HACSPEC: a gateway to high-assurance cryptography

Franziskus Kiefer
Cryspen

Karthikeyan Bhargavan
Lucas Franceschino
Denis Merigoux
Inria Paris

Lasse Letager Hansen
Bas Spitters
Aarhus University

Manuel Barbosa[1]
Antoine Séré[2]
Pierre-Yves Strub[3]
(1) University of Porto
(2) École Polytechnique
(3) Meta

1. High Assurance Cryptography

Cryptographic libraries lie at the heart of the trusted computing base of modern applications, so any bug in this code can lead to a critical security vulnerability. For example, the OpenSSL library has issued 12 CVEs in 2022 (so far). Testing and fuzzing catch some of these bugs but not all. We follow and advocate an alternative approach, which is to use formal verification to systematically prevent large classes of flaws in cryptographic software [14].

Recent years have seen several landmark results in the formal verification of cryptographic libraries. For example, HACL* [24,27] is written and verified in a proof-oriented programming language called F⋆ [26] and compiled to C code optimized for various platforms; Jasmin [10,11] is a programming language that empowers programmers to directly write efficient fine-tuned assembly code and verify it using the EasyCrypt framework [16]; Fiat-Crypto [19] uses Coq [7] to automatically generate verified C and Rust code for finite-field arithmetic, a crucial component of elliptic curve code.

Each of these projects relies on a different toolchain to formally prove that an optimized implementation of a cryptographic algorithm is memory safe, functionally correct with respect to a high-level mathematical specification, and resistant to some class of side-channel attacks. Remarkably, the verified code they produce is as fast as the corresponding unverified code in libraries like OpenSSL. This attractive combination of high performance and high assurance has led to some of this code being adopted by mainstream projects like Chrome, Firefox, and Linux.

Despite these successes, the secure integration and composition of verified cryptographic components within larger unverified applications remains an open challenge. The first problem is that each verification project uses its own formal specification language (F⋆, EasyCrypt, Coq), making its guarantees and assumptions hard for an application developer to read and understand. Second, each verified implementation presents its own low-level API that is easy to misuse. Third, when verified code is embedded within an application written in an unsafe language like C, any memory safety error in the surrounding unverified code may be used to attack the crypto code, potentially nullifying the formal guarantees of verification.

In this talk, we propose a new approach that closes these gaps by integrating specification and verification within the cryptographic software development workflow.

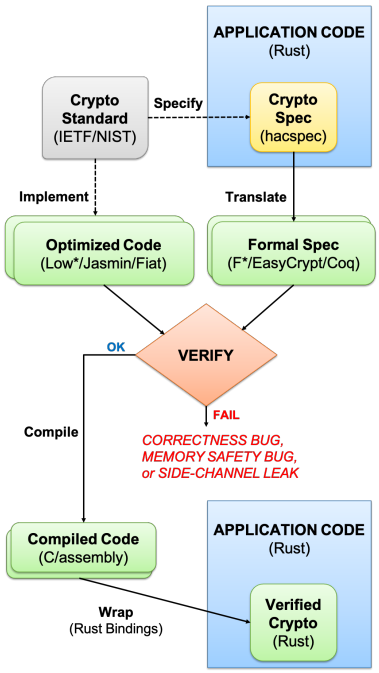

Figure 1. HACSPEC workflow for cryptographic applications in Rust

2. HACSPEC: formal crypto specs in Rust

Our approach (depicted in Figure 1) is built around HACSPEC [23], a new language for writing succinct, executable, formal specifications for cryptographic constructions, which aims to be equally accessible to developers, cryptographers, and verification experts.

Syntactically, HACSPEC is a purely functional subset of safe Rust that disallows mutable references, and hence avoids most of the complexity of stateful Rust programs. HACSPEC is equipped with a library that provides copyable bignums, arrays, and sequences that make it easy to mathematically specify cryptographic schemes at a high level without worrying about low-level representations. We have specified over 15 cryptographic algorithms in HACSPEC, and most specs are under 100 lines of code. Of these, our hash-to-curve spec is used as a reference implementation for the upcoming standard [20].

Writing formal specifications in Rust may seem unusual but has several advantages. First, there is growing enthusiasm for Rust within the cryptographic software community and so the syntax is familiar to both developers

Project	Spec	Code	Algorithms
LIBCRUX	HACSPEC	Rust	High-Level APIs for HMAC, HKDF, HPKE + everything below
HACL* [3]	F*	C	SHA-2, SHA-3, Blake2 Chacha20-Poly1305, P-256, Curve25519, Ed25519
Vale [8]	F*	x86-64	AES-GCM, Curve25519-FF
libjade [4]	jasmin	x86-64	SHA-2, SHA-3, Chacha20, Poly1305, Curve25519
Fiat-Crypto [2]	Coq	C	Curve25519-FF, P-256-FF
AUCurves [1]	Coq	Rust	Ed25519-FF, BLS-FF

TABLE 1. LIBCRUX COMPONENTS
(*-FF DENOTES CODE FOR FINITE-FIELD ARITHMETIC)

and cryptographic engineers. Second, the specifications are executable and hence can be tested against other implementations or published test vectors.[1] Third, the HACSPEC code can directly be used by a Rust application as a prototype implementation of the desired cryptographic component.[2] Finally, by restricting the syntax to a purely-functional subset, it becomes easy to translate HACSPEC to any other formal specification language.

We have developed tools to automatically translate HACSPEC to F* [23], Coq [7], and EasyCrypt [16]. The resulting specifications can then be used to prove properties about the source HACSPEC (for example, that decryption is an inverse of encryption) or can be used as a basis for developing optimized implementations, as depicted in Figure 1. For example, the generated F* specification can be linked to an optimized C implementation in the HACL* library, or the generated EasyCrypt can be used as a specification for a Jasmin assembly implementation. The verified C or assembly implementation is then wrapped within a carefully designed Rust API to create a component that can be swapped into the application as a like-for-like replacement for the original HACSPEC. Importantly, the strong type-safety guarantees of Rust mean that bugs in the unverified application cannot affect the correctness or security of the verified cryptographic code.

The workflow we propose allows application programmers, cryptographers, and verification engineers to smoothly collaborate within the development workflow while maintaining the functionality of the application. Using this approach, we are building a library of verified cryptographic components from different projects distributed under a shared easy-to-understand API.

3. LIBCRUX: composing verified crypto

We present LIBCRUX, the most comprehensive high-assurance cryptographic provider to date, combining verified code from HACL*, Fiat-Crypto, Vale, Jasmin, and AUCurves (see Table 1). LIBCRUX comes with a comprehensive suite of tests (using wycheproof [5]) a continuous integration server, and a benchmarking infrastructure.

1. We use the QuickCheck framework [6] to test other Rust implementations against our HACSPEC specs.

2. A prior verion of HACSPEC was embedded within Python and targeted at cryptographic standards [17], but the use of Python made it hard to integrate the specification within application development.

LIBCRUX takes verified C, assembly, and Rust code from the different projects and combines them behind a defensive Rust API, ensuring that applications respect the assumptions and preserve the guarantees of the verified code. The library automatically picks the best implementation on each platform but also allows fine grained control on which implementation features are enabled.

Each cryptographic construction included in LIBCRUX has a HACSPEC specification which can be used as a prototype implementation when debugging applications. However, the HACSPEC may not be suitable for production use, because of its insufficient performance or lack of side channel protections. So, the library also provides optimized low-level implementations that either replace the full HACSPEC code or just its core computations. For example, the implementations of HMAC and HKDF in LIBCRUX are written in HACSPEC, but they call into verified C and assembly implementations of the underlying hash functions. In each case, we need to prove that the source HACSPEC is equivalent to the the (F*, Coq, EasyCrypt) specification used by the verified implementation.

LIBCRUX is a growing and evolving project. It includes most modern crypto standards, but some equivalence proofs are in progress, and new constructions (e.g. post-quantum KEMs) are being added. Going forward, we see LIBCRUX as a hybrid library consisting of high-level constructions in HACSPEC, low-level primitives in verified C or assembly, and some components written in verified Rust, all sharing HACSPEC specifications and safe APIs.

4. Links to Security Proofs

This talk focuses on verified implementation of cryptography, but our toolchain also enables links between HACSPEC specifications and security proofs for cryptographic constructions and protocols.

We have extended the translation of HACSPEC to Coq to link it with SSProve [9], a Coq library for the verification of cryptographic security proofs in the modular style of state-separating proofs (SSP) [18]. This enables high-level modular proofs of security for cryptographic constructions and protocols written in HACSPEC.

We can also use the translation from HACSPEC to EasyCrypt to develop machine-checked cryptographic security proofs in EasyCrypt. EasyCrypt has previously been used to verify cryptographic standards like SHA-3 [13] and advanced constructions like multi-party computation [12,25]. An extension of EasyCrypt is being used to verify security proofs for post-quantum cryptography [15]. All these proof techniques can now potentially be applied to executable specifications in HACSPEC.

5. Talk Outline

We will present the HACSPEC language [23] using the HPKE standard as a running example [22]. We will describe translations from HACSPEC to F*, Coq, and EasyCrypt, and we will demonstrate the workflow of Figure 1. We will present the first release of LIBCRUX, illustrate its safe APIs, and evaluate its performance. We will conclude with perspectives on future work.

The work we present is based on the HACSPEC technical report [23], which describes the HACSPEC language, type system, libraries and F* translation, on the toolchain [21] translating HACSPEC to Coq, and on the code being actively developed in the libjade [4], HACL* [3], and AUCurves [1] open-source projects.

References

[1] AUCurves - High Assurance Cryptography by means of code synthesis. https://github.com/AU-COBRA/AUCurves.

[2] Fiat-Crypto: Synthesizing Correct-by-Construction Code for Cryptographic Primitives. https://github.com/mit-plv/fiat-crypto.

[3] HACL*: A High-Assurance Cryptographic Library. https://github.com/hacl-star/hacl-star.

[4] libjade: A formally verified cryptographic library written in the jasmin programming language. https://github.com/formosa-crypto/libjade.

[5] Project Wycheproof. https://github.com/google/wycheproof.

[6] quickcheck crate. https://docs.rs/quickcheck/latest/quickcheck/.

[7] The Coq Proof Assistant. https://coq.inria.fr/.

[8] Vale: Verified Assembly Language for Everest. https://github.com/project-everest/vale.

[9] C. Abate, P. G. Haselwarter, E. Rivas, A. V. Muylder, T. Winterhalter, C. Hriţcu, K. Maillard, and B. Spitters. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 1–15, 2021.

[10] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub. Jasmin: High-Assurance and High-Speed Cryptography. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, CCS '17, page 1807–1823, 2017.

[11] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P.-Y. Strub. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In *IEEE Symposium on Security and Privacy (SP)*, pages 965–982, 2020.

[12] J. B. Almeida, M. Barbosa, M. L. Correia, K. Eldefrawy, S. Graham-Lengrand, H. Pacheco, and V. Pereira. Machine-checked ZKP for NP relations: Formally Verified Security Proofs and Implementations of MPC-in-the-Head. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2587–2600, 2021.

[13] J. B. Almeida, C. Baritel-Ruet, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, T. Oliveira, A. Stoughton, and P.-Y. Strub. Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 1607–1622, 2019.

[14] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. SoK: Computer-Aided Cryptography. In *IEEE Symposium on Security and Privacy (SP)*, pages 777–795, 2021. https://eprint.iacr.org/2019/1393.

[15] M. Barbosa, G. Barthe, X. Fan, B. Grégoire, S.-H. Hung, J. Katz, P.-Y. Strub, X. Wu, and L. Zhou. EasyPQC: Verifying Post-Quantum Cryptography. In *ACM SIGSAC Conference on Computer and Communications Security (CCS*, page 2564–2586, 2021.

[16] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology – CRYPTO*, pages 71–90, 2011.

[17] K. Bhargavan, F. Kiefer, and P. Strub. hacspec: Towards Verifiable Crypto Standards. In *Security Standardisation Research*, pages 1–20, 2018.

[18] C. Brzuska, A. Delignat-Lavaud, C. Fournet, K. Kohbrok, and M. Kohlweiss. State separation for code-based game-playing proofs. In *Advances in Cryptology – ASIACRYPT*, pages 222–249, 2018.

[19] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *IEEE Symposium on Security and Privacy (SP)*, pages 1202–1219, 2019.

[20] A. Faz-Hernandez, S. Scott, N. Sullivan, R. S. Wahby, and C. A. Wood. Hashing to Elliptic Curves. IRTF CFRG Internet Draft version 16, 2022. https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-16.

[21] R. Holdsbjerg-Larsen, B. Spitters, and M. Milo. A Verified Pipeline from a Specification Language to Optimized, Safe Rust. CoqPL Workshop, 2022.

[22] F. Kiefer. An Executable HPKE Specification, 2022. https://blog.cryspen.com/blog/hpke_spec/.

[23] D. Merigoux, F. Kiefer, and K. Bhargavan. hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust. Technical report, Inria, Mar. 2021.

[24] M. Polubelova, K. Bhargavan, J. Protzenko, B. Beurdouche, A. Fromherz, N. Kulatova, and S. Zanella-Béguelin. HACLxN: Verified Generic SIMD Crypto (for All Your Favourite Platforms). In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 899–918, 2020.

[25] N. Sidorenco, S. Oechsner, and B. Spitters. Formal security analysis of MPC-in-the-head zero-knowledge protocols. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 1–14, 2021.

[26] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent Types and Multi-Monadic Effects in F*. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270, 2016.

[27] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 1789–1806, 2017.