Hax - Enabling High Assurance Cryptographic Software

Karthikeyan Bhargavan[2], Lucas Franceschino[2], Lasse Letager Hansen[1], Franziskus Kiefer[2], Jonas Schneider-Bensch[2], and Bas Spitters[1]

[1]Aarhus University, [2]Cryspen

1 Introduction

In this talk, we will introduce hax [9], a new toolchain for building and verifying security-oriented software written in Rust. hax translates a subset of safe Rust into the formal languages accepted by proof assistants such as F*, Coq, ProVerif, and SSProve. Backends for other provers, like EasyCrypt, are under construction. Hence, developers can use hax to translate their Rust code to one or more of these backends to prove properties like panic-freedom, functional correctness, protocol-level security, and side-channel resistance. One of the key goals of hax is to build a usable tool for Rust verification that does not force the user to commit to a specific proof environment. Hax extends and expands on the hacspec [11, 10] project, which was presented at RustVerify 2021.

We will present the design of hax and demonstrate its use on a few large applications and case studies. We will showcase how hax supports different backends and their annotations using Rust attributes. We will give examples of its use for both functional correctness and security proofs.
The talk will be for 30 minutes, including a small demo.

2 Hax

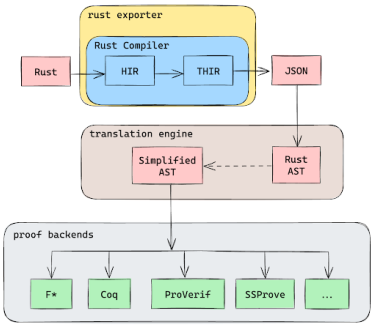

Figure 1: hax toolchain

The hax toolchain [9] allows Rust programmers to verify the correctness and security of their code using a variety of backend provers. The toolchain (Fig 1) consists of three main components:

Rust Exporter. The frontend of the hax toolchain drives the Rust compiler and translates Rust's typed internal representations THIR and MIR to serializable and easy-to-consume abstract syntax trees (ASTs) that are made available via a Rust API or a JSON file.

Rust's internal ASTs are optimized for memory with many levels of indirection that require constant interactive lookups for each identifier. In contrast, our ASTs are self-contained trees packing as much information as possible into each node (e.g. types, attributes, spans), which makes our AST more convenient for verification tools. The Rust Exporter is intended to be usable by other tools outside of hax. For example, the Aeneas [8] project also shares and contributes to this code.

Translation Engine. The main work of hax happens in the translation engine, written in OCaml. The engine consumes the JSON AST corresponding to the THIR representation for any Rust crate (or set of crates). It then translates this AST through a series of typed program transformations, called *phases*, each of which simplifies the code, eventually landing within a sub-language supported by a targeted backend prover. For example, for the F* and Coq backends, the engine uses a dozen different phases, including transforming and functionalizing for-loops; functionalizing local mutation; transforming functions with mutable references as inputs into state-passing code, etc. until it arrives at a purely functional AST that can be seen as an intersection of the purely functional subsets of F* and Gallina. For other backends, such as ProVerif or SSProve [1], which are more imperative in style, we use a different combination of phases.

The translation engine is heavily inspired by our earlier work on hacspec [11, 10], a purely functional subset of Rust that was the first tool to support functional translations from Rust to tools like Coq, F*, and EasyCrypt. hacspec was focused on cryptographic *specifications* in Rust and required programmers to use a highly limited subset of Rust and only use a certain set of hacspec libraries. In contrast, hax is a completely new design that targets a much larger subset of Rust and allows any external library. The primary goal of hax is to target the kind of code used in security-critical applications, like cryptographic libraries, communication protocols, ...

Each phase in the engine is statically typed using a functor style, which reduces the potential for bugs by enforcing inter-phase constraints e.g. an AST that still contains local mutation can never undergo for-loop functionalization.

Proof Backends. The translation engine produces an AST that is as close as possible to the input languages for each backend supported by the toolchain. The final step is a translation from this AST to the concrete syntax of the backend prover. This produces an artifact or model that can be formally analyzed for correctness or security.

Verifying an F*, Coq, or ProVerif model requires a specification of what we would like to prove, and (sometimes) manual annotations to aid the prover. hax supports annotations of the source Rust program with generic pre- and post-conditions as well as in-code assertions, which can be translated to appropriate annotations in the target model. It also supports custom attributes for each backend.

3 Applications

Various projects have used and are using hax for a diverse set of applications and case studies:

The Last Yard. hax was used in [7] to connect a hacspec specification with SSProve and Jasmin [3]. Hax generates a proof that the imperative SSProve translation refines the translation to purely functional Coq. A translation from Jasmin to SSProve further allowed us to formally reason in SSProve about efficient cryptographic implementations in Jasmin. An end-to-end Coq proof of an efficient AES implementation was performed with this methodology.

Libcrux Kyber. hax's F* backend was used to verify correctness of an implementation of the Kyber post-quantum key encapsulation mechanism [5] that has since been adopted by Mozilla for use in its NSS cryptographic library. It has been added to the libcrux library [10].

Smart Contracts. hax was used to prove, for the first time, both functional correctness and cryptographic security of a smart contract, specifically, the Open Vote Network blockchain voting protocol [12]. Functional correctness was proved using ConCert [2] and cryptographic security analysis in SSProve [1]. The same "specify once, verify often" approach was used to analyse functional correctness and security of smart contracts [6].

Bertie TLS. hax is being used to formally analyze the correctness and security of a minimal TLS 1.3 implementation called Bertie [4], using both the F* and ProVerif backends.

References

[1] Carmine Abate et al. "SSProve: A foundational framework for modular cryptographic proofs in Coq". In: *CSF'21*. 2021, pp. 1–15.

[2] Danil Annenkov et al. "Extracting functional programs from Coq, in Coq". In: *Journal of Functional Programming* 32 (2022). DOI: 10.1017/S0956796822000077.

[3] Jasmin Authors. *Jasmin - Language for high-assurance and high-speed cryptography*. 2024. URL: https://github.com/jasmin-lang/jasmin.

[4] Cryspen. *Bertie TLS 1.3 implementation*. 2024. URL: https://github.com/cryspen/bertie.

[5] Cryspen. *Verified ML-KEM (Kyber) in Rust*. 2024. URL: https://cryspen.com/post/ml-kem-implementation/ (visited on 01/16/2024).

[6] Lasse Letager Hansen and Bas Spitters. *Specifying Smart Contract with Hax and ConCert*. CoqPL: The Tenth International Workshop on Coq for Programming Languages. 2024.

[7] Philipp G. Haselwarter et al. "The Last Yard: Foundational End-to-End Verification of High-Speed Cryptography". In: *CCP'24*. ACM, 2024, pp. 30–44. DOI: 10.1145/3636501.3636961.

[8] Son Ho and Jonathan Protzenko. "Aeneas: Rust Verification by Functional Translation". In: *Proc. ACM Program. Lang.* 6.ICFP (2022). DOI: 10.1145/3547647.

[9] Franziskus Kiefer and Lucas Franceschino. *Introducing hax*. 2023. URL: https://hacspec.org/blog/posts/hax-v0-1/ (visited on 10/20/2023).

[10] Franziskus Kiefer et al. "hacspec: A Gateway to High-Assurance Cryptography". Real World Crypto Symposium (RWC) 2023. 2023. URL: https://iacr.org/submit/files/slides/2023/rwc/rwc2023/124/slides.pdf.

[11] Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan. *Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust*. Technical Report. Inria, Mar. 2021. URL: https://inria.hal.science/hal-03176482.

[12] Nikolaj Sidorenco et al. *A formal security analysis of Blockchain voting*. CoqPL: The Tenth International Workshop on Coq for Programming Languages. 2024.