Survey of Verified Cryptography

CSCI-762 - Tom Arnold (4/29/20)

1. Overview

Secure crypto-systems are frequently defeated by flaws in their implementation. Software tools can be used to formally verify implementations of algorithms and cryptographic protocols. In this paper we review several classes of implementation issues common to applied cryptography, as well as a programming language (FStar) designed to help software engineers mitigate these issues.

2. Aspects of Verified Cryptography

The three aspects of verified cryptography that we will review in this paper are (1) Memory Safety, (2) Functional Correctness, and (3) Resistance to Side-Channel Attacks.

2.1. Memory Safety

Memory safety is an issue for low-level programming languages like C and C++ which are often used to write high-performance code. These languages are not memory safe because they allow a developer to work with raw pointers, where a mistake could cause memory corruption or other related issues.

A high-profile example of a memory safety issue affecting crypto-related code occurred in 2014 with the OpenSSL Heartbleed vulnerability[1]. This vulnerability occurred because certain versions of OpenSSL did not validate user input and did not erase secrets before freeing memory. Exploiting this bug, attackers were able to extract an unknown amount of secrets including but not limited to private keys, passwords, and confidential information. This kind of vulnerability unfortunately allows an attacker to make an end run around a secure system by gaining access to the private key.

Newer languages like Java have automatic memory management and are generally not vulnerable to memory safety issues. Unfortunately these languages come with certain

[1] The Heartbleed Bug (https://heartbleed.com/)

tradeoffs, like more complex implementations or different performance characteristics which can make them unsuitable as direct replacements for C/C++. Because of this we look for other alternatives to avoid memory safety issues.

2.2. Functional Correctness

Crypto protocols and algorithms are usually specified in the form of IETF (Internet Engineering Task Force) RFC (Request for Comments) documents. These are technical documents[2] that describe how the protocol or algorithm works, and includes pseudo-code and test vectors when applicable to help implementors meet compliance. An example RFC of relevance is RFC8446[3]: The Transport Layer Security (TLS) Protocol Version 1.3. This document defines TLS 1.3 which is the latest version of the protocol that essentially secures the internet (TLS provides the security for HTTPS). For reference this document is around 160 pages long.

After reading and understanding the specification, the challenge becomes implementing it and verifying compliance. Pseudo-code and test vectors can help here, but the implementation will inevitably diverge from the pseudo-code for efficiency reasons and test-vectors often do not include edge cases and other “interesting” behavior; instead what we need is something approaching an exhaustive test suite, but creating one of these by hand is generally infeasible. Useful tools here are property based testing, where the computer generates test cases, and proof assistants, where the computer reasons about the program and can automatically check certain properties. In this paper we primarily look at the latter.

2.3. Resistance to Side-Channel Attacks

Side channel attacks are where an attacker is able to leverage information they gain from observing external properties like runtime or CPU usage in order to deduce secret knowledge from a system. A simple way to explain this is to observe that without looking at your computer, you know when the fan turns on that some software must be doing something expensive.

---

[2] IETF RFCs (https://ietf.org/standards/rfcs/)

[3] https://tools.ietf.org/html/rfc8446

Side channel vulnerabilities can come from several sources. One is explicit optimizations in the code. For example a developer of a big-number library may include special cases to skip steps when one of the operands is zero, but this is unsafe because an external observer may be able to witness this through reduced power usage and infer something about the inputs. Another source of side channel vulnerabilities is the computer hardware itself; for example the Meltdown and Spectre vulnerabilities[4].

One mitigation for side-channel vulnerabilities is secret independence. The basic idea is to avoid branching or optimizations based on secret values, and to stick to constant time operations for all cryptographic code. We will look at this further in the tools section.

3. Tools

The goal of verified cryptography is to create tools to help implement cryptographic algorithms and protocols according to the specification. In the following section we will look at a programming language and a software library developed specifically for this purpose.

3.1. F* and Low* - Programming language to implement specifications and efficient implementations

F* (FStar) is a programming language developed at INRIA and Microsoft Research as part of Project Everest with the ultimate goal of creating a fully verified TLS 1.3 stack[5]. F* is a functional ML-style programming language with an advanced type system, supporting features such as refinement and dependent types. F* is a high-level language which is garbage collected, but a restricted subset of the language called Low* can be compiled to C for inclusion in existing codebases.

The intention behind the F*/Low* split is that F* code can be high-level and terse, while the Low* code gets to use the features of the advanced type system while being able to target an efficient low-level profile[6]. The approach to implementing and verifying an algorithm in F* is basically to implement it first in high-level F* code as a kind of executable specification that can reasonably be hand-verified by comparing it to the spec, and then implementing it again in a more efficient style using the Low* subset of the language. Finally the two implementations

---

[4] Meltdown and Spectre Side-Channel Vulnerability Guidance (https://www.us-cert.gov/ncas/alerts/TA18-004A)

[5] https://project-everest.github.io/

[6] https://fstarlang.github.io/lowstar/html/Introduction.html

are linked using post-conditions that verify that they agree on output for corresponding input. The goal is to provide a chain of verification from hand-verification of the spec and high-level code to computer-assisted verification of the low-level code.

Refinement and dependent types are tools to support functional correctness. Refinement types allow type declarations to be “refined” to a subset of their domain; an example of a refinement type is restricting an integer to just a specific range of values. Dependent types take this one step further by allowing refinements to depend on other types, so for example you could have a buffer type whose length is guaranteed by the compiler to match another parameter to the function. These tools can be used to enforce invariants in a program and could prevent issues like the OpenSSL Heartbleed vulnerability.

F* is intended as a general purpose language for domains where the most important concerns are correctness and performance. It comes with a standard library that is useful in this endeavor. One component of this library is a Buffers module which is used to model secure heap and stack usage for the generate Low*/C code; the idea behind this module is that the heap is treated as a tree of regions instead of a big array of bytes. Another component is the secure integer module which provides constant-time big number support which is primarily useful in the implementation of cryptographic algorithms.

3.2. HACL* - Verified cryptographic algorithms

HACL* is a library of verified cryptographic primitives being developed as the main consumer of the F* language and tools. The library implements a verified version of ChaCha20-Poly1305 which is an AEAD (Authenticated Encryption with Associated Data) algorithm that can be used as an alternative to AES-GCM in TLS 1.3; it’s primary benefit is for devices that lack AES-NI (AES native instruction hardware support) where it offers improved performance over AES[7].

Generally the performance of the algorithms implemented in HACL* and extracted to C code exceeds that of handwritten C in the OpenSSL library. The tradeoff is that more code is written as part of the executable specification and various type system annotations.

4. Conclusion

This is a promising approach, showing concrete results and a reasonable plan to implementing the goal of a fully verified TLS stack. One open question is over maintenance of the verified code; projects are not likely to adopt F* and the related tooling, but at the same time the extracted C code loses its verification properties if it is hand modified after delivery.

---

[7] https://blog.cloudflare.com/do-the-chacha-better-mobile-performance-with-cryptography/

5. Additional References

- HACL*: A Verified Modern Cryptographic Library (https://eprint.iacr.org/2017/536.pdf)
- Verified Low-Level Programming Embedded in F* (https://arxiv.org/pdf/1703.00053.pdf)
- Implementing and Proving the TLS 1.3 Record Layer (https://eprint.iacr.org/2016/1178.pdf)
- Dependent Types and Multi-monadic Effects in F* (https://eprint.iacr.org/2016/1178.pdf)