Proving Cryptographic C Programs Secure with General-Purpose Verification Tools

François Dupressoir

Department of Computing

The Open University

Partially Supported by a Microsoft Research Ph.D. Scholarship.

A thesis submitted for the degree of

Doctor of Philosophy

September 2012 – Corrected in April 2013

Proving Cryptographic C Programs Secure with General-Purpose Verification Tools

François Dupressoir

Department of Computing

The Open University

Partially Supported by a Microsoft Research Ph.D. Scholarship.

A thesis submitted for the degree of

Doctor of Philosophy

September 2012 – Corrected April 2013

Abstract

Security protocols, such as TLS or Kerberos, and security devices such as the Trusted Platform Module (TPM), Hardware Security Modules (HSMs) or PKCS#11 tokens, are central to many computer interactions. Yet, such security critical components are still often found vulnerable to attack after their deployment, either because the specification is insecure, or because of implementation errors.

Techniques exist to construct machine-checked proofs of security properties for abstract specifications. However, this may leave the final executable code, often written in lower level languages such as C, vulnerable both to logical errors, and low-level flaws.

Recent work on verifying security properties of C code is often based on soundly extracting, from C programs, protocol models on which security properties can be proved. However, in such methods, any change in the C code, however trivial, may require one to perform a new and complex security proof.

Our goal is therefore to develop or identify a framework in which security properties of cryptographic systems can be formally proved, and that can also be used to soundly verify, *using existing general-purpose tools*, that a C program shares the same security properties.

We argue that the current state of general-purpose verification tools for the C language, as well as for functional languages, is sufficient to achieve this goal, and illustrate our argument by developing two verification frameworks around the VCC verifier.

In the symbolic model, we illustrate our method by proving authentication and weak secrecy for implementations of several network security protocols.

In the computational model, we illustrate our method by proving authentication and strong secrecy properties for an exemplary key management API, inspired by the TPM.

Acknowledgements

Firstly, I thank Andy Gordon for the guidance and support he provided from the very first to the very last day of my PhD. I also would like to thank Jan Jürjens and Bashar Nuseibeh for their more distant, but still active supervision efforts: through no fault of their own, we did not speak as much as we should have; I regret this and wish to thank them in particular for their availability, and sometimes active pursuit of conversations, which were always insightful and helped me see my research in a new light on many an occasion.

I would like to thank my wife Heather for her support during the past four years, and in particular for insisting that I take (most) weekends off. Thanks are due to Steve Temple, from the Impington windmill, for providing me with a productive way of doing so, and for our many stimulating conversations.

Finally, I wish to thank Microsoft Research for its support through the PhD Scholarship programme, but also for the very many opportunities, for providing me with a roof to work under in Cambridge, and for the time and expertise of its researchers and staff. In particular, thanks are due to members of the Constructive Security and to the Programming Principles and Tools groups for welcoming me as one of their own. I also wish to thank the verification team in Aachen (with particular thanks to Ernie Cohen, Thomas Santen and Stephan Tobies) and the RiSE team in Redmond (especially Michał Moskal and Patrice Godefroid) for the great discussions, but also the monumental efforts involved in developing and maintaining VCC, and convincing development teams that formal methods are a good thing. I hope my experiments with your tools helped in some small way.

Declarations

Declaration of Originality

Although some of the material presented here was published with co-authors, and I use the pronoun "we" throughout this dissertation, the ideas presented herein are mine, except where explicitly indicated otherwise. In particular, the informal presentation of VCC semantics shown in Section 1.3.2 is based on a formal Coq development by David Naumann (discussed in [Dupressoir, Gordon, Jurjens, and Naumann, 2011]).

Publications

- The Coq formalisation of symbolic cryptographic invariants presented in Sections 2.2 and 2.3 was previously illustrated, but not fully developed, at the workshop on Formal Aspects of Security and Trust (FAST) in 2011 and in the corresponding technical report [Aizatulin, Dupressoir, Gordon, and Jürjens, 2011a];
- a preliminary version of Chapter 3 was published at the Computer Security Foundations Symposium (CSF) in 2011 [Dupressoir et al., 2011], an updated version of which is currently in submission to a special issue of the Journal of Computer Security (JCS);
- a preliminary version of Chapters 4 and 5 were presented during the joint session of the selective workshops on Analysis of Security APIs (ASA), and on Formal and Computational Cryptography (FCC) in 2012; a reduced version of those chapters are currently under submission.

Contents

Contents v

1 Introduction 1
- 1.1 Problem Statement . . . 1
  - 1.1.1 Thesis and Contributions . . . 3
- 1.2 Outline . . . 4
- 1.3 Verification Tools . . . 5
  - 1.3.1 Type Systems . . . 5
  - 1.3.2 Verification of C Programs . . . 6

2 Formalising Symbolic Security 15
- 2.1 Symbolic Models of Cryptography . . . 15
  - 2.1.1 Invariants on Cryptographic Structures . . . 17
- 2.2 A Coq Framework for Symbolic Cryptographic Invariants . . . 18
  - 2.2.1 Usages and Events . . . 18
  - 2.2.2 Invariants . . . 19
- 2.3 Protocol-Specific Invariants . . . 23
  - 2.3.1 Example: Authenticated RPC . . . 23
  - 2.3.2 A High-Level Protocol Description Language . . . 26
  - 2.3.3 Other Examples . . . 30
- 2.4 Discussion and Related Work . . . 33

3 Verifying Symbolic Security of C Code 35
- 3.1 A VCC Framework for Symbolic Cryptography . . . 36
  - 3.1.1 Symbolic Cryptographic Invariants . . . 36
  - 3.1.2 The Representation Table . . . 37
  - 3.1.3 The Hybrid Wrappers . . . 39
- 3.2 Attack Programs . . . 43
- 3.3 An Example Security Theorem: authenticated RPC . . . 45

3.3.1 Verifying Protocol Code . . . 46
3.3.2 Verifiability of the Adversary Shim . . . 47
3.4 Summary of Empirical Results . . . 50
3.4.1 Results . . . 50
3.4.2 Performance . . . 51
3.5 Related Work and Discussion . . . 52
3.5.1 Performance Improvements and Extensions to Stateful Systems . . . 53
3.5.2 Verifying Java Implementations . . . 53
3.5.3 About Code Generation . . . 54
3.5.4 Verifying C Implementations . . . 54
3.5.5 Symbolic and Computational Models of Cryptography . . . 56

4 Computational Security for C Programs **57**
4.1 Computational Models of Cryptography . . . 57
4.1.1 Cryptographic Properties . . . 57
4.1.2 Computational Security for F# Programs . . . 59
4.2 Overview . . . 60
4.2.1 Challenges . . . 60
4.2.2 Our Approach . . . 61
4.3 Probabilistic Program Semantics and Indistinguishability . . . 62
4.3.1 Notations for Possibilistic Semantics . . . 62
4.3.2 Observational Determinism . . . 63
4.3.3 Derandomized Probabilistic Semantics . . . 63
4.3.4 Indistinguishability . . . 64
4.3.5 VCC and Abstract Contracts . . . 64
4.4 Simulation . . . 65
4.4.1 Definition of Simulation . . . 66
4.4.2 Simulation as a Contract . . . 66
4.5 Discussion . . . 67
4.5.1 Related Work . . . 68
4.5.2 Ideal Simulation and Modularity . . . 68
4.5.3 Secure Implementations of Primitives . . . 69
4.5.4 Reference Implementations: From F7 to VCC . . . 69
4.5.5 Other Uses of Simulation in Software Verification . . . 70
4.5.6 Reference Languages for Reference Implementations . . . 71

5 Computational Security of a Key Management System **73**
5.1 The Device . . . 73

5.1.1 Instruction Set . . . 74
5.1.2 Expected Security Properties . . . 75
5.1.3 Cryptographic Assumptions . . . 76
5.1.4 Concrete Security for the Device . . . 78
5.2 Proving Computational Security Properties of the Device . . . 81
5.2.1 Security of the Reference Implementation . . . 82
5.2.2 Observations and Simulation . . . 85
5.2.3 Security of the System Implementation . . . 98
5.3 Discussion . . . 100
5.3.1 On the Size of Specifications and Systems . . . 101
5.3.2 Security for Under-Specified Systems . . . 102

6 Conclusion **103**
6.1 From Secure Model to Usable Specification . . . 104
6.2 Threat Models, Shims and Observation Functions . . . 104
6.3 On Usability and Applicability . . . 105

A Full Coq Framework for Symbolic Cryptography **107**

B Observation Functions and Simulation Contracts - Unabridged **111**
B.1 Encryption of User-Provided Plaintexts . . . 111
B.2 Key Derivation . . . 113
B.3 Encryption of Device Objects . . . 114

C Over-Approximating P.P.T. Adversaries with Non-Determinism **117**

Bibliography **119**

Chapter 1

Introduction

1.1 Problem Statement

Security protocols, such as SSL, TLS or Kerberos, and security devices such as the Trusted Platform Module (TPM), Hardware Security Modules (HSMs) or PKCS#11 tokens, are central to many computer interactions, from providing a secure environment when booting a computer to ensuring the confidentiality of credit card and private information in e-commerce or online banking applications. Yet, these security critical components are still often found vulnerable to unauthorized interactions after their deployment, either because the specification itself is insecure [CVE, 2009; Bruschi, Cavallaro, Lanzi, and Monga, 2005, for example], or because of flaws introduced when implementing a specification [CVE, 2012, for example].

It is therefore essential to develop methods and tools to formally define and reason about the security properties of such cryptographic systems and their implementations. In recent years, techniques have been developed to construct machine-checked, or even automated, proofs of security properties for security protocols and devices. Most of these proof techniques focus on specifications, with a more recent shift towards performing the proofs on executable specifications, for example written in ML. However, this may leave the final executable code, often written in lower level languages such as C, vulnerable both to logical errors (when the code does not implement its specification), and low-level bugs (such as buffer or integer overflows).

Some attention has been given to the verification of cryptographic security properties directly on C

code.

- Some of the considered approaches to the problem aim at dynamically checking for security flaws [Godefroid and Khurshid, 2002; Godefroid, Klarlund, and Sen, 2005; Jeffrey and Ley-Wild, 2006], and do not in general provide formal security guarantees for whole programs.
- Many of the static approaches focus on *extracting* verifiable models from C programs, some as Horn clauses [Goubault-Larrecq and Parrennes, 2005], some as programs in some process calculus [Aizatulin, Gordon, and Jürjens, 2011b, 2012]. If extraction is a promising approach, we believe that it is insufficient on its own. First, any change in the C implementation, however trivial, may cause changes in the extracted model that force the entire security proof to be done from scratch. Second, fully automated techniques are often not very adaptable to new constructs (either on the program side or the security side), as they require changes in the tool itself to cover them, and provide very little control to the end user.
- One particular tool, Pistachio [Udrea, Lumezanu, and Foster, 2008] focuses on proving that a given C program follows a rule-based description of its intended behaviour, extracted from an RFC document. However, RFC documents are often very complex and informal, and provide no formal security guarantees. In addition, the verification tool itself makes unsound abstractions of the C program to improve automation.

Independently of these specialised developments in proving security properties of C code, tools and techniques have been developed that allow the verification of safety and functional properties of real-world C programs [Cohen, Dahlweid, Hillebrand, Leinenbach, Moskal, Santen, Schulte, and Tobies, 2009a; Correnson, Cuoq, Puccetti, and Signoles, 2010]. Such general-purpose tools have recently been shown to be applicable to real-world systems, from avionics [Delmas and Souyris, 2007] to operating systems [Baumann, Beckert, Blasum, and Bormer, 2009; Klein, Elphinstone, Heiser, Andronick, Cock, Derrin, Elkaduwe, Engelhardt, Kolanski, Norrish, Sewell, Tuch, and Winwood, 2009; Shi, He, Zhu, Fang, Huang, and Zhang, 2012] and virtual machine hypervisors [Leinenbach and Santen, 2010]. Their applications to security, however, has so far always stopped where cryptography starts.

Our goal is therefore to develop frameworks in which security properties of cryptographic systems can be formally proved, and that can also be used to soundly verify that a C program has these same security properties. Initially, we focus on performing those security proofs in the symbolic model of

cryptography, and later move on to the computational model of cryptography, providing, for the first time, verification techniques to prove imperfect (probabilistic) observational equivalence properties of C programs.

1.1.1 Thesis and Contributions

Our thesis is that the current state of general-purpose verification tools for the C language, as well as for functional languages, is sufficient to achieve this goal. We illustrate this argument with a particular C verification tool, VCC, used in combination with the Coq proof assistant (for proofs in the symbolic model), and the F7 refinement type-checker for F# (for proofs in the computational model). Both of the verification methodologies presented in this dissertation are designed to allow sharing of the security proof between several implementations, by first proving security on some form of specification (depending on the chosen model of cryptography) and then using a general-purpose verifier to prove that the C implementation follows the specification.

We make the following original contributions:

- We generalise and formalise in Coq the notion of *invariants on cryptographic structures* [Bhargavan, Fournet, and Gordon, 2010]. Our formalisation is modular, reducing the symbolic security of classes of protocols to simple inductive properties on their use of cryptography.
- We show how symbolic security proofs in this framework can be used, in combination with an existing general-purpose verifier to soundly prove secrecy and integrity properties of C programs, applying it in particular to sample security protocols.
- We define a notion of program simulation for (discrete) probabilistic programs, and show how it can be proved with an existing general-purpose verifier. Our notion of simulation allows a tool designed for functional verification to prove probabilistic program equivalence properties.
- By combining existing techniques to prove computational security properties of functional programs with our notion of simulation and a C verification tool, we develop a methodology to prove, for the first time, computational security properties (including equivalence-based properties) of branching, stateful C programs.
- We apply our technique to an exemplary key management system, inspired by the upcoming

Trusted Platform Module (TPM) standard, which includes a reference implementation in C.

Additionally, all code, annotations and formal proofs discussed in this dissertation are available on demand. Some updates may be made to keep the proofs up to date with VCC development and bug fixes.

1.2 Organization of the Dissertation

The rest of this introduction (Chapter 1) is devoted to a brief overview of the verification tools for imperative and functional languages that we rely upon in later Chapters.

In the first part of this dissertation (Chapters 2 and 3), we present techniques that can be used to prove symbolic security properties, under strong assumptions on the cryptographic primitives used. Chapter 2 starts with an overview of symbolic security, including existing models and verification techniques for specifications and implementation, and ending with a novel formalization of symbolic security that permits modular proofs of security on implementations in various languages. Chapter 3 presents the challenges of verifying symbolic security for C programs, and explains how the VCC verifier can be used to prove symbolic security properties of C protocol implementations.

The second part of this dissertation (Chapters 4 and 5) is dedicated to proving security of C programs in the computational model of security, in which cryptographic primitives are no longer assumed to be perfect, and secrecy is information-theoretic rather than formulated logically. In Chapter 4, we present the computational model of cryptography, the associated notions of security, and common proof techniques and tools, and adapt them to systems written in the C language. In Chapter 5, we show how the VCC verifier can be used, in conjunction with the F7 refinement type-checker for F#, to prove security properties, in the computational model of cryptography, of a C program implementing an exemplary key management system. That system, the Device, is intended to exemplify the upcoming TPM's standard cryptographic key management functionalities.

A companion archive containing all code and proofs mentioned in this dissertation is available on demand. An appendix to this dissertation contains code samples that are central to our arguments but are too large to be displayed comfortably within the main text.

1.3 Verification Tools

Since our goal is to use existing program verification techniques and tools, and not to develop our own, we only give here an informal overview of the tools we use, introducing their syntax and discussing what it means for a program to be verified. We do not review all existing general-purpose verification tools.

1.3.1 Type Systems

Type systems are often used to prove or infer properties of functional programs, from basic memory-safety properties (for example, ML's basic type system, due to Hindley, Milner and Damas [Damas and Milner, 1982; Hindley, 1969; Milner, 1978], to functional verification using refinement types [Freeman and Pfenning, 1991], to full-fledged reasoning in higher-order logic using dependent types (such as the Calculus of Inductive Construction (CiC) implemented in Coq [The Coq development team, 2004]).

In this dissertation, we use Coq in Chapter 2, and the F7 refinement type system for F# [Bhargavan, Fournet, and Gordon, 2008] in Chapters 4 and 5. We briefly and informally discuss their syntax and semantics below.

The Coq Proof Assistant

Although Coq has many features, we only make use of its ability to express and prove theorems on inductive datatypes and predicates in a modular way.

Coq does not distinguish between terms and types, but rather sorts them into two categories. The **Prop** sort contains well-formed propositions, which are guaranteed to be true, by Coq's semantics, and predicates (any function that returns a **Prop**) can be defined inductively (using the **Inductive** keyword), or by abstraction (using the **Definition** keyword). The **Type** sort contains datatypes and mathematical structures, which can be defined inductively using the **Inductive** keyword. (Other possibilities exist to define mathematical structures, but we do not use them in this dissertation).

Proving a theorem τ is done by constructing, with the help of *tactics*, a term of type τ, thereby proving that τ has kind **Prop**.

Coq provides a strong module system permitting the parameterisation of definitions and theorems with a set of definitions and proofs, abstracted as simple types, and introduced, in the module signature, using the **Parameter** keyword.

F7: First-Order Refinement Types

Refinement types, in F7, are standard F# types augmented with a first-order refinement formula, given between braces, that constrain the type depending on the values of some program variables. For example, the type of non-negative integers can be defined as the subtype of the type of integers n such that n is non-negative. In F7 notation, this definition is written **type** nat = n: **int** { 0 <= n }.

The refinement formula is an arbitrary first-order formula (with quantifiers), and can also call logical functions and predicates, which are declared using the **function** or **predicate** keyword, and must be specified using axioms.

In addition, the F7 semantics assume the existence of an implicit global shared log of assumptions collected during the execution of the program, and is used to specify some security properties.

Successful type-checking of a function **val** f: x:τ { C } $\rightarrow$ res:τ' { C' } implies, by soundness of the type system, that whenever function f is called and terminates on an argument x of type τ on which the refinement formula C holds in the current log, then it returns a value res of type τ' on which the refinement formula C' (in which both x and res are bound) holds.

For example, the type for addition in nat can be written as follows **val** plus: nat $\rightarrow$ nat $\rightarrow$ nat. When given an implementation of plus, F7 checks that the result is non-negative whenever the arguments are also non-negative, which can then be assumed when type-checking code that calls plus.

1.3.2 Verification of C Programs

If type systems are particularly well-suited to reasoning about functional programs, the lack of abstraction in imperative languages, especially in low-level languages such as C, makes them difficult to apply. In this dissertation, we use the VCC verification tool, which supports a very large subset of the C language, including concurrency. We believe the approaches and techniques described here are applicable to other verification tools such as VeriFast [Jacobs and Piessens, 2008] and Frama-C [Correnson et al., 2010]. Although we make use of some VCC-specific constructs, and in particular its

concurrency model, that may not be directly available in other tools, their use is restricted to the implementation and is often related to practical gains, either in speed or in reducing the number of annotations. We assume the reader is familiar with C syntax, and introduce only novel VCC concepts in the overview below.

The VCC Verifier

VCC is a verification condition generator (VCG) for the C language, soundly abstracting concurrent C programs into a logical model, and using an automatic theorem prover to statically check correctness properties of C code. The correctness properties are expressed in specification, written as function contracts, type invariants and intermediate assertions. The tool is based on a precise model of multithreaded, shared-memory executions of C programs. In this section, we describe the functional specification language, and then discuss the memory model and annotation language used to express and prove properties of C code. We expect the reader is familiar with C syntax.

VCC extends the C syntax with annotations and contracts wrapped in a macro, _(...) that gets stripped away at compile-time, and is checked not to modify the behaviour of the physical code (for example, it cannot write into physical memory locations, or cause any non-termination). As in F7, the specification language is first-order logic with quantification, making use of C's syntax whenever possible.

The VCC Specification Language We start by presenting VCC's specification language, which can be used to describe first-order logic formula, declare uninterpreted function symbols and constrain them with axioms, define mathematical functions and prove theorems. We illustrate this by developing a short library of abstract bytestring operations.

We start by defining a type \Bytes of bytestrings, as a record containing an unbounded array of bytes and a length. One can also define inductive datatypes, or declare abstract types that can later be instantiated, as in F#, with a datatype or a record type.

VCC: Bytestrings

```
_(record \Bytes {
        \Byte bytes[\natural];
        \natural length; })

_(def \bool isBytes(\Bytes s)
  { return ∀\natural i; s.length <= i ⇒s.bytes[i] == (\Byte) 0; })
```

The type \Byte is an alias for the concrete type of unsigned bytes.

We use maps to model unbounded arrays, and later use them to model sets (as boolean maps). Map types are declared with a syntax similar to that of fixed-length arrays, but using the type of their domain in place of the length, and can be defined using λ expressions, as illustrated in the Empty function below. In our case, the bytes field is a map from naturals to bytes, which we restrict, in the refinement predicate isBytes() to be everywhere 0 except within the bytestring's bounds.

In the absence of type refinements in VCC, we use such refinement predicates in function contracts as required. For example, we display below the code for some basic functions on bytestrings, annotated with some simple preconditions, introduced with the **requires** keyword, and postconditions, introduced with the **ensures** keyword. Preconditions restrict the function's domain, and are proved to hold whenever the function is called, whereas postconditions restrict its codomain, and are proved to hold whenever control is returned to the caller.

In postconditions, the keyword **result** is used to refer to the return value. Quantifiers (and the λ) are followed by a list of variable declarations that serve as bindings, and an expression of the appropriate type (boolean for quantifiers, and the map's return type for lambdas). Boolean formulas are expressed using C syntax for conjunction (&&), disjunction (||) and negation (!), augmented with implication and equivalence. As in C, equality tests are denoted with a double equal sign, and the single equal is used for assignment.

VCC: Basic Functions on Bytestrings

```
_(def \natural Length(\Bytes bs)
  { return bs.length; })

_(def \Byte Select(\Bytes bs,\natural i)
  { return bs.bytes[i]; })

_(def \Bytes Update(\Bytes bs,\natural i,\Byte b)
  _(requires isBytes(bs))
  _(requires i < Length(bs))
  _(ensures isBytes(\result))
  { bs.bytes[i] = b;
    return bs; })

_(def \Bytes Empty()
  _(ensures isBytes(\result))
  { return (\Bytes) {
      .length = 0,
      .bytes = λ\natural i; (\Byte) 0 }; })
```

The Empty function illustrates the basic syntax for defining record values inline, by listing field-value bindings.

Many more programming constructs (**if** statements, **switch**-based pattern-matching for inductive datatypes) are available in this specification language. However, loops are prohibited and should be replaced with induction.

VCC automatically proves that **def** functions are pure (and deterministic), total on the domain described by their precondition (this includes

VCC: Logical Form of Length

```
_(abstract \natural Length_l(\Bytes bs))
_(axiom ∀\Bytes bs; Length_l(bs) == bs.length)
```

statically enforcing that pattern-matchings are exhaustive), and terminating. The tool then translates their body into a logical expression, which is passed on to the prover as an axiom, quantified over the function's arguments. For example, the Length() function is translated by VCC into a declaration for the function symbol, along with an axiom defining it, as displayed above in a simplified notation (record operations and pattern matches are internally turned into function calls).

In addition to **def** functions, this small language can be used to define mathematical functions that are not inlined, using the **abstract** keyword. The same proof obligations are discharged, but the function's body is not axiomatised. This is particularly useful for defining abstract interfaces meant to hide implementation details whilst still allowing advanced properties to be used outside of the translation unit.

For example, the contract displayed below can be seen as a lemma, stating that equality on byte strings (on which the refinement predicate holds) is equivalent to equality on their defined domain. Providing a verified implementation of such a function is equivalent to asking VCC to prove that the postcondition is always true.

In this case, we need to give VCC a hint, in the form of a non-trivial return expression that VCC proves true, letting it unwrap the axiomatised definition for Select. When used in conjunction with abstract function definitions, this ability to export theorems and lemmas as well as programming constructs provides a functionality very close to those of Coq modules (without certification, higher-order or polymorphism).

VCC: An Example Theorem

```
_(abstract \bool thm_BytesEquality()
  _(ensures ∀\Bytes bs1,bs2;
      isBytes(bs1) ⇒isBytes(bs2) ⇒
       (bs1 == bs2 <⇒
        (Length(bs1) == Length(bs2) &&
         ∀\natural i;
           i < Length(bs1) ⇒
           Select(bs1,i) == Select(bs2,i))))
  _(returns \true)
  { return ∀\Bytes bs; \natural i; Select(bs,i) == bs.bytes[i]; })
```

The full semantics of the VCC specification language has not been formalised, apart from its implementation in VCC as a translation into the Boogie intermediate verification language. The types and functions displayed here may be used with slightly different names in code in the rest of this dissertation.

Terminology for Concurrent Physical Code Memory blocks are arrays of bytes, but a typed view is imposed in order to simplify reasoning while catering for idiomatic C and standard compilers. The verifier attempts to associate a type with each pointer dereferenced by the program, and imposes

the requirement that distinct pointers reference separate parts of memory. For example two integers cannot partially overlap. Structures may nest as fields inside other structures, in accord with the declared structure types, but distinct values do not otherwise overlap. Annotations can specify, however, the re-interpretation of an integer as an array of bytes, changing the typestate of a union, and so forth.

The declaration of a structure type can be annotated with an invariant: a formula that refers to fields of an instance **this** of the structure. (We often say "invariant" for what are properly called "type invariants".) Invariants need not hold, for example, of uninitialized objects or objects being modified so there is a boolean ghost field that designates whether the object is *open* or *closed*: in each reachable state, every closed object should satisfy the invariant associated with its type. *Ghost state* is disjoint from the concrete state that exists at runtime; syntactic restrictions ensure that it cannot influence concrete state.

Useful invariants often refer to more than one object, but the point of associating invariants with objects is to facilitate local reasoning: when a field is written, the verifier only needs to check the invariants of relevant objects, owing to *admissibility* conditions that VCC imposes on invariants. Invariants and other specifications designate an *ownership* hierarchy: if object o_1 owns o_2 then the invariant of o_1 may refer to the state of o_2 and thus must be maintained by updates of o_2. The state of a thread is modeled by a ghost object. An object is *wrapped* if it is owned by the current thread (object) and closed. The owner of an object is recorded in a ghost field **owns**. VCC provides notations **unwrap** and **wrap** to open/close an object, with **wrap** also asserting the invariant.

Ownership makes manifest that the invariant for one object o_1 may depend on fields of another object o_2, so the VCG can check o_1's invariant when o_2 is updated. Since hierarchical ownership is inadequate for shared objects like locks, VCC provides another way to make manifest that o_1 depends on the state of o_2: it allows that o_1 maintains a *claim* on o_2—a ghost object with no concrete state but an invariant that depends on o_2. Declaring a type to be claimable introduces implicit ghost state used by the VCG to track outstanding claims. The ghost code to create a claim or store it in a field is part of the annotation provided by the programmer.

The term *invariant* encompasses *two-state* predicates for the before and after states of a state transition. In this way, invariants serve as the rely conditions in a form of rely-guarantee reasoning (see Jones [1981], an early formulation of this concept). Usually two-state invariants are written as ordinary

formulas, using the keyword **old** to designate expressions interpreted in the before state. We say an invariant is *one-state* to mean that it does not depend on the before state.

A thread can update an object that it owns, using **unwrap** and **wrap**. However, in many cases such as locks, having a single owner is too restrictive, and another mechanism is needed to allow multiple threads to update the object concurrently. VCC interprets fields marked **volatile** as being susceptible to update by other threads, in accord with the interpretation of the **volatile** keyword by C compilers. An atomic step is allowed to update a volatile field without opening the object, provided that the object is proved closed and the update maintains the object's two-state invariant (that being the interference condition on which interleaved threads rely). The standard idiom for locks is that several threads each maintain a claim that the lock is closed, so they may rely on its invariant; outstanding claims prevent even the owner from unwrapping the object. Atomic blocks are explicitly marked as such. An atomic block may make at most one concrete update, to be sound for C semantics, but may update multiple ghost fields. We do not use **assume** statements in atomic blocks.

Informal VCC Semantics

Several research papers [Cohen et al., 2009a; Cohen, Moskal, Schulte, and Tobies, 2010] document the VCC system but there is no formal model of its semantics of programs and specifications aside from the VCG itself. To be able to formulate a precise specification of the program properties (in particular security properties) verified by VCC, we sketch a conventional operational semantics, in terms of which we specify what we assume about the verifier.

Leaving aside annotations, a program consists of type and function declarations. The body of a function is a sequence of commands including concurrency primitives: thread fork, send, and receive on named channels (all channels being visible to the adversary and under her complete control). Local variables and function parameters (and returns) have declared types. Although VCC provides facitilities to reason about type casts [Cohen, Moskal, Tobies, and Schulte, 2009b], they complicate proofs greatly, and are not strictly necessary to write programs in C. We exclude them from our theoretical model for simplicity, and rely on VCC to soundly reason about type casting operations when they occur in an implementation.

An *execution environment* consists of a self-contained collection of type and function declarations. For

a given execution environment, a runtime *configuration* takes the form (h, ts, qs) where h is the heap, ts is the thread pool, and qs is a map from channel names to message queues. A *thread state* consists of a command (its current continuation) and a local *store* (that is, a mapping of locals and parameters to their current values); a *thread pool* is a finite list of thread states. Thus threads share the heap and the message queues (which hold messages sent but not yet received). A *run* is a series of configurations that are successors in the transition relation.

The only unusual feature of the operational semantics is our treatment of assumptions, which are usually only given an axiomatic semantics. If there is *any* thread poised to execute the command **assume** p, and the condition p does not hold in the current configuration, then there is no transition—we say there is an *assumption failure*. The only blocking configurations are those where every thread is blocked waiting on an empty channel, or where there is an assumption failure.

Execution of **assume** p (when p holds) or **assert** p (even if p does not hold) are no-ops in our semantics. Assertions are effectively labelled skips, in terms of which we formulate correctness.

Definition 1 (Safe Command)**.** *An* assertion failure *is a run in which there is a configuration where some thread's active command is* ***assert*** p *for some* p *that does not hold in that configuration, and there is no assumption failure at that point. A configuration is* safe *if none of its runs is an assertion failure. A command* c *is* safe *under precondition* p *if for states satisfying* p*, the configuration with that initial state and the single thread* c *is a safe configuration.*

Given our treatment of assumptions, safety means that there is no assertion failure unless and until there is an assumption failure.

VCC works in a procedure-modular way: it verifies that each function implementation satisfies its contract, under the assumption of specified contracts for all functions directly called in the body. We describe this in terms of program fragments, for which we use names ending in .c or .h for code or interface texts, as mnemonic for usual file names; but which may in fact be concatenations of multiple files.

Definition 2 (Verifiable)**.** *We write* `api.h` $\vdash$ `p.c` $\rightsquigarrow$ `q.h` *to mean there exists* `p'.c` *that instruments* `p.c` *with additional ghost code (but no assumptions, and no other changes), and* `q'.h` *that may extend* `q.h` *with contracts for additional functions (but not alter those in* `q.h`*) and type invariants, such that VCC successfully verifies the implementation of each function* f *in* `p'.c` *against the contract for* f *in* `q'.h`*, under hypotheses* `api.h` *and* `q'.h`*; moreover admissibility holds for all the type invariants.*

The most common additions to `p.c` are assertions that serve as hints to guide the prover, but claims

and other ghost code can be added. Assumptions would subvert the intended specifications and could even be unsound. The most common additions to q.h are contracts, as q.h may only provide contracts for functions of interest such as main, whereas the code p.c may include other functions, which for modular reasoning must have contracts.

Note that, given headers p.h and api.h and a program p.c, VCC never successfully verifies p.c against p.h with api.h unless p.c successfully compiles and links against p.h with api.h, which in particular means that p.h · api.h are a closed collection of declarations. An immediate consequence of Definition 2 is the following, where the · operator stands for concatenation (for header files) or linking (for code files).

Corollary 1 (VCC Modularity)**.** *If* p.h ⊢ q.c ⇝ q.h *and* p.h · q.h ⊢ r.c ⇝ r.h *then* p.h ⊢ q.c · r.c ⇝ q.h · r.h*.*

The VCC methodology supports verification conditions for sound modular reasoning, but it is not easy to give a VCC-independent semantics for the verifiability judgement p.h ⊢ q.c ⇝ q.h. Fortunately, for our purposes, it is enough to consider soundness for complete concurrent programs. (We consider modular soundness for sequential programs in Section 4.3.) A complete program is verified as ∅ ⊢ m.c ⇝ main.h, where main.h is the header displayed on the right.

main.h

```
void main()
_(requires \program_entry_point())
_(writes \everything());
```

The \program_entry_point() precondition means that all global objects exist and are owned by the current thread at the beginning of this function, as it is the first function that is called when the process is started. Additionally, the main function is allowed to write in the process's entire memory space.

Assumption 1 (VCC Soundness)**.** *If* ∅ ⊢ m.c ⇝ main.h *then the body of function main in* m.c *is safe for the precondition in* main.h*.*

If the soundness of the VCC methodology has been studied extensively [Cohen et al., 2009b, 2010], its implementation may contain flaws that make it unsound. However, we believe this soundness assumption is reasonable as a first approximation, as VCC is being used extensively, and the rare soundness bugs in its implementation are usually found and corrected rapidly.

Since our techniques are independent from the verifier used, we also believe that current efforts [Ap-

pel, 2011; Paul, 2012; Robert and Leroy, 2012] in providing certified or verified static analysers and verifiers for C programs could, when they are ready, be used to increase confidence in our verification results.

Chapter 2

Formalising Symbolic Security

To prove symbolic security properties of C code, we use the notion of *invariants on cryptographic structures* from Bhargavan et al. [2010], and adapt their type-based techniques to a contract-based verifier. After a brief overview of our symbolic model of cryptography (Section 2.1), this chapter presents a novel modular formalization of the notion in Coq, composed of a protocol-independent framework (Section 2.2), and protocol-specific definitions that can be automatically generated from a high-level description of a class of protocols in a domain-specific language (Section 2.3). We end this chapter with a discussion of existing work on proving symbolic security of models and high-level implementations (Section 2.4), comparing it to Bhargavan et al. [2010]'s techniques and ours.

2.1 Symbolic Models of Cryptography

Symbolic models of cryptography, as initially introduced by Dolev and Yao [1983], model cryptographic messages as elements in a cryptographic algebra whose constructors are the cryptographic primitives and a pairing operator.

In the first part of this dissertation, we model cryptography using the algebra displayed right. Messages can be built from atomic byte strings using the Literal constructor, and then inductively from other messages by using an injective pairing operation Pair and four primitive keyed cryptographic operations.

Symbolic Cryptographic Terms

```
Inductive term: Type :=
  | Literal (bs: bytes)
  | Pair (a: term) (b: term)
  | HMac (k: term) (p: term)
  | SEnc (k: term) (p: term)
  | Sign (k: term) (p: term)
  | Enc (k: term) (p: term).
```

Cryptographic Primitives HMac and SEnc are known as *symmetric* cryptographic primitives, since the key to compute them and reverse them (where applicable) is the same.

Symmetric HMACs (Hash-based Message Authentication Codes, modelled as the HMac constructor) are used to prove the origin of a message (*authentication*), by providing evidence of knowledge of a shared secret used as key. Neither the message nor the key are generally extractable from an Hmac term and its validity is checked by comparing the received MAC to a freshly computed one.

Symmetric encryption (SEnc) is meant to protect the secrecy of a message whilst allowing principals who know the encryption key to retrieve the message by decrypting its encryption. In this model, we assume that encryption also provides authentication guarantees (this is discussed further in Section 2.4).

Asymmetric signature and encryption (Sign and Enc, respectively) provide the same functionalities, but use key pairs, one public key for checking the signature or encrypting plaintexts, and one private key for producing a signature or decrypting ciphertexts. We assume that private and public part of asymmetric key pairs are related using a relation AsymPair that allows us to express the fact that the correct verification/decryption key is used.

Cryptographic Properties and Adversary Model We first need to circumscribe what actions the aversaries we consider can perform. We give them full control over the network (including reordering, rerouting, erasing or injecting messages), let them create new principals, and schedule runs of the protocol between selected principals. In addition, we let the adversary derive new messages from messages she has previously observed over the network.

The adversary can build a new term from any two terms she has previously observed, but we limit the information she can obtain on the subterms of a constructed term she observes. For example, no information can be derived from a HMac or Sign term, and we only let the adversary derive the plaintext from a SEnc or Enc term if she knows a decryption key (that is, the encryption key in the case of symmetric encryption, and any key k' such that AsymPair(k',k), where k is the encryption key used to produce the ciphertext, in the case of asymmetric encryption).

Security Protocols and Security Properties Security protocols are structured sequence of messages between several principals (usually two or three), whose goal is to securely exchange some informa-

tion. “Securely”, however, may have different meanings depending on the protocol’s security goals. In this first part, we are concerned with two different security goals:

Authentication expresses that a message believed to be from a principal a was indeed produced by principal a;

Secrecy, or *weak secrecy*, expresses that a message (or part of a message) is never derivable by the adversary.

To reason formally about these security properties, we turn to the literature.

2.1.1 Invariants on Cryptographic Structures

Bhargavan et al. [2010] introduce invariants for cryptographic structure as a method to reason about security properties. Progress through a protocol is recorded using a log of events, similar to the event predicates used by Paulson [1998] or Blanchet [2001], or the history predicates used by Cohen [2003]. Log-dependent predicates indicate whether cryptographic structures, or terms, can appear in the system, or may be known to the adversary. Event correspondences, as introduced by Woo and Lam [1993], can be used to express integrity, proving that end events happen after a corresponding begin event. Secrecy properties can be expressed as correspondences, either by making sure that the begin event is never logged (therefore preventing the end event from happening), or using the begin event to model a compromise action by the adversary.

Events that can be logged include key creation for a particular purpose, and application events such as, say, the fact that principal a intends to send a term x to principal b. Invariants then relate the log-dependent predicates to the presence of particular events in the log. For example, an invariant could say that, if k is a key shared only between principals a and b (we assume unidirectional keys in this description), and it happens that the term HMac k x is seen in the system, then it must be that a intended to send message x to b.

To benefit from these invariants when verifying a program, log-dependent predicates are used to define pre and postconditions on the cryptographic primitives, ensuring that they can only be used by honest protocol participants in ways that preserve the declared invariants. For example, to ensure that the invariant informally described above is maintained by the HMac operation, this function is

annotated with a precondition stating that there are principals a and b such that the key used in the function call is shared between a and b, and a intends to send the message passed as argument to b. Similarly, pre and postconditions on network operations guarantee that protocol participants never violate the invariants by sending non-public terms to the adversary.

The fact that these ideas have already been used to prove security properties of executable code, albeit in F#, makes them a good candidate for our goals. However, many other techniques exist for verifying security protocols, and indeed even for implementations. We discuss them in Section 2.4.

2.2 A Coq Framework for Symbolic Cryptographic Invariants

In this section, we formalize in Coq our protocol-independent theory of cryptographic invariants. The protocol-specific definitions are in this section abstracted as module interfaces, that need to be instantiated before the full Coq proof for a given protocol can be checked. An example of such instantiations is shown and discussed in Section 2.3.

2.2.1 Usages and Events

The general framework is first parameterised by protocol-specific declarations for cryptographic usages, that indicate how a key or nonce can be used by the protocol, and protocol events that record progress through the protocol. These are expected to provide the module interface shown right, requiring a collection of primitive usages for each cryptographic primitive, as well as for nonces, and a collection of protocol events.

Signature: Protocol Usages and Events

```
Module Type ProtocolDefs.
  Parameter nonce_usage: Type.
  Parameters hmac_usage senc_usage: Type.
  Parameters sign_usage enc_usage: Type.
  Parameter pEvent: Type.
End ProtocolDefs.
```

The primitive usage declarations are then augmented with an AdversaryGuess usage for public atoms. This usage models principal names, protocol constants and adversary-chosen values. A distinction is made between private and public keys that share the same primitive usage type.

Similarly, the type of protocol events is augmented with a New event, registering that a given term is meant to be used with a specific usage (either an adversary guess or a primitive usage); and an AsymPair event, registering that two terms are part of the same asymmetric key pair (this dynamically builds the relation discussed previously as key pairs are generated).

In addition to augmenting the protocol-specific definitions, the functor displayed right also defines a type for event logs, here as simple sets of events, as well as shorthand notations for log membership, the inclusion order on logs, and a notion of stability of log-dependent predicates under addition of events to the log (that is, protocol progress). Finally, it also provides a general well-formedness condition stating that any given term can have at most one usage, and that the components of asymmetric keypairs must have the same primitive usage, and use the appropriate usage constructor.

General Usages and Events

```
Module Defs (PD: ProtocolDefs).
  Include PD.

  Inductive usage: Type :=
    | AdversaryGuess
    | Nonce (nu: nonce_usage)
    | HmacKey (hu: hmac_usage)
    | SEncKey (eu: senc_usage)
    | SignKey (su: sign_usage)
    | VerfKey (su: sign_usage)
    | EncKey (eu: enc_usage)
    | DecKey (eu: enc_usage).

  Inductive event: Type :=
    | New (t: term) (u: usage)
    | AsymPair (pk: term) (sk: term)
    | ProtEvent (pe: pEvent).

  Definition log: Type := set event.
  Definition Logged (e: event) (L: log): Prop := set_In e L.
  Definition LoggedP (e: pEvent) (L: log): Prop := Logged (
        ProtEvent e) L.
  Definition leq_log (L L': log): Prop := ∀ e, Logged e L → Logged
        e L'.

  Definition Stable (P: log → Prop) := ∀ L L',
    leq_log L L' → P L →
    P L'.
  Definition WF_log (L: log): Prop :=
    (∀  t u u',
       Logged (New t u) L →
       Logged (New t u') L → u = u') ∧
    (∀  pk sk,
       Logged (AsymPair pk sk) L →
       ((∃  su,
            Logged (New pk (VerfKey su)) L ∧
            Logged (New sk (SignKey su)) L) ∨
        (∃  eu,
            Logged (New pk (EncKey eu)) L ∧
            Logged (New sk (DecKey eu)) L))).
End Defs.
```

2.2.2 Invariants

Given these generalised and parameterised definitions for log operations, the user can now define protocol-specific invariants on logs and terms, which include:

1. an additional well-formedness invariant on the log, LogInvariant, meant to represent conditions enforced by the key management infrastructure (for example, unidirectionality of keys),
2. a release condition *prim*Comp for each kind of primitive usage, and proofs that the release conditions are stable, and
3. a payload condition can*Prim* for each kind of primitive key usage (excluding nonces), also equipped with proofs of stability.

The simplified module signature shown below formalizes those requirements for symmetric cryptography. (Asymmetric cryptography is treated in the same way.)

Signature: Protocol Invariants

```
Module Type ProtocolInvariants (PD: ProtocolDefs).
  Include Defs PD.
  Parameter LogInvariant: log → Prop.

  (* Nonce Predicate *)
  Parameter nonceComp: term → log → Prop.
  Parameter nonceComp_Stable: ∀ t, Stable (nonceComp t).

  (* HMAC Predicates *)
  Parameter hmacComp: term → log → Prop.
  Parameter hmacComp_Stable: ∀ t, Stable (hmacComp t).

  Parameter canHmac: term → term → log → Prop.
  Parameter canHmac_Stable: ∀ k p, Stable (canHmac k p).

  (* SEnc Predicates *)
  Parameter sencComp: term → log → Prop.
  Parameter sencComp_Stable: ∀ t, Stable (sencComp t).

  Parameter canSEnc: term → term → log → Prop.
  Parameter canSEnc_Stable: ∀ k p, Stable (canSEnc k p).
End ProtocolInvariants.
```

The Level predicate

Our security invariants are expressed using a Level predicate, that encodes both the Pub and Bytes originally used by Bhargavan et al. [2010]. Although we use the words low and high, they should not be interpreted as integrity or confidentiality levels, as this Level predicate in fact encodes the derivability rules that are often used to describe how cryptography can be used by honest and dishonest protocol participants. We say that a term t is Low in log L (denoted Level Low t L) whenever it may be made known to the adversary without compromising the protocol's security objectives. We say that a term t is High in log L whenever it can be derived by any honest or dishonest protocol participant (including the adversary). Intuitively, a term is truly secret if it is not Low in the current log. In Chapter 3, we explain in more details how the security of an implementation can be proven essentially by guaranteeing that only Low terms are in fact made available to the adversary.

From the informal definition above, it is clear that we want both Level Low t and Level High t to be stable under log growth (terms, once constructed or given to the adversary, cannot be removed from the system), and that we want, in all logs, any Low term to be High (since the adversary cannot discover terms without their being constructed first). The inductive definition of the Level predicate is broken down below to ease its presentation.

Base Cases: Level for Literals The simplest base case expresses the fact that adversary guesses (terms that have been registered with the AdversaryGuess usage) are Low (and therefore also High) since they may become known to the adversary at any time.

Level Predicate: AdversaryGuess

```
(* AdversaryGuesses are always Low *)
| Level_AdversaryGuess: ∀ l bs L,
    Logged (New (Literal bs) AdversaryGuess) L →
    Level l (Literal bs) L
```

Level Predicate: Nonces and Keys

```
(* Nonces are Low when nonceComp holds *)
| Level_Nonce: ∀ l bs L nu,
  Logged (New (Literal bs) (Nonce nu)) L →
  (l = Low → nonceComp (Literal bs) L) →
  Level l (Literal bs) L
(* HmacKeys are Low when hmacComp holds *)
| Level_HmacKey: ∀ l bs L hu,
  Logged (New (Literal bs) (HmacKey hu)) L →
  (l = Low → hmacComp (Literal bs) L) →
  Level l (Literal bs) L
(* SEncKeys are Low when sencComp holds *)
| Level_SEncKey: ∀ l bs L su,
  Logged (New (Literal bs) (SEncKey su)) L →
  (l = Low → sencComp (Literal bs) L) →
  Level l (Literal bs) L
(* SigKeys are Low when signComp holds *)
| Level_SigKey: ∀ l bs L su,
  Logged (New (Literal bs) (SignKey su)) L →
  (l = Low → sigComp (Literal bs) L) →
  Level l (Literal bs) L
(* VerfKeys are always Low *)
| Level_VerKey: ∀ l bs L su,
  Logged (New (Literal bs) (VerfKey su)) L →
  Level l (Literal bs) L
(* EncKeys are always Low *)
| Level_EncKey: ∀ l bs L eu,
  Logged (New (Literal bs) (EncKey eu)) L →
  Level l (Literal bs) L
(* DecKeys are Low when encComp holds *)
| Level_DecKey: ∀ l bs L eu,
  Logged (New (Literal bs) (DecKey eu)) L →
  (l = Low → encComp (Literal bs) L) →
  Level l (Literal bs) L
```

Keys and nonces are always High, and can only become Low when their release condition holds, or if they are given a public key usage (asymmetric encryption key, or asymmetric signature verification key). These base rules do not apply to constructed terms that may be used as keys or nonces.

Protocols can still make use of such non-atomic keys (the level of which is determined using the appropriate inductive rule as defined below), at the cost of potentially more complex protocol-specific proofs. This point is discussed further in Sections 2.3 and 3.1.3.

Constructed Terms We now discuss the definition of Level for constructed terms, starting with pairs and moving on to cryptographic primitives excluding asymmetric operations, for which the rules are very similar to the symmetric cases. The full definition for the Level predicate is displayed in Appendix A, along with the other definitions and signatures for the general framework.

As expected, a pair is Low (resp. High) if and only if both of its components are Low (resp. High). Both cases can be folded into a single one, universally quantified over the level, as shown right.

Level Predicate: Pairing

```
(* Pairs are as Low as their components *)
| Level_Pair: ∀ l t1 t2 L,
  Level l t1 L →
  Level l t2 L →
  Level l (Pair t1 t2) L
```

For terms resulting from the application of a keyed primitive, we consider two cases, the first when the term is built by honest protocol participants, and therefore follows the protocol's specification; and the second when it is built by the adversary, who can apply the constructor to

Level Predicate: HMACs

```
(* Honest Hmacs are as Low as their payload *)
| Level_Hmac: ∀ l k m L,
  canHmac k m L →
  Level l m L →
  Level l (HMac k m) L
(* Dishonest Hmacs are Low *)
| Level_Hmac_Low: ∀ l k m L,
  Level Low k L →
  Level Low m L →
  Level l (HMac k m) L
```

any two terms it knows. In the first case, and since HMACs do not necessarily provide secrecy, the rule states that the HMAC term is as low as its payload, provided the payload condition holds. In the second case, any two Low terms can be used to compute an HMAC at any level.

Similarly, for symmetric encryptions, two cases need to be distinguished. The first case, modelling encryptions performed by honest participants, require the payload conditions to hold, and turns a plaintext of any level into a ciphertext that can be safely given to the adversary, and therefore has both level Low and level High. The second case, again, computes a ciphertext of any level from a Low plaintext and a Low key. The case for asymmetric encryption is also omitted from this detailed description: although it differs somewhat from the symmetric case, it is still very similar.

Level Predicate: Sym. Encryption

```
(* Honest SEncs are Low *)
| Level_SEnc: ∀ l l' k p L,
  canSEnc k p L →
  Level l' p L →
  Level l (SEnc k p) L
(* Dishonest SEncs are Low *)
| Level_SEnc_Low: ∀ l k p L,
  Level Low k L →
  Level Low p L →
  Level l (SEnc k p) L
```

Generic Invariants

From this definition of the Level predicate, and without knowing more about the protocol-specific invariants than what is expressed in the protocol invariant signature, some simple but powerful general invariants can be proved to hold.

Generic Invariants: Low $\subseteq$ High

```
Theorem Low_High: ∀ t L,
  Level Low t L → Level High t L.
```

The simplest of these invariants is the fact that all Low terms are also High. In other words, all the terms that the adversary can produce by applying its deduction rules can also be produced by applying the adversary rules augmented with the rules for honest protocol participants. In previous versions of the methodology, the fact that *Pub* implied *Bytes* on all terms was a rule in the system, but the Level predicate is set up in such a way that this becomes a theorem.

A second natural invariant is that the Level predicate is in fact positive in the log. This is obviously desirable, since we want that all terms known to any principal in the system are High at any time (and all terms known to the adversary are Low), and we assume that the principals have perfect memory.

Generic Invariants: Level is stable

```
Theorem Level_Stable: ∀ l t L L',
  leq_log L L' → Level l t L →
  Level l t L'.
```

Finally, the basic structural constraints on the Level predicate give rise to strong general inversion properties which, when considered in conjunction with stronger protocol-specific invariants, are central in proving security properties.

Generic Invariants: Level inversion

```
Theorem LowHmacKeyLiteral_Inversion: ∀ L k hu,
  GoodLog L →
  Logged (New (Literal k) (HmacKey hu)) L →
  Level Low (Literal k) L →
  hmacComp (Literal k) L.

Theorem HMac_Inversion: ∀ L l k p,
  Level l (HMac k p) L →
  canHmac k p L ∨ Level Low k L.
```

For example, the LowHmacKeyLiteral_Inversion theorem displayed right states that any Low literal term that is used as an HMAC key in a well-formed log has to be compromised. Similarly, seeing an HMAC term at any level indicates that the MAC was built either by an honest protocol participant, in which case the payload condition must hold; or by the adversary, in which case the key must be Low (as well as the payload).

2.3 Protocol-Specific Invariants

The framework presented above is parameterised by protocol-specific definitions, encoding the various usages made of nonces and cryptographic primitives in runs of a protocol. In this section, we give an example of such a protocol-specific theory, before presenting a small language for giving higher-level description of protocol-specific usages and invariants and generating ready-to-prove skeletons for the corresponding Coq theories.

2.3.1 Example: Authenticated RPC

We now show the usage type and usage predicate definitions for a small Authenticated Remote Procedure Call (Authenticated RPC) protocol, shown below. In this protocol, the client a and server b are assumed to pre-share a secret symmetric key kab, only used for sessions where a is the client and b is the server. The client sends a request req, paired with the HMAC of req under kab. Upon receiving the request and checking its MAC, the server runs its application service on the req, producing a response resp, and replies with resp, paired with the HMAC of the pair (req, resp) under kab. Unique tags are used to prevent the use of response messages as requests. The desired authentication properties are expressed below as event correspondences, where an assertion only succeeds if the asserted event has been logged along the current execution trace.

Protocol Example: an Authenticated RPC Protocol

```
a      : Log(Request(a, b, req))
a → b : req | hmac(kab, Literal([TagRequest])| req)
b      : assert(Request(a, b, req))
b      : Log(Response(a, b, req, resp))
b → a : resp | hmac(kab, Literal([TagRequest])| req | resp)
a      : assert(Response(a, b, req, resp))
```

A-RPC: Additional Parameters

```
Parameter TagRequest: byte.
Parameter TagResponse: byte.
Parameter TagsDistinct: TagRequest <> TagResponse.
```

We parameterise the entire protocol description and its invariants by the request and response tags, additionaly requiring a proof that whichever values are used to instantiate the variables are distinct, as shown on the left. In practice, we can automatically discharge this obligation in VCC when providing the concrete tag values to the implementation (see Section 3.3).

Primitive Usages and Protocol Events

Our Authenticated RPC protocol only uses the HMAC primitive, with a unique key usage, parameterised by the two principals that share it (we choose to have them appear in client-server order). The protocol specification shown above uses two events, Request and Response to model protocol progress, and we add a third, Bad, to model principal compromise, adding partial compromise clauses to the asserted events. The Coq type definitions shown right formalize these remarks, and are proved, using the <: module type annotation, to correctly provide all the definitions expected by our general framework.

A-RPC: Primitive Usages and Protocol Events

```
Module RPCDefs <: ProtocolDefs.
  Definition nonce_usage := False.
  Definition senc_usage := False.
  Definition sign_usage := False.
  Definition enc_usage := False.

  Inductive hmac_usage' :=
    | U_KeyAB (a b: term).
  Definition hmac_usage := hmac_usage'.

  Inductive pEvent' :=
    | Request (a b req: Term.term)
    | Response (a b req resp: Term.term)
    | Bad (p: term).
  Definition pEvent := pEvent'.
End RPCDefs.
```

Log Invariant

For simplicity in this example, we require that only literals are used as keys, turning this simple condition into the log invariant displayed right.

A-RPC: Log Invariant

```
Definition LogInvariant L :=
  ∀ t u, Logged (New t u) L → (∃ bs, t = Literal bs).
```

Usage Conditions

For brevity of later definitions, we first define, for each primitive usage, a log predicate testing whether a given term is associated with that usage. For example, the predicate KeyAB a b k L, displayed right, tests whether the term k has usage HmacKey (U_KeyAB a b) in log L.

A-RPC: Key usage test

```
Definition KeyAB a b k L :=
  Logged (New k (HmacKey (U_KeyAB a b))) L.
```

We want the shared keys to only be released to the adversary if one of the principals that share it is compromised (recorded using the Bad protocol event). This primitive release condition is easily generalized to the generic HMAC usage by existentially quantifying over the primitive usage, as shown on the left.

A-RPC: Release Condition

```
Definition KeyABComp a b L :=
  LoggedP (Bad a) L ∨ LoggedP (Bad b) L.

Definition hmacComp k L :=
  ∃ a, ∃ b, KeyAB a b k L ∧ KeyABComp a b L.
```

The payload conditions are slightly more complicated to infer from the protocol narration. In this case, the shared keys can be used in two different ways by honest principals:

- to compute MACs of tagged requests, on which the Request event holds;
- to compute MACs of tagged request-response pairs, on which the Response event holds.

A-RPC: Payload Condition

```
Definition KeyABPayload a b p L :=
  (∃ req,
    p = Pair (Literal (TagRequest::nil)) req ∧
    LoggedP (Request a b req) L) ∨
  (∃ req, ∃ resp,
    p = Pair (Literal (TagResponse::nil)) (Pair req resp) ∧
    LoggedP (Response a b req resp) L).

Definition canHmac k p L :=
  ∃ a, ∃ b, KeyAB a b k L ∧ KeyABPayload a b p L.
```

All primitive payload conditions we have encountered so far have had a similar shape, that can be divided into a log-independent *format* (here the pairing condition), and a log-dependent *circumstance* (here the fact that Request or Response is in the log). The primitive payload conditions are generalized to generic HMAC usages in the same way as release conditions.

For the authenticated RPC protocol, all other usage conditions are trivially False.

Stability theorems and proofs are omitted from this illustration, but need to be provided for the proof to go through: the proofs in this case simply involve unrolling the definitions, and instantiating the quantifiers until the log inclusion can be applied.

Security theorems

Once the Level predicate instantiated with the protocol-specific definitions shown above, we can prove the following security theorems.

A-RPC: Request Correspondence Theorem

```
Theorem RequestCorrespondence: ∀ a b k req L,
  GoodLog L → KeyAB a b k L →
  Level Low (HMac k (Pair (Literal (TagRequest::nil)) req)) L →
  LoggedP (Request a b req) L ∨
  LoggedP (Bad a) L ∨ LoggedP (Bad b) L.
```

Whenever a valid MAC is observed for a request-tagged message, and the term used to verify the MAC is known to be a valid shared key, we know that either the Request event has been logged on the message (and the MAC was computed by an honest participant), or one of the two principals sharing the key is compromised (and the MAC was computed by the adversary).

A-RPC: Response Correspondence Theorem

```
Theorem ResponseCorrespondence: ∀ a b k req resp L,
  GoodLog L → KeyAB a b k L →
  Level Low (HMac k (Pair (Literal (TagResponse::nil)) (Pair req
      resp))) L →
  LoggedP (Response a b req resp) L ∨
  LoggedP (Bad a) L ∨ LoggedP (Bad b) L.
```

Whenever a valid MAC is observed for a response-tagged pair, and the term used to verify the MAC is known to be a valid shared key, we know that eiter the Response event has been logged on the pair (and the MAC was computed by an honest participant), or one of the two principals sharing the key is compromised (and the MAC was computed by the adversary).

A-RPC: Key Secrecy Theorem

```
Theorem KeySecrecy: ∀ a b k L,
  GoodLog L → KeyAB a b k L → Level Low k L →
  LoggedP (Bad a) L ∨ LoggedP (Bad b) L.
```

Whenever a term known to be a valid shared HMAC key becomes Low, it must be that one of the two principals sharing it is compromised. In cases, such as this one, when the log invariant constrains keys and nonces to be bytestring literals, this theorem is a trivial consequence of Level's inductive definition.

Keyed HMAC Inversion Theorem

```
Theorem KeyedHMac_Inversion: ∀ hu k p L,
  GoodLog L → Logged (New k (HmacKey hu)) L → Level High (
      HMac k p) L →
  canHmac k p L ∨ hmacComp k L.
```

In addition, when this constraint is added to the log invariant, the theorem displayed on the left becomes true, and can be used to greatly simplify other proofs. Its importance is highlighted further in Chapter 3, where it becomes central to providing general cryptographic contracts for the verification of protocol code.

2.3.2 A High-Level Protocol Description Language

The requirement to fulfill several distinct module signatures separately, and in particular the fact that the definition of each cryptographic usage needs to be spread between its declaration as a constructor

in the corresponding primitive usage type, and its proper definition as compromise and payload conditions makes it difficult to develop protocol specifications directly in the formalism presented above. To facilitate this task, we design a small domain-specific language embedded in F#, and presented here under the name T3, the syntax of which is shown below.

T3 Syntax

$t, k ::=$ terms
- x term variable
- *Literal bs* bytestring constant
- *Pair t t* pair
- *Hmac t t* keyed hash
- *SEnc t t* symmetric encryption
- *Sig t t* public key signature
- *Enc t t* public key encryption

$F ::=$ logical formulas
- **true** true constant
- **false** false constant
- $t = t$ term equality
- $!F$ negation
- F && F conjunction
- $F \,||\, F$ disjunction
- $F \Rightarrow F$ implication
- forall (**fun** x $\rightarrow F$) universal quantification
- exists (**fun** x $\rightarrow F$) existential quantification
- $f\ t\ \ldots\ t$ predicate application
- $f\ t\ \ldots\ t\ k$ symmetric key type test
- pubKey $(f\ t\ \ldots\ t)\ k$ public key type test
- prvKey $(f\ t\ \ldots\ t)\ k$ private key type test
- keypair $t\ t$ keypair test
- Logged *ev* log membership test

$U ::=$ nonce and key usages
- nonceLiteral (**fun** n $\rightarrow F$) nonce usage
- hmacKey (**fun** k $\rightarrow F$) (**fun** m $\rightarrow F$) hmac usage
- sencKey (**fun** k $\rightarrow F$) (**fun** m $\rightarrow F$) s. enc. usage
- sigKeyPair (**fun** k $\rightarrow F$) (**fun** m $\rightarrow F$) sig. usage
- encKeyPair (**fun** k $\rightarrow F$) (**fun** m $\rightarrow F$) enc. usage

$D ::=$ declarations
- **let** $f\ x_1\ \ldots\ x_n = F;$ predicate macro
- **let** $f\ x_1\ \ldots\ x_n = U;$ key usage

$l ::=$ security levels
- Low | High

$T ::=$ security theorems
- [< Valid >] **let** $f\ x_1\ \ldots\ x_n = C;$ theorem declaration

where C is F augmented with the following constructions

$C ::=$ derived log predicates
- Level $l\ t$ security level testing
- *prim*Comp t release condition
- can*Prim* $t\ t$ payload condition

where *prim* is one of the crypto primitives.

$P ::=$ **type** event $=$ protocol description
$\quad | ev_1(n_1) \quad \ldots \quad | ev_n(n_n);$
$D \ldots D$
[< LogInvariant >] **let** $f = F;$
$T \ldots T$

A T3 protocol description consists of a declaration of protocol events, followed by a list of predicate and key usage declarations, and ends with a log invariant definition and a list of security theorems.

Predicate and key usage declarations are distinguished by the head symbol: key usages are introduced using one of the five predicate transformers, which take compromise and payload conditions and return a predicate testing whether a term has the defined usage in the current log (for asymmetric keys, the returned predicate tests whether a given term pair has the defined keypair usage in the current log), whereas predicates are simple formulas. The distinction also appears in the types of these expressions, and our embedding in F# forbids ill-typed descriptions where, for example, the

pubKey type test is applied to a symmetric key usage.

Formulas themselves are quantified first-order logic formulas extended with stateful predicates, including the key type tests defined by application of the predicate transformers, a keypair test on terms that check whether two terms are part of the same keypair according to the current log, and a log membership test for user-defined protocol events. We also add a stateless equality test on terms, where terms are members of the same term algebra defined in the Coq framework, but can also be referred to using variables.

The basic type-checking is provided by F#, but is not sufficient to guarantee the desired properties of protocol-specific definitions, as specified in the Coq module signatures shown previously (Section 2.3). In particular, when generating the protocol-specific Coq modules, we also generate proof obligations, in the form of theorems, not only for all declared theorems in the protocol description, but also for all the necessary proof obligations. We also gather all primitive usages, release conditions and payload conditions into their general counterparts and generate the corresponding monotonicity theorems, for the user to prove in Coq.

The following protocol description corresponds to the Authenticated RPC protocol discussed above.

T3 Input for Authenticated RPC

```
type event =
  | Request of term * term * term
  | Response of term * term * term * term
  | Bad of term

let KeyABComp a b =
  Logged (Bad a) || Logged (Bad b)

let KeyABPayload a b m =
  (exists (fun req →
    m = Pair (Literal [|TagRequest|],req) &&
    Logged (Request(a,b,req)))) ||
  (exists2 (fun req resp →
    m = Pair (Literal [|TagResponse|],Pair(req,resp)) &&
    Logged (Response(a,b,req,resp))))

let KeyAB a b =
  hmacKey
    (fun k → KeyABComp a b)
    (fun m → KeyABPayload a b m)

[< LogInvariant >]
let LogInvariant =
  forall2 (fun t (u: usage) →
    u t ⇒ exists (fun bs → t = Literal bs))

[< Valid >]
let KeyedHMAC_Inversion =
  forall4 (fun a b p k →
    GoodLog ⇒ KeyAB a b k ⇒
    Level High (HMac(k,p)) ⇒
    canHmac k p || hmacComp k)

[< Valid >]
let RequestCorrespondence =
  forall4 (fun a b k req →
    GoodLog ⇒ KeyAB a b k ⇒
    Level Low
      (HMac(k,Pair(Literal [|TagRequest|],req))) ⇒
    Logged (Request(a,b,req)) ||
    (Logged (Bad a) || Logged (Bad b)))

[< Valid >]
let ResponseCorrespondence =
  forall5 (fun a b k req resp →
    GoodLog ⇒ KeyAB a b k ⇒
    Level Low
      (HMac (k,Pair(Literal [|TagResponse|],Pair(req,
          resp)))) ⇒
    Logged (Response(a,b,req,resp)) ||
```

```
    (Logged (Bad a) || Logged (Bad b)))

[< Valid >]
let KeySecrecy =
```

```
forall3 (fun a b k →
  GoodLog ⇒ KeyAB a b k ⇒ Level Low k ⇒
  Logged (Bad a) || Logged (Bad b))
```

This description can easily be translated into a Coq theory implementing the module signatures for the protocol-specific definitions, at the cost of some unpleasant variable names.

Events and Usages

The event type is directly translated as the pEvent type definition in Coq.

Key usage declarations are grouped by the predicate transformer used, and the primitive usage types are automatically generated: at this stage, only the name and arity of the declared key usages is used, to directly declare constructors in the corresponding usage type (for example, a key usage declared in T3 using the hmacKey predicate transformer gives rise to a constructor in the hmac_usage type in Coq).

Invariants

Predicates are translated directly as definitions, adding a log argument when they use one of the stateful constructs (log membership test and key usage tests).

The *prim*Comp predicate can then be defined as a disjunction where each disjunct is a conjunction of a key test and the corresponding release condition, with all parameters other than the key and log existentially quantified.

The can*Prim* predicate is defined in a similar way, as a disjunction where each disjunct is a conjunction of a key test and the corresponding payload condition, with all parameters other than the key, payload and log existentially quantified.

The *prim*Comp_Stable and can*Prim*_Stable theorems are automatically declared, but not automatically proved. However, their proof is fairly systematic in all our examples, and heavily relies on the well-formedness condition on logs, that terms can be given at most one usage, allowing us to select the disjunct to be proved in each branch of the hypothesis.

2.3.3 Other Examples

We discuss here some more example protocols, displaying their narration and T3 descriptions, and commenting on their security properties. Not all these protocols are provided with verified implementations in C, but all the stated theorems are proved in Coq.

Encrypted RPC

The following protocol is a variant for the RPC protocol that relies on authenticated encryption instead of pure HMACs to also guarantee some level of secrecy on the requests and responses. The authentication goals, indicated as logging actions and assertions in the following description, are identical to those of the authenticated RPC protocol described in detail above.

Encrypted RPC Protocol Narration

```
a      : Log(Request(a, b, req))
a      : k = keygen()
a → b : senc(kab, req | k)
b      : assert(Request(a, b, req))
b      : Log(Response(a, b, req, resp))
b → a : senc(k, resp)
a      : assert(Response(a, b, req, resp))
```

This protocol requires two different usages for symmetric encryption keys: one for long term keys, pre-shared between the principals (kab above), and one for session keys, which the client generates *for each request*. This latter fact is encoded by parameterising the session key usage with the request as well as the principals, and allows us to prove the desired linking property between request and response in the second correspondence. In addition, we consider two nonce usages, one for requests and another for responses. This is so we can prove some conditional secrecy properties: if the application treats requests and responses as nonces (in particular, if the corresponding plaintext spaces are large enough), then this protocol guarantees that they remain secret unless one of the principals involved is compromised.

All these properties are summarized in the following T3 protocol description.

T3 Input for Encrypted RPC

```
type event =
  | Request of term * term * term
  | Response of term * term * term * term
  | Bad of term

let RequestNComp a b =
  Logged (Bad a) || Logged (Bad b)

let RequestN a b =
  nonceLiteral (fun n → RequestNComp a b)
```

```
let ResponseNComp a b req =
  Logged (Bad a) || Logged (Bad b)

let ResponseN a b req =
  nonceLiteral (fun n → ResponseNComp a b req)

let SessionKeyComp a b req =
  Logged (Bad a) || Logged (Bad b)

let SessionKeyPayload a b req m =
  Logged (Response(a,b,req,m))

let SessionKey a b req =
  sencKey (fun k → SessionKeyComp a b req) (fun m
      → SessionKeyPayload a b req m)

let KeyABComp a b =
  Logged (Bad a) || Logged (Bad b)

let KeyABPayload a b m =
  exists2 (fun req k →
    m = Pair (req,k) &&
    SessionKey a b req k &&
    Logged (Request (a,b,req)))

let KeyAB a b =
  hmacKey
    (fun k → KeyABComp a b)
    (fun m → KeyABPayload a b m)

[< LogInvariant >]
let LogInvariant =
  forall2 (fun t (u: usage) →
    u t ⇒ exists (fun bs → t = Literal bs))
```

```
[< Valid >]
let KeyedHMAC_Inversion =
  forall4 (fun a b p k →
    GoodLog ⇒ KeyAB a b k ⇒
    Level High (HMac(k,p)) ⇒
    canHmac k p || hmacComp k)

[< Valid >]
let RequestCorrespondence =
  forall5 (fun a b kab req k →
    GoodLog ⇒ KeyAB a b kab ⇒
    Level Low (SEnc(kab,Pair(req,k))) ⇒
    (Logged (Request(a,b,req)) &&
     SessionKey a b req k) ||
    (Logged (Bad a) || Logged (Bad b)))

[< Valid >]
let ResponseCorrespondence =
  forall5 (fun a b k req resp →
    GoodLog ⇒ SessionKey a b req k ⇒
    Level Low (SEnc (k,resp)) ⇒
    Logged (Response(a,b,req,resp)) ||
    (Logged (Bad a) || Logged (Bad b)))

[< Valid >]
let KeyABSecrecy =
  forall3 (fun a b k →
    GoodLog ⇒ KeyAB a b k ⇒ Level Low k ⇒
    Logged (Bad a) || Logged (Bad b))

[< Valid >]
let SessionKeySecrecy =
  forall4 (fun a b req k →
    GoodLog ⇒ SessionKey a b req k ⇒ Level Low k ⇒
    Logged (Bad a) || Logged (Bad b))
```

Otway-Rees

Our third example protocol is a variant of the three-party Otway-Rees key exchange protocol by Abadi and Needham [1996]. We assume that keys are created for a fixed purpose. The three parties involved are the initiator i, the responder r, and a trusted key server s, with whom both i and r share a long-term key (ki and kr, respectively). Here again, encryption is assumed to be authenticated.

Otway-Rees(-Abadi-Needham) Protocol Narration

```
i       : ni = fresh()
i → r   : i | ni
r       : nr = fresh()
r → s   : i | r | ni | nr
```

```
s       : kir = keygen()
s       : Log(Initiator(i, ni, kir, r))
s       : Log(Responder(r, nr, kir, i))
s → r   : senc(ki, i | r | kir | ni)| senc(kr, i | r | kir | nr)
r       : Log(Responder(i, nr, kir, r))
r → i   : senc(ki, i | r | kir | ni)
i       : assert(Initiator(i, ni, kir, r))
```

There are two distinct uses for the long-term keys shared with the trusted server: one when the principal is acting as initiator, and the other when the principal is acting as responder. Since nothing in the message format allows to distinguish the two usages, the responder and the trusted server both check that the initiator and responder are distinct principals, which we encode in the usage. In the following T3 description, we fix the usage for the freshly exchanged key, choosing to generate keys that can be used in the authenticated RPC protocol discussed above. Composition of protocols, protocol descriptions, and security proofs is discussed briefly in Section 2.4.

T3 Input for Otway-Rees

```
type event =
  | Request of term * term * term
  | Response of term * term * term * term
  | Initiator of term * term * term * term
  | Responder of term * term * term * term
  | Bad of term

let KeyABComp a b =
  Logged (Bad a) || Logged (Bad b)

let KeyABPayload a b m =
  (exists (fun req →
     m = Pair(Literal [|TagRequest|],req) &&
     Logged (Request(a,b,req)))) ||
  (exists2 (fun req resp →
     m = Pair(Literal [|TagResponse|],Pair(req,resp)) &&
     Logged (Response(a,b,req,resp))))

let KeyAB a b =
  hmacKey
    (fun k → KeyABComp a b)
    (fun m → KeyABPayload a b m)

let PrinKeyComp p k =
  Logged (Bad p)

let PrinKeyPayload p m =
  (exists3 (fun b np kpb →
     p <> b &&
     (m = Pair(p,Pair(b,Pair(kpb,np)))) &&
     KeyAB p b kpb &&
     Logged (Initiator(p,np,kpb,b)))) ||
  (exists3 (fun a np kap →
     p <> a &&
     (m = Pair(a,Pair(p,Pair(kap,np)))) &&
     KeyAB a p kap &&
     Logged (Responder(p,np,kap,a))))

let PrinKey p =
  sencKey
    (fun k → PrinKeyComp p k)
    (fun m → PrinKeyPayload p m)

[< LogInvariant >]
let LogInvariant =
  forall2 (fun t (u: usage) →
    u t ⇒ exists (fun bs → t = Literal bs))

(* Omitting aRPC theorems *)

[< Valid >]
let InitiatorCorrespondence =
  forall5 (fun a b ka na kab →
    GoodLog ⇒ (a <> b) ⇒ PrinKey a ka ⇒
    Level Low (SEnc(ka,Pair(a,Pair(b,Pair(kab,na))))) ⇒
    KeyAB a b kab || Logged (Bad a))

[< Valid >]
let ResponderCorrespondence =
  forall5 (fun a b kb nb kab →
    GoodLog ⇒ (a <> b) ⇒ PrinKey b kb ⇒
    Level Low (SEnc(kb,Pair(a,Pair(b,Pair(kab,nb)))))
      ⇒
    KeyAB a b kab || Logged (Bad b))
```

```
[< Valid >]
let KeyABSecrecy =
  forall3 (fun a b kab →
    GoodLog ⇒ KeyAB a b kab ⇒
    Level Low kab ⇒
    Logged (Bad a) || Logged (Bad b))
```

2.4 Discussion and Related Work

The Inductive Approach Paulson's inductive approach [Paulson, 1998] relies on the protocol being encoded as inductive rules describing how protocol traces can be constructed by building and publishing new terms from previously observed messages. Security properties are expressed in the same way as ours, as correspondences on a log of event predicates (in his case, the entire trace). However, since we wish to separate security concerns from low-level protocol and implementation details, we prefer a proof technique that provides more abstraction from the protocol and security goal.

For example, the TAPS tool (Cohen [2003]) relies on first-order invariants very similar to ours, in order to prove security properties of protocols. Cohen's techniques rely on making as much of the proof as possible protocol-independent, and reducing his central *secrecy invariant*, which states that only messages that are meant to be published are ever published, to simple mathematical properties of derived predicates.

Bhargavan et al. [2010] push these ideas further and even abstract from the protocol itself, only modelling the use of cryptography, therefore modularizing the approach, and making it applicable to the verification of implementations. Cohen's secrecy invariants correspond to Bhargavan et al. [2010]' precondition on network writes, that the sent message is Low. By abstracting the protocol further, we expect to enable some modular reasoning about the protocol itself (and not only its implementation), allowing, for example, to reason soundly about protocol composition. However, we have not yet experimented with this and currently repeat the proofs for Authenticated RPC when using it with keys established by Otway-Rees.

Process Calculi and ProVerif The use of process calculi for the verification of security protocols goes back to Lowe's famous use of the CSP process calculus to find attacks, and prove security [Lowe, 1996]. Since then, similar techniques have been applied to various process calculi.

For example, ProVerif [Blanchet, 2001] takes protocol descriptions written in a variant of the applied π-calculus, translating them into Horn clauses describing the principals' knowledge and proving correspondence properties on this abstract model of the protocol. These later tools refine the free algebra model of cryptography we discussed with equational theories, allowing much more realistic protocols to be analysed, and in particular permitting the use of non-authenticated encryption and arbitrary message formatting. In addition, ProVerif also supports a stronger, observational equivalence-based notion of secrecy, which we discuss further in Section 3.5.

Strand Spaces Strand spaces were introduced by Fábrega [1999] to reason about security protocols in more general terms than the usual goals of trying to prove security, or finding flaws. In particular, they can produce a so-called *complete characterisation* of protocols, that is described all the execution traces that can occur from a given starting configuration (that is, a set of roles). CPSA [Doghmi, Guttman, and Thayer, 2007] and Scyther [Cremers, 2008] are automated tools that work in this model. Although Scyther can also produce complete characterisations, it is mostly focused on proving security properties, but allows more flexible adversary models than previous protocol verification tools (for example, adversaries with access to some of the protocol's internal state, useful for refining and classifying attacks on key exchange protocols).

Computationally Complete Symbolic Attackers In more recent work, Bana and Comon-Lundh [2012] introduced a new kind of symbolic models, based simply on declaring function symbols and constraining them with axioms. The soundness of the approach is ensured by properties of each individual axiom, allowing the model to be built primitive by primitive. However, this approach has not so far been automated, and has only been applied by hand to small protocols.

Chapter 3

Verifying Symbolic Security of C Code

The goal of this chapter is to show how our chosen C verification tool can be used to prove that a given C programs uses cryptography in a way that preserves a set of cryptographic invariants, including secrecy and authentication properties, such as those described in Chapter 2. To do so,

- we encode the T3 protocol description as VCC ghost specification data and code (Section 3.1.1);
- we provide generic annotations and verified ghost instrumentation on the cryptographic libraries, that enforce the assumptions of our symbolic model, the payload conditions when using cryptography, and the fact that data sent over the network is Low (Section 3.1.2);
- we model network adversaries as syntactically restricted C programs (Section 3.2);
- and we explain how the resulting verification conditions can be discharged, and guarantee the desired security properties, by stepping through the authenticated RPC example (Section 3.3).

Section 3.4 summarizes our experiments on some of the example protocols formalised in Section 2.3.3, and Section 3.5 discusses our contributions and their limitations, in view of related work.

In this chapter, we work in the symbolic model discussed in Chapter 2, where cryptographic terms are members of a free algebra with no equational theory introducing additional equalities. We can make use of VCC's inductive datatypes for specifiying this, as shown right.

Term Algebra in VCC

```
_(datatype term {
    case Literal(bytes)
    case Pair(term,term)
    case Hmac(term,term)
    case SEnc(term,term)
    case Sign(term,term)
    case Enc(term,term) })
```

3.1 A VCC Framework for Symbolic Cryptography

3.1.1 Symbolic Cryptographic Invariants

If Bhargavan et al. [2010] treat the cryptographic log as an implicit set of all assumed formulas, we choose to keep it as an explicit set of events in ghost state, and turn F7 **assume** statements into explicit ghost state updates. The definitions in this Chapter mimic the Coq descriptions from Chapter 3.

Usages and Events Declarations in VCC

```
_(type nonce_usage)
_(type hmac_usage)
_(type senc_usage)
_(type sign_usage)
_(type enc_usage)

_(datatype usage {
    case AttackerGuess()
    case Nonce(nonce_usage)
    case HmacKey(hmac_usage)
    case SEncKey(senc_usage)
    case SignKey(sign_usage)
    case VerfKey(sign_usage)
    case EncKey(enc_usage)
    case DecKey(enc_usage) })

_(type pEvent)

_(datatype event {
    case New(term,usage)
    case AsymPair(term,term)
    case ProtEvent(pEvent) })
```

Usages and events are declared in the same way as they are defined in the Coq model, using inductive declarations for the general datatypes, and declaring the types of protocol-specific events and primitive usages forward of their use. These abstract types can be instantiated where necessary using either datatypes or functional record types. In the same way, function symbols with contracts can be declared before being defined, and left abstract for the purpose of general proofs, which is useful to express desired properties of the declared symbols (for example, stability lemmas), in the same way that Coq module signatures let us declare theorems.

Event logs are modelled using a boolean map-based encoding of sets. Testing set membership is done by applying the map, and union and intersection can simply be expressed as lambda expressions combining the two maps by disjunction and conjunction (respectively). We define the same well-formedness condition on logs as in the Coq definitions. Some of the shorthand notations from the Coq formalisation of the framework are omitted from its VCC encoding, since they are actually longer than the direct notation.

Event Logs in VCC

```
_(typedef \bool log[event])

_(def \bool leq_log(log L1, log L2)
  { return ∀event e; L1[e] ⇒L2[e]; })

_(def \bool LoggedP(pEvent e, log L)
  { return L[ProtEvent(e)]; })

_(def \bool WF_log(log L)
  { return
      (∀ term t; usage u1, u2;
        L[New(t,u1)] ⇒L[New(t,u2)] ⇒
        u1 == u2) &&
      (∀ term pk,sk;
        L[AsymPair(pk,sk)] ⇒
        ((∃ sign_usage su;
          L[New(sk,VerfKey(su))] &&
          L[New(pk,SignKey(su))]) ||
         (∃ enc_usage eu;
          L[New(sk,EncKey(eu))] &&
          L[New(pk,DecKey(eu))]))); })
```

The VCC framework, just like its Coq formalisation, is parameterised by protocol-specific predicate definitions, as well as the types mentioned above, that are expected to follow some simple stability theorems. In Coq, both the predicates and the theorems were expressed as typed symbols in a module signature. Similarly, in VCC, we write function prototypes with contracts for the predicates, and use special forms of function prototypes to express the theorems that are expected to hold on the predicates, as illustrated on the right and briefly discussed in Chapter 1. In this dissertation, we choose to express theorems as constantly true boolean functions whose postcondition is the desired property.

VCC Signature for Protocol-Specific Invariants

```
_(abstract \bool LogInvariant(log L))

_(abstract \bool nonceComp(term n, log L))

_(abstract \bool nonceComp_Stable()
  _(ensures ∀term t; log L1, L2;
      leq_log(L1,L2) ⇒nonceComp(t,L1) ⇒
      nonceComp(t,L2))
  _(returns \true))
```

In the current version of the framework, the Level predicate is not encoded directly as an inductive predicate, but is instead given to VCC as a set of axioms, one per inductive case and some for useful inversion theorems. Although it may be possible to give its definition as an inductive predicate, its formal definition does not translate well into the VCC specification language, and the current framework has shown good enough performance on various forms of programs and protocols. Some of the rules and theorems (both keywords are in fact shorthands for the VCC **axiom**, that ignore the name argument) are shown right, illustrating that they are direct syntactic translations of the Coq rules into VCC.

Level in VCC

```
_(def \bool GoodLog(log L)
  { return
      WF_log(L) &&
      LogInvariant(L); })

_(datatype level {
    case Low()
    case High() })

_(abstract \bool Level(level,term,log))

_(rule(Level_AttackerGuess)
  ∀level l; bytes bs; log L;
    L[New(Literal(bs),AttackerGuess())] ⇒
    Level(l,Literal(bs),L))

_(rule(Level_Hmac)
  ∀level l; term k,m; log L;
    canHmac(k,m,L) ⇒
    Level(l,m,L) ⇒
    Level(l,Hmac(k,m),L))

_(theorem(Pair_Level)
   ∀level l; term t1,t2; log L;
     Level(l,Pair(t1,t2),L) ⇒
     Level(l,t1,L) && Level(l,t2,L))
```

3.1.2 The Representation Table

Our symbolic model of cryptography assumes that two distinct symbolic terms are represented as two distinct byte strings, and that fresh literals cannot be guessed by an adversary (enforced by the log invariant, and the fact that guesses made by the adversary are given an AdversaryGuess usage). However, the C programs we verify use concrete cryptographic libraries that manipulate finite byte

strings. Even if C and VCC provided enough type abstraction to hide this fact, the underlying cardinality mismatch between the infinite set of cryptographic terms and the finite set of bytestring values the concrete program can manipulate is an obstacle to any soundness argument, as it introduces a logical inconsistency in the model of the program.

To solve this issue, we instrument the program with specification code that maintains a *representation table*, which tracks the correspondence between concrete byte strings and symbolic terms. We intercept all calls to cryptographic functions with ghost code to update the representation table. We say a *collision* occurs when the table associates a single byte string with two distinct symbolic terms.

For example, suppose x and y are two distinct bytestrings that have the same HMAC, h, under a key k. After the first call to hmac() the table is, for example, as shown. When computing the second HMAC, our instrumented hmac() function attempts to insert the freshly computed h and the corresponding term **Hmac** k y in the table, but detects that h is already associated with a distinct term **Hmac** k x.

Bytestring	Term
k	**Literal** k
x	**Literal** x
y	**Literal** y
h	**Hmac** k x

We make the absence of such collisions an explicit hypothesis in our specification by assuming, via an **assume** statement in the ghost code updating the table, that a collision has not occurred. This removes from consideration any computation following a collision, as was made precise in Section 1.3.2. We treat the event of the adversary guessing a non-public value in a similar way; we assume it does not happen, using an **assume** statement. In this way we prove symbolic security properties of the C code. A separate argument may be made that such collisions only happen with low probability, and is further discussed in Section 3.5.

Representation tables in VCC

```
_(record table {
    \bool DefinedB[bytes];
    term B2T[bytes];

    \bool DefinedT[term];
    bytes T2B[term]; })

_(def \bool valid_table(log L,table T)
  { return
      (∀ bytes b;
        T.DefinedT[Literal(b)] ⇒T.T2B[Literal(b)] == b) &&
      (∀ term t;
        T.DefinedT[t] ⇒Level(High(),t,L)) &&
      (∀ bytes b;
        T.DefinedB[b] ⇒T.T2B[T.B2T[b]] == b) &&
      (∀ term t;
        T.DefinedT[t] ⇒T.B2T[T.T2B[t]] == t) &&
      (∀ bytes b;
        T.DefinedB[b] ⇒T.DefinedT[T.B2T[b]]) &&
      (∀ term t;
        T.DefinedT[t] ⇒T.DefinedB[T.T2B[t]]); })

_(def \bool leq_table(table T1,table T2)
  { return
      (∀ bytes b;
        T1.DefinedB[b] ⇒T2.DefinedB[b]) &&
      (∀ bytes b;
        T1.DefinedB[b] ⇒T1.B2T[b] == T2.B2T[b]) &&
      (∀ term t;
        T1.DefinedT[t] ⇒T2.DefinedT[t]) &&
      (∀ term t;
        T1.DefinedT[t] ⇒T1.T2B[t] == T2.T2B[t]); })
```

Like the log, we define the table as a ghost ob-

ject that is part of the program state and can be directly manipulated in ghost code, this time as a functional record, containing two maps that store the partial bijection between byte strings and terms, and two boolean maps encoding the domain on which the bijection is defined. The predicate valid_table expresses that the B2T and T2B maps of a table do indeed represent a bijection, and also states that all terms that appear in that table (that is, all terms manipulated by the implementation so far) are in fact High in the current log. We also define an order on tables, which is the pointwise order on pairs of partial functions.

Before interfacing these ghost objects with concrete C code, we provide a global wrapper allowing us to manipulate the log and table concurrently.

The two fields are made volatile, and the structure is initially closed to ensure that

1. the one-state invariant, that states that the current log is always good, holds on the initial state; and
2. all updates have to follow the two-state invariants, which states that the log and table grow monotonically over all transitions.

Global cryptographic state

```
_(ghost _(claimable) struct cryptoState_s
{
    volatile log L;
    volatile table T;

    _(invariant WF_log(L) && LogInvariant(L)) // Hint: unroll
          those defs
    _(invariant GoodLog(L))
    _(invariant leq_log(\old(L),L))

    _(invariant valid_table(L,T))
    _(invariant leq_table(\old(T),T))
} CS;)
```

3.1.3 The Hybrid Wrappers

We want to ensure that all cryptographic operations are used in ways that preserve the cryptographic state's invariants. We provide hybrid wrappers around the concrete library functions; wrappers are not only verified to maintain the table's invariants but also serve to give symbolic contracts to a cryptographic interface working with concrete bytes.

To simplify the sample code in this dissertation, and focus on the security aspects of the verification effort, we write the hybrid wrappers to manipulate a structure type bytes_c, displayed right, containing all information pertaining to a byte array: its address and its length, but also the value of the buffer's current contents, as a byte string.

A type for byte strings

```
typedef struct {
  unsigned char *ptr;
  unsigned long len;

  _(ghost bytes value)
  _(invariant \mine((unsigned char[len]) ptr))
  _(invariant value == from_array(ptr,len))
} bytes_c;
```

Idiomatic C code would manipulate the beginning address and length separately. The sample code studied in Aizatulin et al. [2011a] does not use the bytes_c type and manipulates raw byte arrays, and also illustrates how the hybrid instrumentation can be inlined in cases where the application requires it (for example, if some formatting is inlined for performance).

As an example, here is the full contract of our hybrid wrapper for the hmac_sha1() cryptographic function.

Hybrid interface for hmacsha1()

```
int hmacsha1(bytes_c *k, bytes_c *b, bytes_c *res
             _(ghost \claim c))
  // Claim property
_(always c, (&CS)->\closed)
  // Properties of input byte strings
_(maintains \wrapped(k))
_(maintains \wrapped(b))
  // Properties of out parameter
_(writes \extent(res), c)
_(ensures !\result ⇒\wrapped(res))
  // Cryptographic contract
_(requires CS.T.DefinedB[k->value])
_(requires CS.T.DefinedB[b->value])
_(ensures !\result ⇒CS.T.DefinedB[res->value])
    // Cryptographic properties on input terms
_(requires
  canHmac(CS.T.B2T[k->value],CS.T.B2T[b->value],CS.L)
  || (Level(Low(),CS.T.B2T[k->value],CS.L) &&
      Level(Low(),CS.T.B2T[b->value],CS.L)))
    // Cryptographic properties on output term
_(ensures !\result ⇒
  CS.T.B2T[res->value] ==
    Hmac(CS.T.B2T[k->value],CS.T.B2T[b->value]));
```

To ensure that the log and table are kept stable during the call, as well as to allow concurrent updates to the cryptographic states, we pass in, as a ghost parameter, a claim c guaranteeing that the global container CS remains closed whenever the claim is closed. All other desired properties (monotonic growth and validity, in particular) are consequences of this simple fact and the invariants on the cryptographic state, and can be derived by VCC. The next lines of the contract deal with memory-safety concerns, in this case expressing the fact that the arguments are wrapped at call-site and return-site, and that the (typed) memory location pointed to by the third argument is written to by the function, and is wrapped on successful return from the function. The claim c is added to the list of objects that may be written by the function in case it is necessary to strengthen the claimed property to help the proof go through.

The first three lines under "Cryptographic contract" deal only with the table, stating that the input byte strings should appear in the table, and that, upon successful return from the function, the output byte string appears in the table. Finally, we require as a precondition that either canHmac holds on the input parameters in the current cryptographic state (catering for an honest participant's calling conditions), or both the key and payload are Low (catering to calls by the adversary).

On successful return, we guarantee in the postcondition that the output byte string is associated, in the table, with the term obtained by applying the Hmac constructor to the terms associated with the input byte strings.

An honest client, when calling this function, will establish that canHmac() holds on the terms associated with the input byte arrays. We recall that the definition of canHmac is in general a disjunction of clauses of the form "k has HMAC key usage u, and m fulfills the payload condition for u in the log". In the particular case of our authenticated RPC protocol, an honest participant will know that u is indeed U_KeyAB(a,b) for some a and b, and will attempt to prove that the payload is either a well-formatted request on which the Request event has been logged for a and b, or a well-formatted response on which the Response event has been logged for a and b.

A hybrid wrapper for hmacsha1()

```
int hmacsha1(bytes_c *k, bytes_c *b, bytes_c *res
             _(ghost \claim c))
{ _(ghost term tb,tk,th)
  _(ghost \bool collision = \false)
  _(assert Level(High(),Hmac(CS.T.B2T[k->value],CS.T.B2T[b->
       value]),CS.L))
  _(ghost \claim c0 = \make_claim({ c }, (&CS)->\closed &&
       GrowsCS))

  res->len = 20;
  res->ptr = (unsigned char*) malloc(res->len);
  if (res->ptr == NULL)
    return 1;
  sha1_hmac(k->ptr, k->len, b->ptr, b->len, res->ptr);

  _(ghost res->value = from_array(res->ptr,res->len))
  _(ghost \wrap((unsigned char[res->len]) res->ptr))
  _(ghost \wrap(res))
  _(ghost _(atomic c0, &CS) {
      tb = CS.T.B2T[b->value];
      tk = CS.T.B2T[k->value];
      th = Hmac(tk,tb); // Compute the symbolic term

      if ((CS.T.DefinedB[res->value] &&
           CS.T.B2T[res->value] != th) ||
          (CS.T.DefinedT[th] &&
           CS.T.T2B[th] != res->value))
        collision = \true;
      else
      {
        CS.T.DefinedT[th] = \true;
        CS.T.T2B[th] = res->value;
        CS.T.DefinedB[res->value] = \true;
        CS.T.B2T[res->value] = th;
      }
    })
  _(assume !collision) // Our symbolic crypto assumption

  return 0; }
```

A typical hybrid wrapper implementation first performs the concrete operation on byte strings (for instance, by calling a cryptographic library) before performing updates on the ghost state to ensure the cryptographic postconditions, whilst maintaining the log and table invariants. To do so, it first computes the expected cryptographic term by looking up, in the table, the terms associated with the input byte strings and applying the suitable constructor. Once both the concrete byte string and the corresponding terms are computed, the implementation can check for collisions, and in case there are none, update the table (and the log) as expected. In case a collision happens, an **assume** statement expresses that our symbolic cryptography assumptions have been violated.

Such wrappers can only be verified once the protocol-specific definitions are known and given to the C verifier. For example, without definition for the LogInvariant predicate, VCC cannot be expected to prove that the atomic update preserves log validity (since the LogInvariant could forbid that particular update).

Also of particular note is the fact that our wrappers for signature and HMAC verification only verify when the log invariant implies that only literal terms are used as keys and nonces (in fact, whenever

the log invariant by itself ensures that nonces and keys only become Low if their release condition holds). As discussed in Chapter 2, protocols that use non-literals as nonces or keys require more complex protocol-specific proofs, which will be reflected in the wrapper's contract.

Transitive closure of crypto state stability

```
#define GrowsCS\
    (leq_log(\when_claimed(CS.L),CS.L) &&\
     leq_table(\when_claimed(CS.T),CS.T))
```

We use the GrowsCS macro, defined below, which is claimed on the cryptographic state (log and table), to guide VCC through the proof by transitively expanding the stability invariant on cryptographic states: the cryptographic state at the program point where the claim is created is smaller than the cryptographic state at any program point afterwards.

Since the cryptographic state is shared, and its fields marked volatile, all reads and writes from and to the log and table need to occur in atomic blocks guarded by a claim c ensuring, at least, that the global CS object is closed. To strengthen the claim c, we simply create a claim c0 on c, whose property immediately follows from the log and table invariants, and guarantees their monotonic growth despite interference from other threads, by making use of the transitivity argument explained above.

We also provide a function toString() (whose contract is shown right), converting an ordinary string pointer to a bytes_c, the input type for functions like hmacsha1(). It logs a New event with usage AdversaryGuess and assumes the guessed literal does not collide with any other term already in the table.

Contract for the toString() wrapper

```
int toString(unsigned char *in, unsigned long inl, bytes_c *res
        _(ghost \claim c))
_(maintains \thread_local_array(in,inl) && inl != 0)
_(requires \disjoint(\array_range(in,inl),\extent(res)))
_(writes \extent(res))
_(ensures \result ⇒\mutable(res))
_(ensures !\result ⇒\wrapped_with_deep_domain(res))
_(always c, (&CS)->\closed)
_(ensures !\result ⇒CS.T.DefinedB[res->value])
_(ensures !\result ⇒Level(Low(),CS.T.B2T[res->value],CS.L))
_(ensures !\result ⇒res->value == \old(from_array(in,inl)))
_(ensures !\result ⇒CS.T.B2T[res->value] == Literal(res->
        value));
```

Finally, we provide a function bytescmp, that compares two bytes_c objects, and pair and destruct functions that marshal and unmarshal bytes_c objects into and from injective pairs.

As mentioned previously, and illustrated in Aizatulin et al. [2011a], the ghost instrumentation can in fact be pulled out of the wrappers entirely, and added in by hand after each call. In this case, soundness of the approach is maintained by placing proper preconditions on the instrumentation code, at a slight cost in performance due to the additional proof obligations.

3.2 Attack Programs

Adversaries in the symbolic model can intercept messages on unprotected communication links (such as the Internet) and send messages constructed from parts of intercepted messages, as specified by the term algebra. We model the set of all possible attacks, each attack being represented by an *attack program* (or just "attack") which is C code of a particular form.

We define attack programs relative to protocol-specific *adversary interfaces*. The grammar for adversary interfaces is displayed on the right. Such an interface provides some "opaque" type declarations together with some function signatures; these include message send/receive, standard cryptographic operations, and protocol-specific actions for, for example, creating sessions and initiating roles on principals. For an annotated interface $p.h$, we let $erase(p.h)$ be the adversary interface obtained by deleting annotations and the bodies of type declarations.

Adversary interfaces

$T ::=$	type
bool | **unsigned char**$*$ | $X*$	
$\mu ::=$	entry in an interface
typedef X;	type declaration
$T\ f(T_1\ x_1, \ldots, T_n\ x_n)$	function prototype
void $f(T_1\ x_1, \ldots, T_n\ x_n)$	procedure prototype
$\mathcal{I} ::= \mu_1 \ldots \mu_n$	interface ($n \geq 0$)

The protocol-specific shim provides a network adversary interface including generic cryptography and network operations as well as protocol specific functions. For example, we display below the shim for the authenticated RPC protocol, that gives the adversary access to all the constructors and destructors (on byte strings), and lets him setup new sessions (which generates the shared keys and creates the network sockets), run the client and server roles associated with a session, compromise one of the principals associated with a session (which returns the shared key), and hijack the communication channels.

An adversary interface: *erase*(RPCshim.h)

```
typedef bytespub;

bytespub* att_toBytespub(unsigned char* ptr, unsigned long
    len);
bytespub* att_pair(bytespub* b1, bytespub* b2);
bytespub* att_fst(bytespub* b);
bytespub* att_snd(bytespub* b);

bytespub* att_hmacsha1(bytespub* k, bytespub* b);
bool att_hmacsha1Verify(bytespub* k, bytespub* b, bytespub* m);

void att_channel_write(channel* chan, bytespub* b);
bytespub* att_channel_read(channel* chan);

typedef session;

session* att_setup(bytespub* cl, bytespub* se);

void att_run_client(session* s, bytespub* request);
void att_run_server(session* s);

bytespub* att_compromise_client(session*s);
bytespub* att_compromise_server(session*s);

channel* att_getChannel_client(session* s);
channel* att_getChannel_server(session* s);
```

Type bytespub is critical: its invariant constrains its values to be concrete byte arrays that correspond

to Low terms. Verifying the implementation of this adversary interface therefore provides a proof that the set of Low terms is closed under adversary actions. The function contracts (not shown here) are similar to the hybrid wrappers in Section 3.1.3 but oriented to Low data. They are also more complicated, due to memory safety annotations dealing with thread fork and messaging, though that is mostly protocol-independent, and not relevant to the security of the system. An example shim contract appears in Section 3.3.

Attack programs against a given adversary interface are straight-line C programs that are syntactically limited to behave like network adversaries, in particular by syntactically preventing them from breaking the necessary memory abstraction. More importantly, those syntactic restrictions are meant to guarantee that attack programs are verifiable by VCC, although this depends on the chosen adversary interface and cannot be proved in general. Rather, we sketch a proof of the verifiability of attack programs for the authenticated RPC shim in Section 3.3.

The syntactic restrictions we place on attack programs are detailed below.

Attack Programs on $\mathcal{I}$

T ::=	Type
bool | **unsigned char**$*$	
t$*$	where t is declared in $\mathcal{I}$
D ::= T v;	Local variable declaration
C ::=	Command
v = f(v1,...,vn);	where v, v1, ..., vn are variables names and f is declared in $\mathcal{I}$
f(v1,...,vn);	where v1, ..., vn are variables names and f is declared in $\mathcal{I}$
v = s;	where v is a variable name and s is a string literal
	where:
	1. A variable is assigned to at most once, and every variable mentioned in $C_1 \ldots C_m$ is declared in $D_1 \ldots D_n$.
	2. For each function or procedure call, each argument variable is assigned to earlier in the sequence of commands.
P ::= **void** main(){D_1 ... D_n C_1 ... C_m}	3. In each call to a function or procedure f, each argument variable in the call has declared type identical to that of the corresponding parameter of f.
	4. In a function call assignment v = f(...);, the declared type of v is the result type of f.
	5. In a string assignment v = s; the declared type of v is **unsigned char**$*$.

Displayed on the right is an example attack program, describing the "attack" that runs an instance of the RPC protocol to completion, passing the messages around correctly. After declaring all the necessary variables, and initialising some literals for prinipal names and the payload, the adversary starts an authenticated RPC session between the two principals, retrieves their channels and starts

both roles, passing the chosen request to the client. The adversary then reads the request message from the client's channel and writes it onto the server's channel, and performs the opposite operation for the response message, routing it back from the server channel to the client channel.

It is fairly easy to see that the syntactic restrictions do not prevent any of the standard symbolic operations, including control over the scheduling of the roles, full control over the network (including the ability to delay or otherwise stop messages), as well as all the standard symbolic cryptographic constructors and destructors.

Attack program for `RPCshim.h`

```
void main()
{ unsigned char *a,*b,*r;
  bytespub *alice,*bob,*arg,*req,*resp;
  channel *clientC,*serverC;
  session *s;

  a = "Alice"; alice = att_toBytespub(a,5);
  b = "Bob"; bob = att_toBytespub(b,3);
  r = "Request"; arg = att_toBytespub(r,7);
  s = att_setup(alice,bob);
  clientC = att_getChannel_client(s);
  serverC = att_getChannel_server(s);
  att_run_server(s);
  att_run_client(s,arg);
  req = att_channel_read(clientC);
  att_channel_write(serverC,req);
  resp = att_channel_read(serverC);
  att_channel_write(clientC,resp);}
```

3.3 An Example Security Theorem: authenticated RPC

Given the framework from Section 3.1, and the adversary model described in Section 3.2, proving security theorems about a C implementation breaks down into several different tasks:

1. Write the protocol-specific definitions (log invariant, and payload and release conditions) in VCC, and verify the hybrid wrappers with respect to them;

2. Annotate the protocol code with event logging and event assertions expressing the authentication properties;

3. Write the adversary shim and prove that all attack programs against that shim are verifiable in VCC.

Although it is currently done manually, Task 1 can be automated, generating the VCC definitions for a T3 protocol description at the same time as the Coq theory gets produced, and verifying the hybrid wrappers in a single run of VCC (as mentioned above, this verification may be made more complex, or require a rewrite of some of the signature-checking hybrid wrappers, when using non-standard log invariants).

Task 3 is very costly in verification time due to the highly concurrent nature of, for example, the code that forks new threads to run individual roles. However, it can be verified once and for all once the adversary's capabilities and the protocol code's interface have been fixed, and could in fact be a good first sanity check of the protocol interface with respect to the protocol's expected invariants. This verification task does not involve very much security-specific reasoning and could therefore be performed by a non-security specialist, with a good understanding of VCC's concurrency model.

Finally, Task 2 is the central part of the protocol-specific proof. We discuss it in the context of the RPC example, before briefly tackling the proof that all attack programs against the authenticated RPC shim are verifiable.

3.3.1 Verifying Protocol Code

The code displayed on the right shows a slightly simplified version of the annotated code for the client role, where the Request event is logged by the atomic assignment and the final correspondence is asserted as a disjunction of events taking into account the potential compromise of one of the principals involved.

When verifying this code, VCC proves that each of the function calls happens in a state where the function's precondition holds. In particular,

Annotated RPC client code

```
void client(bytes_c *alice, bytes_c *bob, bytes_c *kab, bytes_c *req,
      channel* chan _(ghost \claim c))
_(maintains \wrapped(alice) && \wrapped(bob) &&
            \wrapped(kab) && \wrapped(req))
_(always c, (&CS)->\closed)
_(writes c)
_(requires CS.T.DefinedB[alice->value] &&
           CS.T.DefinedB[bob->value] &&
           CS.T.DefinedB[kab->value] &&
           CS.T.DefinedB[req->value])
_(requires Level(Low(),CS.T.B2T[alice->value],CS.L) &&
           Level(Low(),CS.T.B2T[bob->value],CS.L) &&
           Level(Low(),CS.T.B2T[req->value],CS.L) &&
           Level(High(),table.B2T[kab->value]),CS.L)
_(requires
  RPCKeyAB(CS.T.B2T[alice->value],
           CS.T.B2T[bob->value],
           CS.T.B2T[kab->value],
           CS.L));
{ _(ghost \claim c0 = createRunningClaim(c))
  bytes_c *toMAC1, *mac1, *msg1;
  bytes_c *msg2, *resp, *toMAC2, *mac2;
  // Event
  _(ghost { _(atomic c, &CS)
    CS.L[ProtEvent(Request(CS.T.B2T[a->value],
                           CS.T.B2T[b->value],
                           CS.T.B2T[req->value]))] = \true; })
  // Build and send request message
  _(ghost refreshCryptoState(c0))
  if ((toMAC1 = malloc(sizeof(*toMAC1))) == NULL) return;
  if (request(req, toMAC1 _(ghost c))) return;

  _(ghost refreshCryptoState(c0))
  if ((mac1 = malloc(sizeof(*mac1))) == NULL) return;
  if (hmacsha1(kab, toMAC1, mac1 _(ghost c))) return;

  _(ghost refreshCryptoState(c0))
  if ((msg1 = malloc(sizeof(*msg1))) == NULL) return;
  if (pair(req, mac1, msg1 _(ghost c))) return;

  _(ghost refreshCryptoState(c0))
  if (channel_write(chan, msg1 _(ghost c))) return;

  // Receive and check response message
  _(ghost refreshCryptoState(c0))
  if ((msg2 = malloc(sizeof(*msg2))) == NULL) return;
  if (channel_read(chan, msg2 _(ghost c))) return;

  _(ghost refreshCryptoState(c0))
  if ((resp = malloc(sizeof(*resp))) == NULL) return;
  if ((mac2 = malloc(sizeof(*mac2))) == NULL) return;
  if (destruct(msg2, resp, mac2 _(ghost c))) return;

  _(ghost refreshCryptoState(c0))
  if ((toMAC2 = malloc(sizeof(*toMAC2))) == NULL) return;
  if (response(req, resp, toMAC2 _(ghost c))) return;

  _(ghost refreshCryptoState(c0))
  if (!hmacsha1Verify(kab, toMAC2, mac2 _(ghost c))) return;

  // Correspondence assertion
  _(assert \active_claim(c0))
  _(assert LoggedP(Response(CS.T.B2T[alice->value],
                            CS.T.B2T[bob->value],
                            CS.T.B2T[req->value],
                            CS.T.B2T[resp->value]),CS.L)
        || LoggedP(Bad(CS.T.B2T[a->value]),CS.L)
        || LoggedP(Bad(CS.T.B2T[b->value]),CS.L)); }
```

the call to the channel_write() function yields a proof obligation that the term corresponding to the second argument is Low in the current cryptographic state. The **return** statements are for various kinds of failure, effectively aborting the client in such cases.

As in the hybrid wrappers, we use a local claim c0 (with the same claimed property) to encode the fact that the log and table grow with time. The reference cryptographic state for the transitivity argument is updated, by simply re-claiming the same property in a new state, between all calls to hybrid wrappers.

To prove that the correspondence assertion holds, VCC uses the postconditions of hmacsha1Verify(), stating that a zero return value implies that either canHmac() holds on the key and payload, or the key used for verifying the MAC is Low. In addition, kab is known to correspond to a term that has usage U_KeyAB(a,b) (from the precondition), toMAC2 is known to have a correct reponse format (from the postcondition to the response() function), and mac2 itself is Low, since it is part of a message read from the network. These facts in combination allow VCC to prove the correspondence assertion by unfolding the definitions of canHmac() and inverting the fact that the key is Low to cover the second and third disjunct.

3.3.2 Verifiability of the Adversary Shim

An attack program for the authenticated RPC protocol is a program that relies only on the interface we call `RPCshim.h`, and for which a contract-free version was previously shown. To form an executable, it needs to be combined with `System`, which we define to be the catenation `crypto.c` · `RPChybrids.c` · `RPCprot.c`. Here `crypto.c` is the library of cryptographic algorithms (and we let it subsume OS libraries, e.g., for memory allocation and sockets), which is used in `RPChybrids.c` and `RPCprot.c`.

Before providing the formal results, we informally describe a key property on which soundness of our approach rests. Consider any attack program `M.c` and any run of the program `System` · `RPCshim.c` · `M.c`. It is an invariant that at every step of the run, the representation table holds every term that has arisen by cryptographic computation or by invocation of the toBytesPub() function which an attack must use to convert guessed bytestrings to type bytespub, as needed to invoke the other functions defined in `RPCshim.c`. This is not an invariant that we state in the program annotations; its only role

is to justify our use of assumptions. The only assumptions used are in `RPCshim.c` and `RPChybrids.c`, where collisions are detected. In light of the key invariant, this means that in any run that reaches an assumption failure, the sequence of terms computed includes a hash collision or an adversary guess of a term that is not public according to our symbolic model of cryptography. In short, assumptions are used only to express the Dolev-Yao assumption.

The contracts in `RPCshim.h` all follow a similar pattern; we give one (shown right) for reference in the following proof sketch.

Example contract from `RPCshim.h`

```
bytespub* att_hmacsha1(bytespub* k, bytespub* b _(ghost \claim c
    ))
_(maintains \wrapped(k))
_(maintains \wrapped(b))
_(writes k,b,c)
_(always c, (&CS)->\closed)
_(ensures \wrapped(\result));
```

Attack programs were carefully defined in order to both argue that all symbolic attacks can be written as attack programs, and show that their behaviours are among those of interfering threads encompassed by the verification conditions VCC imposes on protocol code. By soundness Assumption 1, this will be a consequence of the following verifiability result.

Lemma 1. *If M.c is an attack program for erase*(`RPCshim.h`)*, then* `RPCshim.h` $\vdash$ *M.c* $\rightsquigarrow$ `main.h`.

We cannot prove this result under the "pragmatic interpretation" that `RPCshim.h` $\vdash$ *M.c* $\rightsquigarrow$ `main.h` means *M.c* is verifiable by the VCC tool. There are infinitely many attack programs, most of which are too large to even fit in storage much less be processed by the tool. Instead, we consider the "mathematical interpretation"; that is, we show that the verification conditions are valid. Some parts of the proof can still be performed using the verification tool.

Proof. (Sketch) According to Definition 2 we have to show admissibility of the type invariants in `RPCshim.h`; this we have checked using VCC. It remains to prove verifiability of an arbitrary attack program against `main.h`.

Since the contract in `main.h` does not impose a postcondition and its write specification is vacuous, we just need to show that invariants are established and maintained. Let *M.c* be **void** main(){D C}. In accord with Definition 2 we will show verifiability of code C′, that augments the statements of C with two sorts of instrumentation. The first is simply to prefix C with ghost code that initializes the representation table and log.

This code is defined as a ghost function init(), whose contract and body are shown below, where maps are defined using VCC's λ notation, and the constants TagRequest() and TagResponse() are separately defined to be the values of type bytes representing the 1-byte byte strings whose byte is a binary 1 and a binary 2, respectively. We use VCC to verify the body of init(), which serves as proof

that the validity invariants are indeed true on the initial cryptographic state (composed of the empty log, and the table containing only protocol constants, marked as adversary guesses).

The function returns, as an out-parameter, a claim c on the global cryptographic state, that guarantees that it can never be opened during execution, so its invariants (including two-state invariants) are maintained even in the presence of interference from interleaved threads. Thus, the second sort of instrumentation in C′ passes the claim c as ghost parameter to each function and procedure call in C, in accord with their contracts in `RPCshim.h`. For example:

```
att_run_server(s _(ghost c));
```

The init() function

```
_(ghost void init(out \claim c)
  _(writes \extent(&CS))
  _(ensures \wrapped(&CS))
  _(ensures \wrapped(c) && \active_claim(c))
  _(ensures \claims(c, (&CS)->\closed))
  {
    // Initialize to empty log and table
    CS.L = λevent e; \false;
    CS.T.DefinedB = λbytes b; \false;
    CS.T.DefinedT = λterm t; \false;

    // Add protocol constants (tags) to log as attacker
         guesses
    CS.L[New(Literal(TagRequest()),AttackerGuess())] = \true;
    CS.L[New(Literal(TagResponse()),AttackerGuess())] = \true;

    // Add protocol constants (tags) to table as Literalss
    CS.T.DefinedB[TagRequest()] = \true;
    CS.T.B2T[TagRequest()] = Literal(TagRequest());
    CS.T.DefinedT[Literal(TagRequest())] = \true;
    CS.T.T2B[Literal(TagRequest())] = TagRequest();

    CS.T.DefinedB[TagResponse()] = \true;
    CS.T.B2T[TagResponse()] = Literal(TagResponse());
    CS.T.DefinedT[Literal(TagResponse())] = \true;
    CS.T.T2B[Literal(TagResponse())] = TagResponse();

    // Establish invariant and state claim
    \wrap(&CS);
    c = \make_claim({ &CS }, \true);
  })
```

To help us explain why verification goes through, preceding each procedure call f($\overline{v}$) and function call v = f($\overline{v}$) in C′ we assert a conjunction of the form

$$\backslash\textbf{wrapped}(v0)\&\&\ \ldots\ \&\&\ \backslash\textbf{wrapped}(vj)$$

where v0, ... , vj are the pointer variables that have been assigned to up to this point. By induction on the length of C, we argue that each of these assertions holds, and moreover the type invariants are maintained. An assignment, say v = f(v0,...,vn);, satisfies the preconditions of f owing to the added claim, the requirement that v0, ..., vn were previously assigned, and the assertion inserted assertion. By inspection of the contracts for each f in `RPCshim.h` (for example, att_hmacsha1() given above), that is all that is needed. The postcondition of f ensures that results are wrapped, so in particular x is wrapped at the next assertion (and the claim maintained). □

Theorem 1. *Assume* $\emptyset \vdash$ `crypto.c` $\rightsquigarrow$ `crypto.h`. *For any attack program M.c against the interface erase*(`RPCshim.h`), *the program* `System` · `RPCshim.c` · `M.c` *is safe.*

Proof. We have verified with VCC that:

$$\texttt{crypto.h} \vdash (\texttt{RPChybrids.c} \cdot \texttt{RPCprot.c} \cdot \texttt{RPCshim.c}) \rightsquigarrow \texttt{RPCshim.h}$$

By assumption $\emptyset \vdash$ `crypto.c` $\rightsquigarrow$ `crypto.h` and Lemma 1 we get

$$\emptyset \vdash (\texttt{crypto.c} \cdot \texttt{RPChybrids.c} \cdot \texttt{RPCprot.c} \cdot \texttt{RPCshim.c}) \rightsquigarrow \texttt{RPCshim.h}$$

That is, we have $\emptyset \vdash (\texttt{System} \cdot \texttt{RPCshim.c}) \rightsquigarrow \texttt{RPCshim.h}$ by definition of `System`. By Lemma 1, since *M.c* is an attack program for *erase*(`RPCshim.h`), we get $\texttt{RPCshim.h} \vdash \texttt{M.c} \rightsquigarrow \texttt{main.h}$. Hence by Lemma 1 we get

$$\emptyset \vdash (\texttt{System} \cdot \texttt{RPCshim.c} \cdot \texttt{M.c}) \rightsquigarrow \texttt{main.h}$$

So by Assumption 1 the program $\texttt{System} \cdot \texttt{RPCshim.c} \cdot \texttt{M.c}$ is safe. □

This theorem admits the following informal corollary: For all applications *A* verified against `RPCprot.h` and the rest of the API (excluding `RPCshim.h`), $A \cdot \texttt{RPCprot.c} \cdot \texttt{RPChybrids.c} \cdot \texttt{crypto.c}$ is safe in the presence of any active network adversary (in the symbolic model of cryptography).

3.4 Summary of Empirical Results

In this section, we summarize our experimental results on implementations of RPC and of the variant of the Otway-Rees protocol presented by Abadi and Needham Abadi and Needham [1996].

3.4.1 Results

We prove authentication properties of the implementations using non-injective correspondences, expressed as assertions on a log of events, by relying on weak secrecy properties, which we prove formally as invariants of the log. The adversary controls the network, can instantiate an unbounded number of principals, and can run unbounded instances of each protocol role —but can never cause a correspondence assertion to fail and can never break the secrecy invariants, unless the Dolev-Yao assumption (no collisions or lucky guesses) has already been violated. In particular, we prove the following properties about our sample protocol implementations.

RPC

Our implementation of RPC does not let the server reply to unwanted requests, and does not let the client accept a reply that is not related to a previously sent request. Moreover, their shared key remains secret unless either the client or the server is compromised by the adversary.

Otway-Rees

The initiator and responder only accept replies from the trusted server that contain a freshly generated key for their specific usage, and this key remains secret unless either the initiator or the responder is compromised.

As both a consequence and a requirement to using a general purpose verifier, we also prove memory safety properties of our implementations. This can significantly slow verification, especially in parts of the code that handle the building of messages by catenation, and is a large part of the annotation burden.

3.4.2 Performance

Table 3.1 shows verification times, as well as lines of code (LoC) and lines of annotation (LoA) estimations for various implementation files, and offers a comparison of annotation burden and verification time between the original implementation of the methodology as described by Dupressoir et al. [2011] and the one described in this dissertation, that leverages more recent VCC features and performance improvements to provide a more general presentation. Times are given as over-approximations of the verification time (on a mid-end laptop), in seconds. The number of lines of annotation includes the function contracts, but not earlier definitions. For example, when verifying a function in `hybrids.c`, all definitions from `symcrypt.h` can be used but are not counted towards the total. The shim and sample attack programs are verified, as part of the proof of the security theorem, but they are not part of the protocol verification and so are omitted here. However, the significant speedup observed on the protocol code is also observable on the shim and attack code (all functions in the RPC shim verify under 1 second except for the att_run_client() function that performs many copy operations on public byte arrays, having to prove at each step that all invariants are preserved). The current version of the Otway-Rees shim assumes, for simplicity, a special semantics for some function calls, as the threads running the initiator and responder role should be able to return a value to the adversary, which requires some more glue code in C. It is possible to write and verify this glue code using VCC, but it makes the code that much more complex to understand and is not relevant to the protocol's security.

The built-in support for induction in VCC, including the stronger type-checking that comes with

File/Function	LoC	Preliminary Version Dupressoir et al. [2011]		Present Version	
		LoA	Time (s)	LoA	Time (s)
`symcrypt.h`	-	50	≤ 1	5	-
`table.h`	-	50	≤ 15	30	≤ 1
`RPCdefs.h`	-	250	≤ 15	200	≤ 5
`hybrids.c`	150	300	≤ 300	200	≤ 30
destruct()	20	40	≤ 300	20	≤ 5
hmacsha1()	20	20	≤ 10	20	≤ 5
`RPCprot.c`	130	80	≤ 900	130	≤ 60*
client()	40	20	≤ 300	30	≤ 30*
server()	40	10	≤ 600	30	≤ 30*
`ORprot.c`	300	100	~ 6000	100	≤ 200*
initiator()	40	15	≤ 300	18	≤ 30*
responder()	100	100	$\perp$ (out of memory)	30	≤ 140*
server()	40	15	~ 1800	30	≤ 40*

Table 3.1: Comparison showing changes in number of lines of annotation and improvements in verification times between a preliminary version of our system [Dupressoir et al., 2011] and the version presented here, for the same C implementations of Authenticated RPC and Otway-Rees.

it, allows the verifier to heavily optimize the background axiomatization for the inductive types. It also allows a much more succinct and clear definition of the security model, leading to smaller background axiomatizations, and facilitates debugging by making error messages more informative. Additionally, the number of memory-safety-related function contracts has been dramatically reduced by a recent overhaul of the internal axiomatization of the memory model, leading to similar (if not better) verification times with less annotations. The verification times marked with a * above were obtained by passing non-standard options to VCC, changing Z3's case splitting heuristics. Verification times without this option still show significant improvement over the previous figures (the authenticated RPC client then verifies in 60 seconds).

3.5 Related Work and Discussion

We conclude the first part of this dissertation by discussing the results obtained in Chapters 2 and 3 in light of existing and related work on proving *symbolic* security properties of implementations in low-level languages. We conclude with a discussion on extending our work to more flexible and more realistic models of cryptography.

3.5.1 Performance Improvements and Extensions to Stateful Systems

Polikarpova and Moskal [2012] show that the original version of our methodology [Dupressoir et al., 2011] can be adapted to verify security properties of stateful devices. They modify this approach slightly, in that they keep track of subsets of the Level Low and Level High predicates as map-encoded sets. They then use invariants to encode the inductive definition from the top-down: for example, when our inductive rule for pairs states that a low pair can be constructed from two low terms, their invariants would state that if a pair is low, it must be that its two components are low. Since VCC then quantifies over all states of the maps on which the invariants hold, the two approaches are theoretically equivalent. However, in practice, the use of invariants instead of axioms gives their method a significant performance advantage, and a slightly lighter annotation burden, at the cost of performing in VCC some of the security-specific reasoning, which we prefer to do in a more general tool, in line with our objective of separating security proof from low-level code and tools.

We have not compared the performance improvements reported here to those reported by Polikarpova and Moskal [2012], and in particular have not attempted to directly model and verify stateful systems in our symbolic framework. However, we do consider stateful systems in Chapters 4 and 5.

3.5.2 Verifying Java Implementations

There are approaches for verifying implementations of security protocols in other widely-used implementation languages, notably Java.

Jurjens [2006] describes a specialist tool to transform a Java program's control-flow graph to a Dolev-Yao formalization in FOL which is then verified for security properties with automated theorem provers such as SPASS.

O'Shea [2008] translates Java implementations into formal models in the LySa process calculus [Bodei, Buchholtz, Degano, Nielson, and Nielson, 2003] so as to perform a security verification.

The VerifiCard project [Hubbers, Oostdijk, and Poll, 2004] uses the ESC/Java2 static verifier to check conformance of JavaCard applications to protocol models.

Kusters, Truderung, and Graf [2012] present a framework to prove, in the computational model of cryptography, strong equivalence-based secrecy properties of Java programs that use cryptography.

We discuss it further in Chapter 4.

Java and C, as programming languages, are distinct enough that, although some verification issues are common, most of the challenges encountered in one are fairly easily solved in the other (for example, aliasing is non-existent in Java, but omnipresent in C, whereas C does not have dynamic dispatch, which causes issues when verifying Java programs). Therefore, tools for Java cannot trivially solve our problems on the C language, and both languages are widely used for implementing security systems in different environments. We see the two lines of work as complementary, but mostly disjoint.

3.5.3 About Code Generation

Mukhamedov, Gordon, and Ryan [2013] perform a formal analysis of the implementation code of a reference implementation of the TPM's authorization and encrypted transport session protocols in F#, and automatically translate it into executable C code. We believe that such an approach is promising. However, developers of security code often write programs in ways that mitigate, for example timing and power attacks. As there are currently no formal treatments of such concerns in program verification (indeed, many parameters that are not fixed in the source language come into play when considering timing (cache misses, concurrency) or power consumption (hardware details)), we cannot guarantee that the generated code is resilient to these attacks, and any manual modification to the code could break security theorems obtained during the code generation.

Still, annotations for the verification of security properties could be generated along with the code, allowing the developer to re-prove security incrementally as he optimises the program and implements mitigation measures.

3.5.4 Verifying C Implementations

On Dynamic Bug-Finding Jeffrey and Ley-Wild [2006] present DYC, a C API for symbolic cryptographic protocol messages that can be used to generate executable protocol implementations, which dynamically generate constraints that a constraint solver can search for attacks. Code is checked by model-checking a finite state space rather than being fully verified like it is in our approach.

Similarly, white and grey box testing techniques [Godefroid and Khurshid, 2002; Godefroid et al.,

2005] have been used to partially explore the state space of cryptographic programs, searching it for vulnerabilities. If such dynamic and test-based techniques constitute good bug-finding tools, they are not sufficient to prove the absence of vulnerabilities, since they only deal with partial models of the program.

Extracting Verifiable Models As briefly discussed in Chapter 1, some existing and related work on proving symbolic security properties of C programs focuses on extracting, from the C code, models of the implemented protocols on which security-specific tools can be run.

Chaki and Datta [2008] implement a verification framework based on predicate abstraction and model-checking, and successfully apply it to a stripped down version of OpenSSL. Due to the way ASPIER uses model-checking, it is limited to bounded numbers of instantiations (in practice 2 or 3) of the different protocol roles. The authors only describes an abstract process that cannot be applied directly to C code and requires a substantial amount of manual abstraction. Moreover, the approach relies on trusted semantic description of subroutines which are not proven and does not appear to be modular enough to later discharge those assumptions. On the other hand, the use of model-checking provides much flexibility by providing explicit attack candidates when the proof fails. These can then be used to refine the abstraction if they are in fact spurious.

Goubault-Larrecq and Parrennes [2005] do so by abstract interpretation, providing a sound transformation of C code into a decidable subset of first-order logic. The extracted first-order model of the protocol can then be run through an automated theorem prover and weak secrecy properties can be proved. This initial technique does not verify the correctness of the needed annotations and does not support integrity properties. The technique is applied to an implementation of the Needham-Schroeder protocol, failing to prove secrecy on the original, flawed protocol, and succeeding on the fixed Needham-Schroeder-Lowe protocol.

Corin and Manzano [2011] extend the KLEE symbolic execution engine to represent the outcome of cryptographic algorithms symbolically, and other recent work [Aizatulin et al., 2011b, 2012] extracts verifiable ProVerif (or CryptoVerif) models by symbolic execution of C protocol code. If the former tool is not applied to protocol code, the latter set of tools is applied to code similar to the code studied in this Chapter, although only providing security results under the assumption that valid executions of the protocol follow a single "main" control path through the code. However, Aizatulin et al.

[2011b, 2012] can go further in this limited case, and in fact provide security guarantees in the, more realistic, computational model of cryptography.

3.5.5 Symbolic and Computational Models of Cryptography

We believe verification in the symbolic model of cryptography is still relevant: as recent logical flaws in established protocols and implementations illustrate, no amount of attack finding and patching will ever eliminate all security flaws. In addition, proving the absence of symbolic flaws eliminates a vast number of very cheap attacks, forcing a potential intruder to use more advanced techniques to break the system's security, increasing the value of assets that it can protect.

However, recent events have shown that adversaries are becoming more powerful, sometimes with government backing, and can mount more complex attacks that are outside the simple free-algebra-based adversary model we have considered so far.

To remedy this issue, we could, for example, extend our symbolic model with an equational theory such as those used in ProVerif [Blanchet, 2001], or indeed the theory of computationally complete symbolic attackers recently introduced by Bana and Comon-Lundh [2012]. Our early axiomatic encoding of cryptography in VCC [Dupressoir et al., 2011], for example, could be used to encode such models of cryptography. However, reverting to such an encoding would also, at least partially, revert the performance gain reported here, partly due to VCC's ability to optimise our inductive axioms for consumption by Z3. It is also unclear how the generality of the framework can be preserved with finer-grained models of cryptography, which may therefore cause more security-related proof obligations to bleed into the C verification task.

Even with such fine-grained symbolic models, however, the usual computational notion of secrecy, which is equivalence-based, cannot be precisely and soundly abstracted symbolically as a trace property, and can therefore not be handled directly by existing C verification tools. Still, Klein et al. [2009], for example, report on work where a notion of forward simulation is used to prove equivalence-based information-flow properties of C programs using a weakest precondition calculus. In Chapters 4 and 5, we show how a similar notion of simulation can be used to prove security properties of C programs in the computational model of cryptography, at a cost we believe is comparable to the cost of proving those same properties in the fine-grained symbolic models discussed above.

Chapter 4

Computational Security for C Programs

In this chapter, we present some standard notions and proof techniques for security properties in the computational model of cryptography, focusing on recent code-based approaches. We then present general notations and notions that allow us to express and prove computational security properties of C programs.

4.1 Computational Models of Cryptography

Computational models of cryptography present the adversary and algorithms as probabilistic polynomial time (p.p.t.) programs. In such models, notions of security must account for the adversary's bounded computational power as well as for small probabilities of security failures, even when using correct algorithms.

4.1.1 Cryptographic Properties

Security and cryptographic properties are often expressed using cryptographic games [Goldwasser and Micali, 1984], letting a p.p.t. adversary make queries to a set of oracles (abstract probabilistic functions), and expressing the adversary's winning condition as an observable event. An algorithm is said to be secure w.r.t. a given game if the probability of an adversary winning the game is small,

as a function of some security parameters.

We now informally describe standard games for integrity and secrecy.

An Integrity Game For example, an HMAC primitive is defined as a set of three oracles: a key generation oracle Gen, an HMAC oracle Mac, and a verification oracle Verify. A standard property on HMAC schemes is unforgeability under chosen message attack (or INT-CMA), which expresses the fact that an adversary who is allowed to make a polynomial number of queries to the Mac oracle for a fixed key can only create a fresh valid MAC (that was not previously returned by the oracle) with a very small probability.

The CMA game starts with a setup phase, in which the Gen oracle is used to generate the key used in the rest of the game. The adversary is then allowed to make as many queries to Mac as he desires. Finally, the adversary produces two bitstrings: a payload t and a MAC m. The adversary wins if $\mathrm{Verify}(t, m)$ returns true, and the pair (t, m) was not produced by one of the oracle calls.

A Secrecy Game If integrity is still in fact a trace property (of an unknown, but restricted, adversary), secrecy in the computational model is often expressed as an equivalence.

We consider encryption schemes that have three oracles: a key generation oracle Gen, an encryption oracle Enc, and a decryption oracle Dec. The following describes a game modelling the standard property of indistinguishability under chosen plaintext attacks (IND-CPA), where the adversary is allowed to encrypt a large number of plaintexts before attempting to distinguish between the real encryption or an ideal version of it (or, equivalently, between encryptions of two equal-length adversary-provided plaintexts).

In the setup phase, a key is generated using Gen, and a bit b is sampled uniformly at random. The adversary is then allowed to query the Enc oracle with *pairs* of plaintexts (m_0, m_1), and is given in response the encryption of m_b. Finally, the adversary produces a bit b', and wins if $b = b'$.

Stronger notions of secrecy also give the adversary the ability to call the decryption oracle.

4.1.2 Computational Security for F# Programs

If games have historically been used to express security, they are not well-suited to standard techniques for automated reasoning on programs, and although they can be formalised and automated to some extent [Barthe, Grégoire, and Zanella Béguelin, 2009; Barthe, Grégoire, Heraud, and Béguelin, 2011; Blanchet, 2008], the resulting tools often support only very simple languages that are not directly executable.

Fournet, Kohlweiss, and Strub [2011] describe a methodology for proving computational security properties of F# programs by type-checking. If the final security result can be expressed using games, the proof itself is done by proving that the program considered is almost equivalent to some other program, the ideal functionality, on which the security property is easy to establish.

Computational Indistinguishability Approximate program equivalence is expressed using the notion of *computational indistinguishability* [Goldreich, Goldwasser, and Micali, 1986] which we describe informally here, and define formally in Section 4.3.1.

Let $\mathcal{O}$ stand for a set of oracles. We denote with $\mathcal{A}(t, (q_o)_{o\in\mathcal{O}})$ the set of probabilistic adversaries that run in time at most t and make at most q_o queries to oracle o in $\mathcal{O}$.

Let P be a probabilistic program that implements all oracles in $\mathcal{O}$, and A be in $\mathcal{A}(t, (q_o)_{o\in\mathcal{O}})$. We write $\Pr\left[(\langle\mathcal{M}\rangle\ P \cdot A\ \langle b\rangle)\right]$ for the probability that A running in conjunction with P in memory $\mathcal{M}$ returns b.

Given two programs P and P' that both implement all oracles in $\mathcal{O}$, their distinguishing advantage is defined as

$$\mathsf{Adv}_P^{P'}\left(t, (q_o)_{o\in\mathcal{O}}\right) \stackrel{def}{=} \sup_{A\in\mathcal{A}(t,(q_o)_{o\in\mathcal{O}})} \left(\left|\Pr\left[(\langle\mathcal{M}_0\rangle\ P \cdot A\ \langle 0\rangle)\right] - \Pr\left[\left(\langle\mathcal{M}_0\rangle\ P' \cdot A\ \langle 0\rangle\right)\right]\right|\right),$$

where $\mathcal{M}_0$ is the empty memory.

To enable modular automated reasoning about such properties, Fournet et al. [2011] identify some typing conditions on programs using a primitive, under which that primitive can be replaced with its ideal functionality, for some common primitives.

Ideal Functionalities and Perfect Security by Typing Ideal functionalities are sometimes used in cryptography as a way to express security goals [Canetti, 2001]. However, when dealing with executable programs, even in a languages as clean as F#, the ideal functionalities considered are often too large and complex to serve directly as a security definition. Instead, Fournet et al. [2011] identify simple typing conditions that can be used to prove integrity properties (as type-safety with refinement types) and secrecy properties (using relational parametricity).

Security Proofs by Typing Proofs of security for large F# programs [Bhargavan, Fournet, Kohlweiss, Pironti, and Strub, 2013, for example] can then be conducted by successively proving the typing conditions to replace the cryptographic primitives with their ideal functionalities, obtaining an *idealised program* where all cryptography is ideal, and that can be proved perfectly secure by typing.

4.2 Overview

Before giving the details of our method, we discuss the challenges we face in attempting to prove computational security of C programs, and use them to justify our choices, giving a brief abstract overview of our techniques.

4.2.1 Challenges

Probabilistic Reasoning To adapt the F7 type system to be sound in the computational model, Fournet et al. [2011] rely on meta-theorems about a probabilistic semantics of the core RCF language. However, it is difficult to see how such a meta-theorem could be stated, let alone proven, on a complex language such as C. On the other hand, developing a verification tool to reason probabilistically about C programs (in the fashion of EasyCrypt [Barthe et al., 2011]) seems intractable, and made even more difficult by the language's many unspecified behaviours, which can only be soundly modelled with possibilistic non-determinism unless assumptions are made on the compilation and run-time environment.

Equivalence Properties As we've seen, secrecy properties in the computational model are expressed as program equivalence properties. However, all existing general-purpose C verifiers are

designed to prove safety properties and functional correctness, neither of which is sufficient to express equivalence-based security properties directly at the level of the C program.

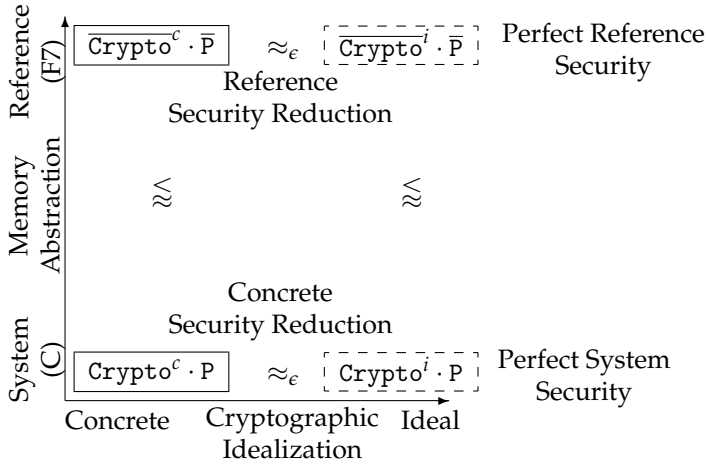
4.2.2 Our Approach

Instead, we choose to perform the security proof on an abstract *reference implementation*, which can then be used as a functional specification for a *system implementation*, which therefore inherits the same security properties.

Security is proved by reducing the security of a program using a cryptographic library $\overline{\texttt{Crypto}}^{c}$ of concrete primitives to the security of the same program, linked with an ideal cryptographic library $\overline{\texttt{Crypto}}^{i}$, as described in Section 4.1.

We then define a notion of program simulation, saying that a system program $\texttt{P}$ simulates a reference program $\overline{\texttt{P}}$ for some observation function whenever all final configurations of $\texttt{P}$ get observed as the final configuration of $\overline{\texttt{P}}$ run on the observation of $\texttt{P}$'s initial observation. We define this notion for probabilistic programs by derandomisation, explicitly passing a random tape in as argument. This allows us to preserve, through the simulation relation, probabilistic properties proved on the reference implementation.

Therefore, our approach operates on and relates four versions of the program, as described in the diagram below.



Perfect Reference Security and Reference Security Reduction can be combined to provide a Concrete

Reference Security theorem, bounding the probability that a probabilistic polynomial time (p.p.t.) adversary breaks the security property of the concrete reference program $\overline{\texttt{Crypto}}^{C} \cdot \overline{\texttt{P}}$.

A general-purpose verifier for the system language can then be used to prove that $\texttt{P}$ simulates $\overline{\texttt{P}}$ whenever the system cryptographic library simulates the reference cryptographic library, and we can conclude, after some meta-reasoning about the adversary and the observation functions involved in the simulation, that the same security properties hold on the system programs.

In the solution presented here, the reference results are proved using the F7 type system as discussed by Fournet et al. [2011], and the simulation proofs are performed in VCC. However, the general notion and approach are defined independently of the languages and tools.

We use the rest of this Chapter to formally define the symbols $\approx_\epsilon$ and $\lessapprox$ used in the diagram.

4.3 Probabilistic Program Semantics and Indistinguishability

We introduce notations for probabilistic program semantics, both for C and F7 programs.

4.3.1 Notations for Possibilistic Semantics

All C and F7 programs are seen as relations between initial configurations and final configurations. We use write P for programs written in an unspecified language, or for mathematical relations, $\texttt{P}$ for programs written in C, and $\overline{\texttt{P}}$ for programs written in F#.

We denote with $\langle a, S\rangle \ P \ \langle a', S'\rangle$ the fact that executing a program P on argument a in initial state S may return value a' in final state S'. In the above, we call $\langle a, S\rangle$ the *initial configuration*, whereas $\langle a', S'\rangle$ is the final configuration, and are later often simply denoted $\langle C\rangle$ and $\langle C'\rangle$. This notation makes it clear that a program implements a relation between initial and final configurations, which we call its *input-output relation* and identify with the program itself, slightly abusing notation.

A *contract* on a program with argument type τ and return type τ' is a pair of predicates $\pi \in \tau \times \mathit{State} \to \mathit{bool}$ and $\rho \in \tau \times \mathit{State} \times \tau \times \mathit{State} \to \mathit{bool}$, and write $\{\pi\}\ P\ \{\rho\}$ when P has argument type τ and return type τ' and whenever $\pi\,(a, S)$ and $\langle a, S\rangle \ P \ \langle a', S'\rangle$, we have $\rho\,(a, S, a', S')$. In the following, we consider that $\langle C\rangle \ P \ \langle C'\rangle$ only when P's precondition holds on $\langle C\rangle$, that is, programs block when run on configurations where their preconditions do not hold.

We extend these notations to open programs, where some function symbols p_i are left to be defined. We write $\{p_i := P_i\}_i \vdash \langle C\rangle\ P\ \langle C'\rangle$ to express the fact that program P, when using program P_i as definition for symbol p_i, may take initial configuration C to final configuration C'. We denote with $P \cdot Q$ the program obtained by using the definitions in P for free symbols in Q, modelling C linking, or F# module composition.

This modularity also extends smoothly to contracts, and we write $\{\{\pi_i\}\ p_i\ \{\rho_i\}\}_i \vdash \{\pi\}\ P\ \{\rho\}$ whenever for any family $(P_i)_i$ such that $\{\pi_i\}\ P_i\ \{\rho_i\}$ for all i, for any C such that $\pi(C)$, and for any C' such that $\{p_i := P_i\}_i \vdash \langle C\rangle\ P\ \langle C'\rangle$, we have $\rho(C, C')$. Intuitively, this expresses the fact that P fulfills its contract whenever each of the p_i is defined as a program that fulfills its contract.

4.3.2 Observational Determinism

Before we can define a simple notion of probabilistic execution that is amenable to automated reasoning, we restrict the set of programs we are interested in, not to fully deterministic programs, but to programs that appear deterministic with respect to a certain observation function α (for example, a set of observable memory locations).

Definition 3 (Observational Determinism)**.** *Given an* observation function α *from final configurations (with value type τ), we say that a program P is* α-deterministic *iff whenever $\langle C\rangle\ P\ \langle C'_0\rangle$ and $\langle C\rangle\ P\ \langle C'_1\rangle$, we have $\alpha\left(C'_0\right) = \alpha\left(C'_1\right)$.*

For example, if π_1 is the first projection, any program that is π_1-deterministic always yields, when run in the same initial configuration, the same return value, although it can finish in different final states.

We denote *execution up to* α with $\alpha \circ \cdot$, defined by composing the observation function, seen as a relation, with the simple reduction relation. It is easy to see that, if a program P is α-deterministic, then $\alpha \circ P$ is in fact a function, by definition.

4.3.3 Derandomized Probabilistic Semantics

In the following, we assume that all configurations contain, in some fixed location, a finite random tape, of a length l which becomes a parameter of the system, and can be chosen once the full program is known. For any configuration $\langle C\rangle$, let $\langle C\rangle\,[r]$ be the configuration whose random tape contains

value r, and is everywhere else equal to $\langle C \rangle$.

Simply counting the number of random tapes that produce a given final configuration does not give rise to a well-founded probabilistic semantics, as the reduction relation $\langle\rangle \cdot \langle\rangle$ is not generally a function. We thus restrict our definition to programs that are observationally deterministic, which we check as part of our machine proof (see Lemma 2).

For an α-observation ω, and an α-deterministic program P, the probability of an execution of P on a in S yielding observation ω is defined by counting the number of random tapes of length l that yield ω, as follows.

$$\Pr\left[\langle a, S\rangle\ \alpha \circ P\ \langle\omega\rangle\right] \stackrel{def}{=} \frac{\left|\left\{r \in \{0,1\}^l \;\middle|\; \langle a, S\rangle\ [r]\ \alpha \circ P\ \langle\omega\rangle\right\}\right|}{2^l}$$

In the rest of this paper, we use the word "(α-)deterministic" to describe programs that behave (α-)deterministically when the random tape is fixed, thus abstracting away all possibilistic non-determinism but modelling probabilistic behaviours.

4.3.4 Indistinguishability

We can now formally define program indistinguishability for deterministic programs.

Definition 4 (Distinguishing Advantage)**.** *Given two deterministic programs P and Q that implement all oracles in an interface $\mathcal{O}$, we define the* distinguishing advantage *for P and Q as*

$$\mathsf{Adv}_P^Q\left(t, (q_o)_{o\in\mathcal{O}}\right) \stackrel{def}{=} \sup_{A \in \mathcal{A}(t,(q_o)_{o\in\mathcal{O}})} \left|\Pr\left[\langle\mathcal{M}_0\rangle\ P \cdot A\ \langle 0\rangle\right] - \Pr\left[\langle\mathcal{M}_0\rangle\ Q \cdot A\ \langle 0\rangle\right]\right|.$$

If P and Q are not deterministic, but are observationally deterministic for some observations α and α' (respectively), then $\alpha \circ P$ and $\alpha' \circ Q$ are deterministic, and the definition above applies provided that $\alpha \circ P$ and $\alpha' \circ Q$ fully implement the oracles in $\mathcal{O}$.

4.3.5 VCC and Abstract Contracts

In this and the following chapter, we consider only sequential programs, and only care about properties of the input-output relation they implement, as opposed to stating our verification goals as

intermediate assertions, as was the case in Chapter 3.

We can therefore simplify the notations introduced in Section 1.3.2 to align them with those describing abstract contracts on relations introduced in this Section. In particular, in the case of sequential programs, the thread pool consists of a single thread (which we recall contains a continuation and a local store), and, in the absence of network communications, the message queue can be dropped.

These simplifications to VCC's semantics allow us to simply consider, when stating the theorems, the simple case where type invariants are irrelevant, and we rely only on function contracts to express our verification goals.

In this context (and in this context only), whenever we can prove in VCC that $\mathtt{P.h} \vdash \mathtt{f} \rightsquigarrow \mathtt{f.h}$, where $\mathtt{f.h}$ is a single function contract $\{\pi\}\ \mathtt{f}\ \{\rho\}$ for the function f, we get that $\{\{\pi_i\}\ \mathtt{P}_i\ \{\rho_i\}\}_i \vdash \{\pi\}\ \mathtt{f}\ \{\rho\}$, where the $\{\pi_i\}\ \mathtt{P}_i\ \{\rho_i\}$ are the function contracts listed in $\mathtt{P}'\mathtt{.h}$.

Intuitively, successful verification for f implies that, for all implementations $\mathtt{P}_i$ of f's dependencies that fulfill their contracts, the program composed of f linked with the $\mathtt{P}_i$ fulfills f's contract.

Note that this does not prevent the VCC verification itself from making use of VCC's advanced features, but only simplifies the theorem statements into simple abstract contracts that can be proved using VCC.

4.4 Simulation

The purpose of this section is to prove Theorem 7, which states that, for each of the Device's commands, the C code implements the same observable input-output function as the F7 code, when the cryptographic libraries they use also behave in the same way.

We define a semantic notion of simulation between a *system program* P and a *reference program* $\overline{\mathtt{P}}$. We instantiate the building blocks of the simulation relation and demonstrate how simulation can be proved using VCC. Finally, we apply the methodology to the Device code, proving Theorem 7 by running VCC.

In the following, we consider programs that can only interact with their random tape through a restricted interface (our interface currently only allows the code to read linearly from the random tape), and we consider only observation functions that preserve the random tape. In practice, the C

programs do not take an explicit random tape, which therefore appears only in ghost code, which cannot be read or written to by the C code, and the F# programs are written with sampling operations that prevent the code from interacting directly with the random tape.

4.4.1 Definition of Simulation

Definition 5 (Function Simulation)**.** *Given a system program* $\mathtt{P} : \tau_S \to \tau'_S$*, and a reference program* $\overline{\mathtt{P}} : \tau_R \to \tau'_R$*, initial observation function* $\alpha \in \langle \tau_S \times State_S \rangle \to \langle \tau_R \times State_R \rangle$ *and final observation function* $\alpha' \in (\tau'_S \times State_S) \to (\tau'_R \times State_R)$*, we say that* P simulates $\overline{\mathtt{P}}$ *for observations* α *and* α'*, denoted* $\mathtt{P} \ {}_{\alpha}\lessapprox_{\alpha'} \overline{\mathtt{P}}$*, iff whenever* $\langle C \rangle \ \mathtt{P} \ \langle C' \rangle$*, we have* $\langle \alpha\,(C) \rangle \ \overline{\mathtt{P}} \ \langle \alpha'\,(C') \rangle$*.*
As with the semantic notations, we write $\left\{ \mathtt{p}_i \ {}_{\alpha_i}\lessapprox_{\alpha'_i} \overline{\mathtt{p}}_i \right\}_i \vdash \mathtt{P} \ {}_{\alpha}\lessapprox_{\alpha'} \overline{\mathtt{P}}$ *to say that, if it is true for all* i *that* $\mathtt{p}_i \ {}_{\alpha_i}\lessapprox_{\alpha'_i} \overline{\mathtt{p}}_i$*, then it is true that* $\mathtt{P} \ {}_{\alpha}\lessapprox_{\alpha'} \overline{\mathtt{P}}$*.*

In general, the initial and final observation functions cannot be equal, since the final configuration may include a return value (that it, τ'_S and τ_S are different). In practice, this becomes particularly useful when considering C programs where output parameters are passed in by reference. In such a case, it is often desirable to ignore the output variable in the initial observation, to express the fact that the final result does not in fact depend on its initial (most likely unknown) value. This can also be used to ignore the input variables (which are also part of the final configuration) in the final observation, allowing the C code to modify memory in place. Examples of such observation functions are given in Section 5.2.2.

4.4.2 Simulation as a Contract

We now show how abstract contracts can be used to express this simulation relation, *for a deterministic terminating reference language*. For an program $\overline{\mathtt{P}}$ written in such a deterministic language, the input-output relation $\langle \cdot \rangle \ \overline{\mathtt{P}} \ \langle \cdot \rangle$ is a total function and we can write $\overline{\mathtt{P}}\,(C) = C'$ instead of $\langle C \rangle \ \overline{\mathtt{P}} \ \langle C' \rangle$.

Given a reference program $\{\pi'\} \ \overline{\mathtt{P}} \ \{\rho'\}$, a system program $\{\pi\} \ \mathtt{P} \ \{\rho\}$, and appropriately typed initial and final observation functions α and α' such that, for all initial (resp. final) system configuration C (resp. C') such that $\pi(C)$ (resp. $\rho(C')$), we have $\pi'(\alpha(C))$ (resp. $\rho'(\alpha'(C'))$) P's contract can be extended as follows, to prove that P simulates $\overline{\mathtt{P}}$ for α and α'.

$$\{\pi\} \cdot \{\rho \wedge \alpha'(C_f) = \overline{\mathtt{P}}\,(\alpha\,(C_i))\}$$

In the formula above, and in all the following, C_i and C_f are the initial and final configurations (respectively).

Both of the original system and reference contracts ($\{\pi\} \cdot \{\rho\}$ and $\{\pi'\} \cdot \{\rho'\}$) are, at this point in the reasoning, considered part of the program, and not part of the proof. In practice, interfaces given to the adversary should only have minimal contracts, with preconditions ensuring correct termination (preventing memory and arithmetic safety violations), and postconditions ensuring that the next oracle queries succeed.

Lemma 2 (Simulation by Contract)**.** *For all reference program* $\overline{\mathtt{P}}$*, whenever a system program* $\mathtt{P}$ *is such that* $\left\{\mathtt{p}_i \ {}_{\alpha_i}\!\lessapprox_{\alpha'_i} \overline{\mathtt{p}}_i\right\}_i \vdash \{\pi\}\ \mathtt{P}\ \{\rho \wedge \alpha'(C_f) = \overline{\mathtt{P}}\,(\alpha\,(C_i))\}$*, then* $\mathtt{P}$ *is* α'*-deterministic, and we have* $\left\{\mathtt{p}_i \ {}_{\alpha_i}\!\lessapprox_{\alpha'_i} \overline{\mathtt{p}}_i\right\}_i \vdash \mathtt{P} \ {}_{\alpha}\!\lessapprox_{\alpha'} \overline{\mathtt{P}}$.

Proof. We recall that the reference program $\overline{\mathtt{P}}$ is deterministic, and that both $\mathtt{P}$ and $\overline{\mathtt{P}}$ always terminate on configurations where their preconditions hold, and note that if the condition on contracts and observation functions mentioned above is not fulfilled, either π' does not hold on $\alpha\,(C_i)$ (and $\overline{\mathtt{P}}\,(\alpha\,(C_i))$ is undefined), or ρ' does not hold on $\alpha'(C_f)$, which can therefore not be a final configuration for $\overline{\mathtt{P}}$.

The hypothesis guarantees that $\mathtt{P}$ is α'-deterministic, since on all initial configurations, its result is α'-observed as the result of the reference function.

The original contract $\{\pi\} \cdot \{\rho\}$ guarantees correct termination, and therefore that there exists a final configuration C_f for any initial configuration C_i on which π holds. Since $\overline{\mathtt{P}}$ is deterministic, there is a unique $\hat{C}'$ such that $\overline{\mathtt{P}}\,(\alpha\,(C_i)) = \hat{C}'$. The additional postcondition guarantees that any final configuration C_f is such that $\alpha'\left(C_f\right) = \overline{\mathtt{P}}\,(\alpha\,(C_i))$, and we can conclude that for all final configuration C_f, we have $\alpha'\left(C_f\right) = \hat{C}'$. □

The first conclusion is important since we later want to reason about the probabilistic interpretation of such programs $\mathtt{P}$, which is only defined for observationally deterministic programs.

4.5 Discussion

The notion of simulation we defined in this chapter does not immediately guarantee that security properties of the reference implementation hold on the system implementation. This depends in particular on the adversary model and on the observation functions chosen for the simulation. We have not identified general conditions on the observations that guarantee this even for standard adversary models. Instead, we illustrate on an example, in Chapter 5, how observation functions

and adversary model interact in the final security proofs for the C programs.

The rest of this chapter is dedicated to general discussions about the methodology outlined here, in relation with existing work.

4.5.1 Related Work

As far as we know, the only prior computational security results for C code are those obtained by Aizatulin et al. [2011b, 2012], which rely on symbolic execution to automatically extract, from the C code, a protocol model verifiable using ProVerif (respectively CryptoVerif). The first paper relies on a computational soundness result which prevents its application to many practical examples. Both papers make a strong assumption on the protocol, forbidding any non-trivial branching (that is, all executions that leave an execution path identified as the main path are assumed to immediately terminate with an error). This prevents, in particular, applications to stateful systems such as the one we study in Chapter 5. In addition, the extraction process to ProVerif and to CryptoVerif is not shown to soundly preserve equivalence-based security properties, although the security-specific tools themselves support them.

Kusters et al. [2012] recently presented a framework for proving computational security properties of Java programs using a general-purpose verification tool. They rely on a tool designed to prove equivalence-based properties of Java programs (such tools do not exist for C in usable forms), but is unaware of cryptography. In addition, they do not support the verification of trace-based properties.

4.5.2 Ideal Simulation and Modularity

It would in fact be sufficient, when verifying a whole program, to prove the simulation only on concrete implementations, instead of proving it both on ideal and concrete implementations, as presented in Section 4.2. Indeed, the simulation result could then be applied to transfer the security properties of the concrete reference program to the concrete system program, proving that the latter is computationally indistinguishable from the ideal reference program. In fact, not all security proofs make use of ideal functionalities, and proof techniques that do not can still be used in combination with our notion of simulation to prove security properties of C code.

However, proving ideal simulation allows us to express intermediate indistinguishability results be-

tween C programs, which may provide some measure of modularity in later work.

Modularity would be greatly helped by the identification of simple criteria on observation functions that guarantee the preservation of security properties by the simulation. There is some previous work classifying adversaries against programs according to their observational power, which could be of some use in developing such general criteria (for example, [Preda, Christodorescu, Jha, and Debray, 2007] identifies classes of adversaries with the most precise observation they can make on the state of a system).

4.5.3 Secure Implementations of Primitives

Verifying that the concrete C functions used to implement the cryptographic algorithms are correct and secure, is beyond the scope of this dissertation. We are not aware of any techniques to prove security of cryptographic primitives implemented in C, but related work includes producing formal proofs of computational security for abstract code-based representations of primitives [Barthe et al., 2009, 2011], and producing proofs of equivalence between a (trusted) naive implementation and an optimized implementation, both written in C [Barbosa, Sousa Pinto, Filliâtre, and Vieira, 2010]. We believe that our technique could be applied to proving simulation properties on implementations of primitives, although we have not yet attempted it.

In combination with a modularity result, this would allow security proofs to be obtained on whole system implemented in C, reducing their security properties to standard complexity assumptions usually used in cryptography.

4.5.4 Reference Implementations: From F7 to VCC

As described in Section 4.2, security proofs are performed on F# programs using the F7 type-checker. However, to prove the simulation using VCC, we need to encode those F# programs as VCC specifications.

If the core languages are very similar, they still have many small differences. In particular:

- The VCC specification language is not polymorphic, and F7 specifications should therefore avoid using polymorphism.

- The VCC specification language can only describe pure functions, and the F7 code should therefore avoid using explicit state.

- The F7 type system, even when the code has no explicit state, carries an implicit log of events (and a random tape) that we model as part of the state in VCC.

Most of these issues can be avoided by writing the F7 specification in a certain style, but doing so would impede the security proof. On the other hand, if F7 code can be systematically translated into VCC specifications, these obstacles prevent an automated translation in many cases.

Formally linking the reference languages

In the following chapter, VCC specifications are manually translated from the F7 implementation, and, in the absence of formal semantics for the VCC specification language, no formal link is made (that is, we assume that the VCC specification is equivalent to the F7 implementation).

However, the language's semantics is embodied in its translation into first-order logic, and the resulting axioms could be used as F7 specification to formalise an equivalence proof.

Another solution would be to formalise and improve VCC's specification language itself to perform the proofs of security by typing directly on the VCC version of the reference implementation.

4.5.5 Other Uses of Simulation in Software Verification

Notions of simulation have been used in other aspects of program verification. For example, the seL4 project [Klein et al., 2009] proves that a C implementation of a micro-kernel simulates (by forward simulation) a prototype implementation written in Haskell. The proofs are done by embedding both C and Haskell semantics in the Isabelle proof assistant, and manually building the simulation proof.

We believe that our notion of simulation may be useful in such a context, and that the automation provided by recent advances in general-purpose verification tools could be leveraged to produce proofs similar or larger in scope than those involved in the seL4 project, at the cost of a larger and more complex TCB which would include VCC.

4.5.6 Reference Languages for Reference Implementations

The use of F7 in this work is indeed incidental, as it appeared to be the most adapted tool to produce security proofs on implementations that could easily be turned into VCC specifications. Cryptography-specific tools, such as CertiCrypt, EasyCrypt [Barthe et al., 2009, 2011], or CryptoVerif [Blanchet, 2008] have languages for describing protocols and primitives that could also be used as reference languages, with the same lack of formal link with VCC.

More generally, however, we believe that functional languages provide a great platform for executable specifications, and that the ready availability of advanced verification tools for them should be leveraged, along with simulation techniques, to use them as such in safety or security-critical contexts.

Chapter 5

Computational Security of a Key Management System

In this chapter, we apply the verification methodology described in Chapter 4 to an exemplary key management system, inspired by the TPM [Tru, 2007] and the more recent TPM 2.0 [Tru, 2013].

5.1 The Device

The main goal of our Device is to provide a basic key management and storage functionality.

The Device is a store for a set of *objects*. An object consists of a fixed-length, public *template* and a *key*, whose length is variable, but bounded by an implementation constant. The template represents metadata for the key. Its first byte specifies whether the associated object is *sensitive*; the key of a sensitive object is only used internally to the Device. Its second byte specifies the length in bytes of the key. The rest of the template is used only as an identifier that may not be unique (and could also encode various public parameters for the key).

The Device can generate fresh objects using a key derivation function (KDF), based on a fixed hardware secret, or it can take public bytes from the user and use them, after minor validation steps, as a non-sensitive key. The internal state of the Device, physically shielded from direct outside observation and interference, consists of an immutable key seed (for key generation), an immutable top-level storage key (for the protection of objects offloaded into untrusted memory), and a fixed-size mutable

array of slots, in which objects can be stored while in use. Objects can be moved from shielded to untrusted memory using the Unload command, and in the other direction using the Load command.

Once loaded onto the Device, objects are identified using a *handle*, which is simply an integer index to the slot in which the object is stored. There is no handle associated with the storage key or the primary seed, which can therefore not be referred to directly by the user.

The Device processes two kinds of ciphertexts or *blobs*. The first kind are objects encrypted with the Device's storage key; these blobs are produced and consumed by the Unload and Load commands, respectively. The second kind are user-provided plaintexts encrypted with the keys of protected objects; these blobs are produced and consumed by the Encrypt and Decrypt commands, respectively.

The Device communicates with the user via an *I/O buffer*, and by calling specific commands. Upon receiving a command call, the Device reads and parses the contents of its buffer, according to the command's expected inputs. The Device then processes the command, which results in possible updates to the internal state, and the writing of a return code and a command-specific return value into the I/O buffer.

5.1.1 Instruction Set

The commands are as follows. (Error codes are not detailed in this description; they are more informative, and taken in consideration in the security proof.)

Create takes as input a template. The command returns an error code if there is no empty slot on the Device. Otherwise, the command creates a fresh object associated with the template, stores it in a free slot, and returns a success code, followed by the handle to the created object.

Import takes as input a template, and a byte array to be interpreted as key. The command returns an error code if the template is for a sensitive object, if the byte array is not of the length indicated by the template, or if there is no empty slot on the Device. Otherwise, the command stores the template and data in a free slot as a new object, and returns a success code, followed by the handle to the imported object.

Export takes as input a handle. The command returns an error code if the indexed slot is empty, or if it contains a sensitive object. Otherwise, the command returns a success code together with

the template and the byte array representation of the key.

Clear takes as input a handle. The command marks the slot as empty (but does not necessarily overwrite its contents) and returns a success code.

Unload takes as input a handle. The command returns an error code if the indexed slot is empty. Otherwise, it returns a success code, followed by a blob consisting of the authenticated encryption of the target object, padded to the block size, under the storage key. For simplicity in this example, unloading an object does not clear its slot.

Load takes as input an encrypted blob. The command returns an error code if there is no empty slot on the Device, if decryption of the blob using the storage key fails, or if parsing the successfully decrypted plaintext fails. Otherwise, the command loads the decrypted object into a free slot and returns a success code, followed by the handle of the newly loaded object.

Encrypt takes as input a handle and a byte array containing a plaintext (for simplicity, the length of plaintexts is an implementation constant). The command returns an error code if the handle points to an empty slot. Otherwise, it returns a success code, followed by a blob consisting of the authenticated encryption, under the indexed object's key, of the input plaintext.

Decrypt takes as input a handle and an encrypted blob. The command returns an error code if the handle points to an empty slot, or if decryption of the blob under the indexed object's key fails. Otherwise, it returns a success code, followed by the decrypted plaintext.

5.1.2 Expected Security Properties

General notions of security for key management systems exist [Cachin and Chandran, 2009; Kremer, Steel, and Warinschi, 2011]. However, we use simpler security notions to keep the security proof as simple as possible, since the focus of this chapter is to transfer security results from F# to C code. We expect and prove the following two security properties for the Device and its C implementation, when the adversary can only call the Device commands, and read and write its I/O buffer.

Load Integrity All sensitive objects stored in the Device internal memory have been created on the Device; all non-sensitive objects stored in the Device internal memory have either been created on the Device, or imported onto it.

Security of Encryption under Sensitive Keys Encryption under sensitive keys guarantees the secrecy of the plaintexts and integrity of the ciphertexts.

Integrity properties are expressed using event correspondences: our events for the Device record key creation (parameterised by the seed and template used, and the resulting key), and key import (parameterised by the template and key). We can then write an invariant on the store, ensuring that a correct event has been logged for each stored key, at any point in the execution (in practice, this is even an invariant on the type of keys).

Secrecy is expressed information theoretically as the fact that, when encrypting under a sensitive key, no information about the plaintext is contained in the ciphertext.

However, neither of these security properties can in fact hold unconditionally or perfectly on a real system, since concrete keys and secrets are bounded bitstrings and could be guessed by unrestricted adversaries. We instead make standard assumptions on the cryptographic primitives used in the Device, and reduce the security of the Device to the security of its cryptographic primitives.

5.1.3 Cryptographic Assumptions

We use two cryptographic operations as concrete primitives: a key derivation function (KDF [Chen, 2009]) and an authenticated encryption scheme (AE [Bellare and Namprempre, 2000]).

Key Derivation Function

To generate encryption keys from the primary seed and the template, we use a single instance of a key derivation function $\overline{\texttt{KDF}}$. Let RF be the ideal random function that lazily samples keys and uses a table to store seed-template-key tuples for lookup when the function is later called with the same arguments. The distinguishing advantage between $\overline{\texttt{KDF}}$ and RF is denoted $\mathsf{Adv}^{\overline{\texttt{KDF}}}_{RF}(t, q_{KDF})$, for adversaries that make at most q_{KDF} queries to the key derivation oracle *with a single seed*. (Other notions may let the adversary generate and use multiple seeds; we do not in this example.) Informally, the advantage $\mathsf{Adv}^{\overline{\texttt{KDF}}}_{RF}(t, q_{KDF})$ is meant to be negligible.

Key Derivation Interface: $\overline{\texttt{I}}_{RF}$

```
type seed
val s0: seed

val KDF: t:template → s:seed { s = s0 } → k:(;t) key
```

The following lemma establishes a typing condition that can be automatically discharged using F7,

and that guarantees that A uses $\overline{\texttt{KDF}}$ and RF as required for the advantage to apply (that is, only uses the key derivation as an oracle). $\overline{\texttt{I}}_{RF}$ stands for the interface displayed above, parameterised by an abstract type for derived keys (indexed by the template the key is derived from), and where the type of seed is made abstract. We do not provide, in this example, a function to generate seeds, but rather declare, in the interface, a constant seed.

Lemma 3 (Ideal Key Derivation: Typing Conditions)**.** *Let $\overline{\texttt{K}}$ be p.p.t. such that $\vdash \overline{\texttt{K}} \rightsquigarrow \overline{\texttt{I}}_{RF}$. For all p.p.t. A such that $\overline{\texttt{I}}_{RF} \vdash A : bool$, we have $|\Pr[\langle\rangle\ \overline{\texttt{K}} \cdot A\ \langle b\rangle] - \Pr[\langle\rangle\ RF \cdot A\ \langle b\rangle]| \leq \mathsf{Adv}^{\overline{\texttt{K}}}_{RF}(t, q_{KDF})$, for all b.*

Proof. A p.p.t. adversary that is well-typed against $\overline{\texttt{I}}_{RF}$ can only call the KDF function with the constant seed declared in the interface. We can therefore represent any such adversary as a p.p.t. adversary making queries to the random function with a single seed. □

Authenticated Encryption

We similarly define the cryptographic assumption on the authenticated encryption scheme, which provides oracles for key generation, encryption and decryption.

Let $AE\ (\overline{\texttt{E}})$ be the ideal authenticated encryption scheme that uses the concrete encryption scheme $\overline{\texttt{E}}$ to encrypt zeroes instead of the plaintext, maintains a table of ciphertext-plaintext pairs and decrypts ciphertexts by looking up the corresponding plaintext in the table, only successfully decrypting ciphertexts that appear in it. Encrypting zeroes ensures that the ciphertexts contain no information about the plaintext (this is the standard notion of CPA security), and only decrypting ciphertexts that were previously encrypted ensures the property known as ciphertext integrity (INT-CTXT). The key generation oracle in the ideal functionality uses $\overline{\texttt{P}}$ to generate a new key (to be used to encrypt zeroes), creates an empty ciphertext-plaintext table and returns a unique identifier for it.

Our Device uses three distinct instances of the authenticated encryption library, which are all concretely implemented in the same way, but whose intended usage and cryptographic properties differ. $\overline{\texttt{EtM}}_p$, is used to encrypt user-provided plaintexts using non-sensitive keys; $\overline{\texttt{EtM}}_s$ is used to encrypt user-provided plaintexts using sensitive keys; and $\overline{\texttt{EtM}}$, is used to protect objects for offloading in untrusted memory. Their distinguishing advantages from the ideal authenticated encryption schemes are, we recall, denoted $\mathsf{Adv}^{\overline{\texttt{EtM}}_p}_{AE\left(\overline{\texttt{EtM}}_s\right)}\left(t, q_{GEN_p}, q_{ENC_p}, q_{DEC_p}\right)$, $\mathsf{Adv}^{\overline{\texttt{EtM}}_s}_{AE\left(\overline{\texttt{EtM}}_s\right)}(t, q_{GEN_s}, q_{ENC_s}, q_{DEC_s})$, and $\mathsf{Adv}^{\overline{\texttt{EtM}}}_{AE\left(\overline{\texttt{EtM}}\right)}(t, q_{GEN}, q_{ENC}, q_{DEC})$, where q_{GEN_i} is the number of calls to the key generation oracle, q_{ENC_i} is the number of calls to the encryption oracle, and q_{DEC_i} is the number of calls to the decryption

oracle. Informally, the distinguishing advantages for $\overline{\mathtt{EtM}}_s(\mathsf{Adv}^{\overline{\mathtt{EtM}}_s}_{AE(\overline{\mathtt{EtM}}_s)}(t, q_{GEN_s}, q_{ENC_s}, q_{DEC_s}))$, and $\overline{\mathtt{EtM}}(\mathsf{Adv}^{\overline{\mathtt{EtM}}}_{AE(\overline{\mathtt{EtM}})}(t, q_{GEN}, q_{ENC}, q_{DEC}))$ are meant to be negligible.

We recall the typing conditions, on programs using an authenticated encryption scheme $\overline{\mathtt{E}}$, that need to be proved in order to substitute the ideal functionality $AE(\overline{\mathtt{E}})$ for the concrete scheme $\overline{\mathtt{E}}$. This result is proved as Theorem 12 by Fournet et al. [2011], and we reproduce it below, using our notation for ideal functionalities (we denote $AE(\overline{\mathtt{E}})$ the module composition $\overline{\mathtt{E}} \cdot \overline{\mathtt{F_E}}^{ae}$).

Secret Interface for $\overline{\mathtt{EtM}}_s$: $\overline{\mathtt{I}}_S$

type plainrepr = b:bytes { Length(b) = plainSize }
private type plain = plainrepr

function val Repr: plain → plainrepr
val plain: pr: plainrepr → p: plain { pr = Repr(p) }

We use the interface $\overline{\mathtt{I}}_S$ (where the S stands for "secrets") as secret interface (in place of their $\overline{\mathtt{I}}_{PLAIN}$). $\overline{\mathtt{I}}_E$ is simply the abstract interface declaring the key generation, encryption and decryption function, as well as an abstract type of keys.

Lemma 4 (Ideal Functionality for AE ([Fournet et al., 2011], Theorem12)). *If $\overline{\mathtt{E}}$ is an encryption scheme such that $\overline{\mathtt{I}}_S \vdash \overline{\mathtt{E}} \rightsquigarrow \overline{\mathtt{I}}_E$, and $\overline{\mathtt{P}}$ is a secret module such that $\vdash \overline{\mathtt{P}} \rightsquigarrow \overline{\mathtt{I}}_S$, for any p.p.t. A such that $\overline{\mathtt{I}}_S, \overline{\mathtt{I}}_E \vdash A$: bool, we have $|\Pr[\langle\rangle\ \overline{\mathtt{P}} \cdot \overline{\mathtt{E}} \cdot A\ \langle b\rangle] - \Pr[\langle\rangle\ \overline{\mathtt{P}} \cdot AE(\overline{\mathtt{E}}) \cdot A\ \langle b\rangle]| \leq \mathsf{Adv}^{\overline{\mathtt{E}}}_{AE(\overline{\mathtt{E}})}(t, q_{GEN}, q_{ENC}, q_{DEC})$ for all b.*

Secure Implementations of Primitives

In practice, we implement $\overline{\mathtt{EtM}}_i$ using Encrypt-then-MAC with AES-CBC and HMAC-SHA1; the key derivation function is implemented using the counter mode described in the NIST recommendation [Chen, 2009] and HMAC-SHA1 as core PRF.

AES-CBC is usually assumed to be a pseudo-random permutation, and HMAC-SHA1 is usually assumed to be unforgeable, therefore providing the desired IND-CPA and INT-CTXT security on the Encrypt-then-MAC construction [Bellare and Namprempre, 2000]. HMAC-SHA1 is also usually assumed to be a good pseudo-random function, justifying its use in the key derivation function.

5.1.4 Concrete Security for the Device

In the same way we expressed the cryptographic assumptions in terms of distinguishing advantage between a concrete functionality and an ideal functionality that is trivially secure, we aim to establish security properties of our Device implementation by giving the adversary oracle access to the Device commands, and letting him attempt to distinguish between the concrete system Device and

its idealization. We call $\mathcal{A}(t, (q_{c,l})_{c \in Cmd, l \in \{0,1\}})$ the set of all p.p.t. adversaries running in time t and making $q_{c,l}$ oracle calls to command c, on an object with sensitive attribute l.

Such adversaries can be written, with the same complexity, as C programs that use the interface displayed below. This interface gives consumer code full control over the I/O buffer and the scheduling of the Device commands. Arguments and results are passed by side-effect on the I/O buffer (an array IOB, of static length IOBLEN) before calling the chosen command.

External Interface for the Device

```
extern BYTE IOB[IOBLEN];

void Create(void);
void Import(void);
void Export(void);
void Clear(void);
void Load(void);
void Unload(void);
void Encrypt(void);
void Decrypt(void)
```

We indiscriminately write A for all implementations of an adversary that have the same complexity (for example, the adversary interacting with the F# code and the one interacting with the C code).

Modelling Security for the Device

Following the approach outlined in Chapter 4, we study four different implementations of the Device, which we introduce and name below.

The verified C code, when linked against a concrete cryptographic library, forms the concrete system implementation of the Device, which we denote $\texttt{Device}^c$. The same verified C code, when linked against an ideal cryptographic library forms an ideal system implementation, denoted $\texttt{Device}^i$. Similarly, we expect our reference implementations in F# to be executable when linked against concrete cryptographic libraries, forming a concrete reference implementation denoted $\overline{\texttt{Device}}^c$. The same reference code, linked against ideal cryptographic libraries, constitutes an ideal reference implementation denoted $\overline{\texttt{Device}}^i$.

Concrete Security The security of the Device is expressed using two games, one to express the Load Integrity property, and the other to express Security of Encryption under Sensitive Keys.

Integrity Game $\mathcal{I}(D)$. Given a program D that implements the Device commands, we call $\mathcal{I}(D)$ the game that gives the adversary oracle access to all commands, and where the adversary wins whenever a query to the Load oracle succeeds on a buffer whose *cipherSize* first bytes (where *cipherSize* is the size of ciphertexts, fixed by the implementation) were not previously returned by a successful

query to Unload.

Secrecy Game $\mathcal{S}(D)$. Given a program D that fully implements the Device commands, we call $Dev(D)$ the functionality obtained by filtering queries to D as follows:

- when receiving an Encrypt query with a handle pointing to a sensitive key, query D for an encryption of zero using the same handle; using the original plaintext and the used key's template, log the triple template-plaintext-ciphertext in a global table (if the query to D was successful); return D's reply to the adversary.
- when receiving a Decrypt query with a handle pointing to a sensitive key, locate in the table a triple whose template and ciphertext components match the query; if it exists, use the logged plaintext for the decryption, otherwise, return an error.

The filter also needs to maintain a partial map from handles to sensitive templates, which can be done easily by recording calls to Create, Load, Unload and Clear.

Let $\mathcal{S}(D)$ be the game that lets the adversary interact either with D, or with $Dev(D)$, depending on a uniformly sampled bit b, and where the adversary wins if she returns a b' such that $b' = b$.

Theorem 2 (Concrete Security)**.** *The probability that an adversary A in $\mathcal{A}(t, (q_{c,l})_{c\in Cmd, l\in\{0,1\}})$ wins either $\mathcal{I}(\texttt{Device}^c)$ or $\mathcal{S}(\texttt{Device}^c)$ is bounded by*

$$\begin{aligned}
&\mathsf{Adv}^{\overline{\texttt{Device}}^i}_{\texttt{Device}^c}\left(t, (q_{c,l})_{c\in Cmd, l\in\{0,1\}}\right) \leq \\
&\mathsf{Adv}^{\overline{\texttt{EtM}}}_{AE\left(\overline{\texttt{EtM}}\right)}\left(P(t), 1, q_{Unload}, q_{Load}\right) + \\
&\mathsf{Adv}^{\overline{\texttt{KDF}}}_{RF}\left(P'(t), q_{Create,s} + q_{Create,p}\right) + \\
&\mathsf{Adv}^{\overline{\texttt{EtM}_s}}_{AE\left(\overline{\texttt{EtM}_s}\right)}\left(P''(t), q_{Create,s}, q_{Encrypt,s}, q_{Decrypt,s}\right),
\end{aligned}$$

where P, P' and P'' are polynomials that take into account the (linear) running time of the command code leading up to the cryptographic oracle queries, as well as the running time (linear in each q_o) of all queries to oracles not involved in the current cryptographic game (for example, calls to the RF do not appear in P', but appear in P and P'').

Perfect Security

In Section 5.2.3, we prove the following theorem, stating that the ideal system Device is perfectly secure.

Theorem 3 (Perfect System Security)**.** *Any adversary A in* $\mathcal{A}(t, (q_{c,l})_{c\in Cmd, l\in\{0,1\}})$ *has a zero probability of winning either of* $\mathcal{I}(\texttt{Device}^i)$ *or* $\mathcal{S}(\texttt{Device}^i)$*.*

This ensures that the ideal system implementation provides perfect Load Integrity (since no adversary can win $\mathcal{I}(\texttt{Device}^i)$), and perfect Security of Encryption under Sensitive Keys (since no adversary can win $\mathcal{S}(\texttt{Device}^i)$).

Security Reduction

In Section 5.2.3, we prove that the concrete system Device can only be distinguished from the ideal system Device inasmuch as the underlying cryptographic primitives can themselves be distinguished. This is formally expressed by explicitly bounding the probability of a p.p.t. adversary distinguishing the two systems.

Theorem 4 (Reduction of System Security to Primitive Security)**.**

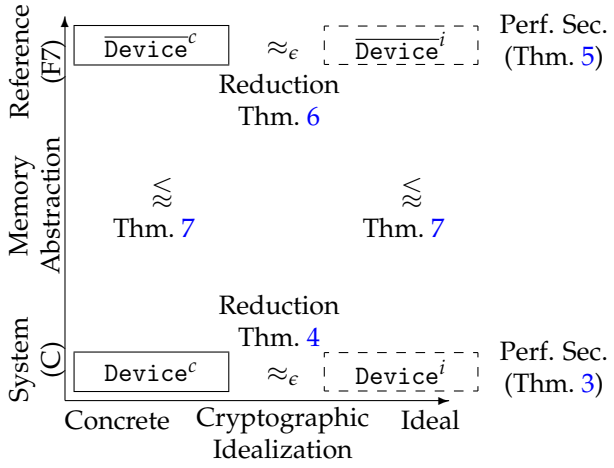
The probability that a p.p.t. adversary in $\mathcal{A}(t, (q_{c,l})_{c\in Cmd, l\in\{0,1\}})$ *distinguishes between the concrete and ideal system Devices is bounded by*

$$\begin{aligned}
&\mathsf{Adv}^{\texttt{Device}^c}_{\texttt{Device}^i}\left(t, (q_{c,l})_{c\in Cmd, l\in\{0,1\}}\right) \leq \\
&\mathsf{Adv}^{\overline{\texttt{EtM}}}_{AE\left(\overline{\texttt{EtM}}\right)}\left(P(t), 1, q_{Unload}, q_{Load}\right) + \\
&\mathsf{Adv}^{\overline{\texttt{KDF}}}_{RF}\left(P'(t), q_{Create,s} + q_{Create,p}\right) + \\
&\mathsf{Adv}^{\overline{\texttt{EtM}_s}}_{AE\left(\overline{\texttt{EtM}_s}\right)}\left(P''(t), q_{Create,s}, q_{Encrypt,s}, q_{Decrypt,s}\right).
\end{aligned}$$

We can then conclude the proof of Theorem 2 by transitivity and the triangle inequality.

5.2 Proving Computational Security Properties of the Device

We summarise all theorems involved in the proof of Theorem 2 in the figure below, placing them in relation to the two abstraction axes introduced in Section 4.2.



Theorem 7 is proved in Section 5.2.2. Theorems 5 and 3, asserting perfect security of the ideal reference and system implementations (respectively), are proved in Section 5.2.3. Theorems 6 and 4, reducing the security of the concrete reference and system implementations to their ideal counterparts, are proved in Section 5.2.3.

5.2.1 Security of the Reference Implementation

To describe the proof, we need to present the structure of the reference implementation, composed of the following modules.

- $\overline{\texttt{T}}$ (for $\overline{\texttt{Template}}$), defines constants of the Device and operations on templates (reading attributes, checking validity).
- $\overline{\texttt{Secrets}}$, declares an abstract type for the user-provided plaintexts for commands Encrypt and Decrypt.
- $\overline{\texttt{EtM}_s}$, instantiates authenticated encryption using sensitive keys.
- $\overline{\texttt{EtM}_p}$, instantiates authenticated encryption using non-sensitive keys.
- $\overline{\texttt{KDF}}$ declares the type of keys as the disjoint union of sensitive and public keys; declares that abstract type of seeds and the key derivation function.
- $\overline{\texttt{Store}}$, declares the type of entries and the type of the store, along with the formatting and parsing functions for entries; the type of entries is declared as abstract to prove secrecy of the

following encryption module.

- $\overline{\texttt{EtM}}$, instantiates authenticated encryption using the root storage key to encrypt store entries.
- $\overline{\texttt{Device}}$, declares the types of the input/output buffer and of the internal state, and implements the commands. We divide it in two modules, a concretely-typed module $\overline{\texttt{API}}$, that parses the input buffer, validates inputs and formats the output buffer, and a module $\overline{\texttt{Device}-}$ whose functions may operate on abstract or refined types, and implements the core of the commands.

In the proof and in the ideal Device functionality, we make use of a variant of the *RF* functionality that uses two disjoint tables, respectively indexed by sensitive and non-sensitive templates. This functionality is perfectly indistinguishable from *RF* (since the two domains considered form a partition of *RF*'s domain). We call it RF_{2s}.

Perfect Security

We use the methodology described by Fournet et al. [2011] to prove perfect security of ideal reference code, written in F7. Perfect secrecy is proved by parametricity and perfect integrity is proved by refinement type-safety.

We prove the following theorem on the ideal F7 Device, mirroring the desired perfect security theorem for the C implementation.

Theorem 5 (Perfect Reference Security)**.** *For all adversary A in $\mathcal{A}(t, (q_{c,l})_{c\in Cmd, l\in\{0,1\}})$, the probability of A winning either $\mathcal{I}(\overline{\texttt{Device}}^i)$ or $\mathcal{S}(\overline{\texttt{Device}}^i)$ is null.*

Proof. We prove that the probability of winning each of the game is 0 for the considered adversaries.

1. For integrity, we log an event Unloaded(entry,blob) whenever a store entry is Unloaded as blob. We then typecheck with F7 that, if the ideal reference implementation of the Load command succeeds on a blob, then the event must have been logged previously for some object. In our proof, we use the ideal authenticated encryption's lookup table to register the event.
2. For secrecy, we need to prove that, for all adversary A in $\mathcal{A}(t, (q_{c,l})_{c\in Cmd, l\in\{0,1\}})$, we have $\overline{\texttt{Device}}^i \cdot A \approx \overline{\texttt{Device}}^i \cdot F \cdot A$, where F is the filter informally described above.

   The key observation is that the table kept by F in the right program contains the same entries as the table kept by the ideal authenticated encryption module in the left program. (We need only consider sensitive Encrypt queries to prove this.) Indeed, when a sensitive Encrypt query is

received on the left, the internal ideal AE samples an IV uniformly at random, encrypts zeros, and logs the plaintext-ciphertext pair in the log corresponding to the key used before returning the ciphertext. On the right, such a query is replaced with a query to encrypt zeros by the filter, which, given the same random tape, yields the same ciphertext and the same entry in the table.

When a sensitive Decrypt query is received on the left, the internal ideal AE's table allocated for the object's template is used to lookup the plaintext; on the right, the filter's table is used. Since they are always equal, Decrypt queries always return the same result.

All other queries are untouched (apart from monitoring code that does not modify the Device's behaviour) by the filter.

This argument can be made more formal using parametricity at the cost of some code refactoring.

□

Security Reduction

The security reduction for the concrete reference implementation follows the methodology presented by Fournet et al. [2011], replacing the concrete cryptographic modules with their ideal counterparts one by one.

Theorem 6 (Reference Security Reduction)**.**

$$\begin{aligned}
&\mathsf{Adv}^{\overline{\texttt{Device}}^{c}}_{\overline{\texttt{Device}}^{c}}\left(t, \left(q_{c,l}\right)_{c\in Cmd, l\in\{0,1\}}\right) \leq \\
&\mathsf{Adv}^{\overline{\texttt{EtM}}}_{AE\left(\overline{\texttt{EtM}}\right)}\left(P(t), 1, q_{Unload}, q_{Load}\right) + \\
&\mathsf{Adv}^{\overline{\texttt{KDF}}}_{RF}\left(P'(t), q_{Create,s} + q_{Create,p}\right) + \\
&\mathsf{Adv}^{\overline{\texttt{EtM}_\texttt{s}}}_{AE\left(\overline{\texttt{EtM}_\texttt{s}}\right)}\left(P''(t), q_{Create,s}, q_{Encrypt,s}, q_{Decrypt,s}\right).
\end{aligned}$$

Proof. The proof comprises the following sequence of game-hopping steps, true for all p.p.t. adversary A.

$$\begin{aligned}
&\overline{\texttt{T}}\cdot\overline{\texttt{Secrets}}\cdot \quad \overline{\texttt{EtM}_s}\cdot\overline{\texttt{EtM}_p}\cdot\overline{\texttt{KDF}}\cdot\overline{\texttt{Store}}\cdot \quad \overline{\texttt{EtM}}\cdot\overline{\texttt{Device}}\cdot A \\
\approx_{\epsilon_1}\ &\overline{\texttt{T}}\cdot\overline{\texttt{Secrets}}\cdot \quad \overline{\texttt{EtM}_s}\cdot\overline{\texttt{EtM}_p}\cdot\underline{\overline{\texttt{KDF}}}\cdot\overline{\texttt{Store}}\cdot AE(\overline{\texttt{EtM}})\cdot\overline{\texttt{Device}}\cdot A && (5.1)\\
\approx_{\epsilon_2}\ &\overline{\texttt{T}}\cdot\overline{\texttt{Secrets}}\cdot \quad \overline{\texttt{EtM}_s}\cdot\overline{\texttt{EtM}_p}\cdot\underline{RF}\cdot\overline{\texttt{Store}}\cdot AE(\overline{\texttt{EtM}})\cdot\overline{\texttt{Device}}\cdot A && (5.2)\\
\approx\ &\overline{\texttt{T}}\cdot\overline{\texttt{Secrets}}\cdot \quad \underline{\overline{\texttt{EtM}_s}}\cdot\overline{\texttt{EtM}_p}\cdot RF_{2s}\cdot\overline{\texttt{Store}}\cdot AE(\overline{\texttt{EtM}})\cdot\overline{\texttt{Device}}\cdot A && (5.3)\\
\approx_{\epsilon_4}\ &\overline{\texttt{T}}\cdot\overline{\texttt{Secrets}}\cdot AE(\overline{\texttt{EtM}_s})\cdot\overline{\texttt{EtM}_p}\cdot RF_{2s}\cdot\overline{\texttt{Store}}\cdot AE(\overline{\texttt{EtM}})\cdot\overline{\texttt{Device}}\cdot A && (5.4)
\end{aligned}$$

where $\approx_{\epsilon_i}$ stands for indistinguishability with the advantages detailed below.

Step (5.1) follows with advantage $\mathsf{Adv}^{\overline{\mathtt{EtM}}}_{AE(\overline{\mathtt{EtM}})}(P(t), 1, q_{Unload}, q_{Load})$. Under the assumption that the formatting and parsing functions for store entries are such that applying the parsing to a formatted entry returns the original entry, we can establish by automated type-checking that, for $\overline{\mathtt{P}'} = (\overline{\mathtt{T}} \cdot \overline{\mathtt{Secrets}} \cdot \overline{\mathtt{EtM}_s} \cdot \overline{\mathtt{EtM}_p} \cdot \overline{\mathtt{KDF}} \cdot \overline{\mathtt{Store}})$ and $\overline{\mathtt{Device}}^i \cdot A = \overline{\mathtt{P}'} \cdot \overline{\mathtt{EtM}} \cdot \overline{\mathtt{Device}} \cdot A$, we have a secret interface $\overline{\mathtt{I}}_{Store}$ such that $\vdash \overline{\mathtt{P}'} \rightsquigarrow \overline{\mathtt{I}}_{Store}$, and $\overline{\mathtt{I}}_{Store}, \overline{\mathtt{I}}_{EtM} \vdash \overline{\mathtt{Device}} \rightsquigarrow \overline{\mathtt{I}}_{Device}$. Thus we have $\overline{\mathtt{I}}_{Store}, \overline{\mathtt{I}}_{EtM} \vdash \overline{\mathtt{Device}} \cdot A : bool$ and we conclude using Lemma 4. The formatting hypothesis is discharged later, as Lemma 6.

Step (5.2) follows with advantage $\mathsf{Adv}^{\overline{\mathtt{KDF}}}_{RF}(P'(t), q_{Create,1})$ by type-checking and Lemma 3.

Step (5.3) follows from rewriting the lazy random function, first so that it uses separate tables for public and sensitive keys (since their templates are disjoint), then so that, for sensitive keys, RF_{2s} calls $\overline{\mathtt{EtM}_s}.\mathtt{GEN}$ (defined as random sampling) instead of direct random sampling. This is just program rewriting.

Step (5.4) follows with advantage $\mathsf{Adv}^{\overline{\mathtt{EtM}_s}}_{AE(\overline{\mathtt{EtM}_s})}(P''(t), q_{Create,1}, q_{Encrypt,1}, q_{Decrypt,1})$ since we can establish by automated type-checking that we have, for $\overline{\mathtt{P}'} = (\overline{\mathtt{T}} \cdot \overline{\mathtt{Secrets}})$, $\overline{\mathtt{Device}}^i \cdot A = \overline{\mathtt{P}'} \cdot \overline{\mathtt{EtM}_s} \cdot (\overline{\mathtt{EtM}_p} \cdot \overline{\mathtt{KDF}} \cdot \overline{\mathtt{Store}} \cdot AE(\overline{\mathtt{EtM}}) \cdot \overline{\mathtt{Device}} \cdot A)$, and we have a secret interface $\overline{\mathtt{I}}_S$ such that $\vdash \overline{\mathtt{P}'} \rightsquigarrow \overline{\mathtt{I}}_S$, and $\overline{\mathtt{I}}_S, \overline{\mathtt{I}}_{EtM_s} \vdash (\overline{\mathtt{EtM}_p} \cdot \overline{\mathtt{KDF}} \cdot \overline{\mathtt{Store}} \cdot AE(\overline{\mathtt{EtM}}) \cdot \overline{\mathtt{Device}}) \rightsquigarrow \overline{\mathtt{I}}_{Device}$. Thus we have $\overline{\mathtt{Secrets}}, \overline{\mathtt{I}}_{\overline{\mathtt{EtM}_s}} \vdash (\overline{\mathtt{EtM}_p} \cdot \overline{\mathtt{KDF}} \cdot \overline{\mathtt{Store}} \cdot AE(\overline{\mathtt{EtM}}) \cdot \overline{\mathtt{Device}} \cdot A) : bool$ and we conclude using Lemma 4.

□

This concludes the security proof for the reference Device implementation. We now prove, by simulation, that the same security properties hold on the system implementation.

5.2.2 Observations and Simulation

We start by defining the initial and final observation functions for the cryptographic primitives, and the corresponding simulation contracts. We then define the observation function for the Device commands, expressing the simulation theorem, and briefly discussing the VCC proof. Finally, we show how VCC can be used to prove other theorems, in particular proving that the contracts on Device commands do not prevent adversaries from calling them, and the formatting condition assumed in the proof for the reference Device (Lemma 6).

To express simulation as a contract as presented in Chapter 4, we need to provide an abstract interface for the reference cryptographic primitives, which we use to express the assumption that the

system cryptography is correctly implemented, and make the reference implementation of the Device available as logical specifications in VCC. We write the abstract interface in such a way that it is met by both the ideal and concrete cryptographic implementations, so that a single run of VCC proves both required program simulations modularly.

The VCC specification language is the simplest notation to express mathematical functions on C program states, and we therefore use it to describe the observation functions, whose definition is central to the final simulation statement and the security proof.

For the argument to make sense, we also display some of the type and function declarations for the reference implementation in VCC. As discussed at the end of Chapter 4, reference implementations in VCC need to be written in store-passing style and are therefore quite verbose.

Abstract Reference Interface for $\overline{\texttt{EtM}}_S$ (excerpt)

```
// Abstract Types
_(type \EtMsKey)
_(abstract \bool isEtMsKey(\Bytes t,\EtMsKey)
  _(requires isTemplate(t)))

// Abstract State
_(type \EtMsState)
_(abstract \bool isEtMsState(\EtMsState))

_(abstract \bool leq(\EtMsState S1,\EtMsState S2)
  _(requires isEtMsState(S1))
  _(requires isEtMsState(S2)))

// Encryption
_(record \EtMsENCIn {
    \EtMsState S; \Bytes t;
    \EtMsKey k; \plain p; })
_(def \bool isEtMsENCIn(\EtMsENCIn i)
  { return isEtMsState(i.S) &&
           isTemplate(i.t) && sensitive(i.t) &&
           isEtMsKey(i.t,i.k) && isPlain(i.p); })

_(record \EtMsENCOut { \EtMsState S; \Bytes c; })
_(def \bool isEtMsENCOut(\EtMsENCOut o)
  { return isEtMsState(o.S) && isEtMsCipher(o.c); })

_(abstract \EtMsENCOut EtMsENC(\EtMsENCIn i)
  _(requires isEtMsENCIn(i))
  _(ensures isEtMsENCOut(\result))
  _(ensures leq(i.S,\result.S)))

// Decryption
_(record \EtMsDECIn {
    \EtMsState S; \Bytes t;
    \EtMsKey k; \Bytes c; })
_(def \bool isEtMsDECIn(\EtMsDECIn i)
  { return isEtMsState(i.S) &&
           isTemplate(i.t) && sensitive(i.t) &&
           isEtMsKey(i.t,i.k) && isEtMsCipher(i.c); })

_(typedef \plainOption \EtMsDECOut)
_(def \bool isEtMsDECOut(\EtMsDECOut o)
  { return isPlainOption(o); })

_(abstract \EtMsDECOut EtMsDEC(\EtMsDECIn i)
  _(requires isEtMsDECIn(i))
  _(ensures isEtMsDECOut(\result)))
```

Authenticated Encryption under Sensitive and Non-Sensitive Keys

We discuss the simulation assumption for the authenticated encryption operations used in the Encrypt and Decrypt commands.

Abstract Interface On the right, we display the abstract interface, in VCC's specification language, for the module performing authenticated encryption using sensitive keys. The $\overline{\texttt{EtM}}_p$ module, performing authenticated encryption using non-sensitive keys, has the same interface, where all instances of the word "secret" are replaced with the word "public', every instance of the predicate sensitive is replaced with its negation, and the abstract type \plain of plaintexts to

encrypt under sensitive keys is replaced with a concrete type for plaintexts to be encrypted under non-sensitive keys. They also share a concrete implementation, but we never idealise encryption using non-sensitive keys.

First, we declare an abstract type for keys, along with its refinement predicate. The refinement predicate takes as argument a template that constrains the length of the key, intuitively meaning that the key is a valid sensitive key for a particular template (in our case, this means that the key has the correct length and sensitive attribute).

Second, we declare an abstract type for this module's cryptographic state. In the concrete implementation, this state will simply contain the random tape, but in the ideal implementation, it will also contain the cryptographic log, encoded using maps. As always, we also define a refinement predicate; in addition, we also equip states with an order relation leq, used to allow VCC to reason about chronology of events.

We can then declare the encryption and decryption functions. In the absence of unnamed tuple types in VCC, we need to declare record types (which are simply named tuples, and use structure-like syntax) whenever a function is meant to return several values. To simplify later discussions, we also make sure that the inputs to all cryptographic functions are passed in as a tuple, allowing the result of observation functions to be passed directly into the encryption functions, as expressed in the abstract contracts from Chapter 4.

This gives rise to record types \EtMsENCIn and \EtMsENCOut for encryption (and similarly for decryption), which we also equip with refinement predicates. Those types are fixed and do not vary between ideal and concrete implementation and can be fully defined instead of being kept abstract.

To express random sampling as a mathematical function, we need it to modify the random tape (for example, by keeping track of the read index on the tape). Since encryption samples the IV at random, we pass the state in, and get a modified copy of the state out (in store-passing style). (In the ideal implementation, the state is also updated by logging the plaintext-ciphertext pair for later decryption.)

Decryption, simply reads the state and does not need to pass it back out. However, it may fail and therefore returns an option type, either reporting a decryption error, or returning a plaintext.

The entire module is parameterised by an abstract type \plain, declared in the secret interface.

Observation Functions and Simulation Contracts For the C code to remain realistic, we use a single function for encryption using sensitive and non-sensitive keys, equipped with a double simulation contract. The system function takes several concrete arguments: pointer and length for a buffer containing the key, pointer and length for a buffer containing the plaintext, and a pointer to a buffer large enough to contain the resulting ciphertext. In addition, we pass the necessary cryptographic states to the function as ghost parameters, along with the key's template, used to decide which simulation contract applies. The cryptographic states that are modified by the reference functions are also passed back out as out parameters (introduced using the **out** keyword). Additional pre and postconditions refining the cryptographic states are added to ensure that the observation functions are well-defined, and we define them as expected.

Simulation Contract for Encryption

```
_(def \EtMsENCIn EtMsENCIn(
    \EtMpState,\EtMsState S, \Bytes t,
    BYTE* k, UINT8 k_len,
    BYTE* p, UINT8 p_len,
    BYTE*)
  _(requires isEtMsState(S))
  _(requires isTemplate(t) && sensitive(t))
  _(requires k_len == keylength(t))
  _(requires p_len == plainSize())
  _(ensures isEtMsENCIn(\result))
  { return (\EtMsENCIn) {
      .S = S, .t = t,
      .k = EtMsKey(t,from_array(k,k_len)),
      .p = plain(from_array(p,p_len)) }; })

_(def \EtMsENCOut EtMsENCOut(
    \EtMpState,\EtMsState S,\Bytes t,
    BYTE*, UINT8, BYTE*, UINT8,
    BYTE* c)
  _(requires isEtMsState(S))
  _(requires isTemplate(t) && sensitive(t))
  _(ensures isEtMsENCOut(\result))
  { return (\EtMsENCOut) {
      .S = S,
      .c = from_array(c,cipherSize()) }; })

void _(assume_correct) ENC(
  _(ghost \EtMpState pS) _(ghost \EtMsState sS)
  _(ghost \Bytes t) BYTE* k,UINT8 k_len,
  BYTE* p,UINT8 p_len,
  BYTE* buffer
  _(out \EtMpState pSOut) _(out \EtMsState sSOut))
_(decreases 0) // Termination Measure
// Reference Contract
_(requires isEtMpState(pS))
_(requires isEtMsState(sS))
_(ensures isEtMpState(pSOut) && leq(pS,pSOut))
_(ensures isEtMsState(sSOut) && leq(sS,sSOut))
_(requires isTemplate(t))
// Writes Clause
_(writes \array_range(buffer,(size_t) cipherSize()))
// Simulation when sensitive
_(ensures sensitive(t) ⇒
  EtMsENCOut(pSOut,sSOut,t,k,k_len,p,p_len,buffer) ==
  EtMsENC(\old(EtMsENCIn(pS,sS,t,k,k_len,p,p_len,buffer))))
_(ensures sensitive(t) ⇒pSOut == pS)
// Simulation when non−sensitive
_(ensures !sensitive(t) ⇒
  EtMpENCOut(pSOut,sSOut,t,k,k_len,p,p_len,buffer) ==
  EtMpENC(\old(EtMpENCIn(pS,sS,t,k,k_len,p,p_len,buffer))))
_(ensures !sensitive(t) ⇒sSOut == sS);
```

For example, the specification code displayed right shows the initial and final (EtMsENCIn/Out(), respectively) observation functions for encryption using sensitive keys.

The initial observation ignores the non-sensitive key-related cryptographic state and the output buffer, and simply preserves the sensitive key-related cryptographic state (including its random tape) and the template, and observes the contents of the key and plaintext buffers, abstracting away from their exact location in memory, before injecting them into the corresponding abstract types.

The final observation similarly ignores the non-sensitive key-related cryptographic state, but this time also ignores the key and plaintext buffers (for example, allowing them to be overwritten with

intermediate results, or performing encryption in place (by passing the same buffer as plaintext and ciphertext buffer)), and simply observes the contents of the ciphertext buffer.

In the previous chapter, observation functions were defined on configurations, but we here define them only on the program's arguments. However, since their body reads from memory, VCC internally passes in an additional argument containing the entire program memory, effectively giving our observation functions the expected type.

With the observation functions defined, we can now write the simulation contract on the system encryption function ENC.

The first line of the contract provides VCC with a termination measure to prove that the function terminates. In this case, the trivial termination measure suffices (it is sufficient for most non-recursive functions).

We then refine the cryptographic states, and the witness template. These preconditions are the same as those on the reference function, and therefore do not restrict the simulation result in any way (as we further argue in Lemma 5).

The following line lets VCC know that the function, during its execution, may write into the ciphertext buffer (excluding all distinct memory locations).

Finally, we state the two simulation postconditions, the first expressing the simulation when the key is sensitive, and the second expressing it when the key is public, exactly as defined in Chapter 4. These state that observing the final configuration (using the out parameters for cryptographic state) is equal to running the reference implementation on the observation of the initial configuration (obtained by evaluating the observation function in the initial memory using the **old** keyword).

The observation functions for the module performing encryption using non-sensitive keys, as well as the simulation contracts for decryption are similar and not shown here.

We call $\mathcal{H}_{\overline{\texttt{EtM}}_{Enc}}$ the simulation hypothesis embodied in the contracts for the system encryption and decryption functions.

Key Derivation

Abstract Reference Interface We use an abstract type \key for Device keys, indexed by the key's template, and use the template to distinguish between sensitive and non-sensitive keys.

Again, we display the abstract reference interface on the right, briefly discussed below.

The abstract type for cryptographic states, \seedState is empty in the concrete implementation (no random tape is used), and contains the lookup tables and event log in the ideal implementation. Again, we declare a refinement predicate and an order relation (both trivially true in the concrete implementation).

As mentioned above, we declare an abstract type for keys, whose refinement predicate is indexed by a template, determining not only the length of the key (as was the case of the encryption module), but also its sensitivity. In addition, the cryptographic state appears in the index, since the ideal refinement on keys expresses the fact that they have been derived on the Device or imported onto it, both of which are logged as events in the state.

Abstract Reference Interface for $\overline{\texttt{KDF}}$ (excerpt)

```
// Cryptographic State
_(type \SeedState)
_(abstract \bool isSeedState(\SeedState S))
_(abstract \bool leq(\SeedState S1,\SeedState S2)
  _(requires isSeedState(S1))
  _(requires isSeedState(S2)))

// Type for Keys
_(type key)
_(abstract \bool isKey(\seedState S,\Bytes t,\key)
  _(requires isSeedState(S))
  _(requires isTemplate(t)))

_(record \KDFIn {
    \EtMsState EtMs_S; \EtMpState EtMp_S;
    \SeedState S; \Bytes t; \seed s; })
_(def \bool isKDFIn(\KDFIn i)
  { return isEtMsState(i.EtMs_S) && isEtMpState(i.EtMp_S) &&
           isSeedState(i.S) && isTemplate(i.t) && isSeed(i.s); })

_(record \KDFOut {
    \EtMsState EtMs_S; \EtMpState EtMp_S;
    \SeedState S; \key k; })
_(def \bool isKDFOut(\Bytes t,\KDFOut o)
  _(requires isTemplate(t))
  { return isEtMsState(o.EtMs_S) && isEtMpState(o.EtMp_S) &&
           isSeedState(o.S) && isKey(o.S,t,o.k); })


_(abstract \KDFOut KDF_r(\KDFIn i)
  _(requires isKDFIn(i))
  _(requires i.s == getPrimarySeed())
  _(ensures isKDFOut(i.t,\result))
  _(ensures leq(i.EtMs_S,\result.EtMs_S))
  _(ensures leq(i.EtMp_S,\result.EtMp_S))
  _(ensures leq(i.S,\result.S)))
```

Finally, we declare the input and output types for the KDF, as well as the KDF itself. Since the ideal KDF, a random function, samples the key uniformly at random to generate it, the underlying encryption libraries' cryptographic states are passed in and out so the correct random tape can be read according to the type of key meant to be derived.

Observations All simulation contracts have the same form, and we therefore only display, from now on, the observation functions.

The concrete KDF takes as arguments a pointer and length for a buffer containing the template, a pointer and length for a buffer containing the seed, a pointer to a buffer large enough to contain a key, and a pointer to an integer whose final value is the length of the derived key, and is unspecified in the

initial configuration (for example, it may contain the actual size of the allocated key buffer to perform additional memory-safety checks). In addition, we pass in the cryptographic states and an abstract template as ghost parameters, and pass the cryptographic states back out as out parameters. The abstract template contains the initial contents of the template buffer; although not strictly necessary, it makes it easier to express the final observation in cases where the template buffer is overwritten.

Both initial and final observation functions preserve the cryptographic states (including the random tapes). In addition, the initial observation function observes the contents of the seed buffer and injects it into the abstract type of seeds, and observes the abstract template, which is separately restricted to be equal to the contents of the template buffer. In the final observation, the buffer containing the freshly derived key is observed and its contents injected into the type of keys, with the correct state and template indexes.

Observations for Key Derivation

```
_(def \KDFIn KDFIn(
    \EtMsState EtMs_S, \EtMpState EtMp_S,
    \SeedState S, \Bytes t, BYTE*, UINT8,
    BYTE* s, UINT8 s_len,
    BYTE* buf, UINT8* buf_len)
  _(requires isEtMsState(EtMs_S) && isEtMpState(EtMp_S))
  _(requires isSeedState(S) && s_len == seedSize())
  _(requires isTemplate(t))
  { return (\KDFIn) {
      .EtMs_S = EtMs_S, .EtMp_S = EtMp_S,
      .S = S, .t = t, .s = seed(from_array(s,s_len)) }; })

_(def \KDFOut KDFOut(
    \EtMsState EtMs_S, \EtMpState EtMp_S,
    \SeedState S, \Bytes t,
    BYTE* tb, UINT8 t_len,
    BYTE* s, UINT8 s_len,
    BYTE* buf, UINT8* buf_len)
  _(requires isEtMsState(EtMs_S) && isEtMpState(EtMp_S))
  _(requires isSeedState(S))
  _(requires isTemplate(t))
  _(requires *buf_len == keylength(t))
  _(ensures isKDFOut(t,\result))
  { return (\KDFOut) {
      .EtMs_S = EtMs_S, .EtMp_S = EtMp_S,
      .S = S, .k = key(S,t,from_array(buf,*buf_len)) }; })
```

We call $\mathcal{H}_{KDF}$ the simulation hypothesis that the system KDF simulates the reference KDF for the observation functions described above.

Authenticated Encryption under the Storage Key

Finally, we discuss the simulation hypothesis for the authenticated encryption primitive as used in the Load and Unload commands. This primitive is in fact concretely the same as the one used in Encrypt and Decrypt, composed with formatting and parsing functions that pad template-key pairs to an implementation-constant size.

Abstract Reference Interface The secret interface for this module is in fact the module that defines the internal key store and store entries. Store entries are defined as an abstract type \entry equipped with a constructing function makeEntry() that takes a seed state, a template and a key and produces an entry (the seed state is only used to refine the type of stored keys).

For brevity, we only show the interface for the reference decryption function, displayed on the right. Cryptographic states for this module are refined by cryptographic states for the $\overline{\texttt{KDF}}$ module, since plaintexts (that appear in the ideal lookup table) are entries, that are refined by the seed state.

Abstract Reference Interface for $\overline{\texttt{EtM}}$ (excerpt)

```
// Keys
_(type \EtMKey)
_(abstract \bool isEtMKey(\EtMKey))

// Cryptographic State
_(type \EtMState)
_(abstract \bool isEtMState(\SeedState sS,\EtMState)
  _(requires isSeedState(sS)))
_(abstract \bool leq(\SeedState sS,\EtMState S1,\EtMState S2)
  _(requires isSeedState(sS))
  _(requires isEtMState(sS,S1))
  _(requires isEtMState(sS,S2)))

// Decryption
_(record \EtMDECIn {
    \SeedState seedS; \EtMState S;
    \Bytes t; \EtMKey k; \Bytes c; })
_(def \bool isEtMDECIn(\EtMDECIn i)
  { return isSeedState(i.seedS) && isEtMState(i.seedS,i.S) &&
           isEtMKey(i.k) && isEtMCipher(i.c); })

_(typedef \entryOption \EtMDECOut)
_(def \bool isEtMDECOut(\SeedState S,\EtMDECOut o)
  _(requires isSeedState(S))
  { return isEntryOption(S,o); })

_(abstract \EtMDECOut EtM_DEC(\EtMDECIn i)
  _(requires isEtMDECIn(i))
  _(ensures isEtMDECOut(i.seedS,\result)))
```

Apart from this complication, the interface is the same as for the other instances of the authenticated encryption library ($\overline{\texttt{EtM}}_S$ and $\overline{\texttt{EtM}}_p$), replacing the plaintexts with entries. In particular, the functions take in all the necessary cryptographic states and pass back out those it modifies (in this case, the decryption function does not modify any state, but its ideal implementation reads from the log to perform decryption).

Observations The system function takes as arguments a pointer and length for a buffer whose contents are used as key, a pointer and length for a buffer containing the ciphertext, and a pointer to one of the Device's internal memory slots. We also add ghost arguments to pass in a seed state and a storage state.

Observations for $\overline{\texttt{EtM}}$ (excerpt)

```
_(def \EtMDECIn EtMDECIn(
    \SeedState Seed_S,\EtMState S,
    BYTE* k,UINT8 kl,
    BYTE* c,UINT8 cl,
    slot_t*)
  _(requires isSeedState(Seed_S))
  _(requires isEtMState(Seed_S,S))
  _(requires kl == EtMKeySize())
  _(requires cl == EtMCipherSize())
  _(ensures isEtMDECIn(\result))
  { return (\EtMDECIn) {
      .seedS = Seed_S, .S = S,
      .k = EtMKey(from_array(k,kl)),
      .c = from_array(c,cl) }; })

_(def \EtMDECOut EtMDECOut(
    \SeedState S,\EtMState,
    BYTE*,UINT8,
    BYTE*,UINT8,
    slot_t* p,int res)
  _(requires isSeedState(S))
  _(requires res == 0 ⇒\inv(p) && p->tmpl[0] < 2)
  _(ensures isEtMDECOut(S,\result))
  { if (res == 0)
      return SoEntry(makeEntry(S,p->t,key(S,p->t,p->kr)));
    return NoEntry(); })
```

Internally, after decrypting the ciphertext, the resulting plaintext is parsed into a template-key pair and stored into the indicated memory slot. The plaintexts are meant to be padded with zeros to a fixed size. However, since we use authenticated encryption, we do not need to check the padding when parsing the decrypted plaintext into a structured object.

The initial observation preserves the cryptographic states and observes the key and ciphertext buffers.

The final observation preserves the cryptographic states, and observes the return value. In case of success (when the return value is 0), the memory slot is observed, and its components used to construct an abstract entry. In case of failure (when the return value is non-null), the observation returns the reference failure value.

We call $\mathcal{H}_{EtM}$ the simulation hypothesis that the system implementation of the Encode-then-Encrypt described here simulates the reference implementation for these observation function.

Device Commands

So far, all the simulation contracts we have seen are hypotheses in our simulation theorem, and can be interpreted as universal quantifications over all reference implementations that typecheck against the reference interface and all system implementations that verify against the simulation contracts for the chosen reference implementation. However, in the case of the Device commands, we wish to prove that the system implementation simulates the particular reference implementation that we proved secure in Section 5.2.1.

Reference Implementations As discussed in Chapter 4, we currently assume that the typechecked F7 code and our VCC specification describe the same mathematical functions. As an example, we display the Create command in both notations below. The Create_cmd_r function is typed with strong refinements and performs all operations on the internal state, whereas Create_r simply parses the input buffer, calls Create_cmd_r and formats its result, implementing the separation between the $\overline{\texttt{Device}-}$ and $\overline{\texttt{API}}$ modules. We make use of several distinct error codes for various error sources, since real-world systems often need to offer detailed feedback to consumer code, or to the user, when errors occur. Since error messages have been known to be a source of security flaws, we see them as a necessary part of the system description when performing security proofs.

Create_cmd_r takes a template and a Device state which includes, implicitly in F7 and explicitly in VCC, abstract cryptographic states for all the modules. After finding the first empty slot (and returning a "store full" error (code 255) in case there are no empty slots), the KDF is called with the primary seed and the template, and the resulting object is placed in the computed slot. The side effects on the cryptographic states (reads from random tape and event logging) are modelled as

updates to the cryptographic states in VCC. The function, when successful, returns the index to the freshly filled slot, along with the new state.

F7 Notation for the Create Command

```
type Create_RET_r =
  | CreateError of rcode
  | CreateSuccess of handle * state

let Create_cmd_r S t =
    match firstFree S with
    | None → CreateError 255
    | Some h →
        let s0 = getPrimarySeed()
        let k = KDF t s0
        let store' =
          update S.store h (Some (makeEntry t k))
        let S' =
        { stKey = S.stKey; store = store' }
        CreateSuccess(h,S')

let Create_r (S,b) =
  match unmarshal_Create_IN_r b with
  | Some t →
    match Create_cmd_r S t with
    | CreateError rc →
      let buf = bcons (int2byte rc) (sub b 1 (ioblen −
          1))
      S,buf
    | CreateSuccess C →
      let (S',h) = C
      let buf = bcons (0uy) (bcons (int2byte h) (sub b
          2 (ioblen − 2)))
      S',buf
  | None →
    let buf = bcons (254uy) (sub b 1 (ioblen − 1))
    S,buf
```

VCC Notation for the Create Command

```
_(datatype Create_RET_r {
    case CreateError(RCODE)
    case CreateSuccess(Handle_r res,state S) })


_(def Create_RET_r Create_cmd_r(state S,Create_IN_r t)
  _(requires isState(S))
  _(requires isCreate_IN_r(t))
  {
    switch (firstFree(S.Seed_S,S.store)) {
      case NoNat():
        return CreateError(RC_STORE_FULL);
      case SoNat(i):
        \seed s0 = getPrimarySeed();
        \KDFIn in = (\KDFIn) { .t = t, .s = s0,
          .EtMs_S = S.EtMs_S, .EtMp_S = S.EtMp_S, .S = S.Seed_S };
        \KDFOut s = KDF_r(in);
        \store newStore =
            update(S.store,i,SoEntry(makeEntry(s.S,t,s.k)));
        state S0 = S / { .Seed_S = s.S, .EtMs_S = s.EtMs_S,
                         .EtMp_S = s.EtMp_S, .store = newStore };
        return CreateSuccess((Handle_r) i, S0); } })


_(def \configuration Create_r(\configuration in)
  _(requires isConfiguration(in))
  _(ensures isConfiguration(\result))
  { switch (unmarshal_Create_IN_r(in.b)) {
      case SoCArgs(t):
        Create_RET_r res = Create_cmd_r(in.S,t);
        switch (res) {
          case CreateError(rc):
            \Bytes oBuf = Bcons(rc,Substring(in.b,1,IOBLEN − 1));
            return in / { .b = oBuf };
          case CreateSuccess(res,S):
            \Bytes oBuf = Bcons(RC_SUCCESS,Bcons(res,Substring(
                in.b,2,IOBLEN − 2)));
            return (\configuration) { .S = S, .b = oBuf }; }
      case NoCArgs():
        \Bytes oBuf = Bcons(RC_UNMARSHAL,Substring(in.b,1,
            IOBLEN − 1));
        return (\configuration) { .S = in.S, .b = oBuf }; } })
```

Create_r takes and returns a configuration, composed of a Device state and an I/O buffer. It first parses the input buffer (in a command-specific way), returning an "unmarshalling" error (code 254) in case parsing fails (in this case, if the buffer does not start with a valid template). The Create_cmd_r function is then called on the Device state and the parsed template, and its return value analysed. In case of error, the error code is copied into the first byte of the output buffer and control is returned to the adversary (no updates are made to the internal state). When successful, the success code 0 is written to the first byte of the output buffer, and the returned handle is written to the second byte of

the output buffer. In this implementation, we choose to leave the irrelevant parts of the I/O buffer untouched. Other implementations may set it to 0, or fill it with randomness; however, whatever behaviour is chosen needs to be fully specified in the reference implementation, as the I/O buffer's contents are directly given to the adversary.

Apart from the differences due to the store-passing transformation (for example, makeEntry reads the Seed module's state directly in F7, whereas it is taken as an argument in VCC notation), some syntactic differences exist between the two languages. For example, in F7, bytes and integers are two different types and casts need to be made explicit, whereas VCC handles them automatically (and automatically discharges the proof obligations to check that any natural number used as a byte is less than 256, for example). In F7, n-ary functions are curried, and we make use of the polymorphic option type, whereas these are not supported in VCC syntax.

Despite those differences, we believe that these two programs are in fact two notations for the same mathematical functions from configurations to configurations.

Observation Functions and Simulation Contracts Our system implementations take no arguments and return no results, instead reading their input and writing their output by side-effect on global variables. Therefore, system configurations for the Device are simply C program states that contain a global buffer BYTE IOB[IOBLEN], and a global state **struct** store_s store, where the structure type **struct** store_s is displayed right.

In addition to the system state (a primary seed, a storage key and an array of memory slots), we give the system state a ghost field containing a copy of the reference state.

The observation function on configurations works by retrieving the cryptographic states from the stored copy of the reference state, building a reference store by observing the internal memory slot by slot, and observing the system seed and storage key. Additional invariants are used to improve the performance of the verification, but are not displayed or discussed here for clarity, and do not change the final verification result, since the observation function used is the same as the one displayed.

Memory slots are considered empty if their template field's first byte is neither 0 nor 1 (the only valid first bytes for templates), and otherwise get observed as expected.

All top-level simulation contracts are similar. For all Device commands, we prove (trivial) termi-

nation and simulation (for the appropriate reference command). In addition, we prove sufficient memory-safety to sequentially compose calls to the commands, which includes proving that all invariants on the global store hold at function boundaries.

We only displayed a sample of the observation functions. As they are central to the statement of the simulation theorem, and later central to the security proof, we display them fully in Appendix B.

We have now reduced the problem of proving simulation to a simple functional verification problem, and can state the corresponding theorem.

Observation and Simulation for $\overline{\texttt{Device}}$

```
struct store_s {
  BYTE pSeed[SEEDSIZE];
  BYTE k[STORAGEKEYSIZE];
  slot_t store[STORE_SIZE];

  // Abstract State
  _(ghost state S) } store;

_(def \entryOption entry(\SeedState S, slot_t e)
  _(requires isSeedState(S))
{ if (e.tmpl[0] == 2) return NoEntry();
  else return SoEntry(makeEntry(S,e.t,key(S,e.t,e.kr))); })

_(def \configuration configuration()
  _(requires \wrapped(&store))
  _(requires \mutable_array(IOB,IOBLEN))
  _(ensures isConfiguration(\result))
{ state S = store.S / {
    .store = λ\natural i;
      (i < storeSize()) ?
        entry(store.S.Seed_S,store.store[i]) :
        NoEntry(),
    .stKey = EtMKey(from_array(store.k,EtMKeySize())),
    .pSeed = seed(from_array(store.seed,seedSize())) };
  return (\configuration) {
    .buffer = from_array(IOB,IOBLEN), .S = S }; })

void Create(void)
  _(decreases 0) // Termination
  _(updates &store) // State Invariants
  _(writes \array_range(IOB,IOBLEN)) // Writes clause
  _(ensures \mutable_array(IOB,IOBLEN))// Memory−safety
  _(ensures configuration() == Create_r(\old(configuration())));
```

Theorems

We prove several theorems relating the C implementation of the Device and its reference implementation in F#.

Device Simulation Our first theorem states that whatever reference implementation $\overline{\texttt{Crypto}}$ of the cryptographic primitives that provides the desired functionalities, and whatever system implementation `Crypto` of these same cryptographic functionalities that is such that $\texttt{Crypto} \lessapprox \overline{\texttt{Crypto}}$, where the observation functions for each function are those listed above, the reference program $\overline{\texttt{Crypto}} \cdot \overline{\texttt{Device}}$, is simulated by the system program $\texttt{Crypto} \cdot \texttt{Device}$ with the configuration() observation function.

Theorem 7 (Device Simulation)**.** *The following property holds, where simulation on modules is short for simulation with the same observation functions for all functions in the module.*

$$\{\mathcal{H}_{EtM_{Enc}}, \mathcal{H}_{KDF}, \mathcal{H}_{EtM}\} \vdash \texttt{Device}\ {}_{\texttt{configuration}()}\lessapprox_{\texttt{configuration}()}\ \overline{\texttt{Device}}$$

Proof. Using VCC, we prove that each of the system Device commands fulfills its contract under the

assumption that the system cryptographic library simulates the reference cryptographic library with the appropriate observation functions, as described above. We conclude by Lemma 2, which, we recall, states that the simulation contract implies semantic simulation. □

VCC is only semi-automated, and many intermediate lemmas, invariants and definitions may be needed to complete the simulation proof. However, given the hypotheses and conclusions in contract form, this constitutes a problem in pure verification, and does not involve any cryptographic expertise.

Robust Device Simulation The theorem above may still place strong preconditions on the commands, and it may seem unreasonable to assume all adversaries to respect them. However, we show that all adversaries with oracle access to the Device commands can in fact be written as *verified* C programs, therefore showing that the simulation result holds even in the presence of an adversary (and in particular a p.p.t. adversary).

Lemma 5 (Robust Device Simulation)**.** *Any p.p.t. adversary A in $\mathcal{A}(t, (q_{c,l})_{c\in Cmd, l\in\{0,1\}})$ can be written as a C program $[\![A]\!]$ such that $\left\{\texttt{Device}\ configuration() \overset{\leq}{\approx} configuration()\ \overline{\texttt{Device}}\right\} \vdash \{True\}\ [\![A]\!]\ \{True\}$. In addition, for all intermediate configuration $\hat{C}$ obtained during the execution of the reference program $\overline{\texttt{Device}} \cdot A$ in an initial configuration where all slots are empty and the storage key and seed are initialised at random, there exists a system configuration C such that $\hat{C} = configuration(C)$.*

Proof. The first result is proved by verifying in VCC a program that loops indefinitely, filling the input buffer and running a command non-deterministically (see Appendix C). All p.p.t. adversaries with oracle access to the commands are clearly refinements of such a program (replacing non-deterministic choice with some local probabilistic computation).

The second result is proved by induction on adversary traces, which are finite in length. By verification, we prove that any such initial reference state is the result of observing an initial system state with the same key and seed. (It should be noted that non-empty initial states can easily be dealt with by adding code to the initialisation function we verify to prove this base case.) The adversary, between oracle queries, can only affect the reference configuration by writing into the I/O buffer, on which the observation function is onto (any reference buffer are the observation of a system buffer containing it). During oracle queries, since we know by induction hypothesis that the initial reference configuration is the result of an observation, the simulation contract applies, and the final system configuration (which exists by termination) is observed as the final reference configuration. □

Entry Formatting Lemma Finally, we prove the formatting lemma on entries, which we recall is used the security reduction proof for the reference Device (Section 5.2.3).

Lemma 6 (Plaintext Side-Condition)**.** *Parsing a formatted store entry yields the original store entry.*

Proof. By verifying in VCC the reference implementation of the displayed function, where format and parse are the **def** encodings of the F# formatting and parsing functions for unloading and loading objects. The SoEntry option constructor is required since parsing is partial. (The seedState needs to be passed in since ideal refinements on entries make use of the Created and Imported predicates.)

Parsing and Formatting of Store Entries

```
_(abstract \bool lemma_parse_format_ID()
  _(ensures ∀\seedState S; \entry e;
        isSeedState(S) ⇒isEntry(S,e) ⇒
        parse(S,format(S,e)) == SoEntry(e))
  _(returns \true))
```

□

5.2.3 Security of the System Implementation

In this section, we prove Theorems 3 and 4, and conclude the proof of security for the system implementation of the Device.

Perfect Security for Ideal C Code

We first prove Theorem 3, which we recall states that the ideal system implementation of the Device is perfectly indistinguishable from the idealized reference functionality $\overline{\texttt{Device}}^i$. This entails, by transitivity and Theorem 5, that no adversary can break the security properties (load integrity and security of encryption under sensitive keys) of the ideal system implementation. The proof works by replacing adversary calls to the ideal C implementation of the Device with queries to the ideal reference oracles, using the verified simulation relation to show that there are no observable differences between the two programs.

Proof of Theorem 3. Given an adversary A in $\mathcal{A}(t, (q_{c,l})_{c \in Cmd, l \in \{0,1\}})$ running with $\overline{\texttt{Device}}^i$. Each query that A makes to an oracle in some configuration $\hat{C}$ can be replaced with a query to the same oracle in $\alpha \circ \texttt{Device}^i$ in *any* configuration C such that $\hat{C} = \alpha(C)$ without changing the full program's behaviour. Since this is true for each random tape, and our observation function preserves the random tapes, this is also true for the probabilistic program. □

System Security Reduction

We prove Theorem 4, which we recall claims that the advantage of an adversary trying to distinguish the ideal and concrete system Device implementations is bounded by a polynomial function of the distinguishing advantages for the cryptographic primitives. We prove that the same bound applies between the system implementations as between the reference implementations (as established in Theorem 6). The adversary-interface-specific proof is similar to that of Theorem 3.

Proof of Theorem 4. Given an adversary A in $\mathcal{A}(t, (q_{c,l})_{c \in Cmd, l \in \{0,1\}})$ running with $\overline{\texttt{Device}}^c$. Each query that A makes to an oracle in some configuration $\hat{C}$ can be replaced with a query to the same oracle in $\alpha \circ \texttt{Device}^c$ in *any* configuration C such that $\hat{C} = \alpha(C)$ without changing the full program's behaviour. Since this is true for each random tape, and our observation function preserves the random tapes, this is also true for the probabilistic program. □

Concrete Security for the System Device

Finally, we can prove the concrete security theorem for the system Device.

Proof of Theorem 2. Theorem 3 states that no adversary in $\mathcal{A}(t, (q_{c,l})_{c \in Cmd, l \in \{0,1\}})$ can ever win either $\mathcal{I}(\texttt{Device}^i)$ or $\mathcal{S}(\texttt{Device}^i)$, that is, no adversary can break either Load Integrity or Security of Encryption under Sensitive Keys of the *ideal* system Device.

Theorem 4 states the following bound on the distinguishing advantage for the ideal and concrete system Devices

$$\begin{aligned}
&\mathsf{Adv}^{\texttt{Device}^c}_{\texttt{Device}^i}\left(t, (q_{c,l})_{c \in Cmd, l \in \{0,1\}}\right) \leq \\
&\mathsf{Adv}^{\overline{\texttt{EtM}}}_{AE\left(\overline{\texttt{EtM}}\right)}\left(P(t), 1, q_{Unload}, q_{Load}\right) + \\
&\mathsf{Adv}^{\overline{\texttt{KDF}}}_{RF}\left(P'(t), q_{Create,s} + q_{Create,p}\right) + \\
&\mathsf{Adv}^{\overline{\texttt{EtM}_s}}_{AE\left(\overline{\texttt{EtM}_s}\right)}\left(P''(t), q_{Create,s}, q_{Encrypt,s}, q_{Decrypt,s}\right).
\end{aligned}$$

By definition of the distinguishing advantage, and of the winning conditions for $\mathcal{I}(\texttt{Device}^c)$ and $\mathcal{S}(\texttt{Device}^c)$, we conclude that the probability of any adversary winning either one of these games is bounded by the distinguishing advantage $\mathsf{Adv}^{\texttt{Device}^c}_{\texttt{Device}^i}\left(t, (q_{c,l})_{c \in Cmd, l \in \{0,1\}}\right)$. □

Informally, we can conclude that, if the distinguishing advantages on the secure primitives are negligible, then the probability of the adversary winning either of the security games is also negligible.

5.3 Discussion

The table below summarises the verification effort involved for our exemplary Device, in terms of lines of code (LoC, counts only executable code), lines of specification (LoS, counts observation functions and reference implementations in VCC notation), and lines of annotations (LoA, denotes contracts, invariants and intermediate assertions inserted for completeness or performance reasons), as well as verification times (shown in seconds, and obtained on a mid-end laptop).

File	LoC	LoS/LoA	Time (s)
template.fs7	-	20	-
template.fs	10	-	5
template.vcc.h	-	30	1
secrets.fs7	-	10	-
secrets.fs	5	-	4
secrets.vcc.h	-	30	-
secrets.vcc.c	-	10	1
EtMs.fs7	-	25	-
EtMs.vcc.h	-	120	1
EtMs.ideal.c	-	100	5
EtMs.concrete.c	-	70	5
EtMp.fs7	-	20	-
EtMp.vcc.h	-	120	1
EtMp.ideal.c	-	100	5
EtMp.concrete.c	-	70	5
EtM_Enc.h	4	100	-

File	LoC	LoS/LoA	Time (s)
seed.fs7	-	42	-
seed.fs	90	-	6
seed.vcc.h	-	200	1
seed.h	4	100	-
store.fs7	-	50	-
store.fs	55	-	5
store.vcc.h	-	200	1
marshal.h	10	200	-
marshal.c	120	20	10
EtM.fs7	-	25	-
EtM.vcc.h	-	150	1
EtM.h	4	20	-
device.fs7	-	20	-
device.fs	100	-	6
api.fs	200	-	6
device.vcc.c	200	-	10
device.c	500	30	**900**

In the table, files with extensions .concrete.c and .ideal.c are not in fact part of the proof, but serve as sanity checks for the abstract reference interfaces. By providing ideal and concrete implementations to the abstract reference interfaces, we prove that the simulation theorem (which we recall is quantified over all implementations of the reference interface) is not vacuously true, and we can additionally argue that it captures the intended ideal and concrete interfaces, although this latter argument is necessarily informal.

As expected, most of the difficulty resides in proving the simulation for the Device commands (simulation for the marshalling function is almost immediate). However, most commands in fact take less than 20 seconds to verify, except for the Create_cmd (500s) and Import_cmd (200s) commands. The slowdown appears to be due to imprecisions in the memory safety and coupling invariants, which we have yet to identify. We believe that the verification times for these two functions can also be

brought down to under 20 seconds with enough engineering work.

Without counting the cryptographic library's implementations (not shown in the table), the C implementation of the Device consists of 700 lines of C code, that are fully verified. The F# implementation is composed of 500 lines of F# code, which are denoted, along with 200 lines of type annotations, into 1100 lines of VCC specifications, counting only the abstract interfaces for cryptographic primitives.

This explosion in the size of the specification, and the current reliance on several tools, highlight some of the shortcomings of our methodology. We now discuss them, and discuss potential solutions to the underlying problems.

5.3.1 On the Size of Specifications and Systems

In many cases, we expect the C code to be the most precise specification of the system to prove secure. This is for example, the case for the new TPM standard, whose normative implementation, written in C, counts 30,000 lines of code [Tru, 2013, Part3]. Manually extracting an F# implementation on which to perform the security proof for such a large system is not reasonably feasible. On the other hand, automated extraction techniques such as those recently presented by Aizatulin et al. [2011b, 2012] can automatically produce almost full specifications of a system from the C code.

We do not believe that automated proofs of security on these extracted models is a viable option on its own, since minor changes in the C code would require the security proof to be restarted from scratch. However, such models could be structured and translated into a VCC specification useful for proving simulation, and transferring any security results obtained on the model to the C implementation. Changes to the C code that do not change the specification could then be dealt with without changing the security proof, and changes to the C code that do affect the specification could be dealt with at a much smaller cost, by incrementally modifying the specification instead of running the extraction process anew.

Finally, if going from C to F# is difficult, in particular when attempting to structure the F# code to facilitate a security proof by typing, we believe that it would be feasible to generate C code from a structured F# specification (as was done by Mukhamedov et al. [2013]), along with a VCC specification that could be used to prove simulation. Optimisations and mitigation measures could then be implemented manually on the C code, incrementally updating the simulation proof without having

to redo the security proof itself.

5.3.2 Security for Under-Specified Systems

In the discussed example, we specify the Device fully. However, we only really need to specify enough of the system that all its unspecified behaviours can be hidden by the observation functions.

For example, if the Device used randomly sampled integers as handles, internally keeping a table mapping handles to memory addresses, the actual memory allocation mechanism could be kept completely unspecified, as its behaviour would be hidden from any adversary with oracle access to the Device commands (we would still have to prove its memory safety properties, and perhaps more precise functional properties to enable the simulation proof for the rest of the system). However, this would require the use of finer-grained observation functions and, in the absence of a simple criterion on observation functions that guarantees the transfer of security properties, would make the proof of Theorems 3 and 4 more complex.

Note that our observation functions already hide unspecified behaviour of this sort, since it does not let the adversary observe the memory addresses of any of the variables manipulated, be they global or local. In this sense, the use of array indices as addresses already provides some abstraction from low-level details. However, the resulting observation functions are still very simple, and intuitively correspond to an adversary with oracle access to the system, in particular if the Device is implemented in hardware rather than software.

Chapter 6

Conclusion

We discuss the contributions outlined in Chapter 1 in the light of their presentation in Chapters 2 to 5.

In Chapter 2, we described a protocol description language that we use to generate Coq theories in which symbolic security properties of the described protocol can be used in a general, modular way. We have shown that the security proof can in fact be organised such that protocol-specific proof obligations are simple, and some of the security proof can be discharged independently of the protocol.

In Chapter 3, we show how such protocol descriptions can be used, within a general-purpose C verifier, to soundly prove security properties of C programs in the symbolic model of cryptography. In particular, we develop a framework that reduces security properties of C programs to a set of verification tasks that we argue can be performed without security expertise.

Further, in Chapter 4, we define a notion of program simulation, that we show is provable using general-purpose verifiers, and that we use, in Chapter 5, to prove security properties of a C program in the computational model of security. Although some security-specific arguments remain to be made once the program simulation property is proved by verification, the process of proving simulation does not require any security expertise.

These results are a step towards using general-purpose C verification to prove security properties of C programs in the symbolic and computational models of security. However, much work is still needed in order for them to be applicable to real-world software.

6.1 From Secure Model to Usable Specification

First and foremost, the transition from T3 and F# code to pure VCC specifications was, during the course of this research, done by hand and trusted. In both cases, the syntactic similarities between the source and target language, along with some simple proof obligations that we discharge automatically on the VCC side serve to reinforce the trust. However, the process of turning a formal model into a VCC specification that can be used to verify C code is expensive and is an obvious impediment to scalability.

It would be easy to automatically generate VCC definitions and axioms from a T3 model in a provable way (given formal semantics for the VCC specification language). However, the transition from F# to VCC is a bit more involved, as it also includes a state-passing transform to make all accesses to the cryptographic state explicit, and also an effects-tracking type system to simplify the resulting specification by ensuring that the states are only mentioned where necessary and therefore avoid unnecessary proof obligations.

The technique demonstrated in Chapter 5 is geared towards proving both concrete and ideal simulation, but such a strong notion of simulation is not in fact necessary: it would be sufficient to prove that the concrete system code simulates the concrete reference code to obtain a concrete bound on the probability that an adversary breaks the security of the concrete system code. In fact, this simplified scenario would also yield much lighter VCC specifications, since the cryptographic state is only used for sampling randomness in the concrete code, and very few operations rely on it. Another advantage of this simpler notion of simulation is that only programs need to be translated from F# to VCC, and not abstract typed interfaces.

6.2 Threat Models, Shims and Observation Functions

In both the symbolic and computational model, most of the difficulty was in giving a reasonable formal definition of what constitutes an adversary. In the symbolic model, we write an application-specific adversary shim to check that standard symbolic adversaries are captured by our model, whereas in the computational model, we use carefully crafted observation functions to represent adversaries that have limited access to the system. In addition, both shims and observation functions

are specific both to the protocol or device under study, and to the threat model considered. For example, the configuration observation function considered in Chapter 5 would not be sufficient to guarantee concrete security in a model where the adversary runs in the same address space as the device code.

Defining good shims and observations for various threat models will need to take into account real attack scenarios. On a device, for example, timing and power channels should be modelled. For a cryptographic library, meant to be run in the same address space as its adversary, not much security could be obtained without formalizing properly, as started by Abadi and Plotkin [2012] and Abadi and Planul [2013], widely deployed operating system countermeasures such as Address Space Layout Randomization or stack canaries.

6.3 On Usability and Applicability

In this first step towards using general-purpose verification tools to prove cryptographic security properties, we did not measure the amount of effort needed to obtain the desired formal security guarantees. In particular, we only looked at small examples, for which we developed the specification, security proof, C code and functional verification in parallel. We expect that, assuming a translation from provable specifications to VCC specifications as discussed in Section 6.1, it should also be possible to use our techniques starting from a high-level specification (for example in T3, F# or EasyCrypt's pWhile language) and then generating both the VCC specification, the C code and an initial simulation proof that can be updated whenever changes are made to the C code.

As shown in the Device example (Chapter 5), inductive specifications are used as invariants to prove simulation on loops, and it is therefore the case that a single C function simulates the composition of several specification functions. It would be difficult, and perhaps impossible without considering intermediate levels of specifications, to prove simulation in the inverse situation, where a single specification function is simulated by the composition of several C functions. In the computational model, we therefore recommend that the specification be written using small functions, to make the C code less rigid and give more liberty to the engineer in the final phases of development.

If the technique we developed for the symbolic model has been shown to be applicable with only minor changes to both protocols ([Dupressoir et al., 2011]) and stateful devices ([Polikarpova and

Moskal, 2012]), this is much less clear for our simulation-based techniques. In particular, we do not yet know how to prove simulation in the presence of dynamic memory allocation (that introduces observable non-determinism unless it can be proven that allocation never fails), and even less so in the presence of concurrency, as is often the case with protocols.

Both of these issues, along with the threat model issues mentioned above are interesting research challenges.

Appendix A

Full Coq Framework for Symbolic Cryptography

From file `common.v`, we display the Coq definitions for the full framework.

Full Coq Framework

```
Module Type ProtocolDefs.
  Parameter nonce_usage: Type.
  Parameters hmac_usage senc_usage: Type.
  Parameters sign_usage enc_usage: Type.
  Parameter pEvent: Type.
End ProtocolDefs.



Module Defs (PD: ProtocolDefs).
  Include PD.

  Inductive usage: Type :=
    | AdversaryGuess
    | Nonce (nu: nonce_usage)
    | HmacKey (hu: hmac_usage)
    | SEncKey (eu: senc_usage)
    | SignKey (su: sign_usage)
    | VerfKey (su: sign_usage)
    | EncKey (eu: enc_usage)
    | DecKey (eu: enc_usage).

  Inductive event: Type :=
    | New (t: term) (u: usage)
    | AsymPair (pk: term) (sk: term)
    | ProtEvent (pe: pEvent).

  Definition log: Type := set event.
  Definition Logged (e: event) (L: log): Prop := set_In e L.
  Definition LoggedP (e: pEvent) (L: log): Prop := Logged (
        ProtEvent e) L.
  Definition leq_log (L L': log): Prop := ∀ e, Logged e L → Logged
        e L'.

  Definition Stable (P: log → Prop) := ∀ L L',
    leq_log L L' → P L →
    P L'.
  Definition WF_log (L: log): Prop :=
    (∀  t u u',
       Logged (New t u) L →
       Logged (New t u') L → u = u') ∧
    (∀  pk sk,
       Logged (AsymPair pk sk) L →
       ((∃  su,
          Logged (New pk (VerfKey su)) L ∧
          Logged (New sk (SignKey su)) L) ∨
        (∃  eu,
          Logged (New pk (EncKey eu)) L ∧
          Logged (New sk (DecKey eu)) L))).
End Defs.



Module Type ProtocolInvariants (PD: ProtocolDefs).
  Include Defs PD.

  Parameter LogInvariant: log → Prop.

  (* Nonce Predicate *)
  Parameter nonceComp: term → log → Prop.
  Parameter nonceComp_Stable: ∀ t, Stable (nonceComp t).

  (* HMAC Predicates *)
  Parameter hmacComp: term → log → Prop.
  Parameter hmacComp_Stable: ∀ t, Stable (hmacComp t).

  Parameter canHmac: term → term → log → Prop.
  Parameter canHmac_Stable: ∀ k p, Stable (canHmac k p).

  (* SEnc Predicates *)
  Parameter sencComp: term → log → Prop.
  Parameter sencComp_Stable: ∀ t, Stable (sencComp t).

  Parameter canSEnc: term → term → log → Prop.
  Parameter canSEnc_Stable: ∀ k p, Stable (canSEnc k p).

  (* Signature Predicates *)
```

```
  Parameter sigComp: term → log → Prop.
  Parameter sigComp_Stable: ∀ t, Stable (sigComp t).

  Parameter canSign: term → term → log → Prop.
  Parameter canSign_Stable: ∀ k p, Stable (canSign k p).

  (* Enc Predicates *)
  Parameter encComp: term → log → Prop.
  Parameter encComp_Stable: ∀ t, Stable (encComp t).

  Parameter canEnc: term → term → log → Prop.
  Parameter canEnc_Stable: ∀ k p, Stable (canEnc k p).
End ProtocolInvariants.


Module CryptographicInvariants (PD: ProtocolDefs) (PI:
      ProtocolInvariants PD).
  Include PI.

  Definition GoodLog (L: log): Prop :=
    WF_log L ∧ LogInvariant L.


  Inductive level := Low | High.


  Inductive Level: level → term → log → Prop :=

    (* AdversaryGuesses are always Low *)
    | Level_AdversaryGuess: ∀ l bs L,
      Logged (New (Literal bs) AdversaryGuess) L →
      Level l (Literal bs) L

    (* Nonces are Low when nonceComp holds *)
    | Level_Nonce: ∀ l bs L nu,
      Logged (New (Literal bs) (Nonce nu)) L →
      (l = Low → nonceComp (Literal bs) L) →
      Level l (Literal bs) L
    (* HmacKeys are Low when hmacComp holds *)
    | Level_HmacKey: ∀ l bs L hu,
      Logged (New (Literal bs) (HmacKey hu)) L →
      (l = Low → hmacComp (Literal bs) L) →
      Level l (Literal bs) L
    (* SEncKeys are Low when sencComp holds *)
    | Level_SEncKey: ∀ l bs L su,
      Logged (New (Literal bs) (SEncKey su)) L →
      (l = Low → sencComp (Literal bs) L) →
      Level l (Literal bs) L
    (* SigKeys are Low when signComp holds *)
    | Level_SigKey: ∀ l bs L su,
      Logged (New (Literal bs) (SignKey su)) L →
      (l = Low → sigComp (Literal bs) L) →
      Level l (Literal bs) L
    (* VerfKeys are always Low *)
    | Level_VerKey: ∀ l bs L su,
      Logged (New (Literal bs) (VerfKey su)) L →
      Level l (Literal bs) L
    (* EncKeys are always Low *)
    | Level_EncKey: ∀ l bs L eu,
      Logged (New (Literal bs) (EncKey eu)) L →
      Level l (Literal bs) L
    (* DecKeys are Low when encComp holds *)
    | Level_DecKey: ∀ l bs L eu,
      Logged (New (Literal bs) (DecKey eu)) L →
      (l = Low → encComp (Literal bs) L) →
      Level l (Literal bs) L

    (* Pairs are as Low as their components *)
    | Level_Pair: ∀ l t1 t2 L,
      Level l t1 L →
      Level l t2 L →
      Level l (Pair t1 t2) L

    (* Honest Hmacs are as Low as their payload *)
    | Level_Hmac: ∀ l k m L,
      canHmac k m L →
      Level l m L →
      Level l (HMac k m) L
    (* Dishonest Hmacs are Low *)
    | Level_Hmac_Low: ∀ l k m L,
      Level Low k L →
      Level Low m L →
      Level l (HMac k m) L

    (* Honest SEncs are Low *)
    | Level_SEnc: ∀ l l' k p L,
      canSEnc k p L →
      Level l' p L →
      Level l (SEnc k p) L
    (* Dishonest SEncs are Low *)
    | Level_SEnc_Low: ∀ l k p L,
      Level Low k L →
      Level Low p L →
      Level l (SEnc k p) L

    (* Honest Sigs are as Low as their payload *)
    | Level_Sig: ∀ l k m L,
      canSign k m L →
      Level l m L →
      Level l (Sign k m) L
    (* Dishonest Sigs are Low *)
    | Level_Sig_Low: ∀ l k m L,
      Level Low k L →
      Level Low m L →
      Level l (Sign k m) L
    (* Honest Encryptions are Low *)
    | Level_Enc: ∀ l k p L,
      canEnc k p L →
      Level High p L →
      Level l (Enc k p) L
    (* Dishonest Encryptions are Low *)
    | Level_Enc_Low: ∀ l k p L,
      Level Low k L →
      Level Low p L →
      Level l (Enc k p) L.


Theorem Low_High: ∀ t L,
  Level Low t L → Level High t L.


Theorem Level_Stable: ∀ l t L L',
  leq_log L L' → Level l t L →
  Level l t L'.


Theorem AbsurdDistinctUsages: ∀ P L t u u',
  GoodLog L →
  u <> u' →
  Logged (New t u) L →
  Logged (New t u') L →
  P.


Theorem LowNonce_Inversion: ∀ L n nu,
  GoodLog L →
  Logged (New (Literal n) (Nonce nu)) L →
  Level Low (Literal n) L →
  nonceComp (Literal n) L.


Theorem LowHmacKeyLiteral_Inversion: ∀ L k hu,
  GoodLog L →
  Logged (New (Literal k) (HmacKey hu)) L →
  Level Low (Literal k) L →
  hmacComp (Literal k) L.
```

Theorem HMac_Inversion: ∀ L l k p,
Level l (HMac k p) L →
canHmac k p L ∨ Level Low k L.

Theorem SEnc_Inversion: ∀ L l k p,
Level l (SEnc k p) L →
canSEnc k p L ∨ Level Low k L.

Theorem Sign_Inversion: ∀ L l k p,
Level l (Sign k p) L →
canSign k p L ∨ Level Low k L.

Theorem Enc_Inversion: ∀ L l k p,
Level l (Enc k p) L →
(canEnc k p L ∧ Level High p L) ∨ Level Low k L.

End CryptographicInvariants.

Appendix B

Observation Functions and Simulation Contracts - Unabridged

This appendix lists the full observation functions for all four cryptographic libraries, and the simulation contracts for the three corresponding system primitives. The aim is to make precise the three simulation hypotheses from Theorem 7.

We do not list the simulation contracts for the Device commands, that are exactly identical except for the reference function they call. We recall that the observation function for configurations was shown in Section 5.2.2.

B.1 Encryption of User-Provided Plaintexts

We display the observation functions for modules $\overline{\texttt{EtM}}_s$ and $\overline{\texttt{EtM}}_p$ (none are needed for the key generation oracle, used only in the ideal random function), and the simulation contracts for the common system encryption function. Together, they form the hypothesis we called $\mathcal{H}_{\overline{\texttt{EtM}}_{Enc}}$.

Observation Functions for Authenticated Encryption using Sensitive Keys ($\overline{\texttt{EtM}}_S$)

```
// Encryption
_(def \EtMsENCIn EtMsENCIn(\EtMpState,\EtMsState S,\Bytes t,BYTE* k,UINT8 k_len,BYTE* p,UINT8 p_len,BYTE*)
  _(requires isEtMsState(S))
  _(requires isTemplate(t) && sensitive(t))
  _(requires k_len == keylength(t))
  _(requires p_len == plainSize())
  _(ensures isEtMsENCIn(\result))
  { return (\EtMsENCIn) {
      .S = S, .t = t,
      .k = EtMsKey(t,from_array(k,k_len)),
      .p = plain(from_array(p,p_len)) }; })
```

```
_(def \EtMsENCOut EtMsENCOut(\EtMpState,\EtMsState S,\Bytes t,BYTE*,UINT8,BYTE*,UINT8,BYTE* c)
  _(requires isEtMsState(S))
  _(requires isTemplate(t) && sensitive(t))
  _(ensures isEtMsENCOut(\result))
  { return (\EtMsENCOut) { .S = S, .c = from_array(c,cipherSize()) }; })

// Decryption
_(def \EtMsDECIn EtMsDECIn(\EtMpState,\EtMsState S,\Bytes t,BYTE* k,UINT8 k_len,BYTE* c,UINT8 c_len,BYTE*)
  _(requires isEtMsState(S))
  _(requires isTemplate(t) && sensitive(t))
  _(requires k_len == keylength(t))
  _(requires c_len == cipherSize())
  _(ensures isEtMsDECIn(\result))
  { return (\EtMsDECIn) {
      .S = S, .t = t,
      .k = EtMsKey(t,from_array(k,k_len)),
      .c = from_array(c,c_len) }; })

_(def \EtMsDECOut EtMsDECOut(\EtMpState,\EtMsState S,\Bytes t,BYTE*,UINT8,BYTE*,UINT8,BYTE* p,UINT8 res)
  _(requires isEtMsState(S))
  _(requires isTemplate(t) && sensitive(t))
  _(ensures isEtMsDECOut(\result))
  { if (res == 0)
      return SoPlain(plain(from_array(p,plainSize())));
    return NoPlain();
  })
```

Observation Functions for Authenticated Encryption using Non-Sensitive Keys ($\overline{\texttt{EtM}}_p$)

```
// Encryption
_(def \EtMpENCIn EtMpENCIn(\EtMpState S,\EtMsState,\Bytes t,BYTE* k,UINT8 k_len,BYTE* p,UINT8 p_len,BYTE*)
  _(requires isEtMpState(S))
  _(requires isTemplate(t) && !sensitive(t))
  _(requires k_len == keylength(t))
  _(requires p_len == plainSize())
  _(ensures isEtMpENCIn(\result))
  { return (\EtMpENCIn) {
      .S = S, .t = t,
      .k = EtMpKey(t,from_array(k,k_len)),
      .p = from_array(p,p_len) }; })

_(def \EtMpENCOut EtMpENCOut(\EtMpState S,\EtMsState,\Bytes t,BYTE*,UINT8,BYTE*,UINT8,BYTE* c)
  _(requires isEtMpState(S))
  _(requires isTemplate(t) && !sensitive(t))
  _(ensures isEtMpENCOut(\result))
  { return (\EtMpENCOut) {
      .S = S, .c = from_array(c,cipherSize()) }; })

// Decryption
_(def \EtMpDECIn EtMpDECIn(\EtMpState S,\EtMsState,\Bytes t,BYTE* k,UINT8 k_len,BYTE* c,UINT8 c_len,BYTE*)
  _(requires isEtMpState(S))
  _(requires isTemplate(t) && !sensitive(t))
  _(requires k_len == keylength(t))
  _(requires c_len == cipherSize())
  _(ensures isEtMpDECIn(\result))
  { return (\EtMpDECIn) {
      .S = S, .t = t,
      .k = EtMpKey(t,from_array(k,k_len)),
      .c = from_array(c,c_len) }; })

_(def \EtMpDECOut EtMpDECOut(\EtMpState S,\EtMsState,\Bytes t,BYTE*,UINT8,BYTE*,UINT8,BYTE* p,UINT8 res)
  _(requires isEtMpState(S))
  _(requires isTemplate(t) && !sensitive(t))
  _(ensures isEtMpDECOut(\result))
  { if (res == 0)
      return SoBytes(from_array(p,plainSize()));
    return NoBytes();
  })
```

Simulation Contracts for the Encryption and Decryption of User-Provided Plaintexts ($\texttt{EtM}_{\texttt{Enc.h}}$)

```
//Encryption
void _(assume_correct) _ENC(_(ghost \EtMpState EtMp_S) _(ghost \EtMsState EtMs_S) _(ghost \Bytes t) BYTE* k,UINT8 k_len, BYTE* p,
      UINT8 p_len, BYTE* buffer _(out \EtMpState EtMp_O) _(out \EtMsState EtMs_O))
_(decreases 0)
_(requires isEtMpState(EtMp_S))
_(requires isEtMsState(EtMs_S))
_(ensures isEtMpState(EtMp_O) && leq(EtMp_S,EtMp_O))
_(ensures isEtMsState(EtMs_O) && leq(EtMs_S,EtMs_O))
_(requires isTemplate(t))
_(writes \array_range(buffer,(size_t) cipherSize()))
  // Simulation when sensitive
_(ensures sensitive(t) ⇒
    EtMsENCOut(EtMp_O,EtMs_O,t,k,k_len,p,p_len,buffer) ==
    EtMs_ENC(\old(EtMsENCIn(EtMp_S,EtMs_S,t,k,k_len,p,p_len,buffer))))
_(ensures sensitive(t) ⇒EtMp_O == EtMp_S)
  // Simulation when public
_(ensures !sensitive(t) ⇒
    EtMpENCOut(EtMp_O,EtMs_O,t,k,k_len,p,p_len,buffer) ==
    EtMp_ENC(\old(EtMpENCIn(EtMp_S,EtMs_S,t,k,k_len,p,p_len,buffer))))
_(ensures !sensitive(t) ⇒EtMs_O == EtMs_S);

// Decryption
UINT8 _(assume_correct) _DEC(_(ghost \EtMpState EtMp_S) _(ghost \EtMsState EtMs_S) _(ghost \Bytes t) BYTE* k,UINT8 k_len, BYTE* c,
      UINT8 c_len, BYTE* buffer)
_(decreases 0)
_(requires isEtMpState(EtMp_S))
_(requires isEtMsState(EtMs_S))
_(requires isTemplate(t))
_(writes \array_range(buffer,(size_t) plainSize()))
  // Simulation when sensitive
_(ensures sensitive(t) ⇒
    EtMsDECOut(EtMp_S,EtMs_S,t,k,k_len,c,c_len,buffer,\result) ==
    EtMs_DEC(\old(EtMsDECIn(EtMp_S,EtMs_S,t,k,k_len,c,c_len,buffer))))
  // Simulation when public
_(ensures !sensitive(t) ⇒
    EtMpDECOut(EtMp_S,EtMs_S,t,k,k_len,c,c_len,buffer,\result) ==
    EtMp_DEC(\old(EtMpDECIn(EtMp_S,EtMs_S,t,k,k_len,c,c_len,buffer))));
```

B.2 Key Derivation

We display the observation functions for module $\overline{\texttt{Seed}}$, and the simulation contracts for the system key derivation function. Together, they form the hypothesis we called $\mathcal{H}_{\overline{\texttt{KDF}}}$.

Observation Functions for Key Derivation ($\overline{\texttt{Seed}}$)

```
_(def \KDFIn KDFIn(\EtMsState EtMs_S, \EtMpState EtMp_S, \SeedState S, \Bytes t, BYTE*, UINT8, BYTE* s, UINT8 s_len, BYTE* buf,
      UINT8* buf_len)
  _(requires isEtMsState(EtMs_S))
  _(requires isEtMpState(EtMp_S))
  _(requires isSeedState(S))
  _(requires isTemplate(t))
  _(requires s_len == seedSize())
  _(ensures isKDFIn(\result))
  { return (\KDFIn) {
      .EtMs_S = EtMs_S, .EtMp_S = EtMp_S,
      .S = S, .t = t, .s = seed(from_array(s,s_len)) }; })
```

```
_(def \KDFOut KDFOut(\EtMsState EtMs_S, \EtMpState EtMp_S, \SeedState S, \Bytes t, BYTE*, UINT8, BYTE* s, UINT8 s_len, BYTE* buf,
      UINT8* buf_len)
  _(requires isEtMsState(EtMs_S))
  _(requires isEtMpState(EtMp_S))
  _(requires isSeedState(S))
  _(requires isTemplate(t))
  _(requires *buf_len == keylength(t))
  _(ensures isKDFOut(t,\result))
  { return (\KDFOut) {
      .EtMs_S = EtMs_S, .EtMp_S = EtMp_S,
      .S = S, .k = key(S,t,from_array(buf,*buf_len)) }; })
```

Simulation Contracts for Key Derivation (`seed.h`)

```
void _(assume_correct) _KDF(_(ghost \EtMsState EtMs_S) _(ghost \EtMpState EtMp_S) _(ghost \SeedState S) _(ghost \Bytes tmpl) BYTE*
       t, UINT8 t_len, BYTE* s, UINT8 s_len, BYTE* buf, UINT8* buf_len _(out \EtMsState EtMs_SOut) _(out \EtMpState EtMp_SOut) _(out
       \SeedState SOut))
  _(decreases 0) // Termination
  _(maintains \thread_local_array(t,t_len)) // Mem
  _(maintains \thread_local_array(s,s_len)) // Mem
  _(ensures \mutable_array(buf,*buf_len)) // Mem
  _(requires isTemplate(tmpl) && tmpl == from_array(t,t_len))
  _(writes \array_range(buf,(size_t) keylength(tmpl)), buf_len) // Writes
  _(ensures *buf_len == keylength(tmpl))
  _(ensures
      KDFOut(EtMs_SOut,EtMp_SOut,SOut,tmpl,t,t_len,s,s_len,buf,buf_len) ==
      KDF_r(\old(KDFIn(EtMs_S,EtMp_S,S,tmpl,t,t_len,s,s_len,buf,buf_len))));
```

B.3 Encryption of Device Objects

We display the observation functions for module $\overline{\texttt{EtM}}$, and the simulation contracts for the system function used for formatting and encrypting (and decrypting and parsing) Device objects. Together, they form the hypothesis we called $\mathcal{H}_{\overline{\texttt{EtM}}}$.

Observation Functions for the Protection of Device Objects ($\overline{\texttt{EtM}}$)

```
// Key Generation
_(def \EtMGENIn EtMGENIn(BYTE* b,\SeedState Seed_S,\EtMState S)
  _(requires isSeedState(Seed_S))
  _(requires isEtMState(Seed_S,S))
  _(ensures isEtMGENIn(\result))
  { return (\EtMGENIn) { .seedS = Seed_S, .S = S }; })

_(def \EtMGENOut EtMGENOut(BYTE* b,\SeedState Seed_S,\EtMState S)
  _(requires isSeedState(Seed_S))
  _(requires isEtMState(Seed_S,S))
  _(requires \mutable_array(b,(size_t) EtMKeySize()))
  _(ensures isEtMGENOut(Seed_S,\result))
  { return (\EtMGENOut) { .S = S, .k = EtMKey(from_array(b,EtMKeySize())) }; })

// Encryption
_(def \EtMENCIn EtMENCIn(\SeedState Seed_S,\EtMState S,BYTE* kb,UINT8 kl,slot_t* plain,BYTE* b)
  _(requires isSeedState(Seed_S))
  _(requires isEtMState(Seed_S,S))
  _(requires kl == EtMKeySize())
  _(requires \inv(plain) && plain->tmpl[0] < 2)
  _(ensures isEtMENCIn(\result))
  {
    \Bytes t = from_array(plain->tmpl,templateSize());
    \Bytes kr = from_array(plain->data,keylength(t));
```

```
    \key k = key(Seed_S,t,kr);
    return (\EtMENCIn) { .seedS = Seed_S, .S = S,
                         .k = EtMKey(from_array(kb,kl)),
                         .p = makeEntry(Seed_S,t,k) }; })

_(def \EtMENCOut EtMENCOut(\SeedState Seed_S,\EtMState S,BYTE*,UINT8,slot_t*,BYTE* b)
  _(requires isSeedState(Seed_S))
  _(requires isEtMState(Seed_S,S))
  _(ensures isEtMENCOut(Seed_S,\result))
  { return (\EtMENCOut) { .S = S, .c = from_array(b,EtMCipherSize()) }; })

//Decryption
//Begin Simulation
_(def \EtMDECIn EtMDECIn(
    \SeedState Seed_S,\EtMState S,
    BYTE* k,UINT8 kl,
    BYTE* c,UINT8 cl,
    slot_t*)
  _(requires isSeedState(Seed_S))
  _(requires isEtMState(Seed_S,S))
  _(requires kl == EtMKeySize())
  _(requires cl == EtMCipherSize())
  _(ensures isEtMDECIn(\result))
  { return (\EtMDECIn) {
      .seedS = Seed_S, .S = S,
      .k = EtMKey(from_array(k,kl)),
      .c = from_array(c,cl) }; })

_(def \EtMDECOut EtMDECOut(
    \SeedState S,\EtMState,
    BYTE*,UINT8,
    BYTE*,UINT8,
    slot_t* p,int res)
  _(requires isSeedState(S))
  _(requires res == 0 ⇒ \inv(p) && p->tmpl[0] < 2)
  _(ensures isEtMDECOut(S,\result))
  { if (res == 0)
      return SoEntry(makeEntry(S,p->t,key(S,p->t,p->kr)));
    return NoEntry(); })
//End Simulation
```

Simulation Contracts for the Protection of Device Objects (`EtM.h`)

```
void _(assume_correct) _GEN(_(ghost \SeedState Seed_S) _(ghost \EtMState S) BYTE* buffer _(out \EtMState outS))
_(decreases 0)
_(writes \array_range(buffer,(size_t) EtMKeySize()))
_(ensures EtMGENOut(buffer,Seed_S,outS) ==
    EtM_GEN(\old(EtMGENIn(buffer,Seed_S,S))));

void _(assume_correct) _slotENC(_(ghost \SeedState Seed_S) _(ghost \EtMState S) BYTE* key,UINT8 keylen, slot_t* plain, BYTE* buffer _(
      out \EtMState SOut))
_(decreases 0) // Termination
_(writes \array_range(buffer,(size_t) EtMCipherSize()))
_(requires \wrapped(plain) && plain->tmpl[0] < 2)
_(ensures EtMENCOut(Seed_S,SOut,key,keylen,plain,buffer) ==
    EtM_ENC(\old(EtMENCIn(Seed_S,S,key,keylen,plain,buffer))));

int _(assume_correct) _slotDEC(_(ghost \SeedState Seed_S) _(ghost \EtMState S) BYTE* key,UINT8 keylen, BYTE* cipher,UINT8 cipherlen,
      slot_t* res)
_(decreases 0)
_(updates res)
_(requires isEtMCipher(from_array(cipher,cipherlen)))
_(ensures \result != 0 <⇒ res->tmpl[0] == 2)
_(ensures EtMDECOut(Seed_S,S,key,keylen,cipher,cipherlen,res,\result) ==
    EtM_DEC(\old(EtMDECIn(Seed_S,S,key,keylen,cipher,cipherlen,res,))));
```

Appendix C

Over-Approximating P.P.T. Adversaries with Non-Determinism

The following code display is fully verified in VCC, proving that all its refinements (including p.p.t. refinements) call the Device commands only in configurations where their preconditions hold.

Non-Deterministic Adversary Over-Approximation

```
void main(void)
_(requires \program_entry_point())
_(writes \array_range(IOB,IOBLEN))
_(writes \extent(&store))
_(writes \extent(&EtMp_S))
_(writes \extent(&EtMs_S))
_(writes \extent(&Seed_S))
_(writes \extent(&EtM_S))
{
  // Initialise the configuration and
  // establish the initial observation
  deviceInit();

  // All p.p.t. adversaries are refinements of
  // the following non-deterministic loop
  while(1)
    _(writes &store)
    _(writes \array_range(IOB,IOBLEN))
    _(invariant \wrapped(&store))
    _(invariant \mutable_array(IOB,IOBLEN))
  { int i = 0;
    int r;
    // Fill the buffer non-deterministically
    while (i < IOBLEN)
      _(invariant i <= IOBLEN)
      _(writes \array_range(IOB,IOBLEN))
      _(invariant \mutable_array(IOB,IOBLEN))
    { unsigned char c;
      IOB[i] = c;
      i++; }

  // Choose the command to run non-deterministically
  switch (r)
  { case 0:
      Create();
      break;
    case 1:
      Import();
      break;
    case 2:
      Export();
      break;
    case 3:
      Clear();
      break;
    case 4:
      Load();
      break;
    case 5:
      Unload();
      break;
    case 6:
      Encrypt();
      break;
    case 7:
      Decrypt();
      break;
    default:
      // Use this to let the adversary skip a turn
  } } }
```

Bibliography

2009. URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3555. 1

2012. URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-5077. 1

Martín Abadi and Roger M. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions in Software Engineering*, 22(1):6–15, 1996. 31, 50

Martín Abadi and Jérémy Planul. On layout randomization for arrays and functions. In *Proceedings of the Second international conference on Principles of Security and Trust*, POST'13, pages 167–185, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-36829-5. doi: 10.1007/978-3-642-36830-1_9. URL http://dx.doi.org/10.1007/978-3-642-36830-1_9. 105

Martín Abadi and Gordon D. Plotkin. On protection by layout randomization. *ACM Transactions on Information and System Security*, 15(2):8:1–8:29, July 2012. ISSN 1094-9224. doi: 10.1145/2240276.2240279. URL http://doi.acm.org/10.1145/2240276.2240279. 105

Mihhail Aizatulin, François Dupressoir, Andrew D. Gordon, and Jan Jürjens. Verifying cryptographic code in C: Some experience and the Csec challenge. Technical Report MSR-TR-2011-118, Microsoft Research, November 2011a. iii, 40, 42

Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 331–340, New York, NY, USA, 2011b. ACM. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046745. URL http://doi.acm.org/10.1145/2046707.2046745. 2, 55, 68, 101

Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Computational verification of C protocol implementations by symbolic execution. In *Proceedings of the 2012 ACM conference on Computer*

and communications security, CCS '12, pages 712–723, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382271. URL http://doi.acm.org/10.1145/2382196.2382271. 2, 55, 56, 68, 101

Andrew W. Appel. Verified software toolchain. In *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software*, ESOP'11/ETAPS'11, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19717-8. URL http://dl.acm.org/citation.cfm?id=1987211.1987212. 13

Gergei Bana and Hubert Comon-Lundh. Towards unconditional soundness: computationally complete symbolic attacker. In *Proceedings of the First international conference on Principles of Security and Trust*, POST'12, pages 189–208, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28640-7. doi: 10.1007/978-3-642-28641-4_11. URL http://dx.doi.org/10.1007/978-3-642-28641-4_11. 34, 56

Manuel Barbosa, Jorge Sousa Pinto, Jean-Christophe Filliâtre, and Bárbara Vieira. A deductive verification platform for cryptographic software. In *Proceedings of the Fourth International Workshop on Foundations and Techniques for Open Source Software Certification*, volume 33 of *Electronic Communications of the Euroepan Association for the Study of Science and Technology*. EASST, 2010. 69

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 90–101, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480894. URL http://doi.acm.org/10.1145/1480881.1480894. 59, 69, 71

Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Proceedings of the 31st annual conference on Advances in cryptology*, CRYPTO'11, pages 71–90, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22791-2. URL http://dl.acm.org/citation.cfm?id=2033036.2033043. 59, 60, 69, 71

Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Formal verification of a microkernel used in dependable software systems. In *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP '09, pages 187–200, Berlin, Heidel-

berg, 2009. Springer-Verlag. ISBN 978-3-642-04467-0. doi: 10.1007/978-3-642-04468-7_16. URL http://dx.doi.org/10.1007/978-3-642-04468-7_16. 2

Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '00, pages 531–545, London, UK, 2000. Springer-Verlag. ISBN 3-540-41404-5. URL http://dl.acm.org/citation.cfm?id=647096.716997. 76, 78

Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. F7: refinement types for F#, September 2008. Available from http://research.microsoft.com/F7/. 5

Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Modular verification of security protocol code by typing. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 445–456, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706350. URL http://doi.acm.org/10.1145/1706299.1706350. 3, 15, 17, 20, 33, 36

Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, SP'13. IEEE, 2013. 60

Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations*, CSFW '01, pages 82–, Washington, DC, USA, 2001. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=872752.873511. 17, 34, 56

Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, October 2008. ISSN 1545-5971. doi: 10.1109/TDSC.2007.1005. URL http://dx.doi.org/10.1109/TDSC.2007.1005. 59, 71

Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Automatic validation of protocol narration. In *Proceedings of the 16th Computer Security Foundations Workshop*, pages 126–140, 2003. 53

Danilo Bruschi, Lorenzo Cavallaro, Andrea Lanzi, and Mattia Monga. Replay attack in tcg specification and solution. In *Proceedings of the 21st Annual Computer Security Applications Conference*, ACSAC '05, pages 127–137, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2461-3. doi: 10.1109/CSAC.2005.47. URL http://dx.doi.org/10.1109/CSAC.2005.47. 1

Christian Cachin and Nishanth Chandran. A secure cryptographic token interface. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, CSF '09, pages 141–153, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3712-2. doi: 10.1109/CSF.2009.7. URL http://dx.doi.org/10.1109/CSF.2009.7. 75

Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd IEEE symposium on Foundations of Computer Science*, FOCS '01, pages 136–, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1390-5. URL http://dl.acm.org/citation.cfm?id=874063.875553. 60

Sagar Chaki and Anupam Datta. ASPIER: an automated framework for verifying security protocol implementations. Technical CMU-CyLab-08-012, CyLab, Carnegie Mellon University, 2008. 55

Lily Chen. *NIST Special Publication 800-108: Recommendation for Key Derivation Using Pseudorandom Functions*. Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899-8900, October 2009. 76, 78

Ernie Cohen. First-order verification of cryptographic protocols. *Journal of Computer Security*, 11 (2):189–216, March 2003. ISSN 0926-227X. URL http://dl.acm.org/citation.cfm?id=859246.859249. 17, 33

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42, Berlin, Heidelberg, 2009a. Springer-Verlag. ISBN 978-3-642-03358-2. doi: 10.1007/978-3-642-03359-9_2. URL http://dx.doi.org/10.1007/978-3-642-03359-9_2. 2, 11

Ernie Cohen, Michał Moskal, Stephan Tobies, and Wolfram Schulte. A precise yet efficient memory model for C. *Electronic Notes in Theoretical Computer Science*, 254:85–103, October 2009b. ISSN 1571-

0661. doi: 10.1016/j.entcs.2009.09.061. URL http://dx.doi.org/10.1016/j.entcs.2009.09.061. 11, 13

Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 480–494, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14294-X, 978-3-642-14294-9. doi: 10.1007/978-3-642-14295-6_42. URL http://dx.doi.org/10.1007/978-3-642-14295-6_42. 11, 13

Ricardo Corin and Andrés Manzano. Efficient symbolic execution for analysing cryptographic protocol implementations. In *Proceedings of the Third international conference on Engineering secure software and systems*, ESSoS'11, pages 58–72, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19124-4. URL http://dl.acm.org/citation.cfm?id=1946341.1946348. 55

Loïc Correnson, Pascal Cuoq, Armand Puccetti, and Julien Signoles. *Frama-C User Manual*, 2010. URL http://frama-c.com/download/frama-c-user-manual.pdf. 2, 6

Cas J. Cremers. The Scyther tool: Verification, falsification, and analysis of security protocols. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 414–418, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70543-7. doi: 10.1007/978-3-540-70545-1_38. URL http://dx.doi.org/10.1007/978-3-540-70545-1_38. 34

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM. ISBN 0-89791-065-6. doi: 10.1145/582153.582176. URL http://doi.acm.org/10.1145/582153.582176. 5

David Delmas and Jean Souyris. Astrée: from research to industry. In *Proceedings of the 14th International Static Analysis Symposium*, SAS'07, pages 437–451. Springer, August 2007. 2

Shaddin F. Doghmi, Joshua D. Guttman, and F. Javier Thayer. Searching for shapes in cryptographic protocols. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'07, pages 523–537, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71208-4. URL http://dl.acm.org/citation.cfm?id=1763507.1763561. 34

Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983. 15

Francois Dupressoir, Andrew D. Gordon, Jan Jurjens, and David A. Naumann. Guiding a general-purpose c verifier to prove cryptographic protocols. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 3–17, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4365-9. doi: 10.1109/CSF.2011.8. URL http://dx.doi.org/10.1109/CSF.2011.8. iii, 51, 52, 53, 56, 105

F. Javier Thayer Fábrega. Strand spaces: proving security protocols correct. *J. Comput. Secur.*, 7(2-3):191–230, March 1999. ISSN 0926-227X. URL http://dl.acm.org/citation.cfm?id=353594.353603. 34

Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 341–350, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046746. URL http://doi.acm.org/10.1145/2046707.2046746. 59, 60, 62, 78, 83, 84

Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. doi: 10.1145/113445.113468. URL http://doi.acm.org/10.1145/113445.113468. 5

Patrice Godefroid and Sarfraz Khurshid. Exploring very large state spaces using genetic algorithms. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02, pages 266–280, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43419-4. URL http://dl.acm.org/citation.cfm?id=646486.694632. 2, 54

Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065036. URL http://doi.acm.org/10.1145/1065010.1065036. 2, 54

Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal*

of the ACM, 33(4):792–807, August 1986. ISSN 0004-5411. doi: 10.1145/6490.6503. URL http://doi.acm.org/10.1145/6490.6503. 59

Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, April 1984. 57

Jean Goubault-Larrecq and Fabrice Parrennes. Cryptographic protocol analysis on real C code. In *Proceedings of the 6th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 363–379, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-24297-X, 978-3-540-24297-0. doi: 10.1007/978-3-540-30579-8_24. URL http://dx.doi.org/10.1007/978-3-540-30579-8_24. 2, 55

Robert Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. 5

Engelbert Hubbers, Martijn Oostdijk, and Erik Poll. Implementing a formally verifiable security protocol in Java Card. In *Proceedings of the First International Conference on Security in Pervasive Computing*, volume 2802 of *Lecture Notes in Computer Science*, pages 213–226, 2004. 53

Bart Jacobs and Frank Piessens. The VeriFast program verifier. Report CS 520, Katholieke Universiteit Leuven, August 2008. 6

Alan Jeffrey and Ruy Ley-Wild. Dynamic model checking of C cryptographic protocol implementations. In *Proceedings of Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis*, 2006. 2, 54

C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25. 10

Jan Jurjens. Security analysis of crypto-based java programs using automated theorem provers. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 167–176, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2579-2. doi: 10.1109/ASE.2006.60. URL http://dx.doi.org/10.1109/ASE.2006.60. 53

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629596. URL http://doi.acm.org/10.1145/1629575.1629596. 2, 56, 70

Steve Kremer, Graham Steel, and Bogdan Warinschi. Security for key management interfaces. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF'11)*, pages 266–280, Cernay-la-Ville, France, June 2011. IEEE Computer Society Press. URL http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/KSW-csf11.pdf. 75

Ralf Kusters, Tomasz Truderung, and Jurgen Graf. A framework for the cryptographic verification of Java-like programs. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, pages 198–212, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4718-3. doi: 10.1109/CSF.2012.9. URL http://dx.doi.org/10.1109/CSF.2012.9. 53, 68

Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In Jean-Raymond Abrial, Michael Butler, Rajeev Joshi, Elena Troubitsyna, and Jim C. P. Woodcock, editors, *09381 Extended Abstracts Collection – Refinement Based Methods for the Construction of Dependable Systems*, number 09381 in Dagstuhl Seminar Proceedings, pages 104–108, Dagstuhl, Germany, 2010. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany. 2

Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '96, pages 147–166, London, UK, UK, 1996. Springer-Verlag. ISBN 3-540-61042-1. URL http://dl.acm.org/citation.cfm?id=646480.693776. 33

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL http://coq.inria.fr. Version 8.0. 5

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978. 5

Aybek Mukhamedov, Andrew D. Gordon, and Mark Ryan. Towards a verified reference implementation of a trusted platform module. In *17th International Workshop on Security Protocols (2009)*, Lecture Notes in Computer Science, pages 69–81. Springer, 2013. 54, 101

Nicholas O'Shea. Using Elyjah to analyse Java implementations of cryptographic protocols. In *Proceedings of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Aanalysis, and Issues in the Ttheory of Ssecurity*, pages 211–223, 2008. 53

Wolfgang Paul. Cyber war, formal verification and certified infrastructure. In *Proceedings of the 4th international conference on Verified Software: theories, tools, experiments*, VSTTE'12, pages 1–1, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-27704-7. doi: 10.1007/978-3-642-27705-4_1. URL http://dx.doi.org/10.1007/978-3-642-27705-4_1. 14

Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128, January 1998. ISSN 0926-227X. URL http://dl.acm.org/citation.cfm?id=353677.353681. 17, 33

Nadia Polikarpova and Michał Moskal. Verifying implementations of security protocols by refinement. In *Proceedings of the 4th international conference on Verified Software: theories, tools, experiments*, VSTTE'12, pages 50–65, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-27704-7. doi: 10.1007/978-3-642-27705-4_5. URL http://dx.doi.org/10.1007/978-3-642-27705-4_5. 53, 105

Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A semantics-based approach to malware detection. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 377–388, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4. doi: 10.1145/1190216.1190270. URL http://doi.acm.org/10.1145/1190216.1190270. 69

Valentin Robert and Xavier Leroy. A formally-verified alias analysis. In *Proceedings of the Second international conference on Certified Programs and Proofs*, CPP'12, pages 11–26, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-35307-9. doi: 10.1007/978-3-642-35308-6_5. URL http://dx.doi.org/10.1007/978-3-642-35308-6_5. 14

Jianqi Shi, Jifeng He, Huibiao Zhu, Huixing Fang, Yanhong Huang, and Xiaoxian Zhang. ORIENTAIS: Formal verified OSEK/VDX real-time operating system. In *Proceedings of the 2012*

IEEE 17th International Conference on Engineering of Complex Computer Systems, ICECCS '12, pages 293–301, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-2-9541-8100-4. doi: 10.1109/ICECCS.2012.27. URL http://dx.doi.org/10.1109/ICECCS.2012.27. 2

TPM Specification version 1.2. Parts 1–3. Trusted Computing Group, 2007. URL https://www.trustedcomputinggroup.org/specs/TPM/. 73

TPM Library Specification, Family "2.0", Level 00, Revision 00.96". Trusted Computing Group, March 2013. URL https://www.trustedcomputinggroup.org/resources/tpm_library_specification. 73, 101

Octavian Udrea, Cristian Lumezanu, and Jeffrey S. Foster. Rule-based static analysis of network protocol implementations. *Information and Computation*, 206(2-4):130–157, February 2008. ISSN 0890-5401. doi: 10.1016/j.ic.2007.05.007. URL http://dx.doi.org/10.1016/j.ic.2007.05.007. 2

Thomas Y. C. Woo and Simon S. Lam. A semantic model for authentication protocols. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy*, SP '93, pages 178–, Washington, DC, USA, 1993. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=882489.884188. 17