A Brief Tour of Formally Secure Compilation

Matteo Busi[1] and Letterio Galletta[2]

[1] Università di Pisa, Pisa, Italy
matteo.busi@di.unipi.it
[2] IMT School for Advanced Studies, Lucca, Italy
letterio.galletta@imtlucca.it

Abstract. Modern programming languages provide helpful high-level abstractions and mechanisms (e.g. types, module, automatic memory management) that enforce good programming practices and are crucial when writing correct and secure code. However, the security guarantees provided by such abstractions are not preserved when a compiler translates a source program into object code. Formally secure compilation is an emerging research field concerned with the design and the implementation of compilers that preserve source-level security properties at the object level. This paper presents a short guided tour of the relevant literature on secure compilation. Our goal is to help newcomers to grasp the basic concepts of this field and, for this reason, we rephrase and present the most relevant results in the literature in a common setting.

1 Introduction

Compilers are one of the fundamental tools for software development. Not only they translate a *source program*, written in a *high-level language*, into an *object* code (low-level), but they also help programmers to catch a variety of errors, and optimize the resulting program. A well-known advantage of using a high-level language is that it usually provides a variety 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 security properties, the resulting code is vulnerable to attacks. Consider for example Fig. 1, where the code on the left is written in a `C`-like language with no pointers, and the code on the right is its translation into a RISC-like assembly (under the assumption that `count` is stored in the register `R0`). In this case the programmer's intention was to use the encapsulation mechanism provided by the `static` qualifier to guarantee *integrity* on the value stored in `count`. However, at the object level, the value is no longer protected: any other object-level module linking our code as a library 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 object level mechanisms

to break high-level abstractions, thus making the reasoning at the source level useless [37,14]. A possible solution to this problem would be to adapt the source to the object language (or vice-versa) so to make them equally powerful, but that is undesirable because, in doing that, we would lose all the advantages that come from having different levels of abstraction in the source and in the object. Moreover, it would not be even sufficient. Indeed, as recently observed by D'Silva et al. [23] and previously known in the cybersecurity community [1,2,3,42], even less radical, seemingly innocuous code transformations that maps a language into itself (e.g. compiler optimizations) may hinder security. Consider for example the snippet in Fig. 2 that checks the correctness of a given PIN number. A *dead-store elimination*, i.e. a transformation that removes assignments to variables that are not used afterwards, optimizes away the line highlighted in red. 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 operating system in which our optimized code is executed), so making it possible to leak the secret.

A way to solve both the above problems is to work on the compiler itself, by guaranteeing that it preserves the source-level security properties. The emerging field of *formally 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 into the object level or, equivalently, that all the attacks that can be carried out at the object level can also be carried out at the source level. In this way, it is enough to reason at the source level to rule out attacks even at the object level. Of course, making program translations secure in general would be a huge step forward for the security of software systems, especially for those that are critical (e.g. avionics or medical software) and possibly written in unsafe languages.

Structure of the tour. The next section discusses *non-robust* secure compilation, and briefly overviews its basic concepts and motivations, together with the recent literature. Similarly, in Section 3 we discuss an extended notion of secure compilation to deal with active attackers. Then, Section 4 reports common dynamic mechanisms, complementary to secure compilation, for protecting programs di-

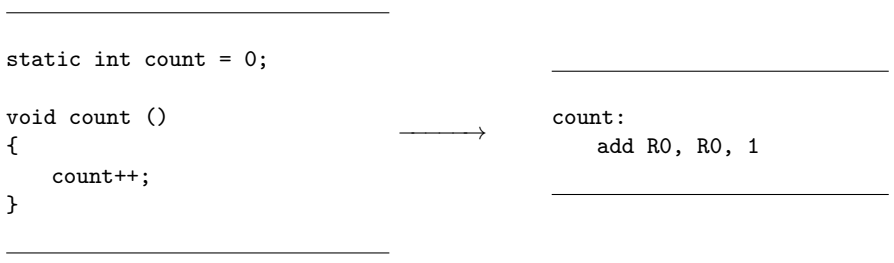

Fig. 1. Non secure compilation.

```
print("Pin:");
pin = read_secret();

if (check(pin))
    print("OK!");
else
    print("KO!");

pin = 0; // reset pin
```

Fig. 2. Non secure dead-store elimination (highlighted in red).

rectly at the hardware level. In Section 5 we briefly discuss the open problems and the challanges of the field. Finally, Section 6 concludes.

Note that, from now onwards, we highlight in blue the elements of the source language, and in red the elements of the object language for better readability.

2 Secure Compilation

In the previous section we informally introduced secure compilation, explaining that it is concerned with guaranteeing that *source level* security properties are preserved when compiling a program into a *object language*. In this section, we specify more precisely what *security preservation* actually means.

First, we need to clarify what kind of security policies we are interested in. To do that, we introduce the concept of *program behaviour*, as the set of sequences of actions/states that a program performs/reaches during its execution. Formally, the behaviour of a program p – written $p{\downarrow}$ – is a subset of the set of all the *finite* and the *infinite* traces Ψ defined on a set Σ of *observables (or events)*. Typically, observables encode information about I/O operations performed by a program, errors occurred during executions, termination, or divergence [26]. Intuitively, a security policy predicates on the program behaviour and establishes which traces are allowed and which not. There are (at least) two different proposals on how to formalise security properties in the literature: *trace properties* [25,10] and *hyperproperties* [17].

The first proposal defines a security policy as the set of permitted traces, more precisely, a trace property is a subset of Ψ. In this case, we say that a program p *satisfies* a trace property P whenever all the executions of p, i.e. $p{\downarrow}$, are allowed by P, in symbols $p{\downarrow} \subseteq P$. Two typical classes of trace properties are *safety* (nothing bad will ever happen) and *liveness* (something good will eventually happen).

The commonly used chinese-wall policy is a safety property, stating that "a system may write to the network as long as it did not read from a file" and it

takes the following form:

$$\{t \in \Psi \mid \neg(\exists i < j \,.\, isFileRead(t_i) \wedge isNetworkWrite(t_j))\}$$

where t_i and t_j are the observables at the steps i and j of the trace t, and *isFileRead* and *isNetworkWrite* are predicates with the expected meaning.

However, trace properties are not enough to express all the interesting security policies. For example, *termination insensitive non-interference* requires that the public part of the state is not affected by the non-public part, otherwise some information of this last kind turns out to be disclosed. While this policy is not a trace property, it can be expressed through a hyperproperty: compare all pairs of traces the initial states of which coincide on their public part, and verify that they reach the same final state, up to their public part.

The needed additional expressive power comes from the ability of predicating on multiple executions of the same program. Indeed, hyperproperties express security policies as a set of program behaviours, i.e. a set of sets of traces. We say that a program p *satisfies* a hyperproperty $\mathbf{H} \subseteq \wp(\Psi)$ whenever the whole behaviour of p is an element of $\mathbf{H}$, i.e. $p{\downarrow} \in \mathbf{H}$. We formalise the above non-interference as follows:

$$\{T \mid \forall \text{ traces } t, t' \in T. \, t_0 =_{public} t'_0 \Rightarrow t_f =_{public} t'_f\}$$

where t_0, t'_0 and t_f, t'_f are the initial and final states of t and t', respectively, and $t_i =_{public} t'_j$ holds if and only if the public part of the states t_i and t'_j coincides.

For conciseness, we shall write $p{\downarrow} \vDash \mathcal{F}$ whenever p satisfies a given security property $\mathcal{F}$: when $\mathcal{F}$ is a trace property P, $p{\downarrow} \vDash \mathcal{F}$ reads as $p{\downarrow} \subseteq P$, and when $\mathcal{F}$ is a hyperproperty $\mathbf{H}$, $p{\downarrow} \in \mathbf{H}$.

We say that a compiler is secure whenever it *preserves* the properties of a source program s under the translation steps. In other words, if s satisfies a given family of properties $\mathbb{F}$ then also its compiled counterpart does:

Definition 1 (Secure compiler). *A compiler* $[\![\cdot]\!]^S_O$ *from a source language* S *to an object language* O *is* secure for $\mathbb{F}$ iff *for any source program* s

$$\forall \mathcal{F} \in \mathbb{F}. \; s{\downarrow} \vDash \mathcal{F} \Rightarrow [\![s]\!]^S_O{\downarrow} \vDash \mathcal{F}$$

Consider first $\mathbb{F}$ to be the set of trace properties. In this case, for proving a compiler secure it suffices proving that is correct, according to the following definition that requires the source program to exhibit a trace when the object code does:

Definition 2 (Compiler correctness [26]). *We say that a compiler* $[\![\cdot]\!]^S_O$ *is* correct iff *for any source program* s*:*

$$t \in s{\downarrow} \Leftarrow t \in [\![s]\!]^S_O{\downarrow}$$

Correctness is often proved by establishing a *backward simulation (or refinement)* between the behaviours of source and object 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 O 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 [26].

A correct compiler is also secure for trace properties:

Theorem 1 ([6]). *Every correct compiler preserves all the trace properties.*

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 trace properties can be turned into SSC hyperproperties) [17].

Theorem 2 ([17]). *Every correct compiler preserves* all and only *the SSC hyperproperties.*

Note that in Theorems 1 and 2 we made the implicit assumptions that the observables we are considering for establishing the correctness are the same on which our security property predicates. Consequently, the correctness proof is monolithic in the sense that it has to deal with both the functional and the non-functional properties, and therefore the observables should 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 possible to reuse off the shelf compilers already proved correct. Take for example CompCert [26], 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 the above notion of non-interference would require to observe also the values of public variables and to re-do the proof. For these reasons, many 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. The proof that a compiler preserves the security properties of interest is then done assuming that it is correct, i.e. that it preserves the I/O semantics of programs.

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 Namjohsi proved that (variants of) popular compiler optimizations preserve the above property of non-interference [18], and introduced in [19] a technique that statically enforces it in SSA-based optimization pipelines. In a framework that generalises [18], Barthe et al. [13] studied the *cryptographic constant-time* policy, which is an instance of observational non-interference and requires that the execution time of a program does not depend on non-public parts of the state. The key ingredient of their proposal are (three variants of) *CT-simulation*, which is a pair of simulations, one considering two source programs and the other the corresponding two object programs.

3 Robust Secure Compilation

Secure compilation as presented in the previous section works well when we consider monolithic programs that incorporate in their code all the functionalities they require to operate. Actually, it is not fully adequate 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. Environments can be conveniently abstracted as *contexts* $C[\cdot]$, i.e. programs with a hole to be filled by the program to be executed. When an environment is malicious, the context acts as an attacker that can actively interact with the program at run-time, rather than passively watching at its execution.

To take into account active attackers and the external environment where the program is plugged in, we resort to the notion of $RHP(X)$ (§ 3.1 of [6]), where X denotes a family of hyperproperties, to update the Definition 1 as follows:

Definition 3 (Robustly secure compiler). *A compiler* $[\![\cdot]\!]_O^S$ *from a source language* S *to an object language* O *is* robustly secure for $\mathbb{F}$ iff *for any source program* s

$$\forall \mathcal{F} \in \mathbb{F}.\ (\forall C_S[\cdot].\ C_S[s]{\downarrow} \vDash \mathcal{F}) \Rightarrow (\forall C_O[\cdot].\ C_O[\,[\![s]\!]_O^S]{\downarrow} \vDash \mathcal{F})$$

The seminal work of Abadi [4] proposed *full abstraction* as a framework to assess this enhanced notion of compiler security. Indeed, a fully abstract compiler preserves and reflects the equivalence of behaviours between original and compiled programs under *any* untrusted context of execution. Equivalence of behaviours is expressed as *contextual* equivalence, often defined as $p \simeq p'$ *iff* $\forall C[\cdot], C[p]{\downarrow} = C[p']{\downarrow}$. We now rephrase the definition of full abstraction (FA) in our terms:

Definition 4 ([4]). *A compiler* $[\![\cdot]\!]_O^S$ *is fully abstract* iff

$$\forall s_1, s_2 \in S\,.\,s_1 \simeq s_2 \Leftrightarrow [\![s_1]\!]_O^S \simeq [\![s_2]\!]_O^S.$$

However, FA has some limitations. The first, most serious drawback is that real-world, off-the-shelf compilers seldom are FA. For example, neither the standard compiler from Java to Bytecode [4] nor the one from C# to CLR language [24] are FA. The second shortcoming is that FA can be hard to prove and even harder to disprove. Indeed, the compiler from System F to the cryptographic λ-calculus [35] was conjectured to be FA, but it was proved not such only 20 years later [21]. Also, enforcing FA requires to instrument the object code, often making it inefficient [34]. Finally, there is no exact characterization of the class of properties an FA compiler (robustly) preserves [6]. Indeed, FA sometimes does not preserve properties that secure compilation chains are expected to, as shown by the following example [33].

Example 1 (FA $\not\Rightarrow$ safety preservation). Consider the language S to be a simple functional language with boolean values and just the constantly `true` function:

$$S \ni s ::= \texttt{true} \mid \texttt{false} \mid \lambda\ \_.\texttt{true} \mid s_1\ s_2$$

Similarly, let O be an object language with booleans, integers and non-recursive functions:

$$O \ni o ::= \texttt{true} \mid \texttt{false} \mid n \mid x \mid \lambda\, x.o \mid o_1\, o_2 \mid \textbf{if}\ o_1 < o_2\ \textbf{then}\ o_3\ \textbf{else}\ o_4$$

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

$$[\![\texttt{true}]\!]^S_O \triangleq \texttt{true} \quad [\![\texttt{false}]\!]^S_O \triangleq \texttt{false} \quad [\![s_1\ s_2]\!]^S_O \triangleq [\![s_1]\!]^S_O\ [\![s_2]\!]^S_O$$
$$[\![\lambda\ \_.\texttt{true}]\!]^S_O \triangleq \lambda\, x.\textbf{if}\ x < 2\ \textbf{then}\ \texttt{true}\ \textbf{else}\ \texttt{false}$$

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 for S stating that "a function never outputs `false`", trivially satisfied by all the programs in S. However, object programs can output `false` depending on x, the object-level input provided by the context, hence invalidating the property.

An extensive treatment of FA compilation is outside the scope of this paper and we refer the interested reader to the survey by Patrignani et al. [31].

To overcome the limitations of FA sketched above, new secure compilation principles have been recently proposed. The first one was *trace-preserving (TP) compilation* for reactive programs [33]. Informally, a compiler $[\![\cdot]\!]^S_O$ is TP if any trace of the compiled program $[\![s]\!]^S_O$ is either a trace of s or is a special *invalid trace*. Depending on how invalid traces are defined, TP compilation comes in two flavours: *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 [33].

There have been interests in the literature in studying tailored classes of trace properties and hyperproperties, devising new principles for their preservation. In particular, Abate et al. [6] carried out a systematic investigation in search of hyperproperty-specific secure compilation principles that come in two forms: the first one, *property-full*, explicitly mentions the family of properties to be preserved, the second one, *property-free*, does not and fosters formal reasoning. The underlying idea is that tailored principles may be helpful to make proofs easier and to reduce the overhead in the generated code. Concentrating on trace properties one finds: *robust trace property preservation*, where the family of security properties $\mathbb{F}$ coincides with all the trace properties; *robust dense property preservation* [6] where $\mathbb{F}$ coincides with properties predicating on all terminating traces; and *robust safety property preservation* and *robustly safe compilation* [6,34], where $\mathbb{F}$ coincides with all the safety properties. Interestingly, robustly safe compilation can be also extended to consider modularity and undefined behaviours [5]. As for hyperproperties, there are a series of principles that we do not report here, e.g. those for relational hyperproperties – hyperproperties that consider multiple runs of multiple systems at the same time. For the sake of simplicity, here we present as an example the most general principle that preserves all the hyperproperties, which is an instance of Definition 3.

Definition 5 (Robust hyperproperty preservation [6]). *Let* s *be a source program. The* property-full *variant of* robust hyperproperty preservation *holds iff for any hyperproperty* $\mathbf{H}$

$$(\forall C_S[\cdot].\ C_S[s]{\downarrow} \vDash \mathbf{H}) \Rightarrow (\forall C_O[\cdot].\ C_O[\,[\![s]\!]^S_O]{\downarrow} \vDash \mathbf{H})$$

Its property-free *variant holds iff*

$$\forall C_O[\cdot].\ \exists C_S[\cdot].\ \forall m.\ m \in (C_O[\,[\![s]\!]^S_O]){\downarrow} \Leftrightarrow m \in (C_S[s]){\downarrow}$$

The proposed principles are elegant and expressive, however, there are no fully automatic provers available yet for establishing them. Some ongoing work [15] is exploring the possibility of extending translation validation [36,28] to automatically verify security properties. Intuitively, the idea is to prove in the style of translation validation that a given run of the compiler succeeds in (robustly) preserving a family of hyperproperties on a specific program, instead of proving the compiler secure for all programs.

4 Low-level Enforcement Mechanisms

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

A first proposal was to use an object language with a rich typed structure in order to preserve the type guarantees of the source language. An example is *typed assembly language (TAL)* [27] that among others has existential types to support closures and other data abstractions. It has been used as object language of a type-preserving compiler for System F. Some researches addressed the development of FA compilers targeting TAL [8,9], but we are not aware of any that respects one of the robust secure compilation principles above.

Other proposals consist 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 control-flow integrity and local-state encapsulation at the hardware level [16,38]. CMs are based on *memory capabilities*, i.e. fat pointers that include some access-control information, tagging registers and memory locations. Besides memory capabilities, many machines also implement *object capabilities*, i.e. sand-boxed executions 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)* [41] extends commodity RISC *Instruction Set Architectures (ISAs)* with capability-based primitives and supports real-world operating systems and applications. Recent research efforts have been devoted to formally

study the properties of this model and to securely compile C code to these machines [39,40,20]. Similarly, *protected module architectures (PMAs)* bring some security guarantees at the hardware level by splitting the memory into a protected and an unprotected section in order isolate sensitive and critical applications from the untrusted ones. The idea was further developed [7,32,30,29] and also implemented in commodity processors by industry, e.g. Intel Secure Guard eXtensions (Intel SGX).

In this line of research a further proposal is the *Sancus 2.0* [29] architecture, implementing PMA features on a low-cost and embedded microprocessor. Sancus achieves, without a *trusted computing base*, (i) software module isolation; (ii) remote attestation; (iii) secure communication and linking; (iv) confidential deployment; and (v) hardware breach confinement. For that, this architecture is equipped with symmetric encryption, key management mechanisms and strictly regulated protected sections (*enclaves*) for software modules. Note that, even though Sancus guarantees some security properties, research is still ongoing to make these mechanisms more secure, e.g. against side-channel attacks [14].

Finally, other proposals attempt to develop new architectures to dynamically enforce security policies. Among them, micro-tagged architectures [22,12,11] enrich each assembly instruction by tagging it with a pointer to a (security) policy to be checked at run-time.

5 Open Challenges

In the above sections, we reviewed some of the recent advances in secure compilation and sketched some relevant formal techniques that can be currently used to achieve it. Indeed, the field of secure compilation is still young, and there are many open problems have not been tackled yet. This section briefly mentions some of them and gives some hints about future research directions in the field.

A first open problem is that of *devising novel secure compilation principles*. Even though many principles have been recently added to the literature, concerning both robust and non-robust secure compilation, there is still the need of new secure compilation principles, possibly tailored to very specific hyperproperties (e.g. variants of non-interference or protecting from side-channel attacks). A key point is to make these new principles suitable to be (rather) easily proved even for realistic compilers. For that they must be provable in a modular fashion, e.g. in the style of [13] that assumes some notion of compiler correctness and builds the security proof on that.

This leads us to the second problem, i.e. that of developing *proof and verification techniques* for secure compilers. On the one hand, it is particularly important to develop novel proof techniques that help the designer of the compiler in showing that her compiler is indeed secure. To be applicable in realistic cases, these new proof techniques must not only be modular (as said above) but also incremental, in the sense that proofs must require minimal adaptations as compilers evolve. On the other hand, to foster the development of secure compilers also for mainstream languages, new *fully automatic, push-button verification*

techniques should emerge. Indeed, the existence of such techniques, for example along the lines of translation validation, may allow to bring secure compilation also to real-world and fully fledged compilers.

Bringing secure compilation into real-world compilers also poses the challenge of separate compilation. The hurdle with separate compilation is that each (malicious) component interacts with the others, built from different source languages by different compilers (hence, with different security guarantees).

Finally, from a theoretical viewpoint, a last open problem consists in establishing the exact family of security properties that full abstraction preserves and under which conditions. Actually, first steps in this direction (at least for safety hyperproperties) were done by Patrignani and Garg [33], but it is not clear yet under which conditions full abstraction may preserve other classes of hyperproperties.

6 Conclusions

This brief tour of formally secure compilation highlighted recent advances in secure compilation.

We first reviewed what it is called *non-robust* secure compilation, that is a simple notion of secure compilation taking into account *passive attackers* that can just observe some of the actions performed by the programs, and its relation with compiler correctness. Also, we rephrased well-known results in the literature, so to make precise the claim that correct compilers *may* preserve security properties and we summarised recent and relevant results that applies when correct compilers do not preserve security properties.

In the third section, we defined the notion of *robust* secure compilation by extending the previous concept to cope with *active attackers*, i.e. contexts (programs with a hole). After giving a short motivation on why the classical principle of full abstraction might not be fully appropriate in some situations, we briefly discussed emerging principles for robust secure compilation and we sketched some ideas on how they can be statically enforced.

Talking of enforcement in Section 4 we reported some mechanisms that allow enforcing security properties directly at the low level, usually at run-time.

Finally, we discussed open problems and known challenges in secure compilation, especially concerning the need of new robust secure compilation principles, (automatic) verification and proof techniques for real-world, off-the-shelf compilers, and full abstraction guarantees.

The picture that emerges is that secure compilation is an expanding and ever growing research field sitting at the intersection of programming languages, program verification and cybersecurity. Given that secure compilation is relatively new and little-known, we hope that this shallow tour may help newcomers to grasp the basics of the field and to navigate the existing literature on the topic.

References

1. CVE-2009-1897. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1897, accessed: 17 Nov 2018
2. CWE-14. https://cwe.mitre.org/data/definitions/14.html, accessed: 17 Nov 2018
3. CWE-733. https://cwe.mitre.org/data/definitions/733.html, accessed: 17 Nov 2018
4. Abadi, M.: Protection in programming-language translations. In: Vitek, J., Jensen, C.D. (eds.) Secure Internet Programming, Security Issues for Mobile and Distributed Objects. Lecture Notes in Computer Science, vol. 1603, pp. 19–34. Springer (1999)
5. Abate, C., de Amorim, A.A., Blanco, R., Evans, A.N., Fachini, G., Hritcu, C., Laurent, T., Pierce, B.C., Stronati, M., Tolmach, A.: When good components go bad: Formally secure compilation despite dynamic compromise. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 1351–1368 (2018)
6. Abate, C., Blanco, R., Garg, D., Hritcu, C., Patrignani, M., Thibault, J.: Journey beyond full abstraction: Exploring robust property preservation for secure compilation. CoRR **abs/1807.04603** (2018)
7. Agten, P., Strackx, R., Jacobs, B., Piessens, F.: Secure compilation to modern processors. In: Chong, S. (ed.) 25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012. pp. 171–185. IEEE Computer Society (2012)
8. Ahmed, A., Blume, M.: Typed closure conversion preserves observational equivalence. In: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008. pp. 157–168 (2008)
9. Ahmed, A., Blume, M.: An equivalence-preserving CPS translation via multi-language semantics. In: Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011. pp. 431–444 (2011)
10. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. **21**(4), 181–185 (1985)
11. de Amorim, A.A., Collins, N., DeHon, A., Demange, D., Hritcu, C., Pichardie, D., Pierce, B.C., Pollack, R., Tolmach, A.: A verified information-flow architecture. Journal of Computer Security **24**(6), 689–734 (2016)
12. de Amorim, A.A., Dénès, M., Giannarakis, N., Hritcu, C., Pierce, B.C., Spector-Zabusky, A., Tolmach, A.: Micro-policies: Formally verified, tag-based security monitors. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. pp. 813–830. IEEE Computer Society (2015)
13. Barthe, G., Grégoire, B., Laporte, V.: Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In: 31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018. pp. 328–343 (2018)
14. Bulck, J.V., Piessens, F., Strackx, R.: Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In: CCS '18: 2018 ACM SIGSAC Conference on Computer & Communications Security, Oct. 15–19, 2018, Toronto, ON, Canada. ACM, New York, NY, USA (2018)
15. Busi, M., Degano, P., Galletta, L.: Translation Validation for Security Properties. CoRR **abs/1901.05082** (2019)

16. Carter, N.P., Keckler, S.W., Dally, W.J.: Hardware support for fast capability-based addressing. SIGPLAN Not. **29**(11), 319–327 (Nov 1994)
17. Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security **18**(6), 1157–1210 (2010)
18. Deng, C., Namjoshi, K.S.: Securing a compiler transformation. In: Rival, X. (ed.) Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9837, pp. 170–188. Springer (2016)
19. Deng, C., Namjoshi, K.S.: Securing the SSA transform. In: Ranzato, F. (ed.) Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10422, pp. 88–105. Springer (2017)
20. Devriese, D., Birkedal, L., Piessens, F.: Reasoning about object capabilities with logical relations and effect parametricity. In: IEEE European Symposium on Security and Privacy, EuroS&P. pp. 147–162. IEEE (2016)
21. Devriese, D., Patrignani, M., Piessens, F.: Parametricity versus the universal type. PACMPL **2**(POPL), 38:1–38:23 (2018)
22. Dhawan, U., Hritcu, C., Rubin, R., Vasilakis, N., Chiricescu, S., Smith, J.M., Jr., T.F.K., Pierce, B.C., DeHon, A.: Architectural support for software-defined metadata processing. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015. pp. 487–502 (2015)
23. D'Silva, V., Payer, M., Song, D.X.: The correctness-security gap in compiler optimization. In: 2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015. pp. 73–87. IEEE Computer Society (2015)
24. Kennedy, A.: Securing the .NET programming model. Theor. Comput. Sci. **364**(3), 311–317 (2006)
25. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Software Eng. **3**(2), 125–143 (1977)
26. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006. pp. 42–54 (2006)
27. Morrisett, J.G., Walker, D., Crary, K., Glew, N.: From system F to typed assembly language. ACM Trans. Program. Lang. Syst. **21**(3), 527–568 (1999)
28. Necula, G.C.: Translation validation for an optimizing compiler. In: Lam, M.S. (ed.) Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000. pp. 83–94. ACM (2000)
29. Noorman, J., Bulck, J.V., Mühlberg, J.T., Piessens, F., Maene, P., Preneel, B., Verbauwhede, I., Götzfried, J., Müller, T., Freiling, F.C.: Sancus 2.0: A low-cost security architecture for iot devices. ACM Trans. Priv. Secur. **20**(3), 7:1–7:33 (2017)
30. Patrignani, M., Agten, P., Strackx, R., Jacobs, B., Clarke, D., Piessens, F.: Secure compilation to protected module architectures. ACM Trans. Program. Lang. Syst. **37**(2), 6:1–6:50 (2015)
31. Patrignani, M., Ahmed, A., Clarke, D.: Formal approaches to secure compilation: A survey of fully abstract compilation and related work. ACM Computing Surveys (2019)
32. Patrignani, M., Clarke, D.: Fully abstract trace semantics for protected module architectures. Computer Languages, Systems & Structures **42**, 22–45 (2015)

33. Patrignani, M., Garg, D.: Secure compilation and hyperproperty preservation. In: 30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017. pp. 392–404. IEEE Computer Society (2017)
34. Patrignani, M., Garg, D.: Robustly safe compilation or, efficient, provably secure compilation. CoRR **abs/1804.00489** (2018)
35. Pierce, B., Sumii, E.: Relating Cryptography and Polymorphism (2000), http://www.kb.ecei.tohoku.ac.jp/~sumii/pub/infohide.pdf
36. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1384, pp. 151–166. Springer (1998)
37. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur. **15**(1), 2:1–2:34 (2012)
38. Shapiro, J.S., Smith, J.M., Farber, D.J.: EROS: a fast capability system pp. 170–185 (1999)
39. Skorstengaard, L., Devriese, D., Birkedal, L.: Reasoning about a machine with local capabilities - provably safe stack and return pointer management. In: Ahmed, A. (ed.) 27th European Symposium on Programming, ESOP 2018, Proceedings. Lecture Notes in Computer Science, vol. 10801, pp. 475–501. Springer (2018)
40. Tsampas, S., El-Korashy, A., Patrignani, M., Devriese, D., Garg, D., Piessens, F.: Towards automatic compartmentalization of c programs on capability machines. In: Workshop on Foundations of Computer Security. pp. 1–14 (2017)
41. Watson, R.N.M., Woodruff, J., Neumann, P.G., Moore, S.W., Anderson, J., Chisnall, D., Dave, N.H., Davis, B., Gudka, K., Laurie, B., Murdoch, S.J., Norton, R., Roe, M., Son, S.D., Vadera, M.: CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. pp. 20–37 (2015)
42. Yang, Z., Johannesmeyer, B., Olesen, A.T., Lerner, S., Levchenko, K.: Dead store elimination (still) considered harmful. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. pp. 1025–1040 (2017)