

hax: Verifying Security-Critical Rust Software using Multiple Provers

Karthikeyan Bhargavan¹, Maxime Buyse¹, Lucas Franceschino¹, Lasse Letager Hansen², Franziskus Kiefer¹, Jonas Schneider-Bensch¹, and Bas Spitters²

¹ Cryspen, France

² Aarhus University, Denmark

Abstract. We present **hax**, a verification toolchain for Rust targeted at security-critical software such as cryptographic libraries, protocol implementations, authentication and authorization mechanisms, and parsing and sanitization code. The key idea behind **hax** is the pragmatic observation that different verification tools are better at handling different kinds of verification goals. Consequently, **hax** supports multiple proof backends, including domain-specific security analysis tools like ProVerif and SSProve, as well as general proof assistants like Coq and F*. In this paper, we present the **hax** toolchain and show how we use it to translate Rust code to the input languages of different provers. We describe how we systematically test our translated models and our models of the Rust system libraries to gain confidence in their correctness. Finally, we briefly overview various ongoing verification projects that rely on **hax**.

1 Verifying Security-Critical Software

A software component is deemed security-critical if any bug or design flaw in it could be exploited by an attacker to break the security of the larger system it is a part of. This definition generally includes any code that performs operations whose inputs are partially or completely controlled by the adversary, such as code that processes packets received over an untrusted network, or code that handles an unauthenticated API call. An attacker may use the public-facing interfaces of such components to craft inputs that cause memory errors, break internal code invariants, bypass security mechanisms, and steal secrets through public interfaces or covert side-channels.

Modern software applications typically rely on a number of security-critical components, such as cryptographic libraries, protocol implementations, parsing and sanitization code, authentication and authorization mechanisms, etc. For example, every Web application relies on an implementation of the Transport Layer Security (TLS) protocol [42], which contains cryptography, protocol state machines, message parsing, and X.509 certificate-based authentication. All of this code becomes part of the trusted computing base of the application, and any bug in this code typically result in a high-profile vulnerability and expensive security updates. Consequently, this kind of code is usually separately audited by security experts and comprehensively tested and fuzzed before being deployed.

Formal Verification: Challenges. Given the high cost of failure, security-critical software components would, in principle, be excellent candidates for the high levels of assurance provided by formal verification and machine-checked proofs, but they come with their own unique challenges.

First of all, many security-critical components need to operate with high privileges, e.g. within operating system kernels or deep within web servers, so that they can have direct access to network buffers or to internal security mechanisms. Furthermore, they need to execute efficiently with minimal overhead, both in terms of processing time and memory usage, so that the attacker cannot overwhelm the system with junk inputs. For both these reasons, security-critical components are typically written in low-level languages like assembly or C with many platform-specific optimizations for different target architectures.

Second, these components often build upon advanced cryptographic mechanisms and protocols that require significant domain expertise to program and to analyze. Cryptographic algorithms rely on efficient implementations of mathematical structures like elliptic curves and lattices that are heavily optimized using single-instruction multiple data (SIMD) parallelization on different platforms. Protocol implementations embed complex state machines that interleave cryptographic operations with network actions and parsing code.

Consequently, to verify (say) a typical implementation of TLS, we need tools that can handle a wide range of tasks: we need to prove that its low-level assembly or C code is memory safe, that it is functionally correct with respect to some high-level mathematical specification, and that it meets its security goals against the class of attackers defined by its threat model. Although many verification tools have been developed to address subsets of these tasks, no single tool is suited to handle all of them and verifying large, complex systems remains a big challenge.

Formal Verification: Approaches. A whole field of study, sometimes called computer-aided cryptography [9], is devoted to the formal analysis of cryptographic designs and implementations, using both general-purpose software verification tools and domain-specific proof tools like symbolic protocol analyzers [15,11,18] and computational cryptographic provers [14,10,7,27].

The most successful projects in this area build customized tools for different proof tasks and link them within a single verification framework. For example, the F* verification framework [43] has been used to implement the HACLS* verified cryptographic library [46], to build verified zero-copy binary parsers [41], and to perform cryptographic security proofs for a TLS implementation [20]. The code for all of these is written in a carefully designed subset of F*, verifies using custom proof libraries, and then compiled to low-level languages like C [40] and WebAssembly [39]. Similar projects link verified cryptographic assembly code written in the Jasmin language [3] with high-level security proofs in EasyCrypt [10], or verified C code in Coq [23] with security proofs in SSProve [27], or verified JavaScript code with proofs in ProVerif and CryptoVerif [13].

Code verified using some of these projects have been widely deployed in mainstream software projects like Google Chrome, Mozilla Firefox, Linux, Python,

WireGuard, etc. However, the key to their success, and also their main limitation, is that they are self contained and do not attempt to verify code written by programmers. Instead, all these projects target code written by verification researchers that are then compiled to C or assembly code that can be deployed by regular software developers who never have to see the proofs. Furthermore, the verification itself relies on deep expertise in the tools used and often takes years of effort by teams of researchers. So, while these projects show what can be done, their methods cannot scale to real-world projects driven by developers.

A key roadblock is that although several frameworks are capable of formally verifying security critical C, e.g. [32,5], and assembly, e.g. [3,16,38], however much of the time and effort for verification is usually spent in proving properties like memory safety, leaving little appetite for verifying higher-level correctness and security guarantees. Furthermore, even if one such component is fully verified, the lack of memory safety and isolation in the overall system means that any bug in another (seemingly non-security-critical) C or assembly component can break all our carefully obtained verification guarantees, by accidentally reading or overwriting the memory used by the verified code.

hax: Verifying Secure Rust Code. The advent of memory-safe systems-oriented languages like Rust has made it possible to write high-assurance high-performance code where memory safety for large swathes of code is automatically ensured by the compiler itself, allowing the programmer and reviewer to focus on higher-level properties of the code. For this reason, Rust is starting to be used in many modern security critical projects³, operating systems⁴, and web browsers⁵. Governmental organizations [1], research institutions⁶, and industry bodies⁷ all now heavily promote the use of memory safety languages like Rust.

There is also a vibrant community of formal verification tools for Rust code [21,6,33,45,37,24,29]. Several of these tools explore the edges of the memory safety guarantees of Rust, such as unsafe code blocks and panic freedom. Many tools also support functional correctness reasoning via model checking or SMT solvers or general proof assistants. As yet, none of these tools support security analysis of cryptographic applications. Furthermore, all these tools are still relatively young and only time will tell which techniques will be most effective on real-world software.

In this paper, we present **hax**, a verification framework targeted towards the formal verification of security-critical Rust software. The development of **hax** began with **hacspec** [34], a domain-specific subset of Rust for writing and analyzing *specifications* of cryptographic algorithms. Over time, **hax** has evolved to support the development, specification, and verification of *implementations* of more general security mechanisms written in idiomatic Rust.

³ <https://cryptography.rs/>

⁴ <https://docs.kernel.org/rust/index.html>

⁵ <https://security.googleblog.com/2023/01/supporting-use-of-rust-in-chromium.html>

⁶ <https://www.darpa.mil/program/translating-all-c-to-rust>

⁷ <https://www.memorysafety.org/>

The key features that drive the design of `hax` are:

- **Support for multiple provers**, including general-purpose proof assistants and security-oriented analyzers for cryptographic code;
- **Formal specifications** for correctness and security embedded within the source Rust code and translated to each proof backend;
- **Formal Rust library model** written and specified once in Rust and translated to each proof backend;
- **Programmer-driven verification** that allows the Rust programmer to embed lemmas, annotations, and proofs within the Rust code and keep them consistent as the code evolves;
- **Translation validation via testing** which allows the programmer and verification engineer to execute and test both the Rust code and the generated models in various backends to gain assurance in the correctness of the `hax` engine and library models.

In particular, `hax` does not promote a single verification framework and instead makes it easy to add new proof backends for different target domains. At the same time, `hax` takes charge of the technical tasks of processing and simplifying the input Rust code, modeling the Rust standard libraries, and providing an integrated development and verification environment for Rust developers that scales.

2 `hax`: methodology and workflow

Figure 1 depicts the high-level architecture of the `hax` framework. The programmer provides a Rust crate containing some code and a formal specification for the code written as pre- or post-conditions, invariants, assertions, or lemmas within the source code. The user would typically also provide tests that can be run on the code. When this crate is compiled, the Rust compiler translates the Rust code to assembly, links it with the Rust standard library and any other external crates the user may rely on, and produces an executable that runs the tests.

The first phase of the `hax` toolchain is the `hax` frontend, which plugs into the Rust compiler and uses it to parse and typecheck the source Rust code before producing a fully annotated abstract syntax tree (AST) for the crate as a JSON file. The frontend is capable of producing both the Typed High-Level Intermediate Representation (THIR) and the Mid-Level Intermediate Representation (MIR) of Rust. Since the Rust compiler and its internal data structures evolve fairly rapidly, the frontend takes on the responsibility of keeping track of compiler changes while producing a stable AST that `hax` and even other tools can use. Indeed, the `hax` frontend is being developed in collaboration with the Charon team [28], and is also used by other Rust verification frameworks like Aeneas [29].

The second phase is the `hax` engine, which imports the Rust THIR AST for a crate and transforms it via a sequence of *phases* to a simplified AST that can be

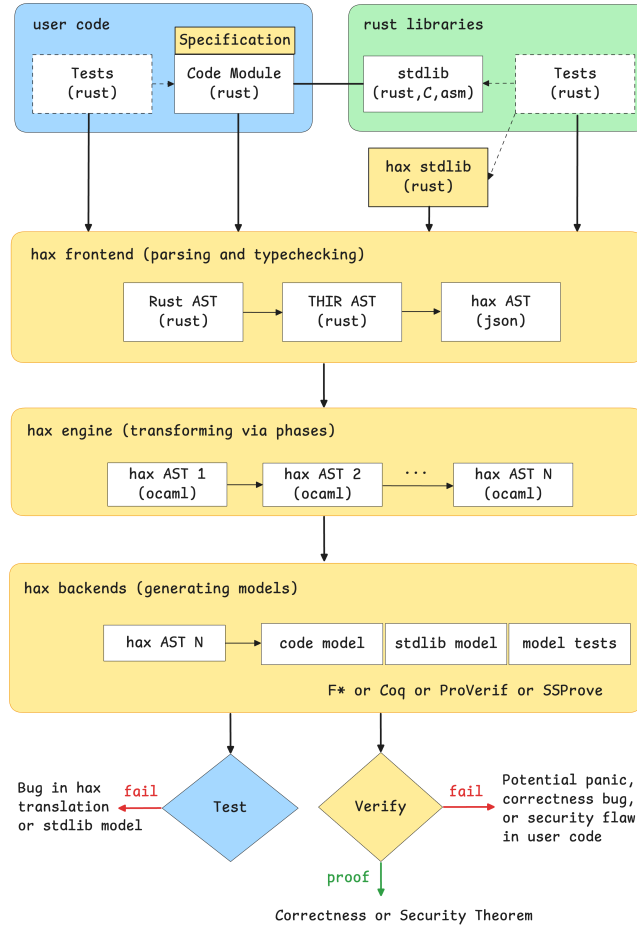


Fig. 1: hax architecture

directly translated to the input languages of various backends. We will describe some of these phases in Section 3.

In the final phase, hax passes on the simplified program to the backend chosen by the programmer. For example, if the programmer chooses F*, the F* backend of hax will generate a purely functional model of the source Rust code and its specification in F*. This model is then linked with F* models of the Rust standard library (and any other external crates) and can be verified for panic freedom and functional correctness against the high-level specification. Completing the proof may require additional annotations, such as loop invariants, or calls to mathematical lemmas. A verification failure may indicate an incomplete proof or a bug in the source code. Other proof backends, such as ProVerif, are completely automated and will either verify the code to produce a security theorem, or generate a counter-example. We describe our current backends in Section 4.

We use `hax` to translate not just the user code, but also handwritten abstract models of the Rust standard library from Rust to various backends. This allows us to model the library once and automatically obtain consistent models for each backend. Modeling the Rust standard library is an incremental, continuous community-driven process. We currently support a few commonly-used libraries, and allow the programmer to extend the library either in Rust or directly in their chosen backend. More details on our model of the Rust libraries are given in Section 5.

Our goal is for all these translations in the `hax` engine and backends to be well-documented and auditable, but we notably do not yet provide formal guarantees for their correctness, which would require us to formalize the semantics of the source Rust and each target language in a proof framework. Instead, we aim to provide pragmatic guarantees based on testing. The generated code for some backends (such as F* and Coq [44]) is executable, so we can compile the tests from the source code to the proof backend and run them to check that the input-output behavior is the same. This gives us additional confidence in the translation and in our model of the Rust standard library. We describe this testing strategy in Section 6.

Several projects are using `hax` to formally verify real-world software. We briefly mention some of these projects in Section 7.

The `hax` project is developed as a community-driven open source project and all our code, libraries, and examples are available online at:

<https://github.com/cryspen/hax>

3 `hax` engine: Transforming and Simplifying Rust Code

The `hax` engine takes as its input the AST produced by the frontend, which is close to the Rust THIR AST, except that all types, trait information, and attributes are inlined. It then performs a series of passes on this AST, called *phases*, that transform the Rust code to a simplified form that is suitable for translation to a proof backend.

3.1 Input Rust AST

Figure 2e presents the input AST in extended Backus-Naur form (EBNF). This figure captures the syntax of Rust as received by the `hax` engine from the frontend. It includes all the familiar constructions from Rust, but does not include features like macros that are eliminated by the Rust compiler.

Literals (`literal`) include strings, integers, booleans, and floating point numbers (although most of our backends do not have any support for floats).

Types (`ty`) include the Rust builtin types: characters, strings, booleans, integers (of size 8, 16, 32, 64, 128 bits or pointer-sized), and floats (16, 32, or 64 bit). They also include composite types such as tuples, fixed length arrays, variable length slices, function types, and named types defined by enums and structs. We currently do not support raw pointer types or dynamic dispatch.

```

string ::= char*
digit ::= [0-9]
uint ::= digit+
int ::= (-)? uint
float ::= int (".")? uint
bool ::= "true" | "false"

local_var ::= ident
global_var ::= rust-path-identifier

literal ::=
| "\" string "\""
| ">" char ">"
| int
| float [d]
| bool

generic_value ::=
| ">" ident
| ty
| expr

goal ::=
| ident "<" (generic_value ",")* ">"

ty ::=
| "bool"
| "char"
| "u8" | "u16" | "u32" | "u64"
| "u128" | "usize"
| "i8" | "i16" | "i32" | "i64"
| "i128" | "isize"
| "f16" | "f32" | "f64" [d]
| "str"
| (ty ",")*
| "[" ty ";" int "]"
| "[" ty "]"
| "*const" ty | "*mut" ty [a]
| "*" expr | "*mut" expr [a]
| ident
| (ty "->")* ty
| dyn (goal)+ [d]

pat ::=
| "_"
| ident "{" (ident ":" pat ",")* "}"
| ident "(" (pat ",")* ")"
| (pat "|")* pat
| "[" (pat ",")* "]" [b]
| "&" pat
| literal
| ("&")? ("mut")? ident ("@" pat)? [c]

modifiers ::=
| ""
| "unsafe" modifiers
| "const" modifiers
| "async" modifiers [a]

guard ::=
| "if" "let" pat (":" ty)? "=" expr

expr ::=
| "if" expr "{" expr "}" ("else" "{" expr "}")?
| "if" "let" pat (":" ty)? "=" expr "{" expr "}" (
  "else" "{" expr "}")?
| expr "(" (expr ",")* ")"
| literal
| "[" (expr ",")* "]" | "[" expr ";" int "]"
| ident "{" (ident ":" expr ";")* "}"
| ident "{" (ident ":" expr ";")* "..." expr "}"
| "match" expr guard "{"
  ((|" pat)* "=" (expr ", " | "{" expr "}")*)
  "}"
| "let" pat (":" ty)? "=" expr ";" expr
| "let" pat (":" ty)? "=" expr "else" "{" expr "}"
  ";" expr
| modifiers "{" expr "}"
| local_var
| global_var
| expr "as" ty
| "loop" "{" expr "}" [e]
| "while" "(" expr ")" "{" expr "}" [e]
| "for" "(" pat "in" expr ")" "{" expr "}" [e]
| "for" "(" "let" ident "in" expr "..." expr ")" "{"
  expr "}" [e]
| "break" expr
| "continue"
| pat "=" expr
| "return" expr
| expr "?"
| "&" ("mut")? expr [c]
| "&" expr "as" "&const _" [a]
| "&mut" expr "as" "&mut _"
| "|" pat "|" expr

impl_item ::=
| "type" ident "=" ty ";"
| modifiers "fn" ident ("<" (generics ",")* ">")?
  "(" (pat ":" ty ",")* ")" (":" ty)? "{" expr "}"

trait_item ::=
| "type" ident ";"
| modifiers "fn" ident ("<" (generics ",")* ">")?
  "(" (pat ":" ty ",")* ")" (":" ty)? "{" expr "}"
  | ";;"

item ::=
| "const" ident "=" expr
| "static" ident "=" expr [a]
| modifiers "fn" ident ("<" (generics ",")* ">")?
  "(" (pat ":" ty ",")* ")" (":" ty)? "{" expr "}"
| "type" ident "=" ty
| "enum" ident ("<" (generics ",")* ">")? "{" (
  ident "(" (ty)* ")"? ",")* "}"
| "struct" ident ("<" (generics ",")* ">")? "{" (
  ident ":" ty ",")* "}"
| "trait" ident ("<" (generics ",")* ">")? "{" (
  trait_item)* "}"
| "impl" ("<" (generics ",")* ">")? ident "for" ty
  "{" (impl_item)* "}"
| "mod" ident "{" (item)* "}"
| "use" path ";";

```

Fig. 2: hax Input Rust AST in EBNF

- (a) no support yet for raw pointers, async/await, static, extern, or union types
- (b) partial support for nested matching and range patterns
- (c) partial support for mutable borrows
- (d) most backends lack support for dynamic dispatch, floating point operations
- (e) some backends only handle specific forms of iterators

Patterns (**pat**) allow matching over the supported types: wildcards, literals, arrays, records, tuples etc. with some limitations in the support for nested patterns and range patterns.

Expressions (**expr**) include literals, variables, type conversions, assignments, array and type constructor applications, and control flow expressions such as conditionals, pattern matches, loops, blocks, and closures. They also include referencing, dereferencing, mutably borrows, and raw pointer operations, although the engine currently does not support raw pointers and only offers limited support for mutable borrows. Specifically, we do not currently support user-written functions that return mutable borrows. Although the engine can handle any kind of loop expression, many of our backends (e.g. ProVerif) have very limited support for loops and so the backend code may impose restrictions on the forms of loops it will accept.

Items (**item**) are the top-level construct in a module and include constants, function definitions, type definitions, trait definitions, trait implementations, modules, and imports. We do not support global static pointers, and we do not model the asynchronicity of **async** functions.

A Rust crate consists of a set of (potentially mutually-recursive) modules, each of which consists of a list of items. A crate may refer to external crates and to the Rust standard library. The engine treats each crate independently: to analyze the crate, we assume that all its dependencies have either been translated or have been modeled by hand for the target backend.

3.2 Transformation Phases

A phase is a typed transformation of AST items: each phase takes a typed AST representing a Rust crate and produces a new typed AST after rewriting some items. The full list of phases implemented by the engine is documented in the source code⁸. Here, we focus on the most important transformations implemented by sets of phases:

- **Order and Bundle Items and Modules.** Rust offers programmers a high degree of flexibility in referencing code and items within and across modules. For example, an item can refer to another item that appears later in the module, or an item within any other module or crate. One can define mutually recursive functions within modules and across modules, but even without recursion, there may be cyclic dependencies between modules. Conversely, most backend proof languages (including all the ones we currently support) allow these kinds of dependencies. Consequently, the engine implements phases that reorder and bundle mutually recursive items so that every item’s dependencies occur before it in the AST. For modules with cyclic dependencies, the engine breaks the cycle by creating a big bundled module with the contents of all the modules in the cycle.

⁸ https://hax.cryspen.com/engine/docs/hax-engine/Hax_engine/Phases/index.html

- **Eliminate Local Mutation.** Rust functions can declare local mutable variables and modify them in conditional and loop expressions, but this kind of mutation is not supported by some backends. The engine contains a phase that eliminates local mutation and replaces it by shadowing. That is, the mutation of a variable `x` gets replaced with a `let` expression that defines a new instance of `x` with the updated value. This transformation is propagated through blocks, loops, and function bodies, so that each expression returns a pair consisting of its original return value and the set of updated values for all mutable variables it modifies. This state-passing transformation is quite straightforward and was also used e.g. in hacspect [34] and Aeneas [29].
- **Eliminate Mutable Borrows.** Each Rust function can have mutably borrowed inputs, mutably borrowed outputs, and local mutable borrows within the function body. The engine implements a transformation that rewrites functions that use mutable borrows as arguments into a state-passing style (in a similar spirit to the elimination local mutation). Conversely, hax has only limited support for functions that create or return mutable borrows. In general, such borrows are only supported as long as they do not create aliases; that is, as long as the mutable borrows are immediately used as function arguments, in which case they are rewritten in a state-passing style.
- **Simplify Control Flow.** Rust programs may contain any combination of conditional, match, and loop expressions, where any deeply nested expression could contain a `return`, `break`, or `continue` which can cause the control flow to jump several layers outwards. Rust also supports the question mark (?) operator that automatically propagates errors out from deep within a function. Most backend provers do not have such expressive control flow, and consequently, the engine implements a set of phases that rearranges expressions so that all these kinds of return expressions are always in leaf position in the control flow graph and so the control flow of each expression is simplified and made explicit in the syntax.
- **Functionalize Iterators.** The Rust compiler desugars all the loop constructions in its surface syntax, such as `for` and `while` loops, into a generic `loop` construction over a generic iterator. The engine implements a phase that propagates the state-passing transformation to loops so that they get transformed into a state-passing `fold` construction that modifies an accumulator at each iteration of the loop. Since proofs about loops often require the most manual intervention, the engine also implements phases that identify common loop patterns and translates them to specialized `fold` constructions. For example, a `for` loop over a range is translated in a way that it is trivial to show that it terminates.

3.3 Choosing and Composing Phases

The hax engine is designed to be modular in that it can be used to execute different sequences of phases to obtain different results. Each phase has a set of preconditions, expressed in terms of features it expects to be present or absent in the input AST, and a post-condition that describes how it changes these features.

These constraints are enforced in the engine using typed OCaml functors and feature variables that together ensure that only sensible compositions of phase transformations can be created.

For each backend, we choose a specific set of phases. For example, to translate Rust code to purely functional models in F* and Coq, we use all the phases described above. ProVerif supports more flexible control flow, so we do not need to perform the control flow transformation. SSProve supports local mutation, and so we do not transform local mutation, while we still use the other phases. Finally, each backend may only have limited support for certain features, like loops or floating point numbers. In these cases, the engine leaves it to the backend to identify and reject code that uses unsupported features.

4 hax backends: Translating Rust to Verifiable Models

Once the hax engine has transformed the input Rust code into a suitable form, we can use the corresponding backend implementation to emit a model in the input language for some prover. The hax backend framework provides a set of convenient libraries that make it easy to add new backends. This includes utilities for formatting the output, mapping locations between the output model and the input Rust source code, and other visualisation and dependency analysis tools that can be shared between backends.

To add a backend, we need to implement rules for translating various syntactic elements (items, expressions, types, etc.) into the corresponding syntax of the target prover. We illustrate how this works for four backends: F*, Coq, SSProve, and ProVerif. Backends for others provers such as EasyCrypt and Lean are currently under development.

4.1 F*

F* [43] is a proof-oriented programming language that has been used to develop verified software for a variety of projects, including cryptography [46], protocols [20], and parsing [41]. Code written in F* can be compiled to OCaml for testing and execution, and some subsets of F* can be compiled to C [40] and WebAssembly [39]. To develop a proof in F*, the user annotates the F* program with assertions, refinement types, invariants, pre- and post-conditions, and lemmas. These are then formally proved using F*'s dependent type system, with the assistance of the Z3 SMT solver [35].

We illustrate the F* backend of hax with an example. Below is a function that implements the Barrett reduction for signed 32-bit integers. This function is taken from a new Rust implementation of the ML-KEM post-quantum cryptographic standard [2] that uses hax for formal verification.

```

1 #[hax::requires((i64::from(value) >= -BARRETT_R && i64::from(value) <= BARRETT_R)]]
2 #[hax::ensures(|result| result > -FIELD_MODULUS && result < FIELD_MODULUS &&
3   result % FIELD_MODULUS == value % FIELD_MODULUS)]
4 pub fn barrett_reduce(value: i32) -> i32 {
5   let mut t = i64::from(value) * BARRETT_MULTIPLIER;

```

```

6      t += BARRETT_R >> 1;
7      let quotient = t >> BARRETT_SHIFT;
8      let sub = (quotient as i32) * FIELD_MODULUS;
9      hax::fstar!(r"Math.Lemmas.cancel_mul_mod (v $quotient) 3329");
10     value - sub
11 }

```

Barrett reduction is a commonly-used algorithm in implementations of modular arithmetic. Here, the function takes an input of type `i32` and performs a series of arithmetic and bitwise operations on it (multiplications, shift-right, addition, subtraction) that implement a modular reduction with respect to the constant `FIELD_MODULUS` (which here is the prime 3329). The reader might wonder why do not directly use the remainder operator of Rust (`%`). The reason is that division and remainder are not constant-time operations—their execution time may depend on the value of their inputs—and hence are vulnerable to side-channel attacks that may the potentially secret input `value`. Indeed, such attacks have been found on similar function in ML-KEM implementations [12].

Panic Freedom. It is also important to remember that while Rust programs are memory safe, they can still panic. In the code above, unless we can prove that every multiplication, addition, and subtraction produces results that are within the target type, the code will potentially panic on some inputs and never return a result. For example, for any input greater or equal to 2147468668 the barrett reduction function above goes out of bounds on line 8 and Rust panics (in debug mode). So, when defining a hax backend, we need to decide whether to generate the model in a way that the programmer must *intrinsically* prove that the code never panics, or to produce a model that may panic and allow the programmer to reason about panics *extrinsically* via lemmas. Different backends may make different choice. In the F* backend we always prove panic-freedom and so ask the programmer to add pre-conditions on the input to ensure the absence of panics.

Correctness Specification. We add a specification to the function in the form of a pre-condition and post-condition. The pre-condition (`hax::requires`) says that the input is within a given range (here $-2^{26} \leq \text{value} \leq 2^{26}$). The post-condition (`hax::ensures`) says that the output computes the signed modulus of the input with respect to the `FIELD_MODULUS`. Proving that the function meets this specification requires a prover that can reason about the mathematical and bitwise operations in the code as well as modular arithmetic.

F* Translation. When we use hax to translate the Rust code above to F*, we obtain the model in Figure 3. There are several notable elements in this translation:

- The Rust compiler elaborates all the type conversions and arithmetic operations to the corresponding library calls, such as `core::convert::from` and `core::ops::arith::neg::neg` and adds the relevant type annotations. These are then translated by the F* backend to the corresponding library functions modeled in F* (e.g. `Core.Convert.f_from`).

```

1 let barrett_reduce (value: i32)
2   : Prims.Pure i32
3   (requires
4     (Core.Convert.f_from #i64 #i32 #FStar.Tactics.Typeclasses.solve value <: i64) ≥
5     (Core.Ops.Arith.Neg.neg v_BARRETT_R <: i64) &&
6     (Core.Convert.f_from #i64 #i32 #FStar.Tactics.Typeclasses.solve value <: i64) ≤
7     v_BARRETT_R)
8   (ensures
9     λresult →
10      let result:i32 = result in
11      result ≥ (Core.Ops.Arith.Neg.neg v_FIELD_MODULUS <: i32) &&
12      result ≤ v_FIELD_MODULUS &&
13      (result %! v_FIELD_MODULUS <: i32) = (value %! v_FIELD_MODULUS <: i32)) =
14   let t:i64 =
15     (Core.Convert.f_from #i64 #i32 #FStar.Tactics.Typeclasses.solve value <: i64) *!
16     v_BARRETT_MULTIPLIER
17   in
18   let t:i64 = t +! (v_BARRETT_R >>! 1! <: i64) in
19   let quotient:i64 = t >>! v_BARRETT_SHIFT in
20   let sub:i32 = (cast (quotient <: i64) <: i32) *! v_FIELD_MODULUS in
21   let _:Prims.unit = Math.Lemmas.cancel_mul_mod (v quotient) 3329 in
22   value -! sub

```

Fig. 3: Barrett Reduction function translated to F* by hax

- The pre-condition and post-condition get translated to the corresponding **requires** and **ensures** clauses in F*.
- All mathematical operations are translated to the *strict* versions of these operations in F* (e.g. +!, −!, *!, >>!) which have pre-conditions stating that their inputs must be within certain ranges to prevent panics.
- Local mutability for the variable `t` (line 6 in Rust) gets translated to variable shadowing in F* (line 18 in Figure 3).

F* Proof. The F* typechecker is able to automatically prove that the code does not panic by using the Z3 SMT solver to reason about the arithmetic operations and their bounds. In fact, it can prove that the function will not panic for any input from -2147468667 to 2147468667 . To prove the post-condition, however, we need to use a mathematical property about modular multiplication called `cancel_mul_mod` in the F* libraries. We inject a call to this lemma within the source Rust code at line 9 and it gets translated to the F* model. With this lemma call, the F* typechecker is able to verify the function.

Backend Features. We have illustrated the F* translation by one example, but more generally, the generated programs in the Pure (i.e. total, terminating, side-effect-free) fragment of the F* language. Since F* is usually more expressive than Rust, most of the translations are straightforward: enums translate to algebraic data types, structs to records, traits to typeclasses, etc. The F* backend includes models for many commonly-used Rust features and libraries, but does not support reasoning about raw pointers or mutable borrows that have not been eliminated by the engine.

4.2 Coq

Coq, recently renamed Rocq, is a fully-featured interactive theorem prover with a rich history and a large user community. Notably, Coq has a small kernel for checking proofs and hence has a much smaller trusted base compared to F* which relies on the correctness of both its typechecker and the Z3 SMT solver.

The Coq backend is very similar to the F* backend, with superficial differences in the notations and libraries used in Coq. By translating Rust code to Coq, we can prove the same kinds of properties as in F* (panic-freedom, functional correctness) but using the tactic-based interactive proof style of Coq. Some examples on the use of the hax Coq backend are given in [26].

4.3 SSProve

The SSProve tool [27] supports computational security proofs about cryptographic constructions, using a technique called State Separating Proofs (SSP) [17]. SSProve is structured as a library within Coq that defines an embedded imperative domain specific language (DSL) that allows mutable local variables, random sampling, and various cryptographic and mathematical operations.

The backend for SSProve follows the same structure as for Coq, except that it produces code within the SSProve DSL, which is restricted to a smaller set of types. Notably SSProve does not support enums and structs, so we need to encode these using tuples and sum types.

Security Proofs with SSProve. To show the use of the SSProve backend, we will go through a simple example also used in the last yard [26]. The example is the classic one-time pad (OTP) construction, implemented in Rust using the XOR operation:

```
1 fn xor(a : u64, b : u64) -> u64 {
2   let x : u64 = a;
3   let y : u64 = b;
4   x ^ y
5 }
```

The SSProve backend translates this Rust function into the following definition in SSProve (within Coq):

```
1 Definition xor (a : both int64) (b : both int64) : both int64 :=
2   xor a b :=
3     letb (x : int64) := a in
4     letb (y : int64) := b in
5     x .^ y : both int64.
```

Next, we model the ideal behavior of this function. That is a purely mathematical formulation of the desired behavior. The idealized function is written by hand in SSProve as follows

```
Definition ideal_xor (a : both int64) (b : both int64) : both int64 :=
  ret_both (is_pure a  $\oplus$  is_pure b)
```

To follow the methodology for state-separating proofs (SSP) [17], we modularize each function into a *package* to isolate its behavior. A *game*, a pair of packages indexed by a Boolean value, is defined from the real and ideal packages

```

Definition IND_CPA_game :=
  fun b => if b then ideal_xor_package else xor_package.

```

Our security statement is: given the above game, it is impossible to find the value of the Boolean, regardless of how you interact with the resulting package. The best you can do is guess. This is called IND-CPA security. In SSProve, this security statement is written as follows:

```

Theorem uncondition_security : ∀ A, Advantage IND_CPA_game A = 0.

```

Linking SSProve with Coq. When proving, it is often useful to have a translation between the imperative SSProve code and the functional Coq code, so that, for example, we can compute functions without needing to interpret the SSProve code, or we can use existing Coq libraries. The SSProve backend automatically generates translations between the generated SSProve and Coq models, along with proofs of equality between the two, allowing the programmer to freely switch between the two backends and safely compose their proofs.

4.4 ProVerif

ProVerif [15] is an automated security protocol verification tool, where protocols are modeled in the applied π -calculus. Given such a protocol model and security goals (such as confidentiality, authentication, privacy) stated as *queries* over the model, ProVerif uses sophisticated algorithms to automatically verify that the protocol satisfies these goals against a large class of *symbolic* or Dolev-Yao adversaries [22]. This threat model is one where the adversary can perform unbounded computation, start and control any number of protocol sessions, read any message sent over the public network, and construct and send messages of any size.

In terms of cryptography, the symbolic model of ProVerif is less precise than the probabilistic computational model used in SSProve: it cannot guess secrets and must treat all cryptographic operations as perfect black boxes. Conversely, this abstraction allows ProVerif to automatically verify a large class of protocols which would require painstaking manual proofs in computational proof backends.

Implementing Protocols. As an example, consider the following Rust function taken from a protocol implementation. Here, the initiator function takes some input keying material (`ikm`) and a pre-shared key (`psk`); it derives an encryption key and initialization vector (`response_key_iv`); it serializes and encrypts this value with the pre-shared key; and it returns the key and a message (`initiator_message`) that must be sent over the public network to the peer.

```

1 pub fn initiate(ikm: &[u8], psk: &KeyIv) -> Result<(Message, KeyIv), Error> {
2   let response_key_iv = derive_key_iv(ikm, RESPONSE_KEY_CONTEXT)?;
3   let serialized_responder_key = serialize_key_iv(&response_key_iv);
4   let initiator_message = encrypt(psk, &serialized_responder_key)?;
5   Ok((initiator_message, response_key_iv))
6 }

```

A protocol implementation typically consists of a list of such functions, each of which either processes or produces a protocol message, using some internal

```

1  letfun proverif_psk__initiate(ikm : bitstring, psk : proverif_psk__t_KeyIv) =
2      let response_key_iv = proverif_psk__derive_key_iv(
3          ikm, proverif_psk__v_RESPONSE_KEY_CONTEXT
4      ) in (
5          let serialized_responder_key =
6              proverif_psk__serialize_key_iv(response_key_iv)
7          in
8              let initiator_message = proverif_psk__encrypt(
9                  psk, serialized_responder_key
10             ) in (initiator_message, response_key_iv)
11             else bitstring_err()
12         )
13     else bitstring_err().

```

Fig. 4: ProVerif Translation of Protocol Initiator

state, cryptographic operations (like `encrypt`) and parsing/serialization functions.

The security goals of the protocol implementation are typically expressed in terms of confidentiality—which variables must remain secret from the adversary—and authentication—which variables must be protected from tampering by unauthorized parties. In the function above, we may wish to ask that `response_key_iv` must remain secret as long as the `psk` is secret, even if the attacker get to read (and tamper) with the `initiator_message` (or any other message sent over the public network).

ProVerif Translation. The Rust function above is translated to a function macro on ProVerif, as depicted in Figure 4. Here, calls to the `derive_key_iv` and `encrypt` functions are translated to calls to our cryptographic library model in ProVerif, where they are modeled using symbolic constructors and destructors.

Serialization and parsing functions, like `serialize_key_iv`, can either be modeled using tuples, constructors, and pattern matching, or the user can abstract them as opaque constructors, depending on the precision of analysis desired.

The translation also shows how certain control-flow constructions in Rust are transformed by the engine and the backend. On lines 2 and 4 of the Rust code, we see the question-mark operator of Rust. This means that the expressions on these lines can return an error and if they do, then the function immediately returns with an error result. These lines are transformed by the hax engine so that they have a more explicit control flow, which is then reflected in the generated ProVerif model, which returns explicit errors when functions fail.

Automated Protocol Security Analysis. To verify security properties on the ProVerif model, we extend the generated model with a verification scenario and security goals as shown below:

```

1  free PSK: proverif_psk__t_KeyIv [private].
2  free SECRET_PAYLOAD: bitstring [private].
3  query attacker(PSK).
4  query attacker(SECRET_PAYLOAD).
5  process

```

6 `Initiator(PSK) | Responder(PSK, SECRET_PAYLOAD)`

Here, `Initiator` and `Responder` are ProVerif processes that call the functions extracted from the Rust code for the two parties in the protocol. Both share a global secret variable `PSK` containing the pre-shared key, and the responder also has a secret payload it encrypts back to the initiator.

The two confidentiality queries ask whether an attacker would be able to obtain the pre-shared key or the secret payload. ProVerif is able to automatically analyze the model and prove that these values are indeed secret. We can also further extend the model and study the security of the protocol with an arbitrary number of keys and payloads, where some pre-shared keys may be compromised, etc. and ProVerif will be able to either prove security or provide a counter-example with a symbolic attack. In some cases, especially where the protocol contains some logical loops or recursive data structures, ProVerif may not terminate and the user would need to encode some abstractions for analysis to terminate.

ProVerif is just one of the many protocol verification tools available in the literature. In the future, one could consider targeting other such verifiers by adding backends for them, or for languages like `SAPIC+` language [18] that unify many such tools under a common syntax.

5 Formal Models for Rust Libraries

Rust programs rely on a number of builtin features and libraries provided by the Rust compiler and the standard libraries: `core`, `alloc`, and `std`.

Primitive types, like machine integers, and operators on them are defined within the compiler. The core library defines a minimal set of features needed by most Rust programs. The alloc library builds on top of core and handles memory allocation and some basic data structures. The std library uses core and alloc to provide a number of data structures.

These libraries are large: core is ~60,000 lines of Rust code (~2300 public functions); alloc is another ~27,500 lines (~800 public functions); and std is ~92,000 lines (~3900 public functions). Not all these libraries are written in Rust; some of them use wrappers around external C and assembly libraries.

To formally verify a Rust program, we must therefore provide models for all its dependencies, including the Rust standard libraries and external third-party crates. Of course, it would be even more desirable to formally verify these external dependencies (see e.g. one ongoing effort to verify std⁹), but even modeling the public functions in these libraries is a mammoth task that requires a incremental community effort.

In the context of `hax`, we need to provide models of the libraries for each backend, which can be both a tedious task and risks creating inconsistencies between different backends. To this end, we employ two strategies towards modeling the Rust libraries. For a minimal set of primitive types and functions, we

⁹ <https://github.com/model-checking/verify-rust-std>

manually write models for each backend in a way that maximally leverages existing libraries and abstractions in that backend. For higher-level libraries, we write models in Rust and compile them using hax itself to generate consistent libraries for each backend.

Hand-written Models for Primitive Types. Many types and functions that are primitive to Rust still need to be mapped to the corresponding types and constructions in various backends. This includes:

- machine integers (e.g. `u8`, `i16`, etc.), booleans, strings
- slices and arrays (`[T]`, `[T; N]`)
- options, results, and panic
- iterators (`loop`, `map`, `enumerate`, etc.)

For each backend we need to manually write the translation of these primitives; see figure 5 for how some of them are mapped in the Coq backend.

```

fn primitives() {
  // bool
  let _: bool = false;
  let _: bool = true;

  // Numerics
  let _: u8 = 12u8;
  let _: u16 = 123u16;
  let _: u32 = 1234u32;
  let _: u64 = 12345u64;
  let _: u128 = 123456u128;
  let _: usize = 32usize;

  let _: i8 = -12i8;
  let _: i16 = 123i16;
  let _: i32 = -1234i32;
  let _: i64 = 12345i64;
  let _: i128 = 123456i128;
  let _: isize = -32isize;

  let _: f32 = 1.2f32;
  let _: f64 = -1.23f64;

  // Textual
  let _: char = 'c';
  let _: &str = "hello world";
}

Definition primitives '(_ : unit) : unit :=
  let _ : bool := (false : bool) in
  let _ : bool := (true : bool) in

  let _ : t_u8 := (12 : t_u8) in
  let _ : t_u16 := (123 : t_u16) in
  let _ : t_u32 := (1234 : t_u32) in
  let _ : t_u64 := (12345 : t_u64) in
  let _ : t_u128 := (123456 : t_u128) in
  let _ : t_usize := (32 : t_usize) in
  let _ : t_i8 := (-12 : t_i8) in
  let _ : t_i16 := (123 : t_i16) in
  let _ : t_i32 := (-1234 : t_i32) in
  let _ : t_i64 := (12345 : t_i64) in
  let _ : t_i128 := (123456 : t_i128) in
  let _ : t_isize := (-32 : t_isize) in

  let _ : float := (1.2%float : float) in
  let _ : float := ((-1.23)%float : float) in

  let _ : ascii := ("c"%char : ascii) in
  let _ : string := ("hello world"%string : string) in
  tt.

```

Fig. 5: Primitives translated to Coq

A key requirement for these hand-written models is that they must be executable, so that we can run and test both these libraries and the code that uses them. Of course, we also need these models to be suitable for verification, and so we often extend these libraries with all the necessary lemmas and tactics to help the user prove properties about their programs.

Generating Library Models from Rust. For most libraries in `core`, `alloc`, and `std`, we advocate writing models of the library directly in Rust and compiling these models to each backend.

In effect, we build a new version of these libraries, layered on top of the Rust standard libraries, but shadowing the namespaces so that we can link them to unmodified Rust code. For example, we implement the `Add` trait in `core::ops`, as a new `hax-core::ops::Add`, and translate it via `hax` to obtain models of `core::ops` in each backend.

To implement traits like `Add` generically for all machine integers in Rust, we first build an architecture for the mathematical interpretation of rust types. We define a Rust library for mathematical integers (represented by the type `HaxInt`), and for each machine integer of type `T`, we define a method `lift()` that computes its underlying integer (`HaxInt`) and a method `lower()` that casts a mathematical integer into the machine integer (if it is within bounds, and panics otherwise).

This notion of abstracting (or lifting) and concretizing (or lowering) Rust data types into mathematical structures is generally useful for writing formal models in Rust and we systematically use it in our library models.

We can now specify libraries like `core::num` and `core::ops` directly in Rust, by lifting the inputs to mathematical integers, doing the operations on `HaxInt` and lowering the result back to machine integers. For example, the equality operation on `u8` is defined in Rust as an implementation of the `PartialEq` trait. We model it in Rust as follows (using a type wrapper `U8`):

```
1 impl<'a> PartialEq for U8<'a> {
2   fn eq(&self, rhs: &Self) -> bool {
3     compare_fun(self.clone().lift(), rhs.clone().lift())
4     == Ordering::Equal
5   }
6 }
```

This then gets translated to each backend using the definitions of `lift`, `lower`, and mathematical integers in that backend. For example, the Coq translation is as follows. The trait implementation translates to a typeclass instance that operates on Coq integers.

```
1 Instance t_PartialEq_774173636 : t_PartialEq ((t_U8)) ((t_U8)) :=
2 {
3   PartialEq_f_eq := fun (self : t_U8) (rhs : t_U8) =>
4     PartialEq_f_eq
5       (haxint_cmp
6         (Abstraction_f_lift (Clone_f_clone (self)))
7         (Abstraction_f_lift (Clone_f_clone (rhs))))
8     (Ordering_Equal);
9 }.
```

The F* implementation is similar, while in ProVerif, all machine integers are modeled as mathematical integers, so lifting and lowering are identity functions.

Mixing the Two Styles. For each library, we always have the choice between using the automatically generated model or manually writing models for different

backends. Where possible, we prefer generated libraries, since they require less work and keep libraries consistent between different backends. However, in some cases we may want to exploit some data structure or proof library that is available in a specific backend. In such cases, we often start with the generated library and then edit it to exploit features of the backend. For example, in the Coq translation above we could replace `haxint_cmp` with the comparison operation in Coq, which might result in simpler proofs.

6 Testing the Generated Models

The `hax` toolchain implements a sequence of translations from Rust to various formal languages. There are many ways of gaining confidence that the models generated by `hax` correctly capture the semantics of the input Rust code.

One could formalize the semantics of the source and target languages and prove that the translation preserves the observable behaviors of the program. This kind of proof effort can be valuable but requires significant effort and is less feasible for frameworks like `hax` that support multiple, widely different backends.

Instead, we take a more pragmatic approach of using a mixture of testing and proof to get more assurance in our methodology.

Verifying Library Annotations. For each function in the Rust library, our library models provide pre- and post-conditions that specify whether and when these functions may panic and what they compute. For the core library functions, we also add specification of various useful properties, and prove these properties for our library models. When generating library models, we can add these lemmas in the Rust source so that they are reflected in all backends.

A simple example is commutativity of addition:

```
#[hax_lib::lemma]
fn add_comm(x: u8, y: u8) -> Proof<{ x + y == y + x }> {}
```

This generates a lemma that must be proven for each backend library.

We have added such lemmas for associativity, commutativity, distributivity, negation, etc. for various combinations of arithmetic and bitwise operators for various numerical types. We define similar lemmas about concatenation and slicing of arrays and slices. These lemmas gives us more confidence that the annotations we use for our proofs are sound with respect to our library models.

Testing Source Annotations. In addition to proving lemmas about source code (and library) annotations, we can also use these annotations to drive property-based tests. We systematically use the `QuickCheck` [19] framework to automatically generate tests based on the pre- and post-conditions on the Rust source code. In particular, this technique is used to generate hundreds of tests for each function in our Rust standard library model, including our models for each arithmetic operation.

Testing Generated Models. An important feature of the many `hax` backends is that the generated models are executable, and hence testable. So, when we

compile some Rust code to (say) F*, we also compile its tests and run them on the generated F*. This gives us confidence in the `hax` translation and in our (executable) library models.

For example, [26] presents a reference implementation of the AES cryptographic algorithm in Rust, and shows how it can be compiled via the `hax` toolchain to SSProve. We test this AES implementation in both Rust and in Coq/SSProve to prove that the encryption and decryption produce the same result in the source code and target model.

Linking Different Models. Another way of gaining confidence in our translations is to formally link the models produced via independent translations. For example, our SSProve backend actually consists of two translations. A functional translation, which is very close to the Coq backend (but uses a smaller universe of types); and an imperative translation with state, making use of the domain specific language (DSL) for code in SSProve. The translations are combined into language constructions, with a projection to each of the translations and a proof of equality between them [26]. In a sense the main difference between the two translation is that one of them uses a few extra functionalization phases, so this proof can be seen as a proof of correctness for those phases.

7 Verifying Rust Applications with `hax`

The `hax` verification framework is used by several projects for the specification and verification of security and correctness properties. In this section, we give a brief overview of some of these applications.

hacspec. `hacspec`¹⁰ is a purely functional subset of Rust that can be used, together with a specification library, to write succinct, executable, and verifiable specifications in Rust, that can then be translated into various formal languages using `hax`. It has been proposed as a general specification language for IETF and NIST standards [8].

The `hacspec` language has recently also been adopted by Crux-Mir [37], a cross-language verification tool for Rust and C/LLVM. Crux-MIR has been used to verify the Ring library implementations of SHA-1 and SHA-2 against their `hacspec` specifications.

Libcrux. The `libcrux` library [31] provides a uniform API for formally verified cryptographic implementations in Rust, C, and assembly. It uses `hacspec` to specify the correctness of its implementations and presents a safe, defensive Rust API to applications. Recently, the post-quantum key encapsulation mechanism ML-KEM [36] was added to `libcrux`¹¹. It was verified using `hax` and its F* backend. This implementation has since been adopted by OpenSSH and by Mozilla for use in its NSS cryptographic library.

¹⁰ <https://hacspec.org>

¹¹ <https://cryspen.com/post/ml-kem-implementation/>

In [30], `hax`’ Coq backend is used to connect the Fiat-cryptography [23] verified compiler for finite field arithmetic in Coq. In this way, a simple specification/reference implementation in `hacspect` can be compiled to a highly optimized implementation in many C-like languages, such as C, Rust, Java, etc. This code has also been integrated into `libcrux`.

Bertie. `hax` is used to extract a ProVerif model from the TLS 1.3 implementation in Bertie¹² to perform a symbolic security analysis¹³. `hax` is also used to compile the parsing and serialization code of Bertie to F^* in order to prove panic freedom and functional correctness.

Smart contracts. Rust is a popular smart contract language, as it allows one to efficiently compile to Wasm which is a popular on-chain virtual machine. In [25], `hax` has been used to verify properties of Rust smart contracts using the ConCert smart contract verification framework [4] in Coq. This is combined with cryptographic proofs in SSProve.

8 Conclusion and Future work

We have presented `hax`: a developer-oriented framework for verifying security critical Rust code. Verification can be done in a wide spectrum of proof backends, ranging from tools for generic program verification (F^* and Coq) to symbolic protocol analyzers (ProVerif) and provers for computational cryptography (SSProve). We use a combination of testing and proving to gain assurance that our specifications, translations, and library models are correct. The `hax` toolchain is being used in many active projects, both in industry and academia.

The design is `hax` makes it compatible and extensible with other proof methodologies and backend provers. The `hacspect` language is used in Crux-Mir, the `hax` frontend is used in Aeneas, and the specifications used in `hax` are compatible with Kani and Creusot. Moreover, our backend framework makes it easy to add new backends. In future work, we would like to add new backends for EasyCrypt and Lean, as well as explore fully automated tools for verifying generic Rust code.

References

1. Back to the building blocks: a path towards secure and measurable software. <https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf> (Feb 2024)
2. Module-Lattice-Based Key-Encapsulation Mechanism Standard (Aug 2024). <https://doi.org/10.6028/NIST.FIPS.203>
3. Almeida, J.B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., Strub, P.: Jasmin: High-assurance and high-speed cryptography. In: CCS. pp. 1807–1823. ACM (2017)

¹² <https://github.com/cryspen/bertie>

¹³ <https://cryspen.com/post/hax-pv/>

4. Annenkov, D., Nielsen, J.B., Spitters, B.: Concert: a smart contract certification framework in coq. In: CPP. pp. 215–228. ACM (2020)
5. Appel, A.W.: Verified software toolchain. In: Goodloe, A., Person, S. (eds.) NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3–5, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7226, p. 2. Springer (2012). https://doi.org/10.1007/978-3-642-28891-3_2
6. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust types for modular specification and verification. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). vol. 3, pp. 147:1–147:30 (2019)
7. Baelde, D., Delaune, S., Jacomme, C., Koutsos, A., Lallemand, J.: The squirrel prover and its logic. ACM SIGLOG News **11**(2), 62–83 (2024)
8. Barbosa, M., Bhargavan, K., Kiefer, F., Schwabe, P., Strub, P., Westerbaan, B.: Formal specifications for certifiable cryptography (2024)
9. Barbosa, M., Barthe, G., Bhargavan, K., Blanchet, B., Cremers, C., Liao, K., Parno, B.: Sok: Computer-aided cryptography. In: SP. pp. 777–795. IEEE (2021)
10. Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., Strub, P.: Easy-crypt: A tutorial. In: FOSAD. Lecture Notes in Computer Science, vol. 8604, pp. 146–166. Springer (2013)
11. Basin, D.A., Cremers, C., Dreier, J., Sasse, R.: Tamarin: Verification of large-scale, real-world, cryptographic protocols. IEEE Secur. Priv. **20**(3), 24–32 (2022)
12. Bernstein, D.J., Bhargavan, K., Bhasin, S., Chattopadhyay, A., Chia, T.K., Kan-wischer, M.J., Kiefer, F., Paiva, T., Ravi, P., Tamvada, G.: KyberSlash: Exploiting secret-dependent division timings in kyber implementations. Cryptology ePrint Archive, Paper 2024/1049 (2024), <https://eprint.iacr.org/2024/1049>
13. Bhargavan, K., Blanchet, B., Kobeissi, N.: Verified models and reference implementations for the TLS 1.3 standard candidate. In: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22–26, 2017. pp. 483–502. IEEE Computer Society (2017). <https://doi.org/10.1109/SP.2017.26>
14. Blanchet, B.: Cryptoverif: Computationally sound mechanized prover for cryptographic protocols. In: Dagstuhl seminar “Formal Protocol Verification Applied. vol. 117, p. 156 (2007)
15. Blanchet, B.: Automatic verification of security protocols in the symbolic model: The verifier proverif. In: FOSAD. Lecture Notes in Computer Science, vol. 8604, pp. 54–87. Springer (2013)
16. Bond, B., Hawblitzel, C., Kapritsos, M., Leino, K.R.M., Lorch, J.R., Parno, B., Rane, A., Setty, S.T.V., Thompson, L.: Vale: Verifying high-performance cryptographic assembly code. In: Kirda, E., Ristenpart, T. (eds.) 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16–18, 2017. pp. 917–934. USENIX Association (2017), <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond>
17. Brzuska, C., Delignat-Lavaud, A., Fournet, C., Kohbrok, K., Kohlweiss, M.: State separation for code-based game-playing proofs. In: Advances in Cryptology – ASIACRYPT 2018. p. 222–249. Springer (2018). https://doi.org/10.1007/978-3-030-03332-3_9
18. Cheval, V., Jacomme, C., Kremer, S., Künnemann, R.: SAPIC+: protocol verifiers of the world, unite! In: USENIX Security Symposium. pp. 3935–3952. USENIX Association (2022)
19. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: ICFP. pp. 268–279. ACM (2000)

20. Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Protzenko, J., Rastogi, A., Swamy, N., Béguelin, S.Z., Bhargavan, K., Pan, J., Zinzindohoue, J.K.: Implementing and proving the TLS 1.3 record layer. In: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017. pp. 463–482. IEEE Computer Society (2017). <https://doi.org/10.1109/SP.2017.58>
21. Denis, X., Jourdan, J.H., Marché, C.: Creusot: a foundry for the deductive verification of rust programs. In: International Conference on Formal Engineering Methods. pp. 90–105. Springer (2022)
22. Dolev, D., Yao, A.C.: On the security of public key protocols. *IEEE Trans. Inf. Theory* **29**(2), 198–207 (1983)
23. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In: IEEE Symposium on Security and Privacy. pp. 1202–1219. IEEE (2019)
24. Gäher, L., Sammler, M., Jung, R., Krebbers, R., Dreyer, D.: Refinedrust: A type system for high-assurance verification of rust programs. *Proceedings of the ACM on Programming Languages* **8**(PLDI), 1115–1139 (2024)
25. Hansen, L.L., Spitters, B.: Specifying smart contract with hax and concert. In: CoqPL (2024), <https://popl24.sigplan.org/details/CoqPL-2024-papers/9/Specifying-Smart-Contract-with-Hax-and-ConCert>
26. Haselwarter, P.G., Hvass, B.S., Hansen, L.L., Winterhalter, T., Hritcu, C., Spitters, B.: The last yard: Foundational end-to-end verification of high-speed cryptography. In: CPP. pp. 30–44. ACM (2024)
27. Haselwarter, P.G., Rivas, E., Muylder, A.V., Winterhalter, T., Abate, C., Sidorenco, N., Hritcu, C., Maillard, K., Spitters, B.: Ssprove: A foundational framework for modular cryptographic proofs in coq. *ACM Trans. Program. Lang. Syst.* **45**(3), 15:1–15:61 (2023)
28. Ho, S., Boisseau, G., Franceschino, L., Prak, Y., Fromherz, A., Protzenko, J.: Charon: An analysis framework for rust (2024), <https://arxiv.org/abs/2410.18042>
29. Ho, S., Protzenko, J.: Aeneas: Rust verification by functional translation. *PACM PL* **6**(ICFP) (2022). <https://doi.org/10.1145/3547647>
30. Holdsbjerg-Larsen, R., Spitters, B., Milo, M.: A verified pipeline from a specification language to optimized, safe rust. In: CoqPL’22 (2022), <https://popl22.sigplan.org/details/CoqPL-2022-papers/5/A-Verified-Pipeline-from-a-Specification-Language-to-Optimized-Safe-Rust>
31. Kiefer, F., Bhargavan, K., Franceschino, L., Merigoux, D., Hansen, L.L., Spitters, B., Barbosa, M., Séré, A., Strub, P.Y.: HACSPEC: a gateway to high-assurance cryptography. *RealWorldCrypto* (2023)
32. Kroening, D., Schrammel, P., Tautschnig, M.: CBMC: the C bounded model checker. *CoRR* **abs/2302.02384** (2023). <https://doi.org/10.48550/ARXIV.2302.02384>
33. Lehmann, N., Geller, A.T., Vazou, N., Jhala, R.: Flux: Liquid types for rust. *Proceedings of the ACM on Programming Languages* **7**(PLDI), 1533–1557 (2023)
34. Merigoux, D., Kiefer, F., Bhargavan, K.: Hacspecc: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust. Technical report, Inria (Mar 2021), <https://inria.hal.science/hal-03176482>
35. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest,

- Hungary, March 29–April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
36. NIST: Module-lattice-based key-encapsulation mechanism standard. Tech. Rep. Federal Information Processing Standards Publications (FIPS PUBS) 203, U.S. Department of Commerce, Washington, D.C. (2024). <https://doi.org/10.6028/NIST.FIPS.203>
 37. Pernsteiner, S., Diatchki, I.S., Dockins, R., Dodds, M., Hendrix, J., Ravich, T., Redmond, P., Scott, R., Tomb, A.: Crux, a precise verifier for rust and other languages. arXiv preprint arXiv:2410.18280 (2024)
 38. Polyakov, A., Tsai, M., Wang, B., Yang, B.: Verifying arithmetic assembly programs in cryptographic primitives (invited talk). In: Schewe, S., Zhang, L. (eds.) 29th International Conference on Concurrency Theory, CONCUR 2018, September 4–7, 2018, Beijing, China. LIPIcs, vol. 118, pp. 4:1–4:16 (2018)
 39. Protzenko, J., Beurdouche, B., Merigoux, D., Bhargavan, K.: Formally verified cryptographic web applications in webassembly. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19–23, 2019. pp. 1256–1274. IEEE (2019). <https://doi.org/10.1109/SP.2019.00064>
 40. Protzenko, J., Zinzindohoué, J.K., Rastogi, A., Ramananandro, T., Wang, P., Béguelin, S.Z., Delignat-Lavaud, A., Hritcu, C., Bhargavan, K., Fournet, C., Swamy, N.: Verified low-level programming embedded in F. Proc. ACM Program. Lang. 1(ICFP), 17:1–17:29 (2017). <https://doi.org/10.1145/3110261>
 41. Ramananandro, T., Delignat-Lavaud, A., Fournet, C., Swamy, N., Chajed, T., Kobeissi, N., Protzenko, J.: Everparse: Verified secure zero-copy parsers for authenticated message formats. In: 28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14–16, 2019. pp. 1465–1482. USENIX Association (2019)
 42. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Aug 2018). <https://doi.org/10.17487/RFC8446>
 43. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P., Kohlweiss, M., Zinzindohoue, J.K., Béguelin, S.Z.: Dependent types and multi-monadic effects in F*. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 256–270. ACM (2016). <https://doi.org/10.1145/2837614.2837655>
 44. The Coq Development Team: The Coq Proof Assistant (2024). <https://doi.org/10.5281/zenodo.11551307>
 45. VanHattum, A., Schwartz-Narbonne, D., Chong, N., Sampson, A.: Verifying dynamic trait objects in rust. In: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice. pp. 321–330 (2022)
 46. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: Hacl*: A verified modern cryptographic library. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 1789–1806. ACM (2017). <https://doi.org/10.1145/3133956.3134043>