HAL open science

A Roadmap for High Assurance Cryptography

Harry Halpin

To cite this version:

Harry Halpin. A Roadmap for High Assurance Cryptography. FPS 2017 - 10th International Symposium on Foundations & Practice of Security, Oct 2017, Nancy, France. pp.1-9. hal-01673294

HAL Id: hal-01673294

https://inria.hal.science/hal-01673294v1

Submitted on 29 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Roadmap for High Assurance Cryptography

Harry Halpin[1]

Inria
2 rue Simone Iff 75012 Paris, France
harry.halpin@inria.fr

Abstract. Although an active area of research for years, formal verification has still not yet reached widespread deployment. We outline the steps needed to move from low-assurance cryptography, as given by libraries such as OpenSSL, to high assurance cryptography in deployment. In detail, we outline the need for a suite of high-assurance cryptographic software with per-microarchitecture optimizations that maintain competitive speeds with existing hand-optimized assembly and the bundling of these cryptographic primitives in a new API that prevents common developer mistakes. A new unified API with both formally verified primitives and an easy-to-use interface is needed to replace OpenSSL in future security-critical applications.

Keywords: high assurance cryptography, formal verification, primitives, security API

1 Introduction

Our increasingly digital society critically relies on the security of software systems, ranging from small IoT devices, through personal computers and smartphones to cars and software in critical infrastructure. For each of those systems we *trust* that they perform certain important tasks, but do not show malicious or unpredictable behavior, even when under attack. Typically, the recommended approach to building such trustworthy systems is the following:

1. first defining clear security goals;
2. then identifying the so-called *trusted code base* (TCB) i.e., the part of the software system that is critical to achieving these goals;
3. isolating the TCB from the rest of the code, and implementing well-defined interfaces between the TCB and the rest of the code; and
4. assuring that the code in the TCB (including the interfaces) achieves the security goals.

Unfortunately, almost none of the above-listed *trusted* systems are systematically built according to this recipe; they are historically grown, without a clear separation between TCB and "non-critical" code; often even without a clear definition of security goals, and essentially everywhere without high-assurance software in the TCB. As a consequence, we are realizing that our society has *trust* in *untrustworthy* low-assurance software.

2 The role of cryptography

Cryptography takes a special role in building trustworthy software. While most other parts of typical TCBs —e.g. operating-system kernels, hypervisors, or critical drivers—implement functionality that is not primarily aiming at security, cryptographic software's sole purpose is to achieve security goals that would not be achievable without cryptography. This inherently places cryptographic code inside the TCB and thus *always* requires high assurance of correctness and security of cryptographic software, including interfaces (i.e. APIs) that cannot be misused to violate the security goals.

Although there has been much academic work on the theoretical security of cryptographic protocols, security proofs constructed by theoreticians rely critically on security and functionality assumptions for underlying primitives, i.e., they implicitly require the existence of interfaces to implementations of those primitives that do not violate the assumptions in the proofs. Indeed, if these mathematical assumptions do not match what is offered by real-world implementations, then the proofs do not apply. When this is the case, there are no security guarantees for the software and catastrophe ensues.

Unfortunately, this happens in practice. Despite the high confidence that society has in cryptography to maintain the security and privacy of their transactions, every year we see devastating attacks against widely deployed cryptographic software. Most of those attacks do not break the cryptographic protection in a mathematical sense, but instead exploit weaknesses in how these primitives are used or implemented. Weaknesses in cryptographic software include mistakes in the implementation of cryptographic primitives, and hard-to-detect bugs in the underlying arithmetic, programming interfaces (APIs) that enable (or even encourage) wrong use, subtle (and sometimes less subtle) flaws in the protocol layer, and side-channel vulnerabilities, that allow an attacker to obtain extra information about secret data through, for example, timing.

The most prominent example of an exploitable bug in an implementation was probably the weakness enabling the Heartbleed attack [13], which allowed a remote attacker to read memory content that in many cases contained secret data. Yet even more common than bugs in cryptographic primitive implementations are the incorrect use of these primitives by developers: A recurring problem is the unjustified trust placed by programmers on API developers to provide good random generation routines and to preconfigure the various cryptographic components with secure parameters by default, as seen in numerous examples of security problems in Android applications due to the incorrect usage of cryptography APIs are given in [11] that range from the usage of weak encryption modes to the fixing of initialization vectors and salt parameters that should be freshly sampled at random for each operation.

All these errors are no longer issues of obscure academic debate, but merit front-page news across the globe as the sensitive data of billions of people can be compromised by a single error in a cryptographic library, causing billions in damages. *How can we make sure that the software we trust for the security of our digital society actually* ***is*** *trustworthy?* Although testing is a relatively

cheap way to eliminate many vulnerabilities, but it will never be able to guarantee the absence of vulnerabilities. In fact, the attacks listed earlier all affected cryptographic software that did undergo serious testing before being deployed. Auditing, the process of careful code review by independent experts, typically reveals more vulnerabilities than automated testing but is much more expensive than testing and therefore does not scale—and does not guarantee the absence of vulnerabilities.

Formal verification is the only approach that can *guarantee correctness and security* of cryptographic software and thus establish the confidence society needs in cryptography. The idea of formally verifying cryptographic software has been an active of research for years, but little of the software deployed on a wide scale today comes with any guarantee of correctness or security. What is needed is a comprehensive plan to aim at formally verify cryptographic software deployed in the real-world with a plan to migrate real-world applications from current "low-assurance" (or, in many cases, no-assurance) cryptography to ***high-assurance cryptography*** that provides the guarantees provided by formal proofs of correctness and security of the needed cryptographic primitives but also provides access to them in an "easy to use" API. In other words, the goal should be to replace OpenSSL with formally verified cryptography.

3 Formally Verified and Optimized Cryptgraphic Primitives

Currently, almost no primitives used in real-world deployed software are formally verified due to the speed lost. Cryptographic software is one of the few examples of software that is commonly hand-optimized at the assembly level. The reason for this is that this approach is, at the same time, feasible and worth the effort, because relatively small portions of code are used to encrypt huge amounts of data, perform many key exchanges, compute many signatures, etc. In particular, on busy servers or on battery-powered devices, even small improvements in performance of some core cryptographic routine translate to noticeable improvements in overall system performance or battery life. Consequently, a very active area of research is devoted to optimizing cryptographic software and essentially every serious cryptographic library (especially OpenSSL) contains hand-optimized assembly routines for the most important primitives and target (micro-)architectures. Yet there have been subtle bugs in low-level arithmetic functions that attackers can exploit in this hand-optimized assembly, such as the multiple carry bugs in big-integer arithmetic in OpenSSL [6]. Yet typically, formally verified primitives are not used in real-world deployment because of a performance penalty, as formal verification is done over models of the code using specialized programming environments such as Coq that do not directly translate into running code, and if so, the code is far too slow to be used.

The largest breakthrough so far has been the HACL* library for high assurance cryptography,[1] which was initially focused on Curve 25519 elliptic curves [18].

[1] `https://github.com/mitls/hacl-star`

The HACL* library is now expanding to include popular stream ciphers (Chacha20, Salsa20, XSalsa20), MACs (Poly1305, HMAC), and more. As it can be compiled in a verified manner down to C, allowing the real world use of verified cryptography [16]. This work is fast enough for real world deployment, as shown by the use of HACL* by Mozilla in their NSS library.[2]

The challenge is to provide at the same time high speed *and* high assurance. Another example of software that comes reasonably close is the hand-optimized assembly implementation of X25519 key exchange presented in [3, 4] (after the bug fix), which has been, to a large extent, proven correct [8]. The proof does not cover the full implementation but only the core loop. More importantly, the proof reveals the main issue with formally proving all widely deployed highly optimized crypto software correct: like also many proofs of less optimized cryptographic software, it required serious manual effort and expert knowledge about both the optimization techniques and the tools used for verification. The amount of code annotations needed for verification by far exceed the amount of actual code. This amount of manual effort does not scale to a larger set of relevant primitives, or to an ever-increasing amount of hand-optimized assembly implementations for an ever-increasing set of microarchitectures.

To allow formally verified cryptography to be usable in practice, there is the need for a verified "low level virtual machine" (LLVM) compiler that optimizes code for micro-architectures in a fully-verified manner in order to permit reaching the performance levels of hand-optimized assembly *and* obtaining formally verified implementations. This includes the verification of the primitives needed by almost all applications—symmetric encryption and authentication, hash functions, key exchange, and digital signatures—while maintaining speed comparable to hand-optimized assembly code for each primitive. This is not impossible if a selected group of modern cryptographic primitives is chosen: Many legacy primitives, such as the MD5 and SHA-1 hash functions, can be broken; so there is no reason to create formally verified implementations of these primitives. Further, although RSA-based cryptography is still widely deployed, it is gradually being replaced by more efficient alternatives that are easier to manage and implement, such as elliptic-curve cryptography (ECC). For example, ECC-based key exchange and digital signatures combined with AES-GCM mode are being adopted in many modern cryptographic deployments on the Internet; this trend is led by large companies such as Amazon, Google, and Microsoft. Curve25519 [2], which will be used for key exchange, encryption/decryption and signature/verification, was recently standardized by the IETF and is used in new versions of TLS and Signal.

The way forward for the formal verification community to accomplish these research goals in terms of cryptographic primitives can be done two phases. A first step towards this goal is to produce possibly slow but formally verified reference implementations in the C programming language of a set of core primitives that are used in state of the art protocols like TLS 1.3 or the Signal secure-messaging protocol. The second step would be to move from C reference imple-

[2] `https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57/`

mentations to formally proven cryptographic software that is hand-optimized on the assembly level. In order to avoid the same issue of scalability that previous efforts have been facing, this goal could be achieved producing tools that allow cryptographic engineers to optimize software then obtain a "click-button" verification of correctness; integrated into typical software build environments. This could be done by working with languages such as Jasmin,[3] a formally verified low-level programming language (largely inspired by the Qhasm programming language by Bernstein [1], which is already today used to write highly optimized cryptographic software). In addition to the efficient register allocator and the instruction-by-instruction translation to assembly offered by Qhasm, Jasmin features a formal specification of its semantics, which allows translation of Jasmin code not only to assembly, and input into formal verification tools that can be produce proofs of equivalence using tools such as GFVerif[4] for elliptic curve cryptography between an optimized Jamin implementation and a (verified) CompCert C reference implementation. The tool will aim primarily at proving equivalence of implementations of symmetric primitives such as permutations, block ciphers, or compression functions. This will not come without work, the programmer will be required to annotate code with statements about operations in the underlying finite field; something that sensible programmers already include now as comments in their code.

4 A Developer-Resistant API

The fastest formally optimized cryptographic primitives will still lead to untrustworthy and broken security if they are incorrectly used. A cryptographic API (Application Programming Interface) is used by programmers to access cryptographic primitives and control cryptographic key material as needed in their applications and higher level protocols. Since APIs usually sit between the primitives themselves and their use in applications, secure API design is an important aspect of secure software engineering. However, as shown by the analysis of Android applications in [11], a huge percentage of applications (88%) tend to have errors in their use of cryptographic APIs. Moreover, existing APIs of libraries such as OpenSSL have been shown to be prone to errors,[5] and these errors can be propagated upwards.

Formal verification is just beginning to be applied to the standardization of security API design, and ad-hoc design by committee should be replaced by a design based on formal foundations. A *security API* consists of a set of functions that are offered to some other program that uphold some security properties, regardless of the functions called or the program calling them [5]. For example, one would hope that an API like PKCS#11 that provides access to key material in hardware tokens would prevent any private key material from being tampered with, regardless of the application [10]. These kinds of security properties are

[3] `https://github.com/jasmin-lang/jasmin`
[4] `http://gfverif.cryptojedi.org/`
[5] `https://www.openssl.org/blog/blog/2016/03/01/an-openssl-users-guide-to-drown/`

particularly critical in many applications, and classically security APIs have been studied in the realm of hardware security modules [5] and increasingly in developer-facing APIs such as the W3C WebCrypto API for Javascript [7]. Most early work did not use generalizable formal techniques, but customized each technique for the API at hand [5], although some work allowed the automatic discovery of common errors in key management [15]. Formal modeling has also been used to successfully reveal a number of API-based attacks on standards, including the commercially available tamper-resistant hardware security tokens PKCS#11 [10]. Although a single program may only use one (or a few) APIs, complex systems such as banking operations consist of thousands of applications, with even more calls to multiple APIs. Of these, although some APIs may be standard, other APIs may be hand-crafted by amateurs, and basic errors such as calling deterministic "random" number generators from the programming language are common

API design should not only be based on sound formal foundations, but also from the concrete results from usability studies of APIs [17]. Almost all APIs across programming languages allow users to make common errors, ranging from nonce re-use to failure to randomize initialization vectors [12]. These account for the vast majority of errors in code and the "top errors" in APIs that have recently been collated by Google's Project Wycheproof[6]. Key management is often underspecified in APIs, and is a common source of errors in systems relying for example on PKCS#11 [10] and the WebCrypto API [7], and simply putting the key material in "trusted hardware" such as hardware tokens may end up having little effect, as shown by errors discovered via formal analysis Yubico's YubiHSM.[7] APIs created by standards committees seem to fare no better: implementations of standardized APIs such as PKCS#11 are often susceptible to multiple attacks.[8] Even worse, when APIs such as PKCS#11 and OpenSSL are used in hardware tokens, errors in the API can cause expensive withdrawal of hardware tokens [14].

The API market today is fractured, with the vast majority of even commercial software being bound to OpenSSL (including the embedding of OpenSSL even in hardware tokens) or various programming-language specific cryptographic APIs. Due to the number of bugs, a number of branches of OpenSSL have happened, ranging from Google's BoringSSL to WolfSSL for lightweight embedded systems. However, none of these efforts have been formally verified, and all of them are under the control of some external entity. IPSec libraries used in VPNs such as OpenSwan and LibreSwan are similarly unverified. Naturally, few other companies want to be dependent on the commercial interests and whims of Google's strategy by becoming tied to BoringSSL.

What is lacking is a flexible API with safe defaults for developers that covers core modern cryptographic primitives – cryptographic primitives that themselves are verified. In order to allow easy "drop-in" replacement of OpenSSL, as much

[6] `https://github.com/google/wycheproof`
[7] `https://www.yubico.com/wp-content/uploads/2012/10/Security-Advisory.pdf`
[8] `https://cryptosense.com/the-untold-story-of-pkcs11-hsm-vulnerabilities/`

of the OpenSSL library should be mapped to the new API as possible for upgrading legacy software. Still, access to these primitives by themselves, even for well-known primitives such as AES-CBC, will almost certainly result in developer errors. So for new software and as a recommended developer-facing API, the API will provide "safe" defaults and layers of abstraction to defend the programmer against their own errors, such as preventing the re-use of nonces and randomly initializing initialization vectors. Furthermore, common errors involving key management, such as key generation, rotation, revocation, and wrapping, can also all be dealt with on a level of abstraction that enforces usages boundaries and sensible "defaults" for best practices for key-handling. For example, if a key is generated, the minimum size as recommended by the ECRYPT "Algorithms, Key Size, and Parameters" report will be used.[9] If there is only a limited number of modern cryptographic primitives verified, then finding "safe" defaults for those primitives and building in proper key-handling (for example, to prevent the same keys for being used in signing and encryption) should be possible in a new high-assurance API. In terms of deployment, a three-pronged strategy is needed 1) The older unverified OpenSSL or other API bindings can be removed and replaced with a high assurance API if the cryptographic primitive is supported 2) Application developers that do not have much experience in cryptography can also use a version with simplified primitives that will automatically chose fast, verified algorithms with "safe defaults" for the developer 3) Advanced developers should be able to override all defaults.

5 Conclusion

In order to make high assurance cryptography a reality, two steps need to be taken. First, formally verified primitives must be comparable in speed to hand-optimized assembly on a per-platform basis. This can be done through formally verified C compilation (including C produced from higher-level verified specifications using languages), and per-architecture optimization using a LLVM that can have equivalence proofs to the formally verified specifications. Second, deploying these primitives in actual applications will require an API that can replace OpenSSL for modern applications, and be easier to use than OpenSSL with safer defaults. Furthermore, as new privacy-preserving primitives such as algebraic MACs and post-quantum primitives reach maturity, these new primitives can be formally verified and added to the API.

It should be noted that the task also extends beyond simply replacing the cryptographic primitives, as the correct usage both OpenSSL and any verified replacement requires the verified correct parsing of data formats, as exemplified by the difficult work of "Project Everest" to parse X.509 certificates in its complete reimplementation of TLS.[10] Without at least one usable API featuring formally verified primitives, there is a little chance of moving beyond OpenSSL.

[9] `https://www.cosic.esat.kuleuven.be/ecrypt/csa/documents/D5.2-AlgKeySizeProt-1.0.pdf`

[10] `https://project-everest.github.io/`

Another aspect that formal verification tools can help address is safe memory management, but further work also needs to be done to ensure that sensitive data such as keys are kept in memory for the minimal needed amount of time. Lastly, we are assuming the processor has correctly implemented the LLVM model and that the LLVM has no features outside the LLVM model capable of being used in an attack, and thus more research needs to be done in formally verifying that actual processors match their specifications [9].

The next step is to present the architecture of a library as sophisticated as OpenSSL and for each building-block of the API explain the security goals and how they can be addressed using formal methods, including the verification of their non-trivial composition in higher-level protocol frameworks such as the Noise framework.[11] With such primitives easily usable by an API, formal verification can serve as the foundation for high assurance cryptography.

6 Acknowledgments

Harry Halpin is funded in part by the European Commission H2020 European Commission through the NEXTLEAP Project (Grant No. 6882). Harry Halpin would like to thank Peter Schwabe for many of the original ideas in this paper and for some of the text itself, while taking sole responsibility for any lack of clarity or problems with this paper.

References

1. Daniel J. Bernstein. qhasm: tools to help write high-speed software. `http://cr.yp.to/qhasm.html` (accessed 2017-04-12).
2. Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography – PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, 2006. `http://cr.yp.to/papers.html\#curve25519`.
3. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *LNCS*, pages 124–142. Springer, 2011. see also full version [4].
4. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012. `http://cryptojedi.org/papers/\#ed25519`, see also short version [3].
5. Mike Bond and Ross Anderson. API-level attacks on embedded systems. *Computer*, 34(10):67–75, October 2001.
6. Billy Bob Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. Practical realisation and elimination of an ecc-related software bug attack. In Orr Dunkelman, editor, *Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings*, volume 7178 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2012.

[11] `http://noiseprotocol.org`

7. Kelsey Cairns, Harry Halpin, and Graham Steel. Security analysis of the W3C web cryptography API. In Lidong Chen, David A. McGrew, and Chris J. Mitchell, editors, *Security Standardisation Research - Third International Conference, SSR 2016, Gaithersburg, MD, USA, December 5-6, 2016, Proceedings*, volume 10074 of *Lecture Notes in Computer Science*, pages 112–140. Springer, 2016.
8. Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, , and Shang-Yi Yang. Verifying curve25519 software. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS'14*, pages 299–309. ACM, 2014. `http://cryptojedi.org/papers/\#verify25519`.
9. Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, et al. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):24, 2017.
10. Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal security analysis of PKCS#11 and proprietary extensions. *J. Comput. Secur.*, 18(6):1211–1245, September 2010.
11. Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 73–84. ACM, 2013.
12. Peter Leo Gorski and Luigi Lo Iacono. Towards the usability evaluation of security apis. In *HAISA*, pages 252–265, 2016.
13. The heartbleed bug, 2014. `http://heartbleed.com`.
14. Robert Künnemann and Graham Steel. YubiSecure? Formal security analysis results for the Yubikey and YubiHSM. In Audun Jøsang, Pierangela Samarati, and Marinella Petrocchi, editors, *Revised Selected Papers of the 8th Workshop on Security and Trust Management (STM'12)*, volume 7783 of *Lecture Notes in Computer Science*, pages 257–272, Pisa, Italy, September 2012. Springer.
15. Dennis Longley and Simon Rigby. An automatic search for security flaws in key management schemes. *Computers & Security*, 11(1):75–89, 1992.
16. Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, et al. Verified low-level programming embedded in f. *Proceedings of the ACM on Programming Languages*, 1(ICFP):17, 2017.
17. Alma Whitten and J Doug Tygar. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *Usenix Security*, volume 1999, 1999.
18. Jean Karim Zinzindohoué, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. A verified extensible library of elliptic curves. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*, pages 296–309. IEEE, 2016.