

Verifying Implementations of Security Protocols in C

François Dupressoir

Probation Report

Submitted to the Open University
as part of the first year probation assessment.

Supervisors: Andy Gordon (Microsoft Research)
Jan Jürjens (Open University
& Microsoft Research)
Bashar Nuseibeh (Open University)

Examiners: Charles Haley (Open University)
Pat Allen (Open University)

Contents

1	Introduction	3
1.1	Research problem	3
1.2	Challenges	3
1.2.1	The challenges of software verification	3
1.2.2	The challenges of security	4
1.2.3	The big challenge	4
1.3	Goals	4
1.3.1	Intended contributions	5
2	Related Work	6
2.1	Verifying low-level implementations of security protocols	6
2.2	Verifying high-level implementations of security protocols	7
2.3	Verifying C code	7
2.3.1	Verifying memory safety	8
2.3.2	General purpose C verifiers	8
3	Current Approach	10
3.1	A sample protocol for message authentication	10
3.1.1	Goals	10
3.1.2	Protocol Specification	10
3.2	Correspondences for security	11
3.3	Proving correspondence properties on C programs	11
3.3.1	Representing C code with functional byte arrays	12
3.3.2	Correspondences and byte arrays	12
3.3.3	Limits and next steps	12
4	Future Plans	13
4.1	General research orientation	13
4.2	Timeline	13
	Bibliography	15
A	Simple Protocol Implementations	19
A.1	Cryptographic and network libraries	19
A.2	Protocol implementations	19

B	Verifying an HMAC-Based Protocol	20
B.1	Predicate definitions	20
B.2	Annotating the cryptographic, network and memory API	21
B.3	Verifying the client	22

Chapter 1

Introduction

1.1 Research problem

Security protocols such as TLS [DR08], a protocol that establishes an authenticated private channel over an untrusted network, are widely used, from the protection of privacy to e-commerce through securing communications between aircrafts. However, if verifying their specifications for vulnerabilities is now a well understood—although still difficult (see Section 1.2.2)—problem, theoretically making them secure against a variety of attackers, it is still unsatisfactory for several reasons.

First, such specifications do not always exist. When dealing with very specific needs, and when the protocol is embedded into a bigger application, a formal model of the protocol such as the ones that are usually verified does not always exist. Moreover, even when dealing with mature implementations of well-specified and well-understood protocols, nothing ensures that no fatal security flaw has been introduced in the program. The most striking example of this is the latest important security advisory for OpenSSL [Ope]—a large open source implementation of the SSL [FKK96] and TLS protocols, that informs of a bad return code check that jeopardises authentication and secrecy, and has existed in the code since the first releases of the project [oCE09].

Our goal is to automatically analyse existing widely-deployed C implementations of security protocols for the presence of security-related bugs.

1.2 Challenges

1.2.1 The challenges of software verification

Software verification in general is a hard problem, in which most of the questions are undecidable. It is especially true when trying to verify very high-level properties such as authentication and secrecy, or full functional correctness, due to the discrepancies that often exist between the idealised specification of a program—what it is supposed to do, if the specification does not exist—and its concrete implementation.

Moreover, dealing with a low-level language such as C multiplies the amount of such discrepancies, even when there is no need for efficient algorithms. The use

of low-level memory management operations, the presence of arbitrary pointer aliasing, the possibility of unstructured control flow are all features of low-level languages that are difficult to reason about automatically, or even to express logically.

1.2.2 The challenges of security

Verifying security properties is also a challenge in itself due to the presence of a malicious attacker that has a certain control over the verification environment [NS78]. The first thing we have to do when dealing with properties that involve an attacker is to define a trust model, that describes what the attacker controls and what elements of the system can be trusted. In the case of security protocols, the underlying cryptographic primitives are often trusted, and the attacker is represented by the network, allowing him to intercept, delay and generate new messages. We will choose that trust model in a first approach, but it would be possible (although much harder) to not even trust the implementations of our cryptographic primitives. In our case, we also decide to trust the compiler (so that analysing the source code—as opposed to the executable—makes sense), and we assume the analysed program is memory-safe (see Section 2.3.1).

We also need to model the attacker’s knowledge, and the way it can deduce new knowledge from a set of known messages. Formal methods traditionally use a Dolev-Yao model, where cryptography is ideal (*i.e.* the attacker cannot get any information about an encrypted message unless he knows the decryption key)[DY83], but such symbolic models are too high-level to model the actual capabilities of real attackers. Cryptographers, on the other hand, use computational models, that deal with the underlying bitstring cryptographic operations, but are hard to reason about (even not automatically). There exists a recent but important body of work on computational soundness of symbolic models [AJ01, CKKW06, BHU09]. Even though there are many cases where the symbolic models are still not sufficient, we will consider a symbolic attacker, at least in a first approach. However, it is important to aim at providing, if at all possible, computational guarantees on programs.

1.2.3 The big challenge

Even more difficulties arise when mixing security and low-level implementations: we are then faced with the challenge of expressing the attacker knowledge in terms of logic formulas over the program state—which can be very complex because of aliasing, or with the challenge of lifting low-level memory operations up to the level of standard symbolic models to represent high-level primitives.

1.3 Goals

However difficult the problem may be, we believe that solving it has been made possible by recent and important advances in related fields in computer science, and especially in verification, both for general software—where applications range from completely automated verification of memory safety properties on big amounts of C code [BCC⁺03] to design-by-contract software engineering [KCH08, BDF⁺03, DMS⁺08]—and for models of security protocols—expressed

either in standard narration [CVB06] or using formal languages such as the π -calculus [Bla09].

We aim at providing a new tool to prove security properties of complex low-level implementations of widely deployed security protocols. If this method is to be used in practice to verify large-scale software, as other static analysis tools are used, we need it to be:

provably sound with respect to a realistic threat model,

scalable to large implementations of security protocols, that usually count hundreds of thousands of lines of code,

automated since security flaws are often due to the difficulties human beings have in understanding security, cryptography in particular, and especially when it comes to low-level implementations, and

modular since implementations of security protocols are usually used as libraries, and verifying their usage—or at least providing means to do so—is as important as verifying their implementation itself.

The modularity of the verification would also allow much more flexibility with the internal trust model, since we could decide arbitrarily to start or stop trusting code at a very fine level of granularity (in VCC, for example, the verification is performed function by function), and provides much of the scalability in most existing industrial verifiers. Unfortunately, it hinders automation, since it often requires large amounts of annotations, but we may subsequently work on inferring those annotations automatically.

1.3.1 Intended contributions

Apart from the interest of formally providing a certain assurance of security when running a given security-critical piece of software, solving this problem could lead to a better understanding, formalisation and verification of general high-level properties in low-level code. It could also give insights about the relation that binds existing approaches to high-level verification, such as Hoare-style contract based verification [Hoa69] and refinement type systems [FP91].

Chapter 2

Related Work

In this chapter, we introduce previous and ongoing research efforts in the domains of software verification of C programs, and automated verification of security protocols. In Section 2.1 we present work on verifying implementations of security protocols in low-level programming languages (*e.g.* C or, arguably, Java). Section 2.2 deals the verification of high-level implementations (or specifications) of such protocols. Section 2.3 then presents previous work on the verification of C programs.

2.1 Verifying low-level implementations of security protocols

The CSur project [GP05] lead the way, by providing a sound transformation of C code into a decidable subset of first-order logic, that can then be used to prove secrecy properties of the implementation. Their work, however, was only applied to a self-made implementation of the Needham-Schroeder protocol, that is not used as is in practice, and they included in their trust model the fact that the server behaved honestly.

Pistachio [ULF06] followed shortly, allowing to verify the compliance of an implementation with a rule-based specification of the communication steps of a protocol, very similar to the classic Alice and Bob notation. This approach allows to enforce constraints on the ordering of events and on the values of variables. Unfortunately, the analysis is unsound due to the heuristics that are used to approximate loops and recursion, and the need for a formal specification makes it impractical in some important cases. However, this work proved that verifying the full compliance of the messages exchanged by an implementation to an RFC specification was possible.

More recently, ASPIER [CD08] was proposed, implementing a verification framework based on a mix of predicate abstraction and model-checking, and successfully applied to a stripped down version of OpenSSL. However, the use of model-checking, although it provides much flexibility by providing explicit attacks—that can then be used to refine the abstraction if they are false positives—is limited to bounded instantiations of the different protocol roles in theory, and is in practice limited to 2 or 3 instances of each role, whereas real communication systems are not limited in such ways. Moreover, the paper only describes a

very abstract process that cannot be applied directly to C code and requires a substantial amount of manual abstraction.

Verifying Java implementations of security protocols has also been studied, and solutions similar to the one we seek have been provided [Jür06]. However, Java’s well-structured control-flow and memory model make it a rather easy target for verification, compared to C, especially when dealing with high-level properties. In particular, the absence of memory overlapping without equality, as well as the absence of unstructured control-flow statements such as `goto` allow to easily express properties of larger memory locations using only the control-flow graph. In that respect, we could almost classify Java as being high-level for verification purposes.

2.2 Verifying high-level implementations of security protocols

ProVerif [Bla09] and CryptoVerif [Bla08] both take a high-level—but running—implementation of a protocol in a variant of the applied π -calculus [AF01], along with a set of correspondences [WL93] to verify, and proves the correspondence properties of the program, or provides an explicit attack in case it finds one. However, there exists a major difference between the two tools: if ProVerif analyses the implementation in a symbolic model, where cryptography is unbreakable, CryptoVerif does the analysis in a computational model, where cryptography is not *efficiently* breakable. This makes it more sound with respect to the reality of attacks than ProVerif, but also makes its results much more unpredictable. Those tools have been used more concretely in the FS2PV [BFGT06] and FS2CV [BFCZ08] tools, to prove security properties of security protocols implemented in the F# functional language, and notably of a TLS implementation [BFCZ08]. Both FS2PV and FS2CV take F# code as input and output the corresponding ProVerif (respectively CryptoVerif) program, and require the use of a specific cryptographic API.

F7 [BBF⁺08] is another tool for the verification of F# implementations of security protocols. It is a refinement type system that can express advanced cryptographic properties on byte arrays, and relies on a theorem prover to prove the correspondence properties required by the type-checker. The use of a type system tends to localise the information as well as the proof, making them much more scalable than the standard verification solution, that represents an entire program in logic before extracting verification conditions. However, it also makes it difficult to get rid of false negatives, since it is much less easy to synthesise an attack from a type-checking error.

2.3 Verifying C code

We can also rely on existing toolsets for the verification of C programs, from advanced type systems to general property software verifiers.

2.3.1 Dialects, type systems, and abstract interpreters for memory safety

Even though we are not interested in low-level properties such as memory safety, it is important to note that establishing it is a necessary step before considering any higher-level static analysis, since the existence of buffer overflow vulnerabilities would allow for arbitrary control-flow and arbitrary code execution. The existence of those tools should enable us to safely assume that the programs we analyse are memory-safe.

CCured [NCH⁺05] is a type-based C code analyser based on the CIL framework. It verifies memory safety properties of existing C code and inserts runtime checks when it fails to prove a fact, thus making sure that the program is free from low-level security issues such as buffer-overflows without preventing it from running by being over-conservative, or cluttering it with run-time checks that adversely impact performance.

Also dealing with memory safety, this time in an entirely static manner, is the Astrée abstract interpreter, that was proved to be able to deal with large quantities of generated safety-critical code [BCC⁺03]. However, even though abstract interpretation [CC77] provides great modularity, it has only seldom been used outside of the realms of static analysis for safety [CC04, DG05].

Like CCured, Deputy [CHA⁺07] is a dependent type system for C that allows to verify memory safety. Unlike CCured, however, it defines a dialect of C, where the different kinds of pointers are represented differently in C, not only preventing it from being applied to existing implementations easily, but also breaking binary compatibility when changing a single pointer from being a standard C pointer to being a “fat” pointer, that holds extra dependent type information. It is however, interesting to see that advanced type systems have been developed for low-level, stateful languages like C, putting an F7-like refinement-type-based approach within reach of the C verifiers.

2.3.2 General purpose C verifiers

All the following tools are general verifiers for C, based on the same standard assertion-based verification approach [Hoa69].

VCC [DMS⁺08] is a contract-based verifier that aims particularly at verifying concurrent C programs. Although we do not need its concurrency-related features for now, its proving power, its clear, sound and simple semantics and memory model for C [CMST08] and its active development team made it a prime choice for testing purposes.

The Havoc project is very similar to VCC in its design, but presents a much more theoretical aspect, in the definition of the tool’s memory model, and in the project’s approach to interprocedural analysis and inference of relevant annotations. Havoc is much better documented than VCC, both technically and academically, but has not been applied to large-scale programs so far.

Unlike VCC and Havoc, Frama-C [BCF⁺08] uses a very simple memory model, relying on its modular plugin-based architecture to verify more precisely and more efficiently. As a tool, it is very well documented, but lacks the uses and the maturity of VCC. It is designed to be an implementation of the ACSL annotation language [BFM⁺08], which is the first standardised annotation language for C.

Typed memory models for C

VCC is built on top of Boogie [BCD⁺06], which was designed to verify programs written using object-oriented languages. Hence it is natural that its memory model is designed to be as close as possible to memory models classically used for those. However, many important differences between C and object-oriented languages make it impossible to simply use existing memory models for C# or Java, for example. Among other problems that are much more complex to solve in C than in object-oriented languages are the issues of pointer aliasing and explicit memory management, that are actually quite central to many C coding techniques. VCC's memory model [CMST08] is equivalent to (sound and complete with respect to) the traditional untyped byte-array based model used in Frama-C, but greatly improves on it on the aspect of efficiency and automation. A similar type system is defined for Havoc, with much more emphasis on the links between contracts and dependent types [CHLQ09].

Chapter 3

Current Approach

In this chapter, we describe our current approach to verify C implementations of security protocols. The general idea of our approach has been to adapt existing work, and notably the verification of correspondence properties in F# implementations, into C using a general verification engine. We chose to focus our efforts on applying an unpublished predicate-based usage of the F7 refinement type system on the C implementation of the same protocol (redefined above) that is used as an example in [BBF⁺08]. We first, in Section 3.1, introduce a simple protocol—slightly adapted from an example used in previous work [BBF⁺08, BFGT06]—that will be used in Section 3.2 to explain the notion of correspondences. Section 3.3 will then show how VCC allows us to verify correspondence properties directly on the C code, in the absence of an attacker and introduce the general framework in which we want to perform the verification.

3.1 A sample protocol for message authentication

We have implemented in C, and will eventually try verifying, a variety of security protocols, with well-studied security properties and flaws (see Appendix A). However, our tool in its current state—and purely for implementation reasons—cannot handle message concatenation. Thus, we will use the protocol defined below as a running example.

3.1.1 Goals

The protocol defined here allows an entity A to send an authenticated message to an entity B with whom he shares a secret. When B receives the message, he can check that A intended to send it to him at some point in the past (at this point, we do not want to ensure the injectivity of that correspondence).

3.1.2 Protocol Specification

Figure 3.1 defines the protocol in the Alice&Bob protocol narration style (see [CVB06]), where *msg* represents the message, k_{AB} is the key shared between A and B, and *hmac*() computes the SHA1 [Nat95] digest of *msg*, keyed with k_{AB} .

It is assumed, for brevity, that the underlying communication mechanism allows

$$\begin{aligned} A \rightarrow B: & \text{ msg} \\ A \rightarrow B: & \text{ hmac}(\text{msg}, k_{AB}) \end{aligned}$$

Figure 3.1: A simple hash-based message authentication protocol

to simply send a message without the risk of two or more messages colliding into one, or one message being fragmented. It is actually fairly straightforward, once the verification of protocols involving message concatenations is possible, to include a message header in the specification and verify that it is correctly sent.

3.2 Correspondences for security

Correspondences are properties of the form “if such event happens, then such other event has happened in the past”. They were introduced to model authentication [WL93], but can also be used to model secrecy and more complex properties [Bla09]. For clarity, we will be dealing only with non-injective correspondences (one-to-many event correspondence), although the lack of injectivity protects against replay attacks. Figure 3.2 shows through an example how correspondences model authentication. In this case, the correspondence we need to prove in order to establish the message authentication property is that whenever *Checked(text)* happens, then *Send(text)* has happened at least once before. To differentiate events from pseudo-code, the former are shown in italics.

<code>client(msg)</code>	<code>server()</code>
<i>Send(msg)</i>	<code>receive(msg)</code>
<code>send(msg)</code>	<code>receive(mac)</code>
<code>send(hmac(msg, kab))</code>	<code>verify(mac, msg, kab)</code>
	<i>Checked(msg)</i>

Figure 3.2: A simple example of correspondence for authentication

3.3 Proving correspondence properties on C programs

The challenge we are now facing is to link high-level correspondence properties such as the one shown in Figure 3.2 to a low-level C implementation, that has to be robust and efficient. Our basic implementation of the example protocol is 40 lines long (as opposed to the 4-line description above), without taking into account the network setup, cryptographic primitives, and system-related features (*e.g.* command-line processing). Moreover, those 40 lines of code contain dynamic memory allocation and advanced use of call-by-reference to produce an out-parameter (see Appendix A for more details). Thus, we need to somehow abstract away these low-level operations in order to cleanly express correspondence properties and verify them.

3.3.1 Representing C code with functional byte arrays

To solve this problem, we used VCC [DMS⁺08] to associate, to all cryptographically significant memory locations, a specification-only functional byte array. The byte arrays are implemented using VCC’s ability to refer to the memory state of the program at any chosen program point by saving it. This way, we can refer to the value of specific bytes, and compare them, or express complex properties on them using predicates. We also implement a set of specification primitives to indicate what particular bytes—in the form of a typed pointer and a length—we want a given byte array to refer to. Ideally, a user should not have to explicitly express this relation, and we can indeed ensure—by expressing pre and post-conditions on functions in a certain style—that this indeed works in simple cases. However, current limitations of VCC and the underlying tools make it necessary to express this relation between low-level program variables and high-level specification values when the deduction of some facts is not immediate (for more details, see Appendix B).

3.3.2 Correspondences and byte arrays

We use this high-level representation of array values to define a set of structural predicates that logically describe the behaviour of cryptographic primitives. Most of those predicates are general and can be defined once, and a small subset of them have to be defined on a protocol-specific basis (for example to encode what a valid message is). The predicates allow us to express cryptographic relationships between the byte array variables (*e.g.* “b is the hash of m under key k”). We also give access to the user to a set of event predicates that express the correspondence-related events. By annotating a C implementation with the event predicates and giving correct pre and post-conditions to memory, cryptographic, and network primitives it is then possible to prove, simply using VCC, that the correspondence is valid in the absence of an attacker.

3.3.3 Limits and next steps

This limitation is due to the fact that nowhere in the process do we encode an actual attacker. We give some information on the attacker model in the definition of the structural predicates and the conditions on cryptographic primitives, but a sound approach would verify those against a formal definition of the attacker’s capabilities (Dolev-Yao or computational). Thus, the immediate next step is to introduce the notion of attacker in the verification process. However, it is unclear which solution would be easier, between expressing the attacker at the level of the C language—to enable the whole verification process to take the attacker into account, and expressing it purely in logic—and introducing it only during the later steps of verification.

Other limitations arise from VCC’s approach to verification. For example, the use of axioms makes it necessary to guide Z3 in some proofs and provide intermediate steps. We will look into ways of mixing automated theorem proving and tactics, since security protocols are often very similar. This could also help solving the issues we are having with the formalisation in C of concatenation.

Chapter 4

Future Plans

4.1 General research orientation

I will now be focusing my efforts on introducing an attacker model into our current approach. For this purpose, and with the added benefit of improving the scalability and modularity of the verification, I believe that making the analysis more local—through a type-based approach—is a good way to go. I will get to read on the links between advanced type systems and Hoare-style verification (Hoare type theory [NM05], typed memory models for C [CMST08, CHLQ09]), and the Havoc property verifier.

Further Reading In order to get a full understanding of the existing work in the domain, I still need to read in details about PCL [DDMR07], a protocol description logic that is used to provide compositional proofs of security properties on protocols, including some computational guarantees[DDMW06].

4.2 Timeline

By August 2009, we will finish implementing the cryptographic and network annotated API. We will try to organise some programming event at the Marktoberdorf summer school to get a varied set of implementations of our sample protocols. For this work to be relevant, we will need to provide realistic pre and post-conditions on the cryptographic primitives, both from a verification and a concrete cryptography point of view. I am currently in the process of reading about the different classes of security properties cryptographic schemes enforce.

By October 2009, we will have evaluated the existing tool on the sample protocols and could submit a paper to TACAS (part of the ETAPS joint event). Even in the absence of an active attacker, the current state of the project shows progress on previous approaches by being applied directly to running code, for which we do not require a specification to exist and, that was written without verification in mind (if we manage to get external implementations).

In the next year, I will work on the theoretical foundations of the verification system I have developed, possibly including a formal proof of soundness with respect to a clearly defined and realistic attacker model. This should give rise to publications in formal methods and security events (*e.g.* ESOP, CSF).

Throughout the process, I will keep in mind the fact that our targets are not simple 500 line implementations but large programs that robustly implement several protocols on very diverse hardware architectures. As soon as we get our system to work cleanly on our simple examples, we should continue our efforts to make it scalable.

Acknowledgements

This is joint work with Mihhail Aizatulin. The project is partly funded by the Microsoft Research PhD Scholarship programme.

Bibliography

- [AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. *ACM SIGPLAN Notices*, 36(3):104–115, 2001.
- [AJ01] Martín Abadi and Jan Jürjens. Formal eavesdropping and its computational interpretation. In *TACS '01: Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software*, page 82–94, London, UK, 2001. Springer-Verlag.
- [BBF⁺08] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffei. Refinement types for secure implementations. In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, page 17–32, Washington, DC, USA, 2008. IEEE Computer Society.
- [BCC⁺03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. *SIGPLAN Not.*, 38(5):196–207, 2003.
- [BCD⁺06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for Object-Oriented programs. volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.
- [BCF⁺08] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL implementation, 2008.
- [BDF⁺03] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M Leino, Wolfram Schulte, K. Rustan, and M. Leino. Verification of Object-Oriented programs with invariants. *JOURNAL OF OBJECT TECHNOLOGY*, 3:2004, 2003.
- [BFCZ08] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zălinescu. Cryptographically verified implementations for TLS. Alexandria, VA, October 2008. ACM.
- [BFGT06] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, page 139–152, Washington, DC, USA, 2006. IEEE Computer Society.

- [BFM⁺08] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C specification language, October 2008.
- [BHU09] Michael Backes, Dennis Hofheinz, and Dominique Unruh. CoSP: a general framework for computational soundness proofs - or - the computational soundness of the applied pi-calculus. Preprint on IACR ePrint 2009/080, February 2009.
- [Bla08] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, 2008.
- [Bla09] Bruno Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 2009. To appear.
- [CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. page 238—252, Los Angeles, California, 1977. ACM Press, New York, NY.
- [CC04] Patrick Cousot and Radhia Cousot. An abstract interpretation-based framework for software watermarking. page 173—185, Venice, Italy, 2004.
- [CD08] Sagar Chaki and Anupam Datta. ASPIER: an automated framework for verifying security protocol implementations. Technical CMU-CyLab-08-012, CyLab, Carnegie Mellon University, 2008.
- [CHA⁺07] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George Necula. *Dependent Types for Low-Level Programming*, pages 520–535. 2007.
- [CHLQ09] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 302–314, Savannah, GA, USA, 2009. ACM.
- [CJ97] John Clark and Jeremy Jacob. A survey of authentication protocol literature. Technical report, 1997.
- [CKKW06] Véronique Cortier, Steve Kremer, Ralf Küsters, and Bogdan Warinschi. *Computationally Sound Symbolic Secrecy in the Presence of Hash Functions*, pages 176–187. 2006.
- [CMST08] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A precise yet efficient memory model for C. October 2008.
- [CVB06] Carlos Caleiro, Luca Vigano, and David Basin. On the semantics of Alice&Bob specifications of security protocols. *Theoretical Computer Science*, 367(1-2):88–122, 2006.

- [DDMR07] A Datta, A Derek, J Mitchell, and A Roy. Protocol composition logic (PCL). *Electronic Notes in Theoretical Computer Science*, 172:311–358, 2007.
- [DDMW06] A. Datta, A. Derek, J.C. Mitchell, and B. Warinschi. Computationally sound compositional logic for key exchange protocols. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 321–334, Venice, Italy, 2006.
- [DG05] Mila Dalla Preda and Roberto Giacobazzi. Semantic-Based code obfuscation by abstract interpretation. pages 1325–1336, 2005.
- [DMS⁺08] Markus Dahlweid, Michał Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: contract-based modular verification of concurrent C. 2008.
- [DR08] T Dierks and E Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246*. August 2008.
- [DY83] D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.
- [FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher. *The SSL Protocol Version 3.0*. November 1996.
- [FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. *SIGPLAN Not.*, 26(6):268–277, 1991.
- [GP05] 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)*, volume 3385 of *Lecture Notes in Computer Science*, page 363–379. Springer, 2005.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- [Jür06] Jan Jürjens. Security analysis of crypto-based Java programs using automated theorem provers. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, page 167–176, Washington, DC, USA, 2006. IEEE Computer Society.
- [KCH08] Joseph Kiniry, Patrice Chalin, and Clément Hurlin. *Integrating Static Checking and Interactive Verification: Supporting Multiple Theories and Provers in Verification*, pages 153–160. 2008.
- [Low96] 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*, pages 147–166. Springer-Verlag, 1996.
- [Nat95] National Institute of Standards and Technology. Secure hash standard, April 1995.

- [NCH⁺05] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.
- [NM05] Aleksandar Nanevski and Greg Morrisett. Dependent type theory of stateful Higher-Order functions. Technical Report TR-24-05, Harvard Computer Science, 2005.
- [NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
- [oCE09] oCERT. oCERT advisory #2008-16 multiple OpenSSL signature verification API misuse, 2009.
- [Ope] The OpenSSL Project. OpenSSL.
- [OR87] Dave Otway and Owen Rees. Efficient and timely mutual authentication. *SIGOPS Oper. Syst. Rev.*, 21(1):8–10, 1987.
- [ULF06] Octavian Udrea, Cristian Lumezanu, and Jeffrey S Foster. Rule-Based static analysis of network protocol implementations. *IN PROCEEDINGS OF THE 15TH USENIX SECURITY SYMPOSIUM*, pages 193–208, 2006.
- [WL93] T.Y.C. Woo and S.S. Lam. A semantic model for authentication protocols. In *Proceedings 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, Oakland, CA, USA, 1993.

Appendix A

Simple Protocol Implementations

As part of the evaluation framework for our tool, we developed a small library of simple working protocols, that are described here.

A.1 Cryptographic and network libraries

We implemented a cryptographic and network library as wrappers around existing libraries to ensure a certain level of consistency in our test implementations and facilitate the initial annotation effort. Once it is stabilised, this API could be used to interact with other projects. Moreover, having a fixed API to work on allows us to focus on developing realistic pre and post-conditions for cryptographic operations.

A.2 Protocol implementations

Built on top of the cryptographic and network API, we provide implementations of several simple protocols:

- the HMAC-SHA1-based protocol used as an example and defined in Section 3.1,
- an authenticated request-response protocol built on top of the previous one, and specified below:

$$\begin{aligned} A \rightarrow B: & \text{ } (req|hmac((1|req), k_{AB})) \\ B \rightarrow A: & \text{ } (res|hmac((2|res|hmac((1|req), k_{AB})), k_{AB})) \end{aligned}$$

- the Otway-Rees key distribution protocol [OR87],
- the Needham-Schroeder-Lowe authentication and key-exchange protocol [NS78] as fixed by Lowe [Low96].

We deal with the inherent typing vulnerabilities, such as the one in the Otway-Rees protocol [CJ97], by using fixed-sizes for everything but the payload and making sure this kind of attacks is not possible on our implementations.

Appendix B

Verifying an HMAC-Based Protocol

This appendix gives more implementation details on our predicates, underlining their strength and their shortcomings by showing them at work on a simple example.

B.1 Predicate definitions

We declare a small set of predicates on byte arrays:

- $Bytes(m)$ means that m is a valid byte array, as opposed to a random sequence of bits,
- $KeyAB(a, b, k)$ means that the bytes k represent a key shared between entities a and b .
- $Pub(m)$ means that the bytes m are meant to be sent over the network, or were received from the network. It is a precondition to sending a byte array over the network that it is public.
- $IsMAC(m, k, w)$ means that the bytes m contain the MAC of bytes w under key k .
- $MACSays(k, m)$ is a protocol-specific predicate that can represent the specific usage that is made of a MAC.
- $Send(a, b, m)$ is the event predicate. It is the only predicate in this set that can be, in a sound approach, assumed (this is the way the event is *logged*, as in [BBF⁺08]).

These predicates are defined axiomatically (VCC forbids induction in specification code) as in Figure B.1, where a and b range over entities, and all other letters range over byte arrays.

Those definitions can be interpreted pretty easily. The first one, for example, states that, if the MAC computation is authorised ($MACSays(k, w)$) and valid ($IsMAC(m, k, w)$), and the key and message are valid byte arrays ($Bytes(k) \wedge$

General definitions

-
- $\forall m, k, w. Bytes(k) \wedge Bytes(w) \wedge MACSays(k, w) \wedge IsMAC(m, k, w) \Rightarrow Bytes(m)$
 - $\forall m, k, w. IsMAC(m, k, w) \Rightarrow (Pub(w) \Rightarrow Pub(m))$
 - $\forall a, b, k. KeyAB(a, b, k) \Rightarrow (\forall m. MACSays(k, m) \Leftrightarrow Send(a, b, m))$

Protocol-specific definitions

-
- $\forall k, w. (\forall a, b. KeyAB(a, b, k) \wedge Send(a, b, w) \Rightarrow MACSays(k, w))$
 - $\forall m, k, w. IsMAC(m, k, w) \Rightarrow MACSays(k, w)$

Figure B.1: Predicate definitions

$Bytes(w)$) then the resulting MAC is also a valid byte array ($Bytes(m)$). In the case of the studied protocol (and in the absence of an attacker), $MACSays()$ is very simple, but can become much more complex when dealing with nested hashes, for example.

B.2 Annotating the cryptographic, network and memory API

This section shows how underlying primitives are annotated with the predicates, by showing and commenting on annotated code snippets from the API.

The simplest example we can provide for interesting predicate annotations is the `getKey()` function from the cryptographic API, shown in Figure B.2. The code snippet shows two annotations: the first one deals with memory safety, and ensures that VCC considers the memory location of the returned structure readable; the second one, more interesting to us, states that the returned key is valid, and is shared between the two argument entities. The `Stored()` predicate ensures the translation between the low-level C representation as a pointer and a length, and the byte array notation used in the specification.

```
// Key stuff
struct keys
{
    size_t length;
    char *key;
};

struct keys getKey(unsigned int a, unsigned int b)
ensures(is_mutable_array(result.key, result.length))
ensures(forall(byte_array k;
               Stored(k, result.key, result.length)
               ==> KeyAB(a, b, k)));
```

Figure B.2: The annotated `getKey()` prototype

Similarly, we can annotate the hash verification function (Figure B.3) to correctly use the predicates to model cryptography, and the network send function

(Figure B.4) to require all sent bytes to be public.

```
int HMACSha1Verify(char *message, size_t length,
                  char *key, size_t key_length, char *MAC)
ensures(result ==> forall(byte_array w, k, m;
                        Stored(w, message, length)
                        && Stored(k, key, key_length)
                        && Stored(m, MAC, MD_LEN)
                        ==> IsMAC(m, k, w)));
```

Figure B.3: The annotated HMACSha1Verify() prototype

```
size_t long sendHL(SOCKET socket, char *sendbuf,
                  size_t length)
requires(forall(byte_array w;
                Stored(w, sendbuf, length)
                ==> Pub(w)));
```

Figure B.4: The annotated sendHL() prototype

Soundness and memory-writing functions

Unfortunately, for soundness, we also need to make sure that all used memory writing functions that do not get verified are annotated with the correct `writes()` clause. Otherwise, VCC just assumes that they do not write into memory and this leads to serious unsoundness problems (see Figure B.5 for an example of such an annotation). This will strongly condition a potential formal proof of soundness for our tool.

```
void memcpy_s(char* _Dst, const size_t _DstSize,
              const char* _Src, const size_t _MaxCount)
writes(array_range(_Dst, _MaxCount))
requires(_DstSize >= _MaxCount)
requires(is_mutable_array(_Dst, _DstSize))
ensures(forall(state_t b;
                Stored(b, _Src, _MaxCount)
                ==> Stored(b, _Dst, _MaxCount)));
```

Figure B.5: The annotated memcpy_s() prototype

B.3 Verifying the client

The predicates are now defined, including `MACSays()`, and the API is annotated. We can verify our first implementation. The code in Figure B.6 is an

implementation of the client side (sender) for the simple protocol we have been studying. Ideally, there would be no explicit mention to byte arrays in the code itself and we could let VCC instantiate the pre and post-conditions to function calls with the correct byte arrays. Unfortunately, because of the incompleteness of the underlying tools, and because we are using axioms, we need to guide the proof at one point and this requires to instantiate those byte array by hand throughout the program. You will also notice a small set of assumptions after the memory allocation, that are due to the way VCC handles arrays. There are ways to get rid of those assumptions (and hence of the possibility of inconsistencies in our logic), but the amount of annotations required is non-trivial, and sometimes actually requires to modify parts of the code. This should be fixed when the memory reinterpretation mechanism described in [CMST08] is implemented in VCC.

```

int WrapAndSend(SOCKET socket, char *message, size_t length)
requires (mutable(message))
requires (forall(size_t i; i < length; mutable(message + i)))
{
    spec(byte_array w, k, h;)
    char *signature;

    struct keys keys = getKey(1, 2);
    size_t key_len = keys.length;
    char *key = keys.key;

    if (length >= DEFAULT_BUFLen - 5 - MD_LEN)
    {
        printf("This message is way too long...\n");
        return 1;
    }

    signature = malloc(MD_LEN);
    assume(Disjoint(signature, MD_LEN, key, key_len));
    assume(Disjoint(signature, MD_LEN, message, length));

    Store(k, key, key_len);
    Store(w, message, length);
    assert(MACSays(k, w));

    HMACSha1(message, length, key, key_len, signature);

    Store(h, signature, MD_LEN);
    assert(IsMAC(h, k, w));
    assert(Pub(h));

    if (sendHL(socket, message, length) <= 0) return 1;
    if (sendHL(socket, signature, MD_LEN) <= 0) return 1;

    return length;
}

```

Figure B.6: The annotated client code