Modernizing FIPS for safe languages and verified libraries

Jonathan Protzenko, Bas Spitters

May 28, 2024

Abstract

Formal verification has a long 30+ year history of making software safer, more secure, and more reliable. Recently, formal verification has extended its reach from aviation and critical systems to cryptography, and notably cryptographic libraries. Combined with a recent executive order that promotes the use of safe languages and of formal verification, the time is ripe for cryptographic libraries to up their game and start embracing state-of-the-art, verified, secure and safe cryptography.

We argue in this paper that unfortunately, the FIPS standard in its current incarnation hampers the adoption of more modern toolchains and languages. Our position is that many of the current requirements, ranging from self-hashing and power-on self-test (POST), to code reviews, provide little to no benefit, while imposing a tax on the FIPS certification of modern cryptographic libraries. In short, FIPS prevents, rather than fosters, the adoption of better cryptography software.

Fortunately, we believe there are many low-hanging fruits that would allow modernizing the FIPS standard. Specifically, we argue that NIST has an opportunity, in concertation with academia and industry, to draft a new set of standards that would talk about the *security* and *design* of modern cryptographic libraries, and would implement the guidelines provided by the White House in their latest executive order.

1 Introduction

Cryptography used to be implemented in hardware, so one wanted to ensure resistance to physical tampering. Today, most crypto is written in software, and/or relies on hardware accelerators that are themselves not subject to FIPS. For example, Intel does not certify its processors for AES-NI. So, we will need a new focus on *software* security. The Whitehouse executive order has very strong recommendations for secure software[1], in line with the CISA initiative on Security by Design [2]. They both strongly recommend using modern methods

[1]https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/memory-safety-statements-of-support/

[2]https://www.cisa.gov/resources-tools/resources/secure-by-design

from programming language theory, such as memory-safe languages and formal methods.

Moreover, quantum computers necessitate an update of the cryptographic protocol stack. For most NIST-approved post-quantum algorithms there is either a formally verified implementation ready or in production. We thus see a twin transition to post-quantum/high-assurance cryptographic software.

We will argue that the current NIST process hampers the adoption of these modern methods. To do so, we will review several of these requirements, and highlight the difficulties that they pose, both in legacy environments (C and C++) and in modern environments (memory-safe languages), especially Rust. We conclude that FIPS, as it stands, is a roadblock towards modernizing cryptographic libraries, and that beyond the known-answer tests, the CMVP seems to provide little actual benefit. We will conclude by giving constructive recommendations on how the process could be modernized.

2 Legacy FIPS requirements

In this section, we review legacy FIPS requirements. While most of these might make sense in the context of a hardware module, or in previous generations of software stacks, we argue that in the context of a software cryptographic library running on a modern OS, these provide few, if any, guarantees, while imposing undue requirements on cryptographic library authors.

2.1 Self-Hashing Requirement

The self-hashing requirement is perhaps the most decried of all the FIPS provisions. Effectively, it requires the cryptographic module to hash its code segment, and compare it to a known value, to ensure that the code of the cryptographic module has not been tampered with.

Issue #1 (threat model). While this makes sense in the context of a hardware module, we do not understand the threat model that would justify such an integrity check in a software context. If an attacker has write access to the system libraries, then the entire machine is compromised, and self-hashing provides no additional defenses. The attacker might as well edit the binary, recompute the hash, and change the key or expected value. If the threat model is random byte corruption on disk, we have yet to find an instance in which none of the other redundancy mechanisms (parity, block integrity hash) fail to detect and/or correct the issue prior to loading the software library. An architect at a large cloud software provider reported (personal communication) that they have not witnessed a single instance of a failing self-hashing test.

Issue #2 (incompatible with dynamic loading). Most modern cryptographic libraries are shipped as dynamic libraries on target systems. This means that these libraries, when stored on disk, contain relocations to be filled by the dynamic linker. Computing a self-hash for this library that ignores the relocations require jumping through a variety of hoops that require raw memory access and manipulations that add the potential for error, not to mention the added build or complexity [3].

Issue #3 (already done, better, by the OS). Modern OSes (macOS, Windows, Linux, iOS, and Android) all establish provenance of their shared libraries using roots of trust and signature certificates. In other words, there already exist modern provenance mechanisms that would already prevent the library from being loaded, should a malicious adversary try to replace the library with a malicious copy. Unfortunately, because FIPS mandates that a boundary be drawn around the library, these mechanisms fall outside of the scope of certification.

Issue #4 (hampers usage of safer languages). A language like Rust largely relies on static linking. While dynamic linking is possible in Rust, it is non idiomatic and generally not well integrated with the build tools. In order to be FIPS-certified, authors of Rust libraries must either self-hash the entire program (static linking), or use dynamic linking and pay the price of a non-standard style of library.

Furthermore, Rust does not provide access to the raw code segment. In order to execute the self-hashing routine, the programmer must either drop into C, or resort to unsafe Rust code, which undermines the guarantees one gets from using a safe language, and affects the perception of the library by end users (Rust programmers see the absence of unsafe as a badge of honor, especially for security-critical libraries like cryptography).

Issue #5 (encourages dangerous practices). We have received reports that in the context of firmware, which are distributed as one large static binary, in order to obtain FIPS certification, a common strategy is to re-implement a simple version of dynamic loading, so as to have the cryptographic code live in a separate object file, which can then be designated as the cryptographic boundary. This thus encourages poor software engineering practices, and once again, increases the attack surface.

2.2 Power-On Self Test (POST)

The power-on self test, just like self-hashing, aims to ascertain that the library is computing the correct results before booting into the system. While we do not take issue with NIST providing known-answer tests (KATs), which are incredibly useful, we remark that running those tests at boot-time causes several concerns.

Issue #1 (threat model). Same as above.

Issue #2 (performance). According to an Oracle blog post[4], POST incurs up to 9% slowdown at boot-time. According to an internal Microsoft analysis, POST incurs up to 60ms penalty for each process creation for those binaries that statically link the cryptographic library (and thus require re-running the

[3] `https://boringssl.googlesource.com/boringssl/+/master/crypto/fipsmodule/FIPS.md`

[4] `https://blogs.oracle.com/solaris/post/is-fips-140-2-actively-harmful-to-software`

self-test). There seems to be confusion as to whether this test should be run once, or every time the library is loaded.

2.3 Manual code review

Issue #1 (discretionary process). Manual code review seems to be left, to a large degree, to the appreciation of the accredited lab. What this means, in practice, is that deviations from the time-tested, "classic" way of writing code are likely going to pose difficulties, or might require cryptographic library authors to pick their lab carefully. We propose an alternative in Section 4.

3 Modernizing legacy requirements

Suggestion #1 (eliminate POST and self-hashing). We recommend that, for software modules, NIST eliminate POST and self-hashing entirely, for the reasons we argued. These should be replaced by a signing requirement that can guarantee, at the OS-level, that the library being loaded is cryptographically equivalent to the one that was originally FIPS-certified and for which all the KATs passed in the validation program.

Suggestion #2 (clarify code review). As we argued, the uncertainty surrounding manual code review is enough to discourage, say, an open-source verified library from ever attempting the FIPS certification. We suggest that NIST clarify that for code-generating, verification-oriented pipelines, the review should focus exclusively on the source specifications, the implementation refinement theorem, and possibly the strength of the code-generating pipeline.

4 Filling in a void: talking about the *security* of cryptographic libraries

We now focus our recommendations on a broader topic: establishing security guarantees for cryptographic libraries. FIPS, to a large extent, focuses on the *conformance* of cryptographic implementations to government-approved standards. This is fundamentally useful, as it i) drives policy, and provides the US government a lever to mandate the adoption of algorithms that meet minimal requirements, and ii) requires software modules to be tested on a very large matrix of CPUs, systems, OSes and build configurations, which is the specific added value that labs provide.

FIPS, however, says very little about the *security* of cryptographic libraries. One only needs to look at the long trail of bugs of every major (FIPS-certified) cryptographic library to conclude that there is room for improvement in their security.

One might look at Common Criteria, which provides guidelines that go beyond merely passing KATs, and talks about full formal verification for their highest certification level (EAL7). However, Common Criteria only talk about a full, self-contained application that uses cryptographic primitives. In the current state of things, one cannot obtain a CC certification for a cryptographic library; one would have to write a test driver, or a client program, in order to meet the certification requirements. The recent certification [3] of the Belenios cryptographic voting protocol provides some interesting background.

Fortunately, formally verified implementations of most of the cryptographic primitives have recently been produced by toolchains like Fiat-Crypto [4], Jasmin [1], Vale [6], HACL* [7], libcrux [5], etc. They have taken to *producing* C, ASM or Rust code out of formally verified specifications. These toolchains have been adopted in industry by major projects (web-browsers, OSes, messenger apps, voting software, ...). These projects clearly satisfy the requirements of security by design demanded by recent US policy.

To modernize the FIPS certification process, we propose to test the *source* specifications, and validate those, along with a review, based on scientific literature, of the code-producing toolchains.[5] In a sense, the generated code is irrelevant, and should not be the target of FIPS certification; the reference specifications, which the final code is proven to implement, should be subjected to manual review, along with the final theorems that the generated code matches the specification. An added benefit is that this would significantly speed up the review process, since these specifications are typically mathematical, concise and short.

We believe there is an opportunity for NIST to fill this gap, and establish a new set of CC-style assurance levels that would talk specifically about cryptographic libraries. We envision various assurance levels, such as “no formally verified code”, “some implementations formally verified, ”all implementations formally verified“, and ”implementations and high-level APIs formally verified“. Some of these levels would be conditional on upgrading to safe code (verified C, or Rust, Go, Ada, or any other), per the executive order. The highest assurance level would be reserved for a library that has been entirely formally verified, all the way to its top-level APIs.

Furthermore, we believe that such a new certification program should provide additional guidelines, such as:

- proper API design, including uniform error management conventions (a notorious pitfall for users of OpenSSL), internal state copies rather than invalidating the client state, and so on.
- API validation, such as rejecting invalid inputs for Diffie-Hellman or P256.
- industry best practices, such as zeroing out memory.

[5]The French ANSSI, for example, provides precise reviewing guidelines [2] for formally verified code.

5 Conclusion

The current NIST certification process is at odds with recent US policy around memory-safety and security by design. We have listed concrete obstacles and proposed a way forward. We believe this is particularly urgent in the twin transition to post-quantum/high assurance software that is happening in industry and open-source projects. We hope that FIPS can be modernized by removing legacy requirements and adding new CC-style assurance levels to promote formal verification and best practices in the design of crypto libraries. We look forward to working with NIST on that.

Acknowledgments

The authors wish to thank friends and colleagues across various large technology companies and government agencies for feedback, insights and commentary on the FIPS certification process. This paper started with a discussion at HACS, the High-Assurance Cryptographic Software workshop. The authors are grateful to the HACS organizers.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. CCS '17, page 1807–1823. ACM, 2017.

[2] Yves Bertot, Maxime Dénès, Arnaud Fontaine, Vincent Laporte, and Thomas Letan. Requirements on the Use of Coq in the Context of Common Criteria Evaluations, 2020.

[3] Angèle Bossuat, Eloïse Brocas, Véronique Cortier, Pierrick Gaudry, Stéphane Glondu, and Nicolas Kovacs. Belenios: the Certification Campaign. In *SSTIC 2024 - Symposium sur la sécurité des technologies de l'information et des communications*, 2024.

[4] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic: With proofs, without compromises. *SIGOPS Oper. Syst. Rev.*, 54(1):23–30, 2020.

[5] Franziskus Kiefer, Karthikeyan Bhargavan, Lucas Franceschino, Denis Merigoux, Lasse Letager Hansen, Bas Spitters, Manuel Barbosa, Antoine Séré, and Pierre-Yves Strub. HACSPEC: a gateway to high-assurance cryptography. In *RWC23*, 2023.

[6] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. Keeping safe rust safe with galeed. In *Proceedings of the*

37th Annual Computer Security Applications Conference, ACSAC '21, page 824–836. ACM, 2021.

[7] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. CCS '17, page 1789–1806. ACM, 2017.