

# Verifying Implementations of Security Protocols in C

## Extended Abstract

François Dupressoir

June, 18-19, 2009  
CRC PhD Students Conference

**Supervisors:** Dr Andrew Gordon (MSRC)  
Dr Jan Jürjens (OU/MSRC)  
Prof Bashar Nuseibeh (OU)

**Department:** Computing

**Status:** Full-time

**Probation viva:** Not Passed

**Starting date:** October 1, 2008

## 1 Introduction

Static program analysis has come a long way since its beginning as an optimisation tool in compilers. It is now being used, not only to find bugs (for example, in device drivers, using the SLAM model checker [BCLR04]), but also to help developing correct applications from the start (see for example ESC/Java2 or Spec#), and even to prove programs bug-free (for example, the ASTRÉE abstract interpreter [BCC<sup>+</sup>03] was used to prove the Airbus A380's flight commands software memory-safe), and all that in industrial code. However, in the domain of computer security, little has been done on verification of programs written in low-level languages such as C. Even though work on proving properties of security protocols on the specification level exists, it remains unsatisfactory since what the security of the computer system relies on in the end is the program that runs, the implementation, and not its specification. A programmer might -willingly or not- introduce security-critical bugs in a piece of software, possibly causing catastrophic consequences. As an example, the latest security hole in openssl, one of the leading implementations of the SSL secure network transport protocol, was caused by a bad check on the return value of the server certificate checking function, causing the client to accept invalid certificates as valid, and possibly sending private information to a third party unknowingly [oCE09].

## 1.1 The issues with security

But analysing security protocols automatically is a hard problem, even when working on specifications, due to the presence of an attacker.

**Modeling the attacker** The first issue that arises from the presence of an attacker is the choice of an attacker model that defines what it is allowed to do. If the model is too weak (does not give it enough power), we will let some important bugs through. On the other hand, a model that is too strong will make the verification too complex and easily outgrow its allocated resources. We will especially have to choose between a computational model, closer to the reality of modern cryptography, and a symbolic model. Under certain assumptions, it can be proved that a symbolic model is computationally sound (see, for example, previous work introducing a general framework for proving computational soundness of symbolic models [BHU09]), but this would then require to prove that the program and the cryptography it uses verify those assumptions. On the other hand, computational properties are not only harder to verify, but also harder to express in ways the standardly used tools understand, although we will see in section 1.2 that some tools now exist for the verification of computational properties. Apart from using the symbolic model, usual assumptions made when verifying cryptographic protocols are that the attacker does not have access to the hardware, or the timing channels, for example (which is usually the case in reality when the protocol is run over the internet).

**Attacker knowledge** Modeling the attacker is not the only problem we have to face, though: most of the interesting security properties actively refer to the knowledge of the attacker, which is not directly known from the state of the program (it may depend on the state of multiple running processes, that the current process does not know of, or need a separate inference step), making it difficult, and sometimes impossible to express them as assertions of closed boolean expressions in the program. Moreover, inferring the attacker knowledge requires, in general, the use of a theorem prover, automated or not, adding much work onto the already resource-intensive verification process, especially in a computational setting. The particulars of the attacker knowledge inference rules are given as part of the attacker model definition.

**Modeling trust** In any secure computer system, there are trusted and untrusted components. When statically analysing security-critical software, it is necessary to clearly state which components are trusted, and for what purpose they exist (so that if they are used in another way, a warning can be issued). This is particularly important when dealing with cryptography, since verifying low-level implementations of cryptographic primitives such as RSA encryption or key generation would require a clean formalisation of advanced mathematical theorems, and perhaps of their proofs. Here again, the choice that was made for the threat model drives the definition of the trust model, since dealing with an attacker that would have write access to the source code (malicious programmer) would void all trust assumptions we could make on the program itself.

## 1.2 Existing Tools

However hard the problem is, recent work exists on the verification of implementations of security protocols.

**Verifying low-level implementations of security protocols** The CSur project [GP05] lead the way, by providing a sound transformation of C code into Horn clauses, 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 the client side of the Needham-Schroeder protocol, that is not used as is in practice.

Pistachio [ULF06] followed shortly, allowing to verify the compliance of an implementation with a rule-based specification of the communication steps of a protocol. 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 analyse the code, and the amount of work required to translate textual specifications (IETF or RFC documents) into rules is too big to be practical.

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 the OpenSSL handshake. However, the use of model-checking, although it provides much more precision by allowing counter-example guided abstraction refinement, is limited to bounded instantiations of the protocol roles in theory, and is in practice limited to 2 or 3 instances of each role, whereas real communication systems are often much larger.

**Verifying high-level implementations of security protocols** ProVerif [Bla09] and CryptoVerif [Bla08] both take a high-level implementation of a protocol in the applied  $\pi$ -calculus, along with a list of correspondences to verify, and proves the correspondence properties of the program, or provides an explicit attack in case it finds one. 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 closer to the reality of modern cryptography than ProVerif, but also makes its results much less predictable. Those tools have been used more concretely in the FS2PV and FS2CV tools, to prove security properties of security protocols implemented in the F# functional language, and notably of a TLS implementation [BFCZ08]. F7 [BBF<sup>+</sup>08] is another tool developed at Microsoft Research 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.

**Verifying C code** Although there are numerous tools and methods for C code analysis, we will only here talk about Microsoft Research's Verified C Compiler (VCC, [DMS<sup>+</sup>08]), that we used in our first efforts to further understand the issues we faced. VCC 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.

### 1.3 Goals

With this project, we aim at providing a new tool to prove security properties of low-level implementations of industrial strength security protocols. If this method is to be used practically to verify industrial-quality software, as other static analysis tools are used, we need it to be:

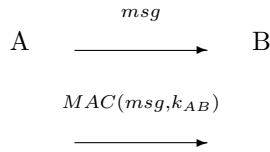
- **sound**, since a single undiscovered security flaw can render the entire security architecture useless,
- **scalable**, since industrial-strength implementations of security protocols usually count hundreds of thousands of lines of code,
- **automated**, since security flaws are often due to the 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.

The modularity of the verification could 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 (using VCC, we could trust a function, but not its callers or its callees, or the functions that belong to the same source file), and provides much of the scalability in most existing industrial verifiers. Unfortunately, it hinders automation, since the modularity often requires rather large amounts of annotations, but we could imagine in a second step working on inferring those annotations automatically.

## 2 Verifying simple implementations

### 2.1 First steps with VCC

As explained earlier, we chose for various reasons to work using VCC, and we tried out the tool on a simple implementation of the Needham-Schroeder-Lowe protocol, the CSur implementation of it being a bit too liberal in its use of type casts for VCC in its state at the time. We however managed to verify it using a rather uncommon approach, by writing an abstract state machine for the protocol in C, and then using VCC to verify the actual code against it, using a symbolic version of cryptography. However, this approach required a very good understanding of the code (to know what calls should trigger the abstract state machine and symbolic cryptographic calls), and required to write an abstract state machine for the protocol in C. Both are easy to do for a simple protocol such as NSL, but becomes rather complex when dealing with big implementations of industrial-strength protocols.



where  $k_{AB}$  is a secret key shared between A and B.

Figure 1: A MAC-based message authentication protocol.

## 2.2 Using predicates

We decided to refine the approach and increase its reusability by implementing a set of predicates in VCC, that expressed cryptographic primitive operations, such as hashing and sending over the network. Using those predicates, we managed to soundly prove, on a simple two-message MAC-based message authentication protocol (see figure 1) that all send operations in the client complied with its specification (a level of verification very similar to the one Pistachio [ULF06] achieves, with added soundness). However, technical limitations in VCC forced rather strict assumptions on the protocol, that could not always be ensured (there could be only one call to each one of the cryptographic primitives), and we did not push this effort further.

## 2.3 Predicates on byte arrays

Instead, we decided to take advantage of recent, still unpublished at this time, developments on using F7 to adapt this work to C. We indeed managed to implement functional byte arrays in VCC, making it much easier to reason about the program, and effectively taking a step towards extracting a faithful higher-level model from it. Once the functional byte arrays are available to use in VCC specification code, and once we have defined some translation between the C code representation of strings and memory (that usually consists of a pair containing a pointer to the array and its length, either explicitly for dynamic arrays, or implicitly for static arrays and null-terminated strings) and the high-level byte-array notation we define, we can then define the cryptographic predicates on byte arrays very much in the style of F7 predicates and start using them to prove properties of the code. At this point in time, our attacker model is very limited and does not allow the attacker to compromise keys, but we managed in this setting to prove the authentication property the protocol in figure 1 should ensure for the server (*i.e.* if the server verifies the MAC on the message successfully, then it comes from the client).

## References

- [BBF<sup>+</sup>08] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 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<sup>+</sup>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.
- [BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*, pages 1–20, 2004.
- [BFCZ08] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zălinescu. Cryptographically verified implementations for TLS. Alexandria, VA, October 2008. ACM.
- [BHU09] Michael Backes, Dennis Hofheinz, and Dominique Unruh. 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.
- [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.
- [CMST08] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A precise yet efficient memory model for C. October 2008.
- [DMS<sup>+</sup>08] Markus Dahlweid, Michał Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: contract-based modular verification of concurrent C. 2008.
- [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.
- [oCE09] oCERT. oCERT advisory #2008-16 multiple OpenSSL signature verification API misuse, 2009.
- [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.