

Verifying Authentication Properties of C Security Protocol Code Using General Verifiers

François Dupressoir

Supervisors Andy Gordon (MSR)
Jan Jürjens (TU Dortmund)
Bashar Nuseibeh (Open University)
Department Computing
Registration Full-Time
Probation Passed

1 Introduction

Directly verifying security protocol code could help prevent major security flaws in communication systems. C is usually used when implementing security software (*e.g.* OpenSSL, cryptlib, PolarSSL...) because it provides control over side-channels, performance, and portability all at once, along with being easy to call from a variety of other languages. But those strengths also make it hard to reason about, especially when dealing with high-level logical properties such as authentication.

Verifying high-level code. The most advanced results on verifying implementations of security protocols tackle high-level languages such as F#. Two main verification trends can be identified on high-level languages. The first one aims at soundly extracting models from the program code, and using a cryptography-specific tool such as ProVerif (*e.g.* fs2pv [BFGT06]) to verify that the extracted protocol model is secure with respect to a given attacker model. The second approach, on the other hand, aims at using general verification tools such as type systems and static analysis to verify security properties directly on the program code. Using general verification tools permits a user with less expert knowledge to verify a program, and also allows a more modular approach to verification, even in the context of security, as argued in [BFG10].

Verifying C code. But very few widely-used security-oriented programs are written in such high-level languages, and lower-level languages such as C are usually favoured. Several approaches have been proposed for analysing C security protocol code [GP05, ULF06, CD08], but we believe them unsatisfactory for several reasons:

- memory-safety assumptions: all three rely on assuming memory-safety

properties,¹

- trusted manual annotations: all three rely on a large amount of trusted manual work,
- unsoundness: both [CD08] and [ULF06] make unsound abstractions and simplifications, which is often not acceptable in a security-critical context,
- scalability issues: [CD08] is limited to bounded, small in practice, numbers of parallel sessions, and we believe [GP05] is limited to small programs due to its whole-program analysis approach.

1.1 Goals

Our goal is to provide a new approach to soundly verify Dolev-Yao security properties of real C code, with a minimal amount of unverified annotations and assumptions, so that it is accessible to non-experts. We do not aim at verifying implementations of encryption algorithms and other cryptographic operations, but their correct usage in secure communication protocols such as TLS.

2 Framework

Previous approaches to verifying security properties of C programs did not define attacker models at the level of the programming language, since they were based on extracting a more abstract model from the analysed C code (CSur and Aspier), or simply verified compliance of the program to a separate specification (as in Pistachio). However, to achieve our scalability goals, we choose to define an attacker model on C programs, that enables a modular verification of the code.

To avoid issues related to the complex, and often very informal semantics of the C language, we use the F7 notion of a refined module (see [BFG10]). In F7, a refined module consists of an imported and an exported interface, containing function declarations and predicate definitions, along with a piece of type-checked F# code. The main result states that a refined module with empty imported interface cannot go wrong, and careful use of assertions allows one to statically verify correspondence properties of the code. Composition results can also be used to combine existing refined modules whilst ensuring that their security properties are preserved.

We define our attacker model on C programs by translating F7 interfaces into annotated C header files. The F7 notion of an opponent, and the corresponding security results, can then be transferred to C programs that implement an F7-translated header. The type-checking phase in F7 is, in the case of C programs, replaced by a verification phase, in our case using VCC. We trust that VCC is sound, and claim that verifying that a given C program correctly implements a given annotated C header entails that there exists an equivalent (in terms of attacks within our attacker model) F7 implementation of that same interface.

¹Which may sometimes be purposefully broken as a source of randomness.

3 Case Study

We show how our approach can be used in practice to verify a simple implementation of an authenticated Remote Procedure Call protocol, that authenticates the pair of communicating parties using a pre-shared key, and links requests and responses together. We show that different styles of C code can be verified using this approach, with varying levels of required annotations, very few of which are trusted by the verifier. We argue that a large part of the required annotations are memory-safety related and would be necessary to verify other properties of the C code, including to verify the memory-safety assumptions made by previous approaches.

4 Conclusion

We define an attacker model for C code by interpreting verified C programs as F7 refined modules. We then describe a method to statically prove the impossibility of attacks against C code in this attacker model using VCC [CDH⁺09], a general C verifier. This approach does not rely on unverified memory-safety assumptions, and the amount of trusted annotations is minimal. We also believe it is as sound and scalable as the verifier that is used. Moreover, we believe our approach can be adapted for use with any contract-based C verifier, and could greatly benefit from the important recent developments in that area.

References

- [BFG10] 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, Madrid, Spain, 2010.
- [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*, pages 139—152, Washington, DC, USA, 2006. IEEE Computer Society.
- [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.
- [CDH⁺09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michal 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*, pages 23—42, Munich, Germany, 2009. Springer-Verlag.
- [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.

- [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.