A Verified Pipeline from a Specification Language to Optimized, Safe Rust

Rasmus Holdsbjerg-Larsen Mikkel Milo Bas Spitters

Concordium Blockchain Research Centre Aarhus (COBRA)

I. Introduction

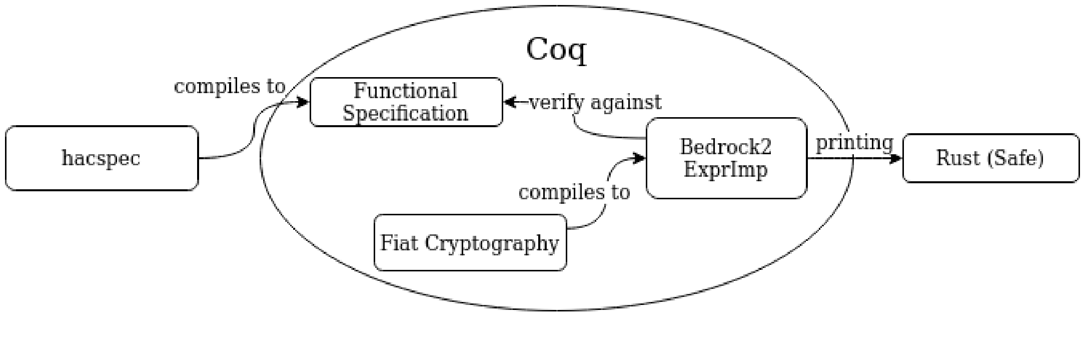
Currently, the IETF standardizes cryptographic primitives using pseudo-code. The correctness of an implementation of such a standard is safe-guarded by unit tests and reference implementations. Hacspec [1], (High Assurance Cryptography Specification language) improves on this by replacing pseudo-code by a subset of rust with a precise operational semantics. Moreover, hacspec comes with a stand-alone type checker and can be translated to both F* and EasyCrypt, which gives hacspec programs also a denotational semantics. We observe that the soundness of the operational semantics wrt. the denotational semantics has not yet been proven, but is expected to be standard. The Rust language does not have a formal specification, even on paper, so hacspec can be seen as a partial formal specification of a subset of Rust.

The goal of hacspec is to serve as a language for writing reference implementations / specifications. Hascspec achieves this as a purely functional language that includes high-level abstractions for mathematical notions commonly used in cryptography such as prime-order fields. As a consequence, low-level optimizations that are critical for efficient implementations are not possible in hacspec. Our approach is to derive, from specifications of primitives written in hacspec, Coq specifications of those primitives, that we then refine into optimized implementations. This is an intended use of hacspec [1].

Similarly, the Fiat Cryptography library [2] allows one to synthesize efficient correct-by-construction implementations (in C, go and java) of field arithmetic from a specification in Coq. In this abstract, we describe a pipeline for standardizing and implementing cryptographic primitives by connecting the two projects and extending Fiat Cryptography to a language rich enough to synthesize code for, for example, elliptic curves.

Fiat Cryptography synthesizes optimized implementations using partial evaluation. This is not easily extended to loops and function calls. Thankfully, Fiat Cryptography compiles to *Bedrock2*[1]. The Bedrock2 project contains a C-like language embedded in Coq, ExprImp, as well as a predicate-transformer semantics specified in terms of a linear separation logic. This not only allows one to verify functional correctness, but also the absence of memory leaks. This combination of Bedrock2 with Fiat Cryptography allows us to synthesize full implementations of cryptographic primitives.

To make sure we synthesize the primitive that fits the standard, we contribute a new Coq back-end for hacspec [1] which allows us to verify ExprImp programs against specifications from hacspec. Finally, we contribute an, as of yet, unverified printer which takes a formally verified ExprImp program and produces *safe* executable Rust. Our printer targets a small subset of safe rust, which includes arrays as the only pointer data structure. Because of this we hope it is feasible to verify the printer in the future.



II. Case Study: The BLS12 pairing groups

As a feasibility study, we consider the group operations on the groups G_1 and G_2 used in the BLS12 digital signature scheme [6]. Hacspec contains specifications of prime order fields, $\mathbb{F}_p$, quadratic extensions of these, $\mathbb{F}_{p^2}$, and pairing groups G_1 and G_2 which depend on these respectively[2]. We compile this entire suite of functions to Coq. The resulting Coq-functions will serve as the concrete specifications against which we will prove our ExprImp programs correct.

[1]https://github.com/mit-plv/bedrock2

[2]This specification was written as part of the bachelor thesis of Kasper Schouborg Nielsen and Mathias Rud Laursen

Functions implementing arithmetic in $\mathbb{F}_p$ are synthesized by Fiat Cryptography and then compiled to Bedrock2. Elements of $\mathbb{F}_p$ are represented as arrays. In Bedrock2 this is done by representing field elements as pointers, and using pre- and post-conditions to state that these pointers actually point to arrays in memory. We use separation logic to reason about pointer data structures. Appel [4] uses a similar simple fragment of VST's separation logic for verifying OpenSSL's implementation of SHA-256. There is no need to reason about function pointer or concurrency, both of which are absent from Bedrock2.

To complete the implementation of extended field arithmetic and of the elliptic curve group operations, we write functions and specifications of higher-level primitives in Bedrock2 and call the base-field functions synthesized by Fiat Cryptography.

A. *Metrics*

To give an estimate of the workload associated with our feasibility study, we divide the project into three parts: theory, tactics and implementations.

The theory part is a high-level Coq formalization of the necessary theory, which includes the theory of quadratic field extensions and a formal proof that numbers represented in multi-limb representations in Montgomery form form a ring.

The tactics part consists of a suite and tactics and auxiliary lemmas used to prove correctness of the Bedrock2 functions. Some are tailored to reasoning about Bedrock2 functions, and some support reasoning about the mathematical constructs such as quadratic field extensions and Montgomery arithmetic. These tactics will be useful for future implementations such as scalar-multiplication and the Miller loop.

The implementations part contain the actual ExprImp functions as well as their specifications and proofs of correctness.

We expect the workload of implementing scalar multiplication to be similar to the present implementation. No high-level formalization of the Miller loop exists yet, so additional effort is needed there.

Part	Theory	Tactics	Implementations	Total
LoC	1784	363	3493	5640

Fig. 1. Lines of code in feasibility study, counted using cloc.

III. Related Work

RustBelt [5] *models* a variant of rust's MIR language in Coq. However, it does not accept either MIR or surface rust code, nor does it produce executable (rust) code. Once RustBelt, or a similar project, moves closer to surface rust, it would be interesting to verify our printer.

The VST project contains CompCert, a formally verified compiler for a subset of C. It would be interesting to make a connection between Bedrock2 and CompCert, although this is not our goal. A primary obstacle in this regard is the strict distinction between pointers and integers made by CompCert; a distinction that is not expressible in bedrock2's uni-typed language.[3]

IV. Acknowledgments

We thank Andres Erbsen and Jade Philipoom for discussions on Bedrock2 and its interface to Fiat Cryptography.
We thank Franziskus Kiefer and Denis Merigoux for discussions around hacspec.

References

[1] K. Bhargavan, F. Kiefer, and P. Strub, "hacspec: Towards verifiable crypto standards", in International Conference on Security Standardisation Research (SSR), ser. LNCS, vol. 11322. Springer, 2018, pp. 1–20.
[2] 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 (S&P). IEEE, 2019, pp. 1202–1219.
[3] A. W. Appel, "Verified software toolchain", in European Symposium on Programming (ESOP), ser. LNCS, vol. 6602. Springer, 2011, pp. 1–17.
[4] A.W. Appel, Verification of a cryptographic primitive: SHA-256. ACM Trans. on Programming Languages and Systems 37, 2 (Apr. 2015), 7:1–7:31.
[5] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. RustBelt: Securing the foundations of the Rust programming language. In PACMPL, 2018.
[6] D. Boneh, B. Lynn, and H. Shacham. "Short signatures from the Weil pairing". In Proceedings of the 7th International Conference on the Theory and Application of Cryptology and information Security: Advances in Cryptology, ASIACRYPT '01, pages 514–532, 2001. Springer.

[3]https://github.com/AbsInt/CompCert/issues/337 https://github.com/mit-plv/bedrock2/issues/78

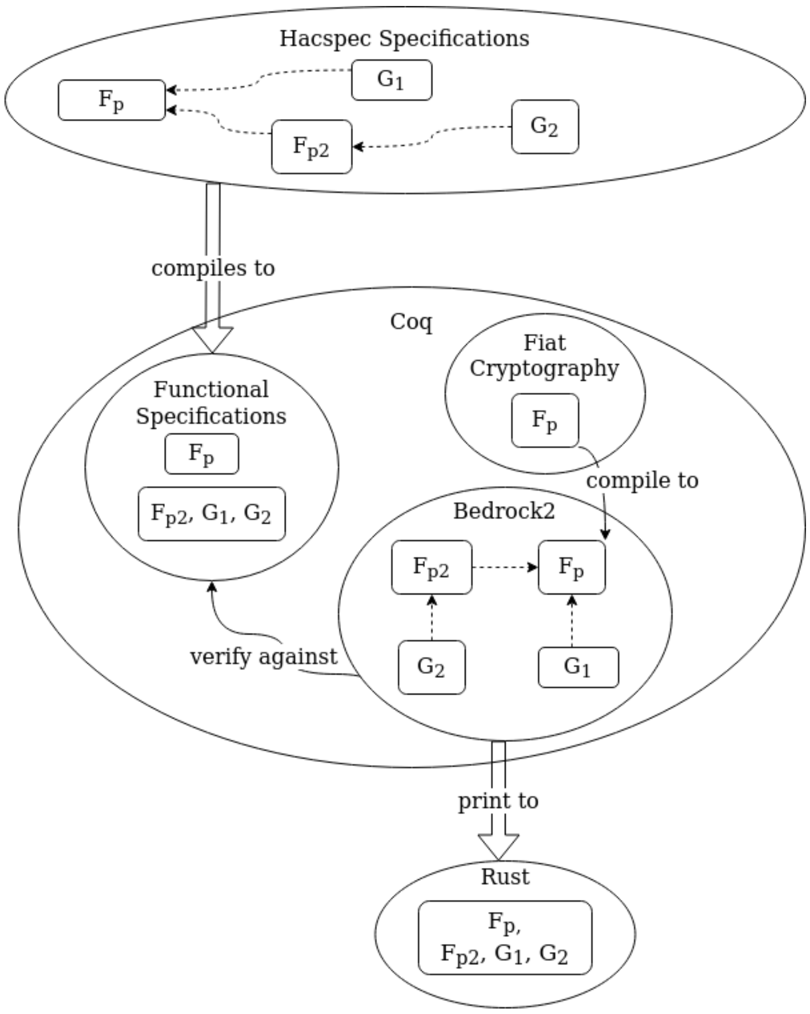


Fig. 2. More detailed version of Fig. I for the case study. Dashed arrows represent function calls.