Università di Pisa
Department of Computer Science
Doctoral Program in Computer Science (XXXIII Cycle)

Ph.D. Thesis

SECURE COMPILATION ALL THE WAY DOWN

SECURE COMPILATION AT DIFFERENT LEVELS OF THE COMPUTATION STACK

Matteo Busi

SUPERVISORS:

Prof. Pierpaolo Degano
(Dept. of Computer Science – Università di Pisa)

Dott. Letterio Galletta
(IMT School for Advanced Studies Lucca)

April 2021

Matteo Busi: *Secure Compilation All the Way Down*

The "Coq symbol" (🐓) and the emojis in the thesis come from the Noto Emoji project.

ABSTRACT

Software is pervasive in our daily lives and we rely on it for many critical tasks. Despite the abundance of (formal) techniques that can be used to prevent bugs, software is often incorrect and insecure. One of the sources of this insecurity is that a gap persists between what the programmer writes and what actually gets executed.

Such a gap is mainly due to the fact that it is often hard to translate (e.g., via compilation) a source language (typically at high-level) into a target language (typically at lower-level), without losing any abstraction.

Indeed, a well-known advantage of using a high-level language is that it usually provides a multiplicity of abstractions and mechanisms (e.g., types, modules, automatic memory management) that enforce good programming practices and ease programmers in writing correct and secure code. However, those high-level abstractions do not always have counterparts at the low-level. This discrepancy can be dangerous when the source level abstractions are used to enforce security properties: if the target language does not provide any mechanism to preserve such properties, the resulting code is vulnerable to attacks.

One tentative solution could be to adapt the source to the target language (or vice versa) to make them equally powerful, but that is undesirable because we possibly lose the advantages that come from having different levels of abstraction in the source and in the target. A better solution is to work on the compiler itself, by guaranteeing that it preserves the source-level security properties. The emerging field of secure compilation has exactly this goal. More precisely, secure compilation is concerned with ensuring that the security properties at the source level are preserved as they are at the target level or, equivalently, that all the (interesting) attacks that can be carried out at the target level have corresponding attacks at the source level. In this way, the reasoning carried out at the source level to rule out attacks suffices to rule out attacks at the target.

This thesis approaches the problem of secure compilation from three different points of view. We argue that secure compilation — being relevant at many levels of the computational stack — needs to be tackled at different levels of abstraction, depending on the security goals that one has in mind.

At the highest level of abstraction we work on verifying that security properties are preserved under program transformations whose target language is the same as the source (e.g., program optimizations or obfuscations). We first follow a classical approach and we manually prove that the widely-used control-flow flattening program obfuscation preserves the constant-time policy, i.e., a security policy that requires that the execution time of programs does not depend on their secret inputs. Then, we move to an automatic and efficient approach. For that, we assume a (security) type system that statically checks whether the property of interest holds or not, and we provide a framework to make its usage incremental, so as to make it possible to perform the analysis after each optimization step without excessively slowing down the compiler.

Moving down from the highest level of abstraction, we weaken the requirement about the source and the target languages being equal and consider transformations that involve a translation step. This line of research is more general and follows an approach similar to that of translation validation to automatically certify that a compiled program has a given security property knowing that its original version enjoys it.

Finally, at the very bottom of the computational stack we deal with low-level attackers, e.g., those that can carry out micro-architectural attacks. More precisely, we provide an instantiation of the well-known principle of full abstraction to prove that an extension to an enclaved-execution architecture is secure with respect to the class of interrupt-based micro-architectural attacks.

ACKNOWLEDGEMENTS

I would first like to thank my advisors Prof. Pierpaolo Degano and Dott. Letterio Galletta for being terrific mentors, since the very beginning of my academic career: if I will ever be a good researcher the merit is mostly yours. A special thanks goes to the members of my internal committee, Prof. Marco Danelutto and Prof. Luca Viganò, for their insightful comments and advices. I am also grateful to Prof. Dominique Devriese and Prof. Cătălin Hrițcu for reading the first version of this thesis: your very detailed comments helped me to improve the quality of this work significantly. I wish also to thank all the people I met during my visit at KU Leuven, especially Prof. Frank Piessens: these three months have been fundamental for me, as a researcher and as a person.

To all the friends I met during my eight years in Pisa, all my flat mates and (more recently) my office mates: you all made Pisa feel like a second home.

I also wish to thank my family, without your help and continuous support I would have never reached my goals. *Grazie!*

Finally, the biggest thank goes to Sara. You have always been there, supporting and loving me, even in the darkest moments. *Thanks.*

CONTENTS

1 INTRODUCTION xiv
  1.1 Published work on our research . . . xvi
2 BACKGROUND 1
  2.1 Formal notions of security . . . 1
    2.1.1 Trace properties and hyperproperties [72] . . . 1
    2.1.2 Security via program equivalences . . . 4
  2.2 Secure compilation . . . 5
    2.2.1 Passive attackers . . . 5
    2.2.2 Active attackers . . . 7
  2.3 Proving and enforcing program properties . . . 11
    2.3.1 Type systems . . . 11
    2.3.2 Translation validation . . . 13
    2.3.3 Security of compiler optimizations . . . 14
    2.3.4 Low-level protection mechanisms . . . 15
3 A CLASSICAL APPROACH TO SECURE COMPILATION 17
  3.1 The constant-time policy . . . 17
  3.2 Control-flow flattening . . . 19
  3.3 A short guide to CT-simulations . . . 20
  3.4 The case of control-flow flattening . . . 22
    3.4.1 The language and its (instrumented) semantics . . . 22
    3.4.2 Control-flow flattening formalization . . . 24
    3.4.3 Correctness and security . . . 25
  3.5 Conclusions . . . 30
4 INCREMENTAL TYPING 32
  4.1 The incremental schema in a nutshell . . . 34
  4.2 Formalizing the incremental schema . . . 36
    4.2.1 Incorporating incrementality . . . 39
  4.3 Making existing typing algorithms incremental . . . 43
    4.3.1 Type checking a functional language . . . 44
    4.3.2 Type inference on a functional language . . . 46
    4.3.3 Type checking non-interference . . . 49
    4.3.4 Type checking robust declassification . . . 53
    4.3.5 Type inference of exceptions . . . 57
    4.3.6 Type checking the dependently-typed λ-calculus . . . 65
    4.3.7 Type checking secrecy in the spi-calculus . . . 67
  4.4 Implementation and experiments . . . 75
    4.4.1 The INCREMENTALIZER . . . 75
    4.4.2 A git-versioned ray tracer . . . 76
    4.4.3 Experimental evaluation . . . 78
  4.5 Conclusions . . . 83
5 SECURE TRANSLATION VALIDATION 87
  5.1 Our proposal . . . 88
  5.2 Secure translation validation . . . 89
  5.3 Effective secure translation validation . . . 90

5.4 A use case for secure translation validation . . . 93
5.4.1 The concrete and abstract source languages and their analysis . . . 94
5.4.2 The concrete and abstract target languages and their analysis . . . 96
5.4.3 An example, formally . . . 106
5.5 Discussion . . . 109
5.6 Conclusions . . . 111
6 SECURE COMPILATION AGAINST MICRO-ARCHITECTURAL ATTACKS 113
6.1 Enclaves and interrupt-based attacks . . . 114
6.1.1 Enclaved execution . . . 114
6.1.2 Interrupt-based attacks . . . 115
6.2 Overview of our approach . . . 115
6.2.1 Sancus model . . . 116
6.2.2 Security definitions . . . 117
6.2.3 Secure interruptible Sancus . . . 121
6.3 The formal model of the architecture . . . 123
6.3.1 Memory and memory layout . . . 123
6.3.2 Register files . . . 124
6.3.3 I/O Devices . . . 125
6.3.4 Software modules, contexts and whole programs . . . 126
6.3.5 Instruction set . . . 126
6.3.6 Configurations . . . 127
6.3.7 CPU mode . . . 128
6.3.8 Memory access control . . . 129
6.4 The semantics of Sancus^{H}, $\textbf{Sancus}^{L}$ and their interrupt logic . . . 129
6.4.1 The Operational Semantics of Sancus^{H} . . . 130
6.4.2 The Operational Semantics of $\textbf{Sancus}^{L}$ . . . 131
6.4.3 A progress theorem . . . 133
6.5 The security theorem . . . 136
6.5.1 Reflection of behaviors . . . 137
6.5.2 Preservation of behaviors . . . 138
6.6 Preservation of hyperproperties . . . 148
6.6.1 Take one: termination-insensitive, time-sensitive non-interference . 150
6.6.2 Take two: termination- and time-sensitive non-interference . . . 150
6.6.3 Take three: stepwise termination- and time-sensitive non-interference 151
6.6.4 Take four: hypersafety . . . 152
6.7 Discussion . . . 153
6.7.1 Full abstraction as a security objective . . . 153
6.7.2 The impact of our simplifications . . . 154
6.8 Conclusions . . . 156
7 CONCLUSION 159
7.1 Future work . . . 160
A INCREMENTAL TYPING 163
A.1 Additional proofs for Section 4.3 . . . 163
A.2 Additional plots for Section 4.4 . . . 166
A.2.1 Original vs. Incremental with simulated changes . . . 166
A.2.2 Original vs. Incremental with inter-dependencies . . . 170
B SECURE COMPILATION AGAINST MICRO-ARCHITECTURAL ATTACKS 171
B.1 The device of Section 6.3.6.1 is deterministic . . . 171
B.2 Complete operational semantics rules of Sancus^{H} . . . 171

B.3 Complete operational semantics rules of Sancus^{L} . . . 174
B.4 Proof of progress of Section 6.4.3 . . . 177
B.5 Proofs and additional definition for Section 6.5.1 . . . 177
B.6 Definitions and proofs for Lemmata 6.3 and 6.4 . . . 178
B.6.1 Properties of Definition 6.9 . . . 178
B.6.2 Properties of Definition 6.8 . . . 182
B.7 Proofs of Lemmata 6.3 and 6.4 . . . 189
B.8 Proof of Proposition 6.3 and Algorithm 3 . . . 191
B.9 Proofs and additional definitions of Section 6.6 . . . 198

BIBLIOGRAPHY 201

LIST OF FIGURES

Figure 1	Code that checks the correctness of a given PIN (dead-store elimination highlighted in red).	xv
Figure 3	Some of the rules of the type checking algorithm $\mathcal{S}$ (with subtyping) for language of [217].	13
Figure 4	A high-level overview of the translation validation approach.	14
Figure 5	Instrumented operational semantics for commands.	23
Figure 6	Definition of $\approx_p$ relation on configurations and its auxiliary relations.	26
Figure 7	Three rules of the original type system $\mathfrak{T}$ (upper part) and their corresponding incremental versions in the type system $\mathcal{I}\mathfrak{T}$. The rest of the rules is in Appendix A.	35
Figure 8	Abstract-syntax tree annotated with types for the factorial program.	36
Figure 9	Incremental type checking of the modified factorial program, where C is as in Table 1.	37
Figure 10	Definition of $buildCache_{\mathfrak{F}}$ for the `FUN` language.	45
Figure 11	Rules defining incremental algorithm $\mathcal{I}\mathfrak{F}$ to type check `FUN`, where $compat_{\mathfrak{F}}$ is as in Definition 4.10.	46
Figure 12	Rules defining algorithm $\mathfrak{W}$ to infer `FUN` types of Section 4.3.2.	47
Figure 13	Definition of $buildCache_{\mathfrak{W}}$ for the incremental type inference of `FUN`.	48
Figure 14	Rules defining incremental algorithm $\mathcal{I}\mathfrak{W}$ to infer `FUN` types, where $compat_{\mathfrak{W}}$ is as in Definition 4.11.	49
Figure 15	The rules of the type checking algorithm $\mathcal{S}$ (with subtyping) for `WHILE` of Section 4.3.4.	50
Figure 16	Definition of $buildCache_{\mathcal{S}}$ for the incremental type checking of `WHILE`.	51
Figure 17	Rules defining the incremental algorithm $\mathcal{I}\mathcal{S}$ to type check `WHILE`, where $compat_{\mathcal{S}}$ is as in Definition 4.12.	52
Figure 18	The original type system $\mathfrak{R}$ of [159].	54
Figure 19	Instantiation of $buildCache_{\mathfrak{R}}$ for $\mathfrak{R}$ [159].	55
Figure 20	The set of rules for using $\mathfrak{R}$ incrementally, where $compat_{\mathfrak{R}}$ is as in Definition 4.13 (part I).	56
Figure 21	The set of rules for using $\mathfrak{R}$ incrementally, where $compat_{\mathfrak{R}}$ is as in Definition 4.13 (part II).	57
Figure 22	Rules defining the inference algorithm for terms of [140].	59
Figure 23	Rules defining the inference algorithm for patterns of [140].	60
Figure 24	Definition of $buildCache_{\mathfrak{I}}$ for the incremental type inference of [140] (upper part for patterns, lower for terms).	62
Figure 25	The rules of the incremental inference algorithm for patterns of [140].	63
Figure 26	The rules of the incrementalized inference algorithm for terms of [140] (part I).	63
Figure 27	The rules of the incrementalized inference algorithm for terms of [140] (part II).	64
Figure 28	The rules for the algorithmic kinding of $\mathfrak{D}$ of λLF [186].	65
Figure 29	The rules for the algorithmic typing of $\mathfrak{D}$ of λLF [186].	66

Figure 30 Definition of $buildCache_{\mathfrak{D}}$ for the incremental kinding and typing of λLF. . . . 66
Figure 31 The rules for the incrementalized version of the kinding of algorithm $\mathfrak{D}$ of λLF [186], where $compat_{\mathfrak{D}}$ is as in Definition 4.15. . . . 67
Figure 32 The rules for the incrementalized version of the typing of algorithm $\mathfrak{D}$ of λLF [186], where $compat_{\mathfrak{D}}$ is as in Definition 4.15. . . . 67
Figure 33 The original type system $\mathscr{P}$ of [3] for terms. . . . 68
Figure 34 The original type system $\mathscr{P}$ of [3] for processes (part I). . . . 69
Figure 35 The original type system $\mathscr{P}$ of [3] for processes (part II). . . . 70
Figure 36 Instantiation of $buildCache_{\mathscr{P}}$ for $\mathscr{P}$ [3]. . . . 71
Figure 37 The incrementalized type system $\mathscr{IP}$ for terms, where $compat_{\mathscr{P}}$ is as in Definition 4.16. . . . 72
Figure 38 The incrementalized type system $\mathscr{IP}$ for processes (part I), where $compat_{\mathscr{P}}$ is as in Definition 4.16. . . . 73
Figure 39 The incrementalized type system $\mathscr{IP}$ for processes (part II), where $compat_{\mathscr{P}}$ is as in Definition 4.16. . . . 74
Figure 40 The graphical report as generated by the incremental typechecker for two successive commits of the explicitly typed MinRT. In green the nodes found in the cache; in cyan the nodes whose type was discovered resorting to the original algorithm; in red the nodes not in the cache. . . . 78
Figure 41 Experimental results comparing the number of re-typings per second vs. the number of nodes of the *diff* sub-tree. The blue, dashed plot is for the original type checking, while the orange, solid one is for the incrementalized one. Dotted plots report the behavior of the incrementalized type checker for various values of the threshold T. The x-axis is logarithmic, while the y-axis is scaled as necessary. The plots on the right consider the maximum number of variables. . . . 81
Figure 42 Experimental results comparing the number of re-typings per second vs. the number of nodes of the *diff* on two unrollings of the factorial function. The dashed, star-marked plot is for the original type checking, the orange, solid one is for the incrementalized one. Dotted plots report the behavior of the incrementalized type checker for various values of the threshold T. The x-axis is logarithmic, while the y-axis is scaled as necessary. . . . 82
Figure 43 Semantics of the μow^* language. . . . 96
Figure 44 Semantics for runtime programs of the $\mu\mathbf{C}^*$ language. . . . 99
Figure 45 Semantics for whole programs of the $\mu\mathscr{C}^*$ language. . . . 100
Figure 46 The type and effect system for extended partial programs of $\mu\mathscr{C}^*$ (part I). . . . 101
Figure 47 The type and effect system for extended partial programs of $\mu\mathscr{C}^*$ (part II). . . . 102
Figure 48 The type and effect system for contexts of $\mu\mathscr{C}^*$. . . . 103
Figure 49 The type and effect system for whole programs of $\mu\mathscr{C}^*$. . . . 103
Figure 50 Interrupt latency traces corresponding to the conditional control-flow paths in Example 6.2. When interrupting after the 7th instruction, the adversary observes a distinct latency difference for the 4-cycle `MOV` instruction vs. the 1-cycle `NOP` instruction. . . . 121

Figure 51 The secure padding scheme. 122
Figure 52 The rules defining the memory access control. 130
Figure 53 Some rules of the main transition system for Sancus^{H}. 134
Figure 54 The transition system for handling interrupts in $\textbf{Sancus}^{\textbf{L}}$. 135
Figure 55 Some rules from the operational semantics of $\textbf{Sancus}^{\textbf{L}}$. 135
Figure 56 An illustration of the proof strategy of preservation of behaviors. 137
Figure 57 The relation $\overset{\alpha}{\Longrightarrow}$ for fine-grained observables. 139
Figure 58 The relation $\overset{\beta}{\Longrightarrow\!\!\!\!\!\!\!\!\Rightarrow}$ for coarse-grained observables. 140
Figure 59 Initial content of unprotected memory as used by Algorithm 2. . . 146
Figure 60 A graphical representation of the algorithm building the I/O device for β_i and β'_i being in the longest common prefix. Here, δ_L denotes the final state of the I/O device being updated, while the final state of the updated device is depicted as a solid, black circle. 148
Figure 61 A graphical representation of the algorithm building the I/O device for β_i and β'_i being the distinguishing observables. Here, δ_L denotes the final state of the I/O device being updated, while the final state of the updated device is depicted as a solid, black circle. 149
Figure 62 Experimental results for trees with $depth = 10$ comparing the number of re-typings per second vs. the number of nodes of the *diff* sub-tree. The blue, dashed plot is for the original type checking, while the orange, solid one is for the incrementalized one. Dotted plots report the behavior of the incrementalized type checker for various values of the threshold T. The x-axis is logarithmic, while the y-axis is scaled as necessary. The plots on the right consider the maximum number of variables. 166
Figure 63 Experimental results for trees with $depth = 12$ comparing the number of re-typings per second vs. the number of nodes of the *diff* sub-tree. The blue, dashed plot is for the original type checking, while the orange, solid one is for the incrementalized one. Dotted plots report the behavior of the incrementalized type checker for various values of the threshold T. The x-axis is logarithmic, while the y-axis is scaled as necessary. The plots on the right consider the maximum number of variables. 167
Figure 64 Experimental results for trees with $depth = 14$ comparing the number of re-typings per second vs. the number of nodes of the *diff* sub-tree. The blue, dashed plot is for the original type checking, while the orange, solid one is for the incrementalized one. Dotted plots report the behavior of the incrementalized type checker for various values of the threshold T. The x-axis is logarithmic, while the y-axis is scaled as necessary. The plots on the right consider the maximum number of variables. 168
Figure 65 Experimental results for trees with $depth = 16$ comparing the number of re-typings per second vs. the number of nodes of the *diff* sub-tree. The blue, dashed plot is for the original type checking, while the orange, solid one is for the incrementalized one. Dotted plots report the behavior of the incrementalized type checker for various values of the threshold T. The x-axis is logarithmic, while the y-axis is scaled as necessary. The plots on the right consider the maximum number of variables. 169

Figure 66 Experimental results comparing the number of re-typings per second vs. the number of nodes of the *diff* on two unrolling of the factorial function. The dashed, star-marked plot is for the original type checking, the orange, solid one is for the incrementalized one. Dotted plots report the behavior of the incrementalized type checker for various values of the threshold T. The x-axis is logarithmic, while the y-axis is scaled as necessary. 170

LIST OF TABLES

Table 1 Tabular representation of the cache C for our example. 35
Table 2 A summary of experimental results. 79
Table 3 Summary of the assembly language considered. 116
Table 4 Definition of $MAC_{\mathcal{L}}(f, \texttt{rght}, t)$ function, where f and t are locations. 129

1

INTRODUCTION

Computing is a pervasive aspect of our daily lives and its applications are getting more and more complex. To tackle this growing complexity, a variety of *design patterns* has been introduced. One of the most common patterns is layered design, where a system is a stack of layers at an increasing level of abstraction: each layer in the stack *abstracts* the complexity of the layer below it, and provides a well-defined set of functionalities to the layer following it. This design pattern works well enough when considering functional properties like correctness, but the gap in abstraction level between adjacent layers raises many security concerns. This view is well expressed by Piessens [188]:

> [...] from a security point of view, the use of abstraction layers can introduce significant vulnerabilities and hence risks for the resulting ICT systems. So-called layer-below attacks, where an attacker exploits implementation details of lower layers to attack one of the upper layers have been common and have been among the most dangerous attacks over the history of computing.

Similar problems arise also when the layers are (programming) languages, as commonly found in software systems. Typically, translating a *source* language S (*usually at high-level*) into a *target* language T (*usually at lower-level*) introduces a gap between what the programmer writes and what actually gets executed [81], thus giving more power to attackers at the target level. For instance, a well-known advantage of using a high-level source language is that it usually provides a multiplicity of abstractions and mechanisms (e.g., types, modules, automatic memory management) that enforce good programming practices and ease programmers in writing correct and secure code. However, those high-level abstractions do not always have counterparts at the low-level. This discrepancy can be dangerous: if the target program is not endowed with mechanisms to enforce source-level properties, target attackers can exploit their additional power and knowledge about the discrepancy to carry out attacks.

More concretely, consider the following piece of code written in a `C`-like language with no pointers:

```
1 static int count = 0;
2
3 void count ()
4 {
5     count++;
6 }
```

In this case, the programmer used the `static` qualifier to limit the scope of the variable `count` to the current file, thus trying to guarantee the *integrity* of its stored value. After compilation to a *RISC*-like assembly, the code above may be transformed into the following (assume for simplicity that `count` is stored in the register `R0`):

```
1 printf("Pin:");
2 pin = read_secret();
3
4 if (check(pin))
5    printf("OK!");
6 else
7    printf("KO!");
8
9 pin = 0; // reset pin
```

Figure 1: Code that checks the correctness of a given PIN (dead-store elimination highlighted in red).

```
1 ...
2
3 count:
4    add R0, R0, 1
```

At the target level, the value in R0 is no longer protected, since any other target-level module linking to our code can read from or write to the register R0. This toy example leverages the very same idea as real-world, potentially dangerous attacks: both use target-level mechanisms to break high-level abstractions, thus making the reasoning at the source level useless [201, 225].

One possible solution would be to adapt the source language to the target one (or vice versa) so making them equally powerful. However, that is undesirable: the source language would lose most of the advantages that come from having different levels of abstraction in the source and in the target. Moreover, it would not even be sufficient. Indeed, as recently observed by D'Silva et al. [81] and previously known in the cyber-security community [60–62, 243], even less radical, seemingly innocuous code transformations that map a language into itself (e.g., compiler optimizations) may hinder security.

Consider for instance the snippet of code in Figure 1, naively checking the correctness of a given PIN. If we apply to it the well-known optimization *dead-store elimination*, the assignment highlighted in red is removed since the variable pin is never used after being assigned. Unfortunately, that assignment ensured the confidentiality of the value of pin, that now might be accessed by any attacker able to read the memory of the program (e.g., an untrusted library linked to our optimized code): dead-store elimination makes it possible to leak the secret.

A way to solve both the above problems is to work on the compiler itself [14, 125], by guaranteeing that it preserves the source-level security properties. The field of *secure compilation* has exactly this goal. More precisely, secure compilation is concerned with granting that the security properties at the source level are preserved as they are at the target level or, equivalently, that all the attacks that can be carried out at the target level also have a corresponding attack at the source. In this way, it is enough to reason at the source level to rule out attacks at the target.

Actually, we argue that secure compilation — being relevant at many levels of abstraction — needs to be tackled following different approaches, also depending on the security goals that one has in mind.

At the highest level of abstraction, we follow two approaches. In Chapter 3 we look at compiler security in the classical way, by *manually* showing that a code transformation preserves an interesting security property (actually, a variant of non-interference). More precisely, we follow the methodology proposed by Barthe et al. [21] and prove that the widely-used code obfuscation *control-flow flattening* [39, 75] preserves the *constant-time policy* [21] in an imperative language with a switch construct.

Still at the highest level of abstraction, we work on *automatically* and *efficiently* verifying that security properties are preserved under program transformations whose target language is the same as the source (e.g., program optimizations). For that, we assume a (security) type system that statically checks whether the property of interest holds or not, and we provide a framework to make its usage incremental (*incrementalization*), so as to make it possible to perform the analysis after *each* optimization step without excessively slowing down the compiler. We remark that preservation of security properties is just one of the envisioned applications of incrementalization. Indeed, this approach could also be applied when the code changes frequently and needs to be checked efficiently (e.g., IDEs or continuous integration). Incrementalization is presented in its full generality in Chapter 4.

Moving down from the highest level of abstraction, we need to lift the constraint that requires the source language to be equal to the target and consider transformations that involve a translation step. To this aim we introduce *secure translation validation*, which is inspired by the translation validation approach [189]. Roughly, with secure translation validation we can *automatically certify* that, in a given execution environment, the translated program enjoys all the safety properties of its source counterpart. Chapter 5 presents secure translation validation and shows a preliminary result on how to successfully apply it to a simple use-case inspired from the literature [193].

Finally, in Chapter 6 we consider the very bottom of the computational stack and we deal with low-level attackers (e.g., those that can break the isolation mechanisms of the processor by exploiting its micro-architectural features). More precisely, we faithfully model the Sancus architecture [168, 170] and extend it with carefully-designed interruptible enclaves. Then, we provide an instantiation of the secure compilation principle of *full abstraction* [2] to prove that our extension is *backward compatible* with the original Sancus model and is *secure* (i.e., the isolation mechanism is not weakened) with respect to the class of interrupt-based micro-architectural attacks (e.g., the Nemesis attack [225]). Additionally, we also show that in our scenario full abstraction actually implies the preservation of various notions of non-interference.

1.1 PUBLISHED WORK ON OUR RESEARCH

This thesis collects, revises and updates the following material which we developed during the Ph.D.:

- Chapter 2 extends the short survey which appeared in
  - Matteo Busi and Letterio Galletta. “A Brief Tour of Formally Secure Compilation.” *Proceedings of the Third Italian Conference on Cyber Security, Pisa, Italy, February 13-15, 2019*. Ed. by Pierpaolo Degano and Roberto Zunino. Vol. 2315. 2019. URL: http://ceur-ws.org/Vol-2315/paper03.pdf
- Chapter 3 includes a polished version of the following paper:

- Matteo Busi, Pierpaolo Degano, and Letterio Galletta. "Control-flow Flattening Preserves the Constant-Time Policy." *Proceedings of the Fourth Italian Conference on Cyber Security, Ancona, Italy, February 4th to 7th, 2020*. Ed. by Michele Loreti and Luca Spalazzi. Vol. 2597. 2020, pp. 82–92. URL: http://ceur-ws.org/Vol-2597/paper-08.pdf

- Chapter 4 updates and joins together the following papers:
  - Matteo Busi, Pierpaolo Degano, and Letterio Galletta. "Using Standard Typing Algorithms Incrementally." *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*. 2019, pp. 106–122. DOI: 10.1007/978-3-030-20652-9_7;
  - Matteo Busi, Pierpaolo Degano, and Letterio Galletta. "Robust Declassification by Incremental Typing." *Foundations of Security, Protocols, and Equational Reasoning - Essays Dedicated to Catherine A. Meadows*. Ed. by Joshua D. Guttman, Carl E. Landwehr, José Meseguer, and Dusko Pavlovic. Vol. 11565. Lecture Notes in Computer Science. Springer, 2019, pp. 54–69. DOI: 10.1007/978-3-030-19052-1_6

  and the journal paper:
  - Matteo Busi, Pierpaolo Degano, and Letterio Galletta. "Mechanical incrementalization of typing algorithms." *Science of Computer Programming* 208 (2021). ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102657

- Contents of Chapter 5 are still unpublished, but our ideas were first presented at the PriSC'19 informal workshop:
  - Matteo Busi, Pierpaolo Degano, and Letterio Galletta. "Translation Validation for Security Properties." *3rd Workshop on Principles of Secure Compilation, PriSC 2019, Cascais, Portugal, January 13, 2019*. 2019. URL: https://arxiv.org/abs/1901.05082

  Moreover, the following paper (which is currently under review) extends the use case of Chapter 5:
  - Matteo Busi, Pierpaolo Degano, and Letterio Galletta. "Secure Translation Validation: Effective Preservation of Robust Safety Properties" (2021)

- Finally, Chapter 6 is taken from:
  - Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. "Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors." *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. 2020, pp. 262–276. DOI: 10.1109/CSF49147.2020.00026

  and its extended version, which is currently under review:
  - Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. "Securing Interruptible Enclaved Execution on Small Microprocessors" (2021)

2

BACKGROUND

This chapter is a gentle introduction to the relevant background for this thesis and secure compilation in general. In the next section we give a non-exhaustive overview on how security can be formally defined. First, we provide a precise definition of security properties in terms of sets of allowed *traces*, then we follow [72] and extend this definition with the ability to predicate about combinations of traces of the same system. We conclude the section by introducing a further notion of security based on *program equivalences*. In Section 2.2, we give a formal definition of compilers and introduce multiple notions of secure compilation. We start with the case of attackers that just observe their victims (i.e., *passive attackers*) and then we move to the more powerful model of *active attackers* (i.e., those that can also interfere with the execution of their victims). Finally, Section 2.3 overviews some common techniques that can be employed to automatically enforce or prove security properties. More specifically, we shallowly cover some well-known static analysis techniques, along with emerging mechanisms that can be used to enforce security directly at the hardware level.

2.1 FORMAL NOTIONS OF SECURITY

The *security* of a system, either hardware or software, can be defined in a variety of ways, depending on the power of the attacker, on the level of detail one is interested in and on the *security policies* to consider.

Although most of the definitions below can be stated for generic systems, for the scope of this thesis we limit ourselves to programs. If not stated otherwise, we assume W to be a *complete and executable* program, namely a *whole program* ($W \in \mathit{Whole}$).

2.1.1 *Trace properties and hyperproperties [72]*

When discussing and defining security policies, it is common to model executions of programs as *sets of traces*. More formally, the behavior of a program W — written $\mathit{beh}(W)$ — is a subset of the set of all traces Ψ:

$$\Psi_{\mathit{fin}} \triangleq \Sigma^* \qquad \Psi_{\mathit{inf}} \triangleq \Sigma^\omega \qquad \Psi \triangleq \Psi_{\mathit{fin}} \cup \Psi_{\mathit{inf}}$$

where Σ is called the set of *observables (or events)*, and Ψ_{fin} and Ψ_{inf} denote finite and infinite sequences of observables. Intuitively, an observable encodes what an external observer learns about W by looking at a single execution step of it. Thus, traces denote the information that is released during a sequence of execution steps of W. Typical examples of observables include I/O operations, errors occurred during the execution of a program, or termination [21, 72].

Example 2.1 ([21])**.** *Consider the* constant-time programming *model which is used to guide the development of secure cryptographic libraries. According to this model, the execution of a program reveals to an external observer (that is, a potential attacker) information about (1) the number and the order of steps they perform; (2) the value of branching conditions they evaluate; and (3) the memory addresses they manipulate. We can express this model using observables and traces by choosing* Σ *as* $\mathit{Addr} \cup \{\mathit{true}, \mathit{false}, \bullet\}$. *Addr encodes the set of valid memory addresses and is used to keep track of memory manipulations, true and false are used to represent the results of branching and* $\bullet$ *allows counting other steps of execution.*

From now onward we indicate the i-th element in a trace t as t_i, and we say that $t \leq t'$ whenever t is a *prefix* of $t \in \Psi$ (if $t \in \Psi_{\mathit{fin}}$ we say that t is a *finite prefix*).

TRACE PROPERTIES The simplest way to define security policies is through the so-called *trace properties* [16, 131]. The intuition is that a trace property is the set of behaviors allowed for a given program. More precisely:

Definition 2.1. *A program* W *satisfies a property* $\pi \in \mathit{Prop} = \wp(\Psi)$ *(written* $W \vDash \pi$*)* iff $\mathit{beh}(W) \subseteq \pi$.

Example 2.2. *Consider for example a policy stating that "a program may write to the network as long as it did not read from a file", this policy can be formalized [72] as*

$$\mathit{NRW} \triangleq \big\{t \in \Psi_{\mathit{inf}} \mid \neg\big(\exists i < j.\ \mathit{isFileRead}(t_i) \wedge \mathit{isNetworkWrite}(t_j)\big)\big\}$$

where isFileRead and isNetworkWrite are predicates on Σ.

In the space of trace properties we can identify two interesting families: *safety* and *liveness properties*. Intuitively, a safety property π_S requires that something *bad and finitely observable* (m) never happens [72]:

$$\begin{aligned}
&\pi_S \in \mathit{Safety} \Leftrightarrow \\
&\quad \forall t \in \Psi_{\mathit{inf}}.\ t \notin \pi_S \Rightarrow \\
&\quad\quad \exists m \in \Psi_{\mathit{fin}}.\ m \leq t \wedge (\forall t' \in \Psi_{\mathit{inf}}.\ m \leq t' \Rightarrow t' \notin \pi_S).
\end{aligned}$$

Dually, a liveness property π_L prescribes that something *good* will *eventually* happen:

$$\pi_L \in \mathit{Liveness} \Leftrightarrow \forall t \in \Psi_{\mathit{fin}}.\ (\exists t' \in \Psi_{\mathit{inf}}.\ t \leq t' \wedge t' \in \pi_L).$$

Note that there exist trace properties that do not belong to the two families above.

Example 2.3. *A trace property stating that "*eventually *a write to the network happens, but that all the previous events were file reads" is neither a safety nor a liveness property [16].*

The above example hints at a general characteristic of trace properties, i.e., each trace property can be written as the intersection between a safety and a liveness property:

Theorem 2.1 (From [16])**.**

$$\forall \pi \in \mathit{Prop}.\ \exists \pi_S \in \mathit{Safety}, \pi_L \in \mathit{Liveness}.\ \pi = \pi_S \cap \pi_L.$$

HYPERPROPERTIES Suppose now that we want to study how confidential data (e.g., secret keys of cryptographic algorithms) affects the observable behavior of a program. For that, split the states of programs into a *low* (or public) part that is accessible to *low users* (i.e., external observers) and a *high* part that must be kept secret from them. Consider the policy of *observational determinism (OD)* that requires a program to appear as *deterministic* to the *low users*. As one can imagine OD may be used to describe the security of many programs and systems, but it is rather straightforward to show that such a security policy cannot be written as a trace property, since establishing that a trace t is allowed in a program requires to compare t with all the other traces of the program in hand. To overcome the limits of trace properties, Clarkson and Schneider proposed *hyperproperties* [72]. In this paragraph we summarize some of the useful definitions and state the analogue of Theorem 2.1 for hyperproperties.

Definition 2.2. *The set of* all *hyperproperties can be defined as*

$$\mathbf{HP} \triangleq \wp(\wp(\Psi)) = \wp(Prop)$$

and we say that a program W satisfies a given hyperproperty $\mathbf{\Pi} \in \mathbf{HP}$ ($W \vDash \mathbf{\Pi}$) iff $beh(W) \in \mathbf{\Pi}$.

Intuitively, a hyperproperty expresses a security policy as the set of program behaviors allowed by that policy.

Trivially, every trace property π can be lifted to a hyperproperty $\wp(\pi)$ that expresses exactly the same policy (i.e., any program satisfying π also satisfies $\wp(\pi)$, and vice versa). However, hyperproperties express much more. Consider again the OD policy, though it cannot be expressed as a trace property, it is easy to express it as a hyperproperty:

$$\mathbf{OD} \triangleq \{T \in Prop \mid \forall t, t' \in T.\, t_0 =_L t'_0 \Rightarrow t \approx_L t'\}$$

where t_0, t'_0 are the initial events of t and t', $t_i =_L t'_j$ holds if and only if the public part of t_i and t'_j coincides and $t \approx_L t'$ holds *iff* t and t' are equivalent according to a low user.

As we did for trace properties we distinguish between classes of hyperproperties. For that, we need to lift the notion of prefix to sets of traces: we say that T is a *prefix* of T' ($T \leq T'$) *iff* $\forall t \in T.\, \exists t' \in T'.\, t \leq t'$ (when $T \in \wp^{fin}(\Psi_{fin})$ we say that T is a *finite prefix* of T'). Then, we define the first class of hyperproperties:

Definition 2.3. *A hyperproperty $\mathbf{\Pi_S}$ is a* safety hyperproperty (hypersafety) *($\mathbf{\Pi_S} \in \mathbf{SHP}$)* iff

$$\begin{aligned} &\forall T \in Prop.\, T \notin \mathbf{\Pi_S} \Rightarrow \\ &\quad \exists M \in \wp^{fin}(\Psi_{fin}).\, M \leq T \wedge (\forall T' \in Prop.\, M \leq T' \Rightarrow T' \notin \mathbf{\Pi_S}) \end{aligned}$$

and the second one:

Definition 2.4. *A hyperproperty $\mathbf{\Pi_L}$ is a* liveness hyperproperty (hyperliveness) *($\mathbf{\Pi_L} \in \mathbf{LHP}$)* iff *it belongs to:*

$$\forall T \in \wp^{fin}(\Psi_{fin}).\, (\exists T' \in Prop.\, T \leq T' \wedge T' \in \mathbf{\Pi_L}).$$

Remarkably, Theorem 2.1 lifts to hyperproperties:

Theorem 2.2 (From [72])**.**

$$\forall \mathbf{\Pi} \in \mathbf{HP}.\, \exists \mathbf{\Pi_S} \in \mathbf{SHP}, \mathbf{\Pi_L} \in \mathbf{LHP}.\, \mathbf{\Pi} = \mathbf{\Pi_S} \cap \mathbf{\Pi_L}$$

2.1.2 *Security via program equivalences*

An alternative way of characterizing the security of a program is to compare its behavior with that of other programs. More concretely, one usually defines a suitable equivalence relation between programs and checks that they behave *indistinguishably* according to an attacker.

Following [2], we define the attacker as a *context* $C \in Ctx$ (i.e., a program with a hole), that must be *linked* with a *partial program* $P \in Partial$ (a program that may require further information to be executed, e.g., a program with free names representing system calls).

Definition 2.5. *The* link *(or* plug*) operator is a function* $\cdot[\cdot] : Ctx \times Partial \to Whole$ *that takes a context* C *and fills its hole with a partial program* P*, producing a whole program* $C[P]$*. Furthermore, we say that* $C[P]$ *converges (written* $C[P]\Downarrow$*)* if and only if *all the possible executions of* $C[P]$ *halt in a finite number of steps.*

In light of these notions, we can then define the equivalence relation encoding indistinguishability:

Definition 2.6. *Let* P, P' *be two partial programs.* P *and* P' *are* contextually equivalent *(written* $P \simeq_{ctx} P'$*)* if and only if *they* equiconverge *in any context, i.e.,*

$$\forall C.\ C[P]\Downarrow \Leftrightarrow C[P']\Downarrow.$$

Note that indistinguishability comes in many different flavors [177], however we follow the tradition of secure compilation and from now onward we identify it with *equiconvergence* [2]. To illustrate why contextual equivalence is relevant for defining security, consider the following example (inspired by [177]).

Example 2.4. *Suppose that we are implementing a simple counter and that we want to allow a callback to be executed as soon as the counter gets updated. The partial program* P *of Figure 2a shows a tentative implementation of a function that resets the counter variable* `cnt`*, invokes the external callback function* `callback` *(to be provided by our untrusted context), and finally checks that the new value of* `cnt` *is indeed* `0` *(and halts if not).*

Now, suppose that we want to make sure that the assertion of Line 7 never fails, independently of the implementation of `callback`*. That is, we need to show that for any possible context* C*, the program* P *is indistinguishable from the program* P' *of Figure 2b (notice the change in the condition of the assertion).*

Depending on how powerful the contexts are, our P *will be either distinguishable or not from* P'*:*

- *If contexts explicitly manipulate the stack (as it happens low level languages), then the context* C *below can tell* P *and* P' *apart:*

```
1  fun callback (int v)
2  {
3     print ("Updated!");
4     int* counter_ptr = /* Inspect the stack to get a pointer
         to counter */;
5     *counter_ptr = 42;
6  }
7
8  fun main ()
9  {
10    reset ();
11    while (true) {};
12 }
```

```
1 int cnt = 0;
2 ...
3 fun reset ()
4 {
5    cnt = 0;
6    callback (cnt);
7    assert (cnt = 0);
8 }
```

(a) The program P from Example 2.4.

```
1 int cnt = 0;
2 ...
3 fun reset ()
4 {
5    cnt = 0;
6    callback (cnt);
7    assert (true);
8 }
```

(b) The program P' from Example 2.4.

Indeed, $C[P']$ always diverges since the assertion succeeds and the infinite loop in the `main` *function of C get executed, while $C[P]$ terminates since the assertion on the value of the counter fails.*

- *Otherwise, if contexts do not explicitly manipulate the stack then they cannot distinguish the two implementations, and P and P' are contextually equivalent. Indeed, in this particular case we conclude that the assertion in P never fails and thus that the integrity of the counter is not violated.*

2.2 SECURE COMPILATION

In Chapter 1 we informally introduced secure compilation and some of its aspects. We now make these intuitions precise, starting from the notion of compiler:

Definition 2.7 (Compiler). *A* compiler $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ *from* S *to* $\mathbf{T}$, *is a function mapping programs written in the* source language S *into those of the* target language $\mathbf{T}$.[1]

In light of this definition and those of Section 2.1, we can now formalize the basics of *secure compilation*, also depending on the attacker model of choice. Intuitively, a secure compiler is one that *preserves* the security of programs it compiles, i.e., if a program satisfies a security policy *before* compilation, then it must satisfy it even *after* compilation.

2.2.1 *Passive attackers*

The simplest non-trivial attacker that suits for secure compilation is that of *passive attacker*. Intuitively, *a passive attacker* observes the behavior of a program and establishes its adherence to a certain (security) policy. Recall now the notion of behavior of a program and define *secure compilers* in case of passive attackers as follows:[2]

Definition 2.8. *A compiler* $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ *is* secure for *a family of hyperproperties* $\mathbf{F}$ iff

$$\forall \mathsf{W} \in \mathsf{Whole}, \mathbf{\Pi} \in \mathbf{F}.\ beh(\mathsf{W}) \models \mathbf{\Pi} \Rightarrow beh([\![\mathsf{W}]\!]^{\mathsf{S}}_{\mathbf{T}}) \models \mathbf{\Pi}.$$

Consider the case in which $\mathbf{F}$ is the set of trace properties. In this case, proving a compiler correct suffices to prove that it is secure too. Indeed, according to the following definition that requires the source program to exhibit a trace when the target code does:

1 For better readability [175] we hereafter highlight in blue, sans-serif font elements of S, in **red, bold** font elements of $\mathbf{T}$ and in black those that are in common.

2 The definition is a simplified version of RHP as proposed in [10]. Moreover, it is given for families of hyperproperties, but we abuse the notation and use the same definition also for families of trace properties.

Definition 2.9 (Compiler correctness [139])**.** *We say that a compiler* $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ *is* correct iff

$$\forall \mathsf{W} \in \mathsf{Whole}, t.\ t \in beh(\mathsf{W}) \Leftarrow t \in beh([\![\mathsf{W}]\!]^{\mathsf{S}}_{\mathbf{T}}).$$

The following theorem holds:

Theorem 2.3 (From [139])**.** *If* $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ *is correct, then it is secure in case of passive attackers for trace properties.*

Correctness can be established via a *backward simulation (or refinement)* between the behaviors of source and target programs. Typically, this is the case when S is an imperative language with some under-specified aspects (e.g., the evaluation order of expressions) and **T** is an assembly-level language. Other notions of compiler correctness, together with some discussion about their suitability for different kind of languages and their relation with backward simulation, are discussed in [139]. Note that, despite being a relatively well-understood notion, that of compiler correctness may be particularly tricky to get right (especially in presence of separate compilation or when the behavior of S and that of **T** use different observables). See [9, 183] for recent advances on these fronts.

Actually, correctness also suffices in proving that a compiler preserves the *subset-closed (SSC) hyperproperties*, i.e., those hyperproperties $\mathbf{H}$ such that, for any $T \in \mathbf{H}$ and $T' \subseteq T$, $T' \in \mathbf{H}$ (note that any trace property can be turned into SSC hyperproperty) [72].

Theorem 2.4 (From [72])**.** *If* $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ *is correct, then it preserves all the SSC hyperproperties.*

Note that Theorem 2.3 and Theorem 2.4 implicitly assume that the observables we considered for establishing the correctness are the same on which the security property predicates. Consequently, the correctness proof is monolithic in the sense that it has to deal with both functional and non-functional properties, and therefore the observables turn out to be rather complex, along with the proof itself. In addition, this approach is not modular and does not scale well when we want to prove the preservation of new security policies: the correctness proof needs to be changed accordingly. Furthermore, it is not straightforward to reuse off the shelf compilers already proved correct. Take for example CompCert [138], a compiler for C that is proved correct assuming as observables I/O operations (calls to library functions and load/store on those variables modelling memory-mapped hardware devices). Proving that it preserves non-interference (roughly, the secret part of the initial state of a program must not affect the public part of its final state [206]) would require to observe also the values of public variables and to re-do the proof. For these reasons, some papers in the literature adopt a different approach advocating a neat separation of concerns between functional and non-functional aspects, allowing for modular and incremental proofs (see e.g., [20, 21, 87, 88]). Then, the proof that a compiler preserves the security properties of interest is done assuming its correctness.

Below, we briefly discuss a couple of proposals that address the security of program optimizations, assuming them correct w.r.t. I/O observables. Deng and Namjoshi [87] proved that (variants of) popular compiler optimizations preserve the above property of non-interference, and introduced in [88] a technique that statically enforces it in SSA-based optimization pipelines. In a framework that generalizes [87], Barthe et al. [21] studied the *cryptographic constant-time* policy, which is an instance of observational non-interference and informally requires that the execution time of a program does not depend on non-public parts of the state (see Chapter 3 for a precise definition). The key ingredients of their proposal are three variants of *CT-simulations*. Intuitively, a CT-simulation is a pair of equivalence relations (one for programs at source level and one for those at the target)

that are stable under reduction. Crucially, their approach is modular: given a compiler with a simulation-based correctness proof, one can re-use parts of the proof and of the simulation relation to prove its security. Barthe et al. [20] proved that a suitably modified version of the CompCert compiler [138] preserves the constant-time policy. For that, they identified the CompCert passes that do not preserve cryptographic constant-time, then they proved them secure using simplified variants of the proof techniques in [21]. Recently, Protzenko et al. [193] proved that the KreMLin compiler, a compiler from a subset of F* to *human-readable* C (and more recently, to WebAssembly [202]) preserves the constant-time policy.

2.2.2 *Active attackers*

Secure compilation as presented in the previous section just considers attackers that cannot interfere with programs during their execution and thus act just as observers. Passive attackers model well the scenario in which we consider monolithic programs that incorporate in their code all the functionalities they require to operate, or when the whole code-base of a system is trusted. However, this attacker model is not fully adequate in many cases, since most of the time programs get executed in an *active, untrusted* environment. Consider for example when we want to preserve properties of programs whose external references cannot be solved at compile time, e.g., because they rely on dynamically linked libraries. These real-world situations require a sharper notion of security, where the environment has to be modeled explicitly. For that, we resort again to *contexts* (Section 2.1.2). Indeed, the context acts as an attacker that can actively interact with the program at run-time, rather than just passively watching its execution.

Building upon the notions of security presented in Section 2.1, secure compilation with *active attackers* can be either based on (hyper)properties (*robust hyperproperty preservation*[3]) or program equivalences (*fully abstract compilation*).

ROBUST HYPERPROPERTY PRESERVATION To take into account active attackers and the external environment where the program is plugged in, we resort to the notion of robust hyperproperty preservation, written $RHP(\mathbf{F})$ [10], where $\mathbf{F}$ denotes a family of hyperproperties:

Definition 2.10 (Robust hyperproperty preservation (RHP)). *A compiler* $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ *from a source language* S *to a target language* $\mathbf{T}$ robustly preserves $\mathbf{F}$ iff *for any partial source program* P

$$\forall \mathbf{\Pi} \in \mathbf{F}.\ (\forall \mathsf{C}.\ \mathsf{C}[\mathsf{P}] \vDash \mathbf{\Pi}) \Rightarrow (\forall \mathbf{C}.\ \mathbf{C}[[\![\mathsf{P}]\!]^{\mathsf{S}}_{\mathbf{T}}] \vDash \mathbf{\Pi}).$$

However, directly proving that a compiler robustly preserves a family of hyperproperties is not at all trivial. To simplify proofs, Abate et al. developed so-called *property-less* characterizations of secure compilers which are equivalent to specialized versions of the criteria in Definition 2.10, but do not explicitly mention the family of hyperproperties they refer to. Among these, the simplest characterization is that of *robustly safe compilers* and it is equivalent to the criteria prescribing the robust preservation of all the *safety properties*.

Example 2.5 (Robust safety property preservation [10, 180]). *Suppose we are interested in preserving all the safety properties (that is,* $\mathbf{F}$ *is the Safety set lifted to hyperproperties). Definition 2.10 can be specialized as follows:*

3 Here, we limit ourselves to hyperproperties. Abate et al. [10] introduce a variety of secure compilation notions to deal with more general classes of security policies.

Definition 2.11 (Robust safety property preservation (RSP))**.** *The compiler* $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ robustly preserves safety properties *(written* $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} \in RSP$*)* iff

$$\forall \mathsf{P}, \pi \in \mathit{Safety}.\ (\forall \mathsf{C}.\ \mathsf{C}[\mathsf{P}] \vDash \pi) \Rightarrow (\forall \mathbf{C}.\ \mathbf{C}[[\![\mathsf{P}]\!]^{\mathsf{S}}_{\mathbf{T}}] \vDash \pi).$$

The property-free version of RSP goes as follows:

Definition 2.12 (Robustly safe compiler (RSC))**.** *A compiler* $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ *is* robustly safe *(written* $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} \in RSC$*)* iff

$$\forall \mathsf{P}, \mathbf{C}, m.\ (\mathbf{C}[[\![\mathsf{P}]\!]^{\mathsf{S}}_{\mathbf{T}}] \rightsquigarrow m) \Rightarrow (\exists \mathsf{C}.\ \mathsf{C}[\mathsf{P}] \rightsquigarrow m),$$

where $C[P] \rightsquigarrow m \triangleq \exists t \geq m.\ t \in beh(C[P])$*.*

Actually, the two definitions characterize the same set of compilers:

Theorem 2.5 (From [10])**.** $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} \in RSP \Leftrightarrow [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} \in RSC.$

Note in passing that the above definition of secure compiler can be further generalized to deal with relational hyperproperties, that is hyperproperties that relate multiple executions of different programs. See [10] for further details. Another interesting extension of the above definitions (including those concerning compiler correctness) permit to have different observables in source and target programs. Patrignani and Garg [179] were the first to consider this scenario for secure compilers. Recently, Abate et al. [9] extended previous work on compiler correctness and security to systematically deal with these situations.

FULLY ABSTRACT COMPILATION Abadi [2] coined the idea of using program equivalences for compiler security. This kind of compiler security takes the name of *fully abstract compilation* and prescribes that a secure compiler must preserve and reflect the equivalence of behaviors between original and compiled programs under *any* untrusted context of execution, i.e., it must preserve and reflect contextual equivalence (see Definition 2.6). We define *fully abstract compilation* as follows:

Definition 2.13 (Fully abstract compilation (FAC) [2])**.** *A compiler* $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ *is fully abstract* iff

$$\forall \mathsf{P}, \mathsf{P}'.\ \mathsf{P} \simeq \mathsf{P}' \Leftrightarrow [\![\mathsf{P}]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq [\![\mathsf{P}']\!]^{\mathsf{S}}_{\mathbf{T}}$$

where $\simeq$ *denotes contextual equivalence.*

Proofs of FAC are usually split in two, correspondingly to the two sides of the double implication of the definition: *equivalence reflection* ($\Leftarrow$) and *equivalence preservation* ($\Rightarrow$).

EQUIVALENCE REFLECTION ($\Leftarrow$). This side of the double implication guarantees that programs that are equivalent *after* compilation were equivalent also before, so guaranteeing that nothing went wrong during compilation. Indeed, reflection is reminiscent of compiler correctness and for that it is sometimes called the *compiler correctness* side of FAC. However, this terminology is improper [177], since compiler correctness is not usually implied by reflection. Intuitively, a compiler that compiles any source program to the same target program is trivially *incorrect*[4] but it reflects contextual equivalence. Instead, the converse typically holds. For instance, we rephrase here the proof sketch for the case in which S and $\mathbf{T}$ are deterministic:

4 We do not consider programs with empty behavior.

Theorem 2.6. *If* S *and* **T** *are deterministic, then compiler correctness implies equivalence reflection.*

Sketch. We set out to prove (by restating equivalence reflection) that, for any P, P′

$$\forall \mathbf{C}.\ \mathbf{C}[[\![\mathsf{P}]\!]^{\mathsf{S}}_{\mathbf{T}}]\Downarrow \Leftrightarrow \mathbf{C}[[\![\mathsf{P}']\!]^{\mathsf{S}}_{\mathbf{T}}]\Downarrow \Rightarrow \forall \mathsf{C}.\ \mathsf{C}[\mathsf{P}]\Downarrow \Leftrightarrow \mathsf{C}[\mathsf{P}']\Downarrow.$$

Determinism guarantees that behaviors of $\mathsf{C}[\mathsf{P}]$ and $\mathsf{C}[\mathsf{P}']$ are singletons. Compiler correctness implies that $beh([\![\mathsf{C}[\mathsf{P}]]\!]^{\mathsf{S}}_{\mathbf{T}}) \subseteq beh(\mathsf{C}[\mathsf{P}])$ and $beh([\![\mathsf{C}[\mathsf{P}']]\!]^{\mathsf{S}}_{\mathbf{T}}) \subseteq beh(\mathsf{C}[\mathsf{P}'])$ (note that the image of the linking operator is a subset of *Whole*). Thus, it must be $beh([\![\mathsf{C}[\mathsf{P}]]\!]^{\mathsf{S}}_{\mathbf{T}}) = beh(\mathsf{C}[\mathsf{P}])$ and $beh([\![\mathsf{C}[\mathsf{P}']]\!]^{\mathsf{S}}_{\mathbf{T}}) = beh(\mathsf{C}[\mathsf{P}'])$, and in particular $\mathsf{C}[\mathsf{P}]\Downarrow \Leftrightarrow \mathsf{C}[\mathsf{P}']\Downarrow$. □

Because of the above theorem and since most fully abstract compilers are defined and proved correct, equivalence reflection is usually trivial and not relevant for secure compilation purposes. However — despite the above results — equivalence reflection remains interesting in some scenarios. For instance, when one has to guarantee *backward compatibility* between different versions of the same language, as it happens in Chapter 6 and its associated papers [58, 59].

EQUIVALENCE PRESERVATION ($\Rightarrow$). To see why this side of the double implication is relevant for security, consider the contrapositive of the statement (after expanding the definition of $\simeq$):

$$\forall \mathsf{P}, \mathsf{P}', \mathbf{C}.\ \mathbf{C}[[\![\mathsf{P}]\!]^{\mathsf{S}}_{\mathbf{T}}]\Downarrow \not\Leftrightarrow \mathbf{C}[[\![\mathsf{P}']\!]^{\mathsf{S}}_{\mathbf{T}}]\Downarrow \Rightarrow \exists \mathsf{C}.\ \mathsf{C}[\mathsf{P}]\Downarrow \not\Leftrightarrow \mathsf{C}[\mathsf{P}']\Downarrow$$

and observe that it requires that for any target-level distinguishing context, i.e., any successful attacker at the target, there must be a corresponding source-level successful attacker. Restating equivalence preservation as its contrapositive also hints the *backtranslation* proof technique for it. A backtranslation is a function mapping any target-level attacker into a source-level one:

Definition 2.14 (Backtranslation). *Let* Ctx *and* **Ctx** *be the sets of all contexts of* S *and* **T**, *respectively. A* backtranslation *is a function* $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} : \mathbf{Ctx} \to \mathsf{Ctx}$ *such that*

$$\forall \mathbf{C}.\ \mathbf{C}[[\![\mathsf{P}]\!]^{\mathsf{S}}_{\mathbf{T}}]\Downarrow \not\Leftrightarrow \mathbf{C}[[\![\mathsf{P}']\!]^{\mathsf{S}}_{\mathbf{T}}]\Downarrow \Rightarrow \langle\!\langle \mathbf{C}\rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}[\mathsf{P}]\Downarrow \not\Leftrightarrow \langle\!\langle \mathbf{C}\rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}[\mathsf{P}']\Downarrow.$$

Intuitively, a backtranslation *witnesses* the existence of a source-level attacker, thus proving the equivalence preservation. Still, despite the power of backtranslation, proving preservation remains in many cases a challenging task. For that, many different proof techniques have been developed to simplify the proof [14, 15, 45, 90, 164, 176, 178, 223]. We refer the interested reader to the online appendix of the survey by Patrignani et al. [177] for a more in-depth treatment of such techniques.

Despite the hurdles highlighted above, in the last two decades FAC has been the gold-standard for compiler security and found several applications. Among others, Ahmed and Blume [14, 15] and New et al. [164] proved that (variants of) classical *closure conversion* and *continuation-passing style conversion* [156] are fully abstract. Abadi and Plotkin [7] introduced a language translation that uses memory layout randomization to achieve a probabilistic variant of FAC. Fournet et al. [100] defined a fully abstract compiler from (a subset of) ML to JavaScript. Bowman and Ahmed [45] defined a translation from DCC [4] to System F and proved it fully abstract, exemplifying a situation in which a fully abstract compiler also preserves non-interference. Full abstraction has also been used to

assess the security of compilers whose targets are architectures with advanced protection feature like capabilities and enclaves (see Section 2.3.4). For instance, Patrignani et al. [176] proposed a fully abstract compiler from an object-oriented source language to an *Instruction Set Architecture (ISA)* with enclaves. Van Strydonck et al. [226] defined a fully abstract translation from a C-like language equipped with contracts based separation-logic to an ISA equipped with (linear) capabilities. More recently, El-Korashy et al. [94] provided a fully abstract *pointers-as-capabilities* compiler, encoding pointers of its source language as capabilities of the CHERI architecture [234], thus guaranteeing that security properties of the source level are carried over to the target. Finally, also our recent work relies on FAC (see Chapter 6 and [58, 59]). More precisely, we prove a modified version of the Sancus architecture [170] secure against *interrupt-based attacks* (e.g., the Nemesis attack [225]). For that, we show that a version of the Sancus architecture without interrupts (which we call $\mathsf{Sancus}^{\mathsf{H}}$) and one with them enabled ($\mathbf{Sancus}^{\mathbf{L}}$) are fully abstract.

RHP VS. FAC Since the (recent) introduction of trace-based secure compilation criteria [10, 179, 180], a standing question in the community of secure compilation has been how these new secure compilation principles and criteria compare to FAC.

FAC, has indeed some limitations. The first and most serious drawback is that real-world, off-the-shelf compilers seldom are fully abstract. For example, neither the standard compiler from Java to byte-code [2] nor the one from C# to the .NET CLR language [125] are fully abstract. The second shortcoming is that FAC can be hard to prove or to disprove (though the same may apply to other criteria). Indeed, the compiler from System F to the cryptographic λ-calculus by Pierce and Sumii [187] was conjectured to be FAC, but it was proved not such only eighteen years later by Devriese et al. [89]. Also, enforcing FAC requires to instrument the target code, often making it inefficient [180]. Finally, FAC sometimes does not preserve properties that secure compilers are expected to, as shown by the following example inspired by [179].

Example 2.6 (FA $\not\Rightarrow$ safety preservation)**.** *Consider the language* S *to be a simple functional language with boolean values and just the constantly* `true` *function:*

$$\mathsf{S} \ni \mathsf{s} ::= \mathtt{true} \mid \mathtt{false} \mid \lambda\,_.\mathtt{true} \mid \mathsf{s}_1\,\mathsf{s}_2.$$

Similarly, let $\mathbf{T}$ *be a target language with booleans, integers and non-recursive functions:*

$$\mathbf{T} \ni \mathbf{t} ::= \mathtt{true} \mid \mathtt{false} \mid \mathbf{n} \mid \mathbf{x} \mid \mathbf{if}\ \mathbf{t}_1 < \mathbf{t}_2\ \mathbf{then}\ \mathbf{t}_3\ \mathbf{else}\ \mathbf{t}_4 \mid \mathbf{t}_1\,\mathbf{t}_2 \mid \lambda\,\mathbf{x}.\mathbf{t}$$

An FA compiler follows, assuming the observables to be the values returned by functions:

$$[\![\mathtt{true}]\!]^{\mathsf{S}}_{\mathbf{T}} \triangleq \mathtt{true} \quad [\![\mathtt{false}]\!]^{\mathsf{S}}_{\mathbf{T}} \triangleq \mathtt{false} \quad [\![\mathsf{s}_1\,\mathsf{s}_2]\!]^{\mathsf{S}}_{\mathbf{T}} \triangleq [\![\mathsf{s}_1]\!]^{\mathsf{S}}_{\mathbf{T}}[\![\mathsf{s}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$$
$$[\![\lambda\,_.\mathtt{true}]\!]^{\mathsf{S}}_{\mathbf{T}} \triangleq \lambda\,\mathbf{x}.\mathbf{if}\ \mathbf{x} < 2\ \mathbf{then}\ \mathtt{true}\ \mathbf{else}\ \mathtt{false}.$$

Actually, it is a trivial compiler except for the constantly `true` *function that is mapped to a function yielding* `true` *or* `false` *depending on its parameter.*

Consider now the (informal) safety property stating that "a function never outputs `false`*", trivially satisfied by all the programs in* S*. However, target programs can output* `false` *depending on* $\mathbf{x}$*, the target-level input provided by the context, hence invalidating the property.*

Despite these limitations, many of the fully abstract compilers from the literature seem to (implicitly) leverage features of their source and target languages to achieve meaningful security and correctness guarantees [179]. The first to explicitly compare a trace-based secure compilation criterion with FAC were Patrignani and Garg [179]. For that, they

define *trace-preserving (TP) compilation* for reactive programs. Informally, a compiler $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ is TP if any trace of the compiled program $[\![\mathsf{P}]\!]^{\mathsf{S}}_{\mathbf{T}}$ is either a trace of the original program P or is a special *invalid trace*. Depending on how invalid traces are defined, TP compilation comes in two flavors: *halting* and *disregarding*. The first one prescribes that invalid traces must stutter after an invalid input; the second one defines invalid traces as those that discard invalid inputs and corresponding outputs. Finally, TP compilers preserve *all* the safety hyperproperties [179].

In light of recent advances on *robust hyperproperty preservation* [9, 10] and applications of FAC [45, 59], Abate and Busi [11] started an in-depth exploration of the relation between FAC and robust hyperproperty preservation, and explicitly characterized which hyperproperties a fully-abstract compiler is expected to preserve. More details about this work-in-progress are in Chapter 7.

2.3 PROVING AND ENFORCING PROGRAM PROPERTIES

In this section we briefly overview the relevant techniques for automatically enforcing and proving the security properties we introduced above.

2.3.1 *Type systems*

Static analysis comprises a broad collection of methodologies and techniques that allow proving properties of programs, without actually executing them. We just touch the basics of *type systems*, a simple *syntax-directed* and *compositional* static analysis technique.

Two components define a type system, a *set of types*, i.e., syntactic entities that intuitively describe the structure of data used and produced by programs, and a *typing relation* — induced by a set of *inference rules* — that takes the form of a *judgment* [65]:

$$\Gamma \vdash t : \tau$$

with the intuitive meaning that the term t has type τ under the type environment Γ (a mapping from free variables of t to their types).

To guarantee that predicted types are "the right ones", when designing a type system we must make sure that such a system is *sound* [185] with respect to the dynamic semantics of the programming language. Though soundness can be established in a variety of ways, the most commonly used approach is the *syntactic* one [237], that requires to prove the following properties:

PROGRESS: Any well-typed term it is either a value or it can take a step of evaluation according to language semantics.

PRESERVATION: Any well-typed term that takes a step of evaluation is evaluated to a well-typed term.

TYPE CHECKING AND TYPE INFERENCE Deciding whether a typing relation holds or not is a well-studied problem, known as *type checking*:

Definition 2.15 (Type checking)**.** *For a given term t of the language, environment Γ, and type τ decide if $\Gamma \vdash t : \tau$.*

Indeed, type checking is (usually) simple since the typing rules define a (correct and complete) algorithm to decide if a given judgment holds.

A different, usually harder, problem is the following

Definition 2.16 (Type inference). *For a given term t of the language and environment Γ, find the* most general type *τ (if any) such that $\Gamma \vdash t\colon \tau$.*

One of the most well-known type systems for which there exists a *decidable and complete* inference algorithm — *Algorithm W* — is the *Hindley-Milner* type system for functional languages.

The general idea behind Algorithm W is to infer the type of the term in hand by first inferring the types of its sub-terms and then by suitably joining them together with the help of unification. Consider, for example, the following rule used by Algorithm W to infer the type of functional abstractions:

$$(\mathcal{W}\text{-Abs})\quad \frac{\Gamma[x \mapsto \alpha_x, f \mapsto \alpha_x \to \alpha_e] \vdash_{\mathcal{W}} e : (\tau_e, \theta_e) \qquad \theta_1 = \mathcal{U}(\tau_e, \theta_e\alpha_e) \wedge (\tau, \theta) = \big((\theta_1\,(\theta_e\,\alpha_x)) \to (\theta_1\,\tau_e), \theta_1 \circ \theta_e\big)}{\Gamma \vdash_{\mathcal{W}} \lambda_f\, x.e : (\tau, \theta)}\ \alpha_x, \alpha_e\ \textit{FRESH}$$

The rule (i) assumes that the formal parameter x has type α_x (fresh) in Γ and f has type α_x to α_e; (ii) infers recursively the type τ_e (of body e) and the substitution θ_e (binding α_x); (iii) reconstructs the overall type by first unifying (if possible) the type τ_e and the type $\theta_e\,\alpha_e$; and finally (iv) builds the functional type and the new substitution to be returned.

Other inference algorithms exist for the Hindley-Milner type system. The two most successful are *Algorithm M* [135] and *HM(X)* [172]. The first uses a top-down approach (whereas Algorithm W uses a bottom-up approach), while the second is a general algorithm parametrized on the *term constraint system X*, used to express the constraints that arise during the inference.

SECURITY TYPE SYSTEMS Type systems are useful in a variety of situations, and here we focus on situations where they can verify that some trace property or some hyperproperty holds.

The most influential example of a type system that can verify a hyperproperty is by Volpano et al. [229]. It allows checking statically if a program enjoys a variant of *non-interference*. The idea is to annotate every term of the program in hand with a label from a *security lattice* and to type check it in search of violations of non-interference. To better illustrate the concept, consider the `WHILE` language from [217]:

$$\begin{aligned}
&a ::= n \mid x \mid a_1\ \texttt{op}\ a_2 \qquad n \in \mathbb{N},\quad \texttt{op}\in \{+, *, -, \dots\},\quad x \in \mathit{Var}\\
&b ::= \texttt{true} \mid \texttt{false} \mid b_1\ \texttt{or}\ b_2 \mid \texttt{not}\ b \mid a_1 \le a_2\\
&c ::= \texttt{skip} \mid x := a \mid c_1\texttt{;}\ c_2 \mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ b\ \texttt{do}\ c\\
&p ::= a \mid b \mid c\\
&\tau ::= H \mid L \quad \varsigma ::= \tau \mid \tau\ \mathit{var} \mid \tau\ \mathit{cmd}\\
&\Gamma ::= \emptyset \mid \Gamma[p \mapsto \varsigma]
\end{aligned}$$

where Var is the countable set of program variables, ς is a security type (belonging to a security lattice whose ordering relation is the sub-typing relation defined in Figure 3), and $\Gamma \in \mathit{TypeEnv}$ is a typing environment that maps variables to their corresponding security type. Judgments of this type system (call it $\mathcal{S}$) have the following form:

$$\Gamma \vdash_{\mathcal{S}} p\ : \varsigma$$

meaning that under the environment Γ, p has security type ς. Figure 3 shows the most interesting rules inducing $\vdash_{\mathcal{S}}$. Of course the type system must be sound as explained

$$(\mathcal{S}\text{-ASSIGN})\quad \frac{\Gamma \vdash_{\mathcal{S}} a : \tau_a \qquad \Gamma(x) = \tau\ var \land \tau = \tau_a \land \varsigma = \tau\ cmd}{\Gamma \vdash_{\mathcal{S}} x := a : \varsigma}$$

$$(\mathcal{S}\text{-IF})\quad \frac{\Gamma \vdash_{\mathcal{S}} b : \tau_b \qquad \Gamma \vdash_{\mathcal{S}} c_1 : \tau_1\ cmd \qquad \Gamma \vdash_{\mathcal{S}} c_2 : \tau_2\ cmd \qquad \tau_b = \tau_1 = \tau_2 \land \varsigma = \tau_b\ cmd}{\Gamma \vdash_{\mathcal{S}} \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \varsigma}$$

$$(\mathcal{S}\text{-SEQ})\quad \frac{\Gamma \vdash_{\mathcal{S}} c_1 : \tau_1\ cmd \qquad \Gamma \vdash_{\mathcal{S}} c_2 : \tau_2\ cmd \qquad \tau_1 = \tau_2 \land \varsigma = \tau_1\ cmd}{\Gamma \vdash_{\mathcal{S}} c_1\texttt{;}\ c_2 : \varsigma} \qquad (\text{SS-SUB})\quad \frac{\Gamma \vdash_{\mathcal{S}} p : \varsigma_1 \qquad \varsigma_1 \subseteq \varsigma_2}{\Gamma \vdash_{\mathcal{S}} p : \varsigma_2}$$

$$(\text{SS-BASE})\quad \frac{}{L \subseteq H} \qquad (\text{SS-CMD})\quad \frac{\tau' \subseteq \tau}{\tau\ cmd \subseteq \tau'\ cmd} \qquad (\text{SS-REFL})\quad \frac{}{\varsigma \subseteq \varsigma} \qquad (\text{SS-TR})\quad \frac{\varsigma_1 \subseteq \varsigma_2 \qquad \varsigma_2 \subseteq \varsigma_3}{\varsigma_1 \subseteq \varsigma_3}$$

Figure 3: Some of the rules of the type checking algorithm $\mathcal{S}$ (with sub-typing) for language of [217].

above and shown in [217], but it also needs to be *correct*, i.e., the given program must satisfy non-interference when the typing relation holds:

Theorem 2.7 (Correctness). *A program p enjoys non-interference under a Γ if there exists a ς such that*

$$\Gamma \vdash_{\mathcal{S}} p : \varsigma$$

Note also that the converse is not true: there might be cases in which a program enjoys non-interference, but for which the typing relation does not hold.

The literature on type systems applied to security is extremely vast, and security type systems come in different flavors. However, most of them rely on the same principles and some of them are even provably equivalent (see e.g., [197]).

Other interesting examples of security type systems include work on *secret declassification*, i.e., systems in which secrets can be declassified and may become public [159], or real-world programming languages with built-in security types, e.g., Jif [194] a Java extension, FlowCaml [213] an OCaml extension, `seclib` [204] and `LIO` [218] for Haskell.

Moreover, type systems are not limited to functional, imperative or object-oriented languages. For instance, Abadi and Blanchet [5] defined a *secrecy type system* for the spi-calculus [6], in which the typing relation is defined in such a way that (roughly) if a process is typable, then secrecy is guaranteed.

Note also that static analysis techniques other than type systems were applied to the analysis of the security of protocols and programs. Among them *control-flow analysis* [42–44, 86], *abstract interpretation* [146] and *type and effect systems* [24, 41].

2.3.2 *Translation validation*

Pnueli et al. [189] first introduced translation validation to automatically verify the correctness of compilers (and in general translation between languages), without having to build correctness proofs from scratch or to adapt the existing ones after each change in the translation. At a very high-level, the approach is as in Figure 4. A program $\mathsf{P} \in \mathsf{S}$

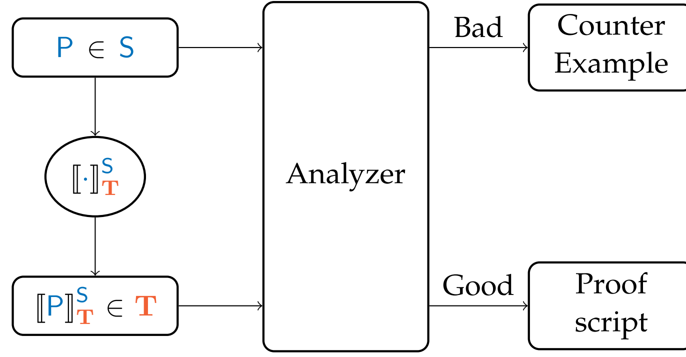


Figure 4: A high-level overview of the translation validation approach.

is compiled to a target program $[\![P]\!]^{S}_{T}$: if the *analyzer* proves that $[\![P]\!]^{S}_{T}$ implements (i.e., refines) the original program P, then it generates a proof script and exits successfully, otherwise it produces a counter-example on which the behavior of P and $[\![P]\!]^{S}_{T}$ differ. It is worth noting that translation validation performs a correctness verification after *each* run of the compiler, so it does not build an *a priori* proof of the correctness of the translation.

The tricky part of this approach resides in the definition of the analyzer. The key to make the process fully automatic lies in first giving a homogeneous semantics to both S and T in terms of a common transition system; then in defining an appropriate notion of *simulation* between the transition systems; and finally in defining a *syntax-based* version of the obtained simulation that entails the original one and that can be built mechanically.

Since the first work by Pnueli et al. [189] translation validation found a vast number of applications. For example, Necula [163] experimented with translation validation on the GCC compiler and Sewell et al. [211] verified the seL4 micro-kernel using the same approach. Variants of this technique were also applied to prove the correctness of compiler optimizations, e.g., Namjoshi and Zuck defined a *stuttering simulation* to perform translation validation of program optimizations [161].

2.3.3 *Security of compiler optimizations*

As sketched in the introduction, the problem of secure compilation remains interesting even when applied to program optimizations or obfuscations as used in compilers.[5] D'Silva et al. [81] were the first to notice that program optimizations passes may not preserve security guarantees, even though the first examples of unsafe compiler optimizations date back to 2002 [200]. To see why optimizations might be dangerous, recall the following example from Chapter 1:

Example 2.7 (Dead-store elimination *is* harmful)**.** *Let* confidentiality *be intuitively expressed as the need of preventing* (secret) information leaks. *Assume to have a compiler pass* $[\![\cdot]\!]_{DSE}$ *from a* C*-like intermediate language to itself, implementing the standard* dead-store elimination (DSE) *that detects and removes dead assignments in programs.*

Now, consider again the program P *from Figure 1. The result of its optimization is* $[\![P]\!]_{DSE}$ *and is the program* P *with the red, boxed assignment removed because it was a dead store, and recognized as such because the compilation of the library happens before linking it with all the other modules.*

Unfortunately, that assignment ensured confidentiality of the secret value of `pin` *that now can be accessed by any context of execution in which* $[\![P]\!]_{DSE}$ *might be plugged in.*

5 In this section, we will denote them as compilers that transform program from an intermediate language to another.

To solve this specific issue Deng and Namjoshi proposed a *confidentiality preserving* version of dead-store elimination [87]. For this, they resort to a taint proof system that infers when a specific dead store can be removed without breaking the confidentiality guarantees of the original code. The same two authors also proposed the so-called *unSSA* algorithm [88] aimed at avoiding the introduction of *new* information leaks caused by compilers that use a *static single assignment* [198] intermediate language (i.e., an intermediate language that requires each variable to be defined before its use and assigned only once). More precisely, consider a compiler that (1) translates the input code written in a source language into an SSA program; (2) optimizes it with an arbitrary number of SSA-based transformations; and (3) outputs the code in the target language. Its problem is that the SSA stage may cause information leaks while transforming multiple assignments to the same variable into different assignments to different and fresh variables. The solution of [88] is to add to the pipeline an additional *unSSA* stage that tries to discover the sets of SSA variables that must be re-joined together in order to remove the newly introduced leaks.

2.3.4 *Low-level protection mechanisms*

In this section, we briefly survey some proposals of target languages, low-level architectural mechanisms and tools that a designer can use to implement secure compilers.

The first proposal is to use a target language endowed with a rich typed structure in order to preserve the type guarantees of the source language. An early example of this is the *typed assembly language (TAL)* [156] that has existential types to support closures and other data abstractions (among other features) and has been used as target language of a type-preserving compiler for System F [156].

Another proposal consists of enriching the target architecture with low-level mechanisms to enforce the security policies during program execution. Along these lines there are *capability machines (CM)*, special microprocessors guaranteeing local-state encapsulation at the hardware level [141]. CMs are based on *memory capabilities* (structures similar to fat pointers that include some access-control information, tagging registers and memory locations). Besides memory capabilities, many machines also implement *object capabilities*, i.e., structures that allow invoking a piece of code without exposing its encapsulated state. Of course, memory and object capabilities neither prevent vulnerabilities in software nor their exploitation, but they do provide strong mitigation mechanisms. Indeed, their main goals are to reduce the attack surface and, in case of successful exploitation, to avoid that attackers gain too many rights over the compromised machine. *Capability Hardware Enhanced RISC Instructions (CHERI)* [234] extends commodity ISAs with capability-based primitives and supports real-world operating systems and applications. Watson et al. [233] proposed a CHERI extension for the `x86` architecture. Also, ARM developed the Morello architecture specification, an ARMv8-A architecture extended with the CHERI protection model [1]. Recent research efforts have been devoted to formally study the properties of this model and to securely compile source code to these machines [94, 103, 215, 216, 226].

Enclaved execution architectures bring some security guarantees at the hardware level by splitting the memory into a protected section (i.e., the *enclave*) and into an unprotected one, in order isolate sensitive and critical applications from the rest of the system. Actually, enclaves enable *secure remote computation* [79]. Upon creation, *enclaves* are initialized with a (software) module, which is isolated from all other software on the same platform (including system software such as the operating system). Furthermore, the correct initialization of modules can be *remotely attested*: a remote party can get cryptographic assurance that an

enclave was properly initialized with a specific module (characterized by a cryptographic hash of the binary module). All these security properties are guaranteed while relying on a small trusted computing base, for instance trusting only the hardware [148, 170] and (possibly) a small hypervisor [99, 147]. This idea was further developed in [13, 169, 176, 178] and also implemented in commodity processors by industry, e.g., Intel Secure Guard eXtensions (Intel SGX). Relevant to our thesis and in this line of research is the *Sancus* [168, 170] architecture, implementing an enclaved execution architecture on the *Texas instrument MSP430*, a low-cost and embedded microprocessor. Sancus achieves (1) software module isolation; (2) remote attestation; (3) secure communication and linking; (4) confidential deployment; and (5) hardware breach confinement. For that, its architecture is equipped with symmetric encryption, key management mechanisms and strictly regulated enclaves for software modules. Note that, even though Sancus provides a number of security mechanisms, a careful design is needed to ensure that these mechanisms are indeed secure [58, 59, 225].

Finally, other proposals attempt to develop new architectures to dynamically enforce security policies. Among them, micro-tagged architectures [17, 18, 92] enrich each word in the machine (both instructions and data) by tagging it with a pointer to a (security) policy to be checked at run-time.

3

A CLASSICAL APPROACH TO SECURE COMPILATION

This chapter shows a rather classical approach to secure compilation, in which a particular code transformation is *manually* proved secure against passive attackers. For that, we focus on obfuscating compilers that are designed to protect a software by obscuring its meaning and impeding the reconstruction of its original source code. Usually, the main concern of these compilers is their robustness against reverse engineering and the performance of the produced code. Very few papers in the literature, e.g., [39], address the problem of proving their correctness and, to the best our knowledge, there are no papers about the preservation of security policies. Here, we offer a first contribution in this direction: we consider a popular program obfuscation (namely *control-flow flattening* [132]) and a specific security policy (namely the *constant-time policy*), and by following [21] we prove that every program satisfying the policy still does after being obfuscated, i.e., the obfuscation preserves the policy. Below, we briefly overview the security policy, the program transformation of interest and the proof technique.

3.1 THE CONSTANT-TIME POLICY

Side-channel attacks allow intruders to extract confidential data by observing the physical behavior of a system. Roughly, an attacker can recover some pieces of confidential information or get indications on which parts are worth their cracking efforts by observing some physical quantity about the execution of the system in hand (the victim). Typically, an attacker measures the physical effects caused by the execution of its victim (e.g., the power it consumes or its execution time) and tries to correlate it with the confidential information the system manipulates (e.g., secret keys). Among these, *timing-based* attacks exploit the execution time of programs to extract confidential information [127]. For instance, if the program in hand branches on a secret, the attacker may restrict the set of values it may assume, whenever the two branches have different execution times and the attacker can measure and compare them. The following toy example illustrates how an attacker may recover confidential information by exploiting a timing-based side-channel.

Example 3.1. *Consider the following function (written in a sugared syntax) taking as an input a secret pin provided by the user, which is checked against the stored one, character by character:*

```
1 fun compare (pin, pin_len):
2    curr := 1;
3    while (curr ≤ pin_len and
4          curr ≤ stored_pin_len and
5          pin(curr) = stored_pin(curr))
6             curr := curr+1;
7
8    return (curr = stored_pin_len+1);
```

Here the constant-time policy is violated since checking a correct pin takes more comparisons between characters than a wrong one. In fact, by measuring the time the attacker can understand which branch was taken, thus restricting the set of values the secret pin may assume.

Many mitigations of timing-based attacks have been proposed, both hardware and software. The *program counter* [155] and the *constant-time* [35] policies are software-based countermeasures, giving rise to the *constant-time programming* discipline. The *constant-time programming* discipline requires that neither the control-flow of programs nor the sequence of their memory accesses depend on secrets, like the value of `pin` in our example. Its goal is to decouple the running time of programs from secrets (i.e., programs are *constant-time* w.r.t. secrets). This task is achieved independently of the platform on which the program is executed (e.g., the operating system or the CPU). Usually, this is formalized as an information flow policy [105] w.r.t. an instrumented semantics that records at least memory accesses and the control-flow of the program [21]. Intuitively, this policy requires that two executions with equivalent initial states (from an attacker's point of view) yield equivalent leakage, making them *indistinguishable* to an attacker. The following example completes the one above, by making the attack more concrete and showing a possible mitigation.

Example 3.2. *Consider again the program from the previous example and a simplified version of the* constant-time policy *leakage model from [21], i.e., an instrumented semantics recording (1) memory accesses (denoted with variable names), (2) arithmetic operations, and (3) branching (either* `true` *or* `false`*). Let now* `stored_pin` *be equal to* `"ps1"`*. If the user inputs* `"pw"`*, the instrumented semantics records the sequence*

$$\boxed{\texttt{curr}} \cdot \boxed{\texttt{true}} \cdot \boxed{+ \cdot \texttt{curr}} \cdot \boxed{\texttt{false}} \cdot \boxed{+}$$

where the first boxed block originates from the initial assignment; the condition of the `while` *leaks the second block and its body leaks the third; the fourth block is observed because the condition of the* `while` *is false for the second character of* `pin`*; the fifth is observed upon evaluation of* `return`*.*

Instead, if the user correctly inputs `"ps1"`*, the same instrumented semantics records*

$$\boxed{\texttt{curr}} \cdot \boxed{\texttt{true}} \cdot \boxed{+ \cdot \texttt{curr}} \cdot \boxed{\texttt{true}} \cdot \boxed{+ \cdot \texttt{curr}} \cdot \boxed{\texttt{true}} \cdot \boxed{+ \cdot \texttt{curr}} \cdot \boxed{\texttt{false}} \cdot \boxed{+}$$

Recall that the value of `pin` *is a secret. The violation of the constant-time policy is now evident: despite starting from equivalent states, the two executions have different leakages (from the fourth boxed block onward). Assuming the length of* `stored_pin` *to be public, the following program is a constant-time version of the one above (assume the operator* `_ ? _ : _` *to be a branch-less conditional operator, e.g., implemented using a* `cmov` *in* `x86` *assembly)*

```
1 fun compare (pin, pin_len):
2    curr := 1;
3    pin_ok := true;
4    while (curr ≤ stored_pin_len)
5       pin_ok := (curr ≤ pin_len and pin(curr) =
            stored_pin(curr)) ? pin_ok : false;
6       curr := curr+1;
7
8    return pin_ok;
```

Indeed, now the leakage does not depend on the value of pin (though it still depends on the length of `stored_pin`*):*

$$\begin{array}{l} \mathit{curr} \cdot \texttt{pin_ok} \cdot \\ \left.\begin{array}{l} \texttt{true} \cdot \mathit{pin_ok} \cdot + \cdot \mathit{curr} \cdot \\ \ldots \\ \texttt{true} \cdot \mathit{pin_ok} \cdot + \cdot \mathit{curr} \cdot \end{array}\right\} \mathit{stored_pin_len}\ \textit{times} \\ \texttt{false} \end{array}$$

3.2 CONTROL-FLOW FLATTENING

Code obfuscation is a program transformation that aims at hiding the intention and the logic of programs by obscuring (portions of) their source or target code. It is used to protect a software making it more difficult to reverse engineer the (source/binary) code, which the attacker can access. In the literature different obfuscations have been proposed. They range from only performing simple syntactic transformations, e.g., renaming variables and functions, to more sophisticated ones that alter both the data, e.g., constant encoding and array splitting [75], and the control flow of the program, e.g., using opaque predicates [75] and inserting dead code.

Control-flow flattening is an advanced obfuscation technique, implemented in state-of-the-art and industrial compilers, e.g., [122]. Intuitively, this transformation re-organizes the *control-flow graph (CFG)* of a program by taking its basic blocks and putting them as cases of a select primitive that dispatches to the right case. In practice, CFG flattening breaks each sequence of statements, nesting of loops and if-statements into single statements, and then hides them in the cases of a large `switch` statement, in turn wrapped inside a `while` loop. In this way, statements originally at different nesting level are put next each other. Finally, to ensure that the control flow of the program during the execution is the same as before, a new variable `pc` is introduced that acts as a program counter, and is also used to terminate the `while` loop mentioned above. The `switch` statement dispatches the execution to one of its cases depending on the value of `pc`. When the execution of a case of the `switch` statement is about to complete `pc` is updated with the value of the next statement to be executed.

The obfuscated version of our constant-time example follows.

```
1  fun compare (pin, pin_len):
2      pc := 1;
3      while(1 ≤ pc)
4          switch(pc):
5              case 1:
6                  curr := 1; pc := 2;
7              case 2:
8                  pin_ok := true; pc := 3;
9              case 3:
10                 if (curr  ≤  stored_pin_len)
11                 then pc := 4;
12                 else pc := 6;
13             case 4:
14                 pin_ok := (curr  ≤  pin_len and pin(curr) =
                       stored_pin(curr)) ? pin_ok : false;
15                 pc := 5;
16             case 5:
```

```
17                  curr := curr+1;
18                  pc := 3;
19              case 6:
20                  skip; pc := 7;
21              case 7:
22                  return pin_ok;
23                  pc := 0; // Never reached
```

In this particular case the new obfuscated program is still constant-time, but we do not know if this is the case also for other programs. In general, we would like to have guarantees that the attacks prevented by the constant-time based countermeasure are not possible in the obfuscated versions.

3.3 A SHORT GUIDE TO CT-SIMULATIONS

Typically, proving the correctness of a compiler requires to prove the existence of a simulation relation between the computations at the source and at the target level [70, 139]. Indeed, if such a relation exists it is guaranteed that the behavior of the target program *refines* that of the source, meaning that they have the same observable behavior.

A general method for proving that constant-time is also preserved by compilation generalizes this approach and is based on the notion of *CT-simulation* [21]. It considers three relations: a simulation relation between source and target, and two equivalences, one between source and the other between target computations. The idea is to prove that, given two computations at source level that are equivalent, they are simulated by two equivalent computations at the target level. Actually, CT-simulations guarantee the preservation of a particular form of non-interference, called *observational non-interference*. In the rest of this section, we briefly survey observational non-interference and how CT-simulations preserve it.

The idea is to model the behavior of programs using a labeled transition system whose transitions have the form $a \xrightarrow{t} b$, where a and b are program configurations and t represents the leakage associated with the execution step between a and b. The semantics is assumed deterministic. Hereafter, let the configurations of the source programs be ranged over by $a, b, \ldots$ and those of the target programs be ranged over by $\mathbf{a}, \mathbf{b}, \ldots$. We will use the dot notation to refer to commands and state inside configurations, e.g., $a.cmd$ refers to the command part of the configuration a.

Recall from Section 2.1 that observables represent what an attacker learns by looking at the execution of a program. Observational non-interference is defined for complete executions (we denote S_f the set of final configurations) and w.r.t. an equivalence relation ϕ on configurations (e.g., states are equivalent on public variables):

Definition 3.1 (Observational non-interference (ONI) [21])**.** *A program P is* observationally non-interferent *w.r.t. a relation ϕ (written $p \models ONI(\phi)$),* iff *for all initial configurations $a, a' \in S_i$ and configurations b, b' and leakages t, t' and $n \in \mathbb{N}$,*

$$a \xrightarrow{t}{}^{n} b \wedge a' \xrightarrow{t'}{}^{n} b' \wedge \phi(a, a') \Rightarrow t = t' \wedge (b \in S_f \text{ iff } b' \in S_f).$$

Consider now a compiler $[\![\cdot]\!]^{S}_{\mathbf{T}}$ from S to $\mathbf{T}$. Intuitively, it preserves ONI when for every program P that enjoys ONI, $[\![P]\!]^{S}_{\mathbf{T}}$ does as well (cfr. Definition 2.8):

Definition 3.2. $[\![\cdot]\!]^{S}_{\mathbf{T}}$ *preserves ONI* iff*, for all programs P*

$$P \models ONI(\phi) \Rightarrow [\![P]\!]^{S}_{\mathbf{T}} \models ONI(\phi).$$

To show that a compiler $[\![\cdot]\!]^{S}_{T}$ is secure, we follow [21]. Actually, Barthe et al. [21] provide three different proof-techniques based on simulations and with a varying degree of generality: *lockstep*, *manystep* and *general CT-simulation*. Roughly, a lockstep simulation requires that each execution step of the source program has a corresponding one at the target. However, this is not applicable to our compiler, since a single instruction in the source is compiled to multiple ones in the target. Manystep simulation generalizes the notion of lockstep simulation by allowing multiple steps at the target for a single step at the source. Still, this is insufficient for control-flow flattening since it may happen that for one step at the source there are no steps at the target. Thus, we resort to a *general CT-simulation* and we build it in two steps.

The first step consists in defining a simulation, called *general simulation*, that relates computations between source and target languages. The idea is to consider related a source and a target configuration whenever, after they perform a certain number of steps, they end up in two still related configurations:

Definition 3.3 (General simulation [21]). *Let* $\text{num-steps}(\cdot,\cdot)$ *be a function mapping source and target configurations to* $\mathbb{N}$. *Also, let* $|\cdot|$ *be a function from source configurations to* $\mathbb{N}$. *The relation* $\approx_P$ *is a* general simulation *w.r.t.* $\text{num-steps}(\cdot,\cdot)$ *whenever:*

1. $(\forall a, b, \mathbf{a}.\ a \to b\ \wedge\ a \approx_P \mathbf{a} \Rightarrow (\exists \mathbf{b}.\ \mathbf{a} \longrightarrow^{\text{num-steps}(a,\mathbf{a})} \mathbf{b} \wedge b \approx_P \mathbf{b})$;
2. $(\forall a, b, \mathbf{a}.\ a \to b\ \wedge\ a \approx_P \mathbf{a}\ \wedge\ \text{num-steps}(a,\mathbf{a}) = 0 \Rightarrow |a| < |b|$;
3. *For any source configuration* $a \in S_f$ *and target configuration* $\mathbf{a}$ *such that* $a \approx_P \mathbf{a}$ *there exists a target configuration* $\mathbf{b} \in \mathbf{S_f}$ *such that* $\mathbf{a} \longrightarrow^{\text{num-steps}(a,\mathbf{a})} \mathbf{b}$ *and* $a \approx_P \mathbf{b}$.

Given two configurations a and $\mathbf{a}$ in the simulation relation, the function $\text{num-steps}(a,\mathbf{a})$ predicts how many steps $\mathbf{a}$ has to perform for reaching a configuration $\mathbf{b}$ related with the configuration b such that $a \to b$. When $\text{num-steps}(a,\mathbf{a}) = 0$, a possibly infinite sequence of source steps is simulated by an empty one at the target level. To avoid these situations the measure function $|\cdot|$ is introduced and the condition 2 of the above definition ensures that the measure of source configuration strictly decreases whenever the corresponding target one stutters.

The second step consists of introducing two equivalence relations between configurations: $\stackrel{c}{\equiv}_S$ relates configurations at the source and $\stackrel{c}{\equiv}_T$ at the target. These two relations form a *general CT-simulation*:

Definition 3.4 (General CT-simulation [21]). *A pair* $(\stackrel{c}{\equiv}_S, \stackrel{c}{\equiv}_T)$ *is a* general CT-simulation *w.r.t.* $\approx_P$, $\text{num-steps}(\cdot,\cdot)$ *and* $|\cdot|$ *whenever:*

1. $(\stackrel{c}{\equiv}_S, \stackrel{c}{\equiv}_T)$ *is a* manysteps CT-diagram, *i.e., if*
   - $a \stackrel{c}{\equiv}_S a'$ *and* $\mathbf{a} \stackrel{c}{\equiv}_T \mathbf{a}'$;
   - $a \xrightarrow{t} b$ *and* $a' \xrightarrow{t} b'$;
   - $\mathbf{a} \xrightarrow{\tau}{}^{\text{num-steps}(a,\mathbf{a})} \mathbf{b}$ *and* $\mathbf{a}' \xrightarrow{\tau'}{}^{\text{num-steps}(a',\mathbf{a}')} \mathbf{b}'$;
   - $a \approx_P \mathbf{a}$, $a' \approx_P \mathbf{a}'$, $b \approx_P \mathbf{b}$ *and* $b' \approx_P \mathbf{b}'$

   then
   - $\tau = \tau'$ *and* $\text{num-steps}(a,\mathbf{a}) = \text{num-steps}(a',\mathbf{a}')$;
   - $b \stackrel{c}{\equiv}_S b'$ *and* $\mathbf{b} \stackrel{c}{\equiv}_T \mathbf{b}'$;
2. *If* a, a' *are initial configurations, with targets* $\mathbf{a}, \mathbf{a}'$, *and* $\phi(a, a')$, *then* $a \stackrel{c}{\equiv}_S a'$ *and* $\mathbf{a} \stackrel{c}{\equiv}_T \mathbf{a}'$;

3. *If* $a \overset{c}{\equiv}_S a'$, *then* $a \in S_f \Leftrightarrow a' \in S_f$;

4. $(\overset{c}{\equiv}_S, \overset{c}{\equiv}_T)$ *is a* final CT-diagram *[21], i.e., if*
   - $a \overset{c}{\equiv}_S a'$ *and* $a \overset{c}{\equiv}_T a'$;
   - a *and* a' *are final;*
   - $a \overset{\tau}{\longrightarrow}_{\text{num-steps}(a,a)} b$ *and* $a' \overset{\tau'}{\longrightarrow}_{\text{num-steps}(a',a')} b'$;
   - $a \approx_P a$, $a' \approx_P a'$, $b \approx_P b$ *and* $b' \approx_P b'$

   then
   - $\tau = \tau'$ *and* $\text{num-steps}(a, a) = \text{num-steps}(a', a')$;
   - $b \overset{c}{\equiv}_T b'$ *and they are both final.*

The idea is that the relations $\overset{c}{\equiv}_S$ and $\overset{c}{\equiv}_T$ are stable under reduction, i.e., preservation of the observational non-interference is guaranteed. The following theorem, referred to in [21] as Theorem 6, gives a sufficient condition to establish constant-time preservation.

Theorem 3.1 (Security). *If* P *is constant-time w.r.t.* ϕ *and there is a general CT-simulation w.r.t. a general simulation, then* $[\![P]\!]^S_T$ *is constant-time w.r.t.* ϕ*.*

3.4 THE CASE OF CONTROL-FLOW FLATTENING

In this section, we apply the technique of *CT-simulations* to prove that control-flow flattening preserves constant-time policy. First, we introduce a small imperative language, its semantics in the form of a LTS and our leakage model. Then, we formalize our obfuscation as a function from syntax to syntax, and finally we prove the preservation of the security policy.

For the sake of presentation, our source language is rather essential, as well as our illustrative examples. As said above, the proof that control-flow flattening is indeed secure follows the approach of [21] (briefly presented in Section 3.3), and only needs paper-and-pencil on our neat, foundational setting. Intuitively, we prove that if two executions of a program on different secret values are indistinguishable (roughly, they take the same time), then also the executions of its obfuscated version are indistinguishable.

Note that extending our results to a richer language will only require to handle more details with no substantial changes in the structure of the proof itself. Furthermore, other security properties besides those already studied in [21] can be accommodated with no particular effort in this framework, and also other program transformations can be proved to preserve security in the same manner.

3.4.1 *The language and its (instrumented) semantics*

We consider a small imperative language with arithmetic and boolean expressions. Let *Var* be a set program identifiers, the syntax is

$$AExp \ni \mathfrak{e} ::= v \mid x \mid \mathfrak{e}_1 \texttt{ op } \mathfrak{e}_2 \qquad v \in \mathbb{Z}, \texttt{op} \in \{+, -, *, /, \texttt{mod}\}, x \in Var$$
$$BExp \ni \mathfrak{b} ::= \texttt{true} \mid \texttt{false} \mid \mathfrak{b}_1 \texttt{ or } \mathfrak{b}_2 \mid \texttt{not } \mathfrak{b} \mid \mathfrak{e}_1 \leq \mathfrak{e}_2 \mid \mathfrak{e}_1 = \mathfrak{e}_2$$
$$Cmd \ni c ::= \texttt{skip} \mid x := \mathfrak{e} \mid c_1 \texttt{;} \ c_2 \mid \texttt{if } \mathfrak{b} \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } \mathfrak{b} \texttt{ do } c$$

$$\frac{}{\mathtt{x} := \mathfrak{e}, \sigma \xrightarrow{leak(\mathfrak{e},\sigma)\cdot \mathtt{x}} \mathtt{skip}, \sigma\{\mathtt{x} \mapsto [\mathfrak{e}]_\sigma\}} \qquad \frac{c_1, \sigma \xrightarrow{t} c_1', \sigma'}{c_1\mathtt{;}\ c_2, \sigma \xrightarrow{t} c_1'\mathtt{;}\ c_2, \sigma'} \qquad \frac{}{\mathtt{skip;}\ c_2, \sigma \xrightarrow{\bullet} c_2, \sigma}$$

$$\frac{[\mathfrak{b}]_\sigma = true}{\mathtt{if}\ \mathfrak{b}\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \sigma \xrightarrow{leak(\mathfrak{b},\sigma)\cdot true} c_1, \sigma} \qquad \frac{[\mathfrak{b}]_\sigma = false}{\mathtt{if}\ \mathfrak{b}\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \sigma \xrightarrow{leak(\mathfrak{b},\sigma)\cdot false} c_2, \sigma}$$

$$\frac{}{\mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ c, \sigma \xrightarrow{\bullet} \mathtt{if}\ \mathfrak{b}\ \mathtt{then}\ (c\mathtt{;}\ \mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ c)\ \mathtt{else}\ \mathtt{skip}, \sigma}$$

Figure 5: Instrumented operational semantics for commands.

For technical reasons we assume that each command in the syntax carries a permanent color $\digamma$, either *white* or not. Also, we stipulate that each `while` statement and all its components get a unique non-white color, and that there is a function *color* yielding the color of a statement. Programs in the language are elements of *Cmd*.

Now, we define the semantics and instantiate the framework of Barthe et al. [21] to the *non-cancelling constant-time policy*, i.e., a variant of the constant-time policy where also the number of execution steps is observed. For that, we define a *leakage model*: we instantiate observables and traces from Chapter 2 to describe the information that an attacker observes during the execution (as already outlined in Example 2.1).

From now onward we follow [21] and denote with $\cdot$ the concatenation of traces. Arithmetic and boolean expressions leak the sequence of operations required to be evaluated; we assume that there is an observable $\underline{op}$, associated with the arithmetic operation being executed, but not with the logical ones (slightly simplifying [21]). Also, we denote with $\bullet$ absence of leaking. Our leakage model is defined by the following function $leak(\cdot, \cdot)$ that given an expression (either arithmetic or boolean) and a state returns the corresponding leakage:

$$\begin{aligned}
leak(v, \sigma) &= leak(\mathtt{x}, \sigma) = leak(\mathtt{true}, \sigma) = leak(\mathtt{false}, \sigma) = \bullet \\
leak(\mathtt{not}\ \mathfrak{b}, \sigma) &= leak(\mathfrak{b}, \sigma) \\
leak(\mathfrak{e}_1\ \mathtt{op}\ \mathfrak{e}_2, \sigma) &= leak(\mathfrak{e}_1, \sigma) \cdot leak(\mathfrak{e}_2, \sigma) \cdot \underline{op} \\
leak(\mathfrak{e}_1 \leq \mathfrak{e}_2, \sigma) &= leak(\mathfrak{e}_1 = \mathfrak{e}_2, \sigma) = leak(\mathfrak{e}_1, \sigma) \cdot leak(\mathfrak{e}_2, \sigma) \\
leak(\mathfrak{b}_1\ \mathtt{or}\ \mathfrak{b}_2, \sigma) &= leak(\mathfrak{b}_1, \sigma) \cdot leak(\mathfrak{b}_2, \sigma)
\end{aligned}$$

Accesses to constants and identifiers leak nothing; boolean and relational expressions leak the concatenation of the leaks of their sub-expressions; the arithmetic expressions append the observable of the applied operator to the leaks of their sub-expressions.

We omit the semantics of arithmetic and boolean expression $[\cdot]_\sigma$ because fully standard [167]; we only assume that each syntactic arithmetic operator `op` has a corresponding semantic operator *op*.

As anticipated in Section 3.3, the semantics of commands is given as a transition relation $\xrightarrow{t}$ between configurations where t is the leakage of that transition step. As usual a configuration is a pair c, σ consisting of a command and a state $\sigma \in Store$ assigning values to program identifiers. Given a program c, the set of initial configurations is $S_i = \{c, \sigma \mid \sigma \in Store\}$, and that of final configurations is $S_f = \{\mathtt{skip}, \sigma \mid \sigma \in Store\}$.

Figure 5 shows the instrumented semantics of the language. The semantics is assumed to keep colors, in particular in the rule for a $\digamma$-colored `while`, all the components of the `if` in the target are also $\digamma$-colored, avoiding color clashes (see the pdf for colors).

3.4.2 *Control-flow flattening formalization*

For the sake of presentation, we will adopt the sugared syntax we used above and represent a sequence of nested conditionals in the obfuscated program as the command $\texttt{switch}\ \mathfrak{e} : \texttt{ls}$, where $\texttt{ls} = [(\mathtt{v}_1, \mathtt{c}_1);\ \ldots;\ (\mathtt{v}_n : \mathtt{c}_n)]$, with semantics

$$\frac{([\mathfrak{e}]_\sigma, \mathtt{c}) \notin \texttt{ls}}{\texttt{switch}\ \mathfrak{e} : \texttt{ls}, \sigma \xrightarrow{leak(\mathfrak{e},\sigma)} \texttt{skip}, \sigma} \qquad \frac{([\mathfrak{e}]_\sigma, \mathtt{c}) \in \texttt{ls}}{\texttt{switch}\ \mathfrak{e} : \texttt{ls}, \sigma \xrightarrow{leak(\mathfrak{e},\sigma)} \mathtt{c}, \sigma}$$

Now, let $\texttt{pc}$ be a fresh identifier, called *program counter*. Then, following [39], the obfuscated version $[\![\mathtt{c}]\!]$ of the command $\mathtt{c}$ is

$$\begin{aligned}
&\texttt{pc} := 1;\\
&\texttt{while}\ 1 \le \texttt{pc}\ \texttt{do}\\
&\qquad \texttt{switch}\ \texttt{pc} : labeled(\texttt{pc}, \mathtt{c}, 1, 0)
\end{aligned}$$

where:

$$\begin{aligned}
&labeled(\texttt{pc}, \texttt{skip}, n, m) = [(n, \texttt{skip};\ \texttt{pc} := m)]\\
&labeled(\texttt{pc}, \mathtt{x} := \mathfrak{e}, n, m) = [(n, \mathtt{x} := [\![\mathfrak{e}]\!]_{\mathfrak{e}};\ \texttt{pc} := m)]\\
&labeled(\texttt{pc}, \mathtt{c}_1;\ \mathtt{c}_2, n, m) =\\
&\qquad labeled(\texttt{pc}, \mathtt{c}_1, n, n + size(\mathtt{c}_1)) \cdot labeled(\texttt{pc}, \mathtt{c}_2, n + size(\mathtt{c}_1), m)\\
&labeled(\texttt{pc}, \texttt{if}\ \mathfrak{b}\ \texttt{then}\ \mathtt{c}_1\ \texttt{else}\ \mathtt{c}_2, n, m) =\\
&\qquad [(n, \texttt{if}\ [\![\mathfrak{b}]\!]_{\mathfrak{b}}\ \texttt{then}\ \texttt{pc} := n + 1\ \texttt{else}\ \texttt{pc} := n + 1 + size(\mathtt{c}_1))] \cdot\\
&\qquad labeled(\texttt{pc}, \mathtt{c}_1, n + 1, m) \cdot labeled(\texttt{pc}, \mathtt{c}_2, n + 1 + size(\mathtt{c}_1), m)\\
&labeled(\texttt{pc}, \texttt{while}\ \mathfrak{b}\ \texttt{do}\ \mathtt{c}, n, m) =\\
&\qquad [(n, \texttt{if}\ [\![\mathfrak{b}]\!]_{\mathfrak{e}}\ \texttt{then}\ \texttt{pc} := n + 1\ \texttt{else}\ \texttt{pc} := n + 1 + size(\mathtt{c}))] \cdot\\
&\qquad labeled(\texttt{pc}, \mathtt{c}, n + 1, n) \cdot [(n + 1 + size(\mathtt{c}), \texttt{skip};\ \texttt{pc} := m)]
\end{aligned}$$

with $size(\cdot)$ defined as follows

$$\begin{aligned}
size(\mathtt{c}) &= 1 \text{ if } \mathtt{c} \in \{\texttt{skip}, \cdot := \cdot\}\\
size(\mathtt{c}_1;\ \mathtt{c}_2) &= size(\mathtt{c}_1) + size(\mathtt{c}_2)\\
size(\texttt{if}\ \mathfrak{b}\ \texttt{then}\ \mathtt{c}_1\ \texttt{else}\ \mathtt{c}_2) &= 1 + size(\mathtt{c}_1) + size(\mathtt{c}_2)\\
size(\texttt{while}\ \mathfrak{b}\ \texttt{do}\ \mathtt{c}) &= 2 + size(\mathtt{c})
\end{aligned}$$

and

$$\begin{aligned}
[\![\texttt{not}\ \mathfrak{b}]\!]_{\mathfrak{b}} &= \texttt{not}\ [\![\mathfrak{b}]\!]_{\mathfrak{b}} & [\![\texttt{true}]\!]_{\mathfrak{b}} &= \texttt{true}\\
[\![\mathfrak{e}_1 \le \mathfrak{e}_2]\!]_{\mathfrak{b}} &= [\![\mathfrak{e}_1]\!]_{\mathfrak{e}} \le [\![\mathfrak{e}_2]\!]_{\mathfrak{e}} & [\![\texttt{false}]\!]_{\mathfrak{b}} &= \texttt{false}\\
[\![\mathfrak{e}_1 = \mathfrak{e}_2]\!]_{\mathfrak{b}} &= [\![\mathfrak{e}_1]\!]_{\mathfrak{e}} = [\![\mathfrak{e}_2]\!]_{\mathfrak{e}} & [\![\mathfrak{b}_1\ \texttt{or}\ \mathfrak{b}_2]\!]_{\mathfrak{b}} &= [\![\mathfrak{b}_1]\!]_{\mathfrak{b}}\ \texttt{or}\ [\![\mathfrak{b}_2]\!]_{\mathfrak{e}}
\end{aligned}$$

$$\begin{aligned}
[\![v]\!]_{\mathfrak{e}} &= v\\
[\![\mathtt{x}]\!]_{\mathfrak{e}} &= \mathtt{x}\\
[\![\mathfrak{e}_1\ \texttt{op}\ \mathfrak{e}_2]\!]_{\mathfrak{e}} &= [\![\mathfrak{e}_1]\!]_{\mathfrak{e}}\ \texttt{op}\ [\![\mathfrak{e}_2]\!]_{\mathfrak{e}}
\end{aligned}$$

The obfuscated version of a command c is a loop with condition $1 \leq$ pc and with body a switch statement. The switch condition is on the values of pc and its cases correspond to the flattened statements, obtained from the function $labeled(\texttt{pc}, \texttt{c}, n, m)$. It returns a list containing the cases of the switch and it is inductively defined on the syntax of commands: the first parameter pc is the identifier to use as program counter; the second is the command c to be flattened; the parameter n represents the value of the guard of the case generated for the first statement of c; and the last parameter m represents the value to be assigned to pc by the last case of the generated switch. For example, the flattening of a sequence c_1; c_2 generates the cases corresponding to c_1 and c_2, and then concatenates them. Note that the values of the program counter for the cases of c_2 start from the value assigned to pc by the last case generated for c_1, i.e., $n + size(c_1)$, where the function $size(\cdot)$ returns the "length" of c_1. For a program c, we use 1 as initial value of n and 0 as last value to be assigned so as to exit from the while loop.

3.4.3 *Correctness and security*

Since the obfuscation does not change the language (apart from sugaring nested if commands); the operational semantics is deterministic; and there are no *unsafe* programs (i.e., a program gets stuck *iff* execution has completed), the correctness of obfuscation directly follows from the existence of a general simulation between the source and the target languages [21]. For that, inspired by [39], we define the relation $\approx_p$ between source and target configurations shown in Figure 6. Intuitively, the relation $\approx_p$ matches source and target configurations with the same behavior, depending on whether they are final (third rule), their execution originated from a loop (Rule (Colored)) or not (Rule (White)). Note that we differentiate white and colored cases as to avoid circular reasoning in the derivations of $\approx_p$. More specifically, our relation matches a configuration a in the source with a corresponding **a** in the target. Actually, **a**.cmd is the while loop of the obfuscated program (fourth premise in Rule (White) and third in Rule (Colored)), whereas **a**.σ is equal to a.σ except for the binding of pc. Its value is mapped to the case of the switch corresponding to the next command in a (first premise in Rule (White) and fifth in Rule (Colored)).

To understand how our simulation works, recall Example 3.1 (Page 17). By Rule (White) we relate the configuration reached at Line 3 at the source level with that of the obfuscated program starting at Line 2 and with a state equal to that of the source level with the additional binding pc $\mapsto 3$. Similarly, we relate the configuration reached at Line 6 at the source level and its obfuscated counterpart (again at Line 2 at the obfuscated level), using Rule (Colored) and noting that the source configuration derives from the execution of a loop.

Now, to prove the obfuscation correct we need to show that $\approx_p$ is a general simulation according to Definition 3.3. Before doing that, we give the definition of the function $\text{num-steps}(\text{a}, \textbf{a})$. Intuitively, given b such that a $\to$ b, this function maps the configurations a and **a** into the number of computation steps needed for **a** to reach a configuration **b** in relation $\approx_p$ with b [21]:

(WHITE)

$$\frac{\mathtt{ls} = \mathit{labeled}(\mathtt{pc}, \mathtt{p}, 1, 0) \quad \begin{array}{c}\mathit{color}(\mathtt{c}) = \mathit{white} \quad \sigma' = \sigma \cup \{\mathtt{pc} \mapsto n\} \\ \mathbf{c}' = \mathtt{while}\ (1 \le \mathtt{pc})\ \mathtt{do}\ (\mathtt{switch}\ \mathtt{pc} : \mathtt{ls})\end{array} \quad -, \mathtt{pc} \vdash c \bowtie \mathtt{ls}[n], m}{\mathtt{c}, \sigma \approx_{\mathtt{p}} \mathbf{c}', \sigma'}$$

(COLORED)

$$\frac{\begin{array}{c}\sigma' = \sigma \cup \{\mathtt{pc} \mapsto n\} \quad \mathtt{ls} = \mathit{labeled}(\mathtt{pc}, \mathtt{p}, 1, 0) \quad \mathbf{c}' = \mathtt{while}\ (1 \le \mathtt{pc})\ \mathtt{do}\ (\mathtt{switch}\ \mathtt{pc} : \mathtt{ls}) \\ \mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ \mathtt{c}'' \in \mathtt{p} \quad \mathit{color}(\mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ \mathtt{c}'') = \mathit{color}(\mathtt{c}) \neq \mathit{white} \\ -, \mathtt{pc} \vdash \mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ \mathtt{c}'' \bowtie \mathtt{ls}[n_0], m' \quad n_0, \mathtt{pc} \vdash \mathtt{c} \diamond \mathtt{ls}[n], m\end{array}}{\mathtt{c}, \sigma \approx_{\mathtt{p}} \mathbf{c}', \sigma'}$$

$$\frac{\sigma' = \sigma \cup \{\mathtt{pc} \mapsto n\}}{\mathtt{skip}, \sigma \approx_{\mathtt{p}} \mathtt{skip}, \sigma'}$$

$$\frac{}{n_0, \mathtt{pc} \vdash \mathtt{skip} \sim \mathtt{ls}[n], m} \qquad \frac{\mathtt{ls}[n] = (n, \mathtt{x} := \mathfrak{e};\ \mathtt{pc} := m)}{n_0, \mathtt{pc} \vdash \mathtt{x} := \mathfrak{e} \sim \mathtt{ls}[n], m}$$

$$\frac{n_0, \mathtt{pc} \vdash \mathtt{c}_1 \sim \mathtt{ls}[n], m' \quad n_0, \mathtt{pc} \vdash \mathtt{c}_2 \sim \mathtt{ls}[m'], m}{n_0, \mathtt{pc} \vdash \mathtt{c}_1;\ \mathtt{c}_2 \sim \mathtt{ls}[n], m}$$

$$\frac{\begin{array}{c}\mathtt{ls}[n] = (n, \mathtt{if}\ \mathfrak{b}\ \mathtt{then}\ \mathtt{pc} := n + 1\ \mathtt{else}\ \mathtt{pc} := n + 1 + \mathit{size}(\mathtt{c}_1)) \\ n_0, \mathtt{pc} \vdash \mathtt{c}_1 \sim \mathtt{ls}[n+1], m \quad n_0, \mathtt{pc} \vdash \mathtt{c}_2 \sim \mathtt{ls}[n + 1 + \mathit{size}(\mathtt{c}_1)], m\end{array}}{n_0, \mathtt{pc} \vdash \mathtt{if}\ \mathfrak{b}\ \mathtt{then}\ \mathtt{c}_1\ \mathtt{else}\ \mathtt{c}_2 \sim \mathtt{ls}[n], m} \qquad \frac{\mathtt{ls}[n] = (n, \mathtt{skip};\ \mathtt{pc} := n_0)}{n_0, \mathtt{pc} \vdash \mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ \mathtt{c} \diamond \mathtt{ls}[n], n_0}$$

$$\frac{}{n_0, \mathtt{pc} \vdash \mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ \mathtt{c} \diamond \mathtt{ls}[n_0], n_0} \qquad \frac{n, \mathtt{pc} \vdash \mathtt{if}\ \mathfrak{b}\ \mathtt{then}\ (\mathtt{c};\ \mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ \mathtt{c})\ \mathtt{else}\ \mathtt{skip} \diamond \mathtt{ls}[n], m}{-, \mathtt{pc} \vdash \mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ \mathtt{c} \bowtie \mathtt{ls}[n], m}$$

where $\sim \in \{\diamond, \bowtie\}$, and the first parameter (n_0) is immaterial in $\bowtie$.

Figure 6: Definition of $\approx_{\mathtt{p}}$ relation on configurations and its auxiliary relations.

Definition 3.5. *For any* $\mathtt{a}$, $\mathbf{a}$ *source and target configurations:*

$$\text{num-steps}(\mathtt{a}, \mathbf{a}) \triangleq \begin{cases} 0 & \text{if } \mathtt{a}.\mathit{cmd} \in \{\mathtt{skip};\ \cdot, \mathtt{while}\ \cdot\ \mathtt{do}\ \cdot\} \\ \text{num-steps}((\mathtt{c}_1, \mathtt{a}.\sigma), \mathbf{a}) & \text{if } \mathtt{a}.\mathit{cmd} = \mathtt{c}_1;\ \mathtt{c}_2 \wedge \mathtt{c}_1 \neq \mathtt{skip} \\ 9 & \text{if } \mathtt{a}.\mathit{cmd} \in \{\mathtt{if}\ \cdot\ \mathtt{then}\ \cdot\ \mathtt{else}\ \cdot\} \\ 7 & o.w. \end{cases}$$

We also define the measure $|\cdot|$ that it used to guarantee that an infinite number of steps at the source level is not matched by a finite number of steps at the target [21, 39]:

Definition 3.6. *For any configuration* $\mathtt{a}$*:*

$$|\mathtt{a}| \triangleq \begin{cases} 2 \cdot |(\mathtt{c}, \mathtt{a}.\sigma)| + 3 & \text{if } \mathtt{a}.\mathit{cmd} = \mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ \mathtt{c} \\ |(\mathtt{c}_2, \mathtt{a}.\sigma)| + 1 & \text{if } \mathtt{a}.\mathit{cmd} = \mathtt{skip};\ \mathtt{c}_2 \\ 0 & o.w. \end{cases}$$

Note that the measure $|\cdot|$ was built with the specific requirements of the proof of Theorem 3.2, i.e., to ensure that $(\forall \mathtt{b}, \mathbf{a}.\ \mathtt{a} \to \mathtt{b}\ \wedge\ \mathtt{a} \approx_{\mathtt{p}} \mathbf{a}\ \wedge\ \text{num-steps}(\mathtt{a}, \mathbf{a}) = 0 \Rightarrow |\mathtt{b}| < |\mathtt{a}|)$. Finally:

Theorem 3.2. *For all programs* $\mathtt{p}$*, the relation* $\approx_{\mathtt{p}}$ *is a general simulation.*

Proof. We show that $\approx_p$ satisfies the three properties of Definition 3.3:

1. $(\forall a, b, \mathbf{a}.\ a \to b \ \wedge\ a \approx_c \mathbf{a} \Rightarrow (\exists \mathbf{b}.\ \mathbf{a} \longrightarrow^{\text{num-steps}(a,\mathbf{a})} \mathbf{b} \wedge b \approx_p \mathbf{b})$.

   The proof goes by induction on the rules of the operational semantics for the configuration a.

   Note that – by definition of $\approx_p$ – any configuration $\mathbf{a}$ related with another a must be such that $\mathbf{a}.\sigma = a.\sigma \cup \{\texttt{pc} \mapsto n\}$ and

$$\mathbf{a}.cmd = \texttt{while}\ 1 \le \texttt{pc}\ \texttt{do}$$
$$\texttt{switch pc} : \texttt{ls}$$

   for some n and $\texttt{ls} = labeled(\texttt{pc}, p, 1, 0)$.

   BASE CASE: $a.cmd = x := \mathfrak{e}$. By definition of $\to$ we know that:

$$a \to b = \texttt{skip}, a.\sigma[x \mapsto [e]_{a.\sigma}].$$

   We have two exhaustive cases, depending on $color(a.cmd)$:

   a) **Case** $color(a.cmd) = white$.
   By Rule (WHITE) we know that for some m, $-, \texttt{pc} \vdash x := \mathfrak{e} \bowtie \texttt{ls}[n], m$.

   From definitions of *labeled* and $\bowtie$ it follows that $\texttt{ls}[n] = (n, x := [\![\mathfrak{e}]\!]_{\mathfrak{e}}\texttt{;}\ \texttt{pc} := m)$ and $\mathbf{a} \longrightarrow^{\text{num-steps}(a,\mathbf{a})} \mathbf{b} = \mathbf{a}.cmd, \sigma[x \mapsto [[\![\mathfrak{e}]\!]_{\mathfrak{e}}]_{\mathbf{a}.\sigma}, \texttt{pc} \mapsto m]$. Finally, we can show that $b \approx_p \mathbf{b}$, since Rule (WHITE) applies again (recall that the color is permanent).

   b) **Case** $color(a.cmd) \neq white$.
   Similarly to the case above, by Rule (COLORED) we have $-, \texttt{pc} \vdash \texttt{while}\ \mathfrak{b}\ \texttt{do}\ c'' \bowtie \texttt{ls}[n_0], m'$ and $n_0, \texttt{pc} \vdash x := \mathfrak{e} \diamond \texttt{ls}[n], m$, for some m', m.

   By definition of *labeled* and $\diamond$, we know that $\texttt{ls}[n] = (n, x := [\![\mathfrak{e}]\!]_{\mathfrak{e}}\texttt{;}\ \texttt{pc} := m)$, thus $\mathbf{a} \longrightarrow^{\text{num-steps}(a,\mathbf{a})} \mathbf{b} = \mathbf{a}.cmd, \sigma[x \mapsto [[\![\mathfrak{e}]\!]_{\mathfrak{e}}]_{\mathbf{a}.\sigma}, \texttt{pc} \mapsto m]$. The thesis then follows by definition of $\approx_p$.

   BASE CASE: $a.cmd = \texttt{skip;}\ c_2$.
   By definition of $\to$ we know that:

$$a \to b = c_2, a.\sigma.$$

   We have two exhaustive cases, depending on $color(a.cmd)$:

   a) **Case** $color(a.cmd) = white$.
   By Rule (WHITE) we know that $-, \texttt{pc} \vdash \texttt{skip;}\ c_2 \bowtie \texttt{ls}[n], m$, i.e., $-, \texttt{pc} \vdash \texttt{skip} \bowtie \texttt{ls}[n], m'$ and $-, \texttt{pc} \vdash c_2 \bowtie \texttt{ls}[m'], m$ for some m, m'. By definition of num-steps$(a, \mathbf{a})$, it follows that $\mathbf{a} \longrightarrow^{\text{num-steps}(a,\mathbf{a})} \mathbf{b}$, with $\mathbf{b} = \texttt{while}\ 1 \le \texttt{pc}\ \texttt{do switch pc} : \texttt{ls}, \sigma[\texttt{pc} \mapsto m']$. Thus, $b \approx_p \mathbf{b}$ trivially holds.

   b) **Case** $color(a.cmd) \neq white$.
   Analogous to the above.

   BASE CASE: $a.cmd = \texttt{while}\ \mathfrak{b}\ \texttt{do}\ c$.
   By definition of $\to$ we know that:

$$a \to b = \texttt{if}\ \mathfrak{b}\ \texttt{then}\ (c\texttt{;}\ \texttt{while}\ \mathfrak{b}\ \texttt{do}\ c)\ \texttt{else skip}, \sigma[x \mapsto [e]_{a.\sigma}].$$

   Again, we have two exhaustive cases, depending on $color(a.cmd)$:

a) **Case** $\mathit{color}(\mathtt{a}.\mathit{cmd}) = \mathit{white}$. By Rule (White) of $\approx_{\mathsf{p}}$ we know that $-, \mathtt{pc} \vdash \mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ \mathtt{c} \bowtie \mathtt{ls}[n], m$, i.e., that

$$n, \mathtt{pc} \vdash \mathtt{if}\ \mathfrak{b}\ \mathtt{then}\ (\mathtt{c;}\ \mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ \mathtt{c})\ \mathtt{else}\ \mathtt{skip} \diamond \mathtt{ls}[n], m. \quad (1)$$

Since $\text{num-steps}(\mathtt{a}, \mathbf{a}) = 0$, then $\mathtt{b} = \mathtt{a}$ and we must show that $\mathtt{b} \approx_{\mathsf{p}} \mathbf{b}$. Indeed, we know $\mathit{color}(\mathtt{b}.\mathit{cmd}) \neq \mathit{white}$ (since it derives from $\mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ \mathtt{c}$), so it suffices to show the following facts

- $\mathbf{b}.\sigma = \mathtt{b}.\sigma \cup \{\mathtt{pc} \mapsto n\}$ and $\mathbf{b}.\mathit{cmd} = \mathtt{while}\ \mathit{1} \leq \mathtt{pc}\ \mathtt{do}\ (\mathtt{switch}\ \mathtt{pc} : \mathtt{ls})$ with $\mathtt{ls} = \mathit{labeled}(\mathtt{pc}, \mathsf{p}, 1, 0)$, that directly follows from $\mathtt{b} = \mathtt{a}$;
- $-, \mathtt{pc} \vdash \mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ \mathtt{c} \bowtie \mathtt{ls}[n_0], m'$, following from (1) and by definition of $\diamond$ since $n_0 = n$, because $\mathbf{a}.\sigma(\mathtt{pc}) = n$, and by choosing $m' = m$;
- $n_0, \mathtt{pc} \vdash \mathtt{if}\ \mathfrak{b}\ \mathtt{then}\ (\mathtt{c;}\ \mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ \mathtt{c})\ \mathtt{else}\ \mathtt{skip} \diamond \mathtt{ls}[n], m$, following directly from the hypotheses.

b) **Case** $\mathit{color}(\mathtt{a}.\mathit{cmd}) \neq \mathit{white}$. Analogous to the above.

Base case: $\mathtt{a}.\mathit{cmd} = \mathtt{if}\ \mathfrak{b}\ \mathtt{then}\ \mathtt{c}_1\ \mathtt{else}\ \mathtt{c}_2$ and $[\mathfrak{b}]_{\mathtt{a}.\sigma} = \mathit{true}$.
Mutatis mutandis.

Base case: $\mathtt{a}.\mathit{cmd} = \mathtt{if}\ \mathfrak{b}\ \mathtt{then}\ \mathtt{c}_1\ \mathtt{else}\ \mathtt{c}_2$ and $[\mathfrak{b}]_{\mathtt{a}.\sigma} = \mathit{false}$.
Mutatis mutandis.

Inductive step: $\mathtt{a}.\mathit{cmd} = \mathtt{c}_1\mathtt{;}\ \mathtt{c}_2$, $\mathtt{c}_1 \neq \mathtt{skip}$.
The induction hypothesis (IHP) reads as follows

$$\forall \mathbf{a}'.\ \mathtt{c}_1, \sigma \to \mathtt{c}_1', \sigma' \wedge \mathtt{c}_1, \sigma \approx_{\mathsf{p}} \mathbf{a}' \Rightarrow (\exists \mathbf{b}'.\ \mathbf{a}' \longrightarrow^{\text{num-steps}((\mathtt{c}_1,\sigma),\mathbf{a}')} \mathbf{b}' \Rightarrow \mathtt{c}_1', \sigma' \approx_{\mathsf{p}} \mathbf{b}')$$

and we have to prove that

$$\forall \mathbf{a}.\ \mathtt{c}_1\mathtt{;}\ \mathtt{c}_2, \sigma \to \mathtt{c}_1'\mathtt{;}\ \mathtt{c}_2, \sigma' \wedge \mathtt{c}_1\mathtt{;}\ \mathtt{c}_2, \sigma \approx_{\mathsf{p}} \mathbf{a} \Rightarrow (\exists \mathbf{b}.\ a \longrightarrow^{\text{num-steps}((\mathtt{c}_1\mathtt{;}\ \mathtt{c}_2,\sigma),\mathbf{a})} \mathbf{b} \Rightarrow \mathtt{c}_1'\mathtt{;}\ \mathtt{c}_2, \sigma \approx_{\mathsf{p}} \mathbf{b}).$$

Again, we have two exhaustive cases, depending on $\mathit{color}(\mathtt{a}.\mathit{cmd})$:

a) **Case** $\mathit{color}(\mathtt{a}.\mathit{cmd}) \neq \mathit{white}$. Note that it must be $\mathbf{a} = \mathbf{a}'$ since they coincide both on commands (by definition of $\approx_{\mathsf{p}}$) and on the store. Also, by the premises of Rule (Colored) we have $-, \mathtt{pc} \vdash \mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ \mathtt{c}'' \bowtie \mathtt{ls}[n_0], m'$ and $n_0, \mathtt{pc} \vdash \mathtt{c}_1\mathtt{;}\ \mathtt{c}_2 \diamond \mathtt{ls}[n], m$. Since by definition $\text{num-steps}((\mathtt{c}_1\mathtt{;}\ \mathtt{c}_2, \sigma), \mathbf{a}) = \text{num-steps}((\mathtt{c}_1, \sigma), \mathbf{a}')$, the operational semantics is deterministic and $\mathbf{a} = \mathbf{a}'$, we have that $\mathbf{b} = \mathbf{b}'$. So, since $\mathit{color}(\mathtt{c}_1') \neq \mathit{white}$, to prove that $\mathtt{c}_1'\mathtt{;}\ \mathtt{c}_2, \sigma \approx_{\mathsf{p}} \mathbf{b}$, it remains to prove the following:

- $-, \mathtt{pc} \vdash \mathtt{while}\ \mathfrak{b}\ \mathtt{do}\ \mathtt{c}'' \bowtie \mathtt{ls}[n_0], m''$ holds by hypothesis with $m'' = m'$;
- $n_0, \mathtt{pc} \vdash \mathtt{c}_1'\mathtt{;}\ \mathtt{c}_2 \diamond \mathtt{ls}[n'], m$ that follows by (IHP) that guarantees that $n_0, \mathtt{pc} \vdash \mathtt{c}_1' \diamond \mathtt{ls}[n_1], m_1$ and by the condition $\mathtt{c}_1\mathtt{;}\ \mathtt{c}_2, \sigma \approx_{\mathsf{p}} \mathbf{a}$ that ensures $n_0, \mathtt{pc} \vdash \mathtt{c}_2 \diamond \mathtt{ls}[m_1], m$.

b) **Case** $\mathit{color}(\mathtt{a}.\mathit{cmd}) = \mathit{white}$. Analogous to the case above.

2. $(\forall \mathtt{a}, \mathtt{b}, \mathbf{a}.\ a \to b \wedge \mathtt{a} \approx_{\mathsf{p}} \mathbf{a} \wedge \text{num-steps}(\mathtt{a}, \mathbf{a}) = 0 \Rightarrow |\mathtt{b}| < |\mathtt{a}|)$.
By construction of the measure function $|\cdot|$.

3. For any final a and obfuscated a there exists a final b such that $\mathsf{a} \longrightarrow^{\text{num-steps}(\mathsf{a},\mathsf{a})} \mathsf{b} \Rightarrow \mathsf{a} \approx_{\mathsf{p}} \mathsf{b}$. Trivial. □

The correctness of the obfuscation now follows as a corollary of Theorem 3.2.

Corollary 3.1 (Correctness [21]). *For all commands* c *and store* σ

$$\mathsf{c}, \sigma \to^* \texttt{skip}, \sigma' \quad \textit{iff} \quad [\![\mathsf{c}]\!], \sigma \to^* \texttt{skip}, \sigma'.$$

The next step is showing that the control-flow flattening obfuscation preserves the constant-time programming policy. For that we follow [21] by defining $\stackrel{\mathsf{c}}{\equiv}$ as

Definition 3.7. *Let a and a' be two configurations, then* $a \stackrel{\mathsf{c}}{\equiv} a'$ iff $a.cmd = a'.cmd$

and showing that $(\stackrel{\mathsf{c}}{\equiv}, \stackrel{\mathsf{c}}{\equiv})$ is a *general CT-simulation* w.r.t. $\approx_{\mathsf{p}}$, $\text{num-steps}(\cdot,\cdot)$ and $|\cdot|$ (Definition 3.4).

To prove that $\stackrel{\mathsf{c}}{\equiv}$ adheres to the definitions above we need two technical lemmata. The first lemma proves that $\stackrel{\mathsf{c}}{\equiv}$ is an invariant of the instrumented semantics of the language:

Lemma 3.1. *Let a, a', b, b' be source or target configurations. If* $a \stackrel{\mathsf{c}}{\equiv} a'$, $a \xrightarrow{t} b$ *and* $a' \xrightarrow{t} b'$ *then* $b \stackrel{\mathsf{c}}{\equiv} b'$.

Proof. Follows directly by case analysis on $a.cmd$ (which is equal to $a'.cmd$ by Definition 3.7). □

The second lemma shows that $\text{num-steps}(\cdot,\cdot)$ gives the same values for configurations equivalent according to $\stackrel{\mathsf{c}}{\equiv}$:

Lemma 3.2. *If* $\mathsf{a} \stackrel{\mathsf{c}}{\equiv} \mathsf{a}'$, $\mathsf{a} \stackrel{\mathsf{c}}{\equiv} \mathsf{a}'$ *then* $\text{num-steps}(\mathsf{a},\mathsf{a}) = \text{num-steps}(\mathsf{a}',\mathsf{a}')$.

Proof. This lemma follows from the fact that the definition of the function $\text{num-steps}(\cdot,\cdot)$ is defined inductively on the syntax of a, a', a and a'. □

We can now state and prove the following theorem:

Theorem 3.3. *The pair* $(\stackrel{\mathsf{c}}{\equiv}, \stackrel{\mathsf{c}}{\equiv})$ *is a* general CT-simulation *w.r.t.* $\approx_{\mathsf{p}}$, $\text{num-steps}(\cdot,\cdot)$ *and* $|\cdot|$.

Proof. We show that $(\stackrel{\mathsf{c}}{\equiv}, \stackrel{\mathsf{c}}{\equiv})$ satisfies the four properties of Definition 3.4:

1. $(\stackrel{\mathsf{c}}{\equiv}, \stackrel{\mathsf{c}}{\equiv})$ is a *manysteps CT-diagram*.
   The definition of *manysteps CT-diagram* states that, if
   - $\mathsf{a} \stackrel{\mathsf{c}}{\equiv}_s \mathsf{a}'$ and $\mathsf{a} \stackrel{\mathsf{c}}{\equiv}_t \mathsf{a}'$;
   - $\mathsf{a} \xrightarrow{t} \mathsf{b}$ and $\mathsf{a}' \xrightarrow{t} \mathsf{b}'$;
   - $\mathsf{a} \xrightarrow{\tau}{}^{\text{num-steps}(\mathsf{a},\mathsf{a})} \mathsf{b}$ and $\mathsf{a}' \xrightarrow{\tau'}{}^{\text{num-steps}(\mathsf{a}',\mathsf{a}')} \mathsf{b}'$;
   - $\mathsf{a} \approx_{\mathsf{p}} \mathsf{a}$, $\mathsf{a}' \approx_{\mathsf{p}} \mathsf{a}'$, $\mathsf{b} \approx_{\mathsf{p}} \mathsf{b}$ and $\mathsf{b}' \approx_{\mathsf{p}} \mathsf{b}'$

   then
   - $\tau = \tau'$ and $\text{num-steps}(\mathsf{a},\mathsf{a}) = \text{num-steps}(\mathsf{a}',\mathsf{a}')$;
   - $\mathsf{b} \stackrel{\mathsf{c}}{\equiv} \mathsf{b}'$ and $\mathsf{b} \stackrel{\mathsf{c}}{\equiv} \mathsf{b}'$;

   The equality of τ and τ' directly follows from the fact that a and a' are syntactically the same by hypothesis and are the obfuscated versions of two configurations that generate the same observable t. From Lemma 3.2 we can derive $\text{num-steps}(\mathsf{a},\mathsf{a}) = \text{num-steps}(\mathsf{a}',\mathsf{a}')$. Finally, Lemma 3.1 entails the last two theses.

2. $\forall c, \sigma, \sigma'.\ \phi((c,\sigma),(c,\sigma')) \Rightarrow (c,\sigma) \stackrel{c}{\equiv} (c,\sigma') \ \wedge\ ([\![c]\!],\sigma) \stackrel{c}{\equiv} ([\![c]\!],\sigma')$ (with σ and σ' in common between the source and the target programs).
   Easily follows from the definition of $\stackrel{c}{\equiv}$ that just requires syntactic equality between configurations.

3. $a \stackrel{c}{\equiv} a' \Rightarrow (a \in S_f \Leftrightarrow a' \in S_f)$.
   Again, follows from definition of $\stackrel{c}{\equiv}$ and S_f.

4. $(\stackrel{c}{\equiv}, \stackrel{c}{\equiv})$ is a *final CT-diagram*.
   The definition of *final CT-diagrams* states that, if
   - $a \stackrel{c}{\equiv}_s a'$ and $a \stackrel{c}{\equiv}_t a'$;
   - $a \in S_f$ and $a' \in S_f$;
   - $a \xrightarrow{\tau}{}^{\text{num-steps}(a,a)}\ b$ and $a' \xrightarrow{\tau'}{}^{\text{num-steps}(a',a')}\ b'$;
   - $a \approx_p a$, $a' \approx_p a'$, $b \approx_p b$ and $b' \approx_p b'$

   then
   - $\tau = \tau'$ and $\text{num-steps}(a,a) = \text{num-steps}(a',a')$;
   - $b \stackrel{c}{\equiv} b'$ and they are both final.

   Since a and a' are final and $a \approx_p a$ and $a' \approx_p a'$, it must be that $\texttt{pc}$ is 0 in both a and a'. Thus, a and a' terminate with $\tau = \tau'$ that just include the check of the `while` condition. The other theses can be derived following the same proof structure as above. □

Finally, the main result of our paper directly follows from the theorem above, because the transformation in Section 3.4.2 satisfies Definition 2.8:

Corollary 3.2 (Constant-time preservation)**.**
The control-flow fattening obfuscation preserves the constant-time policy.

3.5 CONCLUSIONS

In this chapter we illustrated a first approach to secure compilation in the case of passive attackers. In doing that, we applied a methodology from the literature [21] to the advanced obfuscation technique of control-flow flattening and proved that it preserves the constant-time policy. First, we defined what programs leak using an instrumented semantics for our simple imperative language. Then, we have defined the relation $\approx_p$ between source and target configurations — that roughly relates configurations with the same behavior — and proved that it adheres to the definition of general simulation. Finally, we proved that the obfuscation preserves constant time by showing that the pair $(\stackrel{c}{\equiv}, \stackrel{c}{\equiv})$ is a general CT-simulation, as required by the framework we instantiated. As a consequence, the obfuscation based on control-flow flattening is proved to preserve the constant-time policy.

RELATED WORK Program obfuscations are widespread code transformations [33, 75, 76, 122, 132, 247] designed to protect software in settings where the adversary has physical access to the program and can compromise it by inspection or tampering. A great deal of work has been done on obfuscations that are resistant against reverse engineering making the life of attackers harder. However, we do not discuss these papers because they do not consider formal properties of the proposed transformations. We refer the interested reader to [116] for a recent survey.

To the best our knowledge, ours is the first work addressing the problem of security preservation, and therefore here we only focus on those proposals that formally studied the correctness of obfuscations. In [191, 192] a formal framework based on abstract interpretation is proposed to study the effectiveness of obfuscating techniques. This framework not only characterizes when a transformation is correct but also measures its resilience, i.e., the difficulty of undoing the obfuscation. More recently, other work went in the direction of fully verified, obfuscating compilation chains [37–39]. Among these [39] is the most similar to ours, but it only focuses on the correctness of the transformation, and studies it in the setting of the CompCert C compiler. Differently, we adopted a foundational approach by considering a core imperative language and proved that the considered transformation preserves security.

Besides [21], there has recently been an increasing interest in preserving the verification of the constant time policy, e.g., as said above a version of the CompCert C compiler [20] has been released that guarantees the preservation of the policy in each compilation step.

4

INCREMENTAL TYPING

As observed in Chapter 1, one of the goals of secure compilation is to make sure that no new security issues arise as the result of program transformations. In this chapter, we focus on an instance of secure compilation that prescribes the preservation of types, called *type-preserving compilation* [14, 22, 45, 69, 134, 156, 212] in the case in which source and target languages coincide. Also, although program transformations are usually considered to be carried out by compilers, here we consider general source code modifications, be they performed by compilers or by programmers.

The problem of developing type-preserving compilers has been widely studied in the past [69, 134, 156, 212] and type-preservation has been used as a means to achieve strong security guarantees in many cases [14, 22, 45]. However, to the best of our knowledge all the techniques from the literature either require a manual proof effort (which is not always easy or possible), or apply only to specific programming languages (see, e.g., Chen et al. [68]). Instead, our approach requires only a limited amount of effort to derive an incremental typing algorithm from an existing one (provided that the second is syntax-directed). Also, once the incremental typing algorithm has been derived it can be used in the spirit of *translation validation* [189] (see also Chapter 2) to check for the preservation of security properties.

Actually, a first simple way to achieve type preservation with minimal manual effort is to perform the type analysis of interest after each step of code transformation, so as to make sure that the resulting code has a type which is *compatible* (e.g., equal) with the original one. However, the ever-growing size of software code bases hinders this approach due to performance concerns: *incremental type analysis* mitigates this problem by just analyzing the code changes (*diffs*) between the given program before and after a transformation. This incrementality is particularly relevant when the diffs arise from the continuous evolution of software, e.g., when the codebase follows a *perpetual development model* [63].

In the rest of this chapter we introduce incremental typing analysis as a means to achieve type-preserving compilation efficiently and systematically. More specifically, we focus on transformations mapping programs from a language S to itself (as is usually the case with code optimizations used in compilers [87, 88, 198]) and on widespread type systems that permit an early code verification, so reducing errors and also prescribing programmers a clean programming style. Indeed, most of the modern programming languages are equipped with mechanisms for checking or inferring types, and to verify specific properties through them.

Rather than developing *new* typing algorithms that work incrementally, we propose to take an *existing* typing algorithm $\mathscr{A}$ as input, be it a checking or an inference one. Then, $\mathscr{A}$ is *used incrementally*, without re-doing work already done, but exploiting available results through caching and memoization. An advantage of our proposal is that it consists of

an algorithmic schema, namely a *wrapper* that is *independent* of any specific language and type system. In addition, we put forward mild conditions on the results and on the original type system that guarantee that the results of incremental typing match those of the original algorithm.

Roughly, our algorithmic schema works as follows. We start from the abstract syntax tree of a program, where each node is annotated with the result R provided by the original typing algorithm $\mathscr{A}$. We build then a cache, including for each sub-term t both the result R and some relevant contextual information needed by $\mathscr{A}$ to type t (typically a typing environment binding the free variables of t). When the program changes, its annotated abstract syntax tree changes accordingly, and typing the sub-term associated with the changed node is done by reusing the results in the cache whenever possible and by suitably invoking $\mathscr{A}$ upon need. Clearly, the more local the changes, the more reused the information.

Technically, our proposal consists of a set of rule schemata that drive the usage of the cache and of the original algorithm $\mathscr{A}$, as sketched above. Actually, the user has to define the shape of caches and to instantiate a well-confined part of the rule schemata. If the instantiation meets two easy-to-check criteria, the typing results of $\mathscr{A}$ and of the incremental algorithm are guaranteed to be coherent, i.e., the incrementalized algorithm behaves as the non-incremental one. Remarkably, the size of the incrementalized version is always the same of the original version, with a couple of additional rules that take care of cache hits. The overhead in time and space is small, in particular it decreases with the size of *diffs*.

All the above provides us with the guidelines to develop a framework that makes incremental the usage of a given typing algorithm. As a proof-of-concept, our approach has been implemented in OCaml and, along with some examples, is available online.[1] This implementation consists of a parametric module that inputs a type system in a specific format, and outputs its incrementalized version.

Summing up, this chapter includes:

- A parametric, language-independent algorithmic schema that builds a wrapper around an existing typing algorithm $\mathscr{A}$, so as to allow using it incrementally (Section 4.2);
- A formalization of the steps that instantiate the schema and yield the incrementalized version of $\mathscr{A}$: the resulting typing algorithm only types the *diffs* and those parts of the code affected by them (Section 4.2);
- A characterization of a rule format of most typing algorithms in terms of two auxiliary functions tr and $checkJoin$, only (Section 4.2) that, together with the syntax of the current language, suffice for automatically generating the code implementing its (non-incremental) type system;
- Three theorems that under mild conditions guarantee the correctness of the approach (Section 4.2);

1 Available at `https://github.com/matteobusi/incremental-mincaml`.

- The instantiation of the schema on seven typing algorithms for languages in three different programming paradigms: imperative, functional and process calculus. The original type systems are taken from the literature, and cover different aspects: the first two are simple type systems and allow us to illustrate the applicability of our approach, the next two allow to check variants of non-interference, the fifth deals with exception inference, and the last two enable checking of dependent types and protocol security (Section 4.3);
- A module that inputs a type system, rather the two auxiliary functions tr and $checkJoin$ and the syntax of the language in hand, and that outputs its incrementalized version (Section 4.4); and
- Experimental results showing that the time and the space used by the above incrementalized type checker depend on the size of *diffs*, and its performance increases as these become smaller (Section 4.4). This assessment is carried on a prototype of the incremental version of the type checker for MinCaml [219] (Section 4.4).

4.1 THE INCREMENTAL SCHEMA IN A NUTSHELL

Here we overview our algorithmic schema instantiating it on the core of a functional language, with the standard syntax, types, and typing system $\mathcal{T}$ with typing judgments $\Gamma \vdash_{\mathcal{T}} e : \tau$, where Γ is the typing context, e is an expression and $\tau \in \mathit{Type}$ is its type. The upper part of Figure 7 shows the typing rules for the $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ expression, variables and the conditional (ignore for the moment the colored boxes).

Suppose you have typed the following factorial function (in λ_{fact} the index is the name of the recursive function to call):

$$\mathbf{let}\ f = \lambda_{\mathit{fact}}\,(n : \mathit{int}).(\mathbf{if}\ n \geq 1\ \mathbf{then}\, n \times \mathit{fact}\,(n-1)\,\mathbf{else}\, n : \mathit{int})\ \mathbf{in}\, f\, 7$$

and to have stored its abstract-syntax tree annotated with types (aAST), shown in Figure 8. Assume now that f is optimized as follows:

$$\mathbf{let}\ f = \lambda_{\mathit{fact}}\,(n : \mathit{int}).(\mathbf{if}\ n \geq 3\ \mathbf{then}\, n \times \mathit{fact}\,(n-1)\,\mathbf{else}\, n : \mathit{int})\ \mathbf{in}\, f\, 7$$

Our goal is to re-type f using as much as possible the information already computed and stored in the aAST. Our first steps are building a cache from this information, once defined how it is represented.

In this simple case, we represent each element of the cache as a triple (e, Γ, τ). The entire cache for f is in Table 1, where the information needed for typing the relevant expression is given under the heading **Typing context**.

Actually, the cache is built through the specific function $\mathit{buildCache}_{\mathcal{T}}$, tailored on the language in hand. Below, we only show a couple of cases of its instantiation to our functional language:

$$\begin{aligned}
\mathit{buildCache}_{\mathcal{T}}\,((x : \tau_x), \Gamma) &\triangleq \{(x, [x \mapsto \tau_x], \tau_x)\} \\
\mathit{buildCache}_{\mathcal{T}}\,((\mathbf{let}\ x = e_1\ \mathbf{in}\, e_2 : \tau_{let}), \Gamma) &\triangleq \{(\mathbf{let}\ x = e_1\ \mathbf{in}\, e_2, \Gamma_{|\,FV(\mathbf{let}\ x = e_1\ \mathbf{in}\, e_2)}, \tau_{let})\} \\
&\quad \cup (\mathit{buildCache}_{\mathcal{T}}\,((x : \tau_x), \Gamma)) \\
&\quad \cup (\mathit{buildCache}_{\mathcal{T}}\,((e_1 : \tau_1), \Gamma)) \\
&\quad \cup (\mathit{buildCache}_{\mathcal{T}}\,((e_2 : \tau_2), \Gamma[x \mapsto \tau_x]))
\end{aligned}$$

Expression	Typing context	Type
$\textbf{let } f = \lambda_{fact}\,(n : int).(\textbf{if } n \geq 1 \textbf{ then } n \times fact\,(n-1) \textbf{ else } n : int) \textbf{ in } f\,7$	$\emptyset$	int
f	$[f \mapsto int \to int]$	$int \to int$
$fact$	$[fact \mapsto int \to int]$	$int \to int$
$\lambda_{fact}\,(n : int).(\textbf{if } n \geq 1 \textbf{ then } n \times fact\,(n-1) \textbf{ else } n : int)$	$\emptyset$	$int \to int$
$\textbf{if } n \geq 1 \textbf{ then } n \times fact\,(n-1) \textbf{ else } n$	$[fact \mapsto int \to int, n \mapsto int]$	int
$n \geq 1$	$[n \mapsto int]$	$bool$
n	$[n \mapsto int]$	int
1	$\emptyset$	int
$n \times fact\,(n-1)$	$[fact \mapsto int \to int, n \mapsto int]$	int
$fact\,(n-1)$	$[fact \mapsto int \to int, n \mapsto int]$	int
$n-1$	$[n \mapsto int]$	int
$f\,7$	$[f \mapsto int \to int]$	int
7	$\emptyset$	int

Table 1: Tabular representation of the cache C for our example.

We are now ready for our third step: making incremental the original, non-incremental type system. The rules in the new type system $\mathcal{IT}$ have a richer format than the original ones:

$$\Gamma, C \vdash_{\mathcal{IT}} e : \tau \triangleright C'$$

where C is the current cache and C' is the possibly updated one with the newly computed types. Some of the incremental rules are in the lower part of Figure 7, and we briefly comment on them below. First, we have a rule that checks if there is anything in the

$$(\mathcal{T}\text{-Var})\quad \frac{\boxed{x \in dom(\Gamma) \quad \Gamma(x) = \tau}}{\Gamma \vdash_{\mathcal{T}} x : \tau} \qquad (\mathcal{T}\text{-Let})\quad \frac{\boxed{\Gamma} \vdash_{\mathcal{T}} e_1 : \tau_1 \quad \boxed{\Gamma[x \mapsto \tau_1]} \vdash_{\mathcal{T}} e_2 : \tau_2 \quad \boxed{\tau = \tau_2}}{\Gamma \vdash_{\mathcal{T}} \textbf{let } x = e_1 \textbf{ in } e_2 : \tau}$$

$$(\mathcal{T}\text{-If})\quad \frac{\boxed{\Gamma} \vdash_{\mathcal{T}} e_1 : bool \quad \boxed{\Gamma} \vdash_{\mathcal{T}} e_2 : \tau_2 \quad \boxed{\Gamma} \vdash_{\mathcal{T}} e_3 : \tau_3 \quad \boxed{\tau = \tau_2 = \tau_3}}{\Gamma \vdash_{\mathcal{T}} \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}$$

$$(\mathcal{IT}\text{-Hit})\quad \frac{C(e) = \langle \Gamma', \tau \rangle \quad compat_{\mathcal{T}}(\Gamma, \Gamma', e)}{\Gamma, C \vdash_{\mathcal{IT}} e : \tau \triangleright C} \qquad (\mathcal{IT}\text{-VarMiss})\quad \frac{\Gamma \vdash_{\mathcal{T}} x : \tau \quad C' = C \cup \{(x, [x \mapsto \tau], \tau)\}}{\Gamma, C \vdash_{\mathcal{IT}} x : \tau \triangleright C'}\; miss(C, x, \Gamma)$$

$$(\mathcal{IT}\text{-LetMiss})\quad miss(C, \textbf{let } x = e_1 \textbf{ in } e_2, \Gamma)\quad \frac{\boxed{\Gamma}, C \vdash_{\mathcal{IT}} e_1 : \tau_1 \triangleright C_1 \quad \boxed{\Gamma[x \mapsto \tau_1]}, C \vdash_{\mathcal{IT}} e_2 : \tau_2 \triangleright C_2 \quad \boxed{\tau = \tau_2} \quad C' = C \cup C_1 \cup C_2 \cup \{(\textbf{let } x = e_1 \textbf{ in } e_2, \Gamma_{|FV(\textbf{let } x = e_1 \textbf{ in } e_2)}, \tau)\}}{\Gamma, C \vdash_{\mathcal{IT}} \textbf{let } x = e_1 \textbf{ in } e_2 : \tau \triangleright C'}$$

$$(\mathcal{IT}\text{-IfMiss})\quad miss(C, \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, \Gamma)\quad \frac{\boxed{\Gamma}, C \vdash_{\mathcal{IT}} e_1 : bool \triangleright C_1 \quad \boxed{\Gamma}, C \vdash_{\mathcal{IT}} e_2 : \tau_2 \triangleright C_2 \quad \boxed{\Gamma}, C \vdash_{\mathcal{IT}} e_3 : \tau_3 \triangleright C_3 \quad \boxed{\tau = \tau_2 = \tau_3} \quad C' = C \cup C_1 \cup C_2 \cup C_3 \cup \{(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, \Gamma_{|FV(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)}, \tau)\}}{\Gamma, C \vdash_{\mathcal{IT}} \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau_2 \triangleright C'}$$

Figure 7: Three rules of the original type system $\mathcal{T}$ (upper part) and their corresponding incremental versions in the type system $\mathcal{IT}$. The rest of the rules is in Appendix A.

cache for an expression e (rule ($\mathcal{IT}$-Hit)). If we find such a triple, we simply re-use the

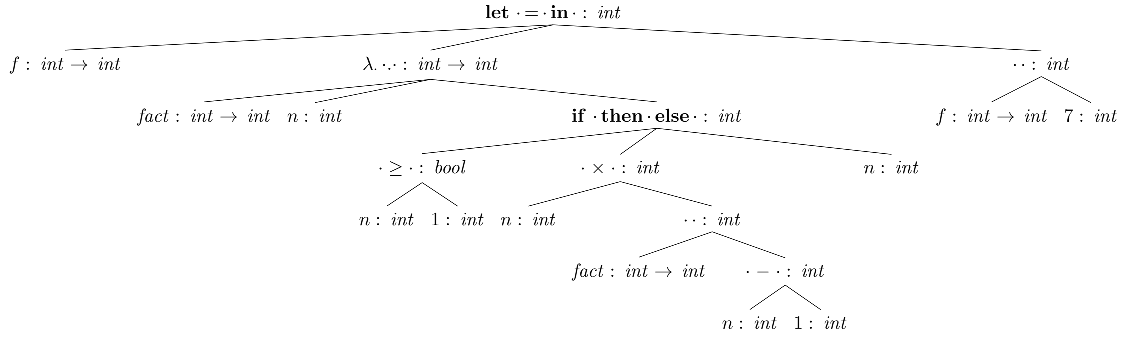


Figure 8: Abstract-syntax tree annotated with types for the factorial program.

cached type, provided that e was typed in the cached typing context Γ' compatible with the current typing context Γ. The notion of compatibility, expressed by the predicate $compat_{\mathcal{T}}(\Gamma, \Gamma', e)$, heavily depends on the type system in hand, and here it simply requires that the two typing contexts coincide on the free variables of e. Of course, the cache C needs no updates.

Next, there are the rules for when the expression in hand has to be re-typed, because the cache has no entry for it or the information available is not compatible. In these rules, the boxed parts (fuchsia and light blue in the pdf) play a role in their automatic generation from the original ones. In the rule ($\mathcal{IT}$-VarMiss), a call to the original type system $\mathcal{T}$ suffices for determining the type of x and for updating the cache. Note that Γ occurs identically in the premises of the original and the incrementalized rules. In the other two rules, the expression in hand has sub-terms that are typed using the new rules. Also here the boxed premises, including the updated version of the typing context for the **let** construct, are the same in $\mathcal{T}$ and $\mathcal{IT}$. There is however a difference, to be made precise later, between the fuchsia boxes and the light blue ones. The first kind of premises identify the (portions of the) typing context needed to type the sub-terms. The second kind of boxed premises specify the constraints that the types computed for these sub-terms have to satisfy and how they can be joined together to get the type of the full term. In this example, the equality between τ_2 and τ_3. It is worth noting that $buildCache_{\mathcal{T}}$ could be simulated by executing the incremental algorithm with an initial empty cache. However, we kept the two steps apart in the framework, so separating the concerns of building the initial cache and performing the actual typing.

Finally, we are left to prove that the types computed by the incrementalized transition system are coherent with those derived by the original one. To do that, one is required to verify two mild conditions on the predicate $compat_{\mathcal{T}}$ and on the applicability of the premises in the light blue boxes. The soundness of the incrementalized type system then follows by instantiating a general theorem, whose hypotheses are exactly the two conditions above.

This is the case in our example, and thus we can re-use most of the information in the cache C for typing the new version of f, yielding the tree in Figure 9, where we omit the light blue premises that are trivial (cache hits are highlighted in green in the pdf).

4.2 FORMALIZING THE INCREMENTAL SCHEMA

In this section we formally present our algorithmic schema that, given a typing algorithm $\mathcal{A}$, yields its incrementalized version $\mathcal{IA}$. We also prove that $\mathcal{IA}$ is sound, in that it computes the same types of $\mathcal{A}$, possibly up to some type manipulations used by $\mathcal{A}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{C(n) = \langle \Gamma_n, int \rangle}{\Gamma, C \vdash n : int \triangleright C} \quad \dfrac{\Gamma \vdash 3 : int}{\Gamma, C \vdash 3 : int \triangleright C'}}{\Gamma, C \vdash n \geq 3 : bool \triangleright C''} \quad \dfrac{C(F_{rec}) = \langle \Gamma, int \rangle}{\Gamma, C \vdash F_{rec} \triangleright C} \quad \dfrac{C(n) = \langle \Gamma_n, int \rangle}{\Gamma_n, C \vdash n \triangleright C}}{\Gamma, C \vdash F_{cond} : int \triangleright C'''}}{\emptyset, C \vdash F_{body} : int \to int \triangleright C^{iv}} \quad \dfrac{C(f\,7) = \langle \Gamma_f, int \rangle}{\Gamma_f, C \triangleright f\,7 : int \triangleright C}}{\emptyset, C \vdash F : int \triangleright C^{iv} \cup \{(F, \emptyset, int)\}}$$

where

$$F = \textbf{let } f = F_{body} \textbf{ in } f\,7 \qquad F_{body} = \lambda_{fact}\,(n : int).(F_{cond} : int)$$
$$F_{cond} = \textbf{if } n \geq 3 \textbf{ then } F_{rec} \textbf{ else } n \qquad F_{rec} = n \times fact\,(n-1)$$

$$\Gamma = [n \mapsto int, fact \mapsto int \to int]$$
$$\Gamma_f = [f \mapsto int \to int] \qquad \Gamma_n = [n \mapsto int]$$

$$C' = C \cup \{(3, \emptyset, int)\} \qquad C'' = C' \cup \{(n \geq 3, \Gamma_n, bool)\}$$
$$C''' = C'' \cup \{(F_{cond}, \Gamma, int)\} \qquad C^{iv} = C''' \cup \{(F_{body}, \emptyset, int)\}.$$

Figure 9: Incremental type checking of the modified factorial program, where C is as in Table 1.

We remark that our proposal is independent of the paradigm of the programming language considered and of the kind of its typing algorithm, be it designed for inferring or for checking types, as exemplified in Section 4.3. We only assume in the formal presentation of this section that the original typing algorithm $\mathscr{A}$ is defined in the usual syntax-directed manner, through inference rules given in a standardized format described below.

The following notation is rather common.

Definition 4.1. *Given a language with typing algorithm $\mathscr{A}$, let*

- $x, y, \ldots \in Name$ *be the set of* names, *e.g., variables or identifiers;*
- $t \in Term$ *denote its* basic elements, *e.g., expressions or statements;*
- $\tau, \tau', \ldots \in Type$ *be the set of (standard)* types, *usually included in Res the set of typing results $R, R', \ldots$ (referred to as* results *hereafter); whenever $Type$ coincides with Res we feel free to refer to results as* types.
- $\Gamma \in TypeCtx$ *be the contextual information needed for typing, which always includes $TypeEnv \ni E : Name \to Type$ (the set of typing environments), and often coincides with it. When the two coincide we feel free to refer to typing contexts as (typing) environments and* vice versa.

Finally, $\Gamma \vdash_{\mathscr{A}} t \colon R$ is the call to $\mathscr{A}$ for typing t.

Now we define the format of the inference rules of $\mathscr{A}$. Remarkably, $\mathscr{A}$ is fully specified *only* by the functions tr and $checkJoin$ used below.

Definition 4.2. *The rules of $\mathscr{A}$ have the following format*

$$\dfrac{\forall i \in \mathbb{I}_t \,.\, \boxed{tr_t^{t_i}(\Gamma, \{R_j\}_{j<i})} \vdash_{\mathscr{A}} t_i : R_i \qquad \boxed{checkJoin_t(\Gamma, \{R_i\}_{i \in \mathbb{I}_t}, \texttt{out}\ R)}}{\Gamma \vdash_{\mathscr{A}} t : R} \quad \textit{where}$$

- *t_i ($i \in \mathbb{I}_t$) is a proper sub-term of t whose result R_i is needed to compute R; in addition, we let $i < j$ if typing t_j requires R_i;*
- *The function $tr_t^{t_i}$ computes the typing context needed by t_i from Γ and the set of results $\{R_j \mid j < i \wedge j \in \mathbb{I}_t\}$;*
- *The (conjunction of) predicate(s) $checkJoin_t$ first check(s) that the results R_i of the sub-terms t_i are compatible with each other, combines them all, and outputs the overall result R.*

A special case occurs when the current term t has no proper sub-terms, i.e., when $\mathbb{I}_t = \emptyset$. In this case, the function tr is empty as well and only the part concerning $checkJoin$ is left.

To enhance readability, above and in the rest of the chapter we will continue highlighting the occurrences of tr (fuchsia in the pdf) and $checkJoin$ (light blue in the pdf), as done in Section 4.1.

The following examples show how some common typing rules in the usual format are rendered in the format above. We remark that, when the type system is supplied in the common syntax-directed, inductive form, both the function tr and the predicate $checkJoin$ (thus the ordering $<$ on sub-terms) could be easily and mechanically extracted from the usual typing rules.

Consider again the example of Section 4.1 and the typing rule ($\mathcal{T}$-VAR) for variables, where $\mathbb{I}_x = \emptyset$:

$$\frac{x \in \mathtt{dom}\,(\Gamma) \qquad \tau = \Gamma(x)}{\Gamma \vdash_{\mathcal{A}} x : \tau}$$

It is rendered in our format as follows

$$\frac{checkJoin_x(\Gamma, \emptyset, \mathtt{out}\ \tau)}{\Gamma \vdash_{\mathcal{A}} x : \tau}$$

where $checkJoin_x(\Gamma, \emptyset, \mathtt{out}\ \tau) \triangleq x \in \mathtt{dom}\,(\Gamma) \ \wedge\ \tau = \Gamma(x)$.

If the rule takes instead the form of the following axiom

$$\frac{}{\Gamma'[x \mapsto \tau] \vdash_{\mathcal{A}} x : \tau}$$

one has $\mathbb{I}_x = \emptyset$ and the same $checkJoin_x$, where $\Gamma'[x \mapsto \tau]$ replaces Γ.

Consider also the rule (($\mathcal{T}$-LET)) for the expression $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$

$$\frac{\Gamma \vdash_{\mathcal{A}} e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash_{\mathcal{A}} e_2 : \tau_2 \qquad \tau = \tau_2}{\Gamma \vdash_{\mathcal{A}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

In our format it becomes as follows (we abuse the set notation, e.g., omitting $\emptyset$ or $\{$ and $\}$ when clear from the context).

$$\frac{\begin{array}{c} tr^{e_1}_{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2}(\Gamma, \emptyset) \vdash_{\mathcal{A}} e_1 : \tau_1 \\ tr^{e_2}_{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2}(\Gamma, \tau_1) \vdash_{\mathcal{A}} e_2 : \tau_2 \qquad checkJoin_{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2}(\Gamma, \tau_1, \tau_2, \mathtt{out}\ \tau) \end{array}}{\Gamma \vdash_{\mathcal{A}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

This example shows that the definition of function tr is immediate; that we need the result of e_1 for typing e_2 (so $\tau_1 < \tau_2$); and that the second parameter of $tr^{e_1}_{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2}$ is empty, because we only need the typing context to type e_1.

$$tr^{e_1}_{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2}(\Gamma, \emptyset) \triangleq \Gamma \qquad tr^{e_2}_{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2}(\Gamma, \tau) \triangleq \Gamma[x \mapsto \tau]. \tag{2}$$

Also, the following definition is immediate

$$checkJoin_{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2}(\Gamma, \{\tau_1, \tau_2\}, \mathtt{out}\ \tau) \triangleq (\tau = \tau_2).$$

4.2.1 *Incorporating incrementality*

Our algorithmic schema operates in four steps. The first defines the shape of the cache that will be used to retrieve the reusable pieces of type information. The second step specifies how a cache is built for a given term. The third step is the most important and generates the rules for incrementally using the original type inference/checking algorithm. Finally, the incremental algorithm is proved to behave as the original one, provided that the contents of the cache and the original typing algorithm each satisfy a mild condition.

From now onward, we assume the language and its original algorithm $\mathscr{A}$ as fixed. Also, we assume $\mathscr{A}$ to be *syntax-directed*, i.e., $\mathscr{A}$ is assumed to be defined inductively on the syntax of the terms of the language.

DEFINING THE SHAPE OF CACHES The shape of the cache is crucial for re-using incrementally portions of the available typing results. A cache associates the input data t to the typing environment Γ and result R, rendered by a set of triples (t, Γ, R), as done in Section 4.1. More formally:

Definition 4.3. *Let the set of caches be* $\wp(\mathit{Term} \times \mathit{TypeCtx} \times \mathit{Res}) \ni C$*, let* $C(t) = \langle \Gamma, R \rangle$ *if the cache has an entry for* t*.*

We remark that caches allow for multiple (compatible) triples for the same expression. The choice of which entry to return when using the functional notation is left to the particular cache implementation.

BUILDING CACHES Given a term t, we assume that the nodes of its abstract syntax tree (called *annotated Abstract Syntax Tree* or *aAST*) are annotated with the result of $\mathscr{A}$ for each of the sub-terms they represent, written $t : R$. Also, let $\Gamma_{|\,FV(t)}$ be the restriction of Γ to the free names of t. Note that $\Gamma_{|\,FV(t)}$ always carries enough information to type t, because of the assumption that the type system is syntax-directed. The construction of the cache relies on the aAST of t, and an example is in Section 4.1. Note that in the definition below, when a term t has no sub-term, the set $\mathbb{I}_t$ is empty.

Definition 4.4. *Let* $\{t_i\}_{i \in \mathbb{I}_t}$ *and* $\boxed{tr_t^{t_i}}$ *be as in Definition 4.2. Then the cache is built by the following function:*

$$\mathit{buildCache}_{\mathscr{A}}\,((t : R), \Gamma) = \{(t, \Gamma_{|\,FV(t)}, R)\} \cup \bigcup_{i \in \mathbb{I}_t} \mathit{buildCache}_{\mathscr{A}}\left((t_i : R_i), \boxed{tr_t^{t_i}(\Gamma, \{R_j\}_{j<i})}\right)$$

The following definition describes the set of caches that represent correct typing information:

Definition 4.5 (Cache well-formedness)**.** *A cache* C *is said to be* well-formed *for* $\mathscr{A}$ *(written* $\Vdash_{\mathscr{A}} C$*)* iff $\forall t, \Gamma, R.\ (t, \Gamma, R) \in C \Rightarrow \Gamma \vdash_{\mathscr{A}} t : R$*.*

The following theorem ensures that each entry of a cache returned by $\mathit{buildCache}_{\mathscr{A}}$ is well-formed for $\mathscr{A}$:

Theorem 4.1 (Cache correctness)**.** *Let* $t \in \mathit{Term}$*,* $R \in \mathit{Res}$ *and* $\Gamma \in \mathit{TypeCtx}$*. If* $\Gamma \vdash_{\mathscr{A}} t : R$*, then* $\Vdash_{\mathscr{A}} \mathit{buildCache}_{\mathscr{A}}\,((t : R), \Gamma)$*.*

Proof. Immediate by induction on the structure of the term t. □

INCREMENTAL TYPING The third step consists of instantiating the rule templates that make typing incremental. We stress that *no change* to the original algorithm $\mathscr{A}$ is needed: its implementation is taken off the shelf and is used as a *gray-box*. Indeed, what matters are just the information registered in the original judgments after a minor syntactic re-wording; the rules that are inspected to derive the function $tr_t^{t_i}$ and the predicate $checkJoin$ of Definition 4.2; and a condition on the algorithm itself and one on the cache contents (see Definitions 4.8 and 4.9 below). In other words, what we propose is a *wrapper* around the original typing algorithm.

The judgments for the incrementalized typing algorithm $\mathscr{IA}$ have the form

$$\Gamma, C \vdash_{\mathscr{IA}} t : R \triangleright C'.$$

The rules of $\mathscr{IA}$ require the predicate $compat_{\mathscr{A}}(\Gamma, \Gamma', t)$ to express the condition that enable us to re-use the cache contents $C(t) = \langle \Gamma', R \rangle$. Intuitively, the predicate holds if Γ' includes, in a broad sense, the information represented by Γ for t, so conveying the compatibility of Γ and Γ'. For example, if $\Gamma(x) = int$ and $\Gamma'(x) = real$, then the two typing contexts are considered compatible with respect to the standard sub-typing relation. According to our experience, the definition of the predicate $compat_{\mathscr{A}}$ can always be easily derived by exploiting suitable relations on results, e.g., equality or sub-typing.

The notion of *cache miss* also helps in the definition of the incrementalized typing algorithm. Intuitively, a cache miss happens whenever there is no association for t in a given cache, or if an entry (t, Γ', R) exists but the typing context Γ' is not compatible with the current Γ. Formally, we express a cache miss (and, dually a cache hit) as follows:

Definition 4.6. *For all C, t, Γ, a* cache miss *occurs whenever the following predicate holds*

$$miss(C, t, \Gamma) \triangleq \nexists \Gamma', R.\ \big(C(t) = \langle \Gamma', R \rangle \wedge compat_{\mathscr{A}}(\Gamma, \Gamma', t)\big).$$

Otherwise, there is a cache hit.

We are now ready to introduce the three different rule templates that make the original typing algorithm $\mathscr{A}$ incremental.

Definition 4.7. *The rules of the typing algorithm $\mathscr{IA}$ are obtained from those of $\mathscr{A}$ by instantiating the following rule templates:*

(TEMPLATE-HIT)
$$\frac{C(t) = \langle \Gamma', R \rangle \qquad compat_{\mathscr{A}}(\Gamma, \Gamma', t)}{\Gamma, C \vdash_{\mathscr{IA}} t : R \triangleright C}$$

(TEMPLATE-MISS-NOSUB)
$$\frac{\Gamma \vdash_{\mathscr{A}} t : R \qquad C' = C \cup \{(t, \Gamma_{\mid FV(t)}, R)\}}{\Gamma, C \vdash_{\mathscr{IA}} t : R \triangleright C'}\ miss(C, t, \Gamma) \wedge \mathbb{I}_t = \emptyset$$

(TEMPLATE-MISS-SUB)
$$\frac{\begin{array}{c}\forall i \in \mathbb{I}_t \,.\, tr_t^{t_i}(\Gamma, \{R_j\}_{j<i}), C \vdash_{\mathscr{IA}} t_i : R_i \triangleright C_i \\ checkJoin_t(\Gamma, \{R_i\}_{i \in \mathbb{I}_t}, \texttt{out}\ R) \qquad C' = \{(t, \Gamma_{\mid FV(t)}, R)\} \cup \bigcup_{i \in \mathbb{I}_t} C_i\end{array}}{\Gamma, C \vdash_{\mathscr{IA}} t : R \triangleright C'}\ miss(C, t, \Gamma) \wedge \mathbb{I}_t \neq \emptyset$$

The rule template (TEMPLATE-HIT) applies when there is a cache hit and the cached result $C(t) = \langle \Gamma', R \rangle$ can be used as it is because the typing context Γ' is compatible with Γ; of course, the cache C needs no updates (e.g., see the rule ($\mathscr{IT}$-HIT) in Figure 7).

The template (TEMPLATE-MISS-NOSUB) is for when there is a cache miss and the current term t has no proper sub-terms, and thus $\mathbb{I}_t = \emptyset$. The new cache C' is obtained by installing in C the new result R for t computed with the original type algorithm $\mathscr{A}$, e.g., see the rule ($\mathscr{IT}$-VARMISS) in Figure 7, where the triple $(x, \emptyset, \tau)$ is added to the current cache.

Finally, the last template (Template-Miss-Sub) applies when there is a cache miss, but the typing result of the term t is inductively defined from those of its sub-terms. In this case, the incremental algorithm $\mathcal{IA}$ is inductively invoked on the sub-terms and composes the obtained results, either available in the cache or inductively computing them (e.g., see the last two rules in Figure 7).[2]

PROVING TYPE COHERENCE We now state the two sufficient conditions for proving that the construction above preserves the correctness of the original algorithm $\mathcal{A}$ and yields its incremental version $\mathcal{IA}$. Both conditions use a binary reflexive relation $\sim$ between results, that is typically instantiated to equality or to sub-typing relation; anyway, $\sim$ is fully determined by the corresponding relation used by $\mathcal{A}$.

Not surprisingly, the first condition involves $compat_{\mathcal{A}}(\Gamma, \Gamma', t)$ and makes sure that it guarantees that Γ and Γ' share all the needed information to correctly type the term t:

Definition 4.8 (Typing context compatibility)**.** *A predicate $compat_{\mathcal{A}}$* expresses compatibility *w.r.t.* $\sim$ iff

$$\forall \Gamma, \Gamma', t.\ compat_{\mathcal{A}}(\Gamma, \Gamma', t) \wedge \Gamma \vdash_{\mathcal{A}} t : R \Rightarrow \\ (\forall R' \text{ such that } \Gamma' \vdash_{\mathcal{A}} t : R' \text{ it holds } R \sim R')$$

The second condition is on the original typing algorithm $\mathcal{A}$, which intuitively (and as expected) must respect the relation $\sim$:

Definition 4.9 (Preservation of $\sim$)**.** *A typing algorithm $\mathcal{A}$* preserves $\sim$ iff *for any term t, $i \in \mathbb{I}_t$, R_i, R'_i such that $R_i \sim R'_i$ it holds that*

$$\forall R, R'.\ checkJoin_t(\Gamma, \{R_i\}_{i \in \mathbb{I}_t}, \texttt{out } R) \wedge checkJoin_t(\Gamma, \{R'_i\}_{i \in \mathbb{I}_t}, \texttt{out } R') \\ \Rightarrow R \sim R'.$$

Note that the above condition is trivially satisfied when $\sim$ is type equality and $checkJoin_t$ is deterministic, which is typically the case. In all our case studies, the function $checkJoin$ also preserves sub-typing, when present. Additionally, this condition and the typing context compatibility, as well, are quite simple to check.

Now we establish the correctness of the incremental algorithm $\mathcal{IA}$, provided that the original one is such. For that, we prove three properties: *coherence*, *completeness* and *well-formedness preservation*. Roughly, *coherence* guarantees that the results of the original typing algorithm and the incrementalized one (if any) agree up to $\sim$:

Theorem 4.2 (Coherence)**.** *Let C be well-formed for $\mathcal{A}$, $compat_{\mathcal{A}}$ be a predicate that expresses compatibility w.r.t. $\sim$, and $\mathcal{A}$ be a typing algorithm that preserves $\sim$. For all t, Γ, R, R', C' it holds that*

$$\Gamma \vdash_{\mathcal{A}} t : R \ \wedge \ \Gamma, C \vdash_{\mathcal{IA}} t : R' \triangleright C' \Rightarrow R \sim R'.$$

Proof. If $C(t) = \langle \Gamma', R' \rangle$ and Γ' is compatible with Γ, we instantiate the rule template (Template-Hit) of Definition 4.7. By well-formedness of C we have that $\Gamma' \vdash_{\mathcal{A}} t : R'$. Thus, by the fact that $compat_{\mathcal{A}}$ expresses compatibility w.r.t. $\sim$, it follows $R \sim R'$ as requested.

Otherwise, a cache miss occurs and we proceed by structural induction on t:

2 Actually, our implementation uses an equivalent format that accumulates the caches obtained by typing t_i with $i < j$ when typing t_j.

BASE CASE: The base case occurs when t has no sub-terms, and in this case the rule template (TEMPLATE-MISS-NOSUB) of Definition 4.7 is used. Since we know by hypothesis that $\Gamma, C \vdash_{\mathcal{I}\mathcal{A}} t : R' \triangleright C'$, by the premises the rule template (TEMPLATE-MISS-NOSUB) it follows that $\Gamma \vdash_{\mathcal{A}} t : R'$. Thus by Definition 4.2 the result R' of the incremental typing algorithm is obtained from $checkJoin_t(\Gamma, \emptyset, \texttt{out}\ R')$. Similarly, the result R of the original algorithm is produced by $checkJoin_t(\Gamma, \emptyset, \texttt{out}\ R)$. The thesis follows since $\mathcal{A}$ preserves $\sim$.

INDUCTIVE STEP: The induction hypothesis allows us to assume that for any sub-term of t the implication holds, i.e., (assuming each i-indexed variable to be universally quantified)

$$\forall i \in \mathbb{I}_t.\ \Gamma_i \vdash_{\mathcal{A}} t_i : R_i \ \wedge\ \Gamma_i, C \vdash_{\mathcal{I}\mathcal{A}} t_i : R'_i \triangleright C'_i \Rightarrow R_i \sim R'_i.$$

For all i, let $\Gamma_i = tr_t^{t_i}(\Gamma, \{R_j\}_{j<i})$. Also, from the hypotheses and the rule template of Definition 4.2 we know that $\Gamma_i \vdash_{\mathcal{A}} t_i : R_i$ (for some R_i) and the result R of the original typing algorithm is given by $checkJoin_t(\Gamma, \{R_i\}_{i\in\mathbb{I}_t}, \texttt{out}\ R)$. Similarly, by hypotheses and the rule template (TEMPLATE-MISS-SUB) of Definition 4.7, we have $\Gamma_i, C \vdash_{\mathcal{I}\mathcal{A}} t_i : R'_i$ (for some R'_i) and the result R' of the incremental algorithm is given by $checkJoin_t(\Gamma, \{R'_i\}_{i\in\mathbb{I}_t}, \texttt{out}\ R')$. Finally, by the induction hypothesis we know that the above $R_i \sim R'_i$ (for all i), thus by the fact that $\mathcal{A}$ preserves $\sim$, it follows $R \sim R'$ as requested. □

Dual to coherence is *completeness*, that ensures that if the original algorithm produces a result R, then also the incremental one produces a result (which can be then related to R by the coherence theorem above, assuming well-formedness of the cache):

Theorem 4.3 (Completeness)**.** *Let C be a cache. For all t, Γ, R it holds that*

$$\Gamma \vdash_{\mathcal{A}} t : R \Rightarrow \exists R', C'.\ \Gamma, C \vdash_{\mathcal{I}\mathcal{A}} t : R' \triangleright C'.$$

Proof. If a cache hit happens, a compatible entry $\langle \Gamma', R'' \rangle$ for t belongs to the cache. In the case we instantiate the rule template (TEMPLATE-HIT) of Definition 4.7, and it easily follows that $\Gamma, C \vdash_{\mathcal{I}\mathcal{A}} t : R'' \triangleright C$ (i.e., $R' = R''$ and $C' = C$).

Otherwise, a cache miss occurs and we proceed by structural induction on t:

BASE CASE: The base case occurs when t has no sub-terms, and the rule template (TEMPLATE-MISS-NOSUB) of Definition 4.7 applies. Since by hypothesis $C \vdash_{\mathcal{I}\mathcal{A}} t : R$, the premises the rule template (TEMPLATE-MISS-NOSUB) entails $\Gamma, C \vdash_{\mathcal{I}\mathcal{A}} t : R' \triangleright C'$ for $R' = R$ and $C' = C \cup \{(t, \Gamma_{|\,FV(t)}, R)\}$.

INDUCTIVE STEP: The induction hypothesis allows us to assume that for any sub-term of t the implication holds, i.e., (assuming each i-indexed variable to be universally quantified)

$$\forall i \in \mathbb{I}_t.\ \Gamma_i \vdash_{\mathcal{A}} t_i : R_i \Rightarrow \exists R'_i, C'_i.\ \Gamma, C \vdash_{\mathcal{I}\mathcal{A}} t_i : R'_i \triangleright C'_i.$$

For any i, let $\Gamma_i = tr_t^{t_i}(\Gamma, \{R_j\}_{j<i})$. From the hypotheses and by the rule template of Definition 4.2, it follows that $\Gamma_i \vdash_{\mathcal{A}} t_i : R_i$ (for some R_i). The thesis then follows by the premises of (TEMPLATE-MISS-SUB) and by the induction hypothesis: R' is given by $checkJoin_t(\Gamma, \{R'_i\}_{i\in\mathbb{I}_t}, \texttt{out}\ R')$ and $C' = \{(t, \Gamma_{|\,FV(t)}, R')\} \cup \bigcup_{i\in\mathbb{I}_t} C'_i$. □

Finally, *well-formedness preservation* guarantees that the incrementalized algorithm keeps the well-formedness of caches when updating them, so enabling the usage of the updated caches in future re-typings:

Theorem 4.4 (Well-formedness preservation)**.** *Let C be a well-formed cache. For all t, Γ, R, C' it holds that*

$$\Gamma, C \vdash_{\mathcal{IA}} t : R \triangleright C' \Rightarrow \Vdash_{\mathcal{A}} C'.$$

Proof. If a cache hit happens, we instantiate the rule template (Template-Hit) of Definition 4.7, and $C' = C$ is well-formed by hypothesis.

Otherwise, a cache miss occurs and we proceed by structural induction on t:

Base case: The base case occurs when t has no sub-terms, and the rule template (Template-Miss-NoSub) of Definition 4.7 applies. By the premises of such a rule template, it follows that $C' = C \cup \{(t, \Gamma_{|\,FV(t)}, R)\}$ and trivially $\Vdash_{\mathcal{A}} C'$.

Inductive step: The induction hypothesis allows us to assume that for any sub-term of t the implication holds, i.e., (assuming each i-indexed variable to be universally quantified)

$$\forall i \in \mathbb{I}_t.\ \Gamma_i, C \vdash_{\mathcal{IA}} t_i : R_i \triangleright C'_i \Rightarrow \Vdash_{\mathcal{A}} C'_i.$$

The well-formedness of $C' = \{(t, \Gamma_{|\,FV(t)}, R)\} \cup \bigcup_{i \in \mathbb{I}_t} C'_i$ easily follows by the premises of the rule template (Template-Miss-Sub) and by the induction hypothesis. □

4.3 MAKING EXISTING TYPING ALGORITHMS INCREMENTAL

In this section we illustrate the flexibility of our proposal and how our algorithmic schema can be easily instantiated to make non-trivial type systems incremental. For that, we consider seven type systems that enforce different properties on programs and we study their incrementalization using our schema:

1. In Section 4.3.1 and Section 4.3.2 we present two use cases that serve as a warm-up and show how to incrementalize a type checking and a type inference algorithm for a simple functional language;

2. As said in the previous chapter, preserving (variants of) non-interference is an important issue, thus Section 4.3.3 applies our framework to the classical security-typed imperative language by [229];

3. Section 4.3.4 extends the previous use case and considers active attackers in the style of Section 2.2.2 by applying our schema to the type system for enforcing robust declassification of Myers et al. [159];

4. Since failing to make sure that the raised exceptions are the same before and after a program transformation may lead to unexpected vulnerabilities, in Section 4.3.5 we illustrate how to incrementally use the type inference algorithm for exceptions by Leroy and Pessaux [140].

5. Dependent types allow to express many interesting properties of programs using types, thus our sixth use case in Section 4.3.6 deals with incremental type checking of the dependently-typed λ-calculus [186];

6. Finally, Section 4.3.7 incrementalizes the type system to check security of protocols in the SPI calculus [3].

Here, we omit all the proofs of theorems and lemmata, which are instead available in Appendix A.

4.3.1 *Type checking a functional language*

Here, we instantiate our schema in order to incrementally type check FUN, a simply typed functional programming language with booleans, integers and recursive lambdas. The syntax, the types and the semantics of FUN are standard, see e.g., [166]. We only recall some relevant aspects of its syntax below.

$$\begin{array}{rll} \mathit{Val} \ni v & ::= c \mid \lambda_f\,(x : \tau_x).(e : \tau_e) & \quad \mathtt{op} \in \{+, *, =, \leq\} \\ \mathit{Expr} \ni e & ::= v \mid x \mid e_1\ \mathtt{op}\ e_2 \mid e_1\, e_2 \mid \mathbf{if}\ e_1\,\mathbf{then}\,e_2\,\mathbf{else}\,e_3 \mid & \\ & \quad \mathbf{let}\ x = e_2\ \mathbf{in}\ e_3 & \\ \mathit{Type} \ni \tau, \tau_x, \tau_e & ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \to \tau_2 & \quad \mathit{TypeEnv} \ni \Gamma ::= \emptyset \mid \Gamma[x \mapsto \tau] \end{array}$$

where the f in the functional abstraction denotes the name of the (possibly) recursive function we are defining, with type $\tau_x \to \tau_e$. Also, we denote with $\tau_{\mathtt{op}}$ the type of the result of the operation op, with $\tau^l_{\mathtt{op}}$ that of its left operand, and with $\tau^r_{\mathtt{op}}$ that of the right one.

Assume as given the type checking algorithm $\mathfrak{F}$, defined by judgments

$$\Gamma \vdash_{\mathfrak{F}} e : \tau$$

We build the type checking algorithm $\mathcal{I}\mathfrak{F}$ that uses $\mathfrak{F}$ incrementally by following the four steps detailed in Section 4.2.

DEFINING THE SHAPE OF CACHES As said in Section 4.2 each entry in the cache is a triple made of a term, a typing environment and a result. Instantiating this to FUN, we get that

$$C \in \mathit{Cache} = \wp(\mathit{Expr} \times \mathit{TypeEnv} \times \mathit{Type}).$$

BUILDING CACHES We build the cache by visiting the aAST and "reconstructing" the typing environment. The function $\mathit{buildCache}_{\mathfrak{F}}$ is in Figure 10, where for brevity we have directly used the results of tr rather than writing the needed invocations. Indeed, tr is the identity almost everywhere, except for **let-in** (see Equation (2)) and for abstraction where it is $\boxed{tr^e_{\lambda_f\,(x:\tau_x).(e:\tau_e)}(\Gamma, \{\tau_x, \tau_f\}) = \Gamma[x \mapsto \tau_x, f \mapsto \tau_f]}$.

INCREMENTAL TYPING By instantiating the patterns of Section 4.2 we obtain judgments of the form

$$\Gamma, C \vdash_{\mathcal{I}\mathfrak{F}} e : \tau \triangleright C'$$

meaning that the expression e has type τ in the environment Γ and using the cache C. The cache C' records new discoveries during the incremental typing.

The incremental rules are in Figure 11. Most of them are trivial as they mimic the behavior of the original algorithm $\mathfrak{F}$. Again, we simply write the results of $\boxed{tr}$ and of $\boxed{\mathit{checkJoin}}$ rather than their invocations. Consider for example the rule ($\mathcal{I}\mathfrak{F}$-Let-Miss): first, the subexpressions e_2 and e_3 are incrementally type checked in the environments prescribed by the relevant calls to the function tr in Equation (2), i.e., $\boxed{\Gamma}$ and $\boxed{\Gamma[x \mapsto \tau_2]}$, respectively. Then, the type of the whole expression **let-in** is computed by $\boxed{\mathit{checkJoin}_{\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3}(\Gamma, \{\tau_2, \tau_3\}, \mathtt{out}\ \tau)}$. The predicate $\mathit{compat}_{\mathfrak{F}}$ is rather simple and uses type equality $=$ as the relation for deciding when two typing contexts are compatible:

Definition 4.10. *Let e be an expression and let Γ, Γ' be two typing contexts. Then we define*

$$\begin{aligned} \mathit{compat}_{\mathfrak{F}}(\Gamma, \Gamma', e) \triangleq\ & \mathtt{dom}\,(\Gamma) \supseteq FV(e) \wedge \mathtt{dom}\,(\Gamma') \supseteq FV(e) \wedge \\ & \forall y \in FV(e).\, \Gamma(y) = \Gamma'(y). \end{aligned}$$

$$
\begin{aligned}
&buildCache_{\mathfrak{F}}\,((c:\tau_c),\Gamma) \triangleq \{(c,\emptyset,\tau_c)\}\\
&buildCache_{\mathfrak{F}}\,((x:\tau_x),\Gamma) \triangleq \{(x,[x\mapsto\tau_x],\tau_x)\}\\
&buildCache_{\mathfrak{F}}\,((\lambda_f\,(x:\tau_x).(e:\tau_e):\tau_f),\Gamma) \triangleq \{(\lambda_f\,x:\tau_x.e:\tau_e,\Gamma_{|\,FV(\lambda_f\,(x:\tau_x).(e:\tau_e))},\tau_f)\}\\
&\qquad \cup (buildCache_{\mathfrak{F}}\,((f:\tau_f),\boxed{\Gamma})) \cup (buildCache_{\mathfrak{F}}\,((x:\tau_x),\boxed{\Gamma}))\\
&\qquad \cup (buildCache_{\mathfrak{F}}\,((e:\tau_e),\boxed{\Gamma[x\mapsto\tau_x,f\mapsto\tau_f]}))\\
&buildCache_{\mathfrak{F}}\,((\mathbf{let}\ x=e_2\ \mathbf{in}\ e_3:\tau_{let}),\Gamma) \triangleq \{(\mathbf{let}\ x=e_2\ \mathbf{in}\ e_3,\Gamma_{|\,FV(\mathbf{let}\ x=e_2\ \mathbf{in}\ e_3)},\tau_{let})\}\\
&\qquad \cup (buildCache_{\mathfrak{F}}\,((x:\tau_x),\boxed{\Gamma})) \cup (buildCache_{\mathfrak{F}}\,((e_2:\tau_2),\boxed{\Gamma}))\\
&\qquad \cup (buildCache_{\mathfrak{F}}\,((e_3:\tau_3),\boxed{\Gamma[x\mapsto\tau_x]}))\\
&buildCache_{\mathfrak{F}}\,((e_1\ \mathtt{op}\ e_2:\tau_{\mathtt{op}}),\Gamma) \triangleq \{(e_1\ \mathtt{op}\ e_2,\Gamma_{|\,FV(e_1 \mathtt{op} e_2)},\tau_{\mathtt{op}})\}\\
&\qquad \cup (buildCache_{\mathfrak{F}}\,((e_1:\tau_1),\boxed{\Gamma})) \cup (buildCache_{\mathfrak{F}}\,((e_2:\tau_2),\boxed{\Gamma}))\\
&buildCache_{\mathfrak{F}}\,((e_1\,e_2:\tau_{app}),\Gamma) \triangleq \{(e_1\,e_2,\Gamma_{|\,FV(e_1\,e_2)},\tau_{app})\}\\
&\qquad \cup (buildCache_{\mathfrak{F}}\,((e_1:\tau_1),\boxed{\Gamma})) \cup (buildCache_{\mathfrak{F}}\,((e_2:\tau_2),\boxed{\Gamma}))
\end{aligned}
$$

Figure 10: Definition of $buildCache_{\mathfrak{F}}$ for the `FUN` language.

PROVING TYPE COHERENCE The following lemma follows easily from the above definitions:

Theorem 4.3.1. *The predicate $compat_{\mathfrak{F}}$ expresses compatibility w.r.t. $=$, and $\mathfrak{F}$ preserves $=$.*

The lemma we just proved together with Theorems 4.2 to 4.4 guarantee the correctness of the incrementalized algorithm:

Corollary 4.1. *Let C be a well-formed cache, the following properties hold for $\mathfrak{I}\mathfrak{F}$:*

- ***Coherence:*** *For all e, Γ, τ, τ', and C'*

$$\Gamma \vdash_{\mathfrak{F}} e:\tau \ \wedge\ \Gamma, C \vdash_{\mathfrak{I}\mathfrak{F}} e:\tau' \triangleright C' \Rightarrow \tau = \tau';$$

- ***Completeness:*** *For all e, Γ, and τ*

$$\Gamma \vdash_{\mathfrak{F}} e:\tau \Rightarrow \exists \tau', C'.\ \Gamma, C \vdash_{\mathfrak{I}\mathfrak{F}} e:\tau' \triangleright C';$$

- ***Well-formedness preservation:*** *For all e, Γ, τ, and C'*

$$\Gamma, C \vdash_{\mathfrak{I}\mathfrak{F}} e:\tau \triangleright C' \Rightarrow \Vdash_{\mathfrak{F}} C'.$$

(𝒥ℱ-Hit)
$$\frac{C(e) = \langle \Gamma', \tau \rangle \qquad compat_{\mathcal{F}}(\Gamma, \Gamma', e)}{\Gamma, C \vdash_{\mathcal{JF}} e : \tau \rhd C}$$

(𝒥ℱ-Const-Miss)
$$\frac{\Gamma \vdash_{\mathcal{F}} c : \tau \qquad C' = C \cup \{(c, \emptyset, \tau)\}}{\Gamma, C \vdash_{\mathcal{JF}} c : \tau \rhd C'} \; miss(C, c, \Gamma)$$

(𝒥ℱ-Var-Miss)
$$\frac{\Gamma \vdash_{\mathcal{F}} x : \tau \qquad C' = C \cup \{(x, \Gamma_{|x}, \tau)\}}{\Gamma, C \vdash_{\mathcal{JF}} x : \tau \rhd C'} \; miss(C, x, \Gamma)$$

(𝒥ℱ-Abs-Miss)
$$\frac{\boxed{\Gamma[x \mapsto \tau_x, f \mapsto \tau_x \to \tau_e]}, C \vdash_{\mathcal{JF}} e : \tau_{body} \rhd C'' \qquad \boxed{\tau_{body} = \tau_e \wedge \tau = \tau_x \to \tau_e} \\ C' = C'' \cup \{(\lambda_f\, x : \tau_x.e : \tau_e, \Gamma_{|FV(\lambda_f\,(x:\tau_x).(e:\tau_e))}, \tau)\}}{\Gamma, C \vdash_{\mathcal{JF}} \lambda_f\,(x : \tau_x).(e : \tau_e) : \tau \rhd C'} \; miss(C, \lambda_f\,(x : \tau_x).(e : \tau_e), \Gamma)$$

(𝒥ℱ-Op-Miss)
$$\frac{\boxed{\Gamma}, C \vdash_{\mathcal{JF}} e_1 : \tau_1 \rhd C'' \qquad \boxed{\Gamma}, C \vdash_{\mathcal{JF}} e_2 : \tau_2 \rhd C''' \\ \boxed{\tau_1 = \tau^l_{\mathsf{op}} \wedge \tau_2 = \tau^r_{\mathsf{op}} \wedge \tau = \tau_{\mathsf{op}}} \qquad C' = C'' \cup C''' \cup \{(e_1\ \mathsf{op}\ e_2, \Gamma_{|FV(e_1 \mathsf{op} e_2)}, \tau)\}}{\Gamma, C \vdash_{\mathcal{JF}} e_1\ \mathsf{op}\ e_2 : \tau \rhd C'} \; miss(C, e_1\ \mathsf{op}\ e_2, \Gamma)$$

(𝒥ℱ-App-Miss)
$$\frac{\boxed{\Gamma}, C \vdash_{\mathcal{JF}} e_1 : \tau_x \to \tau_e \rhd C'' \\ \boxed{\Gamma}, C \vdash_{\mathcal{JF}} e_2 : \tau_2 \rhd C''' \qquad \boxed{\tau_x = \tau_2 \wedge \tau = \tau_e} \qquad C' = C'' \cup C''' \cup \{(e_1\, e_2, \Gamma_{|FV(e_1\, e_2)}, \tau)\}}{\Gamma, C \vdash_{\mathcal{JF}} e_1\, e_2 : \tau \rhd C'} \; miss(C, e_1\, e_2, \Gamma)$$

(𝒥ℱ-If-Miss) $\quad miss(C, \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, \Gamma)$
$$\frac{\boxed{\Gamma}, C \vdash_{\mathcal{JF}} e_1 : \tau_1 \rhd C'' \qquad \boxed{\Gamma}, C \vdash_{\mathcal{JF}} e_2 : \tau_2 \rhd C''' \qquad \boxed{\Gamma}, C \vdash_{\mathcal{JF}} e_3 : \tau_3 \rhd C^{iv} \\ \boxed{\tau_2 = \tau_3 \wedge \tau_1 = bool \wedge \tau = \tau_2} \qquad C' = C'' \cup C''' \cup C^{iv} \cup \{(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, \Gamma_{|FV(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)}, \tau)\}}{\Gamma, C \vdash_{\mathcal{JF}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau \rhd C'}$$

(𝒥ℱ-Let-Miss)
$$\frac{\boxed{\Gamma}, C \vdash_{\mathcal{JF}} e_2 : \tau_2 \rhd C'' \qquad \boxed{\Gamma[x \mapsto \tau_2]}, C \vdash_{\mathcal{JF}} e_3 : \tau_3 \rhd C''' \\ \boxed{\tau = \tau_3} \qquad C' = C'' \cup C''' \cup \{(\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3, \Gamma_{|FV(\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3)}, \tau)\}}{\Gamma, C \vdash_{\mathcal{JF}} \mathbf{let}\ x = e_2\ \mathbf{in}\ e_3 : \tau \rhd C'} \; miss(C, \mathbf{let}\ x = e_2\ \mathbf{in}\ e_3, \Gamma)$$

Figure 11: Rules defining incremental algorithm 𝒥ℱ to type check `FUN`, where $compat_{\mathcal{F}}$ is as in Definition 4.10.

4.3.2 *Type inference on a functional language*

In this sub-section we instantiate our schema in order to use incrementally the type inference algorithm of the language `FUN` of Section 4.3.1. Differently from the previous use case, here we deal with type inference and thus expressions need not be annotated with types. The syntax changes accordingly:

$$\begin{array}{lll} Val \ni v & ::= c \mid \lambda_f\, x.e & \mathsf{op} \in \{+, *, =, \leq\} \\ Expr \ni e & ::= v \mid x \mid e_1\ \mathsf{op}\ e_2 \mid e_1\, e_2 \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \mid \mathbf{let}\ x = e_2\ \mathbf{in}\ e_3 & \\ Type \ni \tau & ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \to \tau_2 \mid \alpha & TypeEnv \ni \Gamma ::= \emptyset \mid \Gamma[x \mapsto \tau] \end{array}$$

As above, the semantics of `FUN` is standard and we denote with τ_{op}, τ^l_{op} and τ^r_{op} the type of the result and of the operands of op. Types are now augmented with type variables $\alpha, \beta, \ldots \in TVar$. We only recall some relevant aspects below. The judgments of the type inference algorithm 𝒲 are

$$\Gamma \vdash_{\mathcal{W}} e \colon (\tau, \theta)$$

where $\theta \colon (TVar \to Type) \in Subst$ is a substitution mapping type variables into (augmented) types. As usual, we write $\theta\,\tau$ to indicate the application of the substitution θ to τ, and $\theta_2 \circ \theta_1$ stands for the composition of substitutions.

$$(\mathcal{W}\text{-Const})\quad \frac{}{\Gamma \vdash_{\mathcal{W}} c : (\tau_c, id)} \qquad (\mathcal{W}\text{-Var})\quad \frac{}{\Gamma \vdash_{\mathcal{W}} x : (\Gamma(x), id)}$$

$$(\mathcal{W}\text{-Abs})\quad \frac{\boxed{\Gamma[x \mapsto \alpha_x, f \mapsto \alpha_x \to \alpha_e]} \vdash_{\mathcal{W}} e : (\tau_e, \theta_e) \quad \boxed{\theta_1 = \mathcal{U}(\tau_e, \theta_e\alpha_e) \wedge (\tau, \theta) = ((\theta_1\,(\theta_e\,\alpha_x)) \to (\theta_1\,\tau_e), \theta_1 \circ \theta_e)}}{\Gamma \vdash_{\mathcal{W}} \lambda_f\, x.e : (\tau, \theta)}\ \alpha_x, \alpha_e \text{ fresh}$$

$$(\mathcal{W}\text{-Op})\quad \frac{\boxed{\Gamma} \vdash_{\mathcal{W}} e_1 : (\tau_1, \theta_1) \quad \boxed{\theta_1\,\Gamma} \vdash_{\mathcal{W}} e_2 : (\tau_2, \theta_2) \quad \boxed{\theta_3 = \mathcal{U}(\theta_2\,\tau_1, \tau^l_{\text{op}}) \wedge \theta_4 = \mathcal{U}(\theta_3\,\tau_2, \tau^r_{\text{op}}) \wedge (\tau, \theta) = (\tau_{\text{op}}, \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1)}}{\Gamma \vdash_{\mathcal{W}} e_1\ \mathbf{op}\ e_2 : (\tau, \theta)}$$

$$(\mathcal{W}\text{-App})\quad \frac{\boxed{\Gamma} \vdash_{\mathcal{W}} e_1 : (\tau_1, \theta_1) \quad \boxed{\theta_1\,\Gamma} \vdash_{\mathcal{W}} e_2 : (\tau_2, \theta_2) \quad \boxed{\theta_3 = \mathcal{U}(\theta_2\,\tau_1, \tau_2 \to \alpha) \wedge (\tau, \theta) = (\theta_3\,\alpha, \theta_3 \circ \theta_2 \circ \theta_1)}}{\Gamma \vdash_{\mathcal{W}} e_1\, e_2 : (\tau, \theta)}\ \alpha \text{ fresh}$$

$$(\mathcal{W}\text{-If})\quad \frac{\boxed{\Gamma} \vdash_{\mathcal{W}} e_1 : (\tau_1, \theta_1) \quad \boxed{\theta_1\,\Gamma} \vdash_{\mathcal{W}} e_2 : (\tau_2, \theta_2) \quad \boxed{\theta_2(\theta_1\,\Gamma)} \vdash_{\mathcal{W}} e_3 : (\tau_3, \theta_3) \quad \boxed{\theta_4 = \mathcal{U}(\theta_3(\theta_2\,\tau_1), bool) \wedge \theta_5 = \mathcal{U}(\theta_4\,\tau_3, \theta_4(\theta_3\,\tau_2)) \wedge (\tau, \theta) = (\theta_5(\theta_4\,\tau_3), \theta_5 \circ \theta_4 \circ \theta_3 \circ \theta_2)}}{\Gamma \vdash_{\mathcal{W}} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : (\tau, \theta)}$$

$$(\mathcal{W}\text{-Let})\quad \frac{\boxed{\Gamma} \vdash_{\mathcal{W}} e_2 : (\tau_2, \theta_2) \quad \boxed{(\theta_2\,\Gamma)[x \mapsto \tau_2]} \vdash_{\mathcal{W}} e_3 : (\tau_3, \theta_3) \quad \boxed{(\tau, \theta) = (\tau_3, \theta_3 \circ \theta_2)}}{\Gamma \vdash_{\mathcal{W}} \mathbf{let}\ x = e_2\ \mathbf{in}\ e_3 : (\tau, \theta)}$$

Figure 12: Rules defining algorithm $\mathcal{W}$ to infer FUN types of Section 4.3.2.

For the sake of completeness, in Figure 12 we show the classical inference algorithm $\mathcal{W}$ (see e.g., [166]), where we assume constants c to have a fixed and known type, and $\mathcal{U}$ to be the standard type unification algorithm. As usual, in the resulting set of rules we have colored and framed the parts that drive the definitions results of $\boxed{tr}$ and $\boxed{checkJoin}$, so making clear that they occur unchanged in the definition of the incremental inference algorithm $\mathcal{IW}$.

DEFINING THE SHAPE OF CACHES Entries in the cache differ slightly from those of the previous sub-section, since now the result of the inference algorithm $\mathcal{W}$ also includes a substitution. Thus, in this use case each entry is a triple $(e, \Gamma, (\tau, \theta))$ and a cache is

$$C \in Cache = \wp(Expr \times TypeEnv \times (Type \times Subst))$$

BUILDING CACHES The function $buildCache_{\mathcal{W}}$ is easily defined in Figure 13, similarly as we did above.

INCREMENTAL TYPING In Figure 14 we display the rules defining the algorithm $\mathcal{IW}$ with judgments of the following form

$$\Gamma, C \vdash_{\mathcal{IW}} e : (\tau, \theta) \triangleright C'$$

Most of the rules mimic the behavior of algorithm $\mathcal{W}$, following the templates of Section 4.2. Consider for example the rule ($\mathcal{IW}$-Let-Miss): first, the types of e_1 and e_2 are incrementally inferred in the environments prescribed by the relevant calls to the function tr. The result associated with the whole expression **let-in** is then the pair $(\tau_2, \theta_2 \circ \theta_1)$, where θ_1 and θ_2 are the substitutions obtained recursively from e_1 and e_2, respectively. Also in this case the

$$
\begin{aligned}
&\mathit{buildCache}_{\mathcal{W}}\,((c : (\tau_c, \theta)), \Gamma) \triangleq \{(c, \emptyset, (\tau_c, \theta))\} \\
&\mathit{buildCache}_{\mathcal{W}}\,((x : (\tau_x, \theta)), \Gamma) \triangleq \{(x, [x \mapsto \tau_x], (\tau_x, \theta))\} \\
&\mathit{buildCache}_{\mathcal{W}}\,((\lambda_f\, x.e : (\tau_f, \theta_f)), \Gamma) \triangleq \{(\lambda_f\, x.e, \Gamma_{|\,FV(\lambda_f\, x.e)}, (\tau_f, \theta_f))\} \\
&\qquad \cup (\mathit{buildCache}_{\mathcal{W}}\,((f : (\tau_f, \theta_f)), \boxed{\Gamma})) \cup (\mathit{buildCache}_{\mathcal{W}}\,((x : (\tau_x, \theta_x)), \boxed{\Gamma})) \\
&\qquad \cup (\mathit{buildCache}_{\mathcal{W}}\,((e : (\tau_e, \theta_e)), \boxed{\Gamma[x \mapsto \tau_x, f \mapsto \tau_f]})) \\
&\mathit{buildCache}_{\mathcal{W}}\,((\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3 : (\tau_{let}, \theta_{let})), \Gamma) \triangleq \\
&\qquad \{(\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3, \Gamma_{|\,FV(\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3)}, (\tau_{let}, \theta_{let}))\} \\
&\qquad \cup (\mathit{buildCache}_{\mathcal{W}}\,((x : (\tau_x, \theta_x)), \boxed{\Gamma})) \cup (\mathit{buildCache}_{\mathcal{W}}\,((e_2 : (\tau_2, \theta_2)), \boxed{\Gamma})) \\
&\qquad \cup (\mathit{buildCache}_{\mathcal{W}}\,((e_3 : (\tau_3, \theta_3)), \boxed{\Gamma[x \mapsto \tau_x]})) \\
&\mathit{buildCache}_{\mathcal{W}}\,((e_1\ \mathtt{op}\ e_2 : (\tau_{\mathtt{op}}, \theta_{\mathtt{op}})), \Gamma) \triangleq \{(e_1\ \mathtt{op}\ e_2, \Gamma_{|\,FV(e_1 \mathtt{op} e_2)}, (\tau_{\mathtt{op}}, \theta_{\mathtt{op}}))\} \\
&\qquad \cup (\mathit{buildCache}_{\mathcal{W}}\,((e_1 : (\tau_1, \theta_1), \boxed{\Gamma})) \cup (\mathit{buildCache}_{\mathcal{W}}\,((e_2 : (\tau_2, \theta_2), \boxed{\Gamma})) \\
&\mathit{buildCache}_{\mathcal{W}}\,((e_1\, e_2 : (\tau_{app}, \theta_{app})), \Gamma) \triangleq \{(e_1\, e_2, \Gamma_{|\,FV(e_1\, e_2)}, (\tau_{app}, \theta_{app}))\} \\
&\qquad \cup (\mathit{buildCache}_{\mathcal{W}}\,((e_1 : (\tau_1, \theta_1)), \boxed{\Gamma})) \cup (\mathit{buildCache}_{\mathcal{W}}\,((e_2 : (\tau_2, \theta_2)), \boxed{\Gamma}))
\end{aligned}
$$

Figure 13: Definition of $\mathit{buildCache}_{\mathcal{W}}$ for the incremental type inference of FUN.

predicate $\mathit{compat}_{\mathcal{W}}$ uses type equality $=$ as the relation for deciding when two contexts are compatible:

Definition 4.11. *Let e be an expression and let Γ, Γ' be two typing environments. Then we define*

$$
\begin{aligned}
\mathit{compat}_{\mathcal{W}}(\Gamma, \Gamma', e) \triangleq\ & \mathtt{dom}\,(\Gamma) \supseteq FV(e) \wedge \mathtt{dom}\,(\Gamma') \supseteq FV(e) \wedge \\
& \forall y \in FV(e)\,.\,\Gamma(y) = \Gamma'(y)
\end{aligned}
$$

For the sake of presentation, we keep our $\mathit{compat}_{\mathcal{W}}$ quite restrictive, even though it fails to detect some cases of compatibility between typing environments (e.g., when $\mathcal{W}$ chooses different fresh variables for the same term). One alternative would be to substitute equality with unificability both in the definition above and in Theorem 4.5 below.

PROVING TYPE COHERENCE The following lemma together with Theorems 4.2 to 4.4 ensure the correctness of $\mathcal{IW}$ (Corollary 4.2):

Theorem 4.5. *The predicate $\mathit{compat}_{\mathcal{W}}$ expresses compatibility w.r.t. $=$, and $\mathcal{W}$ preserves $=$.*

Corollary 4.2. *Let C be a well-formed cache, the following properties hold for $\mathcal{IW}$:*

- ***Coherence:*** *For all e, Γ, (τ, θ), (τ', θ'), and C'*

$$\Gamma \vdash_{\mathcal{W}} e : (\tau, \theta)\ \wedge\ \Gamma, C \vdash_{\mathcal{IW}} e : (\tau', \theta') \triangleright C' \Rightarrow (\tau, \theta) = (\tau', \theta');$$

- ***Completeness:*** *For all e, Γ, and (τ, θ)*

$$\Gamma \vdash_{\mathcal{W}} e : (\tau, \theta) \Rightarrow \exists (\tau', \theta'), C'.\, \Gamma, C \vdash_{\mathcal{IW}} e : (\tau', \theta') \triangleright C';$$

- ***Well-formedness preservation:*** *For all e, Γ, (τ, θ), and C'*

$$\Gamma, C \vdash_{\mathcal{IW}} e : (\tau, \theta) \triangleright C' \Rightarrow \Vdash_{\mathcal{W}} C'.$$

$$(\mathcal{FW}\text{-Hit})\ \frac{C(e) = \langle \Gamma', (\tau, \theta)\rangle \qquad compat_{\mathcal{W}}(\Gamma, \Gamma', e)}{\Gamma, C \vdash_{\mathcal{FW}} e : (\tau, \theta) \triangleright C}$$

$$(\mathcal{FW}\text{-Const-Miss})\ \frac{\Gamma \vdash_{\mathcal{W}} c : (\tau, \theta) \qquad C' = C \cup \{(c, \emptyset, (\tau, \theta))\}}{\Gamma, C \vdash_{\mathcal{FW}} c : (\tau, \theta) \triangleright C'}\ miss(C, c, \Gamma)$$

$$(\mathcal{FW}\text{-Var-Miss})\ \frac{\Gamma \vdash_{\mathcal{W}} x : (\tau, \theta) \qquad C' = C \cup \{(x, \Gamma_{|x}, (\tau, \theta))\}}{\Gamma, C \vdash_{\mathcal{FW}} x : (\tau, \theta) \triangleright C'}\ miss(C, x, \Gamma)$$

$$(\mathcal{FW}\text{-Abs-Miss})\ \frac{\begin{array}{c}\boxed{\Gamma[x \mapsto \alpha_x, f \mapsto \alpha_x \to \alpha_e]}, C \vdash_{\mathcal{FW}} e : (\tau_e, \theta_e) \triangleright C'' \\ \boxed{\theta_1 = \mathcal{U}(\tau_e, \theta_e \alpha_e) \wedge (\tau, \theta) = ((\theta_1\ (\theta_e\ \alpha_x)) \to (\theta_1\ \tau_e), \theta_1 \circ \theta_e)} \\ C' = C'' \cup \{(\lambda_f\ x : \tau_x.e, \Gamma_{|FV(\lambda_f\ x:\tau_x.e)}, (\tau, \theta))\}\end{array}}{\Gamma, C \vdash_{\mathcal{FW}} \lambda_f\ x.e : (\tau, \theta) \triangleright C'}\ miss(C, \lambda_f\ x.e, \Gamma) \wedge \alpha_x, \alpha_e \text{ fresh}$$

$$(\mathcal{FW}\text{-Op-Miss})\ \frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{FW}} e_1 : (\tau_1, \theta_1) \triangleright C'' \qquad \boxed{\theta_1\ \Gamma}, C \vdash_{\mathcal{FW}} e_2 : (\tau_2, \theta_2) \triangleright C''' \\ \boxed{\theta_3 = \mathcal{U}(\theta_2\ \tau_1, \tau^l_{\text{op}}) \wedge \theta_4 = \mathcal{U}(\theta_3\ \tau_2, \tau^r_{\text{op}}) \wedge (\tau, \theta) = (\tau_{\text{op}}, \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1)} \\ C' = C'' \cup C''' \cup \{(e_1 \text{ op } e_2, \Gamma_{|FV(e_1 \text{op} e_2)}, (\tau, \theta))\}\end{array}}{\Gamma, C \vdash_{\mathcal{FW}} e_1 \text{ op } e_2 : (\tau, \theta) \triangleright C'}\ miss(C, e_1 \text{ op } e_2, \Gamma)$$

$$(\mathcal{FW}\text{-App-Miss})\ \frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{FW}} e_1 : (\tau_1, \theta_1) \to \tau_e \triangleright C'' \qquad \boxed{\theta_1\ \Gamma}, C \vdash_{\mathcal{FW}} e_2 : (\tau_2, \theta_2) \triangleright C''' \\ \boxed{\theta_3 = \mathcal{U}(\theta_2\ \tau_1, \tau_2 \to \alpha) \wedge (\tau, \theta) = (\theta_3\ \alpha, \theta_3 \circ \theta_2 \circ \theta_1)} \\ C' = C'' \cup C''' \cup \{(e_1\ e_2, \Gamma_{|FV(e_1\ e_2)}, (\tau, \theta))\}\end{array}}{\Gamma, C \vdash_{\mathcal{FW}} e_1\ e_2 : (\tau, \theta) \triangleright C'}\ miss(C, e_1\ e_2, \Gamma) \wedge \alpha \text{ fresh}$$

$$(\mathcal{FW}\text{-If-Miss})\ \frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{FW}} e_1 : (\tau_1, \theta_1) \triangleright C'' \qquad \boxed{\theta_1\ \Gamma}, C \vdash_{\mathcal{FW}} e_2 : (\tau_2, \theta_2) \triangleright C''' \qquad \boxed{\theta_2(\theta_1\ \Gamma)}, C \vdash_{\mathcal{FW}} e_3 : (\tau_3, \theta_3) \triangleright C^{iv} \\ \boxed{\theta_4 = \mathcal{U}(\theta_3(\theta_2\ \tau_1), bool) \wedge \theta_5 = \mathcal{U}(\theta_4\ \tau_3, \theta_4(\theta_3\ \tau_1)) \wedge (\tau, \theta) = (\theta_5(\theta_4\ \tau_3), \theta_5 \circ \theta_4 \circ \theta_3 \circ \theta_2)} \\ C' = C'' \cup C''' \cup C^{iv} \cup \{(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, \Gamma_{|FV(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)}, (\tau, \theta))\}\end{array}}{\Gamma, C \vdash_{\mathcal{FW}} \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : (\tau, \theta) \triangleright C'}\ miss(C, \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, \Gamma)$$

$$(\mathcal{FW}\text{-Let-Miss})\ \frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{FW}} e_2 : (\tau_2, \theta_2) \triangleright C'' \qquad \boxed{(\theta_1\ \Gamma)[x \mapsto \tau_2]}, C \vdash_{\mathcal{FW}} e_3 : (\tau_3, \theta_3) \triangleright C''' \\ \boxed{(\tau, \theta) = (\tau_3, \theta_3 \circ \theta_1)} \qquad C' = C'' \cup C''' \cup \{(\textbf{let } x = e_2 \textbf{ in } e_3, \Gamma_{|FV(\textbf{let } x = e_2 \textbf{ in } e_3)}, (\tau, \theta))\}\end{array}}{\Gamma, C \vdash_{\mathcal{FW}} \textbf{let } x = e_2 \textbf{ in } e_3 : (\tau, \theta) \triangleright C'}\ miss(C, \textbf{let } x = e_1 \textbf{ in } e_3, \Gamma)$$

Figure 14: Rules defining incremental algorithm $\mathcal{FW}$ to infer FUN types, where $compat_{\mathcal{W}}$ is as in Definition 4.11.

4.3.3 *Type checking non-interference*

Here we show how to make incremental the typing algorithm $\mathcal{S}$ of Volpano-Smith-Irvine [217, 229] for checking non-interference policies, obtaining the algorithm $\mathcal{FS}$. We assume that the variables of programs are classified either as high, H, or low L. Intuitively, a program enjoys the non-interference property when the values of low variables do not depend on those of high ones.

As usual, assume a simple imperative language WHILE, whose syntax is below (Var denotes the set of program variables).

$$\begin{array}{rll}
a & ::= n \mid x \mid a_1\ \mathtt{op_a}\ a_2 & n \in \mathbb{N} \quad \mathtt{op_a} \in \{+, *, -, \ldots\} \quad x \in Var \\
b & ::= \mathtt{true} \mid \mathtt{false} \mid b_1\ \mathtt{or}\ b_2 \mid \mathtt{not}\ b \mid a_1 \leq a_2 & \\
Stmt \ni c & ::= \mathtt{skip} \mid x := a \mid c_1\mathtt{;}\ c_2 \mid \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \mid \mathtt{while}\ b\ \mathtt{do}\ c & \\
Term \ni p & ::= a \mid b \mid c & DType \ni \tau ::= H \mid L \\
PType \ni \varsigma & ::= \tau \mid \tau\ var \mid \tau\ cmd & TypeEnv \ni \Gamma ::= \emptyset \mid \Gamma[p \mapsto \varsigma]
\end{array}$$

The type checking algorithm has judgments of the form

$$\Gamma \vdash_{\mathcal{S}} p : \varsigma$$

where $\varsigma \in PType$, and its rules are in Figure 15, where as usual we have colored and framed the results of $\boxed{tr}$ and $\boxed{checkJoin}$. In the following we assume that the initial typing

$$\text{(S-CONST)}\ \frac{}{\Gamma \vdash_{\mathcal{S}} c : L} \qquad \text{(S-VAR)}\ \frac{\boxed{\Gamma(x) = \tau\ var \wedge \varsigma = \tau}}{\Gamma \vdash_{\mathcal{S}} x : \varsigma} \qquad \text{(S-NOT)}\ \frac{\Gamma \vdash_{\mathcal{S}} b : \tau_b \qquad \boxed{\tau = \tau_b}}{\Gamma \vdash_{\mathcal{S}} \texttt{not}\ b : \tau} \qquad \text{(S-SKIP)}\ \frac{\boxed{\varsigma = H\ cmd}}{\Gamma \vdash_{\mathcal{S}} \texttt{skip} : \varsigma}$$

$$\text{(S-OP)}\ \frac{\boxed{\Gamma} \vdash_{\mathcal{S}} p_0 : \tau_0 \qquad \boxed{\Gamma} \vdash_{\mathcal{S}} p_1 : \tau_1 \qquad \boxed{op \in \{+, *, -, \texttt{or}, \leq, \ldots\} \wedge \tau_0 = \tau_1 \wedge \varsigma = \tau_0}}{\Gamma \vdash_{\mathcal{S}} a_0\ \texttt{op}\ a_1 : \varsigma}$$

$$\text{(S-ASSIGN)}\ \frac{\boxed{\Gamma} \vdash_{\mathcal{S}} a : \tau_a \qquad \boxed{\Gamma(x) = \tau\ var \wedge \tau = \tau_a \wedge \varsigma = \tau\ cmd}}{\Gamma \vdash_{\mathcal{S}} x := a : \varsigma}$$

$$\text{(S-IF)}\ \frac{\boxed{\Gamma} \vdash_{\mathcal{S}} b : \tau_b \qquad \boxed{\Gamma} \vdash_{\mathcal{S}} c_1 : \tau_1\ cmd \qquad \boxed{\Gamma} \vdash_{\mathcal{S}} c_2 : \tau_2\ cmd \qquad \boxed{\tau_b = \tau_1 = \tau_2 \wedge \varsigma = \tau_b\ cmd}}{\Gamma \vdash_{\mathcal{S}} \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \varsigma}$$

$$\text{(S-WHILE)}\ \frac{\boxed{\Gamma} \vdash_{\mathcal{S}} b : \tau_b \qquad \boxed{\Gamma} \vdash_{\mathcal{S}} c_1 : \tau_1\ cmd \qquad \boxed{\tau_b = \tau_1 \wedge \varsigma = \tau_b\ cmd}}{\Gamma \vdash_{\mathcal{S}} \texttt{while}\ b\ \texttt{do}\ c_1 : \varsigma}$$

$$\text{(S-SEQ)}\ \frac{\boxed{\Gamma} \vdash_{\mathcal{S}} c_1 : \tau_1\ cmd \qquad \boxed{\Gamma} \vdash_{\mathcal{S}} c_2 : \tau_2\ cmd \qquad \boxed{\tau_1 = \tau_2 \wedge \varsigma = \tau_1\ cmd}}{\Gamma \vdash_{\mathcal{S}} c_1 ;\ c_2 : \varsigma} \qquad \text{(SS-SUB)}\ \frac{\Gamma \vdash_{\mathcal{S}} p : \varsigma_1 \qquad \varsigma_1 \subseteq \varsigma_2}{\Gamma \vdash_{\mathcal{S}} p : \varsigma_2} \qquad \text{(SS-BASE)}\ \frac{}{L \subseteq H}$$

$$\text{(SS-CMD)}\ \frac{\tau' \subseteq \tau}{\tau\ cmd \subseteq \tau'\ cmd} \qquad \text{(SS-REFL)}\ \frac{}{\varsigma \subseteq \varsigma} \qquad \text{(SS-TR)}\ \frac{\varsigma_1 \subseteq \varsigma_2 \qquad \varsigma_2 \subseteq \varsigma_3}{\varsigma_1 \subseteq \varsigma_3}$$

Figure 15: The rules of the type checking algorithm $\mathcal{S}$ (with subtyping) for WHILE of Section 4.3.4.

environment Γ contains the security level of each variable occurring in the program in hand.

DEFINING THE SHAPE OF CACHES Here, terms are elements of the set $Phrase$, typing environments are in $TypeEnv$, and results belong to $PType$. Thus, the shape of the caches is:

$$C \in Cache = \wp(Phrase \times TypeEnv \times PType).$$

BUILDING CACHES We build the cache by visiting the aAST and "reconstructing" the typing environment. The function $buildCache_{\mathcal{S}}$ is in Figure 16, where for brevity we have directly used the results of tr rather than writing the needed invocations.

INCREMENTAL TYPING In Figure 17 we display the rules defining the algorithm $\mathcal{IS}$ with judgments of the following form

$$\Gamma, C \vdash_{\mathcal{IS}} p : \varsigma \triangleright C'$$

Most of the rules are trivial instantiations of rules in Section 4.2 that mimic those of the original type checking algorithm. Of course, $\mathcal{IS}$ inherits unchanged the subtyping relation of $\mathcal{S}$ and applies it when needed. The predicate $compat_{\mathcal{S}}$ uses again type equality $=$:

$$
\begin{aligned}
&\mathit{buildCaches}_{\mathcal{S}}\left((c : L), \Gamma\right) \triangleq \{(c, \emptyset, L)\} \qquad c \in \mathbb{N} \cup \{\mathit{true}, \mathit{false}\} \\
&\mathit{buildCaches}_{\mathcal{S}}\left((x : \tau), \Gamma\right) \triangleq \{(x, [x \mapsto \tau\ \mathit{var}], \tau)\} \\
&\mathit{buildCaches}_{\mathcal{S}}\left((a_1\ \texttt{op}\ a_2 : \tau), \Gamma\right) \triangleq \{(a_1\ \texttt{op}\ a_2, \Gamma_{|\,FV(a_1 \texttt{op} a_2)}, \tau)\} \\
&\qquad \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((a_1 : \tau_1), \boxed{\Gamma}\right)\right) \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((a_2 : \tau_2), \boxed{\Gamma}\right)\right) \\
&\mathit{buildCaches}_{\mathcal{S}}\left((a_1 \leq a_2 : \tau), \Gamma\right) \triangleq \{(a_1 \leq a_2, \Gamma_{|\,FV(a_1 \leq a_2)}, \tau)\} \\
&\qquad \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((a_1 : \tau_1), \boxed{\Gamma}\right)\right) \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((a_2 : \tau_2), \boxed{\Gamma}\right)\right) \\
&\mathit{buildCaches}_{\mathcal{S}}\left((b_1\ \texttt{or}\ b_2 : \tau), \Gamma\right) \triangleq \{(b_1\ \texttt{or}\ b_2, \Gamma_{|\,FV(b_1 \texttt{or} b_2)}, \tau)\} \\
&\qquad \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((b_1 : \tau_1), \boxed{\Gamma}\right)\right) \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((b_2 : \tau_2), \boxed{\Gamma}\right)\right) \\
&\mathit{buildCaches}_{\mathcal{S}}\left((\texttt{not}\ b : \tau), \Gamma\right) \triangleq \{(\texttt{not}\ b, \Gamma_{|\,FV(\texttt{not} b)}, \tau)\} \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((b : \tau), \boxed{\Gamma}\right)\right) \\
&\mathit{buildCaches}_{\mathcal{S}}\left((\texttt{skip} : H\ \mathit{cmd}), \Gamma\right) \triangleq \{(\texttt{skip}, \emptyset, H\ \mathit{cmd})\} \\
&\mathit{buildCaches}_{\mathcal{S}}\left((x := a : \tau\ \mathit{cmd}), \Gamma\right) \triangleq \{(x := a, \Gamma_{|\,FV(x:=a)}, \tau\ \mathit{cmd})\} \\
&\qquad \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((x : \tau_x), \boxed{\Gamma}\right)\right) \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((a : \tau_a), \boxed{\Gamma}\right)\right) \\
&\mathit{buildCaches}_{\mathcal{S}}\left((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \tau\ \mathit{cmd}), \Gamma\right) \triangleq \\
&\qquad \{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \Gamma_{|\,FV(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)}, \tau\ \mathit{cmd})\} \\
&\qquad \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((b : \tau_b), \boxed{\Gamma}\right)\right) \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((c_1 : \tau_1\ \mathit{cmd}), \boxed{\Gamma}\right)\right) \\
&\qquad \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((c_2 : \tau_2\ \mathit{cmd}), \boxed{\Gamma}\right)\right) \\
&\mathit{buildCaches}_{\mathcal{S}}\left((\texttt{while}\ b\ \texttt{do}\ c : \tau\ \mathit{cmd}), \Gamma\right) \triangleq \{(\texttt{while}\ b\ \texttt{do}\ c, \Gamma_{|\,FV(\texttt{while}\ b\ \texttt{do}\ c)}, \tau\ \mathit{cmd})\} \\
&\qquad \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((b : \tau_b), \boxed{\Gamma}\right)\right) \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((c : \tau_c\ \mathit{cmd}), \boxed{\Gamma}\right)\right) \\
&\mathit{buildCaches}_{\mathcal{S}}\left((c_1\texttt{;}\ c_2 : \tau\ \mathit{cmd}), \Gamma\right) \triangleq \{(c_1\texttt{;}\ c_2, \Gamma_{|\,FV(c_1\texttt{;}\ c_2)}, \tau\ \mathit{cmd})\} \\
&\qquad \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((c_1 : \tau_1\ \mathit{cmd}), \boxed{\Gamma}\right)\right) \cup \left(\mathit{buildCaches}_{\mathcal{S}}\left((c_2 : \tau_2\ \mathit{cmd}), \boxed{\Gamma}\right)\right)
\end{aligned}
$$

Figure 16: Definition of $\mathit{buildCaches}_{\mathcal{S}}$ for the incremental type checking of WHILE.

Definition 4.12. *Let $p \in \mathit{Term}$ and Γ, Γ' be two typing contexts. Then we define*

$$
\begin{aligned}
\mathit{compat}_{\mathcal{S}}(\Gamma, \Gamma', p) \triangleq \texttt{dom}\,(\Gamma) \supseteq FV(p) \wedge \texttt{dom}\,(\Gamma') \supseteq FV(p) \wedge \\
\forall y \in FV(p)\,.\,\Gamma(y) = \Gamma'(y).
\end{aligned}
$$

PROVING TYPE COHERENCE As for the previous two use cases, the following lemma together with Theorems 4.2 to 4.4 ensure the correctness of $\mathcal{IS}$ (Corollary 4.3):

Theorem 4.3.2. *The predicate $\mathit{compat}_{\mathcal{S}}$ expresses compatibility w.r.t. $=$, and $\mathcal{S}$ preserves $=$.*

Corollary 4.3. *Let C be a well-formed cache, the following properties hold for $\mathcal{IS}$:*

- ***Coherence:** For all p, Γ, ς, ς', and C'*

$$\Gamma \vdash_{\mathcal{S}} p : \varsigma \ \wedge\ \Gamma, C \vdash_{\mathcal{IS}} p : \varsigma' \triangleright C' \Rightarrow \varsigma = \varsigma';$$

- ***Completeness:** For all p, Γ, and ς it holds that*

$$\Gamma \vdash_{\mathcal{S}} p : \varsigma \Rightarrow \exists \varsigma', C'.\ \Gamma, C \vdash_{\mathcal{IS}} p : \varsigma' \triangleright C';$$

- ***Well-formedness preservation:** For all p, Γ, ς, and C' it holds that*

$$\Gamma, C \vdash_{\mathcal{IS}} p : \varsigma \triangleright C' \Rightarrow \Vdash_{\mathcal{S}} C'.$$

$$(\mathcal{IS}\text{-Hit})\quad \frac{C(p) = \langle \Gamma', \varsigma \rangle \qquad compat_{\mathcal{S}}(\Gamma, \Gamma', p)}{\Gamma, C \vdash_{\mathcal{IS}} p : \varsigma \triangleright C}$$

$$(\mathcal{IS}\text{-Const-Miss})\quad \frac{\emptyset \vdash_{\mathcal{S}} c : \varsigma \qquad C' = C \cup \{(c, \emptyset, \varsigma)\}}{\Gamma, C \vdash_{\mathcal{IS}} c : \varsigma \triangleright C'}\ miss(C, c, \Gamma)$$

$$(\mathcal{IS}\text{-Var-Miss})\quad \frac{\Gamma \vdash_{\mathcal{S}} x : \varsigma \qquad C' = C \cup \{(x, \Gamma_{|\,x}, \varsigma)\}}{\Gamma, C \vdash_{\mathcal{IS}} x : \varsigma \triangleright C'}\ miss(C, x, \Gamma)$$

$$(\mathcal{IS}\text{-Op-Miss})\quad \frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{IS}} a_1 : \tau_1 \triangleright C'' \qquad \boxed{\Gamma}, C \vdash_{\mathcal{IS}} a_2 : \tau_2 \triangleright C''' \\ \tau_1 = \tau_2 \wedge \varsigma = \tau_1 \qquad C' = C'' \cup C''' \cup \{(a_1 \texttt{ op } a_2, \Gamma_{|\,FV(a_1 \texttt{op} a_2)}, \varsigma)\}\end{array}}{\Gamma, C \vdash_{\mathcal{IS}} a_1 \texttt{ op } a_2 : \varsigma \triangleright C'}\ miss(C, a_1 \texttt{ op } a_2, \Gamma)$$

$$(\mathcal{IS}\text{-BOp-Miss})\quad \frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{IS}} b_1 : \tau_1 \triangleright C'' \qquad \boxed{\Gamma}, C \vdash_{\mathcal{IS}} b_2 : \tau_2 \triangleright C''' \\ \tau_1 = \tau_2 \wedge \varsigma = \tau_1 \qquad C' = C'' \cup C''' \cup \{(b_1 \texttt{ or } b_2, \Gamma_{|\,FV(b_1 \texttt{or} b_2)}, \varsigma)\}\end{array}}{\Gamma, C \vdash_{\mathcal{IS}} b_1 \texttt{ or } b_2 : \varsigma \triangleright C'}\ miss(C, b_1 \texttt{ or } b_2, \Gamma)$$

$$(\mathcal{IS}\text{-Not-Miss})\quad \frac{\boxed{\Gamma}, C \vdash_{\mathcal{IS}} b : \tau \triangleright C'' \qquad C' = C'' \cup \{(\texttt{ not } b, \Gamma_{|\,FV(\texttt{ not} b)}, \tau)\}}{\boxed{\Gamma}, C \vdash_{\mathcal{IS}} \texttt{not } b : \tau \triangleright C'}\ miss(C, \texttt{ not } b, \Gamma)$$

$$(\mathcal{IS}\text{-Leq-Miss})\quad \frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{IS}} a_1 : \tau_1 \triangleright C'' \qquad \boxed{\Gamma}, C \vdash_{\mathcal{IS}} a_2 : \tau_2 \triangleright C''' \\ \tau_1 = \tau_2 \wedge \varsigma = \tau_1 \qquad C' = C'' \cup C''' \cup \{(a_1 \leq a_2, \Gamma_{|\,FV(a_1 \leq a_2)}, \varsigma)\}\end{array}}{\Gamma, C \vdash_{\mathcal{IS}} a_1 \leq a_2 : \varsigma \triangleright C'}\ miss(C, a_1 \leq a_2, \Gamma)$$

$$(\mathcal{IS}\text{-Skip-Miss})\quad miss(C, \texttt{skip}, \Gamma)\quad \frac{\Gamma \vdash_{\mathcal{S}} \texttt{skip} : \varsigma \qquad C' = C \cup \{(\texttt{skip}, \emptyset, \varsigma)\}}{\Gamma, C \vdash_{\mathcal{IS}} \texttt{skip} : \varsigma \triangleright C'}$$

$$(\mathcal{IS}\text{-Assign-Miss})\quad \frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{IS}} x : \tau_x\ var \triangleright C'' \qquad \boxed{\Gamma}, C \vdash_{\mathcal{IS}} a : \tau_a \triangleright C''' \\ \tau_a = \tau_x \wedge \varsigma = \tau_a\ cmd \qquad C' = C'' \cup C''' \cup \{(x := a, \Gamma_{|\,FV(x:=a)}, \varsigma)\}\end{array}}{\Gamma, C \vdash_{\mathcal{IS}} x := a : \varsigma \triangleright C'}\ miss(C, x := a, \Gamma)$$

$$(\mathcal{IS}\text{-If-Miss})\quad miss(C, \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2, \Gamma)\quad \frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{IS}} b : \tau_b \triangleright C'' \qquad \boxed{\Gamma}, C \vdash_{\mathcal{IS}} c_1 : \tau_1\ cmd \triangleright C''' \qquad \boxed{\Gamma}, C \vdash_{\mathcal{IS}} c_2 : \tau_2\ cmd \triangleright C^{iv} \\ \tau_1 = \tau_2 = \tau_b \wedge \varsigma = \tau_1\ cmd \qquad C' = C'' \cup C''' \cup C^{iv} \cup \{(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2, \Gamma_{|\,FV(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2)}, \varsigma)\}\end{array}}{\Gamma, C \vdash_{\mathcal{IS}} \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 : \varsigma \triangleright C'}$$

$$(\mathcal{IS}\text{-While-Miss})\quad \frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{IS}} b : \tau_b \triangleright C'' \qquad \boxed{\Gamma}, C \vdash_{\mathcal{IS}} c : \tau_1\ cmd \triangleright C''' \\ \tau_1 = \tau_b \wedge \varsigma = \tau_1\ cmd \qquad C' = C'' \cup C''' \cup \{(\texttt{while } b \texttt{ do } c, \Gamma_{|\,FV(\texttt{while } b \texttt{ do } c)}, \varsigma)\}\end{array}}{\Gamma, C \vdash_{\mathcal{IS}} \texttt{while } b \texttt{ do } c : \varsigma \triangleright C'}\ miss(C, \texttt{while } b \texttt{ do } c, \Gamma)$$

$$(\mathcal{IS}\text{-Seq-Miss})\quad \frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{IS}} c_1 : \tau_1\ cmd \triangleright C'' \qquad \boxed{\Gamma}, C \vdash_{\mathcal{IS}} c_2 : \tau_2\ cmd \triangleright C''' \\ \tau_1 = \tau_2 \wedge \varsigma = \tau_1\ cmd \qquad C' = C'' \cup C''' \cup \{(c_1 \texttt{;}\ c_2, \Gamma_{|\,FV(c_1 \texttt{;}\ c_2)}, \varsigma)\}\end{array}}{\Gamma, C \vdash_{\mathcal{IS}} c_1 \texttt{;}\ c_2 : \varsigma \triangleright C'}\ miss(C, c_1 \texttt{;}\ c_2, \Gamma)$$

Figure 17: Rules defining the incremental algorithm $\mathcal{IS}$ to type check `WHILE`, where $compat_{\mathcal{S}}$ is as in Definition 4.12.

4.3.4 *Type checking robust declassification*

We consider here the type system $\mathfrak{R}$ for non-interference with robust declassification of Myers et al. [159], briefly surveyed below.

4.3.4.1 *A survey of the type system* $\mathfrak{R}$

Assume a security lattice $\mathcal{L}$, whose ordering specifies the relationship between different security levels $\ell \in \mathcal{L}$. The ordering encodes constraints on how data at a given security level can be used. To reason about both confidentiality and integrity, $\mathcal{L}$ is a product $\mathcal{L}_C \times \mathcal{L}_I$ of confidentiality and integrity lattices. For $x, y \in \mathcal{L}_C$, $x \sqsubseteq_C y$ indicates that data at level x is no more confidential than data at level y. Similarly, $x \sqsubseteq_I y$ for some $x, y \in \mathcal{L}_I$ says that data at level x is not less trustworthy than data at level y. Thus, the elements of $\ell \in \mathcal{L}$ are pairs $\ell = (\ell_C, \ell_I)$, and hereafter we denote with $C(\cdot)$ and $I(\cdot)$ the confidentiality and the integrity part, respectively. The ordering $\sqsubseteq$ of $\mathcal{L}$ is built by using $\sqsubseteq_I$ and the dual of $\sqsubseteq_C$, i.e., for all $\ell, \ell' \in \mathcal{L}$, $\ell \sqsubseteq \ell'$ *iff* $C(\ell) \sqsubseteq_C C(\ell')$ and $I(\ell) \sqsubseteq_I I(\ell')$. The underlying idea for preventing information leaks is to constrain more the usage of high-confidentiality data than low-confidentiality data. Conversely, using low-integrity data is to be constrained more than high-integrity data to prevent information corruption.

Consider a simple imperative language consisting of expressions and commands. Let $val \in Val = \{\texttt{false}, \texttt{true}, 0, 1, \dots\}$ denote a value; let v range over variable names $Name$; let op represent arithmetic and boolean operators; and let $\ell \in \mathcal{L}$ range over the security levels of a lattice $\mathcal{L}$. The syntax of the language is defined by the following grammar:

$$
\begin{aligned}
e \quad &::= val \mid v \mid e \;\texttt{op}\; e' \mid \texttt{declassify}(e, \ell) \\
c \quad &::= \texttt{skip} \mid v := e \mid c_1\texttt{;}\; c_2 \mid \texttt{if}\; e \;\texttt{then}\; c_1 \;\texttt{else}\; c_2 \mid \texttt{while}\; e \;\texttt{do}\; c
\end{aligned}
$$

The language constructs are standard and so is its semantics, except for the expression $\texttt{declassify}(\texttt{e}, \ell)$ that declassifies the security level of the expression e to the level $\ell \in \mathcal{L}$. The result of evaluating $\texttt{declassify}(\texttt{e}, \ell)$ is the same as that of e, but it allows controlling the security level of the computed value. Intuitively, the resulting security level is the join of ℓ and of the security levels associated with the free variables of e.

A security policy is specified by using a typing environment (also called *security environment*) $E\colon Name \to \mathcal{L}$ that assigns each program variable with a security level. Intuitively, the policy permits an information flow from variable x to variable y only if $E(x) \sqsubseteq E(y)$. Hereafter, we assume as given the security environment.

Assume that an attacker can read and write some data manipulated by the program. The power of an attacker to observe and modify a state of the system can be described by an element of the security lattice $\ell_A = (C_A, I_A)$. A passive attacker may read data with a security level of at most $C(\ell_A) = C_A$; whereas an active attacker can also manipulate data with a security level of at least $I(\ell_A) = I_A$ (recall the definition of $\sqsubseteq_I$ above). In [159] a low-integrity piece of code is represented by a hole $\bullet$ that occurs in a program c in the point $c[\bullet]$. An attacker of level ℓ_A can inject in those points a possibly malicious code fragment a, but it cannot insert any declassify. Actually, defining $\bot_C$ as the least element of $\mathscr{L}_C$, makes the code fragment a type check when the program counter is $(\bot_C, I(\ell_A))$. A passive attacker fills all the holes with the low-integrity code of the original program; whereas an active attacker fills the holes in a way that changes the original program behavior. Intuitively, a program satisfies robust declassification when for all program fragments a, a', the attacker's observations about the execution of $c[a']$ does not reveal any secrets apart from what the attacker already knows from observations about $c[a]$.

The type system $\mathfrak{R}$ is defined through two typing relations, one for expressions and one for commands. Hereafter, we denote with pc the value of the program counter, and we assume that the special element OK does not belong to $\mathcal{L}$. For commands, the typing relation is:

$$(E, pc) \vdash_{\mathfrak{R}} c \colon \text{OK}$$

with the intuitive meaning that under E and the program counter pc, the command c enjoys robust declassification.

The program counter pc is immaterial for expressions, and to make the typing rules of commands and expression homogeneous, we introduce the distinguished element _ in $\mathcal{L}$ to represent a *don't care* program counter. With this notation, the typing relation for expressions is

$$(E, _) \vdash_{\mathfrak{R}} e \colon \ell$$

meaning that under the environment E and any program counter, the expression e has the security level $\ell \in \mathcal{L}$.

The typing rules are displayed in Figure 18, and they are quite standard. The only exception is the rule $\mathfrak{R}$-DECLASSIFY that only allows high-integrity data to be declassified when the declassification occurs in a high-integrity program point. Actually, the type system in Figure 18 differs from the original one in [159] because the sub-typing relation has been incorporated in the rules (thus, making the type system completely syntax-directed).

$$(\mathfrak{R}\text{-VAL})\ \frac{}{(E, _) \vdash_{\mathfrak{R}} val \colon \ell} \qquad (\mathfrak{R}\text{-VAR})\ \frac{E(v) = \ell}{(E, _) \vdash_{\mathfrak{R}} v \colon \ell}$$

$$(\mathfrak{R}\text{-OP})\ \frac{\boxed{(E, _)} \vdash_{\mathfrak{R}} e \colon \ell \quad \boxed{(E, _)} \vdash_{\mathfrak{R}} e' \colon \ell' \quad \boxed{\mathsf{op} \colon \ell_{\mathsf{op}} \times \ell_{\mathsf{op}} \to \ell_{\mathsf{op}} \wedge \ell \sqsubseteq \ell_{\mathsf{op}} \wedge \ell' \sqsubseteq \ell_{\mathsf{op}}}}{(E, _) \vdash_{\mathfrak{R}} e \ \mathsf{op}\ e' \colon \ell_{\mathsf{op}}} \qquad (\mathfrak{R}\text{-SKIP})\ \frac{}{(E, pc) \vdash_{\mathfrak{R}} \mathtt{skip} \colon \text{OK}}$$

$$(\mathfrak{R}\text{-ASSIGN})\ \frac{\boxed{(E, _)} \vdash_{\mathfrak{R}} e \colon \ell \quad \boxed{\ell \sqcup pc \sqsubseteq E(v)}}{(E, pc) \vdash_{\mathfrak{R}} v := e \colon \text{OK}} \qquad (\mathfrak{R}\text{-SEQ})\ \frac{\boxed{(E, pc)} \vdash_{\mathfrak{R}} c_1 \colon \text{OK} \quad \boxed{(E, pc)} \vdash_{\mathfrak{R}} c_2 \colon \text{OK}}{(E, pc) \vdash_{\mathfrak{R}} c_1 ; \ c_2 \colon \text{OK}}$$

$$(\mathfrak{R}\text{-IF})\ \frac{\boxed{(E, _)} \vdash_{\mathfrak{R}} e \colon \ell \quad \boxed{(E, \ell \sqcup pc)} \vdash_{\mathfrak{R}} c_1 \colon \text{OK} \quad \boxed{(E, \ell \sqcup pc)} \vdash_{\mathfrak{R}} c_2 \colon \text{OK}}{(E, pc) \vdash_{\mathfrak{R}} \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \colon \text{OK}} \qquad (\mathfrak{R}\text{-WHILE})\ \frac{\boxed{(E, _)} \vdash_{\mathfrak{R}} e \colon \ell \quad \boxed{(E, \ell \sqcup pc)} \vdash_{\mathfrak{R}} c \colon \text{OK}}{(E, pc), C \vdash_{\mathfrak{R}} \mathtt{while}\ e\ \mathtt{do}\ c \colon \text{OK}}$$

$$(\mathfrak{R}\text{-DECLASSIFY})\ \frac{\boxed{(E, _)} \vdash_{\mathfrak{R}} e \colon \ell' \quad \boxed{\ell \sqcup pc \sqsubseteq E(v) \wedge I(\ell) = I(\ell') \wedge pc, \ell' \in \{\ell \mid I_A \not\sqsubseteq_I I(\ell)\}}}{(E, pc), C \vdash_{\mathfrak{R}} v := \mathtt{declassify}(e, \ell) \colon \text{OK}} \qquad (\mathfrak{R}\text{-INJECT})\ \frac{\boxed{C(pc) \sqsubseteq_C C_A}}{(E, pc) \vdash_{\mathfrak{R}} \bullet \colon \text{OK}}$$

Figure 18: The original type system $\mathfrak{R}$ of [159].

4.3.4.2 *Making $\mathfrak{R}$ incremental*

Here we instantiate the four steps of our schema and obtain the incrementalized algorithm $\mathcal{I}\mathfrak{R}$.

DEFINING THE SHAPE OF CACHES Each entry of a cache is a triple, made of a term, a pair $(E, pc) \in \mathit{TypeCtx} = \mathit{TypeEnv} \times \mathcal{L}$, and a result $R \in (\mathcal{L} \cup \{\text{OK}\})$:

$$\mathit{Cache} \triangleq \wp\big((\mathit{Expr} \cup \mathit{Cmd}) \times \mathit{TypeCtx} \times (\mathcal{L} \cup \{\text{OK}\})\big).$$

BUILDING CACHES The second step instantiates the $buildCache_{\mathfrak{R}}$ template to fit the language in hand. The straightforward instantiation is in Figure 19, where for brevity we have used the results of $\boxed{tr}$ rather than the relevant invocations.

$$
\begin{aligned}
&buildCache_{\mathfrak{R}}\,((val : \ell), (E, _)) \triangleq \{(val, (\emptyset, _), \ell)\} \qquad val \in \mathbb{N} \cup \{true, false\}\\
&buildCache_{\mathfrak{R}}\,((v : \ell), (E, _)) \triangleq \{(v, ([v \mapsto \ell], _), \ell)\}\\
&buildCache_{\mathfrak{R}}\,((e \texttt{ op } e' : \ell_{op}), (E, _)) \triangleq \{(e \texttt{ op } e', (E_{|\,FV(ee'\ op\ )}, _), \ell_{op})\}\\
&\qquad \cup (buildCache_{\mathfrak{R}}\,((e : \ell), \boxed{(E, _)})) \cup (buildCache_{\mathfrak{R}}\,((e' : \ell'), \boxed{(E, _)}))\\
&buildCache_{\mathfrak{R}}\,((\texttt{declassify}(e, \ell') : \ell), (E, pc)) \triangleq\\
&\qquad \{(\texttt{declassify}(e, \ell'), (E_{|\,FV(\texttt{declassify}(e,\ell'))}, pc), \ell)\}\\
&\qquad \cup (buildCache_{\mathfrak{R}}\,((e : \ell_e), \boxed{(E, _)}))\\
&buildCache_{\mathfrak{R}}\,((\texttt{skip} : \mathrm{OK}), (E, pc)) \triangleq \{(\texttt{skip}, (\emptyset, pc), \mathrm{OK})\}\\
&buildCache_{\mathfrak{R}}\,((v := e : \mathrm{OK}), (E, pc)) \triangleq \{(v := e, (E_{|\,FV(v:=e)}, pc), \mathrm{OK})\}\\
&\qquad \cup (buildCache_{\mathfrak{R}}\,((v : \ell_v), \boxed{(E, _)})) \cup (buildCache_{\mathfrak{R}}\,((e : \ell_e), \boxed{(E, _)}))\\
&buildCache_{\mathfrak{R}}\,((\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 : \mathrm{OK}), (E, pc)) \triangleq\\
&\qquad \{(\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, (E_{|\,FV(\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2)}, pc), \mathrm{OK})\}\\
&\qquad \cup (buildCache_{\mathfrak{R}}\,((e : \ell_e), \boxed{(E, _)})) \cup (buildCache_{\mathfrak{R}}\,((c_1 : \mathrm{OK}), \boxed{(E, pc)}))\\
&\qquad \cup (buildCache_{\mathfrak{R}}\,((c_2 : \mathrm{OK}), \boxed{(E, pc)}))\\
&buildCache_{\mathfrak{R}}\,((\texttt{while } e \texttt{ do } c : \mathrm{OK}), (E, pc)) \triangleq \{(\texttt{while } e \texttt{ do } c, (E_{|\,FV(\texttt{while } e \texttt{ do } c)}, pc), \mathrm{OK})\}\\
&\qquad \cup (buildCache_{\mathfrak{R}}\,((e : \ell_e), \boxed{(E, _)})) \cup (buildCache_{\mathfrak{R}}\,((c : \ell_c), \boxed{(E, pc)}))\\
&buildCache_{\mathfrak{R}}\,((c_1\texttt{; } c_2 : \mathrm{OK}), (E, pc)) \triangleq \{(c_1\texttt{; } c_2, (E_{|\,FV(c_1\texttt{; } c_2)}, pc), \mathrm{OK})\}\\
&\qquad \cup (buildCache_{\mathfrak{R}}\,((c_1 : \mathrm{OK}), \boxed{(E, pc)})) \cup (buildCache_{\mathfrak{R}}\,((c_2 : \mathrm{OK}), \boxed{(E, pc)}))\\
&buildCache_{\mathfrak{R}}\,((\bullet : \mathrm{OK}), (E, pc)) \triangleq \{(\bullet, (\emptyset, pc), \mathrm{OK})\}
\end{aligned}
$$

Figure 19: Instantiation of $buildCache_{\mathfrak{R}}$ for $\mathfrak{R}$ [159].

INCREMENTAL TYPING The third step of the process consists in instantiating the rule templates of Definition 4.7, which yields the incrementalized type algorithm $\mathcal{I}\mathfrak{R}$ in Figures 20 and 21, the judgments of which have the form $(E, pc), C \vdash_{\mathcal{I}\mathfrak{R}} t\colon R \triangleright C'$. As an example of how Definition 4.7 is instantiated, consider the rule template (TEMPLATE-MISS-SUB) and the rule $\mathcal{I}\mathfrak{R}$-ASSIGNMISS that derives from the rule $\mathfrak{R}$-ASSIGN in Figure 18 and is applied when the cache has no entry for $v := e$. We obtain the incrementalized rule by noting that (1) the command has sub-terms v and e ($\mathbb{I}_t = \{v, e\}$), that only the result of the type checking of e is required according to rule $\mathfrak{R}$-ASSIGN, and therefore $\boxed{tr^{e}_{v:=e}((E, pc), \emptyset) = (E, _)}$, that (2) the compatibility between the cached environments and those necessary for the actual typing is rendered by $\boxed{checkJoin_{v:=e}((E, pc), \{v, e\}, \texttt{out}_) = (\ell \sqcup pc \sqsubseteq E(v))}$, and finally that (3) the new cache C' contains all the triples in C, the new ones in C'' required while typing e, and the triple for the assignment itself. The predicate $compat_{\mathfrak{R}}$ uses equality as the relation for deciding when two types are compatible:

Definition 4.13. *Given a term t, two security environments E, E' and two security levels pc, pc', let*

$$
\begin{aligned}
&compat_{\mathfrak{R}}((E, pc), (E', pc'), t) \triangleq\\
&\quad dom(E) \cap dom(E') \supseteq FV(t) \ \wedge\ E_{|\,FV(t)} = E'_{|\,FV(t)} \ \wedge\ pc' \sqsubseteq pc
\end{aligned}
$$

assuming that $_ \sqsubseteq _$.

$$\frac{C(t) = \langle (E', c'), R\rangle \qquad compat_{\mathfrak{R}}((E,c),(E',c'),t)}{(E,c), C \vdash_{\mathcal{IR}} t\colon R \triangleright C} \;(\mathcal{IR}\text{-Hit})$$

$$\frac{(E,_) \vdash_{\mathfrak{R}} val\colon \ell \qquad C' = C \cup \{(val, (\emptyset, _), \ell)\}}{(E,_), C \vdash_{\mathcal{IR}} val\colon \ell \triangleright C'}\; miss(C, val, (E,_)) \;(\mathcal{IR}\text{-ValMiss})$$

$$\frac{(E,_) \vdash_{\mathfrak{R}} v\colon \ell \qquad C' = C \cup \{(v, (\{v \mapsto \ell\}, _), \ell)\}}{(E,_), C \vdash_{\mathcal{IR}} v\colon \ell \triangleright C'}\; miss(C, v, (E,_)) \;(\mathcal{IR}\text{-VarMiss})$$

$$\frac{\begin{array}{c} \boxed{(E,_)}, C \vdash_{\mathcal{IR}} e\colon \ell \triangleright C'' \\ \boxed{(E,_)}, C \vdash_{\mathcal{IR}} e'\colon \ell' \triangleright C''' \qquad \boxed{\mathtt{op}\colon \ell_{\mathtt{op}} \times \ell_{\mathtt{op}} \to \ell_{\mathtt{op}} \wedge \ell \sqsubseteq \ell_{\mathtt{op}} \wedge \ell' \sqsubseteq \ell_{\mathtt{op}}} \\ C' = C \cup C'' \cup C''' \cup \{(e \;\mathtt{op}\; e', (E_{|\,FV(ee'\;op)}, _), \ell_{\mathtt{op}})\} \end{array}}{(E,_), C \vdash_{\mathcal{IR}} e \;\mathtt{op}\; e'\colon \ell_{\mathtt{op}} \triangleright C'}\; miss(C, e \;\mathtt{op}\; e', (E,_)) \;(\mathcal{IR}\text{-OpMiss})$$

$$\frac{(E,pc) \vdash_{\mathfrak{R}} \mathtt{skip}\colon \mathrm{OK} \qquad C' = C \cup \{(\mathtt{skip}, (\emptyset, pc), \mathrm{OK})\}}{(E,pc), C \vdash_{\mathcal{IR}} \mathtt{skip}\colon \mathrm{OK} \triangleright C'}\; miss(C, \mathtt{skip}, (E,pc)) \;(\mathcal{IR}\text{-SkipMiss})$$

$$\frac{\begin{array}{c} \boxed{(E,_)}, C \vdash_{\mathcal{IR}} e\colon \ell \triangleright C'' \\ \boxed{\ell \sqcup pc \sqsubseteq E(v)} \qquad C' = C \cup C'' \cup \{(v := e, (E_{|\,v:=e}, pc), \mathrm{OK})\} \end{array}}{(E,pc), C \vdash_{\mathcal{IR}} v := e\colon \mathrm{OK} \triangleright C'}\; miss(C, v := e, (E,pc)) \;(\mathcal{IR}\text{-AssignMiss})$$

Figure 20: The set of rules for using $\mathfrak{R}$ incrementally, where $compat_{\mathfrak{R}}$ is as in Definition 4.13 (part I).

PROVING TYPE COHERENCE The lemma below easily follows from the above definition. Consequently, we can instantiate Theorems 4.2 to 4.4 (Corollary 4.4) to guarantee that the results of the original type system $\mathfrak{R}$ and of its incremental version $\mathcal{IR}$ coincide:

Theorem 4.3.3. *$compat_{\mathfrak{R}}$ expresses compatibility w.r.t. $=$, and $\mathfrak{R}$ preserves $=$.*

Corollary 4.4. *Let C be a well-formed cache, The following properties hold for $\mathcal{IR}$:*

- ***Coherence:*** *For all t, (E,pc), R R', and C'*

$$(E,pc) \vdash_{\mathfrak{R}} t : \tau \;\wedge\; (E,pc), C \vdash_{\mathcal{IR}} t : \tau' \triangleright C' \Rightarrow \tau = \tau';$$

- ***Completeness:*** *For all t, (E,pc), and R it holds that*

$$(E,pc) \vdash_{\mathfrak{R}} r : \tau \Rightarrow \exists R', C'.\; (E,pc), C \vdash_{\mathcal{IR}} t : R' \triangleright C';$$

- ***Well-formedness preservation:*** *For all t, (E,pc), R, and C' it holds that*

$$(E,pc), C \vdash_{\mathcal{IR}} t : R \triangleright C' \Rightarrow \Vdash_{\mathfrak{R}} C'.$$

(𝔍ℜ-SeqMiss)

$$\frac{\boxed{(E,pc)}, C \vdash_{\mathfrak{JR}} c_2 : \mathrm{OK} \triangleright C''' \qquad \begin{array}{c}\boxed{(E,pc)}, C \vdash_{\mathfrak{JR}} c_1 : \mathrm{OK} \triangleright C'' \\ C' = C \cup C'' \cup C''' \cup \{(c_1\texttt{;}\ c_2, (E|_{c_1\texttt{;}\ c_2}, pc), \mathrm{OK})\}\end{array}}{(E,pc), C \vdash_{\mathfrak{JR}} c_1\texttt{;}\ c_2 : \mathrm{OK} \triangleright C'} \; \mathit{miss}(C, c_1\texttt{;}\ c_2, (E,pc))$$

(𝔍ℜ-IfMiss) $\quad \mathit{miss}(C, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, (E,pc))$

$$\frac{\begin{array}{c}\boxed{(E,_)}, C \vdash_{\mathfrak{JR}} e : \ell \triangleright C'' \quad \boxed{(E,\ell \sqcup pc)}, C \vdash_{\mathfrak{JR}} c_1 : \mathrm{OK} \triangleright C''' \quad \boxed{(E,\ell \sqcup pc)}, C \vdash_{\mathfrak{JR}} c_2 : \mathrm{OK} \triangleright C^{iv} \\ C' = C \cup C'' \cup C''' \cup C^{iv} \cup \{(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, (E|_{\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2}, pc), \mathrm{OK})\}\end{array}}{(E,pc), C \vdash_{\mathfrak{JR}} \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \mathrm{OK} \triangleright C'}$$

(𝔍ℜ-WhileMiss)

$$\frac{\begin{array}{c}\boxed{(E,_)}, C \vdash_{\mathfrak{JR}} e : \ell \triangleright C'' \quad \boxed{(E,\ell \sqcup pc)}, C \vdash_{\mathfrak{JR}} c : \mathrm{OK} \triangleright C''' \\ C' = C \cup C'' \cup C''' \cup \{(\texttt{while}\ e\ \texttt{do}\ c, (E|_{\texttt{while}\ e\ \texttt{do}\ c}, pc), \mathrm{OK})\}\end{array}}{(E,pc), C \vdash_{\mathfrak{JR}} \texttt{while}\ e\ \texttt{do}\ c : \mathrm{OK} \triangleright C'} \; \mathit{miss}(C, \texttt{while}\ e\ \texttt{do}\ c, (E,pc))$$

(𝔍ℜ-DeclassifyMiss) $\quad \mathit{miss}(C, v := \texttt{declassify}(e,\ell), (E,pc))$

$$\frac{\begin{array}{c}\boxed{(E,_)}, C \vdash_{\mathfrak{JR}} e : \ell' \triangleright C'' \quad \boxed{\ell \sqcup pc \sqsubseteq E(v) \wedge I(\ell) = I(\ell') \wedge pc, \ell' \in \{\ell \mid I_A \not\sqsubseteq_I I(\ell)\}} \\ C' = C \cup C'' \cup \{(v := \texttt{declassify}(e,\ell), (E|_{v:=\texttt{declassify}(e,\ell)}, pc), \mathrm{OK})\}\end{array}}{(E,pc), C \vdash_{\mathfrak{JR}} v := \texttt{declassify}(e,\ell) : \mathrm{OK} \triangleright C'}$$

(𝔍ℜ-InjectMiss)

$$\frac{(E,pc) \vdash_{\mathfrak{R}} \bullet : \mathrm{OK} \qquad C' = C \cup \{(\bullet, (\emptyset, pc), \mathrm{OK})\}}{(E,pc), C \vdash_{\mathfrak{JR}} \bullet : \mathrm{OK} \triangleright C'} \; \mathit{miss}(C, \bullet, (E,pc))$$

Figure 21: The set of rules for using $\mathfrak{R}$ incrementally, where $\mathit{compat}_{\mathfrak{R}}$ is as in Definition 4.13 (part II).

4.3.5 *Type inference of exceptions*

We turn our attention to the functional language with integers, pattern matching and exceptions of Leroy and Pessaux [140] that has the following syntax:

$$\begin{aligned} \mathit{Term} \ni a &::= x \mid i \mid \lambda\, x.a \mid a_1(a_2) \mid \mathbf{match}\ a_1\ \mathbf{with}\ p \to a_2 \mid x \to a_3 \mid \\ &\qquad \mathbf{let}\ x = a_1\ \mathbf{in}\ a_2 \mid c \mid d(a) \mid \mathbf{try}\ a_1\ \mathbf{with}\ x \to a_2 \\ \mathit{Pattern} \ni p &::= x \mid i \mid c \mid d(p) \end{aligned}$$

The construct **match** a_1 **with** $p \to a_2 \mid x \to a_3$ implements pattern matching on a_1: if its value matches the pattern p then a_2 is evaluated, otherwise the term evaluates to a_3. Exceptions are rendered by the construct **try** a_1 **with** $x \to a_2$, with the intuitive meaning that if the execution of a_1 raises an exception, then its value is bound to x and a_2 is evaluated. Also, there is no syntactic construct for raising an exception in the original paper [140] and a predefined **raise** function is assumed. The evaluation is defined as expected by a reduction semantics, and we omit it. The types are designed to statically detect uncaught exceptions, and have the following form (α, ρ, δ are variables for types, rows and presence annotations, respectively):

$$\begin{aligned} \mathit{Type} \ni \tau &::= \alpha \mid \mathit{int}[\varphi] \mid \mathit{exn}[\varphi] \mid \tau_1 \xrightarrow{\varphi} \tau_2 \\ \mathit{TypeScheme} \ni \sigma &::= \forall \alpha_i, \rho_j, \delta_k.\tau \\ \mathit{Row} \ni \varphi &::= \rho \mid \top \mid \varepsilon; \varphi \\ \mathit{RowElement} \ni \varepsilon &::= i : \pi \mid c : \pi \mid d(\tau) \\ \mathit{PresenceAnnotation} \ni \pi &::= \mathit{Pre} \mid \delta \end{aligned}$$

The integer type $int[\varphi]$ is annotated by the set of integers φ; similarly the type of exceptions $exn[\varphi]$ is annotated by a set of exceptions; the functional type carries a latent effect φ, that denotes the set of exceptions that may be raised at runtime; sets of exceptions or of integers are represented as rows: a row may be $\top$ (all the values of the type are possible, e.g., $int[\top]$ denotes all the integers), or a row element followed by another row. A row element may be an integer constant i, a constant exception constructor c, or a parametrized exception construct $d(\tau)$; moreover, constant row elements also carry a presence annotation π. If π equals to Pre, then the element is present in the relevant row; otherwise it is not, but it may be considered as such to satisfy unification constraints (δ). Finally, type schemes are used to encode empty rows and absence of row elements, e.g., $\forall \rho. int \xrightarrow{\rho} int$ denotes a function with no effect and $\forall \delta. int[0 : \delta; \varphi]$ denotes a non-zero integer.
The original type system is equipped with kinds:

$$Kind \ni \kappa ::= \mathtt{INT}(\{i_1, \dots, i_n\}) \mid \mathtt{EXN}(\{c_1, \dots, c_p, d_1 \dots, d_q\})$$

allowing to enforce some invariants over rows and to check for the well-formedness of the types (written $\vdash \varphi :: \kappa$ and $\vdash \tau \; wf$), respectively. We omit the rules for kinding.

The judgments for terms of the type inference algorithm $\mathcal{J}$ are

$$(E, V) \vdash_{\mathcal{J}} a \colon (\tau, \varphi, \theta, V')$$

where $(E, V) \in TypeCtx$, E is a typing environment and V is a set of non-fresh variables; $\theta \in Subst$ is a substitution mapping type variables into types; V' includes V and the newly introduced variables.

Instead, the judgments for patterns of the type inference algorithm $\mathcal{J}$ are

$$(\tau, V) \vdash_{\mathcal{J}} p \colon (E, \tau', \theta, V')$$

with $(\tau, V) \in TypeCtx$, E is a typing environment associating free variables of p to their type; τ' is the type of terms that are not matched by p (*pattern subtraction* in [140]); and θ, V and V' are as above. As usual, $\theta\,\tau$ is the application of the substitution θ to τ, and $\theta_2 \circ \theta_1$ is the composition of substitutions.

From now onward we assume $\mathtt{mgu}_V$ to be the unification function between types; $\mathtt{Gen}(\tau, E, \varphi) = \forall \alpha_i, \rho_j, \delta_k.\tau$ where α_i, ρ_j, and δ_k appear free in τ, but not in E and φ; $\mathtt{Inst}$ to be a function that inputs a type scheme $\forall \alpha_i, \rho_j, \delta_k.\tau$ and a set of non-fresh variables V, and returns τ in which α_i, ρ_j, and δ_k have been substituted with fresh (i.e., not in V) α'_i, ρ'_j, and δ'_k and such that ρ_j and ρ'_j have the same kind for all j; finally, $TypeArg(d)$ to be a function returning the type scheme of the argument of the exception constructor d.

The rules in Figures 22 and 23 are the logical presentation of the original inference algorithm.[3] We now briefly comment on the rules for terms, omitting the details about unification. The rules for variables, and **let-in** are standard. Rule ($\mathcal{J}$-ABS) deals with function abstraction, and is analogous to that of other effect systems. Rules ($\mathcal{J}$-CONST), ($\mathcal{J}$-cCONSTR) are for integers and constant exception constructors, respectively. Intuitively, they return a type that records the presence of the actual value of the expression. Rule ($\mathcal{J}$-dCONSTR) Pattern matching **match** a_1 **with** $p \to a_2 \mid x \to a_3$ is dealt with in rule ($\mathcal{J}$-MATCH). The results for the term a_1 and the pattern p are computed as expected. Using that information, the type and the set of exceptions of a_2 and a_3 are computed. To ensure the precision of the type inference, the results for a_3 are computed under the assumption that x has

3 We assume that the fresh variables are generated algorithmically.

a type which is not matched by p (as given by pattern inference). Finally, inference for **try** a_1 **with** $x \to a_2$ is done using the rule ($\mathcal{J}$-Try). Roughly, the type and the set of raised exceptions φ_1 of a_1 is inferred first, then — under the assumption that x assumes value in φ_1 — the rule computes the overall type and set of exceptions.

($\mathcal{J}$-Const)

$$\frac{V' = V \cup \{\rho, \rho'\} \wedge \vdash \rho :: \mathtt{INT}(\{i\}) \wedge \vdash \rho' :: \mathtt{EXN}(\emptyset)}{(E, V) \vdash_{\mathcal{J}} i : (int[i : \mathtt{Pre}; \rho], \rho', id, V')} \quad \rho, \rho' \notin V$$

($\mathcal{J}$-Var)

$$\frac{(\tau, V') = \mathtt{Inst}(E(x), V \cup \{\rho\}) \wedge \vdash \rho :: \mathtt{EXN}(\emptyset)}{(E, V) \vdash_{\mathcal{J}} x : (\tau, \rho, id, V')} \quad \rho \notin V$$

($\mathcal{J}$-Abs)

$$\frac{E[x \mapsto \alpha], V \cup \{\alpha\} \vdash_{\mathcal{J}} a : (\tau_1, \varphi_1, \theta_1, V_1) \qquad V' = V_1 \cup \{\rho\} \wedge \vdash \rho :: \mathtt{EXN}(\emptyset) \wedge \tau = \theta_1 \alpha \xrightarrow{\varphi_1} \tau_1}{(E, V) \vdash_{\mathcal{J}} \lambda\, x.a : (\tau, \rho, \theta_1, V')} \quad \alpha \notin V, \rho \notin V_1$$

($\mathcal{J}$-App)

$$\frac{\begin{array}{c}(E, V) \vdash_{\mathcal{J}} a_1 : (\tau_1, \varphi_1, \theta_1, V_1) \qquad (\theta_1\, E, V_1) \vdash_{\mathcal{J}} a_2 : (\tau_2, \varphi_2, \theta_2, V_2) \\ (\mu, V_3) = \mathtt{mgu}_{V_2 \cup \{\alpha\}}\{\theta_2 \tau_1 = \theta_2 \xrightarrow{\varphi_2} \alpha, \theta_2 \varphi_1 = \varphi_2\} \wedge \tau = \mu\alpha \wedge \varphi = \mu\varphi_2 \wedge \theta = \mu \circ \theta_2 \circ \theta_1 \wedge V' = V_3\end{array}}{(E, V) \vdash_{\mathcal{J}} a_1\, (a_2) : (\tau, \varphi, \theta, V')} \quad \alpha \notin V_2$$

($\mathcal{J}$-Match)

$$\frac{\begin{array}{c}(E, V) \vdash_{\mathcal{J}} a_1 : (\tau_1, \varphi_1, \theta_1, V_1) \quad (\tau_1, V_1) \vdash_{\mathcal{J}} p : (E', \tau', \psi, V_1') \quad ((\theta_1 \circ \psi\, E)[E'], V_1') \vdash_{\mathcal{J}} a_2 : (\tau_2, \varphi_2, \theta_2, V_2) \\ ((\theta_1 \circ \psi \circ \theta_2\, E)[x \mapsto \theta_2 \tau'], V_2) \vdash_{\mathcal{J}} a_3 : (\tau_3, \varphi_3, \theta_3, V_3) \\ (\mu, V_4) = \mathtt{mgu}_{V_3}\{\theta_3 \tau_2 = \tau_3, \theta_3 \varphi_2 = \varphi_3, (\theta_3 \circ \theta_2 \circ \psi)\varphi_1 = \varphi_3\} \wedge \\ \tau = \mu\tau_3 \wedge \varphi = \mu\varphi_3 \wedge \theta = \mu \circ \theta_3 \circ \theta_2 \circ \psi \circ \theta_1 \wedge V' = V_4\end{array}}{(E, V) \vdash_{\mathcal{J}} \textbf{match}\; a_1\; \textbf{with}\; p \to a_2 \mid x \to a_3 : (\tau, \varphi, \theta, V')}$$

($\mathcal{J}$-Let)

$$\frac{\begin{array}{c}(E, V) \vdash_{\mathcal{J}} a_1 : (\tau_1, \varphi_1, \theta_1, V_1) \quad (\theta_1\, E[x \mapsto \mathtt{Gen}(\tau_1, \theta_1 E, \varphi_1)], V_1) \vdash_{\mathcal{J}} a_2 : (\tau_2, \varphi_2, \theta_2, V_2) \\ (\mu, V_3) = \mathtt{mgu}_{V_2}\{\theta_2 \varphi_1 = \varphi_2\} \wedge \tau = \mu\tau_2 \wedge \varphi = \mu\varphi_2 \wedge \theta = \mu \circ \theta_2 \circ \theta_1 \wedge V' = V_3\end{array}}{(E, V) \vdash_{\mathcal{J}} \textbf{let}\; x = a_1\; \textbf{in}\; a_2 : (\tau, \varphi, \theta, V')}$$

($\mathcal{J}$-cConstr)

$$\frac{\vdash \rho :: \mathtt{EXN}(\{c\}) \wedge \vdash \rho' :: \mathtt{EXN}(\emptyset) \wedge \tau = exn[c : \mathtt{Pre}; \rho] \wedge \varphi = \rho' \wedge \theta = id \wedge V' = V \cup \{\rho, \rho'\}}{(E, V) \vdash_{\mathcal{J}} c : (\tau, \varphi, \theta, V')} \quad \rho, \rho' \notin V$$

($\mathcal{J}$-dConstr)

$$\frac{\begin{array}{c}(E, V) \vdash_{\mathcal{J}} a_1 : (\tau_1, \varphi_1, \theta_1, V_1) \\ (\tau_2, V_2) = \mathtt{Inst}(TypeArg(d), V_1) \wedge (\mu, V_3) = \mathtt{mgu}_{V_2}\{\tau_2 = \tau_1\} \wedge \vdash \rho :: \mathtt{EXN}(\{C\}) \wedge \\ \vdash \rho' :: \mathtt{EXN}(\emptyset) \wedge \tau = exn[d(\mu\tau_1); \rho] \wedge \varphi = \rho' \wedge \theta = \mu \circ \theta_1 \wedge V' = V_3 \cup \{\rho, \rho'\}\end{array}}{(E, V) \vdash_{\mathcal{J}} d(a_1) : (\tau, \varphi, \theta, V')} \quad \rho \notin V_3, \rho' \notin V_3 \cup \{\rho\}$$

($\mathcal{J}$-Try)

$$\frac{\begin{array}{c}(E, V) \vdash_{\mathcal{J}} a_1 : (\tau_1, \varphi_1, \theta_1, V_1) \quad (\theta_1\, E[x \mapsto exn[\varphi_1]], V_1) \vdash_{\mathcal{J}} a_2 : (\tau_2, \varphi_2, \theta_2, V_2) \\ (\mu, V_3) = \mathtt{mgu}_{V_2}\{\theta_2 \varphi_1 = \varphi_2\} \wedge \tau = \mu\tau_2 \wedge \varphi = \mu\varphi_2 \wedge \theta = \mu \circ \theta_2 \circ \theta_1 \wedge V' = V_3\end{array}}{(E, V) \vdash_{\mathcal{J}} \textbf{try}\; a_1\; \textbf{with}\; x \to a_2 : (\tau, \varphi, \theta, V')}$$

Figure 22: Rules defining the inference algorithm for terms of [140].

$$(\mathcal{J}\text{-PVAR})\quad \frac{E = \{x : \tau\} \wedge \tau' = \alpha \wedge \theta = id \wedge V' = V \cup \{\alpha\}}{(\tau, V) \vdash_{\mathcal{J}} x : (E, \tau', \theta, V')}\ \alpha \notin V$$

$$(\mathcal{J}\text{-PINT})\quad \frac{\begin{array}{c}\vdash \rho :: \mathtt{INT}(\{i\}) \wedge (\mu, V_1) = \mathtt{mgu}_{V \cup \{\rho, \delta\}}\{\tau = int[i : \delta; \rho]\} \wedge \\ E = \emptyset \wedge \tau' = int[i : \delta'; \mu\rho] \wedge \theta = \mu \wedge V' = V_1 \cup \{\delta\}\end{array}}{(\tau, V) \vdash_{\mathcal{J}} i : (E, \tau', \theta, V')}\ \rho \notin V, \delta' \notin V \cup \{\rho\} \cup V_1$$

$$(\mathcal{J}\text{-PCCONSTR})\quad \frac{\begin{array}{c}\vdash \rho :: \mathtt{EXN}(\{c\}) \wedge (\mu, V_1) = \mathtt{mgu}_{V \cup \{\rho, \delta\}}\{\tau = exn[c : \delta; \rho]\} \wedge \\ E = \emptyset \wedge \tau' = exn[c : \delta'; \mu\rho] \wedge \theta = \mu \wedge V' = V_1 \cup \{\delta\}\end{array}}{(\tau, V) \vdash_{\mathcal{J}} C : (E, \tau', \theta, V')}\ \rho \notin V, \delta' \notin V \cup \{\rho\} \cup \notin V_1$$

$$(\mathcal{J}\text{-PDCONSTR})\quad \frac{\begin{array}{c}\mathtt{Inst}(\mathit{TypeArg}(d)), V \vdash_{\mathcal{J}} p_1 : (E_1, \tau_1', \theta_1, V_1) \\ \vdash \rho :: \mathtt{EXN}(\{d\}) \wedge (\mu, V_2) = \mathtt{mgu}_{V_1 \cup \{\rho\}}\{\tau = exn[d(\theta_1\tau_1); \rho]\} \wedge \\ E = \mu E_1 \wedge \tau' = exn[d(\mu\tau_1'); \mu\rho] \wedge \theta = \mu \circ \theta_1 \wedge V' = V_2\end{array}}{(\tau, V) \vdash_{\mathcal{J}} d(p_1) : (E, \tau', \theta, V')}\ \rho \notin V$$

Figure 23: Rules defining the inference algorithm for patterns of [140].

DEFINING THE SHAPE OF CACHES The entries of the cache are either $(a, (E, V), (\tau, \varphi, \theta, V'))$ or $(p, (\tau, V), (E, \tau', \theta, V'))$. Thus, a cache C is an element of

$$\begin{aligned} Cache \triangleq \wp\big(&(Term \cup Pattern) \times \\ &((TypeEnv \cup Type) \times \wp(Name)) \times \\ &(((TypeEnv \times Type) \cup (Type \times Row)) \times Subst \times \wp(Name))\big). \end{aligned}$$

BUILDING CACHES The second step instantiates the $buildCache$ template from Section 4.2. The definition has two groups of rules, one for patterns and one for terms and is displayed in Figure 24.

INCREMENTAL TYPING In Figures 25 to 27 we display all the rules defining the algorithm $\mathcal{IJ}$ for patterns and for terms.

Note that here we have unnecessarily duplicated the hit rules for the sake of clarity, because of the uniform presentation of typing for both patterns and terms. All the rules mimic the behavior of algorithm $\mathcal{J}$, following the templates of Section 4.2. The predicate $compat_{\mathcal{J}}$ consists of two parts, one for patterns and one the terms, and it uses equality for deciding when two types are compatible:

Definition 4.14. *Given a pattern p and two pairs $(\tau, V), (\tau', V')$, and a term a and two pairs $(E, V), (E', V')$, let*

$$compat^p_{\mathcal{J}}((\tau, V), (\tau', V'), p) \triangleq \big(\tau = \tau' \wedge V = V'\big)$$

and

$$\begin{aligned} compat^a_{\mathcal{J}}((E, V), (E', V'), a) \triangleq\ & \mathtt{dom}\,(E) \cap \mathtt{dom}\,(E') \supseteq FV(a) \wedge \\ & E_{|\,FV(a)} = E'_{|\,FV(a)} \wedge V = V' \end{aligned}$$

*Finally, define $compat_{\mathcal{J}} = compat^p_{\mathcal{J}}$ **xor** $compat^a_{\mathcal{J}}$.*

PROVING TYPE COHERENCE The following lemma easily follows from Definition 4.14:

Theorem 4.3.4. *$compat_{\mathcal{J}}$ expresses compatibility w.r.t. $=$, and $\mathcal{J}$ preserves $=$.*

Now, we instantiate Theorems 4.2 to 4.4 to guarantee that the results of the original type system $\mathcal{J}$ and of its incremental version $\mathcal{IJ}$ coincide. (We separate the two corollaries for the sake of readability.)

Corollary 4.5 (Patterns)**.** *Let C be a well-formed cache, the following properties hold for $\mathcal{IJ}$:*

- ***Coherence:*** *For any pattern p, any τ, V, E, τ', θ, V', E', τ'', θ', V'', and cache C':*

$$(\tau, V) \vdash_{\mathcal{J}} p : (E, \tau', \theta, V') \;\wedge\; (\tau, V), C \vdash_{\mathcal{IJ}} p : (E', \tau'', \theta', V'') \triangleright C' \\ \Rightarrow (E, \tau', \theta, V') = (E', \tau'', \theta', V'');$$

- ***Completeness:*** *For any pattern p, any τ, V, E, τ', θ, V':*

$$(\tau, V) \vdash_{\mathcal{J}} p : (E, \tau', \theta, V') \Rightarrow \\ \exists (E', \tau'', \theta', V''), C'. \; (\tau, V), C \vdash_{\mathcal{IJ}} p : (E', \tau'', \theta', V'') \triangleright C';$$

- ***Well-formedness preservation:*** *For any pattern p, any τ, V, E, τ', θ, V', and cache C':*

$$(\tau, V), C \vdash_{\mathcal{IJ}} p : (E, \tau', \theta, V') \triangleright C' \Rightarrow \Vdash_{\mathcal{J}} C'.$$

Corollary 4.6 (Terms)**.** *Let C be a well-formed cache, the following properties hold for $\mathcal{IJ}$:*

- ***Coherence:*** *For any term a, any E, V, τ, φ, θ, V', τ', φ', θ', V'', and cache C':*

$$(E, V) \vdash_{\mathcal{J}} a : (\tau, \varphi, \theta, V') \;\wedge\; (E, V), C \vdash_{\mathcal{IJ}} a : (\tau', \varphi', \theta', V'') \triangleright C' \\ \Rightarrow (\tau, \varphi, \theta, V') = (\tau', \varphi'', \theta', V'');$$

- ***Completeness:*** *For any term a, any E, V, τ, φ, θ, V', τ', φ', θ', and V'':*

$$(E, V) \vdash_{\mathcal{J}} a : (\tau, \varphi, \theta, V') \Rightarrow \\ \exists (\tau', \varphi', \theta', V''), C'. \; (\tau, V), C \vdash_{\mathcal{IJ}} a : (\tau', \varphi', \theta', V'') \triangleright C';$$

- ***Well-formedness preservation:*** *For any term a, any E, V, τ, φ, θ, V', and cache C':*

$$(E, V), C \vdash_{\mathcal{IJ}} a : (\tau, \varphi, \theta, V') \triangleright C' \Rightarrow \Vdash_{\mathcal{J}} C'.$$

$$
\begin{aligned}
&buildCache_{\mathcal{J}}\big((x : (E, \tau', \theta, V')), (\tau, V)\big) \triangleq \{(x, (\tau, V), (E, \tau', \theta, V'))\}\\
&buildCache_{\mathcal{J}}\big((i : (E, \tau', \theta, V')), (\tau, V)\big) \triangleq \{(i, (\tau, V), (E, \tau', \theta, V'))\}\\
&buildCache_{\mathcal{J}}\big((c : (E, \tau', \theta, V')), (\tau, V)\big) \triangleq \{(c, (\tau, V), (E, \tau', \theta, V'))\}\\
&buildCache_{\mathcal{J}}\big((d(p_1) : (E, \tau', \theta, V')), (\tau, V)\big) \triangleq \{(d(p_1), (\tau, V), (E, \tau', \theta, V'))\}\\
&\qquad \cup (buildCache_{\mathcal{J}}\big((p_1 : (E_1, \tau_1', \theta_1, V_1')), \boxed{(\mathtt{Inst}(\mathit{TypeArg}(d)), V)}\big))
\end{aligned}
$$

$$
\begin{aligned}
&buildCache_{\mathcal{J}}\big((x : (\tau, \varphi, id, V')), (E, V)\big) \triangleq \{(x, (E, V), (\tau, id, \theta, V'))\}\\
&buildCache_{\mathcal{J}}\big((i : (\tau, \varphi, id, V')), (E, V)\big) \triangleq \{(i, (E, V), (\tau, \varphi, id, V'))\}\\
&buildCache_{\mathcal{J}}\big((\lambda\, x.a : (\tau, \varphi, \theta, V')), (E, V)\big) \triangleq \{(\lambda\, x.a, (E_{|\,FV(\lambda\, x.a)}, V), (\tau, \varphi, \theta, V'))\}\\
&\qquad \cup\, buildCache_{\mathcal{J}}\Big((a\colon (\tau_1, \varphi_1, \theta_1, V_1')), \boxed{(E_{|\,FV(a)\setminus\{x\}}, V')}\Big)\\
&buildCache_{\mathcal{J}}\big((a_1(a_2) : (\tau, \varphi, \theta, V')), (E, V)\big) \triangleq \{(a_1(a_2), (E_{|\,FV(a_1(a_2))}, V), (\tau, \varphi, \theta, V'))\}\\
&\qquad \cup\, buildCache_{\mathcal{J}}\Big((a_1\colon (\tau_1, \varphi_1, \theta_1, V_1)), \boxed{(E_{|\,FV(a_1)}, V')}\Big)\\
&\qquad \cup\, buildCache_{\mathcal{J}}\Big((a_2\colon (\tau_2, \varphi_2, \theta_2, V_2)), \boxed{(\theta_{1|\,FV(a_2)}, V_1)}\Big)\\
&buildCache_{\mathcal{J}}\big((\textbf{match}\ a_1\ \textbf{with}\ p \to a_2 \mid x \to a_3 : (\tau, \varphi, \theta, V')), (E, V)\big) \triangleq\\
&\quad \{(\textbf{match}\ a_1\ \textbf{with}\ p \to a_2 \mid x \to a_3, (E_{|\,FV(\textbf{match}\ a_1\ \textbf{with}\ p\to a_2 \mid x\to a_3)}, V), (\tau, \varphi, \theta, V'))\}\\
&\qquad \cup\, buildCache_{\mathcal{J}}\Big((a_1\colon (\tau_1, \varphi_1, \theta_1, V_1)), \boxed{(E_{|\,FV(a_1)}, V')}\Big)\\
&\qquad \cup\, buildCache_{\mathcal{J}}\Big((p\colon (E', \tau', \psi, V_1')), \boxed{(E, V')}\Big)\\
&\qquad \cup\, buildCache_{\mathcal{J}}\Big((a_2\colon (\tau_2, \varphi_2, \theta_2, V_2)), \boxed{((\theta_1 \circ \psi\, E[E'])_{|\,FV(a_2)}, V_1')}\Big)\\
&\qquad \cup\, buildCache_{\mathcal{J}}\Big((a_3\colon (\tau_3, \varphi_3, \theta_3, V_3)), \boxed{((\theta_1 \circ \psi \circ \theta_2\, E)[x \mapsto \theta_2 \tau']_{|\,FV(a_3)}, V_2)}\Big)\\
&buildCache_{\mathcal{J}}\big((\textbf{let}\ x = a_1\ \textbf{in}\ a_2 : (\tau, \varphi, \theta, V')), (E, V)\big) \triangleq\\
&\qquad \{(\textbf{let}\ x = a_1\ \textbf{in}\ a_2, (E_{|\,FV(\textbf{let}\ x = a_1\ \textbf{in}\ a_2)}, V), (\tau, \varphi, \theta, V'))\}\\
&\qquad \cup\, buildCache_{\mathcal{J}}\Big((a_1\colon (\tau_1, \varphi_1, \theta_1, V_1)), \boxed{(E_{|\,FV(a_1)}, V')}\Big)\\
&\qquad \cup\, buildCache_{\mathcal{J}}\Big((a_2\colon (\tau_2, \varphi_2, \theta_2, V_2)), \boxed{(\theta_1 E[x \mapsto \mathtt{Gen}(\tau_1, \theta_1 E, \varphi_1)]_{|\,FV(a_2)}, V_1)}\Big)\\
&buildCache_{\mathcal{J}}\big((c : (\tau, \varphi, id, V')), (\tau, V)\big) \triangleq \{(c, (\tau, V), (\tau, id, \theta, V'))\}\\
&buildCache_{\mathcal{J}}\big((d(a_1) : (\tau, \varphi, \theta, V')), (E, V)\big) \triangleq \{(d(a_1), (E_{|\,FV(d(a_1))}, V), (\tau, \varphi, \theta, V'))\}\\
&\qquad \cup\, buildCache_{\mathcal{J}}\Big((a_1\colon (\tau_1, \varphi_1, \theta_1, V_1)), \boxed{(E_{|\,FV(a_1)}, V')}\Big)\\
&buildCache_{\mathcal{J}}\big((\textbf{try}\ a_1\ \textbf{with}\ x\ \to a_2 : (\tau, \varphi, \theta, V')), (E, V)\big) \triangleq\\
&\qquad \{(\textbf{try}\ a_1\ \textbf{with}\ x\ \to a_2, (E_{|\,FV(\textbf{try}\ a_1\ \textbf{with}\ x\ \to a_2)}, V), (\tau, \varphi, \theta, V'))\}\\
&\qquad \cup\, buildCache_{\mathcal{J}}\Big((a_1\colon (\tau_1, \varphi_1, \theta_1, V_1)), \boxed{(E_{|\,FV(a_1)}, V')}\Big)\\
&\qquad \cup\, buildCache_{\mathcal{J}}\Big((a_2\colon (\tau_2, \varphi_2, \theta_2, V_2)), \boxed{(\theta_1\, E[x \mapsto exn[\varphi_1]], V_{1|\,FV(a_2)}, V_1)}\Big)
\end{aligned}
$$

Figure 24: Definition of $buildCache_{\mathcal{J}}$ for the incremental type inference of [140] (upper part for patterns, lower for terms).

$$(\mathcal{IJ}\text{-Hit})\quad \frac{C(p) = \langle(\tau', V'), (E', \tau', \theta, V')\rangle \qquad compat_{\mathcal{J}}((\tau, V), (\tau', V'), p)}{(\tau, V), C \vdash_{\mathcal{IJ}} p\colon (E, \tau', \theta, V') \triangleright C}$$

$$(\mathcal{IJ}\text{-PVarMiss})\quad \frac{(\tau, V) \vdash_{\mathcal{J}} x : (E, \tau', \theta, V') \qquad C' = C \cup \{(x, (\tau, V), (E, \tau', \theta, V'))\}}{(\tau, V), C \vdash_{\mathcal{IJ}} x : (E, \tau', \theta, V') \triangleright C'}\ \alpha \notin V, miss(C, x, (\tau, V))$$

$$(\mathcal{IJ}\text{-PIntMiss})\quad \frac{(\tau, V) \vdash_{\mathcal{J}} i : (E, \tau', \theta, V') \qquad C' = C \cup \{(i, (\tau, V), (E, \tau', \theta, V'))\}}{(\tau, V), C \vdash_{\mathcal{IJ}} i : (E, \tau', \theta, V') \triangleright C'}\ miss(C, i, (\tau, V))$$

$$(\mathcal{IJ}\text{-PcConstrMiss})\quad \frac{(\tau, V) \vdash_{\mathcal{J}} c : (E, \tau', \theta, V') \qquad C' = C \cup \{(c, (\tau, V), (E, \tau', \theta, V'))\}}{(\tau, V), C \vdash_{\mathcal{IJ}} c : (E, \tau', \theta, V') \triangleright C'}\ miss(C, c, (\tau, V))$$

$$(\mathcal{IJ}\text{-PdConstrMiss})\quad \frac{\begin{array}{c}\boxed{(\texttt{Inst}(TypeArg(d)), V)}, C \vdash_{\mathcal{IJ}} p_1 : (E_1, \tau_1', \theta_1, V_1) \triangleright C'' \\ \vdash \rho :: \texttt{EXN}(\{d\}) \wedge (\mu, V_2) = \texttt{mgu}_{V_1 \cup \{\rho\}}\{\tau = exn[d(\theta_1 \tau_1); \rho]\} \wedge \\ E = \mu E_1 \wedge \tau' = exn[d(\mu \tau_1'); \mu\rho] \wedge \theta = \mu \circ \theta_1 \wedge V' = V_2 \\ C' = C \cup C'' \cup \{(d(p_1), (\tau, FV(d(p_1))), (E, \tau', \theta, V'))\}\end{array}}{(\tau, V), C \vdash_{\mathcal{IJ}} d(p_1) : (E, \tau', \theta, V') \triangleright C'}\ \rho \notin V, miss(C, d(p_1), (\tau, V))$$

Figure 25: The rules of the incremental inference algorithm for patterns of [140].

$$(\mathcal{IJ}\text{-Hit})\quad \frac{C(a) = \langle(\tau, \rho, \theta, V')\rangle \qquad compat_{\mathcal{J}}((E, V), (E', V'), a)}{(E, V), C \vdash_{\mathcal{IJ}} a\colon (\tau, \rho, \theta, V') \triangleright C}$$

$$(\mathcal{IJ}\text{-VarMiss})\quad \frac{(E, V) \vdash_{\mathcal{J}} x : (\tau, \rho, id, V') \qquad C' = C \cup \{(x, (E_{|\,FV(x)}, V), (\tau, \rho, id, V'))\}}{(E, V), C \vdash_{\mathcal{IJ}} x : (\tau, \rho, id, V') \triangleright C}\ \rho \notin V, miss(C, x, (E, V))$$

$$(\mathcal{IJ}\text{-ConstMiss})\quad \frac{\begin{array}{c}(E, V) \vdash_{\mathcal{J}} i : (int[i : \texttt{Pre}; \rho], \rho', id, V') \\ C' = C \cup \{(i, (E_{|\,FV(i)}, V), (int[i : \texttt{Pre}; \rho], \rho', id, V'))\}\end{array}}{(E, V), C \vdash_{\mathcal{IJ}} i : (int[i : \texttt{Pre}; \rho], \rho', id, V') \triangleright C'}\ \rho, \rho' \notin V, miss(C, i, (E, V))$$

$$(\mathcal{IJ}\text{-AbsMiss})\quad \frac{\begin{array}{c}\boxed{(E[x \mapsto \alpha], V \cup \{\alpha\})}, C \vdash_{\mathcal{IJ}} a : (\tau_1, \varphi_1, \theta_1, V_1) \triangleright C'' \\ V' = V_1 \cup \{\rho\} \wedge \vdash \rho :: \texttt{EXN}(\emptyset) \wedge \tau = \theta_1 \alpha \xrightarrow{\varphi_1} \tau_1 \\ C' = C \cup C'' \cup \{(\lambda\, x.a, (E_{|\,FV(\lambda\, x.a)}, V), (\tau, \rho, \theta_1, V'))\}\end{array}}{(E, V), C \vdash_{\mathcal{IJ}} \lambda\, x.a : (\tau, \rho, \theta_1, V') \triangleright C'}\ \alpha \notin V, \rho \notin V_1, miss(C, \lambda\, x.a, (E, V))$$

Figure 26: The rules of the incrementalized inference algorithm for terms of [140] (part I).

$$(\mathcal{J}\mathcal{J}\text{-AppMiss}) \qquad \alpha \notin V_2, \mathit{miss}(C, a_1(a_2), (E,V))$$

$$\frac{\begin{array}{c} \boxed{(E,V)}, C \vdash_{\mathcal{J}\mathcal{J}} a_1 : (\tau_1, \varphi_1, \theta_1, V_1) \triangleright C'' \quad \boxed{(\theta_1\, E, V_1)}, C \vdash_{\mathcal{J}\mathcal{J}} a_2 : (\tau_2, \varphi_2, \theta_2, V_2) \triangleright C''' \\ (\mu, V_3) = \mathtt{mgu}_{V_2 \cup \{\alpha\}}\{\theta_2\tau_1 = \theta_2 \xrightarrow{\varphi_2} \alpha, \theta_2\varphi_1 = \varphi_2\} \wedge \tau = \mu\alpha \wedge \varphi = \mu\varphi_2 \wedge \theta = \mu \circ \theta_2 \circ \theta_1 \wedge V' = V_3 \\ C' = C \cup C'' \cup C''' \cup \{(a_1\,(a_2), (E_{\mid FV(a_1\,(a_2))}, V), (\tau, \varphi, \theta, V'))\} \end{array}}{(E,V), C \vdash_{\mathcal{J}\mathcal{J}} a_1\,(a_2) : (\tau, \varphi, \theta, V') \triangleright C'}$$

$$(\mathcal{J}\mathcal{J}\text{-MatchMiss}) \qquad \mathit{miss}(C, \mathbf{match}\ a_1\ \mathbf{with}\ p \to a_2 \mid x \to a_3, (E,V))$$

$$\frac{\begin{array}{c} \boxed{(E,V)}, C \vdash_{\mathcal{J}\mathcal{J}} a_1 : (\tau_1, \varphi_1, \theta_1, V_1) \triangleright C_1 \\ \boxed{(\tau_1, V_1)}, C \vdash_{\mathcal{J}\mathcal{J}} p : (E', \tau', \psi, V_1') \triangleright C_2 \quad \boxed{((\theta_1 \circ \psi\, E)[E'], V_1')}, C \vdash_{\mathcal{J}\mathcal{J}} a_2 : (\tau_2, \varphi_2, \theta_2, V_2) \triangleright C_3 \\ \boxed{((\theta_1 \circ \psi \circ \theta_2\, E)[x \mapsto \theta_2\tau'], V_2)}, C \vdash_{\mathcal{J}\mathcal{J}} a_3 : (\tau_3, \varphi_3, \theta_3, V_3) \triangleright C_4 \\ (\mu, V_4) = \mathtt{mgu}_{V_3}\{\theta_3\tau_2 = \tau_3, \theta_3\varphi_2 = \varphi_3, (\theta_3 \circ \theta_2 \circ \psi)\varphi_1 = \varphi_3\} \wedge \\ \tau = \mu\tau_3 \wedge \varphi = \mu\varphi_3 \wedge \theta = \mu \circ \theta_3 \circ \theta_2 \circ \psi \circ \theta_1 \wedge V' = V_4 \\ C' = C \cup \bigcup_{i=1}^{4} C_i \cup \{(\mathbf{match}\ a_1\ \mathbf{with}\ p \to a_2 \mid x \to a_3, (E_{\mid FV(\mathbf{match}\ a_1\ \mathbf{with}\ p \to a_2 \mid x \to a_3)}, V), (\tau, \varphi, \theta, V'))\} \end{array}}{(E,V), C \vdash_{\mathcal{J}\mathcal{J}} \mathbf{match}\ a_1\ \mathbf{with}\ p \to a_2 \mid x \to a_3 : (\tau, \varphi, \theta, V') \triangleright C'}$$

$$(\mathcal{J}\mathcal{J}\text{-LetMiss}) \qquad \mathit{miss}(C, \mathbf{let}\ x = a_1\ \mathbf{in}\, a_2, (E,V))$$

$$\frac{\begin{array}{c} \boxed{(E,V)}, C \vdash_{\mathcal{J}\mathcal{J}} a_1 : (\tau_1, \varphi_1, \theta_1, V_1) \triangleright C'' \quad \boxed{(\theta_1\, E[x \mapsto \mathtt{Gen}(\tau_1, \theta_1 E, \varphi_1)], V_1)}, C \vdash_{\mathcal{J}\mathcal{J}} a_2 : (\tau_2, \varphi_2, \theta_2, V_2) \triangleright C''' \\ (\mu, V_3) = \mathtt{mgu}_{V_2}\{\theta_2\varphi_1 = \varphi_2\} \wedge \tau = \mu\tau_2 \wedge \varphi = \mu\varphi_2 \wedge \theta = \mu \circ \theta_2 \circ \theta_1 \wedge V' = V_3 \\ C' = C'' \cup C''' \cup \{(\mathbf{let}\ x = a_1\ \mathbf{in}\, a_2, (E_{\mid FV(\mathbf{let}\ x = a_1\ \mathbf{in}\, a_2)}, V), (\tau, \varphi, \theta, V'))\} \end{array}}{(E,V), C \vdash_{\mathcal{J}\mathcal{J}} \mathbf{let}\ x = a_1\ \mathbf{in}\, a_2 : (\tau, \varphi, \theta, V') \triangleright C'}$$

$$(\mathcal{J}\mathcal{J}\text{-cConstrMiss})$$

$$\frac{(E,V) \vdash_{\mathcal{J}} c : (\tau, \varphi, \theta, V') \qquad C' = C \cup \{(c, (E_{\mid FV(c)}, V), (\tau, \varphi, \theta, V'))\}}{(E,V), C \vdash_{\mathcal{J}\mathcal{J}} c : (\tau, \varphi, \theta, V') \triangleright C'}\ \rho, \rho' \notin V, \mathit{miss}(C, c, (E,V))$$

$$(\mathcal{J}\mathcal{J}\text{-dConstrMiss}) \qquad \rho \notin V_3, \rho' \notin V_3 \cup \{\rho\}, \mathit{miss}(C, d(a_1), E)$$

$$\frac{\begin{array}{c} \boxed{(E,V)}, C \vdash_{\mathcal{J}\mathcal{J}} a_1 : (\tau_1, \varphi_1, \theta_1, V_1) \triangleright C'' \\ (\tau_2, V_2) = \mathtt{Inst}(\mathit{TypeArg}(d), V_1) \wedge (\mu, V_3) = \mathtt{mgu}_{V_2}\{\tau_2 = \tau_1\} \wedge \vdash \rho :: \mathtt{EXN}(\{c\}) \wedge \\ \vdash \rho' :: \mathtt{EXN}(\emptyset) \wedge \tau = \mathit{exn}[d(\mu\tau_1); \rho] \wedge \varphi = \rho' \wedge \theta = \mu \circ \theta_1 \wedge V' = V_3 \cup \{\rho, \rho'\} \\ C' = C \cup C'' \cup \{(d(a_1), (E_{\mid FV(d(a_1))}, V), (\tau, \varphi, \theta, V'))\} \end{array}}{(E,V), C \vdash_{\mathcal{J}} d(a_1) : (\tau, \varphi, \theta, V') \triangleright C'}$$

$$(\mathcal{J}\mathcal{J}\text{-Try}) \qquad \mathit{miss}(C, \mathbf{try}\ a_1\ \mathbf{with}\ x \to a_2, (E,V))$$

$$\frac{\begin{array}{c} \boxed{(E,V)}, C \vdash_{\mathcal{J}\mathcal{J}} a_1 : (\tau_1, \varphi_1, \theta_1, V_1) \triangleright C'' \quad \boxed{(\theta_1\, E[x \mapsto \mathit{exn}[\varphi_1]], V_1)}, C \vdash_{\mathcal{J}\mathcal{J}} a_2 : (\tau_2, \varphi_2, \theta_2, V_2) \triangleright C''' \\ (\mu, V_3) = \mathtt{mgu}_{V_2}\{\theta_2\varphi_1 = \varphi_2\} \wedge \tau = \mu\tau_2 \wedge \varphi = \mu\varphi_2 \wedge \theta = \mu \circ \theta_2 \circ \theta_1 \wedge V' = V_3 \\ C' = C \cup C'' \cup C''' \cup \{(\mathbf{try}\ a_1\ \mathbf{with}\ x \to a_2, (E_{\mid FV(\mathbf{try}\ a_1\ \mathbf{with}\ x \to a_2)}, V), (\tau, \varphi, \theta, V'))\} \end{array}}{(E,V), C \vdash_{\mathcal{J}\mathcal{J}} \mathbf{try}\ a_1\ \mathbf{with}\ x \to a_2 : (\tau, \varphi, \theta, V') \triangleright C'}$$

Figure 27: The rules of the incrementalized inference algorithm for terms of [140] (part II).

4.3.6 *Type checking the dependently-typed λ-calculus*

Here, we instantiate our algorithmic schema to a type checker with dependent types for the λ-calculus, the evaluation rules of which are standard. We follow the type system of λLF in [186], except that ours presents both kinding and typing relations uniformly. Below, we recall the syntax of terms, types, kinds and contexts.

$$\begin{aligned} \mathit{Term} \ni t &::= x \mid t_1\, t_2 \mid \lambda x : T.\, t \\ \mathit{Type} \ni T &::= X \mid \Pi x : T_1.\, T_2 \mid T\, t \\ \mathit{Kind} \ni K &::= * \mid \Pi x : T.\, K \\ \mathit{TypeCtx} \ni \Gamma &::= \emptyset \mid \Gamma,\, x : T \mid X : K \end{aligned}$$

The algorithmic kinding and type checking of λLF are in Figures 28 and 29, where we assume that (initial) typing contexts and kinds are well-formed and omit the relevant rules.

We now follow the four steps of our schema, and obtain the algorithmic incrementalized version of both the kinding and the type checker of λLF.

DEFINING THE SHAPE OF CACHES Because we have a uniform presentation of kinding and typing relations, the entries of the cache are (A, Γ, R) where either A is a type T and R is a kind K, or A is a term t and R is a type T. Thus, a cache C is an element of

$$\mathit{Cache} \triangleq \wp\big((\mathit{Type} \cup \mathit{Term}) \times \mathit{TypeEnv} \times (\mathit{Kind} \cup \mathit{Type})\big).$$

BUILDING CACHES The second step instantiates the $\mathit{buildCache}_{\mathfrak{D}}$ template, resulting in Figure 30.

INCREMENTAL TYPING Figures 31 and 32 display all the rules defining the algorithm $\mathcal{I}\mathfrak{D}$ for kinding and typing. Note that we have duplicated the rules for hit for the sake of clarity, although this is not formally needed because we have a uniform presentation of kinding and typing. The instantiation is again rather straightforward: all the rules mimic the behavior of algorithm $\mathfrak{D}$, following the templates of Section 4.2.

The definition of the predicate $\mathit{compat}_{\mathfrak{D}}$ is as follows:

Definition 4.15. *Let A be a type or a term, then*

$$\begin{aligned} &\mathit{compat}_{\mathfrak{D}}(\Gamma, \Gamma', A) \triangleq \\ &\quad \mathtt{dom}\,(\Gamma) \cap \mathtt{dom}\,(\Gamma') \supseteq FV(A) \wedge \forall v \in FV(A).\, \Gamma(v) = \Gamma'(v) \end{aligned}$$

($\mathfrak{D}$-KA-VAR)
$$\frac{X : K \in \Gamma}{\Gamma \vdash_{\mathfrak{D}} X : K}$$

($\mathfrak{D}$-KA-PI)
$$\frac{\boxed{\Gamma} \vdash_{\mathfrak{D}} T_1 : * \qquad \boxed{\Gamma,\, x : T_1} \vdash_{\mathfrak{D}} T_2 : *}{\Gamma \vdash_{\mathfrak{D}} \Pi x : T_1.\, T_2 : *}$$

($\mathfrak{D}$-KA-APP)
$$\frac{\boxed{\Gamma} \vdash_{\mathfrak{D}} S : \Pi x : T_1.\, K \qquad \boxed{\Gamma} \vdash_{\mathfrak{D}} t : T_2 \qquad \Gamma \vdash T_1 \equiv T_2 \wedge K' = K\{t/x\}}{\Gamma \vdash_{\mathfrak{D}} S\, t : K'}$$

Figure 28: The rules for the algorithmic kinding of $\mathfrak{D}$ of λLF [186].

$$\frac{\boxed{x : T \in \Gamma}}{\Gamma \vdash_{\mathfrak{D}} x : T}\ (\mathfrak{D}\text{-TA-VAR}) \qquad \frac{\boxed{\Gamma} \vdash_{\mathfrak{D}} S : * \quad \boxed{\Gamma, x : S} \vdash_{\mathfrak{D}} t : T \quad \boxed{T' = \Pi x : S.\, T}}{\Gamma \vdash_{\mathfrak{D}} \lambda x : S.\, t : T'}\ (\mathfrak{D}\text{-TA-ABS})$$

$$\frac{\boxed{\Gamma} \vdash_{\mathfrak{D}} t_1 : \Pi x : S_1.\, T \quad \boxed{\Gamma} \vdash_{\mathfrak{D}} t_2 : S_2 \quad \boxed{\Gamma \vdash S_1 \equiv S_2 \wedge T' = T\{t_2/x\}}}{\Gamma \vdash_{\mathfrak{D}} t_1\, t_2 : T'}\ (\mathfrak{D}\text{-TA-APP})$$

Figure 29: The rules for the algorithmic typing of $\mathfrak{D}$ of λLF [186].

$$\begin{aligned}
&\mathit{buildCache}_{\mathfrak{D}}\,((X : K), \Gamma) \triangleq \{(X, X : K, K)\}\\
&\mathit{buildCache}_{\mathfrak{D}}\,((\Pi x : T_1.\, T_2 : K), \Gamma) \triangleq \{(\Pi x : T_1.\, T_2, \Gamma_{|\,FV(\Pi x:T_1.\,T_2)}, K)\}\\
&\qquad \cup (\mathit{buildCache}_{\mathfrak{D}}\,((T_1 : K_1), \boxed{\Gamma})) \cup (\mathit{buildCache}_{\mathfrak{D}}\,((T_2 : K_2), \boxed{\Gamma, x : T_1}))\\
&\mathit{buildCache}_{\mathfrak{D}}\,((S\,t : K), \Gamma) \triangleq \{(S\,t, \Gamma_{|\,FV(S\,t)}, K)\}\\
&\qquad \cup (\mathit{buildCache}_{\mathfrak{D}}\,((S : K'), \boxed{\Gamma})) \cup (\mathit{buildCache}_{\mathfrak{D}}\,((t : T), \boxed{\Gamma}))\\
&\mathit{buildCache}_{\mathfrak{D}}\,((x : T), \Gamma) \triangleq \{(x, x : T, T)\}\\
&\mathit{buildCache}_{\mathfrak{D}}\,((\lambda x : S.\, t : T), \Gamma) \triangleq \{(\lambda x : S.\, t, \Gamma_{|\,FV(\lambda x:S.\,t)}, T)\}\\
&\qquad \cup (\mathit{buildCache}_{\mathfrak{D}}\,((S : K), \boxed{\Gamma})) \cup (\mathit{buildCache}_{\mathfrak{D}}\,((t : T), \boxed{\Gamma, x : S}))\\
&\mathit{buildCache}_{\mathfrak{D}}\,((t_1\, t_2 : T), \Gamma) \triangleq \{(t_1\, t_2, \Gamma_{|\,FV(t_1\,t_2)}, T)\}\\
&\qquad \cup (\mathit{buildCache}_{\mathfrak{D}}\,((t_1 : S_1), \boxed{\Gamma})) \cup (\mathit{buildCache}_{\mathfrak{D}}\,((t_2 : S_2), \boxed{\Gamma}))
\end{aligned}$$

Figure 30: Definition of $\mathit{buildCache}_{\mathfrak{D}}$ for the incremental kinding and typing of λLF.

PROVING TYPE COHERENCE The following lemma holds trivially:

Theorem 4.3.5. *$compat_{\mathfrak{D}}$ expresses compatibility w.r.t. $=$, and $\mathfrak{D}$ preserves $=$.*

Consequently, we instantiate Theorems 4.2 to 4.4 to achieve the following:

Corollary 4.7. *Let C be a well-formed cache, the following properties hold for $\mathfrak{ID}$:*

- ***Coherence:*** *For any term or type A, Γ, R, R', and C'*

$$\Gamma \vdash_{\mathfrak{D}} A : R \;\wedge\; \Gamma, C \vdash_{\mathfrak{ID}} A : R' \triangleright C' \Rightarrow R = R';$$

- ***Completeness:*** *For any term or type A, Γ, and R it holds that*

$$\Gamma \vdash_{\mathfrak{D}} A : R \Rightarrow \exists R', C'.\, \Gamma, C \vdash_{\mathfrak{ID}} A : R' \triangleright C';$$

- ***Well-formedness preservation:*** *For any term or type A, Γ, R, and C' it holds that*

$$\Gamma, C \vdash_{\mathfrak{ID}} A : R \triangleright C' \Rightarrow \Vdash_{\mathfrak{D}} C'.$$

$$\frac{C(T) = \langle \Gamma', K\rangle \qquad compat_{\mathfrak{D}}(\Gamma, \Gamma', T)}{\Gamma, C \vdash_{\mathcal{I}\mathfrak{D}} T : K \triangleright C} \;(\mathcal{I}\mathfrak{D}\text{-Hit})$$

$$\frac{\Gamma \vdash_{\mathfrak{D}} T : K \qquad C' = C \cup \{(X, \Gamma_{|X}, K)\}}{\Gamma \vdash_{\mathcal{I}\mathfrak{D}} X : K \triangleright C'} \; miss(C, X, \Gamma) \;(\mathfrak{D}\text{-TA-VarMiss})$$

$$\frac{\boxed{\Gamma, x : T_1}, C \vdash_{\mathcal{I}\mathfrak{D}} T_2 : * \triangleright C_2 \qquad \boxed{\Gamma}, C \vdash_{\mathcal{I}\mathfrak{D}} T_1 : * \triangleright C_1 \quad C' = C \cup C_1 \cup C_2 \cup \{(\Pi x : T_1 . T_2, \Gamma_{| FV(\Pi x : T_1 . T_2)}, *)\}}{\Gamma, C \vdash_{\mathcal{I}\mathfrak{D}} \Pi x : T_1 . T_2 : * \triangleright C'} \; miss(C, \Pi x : T_1 . T_2, \Gamma) \;(\mathfrak{D}\text{-KA-PiMiss})$$

$$\frac{\boxed{\Gamma}, C \vdash_{\mathcal{I}\mathfrak{D}} S : \Pi x : T_1 . K \triangleright C_1 \qquad \boxed{\Gamma}, C \vdash_{\mathcal{I}\mathfrak{D}} t : T_2 \triangleright C_2 \qquad \boxed{\Gamma \vdash T_1 \equiv T_2 \wedge K' = K\{t/x\}} \qquad C' = C \cup C_1 \cup C_2 \cup \{(S\,t, \Gamma_{| FV(S\,t)}, K')\}}{\Gamma, C \vdash_{\mathcal{I}\mathfrak{D}} S\,t : K' \triangleright C'} \; miss(C, S\,t, \Gamma) \;(\mathfrak{D}\text{-KA-AppMiss})$$

Figure 31: The rules for the incrementalized version of the kinding of algorithm $\mathfrak{D}$ of λLF [186], where $compat_{\mathfrak{D}}$ is as in Definition 4.15.

$$\frac{C(t) = \langle \Gamma', T\rangle \qquad compat_{\mathfrak{D}}(\Gamma, \Gamma', t)}{\Gamma, C \vdash_{\mathcal{I}\mathfrak{D}} t : T \triangleright C} \;(\mathcal{I}\mathfrak{D}\text{-Hit})$$

$$\frac{\Gamma \vdash_{\mathfrak{D}} x : T \qquad C' = C \cup \{(x, \Gamma_{|x}, T)\}}{\Gamma, C \vdash_{\mathcal{I}\mathfrak{D}} x : T \triangleright C'} \; miss(C, x, \Gamma) \;(\mathfrak{D}\text{-TA-VarMiss})$$

$$\frac{\boxed{\Gamma}, C \vdash_{\mathcal{I}\mathfrak{D}} S : * \triangleright C_1 \qquad \boxed{\Gamma, x : S}, C \vdash_{\mathcal{I}\mathfrak{D}} t : T \triangleright C_2 \qquad \boxed{T' = \Pi x : S . T} \qquad C' = C \cup C_1 \cup C_2 \cup \{(\lambda x : S . t, \Gamma_{| FV(\lambda x : S . t)}, T')\}}{\Gamma, C \vdash_{\mathcal{I}\mathfrak{D}} \lambda x : S . t : T' \triangleright C'} \; miss(C, \lambda x : S . t, \Gamma) \;(\mathcal{I}\mathfrak{D}\text{-TA-AbsMiss})$$

$$\frac{\boxed{\Gamma}, C \vdash_{\mathcal{I}\mathfrak{D}} t_1 : \Pi x : S_1 . T \triangleright C_1 \qquad \boxed{\Gamma}, C \vdash_{\mathcal{I}\mathfrak{D}} t_2 : S_2 \triangleright C_2 \qquad \boxed{\Gamma \vdash S_1 \equiv S_2 \wedge T' = T\{t_2/x\}} \qquad C' = C \cup C_1 \cup C_2 \cup \{(t_1\,t_2, \Gamma_{| FV(t_1\,t_2)}, T')\}}{\Gamma, C \vdash_{\mathcal{I}\mathfrak{D}} t_1\,t_2 : T' \triangleright C'} \; miss(C, t_1\,t_2, \Gamma) \;(\mathcal{I}\mathfrak{D}\text{-TA-AppMiss})$$

Figure 32: The rules for the incrementalized version of the typing of algorithm $\mathfrak{D}$ of λLF [186], where $compat_{\mathfrak{D}}$ is as in Definition 4.15.

4.3.7 *Type checking secrecy in the spi-calculus*

The last use case illustrates how our algorithmic schema can be instantiated also to process calculi, and we consider Abadi's type system for secrecy [3]. We assume a set of variables ranged over by $x, y, \ldots$ and a set of names ranged over by n.

We omit the definition of the semantics, the choice of which is actually immaterial for our treatment, and we only briefly recall the syntax of the spi-calculus:

$$\begin{aligned} L &::= n \mid (M, N) \mid 0 \mid suc(M) \mid x \mid \{M_1, \ldots, M_k\}_N \\ P &::= \overline{M}\langle N_1, \ldots, N_k\rangle.P \mid M(x_1, \ldots, x_k).P \mid nil \mid P|Q \mid !P \mid (\nu n)P \mid \\ &\qquad [M\ is\ N]P \mid let\ (x, y) = M\ in\ P \mid \\ &\qquad case\ M\ of\ 0 : P\ suc(x) : Q \mid case\ L\ of\ \{x_1, \ldots, x_k\}_N\ in\ P \end{aligned}$$

A term L can be a name n; a pair (M, N); the 0 term; the successor of another term $suc(M)$; a variable x; or $\{M_1, \ldots, M_k\}_N$, the symmetric key encryption of the terms $M_1, \ldots, M_k$. Instead, a process may be:

- An output process $\overline{M}\langle N_1, \ldots, N_k\rangle.P$ that outputs the terms $N_1, \ldots, N_k$ on M upon interaction;
- An input process $M(x_1, \ldots, x_k).P$ that reduces to $P[N_1/x_1, \ldots, N_k/x_k]$ when $N_1, \ldots, N_k$ are communicated over M;
- The nil process that does nothing;
- A composition $P|Q$ that intuitively behaves as P and Q run in parallel;
- A replication $!P$, behaving as an infinite composition of P with itself;
- A restriction $(\nu n)P$ that creates a new private name n and then behaves as P;
- A matching $[M\ is\ N]P$ that reduces to P if M and N are the same (stuck otherwise);
- A pair splitting process $let\ (x, y) = M\ in\ P$ that reduces to $P[N/x][L/y]$ if $M = (N, L)$ (stuck otherwise);
- The case process $case\ L\ of\ \{x_1, \ldots, x_k\}_N\ in\ P$, that tries to decrypt L with the key N: if L has the form $\{M_1, \ldots, M_k\}_N$, then process behaves as $P[M_1/x_1, \ldots, M_k/x_k]$ (stuck otherwise);

Names are assigned types T, R in the set $\mathcal{L} = \{Secret, Public, Any\}$, with the expected meaning. Types are partially ordered by $T <: R$ if $T = R$ or R is *Any*, writing $T \sqcup R$ for the join of T and R. If the process P typechecks then it does not leak the values of parameters of level *Any*, typically the payload of the messages of a protocol. The original type system $\mathcal{P}$ is in Figures 33 to 35, where we omit the rules for checking that a type environment E is well-formed and assume it to be always such; we follow [3] and write $n : T :: \{M_1, \ldots, M_k, n\}_N$ for n with type T when a name n is a confounder only used in the term $\{M_1, \ldots, M_k, n\}_N$. As usual, to obtain a fully syntax-directed type system we directly incorporate type subsumption in the rules. We refer the interested reader to Abadi [3] for more details about the original typing algorithm. We now show how to obtain the

$$(\mathcal{P}\text{-Var})\ \frac{T = E(x)}{E \vdash_{\mathcal{P}} x : T} \qquad (\mathcal{P}\text{-Name})\ \frac{n : T :: \{M_1, \ldots, M_k, n\}_N \in E}{E \vdash_{\mathcal{P}} n : T} \qquad (\mathcal{P}\text{-Zero})\ \frac{}{E \vdash_{\mathcal{P}} 0 : Public} \qquad (\mathcal{P}\text{-Suc})\ \frac{E \vdash_{\mathcal{P}} M : T}{E \vdash_{\mathcal{P}} suc(M) : T}$$

$$(\mathcal{P}\text{-Pair})\ \frac{E \vdash_{\mathcal{P}} M : R \quad E \vdash_{\mathcal{P}} N : U \quad T = R \sqcup U}{E \vdash_{\mathcal{P}} (M, N) : T} \qquad (\mathcal{P}\text{-PEncr1})\ \frac{E \vdash_{\mathcal{P}} N : Public}{E \vdash_{\mathcal{P}} \{\}_N : Public}$$

$$(\mathcal{P}\text{-PEncr2})\ \frac{E \vdash_{\mathcal{P}} M_1 : T_1 \ldots E \vdash_{\mathcal{P}} M_k : T_k \quad k > 0 \wedge T = \bigsqcup_{i=1}^{k} T_i \quad E \vdash_{\mathcal{P}} N : Public}{E \vdash_{\mathcal{P}} \{M_1, \ldots, M_k\}_N : T}$$

$$(\mathcal{P}\text{-SEncr})\ \frac{E \vdash_{\mathcal{P}} M_2 : Any \quad E \vdash_{\mathcal{P}} M_3 : Public \quad \begin{array}{c} E \vdash_{\mathcal{P}} M_1 : Secret \\ E \vdash_{\mathcal{P}} N : Secret \end{array} \quad n : T :: \{M_1, M_2, M_3, n\}_N \in E}{E \vdash_{\mathcal{P}} \{M_1, M_2, M_3, n\}_N : Public}$$

Figure 33: The original type system $\mathcal{P}$ of [3] for terms.

incrementalized algorithm $\mathcal{IP}$ by instantiating the four steps of our schema.

$$\text{(}\mathcal{P}\text{-POut)}\quad \frac{\boxed{E} \vdash_{\mathcal{P}} M : Public \quad \boxed{E} \vdash_{\mathcal{P}} M_1 : Public \ldots \boxed{E} \vdash_{\mathcal{P}} M_k : Public \quad \boxed{E} \vdash_{\mathcal{P}} P : \mathrm{OK}}{E \vdash_{\mathcal{P}} \overline{M}\langle M_1, \ldots, M_k \rangle.P : \mathrm{OK}}$$

$$\text{(}\mathcal{P}\text{-SOut)}\quad \frac{\boxed{E} \vdash_{\mathcal{P}} M : Secret \quad \boxed{E} \vdash_{\mathcal{P}} M_1 : Secret \quad \boxed{E} \vdash_{\mathcal{P}} M_2 : Any \quad \boxed{E} \vdash_{\mathcal{P}} M_3 : Public \quad \boxed{E} \vdash_{\mathcal{P}} P : \mathrm{OK}}{E \vdash_{\mathcal{P}} \overline{M}\langle M_1, M_2, M_3 \rangle.P : \mathrm{OK}}$$

$$\text{(}\mathcal{P}\text{-PIn)}\quad \frac{\boxed{E} \vdash_{\mathcal{P}} M : Public \quad \boxed{E} \vdash_{\mathcal{P}} x_1 : Public \ldots E \vdash_{\mathcal{P}} x_k : Public \quad \boxed{E} \vdash_{\mathcal{P}} P : \mathrm{OK}}{E \vdash_{\mathcal{P}} M(x_1, \ldots, x_k).P : \mathrm{OK}}$$

$$\text{(}\mathcal{P}\text{-SIn)}\quad \frac{\boxed{E} \vdash_{\mathcal{P}} M : Secret \quad \boxed{E} \vdash_{\mathcal{P}} x_1 : Secret \quad \boxed{E} \vdash_{\mathcal{P}} x_2 : Any \quad \boxed{E} \vdash_{\mathcal{P}} x_3 : Public \quad \boxed{E} \vdash_{\mathcal{P}} P : \mathrm{OK}}{E \vdash_{\mathcal{P}} M(x_1, x_2, x_3).P : \mathrm{OK}}$$

$$\text{(}\mathcal{P}\text{-Nil)}\quad \frac{}{\boxed{E} \vdash_{\mathcal{P}} nil : \mathrm{OK}} \qquad \text{(}\mathcal{P}\text{-Par)}\quad \frac{\boxed{E} \vdash_{\mathcal{P}} P : \mathrm{OK} \quad \boxed{E} \vdash_{\mathcal{P}} Q : \mathrm{OK}}{E \vdash_{\mathcal{P}} P \mid Q : \mathrm{OK}} \qquad \text{(}\mathcal{P}\text{-Rep)}\quad \frac{\boxed{E} \vdash_{\mathcal{P}} P : \mathrm{OK}}{E \vdash_{\mathcal{P}} !P : \mathrm{OK}}$$

$$\text{(}\mathcal{P}\text{-Res)}\quad \frac{\boxed{E[n : T :: \{M_1, \ldots, M_k, n\}_N]} \vdash_{\mathcal{P}} P : \mathrm{OK}}{E \vdash_{\mathcal{P}} (\nu n)P : \mathrm{OK}}$$

Figure 34: The original type system $\mathcal{P}$ of [3] for processes (part I).

DEFINING THE SHAPE OF CACHES Each entry of a cache is a triple, made of a term or a process, an environment $E \in TypeEnv$ (in this case $TypeEnv$ coincides with $TypeCtx$), and a result $S \in (\mathcal{L} \cup \{\mathrm{OK}\})$:

$$Cache \triangleq \wp\big((Term \cup Processes) \times TypeCtx \times (\{Secret, Public, Any, \mathrm{OK}\})\big).$$

BUILDING CACHES The second step instantiates the $buildCache_{\mathcal{P}}$ template to fit the spi-calculus, see Figure 36. In most of the rules, the instantiation is straightforward, because E in the premises and in the conclusions is unchanged.

INCREMENTAL TYPING The third step of the process consists in instantiating the rule templates of Definition 4.7, which yields the incrementalized type algorithm $\mathcal{IP}$ in Figures 37 to 39. Here, the predicate $compat_{\mathcal{P}}$ uses type equality $=$ as the relation for deciding when two types are compatible:

Definition 4.16. *Let X be a process P or a term M; let v be a name or a variable; and let E, E' be two typing environments. Then we define*

$$compat_{\mathcal{P}}(E, E', X) \triangleq$$
$$\mathtt{dom}\,(E) \cap \mathtt{dom}\,(E') \supseteq FV(X) \,\wedge\, E_{|\,FV(X)} = E'_{|\,FV(X)}$$

The rules for when the cache has a compatible entry are plain. As usual the rules mimic those of the original type system, except that the incremental ones have calls to $\mathcal{IP}$ instead of $\mathcal{P}$ and cache updates. Note again that the needed extensions to the typing environment (dictated by the instantiation of $\boxed{tr}$), as well as the checks on the type T (due to the instantiation of $\boxed{checkJoin}$), are exactly the same as in the original rules.

$$\frac{(\mathcal{P}\text{-Match}) \quad \boxed{E} \vdash_{\mathcal{P}} M : T \quad \boxed{E} \vdash_{\mathcal{P}} N : R \quad \boxed{E} \vdash_{\mathcal{P}} P : \text{OK} \quad T, R \in \{Public, Secret\}}{E \vdash_{\mathcal{P}} [M \text{ is } N] \text{ in } P : \text{OK}}$$

$$\frac{(\mathcal{P}\text{-Split}) \quad \boxed{E} \vdash_{\mathcal{P}} M : T \quad \boxed{E[x : T, y : T]} \vdash_{\mathcal{P}} P : \text{OK} \quad T \in \{Public, Secret\}}{E \vdash_{\mathcal{P}} \text{let } (x, y) = M \text{ in } P : \text{OK}}$$

$$\frac{(\mathcal{P}\text{-Int}) \quad \boxed{E} \vdash_{\mathcal{P}} M : T \quad \boxed{E} \vdash_{\mathcal{P}} P : \text{OK} \quad \boxed{E[x : T]} \vdash_{\mathcal{P}} Q : \text{OK} \quad T \in \{Public, Secret\}}{E \vdash_{\mathcal{P}} \text{case } M \text{ of } 0 : P \text{ suc}(x) : Q : \text{OK}}$$

$$\frac{(\mathcal{P}\text{-PDecr}) \quad \boxed{E} \vdash_{\mathcal{P}} L : T \quad \boxed{E} \vdash_{\mathcal{P}} N : Public \quad \boxed{E[x_1 : T, \ldots, x_k : T]} \vdash_{\mathcal{P}} P : \text{OK} \quad T \in \{Public, Secret\}}{E \vdash_{\mathcal{P}} \text{case } L \text{ of } \{x_1, \ldots, x_k\}_N \text{ in } P : \text{OK}}$$

$$\frac{(\mathcal{P}\text{-SDecr}) \quad \boxed{E} \vdash_{\mathcal{P}} L : T \quad \boxed{E} \vdash_{\mathcal{P}} N : Secret \quad \boxed{E[x_1 : Secret, x_2 : Any, x_3 : Public, x_4 : Any]} \vdash_{\mathcal{P}} P : \text{OK} \quad T \in \{Public, Secret\}}{E \vdash_{\mathcal{P}} \text{case } L \text{ of } \{x_1, x_2, x_3, x_4\}_N \text{ in } P : \text{OK}}$$

Figure 35: The original type system $\mathcal{P}$ of [3] for processes (part II).

PROVING TYPE COHERENCE The following lemma easily follows from the above definitions:

Theorem 4.3.6. *$compat_{\mathcal{P}}$ expresses compatibility w.r.t. $=$, and $\mathcal{P}$ preserves $=$.*

We can then instantiate Theorems 4.2 to 4.4 guarantee the correctness of the incrementalized algorithm. As we did for Section 4.3.5, we split the corollaries for terms and for processes for the sake of readability. Since the result of $\mathcal{P}$ for processes can only be OK (if any), the *coherence* property for processes holds trivially (it is an implication whose consequence is always true) and thus is omitted.

Corollary 4.8 (Terms). *Let C be a well-formed cache, the following properties hold for $\mathcal{IP}$:*

- ***Coherence:*** *For any term M, any E, T, T', and cache C':*

$$E \vdash_{\mathcal{P}} M : T \ \wedge\ E, C \vdash_{\mathcal{IP}} M : T' \triangleright C' \Rightarrow T = T';$$

- ***Completeness:*** *For any term M, any E, and T:*

$$E \vdash_{\mathcal{P}} M : T \Rightarrow \exists T', C'.\ E, C \vdash_{\mathcal{IP}} M : T' \triangleright C';$$

- ***Well-formedness preservation:*** *For any term M, any E, T, and cache C':*

$$E, C \vdash_{\mathcal{IP}} M : T \triangleright C' \Rightarrow \Vdash_{\mathcal{P}} C'.$$

Corollary 4.9 (Processes). *Let C be a well-formed cache, the following properties hold for $\mathcal{IP}$:*

- ***Completeness:*** *For any process P and any E:*

$$E \vdash_{\mathcal{P}} P : \text{OK} \Rightarrow \exists C'.\ E, C \vdash_{\mathcal{IP}} P : \text{OK} \triangleright C';$$

- ***Well-formedness preservation:*** *For any process P, any E, and cache C':*

$$E, C \vdash_{\mathcal{IP}} P : \text{OK} \triangleright C' \Rightarrow \Vdash_{\mathcal{P}} C'.$$

$$
\begin{aligned}
&buildCache_{\mathcal{P}}\,((n : T :: \{M_1,\dots,M_k,n\}_N), E) \triangleq \{(n, E_{|\,n}, T)\}\\
&buildCache_{\mathcal{P}}\,(((M,N)\colon T), E) \triangleq \{((M,N), E_{|\,FV((M,N))}, T)\}\\
&\qquad \cup\, buildCache_{\mathcal{P}}\left((M : T'), \boxed{E_{|\,FV(M)}}\right) \cup buildCache_{\mathcal{P}}\left((N : T''), \boxed{E_{|\,FV(N)}}\right)\\
&buildCache_{\mathcal{P}}\,((0 : Public), E) \triangleq \{(0, \emptyset, Public)\}\\
&buildCache_{\mathcal{P}}\,((suc(M)\colon T), E) \triangleq \{(suc(M), E_{|\,FV(M)}, T)\}\\
&buildCache_{\mathcal{P}}\,((\{M_1,\dots,M_k\}_N\colon T), E) \triangleq (\{M_1,\dots,M_k\}_N, E_{|\bigcup_{i=1}^{k} FV(M_i)}, T\\
&\qquad \cup\, buildCache_{\mathcal{P}}\left((M_1\colon T_1), \boxed{E_{|\,FV(M_1)}}\right) \cup \cdots \cup buildCache_{\mathcal{P}}\left((M_k\colon T_k), \boxed{E_{|\,FV(M_k)}}\right)\\
&buildCache_{\mathcal{P}}\,((x\colon T), E) \triangleq \{(x, E_{|\,x}, T)\}\\
&buildCache_{\mathcal{P}}\,\left((\overline{M_0}\langle M_1,\dots,M_k\rangle.P\colon \mathrm{OK}), E\right) \triangleq (\overline{M_0}, E_{|\bigcup_{i=0}^{k} FV(M_i)}, T)\\
&\qquad \cup\, buildCache_{\mathcal{P}}\left((M_1\colon T_1), \boxed{E_{|\,FV(M_1)}}\right) \cup \cdots \cup buildCache_{\mathcal{P}}\left((M_k\colon T_k), \boxed{E_{|\,FV(M_k)}}\right)\\
&buildCache_{\mathcal{P}}\,((M(x_1,\dots,x_k).P\colon \mathrm{OK}), E) \triangleq (M, E_{|\,FV(M)}, T)\\
&\qquad \cup\, buildCache_{\mathcal{P}}\left((x_1\colon T_1), \boxed{E_{|\,x_1}}\right) \cup \cdots \cup buildCache_{\mathcal{P}}\left((x_k\colon T_k), \boxed{E_{|\,x_k}}\right)\\
&buildCache_{\mathcal{P}}\,((nil\colon \mathrm{OK}), E) \triangleq (nil, E, \mathrm{OK})\\
&buildCache_{\mathcal{P}}\,((P|Q\colon \mathrm{OK}), E) \triangleq (P|Q, E_{|\,FV(P)\cup FV(Q)}, \mathrm{OK})\\
&\qquad \cup\, buildCache_{\mathcal{P}}\left((P\colon \mathrm{OK}), \boxed{E_{|\,FV(P)}}\right) \cup buildCache_{\mathcal{P}}\left((Q\colon \mathrm{OK}), \boxed{E_{|\,FV(Q)}}\right)\\
&buildCache_{\mathcal{P}}\,((\,!P\colon \mathrm{OK}), E) \triangleq (\,!P, E_{|\,FV(P)}, \mathrm{OK})\\
&buildCache_{\mathcal{P}}\,(((\nu n)P\colon \mathrm{OK}), E) \triangleq ((\nu n)P, E, \mathrm{OK})\\
&\qquad \cup\, buildCache_{\mathcal{P}}\left((P\colon \mathrm{OK}), \boxed{E_{|\,FV(P)\cup\{n\}}}\right) \cup buildCache_{\mathcal{P}}\left((n\colon T), \boxed{E}\right)\\
&buildCache_{\mathcal{P}}\,(([M\ is\ N]P\colon \mathrm{OK}), E) \triangleq ([M\ is\ N]P, E_{|\,FV(M)\cup FV(N)\cup FV(P)}, \mathrm{OK})\\
&\qquad \cup\, buildCache_{\mathcal{P}}\left((M\colon T), \boxed{E_{|\,FV(M)}}\right) \cup buildCache_{\mathcal{P}}\left((N\colon T), \boxed{E_{|\,FV(N)}}\right) \cup\\
&\qquad buildCache_{\mathcal{P}}\left((P\colon \mathrm{OK}), \boxed{E_{|\,FV(P)}}\right)\\
&buildCache_{\mathcal{P}}\,((let\ (x,y) = M\ in\ P\colon \mathrm{OK}), E) \triangleq \{(let\ (x,y) = M\ in\ P, E_{|\,FV(M)\cup FV(P)}, \mathrm{OK})\} \cup\\
&\qquad buildCache_{\mathcal{P}}\left((M\colon T), \boxed{E_{|\,FV(M)}}\right) \cup buildCache_{\mathcal{P}}\left((P\colon \mathrm{OK}), \boxed{E_{|\,FV(P)}[x \mapsto T, y \mapsto T]}\right)\\
&buildCache_{\mathcal{P}}\,((case\ M\ of\ 0 : P\ suc(x) : Q\colon \mathrm{OK}), E) \triangleq\\
&\qquad \{case\ M\ of\ 0 : P\ suc(x) : Q, E_{|\,FV(M)\cup\{x\}\cup FV(P)}, \mathrm{OK}\} \cup\\
&\qquad buildCache_{\mathcal{P}}\left((M\colon T), \boxed{E_{|\,FV(M)}}\right) \cup buildCache_{\mathcal{P}}\left((P\colon \mathrm{OK}), \boxed{E_{|\,FV(P)}}\right) \cup\\
&\qquad buildCache_{\mathcal{P}}\left((Q\colon \mathrm{OK}), \boxed{E_{|\,FV(Q)\cup\{x\}}}\right)\\
&buildCache_{\mathcal{P}}\,((case\ L\ of\ \{x_1,\dots,x_k\}_N\ in\ P\colon \mathrm{OK}), E) \triangleq\\
&\qquad \{case\ L\ of\ \{x_1,\dots,x_k\}_N\ in\ P, E_{|\,FV(M)\cup\{x\}\cup FV(P)}, \mathrm{OK}\} \cup\\
&\qquad buildCache_{\mathcal{P}}\left((L\colon T), \boxed{E_{|\,FV(L)}}\right) \cup buildCache_{\mathcal{P}}\left((N\colon \mathrm{OK}), \boxed{E_{|\,FV(N)}}\right) \cup\\
&\qquad buildCache_{\mathcal{P}}\left((P\colon \mathrm{OK}), \boxed{E_{|\,FV(P)\cup\bigcup_{i=0}^{k}\{x_i\}}}\right)
\end{aligned}
$$

Figure 36: Instantiation of $buildCache_{\mathcal{P}}$ for $\mathcal{P}$ [3].

(𝒥𝒫-THit)
$$\frac{C(M) = \langle E', T\rangle \qquad compat_{\mathcal{P}}(E, E', M)}{E, C \vdash_{\mathcal{JP}} M : T \triangleright C}$$

(𝒥𝒫-VarMiss)
$$\frac{E \vdash_{\mathcal{P}} x : T \qquad C' = C \cup \{(x, E_{|\{x\}}, T)\}}{E, C \vdash_{\mathcal{JP}} x : T \triangleright C'} \; miss(C, x, E)$$

(𝒥𝒫-NameMiss)
$$\frac{E \vdash_{\mathcal{P}} n : T \qquad C' = C \cup \{(n, E_{|\{n\}}, T)\}}{E, C \vdash_{\mathcal{JP}} n : T \triangleright C'} \; miss(C, n, E)$$

(𝒥𝒫-ZeroMiss)
$$\frac{E \vdash_{\mathcal{P}} 0 : Public \qquad C' = C \cup \{(0, \emptyset, T)\}}{E, C \vdash_{\mathcal{JP}} 0 : Public \triangleright C'} \; miss(C, 0, E)$$

(𝒥𝒫-SucMiss)
$$\frac{\boxed{E}, C \vdash_{\mathcal{JP}} M : T \triangleright C'' \qquad C' = C \cup C'' \cup \{(suc(M), E_{|FV(M)}, T)\}}{E, C \vdash_{\mathcal{JP}} suc(M) : T \triangleright C'} \; miss(C, suc(M), E)$$

(𝒥𝒫-PairMiss)
$$\frac{\begin{array}{c}\boxed{E}, C \vdash_{\mathcal{JP}} M : R \triangleright C'' \qquad \boxed{E}, C \vdash_{\mathcal{JP}} N : U \triangleright C''' \\ \boxed{T = R \sqcup U} \qquad C' = C \cup C'' \cup C''' \cup \{((M, N), E_{|FV((M,N))}, T)\}\end{array}}{E, C \vdash_{\mathcal{JP}} (M, N) : T \triangleright C'} \; miss(C, (M, N), E)$$

(𝒥𝒫-SEncrMiss) $\quad miss(C, \{M_1, M_2, M_3, n\}_N, E)$
$$\frac{\begin{array}{c}\boxed{E}, C \vdash_{\mathcal{JP}} M_1 : Secret \triangleright C_1 \\ \boxed{E}, C \vdash_{\mathcal{JP}} M_2 : Any \triangleright C_2 \qquad \boxed{E}, C \vdash_{\mathcal{JP}} M_3 : Public \triangleright C_3 \qquad \boxed{E}, C \vdash_{\mathcal{JP}} N : Secret \triangleright C_0 \\ \boxed{n : T :: \{M_1, M_2, M_3, n\}_N \in E} \qquad C' = C \cup \{(\{M_1, M_2, M_3, n\}_N, E_{|FV(\{M_1,M_2,M_3,n\}_N)}, Public)\} \cup \bigcup_{i=0}^{3} C_i\end{array}}{E, C \vdash_{\mathcal{JP}} \{M_1, M_2, M_3, n\}_N : Public \triangleright C'}$$

(𝒥𝒫-PEncr1Miss)
$$\frac{\boxed{E}, C \vdash_{\mathcal{JP}} N : Public \triangleright C'' \qquad C' = C \cup C'' \cup \{(N, E_{|FV(N)}, Public)\}}{E, C \vdash_{\mathcal{JP}} \{\}_N : Public \triangleright C'} \; miss(C, N, E)$$

(𝒥𝒫-PEncr2Miss)
$$\frac{\begin{array}{c}\forall i \in \{1, \dots, k\}. \boxed{E}, C \vdash_{\mathcal{JP}} M_i : T_i \triangleright C_i \\ \boxed{E}, C \vdash_{\mathcal{JP}} N : Public \triangleright C_0 \qquad \boxed{k > 0 \wedge T = \bigsqcup_{i=1}^{k} T_i} \\ C' = C \cup \{(\{M_1, \dots, M_k\}_N, E_{|FV(\{M_1,\dots,M_k\}_N)}, T)\} \cup \bigcup_{i=0}^{k} C_i\end{array}}{E, C \vdash_{\mathcal{JP}} \{M_1, \dots, M_k\}_N : T \triangleright C'} \; miss(C, \{M_1, \dots, M_k\}_N, E)$$

Figure 37: The incrementalized type system 𝒥𝒫 for terms, where $compat_{\mathcal{P}}$ is as in Definition 4.16.

$$\text{(}\mathcal{IP}\text{-PHIT)}\quad \frac{C(M) = \langle E', \text{OK}\rangle \qquad compat_{\mathcal{P}}(E, E', M)}{E, C \vdash_{\mathcal{IP}} M : \text{OK} \triangleright C}$$

$$\text{(}\mathcal{IP}\text{-POUTMISS)}\quad \frac{\begin{array}{c}\forall i \in \{0, \dots, k\}. \boxed{E}, C \vdash_{\mathcal{IP}} M_i : \textit{Public} \triangleright C_i \qquad \boxed{E}, C \vdash_{\mathcal{IP}} P : \text{OK} \triangleright C'' \\ C' = C \cup C'' \cup \{(\overline{M}_0\langle M_1, \dots, M_k\rangle.P, E_{|FV(\overline{M}_0\langle M_1,\dots,M_k\rangle.P)}, \text{OK})\} \cup \bigcup_{i=0}^{k} C_i\end{array}}{E, C \vdash_{\mathcal{IP}} \overline{M}_0\langle M_1, \dots, M_k\rangle.P : \text{OK} \triangleright C'} \; miss(C, \overline{M}_0\langle M_1, \dots, M_k\rangle.P, E)$$

$$\text{(}\mathcal{IP}\text{-SOUTMISS)}\quad \frac{\begin{array}{c}\boxed{E}, C \vdash_{\mathcal{IP}} M_1 : \textit{Secret} \triangleright C_1 \\ \boxed{E}, C \vdash_{\mathcal{IP}} M_2 : \textit{Any} \triangleright C_2 \qquad \boxed{E}, C \vdash_{\mathcal{IP}} M_3 : \textit{Public} \triangleright C_3 \\ \boxed{E}, C \vdash_{\mathcal{IP}} M_0 : \textit{Secret} \triangleright C_0 \qquad \boxed{E}, C \vdash_{\mathcal{IP}} P : \text{OK} \triangleright C'' \\ C' = C \cup C'' \cup \{(\overline{M}_0\langle M_1, \dots, M_k\rangle.P, E_{|FV(\overline{M}_0\langle M_1,\dots,M_k\rangle.P)}, \text{OK})\} \cup \bigcup_{i=0}^{3} C_i\end{array}}{E, C \vdash_{\mathcal{IP}} \overline{M_0}\langle M_1, M_2, M_3\rangle.P : \text{OK} \triangleright C'} \; miss(C, \overline{M_0}\langle M_1, M_2, M_3\rangle.P, E)$$

$$\text{(}\mathcal{IP}\text{-PINMISS)}\quad \frac{\begin{array}{c}\forall i \in \{1, \dots, k\}. \boxed{E}, C \vdash_{\mathcal{IP}} x_i : \textit{Public} \triangleright C_i \\ \boxed{E}, C \vdash_{\mathcal{IP}} M : \textit{Public} \triangleright C_0 \qquad \boxed{E}, C \vdash_{\mathcal{IP}} P : \text{OK} \triangleright C'' \\ C' = C \cup C'' \cup \{(M(x_1, \dots, x_k).P, E_{|FV(M(x_1,\dots,x_k).P)}, \text{OK})\} \cup \bigcup_{i=0}^{k} C_i\end{array}}{E, C \vdash_{\mathcal{IP}} M(x_1, \dots, x_k).P : \text{OK} \triangleright C'} \; miss(C, M(x_1, \dots, x_k).P, E)$$

$$\text{(}\mathcal{IP}\text{-SINMISS)}\quad \frac{\begin{array}{c}\boxed{E}, C \vdash_{\mathcal{IP}} x_1 : \textit{Secret} \triangleright C_1 \\ \boxed{E}, C \vdash_{\mathcal{IP}} x_2 : \textit{Any} \triangleright C_2 \qquad \boxed{E}, C \vdash_{\mathcal{IP}} x_3 : \textit{Public} \triangleright C_3 \\ \boxed{E}, C \vdash_{\mathcal{IP}} M : \textit{Secret} \triangleright C_0 \qquad \boxed{E}, C \vdash_{\mathcal{IP}} P : \text{OK} \triangleright C'' \\ C' = C \cup C'' \cup \{(M(x_1, x_2, x_3).P, E_{|FV(M(x_1,x_2,x_3).P)}, \text{OK})\} \cup \bigcup_{i=0}^{3} C_i\end{array}}{E, C \vdash_{\mathcal{IP}} M(x_1, x_2, x_3).P : \text{OK} \triangleright C'} \; miss(C, M(x_1, x_2, x_3).P, E)$$

$$\text{(}\mathcal{IP}\text{-NILMISS)}\quad \frac{\boxed{E} \vdash_{\mathcal{P}} nil : \text{OK} \qquad C' = C \cup \{(nil, E, \text{OK})\}}{E, C \vdash_{\mathcal{IP}} nil : \text{OK} \triangleright C'} \; miss(C, nil, E)$$

$$\text{(}\mathcal{IP}\text{-PARMISS)}\quad \frac{\begin{array}{c}\boxed{E}, C \vdash_{\mathcal{IP}} P : \text{OK} \triangleright C'' \\ \boxed{E}, C \vdash_{\mathcal{IP}} Q : \text{OK} \triangleright C''' \qquad C' = C \cup C'' \cup C''' \cup \{(P \mid Q, E_{|FV(P|Q)}, \text{OK})\}\end{array}}{E, C \vdash_{\mathcal{IP}} P \mid Q : \text{OK} \triangleright C'} \; miss(C, P \mid Q, E)$$

$$\text{(}\mathcal{IP}\text{-REPMISS)}\quad \frac{\boxed{E}, C \vdash_{\mathcal{IP}} P : \text{OK} \triangleright C'' \qquad C' = C \cup C'' \cup \{(!P, E_{|FV(!P)}, \text{OK})\}}{E, C \vdash_{\mathcal{IP}} !P : \text{OK} \triangleright C'} \; miss(C, !P, E)$$

Figure 38: The incrementalized type system $\mathcal{IP}$ for processes (part I), where $compat_{\mathcal{P}}$ is as in Definition 4.16.

$$(\mathcal{IP}\text{-ResMiss})\quad \frac{\boxed{E[n:T::\{M_1,\dots,M_k,n\}_N]}, C \vdash_{\mathcal{IP}} P: \mathrm{OK} \triangleright C'' \qquad C' = C \cup C'' \cup \{((\nu n)P, E_{|FV((\nu n)P)}, \mathrm{OK})\}}{E, C \vdash_{\mathcal{IP}} (\nu n)P: \mathrm{OK} \triangleright C'} \; miss(C, (\nu n)P, E)$$

$$(\mathcal{IP}\text{-MatchMiss})\quad \frac{\boxed{E}, C \vdash_{\mathcal{IP}} N: R \triangleright C_1 \quad \boxed{E}, C \vdash_{\mathcal{IP}} M: T \triangleright C_0 \quad \boxed{E}, C \vdash_{\mathcal{P}} P: \mathrm{OK} \triangleright C_2 \quad T, R \in \{Public, Secret\} \qquad C' = C \cup \{([M\ is\ N]\ in\ P, E_{|[M\ is\ N]\ in\ P}, \mathrm{OK})\} \cup \bigcup_{i=0}^{2} C_i}{E, C \vdash_{\mathcal{IP}} [M\ is\ N]\ in\ P: \mathrm{OK} \triangleright C'} \; miss(C, [M\ is\ N]\ in\ P, E)$$

$$(\mathcal{IP}\text{-SplitMiss})\quad \frac{miss(C, (let\ (x,y) = M\ in\ P, E)) \quad \boxed{E}, C \vdash_{\mathcal{IP}} M: T \triangleright C'' \quad \boxed{E[x:T, y:T]}, C \vdash_{\mathcal{IP}} P: \mathrm{OK} \triangleright C''' \qquad C' = C \cup C'' \cup C'''\{(let\ (x,y) = M\ in\ P, E_{|FV((let\ (x,y)=M\ in\ P))}, \mathrm{OK})\}}{E, C \vdash_{\mathcal{IP}} let\ (x,y) = M\ in\ P: \mathrm{OK} \triangleright C'}$$

$$(\mathcal{IP}\text{-IntMiss})\quad \frac{miss(C, case\ M\ of\ 0: P\ suc(x): Q, E) \quad \boxed{E}, C \vdash_{\mathcal{IP}} M: T \triangleright C_0 \quad \boxed{E}, C \vdash_{\mathcal{IP}} P: \mathrm{OK} \triangleright C_1 \quad \boxed{E}, C \vdash_{\mathcal{IP}} Q: \mathrm{OK} \triangleright C_2 \quad T \in \{Public, Secret\} \qquad C' = C \cup \{(case\ M\ of\ 0: P\ suc(x): Q, E_{|FV(case\ M\ of\ 0:P\ suc(x):Q)}, \mathrm{OK})\} \cup \bigcup_{i=0}^{2} C_i}{E, C \vdash_{\mathcal{IP}} case\ M\ of\ 0: P\ suc(x): Q: \mathrm{OK} \triangleright C'}$$

$$(\mathcal{IP}\text{-PDecrMiss})\quad \frac{miss(C, case\ L\ of\ \{x_1,\dots,x_k\}_N\ in\ P, E) \quad \boxed{E}, C \vdash_{\mathcal{IP}} L: T \triangleright C_0 \quad \boxed{E}, C \vdash_{\mathcal{IP}} N: Public \triangleright C_1 \quad \boxed{E[x_1:T,\dots,x_k:T]}, C \vdash_{\mathcal{IP}} P: \mathrm{OK} \triangleright C_2 \quad T \in \{Public, Secret\} \qquad C' = C \cup \{(case\ L\ of\ \{x_1,\dots,x_k\}_N\ in\ P, E_{|FV(case\ L\ of\ \{x_1,\dots,x_k\}_N\ in\ P)}, \mathrm{OK})\} \cup \bigcup_{i=0}^{2} C_i}{E, C \vdash_{\mathcal{IP}} case\ L\ of\ \{x_1,\dots,x_k\}_N\ in\ P: \mathrm{OK} \triangleright C'}$$

$$(\mathcal{IP}\text{-SDecrPMiss})\quad \frac{miss(C, case\ L\ of\ \{x_1,x_2,x_3,x_4\}_N\ in\ P, E) \quad \boxed{E}, C \vdash_{\mathcal{IP}} L: T \triangleright C_0 \quad \boxed{E}, C \vdash_{\mathcal{IP}} N: Secret \triangleright C_1 \quad \boxed{E, x_1: Secret, x_2: Any, x_3: Public, x_4: Any} \vdash_{\mathcal{IP}} P: \mathrm{OK} \triangleright C_2 \quad T \in \{Public, Secret\} \qquad C' = C \cup \left(\cup_{i \in \{0,1,2\}} C_i\right) \cup \{(case\ L\ of\ \{x_1,x_2,x_3,x_4\}_N\ in\ P, E_{|FV(case\ L\ of\ \{x_1,x_2,x_3,x_4\}_N\ in\ P)}, \mathrm{OK})\}}{E, C \vdash_{\mathcal{IP}} case\ L\ of\ \{x_1,x_2,x_3,x_4\}_N\ in\ P: \mathrm{OK} \triangleright C'}$$

Figure 39: The incrementalized type system $\mathcal{IP}$ for processes (part II), where $compat_{\mathcal{P}}$ is as in Definition 4.16.

4.4 IMPLEMENTATION AND EXPERIMENTS

This section presents a prototypical implementation of our algorithmic schema as the OCaml module INCREMENTALIZER. We have applied it to the type checker for MinCaml [219], a monomorphic higher-order core of ML. Finally, we report on some experiments that show that the time and space performance of the incrementalized type checker are almost always better than those of the original MinCaml type checker. In particular, the time depends on the size of *diffs* and decreases as these become smaller, and the space overhead is negligible.

4.4.1 *The* INCREMENTALIZER

The INCREMENTALIZER is a proof-of-concept in the form of a couple of OCaml functors that mechanize our algorithmic schema. They take as input a specification of the language in terms of Definition 4.2: the syntax, the function tr, the predicate $checkJoin$, which fully define the original typing algorithm; the input also includes the shape of the typing results and contexts, and the definition of the relation $compat$. The output is the incrementalized typing algorithm (including an implementation of caches and of $buildCache$). For that, we provide the following signature:

```
1  module type LanguageSpecification = sig
2    type 'a term
3    type context
4    type res
5
6    val term_getannot : 'a term -> 'a
7    val term_edit : 'a term -> ('b term) list -> 'b -> 'b term
8    val compute_fv : 'a term -> VarSet.t
9    val compute_hash : 'a term -> int
10   val get_sorted_children : 'a term -> (int * 'a term) list
11
12   val compat : context -> context -> (int * VarSet.t) term -> bool

13   val tr : int -> (int * VarSet.t) term -> (int * VarSet.t) term
         -> context -> res list -> context
14   val checkjoin : (int * VarSet.t) term -> context -> res list ->
         res option

15
16   (* Pretty printing utilities *)
17   val string_of_term : (Format.formatter -> 'a -> unit) -> 'a term
         -> string
18   val string_of_type : res -> string
19   val string_of_context : context -> string
20 end
```

The type declarations `type 'a term`, `type context` and `type res` and functions declared from Lines 12 to 14 define the language and its type system according to Definition 4.2. Lines 6 to 10 are required by the incrementalizer to navigate and modify the AST:

- `term_getannot t` returns the annotation of `t`;

- `term_edit t [a₁; ...; aₙ] a` returns (a new term) `t` annotated with `a`; also, its i-th child is annotated with a_i;
- `compute_fv t` and `compute_hash t` return the set of free variables and a hash of `t`, respectively;
- Finally, the call `get_sorted_children t` returns the list of pairs `[(0, t₀); ...; (n−1, tₙ₋₁)]` with t_i's ordered according to $<$ from Definition 4.2.

The user is required to reify the above signature. For example, the predicate *checkJoin* for the **if-then-else** expression of MinCaml is

```
1 let checkjoin (t : (int * VarSet.t) term) (gamma : context) (rs :
     res list) : res option =
2   match t with
3   (* ... *)
4   | If(_, _, _, _) ->
5       (match (check (List.at rs 0) TBool, check (List.at rs 1)
            (List.at rs 2)) with
6         | (Some _, r) -> r
7         | _ -> None)
8   (* ... *)
```

where `check` takes two types `t1` and `t2` as input and returns `Some t1` if they are equal, `None` otherwise. The function `checkjoin` intuitively returns the type `r` if both branches have the type `r` (i.e., `check (List.at rs 1)(List.at rs 2)`) and if the guard is a boolean (i.e., `check (List.at rs 0)TBool`); otherwise it returns `None`.

Now, let `MyLangSpec` be the provided reified signature. Then the following line of code suffices to obtain the incrementalized type system:

```
module MyIncrementalAlgorithm =
    Incrementalizer.TypeAlgorithm(MyLangSpec)
```

For typing a modified term t_{mod} with the incremental rules just call

```
MyIncrementalAlgorithm.typing cache Γmod tmod
```

where Γ_{mod} is the environment and `cache` is the cache built from the original term `t` through the following:

```
let cache = MyIncrementalAlgorithm.get_empty_cache () in
  MyIncrementalAlgorithm.build_cache t Γ cache
```

Note that the cache is built in place for performance reasons. Also, to keep efficient checking typing context compatibility, we compute the sets of the free variables beforehand, and store them as additional annotations on the aAST.

4.4.2 *A git-versioned ray tracer*

To show the applicability of our methodology to a lifelike case, we experimented with the MinRT ray-tracer[4] of about 1kloc written for the original MinCaml compiler.

First, the implicitly-typed code of MinRT was type-annotated through a simple tool that exploits the frontend of the original MinCaml compiler.[5]

Then, we linked our incremental type checker to the git repository, via the following script (simplified for readability):

4 The original source code is available at https://github.com/esumii/min-caml/tree/master/min-rt.

5 See `main.ml` and `syntax.ml` in https://github.com/matteobusi/min-caml.

```
git show HEAD:minrt.ml > minrt.2.ml
git show HEAD~2:minrt.ml > minrt.1.ml
imc minrt.1.ml minrt.2.ml
```

The first two lines retrieve the last two committed versions of the `minrt.ml` file. The last line invokes the incremental type checker `imc`: first it builds the aAST and the cache for `minrt.1.ml` and then it uses them to type `minrt.2.ml`. For the sake of presentation consider only the *diff* between the two (successive) commits concerning the function `write_rgb`, that replaces:

```
let rec write_rgb (_ : unit) : unit =
  let red = int_of_float rgb.(0) in
  let red = if not (red <= 255) then 255 else red in
  let _ = print_byte red in
  let green = int_of_float rgb.(1) in
  let green = if not (green <= 255) then 255 else green in
  let _ = print_byte green in
  let blue = int_of_float rgb.(2) in
  let blue = if not (blue <= 255) then 255 else blue in
  print_byte blue
```

with

```
let rec write_rgb (_ : unit) : unit =
  let rec cc (v : int) : int = (if v <= 255 then v else 255) in
    print_byte (cc (int_of_float rgb.(0)));
    print_byte (cc (int_of_float rgb.(1)));
    print_byte (cc (int_of_float rgb.(2)))
```

The script above produces the following output

```
Analyzing: Orig: minrt.1.ml ... Mod: minrt.2.ml ...
Lexing ... done
Parsing ... done
Annotating original ... - done (root hash: 355609555)
Annotating modified ... - done (root hash: 994309017)
Initial typing environments ... done
Building the cache ... done
Original typing ... done
Incremental typing ... done
Type: unit - IType: unit
[Visited: 216/3977] O: 10 - H: 99 - M: 0 (I) + 107 (NF) = 107
Printing graphical tree report ... done
```

Lines 1 to 9 include some basic information about the (incremental) typing and its progress. Line 10 shows the result of type checking. Some statistics on the incremental type checking are in Line 11: *(i)* the incremental type checking of `minrt.2.ml` required to visit 216 of *(ii)* the total 3977 nodes; *(iii)* the original typing algorithm was called 10 times (`O`); *(iv)* there were 99 cache hits (`H`) and *(v)* 107 cache misses (`M`). All the misses depend on the absence of the required node in the cache (`NF`) and none on failed compatibility checks (`I`). The last line of the output tells that a graphical report was generated too. This report, in the Graphviz dot format, includes a representation of the aAST of `minrt.2.ml` where hits are highlighted in green, calls to the original typing algorithm are in cyan, and misses due to nodes that are not in cache are highlighted in red (see Figure 40). Furthermore, nodes highlighted in orange are those that caused a miss because of a failed compatibility check (none in Figure 40). Such a graphical representation should be especially useful to visually debug incrementalizations of new type systems.

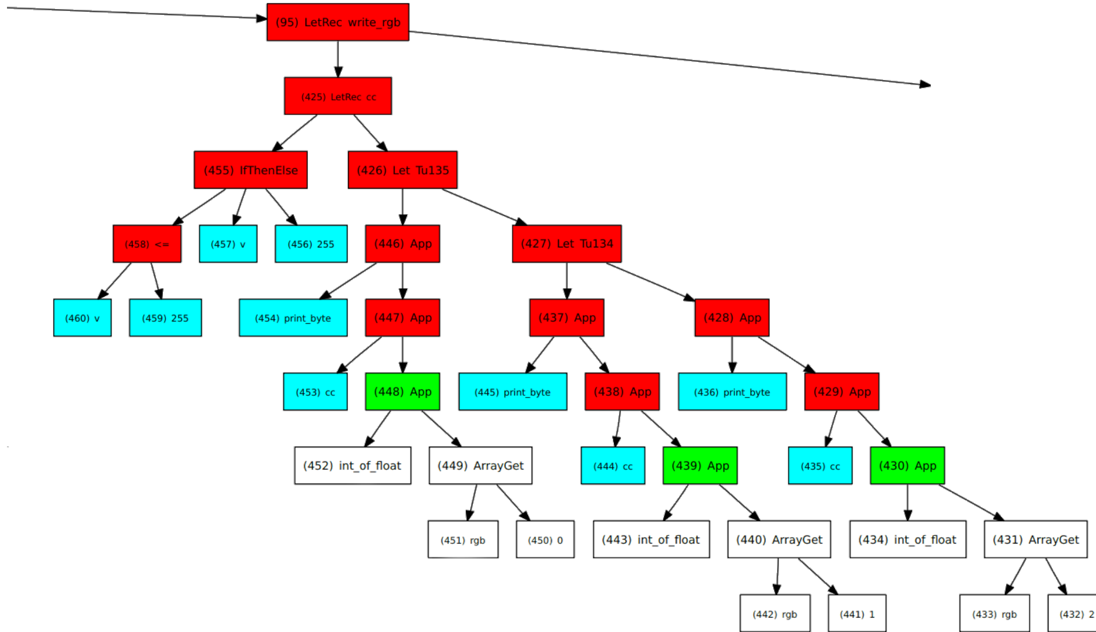


Figure 40: The graphical report as generated by the incremental typechecker for two successive commits of the explicitly typed MinRT. In green the nodes found in the cache; in cyan the nodes whose type was discovered resorting to the original algorithm; in red the nodes not in the cache.

4.4.3 *Experimental evaluation*

We have applied the INCREMENTALIZER to obtain an incremental version of the type checker for MinCaml [219], and we have evaluated its performance.[6] In particular, we made experiments for verifying that (i) the overhead for managing the cache is small; (ii) the time for incrementally checking depends on the size of *diffs*; (iii) the ratio between the time taken by the incremental and the original type checker also decreases with the size of *diffs*; (iv) the memory overhead is negligible.

For that we type checked synthetic programs with (binary and complete) aAST of increasing depth from 8 to 16, and with a number of variables ranging from 1 to 2^{15}. All the internal nodes are binary operators and the leaves are free variables. These test suites are intended to stress our incremental algorithm in the worst, yet artificial case. The measures are obtained using the libraries *Landmarks* and *Core_bench* that however also take into account the OCaml runtime. [7]

Since caches and type environments are implemented as hash-tables, the memory overhead due to the cache is $\mathcal{O}(n \times m)$, where n is the size of the program under analysis and m is the number of its variables. To support the above, we measured the number of MBs allocated by our implementation for the original type checking and by its incremental usage with respect to the size of the synthetic aAST and the number of free variables. Table 2b displays the obtained figures and shows that the space overhead is less than 4%, decreasing with the number of free variables in a program and its size.

6 Available at https://github.com/matteobusi/incremental-mincaml.
7 Available at https://github.com/LexiFi/landmarks and https://github.com/janestreet/core_bench.

Depth	Vars	Original	Incremental	Incremental/Original ratio
16	1	177.87	95681.91	537.94
16	2^7	145.06	4502.24	31.04
16	2^9	143.18	1322.83	9.24
16	2^{11}	139.45	329.90	2.37
16	2^{13}	116.60	69.01	0.59
16	2^{15}	95.33	13.66	0.14

(a) Experimental results about caching in terms of the throughput (i.e., re-checks per second) for the original and the incremental usage of the type checker.

				Original vs. Incremental	
Depth	Vars	Original	Incremental	Ratio	Difference
10	2^9	6.03	6.79	0.8880	0.7602
12	2^{11}	88.20	91.16	0.9675	2.9655
14	2^{13}	1374.05	1385.91	0.9914	11.8610
16	2^{15}	21808.74	21856.18	0.9978	47.4481

(b) Experimental results on the memory overhead in MB.

Table 2: A summary of experimental results.

To test the efficiency of caching we first typed the program with an empty cache to build one. We then typed again the same program with no changes, but starting from the just built cache, so as to compare the difference in time. One reason for time overhead are the inspection and update of caches and environments. We have implemented them as hash-tables to handle them in almost constant time. The other possible time-consuming part concerns checking typing context compatibility. To make also *compat* efficient we compute the sets of the free variables beforehand as mentioned above. Table 2a displays the number of re-typings per second in function of the depth of the aAST and the number of variables in the program. The experiments not only show that the overhead for caching is negligible, but also that caching is beneficial when the number of free variables is not too large with respect to the aAST depth because the results of common subtrees are re-used: just look in the cache for the single, relevant entry. Indeed, when the number of free variables is too large, the cost checking compatibility grows and prevails on the other costs. However, we do not expect this situation to happen very often in practice, since according to our experiments we loose all the advantages of incrementalization only when the number of free variables is at least half the size of the abstract-syntax tree.

Finally, we have simulated program changes by invalidating parts of caches that correspond to the rightmost sub-expression at different depths. Note that invalidating cache entries for the *diff sub-term* t' of t requires to invalidate (i) all the entries for the nodes in the path from the root of the aAST of t to t' and (ii) all the entries for t' and its sub-terms, recursively: therefore the smaller the invalidation depth, the more typing is expected. In this way we simulate that the whole t' is new and never seen before: the bigger t', the more typing is expected. Actually, some sub-trees of t' may appear elsewhere in t. In this case, the number of cache entries invalidated is smaller than the size of t' plus the number of nodes of the path from the root of t to the root of t'.

The plots in Figure 41 show the performance of the incrementalized and of the original type checking algorithm vs. the size of the diff sub-term. The figure displays the throughput for a few choices of the depth of the aASTs (from 12 to 16) and the number of variables (from 2^9 to 2^{15}). The throughput is expressed in terms of the number of aASTs that can be processed per unit of time (i.e., re-typings/s).

The experimental results show that our caching and memoization is in many cases faster than re-typing twice.

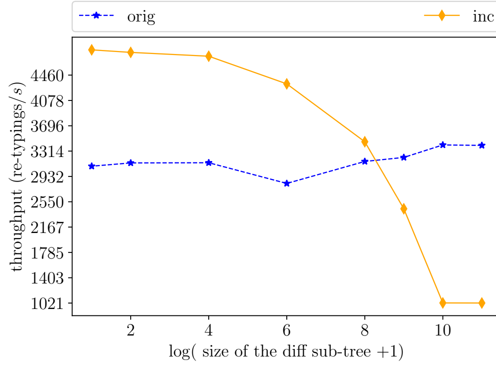
All in all, the advantage of using the incrementalized type checker decreases, as expected, when there is a significant growth of the number of variables or in the size of the diff sub-trees. However, these cases only show up with very big numbers, which are not likely to occur often, especially in the perpetual development model. One of the main reasons is the repeated invocation of the function $compat$. To mitigate this overhead we introduced the possibility of defining a *threshold* on the maximum number of compatibility checks. As an optimization, one can force the incremental algorithm to behave as the original one when the threshold is reached. Dotted, cross-marked plots in Figure 41 report the behavior of the incrementalized type checker on our example for different values of the threshold. As expected, smaller values of the threshold correspond to a delayed degradation of performance due to the size of *diffs*. Note that when the size of the aAST doubles that of the *diffs*, the performance of the incrementalized algorithm with and without threshold coincide since the number of compatibility checks is always 1.

The performance is expected to lower when the code has a lot of inter-dependencies. Indeed, code dependencies are notoriously hard to deal with and they often lead to worst cases in complexity (see e.g., [123]). To experiment on that, we considered programs resulting from unrolling n times the factorial of n:

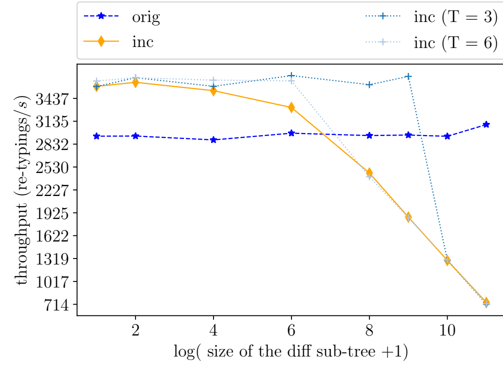
```
1 let x_1 = 1 in
2   ...
3     let x_n = n * x_{n-1} in
4       x_n
```

and we have simulated program changes by invalidating parts of caches that correspond to x_i and its sub-expressions, for different values of i.

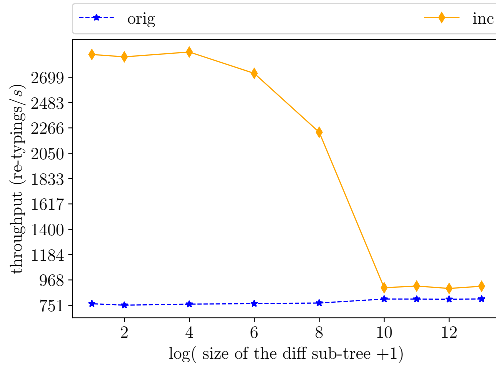
The dashed, star-marked and the solid, diamond-marked plots in Figure 42 report the experimental results for 2559 and 20479 nodes. In this pathological case with a lot of inter-dependencies, both the original and the incrementalized algorithm are much slower than in similarly-sized programs (cfr. Figure 41). The incrementalized algorithm is slightly slower than the original one, because it inspects and updates the cache many times. Yet, the bigger the aAST, the smaller the gap in performance. Actually, the dotted plots in Figure 42 report the results with the optimization discussed above: the smaller the threshold, the closer the incremental algorithm gets to the original one. More plots are available in Appendix A.2.



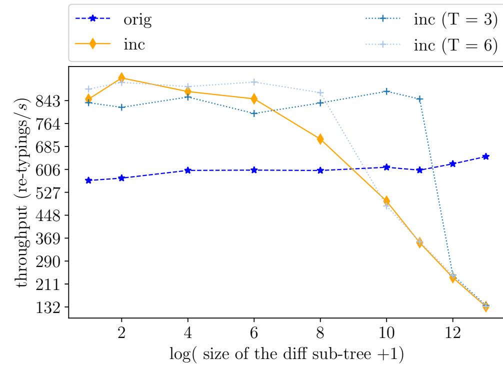
(a) $depth = 12, \#variables = 2^9$



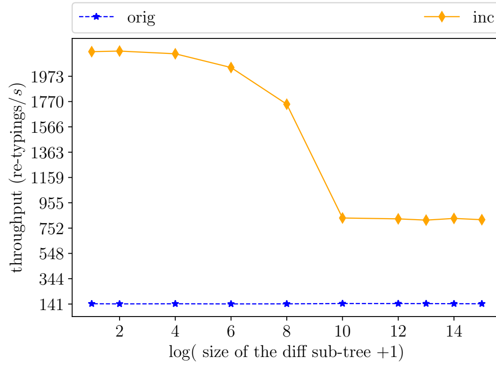
(b) $depth = 12, \#variables = 2^{11}$



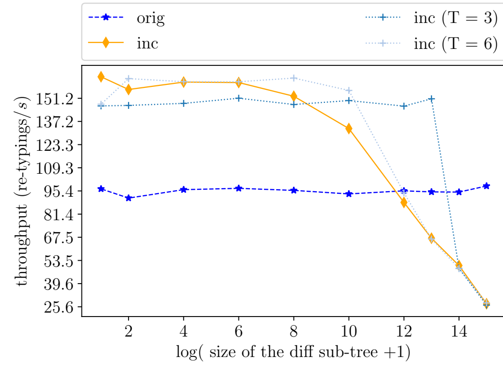
(c) $depth = 14, \#variables = 2^9$



(d) $depth = 14, \#variables = 2^{13}$

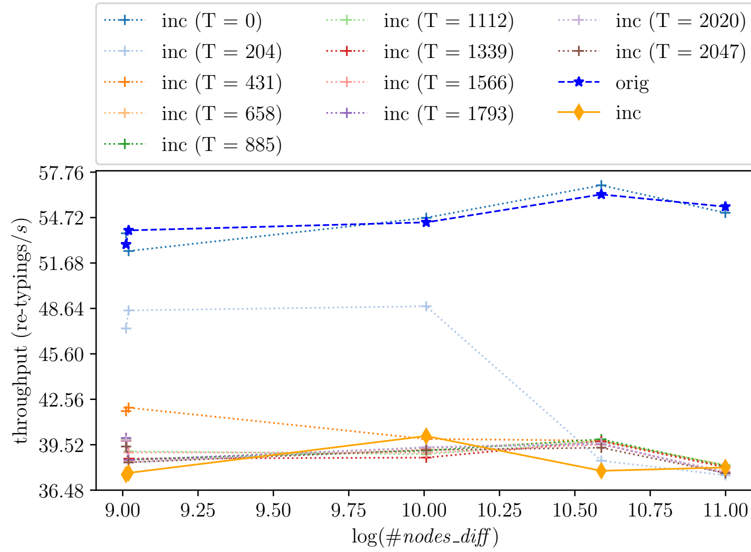


(e) $depth = 16, \#variables = 2^9$

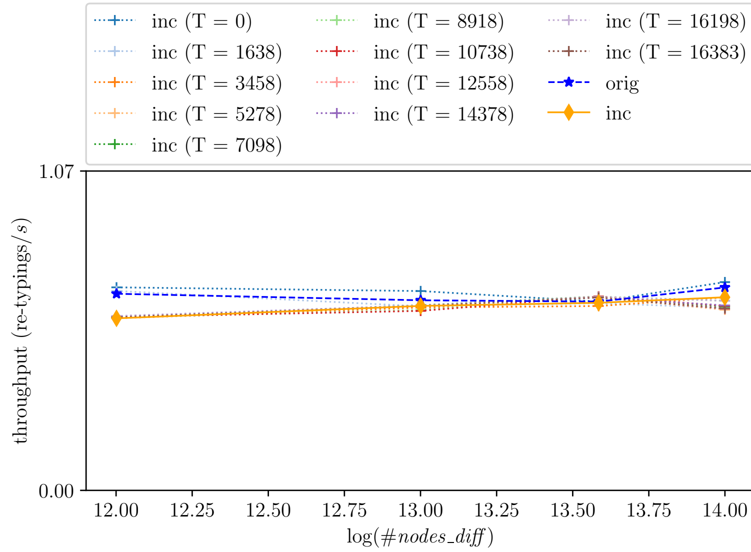


(f) $depth = 16, \#variables = 2^{15}$

Figure 41: Experimental results comparing the number of re-typings per second vs. the number of nodes of the *diff* sub-tree. The blue, dashed plot is for the original type checking, while the orange, solid one is for the incrementalized one. Dotted plots report the behavior of the incrementalized type checker for various values of the threshold T. The x-axis is logarithmic, while the y-axis is scaled as necessary. The plots on the right consider the maximum number of variables.



(a) The aAST considered has 2559 nodes



(b) The aAST considered has 20479 nodes

Figure 42: Experimental results comparing the number of re-typings per second vs. the number of nodes of the *diff* on two unrollings of the factorial function. The dashed, star-marked plot is for the original type checking, the orange, solid one is for the incrementalized one. Dotted plots report the behavior of the incrementalized type checker for various values of the threshold T. The x-axis is logarithmic, while the y-axis is scaled as necessary.

4.5 CONCLUSIONS

In this chapter we presented *incremental type analysis* as a possible way to achieve secure compilation efficiently and automatically. For that, we have presented an algorithmic schema for incrementally using existing type checking and type inference algorithms. Our algorithmic schema is essentially a wrapper that uses the original typing algorithms as gray-boxes. Indeed, assuming that the rules of the type system are given in a specific format (in which most algorithmic presentations fit), only the shape of the input, the output, and some domain-specific knowledge of the original algorithms are relevant. The basic idea of our schema is to re-use as much as possible the available information on the portions of code already typed, which is stored in a cache. Remarkably, the only real effort for defining the incremental algorithm is establishing the notion of compatibility between parts of the environments that are cached and portions that are relevant for re-typing.

Actually, we argue that our approach is mostly mechanizable. Indeed, our recent graduate Federico Pennino developed a parser for automatizing the extraction of the functions tr and $checkJoin$ that specify a typing algorithm, directly from syntax-directed inference rules written in Datalog [184]. We have introduced the basic bricks of our approach and proved three theorems ensuring *coherence*, *completeness* and *cache well-formedness* of *any* incrementalized algorithm. More precisely, these three properties of the incremental algorithm follow from easily checking mild conditions on a notion of typing context compatibility and on the original type system.

To illustrate the approach and show its flexibility, we have then instantiated our proposal on seven typing algorithms for languages in three different programming paradigms (imperative, functional and process calculus). The original type systems are taken from the literature, and cover four different aspects: checking confidentiality and integrity, dependent types, exceptions and protocol security.

We have then implemented an OCaml module that takes as an input two specific functions (namely, tr and $checkJoin$) and the syntax of the language in hand and automatically generates the code of both its original and its incremental type system. As a side result, one can use our module for generating early prototypes of type systems. We have implemented the incremental version of the type checker of MinCaml, and we have assessed it on synthetic programs with varying size and number of variables. The experiments have shown our proposal worth using in situations in which changes between successive re-typings are relatively small w.r.t the size of the program at hand, since only *diffs* are typed, possibly with those parts of the code affected by them. Indeed, e.g., when the size of *diffs* is close to that of the whole program, or when the code has many inter-dependencies, the incrementalized type systems may slow down because of too many compatibility checks. We mitigated this loss in performance by using less cached information when an optional threshold is reached. Additionally, the cost of using the type checker incrementally depends on the size of *diffs*, and its performance increases as these become smaller, a typical situation when applying local transformations, e.g., code-motion, dead code elimination, and code wrapping.

Related work

GENERIC FORMATS FOR TYPING JUDGMENTS A few general formats for specifying programming languages, program analysis tools and type systems exist in the literature, e.g., *PLT Redex* [97] and *K* [203]. However, these proposals are usually much wider in scope than the format we proposed in Definition 4.2 (Page 37). We feel that providing an incrementalization schema based on (restricted versions of) one of these existing formats could foster a wider adoption of our approach but at the same time we think that it would make our framework significantly more complex.

Focussing instead of formats that are specific for type systems, Marino and Millstein [145] propose a very general format with the aim to describe a large class of type and effect systems. More precisely, they provide a set of rule templates to be instantiated to the language at hand by specifying two functions: *adjust* and *check*. Very roughly, the first function works similarly to our tr and the second one is analogous to $checkJoin$. However, differently from us, they are interested in describing and studying the properties of various type (and effect) systems and not in providing a general format useful for making typing algorithms incremental. Also, Cimini and Siek [71] introduce *Gradualizer*, an algorithm that inputs a type system and outputs a gradually typed version of it. For that, they chose λ-prolog [98] to represent the input type system. As said above, a recent graduate of ours followed a similar approach and developed a tool that parses inference rules written in Datalog and automatically extracts a type checker following our format [184].

TYPE-PRESERVING COMPILATION Despite the fact that type-preserving compilation is a special form of secure compilation (see Chapter 2) and although many instances of type-preserving compilers exist (e.g., [69, 134, 212]) only a handful of them have been applied to security problems. One of the most influential examples of type-preserving compilation with applications to security dates back to Morrisett et al. [156] that provide a type-preserving compiler from System F to a typed assembly language (called TAL), via five intermediate steps and four intermediate typed languages. More precisely, let $[\![\cdot]\!]^{\mathsf{I}}_{\mathbf{I}'}$ be one of the intermediate steps, they prove that if a program e has type τ, then the transformed program $[\![\mathsf{e}]\!]^{\mathsf{I}}_{\mathbf{I}'}$ has a type which is *equivalent* to τ. Ahmed and Blume [14] and Bowman and Ahmed [45] respectively show that typed closure-conversion [152] and a translation from DCC [4] to F_ω [185, Chapter 30] are type-preserving and fully abstract, thus proving that they preserve strong security properties. Barthe et al. [22] provide a compiler between two security-typed languages and prove that if the source program satisfies non-interference according to its type system, then its compiled counterpart still satisfies it. Tackling the problem from the point of view of proof-carrying code [162], Barthe et al. [23] provide a compiler that preserves verification conditions as generated on (annotated version of) their source programs at the target. However, to the best of our knowledge, no effort has been made to make the derivation of such type-preserving compilers systematic.

INCREMENTAL ANALYSIS The literature has some proposals for incrementally typing programs. However, these approaches heavily differ from ours, because all of them propose a *new* incremental algorithm for typing, while we *derive* and incremental typing algorithm by wrapping *existing* ones. Additionally, none of the approaches surveyed below use a uniform characterization of type judgments as we do through the metafunctions tr and $checkJoin$.

Meertens [150] proposes an incremental type checking algorithm for the language `B`. Johnson and Walz [120] treat incremental type inference, focussing on identifying where type errors precisely arise. Aditya and Nikhil [12] propose an incremental Hindley/Milner type system supporting incremental type checking of top-level definitions. Our approach instead supports incremental type checking for all kinds of expressions, not only the top-level ones. Miao and Siek [151] introduce an incremental type checker leveraging the fact that, in multi-staged programming, programs are successively refined. Wachsmuth et al. [230] propose a task engine for type checking and name resolution: when a file is modified a task is generated and existing (cached) results are re-used where possible. The proposal by Erdweg et al. [95] and Kuci [130] is the most similar to ours, but, given a type checking algorithm, they describe how to obtain a *new* incremental algorithm. As in our case, they decorate an abstract syntax tree with types and typing environments, represented as sets of constraints, to be suitably propagated when typing. In this way there is no need of dealing with top-down typing context propagation while types flow bottom-up. Recently, Facebook released Pyre [96] a scalable and incremental type checker specifically designed for `Python`. Pacak et al. [173] propose a systematic approach to derive incremental type checkers directly from the inference rules of a type checker. Their idea is to compile the inference rules to the logic programming language Datalog [67], so as to leverage existing efficient and incremental Datalog solvers. Actually, their compilation consists of three steps. The first step translates the given inference rules into a format that it is computable in Datalog. The second step further transforms the rules to enable efficient and incremental type checking (via a specialized form of deforestation of typing contexts [231]). Finally, the last step re-formulates the type checking algorithm to separate error handling from type computation, so avoiding significant re-analyses when checking code changes that fix a type error. Despite sharing our goals, their benchmarking results are not directly comparable with ours since they adopt existing and very efficient Datalog interpreters whereas we use our own prototypical implementation. We are convinced that further research is needed to compare the two approaches, especially concerning their ability to incrementally analyze programs with a lot of internal data dependencies.

Incrementality has also been studied for static analysis other than typing. IncA [220] is a domain-specific language for the definition of incremental program analyses (including typing algorithms), which represents dependencies among the nodes of the abstract syntax tree of the target program as a graph. Infer [118] uses an approach similar to ours in which analysis results are cached to improve performance [36]. Designing incremental data-flow analyzers is a well studied problem and many proposals are based on the technique of the *restarting iteration* [77, 104]. Intuitively, the idea of this technique is to start the fixpoint iteration to solve the data-flow equations from an already computed analysis where the entries corresponding to the changed program points are invalidated. Ryder and Paull [205] present two incremental update algorithms, ACINCB and ACINCF, that allow incremental data-flow analysis. Yur et al. [244] propose an algorithm for an incremental *points-to* analysis. McPeak et al. [149] describe a technique for incremental and parallel static analysis based on *work units* (self-contained atoms of analysis input). The solutions are computed by a sort of processes called *analysis workers*, all coordinated by an *analysis master*. Arzt and Bodden [19] presented Reviser, a tool for incremental interprocedural data-flow analysis based on the IDE/IFDS framework. Given the control-flow graphs of the two program versions, and a previous analysis, Reviser uses a graph-diff algorithm to determine a superset of the changed nodes. Then, the tool recomputes the analysis for them and for those nodes that transitively depend on them. More recently, Nichols et al. [165] presents *fixpoint reuse*, an incremental static analysis technique based on fixpoint

computations for Javascript programs. This technique, given two versions of a program P_o and P_n, and an analysis A_o for P_o, computes a sound approximation A_n of the analysis for P_n. Seidl et al. [210] extend the local generic solver for computing fixpoint solution on which their tool GOBLINT relies to support incremental static analysis of different versions of the same programs.

Also, there are papers that use memoization with a goal similar to the one of our cache, even if they consider different analysis techniques. In particular, Mudduluru and Ramanathan propose, implement, and test an incremental analysis algorithm based on memoization of (equivalent) boolean formulas used to encode paths on programs [157]. Leino and Wüstholz [137] extend the verification machinery of the Dafny language with a cache mechanism to record the results from earlier runs of the verifier. The cache works on the control-flow graph of the program and for each node stores its verification results. Thus, the verification efforts are focused on those parts of the program that were affected by the user's most recent modifications. Although their cache mechanism is similar to ours, they do not provide any formal condition when it is safe re-use cached data. Also, other authors apply memoization techniques to incremental model-checking [133, 241] and incremental symbolic execution [195, 242].

5

SECURE TRANSLATION VALIDATION

In Chapter 4 we presented a first step towards automatization of *secure compilation*. However, extending such an approach to check the preservation of infinite classes of (hyper)properties may be rather impractical, especially when the target and the source language differ. To overcome this limitation, we propose *secure translation validation* (STV) that generalizes the framework of *translation validation* proposed by Pnueli et al. [189] so guaranteeing robust safety property preservation [10]. More precisely, our idea is to automatically detect whether the safety properties of a source program are broken by its compiled version when executed in a given environment (modeled as a *target context*, recall Chapter 2). Indeed, STV can *automatically* detect if a compiler preserves the family of all the safety properties for a specific source program P, under a given target context. STV is carried out at *link time* (i.e., when the compiled program is plugged into its execution context) and we argue that this is the right time. On the one hand, it is *not too early* because one typically wants some security guarantees on a module, e.g., a library, *before* launching a program using it. On the other hand, it is *not too late*, since linking the same program to different contexts may result in different security guarantees.

Example 5.1 (Motivating example)**.** *Consider for instance a functional source language* S *with* I/O primitives*, but none for communication, and a compiler to a target language* $\mathbf{T}$ *that relies on system calls for managing the I/O (on screen, network, etc.). Suppose that a run of the compiler transforms the source program* P

$$\lambda\,\mathsf{i}.\mathbf{if}\ \mathsf{i} \geq 0\,\mathbf{then}\,(\mathbf{display}\,\mathsf{i};\ \mathsf{i})\,\mathbf{else}\,(-1)$$

into the target program $\mathbf{P}$

$$\lambda\,\mathbf{i}.\mathbf{if}\ \mathbf{i} \geq \mathbf{0}\,\mathbf{then}\,(\mathtt{sc_print}\,\mathbf{i};\ \mathbf{i})\,\mathbf{else}\,(-\mathbf{1}).$$

Although intuitively correct, this compilation does not preserve the security property requiring a program to never send a value on the network, which P *enjoys in* any *context, i.e., a program with a single hole. The property still holds when* $\mathbf{P}$ *is plugged into a* non-evil *target context that correctly implements the system calls, however things go wrong when the context is* evil*, i.e., it maliciously implements the system calls. For example, the property is violated when we plug* $\mathbf{P}$ *into* $\mathbf{C}^{\mathbf{evil}}$

$$(\lambda\,\mathbf{i}.\mathbf{let}\ \mathtt{sc_print} = \lambda\,\mathbf{x}.(\mathbf{display}\,\mathbf{x};\ \mathbf{send}\,\mathbf{x})\ \mathbf{in}\,[\cdot]\,\mathbf{i})\ \mathbf{42}.$$

As said above, STV allows to *automatically* decide whether the compiled version of a given source program still enjoys the same safety properties as its source counterpart in a given context of execution.

5.1 OUR PROPOSAL

Recall from Section 2.3.2 that translation validation [189] checks the correctness of the compilation of a given program P, rather than proving the compiler correct for all inputs. Roughly, it works as follows: first, the source and the target languages are endowed with two semantics sharing the same observables; then a suitable simulation is defined between the result of the compilation and the corresponding source program: if such a simulation exists, the compilation is correct; finally, an algorithm effectively computes the required simulation, if any. Remarkably, this algorithm gives a *fully automatic* way of checking the correctness of real-world optimizing compilers [163]. A tempting approach could be to automatically prove also the security of a compiler by showing (the existence of) a (suitable) simulation between the source and the target program. However, the construction of the required simulation, if any, is undecidable when the program at hand is not finite-state [87]. Program analysis comes to our rescue and allows us to devise a mechanical (and approximated) procedure to deal with this problem.

For that, we consider principles guaranteeing the preservation of safety properties, inspired by RSP and RSC (see Definition 2.11 and Definition 2.12, Page 8) proposed by Abate et al. [10]. As said above, RSC considers finite traces produced by the compiled program when plugged in a possibly malicious context: the compiler robustly preserves safety properties *iff* there exists a context in which the source program also produces the same finite traces.

We can effectively check this principle by using STV. Indeed, we can get rid of the universal quantifiers on programs and contexts because STV only considers a *single* program at a time and the check is performed at link time, and proceed as follows. The first key ingredient of STV are two program analyses, one for S and one for **T**. Crucially, we require the analysis for S to produce an *under-approximation* of the behavior of the program under analysis, while that of **T** has to produce an *over-approximation*. Under-approximation at the source and over-approximation at the target are indeed fundamental (see Theorems 5.4 and 5.5) to guarantee the absence of false positives (i.e., target programs classified as secure when they are not). Of course, false negatives may still be produced and we may fail to prove a compilation secure because of the over- and under-approximations of the behavior of programs. For instance, in Section 5.4 we consider the source analysis to be program testing, whereas we instantiate target analysis as a type and effect system [166] that infers *history expressions* [32]. A history expression is a (finite-state) process of a basic process algebra [34], whose actions are, in our case, the observables of the *trace semantics* of the source language, e.g., in the code of P in Example 5.1 the observable of the primitive **display** will be **display** itself.

With our analyses in hand, at link time we plug the compiled program $\mathbf{P} = [\![\mathsf{P}]\!]^{\mathsf{S}}_{\mathbf{T}}$ into the target context **C**, obtaining **C**[**P**]. Once the over-approximated behavior of **C**[**P**] is computed, we verify if the compilation process broke some of its properties of interest. To this aim, starting from **C** we build a source context C witnessing the presence of all the behaviors of **C**[**P**] at the source level, i.e., we check $beh(\mathcal{W}) \subseteq beh(\mathcal{W})$, where $beh(\mathcal{W})$ denotes the over-approximated behavior of **C**[**P**] and $beh(\mathcal{W})$ the under-approximated one associated with C[P]. Note that, to keep this procedure effective, the inclusion of behaviors $\subseteq$ must be decidable, as it is the case for example with traces generated by testing and our instantiation of history expressions.

In summary, in this chapter we

- Introduce STV as a mean for achieving secure compilation with active attackers automatically;

- Define a notion of STV and link it with robust safety property preservation from [10];
- Give two abstract variants of STV, that are more amenable to verification;
- Define our STV algorithm and prove that it entails our notions of abstract STV; and
- Apply STV to a simple use case, whose languages are inspired from Protzenko et al. [193].

The rest of the chapter is organized as follows. Section 5.2 lays the foundations of STV. A concrete use case of our framework is shown in Section 5.4. For that, we formally introduce a source and a target language, their analyses, and show a concrete usage of STV on a simple example. Section 5.5 discusses how to lift some assumptions and limitations of our STV framework and of the presented use case. Finally, Section 5.6 concludes the chapter.

The whole development of Section 5.2 has been mechanized using the Coq proof assistant and is available online at https://github.com/matteobusi/stv/tree/phdthesis.

5.2 SECURE TRANSLATION VALIDATION

In this section we lay the foundations for our *secure translation validation* (STV) approach, making formal the intuitions of the previous two sections.[1]

Consider again the scenario we briefly illustrated at the beginning of the chapter. Suppose to have a compiled program which is about to be linked (e.g., by a static linker or a program loader) with a given context (e.g., its execution environment). We want to make sure that it is safe to execute this program, by showing automatically that any attack that can be carried out by the context in the target can also be carried out by some context in the source.

For our purposes, a (source or target) programming language is defined as follows

Definition 5.1 (Programming language 🐓). *A programming language (or simply, a language) L is a 5-tuple $(Whole, Partial, Ctx, \cdot[\cdot], \cdot \xrightarrow{\cdot} \cdot)$ with:*

1. *$Whole$ is the set of whole programs;*
2. *$Partial$ is the set of partial programs;*
3. *Ctx is the set of contexts;*
4. *$\cdot[\cdot] : Ctx \to Partial \to Whole$ is the linking operator;*
5. *$\cdot \xrightarrow{\cdot} \cdot : Whole \to \Sigma \to Whole$ is a labelled transition* function, *where Σ is the set of observables as in Chapter 2. Also, we denote with $\cdot \xrightarrow{\cdot}^* \cdot$ its reflexive and transitive closure.*

Intuitively, the linking operator completes a partial program with the needed information from the context and produce a whole program, while the transition function describes the execution of whole programs and their observable behavior. Also, we specialize the notion of behavior of a whole program W as the set of all the prefixes of its traces, i.e., $beh(W) \triangleq \{t \in \Psi_{fin} \mid \exists W'.\ W \xrightarrow{t}^* W'\}$. Furthermore, from now onward we will assume to have a source language S and a target language $\mathbf{T}$, along with a compiler $[\![\cdot]\!]$ from S to $\mathbf{T}$ (Definition 2.7).

[1] All the proofs of this section are mechanized in Coq and available online at https://github.com/matteobusi/stv/tree/phdthesis. Links with the symbol 🐓 in the pdf lead to relevant theorems and definitions in the repository.

We are now ready to formalize our notion of STV in the case of safety properties. The notion of STV_{RSP} states when a target program can be executed securely:

Definition 5.2 (STV robust safety property preservation 🐓). *A target program* $\mathbf{P}$ *can be executed securely in* $\mathbf{C}$ *w.r.t. a source program* P *(written* $\mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} STV_{RSP}$*)* iff

$$\forall \pi \in \mathit{Safety}.\ (\forall \mathsf{C}.\ \mathsf{C}[\mathsf{P}] \vDash \pi) \Rightarrow \mathbf{C}[\mathbf{P}] \vDash \pi.$$

Intuitively, $\mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} STV_{RSP}$ holds whenever the target program $\mathbf{P}$ satisfies π under the *given* target context, provided that P does in *any* source context. Perhaps surprisingly, the program $\mathbf{P}$ is *not* required to be the compiled version of P. This is done on purpose, since imposing $\mathbf{P} = [\![\mathsf{P}]\!]$ for a given $[\![\cdot]\!]$ would allow to use the information about $[\![\cdot]\!]$ to prove STV_{RSP}. We do not want this in the above definition, since we want to consider the compiler as a black-box and we want it to be outside the *trusted computing base*.

The following theorem guarantees that our definition of STV_{RSP} corresponds to that of RSP [10] (see Definition 2.11):

Theorem 5.1 ($STV_{RSP} \Leftrightarrow RSP$ 🐓). *Let* $[\![\cdot]\!]$ *be a compiler from* S *to* $\mathbf{T}$.

$$\left(\forall \mathbf{C}, \mathsf{P}.\ [\![\mathsf{P}]\!] \vdash^{\mathbf{C}}_{\mathsf{P}} STV_{RSP}\right) \Leftrightarrow [\![\cdot]\!] \in RSP.$$

Similarly to the principles by Abate et al. [10], we have the following *criterion* that does not explicitly mention the class of all the safety properties:

Definition 5.3 (STV robustly safe compiler 🐓). *A target program* $\mathbf{P}$ *satisfies the* STV criterion for safety properties *in* $\mathbf{C}$ *w.r.t.* P *(written* $\mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} STV_{RSC}$*)* iff

$$\forall t.\ t \in \mathit{beh}(\mathbf{C}[\mathbf{P}]) \Rightarrow \forall m \leq t.\ \left(\exists \mathsf{C}, t'.\ t' \in \mathit{beh}(\mathsf{C}[\mathsf{P}]) \wedge m \leq t'\right).$$

The criterion we just stated turns out to be equivalent to Definition 5.2:

Theorem 5.2 ($STV_{RSP} \Leftrightarrow STV_{RSC}$ 🐓). *For any target program* $\mathbf{P}$*, target context* $\mathbf{C}$ *and source program* P*:*

$$\mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} STV_{RSP} \Leftrightarrow \mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} STV_{RSC}.$$

Also, analogously to STV_{RSP} and RSP (Theorem 5.1) there is a correspondence between STV_{RSC} and RSC (Definition 2.12):

Theorem 5.3 ($STV_{RSC} \Leftrightarrow RSC$ 🐓). *Let* $[\![\cdot]\!]$ *be a compiler from* S *to* $\mathbf{T}$.

$$\left(\forall \mathbf{C}, \mathsf{P}.\ [\![\mathsf{P}]\!] \vdash^{\mathbf{C}}_{\mathsf{P}} STV_{RSC}\right) \Leftrightarrow [\![\cdot]\!] \in RSC.$$

5.3 EFFECTIVE SECURE TRANSLATION VALIDATION

The definitions from Section 5.2 easily follow from the robust secure compilation principles and criteria of Abate et al. [10]. However, they are hardly practical in a context in which one would like to achieve automation since in most cases the set of all traces is difficult or even impossible to extract. To overcome this issue, we introduce three variants of the notion of *abstract* STV and put forward a few conditions for their applicability.

Our notions of abstract STV rely on a notion of *analysis*, which roughly is a mapping between a language and an *abstract* version of it:

Definition 5.4 (Analysis 🐓). *An* analysis $(\!|\cdot|\!)^{L}_{\mathfrak{L}}$ *from a language* L *(called concrete language) to the language* $\mathfrak{L}$ *(called abstract language) is a triple* $(\alpha_{Partial}, \alpha_{Ctx}, \alpha_{Whole})$ *where*

- $\alpha_{Partial}$ *maps partial programs of* L *in partial programs of* $\mathfrak{L}$*;*
- α_{Ctx} *maps contexts of* L *in contexts of* $\mathfrak{L}$*; and*
- α_{Whole} *maps whole programs of* L *in whole programs of* $\mathfrak{L}$*.*

From now onward we will abuse the notation and feel free to use $(\!|\cdot|\!)^{L}_{\mathfrak{L}}$ in place of its components when analyzing partial programs, contexts, or whole programs.

Of course, not all analyses are adequate for our purposes, e.g., think of an analysis mapping any program or context into an empty program. Therefore, we will only use analyses that enjoy a (non-empty combination of) the following three properties: *soundness*, *completeness*, and *modularity*. Intuitively, we say that an analysis is *sound* if the behavior of the *abstract program* (i.e., the one resulting from the analysis) *includes* that of the original one. Dually, *complete analyses* are such that the behavior of the abstract program *is included* in that of the original one. Finally, an analysis is *modular* if the abstracted version of $C[P]$ behaves the same way as the results of the analysis of C and P computed separately and then linked together. Formally:

Definition 5.5 (Sound 🐓, complete 🐓, and modular 🐓 analyses). *An analysis* $(\!|\cdot|\!)^{L}_{\mathfrak{L}}$ *is*

- Sound iff $\forall C, P.\ beh(C[P]) \subseteq beh((\!|C[P]|\!)^{L}_{\mathfrak{L}})$;
- Complete iff $\forall C, P.\ beh(C[P]) \supseteq beh((\!|C[P]|\!)^{L}_{\mathfrak{L}})$;
- Modular iff $\forall C, P.\ beh((\!|C[P]|\!)^{L}_{\mathfrak{L}}) = beh((\!|C|\!)^{L}_{\mathfrak{L}}[(\!|P|\!)^{L}_{\mathfrak{L}}])$.

From now onward we will assume two analyses: $(\!|\cdot|\!)^{\mathsf{S}}_{\mathfrak{S}}$ from S to $\mathfrak{S}$, and $(\!|\cdot|\!)^{\mathbf{T}}_{\mathfrak{T}}$ from $\mathbf{T}$ to $\mathfrak{T}$. With the above notions in hand, one might be tempted to define the abstract criterion as follows:

Definition 5.6 ((Tentative) $aSTV_{RSC}$ 🐓). *A target program* $\mathbf{P}$ *satisfies the* abstract STV criterion for safety properties *in* $\mathbf{C}$ *w.r.t.* P *(written* $\mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} aSTV_{RSC}$*)* iff

$$\forall t.\ t \in beh((\!|\mathbf{C}|\!)^{\mathbf{T}}_{\mathfrak{T}}[(\!|\mathbf{P}|\!)^{\mathbf{T}}_{\mathfrak{T}}]) \Rightarrow \left(\exists\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}.\ t \in beh((\!|\langle\!\langle \mathbf{t}, \mathbf{C}\rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}|\!)^{\mathsf{S}}_{\mathfrak{S}}[(\!|\mathsf{P}|\!)^{\mathsf{S}}_{\mathfrak{S}}])\right)$$

where $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}$ *is a backtranslation function (see* Chapter 2 *and [10, 177, 180]) mapping traces and contexts of* $\mathbf{T}$ *into contexts of* S*.*

Intuitively, this criterion requires that any attack that can be carried out at the abstract target level must have a corresponding attack at the abstract source level. Indeed, this abstract notion of STV arises naturally from the above definitions, and it can be easily proved to imply Definition 5.3 and thus Definition 5.2:

Theorem 5.4 (Definition 5.6 $\Rightarrow$ STV_{RSC} 🐓). *Assume* $(\!|\cdot|\!)^{\mathsf{S}}_{\mathfrak{S}}$ *to be complete and modular, and* $(\!|\cdot|\!)^{\mathbf{T}}_{\mathfrak{T}}$ *to be sound and modular. Then for any* $\mathbf{C}$, $\mathbf{P}$, P*:*

$$\mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} aSTV_{RSC} \Rightarrow \mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} STV_{RSC}.$$

Corollary 5.1 (Definition 5.6 $\Rightarrow$ STV_{RSP} 🐓). *Assume* $(\!|\cdot|\!)^{\mathsf{S}}_{\mathfrak{S}}$ *to be complete and modular, and* $(\!|\cdot|\!)^{\mathbf{T}}_{\mathfrak{T}}$ *to be sound and modular. Then for any* $\mathbf{C}$, $\mathbf{P}$, P*:*

$$\mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} aSTV_{RSC} \Rightarrow \mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} STV_{RSP}.$$

However, the hypotheses for the above theorems are quite strong since the source analysis is required to be modular, complete, and to produce abstractions for source contexts. Consider for instance the set of program traces as the abstract source language, and program testing as the source analysis. Despite being complete (according to Definition 5.5), program testing usually lacks the ability to abstract (i.e., execute) source contexts, and consequently it lacks the modularity property. Fortunately, we can remove the requirement about the modularity of the source analysis by slightly changing Definition 5.6:

Definition 5.7 ($aSTV_{RSC}$ 🐓). *A target program* $\mathbf{P}$ *satisfies the* abstract STV criterion for safety properties *in* $\mathbf{C}$ *w.r.t.* P *(written* $\mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} aSTV_{RSC}$*)* iff

$$\forall t.\ t \in beh(\llparenthesis \mathbf{C} \rrparenthesis^{\mathbf{T}}_{\mathfrak{T}}[\llparenthesis \mathbf{P} \rrparenthesis^{\mathbf{T}}_{\mathfrak{T}}]) \Rightarrow \left(\exists \langle\!\langle \cdot \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}.\ t \in beh(\llparenthesis \langle\!\langle \mathbf{t}, \mathbf{C} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}[\mathsf{P}] \rrparenthesis^{\mathsf{S}}_{\mathcal{S}})\right)$$

where $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}$ *is a backtranslation function [10, 177, 180] mapping traces and contexts of* $\mathbf{T}$ *into contexts of* S*.*

Note that this new definition requires the source program to be available at *linking time*, which is the case in our scenario. As above, we can easily prove that this abstract notion of STV implies Definition 5.3 and thus Definition 5.2:

Theorem 5.5 ($aSTV_{RSC} \Rightarrow STV_{RSC}$ 🐓). *Assume* $\llparenthesis \cdot \rrparenthesis^{\mathsf{S}}_{\mathcal{S}}$ *to be complete, and* $\llparenthesis \cdot \rrparenthesis^{\mathbf{T}}_{\mathfrak{T}}$ *to be sound and modular. Then for any* $\mathbf{C}$, $\mathbf{P}$, P*:*

$$\mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} aSTV_{RSC} \Rightarrow \mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} STV_{RSC}.$$

Corollary 5.2 ($aSTV_{RSC} \Rightarrow STV_{RSP}$ 🐓). *Assume* $\llparenthesis \cdot \rrparenthesis^{\mathsf{S}}_{\mathcal{S}}$ *to be complete, and* $\llparenthesis \cdot \rrparenthesis^{\mathbf{T}}_{\mathfrak{T}}$ *to be sound and modular. Then for any* $\mathbf{C}$, $\mathbf{P}$, P*:*

$$\mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} aSTV_{RSC} \Rightarrow \mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} STV_{RSP}.$$

One could do the same for the target analysis. However, in our scenario it is highly unlikely that the target context and the target program can be analyzed together at linking time. Also, using the above definition of $aSTV_{RSC}$ can be quite demanding. Indeed, to build the wanted source context, the backtranslation function may depend on the information about a specific trace in the behavior of the abstracted target program. If such a dependency is not acceptable (e.g., for performance reasons) we can change the definition above as follows:

Definition 5.8 ($aSTV_{TI\text{-}RSC}$ 🐓). *A target program* $\mathbf{P}$ *satisfies the* abstract trace-independent STV criterion for safety properties *in* $\mathbf{C}$ *w.r.t.* P *(written* $\mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} aSTV_{TI\text{-}RSC}$*)* iff

$$\exists \langle\!\langle \cdot \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}.\ \forall t.\ t \in beh(\llparenthesis \mathbf{C} \rrparenthesis^{\mathbf{T}}_{\mathfrak{T}}[\llparenthesis \mathbf{P} \rrparenthesis^{\mathbf{T}}_{\mathfrak{T}}]) \Rightarrow t \in beh(\llparenthesis \langle\!\langle \mathbf{C} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}[\mathsf{P}] \rrparenthesis^{\mathsf{S}}_{\mathcal{S}})$$

where $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}$ *is now a function mapping contexts of* $\mathbf{T}$ *into contexts of* S*.*

Crucially, this new notion implies the old one (and thus STV_{RSP}, if $\llparenthesis \cdot \rrparenthesis^{\mathsf{S}}_{\mathcal{S}}$ is modular and sound, and $\llparenthesis \cdot \rrparenthesis^{\mathbf{T}}_{\mathfrak{T}}$ is complete):

Theorem 5.6 ($aSTV_{TI\text{-}RSC} \Rightarrow aSTV_{RSC}$ 🐓). *For any* $\mathbf{C}$, $\mathbf{P}$, P*:*

$$\mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} aSTV_{TI\text{-}RSC} \Rightarrow \mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} aSTV_{RSC}.$$

Algorithm 1 Pseudo-code for the STV algorithm 🐓.

function SafeToRun($\mathbf{C} \in \mathbf{T}_{\mathbf{Ctx}}, \mathsf{P} \in \mathsf{S}_{\mathsf{Par}}, \mathbf{P} \in \mathbf{T}_{\mathbf{Par}}$)
  Assume that $\subseteq$ on abstract behaviors is computable
  Assume $(\!|\cdot|\!)^{\mathsf{S}}_{\mathcal{S}}$ to be an analysis from S to $\mathcal{S}$
  Assume $(\!|\cdot|\!)^{\mathbf{T}}_{\mathcal{T}}$ to be an analysis from $\mathbf{T}$ to $\mathcal{T}$
  Let $\flat : \mathbf{T}_{\mathbf{Ctx}} \rightharpoonup \mathsf{S}_{\mathsf{Ctx}}$ be a backtranslation function

  if $\flat(\mathbf{C})$ is undefined **then**
    return (MAYBE_UNSAFE, $\bot$)
  else
    if $beh((\!|\mathbf{C}|\!)^{\mathbf{T}}_{\mathcal{T}}[(\!|\mathbf{P}|\!)^{\mathbf{T}}_{\mathcal{T}}]) \subseteq beh((\!|(\flat(\mathbf{C}))[\mathsf{P}]|\!)^{\mathsf{S}}_{\mathcal{S}})$ **then**
      return (SAFE, $\bot$)
    else
      return (MAYBE_UNSAFE, $\flat(\mathbf{C})$)
    end if
  end if
end function

Building on Definition 5.8, we finally put everything together in Algorithm 1. Assume inclusion between behaviors of programs in $\mathcal{S}$ and $\mathcal{T}$ to be computable, and let $\flat : \mathbf{T}_{\mathbf{Ctx}} \rightharpoonup \mathsf{S}_{\mathsf{Ctx}}$ be a *partial* function mapping target contexts into source ones (intuitively, a partial backtranslation). The algorithm is now rather straightforward. If the function $\flat$ is defined on $\mathbf{C}$, it then checks the conditions of Definition 5.8 and if they hold it returns (SAFE, $\bot$). When these conditions do not hold, the algorithm returns the pair (MAYBE_UNSAFE, $\flat(\mathbf{C})$) meaning that it *may be* unsafe to link the target program in hand with $\mathbf{C}$; it also gives $\flat(\mathbf{C})$ as a possible source-level attacker of P. Finally, if $\flat$ is undefined on $\mathbf{C}$ the same happens, but no counterexample is exhibited. The following theorem proves that if Algorithm 1 returns (SAFE, $\bot$), then it is safe to run the program according to the definition of $aSTV_{TI\text{-}RSC}$:

Theorem 5.7 ((SAFE, $\bot$) $\Rightarrow$ $aSTV_{TI\text{-}RSC}$ 🐓)**.** *Let $\mathbf{P}$ be a partial target program, $\mathbf{C}$ be a target context and P be a source program. If* SafeToRun($\mathbf{C}, \mathsf{P}, \mathbf{P}$) = (SAFE, $\bot$) *then* $\mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} aSTV_{TI\text{-}RSC}$.

Recall now our usage scenario. Let P be the program before compilation and $\mathbf{P} = [\![\mathsf{P}]\!]^{\mathsf{S}}_{\mathbf{T}}$ be the partial target program that we want to execute in the target context $\mathbf{C}$. The following corollary allows us to use Algorithm 1 to assess whether it is secure to run $\mathbf{P}$ in $\mathbf{C}$:

Corollary 5.3 (Algorithm 1 is correct 🐓)**.** *Assume $(\!|\cdot|\!)^{\mathsf{S}}_{\mathcal{S}}$ to be complete, and $(\!|\cdot|\!)^{\mathbf{T}}_{\mathcal{T}}$ to be sound and modular. If* SafeToRun($\mathbf{C}, \mathsf{P}, \mathbf{P}$) = (SAFE, $\bot$) *then* $\mathbf{P} \vdash^{\mathbf{C}}_{\mathsf{P}} STV_{RSP}$.

5.4 A USE CASE FOR SECURE TRANSLATION VALIDATION

This section illustrates a simple use case for our STV framework. We focus here on source and target languages that simplify λow^* and C^* from Protzenko et al. [193] by removing heap-related constructs. Although simple enough to keep the presentation manageable, we remark that these two languages are used in practice [193].

The source language μow^* is a simple first-order functional language. The contexts of μow^* are sequences of definitions introducing environment variables and functions, ending with a hole. Partial programs are expressions (as found in many functional languages), with the additional ability to invoke I/O primitives in the set $Prim$ (e.g., for writing on screen). Whole programs are obtained by filling the hole of a context with a partial program.

The target language $\mu\mathbf{C}^*$ is instead an imperative language. The definitions of contexts and whole programs are very similar to the ones of μow^*, partial programs are instead lists of statements. The special syntactic construct `call` is used to invoke system calls taken from a given set $SCName$. Also, we assume that $SCName \supseteq Prim$. Note that this inclusion is reasonable: it often happens that source, high-level languages forbid access to some system functionalities that are available to lower-level languages, e.g., for portability reasons.

Both μow^* and $\mu\mathbf{C}^*$ are deterministic languages, whose observable behavior is defined as the sequence of system calls performed by a program during its execution. Formally, a trace t is a sequence of names taken from the set $\Sigma = SCName \supseteq Prim$. Also, let ϵ be empty sequence, that is the neutral element of the sequence concatenation operator $\cdot$ (i.e., $\epsilon \cdot t = t \cdot \epsilon = t$).

A said above, to make STV practical we also need two abstract languages and two analyses mapping programs from μow^* and $\mu\mathbf{C}^*$ to their abstract counterparts.

The abstract source language is called $\mu\mathfrak{ow}^*$. Since we are going to use $aSTV_{TI\text{-}RSC}$, we slightly abuse Definition 5.1 and just define the set of whole programs of $\mu\mathfrak{ow}^*$ (which actually coincides with the set of sets of finite traces). The analysis from μow^* to $\mu\mathfrak{ow}^*$ is $(\!|\cdot|\!)^{\mu ow^*}_{\mu\mathfrak{ow}^*}$ and it maps a whole source program to the set of the finite prefixes of its behavior.

The abstract target language $\mu\mathscr{C}^*$ is instead that of history expressions (see [32] for further details), with observables over the set $\Sigma = SCName$. The analysis is in this case a *type and effect system* inferring histories directly from programs and contexts.

5.4.1 *The concrete and abstract source languages and their analysis*

In the following, we present the source language μow^*, its abstract counterpart $\mu\mathfrak{ow}^*$, and the (complete) analysis relating them.

THE SOURCE LANGUAGE: μow^* As said above μow^* is a simply-typed, first-order functional language inspired by the λow^* language from [193]. The only relevant difference with λow^* is that variables in their language can be allocated on the heap. Let $Prim$ be the set of the names of available language primitives, included in the set of the observables. For the sake of simplicity, we model primitives as opaque procedures with no return value. Also, since the type system is standard, we omit it and assume that all programs

are well-typed. The definition of $\mu ow^* = (\mathsf{Whole}_{\mu ow^*}, \mathsf{Partial}_{\mu ow^*}, \mathsf{Ctx}_{\mu ow^*}, \cdot[\cdot], \cdot \xrightarrow{\cdot}_{\mu ow^*} \cdot)$ is rather unsurprising. Its syntax is defined as follows:

$$f_p \in \mathit{Prim} \quad x, f \in \mathit{Name} \quad f_c \in \mathit{CName}$$
$$\mathit{Prim} \cap \mathit{Name} = \emptyset \quad \mathit{Prim} \cap \mathit{CName} = \emptyset \quad \mathit{Name} \cap \mathit{CName} = \emptyset$$
$$v \in \mathit{Val} = \mathbb{Z} \cup \{\,()\,\} \quad \mathsf{op} \in \{+, *, -\} \quad \tau, \tau_1 \in \{\mathtt{int}, \mathtt{unit}\}$$
$$\begin{aligned}
\mathsf{Partial}_{\mu ow^*} \ni \mathsf{P} \quad &::= \mathbf{let}\ x : \tau = v\ \mathbf{in}\ \mathsf{P} \mid \mathbf{let}\ f = \lambda\, x : \tau.\mathsf{e} : \tau_1\ \mathbf{in}\ \mathsf{P} \mid \mathsf{e} \\
\mathsf{e} \quad &::= v \mid x \mid \mathsf{e}_1\ \mathsf{op}\ \mathsf{e}_2 \mid \mathbf{let}\ x : \tau = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2 \mid \mathbf{let}\ x : \tau = f\ \mathsf{e}\ \mathbf{in}\ \mathsf{e}_2 \mid \\
&\quad\ \mathbf{let}\ x : \tau = f_c\ \mathsf{e}\ \mathbf{in}\ \mathsf{e}_2 \mid \mathbf{let}\ x : \mathtt{unit} = f_p\ \mathsf{e}\ \mathbf{in}\ \mathsf{e}_2 \mid \\
&\quad\ \mathbf{if}\ \mathsf{e}_1\ \mathbf{then}\ \mathsf{e}_2\ \mathbf{else}\ \mathsf{e}_3 \\
\mathsf{Ctx}_{\mu ow^*} \ni \mathsf{C} \quad &::= \mathbf{let}\ f_c = \lambda\, x : \tau.\mathsf{e} : \tau_1\ \mathbf{in}\ \mathsf{C} \mid [\cdot]
\end{aligned}$$

Given a context and a program, we can easily define the linking operator:

$$(\mathbf{let}\ f_c = \lambda\, x : \tau_1.\mathsf{e} : \tau_2\ \mathbf{in}\ \mathsf{C})[\mathsf{P}] = \mathbf{let}\ f_c = \lambda\, x : \tau_1.\mathsf{e} : \tau_2\ \mathbf{in}\ \mathsf{C}[\mathsf{P}]$$
$$([\cdot])[\mathsf{P}] = \mathsf{P}.$$

The set of whole programs, $\mathsf{Whole}_{\mu ow^*}$, is then the set of programs generated by the linking operator above. The rules defining the call-by-value operational semantics for whole programs are in Figure 43, where $\{\cdot/\cdot\}$ is the usual capture-avoiding substitution operator. Most of the rules are unsurprising, the only interesting cases being those for the invocation of primitives. When a program calls a primitive, it first reduces its parameter to a value (rule (μow^*-PRIM0)) and then it invokes the actual primitive f_p, emitting f_p as the observable (rule (μow^*-PRIM1)).

THE ABSTRACT SOURCE LANGUAGE: $\mu\mathcal{OW}^*$ Now that our source language μow^* is fully defined we can introduce its abstract counterpart $\mu\mathcal{OW}^*$. As said above, we abuse Definition 5.1 and just define whole programs of $\mu\mathcal{OW}^*$. Since our analysis is program testing (see below), we define $\mathsf{Whole}_{\mu\mathcal{OW}^*}$ as the set of all finite program traces, formally:

$$\mathsf{Whole}_{\mu\mathcal{OW}^*} = \wp(\mathit{Prim}^*).$$

The semantics of $\mu\mathcal{OW}^*$ is as follows (where $\setminus$ is the usual set difference operation)

$$\frac{t \in \mathcal{W}}{\mathcal{W} \xrightarrow{t}_{\mu\mathcal{OW}^*} \mathcal{W} \setminus \{t\}}$$

THE ANALYSIS We are now ready to define the analysis $(\!|\cdot|\!)^{\mu ow^*}_{\mu\mathcal{OW}^*}$ from μow^* to $\mu\mathcal{OW}^*$. As said above, $(\!|\cdot|\!)^{\mu ow^*}_{\mu\mathcal{OW}^*}$ is defined as program testing on whole programs of μow^*, i.e.,

$$(\!|\mathsf{W}|\!)^{\mu ow^*}_{\mu\mathcal{OW}^*} = \{t \mid \mathsf{W} \xrightarrow{t}{}^{*}_{\mu ow^*} \mathsf{W}'\}.$$

Note that the above set is always finite because all μow^* programs are total: the language has no recursion, no recursive types and no higher-order state. To see our definitions at work, consider the following simple example.

Example 5.2. *Assume a primitive* `display` *that shows a given integer on the screen and let* P *be the following partial* μow^* *program:*

$$\begin{aligned}
&\mathbf{let}\ x : \mathtt{int} = 3\ \mathbf{in} \\
&\mathbf{let}\ dx : \mathtt{int} = d\, x\ \mathbf{in} \\
&\quad \mathbf{if}\ dx\ \mathbf{then}\ (\mathtt{display}\ x)\ \mathbf{else}\ (\mathtt{display}\ 42)
\end{aligned}$$

(μow^*-LetVar0)

$$\frac{e_1 \xrightarrow{o}_{\mu ow^*} e_1'}{\textbf{let } x : \tau = e_1 \textbf{ in } e_2 \xrightarrow{o}_{\mu ow^*} \textbf{let } x : \tau = e_1' \textbf{ in } e_2}$$

(μow^*-LetVar1)

$$\frac{}{\textbf{let } x : \tau = v \textbf{ in } e_2 \xrightarrow{\epsilon}_{\mu ow^*} e_2\{v/x\}}$$

(μow^*-LetAbs)

$$\frac{f \in \mathit{Name} \cup \mathit{CName}}{\textbf{let } f = \lambda\, x : \tau.e_1 : \tau_1 \textbf{ in } e_2 \xrightarrow{\epsilon}_{\mu ow^*} e\{(\lambda\, x : \tau.e_1 : \tau_1)\, /\, f\}}$$

(μow^*-Op0)

$$\frac{e_1 \xrightarrow{o}_{\mu ow^*} e_1'}{e_1 \text{ op } e_2 \xrightarrow{o}_{\mu ow^*} e_1' \text{ op } e_2}$$

(μow^*-Op1)

$$\frac{e_2 \xrightarrow{o}_{\mu ow^*} e_2'}{n_1 \text{ op } e_2 \xrightarrow{o}_{\mu ow^*} n_1 \text{ op } e_2'}$$

(μow^*-Op2)

$$\frac{n = n_1 \; op \; n_2}{n_1 \text{ op } n_2 \xrightarrow{o}_{\mu ow^*} n}$$

(μow^*-App0)

$$\frac{e_2 \xrightarrow{o}_{\mu ow^*} e_2'}{\textbf{let } x : \tau = (\lambda\, y : \tau_1.e_1 : \tau)\, e_2 \textbf{ in } e_3 \xrightarrow{o}_{\mu ow^*} \textbf{let } x : \tau = (\lambda\, y : \tau_1.e_1 : \tau)\, e_2' \textbf{ in } e_3}$$

(μow^*-App1)

$$\frac{}{\textbf{let } x : \tau = (\lambda\, y : \tau_1.e_1 : \tau)\, v \textbf{ in } e_3 \xrightarrow{\epsilon}_{\mu ow^*} \textbf{let } x : \tau = e_1\{v/y\} \textbf{ in } e_3}$$

(μow^*-Prim0)

$$\frac{f_p \in \mathit{Prim} \qquad e \xrightarrow{o}_{\mu ow^*} e'}{\textbf{let } x : \texttt{unit} = f_p\, e \textbf{ in } e_2 \xrightarrow{o}_{\mu ow^*} \textbf{let } x : \texttt{unit} = f_p\, e' \textbf{ in } e_2}$$

(μow^*-Prim1)

$$\frac{f_p \in \mathit{Prim}}{\textbf{let } x : \texttt{unit} = f_p\, v \textbf{ in } e_2 \xrightarrow{f_p}_{\mu ow^*} e_2}$$

(μow^*-If)

$$\frac{e_1 \xrightarrow{o}_{\mu ow^*} e_1'}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \xrightarrow{o}_{\mu ow^*} \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3}$$

(μow^*-If0)

$$\frac{}{\textbf{if } n \textbf{ then } e_2 \textbf{ else } e_3 \xrightarrow{\epsilon}_{\mu ow^*} e_3}\; n = 0$$

(μow^*-If1)

$$\frac{}{\textbf{if } n \textbf{ then } e_2 \textbf{ else } e_3 \xrightarrow{\epsilon}_{\mu ow^*} e_2}\; n = 1$$

Figure 43: Semantics of the μow^* language.

Now, let $C[P] \in \text{Whole}_{\mu ow^*}$ *be the whole program obtained by linking* P *with the context* $C = (\textbf{let } d = (\lambda\, y : \texttt{int}.y * 2 : \texttt{int}) \textbf{ in } [\cdot])$. *Analyzing* $C[P]$, *we get:*

$$(\!|C[P]|\!)^{\mu ow^*}_{\mu \mathscr{O}\mathscr{W}^*} = \{m \mid m \leq \epsilon \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot \texttt{display}\} = \{\epsilon, \texttt{display}\}.$$

Now, rather trivially, the following theorem follows:

Theorem 5.8. $(\!|\cdot|\!)^{\mu ow^*}_{\mu \mathscr{O}\mathscr{W}^*}$ *is a complete analysis.*

Proof. Easily from the definition of the analysis and the semantics of μow^* and $\mu \mathscr{O}\mathscr{W}^*$. □

5.4.2 *The concrete and abstract target languages and their analysis*

Here we present our target language μC^* and its abstract counterpart $\mu \mathscr{C}^*$. Also, we introduce the analysis as a sound type and effect system inferring $\mu \mathscr{C}^*$ (partial) programs and contexts starting from concrete ones in μC^*.

THE TARGET LANGUAGE: $\mu\mathbf{C}^*$ $\mu\mathbf{C}^*$ is an imperative language inspired by the C^* language by Protzenko et al. [193]. Just as in [193] and μow^*, in C^* we forbid nested function declarations. Let $SCName$ be the set of the names of available system calls, coinciding with the set of the observables of $\mu\mathbf{C}^*$. Similarly to primitives of μow^*, we model system calls as opaque procedures with no return value. Contexts and partial programs of $\mu\mathbf{C}^*$ are defined as follows:

$$f_{sc} \in SCName \quad x, y, f \in Name \quad f_c \in CName \quad n \in \mathbb{Z}$$
$$SCName \cap Name = \emptyset \quad SCName \cap CName = \emptyset \quad Name \cap CName = \emptyset$$
$$v \in Val = \mathbb{Z} \cup \{\, () \,\} \quad \mathtt{op} \in \{+, *, -\} \quad \tau, \tau_1 \in \{\mathtt{int}, \mathtt{unit}\}$$
$$\begin{aligned}
\mathbf{a} \quad &::= n \mid x \mid \mathbf{a_1}\ \mathtt{op}\ \mathbf{a_2} \\
\mathbf{Expr} \ni \mathbf{e} \quad &::= \mathbf{a} \mid () \\
\mathbf{Partial}_{\mu\mathbf{C}^*} \ni \mathbf{P} \quad &::= \tau\ x = v;\ \mathbf{P} \mid \mathtt{fun}\, f\,(y : \tau) : \tau_1\ \{\mathbf{s}\};\ \mathbf{P} \mid \mathbf{s} \\
\mathbf{s}, \mathbf{s_1}, \mathbf{s_2} \quad &::= \tau\ x = \mathbf{e};\ \mathbf{s} \mid \tau\ x = f\,(\mathbf{e});\ \mathbf{s} \mid \tau\ x = f_c\,(\mathbf{e});\ \mathbf{s} \mid \mathtt{call}\,(f_{sc}, \mathbf{e});\ \mathbf{s} \mid \\
&\quad\ \mathtt{if}\ (\mathbf{e})\ \{\mathbf{s_1}\}\ \mathtt{else}\ \{\mathbf{s_2}\};\ \mathbf{s} \mid \mathtt{return}\,\mathbf{e};\ \mathbf{s} \mid \mathtt{return}\,\mathbf{e} \\
Ctx_{\mu\mathbf{C}^*} \ni \mathbf{C} \quad &::= \mathtt{fun}\, f_c\,(y : \tau) : \tau_1\ \{\mathbf{s}\};\ \mathbf{C} \mid [\cdot]
\end{aligned}$$

Names and values are shared with the source language μow^*. Arithmetic expressions and expressions are standard and have no side effects. Partial programs are sequences of declarations terminating with a statement. The statements include declarations of variables, function calls ($x = f\,(\mathbf{e})$), calls to functions defined in the context ($x = f_c\,(\mathbf{e})$), and conditionals. All the statements end with a `return`. Returns appearing before the end of a statement are simply ignored. Similarly to C^*, variables are immutable and functions can only access their local variables and parameters. Additionally, our language has a special syntactic construct $\mathtt{call}\,(f_{sc}, \mathbf{e})$ used to invoke the system call f_{sc} passing to it the value obtained by evaluating $\mathbf{e}$.

Given a target context and a partial program, we define the linking operator as:

$$(\mathtt{fun}\, f_c\,(y : \tau) : \tau_1\ \{\mathbf{s}\};\ \mathbf{C})[\mathbf{P}] = \mathtt{fun}\, f_c\,(y : \tau) : \tau_1\ \{\mathbf{s}\};\ \mathbf{C}[\mathbf{P}]$$
$$([\cdot])[\mathbf{P}] = \mathbf{P}$$

Again, the set of whole programs $\mathbf{Whole}_{\mu\mathbf{C}^*}$ is the set of programs generated by the linking operator above. Before defining the semantics of $\mu\mathbf{C}^*$ we need to define the set $\mathbf{Runtime}_{\mu\mathbf{C}^*}$, corresponding to the set of configurations that may arise at runtime. To do that we define *extended partial programs* of $\mu\mathbf{C}^*$, built upon *extended statements* that enrich standard statements with the following two alternatives:

$$\begin{aligned}
\mathbf{s}, \mathbf{s_1} ::= \ldots \mid\ & \tau\ x = (\mathtt{fn}\ (y : \tau_y)\ \{\mathbf{s}\}_{\mathbf{\Gamma}})\ ;\ \mathbf{s_1} \mid \\
& \tau\ x = (\mathtt{fn}\ (y : \tau_y)\ \{\mathtt{return}\ \mathbf{e}\}_{\mathbf{\Gamma}})\ ;\ \mathbf{s_1}
\end{aligned}$$

where $\mathbf{\Gamma}$ is a typing environment used by our type and effect system to keep track of the types of the variables inside functions (see below).

The transition relation of the semantics has the form

$$(\bar{\sigma}, \mathbf{\Pi}), \mathbf{R} \xrightarrow{o}_{\mu\mathbf{C}^*} (\bar{\sigma}', \mathbf{\Pi}'), \mathbf{R}'$$

where (1) $\bar{\sigma}, \bar{\sigma}' \in \mathbf{Stack} = (Name \rightharpoonup Val)^*$ denotes a *stack*, i.e., a finite list of frames. A *frame* is a partial map from variables to their values; (2) $\mathbf{\Pi}, \mathbf{\Pi}'$ are *function environments* and associate function names with their definitions; (3) $\mathbf{R}, \mathbf{R}' \in \mathbf{Runtime}_{\mu\mathbf{C}^*}$ is a runtime program; and (4) $o \in \Sigma$ is an observable. Intuitively, a transition from $(\bar{\sigma}, \mathbf{\Pi}), \mathbf{R}$ to

$(\bar{\sigma}', \mathbf{\Pi}'), \mathbf{R}'$ with observable o denotes that the program $\mathbf{R}$ evolves into $\mathbf{R}'$ by emitting o as its observable behavior and changing $\bar{\sigma}$ and $\mathbf{\Pi}$ into $\bar{\sigma}'$ and $\mathbf{\Pi}'$. Figure 44 shows the rules defining the operational semantics of programs of $\mu\mathbf{C}^*$, where the evaluation function for arithmetic expressions $[\cdot]_\cdot : \mathbf{Expr} \times \mathbf{Stack} \to \mathit{Val}$ is defined as

$$
\begin{aligned}
[n]_{\bar{\sigma}} &= n \\
[x]_{\sigma' \cdot \bar{\sigma}} &= \sigma'(x) && \text{if } x \in \mathtt{dom}(\sigma') \\
[\mathbf{a_1} \ \mathtt{op} \ \mathbf{a_2}]_{\bar{\sigma}} &= [\mathbf{a_1}]_{\bar{\sigma}} \ \mathit{op} \ [\mathbf{a_2}]_{\bar{\sigma}}
\end{aligned}
$$

Also, we assume the usual concatenation operator on lists (denoted with $\cdot$); the update operator for mappings (written $\cdot[\cdot \mapsto \cdot]$); and the update operator on stacks to always operate on the topmost frame. Most of the rules are standard and reminiscent of those of Protzenko et al. [193]. The only interesting case is that of system calls: first, we reduce its parameter to a value and then we perform the actual call to f_{sc}, emitting the observable f_{sc} (rule ($\mu\mathbf{C}^*$-Call)). Behaviors are then defined as the finite traces that can be generated starting from a stack with a single, empty frame and an empty function environment:

$$
\mathit{beh}(\mathbf{R}) = \{t \mid \exists \mathbf{R}', \bar{\sigma}, \mathbf{\Pi}.\ (\{\}, \emptyset), \mathbf{R} \xrightarrow{t}{}^{*} \mu\mathbf{C}^*(\bar{\sigma}, \mathbf{\Pi}), \mathbf{R}'\}.
$$

THE ABSTRACT TARGET LANGUAGE: $\mu\mathscr{C}^*$ We now move to the abstract target language, $\mu\mathscr{C}^*$. Both partial and whole programs of $\mu\mathscr{C}^*$ are defined as history expressions:

$$
\begin{aligned}
\mathbf{Partial}_{\mu\mathscr{C}^*} \ni \mathbf{H_S}, \mathbf{H_S'} &::= \bullet^{\tau} \mid \hbar \mid \mathbf{H_S};\ \mathbf{H_S'} \mid \mathbf{H_S} + \mathbf{H_S'} \mid f_{sc}^{\tau} \\
\mathbf{Whole}_{\mu\mathscr{C}^*} \ni \mathbf{H_W} &::= \bullet^{\tau} \mid \mathbf{H_W};\ \mathbf{H_W'} \mid \mathbf{H_W} + \mathbf{H_W'} \mid f_{sc}^{\tau} \mid \top.
\end{aligned}
$$

Where $\bullet^{\tau}$ denotes an abstract program with empty behavior, whose type annotation τ intuitively corresponds to the type of the concrete statement the empty program abstracts; variables $\hbar$ indicate an unknown effect coming from the context; f_{sc}^{τ} denotes a system call, whose annotation τ corresponds to the type of the parameter of the system call; the operator $;$ is for sequencing; and $+$ denotes the non-deterministic choice; $\top$ is the history expression whose behavior is the set of all the possible prefixes of the traces over SCName.

Also, for any τ we let $(\mathbf{Partial}_{\mu\mathscr{C}^*}, \mathbf{op}, \bullet^{\tau})$ and $(\mathbf{Whole}_{\mu\mathscr{C}^*}, \mathbf{op}, \bullet^{\tau})$ be monoids if $\mathbf{op} = ;$, and commutative monoids if $\mathbf{op} = +$ (which is considered idempotent). Furthermore, for both $\mathbf{Partial}_{\mu\mathscr{C}^*}$ and $\mathbf{Whole}_{\mu\mathscr{C}^*}$ we assume the following axiom:

$$
\mathbf{o}; (\mathbf{H} + \mathbf{H}') = \mathbf{o}; \mathbf{H} + \mathbf{o}; \mathbf{H}'.
$$

(Note that from the above assumptions it is easy to define a rewriting system that reduces history expressions to a normal form, up to associativity and commutativity, see e.g., [32].)

Finally, from now onward we will stipulate that any context name $f_c \in \mathit{CName}$ is associated with the unique variable $\hbar_{f_c}$ and we consider histories up to the above axioms and assumptions.

(μC*-FunDecl)
$$\frac{f \in \mathit{Name} \cup \mathit{CName}}{(\bar{\sigma}, \mathbf{\Pi}), \texttt{fun}\, f\ (y : \tau_f) : \tau_1\ \{\mathrm{s}_1\};\ \mathrm{s} \xrightarrow{\epsilon}_{\mu\mathbf{C}^*} (\bar{\sigma}, \mathbf{\Pi}[f \mapsto (\texttt{fn}\ (y : \tau_y)\ \{\mathrm{s}_1\})]), \mathrm{s}}$$

(μC*-VarDecl)
$$\frac{[\mathrm{e}]_{\sigma' \cdot \bar{\sigma}} = v \qquad x \notin \texttt{dom}\,(\sigma' \cdot \bar{\sigma})}{(\bar{\sigma}, \mathbf{\Pi}), \tau\ x = \mathrm{e};\ \mathrm{s} \xrightarrow{\epsilon}_{\mu\mathbf{C}^*} (\bar{\sigma}[x \mapsto v], \mathbf{\Pi}), \mathrm{s}}$$

(μC*-App0)
$$\frac{f \in \mathit{Name} \cup \mathit{CName} \qquad \mathbf{\Pi}(f) = \texttt{fn}\ (y : \tau_y)\ \{\mathrm{s}_1\}}{(\bar{\sigma}, \mathbf{\Pi}), \tau\ x = f\ (\mathrm{e});\ \mathrm{s} \xrightarrow{\epsilon}_{\mu\mathbf{C}^*} (\{\mathbf{y} \mapsto [\mathrm{e}]_{\bar{\sigma}}\} \cdot \bar{\sigma}, \mathbf{\Pi}), \tau\ x = (\texttt{fn}\ (y : \tau_y)\ \{\mathrm{s}_1\}_{\emptyset})\ ;\ \mathrm{s}}$$

(μC*-App1-Decl)
$$\frac{(\bar{\sigma}, \mathbf{\Pi}), \tau\ x = \mathrm{e};\ \mathrm{s}'_1 \xrightarrow{o}_{\mu\mathbf{C}^*} (\bar{\sigma}', \mathbf{\Pi}'), \mathrm{s}'_1 \qquad \mathbf{\Gamma}' = \mathbf{\Gamma}[x \mapsto \tau]}{(\bar{\sigma}, \mathbf{\Pi}), \tau\ x = \left(\texttt{fn}\ (y : \tau_y)\ \{\tau\ x = \mathrm{e};\ \mathrm{s}'_1\}_{\mathbf{\Gamma}}\right)\ ;\ \mathrm{s} \xrightarrow{o}_{\mu\mathbf{C}^*} (\bar{\sigma}', \mathbf{\Pi}'), \tau\ x = \left(\texttt{fn}\ (y : \tau_y)\ \{\mathrm{s}'_1\}_{\mathbf{\Gamma}'}\right)\ ;\ \mathrm{s}}$$

(μC*-App1)
$$\frac{\mathrm{s}_1 \notin \{\texttt{return}\ \mathrm{e}, \tau\ x = \mathrm{e};\ \mathrm{s}'_1\} \qquad (\bar{\sigma}, \mathbf{\Pi}), \mathrm{s}_1 \xrightarrow{o}_{\mu\mathbf{C}^*} (\bar{\sigma}', \mathbf{\Pi}'), \mathrm{s}'_1}{(\bar{\sigma}, \mathbf{\Pi}), \tau\ x = (\texttt{fn}\ (y : \tau_y)\ \{\mathrm{s}_1\}_{\mathbf{\Gamma}})\ ;\ \mathrm{s} \xrightarrow{o}_{\mu\mathbf{C}^*} (\bar{\sigma}', \mathbf{\Pi}'), \tau\ x = \left(\texttt{fn}\ (y : \tau_y)\ \{\mathrm{s}'_1\}_{\mathbf{\Gamma}}\right)\ ;\ \mathrm{s}}$$

(μC*-App2)
$$\frac{}{(\sigma' \cdot \bar{\sigma}, \mathbf{\Pi}), \tau\ x = (\texttt{fn}\ (y : \tau_y)\ \{\texttt{return}\ \mathrm{e}\}_{\mathbf{\Gamma}})\ ;\ \mathrm{s} \xrightarrow{\epsilon}_{\mu\mathbf{C}^*} (\bar{\sigma}, \mathbf{\Pi}), \tau\ x = v;\ \mathrm{s}}\ [\mathrm{e}]_{\sigma' \cdot \bar{\sigma}} = v$$

(μC*-Call)
$$\frac{}{(\bar{\sigma}, \mathbf{\Pi}), \texttt{call}(f_{sc}, \mathrm{e});\ \mathrm{s} \xrightarrow{f_{sc}}_{\mu\mathbf{C}^*} (\bar{\sigma}, \mathbf{\Pi}), \mathrm{s}}\ [\mathrm{e}]_{\bar{\sigma}} = v$$

(μC*-If0)
$$\frac{}{(\bar{\sigma}, \mathbf{\Pi}), \texttt{if}\ (\mathrm{e})\ \{\mathrm{s}_1\}\ \texttt{else}\ \{\mathrm{s}_2\};\ \mathrm{s} \xrightarrow{\epsilon}_{\mu\mathbf{C}^*} (\bar{\sigma}, \mathbf{\Pi}), \mathrm{s}_2;\ \mathrm{s}}\ [\mathrm{e}]_{\bar{\sigma}} = 0$$

(μC*-If1)
$$\frac{}{(\bar{\sigma}, \mathbf{\Pi}), \texttt{if}\ (\mathrm{e})\ \{\mathrm{s}_1\}\ \texttt{else}\ \{\mathrm{s}_2\};\ \mathrm{s} \xrightarrow{\epsilon}_{\mu\mathbf{C}^*} (\bar{\sigma}, \mathbf{\Pi}), \mathrm{s}_1;\ \mathrm{s}}\ [\mathrm{e}]_{\bar{\sigma}} \neq 0$$

(μC*-Ret)
$$\frac{}{(\bar{\sigma}, \mathbf{\Pi}), \texttt{return}\ \mathrm{e} \xrightarrow{\epsilon}_{\mu\mathbf{C}^*} (\bar{\sigma}, \mathbf{\Pi}), \mathrm{v}}\ [\mathrm{e}]_{\bar{\sigma}} = v$$

(μC*-RetSeq)
$$\frac{}{(\bar{\sigma}, \mathbf{\Pi}), \texttt{return}\ \mathrm{e};\ \mathrm{s} \xrightarrow{\epsilon}_{\mu\mathbf{C}^*} (\bar{\sigma}, \mathbf{\Pi}), \mathrm{s}}$$

Figure 44: Semantics for runtime programs of the μC^* language.

Abstract contexts are just special kind typing contexts mapping function names into their type and latent effects:

$$Ctx_{\mu\mathscr{C}^*} \ni \mathbf{H_C} ::= \emptyset \mid \mathbf{H_C}[f \mapsto \tau \xrightarrow{\mathbf{H_S}} \tau_1]$$

The linking operator is defined by cases as follows:

$$\mathbf{H_C}[\mathbf{H}] = \begin{cases} \mathbf{H_W} & \text{if } \mathbf{H_W} = subst(\mathbf{H_C}, \mathbf{H}) \wedge FV(\mathbf{H_W}) = \emptyset \\ \top & \text{o.w.} \end{cases}$$

where $subst$ is

$$subst(\emptyset, \mathbf{H}) = \mathbf{H}$$
$$subst(\mathbf{H_C}[\mathbf{f_c} \mapsto \tau \xrightarrow{\mathbf{H}_{f_c}} \tau_1], \mathbf{H}) = subst(\mathbf{H_C}, \mathbf{H}\{\mathbf{H}_{f_c}/\hbar_{f_c}\}).$$

It is easy to see that $\cdot[\cdot]$ always produces a whole program of $\mu\mathscr{C}^*$. Since we allow the attacker to freely choose the (abstract) context, when the current context does not provide enough information to complete the linking (i.e., when $FV(\mathbf{H_W}) \neq \emptyset$), the linking operator produces the whole abstract program $\top$ to reflect the fact. The semantics of whole programs of $\mu\mathscr{C}^*$ is in Figure 45. Intuitively, the rule ($\mu\mathscr{C}^*$-Eps) allows any history expression to emit an empty observable. The second rule describes the behavior of $\top$, which can move to itself by emitting *any* observable. The rule ($\mu\mathscr{C}^*$-Seq) is for sequencing and the fourth rule reduces histories denoting system calls. Finally, rule ($\mu\mathscr{C}^*$-Choice) deals with the choice operator by reducing it to its right operand. Note that there is no need of a symmetric rule (i.e., one that reduces the choice operator to the left operand) since $+$ is commutative monoid.

THE ANALYSIS For the purpose of applying $aSTV_{TI\text{-}RSC}$ we now need a (sound and modular) analysis that abstracts (partial and runtime) programs and contexts of $\mu\mathbf{C}^*$ into those of $\mu\mathscr{C}^*$. As said above, our analysis is based on a type and effect system that extracts abstract programs from concrete ones. Types are either τ as in $\mu\mathbf{C}^*$, or functional types $\tau \xrightarrow{\mathbf{H}} \tau_1$, with $\mathbf{H}$ being the latent effect, i.e., an effect that is observable upon the execution of the function associated with it. Figures 46 and 47 display the type and effect system for extended partial programs of $\mu\mathbf{C}^*$, whose judgments have the form:

$$\mathbf{\Gamma}, \mathbf{\Delta} \vdash \mathbf{P} : \tau \,\&\, \mathbf{H}$$

($\mu\mathscr{C}^*$-Eps)
$$\mathbf{H_W} \xrightarrow{\epsilon}_{\mu\mathscr{C}^*} \mathbf{H_W}$$

($\mu\mathscr{C}^*$-Top)
$$\frac{o \in \Sigma}{\top \xrightarrow{o}_{\mu\mathscr{C}^*} \top}$$

($\mu\mathscr{C}^*$-Seq)
$$\frac{\mathbf{H_W} \xrightarrow{o}_{\mu\mathscr{C}^*} \mathbf{H''_W}}{\mathbf{H_W};\, \mathbf{H'_W} \xrightarrow{o}_{\mu\mathscr{C}^*} \mathbf{H''_W};\, \mathbf{H'_W}}$$

($\mu\mathscr{C}^*$-SysCall)
$$\frac{}{f^{\tau}_{sc} \xrightarrow{f_{sc}}_{\mu\mathscr{C}^*} \bullet^{\tau}}$$

($\mu\mathscr{C}^*$-Choice)
$$\frac{}{\mathbf{H_W} + \mathbf{H'_W} \xrightarrow{\epsilon}_{\mu\mathscr{C}^*} \mathbf{H_W}}$$

Figure 45: Semantics for whole programs of the $\mu\mathscr{C}^*$ language.

(AP-INT)
$$\Gamma, \Delta \vdash n : \texttt{int} \ \&\ \bullet^{\texttt{int}}$$

(AP-UNIT)
$$\Gamma, \Delta \vdash () : \texttt{unit} \ \&\ \bullet^{\texttt{void}}$$

(AP-VAR)
$$\frac{\Gamma(x) = \tau}{\Gamma, \Delta \vdash x : \tau \ \&\ \bullet^{\tau}}$$

(AP-OP)
$$\frac{\Gamma, \Delta \vdash \mathbf{a_1} : \texttt{int} \ \&\ \bullet^{\texttt{int}} \qquad \Gamma, \Delta \vdash \mathbf{a_2} : \texttt{int} \ \&\ \bullet^{\texttt{int}}}{\Gamma, \Delta \vdash \mathbf{a_1} \ \texttt{op} \ \mathbf{a_2} : \texttt{int} \ \&\ \bullet^{\texttt{int}}}$$

(AP-DECLVAR) + case for $\mathbf{P}$ in place of $\mathbf{s}$
$$\frac{\Gamma, \Delta \vdash \mathbf{e} : \tau \ \&\ \bullet^{\tau} \qquad \Gamma[x \mapsto \tau], \Delta \vdash \mathbf{s} : \tau_{\mathbf{s}} \ \&\ \mathbf{H}}{\Gamma, \Delta \vdash \tau \ x = \mathbf{e};\ \mathbf{s} : \tau_{\mathbf{s}} \ \&\ \mathbf{H}}$$

(AP-DECLFUN) + case for $\mathbf{P}$ in place of $\mathbf{s}$
$$\frac{f \in Name \cup CName \qquad [y \mapsto \tau_y], \Delta \vdash \mathbf{s_1} : \tau_f \ \&\ \mathbf{H}_f \qquad \Gamma, \Delta[f \mapsto (\tau_y \xrightarrow{\mathbf{H}_f} \tau_f)] \vdash \mathbf{s} : \tau_{\mathbf{s}} \ \&\ \mathbf{H_s}}{\Gamma, \Delta \vdash \texttt{fun}\, f\ (y : \tau_y) : \tau_f\ \{\mathbf{s_1}\};\ \mathbf{s} : \tau_{\mathbf{s}} \ \&\ \mathbf{H_s}}$$

Figure 46: The type and effect system for extended partial programs of $\mu\mathscr{C}^*$ (part I).

where $\mathbf{P}$ is an extended partial program, and Γ and Δ are typing environments mapping names of variables and names of functions into their types. We now briefly comment on the most interesting rules of the type and effect system. Rules (AP-INT)-(AP-OP) allow us to infer that expressions never have observable behavior. Rules (AP-DECLVAR) and (AP-DECLFUN) deal with the declarations of variables and functions and are standard. The rule (AP-APP0) typechecks function calls using the typing information coming from Δ and uses the latent effect of the function being executed to infer the effect of the whole program in hand. Rules (AP-APP1) and (AP-APP2) are similar, but deal with syntactic configurations belonging to extended statements. Crucially, the additional typing environment Γ_1 is used to check types and infer effects of the body of the function being executed, so faithfully simulating the stack of the dynamic semantics. Rule (AP-APPNF) is used in partial programs only and puts a placeholder in place of the effect of functions coming from the context (to be substituted upon linking). Effects of system calls are inferred by rule (AP-CALL). Finally, rules (AP-IF), (AP-RET), and (AP-RETSEQ) are standard and deal with conditionals and returns. Figure 48 displays the rather trivial type and effect system for contexts of $\mu\mathbf{C}^*$. Its judgments have the form:

$$\Gamma, \Delta \vdash \mathbf{C} : \tau \ \&\ \mathbf{H}$$

where typing environments are as above, $\mathbf{C}$ is a context and $\mathbf{H_C}$ is a context of $\mu\mathscr{C}^*$ (i.e., a typing environment mapping names from *CName* to their type and latent effect). The final part of the type and effect system concerns programs and it just puts together the results coming from the analysis of context and partial programs, Figure 49 displays it. We are finally ready to state and prove the modularity and the soundness theorems. Recall that modularity prescribes that the behavior of the analysis of a context linked with a partial program equals to that of the analyses of the context and of the program performed separately and then linked together:

Theorem 5.9 (Modularity). *Let* $\mathbf{C}$ *be a context and* $\mathbf{P}$ *be a partial program. If* $\Gamma \vdash \mathbf{C} \ \&\ \mathbf{H_C}$, $\Gamma \vdash \mathbf{P} : \tau_{\mathbf{P}} \ \&\ \mathbf{H_P}$, *and* $\Gamma \vdash \mathbf{C}[\mathbf{P}] : \tau_{\mathbf{W}} \ \&\ \mathbf{H_W}$ *then* $beh(\mathbf{H_W}) = beh(\mathbf{H_C}[\mathbf{H_P}])$.

(AP-App0)

$$\frac{f \in Name \cup CName \qquad \Gamma \vdash \mathbf{e} : \tau_{\mathbf{e}} \,\&\, \bullet^{\tau_{\mathbf{e}}} \qquad \Delta(f) = \tau_{\mathbf{e}} \xrightarrow{\mathbf{H}_f} \tau \qquad \Gamma[x \mapsto \tau], \Delta \vdash \mathbf{s} : \tau_{\mathbf{s}} \,\&\, \mathbf{H}_{\mathbf{s}}}{\Gamma, \Delta \vdash \tau\ x = f\,(\mathbf{e});\ \mathbf{s} : \tau_{\mathbf{s}} \,\&\, \mathbf{H}_f;\ \mathbf{H}_{\mathbf{s}}}$$

(AP-App1)

$$\frac{\mathbf{s}_1 \neq \mathtt{return}\ \mathbf{e} \qquad \Gamma_1[y \mapsto \tau_y], \Delta \vdash \mathbf{s}_1 : \tau \,\&\, \mathbf{H}_{\mathbf{s}_1} \qquad \Gamma[x \mapsto \tau], \Delta \vdash \mathbf{s} : \tau_{\mathbf{s}} \,\&\, \mathbf{H}_{\mathbf{s}}}{\Gamma, \Delta \vdash \tau\ x = (\mathtt{fn}\ (y : \tau_y)\ \{\mathbf{s}_1\}_{\Gamma_1});\ \mathbf{s} : \tau_{\mathbf{s}} \,\&\, \mathbf{H}_{\mathbf{s}_1};\ \mathbf{H}_{\mathbf{s}}}$$

(AP-App2)

$$\frac{\Gamma_1[y \mapsto \tau_y], \Delta \vdash \mathbf{e} : \tau \,\&\, \bullet^{\tau} \qquad \Gamma[x \mapsto \tau], \Delta \vdash \mathbf{s} : \tau_{\mathbf{s}} \,\&\, \mathbf{H}_{\mathbf{s}}}{\Gamma, \Delta \vdash \tau\ x = (\mathtt{fn}\ (y : \tau_y)\ \{\mathtt{return}\ \mathbf{e}\}_{\Gamma_1});\ \mathbf{s} : \tau_{\mathbf{s}} \,\&\, \mathbf{H}_{\mathbf{s}}}$$

(AP-AppNF)

$$\frac{\Gamma, \Delta \vdash \mathbf{e} : \tau_{\mathbf{e}} \,\&\, \bullet^{\tau_{\mathbf{e}}} \qquad f_c \notin \mathtt{dom}\,(\Delta) \qquad \Gamma[x \mapsto \tau_x], \Delta \vdash \mathbf{s} : \tau_{\mathbf{s}} \,\&\, \mathbf{H}_{\mathbf{s}}}{\Gamma, \Delta \vdash \tau\ x = f_c\,(\mathbf{e});\ \mathbf{s} : \tau_{\mathbf{s}} \,\&\, \hbar_{\mathbf{f}_c};\ \mathbf{H}_{\mathbf{s}}}$$

(AP-Call)

$$\frac{\Gamma, \Delta \vdash \mathbf{e} : \tau_{\mathbf{e}} \,\&\, \bullet^{\tau_{\mathbf{e}}} \qquad \Gamma, \Delta \vdash \mathbf{s} : \tau_{\mathbf{s}} \,\&\, \mathbf{H}_{\mathbf{s}}}{\Gamma, \Delta \vdash \mathtt{call}\,(f_{sc}, \mathbf{e});\ \mathbf{s} : \tau_{\mathbf{s}} \,\&\, f_{sc}^{\tau_{\mathbf{e}}};\ \mathbf{H}_{\mathbf{s}}}$$

(AP-If)

$$\frac{\Gamma, \Delta \vdash \mathbf{e} : \mathtt{int} \,\&\, \bullet^{\mathtt{int}} \qquad \Gamma, \Delta \vdash \mathbf{s}_1 : \tau \,\&\, \mathbf{H}_{\mathbf{s}_1} \qquad \Gamma, \Delta \vdash \mathbf{s}_2 : \tau \,\&\, \mathbf{H}_{\mathbf{s}_2} \qquad \Gamma, \Delta \vdash \mathbf{s} : \tau_{\mathbf{s}} \,\&\, \mathbf{H}_{\mathbf{s}}}{\Gamma, \Delta \vdash \mathtt{if}\ (\mathbf{e})\ \{\mathbf{s}_1\}\ \mathtt{else}\ \{\mathbf{s}_2\};\ \mathbf{s} : \tau_{\mathbf{s}} \,\&\, (\mathbf{H}_{\mathbf{s}_1} + \mathbf{H}_{\mathbf{s}_2});\ \mathbf{H}_{\mathbf{s}}}$$

(AP-Ret)

$$\frac{\Gamma, \Delta \vdash \mathbf{e} : \tau \,\&\, \bullet^{\tau}}{\Gamma, \Delta \vdash \mathtt{return}\ \mathbf{e} : \tau \,\&\, \bullet^{\tau}}$$

(AP-RetSeq)

$$\frac{\Gamma, \Delta \vdash \mathbf{e} : \tau \,\&\, \bullet^{\tau} \qquad \Gamma, \Delta \vdash \mathbf{s} : \tau_{\mathbf{s}} \,\&\, \mathbf{H}_{\mathbf{s}}}{\Gamma, \Delta \vdash \mathtt{return}\ \mathbf{e};\ \mathbf{s} : \tau_{\mathbf{s}} \,\&\, \mathbf{H}_{\mathbf{s}}}$$

Figure 47: The type and effect system for extended partial programs of $\mu\mathscr{C}^*$ (part II).

Proof. By the definition of the *subst* function and the rules from Figures 46, 47 and 49, it easily follows that $\mathbf{H}_{\mathbf{W}} = \mathbf{H}_{\mathbf{C}}[\mathbf{H}_{\mathbf{P}}]$, thus the thesis. □

Soundness requires instead that the behavior of abstract programs over-approximates that of their concrete counterpart. To prove that we first need to introduce three new auxiliary definitions and a preservation lemma. The first definition introduces the relation $\asymp$, linking a function environment Π with the typing environment for functions Δ. Intuitively, $\asymp$ ensures that the type and the latent effect stored in Π for a function f correspond to the definition of f in Δ:

Definition 5.9. *Let Π be a function environment and Δ be a typing environment for functions. $\Pi \asymp \Gamma$ holds* iff *for any $f \in Name \cup CName$*

$$\Pi(f) = \mathtt{fn}\ (y : \tau_y)\ \{\mathbf{s}\} \ \wedge\ \Delta\,(f) = \tau_y \xrightarrow{\mathbf{H}} \tau \implies [y \mapsto \tau_y] \vdash \mathbf{s} : \tau \,\&\, \mathbf{H}.$$

(AC-HOLE)
$$\mathbf{\Gamma}, \mathbf{\Delta} \vdash [\cdot] : \emptyset$$

(AC-FUNDECL)
$$\frac{\mathbf{\Gamma}[x \mapsto \tau_x], \mathbf{\Delta} \vdash \mathbf{s} : \tau_{f_c} \,\&\, \mathbf{H_s} \qquad \mathbf{\Gamma}, \mathbf{\Delta}[f_c \mapsto (\tau_x \xrightarrow{\mathbf{H_s}} \tau_{f_c})] \vdash \mathbf{C} \,\&\, \mathbf{H_C}}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash \texttt{fun}\ f_c\ (x : \tau_x) : \tau_{f_c}\ \{\mathbf{s}\};\ \mathbf{C} \,\&\, \mathbf{H_C}[f_c \mapsto (\tau_x \xrightarrow{\mathbf{H_s}} \tau_{\mathbf{s}})]}$$

Figure 48: The type and effect system for contexts of $\mu\mathscr{C}^*$.

(AW-OK)
$$\frac{\mathbf{\Gamma}, \mathbf{\Delta} \vdash \mathbf{C} \,\&\, \mathbf{H_C} \qquad \mathbf{\Gamma}, \mathbf{H_C} \vdash \mathbf{C}[\mathbf{P}] : \tau \,\&\, \mathbf{H} \qquad FV(\mathbf{H}) = \emptyset}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash \mathbf{C}[\mathbf{P}] : \tau \,\&\, \mathbf{H}}$$

(AW-TOP)
$$\frac{\mathbf{\Gamma}, \mathbf{\Delta} \vdash \mathbf{C} \,\&\, \mathbf{H_C} \qquad \mathbf{\Gamma}, \mathbf{H_C} \vdash \mathbf{C}[\mathbf{P}] : \tau \,\&\, \mathbf{H} \qquad FV(\mathbf{H}) \neq \emptyset}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash \mathbf{C}[\mathbf{P}] : \tau \,\&\, \top}$$

Figure 49: The type and effect system for whole programs of $\mu\mathscr{C}^*$.

The second definition introduces the function γ. Intuitively, γ takes a typing environment for variables $\mathbf{\Gamma}$ and a runtime program $\mathbf{R}$ and transforms its first parameter into a typing environment with enough information to type the program obtained from $\mathbf{R}$ in one step of computation. To this aim, γ adds to $\mathbf{\Gamma}$ all the bindings that are introduced by one evaluation step of $\mathbf{R}$:

Definition 5.10. *The function γ is defined as follows:*

$$\gamma(\mathbf{\Gamma}, \tau\ x = \mathbf{e};\ \mathbf{s}) = \mathbf{\Gamma}[x \mapsto \tau]$$
$$\gamma(\mathbf{\Gamma}, \mathbf{R}) = \mathbf{\Gamma}$$

The third definition is analogous to the previous one, but applies to typing environments for functions:

Definition 5.11. *The function δ is defined as follows:*

$$\delta(\mathbf{\Delta}, (\texttt{fun}\ f\ (y : \tau_y) : \tau_f\ \{\mathbf{s_1}\};\ \mathbf{s})) = \mathbf{\Delta}[f \mapsto \tau_y \xrightarrow{\mathbf{H}_f} \tau_f]$$
$$\text{with } \mathbf{H}_f \text{ s.t. } [y \mapsto \tau_y], \mathbf{\Delta} \vdash \mathbf{s_1} : \tau_f \,\&\, \mathbf{H}_f$$
$$\delta(\mathbf{\Delta}, \mathbf{R}) = \mathbf{\Delta}$$

We are now ready to prove the preservation lemma, stating that types are preserved by the semantics of $\mu\mathbf{C}^*$ and effects reflect the actual behavior of programs in the target language:

Lemma 5.1 (Subject reduction). *For any $\mathbf{R}, \mathbf{\Gamma}, \mathbf{\Delta}, \bar{\sigma}, \mathbf{\Pi}, \bar{\sigma}', \mathbf{\Pi}', o, \mathbf{R}', \tau_{\mathbf{R}}, \mathbf{H_R} \neq \top$, if*

$$(\bar{\sigma}, \mathbf{\Pi}), \mathbf{R} \xrightarrow{o}_{\mu\mathbf{C}^*} (\bar{\sigma}', \mathbf{\Pi}'), \mathbf{R}' \ \wedge\ \mathbf{\Gamma}, \mathbf{\Delta} \vdash \mathbf{R} : \tau_{\mathbf{R}} \,\&\, \mathbf{H_R} \ \wedge\ \mathbf{\Pi} \asymp \mathbf{\Delta}$$

then there exists $\mathbf{H}'_{\mathbf{R}}$

$$\mathbf{H_R} \xrightarrow{o}_{\mu\mathscr{C}^*} \mathbf{H}'_{\mathbf{R}} \ \wedge\ \gamma(\mathbf{\Gamma}, \mathbf{R}), \delta(\mathbf{\Delta}, \mathbf{R}) \vdash \mathbf{R}' : \tau_{\mathbf{R}} \,\&\, \mathbf{H}'_{\mathbf{R}} \ \wedge\ \mathbf{\Pi}' \asymp \delta(\mathbf{\Delta}, \mathbf{R}).$$

Proof. Below, if not specified we assume all variables to be universally quantified. The proof goes by induction on the derivations of $\Gamma, \Delta \vdash R : \tau_R \;\&\; H_R$. Since we assume that $H_R \neq \top$, and since updates to Δ performed by rules in Figure 48 produce the same results as updates performed by those in Figures 46 and 47, we just consider the rules from the latter figures.

BASE CASES: (AP-INT) AND (AP-UNIT). Values cannot progress, thus the implication follows trivially.

BASE CASE: (AP-VAR) AND (AP-OP). By definition it never happens that R is a variable or an arithmetic expression.

BASE CASE: (AP-RET). Let $H_{R'} = \bullet^{\tau}$. Also, observe that in this case the program reduces to a value v of the same type τ of e, thus $\gamma(\Gamma, \texttt{return e}), \delta(\Delta, \texttt{return e}) \vdash v : \tau_R \;\&\; \bullet^{\tau}$ trivially holds. Finally, $\delta(\Delta, \texttt{return e}) = \Delta$ and $\Pi' = \Pi$ so $\Pi' \asymp \delta(\Delta, \texttt{return e})$.

INDUCTIVE CASE: (AP-RETSEQ). Trivially by the fact that $\texttt{return e; s}$ reduces in one step to s emitting ϵ.

INDUCTIVE CASE: (AP-DECLVAR). We must show that:

$$\begin{aligned}
&(\bar{\sigma}_R, \Pi_R), \tau\ x = e;\ s \xrightarrow{\epsilon}_{\mu C^*} (\bar{\sigma}_R[x \mapsto [e]_{\bar{\sigma}}], \Pi_R), s \;\wedge \\
&\quad \Gamma_R, \Delta_R \vdash \tau\ x = e;\ s : \tau_R \;\&\; H_R \wedge \Pi_R \asymp \Delta_R \implies \\
&\qquad \exists H_{R'}.\ H_R \xrightarrow{\epsilon}_{\mu\mathscr{C}^*} H_{R'} \;\wedge \\
&\qquad\quad \gamma(\Gamma_R, \tau\ x = e;\ s), \delta(\Delta_R, \tau\ x = e;\ s) \vdash s : \tau_R \;\&\; H_{R'} \wedge \\
&\qquad\quad \Pi_{R'} \asymp \delta(\Delta_R, \tau\ x = e;\ s).
\end{aligned}$$

The existence of some $H_{R'}$ is given by choosing it to be H_R. Also, noting that $\gamma(\Gamma_R, \tau\ x = e;\ s) = \Gamma_R[x \mapsto \tau]$ and $\delta(\Delta_R, \tau\ x = e;\ s) = \Delta_R$ (due to the premises of (AP-DECLVAR)) we deduce that $\Gamma_R[x \mapsto \tau], \Delta_R \vdash s : \tau_R \;\&\; H_R$. Finally, since the Π_R is left unchanged by the semantics, thus $\Pi_R \asymp \Delta_R = \delta(\Delta_R, \tau\ x = e;\ s)$.

INDUCTIVE CASE: (AP-DECLFUN). We must show that:

$$\begin{aligned}
&(\bar{\sigma}_R, \Pi_R), \texttt{fun}\ f\ (y : \tau_y) : \tau_f\ \{s_1\};\ s \xrightarrow{\epsilon}_{\mu C^*} (\bar{\sigma}_R, \Pi_R[f \mapsto (\tau_y \xrightarrow{H_f} \tau_f)]), s \;\wedge \\
&\quad \Gamma_R, \Delta_R \vdash \texttt{fun}\ f\ (y : \tau_y) : \tau_f\ \{s_1\};\ s : \tau_R \;\&\; H_R \wedge \Pi_R \asymp \Delta_R \implies \\
&\qquad \exists H_{R'}.\ H_R \xrightarrow{\epsilon}_{\mu\mathscr{C}^*} H_{R'} \;\wedge \\
&\qquad\quad \gamma(\Gamma_R, \texttt{fun}\ f\ (y : \tau_y) : \tau_f\ \{s_1\};\ s), \\
&\qquad\qquad \delta(\Delta_R, \texttt{fun}\ f\ (y : \tau_y) : \tau_f\ \{s_1\};\ s) \vdash s : \tau_R \;\&\; H_{R'} \wedge \\
&\qquad\quad \Pi_{R'} \asymp \delta(\Delta_R, \texttt{fun}\ f\ (y : \tau_y) : \tau_f\ \{s_1\};\ s).
\end{aligned}$$

As in the previous case, let $H_{R'} = H_R$.

Note that $\gamma(\Gamma_R, \texttt{fun}\ f\ (y : \tau_y) : \tau_f\ \{s_1\};\ s) = \Gamma_R$.

Also, $\delta(\Delta_R, \texttt{fun}\ f\ (y : \tau_y) : \tau_f\ \{s_1\};\ s) = \Delta_R[f \mapsto \tau_y \xrightarrow{H_{s_1}} \tau_f]$ where H_{s_1} is such that $[y \mapsto \tau_y], \Delta_R \vdash s_1 : \tau_f \;\&\; H_{s_1}$. Thus, by the premises of (AP-DECLFUN) we can conclude $\Gamma_R, \Delta_R[f \mapsto \tau_y \xrightarrow{H_{s_1}} \tau_f] \vdash s : \tau_R \;\&\; H_R$. Finally, $\Pi_{R'} \asymp \delta(\Delta_R, \texttt{fun}\ f\ (y : \tau_y) : \tau_f\ \{s_1\};\ s)$ by the definition of δ.

INDUCTIVE CASE: (AP-App0). We must show that:

$$\begin{aligned}
&(\bar{\sigma}_{\mathbf{R}}, \mathbf{\Pi}_{\mathbf{R}}), \tau\ x = f\ (\mathrm{e});\ \mathrm{s} \xrightarrow{\epsilon}_{\mu\mathbf{C}^*} (\bar{\sigma}_{\mathbf{R}}, \mathbf{\Pi}_{\mathbf{R}}), \mathbf{R}' \wedge \\
&\quad \mathbf{\Gamma}_{\mathbf{R}}, \mathbf{\Delta}_{\mathbf{R}} \vdash \tau\ x = f\ (\mathrm{e});\ \mathrm{s} : \tau_{\mathbf{R}}\ \&\ \mathbf{H}_{\mathbf{R}} \wedge \mathbf{\Pi}_{\mathbf{R}} \asymp \mathbf{\Delta}_{\mathbf{R}} \implies \\
&\qquad \exists \mathbf{H}_{\mathbf{R}'}.\ \mathbf{H}_{\mathbf{R}} \xrightarrow{\epsilon}_{\mu\mathscr{C}^*} \mathbf{H}_{\mathbf{R}'} \wedge \\
&\qquad\quad \gamma(\mathbf{\Gamma}_{\mathbf{R}}, \tau\ x = f\ (\mathrm{e});\ \mathrm{s}), \delta(\mathbf{\Delta}_{\mathbf{R}}, \tau\ x = f\ (\mathrm{e});\ \mathrm{s}) \vdash \mathrm{s} : \tau_{\mathbf{R}}\ \&\ \mathbf{H}_{\mathbf{R}'} \wedge \\
&\qquad\quad \mathbf{\Pi}_{\mathbf{R}'} \asymp \delta(\mathbf{\Delta}_{\mathbf{R}}, \tau\ x = f\ (\mathrm{e});\ \mathrm{s}).
\end{aligned}$$

Observe now that $\mathbf{H}_{\mathbf{R}} = \mathbf{H}_f;\ \mathbf{H}_{\mathrm{s}}$. $\mathbf{H}_f$ is the latent effect of f in $\mathbf{\Delta}_{\mathbf{R}}$ and by the fact that $\mathbf{\Pi}_{\mathbf{R}} \asymp \mathbf{\Delta}_{\mathbf{R}}$, it is such that $[y \mapsto \tau_y], \mathbf{\Delta}_{\mathbf{R}} \vdash \mathrm{s}_1 : \tau\ \&\ \mathbf{H}_f$. Instead, $\mathbf{H}_{\mathrm{s}}$ is such that $\mathbf{\Gamma}_{\mathbf{R}}[x \mapsto \tau], \mathbf{\Delta}_{\mathbf{R}} \vdash \mathrm{s} : \tau_{\mathrm{s}}\ \&\ \mathbf{H}_{\mathrm{s}}$ by the premises of (AP-App0). Let now $\mathbf{H}_{\mathbf{R}'} = \mathbf{H}_{\mathbf{R}}$ and note that $\gamma(\mathbf{\Gamma}_{\mathbf{R}}, \tau\ x = f\ (\mathrm{e});\ \mathrm{s}) = \mathbf{\Gamma}_{\mathbf{R}}$ and $\delta(\mathbf{\Delta}_{\mathbf{R}}, \tau\ x = f\ (\mathrm{e});\ \mathrm{s}) = \mathbf{\Delta}_{\mathbf{R}}$, thus by the premises of (AP-App0) we can conclude $\mathbf{\Gamma}_{\mathbf{R}}, \mathbf{\Delta}[f \mapsto \tau_y \xrightarrow{\mathbf{H}_{\mathrm{s}_1}} \tau_f] \vdash \mathrm{s} : \tau_{\mathbf{R}}\ \&\ \mathbf{H}_{\mathbf{R}}$. Finally, $\mathbf{\Pi}_{\mathbf{R}'} = \mathbf{\Pi}_{\mathbf{R}} \asymp \mathbf{\Delta} = \delta(\mathbf{\Delta}_{\mathbf{R}}, \tau\ x = f\ (\mathrm{e});\ \mathrm{s})$.

INDUCTIVE CASE: (AP-App1). We must show that:

$$\begin{aligned}
&(\bar{\sigma}_{\mathbf{R}}, \mathbf{\Pi}_{\mathbf{R}}), \tau\ x = (\mathtt{fn}\ (y : \tau_y)\ \{\mathrm{s}_1\}_{\mathbf{\Gamma}_1});\ \mathrm{s} \xrightarrow{o}_{\mu\mathbf{C}^*} \\
&\qquad\qquad (\bar{\sigma}_{\mathbf{R}'}, \mathbf{\Pi}_{\mathbf{R}}), \tau\ x = \left(\mathtt{fn}\ (y : \tau_y)\ \{\mathrm{s}'_1\}_{\mathbf{\Gamma}'_1}\right);\ \mathrm{s} \wedge \\
&\quad \mathbf{\Gamma}_{\mathbf{R}}, \mathbf{\Delta}_{\mathbf{R}} \vdash \tau\ x = f\ (\mathrm{e});\ \mathrm{s} : \tau_{\mathbf{R}}\ \&\ \mathbf{H}_{\mathbf{R}} \wedge \mathbf{\Pi}_{\mathbf{R}} \asymp \mathbf{\Delta}_{\mathbf{R}} \implies \\
&\qquad \exists \mathbf{H}_{\mathbf{R}'}.\ \mathbf{H}_{\mathbf{R}} \xrightarrow{o}_{\mu\mathscr{C}^*} \mathbf{H}_{\mathbf{R}'} \wedge \\
&\qquad\quad \gamma(\mathbf{\Gamma}_{\mathbf{R}}, \tau\ x = (\mathtt{fn}\ (y : \tau_y)\ \{\mathrm{s}_1\}_{\mathbf{\Gamma}_1});\ \mathrm{s}), \\
&\qquad\qquad \delta(\mathbf{\Delta}_{\mathbf{R}}, \tau\ x = (\mathtt{fn}\ (y : \tau_y)\ \{\mathrm{s}_1\}_{\mathbf{\Gamma}_1});\ \mathrm{s}) \\
&\qquad\qquad\quad \vdash \tau\ x = \left(\mathtt{fn}\ (y : \tau_y)\ \{\mathrm{s}'_1\}_{\mathbf{\Gamma}'_1}\right);\ \mathrm{s} : \tau_{\mathbf{R}}\ \&\ \mathbf{H}_{\mathbf{R}'} \wedge \\
&\qquad\qquad \mathbf{\Pi}_{\mathbf{R}'} \asymp \delta(\mathbf{\Delta}_{\mathbf{R}}, \tau\ x = (\mathtt{fn}\ (y : \tau_y)\ \{\mathrm{s}_1\}_{\mathbf{\Gamma}_1});\ \mathrm{s})
\end{aligned}$$

under the (induction) hypothesis that the property holds for $\mathbf{\Gamma}'_1[y \mapsto \tau_y], \mathbf{\Delta}_{\mathbf{R}} \vdash \mathrm{s}_1 : \tau_{\mathrm{s}_1}\ \&\ \mathbf{H}_{\mathrm{s}_1}$ and $\mathbf{\Gamma}_{\mathbf{R}}[x \mapsto \tau], \mathbf{\Delta}_{\mathbf{R}} \vdash \mathrm{s} : \tau_{\mathbf{R}}\ \&\ \mathbf{H}_{\mathrm{s}}$. By the first premise of (AP-App1) we have that

$$(\bar{\sigma}_{\mathbf{R}}, \mathbf{\Pi}_{\mathbf{R}}), \mathrm{s}_1 \xrightarrow{o}_{\mu\mathbf{C}^*} (\bar{\sigma}_{\mathbf{R}'}, \mathbf{\Pi}_{\mathbf{R}}), \mathrm{s}'_1.$$

By the induction hypothesis, it follows that $\mathbf{H}_{\mathrm{s}_1} \xrightarrow{o}_{\mu\mathbf{C}^*} \mathbf{H}_{\mathrm{s}'_1}$, where $\mathbf{H}_{\mathrm{s}_1}$ and $\mathbf{H}_{\mathrm{s}'_1}$ are the effects of s_1 and s'_1, respectively. Observe now that $\mathbf{H}_{\mathbf{R}} = \mathbf{H}_{\mathrm{s}_1};\ \mathbf{H}_{\mathrm{s}}$ ($\mathbf{H}_{\mathrm{s}}$ is the effect of s), and thus let $\mathbf{H}_{\mathbf{R}'} = \mathbf{H}_{\mathrm{s}'_1};\ \mathbf{H}_{\mathrm{s}}$: by rule $\mu\mathscr{C}^*$-Seq the transition happens and emit observable o, as requested. Note that $\gamma(\mathbf{\Gamma}_{\mathbf{R}}, \tau\ x = (\mathtt{fn}\ (y : \tau_y)\ \{\mathrm{s}_1\});\ \mathrm{s}) = \mathbf{\Gamma}_{\mathbf{R}}$ and $\delta(\mathbf{\Delta}_{\mathbf{R}}, \tau\ x = (\mathtt{fn}\ (y : \tau_y)\ \{\mathrm{s}_1\});\mathrm{s}) = \mathbf{\Delta}_{\mathbf{R}}$, thus by the premises of rule (AP-App1) and by induction hypothesis we can conclude that $\mathbf{\Gamma}_{\mathbf{R}}, \mathbf{\Delta}_{\mathbf{R}} \vdash \tau\ x = (\mathtt{fn}\ (y : \tau_y)\ \{\mathrm{s}_1\});\ \mathrm{s} : \tau_{\mathbf{R}}\ \&\ \mathbf{H}_{\mathbf{R}'}$. Finally, $\mathbf{\Pi}_{\mathbf{R}'} = \mathbf{\Pi}_{\mathbf{R}} \asymp \mathbf{\Delta} = \delta(\mathbf{\Delta}_{\mathbf{R}}, \tau\ x = f\ (\mathrm{e});\ \mathrm{s})$.

INDUCTIVE CASE: (AP-App2). Analogous to the previous case, with observable ϵ instead of o.

INDUCTIVE CASE: (AP-AppNF). This rule is never used since we assume that $\mathbf{H}_{\mathbf{R}} \neq \top$, i.e., contexts fully specify functions needed by programs.

INDUCTIVE CASE: (AP-Call). The theses trivially follow from the premise of the rule.

INDUCTIVE CASE: (AP-If). We have two cases:

1. *Case* $[\mathrm{e}]_{\bar{\sigma}_R} \neq 0$. In this case, we must show that:

$$
\begin{aligned}
&(\bar{\sigma}_R, \Pi_R), \texttt{if (e) \{s}_1\texttt{\} else \{s}_2\texttt{\}; s} \xrightarrow{\epsilon}_{\mu C^*} (\bar{\sigma}_R, \Pi_R), \texttt{s}_1\texttt{; s} \land \\
&\quad \Gamma_R, \Delta_R \vdash \texttt{if (e) \{s}_1\texttt{\} else \{s}_2\texttt{\}; s} : \tau_R \,\&\, H_R \land \Pi_R \asymp \Delta_R \\
&\qquad \implies \exists H_{R'}.\ H_R \xrightarrow{\epsilon}_{\mu\mathscr{C}^*} H_{R'} \land \\
&\qquad\quad \gamma(\Gamma_R, \texttt{if (e) \{s}_1\texttt{\} else \{s}_2\texttt{\}; s}), \\
&\qquad\qquad \delta(\Delta_R, \texttt{if (e) \{s}_1\texttt{\} else \{s}_2\texttt{\}; s}) \vdash \texttt{s}_1\texttt{; s} : \tau_R \,\&\, H_{R'} \land \\
&\qquad\qquad \Pi_R \asymp \delta(\Delta_R, \texttt{if (e) \{s}_1\texttt{\} else \{s}_2\texttt{\}; s}).
\end{aligned}
$$

Observe that $H_R = (H_{s_1} + H_{s_2});\ H_s$ where H_{s_1}, H_{s_2}, and H_s are the effects of $\texttt{s}_1$, $\texttt{s}_2$, and $\texttt{s}$ (resp.). Thus, we can choose $H_{R'} = H_{s_1};\ H_s$ which is reachable by rule $\mu\mathscr{C}^*$-Choice by emitting ϵ. Noting that $\delta(\Delta_R, \texttt{if (e) \{s}_1\texttt{\} else \{s}_2\texttt{\}; s}) = \Delta_R$ and $\gamma(\Gamma_R, \texttt{if (e) \{s}_1\texttt{\} else \{s}_2\texttt{\}; s}) = \Gamma_R$, we need to show that $\Gamma_R, \Delta_R \vdash \texttt{s}_1\texttt{;s} : \tau_R \,\&\, H_{s_1};\ H_s$ which follows by cases on $\texttt{s}_1$ and by the premises of rule (AP-If). Finally, $\Pi \asymp \gamma(\Gamma_R, \texttt{if (e) \{s}_1\texttt{\} else \{s}_2\texttt{\}; s}) = \Gamma_R$ follows by hypothesis.

2. *Case* $[\mathrm{e}]_{\bar{\sigma}_R} = 0$. Follows by commutativity of $+$.

□

Finally, soundness follows easily:

Theorem 5.10 (Soundness)**.** *Let* $\mathbf{W}$ *be a whole* $\mu\mathscr{C}^*$ *program. If* $\emptyset, \emptyset \vdash \mathbf{W} : \tau_W \,\&\, H_W$*, then* $beh(\mathbf{W}) \subseteq beh(H_W)$.

Proof. The theorem holds trivially when $\mathbf{W}$ is a value or when $H_W = \top$. Otherwise, it follows by induction on the length of traces in $beh(\mathbf{W})$ by repeatedly applying Lemma 5.1 and by noting that $\emptyset \asymp \emptyset$, where $\asymp$ is as in the statement of Lemma 5.1. □

5.4.3 *An example, formally*

We now illustrate our instantiation of STV to the concrete case of compilation from μow^* to μC^* with an example.

For the sake of this example, let $Prim = \{\texttt{display}\}$ and $SCName = Prim \cup \{\texttt{send}\}$. Intuitively, `display` shows the given value on the screen, and `send` sends it over the network. Let P be a source program that calls an encryption function provided by the context on a secret value and shows the result of such a call on the screen (via the `display` primitive):

$$
\begin{aligned}
&\textbf{let}\ secret : \texttt{int} = 42\ \textbf{in} \\
&\textbf{let}\ enc : \texttt{int} = encrypt_c\ secret\ \textbf{in} \\
&\quad \texttt{display}\ enc
\end{aligned}
$$

A correct compiler may compile P to the following μC^* program, $\mathbf{P}$:

$$
\begin{aligned}
&\texttt{int}\ secret = 42; \\
&\texttt{int}\ enc = encrypt_c\,(secret); \\
&\texttt{call}\,(\texttt{display}, enc); \\
&\texttt{return}\,()
\end{aligned}
$$

As said in the introduction we want more than correctness: we want to make sure that executing the compiled program in a given environment (i.e., context) does not break security guarantees that the source program had. We apply Algorithm 1 and the analyses above on a few examples of contexts to illustrate when (1) $\mathbf{P}$ can be executed securely; (2) it could but our analysis generates false positives; and (3) it cannot.

According to $aSTV_{TI\text{-}RSC}$, we need to define a partial $\flat$ function that *backtranslates* target contexts into source ones. Actually, we define our $\flat$ function from *abstract target* contexts to *concrete source* ones, so that one can compute the abstract version of a context beforehand (e.g., in the spirit of proof-carrying compilation) and re-use it without recomputing the backtranslation. The backtranslation function is inductively defined on abstract contexts as follows:

$$\flat(\emptyset) = [\cdot]$$

$$\flat(\mathbf{H_C}[f_c \mapsto (\tau \xrightarrow{\mathbf{H}} \tau_1)]) = \mathbf{let}\ f_c = \lambda\, x : \tau .\, s : \tau_1\ \mathbf{in}\ \flat(\mathbf{H_C})$$
$$\text{where } (s, \tau_1) = \flat_S(\mathbf{H}) \text{ if } \flat_S(\mathbf{H}) \text{ is defined}$$

where

$$\begin{aligned}
\flat_S(\bullet^{\texttt{unit}}) &= ((), \texttt{unit}) && \\
\flat_S(\bullet^{\texttt{int}}) &= (42, \texttt{int}) && \\
\flat_S(f_{sc}^{\texttt{unit}}) &= (f_{sc}\ (), \texttt{unit}) && \text{if } f_{sc} \in Prim \\
\flat_S(f_{sc}^{\texttt{int}}) &= (f_{sc}\ 42, \texttt{unit}) && \text{if } f_{sc} \in Prim \\
\flat_S(\mathbf{H_S};\ \mathbf{H'_S}) &= (\mathbf{let}\ x : \tau_s = s\ \mathbf{in}\ s', \tau'_s) && \text{if } \flat_S(\mathbf{H_S}) = (s, \tau_s), \\
&&& \flat_S(\mathbf{H'_S}) = (s', \tau'_s), \text{ and} \\
&&& x \text{ fresh in } s, s' \\
\flat_S(\mathbf{H_S} + \mathbf{H'_S}) &= (\mathbf{if}\ 42\ \mathbf{then}\ s\ \mathbf{else}\ s', \tau) && \text{if } \flat_S(\mathbf{H_S}) = (s, \tau), \text{ and} \\
&&& \flat_S(\mathbf{H'_S}) = (s', \tau)
\end{aligned}$$

The function $\flat_S$ is a partial mapping from histories into concrete source programs. Also, $\flat_S$ carries additional typing information to ensure the well-typedness of the backtranslated source program. For the sake of the example, we made arbitrary choices for values: the backtranslation needs just to testify the existence of a source context, without necessarily trying to correctly implement the concrete behavior of the original one. Note that this is a source of false positives, e.g., think of a case in which the target context never takes a branch of a conditional, whereas its backtranslated counterpart always does.

Also note that we omit the algorithm for checking the inclusion between target and source abstract behaviors, since it is standard. Indeed, both the set of behaviors of abstract source programs and the set of behaviors of target ones are regular languages (the first being a finite set and the second being the language generated by a history expression with no recursion).

The last step before applying Algorithm 1 is to analyze the program and extract its abstract counterpart. Rather straightforwardly, applying the rules from Figure 46 we get $\mathbf{H_P}$:

$$\hbar_{encrypt_c};\ \texttt{display};\ \bullet^{\texttt{unit}};\ \bullet^{\texttt{unit}}.$$

Note the variable $\hbar_{encrypt_c}$, that denotes the fact that the implementation of $encrypt_c$ is still unknown.

We are finally ready to apply the STV algorithm to different contexts.

A GOOD CONTEXT Consider the following μC^* context C_{good}, implementing the $encrypt_c$ as the encryption function of the *Caesar's cipher* (i.e., performed by adding 3 to the clear text):

$$\begin{array}{l} \textbf{fun}\ encrypt_c\,(s : \texttt{int}) : \texttt{int}\ \{ \\ \quad \texttt{return}\ s + 3 \\ \};[\cdot] \end{array}$$

Analyzing C_{good}, we get:

$$[encrypt_c \mapsto (\texttt{int} \xrightarrow{\bullet^{\texttt{int}}} \texttt{int})].$$

Since our backtranslation is defined on abstract contexts, we now need to check whether C_{good} can be backtranslated. First, note that the backtranslated body of $encrypt_c$ is $b_S(\bullet^{\texttt{int}}) = (42, \texttt{int})$ thus the backtranslation $C_{good} = b(C_{good})$ is defined as

$$\textbf{let}\ encrypt_c = \lambda\,x : \texttt{int}.\ 42 : \texttt{int}\ \textbf{in}\ [\cdot].$$

Also, it is easy to compute the linking at the target abstract level, which equals to

$$\bullet^{\texttt{int}};\ \texttt{display};\ \bullet^{\texttt{unit}};\ \bullet^{\texttt{unit}}.$$

Linking the backtranslated context with P, we get that the behavior of its analyzed version is the set of all prefixes of the following trace:

$$\epsilon \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot \texttt{display} = \texttt{display}.$$

Thus, it easily follows that the behavior at the source includes that of $C_{good}[H_P]$ (up to the neutral element of traces, ϵ) and that, according to our algorithm, it is safe to execute $\mathbf{P}$ in C_{good} w.r.t. P.

A FALSE POSITIVE Consider the following μC^* context C'_{good}, implementing the $encrypt_c$ as the encryption function of the *Caesar's cipher*, but such that it returns -1 and displays its parameter on the screen when it is different from 0:

$$\begin{array}{l} \textbf{fun}\ encrypt_c\,(s : \texttt{int}) : \texttt{int}\ \{ \\ \quad \texttt{if}\ (s)\ \{ \\ \quad\quad \texttt{call}\ (\texttt{display}, s); \\ \quad\quad \texttt{return}\ -1 \\ \quad \}\ \texttt{else}\ \{ \\ \quad\quad \texttt{return}\ s + 3 \\ \};[\cdot] \end{array}$$

Analyzing C'_{good}, we get:

$$[encrypt_c \mapsto (\texttt{int} \xrightarrow{(\texttt{display};\ \bullet^{\texttt{int}}) + \bullet^{\texttt{int}}} \texttt{int})].$$

Following a reasoning similar to the one above, we get $\mathsf{C}'_{\mathsf{good}}$:

$$
\begin{aligned}
&\textbf{let } \mathit{encrypt}_c = \lambda x : \texttt{int}.\\
&\quad (\textbf{let } y : \texttt{int} =\\
&\qquad \textbf{if } \mathit{42} \textbf{ then}\\
&\qquad\quad \textbf{let } z : \texttt{unit} = \texttt{display } \mathit{42} \textbf{ in } \mathit{42}\\
&\qquad \textbf{else } \mathit{42}\\
&\quad \textbf{in } \mathit{42}) : \texttt{int}\\
&\textbf{in } [\cdot]
\end{aligned}
$$

Now, the linking at the abstract target level equals to

$$((\texttt{display};\ \bullet^{\texttt{int}}) + \bullet^{\texttt{int}});\ \texttt{display};\ \bullet^{\texttt{unit}};\ \bullet^{\texttt{unit}}.$$

Instead, the behavior of the analysis of $\mathsf{C}'_{\mathsf{good}}[\mathsf{P}]$ is the set of all prefixes of the following trace:

$$\epsilon \cdot \epsilon \cdot \epsilon \cdot \texttt{display} \cdot \epsilon \cdot \texttt{display} = \texttt{display} \cdot \texttt{display}$$

Thus, it is easy to see that the behavior at the source *does not include* that of $\mathbf{C}'_{\mathbf{good}}[\mathbf{H_P}]$ and that, according to our algorithm, it *might* not be secure to execute $\mathbf{P}$ in $\mathbf{C}_{\mathbf{good}}$ w.r.t. P. Indeed, this is a *false positive*: a more precise backtranslation (e.g., one acting directly on concrete target contexts) combined with a more precise type and effect system (e.g., one also keeping track of guards of conditionals) would allow us to classify correctly this case.[2]

AN ACTUALLY MALICIOUS CONTEXT Finally, consider the target context $\mathbf{C}_{\mathbf{evil}}$ that implements $\mathit{encrypt}_c$ as the encryption function of the *Caesar's cipher*, but that also sends the value of its parameter on the network:

$$
\begin{aligned}
&\textbf{fun } \mathit{encrypt}_c\,(s : \texttt{int}) : \texttt{int}\ \{\\
&\quad \texttt{call}\,(\texttt{send}, s);\\
&\quad \texttt{return } s + \mathit{3}\\
&\};[\cdot]
\end{aligned}
$$

In this case, our backtranslation function is undefined since `send` is not a primitive in $\mu o w^*$. Thus, our algorithm correctly tells us that it may be unsafe to execute $\mathbf{P}$ in $\mathbf{C}_{\mathbf{evil}}$.

5.5 DISCUSSION

ASSUMPTIONS AND LIMITATIONS OF THE STV FRAMEWORK Throughout the chapter, we assumed that the partial target program is trusted and that the compiled code of the context is available at link time. In this case, we can extract the abstraction of their behavior. However, the code of the context may be very large and its analysis may become infeasible in practice, e.g., because the time spent in the analysis grows too high. A way for overcoming this issue could be to compute the abstraction of the context beforehand (e.g.,

2 In our case the simplicity of the type and effect system is to blame: the backtranslated context just takes one of the two branches of the conditional, whereas in history expressions we *always* consider both branches as possible.

during its compilation), and re-use it on need. Of course, this would require to certify that the provided context abstraction is correct and that the code of the context does not change over time. A more realistic alternative could be to reduce the size of the context that needs to be checked at link time by certifying some of its parts using other mechanisms, e.g., remote attestation [74]. In this way, the certified context parts could be included in the trusted computing base. Another way to limit the size of the context, could be to resort to other dynamic defense mechanisms to complete the protection of the program. For example, a possible implementation could couple STV together with enclaved execution [148, 170]. More precisely, STV may check that a partial program P is not linked and loaded in the same enclave as possibly malicious code, leaving to the enclave mechanism the protection from other malicious (system) software. Studying a possible integration between STV and enclaves is an interesting future work. In conclusion, an actual implementation of our framework should trade security guarantees off for performance (as usual in program analysis) and should also cooperate with existing countermeasures.

A second assumption made is that the threat model we consider is *static*, meaning that we classify a context as malicious or not just by inspecting its code. Indeed, we do not consider cases in which a harmless context becomes malicious at run-time, e.g., due to undefined behavior. Again, a more general threat model could be faced by integrating STV with other defense mechanisms, e.g., software sandboxing [199], software fault isolation [232], or robustly safe compartmentalizing compilation [8]. An interesting future work consists of relaxing the assumption on the separation between the trusted partial program and the (malicious) context. In such a setting, we may consider a set of mutually distrustful programs that can be compromised at run-time and become attack vectors themselves [8].

Another limitation of our STV framework is that it currently only works for the robust preservation of safety properties. We believe that extending the framework to safety hyperproperties is straightforward, whereas the applicability of STV to other security properties (like liveness hyperproperties) is still under investigation.

USING STV FOR MORE COMPLEX USE CASES The most evident limitation of our current use case is that the languages are not Turing-complete. We considered these languages since we think that the problem of checking the robust preservation of all the safety properties is of practical interest in the scenario of Protzenko et al. [193]. Actually, with a limited amount of additional work, it is possible to apply STV to richer languages. Indeed, in [56] we use STV to check whether the isolation properties provided by a Turing-complete compartmentalized source language (inspired by that of Juglaret et al. [121]) are preserved when compiling to a low-level language without isolation guarantees.

Another limitation of our current use case is that our analyses lack some precision, and that affects the rate of false positives (i.e., safe programs that are flagged as insecure). A major source of false positives in our analyses is that they track the control-flow of programs but not their data-flow (roughly, how programs manipulate data during their execution). An immediate way of mitigating this issue could be to combine the current analyses with ones that also track the data (e.g., data-flow analyses [166] or abstract interpretation [80]). By doing that, we would also be able to use STV on languages endowed with richer observables, e.g., recording system call arguments and their return values. One possible downside of these more accurate analyses is that they can be computationally expensive.

However, this is the price we have to pay if we want a *sound* push-button verification procedure and we believe that this issue can mitigated by leveraging the results from the (very vast) literature on program analysis and verification. In the future, we plan to investigate more in-depth which classes of static analyses may be interesting for STV.

A further direction of investigation is to understand how the requirement of the inclusion between an *over-approximation* of the behavior of a target program and an *under-approximation* of that of a source one may impact practical implementations of our framework. To us this requirement is reasonable, since the under-approximation of the behavior of the source also under-approximates the behavior of the backtranslated context. Thus, one can choose a backtranslation that produces contexts having more behaviors than (strictly) necessary. Moreover, the source and the target language (usually) operate at very different levels of abstractions and the gap between them may be useful to make the above requirement more realistic.

5.6 CONCLUSIONS

In this chapter we introduced the idea of *secure translation validation* (STV), which constitutes a step forward in the direction of secure compilation and its automatic enforcement. In doing that, we introduced a simple and general theory for STV and mechanized it in the Coq theorem prover. Summing up, we built upon the principle of *robust safety preservation* by Abate et al. [10] and adapted it to be used in the setting of translation validation. Also, we provided sufficient conditions for the STV to work correctly, i.e., ensuring that no non-secure code is ever executed. Finally, we illustrated our idea on a simple case study, inspired by the recent work on the KreMLin compiler by Protzenko et al. [193]. Differently from them, we consider the compiler as a *black-box* and we just compared the behavior of the compiled program in a given environment with that of the corresponding source program. Of course, one could *directly* analyze the target program to check whether it enforces or not some security properties. But that would mean to be unable to (easily) check the security of compiled programs for potentially infinite classes of hyperproperties and to be unable to relate insecure behaviors at the target with that of the source.

We are aware that more realistic use cases (and implementations) are in order to further assess the real-world applicability of the approach. As a first case study, in [56] we applied the STV framework to ensure that a carefully defined compiler preserves the isolation properties between a Turing-complete compartmentalized source language and a low-level language without any mechanism that enforce isolation.

Related work

Traditionally, secure compilation follows the approach by Abadi [2] and define secure compilation as a full abstraction property of compilers. Indeed, in the last two decades most of the research has been directed towards defining secure compilers and proving them fully abstract. Recently, Patrignani and Garg [179] highlighted some shortcomings of the traditional approach and worked towards the definition of alternative approaches to secure compilation [10, 180]. Differently from full abstraction, that requires preservation and reflection of a congruence relation under compilation, their principles are tailored to the specific family of (hyper)properties that one wants to preserve. In particular, in this chapter we built on *robust safety preservation* [10, 180], prescribing the preservation

of robust satisfaction of safety properties. Recently, Abate et al. [9] further extended the notions of compiler correctness and security (as introduced in [10]) to deal with source and target languages whose observables are different and whose traces are related by a relation $\sim$. For a longer summary on secure compilation, we refer the interested reader to Chapter 2.

We first proposed to apply translation validation in the field of secure compilation in [52]. Independently, Namjoshi and Tabajara [160] developed a translation validation schema based on refinements. They develop two automaton-based refinement schemata to handle hyperproperty preservation in the case of passive attackers. In both schemata the idea is roughly as follows: if a *witness* refinement relation exists between the (suitably encoded) source and target systems, then the hyperproperty of choice has been preserved.

Our idea of abstracting the behavior of programs using history expressions so as to be able to prove their properties dates back to Bartoletti et al. [26–28], which in turn built upon history effects by Skalka and Smith [214]. Actually, history expressions have been used in a variety of applications. For instance, [27, 28] use histories to check access control policies statically. Also, history expressions have been used for checking secure web services composition and for web service orchestration [24–26, 29, 31, 78]. Furthermore, Bartoletti et al. [30, 32] applied history expressions to resource usage analysis. More recently, histories have also been applied in the scenario of context-oriented programming [115] for checking both adaptation of programs [82–85, 101] and their security [40, 41, 50].

6

SECURE COMPILATION AGAINST MICRO-ARCHITECTURAL ATTACKS

Isolation mechanisms like process isolation, virtual memory, or enclaved execution [148] are among the standard features of modern microprocessors. Indeed, isolation mechanisms are intended to confine the interactions between different programs to a well-defined communication interface. In this way, different parties can safely deploy their applications on the same system without the need of mutual trust.

However, a recent wave of attacks has shown that many of these isolation mechanisms can be attacked via *software-exploitable side-channels*. Over the past few years, many major isolation mechanisms have been successfully broken using side-channels. Among others: Meltdown [142] broke the user/kernel isolation hindering the confidentiality of data of victim programs; Spectre [128] did the same by exploiting pitfalls in the process isolation mechanism; and Foreshadow [46] broke confidentiality of enclaved executions on Intel processors. Actually, side-channels allowed attackers to violate integrity [126, 158, 221] and confidentiality of programs on both high-end [46, 102, 128, 142] and small microprocessors [225].

Because of the complexity and variety of software-exploitable side-channel attacks, we cannot expect *silver-bullet solutions* against them. Indeed, these attacks often rely on, or even exploit, specific hardware features and implementation details. Therefore, the attack surface exposed by modern microprocessors is wide and hard to fix (see e.g., Canella et al. [64] for an overview of some of these attacks). Also, the potential attack vectors vary with the attacker model that a specific isolation mechanism considers. For instance: enclaved execution is designed to protect enclaved code from malicious operating system software, whereas process isolation assumes that the operating system is trusted and not under the control of the attacker. As a consequence, protection against software-exploitable side-channel attacks is much harder for enclaved execution [239]. On top of that, solutions will also depend on performance versus security trade-offs, e.g., whether it is acceptable or not to disable some performance-enhancing features.

In this chapter we study how to design and prove secure these countermeasures. In particular, we rigorously study the resistance of enclaved execution on small microprocessors against interrupt-based attacks [48, 114, 225]. We base our study on the existing open-source Sancus platform [168, 170], a small microprocessor with predictable timing of individual instructions, that supports *non-interruptible* enclaved execution. Through a variety of attacks enabled by supporting interruptibility of enclaves, we illustrate that it is non-trivial to achieve security. Next, we provide a formal model of the existing Sancus and we extend it with interrupts. To show that this extension does not break isolation properties we prove our countermeasure secure by instantiating full abstraction [2] (briefly summarized in Chapter 2).

Roughly, we show that what the attacker can learn from (or do to) an enclave is exactly the same *before* and *after* adding the support for interrupts. In other words, adding interruptibility does not open *new avenues of attack*.

In summary, in this chapter we propose:

- A specific design for extending Sancus, an existing enclaved execution system, with interrupts;
- To use full abstraction [2, 177] as a formal criterion of what it means to maintain the security of isolation mechanisms under processor extensions. Also, we instantiate it for proving that the mechanism of enclaved execution, extended to support interrupts, complies with our security definition; and
- A specialization of the *backtranslation* proof technique (recall Chapter 2 or see [176]) to encode the attack logic within the attacker-controlled device. The novelty of our backtranslation consists in reflecting the capability the attacker has to exploit the device it controls (that has no limit in size), when the memory of the attacker is inadequate (limited to 64 KB).

The chapter is structured as follows: in Section 6.1 we provide background information on enclaved execution and interrupt-based attacks. Section 6.2 provides an informal overview of our approach. Section 6.3 introduces our formalization, and Section 6.4 presents the semantics of Sancus without and with interrupts. The proof that enclaved executions are resistant to interrupt-based attacks is in Section 6.5 (auxiliary and technical definitions and proofs are presented in full detail in the Appendix B). Section 6.6 shows how our full abstraction result implies some other security notions when tailored to our setting. In Section 6.7 we discuss limitations of our proposal and Section 6.8 concludes the chapter and overviews the related work.

Finally, our co-authors from KU Leuven implemented the mechanism we describe in this chapter in the real Sancus architecture. We discuss it in detail in our recent papers [58, 59] and its code is open source and available online at `https://github.com/sancus-pma/sancus-core/tree/nemesis`.

6.1 enclaves and interrupt-based attacks

6.1.1 *Enclaved execution*

Recall the notion of *enclaves* from Chapter 2. Roughly, an enclave is a protected section of memory initialized with a software module, which is isolated from all other software on the same platform and whose correct initialization can be *remotely attested*. We will focus here just on the isolation aspect of enclaves. In fact, even though remote attestation is important for the secure initialization of enclaves and for setting up secure communication channels with them, it does not play an important role for the interrupt-driven attacks that we study in this chapter. For further details on remote attestation and secure communication, we refer the interested reader to [79] for large systems and to [129, 170] for small ones.

An *enclaved software module* (or simply *module*) consists of two contiguous memory sections, a *code section*, initialized with the machine code of the module, and a *data section*. The data section is initialized to zero, and the loading of confidential data happens through a secure channel, after attesting the correct initialization of the module. For instance, confidential data can be restored from cryptographically sealed storage, or can be obtained from a remote trusted party.

The enclaved execution platform guarantees that: (1) the code and data section of an enclave are *only* accessible while executing code from the code section; and (2) the code section can only be entered through one or more designated *entry points*. These isolation guarantees are simple, but they offer the useful property that *data of a module can only be manipulated by code of the same module*, i.e., an encapsulation property similar to what programming languages offer through classes and objects. Actually, untrusted code may reside in the same address space as the enclave, but outside its code and data sections. Untrusted code can only interact with the enclave by jumping to an entry point. The enclave can return control (and computation results) to the untrusted code by jumping back out.

6.1.2 *Interrupt-based attacks*

As explained above, the attacker model considered for enclaved execution is a very strong attacker that controls all the other software on the platform, including privileged system software. Even though the isolation mechanisms for enclaves are well-understood at the architectural level (including some successful formal verification efforts [99, 171]), it is still a challenge to protect enclaves against side-channels. For instance, the recent research on *controlled-channel attacks* [48, 49, 136, 154, 225, 239] proved that the increased control of the attacker over privileged software allows the adversary to exploit a new class of powerful, low-noise side-channels.

An additional consequence of this strong attacker model is that the scheduling and handling of interrupts is also under the control of the adversary. For instance, this power has been exploited to single-step through an enclave [48], or to mount a new class of *interrupt latency* attacks [114, 225] that derive individual enclaved instruction timings from the time it takes to dispatch to the interrupt handler of the untrusted operating system. We provide concrete examples of interrupt-based attacks in the next section, after detailing our model of enclaved execution.

Pure interrupt-based attacks such as interrupt latency measurements are the *only* known controlled-channel attack against low-end enclaved execution platforms and all the designs supporting interruptibility of enclaves [73, 129] are vulnerable to them. Moreover, they have been shown to be very powerful, for instance Van Bulck et al. [225] used them to efficiently extract secrets (like passwords or PINs) from embedded enclaves. Some designs avoid the problem of interrupt-based attacks by completely disabling interrupts during enclaved executions [170, 171]. However, this solution has the important downside that system software can no longer guarantee availability: if an enclaved module goes into an infinite loop, the system cannot progress.

6.2 OVERVIEW OF OUR APPROACH

We set out to design an interruptible enclaved execution system that is provably resistant against interrupt-based attacks. This section discusses our approach informally, later sections discuss a formalization with security proofs.

As said above we base our design on Sancus [170], an existing open-source enclaved execution system. We first describe our Sancus model, and discuss how extending Sancus with interrupts leads to the attacks mentioned in Section 6.1.2. In other words, we show how extending Sancus with interrupts breaks some of the isolation guarantees provided by the original architecture.

Instr. i	Meaning	Cycles	Size (Words)
RETI	Returns from interrupt.	5	1
NOP	No-operation.	1	1
HLT	Halt.	1	1
NOT $\mathtt{r}$	$\mathtt{r} \leftarrow \neg\mathtt{r}$. (Emulated in MSP430)	2	2
IN $\mathtt{r}$	Reads word from the device and puts it in $\mathtt{r}$.	2	1
OUT $\mathtt{r}$	Writes word in register $\mathtt{r}$ to the device.	2	1
AND $\mathtt{r}_1$ $\mathtt{r}_2$	$\mathtt{r}_2 \leftarrow \mathtt{r}_1 \,\&\, \mathtt{r}_2$.	1	1
JMP $\&\mathtt{r}$	Sets $\mathtt{pc}$ to the value in $\mathtt{r}$.	2	1
JZ $\&\mathtt{r}$	Sets $\mathtt{pc}$ to the value in $\mathtt{r}$ if bit 0 in $\mathtt{sr}$ is set.	2	1
MOV $\mathtt{r}_1$ $\mathtt{r}_2$	$\mathtt{r}_2 \leftarrow \mathtt{r}_1$.	1	1
MOV $@\mathtt{r}_1$ $\mathtt{r}_2$	Loads in $\mathtt{r}_2$ the word starting in location pointed to by $\mathtt{r}_1$.	2	1
MOV $\mathtt{r}_1$ $0(\mathtt{r}_2)$	Stores the value of $\mathtt{r}_1$ starting at location pointed to by $\mathtt{r}_2$.	4	2
MOV $\#w$ $\mathtt{r}_2$	$\mathtt{r}_2 \leftarrow w$.	2	2
ADD $\mathtt{r}_1$ $\mathtt{r}_2$	$\mathtt{r}_2 \leftarrow \mathtt{r}_1 + \mathtt{r}_2$.	1	1
SUB $\mathtt{r}_1$ $\mathtt{r}_2$	$\mathtt{r}_2 \leftarrow \mathtt{r}_1 - \mathtt{r}_2$.	1	1
CMP $\mathtt{r}_1$ $\mathtt{r}_2$	Zero bit in $\mathtt{sr}$ set if $\mathtt{r}_2 - \mathtt{r}_1$ is zero.	1	1

Table 3: Summary of the assembly language considered.

Then, we propose a formal security criterion that defines what it means for interruptibility to *preserve the isolation properties*, and we illustrate that definition with examples.

Finally, we propose a design for an interrupt handling mechanism that is resistant against the considered attacks and that satisfies our security definition. Crucial to our design is the assumption that the timing of individual instructions is predictable, which is typical of "small" microprocessors, like Sancus. Our approach of ensuring that the same attacks are possible before and after an architecture extension is tailored here on a specific architecture, however we expect it to be applicable in more complex settings too, as we briefly explained in Section 6.7.

6.2.1 *Sancus model*

PROCESSOR Sancus is based on the TI MSP430 16-bit microprocessor [119], with a classic von Neumann architecture where code and data share the same address space. We formalize the subset of instructions summarized in Table 3 that is rich enough to model all the attacks on Sancus we care about (see also Section 6.7). We have a subset of memory-to-register and register-to-memory transfer instructions; a comparison instruction; an unconditional and a conditional jump; and basic arithmetic instructions.

MEMORY Sancus has a byte-addressable memory of at most 64KB, where a finite number of enclaves can be defined. The bound on the number of enclaves is a parameter set at processor synthesis time. In our model, we assume that there is a single enclave, made of a *code section*, initialized with the machine code of the module, and a *data section*. The data section is securely provisioned with data by relying on remote attestation and secure

communication, not modeled here as they play no role in the interrupt-based attacks of interest in this paper. Instead, our model allows direct initialization of the data section with confidential enclave data. The rest of the memory is *unprotected memory* and is under full control of the attacker.

Enclaves have a single entry point; the enclave can only be entered by jumping to the first address of the code section. Multiple *logical entry points* can easily be implemented on top of this single physical entry point. Control flow can leave the enclave by jumping to any address in unprotected memory. Obviously, a compiler can implement higher-level abstractions such as enclave function calls and returns, or out-calls from the enclave to functions in the untrusted code [170].

Sancus enforces memory access control based on program counter. If the program counter points to unprotected memory, the processor cannot access any memory location within the enclave: the only way to interact with the enclave is to jump to its entry point. If the program counter is within the code section of the enclave, the processor can only access the enclave data section for reading/writing and the enclave code section for execution. This access control is faithfully rendered in our model, see Section 6.3.8 for the full definition of the relevant predicate.

I/O DEVICES Sancus uses memory-mapped I/O to interact with peripherals. One important example of a peripheral for the attacks we study is a cycle-accurate timer, which allows software to measure time in terms of the number of CPU cycles. In our model, we include a single very general I/O device that behaves as a state machine running synchronously to CPU execution. In particular, it is trivial to instantiate this general I/O device to a cycle-accurate timer.

Instead of modeling memory-mapped I/O, we introduce the two special instructions `IN` and `OUT` that allow writing/reading a word to/from the device (see Table 3). Actually these instructions could be implemented as memory operations, at the price of dealing with special cases in the execution semantics. For instance, software could read the current cycle timer value from a timer peripheral by using the `IN` instruction. The I/O devices can request to interrupt the processor with single-cycle accuracy.

The original Sancus disables interrupts during enclaved execution. One of the key objectives of this chapter is to propose a Sancus extension that does handle such interrupts without weakening security.

6.2.2 *Security definitions*

ATTACKER MODEL An attacker controls the entire *context* of an enclave, i.e., (1) the whole unprotected memory (including code interacting with the enclave, as well as data in unprotected memory); and (2) the connected device. This is the standard attacker model for enclaved execution. In particular, it implies that the attacker has complete control over the interrupt service routines, i.e., pieces of code that the CPU invokes when an interrupt is raised.

CONTEXTUAL EQUIVALENCE FORMALIZES ISOLATION Informally, our security objective is extending the Sancus processor without weakening the isolation it provides to enclaves. Intuitively, isolation of enclaved execution should guarantee that attackers cannot see "inside" an enclave, so enabling them to "hide" enclave data or implementation details

from the attacker. We formalize this concept of isolation precisely by using the notion of *contextual equivalence* or *contextual indistinguishability*, as done Abadi [2]. Two enclaved modules M_1 and M_2 are contextually equivalent if there exists no context that tells them apart. We discuss this on the following example.

Example 6.1 (Start-to-end timing)**.** *The following enclave compares a user-provided password in* R_{15} *with a secret in-enclave password at address* pwd_addr*, and stores the user-provided value in* R_{14} *into the enclave location at* store_addr *if the user password was correct.*

```
 1 enclave_entry:
 2     /* Load addresses for comparison */
 3     MOV #store_addr, R10
 4     MOV #access_ok, R11
 5     MOV #endif, R12
 6     MOV #pwd_addr, R13
 7     /* Compare user vs. enclave password */
 8     MOV @R13, R13
 9     CMP R13, R15
10     JZ &R11
11 access_fail: /* Password fail: return */
12     JMP &R12
13 access_ok: /* Password ok: store user val */
14     MOV R14, 0(R10)
15 endif: /* Clear secret enclave password */
16     SUB R13, R13
17 enclave_exit:
```

First, consider the case in which attackers have no access to a timer device. In such a case the above enclave code successfully hides the in-enclave password. Indeed, if we take modules M_1 *and* M_2 *to be two instances of the above only differing in the value of the secret password, then* M_1 *and* M_2 *are indistinguishable for any context that does not have access to a cycle-accurate timer: all a context can do is calling the entry point, but it gets no indication whether the user-provided password was correct. This means that the enclave isolation successfully "hides" the password.*

Things change with the help of a cycle-accurate timer. In this scenario, an attacker can distinguish M_1 *and* M_2 *by creating a context that measures the start-to-end execution time of an enclave call. Right before jumping to the entry-point of the enclave the context reads and stores the value of the timer. Afterwards, when the enclave exits the context reads the timer again and computes the total time spent in the enclave.*

We represent enclaved executions as a list of individual instructions along with the number of cycles they take (recall Table 3, that conveniently specifies how many cycles it takes to execute a given instruction). The first possible control-flow path is for the access_ok *branch of the above program:*

```
MOV #store_addr, R10   →  2 cycles
MOV #access_ok, R11    →  2 cycles
MOV #endif, R12        →  2 cycles
MOV #pwd_addr, R13     →  2 cycles
MOV @R13, R13          →  2 cycles
CMP R13, R15           →  1 cycle
JZ &R11                →  2 cycles
MOV R14, 0(R10)        →  4 cycles
SUB R13, R13           →  1 cycle
```

and takes a total of 18 *cycles. The other possible path is for the* `access_fail` *branch and goes as follows*

```
MOV #store_addr, R10   →  2 cycles
MOV #access_ok, R11    →  2 cycles
MOV #endif, R12        →  2 cycles
MOV #pwd_addr, R13     →  2 cycles
MOV @R13, R13          →  2 cycles
CMP R13, R15           →  1 cycle
JZ &R11                →  2 cycles
JMP &R12               →  2 cycles
SUB R13, R13           →  1 cycle
```

requiring just 16 *cycles. The context can then distinguish the two control-flow paths, and hence* M_1 *from* M_2*. Finally, by launching a brute-force attack [106], attackers can also extract the secret password.*

The above (slightly artificial) example illustrates how contextual equivalence formalizes isolation. It also shows that the original Sancus already has some side-channel vulnerabilities under our attacker model. Since we assume attackers can use any I/O device, they have the ability to setup a timer device and mount the start-to-end timing attack we discussed.

It is important to note that it is *not* our goal to close these existing side-channel vulnerabilities in Sancus. Our objective is to make sure that adding interrupts does not introduce *additional* side-channels, i.e., this does not *weaken* the isolation properties of Sancus.

For existing side-channels, countermeasures can be applied by the enclave programmer or by a security-aware compiler, e.g., [66, 190, 238]. For instance, the programmer can balance out the various secret-dependent control-flow paths as in Example 6.2.

Example 6.2 (Interrupt latency). *Consider the program of Example 6.1, balanced in terms of overall execution time by adding two* `NOP` *instructions before Line 14:*

```
 1 enclave_entry:
 2     /* Load addresses for comparison */
 3     MOV #store_addr, R10
 4     MOV #access_ok, R11
 5     MOV #endif, R12
 6     MOV #pwd_addr, R13
 7     /* Compare user vs. enclave password */
 8     MOV @R13, R13
 9     CMP R13, R15
10     JZ &R11
11 access_fail:
12     /* Password fail: constant time return */
13     NOP
14     NOP
15     JMP &R12
16 access_ok: /* Password ok: store user val */
17     MOV R14, 0(R10)
18 endif: /* Clear secret enclave password */
19     SUB R13, R13
20 enclave_exit:
```

The control-flow path for `access_ok` *remains unchanged, whereas that for* `access_fail` *becomes:*

```
MOV #store_addr, R10   →  2 cycles
MOV #access_ok, R11    →  2 cycles
MOV #endif, R12        →  2 cycles
MOV #pwd_addr, R13     →  2 cycles
MOV @R13, R13          →  2 cycles
CMP R13, R15           →  1 cycle
JZ &R11                →  2 cycles
NOP                    →  1 cycle
NOP                    →  1 cycle
JMP &R12               →  2 cycles
SUB R13, R13           →  1 cycle
```

Since now both branches take the same amount of time (18 *cycles), the start-to-end timing attack is mitigated.*

INTERRUPTS CAN WEAKEN ISOLATION We now show that a straightforward implementation of interrupts in the Sancus processor would significantly weaken isolation. Consider an implementation of interrupts similar to the TI MSP430: on arrival of an interrupt, the processor first completes the ongoing instruction, and then it jumps to an interrupt service routine.

The program in Example 6.2 is secure on Sancus without interrupts. However, it is not secure against a malicious context that can schedule interrupts to be handled while the enclave executes. To see why, assume that an interrupt is scheduled by the malicious context to arrive within the first cycle after the conditional jump at Line 10. If the jump was taken then the current instruction is the 4-cycles `MOV`at Line 17, otherwise the current instruction is the 1-cycle `NOP` at Line 13. Now, since the interrupt service routine of the attacker will only be called *after* completion of the current instruction, the adversary observes an interrupt latency difference of 3 cycles, depending on the secret branch condition inside the enclave. Van Bulck et al. [225] have shown that this difference in latency when handling interrupts can be practically measured to precisely reconstruct individual enclave instruction timings on both high-end and low-end enclave processors.

Using this attack technique, illustrated in Figure 50, a context can again distinguish two instances of the module with a different password, and hence the addition of interrupts has *weakened* isolation.

One could be tempted to fix the above timing leakage by modifying the implementation of interrupt handling in the processor to always dispatch interrupt service routines in constant time `T`, i.e., regardless of the duration of the interrupted instruction. However, this is a necessary but not sufficient condition:

Example 6.3 (Resume-to-end timing)**.** *Consider the program from Example 6.2 executed on a processor which always dispatches interrupts in constant time* `T`*. The attacker schedules an interrupt to arrive in the first cycle after the* `JZ` *instruction, yielding constant interrupt latency* `T`*. Next, the context resumes the enclave and measures the time it takes to let the enclave run to completion* without *further interrupts. While interrupt latency timing differences are properly masked, the time to complete enclave execution after resume from the interrupt is* 1 *cycle for the* `access_ok` *path and* 4 *cycles for the* `access_fail` *path (cfr. Figure 50).*

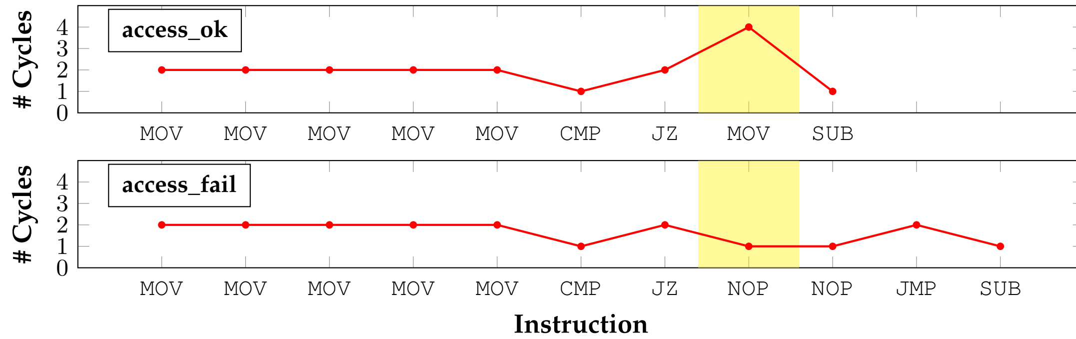


Figure 50: Interrupt latency traces corresponding to the conditional control-flow paths in Example 6.2. When interrupting after the 7th instruction, the adversary observes a distinct latency difference for the 4-cycle MOV instruction vs. the 1-cycle NOP instruction.

Another possibility of attack comes from *interrupt-counting*:

Example 6.4 (Interrupt-counting attack)**.** *An alternative way to attack the program from Example 6.2 even when interrupt latency is constant is the following, is to* count *how often the enclave execution can be interrupted, e.g., by scheduling a new interrupt* 1 *cycle after resuming from the previous one. Since interrupts are handled on instruction boundaries, this lets the attacker count the number of instructions executed in the enclave, and hence to distinguish the two possible control-flow paths (cfr. Figure 50). Such interrupt-counting attacks [154] have been shown to be dangerous even on enclaved execution systems where timing measurements are noisy (like Intel SGX).*

DEFINING THE SECURITY OF AN EXTENSION The examples above show how a new processor feature (like interrupts) can weaken isolation of an existing isolation mechanism (like enclaved execution), and this is exactly what we want to avoid. Here we propose and implement a defense against these attacks and formally prove that it is indeed secure. Our security definition should now be clear: given an original system (like Sancus), and an extension of that system (like interruptible Sancus), that extension is secure if and only if it does not change the contextual equivalence of enclaves. Enclaves that are contextually equivalent in the original system must be contextually equivalent in the extended system and vice versa (we shall formalize this as a *full abstraction* property later on).

6.2.3 *Secure interruptible Sancus*

Designing an interrupt handling mechanism that is secure according to our definition above is quite subtle. We illustrate some of the subtleties. In particular, we provide an intuition on how an appropriate use of time padding can handle the various attacks discussed above. We also discuss how other design aspects are crucial for achieving security. In this section, we just provide intuition and examples. The ultimate argument that our design is secure is our proof, discussed later.

PADDING We already discussed that it is insufficient for security to naively pad interrupt latency to make it constant, thus we need a padding approach that handles all kinds of attacks.

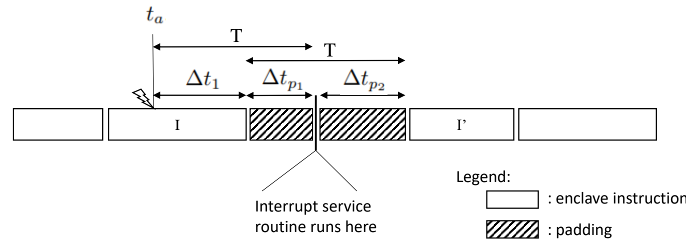


Figure 51: The secure padding scheme.

Our padding scheme (see Figure 51) is as follows. Suppose the attacker schedules the interrupt to arrive at t_a, during the execution of instruction I in the enclave. Let Δt_1 be the time needed to complete the execution of I. To make sure the attacker cannot learn anything from the interrupt latency, we introduce padding for Δt_{p_1} cycles where Δt_{p_1} is computed by the interrupt handling logic such that $\Delta t_1 + \Delta t_{p_1}$ is a constant value T. This value T should be chosen as small as possible to avoid wasting unnecessary time, but must be larger than or equal to the maximal instruction cycle time `MAX_TIME` (to make sure that no negative padding is required, even when an interrupt arrives right at the start of an instruction with the maximal cycle time). This first padding ensures that an attacker always measures a constant interrupt latency.

However, the above mitigation is not enough, as an attacker can now measure resume-to-end time as in Example 6.3. Thus, we provide a second kind of padding. On return from an interrupt, the interrupt handling logic will pad again for Δt_{p_2} cycles, ensuring that $\Delta t_{p_1} + \Delta t_{p_2}$ is again the constant value T (i.e., $\Delta t_{p_2} = \Delta t_1$). This makes sure that the resume-to-end time measured by the attacker does not depend on the instruction being interrupted.

This description of our padding scheme is still incomplete. Crucially, we need to specify what happens if a new interrupt arrives while the processor is still performing the padding because of a previous interrupt or the interrupt handling routine is still running (actually, interrupts are masked when one is handled). This is important to counter attacks like that of Example 6.4.

Intuitively, our mitigation ensures that (1) an attacker can schedule an interrupt at any time t_a during enclave execution; (2) that interrupt will always be handled with a constant latency T; and (3) the resume-to-end time is always exactly the time the enclave still would have needed to complete execution from point t_a if it had not been interrupted. Despite being fundamental, the above double padding scheme just one of the ingredients of our secure interrupt handling mechanism. Many other aspects of the design are important for security. We now briefly discuss a number of other issues that came up during the security proof, leading to the refinement of the implementation of Sancus.

SAVING EXECUTION STATE ON INTERRUPT When an enclaved execution is interrupted, the processor state (contents of the registers) is saved (to allow resuming the execution once the interrupt is handled) and is cleared (to avoid leaking confidential register contents to the context). A straightforward implementation would be to store the processor state on a stack in the enclave accessible memory. However, the proof of our security theorem showed that this solution is not secure: consider two enclaved modules that monitor the content of the memory area where processor state is saved, and behave differently on observing a change in the content of this memory area. These modules are contextually equivalent in the absence of interrupts (as the contents of this memory area will never

change), but become distinguishable in the presence of interrupts. Note that this is an actual security issue and not an artifact of the formalization. For instance, consider what could happen if the module behaves differently depending on whether a secret is stored or not in said area: an attacker could exploit this dependency to leak (parts of) that secret. Hence, our design saves processor state in a storage area which is *inaccessible* to software.

NO ACCESS TO UNPROTECTED MEMORY FROM WITHIN AN ENCLAVE Most designs of enclaved execution allow an enclave to access unprotected memory. However, for a single core processor, interruptibility significantly weakens contextual equivalence for enclaves that can access unprotected memory. Consider an enclave M_1 that always returns a constant 0, and an enclave M_2 that reads twice from the same unprotected address and returns the difference of the values read. On a single-core processor without interrupts, M_2 will also always return 0, and hence is indistinguishable from M_1. But an interrupt scheduled to occur between the two reads from M_2 can change the value returned by the second read, and hence M_1 and M_2 become distinguishable. Hence, our design forbids enclaves to access unprotected memory.

For similar reasons, our design forbids an interrupt handler to reenter the enclave while it has been interrupted, and forbids the enclave to directly interact with I/O devices.

Finally, we prevent the interrupt enable bit in the status register from being changed by the software in the enclave, as such changes are unobservable in the original Sancus and they would be observable once interruptibility is added.

While the security proof is a significant amount of effort, an important benefit of this formalization is that it forced us to consider all these cases and to think about secure ways of handling them. We made our design choices to keep the model simple and the proof manageable, although some of them may seem restrictive. Section 6.7 discusses the practical impact of these choices and possible ways of relaxing some limitations.

6.3 THE FORMAL MODEL OF THE ARCHITECTURE

Here we set up the formal model of the architecture that runs both the original, uninterruptible Sancus (Sancus^{H}, Sancus-High) and the secure interruptible Sancus ($\mathbf{Sancus}^{\mathbf{L}}$, Sancus-Low). The next section will define the semantics of Sancus^{H} and $\mathbf{Sancus}^{\mathbf{L}}$, and then we will formally show that the two versions of Sancus actually provide the same security guarantees, i.e., the isolation mechanism is not broken by adding our carefully designed interruptible enclaved execution.

6.3.1 *Memory and memory layout*

Recall from Section 6.2.1 that MSP430 has a 16-bit architecture, thus we model its memory as a (finite) function mapping 2^{16} locations to bytes b. Given a memory $\mathcal{M}$, we denote the operation of retrieving the byte associated with the location l as $\mathcal{M}(l)$. On top of that, we define read and write operations on words (i.e., pairs of bytes) and we write $w = b_1 b_0$ to denote that the most significant byte of a word w is b_1 and its least significant byte is b_0.

The read operation is standard: it retrieves two consecutive bytes from a given memory location l (in a little-endian fashion, as in the MSP430):

$$\mathcal{M}[l] \triangleq b_1 b_0 \quad \text{if } \mathcal{M}(l) = b_0 \wedge \mathcal{M}(l+1) = b_1$$

We define the write operation as follows

$$(\mathcal{M}[l \mapsto b_1 b_0])(l') \triangleq \begin{cases} b_0 & \text{if } l' = l \\ b_1 & \text{if } l' = l+1 \\ \mathcal{M}(l') & \text{o.w.} \end{cases}$$

Writing $b_0 b_1$ in location l in $\mathcal{M}$ means to build an updated memory mapping l to b_0, $l+1$ to b_1 and unchanged otherwise.

Note that reads and writes in $l = \texttt{0xFFFF}$ are undefined ($l+1$ would overflow hence it is undefined). The memory access control explicitly forbids these accesses (see below). Also, the write operation deals with unaligned memory accesses (cfr. case $l' = l+1$). We faithfully model these aspects to prove that they do not lead to potential attacks.

Since modeling the memory as a function gives no clues on how the enclave is organized, we assume a fixed *memory layout* $\mathcal{L} \triangleq \langle ts, te, ds, de, isr \rangle$. It describes how the enclave and the *interrupt service routine* (ISR) are placed in non-fragmented portions of memory and is used to check memory accesses during the execution of each instruction (see below). To reflect the memory segmentation of the real Sancus, we have two protected memory sections, containing the code and the data of the enclave. The protected code section is denoted by $[ts, te)$, while $[ds, de)$ is the protected data section, and they are placed in non-overlapping memory sections. The first address of the protected code section is the single entry point of the enclave. The last component of the tuple $\mathcal{L}$, isr, is the address of the ISR. Finally, we reserve the location 0xFFFE to store the address of the first instruction to be executed when the CPU starts or when an exception happens, reflecting the behavior of the MSP430. Thus, 0xFFFE must be outside the enclave sections and different from isr. Note that memory operations enforce no memory access control w.r.t. $\mathcal{L}$, since these checks are performed during the execution of each instruction (see below).

Summing up, a memory layout is defined as

$$\mathcal{L} \triangleq \langle ts, te, ds, de, isr \rangle, \quad \text{where}$$

- $[ts, te)$ and $[ds, de)$ are the protected code and data sections, resp., with $[ts, te) \cap [ds, de) = \emptyset$;
- $isr \notin [ts, te) \cup [ds, de)$ is the entry point for the ISR; and
- $isr \neq \texttt{0xFFFE}$, and $\texttt{0xFFFE} \notin [ts, te) \cup [ds, de)$. The address 0xFFFE stores the address where the CPU starts executing on boot, or on an exception.

6.3.2 *Register files*

Sancus[H], just like the original Sancus, has sixteen 16-bit registers three of which $\texttt{R}_0$, $\texttt{R}_1$, $\texttt{R}_2$ are used for dedicated functions, whereas the others are for general use (actually, $\texttt{R}_3$ is a constant generator in the real MSP430 machine, but we ignore that use in our formalization). More precisely, $\texttt{R}_0$ (hereafter denoted as pc) is the program counter and points to the next instruction to be executed. Instruction accesses are performed by word and the pc is aligned to even addresses. The register $\texttt{R}_1$ (sp hereafter) is the stack pointer and it is used, as usual, by the CPU to store the pointer to the activation record of the current procedure. Also the stack pointer is aligned to even addresses. The register $\texttt{R}_2$ (sr

hereafter) is the status register and contains different pieces of information encoded as flags. The most important here is the fourth bit, called GIE, set to 1 when interrupts are enabled. Other bits are set, e.g., when an operation produces a carry or when the result of an operation is zero.

Formally, our *register file* $\mathcal{R}$ is a function that maps each register r to a word. The read operation is standard:

$$\mathcal{R}[\mathtt{r}] \triangleq w \text{ if } \mathcal{R}(\mathtt{r}) = w$$

The write operation requires instead accommodating the hardware itself and our security requirements:

$$\mathcal{R}[\mathtt{r} \mapsto w] \triangleq \lambda[\mathtt{r}'].\begin{cases} w\&\mathtt{0xFFFE} & \text{if } \mathtt{r}' = \mathtt{r} \wedge (\mathtt{r} = \mathtt{pc} \vee \mathtt{r} = \mathtt{sp}) \\ (w\&\mathtt{0xFFF7}) \mid (\mathcal{R}[\mathtt{sr}]\&\mathtt{0x8}) & \text{if } \mathtt{r}' = \mathtt{r} = \mathtt{sr} \wedge \mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{PM} \\ w & \text{if } \mathtt{r}' = \mathtt{r} \wedge (\mathtt{r} \neq \mathtt{pc} \wedge \mathtt{r} \neq \mathtt{sp}) \wedge \\ & \quad (\mathtt{r} \neq \mathtt{sr} \vee \mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{UM}) \\ \mathcal{R}[\mathtt{r}'] \quad \text{o.w.} & \end{cases}$$

In the definition above $\&$ and $\mid$ denote the standard *and* and *or* bitwise operators, and we use the relation $\mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{m}$, for $m \in \{\mathtt{PM}, \mathtt{UM}\}$ that is defined in Section 6.3.7. It indicates that the execution is carried on in protected or in unprotected mode. Note that word alignment is enforced because the least-significant bit of the program counter and of the stack pointer are *always* masked to 0 (as it happens in the MSP430). Also, the GIE bit of the status register is always masked to its previous value when in protected mode, i.e., it cannot be changed when the CPU is running in protected code (resulting from the bitwise *or* between $w\&\mathtt{0xFFF7}$ — masking the GIE bit of w — and $\mathcal{R}[\mathtt{sr}]\&\mathtt{0x8}$ — masking everything except the value of the GIE bit of the status register).

Finally, it is convenient defining the following special register files:

$$\mathcal{R}_0 \triangleq \{\mathtt{pc} \mapsto 0, \mathtt{sp} \mapsto 0, \mathtt{sr} \mapsto 0, \mathtt{R}_3 \mapsto 0, \ldots, \mathtt{R}_{15} \mapsto 0\}$$
$$\mathcal{R}^{init}_{\mathcal{M}} \triangleq \{\mathtt{pc} \mapsto \mathcal{M}[\mathtt{0xFFFE}], \mathtt{sp} \mapsto 0, \mathtt{sr} \mapsto \mathtt{0x8}, \mathtt{R}_3 \mapsto 0, \ldots, \mathtt{R}_{15} \mapsto 0\}$$

where

- pc is set to $\mathcal{M}[\mathtt{0xFFFE}]$ as it does in the MSP430;
- sp is set to 0 and we expect untrusted code to set it up in a setup phase, if any;
- sr is set to 0x8, i.e., register is clear except for the GIE flag.

Register file $\mathcal{R}_0$ is used when we jump out from the enclave to zero the processor state; $\mathcal{R}^{init}_{\mathcal{M}}$ denotes the initial register file of the CPU, when it starts executing.

6.3.3 *I/O Devices*

Recall from the previous section that the attacker can raise an interrupt and observe the effects it has on the execution of the enclave. This kind of attack usually requires a software component and a hardware one. The software component is settled in the unprotected memory and is detailed below. The hardware component is a physical device that interacts with the processor through synchronous I/O operations. Additionally, the

progress of I/O devices is tied to that the of the CPU, making them cycle-accurate and allowing to model the full power of the attacker considered in the real Sancus (e.g., to use a cycle-accurate timer). In our case it is a Sancus I/O device, and we model it as a (simplified) *deterministic I/O automaton* [144], as follows:

$$\mathcal{D} \triangleq \langle \Delta, \delta_{\text{init}}, \overset{a}{\rightsquigarrow}_D \rangle, \quad \text{where}$$

- $a \in A$, with A a signature that includes the following actions (below w is a word):
  - ϵ, a silent, internal action;
  - $rd(w)$, an output action (i.e., read request from the CPU);
  - $wr(w)$, an input action (i.e., write request from the CPU);
  - $int?$ an output action telling that an interrupt was raised in the last state;
- $\Delta \neq \emptyset$ is the *finite* set of internal states of the device;
- $\delta_{\text{init}} \in \Delta$ is the *single* initial state;
- $\delta \overset{a}{\rightsquigarrow}_D \delta' \subseteq \Delta \times A \times \Delta$ is the transition function that takes one step in the device while doing action $a \in A$, starting in state δ and ending in state δ'. (We write $\overline{a}$ for a string of actions and we omit ϵ when unnecessary.) The transition function is such that $\forall \delta$ either $\delta \overset{\epsilon}{\rightsquigarrow}_D \delta'$ or $\delta \overset{int?}{\rightsquigarrow}_D \delta''$ (i.e., one and only one of the two transitions must be possible). Also, at most one $rd(w)$ action must be possible starting from a given state.

6.3.4 *Software modules, contexts and whole programs*

A module contains both protected code and protected data.

Definition 6.1. *A* software module *is a memory $\mathcal{M}_M$ containing both protected code and protected data sections.*

Intuitively, the context is the part of the whole program that can be manipulated by an attacker, i.e., the software component and the physical device:

Definition 6.2. *A* context *C is a pair $\langle \mathcal{M}_C, \mathcal{D} \rangle$, where $\mathcal{D}$ is a device and $\mathcal{M}_C$ defines the contents of all memory locations* outside *the protected sections of the layout.*

Filling in a context hole with a software module yields a whole program.

Definition 6.3. *Given a context $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$ and a software module $\mathcal{M}_M$ such that* $\texttt{dom}\,(\mathcal{M}_C) \cap \texttt{dom}\,(\mathcal{M}_M) = \emptyset$, *a* whole program *is*

$$C[\mathcal{M}_M] \triangleq \langle \mathcal{M}_C \uplus \mathcal{M}_M, \mathcal{D} \rangle.$$

6.3.5 *Instruction set*

The instruction set *Inst* is the same for both Sancus[L] and Sancus[H] and it is (almost) that of the MSP430. An overview of the instruction set is in Table 3 (Page 116). For each instruction i the table includes its operands, an intuitive meaning of its semantics, its duration and the number of words it occupies in memory. The durations are used to define the function $cycles(i)$ and implicitly determine a value MAX_TIME, greater than or

equal to the duration of the longest instruction. Here we choose MAX_TIME $= 6$, in order to keep the compatibility with MSP430 (whose longest instruction takes 6 cycles). Since instructions are stored in memory, for getting them we use the meta-function $decode(\mathcal{M}, l)$ that decodes the contents of the cell(s) starting at location l, returning an instruction in the table if any and $\perp$ otherwise.

6.3.6 *Configurations*

Given an I/O device $\mathcal{D}$, the internal state of the entire system is described by configurations of the form:

$$c \triangleq \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \in \mathbb{C}, \quad \text{where}$$

- δ is the current state of the I/O device;
- t is the current clock cycle, i.e., a natural number denoting the time elapsed since the CPU started its execution;
- t_a is the arrival time (clock cycle) of the last pending interrupt, set to $\perp$ if there are none;
- $\mathcal{M}$ is the current memory;
- $\mathcal{R}$ is the current content of the registers;
- pc_{old} is the value of the program counter *before* executing the current instruction; and
- $\mathcal{B}$ is the *backup* that can assume the following values:
  - $\perp$, indicating that the CPU is either handling no interrupt or it is handling one originated in unprotected mode;
  - $\langle \mathcal{R}, pc_{old}, t_{pad} \rangle$, indicating that the interrupt handler is managing an interrupt raised in protected mode. The triple includes the register file $\mathcal{R}$, the program counter pc_{old} at the time the interrupt was originated, and the value t_{pad}, which indicates the remaining padding time that must be applied before returning into protected mode;
  - $\langle \perp, \perp, t_{pad} \rangle$, indicating that the CPU is currently padding the resumption from an interrupt.

The initial states of the CPU are represented by the initial configurations from which the computation starts. The initial configuration for a whole program $C[\mathcal{M}_M] = \langle \mathcal{M}, \mathcal{D} \rangle$ is:

$$\text{INIT}_{C[\mathcal{M}_M]} \triangleq \langle \delta_{\text{init}}, 0, \perp, \mathcal{M}, \mathcal{R}^{init}_{\mathcal{M}_C}, \texttt{0xFFFE}, \perp \rangle \text{ where}$$

- the state of the I/O device $\mathcal{D}$ is δ_{init};
- the initial value of the clock is 0 and no interrupt has arrived yet;
- the memory is initialized to the whole program memory $\mathcal{M}_C \uplus \mathcal{M}_M$;
- all the registers are initialized to 0, their initial value, except that pc is set to 0xFFFE (the address from which the CPU gets the initial program counter), and that sr is set to 0x8 (the register is clear except for the GIE flag);
- the "old" program counter is also initialized to 0xFFFE; and

- the backup is set to $\perp$, as no interrupt has been raised yet.

Dually, HALT is the only configuration denoting termination. More precisely, we feel free to use this distinguished and opaque configuration for representing graceful termination.

Also, we define *exception handling* configurations, that model what happens on soft reset of the machine (e.g., on a memory access violation, or a halt in protected mode). On such a soft reset, control returns to the attacker by jumping to the address stored in location `0xFFFE`:

$$\mathrm{EXC}_{\langle \delta,t,t_a,\mathcal{M},\mathcal{R},pc_{old},\mathcal{B}\rangle} \triangleq \langle \delta, t, \perp, \mathcal{M}, \mathcal{R}_0[\mathtt{pc} \mapsto \mathcal{M}[\mathtt{0xFFFE}]], \mathtt{0xFFFE}, \perp \rangle.$$

6.3.6.1 *I/O device wrapper*

Since the class of interrupt-based attacks requires a cycle-accurate timer, it is convenient to synchronize the CPU and the device time by forcing the device to take as many steps as the number of cycles consumed for each instruction by the CPU. The following "wrapper" around the device $\mathcal{D}$ models this synchronization:

$$\mathcal{D} \vdash \delta, t, t_a \curvearrowright^k_D \delta', t', t'_a$$

Intuitively, assume the device be in state δ, the clock time be t and the last interrupt be raised at time t_a. Then, after k cycles the new clock time will be $t' = t + k$, the last interrupt was raised at time t'_a and the new state will be δ'; when no interrupt has to be handled, $t_a = t'_a = \perp$. Formally:

$$\frac{a \in \{\epsilon, int?\} \qquad \bigwedge_{i=0}^{k-1} \delta_i \overset{a}{\rightsquigarrow}_D \delta_{i+1} \qquad t'_a = \begin{cases} t + j & \text{if } \exists 0 \leq j < k.\, \delta_j \overset{int?}{\rightsquigarrow}_D \delta_{j+1} \wedge \\ & \quad \forall j' < j.\, \delta_{j'} \overset{\epsilon}{\rightsquigarrow}_D \delta_{j'+1} \\ t_a & \text{o.w.} \end{cases}}{\mathcal{D} \vdash \delta_0, t, t_a \curvearrowright^k_D \delta_k, (t+k), t'_a}$$

6.3.7 *CPU mode*

We now specify when the CPU is running in protected or in unprotected mode. Actually, the mode $\mathtt{m} \in \{\mathtt{PM}, \mathtt{UM}\}$ is determined by the value of the program counter, which can be in either code section. Formally:

$$\frac{pc \in [\mathcal{L}.ts, \mathcal{L}.te)}{pc \vdash_{mode} \mathtt{PM}} \qquad \frac{pc \notin [\mathcal{L}.ts, \mathcal{L}.te) \cup [\mathcal{L}.ds, \mathcal{L}.de)}{pc \vdash_{mode} \mathtt{UM}}$$

Also, we lift the definition to configurations as follows:

$$\frac{\mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{m}}{\langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \vdash_{mode} \mathtt{m}} \qquad \frac{}{\mathrm{HALT} \vdash_{mode} \mathtt{UM}}$$

Note in passing that no mode is defined when the program counter points within the data section, because the memory access control introduced below prevents the program counter to assume values therein.

		t			
		Entry Point	Prot. code	Prot. Data	Other
f	Entry Point/Prot. code	r-x	r-x	rw-	–x
	Other	–x	—	—	rwx

Table 4: Definition of $MAC_{\mathcal{L}}(f, \texttt{rght}, t)$ function, where f and t are locations.

6.3.8 *Memory access control*

We formalize the *memory access control* (MAC) mechanism of Sancus using the predicate $MAC_{\mathcal{L}}(f, \texttt{rght}, t)$ in Table 4. Roughly, the predicate holds whenever the address the CPU is trying to read is within the same memory partition as the program counter of the last completed instruction (pc_{old}). More precisely, the predicate holds whenever from the location f (usually pc_{old}) we have the rights rght on location t, reflecting the mechanism provided by Sancus. Note that when f is within unprotected code, $MAC_{\mathcal{L}}(f, \texttt{rght}, t)$ grants it no rights on a location t in the protected memory.

Building on the above, we define the following relation

$$i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK}$$

that holds whenever the instruction i can be executed in a CPU configuration in which the previous program counter is pc_{old}, the registers are $\mathcal{R}$ and the backup is $\mathcal{B}$. We check that (1) when transitioning from pc_{old} to $\mathcal{R}[\texttt{pc}]$, the CPU has execution rights to execute instruction i, i.e., $MAC_{\mathcal{L}}(pc_{old}, \texttt{x}, \mathcal{R}[\texttt{pc}] + j)$ for $j \in \{0, ..., size(i) - 1\}$; (2) if i is an I/O instruction, it can be executed in current CPU mode; and (3) if i is a memory operation (i.e., either $\texttt{MOV r}_1\ \texttt{0(r}_2\texttt{)}$ or $\texttt{MOV @r}_1\ \texttt{r}_2$) from $\mathcal{R}[\texttt{pc}]$ we have the appropriate rights to perform it.

The predicate MAC is the minimal relation satisfying the inference rules in Figure 52. Note that (1) for each word that is accessed in memory we also check that the first location is not the last byte of the memory (except for the program counter, for which the decode function would fail since it would try to access undefined memory); (2) word accesses must be checked once for each byte of the word; and (3) checks on pc guarantee that a memory violation does not happen while decoding. We briefly comment on the rule for $i \in \{\texttt{IN r}, \texttt{OUT r}\}$. The preconditions say that (1) the current value of the program counter is in unprotected mode; (2) that the instructions pointed to by pc_{old} and $\mathcal{R}[\texttt{pc}]$ are executable, according to $MAC_{\mathcal{L}}$; and (3) that the same holds for pc_{old} and $\mathcal{R}[\texttt{pc}] + 1$, i.e., the next instruction.

6.4 THE SEMANTICS OF SANCUS^{H}, SANCUS^{L} AND THEIR INTERRUPT LOGIC

As anticipated, we proceed to formally define the semantics of Sancus^{H} and **Sancus^{L}**. The two share most of their structure and just differ in the way they deal with interrupts: Sancus^{H} has none of them and so the handler is trivial, while **Sancus^{L}** has an appropriate interrupt logic, based on the mitigation sketched above. Each version of Sancus is endowed with two transitions systems: the main one specifies the operational semantics of instructions, while the other is auxiliary and describes the relevant interrupt logic. Therefore, we will factorize as much as possible the inference rules shared by the main transition systems, and only indicate the differences using blue, sans-serif font for Sancus^{H} and **red, bold** for **Sancus^{L}**.

$$\frac{\begin{array}{c}\mathcal{R}[\mathtt{sp}] \neq \mathtt{0xFFFF} \\ \mathcal{R}[\mathtt{sp}]+2 \neq \mathtt{0xFFFF} \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+1) \quad MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{r}, \mathcal{R}[\mathtt{sp}]) \\ MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{r}, \mathcal{R}[\mathtt{sp}]+1) \quad MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{r}, \mathcal{R}[\mathtt{sp}]+2) \quad MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{r}, \mathcal{R}[\mathtt{sp}]+3)\end{array}}{\mathtt{RETI}, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \mathtt{OK}}$$

$$\frac{\begin{array}{c} i \in \{\mathtt{NOP}, \mathtt{AND}\ \mathtt{r}_1\ \mathtt{r}_2, \mathtt{ADD}\ \mathtt{r}_1\ \mathtt{r}_2, \mathtt{SUB}\ \mathtt{r}_1\ \mathtt{r}_2, \mathtt{CMP}\ \mathtt{r}_1\ \mathtt{r}_2, \mathtt{MOV}\ \mathtt{r}_1\ \mathtt{r}_2, \mathtt{JMP}\ \&\mathtt{r}, \mathtt{JZ}\ \&\mathtt{r}\} \\ MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+1)\end{array}}{i, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \mathtt{OK}}$$

$$\frac{\begin{array}{c} i \in \{\mathtt{NOT}\ \mathtt{r}, \mathtt{MOV}\ \#w\ \mathtt{r}\} \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]) \\ MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+1) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+2) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+3)\end{array}}{i, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \mathtt{OK}}$$

$$\frac{i \in \{\mathtt{IN}\ \mathtt{r}, \mathtt{OUT}\ \mathtt{r}\} \quad \mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{UM} \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+1)}{i, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \mathtt{OK}}$$

$$\frac{\begin{array}{c}\mathcal{R}[\mathtt{r}_1] \neq \mathtt{0xFFFF} \quad \mathcal{R}[\mathtt{r}_1]+1 \neq \mathtt{0xFFFF} \\ MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{r}, \mathcal{R}[\mathtt{r}_1]) \quad MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{r}, \mathcal{R}[\mathtt{r}_1]+1) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+1)\end{array}}{\mathtt{MOV}\ @\mathtt{r}_1\ \mathtt{r}_2, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \mathtt{OK}}$$

$$\frac{\begin{array}{c}\mathcal{R}[\mathtt{r}_2] \neq \mathtt{0xFFFF} \\ \mathcal{R}[\mathtt{r}_2]+1 \neq \mathtt{0xFFFF} \quad MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{w}, \mathcal{R}[\mathtt{r}_2]) \quad MAC_{\mathcal{L}}(\mathcal{R}[\mathtt{pc}], \mathtt{w}, \mathcal{R}[\mathtt{r}_2]+1) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]) \\ MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+1) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+2) \quad MAC_{\mathcal{L}}(pc_{old}, \mathtt{x}, \mathcal{R}[\mathtt{pc}]+3)\end{array}}{\mathtt{MOV}\ \mathtt{r}_1\ \mathtt{0}(\mathtt{r}_2), \mathcal{R}, pc_{old}, \bot \vdash_{mac} \mathtt{OK}}$$

$$\frac{i \neq \mathtt{RETI} \quad \mathcal{B} \neq \bot \quad i, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \mathtt{OK} \quad \mathcal{R}[\mathtt{sr}].\mathtt{GIE} = \mathtt{0} \quad \mathcal{R}[\mathtt{pc}] \neq ts}{i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK}} \qquad \frac{\mathcal{B} \neq \bot}{\mathtt{RETI}, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK}}$$

Figure 52: The rules defining the memory access control.

More precisely, assume hereafter as given a context $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$, where M_C defines the contents of the memory locations of the unprotected section and $\mathcal{D}$ is an I/O device, and let $c, c' \in \mathbb{C}$ be two configurations. Then, the main transition system of Sancus$^{\mathsf{H}}$ has the transitions on the left and its auxiliary one the transitions on the right:

$$\mathcal{D} \vdash c \rightarrow c' \qquad\qquad \mathcal{D} \vdash c \hookrightarrow_{\mathsf{I}} c'$$

while the main and the auxiliary transition systems of Sancus$^{\mathsf{L}}$ have the transitions on the left and on the right, respectively:

$$\mathcal{D} \vdash c \rightarrow c' \qquad\qquad \mathcal{D} \vdash c \hookrightarrow_{\mathbf{I}} c'$$

6.4.1 *The Operational Semantics of* Sancus$^{\mathsf{H}}$

We first present the auxiliary transition system implementing the logic that decides what happens when an interrupt arrives, and then we formalize how the instructions are executed in Sancus$^{\mathsf{H}}$.

6.4.1.1 *Interrupts in* Sancus$^{\mathsf{H}}$

As said above, interrupts in Sancus$^{\mathsf{H}}$ are *always* ignored, thus the configuration is left unchanged, and we have the following trivial rule:

INT

$$\frac{}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathsf{I}} \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}$$

6.4.1.2 *Main transition system*

The transitions of the main transition system describe how the Sancus^H configurations evolve during the execution. Figure 53 shows selected inference rules of the transition system, on which we briefly comment below; the other rules can be found in Appendix B.2.

The rule (CPU-HLT-UM) is the only one that halts the CPU and only applies when an HLT instruction is executed in unprotected mode. Dually the rule (CPU-HLT-PM) deals with the case in which an HLT instruction is to be executed in protected mode. In such a case, the *exception handling* configuration is reached, allowing for a cleanup and a graceful termination. The rule (CPU-VIOLATION-PM) takes care of the violations in protected mode: the transition in the conclusion of the rule leads to the *exception handling* configuration if there is a non-empty backup (first premise) and if the instruction i does not pass the memory-access control relation (second premise); Rule (CPU-MOVL) is for when the current instruction i loads in $\mathtt{r}_2$ the word in memory at the position pointed to by $\mathtt{r}_1$. Its first premise checks that the CPU is not currently padding interrupt resumption time (more details on that later on, it can be safely ignored for now); the second one if the instruction can be executed; the third one increments the program counter by 2 and loads in $\mathtt{r}_2$ the value $\mathcal{M}[\mathtt{r}_1]$; the fourth premise registers in the device that i requires $cycles(i)$ cycles to complete; and the last one executes the interrupt logic to check whether an interrupt needs to be handled or not (see below). Rules dealing with jumps are quite standard. Upon a JZ $\&\mathtt{r}$ instruction (*jump if zero*), the CPU checks the content of the Z (*zero*) bit of the status register. If $\mathcal{R}[\mathtt{sr}].Z$ is 0, then the rule (CPU-JZ0) is triggered and $\mathcal{R}[\mathtt{pc}]$ is left unchanged, otherwise the rule (CPU-JZ1) applies and the content of the register $\mathtt{r}$ is copied into pc, so performing the jump. Another interesting rule is (CPU-IN) that deals with the case in which the instruction reads a word from the device and puts the result in $\mathtt{r}$. Its third premise holds when the device sends the word w to the CPU; the others are similar to those of (CPU-MOVL). Dually, the rule (CPU-OUT) deals with outputs to the device. Note that, the CPU is forced to halt when the I/O device is not ready for a read or a write (rules (CPU-NOIN) and (CPU-NOOUT)). As a matter of fact, this can only happen in unprotected mode, since the MAC relation forbids I/O operations inside enclaves. Note also that the current time of the CPU is *always* incremented by the time needed to complete the current instruction.

6.4.2 *The Operational Semantics of* Sancus^L

In Sancus^L interrupts can be raised and must be properly handled securely both in protected and unprotected mode, and for that we define a non-trivial auxiliary transition system. Although the rules of the main transition system are largely the same as Sancus^H, the new auxiliary transitions affect the behavior of the instruction for returning from interrupts.

6.4.2.1 *Interrupts in* Sancus^L

The inference rules in Figure 54 formalize the mitigation outlined in Section 6.2 as a defense against interrupt-based attacks, regardless of the CPU being in unprotected or protected mode. For that, all the rules have a premise checking the mode in which the last instruction was executed ($pc_{old} \vdash_{mode}$ UM or $pc_{old} \vdash_{mode}$ PM).

The rules (INT-UM-NP) and (INT-PM-NP) take care of when the GIE bit of the status register is set to 0, i.e., interrupts are disabled, or there is none ($t_a = \bot$). In this case the configurations are simply left untouched.

When instead $\texttt{GIE} = 1$ and an interrupt is on ($t_a \neq \bot$), either rule (INT-UM-P) or (INT-PM-P) handles it. When in unprotected mode, a premise of (INT-UM-P) concerns registers: the program counter gets the entry point of the handler; the status register gets 0; and the top of the stack is moved 4 positions ahead to allocate the activation record of the interrupt handler.

Accordingly, the new memory $\mathcal{M}'$ updates the locations pointed by the relevant elements of the stack with the current program counter and the contents of the status register. The last premise reflects that setting up this interrupt handling takes 6 cycles.

The rule (INT-PM-P) is for protected mode and it is more interesting. Besides assigning the entry point of the handler to the program counter, it computes the padding time for mitigation of interrupt-based timing attacks and saves the backup in $\mathcal{B}'$. The padding k is then used, causing interrupt handling to take $6 + k$ steps. Such a padding implements the first part of the mitigation (see Section 6.2.3) and is computed so as to make the dispatching time of interrupts constant. Note that the padding never gets negative. When an interrupt arrives in protected mode two cases may arise. Either $\texttt{GIE} = 1$, and the padding is non-negative because the interrupt is handled at the end of the current instruction; or $\texttt{GIE} = 0$, and no padding is needed because the interrupt is handled as soon as GIE becomes 1, which is only possible in unprotected mode. The backup stores part of the CPU configuration ($\mathcal{R}$ and pc_{old}) and $t_{pad} = t - t_a$. The value of t_{pad} will then be used as further padding before returning, thus fully implementing the mitigation (cfr. Section 6.2.3). Recall that the register file $\mathcal{R}_0$ is $\{\texttt{pc} \mapsto 0, \texttt{sp} \mapsto 0, \texttt{sr} \mapsto 0, \texttt{R}_3 \mapsto 0, \ldots, \texttt{R}_{15} \mapsto 0\}$.

It is worth to briefly describe what happens upon “corner cases:”

- Whenever an interrupt has to be handled in protected mode, but the current instruction drives the CPU in unprotected mode, the padding mechanism is applied *fully including* the padding after the RETI (see rule (CPU-RETI)). Indeed, if partial padding (resp. no padding at all) was applied then the duration of the padding (resp. of the last instruction) would be leaked to the attacker (cf. Figure 54).

- Interrupts are ignored when arising during the time spent in padding and *before* invoking the interrupt service routine. This is because the padding duration and the instruction duration would be leaked otherwise. To avoid that, the rule (INT-PM-P) ignores any interrupts raised during the cycles needed for the interrupt logic and for the padding. A viable alternative would require to buffer interrupts and handle them later on.

- Interrupts happening *during* the execution of the interrupt service routine are simply “chained” and handled as soon as the current routine completes (see rule (CPU-RETI-CHAIN)).

- Finally, interrupts raised during the padding time and *after* the interrupt service routine are handled as any other interrupt happening in protected mode (see rule (CPU-RETI-PAD)).

6.4.2.2 *Main transition system*

The rules of the main transition system of Sancus^{L} are exactly the same used for the semantics of Sancus^{H}, except for the blue arrows turned into red. Notably, those for the interrupt logic: the red arrow $\hookrightarrow_{\text{I}}$ replaces the blue arrow $\hookrightarrow_{\text{I}}$ in the premises. Figure 55 shows the rules dealing with the cases that may happen when the interrupt handler returns and the processor gives the control back to the code that was executing before the interrupt was raised. The first rule (CPU-RETI), deals with the actual return from an interrupt. In this case the processor restores the status register and sets the program counter to the instruction following the interrupted one. The values to bring back have been stored in the current activation record on the stack (i.e., $\mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{M}[\mathcal{R}[\mathtt{sp}] + 2], \mathtt{sr} \mapsto \mathcal{M}[\mathcal{R}[\mathtt{sp}]]]$). Instead, rule (CPU-RETI-CHAIN) applies if an interrupt arrived while returning from handling an interrupt raised in protected mode (third and fifth premises). In this case the CPU directly jumps to the handler of the new interrupt with no further padding. Finally, we discuss the rules (CPU-RETI-PREPAD) and (CPU-RETI-PAD). Their combination deals with the case in which the CPU is returning from handling an interrupt raised in protected mode, and no new interrupt arrived afterwards (or the `GIE` bit is off, cf. the fourth premise of rule (CPU-RETI-PREPAD)). First, the rule (CPU-RETI-PREPAD) restores registers and pc_{old} from the backup $\mathcal{B}$, so enabling the application of the rule (CPU-RETI-PAD) (note that no other rule is applicable because of the contents of $\mathcal{B}$). Then, through the rule (CPU-RETI-PAD) the remaining padding (recorded in the backup) is applied so to prevent resume-to-end timing attacks (note that this last padding is interruptible, as witnessed by the last premise). This last padding is applied even though the configuration reached through rule (CPU-RETI-PREPAD) is in unprotected mode (i.e., when the interrupted instruction was a jump out of protected mode). Otherwise, the attacker may discover the value of the padding applied *before* the interrupt service routine. Actually, we model the mechanism of restoring registers, pc_{old} and of applying the remaining padding with two rules instead of just one for technical reasons.

6.4.3 *A progress theorem*

As a sanity check we prove the following progress theorem showing that both Sancus^{H} and Sancus^{L} get stuck only if the CPU reaches the distinguished configuration HALT. Its proof is in Appendix B.4:

Theorem 6.1 (Progress). *For all $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$, $\mathcal{M}_M$ and configuration c*

- $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c \not\rightarrow \implies c = \text{HALT}$
- $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c \not\rightarrow \implies c = \text{HALT}.$

(CPU-HLT-UM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \vdash_{mode} \texttt{UM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \text{HALT}} \; decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{HLT}$$

(CPU-NoIN)

$$\frac{\delta \overset{rd(w)}{\not\rightsquigarrow}_D}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \text{HALT}} \; decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{IN r}$$

(CPU-NoOUT)

$$\frac{\delta \overset{wr(\mathcal{R}[\texttt{r}])}{\not\rightsquigarrow}_D}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \text{HALT}} \; decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{OUT r}$$

(CPU-HLT-PM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \vdash_{mode} \texttt{PM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \text{EXC}_{\langle \delta, t+cycles(i), t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{HLT}$$

(CPU-Decode-Fail)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \bot}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \text{EXC}_{\langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}}$$

(CPU-Violation-PM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \not\vdash_{mac} \texttt{OK}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \text{EXC}_{\langle \delta, t+cycles(i), t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) \neq \bot$$

(CPU-MovL)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \\ \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 2][\texttt{r}_2 \mapsto \mathcal{M}[\mathcal{R}[\texttt{r}_1]]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_{\text{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{MOV @r}_1\ \texttt{r}_2$$

(CPU-Jz0)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \\ \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 2] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_{\text{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{JZ \&r} \wedge \mathcal{R}[\texttt{sr}].Z = 0$$

(CPU-Jz1)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \\ \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{r}]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_{\text{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{JZ \&r} \wedge \mathcal{R}[\texttt{sr}].Z = 1$$

(CPU-In)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \delta \overset{rd(w)}{\rightsquigarrow}_D \delta' \\ \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 2][\texttt{r} \mapsto w] \qquad \mathcal{D} \vdash \delta', t, t_a \curvearrowright_D^{cycles(i)} \delta'', t', t'_a \\ \mathcal{D} \vdash \langle \delta'', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_{\text{I}} \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{IN r}$$

(CPU-Out)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \\ \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 2] \qquad \delta \overset{wr(\mathcal{R}[\texttt{r}])}{\rightsquigarrow}_D \delta' \qquad \mathcal{D} \vdash \delta', t, t_a \curvearrowright_D^{cycles(i)} \delta'', t', t'_a \\ \mathcal{D} \vdash \langle \delta'', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_{\text{I}} \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{OUT r}$$

Figure 53: Some rules of the main transition system for SancusH.

(INT-UM-P)

$$\frac{pc_{old} \vdash_{mode} \texttt{UM} \quad \mathcal{R}[\texttt{sr}].\texttt{GIE} = 1 \quad t_a \neq \bot \quad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto isr, \texttt{sr} \mapsto 0, \texttt{sp} \mapsto \mathcal{R}[\texttt{sp}] - 4] \quad \mathcal{M}' = \mathcal{M}[\mathcal{R}[\texttt{sp}] - 2 \mapsto \mathcal{R}[\texttt{pc}], \mathcal{R}[\texttt{sp}] - 4 \mapsto \mathcal{R}[\texttt{sr}]] \quad \mathcal{D} \vdash \delta, t, \bot \curvearrowright_D^{6} \delta', t', t'_a}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathtt{I}} \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc_{old}, \mathcal{B} \rangle}$$

(INT-UM-NP)

$$\frac{pc_{old} \vdash_{mode} \texttt{UM} \quad (\mathcal{R}[\texttt{sr}].\texttt{GIE} = 0 \vee t_a = \bot)}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathtt{I}} \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}$$

(INT-PM-P)

$$\frac{k = \texttt{MAX_TIME} - (t - t_a) \quad pc_{old} \vdash_{mode} \texttt{PM} \quad \mathcal{R}[\texttt{sr}].\texttt{GIE} = 1 \quad t_a \neq \bot \quad \mathcal{R}' = \mathcal{R}_0[\texttt{pc} \mapsto isr] \quad \mathcal{D} \vdash \delta, t, \bot \curvearrowright_D^{6+k} \delta', t', t'_a \quad \mathcal{B}' = \langle \mathcal{R}, pc_{old}, t - t_a \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathtt{I}} \langle \delta', t', \bot, \mathcal{M}, \mathcal{R}', pc_{old}, \mathcal{B}' \rangle}$$

(INT-PM-NP)

$$\frac{pc_{old} \vdash_{mode} \texttt{PM} \quad (\mathcal{R}[\texttt{sr}].\texttt{GIE} = 0 \vee t_a = \bot)}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathtt{I}} \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}$$

Figure 54: The transition system for handling interrupts in SancusL.

(CPU-Reti)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \quad i, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \texttt{OK} \quad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{M}[\mathcal{R}[\texttt{sp}] + 2], \texttt{sr} \mapsto \mathcal{M}[\mathcal{R}[\texttt{sp}]], \texttt{sp} \mapsto \mathcal{R}[\texttt{sp}] + 4] \quad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \bot \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{RETI}$$

(CPU-Reti-Chain)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \quad \mathcal{B} \neq \bot \quad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \quad \mathcal{R}[\texttt{sr.GIE}] = 1 \quad t'_a \neq \bot \quad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}, \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathtt{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{RETI}$$

(CPU-Reti-PrePad)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \quad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \quad \mathcal{B} \neq \bot \quad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \quad (\mathcal{R}[\texttt{sr.GIE}] = 0 \;\vee\; t'_a = \bot)}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta', t', t'_a, \mathcal{M}, \mathcal{B}.\mathcal{R}, \mathcal{B}.pc_{old}, \langle \bot, \bot, \mathcal{B}.t_{pad} \rangle \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{RETI}$$

(CPU-Reti-Pad)

$$\frac{\mathcal{B} = \langle \bot, \bot, t_{pad} \rangle \quad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{t_{pad}} \delta', t', t'_a \quad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \hookrightarrow_{\mathtt{I}} \langle \delta'', t'', t''_a, \mathcal{M}, \mathcal{R}', pc_{old}, \mathcal{B}' \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}, \mathcal{R}', pc_{old}, \mathcal{B}' \rangle}$$

Figure 55: Some rules from the operational semantics of SancusL.

6.5 THE SECURITY THEOREM

In this section we establish that Sancus$^{\mathbf{L}}$ enjoys the following security property: what an attacker can learn from an enclave is exactly the same before and after adding the support for interrupts. Technically, we show that the semantics of Sancus$^{\mathbf{L}}$ is *fully abstract* w.r.t. the semantics of Sancus$^{\mathsf{H}}$, i.e., in other words all the attacks that can be carried out in Sancus$^{\mathbf{L}}$ can also be carried out in Sancus$^{\mathsf{H}}$, and vice versa. Even though the technical details are specific to our case study, we are confident that the security definition applies also to other architectures, as outlined in Section 6.7.

Before stating the full abstraction theorem, we introduce some further notations, which also help in the main steps of its proof. Additional, minor lemmata and definitions for completing the proofs are in Appendices B.5 to B.9. Recall from Section 6.3.4 that $C[\mathcal{M}_M]$ is a whole program, where $\mathcal{M}_M$ is the software module and $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$ represents the context ($\mathcal{M}_C$ contains the unprotected program and data and $\mathcal{D}$ is the I/O device).

We first define the notion of convergence of whole programs.

Definition 6.4. *Let $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$ be a context, and $\mathcal{M}_M$ be a software module. A whole program $C[\mathcal{M}_M]$ converges in Sancus$^{\mathsf{H}}$ (written $C[\mathcal{M}_M]\Downarrow^{\mathsf{H}}$)* iff

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* \mathrm{HALT}.$$

Similarly, the same whole program converges in Sancus$^{\mathbf{L}}$ (written $C[\mathcal{M}_M]\Downarrow^{\mathbf{L}}$) iff

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* \mathrm{HALT}.$$

The following definition introduces the notion of contextual equivalence of two software modules. Roughly, the notion of contextual equivalence formalizes the intuitive notion of *indistinguishability*: two modules are contextually equivalent if they behave in the same way when they interact with an arbitrary, attacker-controlled context. Due to the quantification over *all* contexts, it suffices to consider just terminating and non-terminating executions as distinguishable, since any other distinction can be reduced to it.

Definition 6.5. *Two software modules $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ are contextually equivalent in Sancus$^{\mathsf{H}}$, written $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$,* iff

$$\forall C. \left(C[\mathcal{M}_M]\Downarrow^{\mathsf{H}} \iff C[\mathcal{M}_{M'}]\Downarrow^{\mathsf{H}} \right).$$

Similarly, $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ are contextually equivalent in Sancus$^{\mathbf{L}}$, written $\mathcal{M}_M \simeq^{\mathbf{L}} \mathcal{M}_{M'}$, iff

$$\forall C. \left(C[\mathcal{M}_M]\Downarrow^{\mathbf{L}} \iff C[\mathcal{M}_{M'}]\Downarrow^{\mathbf{L}} \right).$$

Eventually we state and prove the main theorem establishing the correctness of our mitigation:

Theorem 6.2 (Full abstraction)**.** $\forall \mathcal{M}_M, \mathcal{M}_{M'}. \ (\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'} \iff \mathcal{M}_M \simeq^{\mathbf{L}} \mathcal{M}_{M'})$.

Proof. Here we only present the "surface" of the proof by stating the main properties, whose proofs often require many other auxiliary definitions and properties that are detailed in the Appendix. Actually, the proof that our mitigation guarantees absence of interrupt-based attacks is rather long, and has the following steps. We first establish reflection of behaviors: $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'} \Leftarrow \mathcal{M}_M \simeq^{\mathbf{L}} \mathcal{M}_{M'}$ (Lemma 6.2 in Section 6.5.1). Then, the other implication, i.e., preservation of behaviors is proved by Lemma 6.8

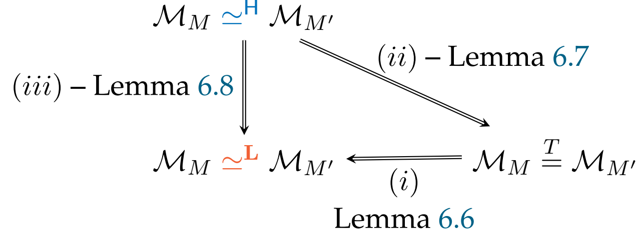


Figure 56: An illustration of the proof strategy of preservation of behaviors.

in Section 6.5.2 following the strategy summarized in Figure 56. We rely on the well-known notion of traces, i.e., the sequences of actions performed by a module $\mathcal{M}_M$ plugged in a context that can be observed by an attacker. In particular, we focus on the invocations of $\mathcal{M}_M$ and on the returns from it. In both cases our traces also carry information about the contents of the registers and for returns also the flow of time. We then say that two modules $\mathcal{M}_M$ and $\mathcal{M}'_M$ are trace equivalent, in symbols $\mathcal{M}_M \stackrel{T}{=} \mathcal{M}'_M$, if they exhibit the same traces (see Definition 6.7). Proving preservation is then done in two steps, the composition of which gives (iii) in Figure 56. First Lemma 6.7 establishes (ii) in Figure 56: two modules equivalent in $\text{Sancus}^{\mathsf{H}}$ are trace equivalent. The proof technique that we adopt specializes backtranslation of [176], applied to the contrapositive of (ii). Roughly, we construct a context in $\text{Sancus}^{\mathsf{H}}$ distinguishing two modules when they are not trace equivalent. Then Lemma 6.6 establishes (i) in Figure 56: two modules that are trace equivalent are also equivalent in $\mathbf{Sancus}^{\mathbf{L}}$. The proof of this lemma is rather technical: essentially, it consists in showing that neither the context affects the behavior of the module, nor the module affects that of the context.

Summing up:

- *Case* $\Leftarrow$. Reflection of behaviors follows from Lemma 6.2 in Section 6.5.1.
- *Case* $\Rightarrow$. Preservation of behaviors follows from Lemma 6.8 in Section 6.5.2. □

6.5.1 *Reflection of behaviors*

Recall that $\mathbf{Sancus}^{\mathbf{L}}$ differs from $\text{Sancus}^{\mathsf{H}}$ because of its interrupt handling mechanism, only. Consequently, to prove the reflection of behaviors, i.e., that for all $\mathcal{M}_M, \mathcal{M}_{M'}. \mathcal{M}_M \simeq^{\mathbf{L}} \mathcal{M}_{M'}$ implies $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$ it suffices to inhibit interrupts in $\mathbf{Sancus}^{\mathbf{L}}$, and for that we introduce the notion of *interrupt-less context* $C_{\not I}$ for a context C. In other words, $C_{\not I}$ behaves as C but never raises any interrupt. When a module is plugged in an interrupt-less context, it terminates according to the low level semantics if and only if it does in the high level semantics. Technically, to obtain the interrupt-less version of a context C it suffices removing in the device the transitions that may raise an interrupt.

Definition 6.6. *Let* $\mathcal{D} = \langle \Delta, \delta_{\text{init}}, \stackrel{a}{\leadsto}_D \rangle$ *be an I/O device. Given a context* $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$*, we define its corresponding* interrupt-less context *as* $C_{\not I} = \langle \mathcal{M}_C, \stackrel{a}{\leadsto}_{D_{\not I}} \rangle$ *where:*

- $\mathcal{D}_{\not I} = \langle \Delta, \delta_{\text{init}}, \stackrel{a}{\leadsto}_{D_{\not I}} \rangle$*, and*
- $\stackrel{a}{\leadsto}_{D_{\not I}} \triangleq \stackrel{a}{\leadsto}_D \cup \{(\delta, \epsilon, \delta') \mid (\delta, int?, \delta') \in \stackrel{a}{\leadsto}_D\} \setminus \{(\delta, int?, \delta') \mid (\delta, int?, \delta') \in \stackrel{a}{\leadsto}_D\}$.

Note that $\mathcal{D}_{\not I}$ is actually a device, due to the constraints on its transition function.

The behavior of interrupt-less contexts in $\mathbf{Sancus}^{\mathbf{L}}$ directly correspond to the behavior of their standard counterparts in $\text{Sancus}^{\mathsf{H}}$ as stated below.

Lemma 6.1. *For any module $\mathcal{M}_M$, context C, and corresponding interrupt-less context $C_{\not I}$:*

$$C_{\not I}[\mathcal{M}_M]\Downarrow^{\mathbf{L}} \iff C[\mathcal{M}_M]\Downarrow^{\mathsf{H}}$$

Reflection now follows, because whole programs in $\mathsf{Sancus}^{\mathsf{H}}$ behave just like a subset of whole programs in $\mathbf{Sancus}^{\mathbf{L}}$.

Lemma 6.2 (Reflection)**.** $\forall \mathcal{M}_M, \mathcal{M}_{M'}. (\mathcal{M}_M \simeq^{\mathbf{L}} \mathcal{M}_{M'} \implies \mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'})$.

6.5.2 *Preservation of behaviors*

Here, we prove the preservation of behaviors, i.e., the chain of implications (ii) and then (i), resulting in (iii) in Figure 56. More precisely we perform the following steps.

In Section 6.5.2.1 we first define two notions of traces: the *fine-grained* and *coarse-grained* traces. The first is an auxiliary notion that directly derives from the semantics of $\mathbf{Sancus}^{\mathbf{L}}$ and facilitates the proofs. Intuitively, it takes into account all the actions performed by the system. The second kind of traces only records the actions that attackers can observe, and are easily derived from the fine-grained ones. Also, we call trace equivalent two modules with the same set of coarse-grained traces. Using the fine-grained traces, we state and prove the key yet rather technical Proposition 6.1 ensuring that our mitigation reflects the intuition described in Figure 51.

Then we prove in Section 6.5.2.2 that trace equivalence implies contextual equivalence at $\mathbf{Sancus}^{\mathbf{L}}$ (the implication (i) of Figure 56). For that Lemma 6.5 is crucial, since it ensures that two trace equivalent modules still produce the same traces when plugged in a given context.

Next, in Section 6.5.2.3 we prove that contextual equivalence implies trace equivalence at $\mathsf{Sancus}^{\mathsf{H}}$ (the implication (ii) of Figure 56). This is achieved by defining a *backtranslation* [176] that given an attacker (a context that differentiate two modules) at $\mathbf{Sancus}^{\mathbf{L}}$ returns an attacker at $\mathsf{Sancus}^{\mathsf{H}}$.

Finally, Section 6.5.2.4 immediately concludes our proof of item (iii) in Figure 56.

6.5.2.1 *Fine-grained and coarse-grained traces*

It is convenient to consider two kinds of traces: the fine-grained and the coarse-grained ones. The first records the relevant actions performed by the processors including those concerned with interrupt handling. The coarse-grained, instead, presents what the attacker is able to observe, i.e., jumping in and out an enclave.

The fine-grained observables are defined as follows:

$$\alpha ::= \xi \mid \tau(k) \mid \texttt{reti?}(k) \mid \texttt{handle!}(k) \mid \bullet \mid \texttt{jmpIn?}(\mathcal{R}) \mid \texttt{jmpOut!}(k; \mathcal{R}).$$

Above, $k \in \mathbb{N}$ indicates that the observed action takes k cycles. Intuitively, ξ denotes unobservable actions performed by the context; $\tau(k)$ indicates an internal action; $\texttt{handle!}(k)$ and $\texttt{reti?}(k)$ denote when the processor starts executing the interrupt service routine from protected mode and when it returns from it, respectively. Then, the observable $\bullet$ indicates that termination occurred; $\texttt{jmpIn?}(\mathcal{R})$ and $\texttt{jmpOut!}(k; \mathcal{R})$ record when the CPU enters and exits from protected mode, respectively, where $\mathcal{R}$ is the contents of the register file when the action ends.

The relation $\overset{\alpha}{\Longrightarrow}$ in Figure 57 extracts observables from the execution of a whole program. Note that each transition $\mathcal{D} \vdash c \rightarrow c'$ has a corresponding transition $\mathcal{D} \vdash c \overset{\alpha}{\Longrightarrow} c'$ for some α, possibly the non-observable ξ. The transitive and reflexive closure of $\overset{\alpha}{\Longrightarrow}$ is $\overset{\overline{\alpha}}{\Longrightarrow}^*$, where $\overline{\alpha}$ is a trace, i.e., a sequence of actions (ϵ is the empty trace).

(Obs-Internal-PM)

$$\frac{\mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{PM} \qquad \mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \bot \rangle \qquad \mathcal{R}'[\mathtt{pc}] \vdash_{mode} \mathtt{PM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \xRightarrow{\tau(k)} \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \bot \rangle}$$

(Obs-JmpIn)

$$\frac{\mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{UM} \qquad \mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \rightarrow \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \bot \rangle \qquad \mathcal{R}'[\mathtt{pc}] \vdash_{mode} \mathtt{PM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \xRightarrow{\mathtt{jmpIn?}(\mathcal{R}')} \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \bot \rangle}$$

(Obs-Reti)

$$\frac{\mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{UM} \qquad \mathcal{B} \neq \bot \qquad \mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \langle \bot, \bot, t_{pad} \rangle \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \xRightarrow{\mathtt{reti?}(k)} \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \langle \bot, \bot, t_{pad} \rangle \rangle}$$

(Obs-JmpOut)

$$\frac{\mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{PM} \qquad \mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \rightarrow \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \bot \rangle \qquad \mathcal{R}'[\mathtt{pc}] \vdash_{mode} \mathtt{UM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \xRightarrow{\mathtt{jmpOut!}(k;\mathcal{R}')} \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \mathcal{B}' \rangle}$$

(Obs-JmpOut-PostPoned)

$$\frac{\mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{UM} \quad \mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \langle \bot, \bot, t_{pad} \rangle \rangle \rightarrow \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \bot \rangle \qquad \mathcal{R}'[\mathtt{pc}] \vdash_{mode} \mathtt{UM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \langle \bot, \bot, t_{pad} \rangle \rangle \xRightarrow{\mathtt{jmpOut!}(k;\mathcal{R}')} \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \mathcal{B}' \rangle}$$

(Obs-Handle)

$$\frac{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \rightarrow \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \mathcal{B}' \rangle \qquad \mathcal{R}'[\mathtt{pc}] \vdash_{mode} \mathtt{UM} \qquad \mathcal{B}' \neq \bot}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \xRightarrow{\mathtt{handle!}(k)} \langle \delta', t+k, t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \mathcal{B}' \rangle}$$

(Obs-Internal-UM)

$$\frac{\mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{UM} \qquad \mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \mathcal{B} \rangle \qquad \mathcal{R}'[\mathtt{pc}] \vdash_{mode} \mathtt{UM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \xRightarrow{\xi} \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc'_{old}, \mathcal{B} \rangle}$$

(Obs-Final)

$$\frac{\mathcal{R}[\mathtt{pc}] \vdash_{mode} \mathtt{UM} \qquad \mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{HALT}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \xRightarrow{\bullet} \mathrm{HALT}}$$

Figure 57: The relation $\xRightarrow{\alpha}$ for fine-grained observables.

$$\frac{\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\xi\cdots\xi\cdot\texttt{jmpIn?}(\mathcal{R})}{}^{*} c}{\mathcal{D} \vdash_{C[\mathcal{M}_M]} \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\texttt{jmpIn?}(\mathcal{R})} c} \qquad \frac{\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\xi\cdots\xi\cdot\bullet}{}^{*} \text{HALT}}{\mathcal{D} \vdash_{C[\mathcal{M}_M]} \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\bullet} \text{HALT}}$$

$$\frac{\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \rightarrow^{*} c \qquad \mathcal{D} \vdash_{C[\mathcal{M}_M]} c \xRightarrow{\texttt{jmpOut!}(\Delta t;\mathcal{R}')} c' \qquad \mathcal{D} \vdash c' \xRightarrow{\xi\cdots\xi\cdot\texttt{jmpIn?}(\mathcal{R}'')}{}^{*} c''}{\mathcal{D} \vdash_{C[\mathcal{M}_M]} c' \xRightarrow{\texttt{jmpIn?}(\mathcal{R}'')} c''}$$

$$\frac{\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \rightarrow^{*} c \qquad \mathcal{D} \vdash_{C[\mathcal{M}_M]} c \xRightarrow{\texttt{jmpOut!}(\Delta t;\mathcal{R}')} c' \qquad \mathcal{D} \vdash c' \xRightarrow{\xi\cdots\xi\cdot\bullet}{}^{*} \text{HALT}}{\mathcal{D} \vdash_{C[\mathcal{M}_M]} c' \xRightarrow{\bullet} \text{HALT}}$$

$$\frac{\begin{array}{c}\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \rightarrow^{*} c \qquad \mathcal{D} \vdash_{C[\mathcal{M}_M]} c \xRightarrow{\texttt{jmpIn?}(\mathcal{R}')} c' \\ \mathcal{D} \vdash c' \xRightarrow{\alpha^{(0)}\cdots\alpha^{(n-1)}\cdot\texttt{jmpOut!}(k'';\mathcal{R}'')}{}^{*} c'' \qquad \forall 0 \leq i < n.\, \alpha_i \notin \{\texttt{jmpOut!}(_;_), \bullet\} \qquad \Delta t = k'' + \sum_{i=0}^{n-1} \mathit{time}(\alpha^{(i)})\end{array}}{\mathcal{D} \vdash_{C[\mathcal{M}_M]} c' \xRightarrow{\texttt{jmpOut!}(\Delta t;\mathcal{R}'')} c''}$$

$$\frac{\begin{array}{c}\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \rightarrow^{*} c \\ \mathcal{D} \vdash_{C[\mathcal{M}_M]} c \xRightarrow{\texttt{jmpIn?}(\mathcal{R}')} c' \qquad \mathcal{D} \vdash c' \xRightarrow{\alpha_0\cdots\alpha_{n-1}\cdot\bullet}{}^{*} \text{HALT} \qquad \forall 0 \leq i < n.\, \alpha_i \notin \{\texttt{jmpOut!}(_;_), \bullet\}\end{array}}{\mathcal{D} \vdash_{C[\mathcal{M}_M]} c' \xRightarrow{\bullet} \text{HALT}}$$

$$\text{where} \quad \mathit{time}(\alpha) = \begin{cases} k & \text{if } \alpha \in \{\texttt{reti?}(k), \texttt{handle!}(k), \tau(k), \texttt{jmpOut!}(k;\mathcal{R})\} \\ 0 & \text{o.w.} \end{cases}$$

Figure 58: The relation $\xRightarrow{\beta}$ for coarse-grained observables.

Note that in any trace $\overline{\alpha}$, *only* the observables $\tau(k), \texttt{reti?}(k)$ or $\texttt{handle!}(k)$ may occur between a $\texttt{jmpIn?}(\mathcal{R})$ and a $\texttt{jmpOut!}(k;\mathcal{R})$. When an interrupt has to be handled, the observed trace starts with $\texttt{handle!}(\cdot)$, followed by a sequence of ξ and then a $\texttt{reti?}(k)$, provided that a RETI is executed (k always has value $\mathit{cycles}(\texttt{RETI})$). If the interrupted instruction was a jump from protected to unprotected mode, the $\texttt{reti?}(\cdot)$ is followed by a $\texttt{jmpOut!}(\cdot;\cdot)$ (cfr. rules (Obs-Handle), (Obs-Internal-UM), (Obs-Reti) and (Obs-JmpOut-PostPoned)); otherwise a $\tau(\cdot)$ – or a $\texttt{handle!}(\cdot)$ if an interrupt has to be handled.

Actually, an attacker (i.e., the context) cannot observe all α's, but only the following coarse-grained observables, where $\texttt{jmpIn?}(\mathcal{R})$ and $\texttt{jmpOut!}(\Delta t;\mathcal{R})$ represent invoking a module and returning from it.

$$\beta ::= \bullet \mid \texttt{jmpIn?}(\mathcal{R}) \mid \texttt{jmpOut!}(\Delta t;\mathcal{R}).$$

In Figure 58 we define the relation $\mathcal{D} \vdash_{C[\mathcal{M}_M]} c \xRightarrow{\beta} c'$, where $C[\mathcal{M}_M]$ is the initial whole program from which the computation started; $\mathcal{D}$ is the device specified by the context C; and c and c' are two configurations. From now onward, we feel free to abuse the notation and omit the index $C[\mathcal{M}_M]$ when clear from the context. Essentially, we remove the observables for interrupts and silent actions from the fine-grain traces, making them not visible any longer. More in detail, all the actions in between a $\texttt{jmpIn?}(\cdot)$ and the immediately following $\texttt{jmpOut!}(k;\mathcal{R})$ (or a $\bullet$) are dropped; similarly for the fine-grained observables in between a $\texttt{jmpOut!}(k;\mathcal{R})$ (or the very first observable from the initial configuration) and the next $\texttt{jmpIn?}(\cdot)$. In addition, the parameter k is replaced by Δt in the observable $\texttt{jmpOut!}(\Delta t;\mathcal{R})$ to model that an attacker can only measure the end-to-end time of a piece of code running in protected mode. Taking $\xRightarrow{\overline{\beta}}{}^{*}$ as the reflexive and transitive closure of the relation $\xRightarrow{\beta}$ defined in Figure 58 (where traces $\overline{\beta}$ are strings of β's), we eventually define when two modules are trace equivalent:

Definition 6.7. *Two modules are* (coarse-grained) trace equivalent, *written* $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$, iff

$$Tr(\mathcal{M}_M) = Tr(\mathcal{M}_{M'}).$$

where $Tr(\mathcal{M}_M) \triangleq \{\overline{\beta} \mid \exists C = \langle \mathcal{M}_C, \mathcal{D} \rangle . \mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}^* c'\}$.

NOTATION Hereafter let $x \in \{1, 2\}$; let $c, c_1, c_2, \ldots$, possibly dashed, be configurations; and let $c_x^{(n)} = \langle \delta_x^{(n)}, t_x^{(n)}, t_{a_x}^{(n)}, \mathcal{M}_x^{(n)}, \mathcal{R}_x^{(n)}, pc_{old\,x}^{(n)}, \mathcal{B}_x^{(n)} \rangle$ be the configuration reached after n execution steps from the initial configuration $c_x^{(0)}$. We will index the elements of a trace and the components of a context C_x in a similar way. Finally, let $c_x^{(i)}$ be the configuration *right before* the action of index i in a given (fine- or coarse-grained) trace.

To prove a crucial property of our mitigation, it is convenient to introduce the notion of *complete interrupt segments* of a fine-grained trace, which are those starting with an $\texttt{handle!}(\cdot)$ action and ending with a $\texttt{reti?}(\cdot)$ action (see Definition B.1 in Appendix B.5). Also, let $|\mathbb{I}_{\overline{\alpha}}|$ be the number of the complete interrupt segments in a given trace $\overline{\alpha}$.

The proposition below characterizes how our mitigation affects the execution time of a module (thus excluding the time spent executing the attacker's code). Intuitively, it ensures that handling each interrupt contributes to the time spent in *protected mode* with a constant number of cycles equal to $11 + \text{MAX_TIME}$. This is crucial to guarantee a constant delay *before and after* interrupt handling, otherwise an attacker would be able to observe different timings as it happens in Examples 6.1 and 6.3.

Proposition 6.1. *If* $c^{(0)} \vdash_{mode} \texttt{PM}$ *and* $\mathcal{D} \vdash c^{(0)} \overset{\overline{\alpha}}{\Longrightarrow}^* c^{(n+1)}$, *with* $\overline{\alpha} = \alpha^{(0)} \cdots \alpha^{(n-1)} \cdot \texttt{jmpOut!}(k^{(n)}; \mathcal{R}')$, *then* $k^{(n)} + \sum_{i=0}^{n-1} time(\alpha^{(i)}) = \sum_{i=0}^{n} \gamma(c^{(i)}) + (11 + \text{MAX_TIME}) \cdot |\mathbb{I}_{\overline{\alpha}}|$, *where*

$$\gamma(c) \triangleq \begin{cases} cycles(decode(\mathcal{M}, \mathcal{R}[\texttt{pc}])) & \text{if } c \vdash_{mode} \texttt{PM} \wedge \mathcal{B} = \bot \\ 0 & o.w. \end{cases}$$

Proof. By definition of the interrupt logic and the operational semantics of Sancus$^{\mathbf{L}}$, for each interrupt handled in protected mode we perform a $0 \leq k \leq \text{MAX_TIME}$ padding *before* invoking the interrupt service routine and an additional padding of $(\text{MAX_TIME} - k)$ cycles *after* its execution, i.e., the padding time introduced for each complete interrupt segment amounts to MAX_TIME. Also, since the interrupt logic always requires 6 cycles to jump to the interrupt service routine and 5 cycles are required upon RETI it easily follows that:

$$k^{(n)} + \sum_{i=0}^{n-1} time(\alpha^{(i)}) = \sum_{i=0}^{n} \gamma(c^{(i)}) + (11 + \text{MAX_TIME}) \cdot |\mathbb{I}_{\overline{\alpha}}|. \qquad \square$$

6.5.2.2 *Trace equivalence implies contextual equivalence at* Sancus$^{\mathbf{L}}$

Here we prove the implication (i) of Figure 56, i.e., that $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'} \implies \mathcal{M}_M \simeq^{\mathbf{L}} \mathcal{M}_{M'}$. We rely on the following proposition to ensure that a terminating program generates a coarse-grained trace ending with $\bullet$, and vice versa.

Proposition 6.2. $C[\mathcal{M}_M]\Downarrow^{\mathbf{L}}$ iff $\exists \overline{\beta}.\ \mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}\cdot\bullet}{\Longrightarrow}^* \text{HALT}$.

Proof. The *only-if* part holds trivially. For the other direction, the definition of $C[\mathcal{M}_M]\Downarrow^{\mathbf{L}}$ implies that $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \rightarrow^* \text{HALT}$ and the definitions of fine- and coarse-grained traces (Figures 57 and 58) guarantee that the last observed action is $\bullet$ as requested. $\square$

Consider two whole programs that share the same context. The lemma below states that if they perform the same sequence of actions reaching an unprotected configuration, then their next action, if any, will be the same. Intuitively, this is because the context is deterministic and because our mitigation makes the context behavior independent of the module. Recall that coarse-grained traces record timing information, and therefore this lemma and the next one also express timing independence between contexts and modules.

Lemma 6.3. *Let* $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$*. If* $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}}{}^{*} c_1 \xRightarrow{\beta} c_1'$*,* $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}}{}^{*} c_2$*,* $c_1 \vdash_{mode} \texttt{UM}$ *and* $c_2 \vdash_{mode} \texttt{UM}$*, then* $\mathcal{D} \vdash c_2 \xRightarrow{\beta} c_2'$*.*

We introduce below the (simulation based) equivalence $\overset{U}{\approx}$ that helps in proving this lemma, because this equivalence is preserved when moving from one unprotected configuration to another (Proposition B.19). Roughly, $\overset{U}{\approx}$ relates configurations that are equivalent from an attacker's point of view (e.g., whenever the two configurations share the contents of registers or of the unprotected memory):

Definition 6.8 (Definition 6.8)**.** *We say that two configurations are* U-equivalent *(written* $c \overset{U}{\approx} c'$*)* iff

$$(c = c' = \text{HALT}) \vee$$
$$(c = \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \ \wedge \ c' = \langle \delta', t', t_a', \mathcal{M}', \mathcal{R}', pc_{old}', \mathcal{B}' \rangle \ \wedge \ \mathcal{M} \overset{U}{=} \mathcal{M}' \ \wedge$$
$$c \vdash_{mode} \texttt{m} \ \wedge \ c' \vdash_{mode} \texttt{m} \ \wedge \ \delta = \delta' \ \wedge \ t = t' \ \wedge \ t_a = t_a' \ \wedge \ \mathcal{R} \overset{\texttt{UM}}{\asymp}_{\texttt{m}} \mathcal{R}' \ \wedge \ \mathcal{B} \bowtie \mathcal{B}')$$

where

- $\mathcal{M} \overset{U}{=} \mathcal{M}'$ *iff* $\forall l \notin [ts, te) \cup [ds, de).\ M[l] = \mathcal{M}'[l]$
- $\mathcal{R} \overset{\texttt{UM}}{\asymp}_{\texttt{m}} \mathcal{R}'$ *iff* $(\texttt{m} = \texttt{UM} \implies \mathcal{R} = \mathcal{R}') \ \wedge \ \mathcal{R}[\texttt{sr.GIE}] = \mathcal{R}'[\texttt{sr.GIE}]$
- $\mathcal{B} \bowtie \mathcal{B}'$ *iff* $(\mathcal{B} = \bot \iff \mathcal{B}' = \bot)$

Conversely, the following lemma shows that the behavior of the module is not influenced by the context (due to the isolation mechanism offered by the enclave):

Lemma 6.4. *Let* $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$*. If* $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$*,* $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}}{}^{*} c_1'' \xRightarrow{\texttt{jmpIn?}(\mathcal{R}_1)} c_1 \xRightarrow{\beta} c_1'$ *and* $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}}{}^{*} c_2'' \xRightarrow{\texttt{jmpIn?}(\mathcal{R}_2)} c_2$*, then* $\mathcal{D} \vdash c_2 \xRightarrow{\beta} c_2'$*.*

Analogously to Lemma 6.3, also this proof is based on a simulation argument. In this case we prove the preservation of the relation $\overset{P}{\approx}$, which intuitively relates configurations that are equivalent from a module's point of view:

Definition 6.9. *We say that two configurations are* P-equivalent *(written* $c \overset{P}{\approx} c'$*)* iff

$$(c = c' = \text{HALT}) \vee$$
$$(c = \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \ \wedge \ c' = \langle \delta', t', t_a', \mathcal{M}', \mathcal{R}', pc_{old}', \mathcal{B}' \rangle \ \wedge \ \mathcal{M} \overset{P}{=} \mathcal{M}' \ \wedge$$
$$pc_{old} \vdash_{mode} \texttt{m} \ \wedge \ pc_{old}' \vdash_{mode} \texttt{m} \ \wedge \ \mathcal{R} \overset{\texttt{PM}}{\asymp}_{\texttt{m}} \mathcal{R}' \ \wedge \ \mathcal{B} \bowtie \mathcal{B}')$$

where

- $\mathcal{M} \stackrel{P}{=} \mathcal{M}'$ *iff* $\forall l \in [ts, te) \cup [ds, de).\ M[l] = \mathcal{M}'[l]$.
- $\mathcal{R} \overset{\texttt{PM}}{\asymp}_{\texttt{m}} \mathcal{R}'$ *iff* $(\texttt{m} = \texttt{PM} \implies \mathcal{R} = \mathcal{R}')$
- $\mathcal{B} \bowtie \mathcal{B}'$ *iff* $(\mathcal{B} = \bot \iff \mathcal{B}' = \bot)$.

The two lemmata above imply the following:

Lemma 6.5. *Given a context $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$ and two modules $\mathcal{M}_M$ and $\mathcal{M}_{M'}$. If $\mathcal{M}_M \stackrel{T}{=} \mathcal{M}_{M'}$ and $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}}{}^* c_1$, then there exists a c_2 such that $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}}{}^* c_2$.*

Proof. We show this by induction on the length n of $\overline{\beta}$.

- *Case* $n = 0$. Since $\overline{\beta} = \varepsilon$, by definition of $\xRightarrow{\cdot}{}^*$, we have $c_1 = \text{INIT}_{C[\mathcal{M}_M]} = c_1$. Again, by definition of $\xRightarrow{\cdot}{}^*$, we choose $c_2 = \text{INIT}_{C[\mathcal{M}_{M'}]}$ and get the thesis.
- *Case* $n = n' + 1$. The induction hypothesis (IHP) is then:

  $$\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}'}{}^* c_1' \Rightarrow \mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}'}{}^* c_2'$$

  and we must show that

  $$\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}'}{}^* c_1' \xRightarrow{\beta} c_1 \Rightarrow \mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}'}{}^* c_2' \xRightarrow{\beta} c_2$$

  By cases on the CPU mode in c_1' and c_2':
  - $\mathcal{R}_1'[\texttt{pc}] \vdash_{mode} \texttt{UM}$ *and* $\mathcal{R}_2'[\texttt{pc}] \vdash_{mode} \texttt{UM}$: Follows by (IHP) and Lemma 6.3;
  - $\mathcal{R}_1'[\texttt{pc}] \vdash_{mode} \texttt{PM}$ *and* $\mathcal{R}_2'[\texttt{pc}] \vdash_{mode} \texttt{PM}$: Follows by (IHP) and Lemma 6.4;
  - $\mathcal{R}_1'[\texttt{pc}] \vdash_{mode} \texttt{m}$ *and* $\mathcal{R}_2'[\texttt{pc}] \vdash_{mode} \texttt{m}'$ *and* $\texttt{m} \neq \texttt{m}'$: It never happens, as observed in Proposition B.21. □

We eventually conclude the proof that if two modules are trace equivalent then they are contextually equivalent in Sancus$^{\mathbf{L}}$ (arrow (i) in Figure 56):

Lemma 6.6. *If $\mathcal{M}_M \stackrel{T}{=} \mathcal{M}_{M'}$ then $\mathcal{M}_M \simeq^{\mathbf{L}} \mathcal{M}_{M'}$.*

Proof. Expanding the definition of $\simeq^{\mathbf{L}}$, the statement becomes:

$$\mathcal{M}_M \stackrel{T}{=} \mathcal{M}_{M'} \Rightarrow (\forall C = \langle \mathcal{M}_C, \mathcal{D} \rangle . C[\mathcal{M}_M]\Downarrow^{\mathbf{L}} \iff C[\mathcal{M}_{M'}]\Downarrow^{\mathbf{L}})$$

We split the double implication and we show the two cases independently.

- *Case* $\Rightarrow$. By Proposition 6.2 there exists $\overline{\beta}$ such that $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\overline{\beta}\cdot\bullet}{}^* \text{HALT}$. Since $\mathcal{M}_M \stackrel{T}{=} \mathcal{M}_{M'}$, we know by Lemma 6.5 that $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}\cdot\bullet}{}^* \text{HALT}$. Thus, again by Proposition 6.2, we have $C[\mathcal{M}_{M'}]\Downarrow^{\mathbf{L}}$;
- *Case* $\Leftarrow$. Symmetric to the previous one. □

6.5.2.3 *Contextual equivalence at* Sancus$^{\mathsf{H}}$ *implies trace equivalence*

Here we prove by contraposition that $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'} \implies \mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$, i.e., implication (ii) of Figure 56.

We first define those traces, if any, that *distinguish* a given a pair of modules, i.e., one converges while the other does not. Given a context in **Sancus$^{\mathbf{L}}$** that keeps two modules apart through two such traces, we define two algorithms: the first builds a memory and the other a device. Once put together, they implement a *backtranslation* [176] and return a context differentiating the two modules in Sancus$^{\mathsf{H}}$. Because of the strong limitations of MSP430 (e.g., it only has 64KB of memory) building such a context in unprotected memory only is infeasible. Since the attacker model we assumed has the strong power of controlling everything except the enclave, it is also assumed to control the I/O device that has unlimited memory. Therefore, the backtranslation takes full advantage of such a strength to build a distinguishing context.

We start from two distinguishing traces, that consist of a common prefix followed by two further traces starting with two different observables. We then make sure that there always exist such traces for two modules that are kept apart in **Sancus$^{\mathbf{L}}$**.

Definition 6.10 (Distinguishing traces)**.** *Let $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ be two modules, and let $\overline{\beta} = \overline{\beta}_s \cdot \beta \cdot \overline{\beta}_e \in Tr(\mathcal{M}_M)$ and $\overline{\beta}' = \overline{\beta}_s \cdot \beta' \cdot \overline{\beta}'_e \in Tr(\mathcal{M}_{M'})$. We say that $\overline{\beta}$ and $\overline{\beta}'$ are* distinguishing traces for $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ *iff there exist a context $C^L = \langle \mathcal{M}_{C^L}, \mathcal{D}^L \rangle$ such that*

- $\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c$ *and* $\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_{M'}]} \overset{\overline{\beta}'}{\Longrightarrow}{}^* c'$*, for some* c, c'*;*
- $\overline{\beta} \notin Tr(\mathcal{M}_{M'})$*,* $\overline{\beta}' \notin Tr(\mathcal{M}_M)$ *and* $\beta \neq \beta'$*.*

Proposition 6.3. *If $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ are two modules such that $\mathcal{M}_M \not\simeq^{\mathbf{L}} \mathcal{M}_{M'}$, then there always exist $\overline{\beta}$ and $\overline{\beta}'$ that are distinguishing traces for $\mathcal{M}_M$ and $\mathcal{M}_{M'}$.*

FIRST ALGORITHM: MEMORY INITIALIZATION. Intuitively, given two modules and two distinguishing traces $\overline{\beta} = \overline{\beta}_s \cdot \beta \cdot \overline{\beta}_e$ and $\overline{\beta}' = \overline{\beta}_s \cdot \beta' \cdot \overline{\beta}'_e$ for them, the Algorithm 2 builds the memory of the wanted distinguishing context. Actually, this memory only contains the code that cooperates with the I/O device built in Algorithm 3 to mimic the target context and to differentiate the two modules in hand. Intuitively, the generated code communicates the state of the CPU to the I/O device enabling it to drive the context execution and thus the behavior of the processor.

Assume as given an *assembler* function *encode* that returns the encoding of any assembly instruction as one or two words – according to the size specified in Table 3 (Page 116). Also, assume that the unprotected memory is large enough to contain the code of the context we are building (there is almost no loss of generality since the space required for this code is bounded by a constant (≤ 25 words) plus the number of different addresses to which the protected code jumps – kept anyway in the unprotected memory). Suppose also to have the five constants `A_HALT`, `A_LOOP`, `A_JIN`, `A_EP` and `A_RDIFF` representing addresses in the unprotected memory: they are assumed different from (i) each other, (ii) `0xFFFE` and (iii) any address $\mathcal{R}[\texttt{pc}]$ such that $\texttt{jmpOut!}(\Delta t; \mathcal{R})$ occurs in either input distinguishing traces. Finally, assume for simplicity that the modules never jump to `0xFFFE`.[1]

1 Slightly changing Algorithm 2 suffices to remove this limitation: upon the jump into protected mode right before jumping to `0xFFFE`, the context writes the right code to deal with it in `0xFFFE` and, afterwards, restores the old content of that address.

First, the algorithm initializes the memory $\mathcal{M}_C$ by filling it with the code in Figure 59. It consists of five parts. The first two are for convergence (Line 1) and divergence (Line 3). The next part (Lines 5 to 20) inputs the registers values from the device and then jumps into the enclave. Line 25 specifies that the first instruction to be executed is at the address specified by `A_EP`. Finally, the code in Lines 22 and 23 interacts with the device to get the next instruction to execute.

Then, the algorithm compares β and β'. If they are both $\texttt{jmpOut!}(\cdot;\cdot)$ and at least one register has different values in the observables, two cases arise:

- If one of the registers differentiating β and β' is $\texttt{r} \neq \texttt{pc}$, then we add the code (Lines 5 to 14, Algorithm 2) to request a new program counter (that will depend on the value of `r`) to the device, starting at address `A_RDIFF`;

- Otherwise, the register differentiating $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R})$ and $\beta' = \texttt{jmpOut!}(\Delta t; \mathcal{R}')$ is `pc`. In this case, we store in $\mathcal{M}_C$ the instructions to ask the device a new program counter at the address $\mathcal{R}[\texttt{pc}]$ and $\mathcal{R}'[\texttt{pc}]$ for the first and second module, respectively (Lines 15 to 19, Algorithm 2). Also, we record the differentiating values of the program counter in $joutd$ and $joutd'$, to be used by Algorithm 3.

Finally, the algorithm adds the code to deal with jumps out from the protected module to unprotected code for any $\texttt{jmpOut!}(\Delta t; \mathcal{R})$ in $\overline{\beta}_s \cdot \beta$ or $\overline{\beta}_s \cdot \beta'$ such that $\mathcal{R}[\texttt{pc}] \neq joutd$ and $\mathcal{R}[\texttt{pc}] \neq joutd'$. Since the code cannot track timing directly, we delegate the device to deal with the case when the observables differ on timings, i.e., when $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R})$ and $\beta' = \texttt{jmpOut!}(\Delta t'; \mathcal{R})$ with $\Delta t \neq \Delta t'$ (see Algorithm 3). Eventually, the algorithm returns the memory built and the values of $joutd$ and $joutd'$ (if any), used by Algorithm 3 to build the distinguishing device.

Algorithm 2 Builds the memory of the distinguishing context.

```
 1: procedure BuildMem(β̄ = β̄_s · β · β̄_e, β̄' = β̄_s · β' · β̄'_e)
 2:     ▷ β̄ and β̄' are distinguishing traces w. common prefix β̄_s
 3:     joutd = joutd' = ⊥
 4:     M_C = filled as described in Figure 59
 5:     if β = jmpOut!(Δt; R) ∧ β' = jmpOut!(Δt; R') ∧ (∃r. R[r] ≠ R'[r]) then
 6:         if r ≠ pc then
 7:             M_C = M_C[A_RDIFF ↦ encode(OUT r), A_RDIFF + 1 ↦ encode(IN pc)]
 8:         else
 9:             joutd = R[pc]
10:             joutd' = R'[pc]
11:             M_C = M_C[joutd ↦ encode(OUT pc), joutd + 1 ↦ encode(IN pc)]
12:             M_C = M_C[joutd' ↦ encode(OUT pc), joutd' + 1 ↦ encode(IN pc)]
13:         end if
14:     end if
15:     for jmpOut!(Δt; R) ∈ β̄_s · β, β̄_s · β' do
16:         if R[pc] ≠ joutd ∧ R[pc] ≠ joutd' then
17:             M_C = M_C[R[pc] ↦ encode(IN pc)]
18:         end if
19:     end for
20:     return (M_C, joutd, joutd')
21: end procedure
```

```
1  A_HALT. HLT
2
3  A_LOOP. JMP pc
4
5  A_JIN . IN sp
6        . IN sr
7        . IN R3
8        . IN R4
9        . IN R5
10       . IN R6
11       . IN R7
12       . IN R8
13       . IN R9
14       . IN R10
15       . IN R11
16       . IN R12
17       . IN R13
18       . IN R14
19       . IN R15
20       . IN pc
21
22 A_EP  . OUT pc
23       . IN pc
24
25 0xFFFE. A_EP ; the content of 0xFFFE is A_EP
```

Figure 59: Initial content of unprotected memory as used by Algorithm 2.

SECOND ALGORITHM: DEVICE CONSTRUCTION. This second algorithm iteratively builds a device that cooperates with the memory of the context given by Algorithm 2 to distinguish $\mathcal{M}_M$ from $\mathcal{M}_{M'}$. The algorithm is in Appendix B.8, and here we only briefly comment on it.

Let *joutd* and *joutd'* be the addresses returned by Algorithm 2 (if any) and that represent the differentiating values of the program counter; let $\overline{\beta} = \overline{\beta}_s \cdot \beta \cdot \overline{\beta}_e$ and $\overline{\beta}' = \overline{\beta}_s \cdot \beta' \cdot \overline{\beta}'_e$ ($\beta \neq \beta'$) be two distinguishing traces for $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ under C^L; finally, let *term* (resp. *term'*) denote whether $\mathcal{M}_M$ (resp. $\mathcal{M}_{M'}$) converges in a context with no interrupts after the last jump into protected mode. The algorithm starts with an empty device and iterates over the observables β_i in $\overline{\beta}_s$:

- *Case* $\beta_i = \texttt{jmpIn?}(\mathcal{R})$.
  In this case either this is the first observable or $\beta_{i-1} = \texttt{jmpOut!}(\cdot;\cdot)$. According to Algorithm 2, in both cases we reach the instruction `IN pc` (either at address `A_EP` or those of jumps out of protected mode), waiting for the next program counter. The algorithm appends the behavior described in Figure 60a to the device built so far. Intuitively, the device ignores possible write operations and outputs the special address `A_JIN`. Then, it starts sending the values of the registers in $\mathcal{R}$, so to simulate in Sancus$^{\text{H}}$ what happens in Sancus$^{\text{L}}$ and to match the code requests.

- *Case* $\beta_i = \texttt{jmpOut!}(\Delta t; \mathcal{R})$.
  The device is simply updated with an ϵ-loop on the last added state δ_L and ignores write operations (to deal with $\mathcal{R}[\texttt{pc}] = joutd$ or $\mathcal{R}[\texttt{pc}] = joutd'$). Figure 60b pictorially represents this case.

Then, as soon as β and β' show up, the algorithm sets up the device to differentiate the modules:

- *Case* $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \beta' = \texttt{jmpOut!}(\Delta t'; \mathcal{R}') \wedge (\exists \texttt{r}.\, \mathcal{R}[\texttt{r}] \neq \mathcal{R}'[\texttt{r}])$.
  In this case the differentiation is due to a register, and two further sub-cases may arise. If the register is `pc` then the device gets the differentiating value from the context (executing code at $joutd$ and $joutd'$ by construction); based on that value, it outputs either `A_HALT` or `A_LOOP` (see Figure 61a). For any other register than `pc`, the context waits for the next program counter and replies with the address `A_RDIFF`. This address points to the code that sends the differentiating register value and, based on that, the device replies with either `A_HALT` or `A_LOOP` (see Figure 61b).

- *Case* $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \beta' = \texttt{jmpOut!}(\Delta t'; \mathcal{R}) \wedge \Delta t \neq \Delta t'$.
  Since different timings in Sancus$^{\mathrm{L}}$ correspond to different timings in Sancus$^{\mathsf{H}}$ (see Proposition B.24), we program the device so as to either reply with `A_HALT` or with `A_LOOP` depending on the time value (Figure 61c).

- *Case* $\beta = \bullet \wedge \beta' = \texttt{jmpOut!}(\Delta t; \mathcal{R})$.
  In this case $\bullet$ may occur during an interrupt service routine. Two sub-cases may arise, depending on whether the first module $\mathcal{M}_M$ terminates or not when executed in a context that raises no interrupts after the last jump into protected mode. Note that the value of $term$ differentiates the two sub-cases. When $term$ holds, $\mathcal{M}_M$ causes a transition to an exception handling configuration from which there is a a jump to `A_EP`, and the device instructs the code to jump to `A_HALT`. Instead, the second module jumps to another location different from the distinguished address `A_EP`, thus a jump to `A_LOOP` occurs (Figure 61d). When $term$ does not hold, $\mathcal{M}_M$ diverges and the second module makes the CPU jump to a location in unprotected code and the CPU is instructed to jump to `A_HALT` (Figure 61e).

- *Case* $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \beta' = \varepsilon$.
  Analogous to the above, with $term'$ replacing $term$.

- *Otherwise.* No other cases may arise (see Proposition B.23).

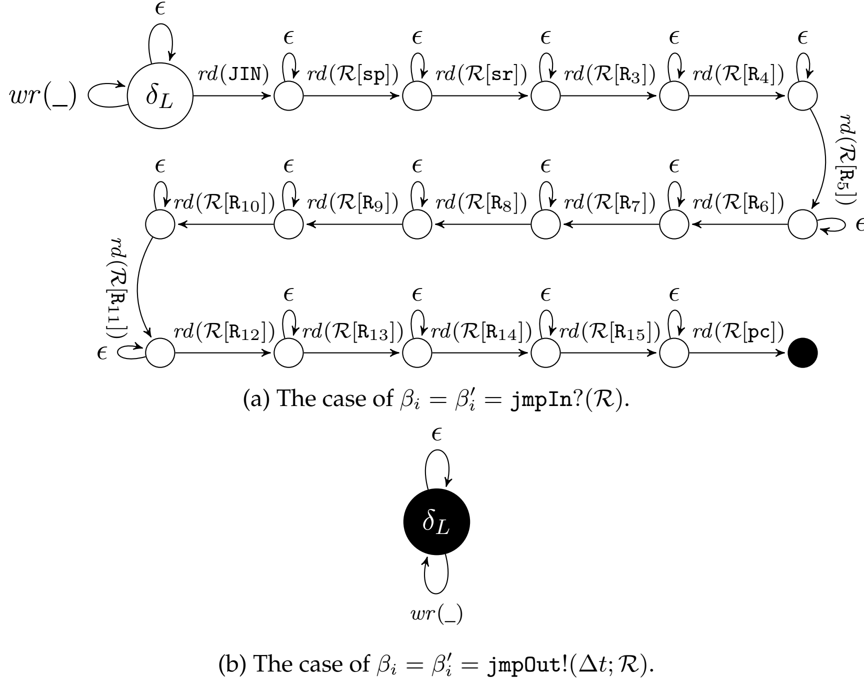
At the end, the algorithm returns a device built as just summarized.

The correctness of the two algorithms is established by the Propositions B.23 and B.24 in Appendix B. The first states that, under the stated conditions, BuildDevice always produces an actual I/O device. The second property guarantees that the context built by joining together the results of the two algorithms is indeed a distinguishing one.

We finally prove that if two modules are contextually equivalent in Sancus$^{\mathsf{H}}$, then they are trace equivalent (implication (ii) in Figure 56).

Lemma 6.7. *If* $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$ *then* $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$.

Proof. We prove the contrapositive: $\mathcal{M}_M \overset{T}{\neq} \mathcal{M}_{M'}$ then $\mathcal{M}_M \not\simeq^{\mathsf{H}} \mathcal{M}_{M'}$. By Proposition 6.3, since $\overset{T}{\neq}$ there exists a pair of distinguishing traces for $\mathcal{M}_M$ and $\mathcal{M}_{M'}$. Algorithm 2 and Algorithm 3 witness the existence of a context C^H that is an actual context and is guaranteed to differentiate $\mathcal{M}_M$ from $\mathcal{M}_{M'}$, i.e., $C^H[\mathcal{M}_M]\Downarrow^{\mathsf{H}}$ and $C^H[\mathcal{M}_{M'}]\not\Downarrow^{\mathsf{H}}$ (or vice versa). Thus, by definition of contextually equivalent modules in Sancus$^{\mathsf{H}}$, we get $\mathcal{M}_M \not\simeq^{\mathsf{H}} \mathcal{M}_{M'}$ as requested. □



(a) The case of $\beta_i = \beta'_i = \texttt{jmpIn?}(\mathcal{R})$.

(b) The case of $\beta_i = \beta'_i = \texttt{jmpOut!}(\Delta t; \mathcal{R})$.

Figure 60: A graphical representation of the algorithm building the I/O device for β_i and β'_i being in the longest common prefix. Here, δ_L denotes the final state of the I/O device being updated, while the final state of the updated device is depicted as a solid, black circle.

6.5.2.4 *Preservation of behaviors*

The last step of this long proof consists in stating and proving the lemma that guarantees preservation of behaviors, i.e., the implication (iii) in Figure 56:

Lemma 6.8 (Preservation)**.**

$$\forall \mathcal{M}_M, \mathcal{M}_{M'}. (\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'} \Rightarrow \mathcal{M}_M \simeq^{\mathbf{L}} \mathcal{M}_{M'}).$$

Proof. Just compose the implications (i) and (ii) of Figure 56 (i.e., Lemmata 6.6 and 6.7, resp.). □

6.6 PRESERVATION OF HYPERPROPERTIES

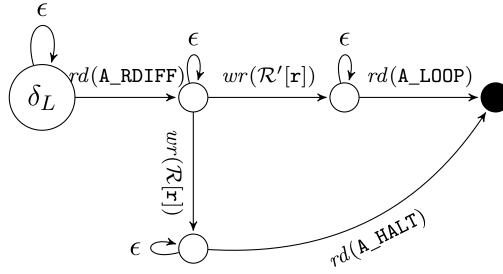
This section shows that our full abstraction result allows us to easily derive the preservation of some notions of *non-interference* and *hypersafety* when passing from $\text{Sancus}^{\mathsf{H}}$ to $\text{Sancus}^{\mathbf{L}}$. Since we are dealing with enclaves, the standard notions will be adapted to our framework.

From now onward, we will use the following equivalence relation — less demanding than $\overset{U}{\approx}$ (Definition 6.8, Page 142) — to express equivalent configurations:

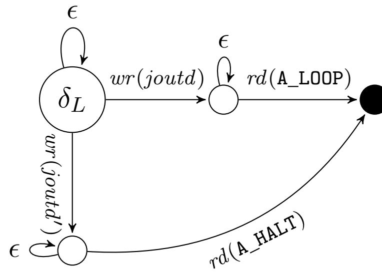
Definition 6.11. *Let c and c' be two configurations. We write $c \overset{L}{=} c'$ iff $(c = c' = \mathrm{HALT}) \vee (c.\delta = c'.\delta \ \wedge\ c.t = c'.t \ \wedge\ c.t_a = c'.t_a \ \wedge\ c.\mathcal{M} \overset{U}{=} c'.\mathcal{M} \ \wedge\ c.\mathcal{R} = c'.\mathcal{R})$, with $\overset{U}{=}$ as in Definition 6.8.*

The following proposition holds trivially:

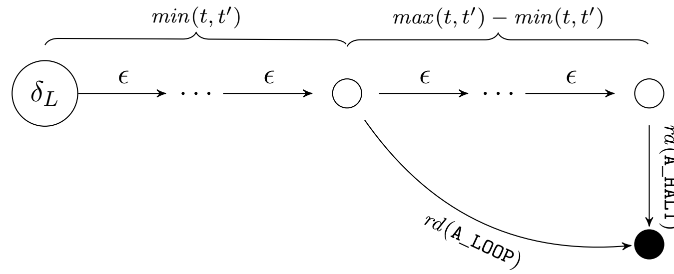
Proposition 6.4. *Let c and c' be configurations such that $c, c' \vdash_{mode} \texttt{UM}$. If $c \overset{U}{\approx} c'$ then $c \overset{L}{=} c'$.*



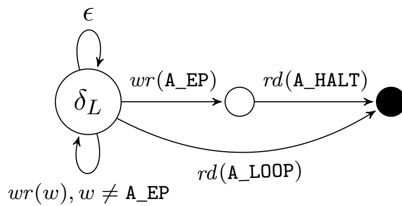
(a) The case of $\beta_i = \mathtt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \beta'_i = \mathtt{jmpOut!}(\Delta t'; \mathcal{R}') \wedge (\exists \mathtt{r} \neq \mathtt{pc}.\, \mathcal{R}[\mathtt{r}] \neq \mathcal{R}'[\mathtt{r}])$.



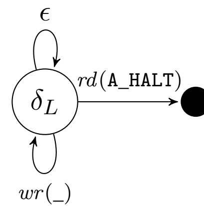
(b) The case of $\beta_i = \mathtt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \beta'_i = \mathtt{jmpOut!}(\Delta t'; \mathcal{R}') \wedge \mathcal{R}[\mathtt{pc}] \neq \mathcal{R}'[\mathtt{pc}]$.



(c) The case of $\beta_i = \mathtt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \beta'_i = \mathtt{jmpOut!}(\Delta t'; \mathcal{R}) \wedge \Delta t \neq \Delta t'$. Let t and t' be the timing differences observed by the attacker at Sancus$^{\mathrm{L}}$, see Algorithm 3 in the Appendix for details.



(d) The case of $\beta_i = \bullet \wedge \beta'_i = \mathtt{jmpOut!}(\Delta t; \mathcal{R}) \wedge\ term$.



(e) The case of $\beta_i = \bullet \wedge \beta'_i = \mathtt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \neg term$.

Figure 61: A graphical representation of the algorithm building the I/O device for β_i and β'_i being the distinguishing observables. Here, δ_L denotes the final state of the I/O device being updated, while the final state of the updated device is depicted as a solid, black circle.

6.6.1 *Take one: termination-insensitive, time-sensitive non-interference*

We now tailor the standard notion of termination-insensitive, time-sensitive non-interference (inspired by [124]) to fit in our framework. Roughly, two modules are non-interferent if and only if no context can distinguish them, because right before termination their public memories are equivalent. Formally:

Definition 6.12. *Two modules $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ are termination-insensitive, time-sensitive non-interferent* (ISNI) *in* Sancus$^{\mathsf{H}}$ *(written $\mathcal{M}_M \approx_{\mathsf{IS}} \mathcal{M}_{M'}$)* iff *for all contexts $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$*

$$\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \to^* c \to \text{HALT} \;\wedge\; \mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \to^* c' \to \text{HALT} \implies c \stackrel{L}{=} c'.$$

Similarly, we define ISNI *in* **Sancus$^{\mathbf{L}}$**, $\mathcal{M}_M \approx_{\mathbf{IS}} \mathcal{M}_{M'}$.

Commonly termination-insensitive non-interference is a property of a single program, to which our definition actually reduces when considering initial configurations as programs whose public input is a context and secret input is a module. Indeed, this is a good model of what happens in reality: contexts are controlled by the attackers, whereas modules are securely deployed (i.e., we model the situation where both code and data are confidentially deployed, as can be done in Sancus 2.0 [170] and in Soteria [108]). Note in passing that our definition and results still hold if the code and a portion of the data are made public before being loaded in the enclave (see also the discussion at the end of Section 6.6.2).

In the following, we clarify the relation between non-interference as defined in Definition 6.12 and our instance of full abstraction established in Theorem 6.2. First, we relate contextual equivalence with non-interference in **Sancus$^{\mathbf{L}}$**:

Lemma 6.9. *If $\mathcal{M}_M \simeq^{\mathbf{L}} \mathcal{M}_{M'}$ then $\mathcal{M}_M \approx_{\mathbf{IS}} \mathcal{M}_{M'}$.*

From that it easily follows that non-interference in **Sancus$^{\mathbf{L}}$** is guaranteed when two modules are contextually equivalent in Sancus$^{\mathsf{H}}$:

Theorem 6.3. *If $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$ then $\mathcal{M}_M \approx_{\mathbf{IS}} \mathcal{M}_{M'}$.*

6.6.2 *Take two: termination- and time-sensitive non-interference*

In this section we consider a notion of non-interference inspired by Devriese and Piessens [91] and distinguishes terminating modules from non-terminating ones. In the standard notion the program is public and the memory is split in a public and a secret segment: an attacker cannot discover any secret data by running the code with different public data. In our framework however also the code as well as (public) data are protected, being hosted in the enclave. Therefore, we first adapt the standard definition to our case, where the entire module is protected. At the end of this section, we then discuss how to recover the classic notion.

Definition 6.13 (SSNI). *Two modules $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ are termination- and time-sensitive non-interferent* (SSNI) *in* Sancus$^{\mathsf{H}}$ *(written $\mathcal{M}_M \approx_{\mathsf{SS}} \mathcal{M}_{M'}$)* iff *for all contexts $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$, and configurations c both implications hold:*

- $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \to^* c \to \text{HALT} \implies \exists c'. (\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \to^* c' \to \text{HALT} \;\wedge\; c \stackrel{L}{=} c')$
- $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \to^* c \to \text{HALT} \implies \exists c'. (\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \to^* c' \to \text{HALT} \;\wedge\; c \stackrel{L}{=} c')$

Similarly, we define SSNI *in* Sancus$^{\mathrm{L}}$, $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$.

The following theorem is easily established:

Theorem 6.4.

1. *If* $\mathcal{M}_M \simeq^{\mathrm{L}} \mathcal{M}_{M'}$, *then* $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$; *and*
2. *if* $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$, *then* $\mathcal{M}_M \simeq^{\mathrm{H}} \mathcal{M}_{M'}$.

Thus, due to Theorem 6.2, the preservation of *SSNI* holds:

Corollary 6.1. $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'} \implies \mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$.

To recover the standard notion of non-interference, we first formalize when two modules share the same code, but may differ in their data sections. Note that the code needs not to be kept confidential, even though it is part of the enclave. Recall the notion of layout $\mathscr{L} = \langle ts, te, ds, de, isr \rangle$, which is fixed in our model.

Definition 6.14. *We say that two modules* $\mathcal{M}_M$ *and* $\mathcal{M}_{M'}$ share the code *(written* $\mathcal{M}_M =_{sc} \mathcal{M}_{M'}$*)* iff $\forall i \in [ts, te).\ \mathcal{M}_M(i) = \mathcal{M}_{M'}(i)$.

Suppose as given two enclaves hosting two modules $\mathcal{M}_M$ and $M_{M'}$ such that (i) they share a piece of code considered public in their code section, i.e., $\mathcal{M}_M =_{sc} \mathcal{M}_{M'}$; and (ii) they have possibly different values in the data section, which are confidential. Under these assumptions, Definition 6.13 reduces to the standard notion of termination- and time-sensitive non-interference. Also, since $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'} \wedge \mathcal{M}_M =_{sc} \mathcal{M}_{M'} \implies \mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$ is an immediate consequence of Corollary 6.1, we easily guarantee the preservation of the standard notion of non-interference in our enclave setting.

Corollary 6.2. *For all* $\mathcal{M}_M, \mathcal{M}_{M'}$ *such that* $\mathcal{M}_M =_{sc} \mathcal{M}_{M'}$ *it is*

$$\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'} \implies \mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}.$$

Note that Theorem 6.4 and its corollaries hold under the hypothesis that Sancus$^{\mathrm{H}}$ and Sancus$^{\mathrm{L}}$ are fully abstract. Indeed, under the same hypothesis one could also prove the vice versa of the cases of Theorem 6.4 and get that *SSNI* is equivalent to our notion of contextual equivalence, especially for enclaves with the same code section.

6.6.3 *Take three: stepwise termination- and time-sensitive non-interference*

Since the attacker in our model is able to interrupt execution at *every* CPU cycle, one might wonder about the preservation of a stronger, *stepwise* notion of non-interference.

For that, we start from *SSNI* and introduce *stepwise termination-and time-sensitive non-interference*. It stipulates that two modules are non-interferent whenever their public memories are kept equivalent while stepping between successive unprotected configurations. For that, we first need the following definition:

Definition 6.15. $\mathcal{D} \vdash c \twoheadrightarrow_k c'$ iff

$$\mathcal{D} \vdash c \rightarrow c_1 \rightarrow \ldots \rightarrow c_n \rightarrow c' \ \wedge\ c, c' \vdash_{mode} \texttt{UM} \wedge$$
$$\forall 1 \le i \le n.\ c_i \vdash_{mode} \texttt{PM} \ \wedge k = \begin{cases} 0 & n = 0 \\ 2 & o.w. \end{cases}$$

Also, let $\mathcal{D} \vdash c_1 \twoheadrightarrow^t_K c_t$*, where* $K = \sum_{i=1}^t k_i$*, be the shorthand for* $\mathcal{D} \vdash c_1 \twoheadrightarrow_{k_1} \ldots \twoheadrightarrow_{k_t} c_t$.
Similarly, we define $\mathcal{D} \vdash c \twoheadrightarrow_k c'$ *and* $\mathcal{D} \vdash c \twoheadrightarrow^t_K c'$.

Intuitively k counts the interactions between the context and the module ($k = 0$ if there are none and $k = 2$ if there is one entry and one exit) whereas the arrows $\twoheadrightarrow_k$ and $\twoheadrightarrow_k$ ignore all the steps taken in protected mode and just take into account the actions of the context.

We can now define the new notion of stepwise termination- and time-sensitive non-interference:

Definition 6.16. *Two modules $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ are stepwise termination- and time-sensitive non-interferent* (SSSNI) *in* Sancus$^{\mathsf{H}}$ *(written $\mathcal{M}_M \approx_{\mathsf{SSS}} \mathcal{M}_{M'}$)* iff *for all contexts $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$ both implications hold*

- $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \twoheadrightarrow^t_K c \implies \exists c'. \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \twoheadrightarrow^t_K c' \ \wedge \ c \overset{L}{=} c'$
- $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \twoheadrightarrow^t_K c' \implies \exists c. \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \twoheadrightarrow^t_K c \ \wedge \ c \overset{L}{=} c'$

Similarly, we define SSSNI *in* **Sancus$^{\mathsf{L}}$**, $\mathcal{M}_M \approx_{\mathbf{SSS}} \mathcal{M}_{M'}$.

Since the arrows $\twoheadrightarrow^t_K$ and $\twoheadrightarrow^t_K$ are in a clear relation with $\overset{\overline{\beta}}{\Longrightarrow}$ (see Proposition B.25), we can prove the following results, leading to the preservation of *SSSNI*:

Lemma 6.10.

1. *If $\mathcal{M}_M \simeq^{\mathbf{L}} \mathcal{M}_{M'}$ then $\mathcal{M}_M \approx_{\mathbf{SSS}} \mathcal{M}_{M'}$; and*
2. *if $\mathcal{M}_M \approx_{\mathsf{SSS}} \mathcal{M}_{M'}$ then $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$.*

Thus, due to Theorem 6.2, the preservation of *SSSNI* holds:

Corollary 6.3. *If $\mathcal{M}_M \approx_{\mathsf{SSS}} \mathcal{M}_{M'}$ then $\mathcal{M}_M \approx_{\mathbf{SSS}} \mathcal{M}_{M'}$.*

We note in passing that the same considerations made at the end of Section 6.6.2 suffice to show that our contextual-equivalence coincides with this notion of non-interference when the code and some data are deemed public.

6.6.4 *Take four: hypersafety*

In this section we briefly sketch how to reduce our notion of full abstraction to the preservation of a much wider family of security properties than just non-interference, building on the work of Patrignani and Garg [179].

We first recall some notation. A compiler is seen in [179] as a mapping $[\![\cdot]\!]$ from source to target programs. Our compiler is actually the identity function, since any module $\mathcal{M}_M$ in Sancus$^{\mathsf{H}}$ is mapped into the *same* module $\mathcal{M}_M$ in **Sancus$^{\mathsf{L}}$**. Also, Patrignani and Garg [179] give the following notion of trace equivalence, which we call *whole program trace equivalence*:

Definition 6.17 (Definition 19 [179]). *We say that*

$$\mathcal{M}_M \overset{WT}{=} \mathcal{M}_{M'} \iff \forall C. WTr(C[\mathcal{M}_M]) = WTr(C[\mathcal{M}_{M'}]),$$

where $WTr(C[\mathcal{M}_M]) \triangleq \{\overline{\beta} \mid \exists c. \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c\}$.

Contrary to our notion of Definition 6.7, their definition of trace equivalence requires the programs to produce the same set of traces under a *fixed context* (i.e., it is defined on whole programs).

Crucially, the following theorem links the notion of whole program trace equivalence with ours:

Theorem 6.5. *The following relations are equivalent:*

$$1.\ \mathcal{M}_M \overset{WT}{=} \mathcal{M}_{M'} \qquad 2.\ \mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$$
$$3.\ \mathcal{M}_M \simeq^{\mathbf{L}} \mathcal{M}_{M'} \qquad 4.\ \mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$$

Thus, as a consequence of Theorem 6.5, our coarse-grained traces (Figure 58) are a fully abstract trace semantics for both $\mathsf{Sancus}^{\mathsf{H}}$ and $\mathbf{Sancus}^{\mathbf{L}}$, according to Definition 20 of [179]:

Corollary 6.4.

1. $\forall \mathcal{M}_M, \mathcal{M}_{M'}.\ \mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'} \iff \mathcal{M}_M \overset{WT}{=} \mathcal{M}_{M'}$;
2. $\forall \mathcal{M}_M, \mathcal{M}_{M'}.\ \mathcal{M}_M \simeq^{\mathbf{L}} \mathcal{M}_{M'} \iff \mathcal{M}_M \overset{WT}{=} \mathcal{M}_{M'}$.

Due to this corollary, both **Assumption 1** (i.e., trace semantics of $\mathsf{Sancus}^{\mathsf{H}}$ is fully abstract) and **Assumption 2** (i.e., trace semantics of $\mathbf{Sancus}^{\mathbf{L}}$ is fully abstract) from [179] hold. Recall that the compiler from $\mathsf{Sancus}^{\mathsf{H}}$ to $\mathbf{Sancus}^{\mathbf{L}}$ implicitly used throughout the paper *is* an identity compiler (mapping a module in itself). By the assumption above, it also follows that $\mathsf{Sancus}^{\mathsf{H}}$ and $\mathbf{Sancus}^{\mathbf{L}}$ share the same set of fully abstract traces. Our identity compiler is therefore trivially *correct*, and also is a *fail-safe-behavior compiler* — roughly, a compiler producing target programs that always halt after an invalid input (Definition 16 of [179]). We finally conclude that all the safety hyperproperties that hold for whole programs in $\mathsf{Sancus}^{\mathsf{H}}$ also hold in $\mathbf{Sancus}^{\mathbf{L}}$ (Theorems 10 and 6 of [179]).

Note that the above is just a sketch of how one could prove the preservation of hypersafety following the approach of [179]. Indeed, a more formal and complete treatment of hyperproperty preservation would require traces to be the *ground truth* concerning what an attacker might observe. This might call for a complete reworking of the notion of traces in our setting, that are now a mere tool, beneficial to the proof, only. The same considerations hold for other principles of secure compilation based on (robust) hyperproperty preservation, such as those in [9, 10]; see also Section 6.7.1.

6.7 DISCUSSION

6.7.1 *Full abstraction as a security objective*

The security guarantee that our approach offers is quite strong: an attack is possible in $\mathsf{Sancus}^{\mathsf{H}}$ if and only if it is possible in $\mathbf{Sancus}^{\mathbf{L}}$. Recall from Section 6.2 that isolation is defined in terms of contextual equivalence: full abstraction fits nicely in our setting, since it ensures preservation and reflection of contextual equivalence.

The preservation part (i.e., contextual equivalence in $\mathsf{Sancus}^{\mathsf{H}}$ implies it in $\mathbf{Sancus}^{\mathbf{L}}$) guarantees that extending $\mathsf{Sancus}^{\mathsf{H}}$ with interrupts opens no new attack avenues. Vice versa, reflection (i.e., contextual equivalence in $\mathbf{Sancus}^{\mathbf{L}}$ implies it in $\mathsf{Sancus}^{\mathsf{H}}$) is needed because otherwise two enclaves that are distinguishable in $\mathsf{Sancus}^{\mathsf{H}}$ *become* indistinguishable in $\mathbf{Sancus}^{\mathbf{L}}$, making the extension not fully "backward compatible". Although this is mainly a functionality concern, adding interruptible enclaves without guaranteeing reflection requires the enclave programmer to be aware of low-level implementation details of the target (micro-)architecture. However, reflection rules out mechanisms that while closing the interrupt side-channels also close other channels. We believe the situation is very similar for other extensions: adding caches, pipelining, etc. should not strengthen existing isolation mechanisms either.

Actually, full abstraction enables us to take the security guarantees of $\mathsf{Sancus}^{\mathsf{H}}$ as the specification of the isolation required after an extension is added.

An alternative approach to full abstraction would be to require (a non-interactive version of) robust preservation of timing-sensitive non-interference [10]. This can also guarantee resistance against the example attacks in Section 6.2. However, this approach offers a strictly weaker guarantee: our full abstraction result implies that timing-sensitive non-interference properties of $\mathsf{Sancus}^{\mathsf{H}}$ programs are preserved in $\mathbf{Sancus}^{\mathbf{L}}$, provided that non-interference takes as secret the data of the enclave, i.e., its memory, code, and initial state (see also the discussion in Section 6.6 about the role of full abstraction in preservation of non-interference).

In addition, full abstraction implies that isolation properties that rely on code confidentiality are preserved, and this matters for enclave systems that guarantee code confidentiality, like the Soteria system [108]. An advantage however is that robust preservation of timing-sensitive non-interference should be easier to prove.

In case full abstraction is considered too strong as a security criterion, it is possible to selectively weaken it by modifying $\mathsf{Sancus}^{\mathsf{H}}$. For instance, to specify that code confidentiality is not important, one can modify $\mathsf{Sancus}^{\mathsf{H}}$ to allow contexts to read the code of an enclave.

6.7.2 *The impact of our simplifications*

The model we discussed in this chapter makes several simplifying assumptions w.r.t. the actual Sancus architecture. We believe that some of them are non-essential and could be removed with additional work, but without providing important new insights. For instance, supporting more MSP430 instructions would not affect the strong security guarantees offered by our approach, and only requires straightforward, yet tedious technical work. Also, the limitation that forbids enclaves to access unprotected memory can be lifted by making reads and writes from/to unprotected memory observable to the untrusted context in $\mathsf{Sancus}^{\mathsf{H}}$. This means that these reads and writes become interactions with the adversary and that the burden of considering attacks scenarios involving them is moved to the enclave programmer. An alternative approach which puts less security responsibility on the developer is to rely on a trusted runtime that can access unprotected memory to copy in/out parameters and results, and then turn off access to unprotected memory before calling enclaved code. Our model could easily be extended to deal with such a trusted runtime by considering memory copied in/out as a large CPU register. However, it is important to emphasize that the implementation of such trusted enclave runtime environments has been shown to be error-prone [47]. A further alternative is considering the secure compartmentalizing compilation proposed by Juglaret et al. [121], who also use full abstraction to prove security.

Another non-essential limitation is the fact that we do not support nested interrupts nor interrupt priority. It is straightforward to extend our model with the possibility of multiple pending interrupts and a policy to select which of these pending interrupts to handle. One only has to take care that the interrupt arrival time used to compute padding is the arrival time of the interrupt that will be handled first.

Other assumptions are instead more essential, and removing these would require additional research. Here, we discuss their impact on the applicability of our results to real systems.

First, our model made some simplifying assumptions about the enclave-based isolation mechanism. We do not support cryptographic operations and attestation. Despite not being fundamental to deal with interrupt-based attacks, they should be added to have a full model of Sancus. Indeed, not supporting these features means that we assume that loading and initializing an enclave can be done as securely in **Sancus$^{\mathbf{L}}$** as it can be done in Sancus$^{\mathsf{H}}$. Our choice separates concerns and it is independent of the security criterion adopted. Modelling both memory access control and cryptography would only increase the complexity of the model, as two security mechanisms rather than one would be in order. Also their interactions should be considered to prevent, among others, leaks of cryptographic keys unveiling secrets protected by memory access control, and vice versa.

Furthermore, we assumed the simple setting where only a single enclave is supported. We believe these simplifications are acceptable, as they reduce the complexity of the model significantly, and as none of the known interrupt-driven attacks rely on these features. Another limitation of our model is that it forbids jumps into the enclave from the interrupt service routine (via $\vdash_{mac}$). Allowing re-entrancy would cause the same complications as allowing multi-threaded enclaves, and these are substantial complications that also lead to new kinds of attacks [235]. We leave investigation of these issues to future work.

Finally, we scoped our work to only consider "small" microprocessors. Both our formal models, as well as the design and security proof of our interrupt padding countermeasure focus very much on enclaved execution on small microprocessors, like the Sancus system. An interesting question is to what extent the insights of our countermeasure design can be applied to more complex platforms like Intel SGX.

The first difference we do not capture in our model is that the execution time of instructions in high-end architectures appears to attackers as non-deterministic. This is mainly a disadvantage for the attacker, that needs to take the average of timings of multiple runs to reliably compute execution times and draw their conclusion. However, this has been shown to be feasible for interrupt-based attacks (even on high-end x86 processors) by Van Bulck et al. [225]. Actually, we think that some form of padding might still be useful even in these cases. On the one hand, the non-deterministic nature of instruction execution time makes it hard to choose a good value for `MAX_TIME` and since the *worst-case* execution time on complex processors can be quite high, choosing `MAX_TIME` to be higher than any possible instruction may be prohibitively expensive. On the other hand, it might be fine to choose `MAX_TIME` to be smaller than the actual worst-case longest instruction execution time. In this case, `MAX_TIME` would be a trade-off between performance and security: the higher the `MAX_TIME`, the less an attacker can learn from a specific interrupt latency measurement. However, in cases where the attacker can influence the execution time of instructions (for instance, by flushing caches), the precise security gains are hard to estimate. One could also think of adding a *random* padding to interrupt handling and resume, together with measures to make it impossible for the attacker to execute the same measurement many times (thus making it impossible to do statistical analysis).

The second difference is that optimizations (e.g., caches or speculative execution) open new side-channels which are observable to the attacker. It is hard to estimate the impact of padding mechanisms on these side-channels. For instance, if the attacker can distinguish padding from regular instructions by observing side effects (e.g., monitoring for loads from memory, which do not happen for padding), the countermeasure becomes useless. Since removing all the possible side effects through which enclaved executions leak information may be infeasible (e.g., for performance reasons) the question becomes instead how much leakage we are willing to allow. This could again be formulated as a full abstraction theorem, where we model the side-channels that we accept in the high model (similarly to

what we did for end-to-end timing attacks in Sancus^{H}). We believe that the models in this paper provide a good basis for such further developments, but we leave the (challenging) elaboration of these ideas to future work. For an in-depth discussion on the applicability of our countermeasure to high-end processors, we refer to Van Bulck's PhD thesis [224] and Busi et al. [59].

In summary, to provide hard mathematical security guarantees, one often abstracts from some details and provable security only provides assurance to the extent that the assumptions made are valid and the simplifications non-essential. The discussion above shows that this is the case for a relevant class of attacks and systems, and hence that our countermeasure for these attacks is well-designed. Of course, attacks remain possible for more complex systems (e.g., including caches and speculation), or considering more powerful attackers (e.g., with physical access to the system).

6.8 CONCLUSIONS

We have proposed an approach to formally assure that extending a microprocessor with a new feature does not weaken the isolation mechanisms that the processor offers. More precisely, we advocate full abstraction as a formal criterion of what it means to maintain the security of isolation mechanisms under processor extensions. We have applied our approach to an embedded microprocessor: first, we have designed an extension of Sancus with interruptible enclaves ($\textbf{Sancus}^{\textbf{L}}$) and then we have proved it fully abstract with respect to the original Sancus without them (Sancus^{H}). Remarkably, the full abstraction proof relies on the strong power of our attacker that controls the unprotected memory, which is limited to 64 KB, and the I/O device which instead has unlimited memory. Indeed, the backtranslation encodes the attack logic within the I/O device that then drives a fixed piece of code in unprotected memory, namely the software component of the attacker.

To further assess our full abstraction-based security criterion we have compared its guarantees with those of some notions of non-interference preservation presented in the literature: we have proved that they are implied by our full abstraction theorem. We also proved that our results preserve hyperproperties, thus ensuring that modules executed in interruptible Sancus enjoy the same hyperproperties as they would when executed by the uninterruptible one.

The mitigation here proposed was also implemented by our co-authors from KU Leuven, so demonstrating its feasibility. An in-depth description of the implementation is available in [58, 59] and its source code is available online at `https://github.com/sancus-pma/sancus-core/tree/nemesis`.

RELATED WORK Our work is motivated by the recent wave of software-based side-channel attacks and controlled-channel attacks that rely on architectural or micro-architectural processor features. Recent surveys of the area include Ge et al. [102] for timing attacks, Gruss's PhD thesis [110] for software-based micro-architectural attacks before Spectre/Meltdown, and Canella et al. [64] for transient execution based attacks.

Despite their strong security measures, enclaves are vulnerable to a variety of attacks. For instance, a successful attack technique that allows to extract secrets from enclaves (see e.g., [107, 113, 153, 209]) is PRIME+PROBE [117]. In its simplest form, PRIME+PROBE allows attackers to discover memory accesses of their victims as follows. First, the attacker fills the cache with its own data (*primed data*); then it waits for the victim to be executed; finally, it checks (*probes*) whether the cache lines of the primed data are still in the cache: if this

is not the case it means that the victim accessed the relevant memory addresses. Also, controlled side-channel attacks [240] proved to be quite effective (e.g., [48, 114, 154, 225]). Among these, we find interrupt-based attacks that are the most relevant for this chapter. Van Bulck et al. [225] were the first to show how that measuring interrupt latency can lead to powerful attacks against both high-end enclaved execution systems (like Intel SGX) and against low-end systems like Sancus. Independently, He et al. [114] designed a similar attack just for Intel SGX. Intel SGX is also vulnerable to *transient-execution attacks*, i.e., attacks that exploit the side effects caused by speculative and out-of-order execution. One of the first attacks exploiting transient-execution is Foreshadow [46, 236], which allows an attacker to extract secrets from an enclave residing in the same core as the attacker. Other, more recent vulnerabilities leveraging different sources of side effects include ZombieLoad [208], RIDL [207], and CrossTalk [196]. The above list is by no means exhaustive and for a more in-depth survey on Intel SGX attacks we refer the interested reader to Van Bulck's PhD thesis [224].

There is an extensive body of work also on defenses against software-based side-channel attacks. The four surveys mentioned above ([64, 102, 110, 224]) also survey defenses, including both software-based defenses like the constant-time programming model and hardware-based defenses such as cache-partitioning. To the best of our knowledge, our work proposes the first defense specifically designed and proved to protect against pure interrupt-based side-channel attacks. Clercq et al. [73] have proposed a design for secure interruptibility of enclaved execution, but they have not considered side-channels — their main concern is to make sure that there are no direct leaks (e.g., of registers) on interrupts. Most closely related to ours is the work on SecVerilog [246] that also aims for formal assurances. To guarantee timing-sensitive non-interference properties, SecVerilog uses a security-typed hardware description language. However, this approach has not yet been applied to the issue of interrupt-based attacks. Similarly, Zagieboylo et al. [245] describe an ISA with information-flow labels and use it to guarantee timing-insensitive information flow at the architectural level.

An alternative approach to interruptible secure remote computation is pursued by VRASED [171]. In contrast to enclaved execution, their design only relies on memory access control for the attestation key, not for the software modules being attested. They prove that a carefully designed hardware/software co-design can securely do remote attestation.

Our security criterion is directly influenced by a long line of work that considers *full abstraction* as a criterion for secure compilation as already briefly surveyed in Chapter 2. The idea was first coined by Abadi [2], and has been applied in many settings, including compilation to JavaScript [100], various intermediate compiler passes [14, 15], and compilation to platforms that support enclaved execution [13, 176, 178]. None of the above works consider timing-sensitivity nor interrupts and they study compilations higher up the software stack than what we consider in this chapter. Still higher up in the computational stack, Tomé Cortiñas et al. [222] extended the MAC library [228] — a Haskell *information-flow control* library — with asynchronous exceptions. Akin to interrupts in our setting, asynchronous exceptions can be raised at any time and may break security properties of the running code. To ensure that this never happens, they introduced a variant of non-interference and proved that it is satisfied by their extension of the MAC library.

Other authors applied secure compilation techniques to prove security against side-channel attacks. For instance, Barthe et al. [20] proved that a suitably modified version of the CompCert compiler [139] preserves the constant-time policy. For that, they identified the passes of CompCert that did not preserve constant-time and modified them accordingly; afterwards, they proved them to be constant-time preserving using variants of the proof techniques proposed in [21] (and briefly surveyed in Chapter 3). Very recently, Patrignani and Guarnieri [182] proved secure a couple of mitigations against Spectre v1 [128] by specializing hyperproperty preservation principles of [10] to preserve *speculative non-interference* [112].

Another active area of research is that of detecting (and possibly fixing) speculative leaks (e.g., leaks caused by Spectre attacks [128]) using programming-languages techniques. For example, SPECTECTOR [112] is a symbolic execution tool that analyses `x86_64` assembly programs and detects the presence of possible speculative leaks or proves their absence. Guanciale et al. [111] present a formal model capturing out-of-order execution and speculation in single core processors. Using this model they discover three new (possible) vulnerabilities and assess the security of existing countermeasures. Vassena et al. [227] define a static type system that labels each expression of their language as either *transient* or *stable* (i.e., that may include transient values or not, respectively). Crucially, their type system rejects programs that possibly contain speculative leaks. Also, they introduce the **protect** primitive that ensures that assignments containing it are performed only once their right-hand-side is stable. Furthermore, Vassena et al. provide an algorithm that automatically synthesizes the minimal number of **protect**s to be inserted in the given program to fix all the potential speculative leaks.

7

CONCLUSION

In this thesis, we have argued that secure compilation is *not just* a concern of compilers and their developers. Indeed, language translations and code transformations are common across all the levels of the computational stack, from the high level down to the micro-architecture. Thus, we have studied different techniques guaranteeing a variety of security objectives and that are applicable at different levels of abstraction.

At the highest level, we have considered transformations whose target language is the same as the source (e.g., program optimizations or obfuscations) and we have verified that they preserve the security properties of source programs. In Chapter 3 we have followed a classical approach and manually proved that the control-flow flattening program obfuscation preserves the constant-time policy. However, this is not satisfactory: this approach requires the user to prove her results manually [20, 21]. Our point is that we need secure compilation to be achievable with the least human effort as possible to make it practical. Thus, in Chapter 4 we have introduced an automatic approach that makes secure compilation easier to achieve in the particular case of type-preserving compilation. Indeed, our approach allows to efficiently check whether a given program transformation preserves the type of a particular transformed program. Given a (security) type system that statically checks whether the property of interest holds, we have provided a framework to make its usage incremental. Since the incrementalized algorithm just analyses the *diffs* between the transformed and the original program, the incrementalization allows to re-analyze the code after *each* optimization step without excessively slowing down the compiler.

Despite being efficient and automatic, the incremental approach lacks the possibility to deal with cases in which the source language differs from the target, as typical at lower levels of the computational stack. In Chapter 5 we have introduced *secure translation validation* and lifted the requirement about the source and the target languages being equal. Secure translation validation extends translation validation [189] to deal with security properties. More precisely, it enables to automatically certify that a given compiled program can be safely executed in a given environment (i.e., target context). Actually, we verify that the compiled program plugged into the target context still enjoys the same safety properties as its source counterpart. If not, our technique also provides (in some cases) the source context that performs the attack invalidating the same safety that is broken at the target level. We also instantiated our framework to a simple use case, whose languages are inspired from Protzenko et al. [193]. Despite encouraging results, a more realistic use case and its actual implementation could be useful to further assess our approach. We did a first step in this direction in [56], where we apply STV to a pair of Turing-complete programming languages so as to ensure the preservation of the isolation properties provided by the source language into the target.

Finally, in Chapter 6 we have moved to the very bottom of the computational stack. At this level, we have studied how to *securely* extend a given architecture with new features while keeping *backward compatibility*. In this setting, we have considered attackers that can exploit controlled side-channels to break the isolation mechanisms provided by architectures. By observing that *contextual equivalence* is a good model for the isolation properties provided by those isolation mechanisms, we have concluded that to ensure security and keep backward compatibility it is necessary to *preserve* and *reflect* contextual equivalence, i.e., we need to prove a *full abstraction* theorem. Concretely, we have first formally modeled the Sancus architecture [168, 170]; then, we have extended it with carefully-designed interruptible enclaves; finally, we have proved a full abstraction theorem between the two. These three steps ensure that the extended version of our model is resistant to interrupt-based attacks (e.g., Nemesis [225]), while keeping backward compatibility. Also, we have shown that our result actually implies (variants of) the more classical notions of non-interference preservation.

7.1 FUTURE WORK

Here we briefly outline our in-progress and future work. As said in the introduction, the goal of this thesis was to explore how the ideas of secure compilation can be applied at different levels of the computational stack, with an eye to its automatization. However, still a lot of interesting questions remain open.

FOUNDATIONS OF SECURE COMPILATION From our point of view, a crucial open question about the foundations of secure compilation concerns the relation between the different secure compilation approaches. In particular, we are interested in comparing *fully-abstract compilation* [2] and *trace-based compiler security principles* [9, 10]. A complete answer to the above question would allow to re-use the results from one approach into the other, and would help developers of secure compilers to choose the right principles to use. In collaboration with Carmine Abate [11], we have done some preliminary steps towards bridging the two views and our initial results are quite promising. Recall that a compiler is fully abstract if and only if it *preserves and reflects* contextual equivalence (that in [11] corresponds to trace equivalence) and that a hyperproperty is a set of sets of traces (see Chapter 2 and [72]). Consider now the definition of *robust hyperproperty preservation* from [9]:

Definition 7.1 (Theorem 1 of [11]).

$$(\forall \mathsf{C}.\ \mathsf{C}[\mathsf{P}] \vDash H) \Rightarrow (\forall \mathbf{C}.\ \mathbf{C}[\mathbf{P}] \vDash \tilde{\tau}(H))$$

where $\tilde{\tau}$ is a map from source hyperproperties to target ones.

As a consequence of the results by Parrow [174], it is possible to show that a fully abstract compiler also ensures robust hyperproperty preservation for some $\tilde{\tau}$:

Theorem 7.1. *If $[\![\cdot]\!]$ is a fully abstract compiler, then there exists a $\tilde{\tau}$ such that $[\![\cdot]\!]$ preserves the robust satisfaction of hyperproperties. Moreover, $\tilde{\tau}$ is the smallest (pointwise) with this property*

and its corollary (assuming $\tilde{\tau}$ from Theorem 7.1):

Corollary 7.1 (Corollary 1 of [11]). *If $[\![\cdot]\!]$ is a fully abstract compiler and H is a hyperproperty, then $[\![\cdot]\!]$ preserves the robust satisfaction of H iff $\tilde{\tau}(H) \subseteq H$.*

Despite these initial and encouraging results, many questions remain unanswered. The first problem is that the proof of Theorem 7.1 defines $\tilde{\tau}$ using its knowledge about the behavior of the compiler, thus making the compiler itself part of the trusted computing base. We are convinced that one could consider an *over-approximation* of $\tilde{\tau}$ (i.e., a larger map) to remove the compiler from the trusted computing base, however further investigation is needed to completely solve the issue. Another problem is that the computation of $\tilde{\tau}$ can be quite prohibitive: in [11] we propose to compute it by successive approximations, however it is still unclear how to apply this methodology in more general cases. Another open question is under which conditions (a relaxed version of) the vice versa of Theorem 7.1 holds. Indeed, turning such a theorem into a characterization would allow for a better unifying view of the two secure compilation approaches.

SECURITY OF PROGRAM OBFUSCATIONS In Chapter 3 we have proved that control-flow flattening preserves the constant-time policy in a very simple setting. In the future, we would like to extend this study to more complex scenarios. First, it would be interesting to mechanize our formalization and to investigate how to integrate it with recent evolutions of the CompCert compiler that already preserve the constant-time policy [20]. Also, we would like to consider the security of other obfuscations techniques (e.g., those in [76]) and different security properties, e.g., general safety properties or hypersafety. Also, in Chapter 3 we have just considered a passive attacker that can only observe the leakage. An interesting problem would then be to explore if our result and the current proof technique scale to a setting with active attackers that also interfere with the execution of programs, e.g., those from [9, 10].

AUTOMATIZATION OF SECURE COMPILATION As for incremental typing, we wish to address a number of questions that remain open.

First, we are planning to extend our approach to combine our proposal with that of [109], so as to also support the correct definition of type systems. Another issue to investigate comes from the observation that other analyses could be made incremental. Supporting type and effect systems should be rather straightforward, instead we expect that more work is needed to apply our ideas to other syntax-directed static analyses, e.g., control flow analysis because it requires fixed-point computations. Also, some type systems do not fit easily in the format we have proposed in Definition 4.2. A particularly relevant example are the bidirectional type systems, e.g., that for dependent types in [143]. A possible generalization of the rule format used here to include this kind of rules is the following

$$\frac{\bigwedge_{k \in K_t} \left(\bigwedge_{i \in \mathbb{I}_k} \boxed{tr_t^{k,t_i}(\Gamma, \{R_{k,j}\}_{j<i})} \vdash_{\mathscr{A}} t_i : R_{k,i} \wedge \boxed{checkJoin_t^k(\Gamma, \{R_{k,i}\}_{i \in \mathbb{I}_k}, \mathtt{out}\ R_k)} \right)}{\Gamma \vdash_{\mathscr{A}} t : \boxed{join_t(\{R_k\}_{k \in K_t})}}$$

where $K_t \subseteq \mathbb{N}$ and $\forall k \in K_t.\ \mathbb{I}_k \subseteq \mathbb{I}^t$, and the function $\boxed{join_t}$ combines the partial results R_k to produce the overall result. The intuition is that with this new format one can express rules that involve multiple and different instances of *checkJoin*. Finally, we would like to investigate how to deal with typing errors, in particular we are interested in devising which is the most appropriate policy for updating the cache when type errors arise.

The other automatic proposal of this thesis is *secure translation validation*, but it is still preliminary work. For instance, we would like to further investigate the applicability of our ideas to robust hypersafety preservation [10], which we feel to be at easy reach. Also, our current case study is just a toy. First, one should extend the languages to support

more realistic features (such as memory management or recursion) while, at the same time, increasing the precision of the source and target analyses to lower the number of false positives. Moreover, complete mechanization and implementation still lack: to foster the adoption of secure translation validation one should at least have a working prototype that allows one to check the security of life-like programs. Finally, we would like to explore the applicability of this technique to languages lower in the computations stack. For example, it would be interesting to check the security of assembly programs executed on a processor equipped with ISA-level security features such as enclaves or capabilities.

SECURE COMPILATION AGAINST MICRO-ARCHITECTURAL ATTACKS As seen in Chapter 6 for Nemesis [225] and in other recent work [181] for Spectre [128], secure compilation may be helpful in mitigating some side-channel and micro-architectural attacks. While our approach makes it possible to rigorously reason about timing-based side-channels, scaling it to larger processors is not trivial at all. To handle larger processors, we need models that can abstract away many details of the processor implementation, yet keeping enough details to model micro-architectural attacks of interest. A promising example of a model with such features and that could replace our cycle-accurate one was proposed by Disselkoen et al. [93]. As a matter of fact, our model and our full abstraction result seem to be a good starting point, although they currently apply only to “small” microprocessors for which we can define a cycle-accurate operational semantics.

Furthermore, in our proposal the security criterion is binary: an extension is either secure, or it is not. Therefore, *low bandwidth* side-channels are not kept apart from *high-bandwidth* side-channels. An important challenge for future work is to introduce some kind of *measure* on the weakening of security, so as to allow security policies that consider some bounded amount of leakage acceptable.

A

INCREMENTAL TYPING

A.1 ADDITIONAL PROOFS FOR SECTION 4.3

Theorem 4.3.1. *The predicate $compat_{\mathcal{F}}$ expresses compatibility w.r.t. $=$, and $\mathcal{F}$ preserves $=$.*

Proof.

- The first thesis follows trivially, since $compat_{\mathcal{F}}(\Gamma, \Gamma', e)$ requires Γ and Γ' to coincide on the free variables of e, on which the typing of e only depends.
- By inspection of *checkJoin* of Figure 11, it easily follows that the `out` parameter is uniquely determined by the other parameters of *checkJoin*. Thus, since by hypothesis they are all equal, the second thesis follows. □

Theorem 4.5. *The predicate $compat_{\mathcal{W}}$ expresses compatibility w.r.t. $=$, and $\mathcal{W}$ preserves $=$.*

Proof.

- The first thesis follows from the fact that $\mathcal{W}$ is syntax-driven, thus both the tree and the rules in a deduction (if any) only depend on e. Indeed, the implication in Definition 4.8 holds because all the premises that use Γ still hold when Γ' is used instead, since $\Gamma(y)$ equals to $\Gamma'(y)$ for all free variables y.
- Again, the second thesis follows since the `out` parameter is uniquely determined by the other parameters of *checkJoin*. □

Theorem 4.3.2. *The predicate $compat_{\mathcal{S}}$ expresses compatibility w.r.t. $=$, and $\mathcal{S}$ preserves $=$.*

Proof.

- Again, the first thesis follows trivially: $compat_{\mathcal{FS}}(\Gamma, \Gamma', p)$ requires Γ and Γ' to coincide on the free variables of p, on which the typing of p only depends.
- By inspection of *checkJoin* of Figure 11, it easily follows that the `out` parameter is uniquely determined by the other parameters of *checkJoin*. Thus, since by hypothesis they are all equal, the second thesis follows. □

Theorem 4.3.3. *$compat_{\mathcal{R}}$ expresses compatibility w.r.t. $=$, and $\mathcal{R}$ preserves $=$.*

Proof.

- We show the first part of the statement by induction on the rules of $\mathcal{R}$. The base cases are as follows:

  $\mathcal{R}$-VAL: since *val* has a fixed level, the thesis follows trivially.

  $\mathcal{R}$-VAR: since x is a free variable and $E(x) = E'(x)$ by hypothesis, the thesis follows.

$\mathfrak{R}$-SKIP: trivial.

$\mathfrak{R}$-INJECT: Since $pc' \sqsubseteq pc$ by hypothesis and since $C(pc) \sqsubseteq_C C_A$, by definition of $\sqsubseteq_C$ it follows that $C(pc') \sqsubseteq C(pc)$, thus by transitivity of C_A, we have $C(pc') \sqsubseteq_C C_A$.

The inductive step requires to show that, if for all t' appearing in the premises of the rule for t, where the induction hypothesis (IHP) reads as follows:

$$\begin{aligned}&\forall (E_{t'}, pc_{t'}), (E'_{t'}, pc'_{t'}).\\ &\quad compat((E_{t'}, pc_{t'}), (E'_{t'}, pc'_{t'}), t') \ \wedge \ (E_{t'}, pc_{t'}) \vdash_{\mathfrak{R}} t' : \ell\\ &\qquad \Rightarrow (E'_{t'}, pc'_{t'}) \vdash_{\mathfrak{R}} t' : \ell\end{aligned}$$

implies

$$\begin{aligned}&\forall (E_t, pc_t), (E'_t, pc'_t).\\ &\quad compat((E_t, pc_t), (E'_t, pc'_t), t) \ \wedge \ (E_t, pc_t) \vdash_{\mathfrak{R}} t : \ell\\ &\qquad \Rightarrow (E'_t, pc'_t) \vdash_{\mathfrak{R}} t : \ell.\end{aligned}$$

For that, we have the following cases:

$\mathfrak{R}$-OP: First, observe that $E_{t'} = E_t$ and $E'_{t'} = E'_t$. By IHP we get that $(E'_{t'}, _) \vdash_{\mathfrak{R}} e : \ell$ and $(E'_{t'}, _) \vdash_{\mathfrak{R}} e' : \ell'$, thus the thesis follows.

$\mathfrak{R}$-ASSIGN: As above, observe that $E_{t'} = E_t$ and $E'_{t'} = E'_t$. By (IHP) we get that $(E'_{t'}, _) \vdash_{\mathfrak{R}} e : \ell$. Also, by monotonicity of $\sqcup$, we have $\ell \sqcup pc'_t \sqsubseteq \ell \sqcup pc_t$. Finally, by transitivity of $\sqsubseteq$ the thesis follows.

$\mathfrak{R}$-SEQ, $\mathfrak{R}$-IF AND $\mathfrak{R}$-WHILE: follow directly by (IHP).

$\mathfrak{R}$-DECLASSIFY: similar to the case of $\mathfrak{R}$-ASSIGN, the only additional and non-trivial additional requirement is that we need to show that $I_A \not\sqsubseteq_I I(pc'_t)$ (note that $pc_t, pc'_t \neq _$). If $I(pc'_t) = I(pc_t)$, then the thesis follows easily. Otherwise note that under the hypothesis $I(pc'_t) \neq I(pc_t)$, $\not\sqsubseteq_I$ coincides with the inverse relation of $\sqsubseteq_I$ thus it is still a transitive relation, hence $I_A \not\sqsubseteq_I I(pc'_t)$.

- Determinism of $\mathfrak{R}$ suffices to show that it preserves $=$. □

Theorem 4.3.4. *$compat_{\mathfrak{I}}$ expresses compatibility w.r.t. $=$, and $\mathfrak{I}$ preserves $=$.*

Proof.

- We show the first part of the statement by structural induction on patterns and terms. The base cases are trivial:

$\mathfrak{I}$-PVAR, $\mathfrak{I}$-PINT AND $\mathfrak{I}$-PCCONSTR: easy by definition of $compat^p_{\mathfrak{I}}$ and since fresh variable are chosen algorithmically.

$\mathfrak{I}$-CONST, $\mathfrak{I}$-VAR AND $\mathfrak{I}$-CCONSTR: easy by definition of $compat^a_{\mathfrak{I}}$ and since fresh variable are chosen algorithmically.

The inductive step requires to show that, if for all t' appearing in the premises of the rule for t:

$$\forall \Gamma_{t'}, \Gamma'_{t'}.\ compat(\Gamma_{t'}, \Gamma'_{t'}, t') \ \wedge \ \Gamma \vdash_{\mathfrak{I}} t' : R \Rightarrow \Gamma'_{t'} \vdash_{\mathfrak{I}} t' : R \qquad (IHP)$$

implies

$$\forall \Gamma_t, \Gamma'_t.\ compat(\Gamma_t, \Gamma'_t, t) \ \wedge \ \Gamma \vdash_{\mathfrak{I}} t : R \Rightarrow \Gamma'_t \vdash_{\mathfrak{I}} t : R$$

For that, we have the following cases:

$\mathcal{I}$-PdConstr: directly follows from (IHP) and the fact that fresh variable are chosen algorithmically.

$\mathcal{I}$-Abs and $\mathcal{I}$-dConstr: follow from (IHP), the determinism of mgu and the fact that fresh variable are chosen algorithmically.

$\mathcal{I}$-App, $\mathcal{I}$-Match, $\mathcal{I}$-Let and $\mathcal{I}$-Try: as above, with the additional observation that if for some set of variables S, $E_{|S} = E'_{|S}$ then for any substitution θ it holds $(\theta E)_{|S} = (\theta E')_{|S}$.

- Determinism of $\mathcal{I}$ suffices to show that it preserves $=$. □

Theorem 4.3.5. *$compat_{\mathcal{D}}$ expresses compatibility w.r.t. $=$, and $\mathcal{D}$ preserves $=$.*

Proof.

- Easily follows by induction on the rules of the type system.
- Determinism of $\mathcal{D}$ suffices to show that it preserves $=$. □

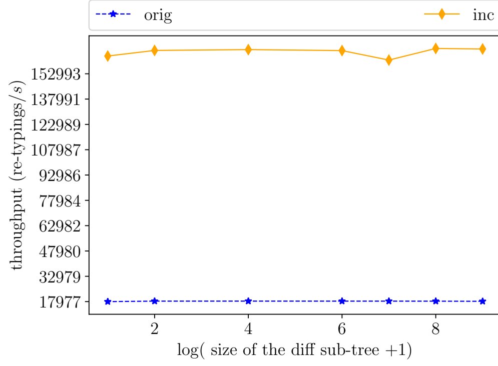
Theorem 4.3.6. *$compat_{\mathcal{P}}$ expresses compatibility w.r.t. $=$, and $\mathcal{P}$ preserves $=$.*

Proof.

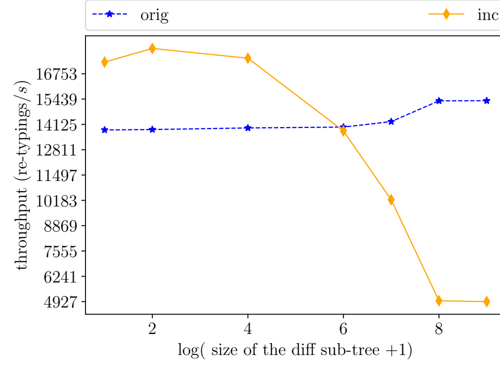
- Easily follows by induction on the rules of the type system.
- Determinism of $\mathcal{P}$ suffices to show that it preserves $=$. □

A.2 ADDITIONAL PLOTS FOR SECTION 4.4

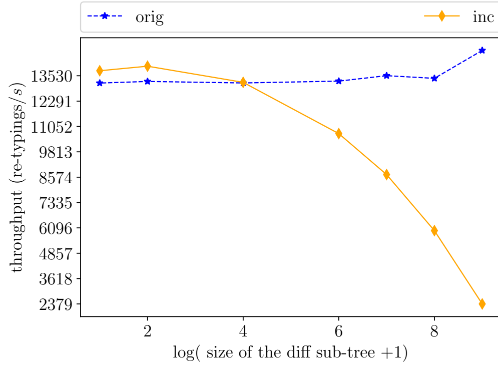
A.2.1 *Original vs. Incremental with simulated changes*



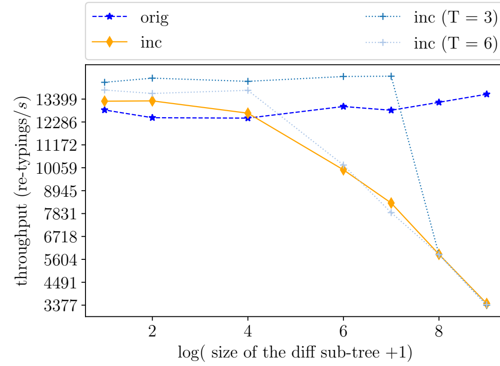
(a) $\#variables = 1$



(b) $\#variables = 2^7$

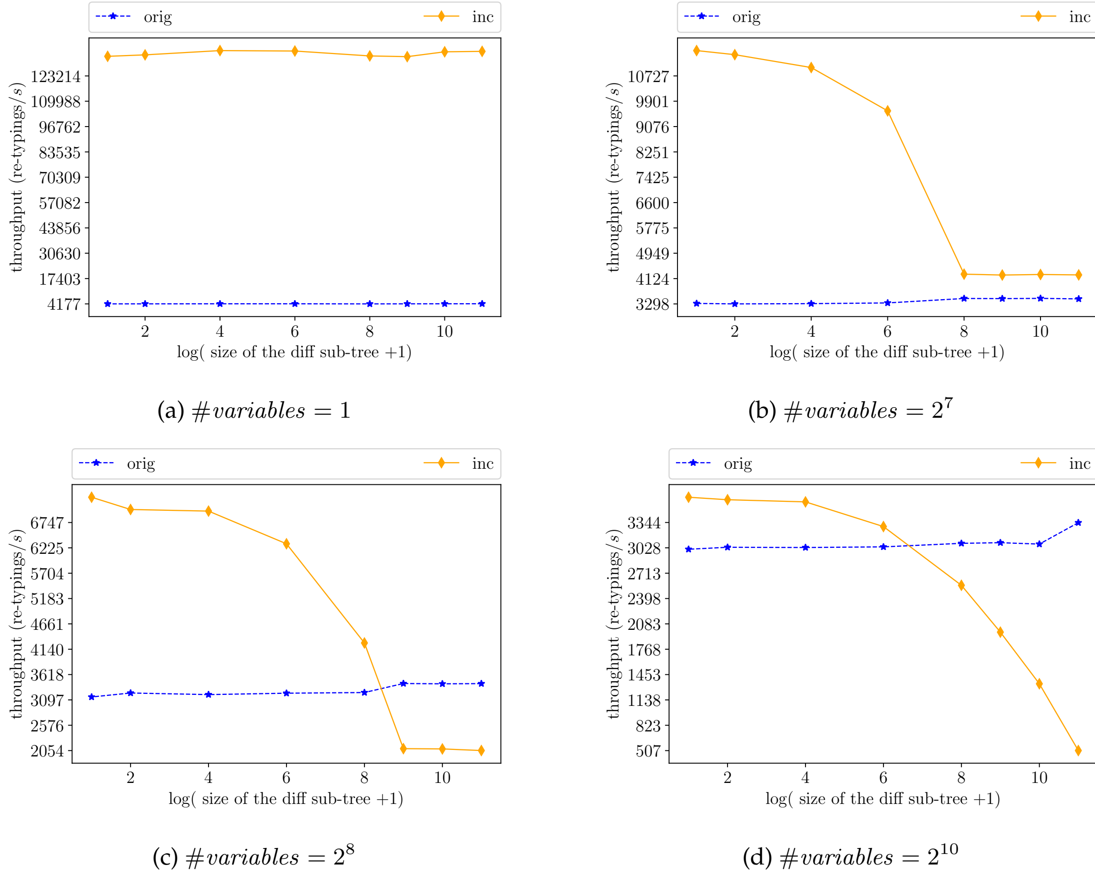


(c) $\#variables = 2^8$



(d) $\#variables = 2^9$

Figure 62: Experimental results for trees with $depth = 10$ comparing the number of re-typings per second vs. the number of nodes of the *diff* sub-tree. The blue, dashed plot is for the original type checking, while the orange, solid one is for the incrementalized one. Dotted plots report the behavior of the incrementalized type checker for various values of the threshold T. The x-axis is logarithmic, while the y-axis is scaled as necessary. The plots on the right consider the maximum number of variables.



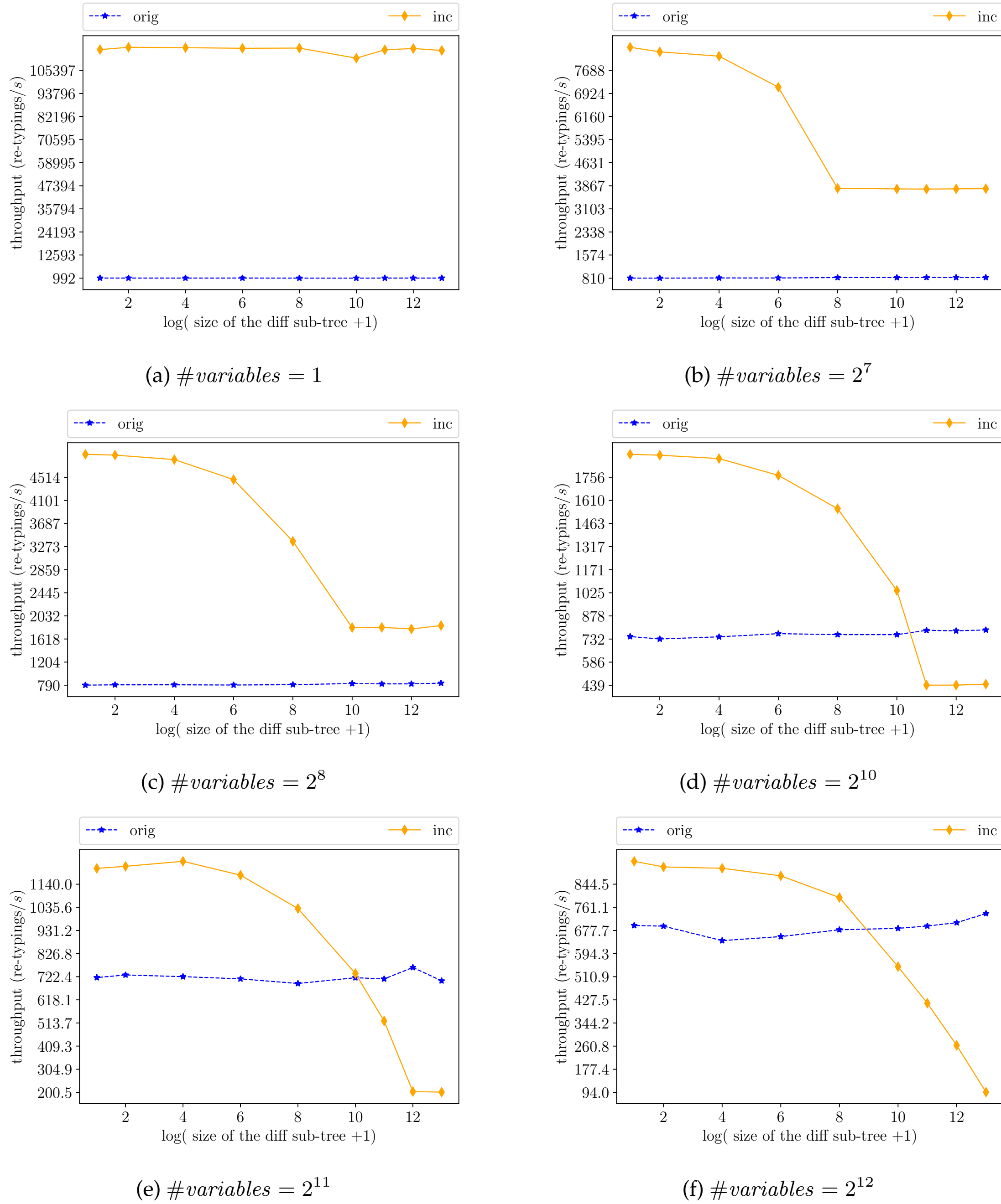
(a) $\#variables = 1$

(b) $\#variables = 2^7$

(c) $\#variables = 2^8$

(d) $\#variables = 2^{10}$

Figure 63: Experimental results for trees with $depth = 12$ comparing the number of re-typings per second vs. the number of nodes of the *diff* sub-tree. The blue, dashed plot is for the original type checking, while the orange, solid one is for the incrementalized one. Dotted plots report the behavior of the incrementalized type checker for various values of the threshold T. The x-axis is logarithmic, while the y-axis is scaled as necessary. The plots on the right consider the maximum number of variables.

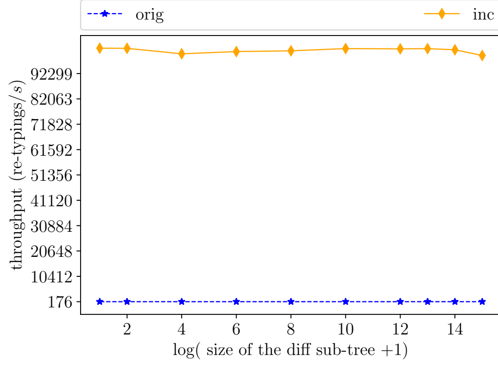


(a) $\#variables = 1$ (b) $\#variables = 2^7$

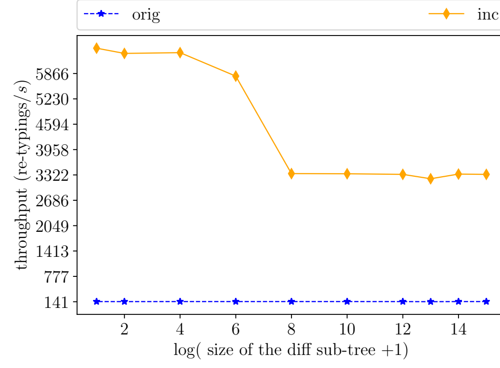
(c) $\#variables = 2^8$ (d) $\#variables = 2^{10}$

(e) $\#variables = 2^{11}$ (f) $\#variables = 2^{12}$

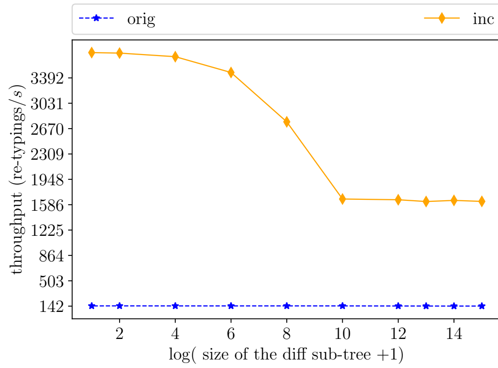
Figure 64: Experimental results for trees with $depth = 14$ comparing the number of re-typings per second vs. the number of nodes of the *diff* sub-tree. The blue, dashed plot is for the original type checking, while the orange, solid one is for the incrementalized one. Dotted plots report the behavior of the incrementalized type checker for various values of the threshold T. The x-axis is logarithmic, while the y-axis is scaled as necessary. The plots on the right consider the maximum number of variables.



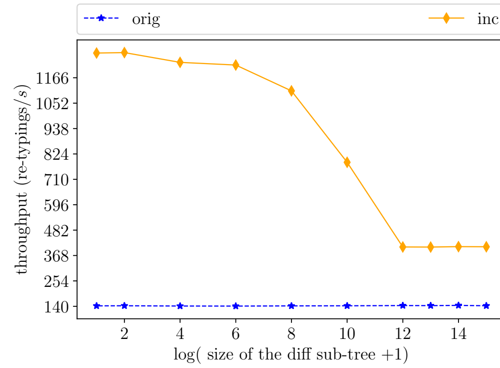
(a) $\#variables = 1$



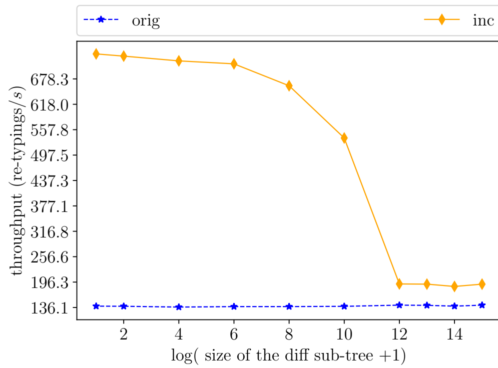
(b) $\#variables = 2^7$



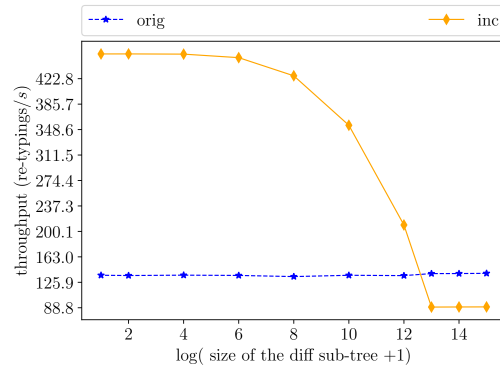
(c) $\#variables = 2^8$



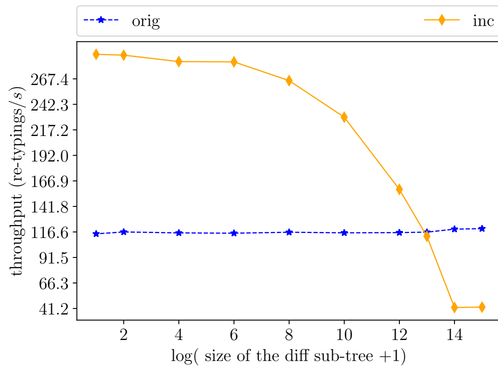
(d) $\#variables = 2^{10}$



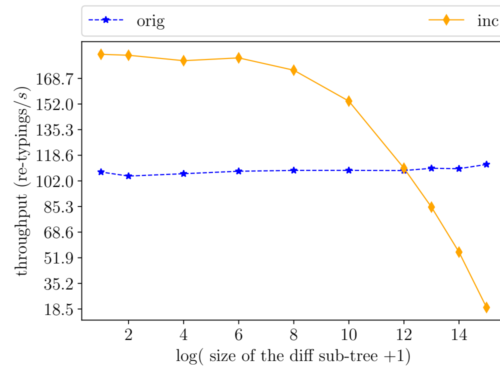
(e) $\#variables = 2^{11}$



(f) $\#variables = 2^{12}$



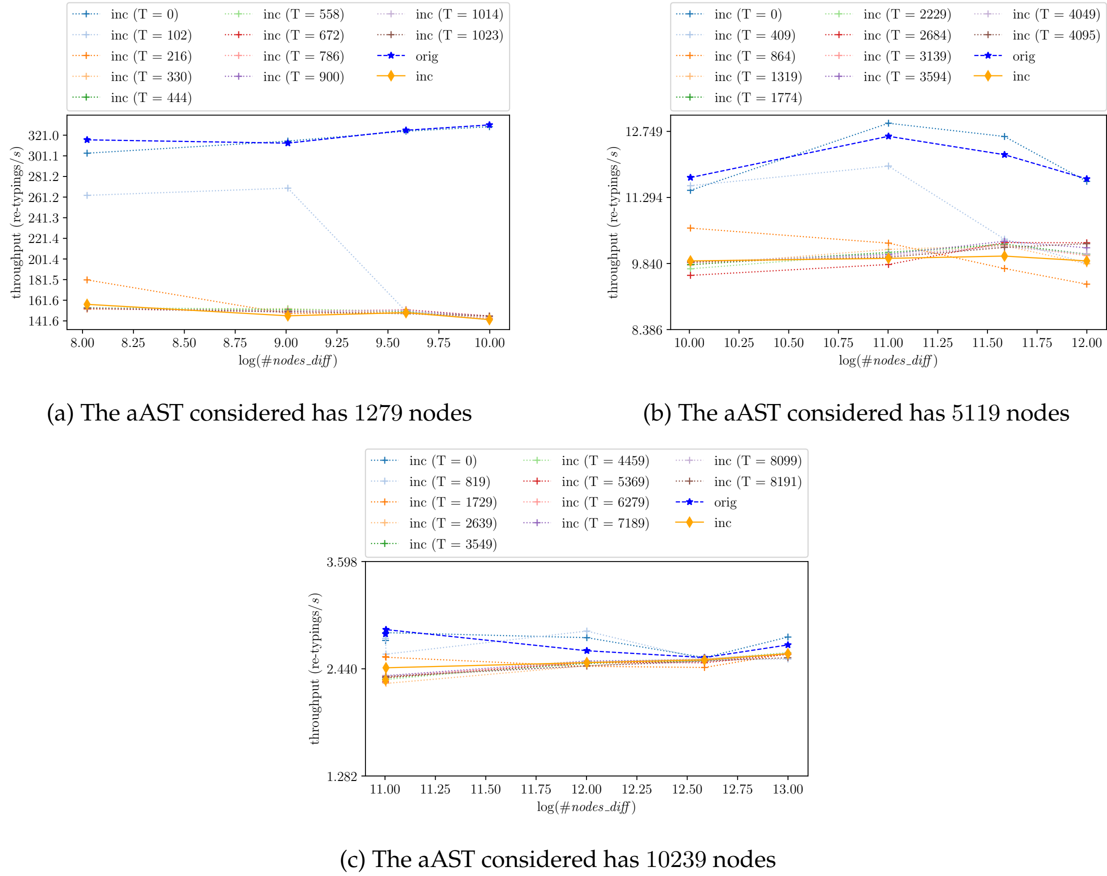
(g) $\#variables = 2^{13}$



(h) $\#variables = 2^{14}$

Figure 65: Experimental results for trees with $depth = 16$ comparing the number of re-typings per second vs. the number of nodes of the *diff* sub-tree. The blue, dashed plot is for the original type checking, while the orange, solid one is for the incrementalized one. Dotted plots report the behavior of the incrementalized type checker for various values of the threshold T. The x-axis is logarithmic, while the y-axis is scaled as necessary. The plots on the right consider the maximum number of variables.

A.2.2 *Original vs. Incremental with inter-dependencies*



(a) The aAST considered has 1279 nodes

(b) The aAST considered has 5119 nodes

(c) The aAST considered has 10239 nodes

Figure 66: Experimental results comparing the number of re-typings per second vs. the number of nodes of the *diff* on two unrolling of the factorial function. The dashed, star-marked plot is for the original type checking, the orange, solid one is for the incrementalized one. Dotted plots report the behavior of the incrementalized type checker for various values of the threshold T. The x-axis is logarithmic, while the y-axis is scaled as necessary.

B

SECURE COMPILATION AGAINST MICRO-ARCHITECTURAL ATTACKS

B.1 THE DEVICE OF SECTION 6.3.6.1 IS DETERMINISTIC

Proposition B.1. *If* $\mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^k \delta', t', t'_a$ *and* $\mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^k \delta'', t'', t''_a$*, then* $\delta' = \delta''$*,* $t' = t''$ *and* $t'_a = t''_a$*.*

Proof. Trivial. □

B.2 COMPLETE OPERATIONAL SEMANTICS RULES OF SANCUS$^{\mathrm{H}}$

INT

$$\frac{}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}$$

(CPU-HLT-UM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \vdash_{mode} \mathtt{UM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{HALT}} \; decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{HLT}$$

(CPU-NoIN)

$$\frac{\delta \overset{rd(w)}{\not\curvearrowright}_D}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{HALT}} \; decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{IN}\ \mathtt{r}$$

(CPU-NoOUT)

$$\frac{\delta \overset{wr(\mathcal{R}[\mathtt{r}])}{\not\curvearrowright}_D}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{HALT}} \; decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{OUT}\ \mathtt{r}$$

(CPU-HLT-PM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \vdash_{mode} \mathtt{PM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{EXC}_{\langle \delta, t+cycles(i), t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{HLT}$$

(CPU-DECODE-FAIL)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \bot}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{EXC}_{\langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}}$$

(CPU-VIOLATION-PM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \not\vdash_{mac} \mathtt{OK}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{EXC}_{\langle \delta, t+cycles(i), t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) \neq \bot$$

(CPU-MovL)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \\ \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 2][\texttt{r}_2 \mapsto \mathcal{M}[\mathcal{R}[\texttt{r}_1]]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B}\rangle \hookrightarrow_{\text{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}'\rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}'\rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{MOV @r}_1\ \texttt{r}_2$$

(CPU-MovS)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 4] \\ \mathcal{M}' = \mathcal{M}[\mathcal{R}[\texttt{r}_2] \mapsto \mathcal{R}[\texttt{r}_1]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B}\rangle \hookrightarrow_{\text{I}} \langle \delta'', t'', t''_a, \mathcal{M}'', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}'\rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}'', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}'\rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{MOV r}_1\ \texttt{0(r}_2\texttt{)}$$

(CPU-Mov)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \\ \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 2][\texttt{r}_2 \mapsto \mathcal{R}[\texttt{r}_1]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B}\rangle \hookrightarrow_{\text{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}'\rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}'\rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{MOV r}_1\ \texttt{r}_2$$

(CPU-MovI)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \\ \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 4][\texttt{r} \mapsto w] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B}\rangle \hookrightarrow_{\text{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}'\rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}'\rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{MOV}\ \#w\ \texttt{r}$$

(CPU-Cmp) $\quad i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{CMP r}_1\ \texttt{r}_2$

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 2][\texttt{r}_2 \mapsto \mathcal{R}[\texttt{r}_1] - \mathcal{R}[\texttt{r}_2]] \\ \mathcal{R}'' = \mathcal{R}'[\texttt{sr.N} \mapsto (\mathcal{R}'[\texttt{r}_2] < 0), \texttt{sr.Z} \mapsto (\mathcal{R}'[\texttt{r}_2] == 0), \texttt{sr.C} \mapsto (\mathcal{R}'[\texttt{r}_2] \neq 0), \texttt{sr.V} \mapsto overflow(\mathcal{R}[\texttt{r}_1] - \mathcal{R}[\texttt{r}_2])] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}\rangle \hookrightarrow_{\text{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\texttt{pc}], \mathcal{B}'\rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\texttt{pc}], \mathcal{B}'\rangle}$$

(CPU-Nop)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \\ \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 2] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B}\rangle \hookrightarrow_{\text{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}'\rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}'\rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{NOP}$$

(CPU-Reti-Chain)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle \qquad \mathcal{B} \neq \bot \\ i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{R}[\texttt{sr.GIE}] = 1 \\ t'_a \neq \bot \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}, \mathcal{R}[\texttt{pc}], \mathcal{B}\rangle \hookrightarrow_{\text{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B}\rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B}\rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{RETI}$$

(CPU-Reti-PrePad)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle \qquad \mathcal{B} \neq \bot \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad (\mathcal{R}[\texttt{sr.GIE}] = 0 \ \vee \ t'_a = \bot)\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \langle \delta', t', t'_a, \mathcal{M}, \mathcal{B}.\mathcal{R}, \mathcal{B}.pc_{old}, \langle \bot, \bot, \mathcal{B}.t_{pad}\rangle\rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{RETI}$$

(CPU-Reti-Pad)

$$\frac{\begin{array}{c}\mathcal{B} = \langle \bot, \bot, t_{pad}\rangle \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{t_{pad}} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot\rangle \hookrightarrow_{\text{I}} \langle \delta'', t'', t''_a, \mathcal{M}, \mathcal{R}', pc_{old}, \mathcal{B}'\rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}, \mathcal{R}', pc_{old}, \mathcal{B}'\rangle}$$

(CPU-RETI)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \texttt{OK} \\ \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{M}[\mathcal{R}[\texttt{sp}]+2], \texttt{sr} \mapsto \mathcal{M}[\mathcal{R}[\texttt{sp}]], \texttt{sp} \mapsto \mathcal{R}[\texttt{sp}]+4] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t_a' \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \rightarrow \langle \delta', t', t_a', \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \bot \rangle}\ i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{RETI}$$

(CPU-JZO)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \\ \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}]+2] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t_a' \\ \mathcal{D} \vdash \langle \delta', t', t_a', \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}\ i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{JZ}\ \&\texttt{r} \wedge \mathcal{R}[\texttt{sr}].Z = 0$$

(CPU-JZ1)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \\ \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{r}]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t_a' \\ \mathcal{D} \vdash \langle \delta', t', t_a', \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}\ i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{JZ}\ \&\texttt{r} \wedge \mathcal{R}[\texttt{sr}].Z = 1$$

(CPU-JMP)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \\ i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{r}]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t_a' \\ \mathcal{D} \vdash \langle \delta', t', t_a', \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}\ i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{JMP}\ \&\texttt{r}$$

(CPU-IN)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \delta \overset{rd(w)}{\rightsquigarrow}_D \delta' \\ \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}]+2][\texttt{r} \mapsto w] \qquad \mathcal{D} \vdash \delta', t, t_a \curvearrowright_D^{cycles(i)} \delta'', t', t_a' \\ \mathcal{D} \vdash \langle \delta'', t', t_a', \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta''', t'', t_a'', \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta''', t'', t_a'', \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}\ i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{IN r}$$

(CPU-OUT)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \\ \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}]+2] \qquad \delta \overset{wr(\mathcal{R}[\texttt{r}])}{\rightsquigarrow}_D \delta' \qquad \mathcal{D} \vdash \delta', t, t_a \curvearrowright_D^{cycles(i)} \delta'', t', t_a' \\ \mathcal{D} \vdash \langle \delta'', t', t_a', \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta''', t'', t_a'', \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta''', t'', t_a'', \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}\ i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{OUT r}$$

(CPU-NOT)

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \\ \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}]+2][\texttt{r} \mapsto \neg\mathcal{R}[\texttt{r}]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t_a' \\ \mathcal{D} \vdash \langle \delta', t', t_a', \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}\ i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{NOT r}$$

(CPU-AND) $\quad i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{AND}\ \texttt{r}_1\ \texttt{r}_2$

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}]+2][\texttt{r}_2 \mapsto \mathcal{R}[\texttt{r}_1] \& \mathcal{R}[\texttt{r}_2]] \\ \mathcal{R}'' = \mathcal{R}'[\texttt{sr.N} \mapsto \mathcal{R}'[\texttt{r}_2] \& \texttt{0x8000}, \texttt{sr.Z} \mapsto (\mathcal{R}'[\texttt{r}_2] == 0), \texttt{sr.C} \mapsto (\mathcal{R}'[\texttt{r}_2] \neq 0), \texttt{sr.V} \mapsto 0] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t_a' \qquad \mathcal{D} \vdash \langle \delta', t', t_a', \mathcal{M}, \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}''', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}''', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}$$

(CPU-ADD) $\quad i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{ADD}\ \texttt{r}_1\ \texttt{r}_2$

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}]+2][\texttt{r}_2 \mapsto \mathcal{R}[\texttt{r}_1] + \mathcal{R}[\texttt{r}_2]] \\ \mathcal{R}'' = \mathcal{R}'[\texttt{sr.N} \mapsto (\mathcal{R}'[\texttt{r}_2] < 0), \texttt{sr.Z} \mapsto (\mathcal{R}'[\texttt{r}_2] == 0), \texttt{sr.C} \mapsto (\mathcal{R}'[\texttt{r}_2] \neq 0), \texttt{sr.V} \mapsto overflow(\mathcal{R}[\texttt{r}_1] + \mathcal{R}[\texttt{r}_2])] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t_a' \qquad \mathcal{D} \vdash \langle \delta', t', t_a', \mathcal{M}, \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}''', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}''', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}$$

(CPU-SUB) $\quad i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{SUB}\ \texttt{r}_1\ \texttt{r}_2$

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}]+2][\texttt{r}_2 \mapsto \mathcal{R}[\texttt{r}_1] - \mathcal{R}[\texttt{r}_2]] \\ \mathcal{R}'' = \mathcal{R}'[\texttt{sr.N} \mapsto (\mathcal{R}'[\texttt{r}_2] < 0), \texttt{sr.Z} \mapsto (\mathcal{R}'[\texttt{r}_2] == 0), \texttt{sr.C} \mapsto (\mathcal{R}'[\texttt{r}_2] \neq 0), \texttt{sr.V} \mapsto overflow(\mathcal{R}[\texttt{r}_1] - \mathcal{R}[\texttt{r}_2])] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t_a' \qquad \mathcal{D} \vdash \langle \delta', t', t_a', \mathcal{M}, \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}''', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t_a'', \mathcal{M}', \mathcal{R}''', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}$$

B.3 COMPLETE OPERATIONAL SEMANTICS RULES OF Sancus^{L}

(INT-UM-P)

$$\frac{pc_{old} \vdash_{mode} \texttt{UM} \quad \mathcal{R}[\texttt{sr}].\texttt{GIE} = 1 \quad t_a \neq \bot \quad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto isr, \texttt{sr} \mapsto 0, \texttt{sp} \mapsto \mathcal{R}[\texttt{sp}] - 4] \quad \mathcal{M}' = \mathcal{M}[\mathcal{R}[\texttt{sp}] - 2 \mapsto \mathcal{R}[\texttt{pc}], \mathcal{R}[\texttt{sp}] - 4 \mapsto \mathcal{R}[\texttt{sr}]] \quad \mathcal{D} \vdash \delta, t, \bot \curvearrowright^{6}_{D} \delta', t', t'_a}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', pc_{old}, \mathcal{B} \rangle}$$

(INT-UM-NP)

$$\frac{pc_{old} \vdash_{mode} \texttt{UM} \quad (\mathcal{R}[\texttt{sr}].\texttt{GIE} = 0 \vee t_a = \bot)}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}$$

(INT-PM-P)

$$\frac{k = \texttt{MAX_TIME} - (t - t_a) \quad pc_{old} \vdash_{mode} \texttt{PM} \quad \mathcal{R}[\texttt{sr}].\texttt{GIE} = 1 \quad t_a \neq \bot \quad \mathcal{R}' = \mathcal{R}_0[\texttt{pc} \mapsto isr] \quad \mathcal{D} \vdash \delta, t, \bot \curvearrowright^{6+k}_{D} \delta', t', t'_a \quad \mathcal{B}' = \langle \mathcal{R}, pc_{old}, t - t_a \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta', t', \bot, \mathcal{M}, \mathcal{R}', pc_{old}, \mathcal{B}' \rangle}$$

(INT-PM-NP)

$$\frac{pc_{old} \vdash_{mode} \texttt{PM} \quad (\mathcal{R}[\texttt{sr}].\texttt{GIE} = 0 \vee t_a = \bot)}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}$$

(CPU-HLT-UM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \quad \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \vdash_{mode} \texttt{UM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{HALT}} \; decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{HLT}$$

(CPU-NoIN)

$$\frac{\delta \overset{rd(w)}{\not\curvearrowright}_{D}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{HALT}} \; decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{IN r}$$

(CPU-NoOUT)

$$\frac{\delta \overset{wr(\mathcal{R}[\texttt{r}])}{\not\curvearrowright}_{D}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{HALT}} \; decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{OUT r}$$

(CPU-HLT-PM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \quad \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \vdash_{mode} \texttt{PM}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{EXC}_{\langle \delta, t + cycles(i), t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{HLT}$$

(CPU-Decode-Fail)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \bot}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{EXC}_{\langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}}$$

(CPU-Violation-PM)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \not\vdash_{mac} \mathtt{OK}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \mathrm{EXC}_{\langle \delta, t+cycles(i), t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle}} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) \neq \bot$$

(CPU-MovL)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \\ \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}] + 2][\mathtt{r}_2 \mapsto \mathcal{M}[\mathcal{R}[\mathtt{r}_1]]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{MOV}\ @\mathtt{r}_1\ \mathtt{r}_2$$

(CPU-MovS)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}] + 4] \\ \mathcal{M}' = \mathcal{M}[\mathcal{R}[\mathtt{r}_2] \mapsto \mathcal{R}[\mathtt{r}_1]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}', \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}'', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}'', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{MOV}\ \mathtt{r}_1\ \mathtt{0}(\mathtt{r}_2)$$

(CPU-Mov)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \\ \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}] + 2][\mathtt{r}_2 \mapsto \mathcal{R}[\mathtt{r}_1]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{MOV}\ \mathtt{r}_1\ \mathtt{r}_2$$

(CPU-MovI)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \\ \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}] + 4][\mathtt{r} \mapsto w] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{MOV}\ \#w\ \mathtt{r}$$

(CPU-Cmp) $\quad i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{CMP}\ \mathtt{r}_1\ \mathtt{r}_2$

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \qquad \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}] + 2][\mathtt{r}_2 \mapsto \mathcal{R}[\mathtt{r}_1] - \mathcal{R}[\mathtt{r}_2]] \\ \mathcal{R}'' = \mathcal{R}'[\mathtt{sr.N} \mapsto (\mathcal{R}'[\mathtt{r}_2] < 0), \mathtt{sr.Z} \mapsto (\mathcal{R}'[\mathtt{r}_2] == 0), \mathtt{sr.C} \mapsto (\mathcal{R}'[\mathtt{r}_2] \neq 0), \mathtt{sr.V} \mapsto overflow(\mathcal{R}[\mathtt{r}_1] - \mathcal{R}[\mathtt{r}_2])] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle}$$

(CPU-Nop)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \\ \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}] + 2] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \\ \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{NOP}$$

(CPU-Reti-Chain)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad \mathcal{B} \neq \bot \\ i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{R}[\mathtt{sr.GIE}] = \mathtt{1} \\ t'_a \neq \bot \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}, \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}', \mathcal{R}[\mathtt{pc}], \mathcal{B} \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{RETI}$$

(CPU-Reti-PrePad)

$$\frac{\begin{array}{c} \mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad \mathcal{B} \neq \bot \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad (\mathcal{R}[\mathtt{sr.GIE}] = \mathtt{0} \ \vee \ t'_a = \bot) \end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta', t', t'_a, \mathcal{M}, \mathcal{B}.\mathcal{R}, \mathcal{B}.pc_{old}, \langle \bot, \bot, \mathcal{B}.t_{pad} \rangle \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{RETI}$$

(CPU-Reti-Pad)

$$\frac{\mathcal{B} = \langle \bot, \bot, t_{pad} \rangle \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{t_{pad}} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \hookrightarrow_I \langle \delta'', t'', t''_a, \mathcal{M}, \mathcal{R}', pc_{old}, \mathcal{B}' \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}, \mathcal{R}', pc_{old}, \mathcal{B}' \rangle}$$

(CPU-Reti)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \bot \vdash_{mac} \texttt{OK} \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{M}[\mathcal{R}[\texttt{sp}] + 2], \texttt{sr} \mapsto \mathcal{M}[\mathcal{R}[\texttt{sp}]], \texttt{sp} \mapsto \mathcal{R}[\texttt{sp}] + 4] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \bot \rangle \rightarrow \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \bot \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{RETI}$$

(CPU-Jz0)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 2] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{JZ \&r} \land \mathcal{R}[\texttt{sr}].Z = 0$$

(CPU-Jz1)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{r}]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{JZ \&r} \land \mathcal{R}[\texttt{sr}].Z = 1$$

(CPU-Jmp)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{r}]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{JMP \&r}$$

(CPU-In)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \delta \overset{rd(w)}{\rightsquigarrow}_D \delta' \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 2][\texttt{r} \mapsto w] \qquad \mathcal{D} \vdash \delta', t, t_a \curvearrowright_D^{cycles(i)} \delta'', t', t'_a \qquad \mathcal{D} \vdash \langle \delta'', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{IN r}$$

(CPU-Out)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 2] \qquad \delta \overset{wr(\mathcal{R}[\texttt{r}])}{\rightsquigarrow}_D \delta' \qquad \mathcal{D} \vdash \delta', t, t_a \curvearrowright_D^{cycles(i)} \delta'', t', t'_a \qquad \mathcal{D} \vdash \langle \delta'', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta''', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{OUT r}$$

(CPU-Not)

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 2][\texttt{r} \mapsto \neg \mathcal{R}[\texttt{r}]] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle} \; i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{NOT r}$$

(CPU-And) $i = decode(\mathcal{M}, \mathcal{R}[\texttt{pc}]) = \texttt{AND r}_1\, \texttt{r}_2$

$$\frac{\mathcal{B} \neq \langle \bot, \bot, t_{pad} \rangle \qquad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \texttt{OK} \qquad \mathcal{R}' = \mathcal{R}[\texttt{pc} \mapsto \mathcal{R}[\texttt{pc}] + 2][\texttt{r}_2 \mapsto \mathcal{R}[\texttt{r}_1] \& \mathcal{R}[\texttt{r}_2]] \qquad \mathcal{R}'' = \mathcal{R}'[\texttt{sr.N} \mapsto \mathcal{R}'[\texttt{r}_2] \& \texttt{0x8000}, \texttt{sr.Z} \mapsto (\mathcal{R}'[\texttt{r}_2] == 0), \texttt{sr.C} \mapsto (\mathcal{R}'[\texttt{r}_2] \neq 0), \texttt{sr.V} \mapsto 0] \qquad \mathcal{D} \vdash \delta, t, t_a \curvearrowright_D^{cycles(i)} \delta', t', t'_a \qquad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}'', \mathcal{R}[\texttt{pc}], \mathcal{B} \rangle \hookrightarrow_I \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B} \rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\texttt{pc}], \mathcal{B}' \rangle}$$

(CPU-ADD) $i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{ADD}\ \mathtt{r}_1\ \mathtt{r}_2$

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle\bot,\bot,t_{pad}\rangle \quad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \quad \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}]+2][\mathtt{r}_2 \mapsto \mathcal{R}[\mathtt{r}_1] + \mathcal{R}[\mathtt{r}_2]] \\ \mathcal{R}'' = \mathcal{R}'[\mathtt{sr.N} \mapsto (\mathcal{R}'[\mathtt{r}_2] < 0), \mathtt{sr.Z} \mapsto (\mathcal{R}'[\mathtt{r}_2] == 0), \mathtt{sr.C} \mapsto (\mathcal{R}'[\mathtt{r}_2] \neq 0), \mathtt{sr.V} \mapsto overflow(\mathcal{R}[\mathtt{r}_1] + \mathcal{R}[\mathtt{r}_2])] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_{D}^{cycles(i)} \delta', t', t'_a \quad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}\rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathtt{pc}], \mathcal{B}'\rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathtt{pc}], \mathcal{B}'\rangle}$$

(CPU-SUB) $i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \mathtt{SUB}\ \mathtt{r}_1\ \mathtt{r}_2$

$$\frac{\begin{array}{c}\mathcal{B} \neq \langle\bot,\bot,t_{pad}\rangle \quad i, \mathcal{R}, pc_{old}, \mathcal{B} \vdash_{mac} \mathtt{OK} \quad \mathcal{R}' = \mathcal{R}[\mathtt{pc} \mapsto \mathcal{R}[\mathtt{pc}]+2][\mathtt{r}_2 \mapsto \mathcal{R}[\mathtt{r}_1] - \mathcal{R}[\mathtt{r}_2]] \\ \mathcal{R}'' = \mathcal{R}'[\mathtt{sr.N} \mapsto (\mathcal{R}'[\mathtt{r}_2] < 0), \mathtt{sr.Z} \mapsto (\mathcal{R}'[\mathtt{r}_2] == 0), \mathtt{sr.C} \mapsto (\mathcal{R}'[\mathtt{r}_2] \neq 0), \mathtt{sr.V} \mapsto overflow(\mathcal{R}[\mathtt{r}_1] - \mathcal{R}[\mathtt{r}_2])] \\ \mathcal{D} \vdash \delta, t, t_a \curvearrowright_{D}^{cycles(i)} \delta', t', t'_a \quad \mathcal{D} \vdash \langle \delta', t', t'_a, \mathcal{M}, \mathcal{R}'', \mathcal{R}[\mathtt{pc}], \mathcal{B}\rangle \hookrightarrow_{\mathrm{I}} \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathtt{pc}], \mathcal{B}'\rangle\end{array}}{\mathcal{D} \vdash \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \rightarrow \langle \delta'', t'', t''_a, \mathcal{M}', \mathcal{R}''', \mathcal{R}[\mathtt{pc}], \mathcal{B}'\rangle}$$

B.4 PROOF OF PROGRESS OF SECTION 6.4.3

Theorem 6.1 (Progress). *For all* $C = \langle \mathcal{M}_C, \mathcal{D}\rangle$, $\mathcal{M}_M$ *and configuration* c

- $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c \not\rightarrow \implies c = \mathrm{HALT}$
- $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c \not\rightarrow \implies c = \mathrm{HALT}$.

Proof. Since no conclusion of the Sancus$^{\mathrm{H}}$ and Sancus$^{\mathrm{L}}$ semantic rules has HALT as starting configuration, this distinguished configuration is trivially stuck.

Also, HALT is the only stuck configuration because any $c = \langle \delta, t, t_a, \mathcal{M}, \mathcal{R}, pc_{old}, \mathcal{B}\rangle \neq$ HALT can progress. We show this for Sancus$^{\mathrm{H}}$; for Sancus$^{\mathrm{L}}$ just substitute $\rightarrow$ for $\rightarrow$.

If $\mathcal{B} \neq \langle \bot, \bot, t_{pad}\rangle$, the following four cases arise:

1. If $decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) = \bot$, then (CPU-DECODE-FAIL) applies.
2. If $decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}]) \neq \bot \wedge i, \mathcal{R}, pc_{old}, \mathcal{B} \not\vdash_{mac} \mathtt{OK}$, then (CPU-VIOLATION-PM) applies.
3. If the device is not willing to synchronize with the CPU, either rule (CPU-NOIN) or rule (CPU-NOOUT) applies.
4. Otherwise, there is a rule for each $i = decode(\mathcal{M}, \mathcal{R}[\mathtt{pc}])$ leading to a target configuration. Indeed, all the cases that may arise are covered by the premises that
   - check well-formedness of i and non-violation of MAC;
   - are all mutually exclusive (e.g., $\mathcal{B} \neq \bot$ in (CPU-RETI-CHAIN) and (CPU-RETI-PREPAD) is dealt with in rule (CPU-RETI) or the requirements of the values of $\mathcal{R}[\mathtt{sr.GIE}]$ and t'_a in (CPU-RETI-CHAIN) appear negated in (CPU-RETI-PREPAD)); and
   - require the existence of values either built explicitly (e.g., the value of $\mathtt{sr.N}$ in (CPU-AND)) or through relations that are always defined (e.g., through the transition system for interrupts).

Otherwise, $\mathcal{B} = \langle \bot, \bot, t_{pad}\rangle$ and the rule (CPU-RETI-PAD) applies. □

B.5 PROOFS AND ADDITIONAL DEFINITION FOR SECTION 6.5.1

Lemma 6.1. *For any module* $\mathcal{M}_M$*, context* C*, and corresponding interrupt-less context* $C_{\not I}$*:*

$$C_{\not I}[\mathcal{M}_M]\Downarrow^{\mathrm{L}} \iff C[\mathcal{M}_M]\Downarrow^{\mathrm{H}}$$

Proof. By definition of $\mathcal{D} \vdash \cdot \curvearrowright^k_D \cdot$, the value t_a in the CPU configuration (that signals the presence of an unhandled interrupt) is changed only when an interrupt has been raised since the last time it was checked.

Since any $int?$ action has been substituted with an ϵ, t_a is never changed from its initial $\bot$ value.

Since the only difference in behavior between the two levels is in the interrupt logic, and since the ISR in $C_{\mathcal{I}}$ is never invoked (thus, it does not affect the program behavior), $\mathcal{D} \vdash \cdot \hookrightarrow_{\mathbf{I}} \cdot$ behaves exactly as $\mathcal{D} \vdash \cdot \hookrightarrow_{\mathsf{I}} \cdot$. So, $C_{\mathcal{I}}[\mathcal{M}_M]{\Downarrow}^{\mathbf{L}}$ implies $C[\mathcal{M}_M]{\Downarrow}^{\mathsf{H}}$ and vice versa. □

Lemma 6.2 (Reflection). $\forall \mathcal{M}_M, \mathcal{M}_{M'}. (\mathcal{M}_M \simeq^{\mathbf{L}} \mathcal{M}_{M'} \implies \mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'})$.

Proof. We can expand the hypothesis using the definition of $\simeq^{\mathbf{L}}$ and $\simeq^{\mathsf{H}}$ as follows:

$$(\forall C. C[\mathcal{M}_M]{\Downarrow}^{\mathbf{L}} \iff C[\mathcal{M}_{M'}]{\Downarrow}^{\mathbf{L}}) \implies (\forall C'. C'[\mathcal{M}_M]{\Downarrow}^{\mathsf{H}} \iff C'[\mathcal{M}_{M'}]{\Downarrow}^{\mathsf{H}}).$$

For any C' we can build the corresponding interrupt-less context $C'_{\mathcal{I}}$.
Since interrupt-less contexts are a (strict) subset of all the contexts, by hypothesis:

$$C'_{\mathcal{I}}[\mathcal{M}_M]{\Downarrow}^{\mathbf{L}} \iff C'_{\mathcal{I}}[\mathcal{M}_{M'}]{\Downarrow}^{\mathbf{L}}.$$

But from Lemma 6.1 it follows that

$$C'[\mathcal{M}_M]{\Downarrow}^{\mathsf{H}} \iff C'[\mathcal{M}_{M'}]{\Downarrow}^{\mathsf{H}}.$$ □

Definition B.1 (Complete interrupt segments). *Let $\overline{\alpha} = \alpha_0 \cdots \alpha_n$ be a fine-grained trace. The set $\mathbb{I}_{\overline{\alpha}}$ of* complete interrupt segments *of $\overline{\alpha}$ is defined as follows:*

$$\mathbb{I}_{\overline{\alpha}} \triangleq \{(i, j) \mid \alpha_i = \mathtt{handle!}(k) \land \alpha_j = \mathtt{reti?}(k') \land i < j \land \forall i < l < j. \alpha_l = \xi\}.$$

B.6 DEFINITIONS AND PROOFS FOR LEMMATA 6.3 AND 6.4

The following proposition easily follows from the above definitions:

Proposition B.2. *Both $\overset{P}{\approx}$ and $\overset{U}{\approx}$ (Definition 6.8, Page 142) are equivalence relations.*

Proof. Trivial. □

B.6.1 *Properties of Definition 6.9*

The first proposition says that if a configuration can take a step, also another P-equivalent configuration can.

Proposition B.3. *If $c_1 \overset{P}{\approx} c_2$, $c_1 \vdash_{mode} \mathtt{PM}$, $\mathcal{D}' \vdash c_1 \rightarrow c'_1$ then $decode(\mathcal{M}_1, \mathcal{R}_1[\mathtt{pc}]) = decode(\mathcal{M}_2, \mathcal{R}_2[\mathtt{pc}])$ and $\mathcal{D}' \vdash c_2 \rightarrow c'_2$.*

Proof. Since $c_1 \overset{P}{\approx} c_2$ and $c_1 \vdash_{mode} \mathtt{PM}$, it also holds that $c_2 \vdash_{mode} \mathtt{PM}$. Also, the instruction $decode(\mathcal{M}_1, \mathcal{R}_1[\mathtt{pc}])$ is decoded in both $\mathcal{M}_1$ and $\mathcal{M}_2$ at the same protected address, hence $decode(\mathcal{M}_1, \mathcal{R}_1[\mathtt{pc}]) = decode(\mathcal{M}_2, \mathcal{R}_2[\mathtt{pc}])$, and $\mathcal{D}' \vdash c_2 \rightarrow c'_2$. □

Proposition B.4. *If $c_1 \overset{P}{\approx} c_2$, $c_1 \vdash_{mode} \mathtt{PM}$, $\mathcal{D} \vdash c_1 \rightarrow c'_1$, $\mathcal{D}' \vdash c_2 \rightarrow c'_2$ and $\mathcal{B}'_1 \bowtie \mathcal{B}'_2$ then $c'_1 \overset{P}{\approx} c'_2$.*

Proof.
Since $c_1 \overset{P}{\approx} c_2$, $c_1 \vdash_{mode}$ PM and $\mathcal{D} \vdash c_1 \to c_1'$, by Proposition B.3, $i = decode(\mathcal{M}_1, \mathcal{R}_1[\texttt{pc}]) = decode(\mathcal{M}_2, \mathcal{R}_2[\texttt{pc}])$ and $\mathcal{D}' \vdash c_2 \to c_2'$.

Since $\mathcal{B}_1' \bowtie \mathcal{B}_2'$, we have two cases:

1. *Case* $\mathcal{B}_1' = \mathcal{B}_2' = \bot$. In this case we know that no interrupt handling started during the step, and by exhaustive cases on i we can show $c_1' \overset{P}{\approx} c_2'$:
   - *Case* $i \in \{\texttt{HLT}, \texttt{IN r}, \texttt{OUT r}\}$. In both cases we have $c_1' = \text{EXC}_{c_1} \overset{P}{\approx} \text{EXC}_{c_2} = c_2'$.
   - *Otherwise.* The relevant values in c_1' and c_2' just depend on values that coincide also in c_1 and c_2. Hence, by determinism of the rules, we get $c_1' \overset{P}{\approx} c_2'$.
2. *Case* $\mathcal{B}_1' \neq \bot$ *and* $\mathcal{B}_2' \neq \bot$. In this case an interrupt was handled, but the same instruction was indeed executed in protected mode, hence $\mathcal{M}_1' \overset{P}{=} \mathcal{M}_2'$. Also, $\mathcal{R}_1' \overset{\texttt{PM}}{\asymp}_{\texttt{UM}} \mathcal{R}_2'$ holds trivially, $\mathcal{B}_1' \bowtie \mathcal{B}_2'$ by hypothesis and $pc'_{old\,1} \vdash_{mode}$ UM and $pc'_{old\,2} \vdash_{mode}$ UM. Thus, $c_1' \overset{P}{\approx} c_2'$.

□

Some sequences of fine-grained traces preserve P-equivalence.

Proposition B.5. *If* $c_1 \overset{P}{\approx} c_2$, $\mathcal{D} \vdash c_1 \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_1}}{}^{*} c_1' \xRightarrow{\texttt{jmpIn?}(\mathcal{R})} c_1''$, $\mathcal{D}' \vdash c_2 \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2}}{}^{*} c_2' \xRightarrow{\texttt{jmpIn?}(\mathcal{R})} c_2''$, *then* $c_1'' \overset{P}{\approx} c_2''$.

Proof. We show by Noetherian induction over (ℓ_1, ℓ_2) that $\mathcal{M}_1' \overset{P}{=} \mathcal{M}_2'$. For that, we use well-founded relation $(\ell_1, \ell_2) \prec (\ell_1', \ell_2')$ *iff* $\ell_1 < \ell_1' \wedge \ell_2 < \ell_2'$.

- *Case* $(0,0)$. Trivial.
- *Case* $(0, \ell_2)$*, with* $\ell_2 > 0$*. (and symmetrically* $(\ell_1, 0)$*, with* $\ell_1 > 0$*)* We have to show that

  $$\mathcal{D} \vdash c_1 \xRightarrow{\varepsilon}{}^{*} c_1' \wedge \mathcal{D}' \vdash c_2 \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2}}{}^{*} c_2' \Rightarrow \mathcal{M}_1' \overset{P}{=} \mathcal{M}_2'$$

  Since from c_1 there is no step, $c_1 = c_1'$. Moreover a sequence of ξ was observed starting from c_2, and since both configurations are in unprotected mode and no violation occurred (see Table 4) the protected memory is unchanged. Thus, by transitivity of $\overset{P}{=}$, we have $\mathcal{M}_1' = \mathcal{M}_1 \overset{P}{=} \mathcal{M}_2 \overset{P}{=} \mathcal{M}_2'$.
- *Case* $(\ell_1, \ell_2) = (\ell_1' + 1, \ell_2' + 1)$. If

  $$\mathcal{D} \vdash c_1 \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_1'}}{}^{*} c_1''' \wedge \mathcal{D}' \vdash c_2 \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2'}}{}^{*} c_2''' \Rightarrow \mathcal{M}_1''' \overset{P}{=} \mathcal{M}_2''' \text{ (IHP)}$$

  then

  $$\mathcal{D} \vdash c_1 \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_1'}}{}^{*} c_1''' \xRightarrow{\xi} c_1' \wedge \mathcal{D}' \vdash c_2 \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2'}}{}^{*} c_2''' \xRightarrow{\xi} c_2' \Rightarrow \mathcal{M}_1' \overset{P}{=} \mathcal{M}_2'.$$

  By (IHP) we know that $\mathcal{M}_1''' \overset{P}{=} \mathcal{M}_2'''$. Indeed, since we observed ξ it means that $pc_{old}{}_1' \vdash_{mode} \texttt{m} \wedge pc_{old}{}_2' \vdash_{mode} \texttt{m}$. Moreover (see Figure 57) since ξ was observed starting from c_1''' and from c_2''' and since both configurations are in unprotected mode, protected memory is unchanged. Thus, $\mathcal{M}_1' \overset{P}{=} \mathcal{M}_1''' \overset{P}{=} \mathcal{M}_2''' \overset{P}{=} \mathcal{M}_2'$.

Since the instruction generating $\alpha = \texttt{jmpIn?}(\mathcal{R})$ was executed in unprotected mode, we have that $\mathcal{M}_1'' \stackrel{P}{=} \mathcal{M}_2''$. Also $\mathcal{R}_1'' = \mathcal{R} \stackrel{\texttt{PM}}{\asymp}_{\texttt{PM}} \mathcal{R} = \mathcal{R}_2''$, $pc'_{old1}{}'' \vdash_{mode} \texttt{UM}$, $pc'_{old2}{}'' \vdash_{mode} \texttt{UM}$ and $\mathcal{B}_1'' \bowtie \mathcal{B}_2''$. □

Proposition B.6. *If* $c_1 \stackrel{P}{\approx} c_2$, $\mathcal{D} \vdash c_1 \xRightarrow{\texttt{handle!}(k_1)}{}^* c_1' \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_1}}{}^* c_1'' \xRightarrow{\texttt{reti?}(k_1')} c_1'''$, $\mathcal{D}' \vdash c_2 \xRightarrow{\texttt{handle!}(k_2)}{}^* c_2' \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2}}{}^* c_2'' \xRightarrow{\texttt{reti?}(k_2')} c_2'''$, *then* $c_1''' \stackrel{P}{\approx} c_2'''$.

Proof. Since upon observation of $\texttt{handle!}(k_x)$ the protected memory cannot be modified, we know that $\mathcal{M}_1' \stackrel{P}{=} \mathcal{M}_2'$.

We show by Noetherian induction over (ℓ_1, ℓ_2) that $\mathcal{M}_1'' \stackrel{P}{=} \mathcal{M}_2''$. For that, we use well-founded relation $(\ell_1, \ell_2) \prec (\ell_1', \ell_2')$ *iff* $\ell_1 < \ell_1' \wedge \ell_2 < \ell_2'$.

- *Case* $(0, 0)$. Trivial.
- *Case* $(0, \ell_2)$*, with* $\ell_2 > 0$ *(and symmetrically* $(\ell_1, 0)$*, with* $\ell_1 > 0$*).* We have to show that

$$\mathcal{D} \vdash c_1' \xRightarrow{\varepsilon}{}^* c_1'' \wedge \mathcal{D}' \vdash c_2' \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2}}{}^* c_2'' \Rightarrow \mathcal{M}_1'' \stackrel{P}{=} \mathcal{M}_2''$$

  Since from c_1' there is no step, $c_1'' = c_1'$. Moreover a sequence of ξ was observed starting from c_2', and since both configurations are in unprotected mode and no violation occurred (see Table 4) the protected memory is unchanged. Thus, by transitivity of $\stackrel{P}{=}$, we have $\mathcal{M}_1'' = \mathcal{M}_1' \stackrel{P}{=} \mathcal{M}_2' \stackrel{P}{=} \mathcal{M}_2''$.
- *Case* $(\ell_1, \ell_2) = (\ell_1' + 1, \ell_2' + 1)$. If

$$\mathcal{D} \vdash c_1' \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_1'}}{}^* c_1^{iv} \wedge \mathcal{D}' \vdash c_2' \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2'}}{}^* c_2^{iv} \Rightarrow \mathcal{M}_1^{iv} \stackrel{P}{=} \mathcal{M}_2^{iv} \text{ (IHP)}$$

  then

$$\mathcal{D} \vdash c_1' \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_1'}}{}^* c_1^{iv} \xRightarrow{\xi} c_1'' \wedge \mathcal{D}' \vdash c_2' \xRightarrow{\overbrace{\xi \cdots \xi}^{\ell_2'}}{}^* c_2^{iv} \xRightarrow{\xi} c_2'' \Rightarrow \mathcal{M}_1'' \stackrel{P}{=} \mathcal{M}_2''.$$

  By (IHP) we know that $\mathcal{M}_1^{iv} \stackrel{P}{=} \mathcal{M}_2^{iv}$. Indeed, since we observed ξ it means that $pc_{old}{}_1'' \vdash_{mode} \texttt{UM} \wedge \vdash_{mode} \texttt{UM} pc_{old}{}_2''$. Moreover (see Figure 57) since ξ was observed starting from c_1^{iv} and from c_2^{iv} and since both configurations are in unprotected mode, no violation occurred and by Table 4 protected memory is unchanged. Thus, by transitivity of $\stackrel{P}{=}$, we have $\mathcal{M}_1'' \stackrel{P}{=} \mathcal{M}_1^{iv} \stackrel{P}{=} \mathcal{M}_2^{iv} \stackrel{P}{=} \mathcal{M}_2''$.

Thus, we have that $\mathcal{M}_1''' \stackrel{P}{=} \mathcal{M}_2'''$, since $\alpha = \texttt{reti?}(\cdot)$ does not modify protected memory. Also $\mathcal{R}_1''' \stackrel{\texttt{PM}}{\asymp}_{\texttt{UM}} \mathcal{R}_2'''$, $\mathcal{B}_1''' \bowtie \mathcal{B}_2'''$, $pc'_{old1} \vdash_{mode} \texttt{UM}$ and $pc'_{old2} \vdash_{mode} \texttt{UM}$, by definition of $\alpha = \texttt{reti?}(\cdot)$. □

Proposition B.7. *If* $c_1 \stackrel{P}{\approx} c_2$, $c_1 \vdash_{mode} \texttt{PM}$, $\mathcal{D} \vdash c_1 \xRightarrow{\alpha_1} c_1'$, $\mathcal{D}' \vdash c_2 \xRightarrow{\alpha_2} c_2'$, $\alpha_1, \alpha_2 \neq \texttt{handle!}(\cdot)$ *then* $\alpha_1 = \alpha_2$ *and* $c_1' \stackrel{P}{\approx} c_2'$.

Proof. By definition of fine-grained traces we know that the transition leading to the observation of α_1 happens upon the execution of an instruction that must also be executed starting from c_2 (by Proposition B.3) and that $c_1' \overset{P}{\approx} c_2'$ (by Proposition B.4). Also, since $c_1 \vdash_{mode} \texttt{PM}$, we know that $\alpha_1 \in \{\tau(k_1), \texttt{jmpOut!}(k_1; \mathcal{R}_1)\}$. Thus, in both cases and since by hypothesis $\alpha_2 \neq \texttt{handle!}(\cdot)$, it must be that $\alpha_2 = \alpha_1$. □

Proposition B.8. *If* $c_1 \overset{P}{\approx} c_2, \mathcal{D} \vdash c_1 \xRightarrow{\tau(k_1^{(0)}) \cdots \tau(k_1^{(n_1-1)}) \cdot \alpha_1}{}^* c_1', \mathcal{D}' \vdash c_2 \xRightarrow{\tau(k_2^{(0)}) \cdots \tau(k_2^{(n_2-1)}) \cdot \alpha_2}{}^* c_2'$*, and* $\alpha_1, \alpha_2 \neq \texttt{handle!}(\cdot)$ *then* $\tau(k_1^{(0)}) \cdots \tau(k_1^{(n_1-1)}) \cdot \alpha_1 = \tau(k_2^{(0)}) \cdots \tau(k_2^{(n_2-1)}) \cdot \alpha_2$ *and* $c_1' \overset{P}{\approx} c_2'$.

Proof. Corollary of Proposition B.7. □

P-equivalence is preserved by complete interrupt segments (recall Definition B.1). Indeed, from now onward denote

$$\begin{aligned}\overline{\alpha}_x \in \{\varepsilon\} \cup \\ \{\alpha_x^{(0)} \cdots \alpha_x^{(n_x-1)} \mid n_x \geq 1 \wedge \alpha_x^{(n_x-1)} = \texttt{reti?}(k_x^{(n_x-1)}) \wedge \\ \forall i.\, 0 \leq i \leq n_x - 1.\, \alpha_x^{(i)} \notin \{\bullet, \texttt{jmpIn?}(\mathcal{R}_x^{(i)}), \texttt{jmpOut!}(k_x^{(i)}; \mathcal{R}_x^{(i)})\}\}.\end{aligned}$$

Proposition B.9. *Let* $\mathcal{D}$ *and* $\mathcal{D}'$ *be two devices.*
If $c_1^{(0)} \overset{P}{\approx} c_2^{(0)}, \mathcal{D} \vdash c_1 \xRightarrow{\texttt{jmpIn?}(\mathcal{R})} c_1^{(0)} \xRightarrow{\overline{\alpha}_1}{}^* c_1^{(n_1)}$ *and* $\mathcal{D}' \vdash c_2 \xRightarrow{\texttt{jmpIn?}(\mathcal{R})} c_2^{(0)} \xRightarrow{\overline{\alpha}_2}{}^* c_2^{(n_2)}$ *then* $c_1^{(n_1)} \overset{P}{\approx} c_2^{(n_2)}$.

Proof. We first show by induction on $|\mathbb{I}_{\overline{\alpha}_1}|$ (see Definition B.1) that

$$\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}_1}{}^* c_1^{(n_1)} \wedge \mathcal{D}' \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}_2}{}^* c_2^{(n_2)} \Rightarrow c_1^{(n_1)} \overset{P}{\approx} c_2^{(n_2)}$$

assuming wlog that $|\mathbb{I}_{\overline{\alpha}_2}| \leq |\mathbb{I}_{\overline{\alpha}_1}|$.

- *Case* $|\mathbb{I}_{\overline{\alpha}_1}| = 0$. Trivial.
- *Case* $|\mathbb{I}_{\overline{\alpha}_1}| = |\mathbb{I}_{\overline{\alpha}_1'}| + 1$. If

$$\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}_1'}{}^* c_1^{(n_1')} \wedge \mathcal{D}' \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}_2'}{}^* c_2^{(n_2')} \Rightarrow c_1^{(n_1')} \overset{P}{\approx} c_2^{(n_2')} \text{ (IHP)}$$

then

$$\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}_1}{}^* c_1^{(n_1)} \wedge \mathcal{D}' \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}_2}{}^* c_2^{(n_2)} \Rightarrow c_1^{(n_1)} \overset{P}{\approx} c_2^{(n_2)}$$

Now let (i_1, j_1) be the new interrupt segment of $\overline{\alpha}_1$ that we split it as follows:

$$\overline{\alpha}_1 = \overline{\alpha}_1' \cdot \tau(k_1^{(n_1')}) \cdots \tau(k_1^{(i_1-1)}) \cdot \texttt{handle!}(k_1^{(i_1)}) \cdots \texttt{reti?}(k_1^{(j_1)})$$

The following two exhaustive cases may arise.

1. *Case* $|\mathbb{I}_{\overline{\alpha}_1}| = |\mathbb{I}_{\overline{\alpha}_2}|$. For some (i_2, j_2) we then have:

$$\overline{\alpha}_2 = \overline{\alpha}_2' \cdot \tau(k_2^{(n_2')}) \cdots \tau(k_2^{(i_2-1)}) \cdot \texttt{handle!}(k_2^{(i_2)}) \cdots \texttt{reti?}(k_2^{(j_2)})$$

By Propositions B.6 and B.8 we know that $c_1^{(n_1)} \overset{P}{\approx} c_2^{(n_2)}$, being reached through $\alpha_1^{(j_1)}$ and $\alpha_2^{(j_2)}$.

2. *Case* $|\mathbb{I}_{\overline{\alpha}_2}| < |\mathbb{I}_{\overline{\alpha}_1}|$. In this case we have

$$\overline{\alpha}_2 = \overline{\alpha}_2' \cdot \tau(k_2^{(n_2')}) \cdots \tau(k_2^{(n_2-2)}) \cdot \tau(k_2^{(n_2-1)})$$

with $c_1^\ell \overset{P}{\approx} c_2^\ell$ for $n_2' \leq \ell \leq n_2 - 2 = i_1 - 1$, where the last equality holds because the module is executing from configurations that are P-equivalent. As soon as the interrupt arrives, the same instruction is executed (Proposition B.3) that causes the same changes in the registers, the old program counter and the protected memory. In turn the first two are stored in the backup before handling the interrupt. They are then restored by the `RETI`, observed as $\alpha_1^{(j_1)}$, while the protected memory is left untouched. Consequently, we have that $c_1^{(n_1)} \overset{P}{\approx} c_2^{(n_2)}$, that are the configurations reached through $\alpha_1^{(j_1)}$ and $\tau(k_2^{(n_2)-1})$.

□

Finally, we can show that P-equivalence is preserved by coarse-grained traces:

Proposition B.10.
If $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \xRightarrow{\texttt{jmpIn?}(\mathcal{R})} c_1$ *and* $\mathcal{D}' \vdash \text{INIT}_{C'[\mathcal{M}_M]} \xRightarrow{\texttt{jmpIn?}(\mathcal{R})} c_2$ *then* $c_1 \overset{P}{\approx} c_2$.

Proof. By definition of coarse-grained traces, we have that in both premises $\texttt{jmpIn?}(\mathcal{R})$ is preceded by a sequence of ξ actions (possibly in different numbers). Since neither ξ actions nor $\texttt{jmpIn?}(\mathcal{R})$ ever change the protected memory (by definition of memory access control) and since the $\texttt{jmpIn?}(\mathcal{R})$ sets the registers to the values in $\mathcal{R}$, it follows that $c_1 \overset{P}{\approx} c_2$. □

The following definition gives an equality up to timings among coarse-grained traces:

Definition B.2. *Let* $\overline{\beta} = \beta_0 \dots \beta_n$ *and* $\overline{\beta}' = \beta_0' \dots \beta_{n'}'$ *be two coarse-grained traces. We say that* $\overline{\beta}$ *is* equal up to timings *to* $\overline{\beta}'$ *(written* $\overline{\beta} \approx \overline{\beta}'$*)* iff

$$n = n' \wedge (\forall i \in \{0, \dots, n\}.\, \beta_i = \beta_i' \vee (\beta_i = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \beta_i' = \texttt{jmpOut!}(\Delta t'; \mathcal{R}))).$$

Finally, the proposition below shows that the traces that are equal up to timings preserve the P-equivalence:

Proposition B.11. *If* $c_1 \overset{P}{\approx} c_2$, $\mathcal{D} \vdash c_1 \overset{\overline{\beta}}{\Longrightarrow}{}^* c_1'$, $\mathcal{D}' \vdash c_2 \overset{\overline{\beta}'}{\Longrightarrow}{}^* c_2'$ *and* $\overline{\beta} \approx \overline{\beta}'$ *then* $c_1' \overset{P}{\approx} c_2'$.

Proof. The thesis easily follows from Proposition B.5 and Proposition B.9. □

B.6.2 *Properties of Definition 6.8*

Also for U-equivalent configurations it holds that when one takes a step, also the other does.

Proposition B.12. *If* $c_1 \overset{U}{\approx} c_2$, $c_1 \vdash_{mode} \texttt{UM}$ *then* $decode(\mathcal{M}_1, \mathcal{R}_1[\texttt{pc}]) = decode(\mathcal{M}_2, \mathcal{R}_2[\texttt{pc}])$.

Proof. Since $c_1 \overset{U}{\approx} c_2$ and $c_1 \vdash_{mode} \texttt{UM}$, it also holds that $c_2 \vdash_{mode} \texttt{UM}$. Also, the instruction $decode(\mathcal{M}_1, \mathcal{R}_1[\texttt{pc}])$ is decoded in both $\mathcal{M}_1$ and $\mathcal{M}_2$ at the same unprotected address, hence $decode(\mathcal{M}_1, \mathcal{R}_1[\texttt{pc}]) = decode(\mathcal{M}_2, \mathcal{R}_2[\texttt{pc}])$. □

Next we prove that $\overset{U}{\approx}$ is preserved by unprotected-mode steps of the Sancus$^{\mathrm{L}}$ operational semantics:

Proposition B.13. *If* $c_1 \overset{U}{\approx} c_2$, $c_1 \vdash_{mode}$ UM *and* $\mathcal{D} \vdash c_1 \rightarrow c_1'$, *then* $\mathcal{D} \vdash c_2 \rightarrow c_2' \wedge c_1' \overset{U}{\approx} c_2'$.

Proof.
Since $c_1 \overset{U}{\approx} c_2$, $c_1 \vdash_{mode}$ UM and $\mathcal{D} \vdash c_1 \rightarrow c_1'$, by Proposition B.12, $i = decode(\mathcal{M}_1, \mathcal{R}_1[\mathtt{pc}]) = decode(\mathcal{M}_2, \mathcal{R}_2[\mathtt{pc}])$.

To show that $c_1' \overset{U}{\approx} c_2'$, we consider the following exhaustive cases:

- *Case* $i = \bot$. Since $c_1 \overset{U}{\approx} c_2$ we get $c_2 \vdash_{mode}$ UM and by definition of $\cdot \vdash \cdot \rightarrow \cdot$ we get $c_1' = \text{EXC}_{c_1}$ and $c_2' = \text{EXC}_{c_2}$. However, by definition of EXC., we have that $\mathcal{M}_1' \overset{U}{\doteq} \mathcal{M}_2'$, $c_1' \vdash_{mode}$ UM, $c_2' \vdash_{mode}$ UM, $\delta_1' = \delta_1 = \delta_2 = \delta_2'$, $t_1' = t_1 = t_2 = t_2'$, $t_{a_1}' = t_{a_1} = t_{a_2} = t_{a_2}'$, $\mathcal{R}_1' \overset{\mathtt{UM}}{\asymp}_{\mathtt{m}} \mathcal{R}_2'$, and $\bot = \mathcal{B}_1' \bowtie \mathcal{B}_2' = \bot$, i.e., $c_1' \overset{U}{\approx} c_2'$.
- *Case* $i = \mathtt{HLT}$. Trivial, since $c_1' = \text{HALT} = c_2'$.
- *Case* $i \neq \bot$. We have the following exhaustive sub-cases, depending on c_1':
  - *Case* $c_1' = \text{EXC}_{c_1}$. In this case a violation occurred, i.e., $i, \mathcal{R}_1, pc_{old1}, \mathcal{B}_1 \nvdash_{mac}$ OK. However, the same violation also occurs for c_2, since the only parts that may keep c_1 apart from c_2 are pc_{old} and $\mathcal{B}$, and thus $c_1' \overset{U}{\approx} c_2'$ because:
    * $pc_{old2} \neq pc_{old1}$, cannot cause a failure since unprotected code is executable from anywhere,
    * $\mathcal{B}_1 = \langle \mathcal{R}_1, pc_{old1}, t_{pad_1} \rangle \neq \langle \mathcal{R}_2, pc_{old2}, t_{pad_2} \rangle = \mathcal{B}_2$, cannot cause a failure since the additional conditions on the configuration imposed by the memory access control only concern values that are the same in both configurations.
  - *Case* $c_1' \neq \text{EXC}_{c_1}$ *and* $i = \mathtt{RETI}$. If $\mathcal{B}_1 = \bot$, then $\mathcal{B}_1 = \mathcal{B}_2 = \mathcal{B}_1' = \mathcal{B}_2' = \bot$, hence rule (CPU-Reti) applies and we get $c_1' \overset{U}{\approx} c_2'$ since $\mathcal{R}_1' = \mathcal{R}_2'$ and $\mathcal{D} \vdash \cdot \curvearrowright_D \cdot$ is a deterministic relation (Proposition B.1). If $\mathcal{B}_1 \neq \bot$ it must also be that $\mathcal{B}_2 \neq \bot$ by U-equivalence, so either rule (CPU-Reti-Chain) or rule (CPU-Reti-PrePad) applies. In the first case we get $c_1' \overset{U}{\approx} c_2'$ because $c_1 \overset{U}{\approx} c_2$ and by determinism of $\mathcal{D} \vdash \cdot \curvearrowright_D \cdot$ and $\mathcal{D} \vdash \cdot \hookrightarrow_{\mathbf{I}} \cdot$. In the second case we get $c_1' \overset{U}{\approx} c_2'$ since $\langle \bot, \bot, t_{pad_1}' \rangle = \mathcal{B}_1' \bowtie \mathcal{B}_2' = \langle \bot, \bot, t_{pad_2}' \rangle$ and $\mathcal{R}_1' \overset{\mathtt{UM}}{\asymp}_{\mathtt{PM}} \mathcal{R}_2'$ holds since we restored the register files from backups in which the interrupts were enabled (otherwise the CPU would not have handled the interrupt it is returning from).
  - *Case* $c_1' \neq \text{EXC}_{c_1}$ *and* $i \notin \{\bot, \mathtt{HLT}, \mathtt{RETI}\}$. All the other rules depend on both (1) parts of the configurations that are equal due to $c_1 \overset{U}{\approx} c_2$, and on (2) $\mathcal{D} \vdash \cdot \curvearrowright_D^5 \cdot$ and $\mathcal{D} \vdash \cdot \hookrightarrow_{\mathbf{I}} \cdot$ which are deterministic and have the same inputs (since $c_1 \overset{U}{\approx} c_2$). Hence, $c_1' \overset{U}{\approx} c_2'$ as requested. □

The above proposition carries on fine-grained traces, provided that the computation is carried on in unprotected mode:

Proposition B.14. *If* $c_1 \overset{U}{\approx} c_2$, $c_1 \vdash_{mode}$ UM, $\mathcal{D} \vdash c_1 \overset{\alpha}{\Longrightarrow} c_1'$ *then* $\mathcal{D} \vdash c_2 \overset{\alpha}{\Longrightarrow} c_2'$ *and* $c_1' \overset{U}{\approx} c_2'$.

Proof.
By Propositions B.12 and B.13, $c_1' \overset{U}{\approx} c_2'$ and $i = decode(\mathcal{M}_1, \mathcal{R}_1[\mathtt{pc}]) = decode(\mathcal{M}_2, \mathcal{R}_2[\mathtt{pc}])$.

Thus, since the same i is executed under U-equivalent configurations and since $c_1' \overset{U}{\approx} c_2'$, we have that $\mathcal{D} \vdash c_2 \overset{\alpha}{\Longrightarrow} c_2'$. □

Proposition B.15.
If $c_1 \overset{U}{\approx} c_2$*,* $c_1 \vdash_{mode}$ UM*,* $\mathcal{D} \vdash c_1 \xRightarrow{\xi \cdots \xi \cdot \alpha}{}^{*} c_1'$ *and* $\alpha \in \{\xi, \bullet, \mathtt{jmpIn?}(\mathcal{R}), \mathtt{reti?}(k)\}$ *then* $\mathcal{D} \vdash c_2 \xRightarrow{\xi \cdots \xi \cdot \alpha}{}^{*} c_2'$ *and* $c_1' \overset{U}{\approx} c_2'$.

Proof. The proof goes by induction on the length n of $\xi \cdots \xi$.

- *Case* $n = 0$. Proposition B.14 applies.
- *Case* $n' = n + 1$. By induction hypothesis for some c_1''', c_2''', c_1'' and c_2'' we have $\mathcal{D} \vdash c_1 \xRightarrow{\overbrace{\xi \cdots \xi}^{n'}} c_1''' \xRightarrow{\alpha} c_1''$, $\mathcal{D} \vdash c_2 \xRightarrow{\overbrace{\xi \cdots \xi}^{n'}} c_2''' \xRightarrow{\alpha} c_2''$ and $c_1'' \overset{U}{\approx} c_2''$. Thus, if $\mathcal{D} \vdash c_1''' \xRightarrow{\xi} c_1^{iv}$ (i.e., we observe a further ξ starting from c_1), by Proposition B.14 we get $\mathcal{D} \vdash c_2''' \xRightarrow{\xi} c_2^{iv}$ and $c_1^{iv} \overset{U}{\approx} c_2^{iv}$. Finally, by Proposition B.14 applies on c_1^{iv} and c_2^{iv} we get the thesis.

□

Now we move our attention to $\mathtt{handle!}(\cdot)$.

Proposition B.16. *If* $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$*,* $\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\tau(k_1^{(0)}) \cdots \tau(k_1^{(n_1-1)}) \cdot \mathtt{handle!}(k_1^{(n_1)})}{}^{*} c_1^{(n_1+1)}$ *and* $\mathcal{D} \vdash c_2^{(0)} \xRightarrow{\tau(k_2^{(0)}) \cdots \tau(k_2^{(n_2-1)}) \cdot \mathtt{handle!}(k_2^{(n_2)})}{}^{*} c_2^{(n_2+1)}$ *then* $c_1^{(n_1+1)} \overset{U}{\approx} c_2^{(n_2+1)}$.

Proof.

- By definition of fine-grained semantics, $\mathtt{handle!}(k_x^{(n_x)})$ only happens when an interrupt is handled with $c_x^{(n_x)}$ in protected mode.
- By definition of $\mathcal{D} \vdash \cdot \hookrightarrow_{\mathbf{I}} \cdot$, $\mathcal{R}_1^{(n_1+1)} = \mathcal{R}_2^{(n_2+1)} = \mathcal{R}_0[\mathtt{pc} \mapsto isr]$.
- Since unprotected memory cannot be changed by protected mode actions without causing a violation (that would cause the observation of a $\mathtt{jmpOut!}(\cdot;\cdot)$) and is not changed upon RETI when it happens in a configuration with backup different from $\bot$ (cfr. rules **(CPU-Reti-*)**), $\mathcal{M}_1^{(n_1+1)} \overset{U}{=} \mathcal{M}_2^{(n_2+1)}$.
- Since we observe $\mathtt{handle!}(k_x^{(n_x)})$ it must be that $\mathtt{GIE} = \mathtt{1}$ and it had to be such also in $c_x^{(0)}$ (because by definition the operations on registers cannot modified this flag in protected mode). Hence, $t_{a_x}^i = \bot$ for $0 \le i \le n_x$. Let t_{a1}^{int} and t_{a2}^{int} be the arrival times of the interrupt that originated the observations $\mathtt{handle!}(k_1^{(n_1)})$ and $\mathtt{handle!}(k_2^{(n_2)})$, resp. By definition of $\mathcal{D} \vdash \cdot \curvearrowright_D \cdot$, t_{a1}^{int} and t_{a2}^{int} are the first absolute times after $t_1^{(n_1)}$ and $t_2^{(n_2)}$ in which an interrupt was raised and, since $\mathcal{D}$ is deterministic and $t_{a_x}^{(i)} = \bot$ for $0 \le i \le n_x$, it must be that $t_{a1}^{int} = t_{a2}^{int} = t^{int}$ (recall that $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$ and that INor OUTinstructions are forbidden in protected mode).

  Assume now that the instruction during which the interrupt occurred ended at time t_x^f. Then we can write $t^{(n_x+1)}$ as:

$$
\begin{aligned}
t^{(n_x+1)} &= t^{(n_x)} + k_x^{(n_x)} \\
&= t^{(n_x)} + \underbrace{t^{int} - t^{(n_x)} + t_x^f - t^{int}}_{\text{Duration of the instruction}} + \underbrace{\mathtt{MAX_TIME} - t_x^f + t^{int}}_{\text{Mitigation from (INT-PM-P)}} + 6 \\
&= \cancel{t^{(n_x)}} + t^{int} - \cancel{t^{(n_x)}} + \cancel{t_x^f} - \cancel{t^{int}} + \mathtt{MAX_TIME} - \cancel{t_x^f} + \cancel{t^{int}} + 6 \\
&= t^{int} + \mathtt{MAX_TIME} + 6
\end{aligned}
$$

  and therefore $t^{(n_1+1)} = t^{(n_2+1)}$.

- Since $t^{(n_1+1)} = t^{(n_2+1)}$, $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$ and no interaction with $\mathcal{D}$ via INor OUTcan occur in protected mode, the deterministic device $\mathcal{D}$ performed the same number of steps in both computations, and then $t_{a_1}^{(n_1+1)} = t_{a_2}^{(n_2+1)}$ and $\delta_1^{(n_1+1)} = \delta_2^{(n_2+1)}$.

Hence, $c_1^{(n_1+1)} \overset{U}{\approx} c_2^{(n_2+1)}$ as requested. □

The following properties show that the combination of U-equivalence and trace equivalence induces some useful properties of modules and sequences of complete interrupt segments. Before doing that we define the $(\overline{a}, n)$-interrupt-limited version of a context C as the context that behaves as C but such that (1) the transition relation of its device results from unrolling at most n steps of its transition relation and (2) its device never raises interrupts *after* observing the sequence of actions $\overline{a}$:

Definition B.3. *Let $\mathcal{D} = \langle \Delta, \delta_{\text{init}}, \overset{a}{\rightsquigarrow}_D \rangle$ be an I/O device. Let $\overline{a}$ be a string over the signature A of I/O devices and denote ℓ as the function that associates to each string over A a unique natural number (e.g., its position in a suitable lexicographic order). Given a context $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$, we define its corresponding $(\overline{a}, n)$-*interrupt-limited context *as $C_{\leq \overline{a},n} = \langle \mathcal{M}_C, \mathcal{D}_{\leq \overline{a},n} \rangle$ where $\mathcal{D}_{\leq \overline{a},n} = \langle img(\overset{a}{\rightsquigarrow}_{D \leq \overline{a},n}) \cup \mathtt{dom}\left(\overset{a}{\rightsquigarrow}_{D \leq \overline{a},n}\right), 0, \overset{a}{\rightsquigarrow}_{D \leq \overline{a},n} \rangle$ and*

$$
\begin{aligned}
\overset{a}{\rightsquigarrow}_{D \leq \overline{a},n} \triangleq & \\
& \Big(\{ (p, a, p') \mid \forall \overline{a}'.\, p = \ell(\overline{a}') \wedge p' = \ell(\overline{a}' \cdot a) \wedge \delta_{init} \overset{\overline{a}'}{\rightsquigarrow}{}^*_D\, \delta \overset{a}{\rightsquigarrow}_D \delta' \wedge |\overline{a}' \cdot a| \leq n \} \setminus \\
& \quad \{ (p, int?, p') \mid \forall \overline{a}'.\, p = \ell(\overline{a} \cdot \overline{a}') \wedge p' = \ell(\overline{a} \cdot \overline{a}' \cdot int?) \} \Big) \cup \\
& \{ (p, \epsilon, p') \mid \forall \overline{a}'.\, p = \ell(\overline{a} \cdot \overline{a}') \wedge p' = \ell(\overline{a} \cdot \overline{a}' \cdot int?) \wedge \\
& \qquad \delta_{init} \overset{\overline{a} \cdot \overline{a}'}{\rightsquigarrow}{}^*_D\, \delta \overset{int?}{\rightsquigarrow}_D \delta' \wedge |\overline{a} \cdot \overline{a}' \cdot int?| \leq n \}.
\end{aligned}
$$

(Note that any $(\overline{a}, n)$-interrupt-limited context is actually a device, due to the constraint on its transition function).

Now, let

$$
\begin{aligned}
\overline{\alpha}_x \in \{\varepsilon\} \cup \{ \alpha_x^{(0)} \cdots \alpha_x^{(n_x-1)} \mid n_x \geq 1 \wedge \alpha_x^{(n_x-1)} = \mathtt{reti?}(k_x^{(n_x-1)}) \wedge \\
\forall i.\, 0 \leq i \leq n_x - 1.\, \alpha_x^{(i)} \notin \{ \bullet, \mathtt{jmpIn?}(\mathcal{R}_x^{(i)}), \mathtt{jmpOut!}(k_x^{(i)}; \mathcal{R}_x^{(i)}) \} \}.
\end{aligned}
$$

Proposition B.17. *If*

- $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$
- $\mathcal{D} \vdash \text{INIT}_{C[M_M]} \xRightarrow{\overline{\beta} \cdot \mathtt{jmpIn?}(\mathcal{R})}{}^* c_1^{(0)}$
- $\mathcal{D} \vdash \text{INIT}_{C[M_{M'}]} \xRightarrow{\overline{\beta} \cdot \mathtt{jmpIn?}(\mathcal{R})}{}^* c_2^{(0)}$
- $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$
- *for some* $m_1 \geq 0$, $\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}_1 \cdot \tau(k_1^{(n_1)}) \cdots \tau(k_1^{(n_1+m_1-1)}) \cdot \mathtt{jmpOut!}(k_1^{(n_1+m_1)}; \mathcal{R}')}{}^* c_1^{(n_1+m_1+1)}$
- *for some* $m_2 \geq 0$, $\mathcal{D} \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}_2 \cdot \tau(k_2^{(n_2)}) \cdots \tau(k_2^{(n_2+m_2-1)}) \cdot \mathtt{jmpOut!}(k_2^{(n_2+m_2)}; \mathcal{R}')}{}^* c_2^{(n_2+m_2+1)}$

then $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) = \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)})$.

Proof. We show this proposition by contraposition, by showing that $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) \neq \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)})$ then $\mathcal{M}_M \overset{T}{\neq} \mathcal{M}_{M'}$. For that it suffices to show that

$$\exists C'.\mathcal{D}' \vdash \mathrm{INIT}_{C'[\mathcal{M}_M]} \xRightarrow{\overline{\beta}\cdot\mathtt{jmpIn?}(\mathcal{R})}{}^{*} c_3^{(0)} \xRightarrow{\mathtt{jmpOut!}(\Delta t_3;\mathcal{R}_3^{(n_3+m_3)})} c_3^{(n_3+m_3+1)}$$

(i.e., $\mathcal{D} \vdash c_3^{(0)} \xRightarrow{\overline{\alpha}_3\cdot\tau(k_3^{(n_3)})\cdots\tau(k_3^{(n_3+m_3-1)})\cdot\mathtt{jmpOut!}(k_3^{(n_3+m_3)};\mathcal{R}_3^{(n_3+m_3)})}{}^{*} c_3^{(n_3+m_3+1)}$)
such that

$$\forall C''.\mathcal{D}'' \vdash \mathrm{INIT}_{C''[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}\cdot\mathtt{jmpIn?}(\mathcal{R})}{}^{*} c_4^{(0)} \xRightarrow{\mathtt{jmpOut!}(\Delta t_4;\mathcal{R}_4^{(n_4+m_4+1)})} c_4^{(n_4+m_4+1)}$$

with $\Delta t_3 \neq \Delta t_4$

(i.e., $\mathcal{D} \vdash c_4^{(0)} \xRightarrow{\overline{\alpha}_4\cdot\tau(k_4^{(n_4)})\cdots\tau(k_4^{(n_4+m_4-1)})\cdot\mathtt{jmpOut!}(k_4^{(n_4+m_4)};\mathcal{R}_4^{(n_4+m_4)})}{}^{*} c_4^{(n_4+m_4+1)}$).

Assume wlog that $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) < \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)})$. Noting that the first observable of $\overline{\beta}\cdot\mathtt{jmpIn?}(\mathcal{R})$ must be a $\mathtt{jmpIn?}(\cdot)$, by Propositions B.10 and B.11, we have that $c_1^{(0)} \overset{P}{\approx} c_3^{(0)}$ and, similarly, $c_2^{(0)} \overset{P}{\approx} c_4^{(0)}$. Thus, as a consequence of Propositions B.3, B.8 and B.9, $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) = \sum_{i=0}^{n_3+m_3} \gamma(c_3^{(i)})$ and $\sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)}) = \sum_{i=0}^{n_4+m_4} \gamma(c_4^{(i)})$.

Let $n \in \mathbb{N}$ be greater than the number of steps over the relation $\dot{\rightsquigarrow}_D$ in the computation $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^{*} c_1^{(n_1+m_1+1)}$ and let $\overline{a}$ be the sequence of actions over $\dot{\rightsquigarrow}_D$ in the computation $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^{*} c_1^{(0)}$. Choosing $C' = C_{\leq \overline{a},n}$ we get $\Delta t_3 = \sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) = \sum_{i=0}^{n_3+m_3} \gamma(c_3^{(i)})$. Any other context C'' that allows to observe the same $\overline{\beta}\cdot\mathtt{jmpIn?}(\mathcal{R})$ from $\mathrm{INIT}_{C''[\mathcal{M}_{M'}]}$ raises 0 or more interrupts "after" c_4^0, hence taking additional $S \geq 0$ cycles on top of those required for the instructions to be executed. Thus $\mathcal{M}_M \overset{T}{\neq} \mathcal{M}_{M'}$, since $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) < \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)})$ and $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) = \Delta t_3 < \Delta t_4 = \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)}) + S$. □

Proposition B.18. *If*

- $\mathcal{D} \vdash \mathrm{INIT}_{C[M_M]} \xRightarrow{\overline{\beta}\cdot\mathtt{jmpIn?}(\mathcal{R})}{}^{*} c_1^{(0)}$
- $\mathcal{D} \vdash \mathrm{INIT}_{C[M_{M'}]} \xRightarrow{\overline{\beta}'\cdot\mathtt{jmpIn?}(\mathcal{R})}{}^{*} c_2^{(0)}$
- $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$
- $\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}_1\cdot\tau(k_1^{(n_1)})\cdots\tau(k_1^{(n_1+m_1-1)})\cdot\alpha_1}{}^{*} c_1^{(n_1+m_1+1)}$ *for some* $m_1 \geq 0$ *and* $\alpha_1 \in \{\mathtt{jmpOut!}(k_1^{(n_1+m_1)};\mathcal{R}'), \mathtt{handle!}(k_1^{(n_1+m_1)})\}$
- $\mathcal{D} \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}_2\cdot\tau(k_2^{(n_2)})\cdots\tau(k_2^{(n_2+m_2-1)})\cdot\alpha_2}{}^{*} c_2^{(n_2+m_2+1)}$ *for some* $m_2 \geq 0$ *and* $\alpha_2 \in \{\mathtt{jmpOut!}(k_2^{(n_2+m_2)};\mathcal{R}'), \mathtt{handle!}(k_2^{(n_2+m_2)})\}$

then

1. $|\mathbb{I}_{\overline{\alpha}_1}| = |\mathbb{I}_{\overline{\alpha}_2}|$
2. $c_1^{(n_1)} \overset{U}{\approx} c_2^{(n_2)}$.

Proof. Assume wlog that $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) \leq \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)})$, and we prove by induction on $|\mathbb{I}_{\overline{\alpha}_1}|$ that

$$\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}_1}{}^{*} c_1^{(n_1)} \wedge \mathcal{D} \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}_2}{}^{*} c_1^{(n_2)} \quad \text{imply} \quad c_1^{(n_1)} \overset{U}{\approx} c_2^{(n_2)} \wedge |\mathbb{I}_{\overline{\alpha}_1}| = |\mathbb{I}_{\overline{\alpha}_2}|$$

- *Case* $|\mathbb{I}_{\overline{\alpha}_1}| = 0$. Since no complete interrupt segment was observed it means that $\overline{\alpha}_1$ cannot end with a `reti?`$(\cdot)$, so it must be $\overline{\alpha}_1 = \varepsilon$. Moreover, since $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$ and the value of the `GIE` bit cannot be changed in protected mode, we know that:
  - *Case* $\mathcal{R}_1^{(0)}[\texttt{sr.GIE}] = \mathcal{R}_2^{(0)}[\texttt{sr.GIE}] = \texttt{0}$. Then no `handle!`$(\cdot)$ can be observed in $\overline{\alpha}_2$, hence it must be that $\overline{\alpha}_2 = \varepsilon$ and the two thesis easily follow.
  - *Case* $\mathcal{R}_1^{(0)}[\texttt{sr.GIE}] = \mathcal{R}_2^{(0)}[\texttt{sr.GIE}] = \texttt{1}$. Then it means that no interrupt was raised by the device in the computation starting with $c_1^{(0)}$ and the same must happen in $c_2^{(0)}$ because of U-equivalence and $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) \leq \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)})$. Hence it must be that $\overline{\alpha}_2 = \varepsilon$ and the two thesis easily follow.
- *Case* $|\mathbb{I}_{\overline{\alpha}_1}| = |\mathbb{I}_{\overline{\alpha}'_1}| + 1$. If

  $$\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}'_1}{}^{*} c_1^{(n'_1)} \wedge \mathcal{D} \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}'_2}{}^{*} c_2^{(n'_2)} \quad \text{imply} \quad c_1^{(n'_1)} \overset{U}{\approx} c_2^{(n'_2)} \wedge |\mathbb{I}_{\overline{\alpha}'_1}| = |\mathbb{I}_{\overline{\alpha}'_2}| \text{ (IHP)}$$

  then

  $$\mathcal{D} \vdash c_1^{(0)} \xRightarrow{\overline{\alpha}_1}{}^{*} c_1^{(n_1)} \wedge \mathcal{D} \vdash c_2^{(0)} \xRightarrow{\overline{\alpha}_2}{}^{*} c_2^{(n_2)} \quad \text{imply} \quad c_1^{(n_1)} \overset{U}{\approx} c_2^{(n_2)} \wedge |\mathbb{I}_{\overline{\alpha}_1}| = |\mathbb{I}_{\overline{\alpha}_2}|$$

  Now let (i_1, j_1) be the new interrupt segment of $\overline{\alpha}_1$, that we split as follows:

  $$\overline{\alpha}_1 = \overline{\alpha}'_1 \cdot \tau(k_1^{(n'_1)}) \cdots \tau(k_1^{(i_1-1)}) \cdot \texttt{handle!}(k_1^{(i_1)}) \cdots \texttt{reti?}(k_1^{(j_1)}).$$

  Since by (IHP) $c_1^{(n'_1)} \overset{U}{\approx} c_2^{(n'_2)}$ and $\mathcal{D}$ is deterministic and no successfully I/O ever happens in protected mode, the first new interrupt (i.e., the one leading to the observation of `handle!`$(k_1^{(i_1)})$) is raised at the same cycle in both computations. Call $c_2^{(i_2)}$ the configuration at the beginning of the step of computation in which such interrupt was raised (the choice of indexes will be clear below). From this configuration only three cases for the fine-grained action might be observed:
  - *Case* $\tau(\cdot)$ *and* `jmpOut!`$(\cdot;\cdot)$. Never happens, since $\mathcal{B}_2^{(i_2+1)} \neq \bot$.
  - *Case* `handle!`$(k_2^{(i_2)})$. Proposition B.16 ensures that $c_2^{(i_2+1)} \overset{U}{\approx} c_1^{(i_1+1)}$, and Proposition B.15 that at some index j_2 a `reti?`$(k_2^{(j_2)})$ is observed in $\overline{\alpha}_2$, i.e., a new interrupt segment (i_2, j_2) is observed. Thus, $|\mathbb{I}_{\overline{\alpha}_2}| = |\mathbb{I}_{\overline{\alpha}'_2}| + 1 = |\mathbb{I}_{\overline{\alpha}'_1}| + 1 = |\mathbb{I}_{\overline{\alpha}_1}|$ (where the second equality holds by (IHP)). Finally, by definition of $\overline{\alpha}_2$, we have that $n_1 = j_1 + 1$ and $n_2 = j_2 + 2$, hence $c_1^{(n_1)} \overset{U}{\approx} c_2^{(n_2)}$.

□

The following proposition states that U-equivalent unprotected-mode configurations perform the same single coarse-grained action:

Proposition B.19. *If* $c_1 \overset{U}{\approx} c_2$, $c_1 \vdash_{mode} \texttt{UM}$ *and* $\mathcal{D} \vdash c_1 \overset{\beta}{\Longrightarrow} c'_1$, *then* $\mathcal{D} \vdash c_2 \overset{\beta}{\Longrightarrow} c'_2$ *and* $c'_1 \overset{U}{\approx} c'_2$.

Proof. Since $c_1 \vdash_{mode}$ UM, the segment of fine-grained trace that originated β (see Figure 58) is in the form:

$$\mathcal{D} \vdash c_1 \xRightarrow{\xi \cdots \xi \cdot \alpha}{}^* c_1'$$

with either $\alpha = \bullet$ or $\alpha = \texttt{jmpIn?}(\mathcal{R})$.
Proposition B.15 guarantees that:

$$\mathcal{D} \vdash c_2 \xRightarrow{\xi \cdots \xi \cdot \alpha}{}^* c_2' \wedge c_1' \overset{U}{\approx} c_2'.$$

Thus, $\mathcal{D} \vdash c_2 \overset{\beta}{\Longrightarrow\!\!\!\!\!\!\!\!\Longrightarrow} c_2'$ and $c_1' \overset{U}{\approx} c_2'$. □

Finally, we can show that U-equivalence is preserved by coarse-grained traces:

Proposition B.20. *If* $c_1 \overset{U}{\approx} c_2$, $c_1 \vdash_{mode}$ UM, $\mathcal{D} \vdash c_1 \overset{\overline{\beta}}{\Longrightarrow\!\!\!\!\!\!\!\!\Longrightarrow}{}^* c_1'$, $\mathcal{D} \vdash c_2 \overset{\overline{\beta}}{\Longrightarrow\!\!\!\!\!\!\!\!\Longrightarrow}{}^* c_2'$, $c_1' \vdash_{mode}$ UM *and* $c_2' \vdash_{mode}$ UM *then* $c_1' \overset{U}{\approx} c_2'$.

Proof. We show the proposition by induction on n, the length of $\overline{\beta}$:

- *Case* $n = 0$. By definition of $\overset{\varepsilon}{\Longrightarrow\!\!\!\!\!\!\!\!\Longrightarrow}{}^*$ we know that it must be $c_1' = c_1$ and $c_2' = c_2'$ and the thesis easily follows.
- *Case* $n = n' + 1$. The only case in which a coarse-grained trace can be extended by just one action, while remaining in unprotected mode, is when the action is $\bullet$. In this case the hypothesis easily follows from the definition of $\bullet$ and U-equivalence.
- *Case* $n = n' + 2$. If

$$\mathcal{D} \vdash c_1 \overset{\overline{\beta}}{\Longrightarrow\!\!\!\!\!\!\!\!\Longrightarrow}{}^* c_1'' \wedge \mathcal{D} \vdash c_2 \overset{\overline{\beta}}{\Longrightarrow\!\!\!\!\!\!\!\!\Longrightarrow}{}^* c_2'' \wedge \mathcal{R}_1''[\texttt{pc}] \vdash_{mode} \texttt{UM} \wedge \mathcal{R}_2''[\texttt{pc}] \vdash_{mode} \texttt{UM} \text{ imply } c_1'' \overset{U}{\approx} c_2''$$

then

$$\mathcal{D} \vdash c_1 \overset{\overline{\beta}}{\Longrightarrow\!\!\!\!\!\!\!\!\Longrightarrow}{}^* c_1'' \overset{\beta\beta'}{\Longrightarrow\!\!\!\!\!\!\!\!\Longrightarrow} c_1' \;\wedge\; \mathcal{D} \vdash c_2 \overset{\overline{\beta}}{\Longrightarrow\!\!\!\!\!\!\!\!\Longrightarrow}{}^* c_2'' \overset{\beta\beta'}{\Longrightarrow\!\!\!\!\!\!\!\!\Longrightarrow} c_2' \;\wedge\; \mathcal{R}_1'[\texttt{pc}] \vdash_{mode} \texttt{UM} \wedge$$
$$\mathcal{R}_2'[\texttt{pc}] \vdash_{mode} \texttt{UM}\text{, imply } c_1' \overset{U}{\approx} c_2'.$$

By cases on $\beta\beta'$:

  – *Case* $\beta\beta' = \texttt{jmpIn?}(\mathcal{R})\,\bullet$. Directly follows from definition of $\bullet$ and $\overset{U}{\approx}$.
  – *Case* $\beta\beta' = \texttt{jmpIn?}(\mathcal{R})\;\texttt{jmpOut!}(\Delta t; \mathcal{R}')$. By definition they are originated by

$$\mathcal{D} \vdash c_1'' \xRightarrow{\xi \cdots \xi \cdot \texttt{jmpIn?}(\mathcal{R})}{}^* c_1^{(0)} \xRightarrow{\alpha_1^{(0)} \cdots \alpha_1^{(n_1-1)}}{}^* c_1^{(n_1)} \xRightarrow{\texttt{jmpOut!}(k_1^{(n_1)}; \mathcal{R}')} c_1'$$
$$\mathcal{D} \vdash c_2'' \xRightarrow{\xi \cdots \xi \cdot \texttt{jmpIn?}(\mathcal{R})}{}^* c_2^{(0)} \xRightarrow{\alpha_2^{(0)} \cdots \alpha_2^{(n_2-1)}}{}^* c_2^{(n_2)} \xRightarrow{\texttt{jmpOut!}(k_2^{(n_2)}; \mathcal{R}')} c_2'.$$

  By *(IHP)* and by Proposition B.15 we can conclude that $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$.
  Let $c_x^{(M_x)}$ be the configuration generated by the last $\texttt{reti?}(\cdot)$ in $\alpha_x^{(0)} \cdots \alpha_x^{(n_x-1)}$. By Proposition B.18 the number of completely handled interrupts is the same in the two traces and $c_1^{(M_1)} \overset{U}{\approx} c_2^{(M_2)}$. Also:

    * By definition of $\texttt{jmpOut!}(k_1^{(n_1)}; \mathcal{R}')$ and $\texttt{jmpOut!}(k_2^{(n_2)}; \mathcal{R}')$ we trivially get $\mathcal{R}_1' = \mathcal{R}_2' = \mathcal{R}'$.

* Since unprotected memory cannot be changed in protected mode (see Table 4) and $c_1^{(M_1)} \overset{U}{\approx} c_2^{(M_2)}$, $\mathcal{M}'_1 \overset{U}{=} \mathcal{M}'_2$.
* Let $\overline{\alpha}_x = \alpha_x^{(0)} \cdots \alpha_x^{(n_x-1)} \cdot \texttt{jmpOut!}(k_x^{(n_x)}; \mathcal{R}')$.
  By definition of $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R}')$:

$$t'_1 = t_1^{(0)} + \Delta t + \sum_{(i_1,j_1) \in |\mathbb{I}_{\overline{\alpha}_1}|} (t_1^{(j_1)} - t_1^{(i_1+1)})$$

$$t'_2 = t_2^{(0)} + \Delta t + \sum_{(i_2,j_2) \in |\mathbb{I}_{\overline{\alpha}_2}|} (t_2^{(j_2)} - t_2^{(i_2+1)})$$

  But $t_1^{(0)} = t_2^{(0)}$ since $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$. Also, each operand in $(t_1^{(j_1)} - t_1^{(i_1+1)})$ equals the corresponding $(t_2^{(j_2)} - t_2^{(i_2+1)})$ because for each (p^{th} element) $(i_1, j_1) \in \mathbb{I}_{\overline{\alpha}_1}$ and corresponding $(i_2, j_2) \in \mathbb{I}_{\overline{\alpha}_2}$, Proposition B.16 guarantees that $t_1^{(i_1+1)} = t_2^{(i_2+1)}$ and Proposition B.15 guarantees that $t_1^{(j_1)} = t_2^{(j_2)}$.
* Finally, since no interaction with $\mathcal{D}$ via `IN`or `OUT`occurs in protected mode and since the same deterministic device performed the same number of steps (starting from $c_1^{(0)} \overset{U}{\approx} c_2^{(0)}$), it follows that $t'_{a_1} = t'_{a_2}$ and $\delta'_1 = \delta'_2$.

□

B.7 PROOFS OF LEMMATA 6.3 AND 6.4

Proposition B.21. *Let $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$. If $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c_1$ and $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c_2$, then $c_1 \vdash_{mode} \texttt{m}$ and $c_2 \vdash_{mode} \texttt{m}$.*

Proof. Let β the last observable of $\overline{\beta}$. By definition c_1 and c_2 are such that, for some c'_1 and c'_2:

$$\mathcal{D} \vdash c'_1 \overset{\alpha}{\Longrightarrow} c_1 \qquad \mathcal{D} \vdash c'_2 \overset{\alpha}{\Longrightarrow} c_2$$

with α equal to $\bullet$, $\texttt{jmpIn?}(\cdot)$ or $\texttt{jmpOut!}(\cdot;\cdot)$ (depending on the value of β). In either case, since c'_1 and c'_1 are the configuration *right after* α and by definition of fine-grained traces, we have $c_1 \vdash_{mode} \texttt{m}$ and $c_2 \vdash_{mode} \texttt{m}$. □

Proposition B.22. *For any context $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$ and module $\mathcal{M}_M$, if $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \overset{\beta_0 \cdots \beta_n}{\Longrightarrow}{}^* c$ with $n \geq 0$, then the observables occurring*

(1) in even positions (β_0, β_2, ...) are either $\bullet$ or $\texttt{jmpIn?}(\mathcal{R})$ (for some $\mathcal{R}$)

(2) in odd positions (β_1, β_3, ...) are either $\bullet$ or $\texttt{jmpOut!}(\Delta t; \mathcal{R})$ (for some Δt and $\mathcal{R}$)

Proof. Both easily follow from Figures 57 and 58. □

First, we show that, due to the mitigation, the behavior of the context does not depend on the behavior of the module:

Lemma 6.3. *Let $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$. If $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c_1 \overset{\beta}{\Longrightarrow} c'_1$, $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c_2$, $c_1 \vdash_{mode} \texttt{UM}$ and $c_2 \vdash_{mode} \texttt{UM}$, then $\mathcal{D} \vdash c_2 \overset{\beta}{\Longrightarrow} c'_2$.*

Proof. First, observe that $\text{INIT}_{C[\mathcal{M}_M]} \overset{U}{\approx} \text{INIT}_{C[\mathcal{M}_{M'}]}$, because

$$\text{INIT}_{C[\mathcal{M}_M]} = \langle \delta_{\text{init}}, 0, \bot, \mathcal{M}_C \uplus \mathcal{M}_M, \mathcal{R}^{init}_{\mathcal{M}_C}, \texttt{0xFFFE}, \bot \rangle$$
$$\text{INIT}_{C[\mathcal{M}_{M'}]} = \langle \delta_{\text{init}}, 0, \bot, \mathcal{M}_C \uplus \mathcal{M}_{M'}, \mathcal{R}^{init}_{\mathcal{M}_C}, \texttt{0xFFFE}, \bot \rangle.$$

Since $\text{INIT}_{C[\mathcal{M}_M]} \vdash_{mode} \texttt{UM}$, $\text{INIT}_{C[\mathcal{M}_M]} \overset{U}{\approx} \text{INIT}_{C[\mathcal{M}_{M'}]}$, $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c_1$, $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c_2$, $c_1 \vdash_{mode} \texttt{UM}$ and $c_2 \vdash_{mode} \texttt{UM}$, by Proposition B.20 we have $c_1 \overset{U}{\approx} c_2$. Finally, since $\mathcal{D} \vdash c_1 \overset{\beta}{\Longrightarrow} c_1'$ and by Proposition B.19 we get $\mathcal{D} \vdash c_2 \overset{\beta}{\Longrightarrow} c_2'$. □

Then the following lemma shows that the isolation mechanism offered by the enclave guarantees that the behavior of the module is not influenced by the one of the context:

Lemma 6.4. *Let $C = \langle \mathcal{M}_C, \mathcal{D} \rangle$. If $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$, $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c_1'' \xRightarrow{\texttt{jmpIn?}(\mathcal{R}_1)} c_1 \overset{\beta}{\Longrightarrow} c_1'$ and $\mathcal{D} \vdash \text{INIT}_{C[\mathcal{M}_{M'}]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c_2'' \xRightarrow{\texttt{jmpIn?}(\mathcal{R}_2)} c_2$, then $\mathcal{D} \vdash c_2 \overset{\beta}{\Longrightarrow} c_2'$.*

Proof. Noting that $c_1 \vdash_{mode} \texttt{PM}$ and that the last observable of $\overline{\beta}$ is a $\texttt{jmpIn?}(\cdot)$, by definition of coarse-grained traces (see Figure 58) we have the following fine-grained traces starting from c_1'':

$$\mathcal{D} \vdash c_1'' \xRightarrow{\xi \cdots \xi \cdot \texttt{jmpIn?}(\mathcal{R}_1)}{}^* c_1 \xRightarrow{\overline{\alpha}_1}{}^* c_1^{(n_1)} \xRightarrow{\tau(k_1^{(n_1)}) \cdots \tau(k_1^{(n_1+m_1-1)}) \cdot \overline{\alpha}_1'}{}^* c_1'$$

with $\overline{\alpha}_1' \in \{\texttt{jmpOut!}(k_1; \mathcal{R}_1'), \texttt{handle!}(k_1) \cdot \xi \cdots \xi \cdot \bullet\}$.

Similarly for c_2 it must be:

$$\mathcal{D} \vdash c_2'' \xRightarrow{\xi \cdots \xi \cdot \texttt{jmpIn?}(\mathcal{R}_2)}{}^* c_2 \xRightarrow{\overline{\alpha}_2}{}^* c_2^{(n_2)} \xRightarrow{\tau(k_2^{(n_2)}) \cdots \tau(k_1^{(n_2+m_2-1)}) \cdot \overline{\alpha}_2'}{}^* c_2'.$$

with $\overline{\alpha}_2' \in \{\texttt{jmpOut!}(k_2; \mathcal{R}_2'), \texttt{handle!}(k_2) \cdot \xi \cdots \xi \cdot \bullet\}$.

We have now two cases:

- *Case* $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R})$. $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$ implies the existence of a context $C' = \langle \mathcal{M}_{C'}, \mathcal{D}' \rangle$ that allow us to observe $\mathcal{D}' \vdash \text{INIT}_{C'[\mathcal{M}_{M'}]} \overset{\overline{\beta}}{\Longrightarrow} c_3 \overset{\beta}{\Longrightarrow} c_3'$, i.e.,

  $$\mathcal{D}' \vdash c_3 \xRightarrow{\overline{\alpha}_3}{}^* c_3^{(n_3)} \xRightarrow{\tau(k_3^{(n_3)}) \cdots \tau(k_3^{(n_3+m_3-1)}) \cdot \overline{\alpha}_3'} c_3'$$

  with $\overline{\alpha}_3' \in \{\texttt{jmpOut!}(k_3; \mathcal{R}_3'), \texttt{handle!}(k_3) \cdot \xi \cdots \xi \cdot \bullet\}$.

  By Propositions B.10 and B.11 we have that $c_2 \overset{P}{\approx} c_3$, and by Proposition B.9 we conclude that $c_3^{(n_3)} \overset{P}{\approx} c_2^{(n_2)}$.

  Proposition B.8 guarantees that

  $$\tau(k_2^{(n_2)}) \cdots \tau(k_2^{(n_2+m_2-1)}) \cdot \overline{\alpha}_2' = \tau(k_3^{(n_3)}) \cdots \tau(k_3^{(n_3+m_3-1)}) \cdot \overline{\alpha}_3'.$$

  Since $\overline{\alpha}_2' = \overline{\alpha}_3' = \texttt{jmpOut!}(k_3; \mathcal{R}_1)$, we know that $\mathcal{D} \vdash c_2^{(n_2)} \xRightarrow{\texttt{jmpOut!}(\Delta t'; \mathcal{R}_1)} c_2'$.

  By Proposition 6.1, we have

  $$\Delta t = \sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) + (11 + \texttt{MAX_TIME}) \cdot |\mathbb{I}_{\overline{\alpha}_1}|$$
  $$\Delta t' = \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)}) + (11 + \texttt{MAX_TIME}) \cdot |\mathbb{I}_{\overline{\alpha}_2}|.$$

  Since by Propositions B.17 and B.18 we have $\sum_{i=0}^{n_1+m_1} \gamma(c_1^{(i)}) = \sum_{i=0}^{n_2+m_2} \gamma(c_2^{(i)})$ and $|\mathbb{I}_{\overline{\alpha}_1}| = |\mathbb{I}_{\overline{\alpha}_2}|$, we get $\Delta t = \Delta t'$ as requested.

- *Case* $\beta = \bullet$. It must be that $\overline{\alpha}'_1 = \texttt{handle!}(k_1) \cdot \xi \cdots \xi \cdot \bullet$ and $\overline{\alpha}'_2 = \texttt{handle!}(k_2) \cdot \xi \cdots \xi \cdot \bullet$. If this was not the case (i.e., if $\overline{\alpha}'_2 = \texttt{jmpOut!}(k_2; \mathcal{R}'_2)$), then c_2 could be swapped with c_1 (and c_1 with c_2) in the statement of this Lemma and the previous case would apply. Thus, the thesis follows.

□

B.8 PROOF OF PROPOSITION 6.3 AND ALGORITHM 3

From now onward, we simply write $\beta = \varepsilon$ (resp. $\beta' = \varepsilon$) if $\overline{\beta}$ (resp. $\overline{\beta}'$) is shorter than $\overline{\beta}'$ (resp. $\overline{\beta}$).

Proposition 6.3. *If $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ are two modules such that $\mathcal{M}_M \not\simeq^{\mathbf{L}} \mathcal{M}_{M'}$, then there always exist $\overline{\beta}$ and $\overline{\beta}'$ that are distinguishing traces for $\mathcal{M}_M$ and $\mathcal{M}_{M'}$.*

Proof. From the contrapositive of Lemma 6.6 we know that $\mathcal{M}_M \overset{T}{\neq} \mathcal{M}_{M'}$, i.e., there exist $\overline{\beta} \in Tr(\mathcal{M}_M)$ and $\overline{\beta}' \in Tr(\mathcal{M}_{M'})$ such that $\overline{\beta} \notin Tr(\mathcal{M}_{M'})$ and $\overline{\beta} \in Tr(\mathcal{M}_M)$. Also, since $\mathcal{M}_M \not\simeq^{\mathbf{L}} \mathcal{M}_{M'}$, we have that there exists a context C^L such that $C^L[\mathcal{M}_M]\Downarrow^{\mathbf{L}}$ and $C^L[\mathcal{M}_{M'}]\not\Downarrow^{\mathbf{L}}$ (or vice versa) – assume wlog $C^L[\mathcal{M}_M]\Downarrow^{\mathbf{L}}$ and $C^L[\mathcal{M}_{M'}]\not\Downarrow^{\mathbf{L}}$.

Thus, by Proposition 6.2:

$$\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_M]} \overset{\overline{\beta}''}{\Longrightarrow}{}^* \text{ HALT}$$
$$\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_{M'}]} \overset{\overline{\beta}'''}{\Longrightarrow}{}^* c \neq \text{HALT}$$

for some $\overline{\beta}''$ (ending in $\bullet$), c and for all $\overline{\beta}'''$ that can be observed.
Indeed, we can always write that $\overline{\beta}'' = \overline{\beta}_s \cdot \beta \cdot \overline{\beta}_e$ and $\overline{\beta}''' = \overline{\beta}_s \cdot \beta' \cdot \overline{\beta}'_e$ where:

- $\overline{\beta}_s$ is the longest (possibly empty) common prefix of the two traces
- β and $\beta' \neq \bullet$ are the first different observables – one of the two may be ε or, by Proposition 6.2, it may be $\beta = \bullet$
- $\overline{\beta}_e$ and $\overline{\beta}'_e$ are the (possibly empty) remainders of the two traces.

Thus, since $\overline{\beta}''$ and $\overline{\beta}'''$ are also observed under the same context C^L, they are distinguishing traces. □

The first two parameters of BUILDDEVICE – $joutd$ and $joutd'$ – are differentiating $\texttt{jmpOut!}(\cdot;\cdot)$ addresses (if any), as returned by the BUILDMEM (Algorithm 2). Parameters $\overline{\beta}$ and $\overline{\beta}'$ are distinguishing traces for $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ generated under the context C^L (cfr. Definition 6.10). Finally, $term$ (resp. $term'$) denotes whether $\mathcal{M}_M$ (resp. $\mathcal{M}_{M'}$) converges in a context with no interrupts after the last jump into protected mode.

The first two lines define the initial set of states, which will be a finite subset of $\mathbb{N}$ in the end, and the initial *empty* transition function.

Line 7 defines δ_L that records the last state that was added to the I/O device. At the beginning it is initialized to 0.

The algorithm then proceeds by iterating over all the observables in $\overline{\beta}_s$ (all the steps below also update Δ and δ_L, but we omit to state it explicitly):

Algorithm 3 Builds the device of the distinguishing context.

1: **procedure** BUILDDEVICE($joutd, joutd', \overline{\beta} = \beta_0 \cdots \beta_{n-1} \cdot \beta \cdot \overline{\beta}_e, \overline{\beta}' = \beta_0 \cdots \beta_{n-1} \cdot \beta' \cdot \overline{\beta}'_e, term, term', C^L$)
2: ▷ $joutd, joutd'$ are differentiating $\texttt{jmpOut!}(\cdot;\cdot)$ addresses, if any
3: ▷ $\overline{\beta}$ and $\overline{\beta}'$ are distinguishing traces generated by the context C^L
4: ▷ $term$ (resp. $term'$) denotes whether $\mathcal{M}_M$ (resp. $\mathcal{M}_{M'}$) converges in a context with no interrupts after the last jump into protected mode
5: $\Delta = \{0\}$
6: $\dot{\rightsquigarrow}_D = \emptyset$
7: $\delta_L = 0$ ▷ This variable keeps track of the last added device state.
8: **for** $i \in 0..n-1$ **do**
9: **if** $\beta_i = \texttt{jmpIn?}(\mathcal{R})$ **then**
10: $\Delta = \Delta \cup \{\delta_L + 1, \ldots, \delta_L + 17\}$
11: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L, wr(w), \delta_L) \mid w \in \mathit{Word}\}$
12: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L, rd(\texttt{A_JIN}), \delta_L + 1)\}$
13: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + 1, rd(\mathcal{R}[\texttt{sp}]), \delta_L + 2)\}$
14: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + 2, rd(\mathcal{R}[\texttt{sr}]), \delta_L + 3)\}$
15: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + i, rd(\mathcal{R}[\texttt{i}]), \delta_L + i + 1) \mid 3 \leq i \leq 15\}$
16: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + 16, rd(\mathcal{R}[\texttt{pc}]), \delta_L + 17)\}$
17: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + i, \epsilon, \delta_L + i) \mid 0 \leq i \leq 16\}$
18: $\delta_L = \delta_L + 17$
19: **else if** $\beta_i = \texttt{jmpOut!}(\Delta t; \mathcal{R})$ **then**
20: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L, \epsilon, \delta_L)\} \cup \{(\delta_L, wr(w), \delta_L) \mid w \in \mathit{Word}\}$
21: **end if**
22: **end for**
23: **if** $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \beta' = \texttt{jmpOut!}(\Delta t'; \mathcal{R}') \wedge (\exists \texttt{r}.\, \mathcal{R}[\texttt{r}] \neq \mathcal{R}'[\texttt{r}])$ **then**
24: **if** $\texttt{r} \neq \texttt{pc}$ **then**
25: $\Delta = \Delta \cup \{\delta_L + 1, \ldots, \delta_L + 4\}$
26: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L, rd(\texttt{A_RDIFF}), \delta_L + 1)\}$
27: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + 1, wr(\mathcal{R}[\texttt{pc}]), \delta_L + 2)\}$
28: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + 1, wr(\mathcal{R}'[\texttt{pc}]), \delta_L + 3)\}$
29: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + 2, rd(\texttt{A_HALT}), \delta_L + 4)\}$
30: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + 3, rd(\texttt{A_LOOP}), \delta_L + 4)\}$
31: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + i, \epsilon, \delta_L + i) \mid 0 \leq i \leq 3\}$
32: $\delta_L = \delta_L + 4$
33: **else**
34: $\Delta = \Delta \cup \{\delta_L + 1, \ldots, \delta_L + 3\}$
35: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L, wr(joutd), \delta_L + 1)\}$
36: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L, wr(joutd'), \delta_L + 2)\}$
37: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + 1, rd(\texttt{A_HALT}), \delta_L + 3)\}$
38: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + 2, rd(\texttt{A_LOOP}), \delta_L + 3)\}$
39: $\dot{\rightsquigarrow}_D = \dot{\rightsquigarrow}_D \cup \{(\delta_L + i, \epsilon, \delta_L + i) \mid 0 \leq i \leq 2\}$
40: $\delta_L = \delta_L + 3$
41: **end if**
42: continues ...

```
43:         ... continued
44:     else if β = jmpOut!(Δt; R) ∧ β' = jmpOut!(Δt'; R) ∧ Δt ≠ Δt' then
45:             ▷ Let D^L ⊢ INIT_{C[M_M]} =β̄_s=>* c_1 and D^L_I ⊢ c_1 =jmpOut!(Δt_I;R)=> c'_1.
46:             ▷ Let D^L ⊢ INIT_{C[M_M']} =β̄_s=>* c_2 and D^L_I ⊢ c_2 =jmpOut!(Δt'_I;R)=> c'_2.
47:         t = t'_1 − t_1
48:         t' = t'_2 − t_2
49:         Δ = Δ ∪ {δ_L + 1, ..., δ_L + max(t, t') + 1}
50:         ⇝_D = ⇝_D ∪ {(δ_L + min(t, t'), rd(A_HALT), δ_L + max(t, t') + 1)}
51:         ⇝_D = ⇝_D ∪ {(δ_L + max(t, t'), rd(A_LOOP), δ_L + max(t, t') + 1))}
52:         ⇝_D = ⇝_D ∪ {(δ_L + k, ϵ, δ_L + k + 1) | 0 ≤ k ≤ max(i, i')}
53:         δ_L = δ_L + max(t, t') + 1
54:     else if β = • ∧ β' = jmpOut!(Δt; R) then
55:         if term then
56:             Δ = Δ ∪ {δ_L + 1, ..., δ_L + 2}
57:             ⇝_D = ⇝_D ∪ {(δ_L, wr(A_EP), δ_L + 1)}
58:             ⇝_D = ⇝_D ∪ {(δ_L + 1, rd(A_HALT), δ_L + 2)}
59:             ⇝_D = ⇝_D ∪ {(δ_L, rd(A_LOOP), δ_L + 2)}
60:             ⇝_D = ⇝_D ∪ {(δ_L, wr(w), δ_L) | w ∈ Word \ {A_EP}}
61:             ⇝_D = ⇝_D ∪ {(δ_L + i, ϵ, δ_L + i) | 0 ≤ i ≤ 1}
62:             δ_L = δ_L + 2
63:         else
64:             Δ = Δ ∪ {δ_L + 1}
65:             ⇝_D = ⇝_D ∪ {(δ_L, rd(A_HALT), δ_L + 1)}
66:             ⇝_D = ⇝_D ∪ {(δ_L, wr(w), δ_L) | w ∈ Word}
67:             ⇝_D = ⇝_D ∪ {(δ_L, ϵ, δ_L)}
68:             δ_L = δ_L + 2
69:         end if
70:     else if β = jmpOut!(Δt; R) ∧ β' = ε then
71:             ▷ As the previous case, with term' in place of term.
72:     else
73:         return ⊥
74:     end if
75:     D = ⟨Δ, 0, ⇝_D⟩
76:     return D
77: end procedure
```

- *Case* $\beta_i = \beta'_i = \texttt{jmpIn?}(\mathcal{R})$. In this case we know that either this is the first observable or previous one was a $\texttt{jmpOut!}(\cdot;\cdot)$. Since the memory is obtained following Algorithm 2, we know that in both cases we reach the instruction `IN pc` (either at address `A_EP` or those of jumps out of protected mode), waiting for the next program counter (sometimes before that we perform a write, which shall be ignored). Thus, the device ignores any write operation and replies with `A_JIN` (Line 12). Then it starts to send the values of the registers in $\mathcal{R}$, to simulate in Sancus$^{\mathrm{H}}$ what happens in Sancus$^{\mathrm{L}}$ and to match the requests from the code. To help the intuition Figure 60a depicts how the transition function looks after the update (the solid black state denotes the new value of δ_L).

- *Case* $\beta_i = \beta'_i = \texttt{jmpOut!}(\Delta t; \mathcal{R})$. The device is simply updated with a loop on δ_L with action ϵ and ignores any write operation (so as to deal with $\mathcal{R}[\texttt{pc}] = joutd$ or $\mathcal{R}[\texttt{pc}] = joutd'$). Figure 60b pictorially represents this case.

Then, when $\overline{\beta}_s$ ends, the algorithm analyses β and β' and sets up the device to differentiate the two modules:

- *Case* $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \beta' = \texttt{jmpOut!}(\Delta t'; \mathcal{R}') \wedge (\exists \texttt{r}.\, \mathcal{R}[\texttt{r}] \neq \mathcal{R}'[\texttt{r}])$. In this case the differentiation is due to a register, and two further sub-cases may arise, depending on whether it is `pc`. If the register is `pc` then the device waits for the differentiating value for the context (that is executing code at $joutd$ and $joutd'$ by construction) and based on that value, it replies with either `A_HALT` (Line 37) or `A_LOOP` (Line 38). Instead, if the differentiation register is not `pc` then the code of the context is waiting for the next program counter and the context replies with `A_RDIFF`. From this address we find the code that sends the differentiating register and, based on that value, the device replies with either `A_HALT` (Line 29) or `A_LOOP` (Line 30). Figures 61a and 61b may help the intuition.

- *Case* $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \beta' = \texttt{jmpOut!}(\Delta t'; \mathcal{R}) \wedge \Delta t \neq \Delta t'$. This case is probably the most interesting since differentiation happens in Sancus$^{\mathrm{L}}$ due to timings. However, different timings in Sancus$^{\mathrm{L}}$ correspond to different timings in Sancus$^{\mathrm{H}}$ (as observed in proof of Proposition B.24), and the device is programmed to reply with either `A_HALT` (Line 50) or `A_LOOP` (Line 51) depending on the time value. Figure 61c intuitively depicts this situation.

- *Case* $\beta = \bullet \wedge \beta' = \texttt{jmpOut!}(\Delta t; \mathcal{R})$. In this case $\bullet$ may occur during an interrupt service routine. We then have two sub-cases, depending on whether the first module terminates when executed in a context with no interrupts after the last jump into protected mode or not (i.e., encoded by the value of $term$). When $term$ holds, the first module makes the CPU go through an exception handling configuration that jumps to `A_EP` and the device instructs the code to jump to `A_HALT` (Line 58), while for the second module the CPU jumps to any other location (`A_EP` is chosen to be different from any other jump out address!) and is instructed to jump to `A_LOOP` (Line 59). When $term$ does not hold, the first module diverges, while for the second module the CPU jumps to a location in unprotected code and it is instructed to jump to `A_HALT` (Line 65). Figures 61d and 61e may help the intuition.

- *Case* $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \wedge \beta' = \varepsilon$. Analogous to the previous case.

- *Otherwise.* No other cases may arise, as noted in Proposition B.23.

Finally, the algorithm returns a device with the set of states Δ, the initial state 0 and the transition function built as just explained.

Proposition B.23. *Let $\mathcal{M}_M \overset{T}{\neq} \mathcal{M}_{M'}$, $\overline{\beta}, \overline{\beta}'$ be distinguishing traces of $\mathcal{M}_M$ and $\mathcal{M}_{M'}$ originated by some context C^L and let $term$ and $term'$ be any pair of booleans, then $\mathcal{D} = \textsc{BuildDevice}(\overline{\beta}, \overline{\beta}', joutd, joutd', term, term', C^L) \neq \bot$ and $\mathcal{D}$ is an I/O device.*

Proof. We first show that BuildDevice never returns $\bot$ when $\overline{\beta}$ and $\overline{\beta}'$ are distinguishing traces. For that, let $\overline{\beta} = \overline{\beta}_s \cdot \beta \cdot \overline{\beta}_e$ and $\overline{\beta}' = \overline{\beta}_s \cdot \beta' \cdot \overline{\beta}'_e$, and note that the only cases for which $\bot$ is returned are the following:

- *Case* $\beta = \beta' = \bullet$. Since $\beta \neq \beta'$ by hypothesis, this case never happens.
- *Case* $\beta = \texttt{jmpOut!}(\Delta t; \mathcal{R})$ *and* $\beta' = \texttt{jmpIn?}(\mathcal{R}')$ *(or vice versa)*. This case never happens due to Proposition B.22.
- *Case* $\{\bullet, \texttt{jmpIn?}(\mathcal{R})\} \ni \beta \neq \beta' \in \{\bullet, \texttt{jmpIn?}(\mathcal{R}')\}$. Roughly, this means that the *same* context performed two different actions upon observation of the same trace ($\overline{\beta}_s$). Formally, we know by hypothesis that for the context $C^L = \langle \mathcal{M}_C, \mathcal{D}^L \rangle$

$$\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_M]} \overset{\overline{\beta}_s}{\Longrightarrow}{}^* c_1$$
$$\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_{M'}]} \overset{\overline{\beta}_s}{\Longrightarrow}{}^* c_2.$$

  with $c_1 \vdash_{mode} \texttt{UM}$ and $c_2 \vdash_{mode} \texttt{UM}$. Proposition B.20 guarantees that $c_1 \overset{U}{\approx} c_2$, thus by Proposition B.19 the same observable must originate from both c_1 and c_2, but that is against the hypothesis that $\beta \neq \beta'$.

Finally, it is easy to see that $\mathcal{D}$ returned by BuildDevice is an actual device. Indeed, its set of states Δ is finite (the algorithm always terminates in a finite number of steps and each step adds a finite number of state); its initial state 0 belongs to Δ; no $int?$ transitions are ever added and a single $rd(w)$ transition outgoes from any given state: thus the transition relation respects the definition of I/O devices. □

The following proposition states that the context built by joining together the results of the two algorithms above is a distinguishing one:

Proposition B.24. *Let $\mathcal{M}_M \overset{T}{\neq} \mathcal{M}_{M'}$; let $C^L = \langle \mathcal{M}_C, \mathcal{D}^L \rangle$; let*

$$\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_M]} \overset{\overline{\beta}_s}{\Longrightarrow}{}^* c'_1 \overset{\beta}{\Longrightarrow} c_1$$
$$\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_{M'}]} \overset{\overline{\beta}_s}{\Longrightarrow}{}^* c'_2 \overset{\beta'}{\Longrightarrow} c_2$$

be such that $\overline{\beta} = \overline{\beta}_s \cdot \beta \cdot \overline{\beta}_e$ and $\overline{\beta}' = \overline{\beta}_s \cdot \beta' \cdot \overline{\beta}_e$ distinguishing traces of $\mathcal{M}_M$ and $\mathcal{M}_{M'}$; and let

$$term \iff \mathcal{D}^L{}_{\mathcal{I}} \vdash c'_1 \rightarrow^* \text{HALT}$$
$$term' \iff \mathcal{D}^L{}_{\mathcal{I}} \vdash c'_2 \rightarrow^* \text{HALT}.$$

If $C^H = \langle \mathcal{M}_C, \mathcal{D} \rangle$, $(\mathcal{M}_C, joutd, joutd') = \textsc{BuildMem}(\overline{\beta}, \overline{\beta}')$, and $\mathcal{D} = \textsc{BuildDevice}(\overline{\beta}, \overline{\beta}', joutd, joutd', term, term')$, then $C^H[\mathcal{M}_M]\Downarrow^{\mathsf{H}}$ and $C^H[\mathcal{M}_{M'}]\not\Downarrow^{\mathsf{H}}$ (or vice versa).

Proof. Assume wlog that $C^L[\mathcal{M}_M]\Downarrow^{\mathbf{L}}$ and $C^L[\mathcal{M}_{M'}]\not\Downarrow^{\mathbf{L}}$. By Lemma 6.1

$$C^H[\mathcal{M}_M]\Downarrow^{\mathsf{H}} \iff C^H{}_{\mathcal{I}}[\mathcal{M}_M]\Downarrow^{\mathbf{L}} \quad \text{and} \quad C^H[\mathcal{M}_{M'}]\Downarrow^{\mathsf{H}} \iff C^H{}_{\mathcal{I}}[\mathcal{M}_{M'}]\Downarrow^{\mathbf{L}}$$

It suffices thus proving that $C^H{}_{\mathcal{I}}$ distinguishes $\mathcal{M}_M$ and $\mathcal{M}_{M'}$, i.e., $C^H{}_{\mathcal{I}}[\mathcal{M}_M]\Downarrow^{\mathbf{L}}$ and $C^H{}_{\mathcal{I}}[\mathcal{M}_{M'}]\not\Downarrow^{\mathbf{L}}$ or vice versa.

We show by induction on the length $2n+1$ of $\overline{\beta}_s$ that if

$$\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s}{}^{*} c_1'$$
$$\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}_s}{}^{*} c_2'$$

then $\exists \overline{\beta}_s'$ s.t.

$$D^H{}_{\mathcal{I}} \vdash \text{INIT}_{C^H{}_{\mathcal{I}}[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s'}{}^{*} c_3 \text{ and}$$
$$D^H{}_{\mathcal{I}} \vdash \text{INIT}_{C^H{}_{\mathcal{I}}[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}_s'}{}^{*} c_4 \text{ with } \overline{\beta}_s' \approx \overline{\beta}_s \text{ (see Definition B.2).}$$

Note that the length of $\overline{\beta}_s$ must be odd as a consequence of Propositions B.19 and B.20 and no $\bullet$ appears in it since otherwise it would mean that $\overline{\beta} = \overline{\beta}'$.

- *Case* $n = 0$. Then, $\overline{\beta}_s$ is $\texttt{jmpIn?}(\mathcal{R})$. Thus, Algorithm 2 guarantees that the current instruction is `IN pc` (at address `A_EP`) and its execution leads to address `A_JIN` (by Algorithm 3) and the same $\texttt{jmpIn?}(\mathcal{R})$ is observed starting from both $\text{INIT}_{C^H{}_{\mathcal{I}}[\mathcal{M}_M]}$ and $\text{INIT}_{C^H{}_{\mathcal{I}}[\mathcal{M}_{M'}]}$ and also $\overline{\beta}_s' \approx \overline{\beta}_s$.
- *Case* $n = n' + 1$. If

$$\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s''}{}^{*} c_1'' \wedge \mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}_s''}{}^{*} c_2''$$
$$\Downarrow$$
$$D^H{}_{\mathcal{I}} \vdash \text{INIT}_{C^H{}_{\mathcal{I}}[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s'''}{}^{*} c_3' \wedge D^H{}_{\mathcal{I}} \vdash \text{INIT}_{C^H{}_{\mathcal{I}}[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}_s'''}{}^{*} c_4' \ \wedge \overline{\beta}_s''' \approx \overline{\beta}_s'' \text{ (IHP)}$$

then

$$\mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s''}{}^{*} c_1'' \xRightarrow{\overline{\beta}''}{}^{*} c_1' \wedge \mathcal{D}^L \vdash \text{INIT}_{C^L[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}''}{}^{*} c_2'' \xRightarrow{\overline{\beta}''}{}^{*} c_2'$$
$$\Downarrow$$
$$D^H{}_{\mathcal{I}} \vdash \text{INIT}_{C^H{}_{\mathcal{I}}[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s'''}{}^{*} c_3' \xRightarrow{\overline{\beta}'''}{}^{*} c_3 \ \wedge \ D^H{}_{\mathcal{I}} \vdash \text{INIT}_{C^H{}_{\mathcal{I}}[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}_s'''}{}^{*} c_4' \xRightarrow{\overline{\beta}'''}{}^{*} c_4$$
$$\wedge \overline{\beta}_s''' \cdot \overline{\beta}''' \approx \overline{\beta}_s'' \cdot \overline{\beta}''.$$

Note that it must be that $\overline{\beta}'' = \texttt{jmpOut!}(\Delta t; \mathcal{R}) \cdot \texttt{jmpIn?}(\mathcal{R}')$ by Proposition B.22 and because we never observe $\bullet$ in the common prefix. By (IHP) and Proposition B.11 we have $c_1'' \overset{P}{\approx} c_3'$ and $c_2'' \overset{P}{\approx} c_4'$. Thus, by Propositions B.8 and B.9, it must be that $\texttt{jmpOut!}(\Delta t'; \mathcal{R})$ is observed when starting in c_3' and $\texttt{jmpOut!}(\Delta t''; \mathcal{R})$ is observed when starting in c_4' (for some $\Delta t'$ and $\Delta t''$).

By definition of coarse-grained traces, each of the computations above is generated by fine-grained trace in the form (we write _ to denote a generic configuration):

$$\mathcal{D}^L \vdash _ \xRightarrow{\texttt{jmpIn?}(\mathcal{R}'')} c_1'' = c_1^{(0)} \xrightarrow{\alpha_1^{(0)}} \cdots \xrightarrow{\alpha_1^{(n_1-1)}} c_1^{(n_1)} \xRightarrow{\texttt{jmpOut!}(k_1^{(n_1)};\mathcal{R})} c^{(n_1)+1} \xRightarrow{\xi\cdots\xi\texttt{jmpIn?}(\mathcal{R}')}{}^{*} c_1'$$

$$\mathcal{D}^L \vdash _ \xRightarrow{\texttt{jmpIn?}(\mathcal{R}'')} c_2'' = c_2^{(0)} \xrightarrow{\alpha_2^{(0)}} \cdots \xrightarrow{\alpha_2^{(n_2-1)}} c_2^{(n_2)} \xRightarrow{\texttt{jmpOut!}(k_2^{(n_2)};\mathcal{R})} c^{(n_2)+1} \xRightarrow{\xi\cdots\xi\texttt{jmpIn?}(\mathcal{R}')}{}^{*} c_2'$$

$$D^H{}_I \vdash _ \xRightarrow{\texttt{jmpIn?}(\mathcal{R}'')} c_3' = c_3^{(0)} \xrightarrow{\alpha_3^{(0)}} \cdots \xrightarrow{\alpha_3^{(n_3-1)}} c_3^{(n_3)} \xRightarrow{\texttt{jmpOut!}(k_3^{(n_3)};\mathcal{R})} c_3^{(n_3+1)}$$

$$D^H{}_I \vdash _ \xRightarrow{\texttt{jmpIn?}(\mathcal{R}'')} c_4' = c_4^{(0)} \xrightarrow{\alpha_4^{(0)}} \cdots \xrightarrow{\alpha_4^{(n_4-1)}} c_4^{(n_4)} \xRightarrow{\texttt{jmpOut!}(k_4^{(n_4)};\mathcal{R})} c_4^{(n_4+1)}.$$

Thus, due to Proposition 6.1 and by hypothesis, it holds that $\Delta t = \sum_{i=0}^{n_1} \gamma(c_1^{(i)}) + (11 + \texttt{MAX_TIME}) \cdot |\mathbb{I}_{\alpha_1^{(0)}\cdots\alpha_1^{(n_1)}}| = \sum_{i=0}^{n_2} \gamma(c_2^{(i)}) + (11 + \texttt{MAX_TIME}) \cdot |\mathbb{I}_{\alpha_2^{(0)}\cdots\alpha_2^{(n_2)}}|$. Also, since by (IHP) and Propositions B.19 and B.20 it follows that $c_1^{(0)} = c_1'' \overset{U}{\approx} c_2'' = c_2^{(0)}$, we know $|\mathbb{I}_{\alpha_1^{(0)}\cdots\alpha_1^{(n_1)}}| = |\mathbb{I}_{\alpha_2^{(0)}\cdots\alpha_2^{(n_2)}}|$ (by Proposition B.18) and thus $\sum_{i=0}^{n_1} \gamma(c_1^{(i)}) = \sum_{i=0}^{n_2} \gamma(c_2^{(i)})$. Moreover, by (IHP) and Proposition B.11, we get $c_1^{(0)} = c_1'' \overset{P}{\approx} c_3' = c_3^{(0)}$ and $c_2^{(0)} = c_2'' \overset{P}{\approx} c_4' = c_4^{(0)}$. Now, as a consequence of Propositions B.3, B.8 and B.9 we know that $\Delta t' = \sum_{i=0}^{n_3} \gamma(c_3^{(i)}) = \sum_{i=0}^{n_1} \gamma(c_1^{(i)}) = \sum_{i=0}^{n_2} \gamma(c_2^{(i)}) = \sum_{i=0}^{n_3} \gamma(c_3^{(i)}) = \Delta t''$. By (IHP) and since the first observable after c_3' and c_4' is the same, by Proposition B.20 it follows $c_3^{(n_3+1)} \overset{U}{\approx} c_4^{(n_4+1)}$. Thus, due to Proposition B.19, we get that the same coarse-grained observable $\texttt{jmpIn?}(\mathcal{R}''')$ is observed after $c_3^{(n_3+1)}$ and $c_4^{(n_4+1)}$. Finally, $\mathcal{R}'''$ is equal to $\mathcal{R}'$ since after any $\texttt{jmpOut!}(\cdot;\cdot)$ a `IN pc` instruction is executed and its execution leads to address `A_JIN` (by Algorithm 3) that performs $\texttt{jmpIn?}(\mathcal{R})$, and the thesis follows.

Since we proved that

$$D^H{}_I \vdash \text{INIT}_{C^H{}_I[\mathcal{M}_M]} \xRightarrow{\overline{\beta}_s'}{}^{*} c_3 \text{ and}$$

$$D^H{}_I \vdash \text{INIT}_{C^H{}_I[\mathcal{M}_{M'}]} \xRightarrow{\overline{\beta}_s'}{}^{*} c_4$$

we also have that $c_3 \overset{U}{\approx} c_4$ by Propositions B.19 and B.20.

Let $D^H{}_I \vdash c_3 \xRightarrow{\overline{\beta}_3}{}^{*} c_3''$ and $D^H{}_I \vdash c_4 \xRightarrow{\overline{\beta}_4}{}^{*} c_4''$, with $\overline{\beta}_3$ and $\overline{\beta}_4$ either empty or made of a single observable (either $\bullet$ or $\texttt{jmpOut!}(\cdot;\cdot)$, since no difference cannot be observed upon $\texttt{jmpIn?}(\cdot)$ as observed above). By exhaustive cases on β and β' we have:

- *Case* $\beta = \bullet$ *and* $\beta' = \texttt{jmpOut!}(\Delta t'''; \mathcal{R}'')$. Note that, since *term* $\iff \mathcal{D}^L{}_I \vdash c_1' \rightarrow^{*}$ HALT and $c_1' \overset{P}{\approx} c_3$ (by Propositions B.10 and B.11), we get *term* $\iff \mathcal{D}^H{}_I \vdash c_3 \rightarrow^{*}$ HALT by Proposition B.8 and since neither $\mathcal{D}^L{}_I$ nor $\mathcal{D}^H{}_I$ raise any interrupt. Thus, by definition of $\mathcal{D}^L$ (cfr. Algorithm 3) the context C^H distinguishes the two modules.

- *Case* $\beta = \texttt{jmpOut!}(\Delta t'''; \mathcal{R}'')$ *and* $\beta' = \varepsilon$. Similar to the previous case (with *term'* in place of *term*).

- *Case* $\beta = \texttt{jmpOut!}(\Delta t'''; \mathcal{R}'')$ *and* $\beta' = \texttt{jmpOut!}(\Delta t'''; \mathcal{R}''')$ *with* $\mathcal{R}'' \neq \mathcal{R}'''$. Since $c_1' \overset{P}{\approx} c_3$ and $c_2' \overset{P}{\approx} c_4$, it must be that $\overline{\beta}_3 = \texttt{jmpOut!}(\Delta t^{v}; \mathcal{R}'')$ and $\overline{\beta}_4 = \texttt{jmpOut!}(\Delta t^{vi}; \mathcal{R}'')$. Thus, by Algorithms 2 and 3, C^H distinguishes the two modules.
- *Case* $\beta = \texttt{jmpOut!}(\Delta t'''; \mathcal{R}'')$ *and* $\beta' = \texttt{jmpOut!}(\Delta t^{iv}; \mathcal{R}'')$. In this case it holds that $\overline{\beta}_3 = \texttt{jmpOut!}(\Delta t^{v}; \mathcal{R}'')$ and $\overline{\beta}_4 = \texttt{jmpOut!}(\Delta t^{vi}; \mathcal{R}'')$ with the same timings of the instructions (by Proposition 6.1). Since $c_3 \overset{U}{\approx} c_4$, the two times must differ one from each other otherwise, by the contrapositive of Proposition B.17, we would get $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$. Again, by definition of Algorithms 2 and 3, one computation converges and one diverges, hence C^H distinguishes the two modules.

□

B.9 PROOFS AND ADDITIONAL DEFINITIONS OF SECTION 6.6

Proposition 6.4. *Let c and c' be configurations such that $c, c' \vdash_{mode} \texttt{UM}$. If $c \overset{U}{\approx} c'$ then $c \overset{L}{=} c'$.*

Proof. Since $c, c' \vdash_{mode} \texttt{UM}$, the proposition follows directly from Definitions 6.8 and 6.11. □

Lemma 6.9. *If $\mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'}$ then $\mathcal{M}_M \approx_{\mathrm{IS}} \mathcal{M}_{M'}$.*

Proof. Assuming contextual equivalence in Sancus$^{\mathrm{L}}$ and that:

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c \rightarrow \mathrm{HALT} \;\wedge\; \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \rightarrow^* c' \rightarrow \mathrm{HALT},$$

our goal is to prove that $c \overset{L}{=} c'$. From contextual equivalence it follows that $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$. By Lemma 6.5 we also know that for some c'':

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c \;\wedge\; \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c''.$$

Proposition B.21 and Proposition B.20 guarantee that $c \overset{U}{\approx} c''$. Then, since $c \rightarrow \mathrm{HALT}$, it must be $c'' \rightarrow \mathrm{HALT}$. For that and by determinism of the operational semantics of Sancus$^{\mathrm{L}}$ we have that $c' = c''$ and $c \overset{U}{\approx} c'$, which by Proposition 6.4 implies $c \overset{L}{=} c'$. □

Theorem 6.3. *If $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$ then $\mathcal{M}_M \approx_{\mathrm{IS}} \mathcal{M}_{M'}$.*

Proof. Since $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$, by Theorem 6.2 we also have that $\mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'}$ and Lemma 6.9 concludes the proof. □

Theorem 6.4.

1. *If $\mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'}$, then $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$; and*
2. *if $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$, then $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$.*

Proof.

1. Lemma 6.9 guarantees that $\mathcal{M}_M \approx_{\mathrm{IS}} \mathcal{M}_{M'}$. We now set out to show that $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$ is implied by $\mathcal{M}_M \approx_{\mathrm{IS}} \mathcal{M}_{M'}$ in our setting. Indeed, by definition of SSNI we can assume (wlog) that $\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \rightarrow^* c \rightarrow \mathrm{HALT}$, i.e., $C[\mathcal{M}_M]{\Downarrow}^{\mathsf{L}}$. By hypothesis it also follows that $C[\mathcal{M}_{M'}]{\Downarrow}^{\mathsf{L}}$. For that and by definition of ISNI, it then follows $\mathcal{M}_M \approx_{\mathrm{SS}} \mathcal{M}_{M'}$.

2. By definition of SSNI it follows that for any C if $C[\mathcal{M}_M]{\Downarrow}^{\mathsf{H}}$, then $C[\mathcal{M}_{M'}]{\Downarrow}^{\mathsf{H}}$ and vice versa, i.e., $C[\mathcal{M}_M]{\Downarrow}^{\mathsf{H}} \iff C[\mathcal{M}_{M'}]{\Downarrow}^{\mathsf{H}}$ which coincides with the definition of $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$. □

Proposition B.25.

1. *If $\mathcal{D} \vdash c \overset{\overline{\beta}}{\Longrightarrow}{}^* c'$ then $\exists! t, K.\ \mathcal{D} \vdash c \twoheadrightarrow^t_K c' \wedge \overline{\beta} \propto K$.*

2. *If $\mathcal{D} \vdash c \twoheadrightarrow^t_K c'$ then $\exists \overline{\beta}.\ \mathcal{D} \vdash c \overset{\overline{\beta}}{\Longrightarrow}{}^* c' \wedge \overline{\beta} \propto K$.*

where

$$\overline{\beta} \propto K \text{ iff } |\overline{\beta}| = \begin{cases} K & \overline{\beta} \neq \overline{\beta}' \cdot \bullet \\ K+1 & o.w. \end{cases}$$

Proof.

1. By determinism, there is a single computation χ from c to c' generating $\overline{\beta}$. From uniqueness of χ and by Definition 6.15, one gets existence and uniqueness of t and K.

2. Directly follows from Definition 6.15 and Figure 58. □

Lemma 6.10.

1. *If $\mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'}$ then $\mathcal{M}_M \approx_{\mathrm{SSS}} \mathcal{M}_{M'}$; and*

2. *if $\mathcal{M}_M \approx_{\mathrm{SSS}} \mathcal{M}_{M'}$ then $\mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$.*

Proof.

1. Assuming contextual equivalence in $\mathbf{Sancus}^{\mathsf{L}}$ and that:

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \twoheadrightarrow^t_K c,$$

our goal is to prove that $c \overset{L}{=} c'$. From contextual equivalence it follows that $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$. By Lemma 6.5 we also know that for some c'':

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c \implies \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c''.$$

Proposition B.21 and Proposition B.20 guarantee that $c \overset{U}{\approx} c''$. By Proposition B.25.(1) there exist unique t and K such that

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \twoheadrightarrow^t_K c$$

and

$$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \twoheadrightarrow^t_K c''$$

thus by determinism of operational semantics of $\mathbf{Sancus}^{\mathsf{L}}$, we have that $c'' = c'$ and $c \overset{U}{\approx} c'$, which by Proposition 6.4 implies $c \overset{L}{=} c'$.

2. Suppose that $C[\mathcal{M}_M]\Downarrow^{\mathsf{H}}$ and $C[\mathcal{M}_{M'}]\not\Downarrow^{\mathsf{H}}$. But then they cannot be $\mathcal{M}_M \approx_{\mathsf{SSS}} \mathcal{M}_{M'}$ (since HALT is in relation just with itself), which contradicts the hypothesis. □

Theorem 6.5. *The following relations are equivalent:*

$$1.\ \mathcal{M}_M \overset{WT}{=} \mathcal{M}_{M'} \qquad 2.\ \mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$$
$$3.\ \mathcal{M}_M \simeq^{\mathsf{L}} \mathcal{M}_{M'} \qquad 4.\ \mathcal{M}_M \simeq^{\mathsf{H}} \mathcal{M}_{M'}$$

Proof. We only prove $(1) \iff (2)$; The other equivalences follow from Theorem 6.2.

- $(1) \Leftarrow (2)$. Since $\mathcal{M}_M \overset{T}{=} \mathcal{M}_{M'}$, by Lemma 6.5 we know that:

  $$\mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_M]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c \iff \mathcal{D} \vdash \mathrm{INIT}_{C[\mathcal{M}_{M'}]} \overset{\overline{\beta}}{\Longrightarrow}{}^* c'.$$

  Thus, $\forall C. WTr(C[\mathcal{M}_M]) = WTr(C[\mathcal{M}_{M'}])$ as requested.

- $(1) \Rightarrow (2)$. Easy. □

BIBLIOGRAPHY

[1] ARM. *Arm Architecture Reference Manual Supplement Morello for A-profile Architecture*. https://developer.arm.com/documentation/ddi0606/ah/?lang=en.

[2] Martín Abadi. "Protection in Programming-Language Translations." *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*. Ed. by Jan Vitek and Christian Damsgaard Jensen. Vol. 1603. Lecture Notes in Computer Science. Springer, 1999, pp. 19–34.

[3] Martín Abadi. "Secrecy by Typing in Security Protocols." *J. ACM* 46.5 (1999), pp. 749–786.

[4] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. "A Core Calculus of Dependency." *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. 1999, pp. 147–160. doi: 10.1145/292540.292555.

[5] Martín Abadi and Bruno Blanchet. "Secrecy types for asymmetric communication." *Theor. Comput. Sci.* 298.3 (2003), pp. 387–415.

[6] Martín Abadi and Andrew D. Gordon. "A Calculus for Cryptographic Protocols: The Spi Calculus." *CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1-4, 1997*. 1997, pp. 36–47.

[7] Martín Abadi and Gordon D. Plotkin. "On Protection by Layout Randomization." *ACM Trans. Inf. Syst. Secur.* 15.2 (2012), 8:1–8:29.

[8] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. "When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise." *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2018, pp. 1351–1368.

[9] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Catalin Hritcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. "Trace-Relating Compiler Correctness and Secure Compilation." *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. 2020, pp. 1–28. doi: 10.1007/978-3-030-44914-8_1.

[10] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. "Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation." *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. 2019, pp. 256–271.

[11] Carmine Abate and Matteo Busi. "The Fox and the Hound: Comparing Fully Abstract and Robust Compilation." *CoRR* abs/2006.14969 (2020). arXiv: 2006.14969. url: https://arxiv.org/abs/2006.14969.

[12] Shail Aditya and Rishiyur S. Nikhil. “Incremental Polymorphism.” *Functional Programming Languages and Computer Architecture, 5th ACM, Proceedings*. Ed. by John Hughes. Vol. 523. LNCS. Springer, 1991, pp. 379–405.

[13] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. “Secure Compilation to Modern Processors.” *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. Ed. by Stephen Chong. IEEE Computer Society, 2012, pp. 171–185.

[14] Amal Ahmed and Matthias Blume. “Typed closure conversion preserves observational equivalence.” *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. 2008, pp. 157–168.

[15] Amal Ahmed and Matthias Blume. “An equivalence-preserving CPS translation via multi-language semantics.” *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 2011, pp. 431–444.

[16] Bowen Alpern and Fred B. Schneider. “Defining Liveness.” *Inf. Process. Lett.* 21.4 (1985), pp. 181–185.

[17] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Catalin Hritcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. “A verified information-flow architecture.” *Journal of Computer Security* 24.6 (2016), pp. 689–734.

[18] Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. “Micro-Policies: Formally Verified, Tag-Based Security Monitors.” *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 813–830.

[19] Steven Arzt and Eric Bodden. “Reviser: Efficiently Updating IDE-/IFDS-Based Data-Flow Analyses in Response to Incremental Program Changes.” *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. New York, NY, USA: ACM, 2014, 288–298. ISBN: 9781450327565. DOI: 10.1145/2568225.2568243.

[20] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. “Formal verification of a constant-time preserving C compiler.” *Proc. ACM Program. Lang.* 4.POPL (2020), 7:1–7:30. DOI: 10.1145/3371075.

[21] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. “Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time".” *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. 2018, pp. 328–343.

[22] Gilles Barthe, Tamara Rezk, and Amitabh Basu. “Security types preserving compilation.” *Computer Languages, Systems & Structures* 33.2 (2007), pp. 35–59.

[23] Gilles Barthe, Tamara Rezk, and Ando Saabas. “Proof Obligations Preserving Compilation.” *Formal Aspects in Security and Trust, Third International Workshop, FAST 2005, Newcastle upon Tyne, UK, July 18-19, 2005, Revised Selected Papers*. 2005, pp. 112–126.

[24] Massimo Bartoletti, Gabriele Costa, Pierpaolo Degano, Fabio Martinelli, and Roberto Zunino. "Securing Java with Local Policies." *J. Object Technol.* 8.4 (2009), pp. 5–32. DOI: 10.5381/jot.2009.8.4.a1.

[25] Massimo Bartoletti, Gabriele Costa, Pierpaolo Degano, Fabio Martinelli, and Roberto Zunino. "Securing Java with Local Policies." *J. Object Technol.* 8.4 (2009), pp. 5–32. DOI: 10.5381/jot.2009.8.4.a1.

[26] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. "Enforcing Secure Service Composition." *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*. 2005, pp. 211–223. DOI: 10.1109/CSFW.2005.17.

[27] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. "History-Based Access Control with Local Policies." *Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. 2005, pp. 316–332. DOI: 10.1007/978-3-540-31982-5_20.

[28] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. "Policy framings for access control." *Proceedings of the POPL 2005 Workshop on Issues in the Theory of Security, WITS 2005, Long Beach, California, USA, January 10-11, 2005*. 2005, pp. 5–11. DOI: 10.1145/1045405.1045407.

[29] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. "Types and Effects for Secure Service Orchestration." *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. 2006, pp. 57–69. DOI: 10.1109/CSFW.2006.31.

[30] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. "Types and Effects for Resource Usage Analysis." *Foundations of Software Science and Computational Structures, 10th International Conference, FOSSACS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24-April 1, 2007, Proceedings*. 2007, pp. 32–47. DOI: 10.1007/978-3-540-71389-0_4.

[31] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. "Semantics-Based Design for Secure Web Services." *IEEE Trans. Software Eng.* 34.1 (2008), pp. 33–49. DOI: 10.1109/TSE.2007.70740.

[32] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. "Local policies for resource usage analysis." *ACM Trans. Program. Lang. Syst.* 31.6 (2009), 23:1–23:43. DOI: 10.1145/1552309.1552313.

[33] Mihai Bazon. *UglifyJS - JavaScript parser, compressor, minifier written in JS*. http://lisperator.net/uglifyjs/. Online; last access Mar 2021.

[34] Jan A. Bergstra and Jan Willem Klop. "Algebra of Communicating Processes with Abstraction." *Theor. Comput. Sci.* 37 (1985), pp. 77–121. DOI: 10.1016/0304-3975(85)90088-X.

[35] Daniel J. Bernstein. *Cache-timing attacks on AES*. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf. Online; last access Mar 2021. 2005.

[36] Sam Blackshear, Dino Di Stefano, Martino Luca, Peter O'Hearn, and Jules Villard. *Finding inter-procedural bugs at scale with Infer static analyzer*. 2017. URL: https://code.facebook.com/posts/1537144479682247/finding-inter-procedural-bugs-at-scale-with-infer-static-analyzer/.

[37] Sandrine Blazy and Roberto Giacobazzi. "Towards a formally verified obfuscating compiler." *SSP 2012 - 2nd ACM SIGPLAN Software Security and Protection Workshop*. 2012.

[38] Sandrine Blazy and Rémi Hutin. "Formal verification of a program obfuscation based on mixed Boolean-arithmetic expressions." *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2019, pp. 196–208. DOI: 10.1145/3293880.3294103.

[39] Sandrine Blazy and Alix Trieu. "Formal verification of control-flow graph flattening." *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*. 2016, pp. 176–187. DOI: 10.1145/2854065.2854082.

[40] Chiara Bodei, Pierpaolo Degano, Letterio Galletta, and Francesco Salvatori. "Linguistic Mechanisms for Context-Aware Security." *Theoretical Aspects of Computing - ICTAC 2014 - 11th International Colloquium, Bucharest, Romania, September 17-19, 2014. Proceedings*. 2014, pp. 61–79. DOI: 10.1007/978-3-319-10882-7_5.

[41] Chiara Bodei, Pierpaolo Degano, Letterio Galletta, and Francesco Salvatori. "Context-aware security: Linguistic mechanisms and static analysis." *Journal of Computer Security* 24.4 (2016), pp. 427–477.

[42] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. "Control Flow Analysis for the pi-calculus." *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*. 1998, pp. 84–98.

[43] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. "Static Analysis of Processes for No and Read-Up nad No Write-Down." *Foundations of Software Science and Computation Structure, Second International Conference, FoSSaCS'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*. 1999, pp. 120–134.

[44] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. "Static Analysis for the pi-Calculus with Applications to Security." *Inf. Comput.* 168.1 (2001), pp. 68–92.

[45] William J. Bowman and Amal Ahmed. "Noninterference for free." *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 2015, pp. 101–113. DOI: 10.1145/2784731.2784733.

[46] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.* Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 991–1008. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[47] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes." *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019, pp. 1741–1758. DOI: 10.1145/3319535.3363206.

[48] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control." *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. ACM, 2017, 4:1–4:6. DOI: 10.1145/3152701.3152706.

[49] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. "Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution." *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 1041–1056. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck.

[50] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. "A Semantics for Disciplined Concurrency in COP." *Proceedings of the 17th Italian Conference on Theoretical Computer Science, Lecce, Italy, September 7-9, 2016*. 2016, pp. 177–189. URL: http://ceur-ws.org/Vol-1720/full13.pdf.

[51] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. "Robust Declassification by Incremental Typing." *Foundations of Security, Protocols, and Equational Reasoning - Essays Dedicated to Catherine A. Meadows*. Ed. by Joshua D. Guttman, Carl E. Landwehr, José Meseguer, and Dusko Pavlovic. Vol. 11565. Lecture Notes in Computer Science. Springer, 2019, pp. 54–69. DOI: 10.1007/978-3-030-19052-1_6.

[52] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. "Translation Validation for Security Properties." *3rd Workshop on Principles of Secure Compilation, PriSC 2019, Cascais, Portugal, January 13, 2019*. 2019. URL: https://arxiv.org/abs/1901.05082.

[53] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. "Using Standard Typing Algorithms Incrementally." *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*. 2019, pp. 106–122. DOI: 10.1007/978-3-030-20652-9_7.

[54] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. "Control-flow Flattening Preserves the Constant-Time Policy." *Proceedings of the Fourth Italian Conference on Cyber Security, Ancona, Italy, February 4th to 7th, 2020*. Ed. by Michele Loreti and Luca Spalazzi. Vol. 2597. 2020, pp. 82–92. URL: http://ceur-ws.org/Vol-2597/paper-08.pdf.

[55] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. "Mechanical incrementalization of typing algorithms." *Science of Computer Programming* 208 (2021). ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102657.

[56] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. "Secure Translation Validation: Effective Preservation of Robust Safety Properties." *Under review* (2021).

[57] Matteo Busi and Letterio Galletta. "A Brief Tour of Formally Secure Compilation." *Proceedings of the Third Italian Conference on Cyber Security, Pisa, Italy, February 13-15, 2019*. Ed. by Pierpaolo Degano and Roberto Zunino. Vol. 2315. 2019. URL: http://ceur-ws.org/Vol-2315/paper03.pdf.

[58] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. "Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors." *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. 2020, pp. 262–276. DOI: 10.1109/CSF49147.2020.00026.

[59] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. "Securing Interruptible Enclaved Execution on Small Microprocessors." *Under review* (2021).

[60] *CVE-2009-1897*. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1897. Accessed: 17 Nov 2018.

[61] *CWE-14*. https://cwe.mitre.org/data/definitions/14.html. Accessed: 17 Nov 2018.

[62] *CWE-733*. https://cwe.mitre.org/data/definitions/733.html. Accessed: 17 Nov 2018.

[63] Cristiano Calcagno, Dino Di Stefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. "Moving Fast with Software Verification." *NASA Formal Methods*. Ed. by Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. 2015, pp. 3–11. DOI: 10.1007/978-3-319-17524-9_1.

[64] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses." *28th USENIX Security Symposium, USENIX Security 2019*. 2019.

[65] Luca Cardelli. "Type Systems." *The Computer Science and Engineering Handbook*. 1997, pp. 2208–2236.

[66] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. "FaCT: a DSL for timing-sensitive computation." *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Kathryn S. McKinley and Kathleen Fisher. ACM, 2019, pp. 174–189. URL: https://doi.org/10.1145/3314221.3314605.

[67] Stefano Ceri, Georg Gottlob, and Letizia Tanca. "What you Always Wanted to Know About Datalog (And Never Dared to Ask)." *IEEE Trans. Knowl. Data Eng.* 1.1 (1989), pp. 146–166. DOI: 10.1109/69.43410.

[68] Juan Chen, Chris Hawblitzel, Frances Perry, Michael Emmi, Jeremy Condit, Derrick Coetzee, and Polyvios Pratikakis. "Type-preserving compilation for large-scale optimizing object-oriented compilers." *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 2008, pp. 183–192. DOI: 10.1145/1375581.1375604.

[69] Adam Chlipala. "A certified type-preserving compiler from lambda calculus to assembly language." *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 2007, pp. 54–65. DOI: 10.1145/1250734.1250742.

[70] Adam Chlipala. *Formal reasoning about programs*. 2017.

[71] Matteo Cimini and Jeremy G. Siek. "The gradualizer: a methodology and algorithm for generating gradual type systems." *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 443–455. DOI: 10.1145/2837614.2837632.

[72] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties." *Journal of Computer Security* 18.6 (2010), pp. 1157–1210.

[73] Ruan de Clercq, Frank Piessens, Dries Schellekens, and Ingrid Verbauwhede. "Secure interrupts on low-end microcontrollers." *IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2014, Zurich, Switzerland, June 18-20, 2014*. IEEE Computer Society, 2014, pp. 147–152. DOI: 10.1109/ASAP.2014.6868649.

[74] George Coker, Joshua D. Guttman, Peter Loscocco, Amy L. Herzog, Jonathan K. Millen, Brian O'Hanlon, John D. Ramsdell, Ariel Segall, Justin Sheehy, and Brian T. Sniffen. "Principles of remote attestation." *Int. J. Inf. Sec.* 10.2 (2011), pp. 63–81.

[75] Christian S. Collberg and Jasvir Nagra. *Surreptitious Software - Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2010. ISBN: 978-0-321-54925-9.

[76] Christian Collberg. *The Tigress C Diversifier/Obfuscator*. https://tigress.wtf/. Online; last access Mar 2021.

[77] Keith Daniel Cooper. "Interprocedural Data Flow Analysis in a Programming Environment." PhD thesis. Rice University, 1983.

[78] Gabriele Costa, Pierpaolo Degano, and Fabio Martinelli. "Modular plans for secure service composition." *J. Comput. Secur.* 20.1 (2012), pp. 81–117. DOI: 10.3233/JCS-2011-0430.

[79] Victor Costan and Srinivas Devadas. "Intel SGX Explained." *IACR Cryptology ePrint Archive* 2016 (2016), p. 86. URL: http://eprint.iacr.org/2016/086.

[80] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[81] Vijay D'Silva, Mathias Payer, and Dawn Xiaodong Song. "The Correctness-Security Gap in Compiler Optimization." *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*. IEEE Computer Society, 2015, pp. 73–87.

[82] Pierpaolo Degano, Gian Luigi Ferrari, and Letterio Galletta. "A Two-Component Language for COP." *Proceedings of 6th International Workshop on Context-Oriented Programming, COP@ECOOP 2014, Uppsala, Sweden, July 28 - August 1, 2014*. 2014, 6:1–6:7. DOI: 10.1145/2637066.2637072.

[83] Pierpaolo Degano, Gian Luigi Ferrari, and Letterio Galletta. "A Two-Component Language for Adaptation: Design, Semantics and Program Analysis." *IEEE Trans. Software Eng.* 42.6 (2016), pp. 505–529. DOI: 10.1109/TSE.2015.2496941.

[84] Pierpaolo Degano, Gian Luigi Ferrari, Letterio Galletta, and Gianluca Mezzetti. "Types for Coordinating Secure Behavioural Variations." *Coordination Models and Languages - 14th International Conference, COORDINATION 2012, Stockholm, Sweden, June 14-15, 2012. Proceedings*. 2012, pp. 261–276. DOI: 10.1007/978-3-642-30829-1_18.

[85] Pierpaolo Degano, Gian Luigi Ferrari, Letterio Galletta, and Gianluca Mezzetti. "Typing Context-Dependent Behavioural Variation." *Proceedings Fifth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2012, Tallinn, Estonia, 31 March 2012*. 2012, pp. 28–33. DOI: 10.4204/EPTCS.109.5.

[86] Pierpaolo Degano, Francesca Levi, and Chiara Bodei. "Safe Ambients: Control Flow Analysis and Security." *Advances in Computing Science - ASIAN 2000, 6th Asian Computing Science Conference, Penang, Malaysia, November 25-27, 2000, Proceedings*. 2000, pp. 199–214.

[87] Chaoqiang Deng and Kedar S. Namjoshi. "Securing a Compiler Transformation." *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*. Ed. by Xavier Rival. Vol. 9837. Lecture Notes in Computer Science. Springer, 2016, pp. 170–188.

[88] Chaoqiang Deng and Kedar S. Namjoshi. "Securing the SSA Transform." *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*. Ed. by Francesco Ranzato. Vol. 10422. Lecture Notes in Computer Science. Springer, 2017, pp. 88–105.

[89] Dominique Devriese, Marco Patrignani, and Frank Piessens. "Parametricity versus the universal type." *PACMPL* 2.POPL (2018), 38:1–38:23.

[90] Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel. "Modular, Fully-abstract Compilation by Approximate Back-translation." *Log. Methods Comput. Sci.* 13.4 (2017). DOI: 10.23638/LMCS-13(4:2)2017.

[91] Dominique Devriese and Frank Piessens. "Noninterference through Secure Multi-execution." *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010, pp. 109–124. DOI: 10.1109/SP.2010.15.

[92] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight Jr., Benjamin C. Pierce, and André DeHon. "Architectural Support for Software-Defined Metadata Processing." *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*. 2015, pp. 487–502.

[93] C. Disselkoen, R. Jagadeesan, A. S. A. Jeffrey, and J. Riely. "The Code That Never Ran: Modeling Attacks on Speculative Evaluation." *Proc. IEEE Symp. Security and Privacy*. 2019.

[94] Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. “CapablePtrs: Securely Compiling Partial Programs using the Pointers-as-Capabilities Principle.” *CoRR* abs/2005.05944 (2020). arXiv: 2005.05944. URL: https://arxiv.org/abs/2005.05944.

[95] Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. “A co-contextual formulation of type rules and its application to incremental type checking.” *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2015, pp. 880–897. DOI: 10.1145/2814270.2814277.

[96] Facebook. *Pyre - A performant type-checker for Python 3*. URL: https://pyre-check.org/.

[97] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009. ISBN: 978-0-262-06275-6. URL: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2\&tid=11885.

[98] Amy P. Felty, Elsa L. Gunter, John Hannan, Dale Miller, Gopalan Nadathur, and Andre Scedrov. “Lambda-Prolog: An Extended Logic Programming Language.” *9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings*. Ed. by Ewing L. Lusk and Ross A. Overbeek. Vol. 310. Lecture Notes in Computer Science. Springer, 1988, pp. 754–755. DOI: 10.1007/BFb0012882.

[99] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. “Komodo: Using verification to disentangle secure-enclave hardware from software.” *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 287–305. DOI: 10.1145/3132747.3132782.

[100] Cédric Fournet, Nikhil Swamy, Juan Chen, Pierre-Évariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. “Fully abstract compilation to JavaScript.” *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 371–384.

[101] Letterio Galletta. “Adaptivity: linguistic mechanisms and static analysis techniques.” PhD thesis. 2014.

[102] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware.” *J. Cryptographic Engineering* 8.1 (2018), pp. 1–27. DOI: 10.1007/s13389-016-0141-6.

[103] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. “Efficient and provable local capability revocation using uninitialized capabilities.” *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–30. DOI: 10.1145/3434287.

[104] Vida Ghodssi. “Incremental analysis of programs.” PhD thesis. University of Central Florida, 1983.

[105] J. A. Goguen and J. Meseguer. “Security Policies and Security Models.” *IEEE Symposium on Security and Privacy*. 1982, pp. 11–20.

[106] Travis Goodspeed. “Practical attacks against the MSP430 BSL.” *Twenty-Fifth Chaos Communications Congress*. 2008.

[107] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. "Cache Attacks on Intel SGX." *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017*. 2017, 2:1–2:6. DOI: 10.1145/3065913.3065915.

[108] Johannes Götzfried, Tilo Müller, Ruan de Clercq, Pieter Maene, Felix Freiling, and Ingrid Verbauwhede. "Soteria: Offline Software Protection Within Low-cost Embedded Devices." *Proceedings of the 31st Annual Computer Security Applications Conference*. ACSAC 2015. Los Angeles, CA, USA: ACM, 2015, pp. 241–250. ISBN: 978-1-4503-3682-6. DOI: 10.1145/2818000.2856129.

[109] Sylvia Grewe, Sebastian Erdweg, Pascal Wittmann, and Mira Mezini. "Type Systems for the Masses: Deriving Soundness Proofs and Efficient Checkers." *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* Onward! 2015. ACM, 2015, 137–150.

[110] Daniel Gruss. "Software-based Microarchitectural Attacks." PhD thesis. Graz University of Technology, June 2017.

[111] Roberto Guanciale, Musard Balliu, and Mads Dam. "InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis." *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. 2020, pp. 1853–1869. DOI: 10.1145/3372297.3417246.

[112] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. "Spectector: Principled Detection of Speculative Information Flows." *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020, pp. 1–19. DOI: 10.1109/SP40000.2020.00011.

[113] Marcus Hähnel, Weidong Cui, and Marcus Peinado. "High-Resolution Side Channels for Untrusted Operating Systems." *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. 2017, pp. 299–312. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel.

[114] Wenjian He, Wei Zhang, Sanjeev Das, and Yang Liu. "SGXlinger: A New Side-Channel Attack Vector Based on Interrupt Latency Against Enclave Execution." *36th IEEE International Conference on Computer Design, ICCD 2018, Orlando, FL, USA, October 7-10, 2018*. IEEE Computer Society, 2018, pp. 108–114. DOI: 10.1109/ICCD.2018.00025.

[115] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. "Context-oriented Programming." *J. Object Technol.* 7.3 (2008), pp. 125–151. DOI: 10.5381/jot.2008.7.3.a4.

[116] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. "Diversification and obfuscation techniques for software security: A systematic literature review." *Information and Software Technology* 104 (2018), pp. 72–93.

[117] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. "Cache Attacks Enable Bulk Key Recovery on the Cloud." *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. 2016, pp. 368–388. DOI: 10.1007/978-3-662-53140-2_18.

[118] Facebook Infer. *Infer static analyzer*. URL: http://fbinfer.com/.

[119] Texas Instruments. *MSP430x1xx Family: User Guide*. http://www.ti.com/lit/ug/slau049f/slau049f.pdf.

[120] Gregory F. Johnson and Janet A. Walz. "A Maximum-Flow Approach to Anomaly Isolation in Unification-Based Incremental Type Inference." *Proceedings of the 13th Symposium on Principles of Programming Languages*. ACM, 1986, pp. 44–57.

[121] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, Boris Eng, and Benjamin C. Pierce. "Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation." *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. 2016, pp. 45–60. DOI: 10.1109/CSF.2016.11.

[122] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. "Obfuscator-LLVM–software protection for the masses." *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE. 2015, pp. 3–9.

[123] Paris C. Kanellakis and John C. Mitchell. "Polymorphic Unification and ML Typing." *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. 1989, pp. 105–115. DOI: 10.1145/75277.75286.

[124] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. "Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach." *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. IEEE Computer Society, 2011, pp. 413–428. DOI: 10.1109/SP.2011.19.

[125] Andrew Kennedy. "Securing the .NET programming model." *Theor. Comput. Sci.* 364.3 (2006), pp. 311–317.

[126] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors." *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014, pp. 361–372. DOI: 10.1109/ISCA.2014.6853210.

[127] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems." *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*. Vol. 1109. LNCS. Springer-Verlag, 1996, pp. 104–113.

[128] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution." *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[129] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. "TrustLite: a security architecture for tiny embedded devices." *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*. Ed. by Dick C. A. Bulterman, Herbert Bos, Antony I. T. Rowstron, and Peter Druschel. ACM, 2014, 10:1–10:14. DOI: 10.1145/2592798.2592824.

[130] Edlira Kuci. "Co-Contextual Type Systems: Contextless Deductive Reasoning for Correct Incremental Type Checking." PhD thesis. Darmstadt: Technische Universität, 2020. URL: http://tuprints.ulb.tu-darmstadt.de/11419/.

[131] Leslie Lamport. "Proving the Correctness of Multiprocess Programs." *IEEE Trans. Software Eng.* 3.2 (1977), pp. 125–143.

[132] Tımea László and Ákos Kiss. "Obfuscating C++ programs via control flow flattening." *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica* 30.1 (2009), pp. 3–19.

[133] Steven Lauterburg, Ahmed Sobeih, Darko Marinov, and Mahesh Viswanathan. "Incremental state-space exploration for programs with dynamically allocated data." *30th International Conference on Software Engineering (ICSE 2008)*. 2008, pp. 291–300. DOI: 10.1145/1368088.1368128.

[134] Christopher League, Zhong Shao, and Valery Trifonov. "Type-preserving compilation of Featherweight Java." *ACM Trans. Program. Lang. Syst.* 24.2 (2002), pp. 112–152. DOI: 10.1145/514952.514954.

[135] Oukseh Lee and Kwangkeun Yi. "Proofs about a Folklore Let-Polymorphic Type Inference Algorithm." *ACM Trans. Program. Lang. Syst.* 20.4 (1998), pp. 707–723.

[136] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing." *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 557–574.

[137] K. Rustan M. Leino and Valentin Wüstholz. "Fine-Grained Caching of Verification Results." *Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. LNCS. Springer, 2015, pp. 380–397.

[138] Xavier Leroy. "Formal certification of a compiler back-end or: programming a compiler with a proof assistant." *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. 2006, pp. 42–54.

[139] Xavier Leroy. "A Formally Verified Compiler Back-end." *J. Autom. Reason.* 43.4 (2009), pp. 363–446. DOI: 10.1007/s10817-009-9155-4.

[140] Xavier Leroy and François Pessaux. "Type-based analysis of uncaught exceptions." *ACM Trans. Program. Lang. Syst.* 22.2 (2000), pp. 340–377. DOI: 10.1145/349214.349230.

[141] Henry M. Levy. *Capability-based computer systems*. Digital Press, 1984. URL: https://homes.cs.washington.edu/~levy/capabook/.

[142] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space." *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 973–990. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[143] Andres Löh, Conor McBride, and Wouter Swierstra. "A Tutorial Implementation of a Dependently Typed Lambda Calculus." *Fundam. Inform.* 102.2 (2010), pp. 177–207.

[144] Nancy A Lynch and Mark R Tuttle. *An introduction to input/output automata*. Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[145] Daniel Marino and Todd D. Millstein. "A generic type-and-effect system." *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*. Ed. by Andrew Kennedy and Amal Ahmed. ACM, 2009, pp. 39–50. DOI: 10.1145/1481861.1481868.

[146] Isabella Mastroeni. "Abstract interpretation-based approaches to Security - A Survey on Abstract Non-Interference and its Challenging Applications." *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013.* 2013, pp. 41–65.

[147] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. "TrustVisor: Efficient TCB Reduction and Attestation." *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010, pp. 143–158. DOI: 10.1109/SP.2010.17.

[148] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. "Innovative instructions and software model for isolated execution." *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*. Ed. by Ruby B. Lee and Weidong Shi. ACM, 2013, p. 10. DOI: 10.1145/2487726.2488368.

[149] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. "Scalable and incremental software bug detection." *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13*. 2013, pp. 554–564. URL: http://doi.acm.org/10.1145/2491411.2501854.

[150] Lambert G. L. T. Meertens. "Incremental Polymorphic Type Checking in B." *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*. Ed. by John R. Wright, Larry Landweber, Alan J. Demers, and Tim Teitelbaum. ACM, 1983, pp. 265–275.

[151] Weiyu Miao and Jeremy G. Siek. "Incremental type-checking for type-reflective metaprograms." *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*. Ed. by Eelco Visser and Jaakko Järvi. ACM, 2010, pp. 167–176.

[152] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. "Typed Closure Conversion." *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. 1996, pp. 271–283. DOI: 10.1145/237721.237791.

[153] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks." *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. 2017, pp. 69–90. DOI: 10.1007/978-3-319-66787-4_4.

[154] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves." *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, 2020.

[155] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. “The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks.” *Information Security and Cryptology - ICISC 2005, 8th International Conference*. Ed. by Dongho Won and Seungjoo Kim. Vol. 3935. LNCS. 2005, pp. 156–168.

[156] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. “From system F to typed assembly language.” *ACM Trans. Program. Lang. Syst.* 21.3 (1999), pp. 527–568.

[157] Rashmi Mudduluru and Murali Krishna Ramanathan. “Efficient Incremental Static Analysis Using Path Abstraction.” *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014*. 2014, pp. 125–139. URL: https://doi.org/10.1007/978-3-642-54804-8_9.

[158] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. “Plundervolt: Software-based Fault Injection Attacks against Intel SGX.” *41st IEEE Symposium on Security and Privacy (S&P’20)*. 2020.

[159] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. “Enforcing Robust Declassification and Qualified Robustness.” *Journal of Computer Security* 14.2 (2006), pp. 157–196.

[160] Kedar S. Namjoshi and Lucas M. Tabajara. “Witnessing Secure Compilation.” *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings*. 2020, pp. 1–22. DOI: 10.1007/978-3-030-39322-9_1.

[161] Kedar S. Namjoshi and Lenore D. Zuck. “Witnessing Program Transformations.” *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. 2013, pp. 304–323.

[162] George C. Necula. “Proof-Carrying Code.” *Conference Record of POPL’97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*. 1997, pp. 106–119. DOI: 10.1145/263699.263712.

[163] George C. Necula. “Translation validation for an optimizing compiler.” *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*. Ed. by Monica S. Lam. ACM, 2000, pp. 83–94.

[164] Max S. New, William J. Bowman, and Amal Ahmed. “Fully abstract compilation via universal embedding.” *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 2016, pp. 103–116. DOI: 10.1145/2951913.2951941.

[165] Lawton Nichols, Mehmet Emre, and Ben Hardekopf. “Fixpoint Reuse for Incremental JavaScript Analysis.” *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. SOAP 2019. New York, NY, USA: ACM, 2019, 2–7. ISBN: 9781450367202. DOI: 10.1145/3315568.3329964.

[166] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999. URL: https://doi.org/10.1007/978-3-662-03811-6.

[167] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, 2007. ISBN: 978-1-84628-691-9. DOI: 10.1007/978-1-84628-692-6.

[168] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base." *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*. Ed. by Samuel T. King. USENIX Association, 2013, pp. 479–494. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/noorman.

[169] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix C. Freiling. "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices." *ACM Trans. Priv. Secur.* 20.3 (2017), 7:1–7:33.

[170] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices." *ACM Trans. Priv. Secur.* 20.3 (July 2017), 7:1–7:33. ISSN: 2471-2566. DOI: 10.1145/3079763.

[171] Ivan Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. "VRASED: A Verified Hardware/Software Co-Design for Remote Attestation." *28th USENIX Security Symposium, USENIX Security 2019*. 2019.

[172] Martin Odersky, Martin Sulzmann, and Martin Wehr. "Type Inference with Constrained Types." *TAPOS* 5.1 (1999), pp. 35–55.

[173] André Pacak, Sebastian Erdweg, and Tamás Szabó. "A Systematic Approach to Deriving Incremental Type Checkers." *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428195.

[174] Joachim Parrow. "General conditions for full abstraction." *Math. Struct. Comput. Sci.* 26.4 (2016), pp. 655–657. DOI: 10.1017/S0960129514000280.

[175] Marco Patrignani. "Why Should Anyone use Colours? or, Syntax Highlighting Beyond Code Snippets." *CoRR* abs/2001.11334 (2020). arXiv: 2001.11334. URL: https://arxiv.org/abs/2001.11334.

[176] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. "Secure Compilation to Protected Module Architectures." *ACM Trans. Program. Lang. Syst.* 37.2 (2015), 6:1–6:50.

[177] Marco Patrignani, Amal Ahmed, and Dave Clarke. "Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work." *ACM Comput. Surv.* 51.6 (2019). ISSN: 0360-0300. DOI: 10.1145/3280984.

[178] Marco Patrignani and Dave Clarke. "Fully abstract trace semantics for protected module architectures." *Computer Languages, Systems & Structures* 42 (2015), pp. 22–45.

[179] Marco Patrignani and Deepak Garg. "Secure Compilation and Hyperproperty Preservation." *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 2017, pp. 392–404.

[180] Marco Patrignani and Deepak Garg. "Robustly Safe Compilation" (2019), pp. 469–498. DOI: 10.1007/978-3-030-17184-1_17.

[181] Marco Patrignani and Marco Guarnieri. "Exorcising Spectres with Secure Compilers." *CoRR* abs/1910.08607 (2019). arXiv: 1910.08607. URL: http://arxiv.org/abs/1910.08607.

[182] Marco Patrignani and Marco Guarnieri. "Exorcising Spectres with Secure Compilers." *CoRR* abs/1910.08607 (2020). arXiv: 1910.08607. URL: http://arxiv.org/abs/1910.08607.

[183] Daniel Patterson and Amal Ahmed. "The next 700 compiler correctness theorems (functional pearl)." *Proc. ACM Program. Lang.* 3.ICFP (2019), 85:1–85:29. DOI: 10.1145/3341689.

[184] Federico Pennino. *CADL: generating a type-checking module from Datalog to OCaml*. Bachelor's thesis, code available at https://github.com/freek9807/CADL. 2020.

[185] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN: 978-0-262-16209-8.

[186] Benjamin C Pierce. *Advanced topics in types and programming languages*. MIT press, 2005.

[187] Benjamin Pierce and Eijiro Sumii. "Relating Cryptography and Polymorphism" (2000). URL: http://www.kb.ecei.tohoku.ac.jp/~sumii/pub/infohide.pdf.

[188] Frank Piessens. "Security across abstraction layers: old and new examples." *IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2020, Genoa, Italy, September 7-11, 2020*. 2020, pp. 271–279. DOI: 10.1109/EuroSPW51379.2020.00043.

[189] Amir Pnueli, Michael Siegel, and Eli Singerman. "Translation Validation." *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. Ed. by Bernhard Steffen. Vol. 1384. Lecture Notes in Computer Science. Springer, 1998, pp. 151–166.

[190] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. "Liquid information flow control." *Proc. ACM Program. Lang.* 4.ICFP (2020), 105:1–105:30. DOI: 10.1145/3408987.

[191] Mila Dalla Preda and Roberto Giacobazzi. "Control Code Obfuscation by Abstract Interpretation." *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), 7-9 September 2005, Koblenz, Germany*. 2005, pp. 301–310. DOI: 10.1109/SEFM.2005.13.

[192] Mila Dalla Preda and Roberto Giacobazzi. "Semantics-based code obfuscation by abstract interpretation." *Journal of Computer Security* 17.6 (2009), pp. 855–908. DOI: 10.3233/JCS-2009-0345.

[193] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. "Verified low-level programming embedded in F*." *PACMPL* 1.ICFP (2017), 17:1–17:29. DOI: 10.1145/3110261.

[194] Kyle Pullicino. "Jif: Language-based Information-flow Security in Java." *CoRR* abs/1412.8639 (2014).

[195] Rui Qiu, Guowei Yang, Corina S. Pasareanu, and Sarfraz Khurshid. "Compositional Symbolic Execution with Memoized Replay." *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Volume 1*. 2015, pp. 632–642. URL: https://doi.org/10.1109/ICSE.2015.79.

[196] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "CrossTalk: Speculative Data Leaks Across Cores Are Real." *IEEE S&P 2021*. May 2021. URL: https://download.vusec.net/papers/crosstalk_sp21.pdf.

[197] Vineet Rajani and Deepak Garg. "Types for Information Flow Control: Labeling Granularity and Semantic Models." *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. 2018, pp. 233–246.

[198] Fabrice Rastello. *SSA-based Compiler Design*. 1st. Springer Publishing Company, Incorporated, 2016. ISBN: 1441962018, 9781441962010.

[199] Charles Reis and Steven D. Gribble. "Isolating Web Programs in Modern Browser Architectures." *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. Nuremberg, Germany: Association for Computing Machinery, 2009, 219–232. ISBN: 9781605584829. DOI: 10.1145/1519065.1519090.

[200] Tom Rini. *Re: [PATCH] C undefined behavior fix*. https://gcc.gnu.org/ml/gcc/2002-01/msg00035.html.

[201] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. "Return-Oriented Programming: Systems, Languages, and Applications." *ACM Trans. Inf. Syst. Secur.* 15.1 (2012), 2:1–2:34.

[202] Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J. F. Bastien, and Michael Holman. "Bringing the web up to speed with WebAssembly." *Commun. ACM* 61.12 (2018), pp. 107–115. DOI: 10.1145/3282510.

[203] Grigore Rosu and Traian-Florin Serbanuta. "An overview of the K semantic framework." *J. Log. Algebraic Methods Program.* 79.6 (2010), pp. 397–434. DOI: 10.1016/j.jlap.2010.03.012.

[204] Alejandro Russo, Koen Claessen, and John Hughes. "A library for light-weight information-flow security in haskell." *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. 2008, pp. 13–24.

[205] Barbara G. Ryder and Marvin C. Paull. "Incremental Data-Flow Analysis." *ACM Trans. Program. Lang. Syst.* 10.1 (1988), pp. 1–50. URL: http://doi.acm.org/10.1145/42192.42193.

[206] Andrei Sabelfeld and Andrew C. Myers. "Language-based information-flow security." *IEEE J. Sel. Areas Commun.* 21.1 (2003), pp. 5–19. DOI: 10.1109/JSAC.2002.806121. URL: https://doi.org/10.1109/JSAC.2002.806121.

[207] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue In-flight Data Load." *S&P*. May 2019.

[208] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling." *CCS*. 2019.

[209] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Malware Guard Extension: abusing Intel SGX to conceal cache attacks." *Cybersecur.* 3.1 (2020), p. 2. DOI: 10.1186/s42400-019-0042-y.

[210] Helmut Seidl, Julian Erhard, and Ralf Vogler. "Incremental Abstract Interpretation." *From Lambda Calculus to Cybersecurity Through Program Analysis: Essays Dedicated to Chris Hankin on the Occasion of His Retirement*. Cham: Springer International Publishing, 2020, pp. 132–148. DOI: 10.1007/978-3-030-41103-9_5.

[211] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. "Translation validation for a verified OS kernel." *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 2013, pp. 471–482.

[212] Zhong Shao and Andrew W. Appel. "A Type-Based Compiler for Standard ML." *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*. 1995, pp. 116–129. DOI: 10.1145/207110.207123.

[213] Vincent Simonet. *Flow Caml*. https://www.normalesup.org/~simonet/soft/flowcaml/.

[214] Christian Skalka and Scott F. Smith. "History Effects and Verification." *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*. 2004, pp. 107–128. DOI: 10.1007/978-3-540-30477-7_8.

[215] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. "Reasoning About a Machine with Local Capabilities - Provably Safe Stack and Return Pointer Management." *27th European Symposium on Programming, ESOP 2018, Proceedings*. Ed. by Amal Ahmed. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 475–501.

[216] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. "StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities." *Proc. ACM Program. Lang.* 3.POPL (2019), 19:1–19:28. DOI: 10.1145/3290332.

[217] Geoffrey Smith. "Principles of Secure Information Flow Analysis." *Malware Detection*. Springer, 2007, pp. 291–307. URL: https://doi.org/10.1007/978-0-387-44599-1_13.

[218] Deian Stefan, David Mazières, John C. Mitchell, and Alejandro Russo. "Flexible dynamic information flow control in the presence of exceptions." *J. Funct. Program.* 27 (2017). DOI: 10.1017/S0956796816000241.

[219] Eijiro Sumii. "MinCaml: a simple and efficient compiler for a minimal functional language." *Proceedings of the 2005 workshop on Functional and declarative programming in education*. Ed. by Robby Bruce Findler, Michael Hanus, and Simon Thompson. ACM, 2005, pp. 27–38.

[220] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. "IncA: a DSL for the definition of incremental program analyses." *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 2016, pp. 320–331. URL: http://doi.acm.org/10.1145/2970276.2970298.

[221] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management." *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 1057–1074. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

[222] Carlos Tomé Cortiñas, Marco Vassena, and Alejandro Russo. "Securing Asynchronous Exceptions." *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. 2020, pp. 214–229. DOI: 10.1109/CSF49147.2020.00023.

[223] Stelios Tsampas, Andreas Nuyts, Dominique Devriese, and Frank Piessens. "A Categorical Approach to Secure Compilation." *Coalgebraic Methods in Computer Science - 15th IFIP WG 1.3 International Workshop, CMCS 2020, Colocated with ETAPS 2020, Dublin, Ireland, April 25-26, 2020, Proceedings*. Ed. by Daniela Petrisan and Jurriaan Rot. Vol. 12094. Lecture Notes in Computer Science. Springer, 2020, pp. 155–179. DOI: 10.1007/978-3-030-57201-3_9.

[224] Jo Van Bulck. "Microarchitectural Side-Channel Attacks for Privileged Software Adversaries." PhD thesis. KU Leuven, Leuven, Belgium, Sept. 2020. URL: https://lirias.kuleuven.be/3047121.

[225] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic." *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: ACM, pp. 178–195. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243822.

[226] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. "Linear capabilities for fully abstract compilation of separation-logic-verified code." *Proc. ACM Program. Lang.* 3.ICFP (2019), 84:1–84:29. DOI: 10.1145/3341688.

[227] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. "Automatically eliminating speculative leaks from cryptographic code with blade." *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–30. DOI: 10.1145/3434330.

[228] Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. "Mac a verified static information-flow control library." *Journal of logical and algebraic methods in programming* 95 (2018), pp. 148–180.

[229] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. "A Sound Type System for Secure Flow Analysis." *Journal of Computer Security* 4.2/3 (1996), pp. 167–188. URL: https://doi.org/10.3233/JCS-1996-42-304.

[230] Guido Wachsmuth, Gabriël D. P. Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. "A Language Independent Task Engine for Incremental Name and Type Analysis." *Software Language Engineering - 6th International Conference*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Vol. 8225. LNCS. Springer, 2013, pp. 260–280.

[231] Philip Wadler. "Deforestation: Transforming Programs to Eliminate Trees." *Theor. Comput. Sci.* 73.2 (1990), pp. 231–248. DOI: 10.1016/0304-3975(90)90147-A.

[232] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient Software-Based Fault Isolation." *SIGOPS Oper. Syst. Rev.* 27.5 (Dec. 1993), 203–216. ISSN: 0163-5980. DOI: 10.1145/173668.168635.

[233] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. *Capability*

Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7). Tech. rep. UCAM-CL-TR-927. University of Cambridge, Computer Laboratory, June 2019. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf.

[234] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization." *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015, pp. 20–37.

[235] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves." *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*. 2016, pp. 440–457. DOI: 10.1007/978-3-319-45744-4_22.

[236] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution." *Technical report* (2018).

[237] Andrew K. Wright and Matthias Felleisen. "A Syntactic Approach to Type Soundness." *Inf. Comput.* 115.1 (1994), pp. 38–94. DOI: 10.1006/inco.1994.1093.

[238] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. "Eliminating timing side-channel leaks using program repair." *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. Ed. by Frank Tip and Eric Bodden. ACM, 2018, pp. 15–26. DOI: 10.1145/3213846.3213851.

[239] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems." *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 640–656. DOI: 10.1109/SP.2015.45.

[240] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems." *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015, pp. 640–656. DOI: 10.1109/SP.2015.45.

[241] Guowei Yang, Matthew B. Dwyer, and Gregg Rothermel. "Regression model checking." *25th IEEE International Conference on Software Maintenance (ICSM 2009)*. 2009, pp. 115–124. URL: https://doi.org/10.1109/ICSM.2009.5306334.

[242] Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. "Directed Incremental Symbolic Execution." *ACM Trans. Softw. Eng. Methodol.* 24.1 (2014), 3:1–3:42. URL: http://doi.acm.org/10.1145/2629536.

[243] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. "Dead Store Elimination (Still) Considered Harmful." *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. 2017, pp. 1025–1040.

[244] Jyh-Shiarn Yur, Barbara G. Ryder, and William Landi. "An Incremental Flow- and Context-Sensitive Pointer Aliasing Analysis." *Proceedings of the 1999 International Conference on Software Engineering*. 1999, pp. 442–451. URL: http://doi.acm.org/10.1145/302405.302676.

[245] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. "Using Information Flow to Design an ISA that Controls Timing Channels." *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. 2019, pp. 272–287. DOI: 10.1109/CSF.2019.00026.

[246] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. "A Hardware Design Language for Timing-Sensitive Information-Flow Security." *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*. Ed. by Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas. ACM, 2015, pp. 503–516. DOI: 10.1145/2694344.2694372.

[247] WebAssembly team. *Binaryen - Compiler infrastructure and toolchain library for WebAssembly*. https://github.com/WebAssembly/binaryen. Online; last access Mar 2021.