High Assurance Cryptographic Software using Rust and Rocq

Lasse Letager Hansen

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

High Assurance Cryptographic Software using Rust and Rocq

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Lasse Letager Hansen
2025-08-31

Abstract

As technology is advancing, more domains are moving to the electronic and online world. This introduces both more complexity, which can introduce bugs, and also allows for the ability for attacks, especially as computers are getting more powerful and the world is getting more connected. Thus, we should ensure not only the correctness but also the security of our technology.

We have built *Hax*, a multi-prover developer-driven framework, to take code written in a safe subset of Rust, i.e., memory- and type-safe, and translate it into a selection of backends. To help with the scalability of Hax, we develop a verified version of parts of the Rust core library.

To do formal cryptographic proofs in a game-hopping style, we use *SSProve*, a Rocq library, which is based on state-separating proofs (SSP). We extend Hax with a Rocq and SSProve backend to allow us to use Rust as a specification and implementation language.

Using this basis, we first developed a framework to do *end-to-end formal verification of efficient cryptographic implementations*. We achieve this by writing the efficient implementation in Jasmin and the specification in Rust and then translating both to SSProve. We apply this framework to AES and show the implementation equivalent to the specification and prove security (IND-CPA).

Next, we build a framework for *proving security and correctness of practical cryptographic protocols*. We implement Bert13, a portable, post-quantum TLS 1.3 implementation in Rust. We use Hax to translate it into multiple backends. We prove parser correctness and panic freedom in F*. ProVerif is used to show protocol-level guarantees, such as server authentication and session key forward secrecy. Finally, we use SSProve to formalize parts of the security proof of the TLS 1.3 key schedule, strengthening the results of an existing paper proof.

In the final paper, we introduce a framework to do *formal verification of security and correctness properties of smart contracts*. The framework uses Hax to translate smart contracts into SSProve and ConCert. We apply it to a realistic voting protocol, the open vote network (OVN) protocol, for which we show maximum ballot secrecy (MBS), a security property, in SSProve, and self-tallying, a correctness property, in ConCert. After the paper, we present an alternative proof of the MBS and improvements to the smart contract.

We finish the thesis with related work, a general discussion of the results from the thesis, and future work.

Resumé

I denne afhandling vil vi introducere nogle værktøjer og teknikker til at lave formel verifikation af kryptografisk kode. Et stort fokus for teknikkerne er brugen af *Hax* og *SSProve*.

Vi starter afhandlingen med at beskrive Hax, som er et værktøj, der tager kode skrevet i en sikker del af Rust og oversætter det til en bevisassistent (ProVerif, $F^\star$, L∀∃N, EasyCrypt, Rocq, and SSProve). Vi viser derefter nogle eksempler på, hvordan man kan bruge SSProve, et kodebibliotek til Rocq, til at bevise den kryptografiske sikkerhed af Rust-koden, vi har oversat med Hax. Eksemplerne vi kigger på er kryptografiske primitiver, protokoller og en smart kontrakt.

Først viser vi at en effektiv implementering af den avancerede krypteringsstandard (AES) gør det samme som en specifikation og viser implementeringen sikker i SSProve.

Derefter viser vi sikkerheden af transportlagssikkerhedsprotokollen (TLS). Her bruger vi flere værktøjer til at vise forskellige sikkerhedsegenskaber. Vi bruger $F^\star$ til at vise, at TLS koden ikke laver fortolkningsfejl af de modtagne beskeder, og at koden ikke har fatale fejl. ProVerif bruges til at vise at TLS protokollen overholder serverautentifikation og den fremadrettede sikkerhed af sessionsnøglerne. Vi går i dybden med formaliseringen af sikkerhedsbeviset for, hvordan TLS genererer sikkerhedsnøgler.

Til sidst tilføjer vi muligheden for at bruge Hax til at oversætte smarte kontrakter skrevet i Rust. Vi bruger dette til at vise at Open Vote Network (OVN)-protokollen både er sikker og korrekt.

Afhandlingen slutter af med at sammenligne med andre værktøjer og andre projekter, og vi diskuterer, hvor vi tænker, man skal fokusere fremtidigt arbejde.

Acknowledgments

I would like to thank **Bas Spitters** for holding me to a high academic standard. I am grateful to my support group – **Claudio Orlandi**, **Panagiotis Karras**, and **Michael Wessely** – for guiding me through stressful times. I would like to thank the **Logic and Semantics research group** for letting me join seminars even during my bachelor's and master's and generally being welcoming and friendly people.

I would like to thank **Cryspen** and **Inria** for making my stay abroad pleasant. I especially want to thank **Karthikeyan Bhargavan** for the collaboration and help leading up to, during, and after the mandatory research environment change.

Having a dedicated place to relax and do silly things is very important to stay sane; TÅGEKAMMERET is such a place. I especially enjoyed the goofy events and lighthearted chats that took my mind off work. A heartfelt thanks to the people of **Regnecentralen** (), the community around our social student kitchen, for also being a place to relax and talk with students about everything and nothing. **The coffee ping chat** (☕) provided a welcome break from work and a chance to discuss challenges with other PhD and master's students from outside my research group. I appreciate the computer science Friday bar, **Fredagscaféen** (), for always being open on Fridays and for having wonderful people to talk with. I would especially like to thank my **"Don't Panic"** sticky note for keeping me company during late hours at the office. Finally, I would like to thank **family and friends** for being a source of joy and positivity.

Lasse Letager Hansen,
Aarhus, 2025-08-31.

Contents

Part I

Overview

Chapter 1

Introduction

Throughout the thesis we will look at a selection of examples illustrating the goals and ideas for verifying secure primitives and protocols and programs using these. The topics throughout the thesis can be seen as separated into the following topics:

- Implementation
- Testing and Correctness
- Security Properties and Proof
- Smart Contract and Blockchain

The presentation of the frameworks will be example driven, using the Hax framework, which allows one to write Rust programs and translate them to a set of backends (Rocq, SSProve, F*, ProVerif). Thus, we start by introducing Hax and how it can be used.

For the examples, we will start at the low levels of abstraction. Looking at how to write primitive specifications in Rust, connect an efficient assembly implementation, and show security properties of primitives. As an example of this, we show that an implementation of the advanced encryption standard (AES) in Jasmin adheres to an AES specification that we have written in Rust. The specification is based on the National Institute of Standards and Technology (NIST) specification.

Next we will look at how to use these primitives in the construction of protocols while ensuring the security and correctness of computations. The example we will use for this is Transport Layer Security (TLS) 1.3.

Finally, we will look at the use of cryptographic primitives in the setting of smart contracts. The example we will look at is the Open Vote Network (OVN) protocol.

1.1 Structure of Thesis and Overview of Chapters

The first part of the thesis contains background theory and a description of the state of high-assurance cryptography at the start of the thesis. The second part of the thesis is all the papers submitted throughout the thesis. Finally, we will conclude with an overview of the current state of high-assurance cryptography, related work, a summary and discussion of the contributions of this thesis, and future work.

The background theory chapter (Chapter 2) will introduce general concepts that serve as a more detailed introduction to a selection of topics. This is intended to establish a basis of knowledge for readers with no prior knowledge.

The next section will present the field of high-assurance cryptography and introduce the open problems and existing solutions before the work of this thesis. It is meant to frame this work and motivate the problems we are trying to solve and how we are trying to solve them. This further serves as a guide to what we are not trying to solve and gives a snapshot of the landscape of high-assurance cryptography. This snapshot will be updated to a more current view in the discussion chapter, where we place this work in relation to other projects and tools.

We introduce each paper as a separate chapter. Each chapter will start with a detailed background on the work and the problems it is trying to solve. This will be followed with a copy of the submitted/accepted version of the paper. Finally, we expand upon the work. This is done by adding perspective on what alternative proof and formalization or possible improvements could be made to the tool, artifacts, and frameworks we introduce. We also place the work on each paper into the broader effort and goals of the thesis as a whole.

We first present the paper on Hax (see Chapter 3) to introduce the framework used for writing specifications and implementation of the primitives and protocols used in the remaining papers. The remaining papers will introduce a framework for doing different types of formalizations together with an example. We go from low to high level, thus starting by presenting an end-to-end formalization for an efficient primitive (AES); see Chapter 4. Next we present the work on verifying cryptographic protocols (Chapter 5), and finally we present the work on proving security and correctness of smart contracts (Chapter 6).

Finally, we have the related work (Chapter 7), discussion (Chapter 8), and future work (Chapter 9) chapters to put the work of this thesis into perspective.

1.2 Motivation

Computers are becoming a universal part of all aspects of society. Almost all bank transactions are now automated or done online. Communication is done via email or messaging apps. Even elections are moving to electronic voting, which enables faster and more precise voting.

As we are transitioning to online and electronic technologies for all these applications the correctness is becoming very important. The types of attacks are ever-increasing, as the technologies become more complex. To build *high-assurance software*, we thus need techniques, tools, and frameworks for ensuring *correctness* and *security*. A lot of work has already been done for verifying classes of correctness, resulting in, e.g.,

- verified compilers [65]
- static analysis tools [51, 53, 63, 95]
- continuous integration (CI) with fuzzing, unit testing, and property-based testing [27]
- trace-based analysis for reasoning about complex properties and systems [3, 9]

While all of these are a part of producing high-assurance software, some parts of software require even stronger guarantees, like security. As a voting protocol can be correctly implemented but still be insecure by leaking people's votes or allowing other parties to manipulate the result. Thus, we also need a way to talk about the specification of protocols and prove security about the implementation of such protocols. Places where a gap exists include

- the implementation not following the specification
- the specification not being secure, i.e., not guaranteeing the properties we want
- missing security properties

Ensuring these properties are usually done by having the cryptographers, who design the protocol, prove the specification is secure in some cryptographic model of security. Then software engineers implement the protocol and ensure adherence to the specification. This often builds on test vectors, which are a set of tests describing the intended output for some given input.

The issue with this process is that there might be mistakes in the cryptographic proof not caught by reviewers. The implementation might have subtle *bugs* not caught by the test vectors, or the implementation might introduce alternative attacks. Thus, we would want to formally verify the security proof and prove that the implementation follows the specification, not just test it. In this work, we will use *interactive theorem provers* to formally verify these properties in a machine-checkable way. This rules out issues with the specification. We will also introduce frameworks for using multiple specialized tools to make this process easier, as doing formal proofs is often very costly and out of scope for most software. We do, however, reason that for cryptographic protocols and the part of software with high importance, formal methods should be applied.

The workshop on high-assurance crypto software (HACS)[1] has been running for 10 years now, and many successful projects have come from this. These projects have focused on formalizing primitives and protocols in both the symbolic and computation models; however, a lot is still to be done for building not only secure and correct implementations but also efficient and realistic implementations.

[1]`https://www.hacs-workshop.org/`

The Everest project[2] [14, 88] is another large-scale verification effort. The project aims to create protocols and primitives used in the HTTPS ecosystem, making industry-strength secure drop-in replacements.

We contribute to HACS community and try to push formalization towards real protocols and efficient implementations.

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

Chapter 2

Background Theory

We will start by introducing some general background about interactive theorem provers, computer-aided cryptography, and Rust. This will lay the foundations to understand the topics in the rest of the thesis. Whereas any theory more specifically related to a paper will be introduced in the relevant chapter.

2.1 Dependent Type Theory

We start by introducing dependent type theory [2, 94], which is the basis for most formal proofs, tools, frameworks, and techniques.

Curry-Howard Correspondence

Haskell Brooks Curry and William Alvin Howard have observed [34, 35, 52] a relation called the *Curry-Howard correspondence*, which states that proofs and computations are two sides of the same coin. That is, for any model of computation (e.g., Turing machines, λ-calculus, etc.), there is a logic system (e.g., intuitionistic logic, type theory, etc.) for which it coincides.

The constructive interpretation that a proof of a proposition is regarded as true only if it is possible to construct a proof of its parts [92], originally posed by Luitzen Egbertus Jan Brouwer and Arend Heyting [49], and Andrey Nikolaevich Kolmogorov [62], is known as the *Brouwer–Heyting–Kolmogorov interpretation*.

The description of these ideas is also collected under the *propositions as types paradigm*, which states that propositions are types. This is the basis we will use for formally reasoning about proofs, as we can build on these ideas and observations to use, e.g., λ-calculus as a framework for doing constructive proofs, as it has a direct correspondence with type theory.

Church-Turing Thesis

Another influential result, derived from the work of Alonzo Church and Alan Turing, is the *Church-Turing thesis* [24–26, 93], which states that simply typed λ-calculus and

Turing machines are equally expressive, i.e., equivalent to general recursive functions. Said in another way, general computation can be modeled by both Turing machines and λ-calculus, among others. Thus, there are many different ways to build a system of computations, which, given the above results, can be used to construct proofs. We can therefore reason about general computation, e.g., what happens in a real computer, using the more abstract mathematical formulations like λ-calculus. Furthermore, since code and proofs are strongly connected, we can use a programming language to reason about proofs and a proof framework to reason about code.

Martin Löf Type Theory (MLTT)

Martin Löf Type Theory (MLTT) [2, Appendix A; 68; 94, Chapter 1] builds a dependent type theory, which can be used as an alternative to standard Zermelo–Fraenkel (ZFC) set theory [40, 41, 54, 98].

In a dependent type theory, we state lemmas and theorems as types, and prove them by constructing a term/element of the given type. The type theory being dependent means we allow types to depend on other terms or types. From the Curry-Howard correspondence we can build a programming language or calculus for reasoning about such proofs.

Calculus of Inductive Construction (CIC)

A version of MLTT is the Calculus of Inductive Construction (CIC) [32, 78], which is a constructive foundation for mathematics that can be used as a dependent typed programming language. It is a higher-order typed λ-calculus and is part of the λ-cube (upper top right part λC), which classifies different type theories and their theoretic

$$\begin{tikzcd}
& \lambda\omega \arrow[rr] & & \lambda C \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
& \lambda\underline{\omega} \arrow[uu, dashed] \arrow[rr, dashed] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda^{\rightarrow} \arrow[uu] \arrow[ur, dashed] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Figure 2.1: Lambda Cube
dependent types ($\rightarrow$), polymorphism ($\uparrow$), type binding ($\nearrow$)

strengths. Thus, CIC can be used for formalizing mathematical proofs. Furthermore, interactive proof assistants (like Rocq and L∀∃N) have been built on top of this theory.

2.2 Interactive Theorem Provers (ITP)

There have been multiple examples of mathematical results being wrong. Even results, which have been rigorously peer reviewed and accepted at high-prestige conferences. Furthermore, as computers and automation techniques are getting better, proof automation is a topic of increasing interest. This has led to the field of constructive mathematics and the introduction of interactive theorem provers (ITP). Constructive mathematics is a way of writing mathematical proofs and statements in a way that the proof can be algorithmically constructed. Interactive theorem provers are then a way of writing these mathematical proofs such that a computer validates the correctness while giving the user feedback. This enables automation and strong guarantees.

The core of the system is a type checker, which validates that terms are of the correct type. This is used to show that we have a proof (the term) for some statement (the type) and can be seen as an example of the Curry-Howard correspondence, as we can complete proofs using code and tactics. A full system can then be built around the rather small core of a type checker.

An example of an interactive theorem prover is the *Rocq* proof assistant (formerly known as Coq) [31, 76, 91]. It builds on the Calculus of Inductive Construction (CIC).

Another example of a theorem prover is $F^\star$ [85, 86, 90]. It is a dependently typed programming language, which does not build on CIC; instead, it uses SMT solvers to discharge proof obligations. The trusted core of $\mathrm{F}^\star$ is larger than that of more rigorous systems; however, it is still based on a sound foundation but with a focus on automation, allowing it to scale to large and complex code bases.

2.3 Computer-Aided Cryptography

In this section we will introduce some different types of computer-aided cryptography. We start with computational and constructive cryptography [73], explaining state separating proofs [21] and a brief introduction to the Joy of Cryptography book [81], which teaches this style of cryptographic proofs. Some of these descriptions will be repeated in Subsection 2.4 of the OVN paper. Next, we introduce the symbolic (or Dolev-Yao) model [37] of cryptography. Finally, we describe some of the common assumptions and models used in cryptographic proofs.

The Computational Model of Cryptography

The computational model [72], also known as the standard model, requires one to prove security against a pretty strong adversary. The adversary has access to the implementation of the primitive or protocol. Furthermore, the adversary is allowed to do any polynomial amount of work based on the security parameters. Thus, the security needs to scale super polynomially (e.g., exponentially) in the security parameter, such that a value for the security parameter can always be found, ensuring security against current adversaries.

Constructive Cryptography

Constructive proof theories state that proofs should be constructible; that is, one needs to give some algorithm or construction to build the proof. Thus, if there are any existential statements, one should also be able to exemplify those. The strength of constructive proofs is that they can be machine checked and even automated to some point. The field of constructive cryptography [73] builds on similar ideas, focusing on the excitability and constructability of cryptographic proofs.

State Separating Proofs (SSP)

A framework for doing constructive proofs is the state-separating proof framework [21]. To ensure the framework is constructive, it uses the fact that cryptographic primitives and protocols can be defined by code. An SSP proof then proceeds by taking the code and splitting it into modular parts called packages. A package can then be shown to be observationally equivalent to another package. That is, an adversary cannot distinguish the behavior of using one or the other. We call such a pair of packages a game if they have no dependencies and define the same functionality. This allows us to swap one package for another, also known as a game jump. The goal is then to build a sequence of game jumps to connect the real implementation of a protocol with an ideal behavior of the protocol.

The Joy of Cryptograpy

Similarly to SSP, the Joy of Cryptography book [81] uses a framework for reasoning about constructive cryptography in a modular style. The goal of the Joy of Cryptography is to teach cryptography consistently. Proofs use a reduction-based reasoning style, where we want to prove the security of the full package. Sometimes this means showing security of parts (libraries), as is done in SSP; other times the focus is on reasoning about the equivalence in the output distributions.

The Symbolic Model of Cryptography

In the symbolic model, also known as the Dolev-Yao model [37], of cryptography, the adversary no longer has access to the inner workings of the algorithms. Rather, the adversary can read, create, block, or change messages. Thus, we look at the algorithms as a form of black box that might leak information. This gives a symbolic model allowing for stronger techniques to be applied, making it easier to formalize larger protocols, at the cost of some fine grained details. Tools for this model can show that a protocol follows some guarantees or produce a counterexample showing the protocol is broken.

Assumptions of Cryptography

A primitive that is used a lot in modern cryptography is hashing. Cryptographic hashing takes a value as input and produces a (possibly smaller) output. The hope is that given this output, it is hard to find the input, but given an input, it produces a deterministic output, i.e., inputting the same value twice will give the same output. This is very useful, as true randomness is very hard to come by.

The random oracle model [11] assumes the existence of an oracle that, given a fresh input, produces a uniformly random output; however, given an already seen input, produces the same value. Usually, this random oracle is instantiated with a hashing function, even though there is no proof that it is truly random.

2.4 Rust

In this section we will give an introduction to the philosophy and language features of the Rust programming language [59, 60, 69].

Types

We will start by giving an account of the types in Rust [82] to form a basis for discussing the language features. Rust supports primitive types:

- Booleans (`bool`),
- numerics – integers (`u8`, `i8`, ..., `u128`, `i128`) and floats (`f32` and `f64`),
- textual – char (`chr`) and string (`str`), and
- never (`!`).

The *never* type is used to represent infinite computations; that is, a type without a value, thus not allowing for returns.

These primitive types can be grouped using *tuples*, *arrays*, and *slices*. Tuples allow one to group 0 or more things of different types together. The size of a tuple is given directly in the type, e.g., if we want to return 2 Booleans and a string, then the type would be (`bool`, `bool`, `str`). Tuples are also allowed of size 0, representing the unit type, which only has a single element `()`.

Arrays are a specific-sized grouping of the same type; e.g., a five-element-long list of `u8` would be [u8; 5]. Thus, the size of both tuples and arrays is known at compile time.

In contrast, we have dynamically sized lists, known as slices. These have the same syntax as arrays, but without a size, e.g., [u8]. We can combine these sequence types to make arrays of tuples, or slices of arrays, or whatever anyone can dream up.

Rust has *structs* similar to those in C. This allows one to combine types into a named user-defined type. For example:

```
struct Foo {
  foo : u8,
  bar : (str, bool),
  baz : [f64; 12],
}
```

To construct a struct, one needs to construct all the parts and instantiate the named fields. The values can then be updated or projected out of the struct as needed. Thus, structs are a way to organize and structure the types.

Rust also supports *enums* to enable inductive reasoning, which comes to expression in the combination with the pattern matching functionality of Rust. The goal is to support full algebraic data types; however, this is not yet fully supported. Enums currently allow one to define possible cases for a type. For example:

```
enum Foo {
  Bar,
  Baz(u8, str, baz),
}
```

Rust also supports *unions*, which are a lot like structs, also sharing the syntax, but using `union` instead of `struct`. However, all fields in a union share the same memory. Unions should, therefore, be seen as an optimization tool for accessing parts of a common data type and combining different structures with possibly overlapping values.

Next, we have the low-level types for working with the operating system. These are *references*, *raw pointers*, and *function pointers*. Raw pointers are the memory pointers used by the operating system and thus are generally unsafe to use and manipulate. This is reflected by the Rust requirement to explicitly declare code that does this `unsafe`. Similarly, function pointers are used for enabling a more functional style of coding.

Function pointers can be given to, e.g., `map` or `fold` functions. Having function pointers as objects also allows us to define anonymous functions (also known as lambda functions), enabling functionality like callback functions.

One of the main ideas of Rust is the use of references. In Rust having a reference to an object implies ownership of that object. There are two types of references: a shared reference (&) for reading the value of an object and a mutable reference (`&mut`) allowing for modification of the object. Allowing the user to access and use references ensures the ability to construct efficient code on par with C. By controlling the access patterns or ownership model, Rust ensures memory safety without the use of a garbage collector. This is achieved by having the owner of an object be responsible for clearing the memory use of that object. Thus, if the owner of an object goes out of scope, that object will also be removed. This mechanism is captured by the Rust borrow checker.

To properly define and use function pointers, Rust allows one to define function types and closures, which describe the input and output types together with how the elements and memory are captured. General functions in Rust can be defined over *generic values* and trait implementations; however, a function pointer must be fully instantiated.

The final tool in Rust to help structure code is the *trait system*. Traits are a collection of generalized functions and types, which allows one to create an abstract model of some behavior. These traits can then be implemented for some given types. One of the selling points of the Rust trait system is that

- implementation for external traits can only be done for types defined in the current crate;
- implementation for external types can only be done for traits defined in the current crate.

These properties ensure the predictability of crate inclusions, in that including an extra crate cannot break existing functionality.

Control Flow

Rust allows for basic as well as more complex control flow [82]. Using *branching*, *repetition*, and *early returns*, maintain the safety of lifetimes, borrowing, and memory.

Unsafe Rust

When Rust wants to interact with the operating system, it calls out of its encapsulated workflow with all its guarantees. Thus, this might cause undefined or unintended behavior. To make this unsafety explicit, Rust uses the `unsafe` keyword. There is an entire book, the Rustonomicon [89], explaining the correct use of `unsafe` and where and why it is necessary. Some discussion of tools and projects related to ensuring correctness of unsafe code can be found in §7.1.

Part II

Publications

Chapter 3

Hax: Verifying Security-Critical Rust Software using Multiple Provers

We first introduce the subset of the Rust programming language that is supported by Hax. Then we give some additional background knowledge related to the Hax framework. We then give an overview of the Hax paper [15], followed by the paper in full. Afterwards we summarize the paper and extend the description of the *annotated core library* and discuss how to combine Hax with other tools and automation.

3.1 Hax

Hax is a tool that takes Rust code and translates it into a selection of tools and backends. The reason to do this is to formally reason about the correctness and safety of the code. Hax currently supports a large subset of Rust. Since this subset is described in most papers, we will give a detailed and combined description here. Thus, the Hax section in the other papers can be skipped for brevity.

The Hax Subset of Rust

We will describe the Hax subset of Rust by going over the language features described in §2.4. We start with the types. Hax supports all the basic types of Rust, e.g., Booleans, numerics, textuals, and the `never` type. However, for most high-assurance applications, textuals are rarely used. Furthermore, few of the Hax backends support non-termination, so the `never` type is not that relevant.

Hax represents structs and enums as a combined type representing algebraic data types. The types are modeled by records and inductive types, respectively, in most of the backends. Unions are semantically very close to structs; however, Hax does not handle the memory model of Rust. Therefore, unions are currently not handled by Hax but are represented in the input to the Hax framework. This also means raw pointers are not translated. There is some support for function pointers and mutable references; however, the more advanced functionality is not yet handled. Hax does support closures and lambda functions, which allows us to write Rust code using the functional paradigm.

Hax supports traits; however, these are translated into type classes in most backends. This results in some of the more advanced features of traits not being translatable.

Specification

The translation of the Rust code starts by hooking into the (typed) higher intermediate representation (THIR) and extracts the terms (as JSON). Next the entire code is structured into an Abstract Syntax Tree (AST).

The AST encodes features to be used in all transformations done to map it to the different backends. This is done by having the AST be defined over a feature set, which can enable and disable different types. The backends then accept a specific combination of features, which is a result of a series of reduction phases. The surface AST can represent the following features:

- Loops (for, for-range, while, loop),
- Traits and implementations,
- Mutable variables, and
- Type definitions (enums, structs, unions).

The phases then transform the AST; most of the proof assistants support a functional feature set with bounded recursion and no mutable variables. We add phases that panic if anything not allowed is encountered, thus restricting the translation. Writing code in the Hax subset is therefore not entirely like writing Rust, but the differences are small, especially for the use cases examined throughout the papers of this thesis. For more details see the paper following §3.2.

Hacspec

The domain of cryptography and general algorithmic specifications rarely requires the advanced features of Rust. Thus, by further restricting the Hax subset, we can define a specification language, which ensures simplicity and correctness. Since such a specification would still be in the Hax subset, we can translate it into the different backends of Hax. Thus, we achieve executable specifications with simple semantics that can be formally verified as correct. Any efficient implementation can then be tested and verified against the specification. For more on this methodology, see §4.1 or, more generally, Chapter 4.

As part of the Hacspec project [7, 74], a constant-time library was built for simplifying the specification of constant-time primitives and protocols. Having a common set of libraries used for writing executable specifications would benefit the high assurance community greatly.

3.2 The Paper

The work on Hax originates from the Python implementation of Hacspec [74]. The further development of a high-assurance cryptography library, libcrux, and the Hacspec specification language was presented [58] at the Real World Crypto (RWC'23) Symposium. The introduction of Hax as a framework and explanation of further improvements was presented [16] at RustVerify. A full paper on the current version of Hax is accepted at Verified Software: Theories, Tools, and Experiments (VSTTE'24). This paper is accompanying the invited talk of Karthikeyan Bhargavan (co-author) at VSTTE'24. The following pages will contain the paper in full [15], after which we will summarize the content of the paper and frame its relevance in §3.3.

hax: Verifying Security-Critical Rust Software Using Multiple Provers

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

[1] Cryspen, Paris, France
karthik@cryspen.com
[2] Aarhus University, Aarhus, 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

J. Protzenko and A. Raad (Eds.): VSTTE 2024, LNCS 15525, pp. 96–119, 2025.
https://doi.org/10.1007/978-3-031-86695-1_7

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 [11,15,18] and computational cryptographic provers [7,10,14,27].

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 HACL* 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. [5,32], 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 projects[1], operating systems[2], and web browsers[3]. Governmental organizations [1], research institutions[4], and industry bodies[5] 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 [6, 21,24,29,33,37,45]. 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

[1] https://cryptography.rs/.
[2] https://docs.kernel.org/rust/index.html.
[3] https://security.googleblog.com/2023/01/supporting-use-of-rust-in-chromium.html.
[4] https://www.darpa.mil/program/translating-all-c-to-rust.
[5] https://www.memorysafety.org/.

to support the development, specification, and verification of *implementations* of more general security mechanisms written in idiomatic Rust.

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 other tools can use. As a result, the hax frontend is an independently useful tool and is also used by other Rust verification frameworks like Aeneas [28].

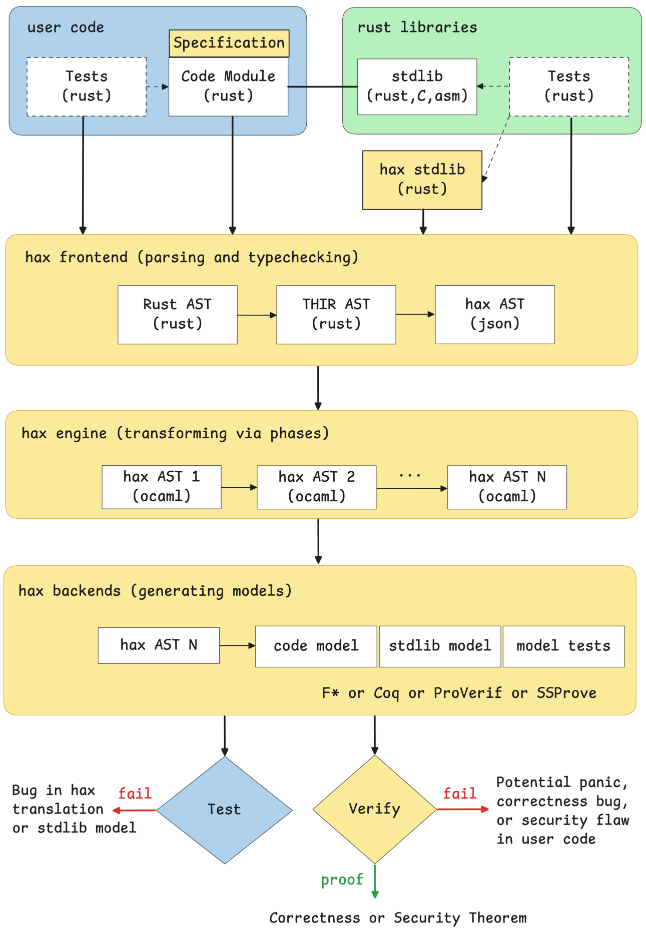


Fig. 1. hax architecture

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 directly translated to the input languages of various backends. We will describe some of these phases in Sect. 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 Sect. 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 Sect. 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 Sect. 6.

Several projects are using hax to formally verify real-world software. We briefly mention some of these projects in Sect. 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/hacspec/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

Fig. 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

```
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 ::=
|

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.

length slices, function types, and named types defined by enums and structs. We currently do not support raw pointer types or dynamic dispatch.

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 code[6]. 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.

[6] https://hacspec.org/hax/engine/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 hacspec [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} `<= value <=` 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 Fig. 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

```
 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 ≫! 1l <: 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

are then translated by the F* backend to the corresponding library functions modeled in F* (e.g. Core.Convert.f_from).

- 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 Fig. 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:

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

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

```
Definition xor (a : both int64) (b : both int64) : both int64 :=
  xor a b :=
    letb (x : int64) := a in
    letb (y : int64) := b in
    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 ⊕ 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 satisies these goals against a large class of *symbolic* or Dolev-Yao adversaries [22]. This threat model is one where the adversary can perform unbounded computatation, 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.

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

A protocol implementation typically consists of a list of such functions, each of which either processes or produces a protocol message, using some internal state, cryptographic operations (like `encrypt`) and parsing/serialization functions.

```
letfun proverif_psk__initiate(ikm : bitstring, psk : proverif_psk__t_KeyIv) =
       let response_key_iv = proverif_psk__derive_key_iv(
         ikm, proverif_psk__v_RESPONSE_KEY_CONTEXT
       ) in (
         let serialized_responder_key =
           proverif_psk__serialize_key_iv(response_key_iv)
          in
         let initiator_message = proverif_psk__encrypt(
           psk, serialized_responder_key
         ) in (initiator_message, response_key_iv)
         else bitstring_err()
       )
       else bitstring_err().
```

Fig. 4. ProVerif Translation of Protocol Initiator

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 Fig. 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:

```
free PSK: proverif_psk__t_KeyIv [private].
free SECRET_PAYLOAD: bitstring [private].
query attacker(PSK).
query attacker(SECRET_PAYLOAD).
process
  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 verificaiton 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 std[7]), 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 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

[7] https://github.com/model-checking/verify-rust-std.

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 Fig. 5 for how some of them are mapped in the Coq backend.

```rust
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";
}
```

$\Longrightarrow$

```coq
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`):

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

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.

```coq
Instance t_PartialEq_774173636 : t_PartialEq ((t_U8)) ((t_U8)) :=
{
  PartialEq_f_eq := fun  (self : t_U8) (rhs : t_U8) ⇒
    PartialEq_f_eq
      (haxint_cmp
        (Abstraction_f_lift (Clone_f_clone (self)))
        (Abstraction_f_lift (Clone_f_clone (rhs))))
      (Ordering_Equal);
}.
```

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 out 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[8] 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[9]. 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.

[8] https://hacspec.org.
[9] 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 hacspec 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 Bertie[10] to perform a symbolic security analysis[11]. 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 hacspec 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 Easy-Crypt 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 (2024). https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf
2. Module-Lattice-based key-encapsulation mechanism standard (2024). https://doi.org/10.6028/NIST.FIPS.203
3. Almeida, J.B., et al.: Jasmin: high-assurance and high-speed cryptography. In: CCS, pp. 1807–1823. ACM (2017)

[10] https://github.com/cryspen/bertie.
[11] 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. LNCS, 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., et al.: Sok: computer-aided cryptography. In: SP, pp. 777–795. IEEE (2021)
10. Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., Strub, P.: Easycrypt: a tutorial. In: FOSAD. LNCS, 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., et al.: 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. LNCS, vol. 8604, pp. 54–87. Springer (2013)
16. Bond, B., et al.: 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: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018. LNCS, vol. 11274, pp. 222–249. Springer, Cham (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., et al.: 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. Proc. ACM Program. Lang. **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., et al.: 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. arXiv preprint arXiv:2410.18042 (2024)
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., et al.: HACSPEC: a gateway to high-assurance cryptography. RealWorldCrypto (2023)
32. Kroening, D., Schrammel, P., Tautschnig, M.: CBMC: the C bounded model checker. arXiv preprint arXiv:2302.02384 (2023)
33. Lehmann, N., Geller, A.T., Vazou, N., Jhala, R.: Flux: liquid types for rust. Proc. ACM Program. Lang. **7**(PLDI), 1533–1557 (2023)
34. Merigoux, D., Kiefer, F., Bhargavan, K.: Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust, Technical report, Inria (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. LNCS, vol. 4963, pp. 337–340. Springer (2008)
36. NIST: Module-lattice-based key-encapsulation mechanism standard, Technical report, 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., et al.: 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., et al.: 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., et al.: 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 (2018). https://doi.org/10.17487/RFC8446
43. Swamy, N., et al.: 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

3.3 Summary

We present the Hax framework as a tool for writing verifiable implementations and specifications in Rust. The tool allows us to translate code written in a subset of Rust into a selection of backends: ProVerif, F*, L∀∃N, EasyCrypt, Rocq, and SSProve. The translation happens in a sequence of translation phases. Thus, the framework is extensible and allows the implementation of new backends or adding support for more of Rust by writing new translation phases.

3.4 The Technical Details of the Rocq Backends

Rocq/Coq

Many of the backends of Hax use common patterns when printing the AST. Thus, Cryspen has introduced the generic printer, which enables one to write the translation case by case given the translation of all sub-terms.

The Rocq backend builds on the generic printer and enables a lot of the simplification phases as described in the paper. Taking the final AST and translating it into Rocq is done in a couple of steps.

The supported primitives of the Rocq backend are the unit type, Boolean type, and number types. We rarely handle chars, strings, and floats in the Rocq backend; however, they do get translated. For example, see Figure 5 of the paper.

Types are translated straightforwardly. Enums become inductive types. Structs become records, but we use the `coq-record-update` library [22] for supporting record updates. Aliases are defined using notation, and functions become definitions.

Rust traits are translated into type classes, and instances are resolved using Rocq's type class resolution mechanism. This mapping can cause issues, as something that did resolve in Rust might fail in Rocq. This could be made more faithful by, e.g., implementing the trait resolution mechanism in Rocq. Some efforts exist, see Chalk [46] and a-mir-formality [45], to formalize the trait system by using logical programming. This could be ported to Rocq, e.g., using ELPI. This style of having an implementation on the Rust side and an equivalent model after the translation is quite common for Hax.

SSProve

For translating to SSProve, we have two options. Reusing the Rocq translation, as Rocq code is allowed in SSProve. However, this makes it harder to work with SSProve and does not take advantage of SSProve being imperative. The other option is translating into SSProve directly, requiring a new translation and embedding of the Rust types into the `choice_type` used by SSProve. We do a combination, where we build the translation between the two during translation. For more details see §4.5.

An alternative would be to define a new Domain Specific Language (DSL) and write the translation from that to SSProve within Rocq. This would allow us to prove guarantees of the translation, allowing for stronger reasoning about memory, as we can use the memory guarantees from Rust.

This requires us to write a deep embedding; however, the current shallow embedding into SSProve already behaves much like a deep embedding into Rocq.

3.5 The Annotated Rust Core Library

The annotated Rust core library aims to unify the *specification of the standard library* for the different proof assistant backends of Hax.

One of the core principles of this approach is that we want to write a functional specification, which is closer to the backend version, and implement the imperative types and functions using the function definition. An example could be implementing `u8` using $\mathbb{N}$. To do this we first need to implement $\mathbb{N}$ (bootstrapping issues ensue). The idea is then to replace the actual Rust implementation of $\mathbb{N}$ with the backend implementation (and not the translation, solving bootstrapping). Now we can implement `u8` as one would in any of the backends, but this will align the *implementations across all the backends* and allow the automatic translation of the specification to any old or new backend. This reduces the implementation effort of the standard library to just some common types and simple properties of those types. E.g., the constructors of $\mathbb{N}$ and pattern matching, we get induction by defining things recursively.

Another benefit from implementing the standard library in this way is that we can (efficiently) *run the model* of the core library. This also allows us to reuse existing tests for the standard library and even translate them to the different backends for re-running the test after translation. Thus, we can actually validate and check the efficient implementation in the standard library against our mathematical model in the backends.

We can also go one layer deeper, as $\mathbb{N}$ is defined using the standard library, so we can instantiate that implementation another time. Thereby pushing the bootstrapping one layer deeper. Thus, we can also check if our implementation of the functional types is correct, using the translation of the implementation itself. This is in the style of projects like MetaRocq [1].

Supported Functionality

To define the mathematical number types ($\mathbb{N}$, $\mathbb{N}_+$, and $\mathbb{Z}$) in Rust, we start by defining a common subtype, namely a (named) byte array. This definition makes use of the `alloc` library as we are dynamically allocating memory. This causes a bootstrapping issue, which is solved by replacing the definitions during translation, thus removing the use of the byte array and thereby the recursive definition. To define the $\mathbb{N}$, $\mathbb{N}_+$, and $\mathbb{Z}$, we wrap the common subtype in a constructor to represent that the byte array is being interpreted in a specific way. Now that we have the types, we need to define constructors and destructors. These constructors will need to manipulate the byte array, which should not exist in the `core` library; thus, we again need to *replace these constructors* during translation. We could leave these definitions empty or undefined;

however, giving them an *actual implementation* allows us to *validate* the correctness of the specification for the core library. The constructors for $\mathbb{N}$ are 0 and `succ`, and are implemented as bit operations on the underlying byte array. Finally, we define pattern matching (i.e., destructors) for $\mathbb{N}$, by defining a `match` function going from $\mathbb{N}$ to $\mathbb{N}_{\textbf{enum}}$. Thus, given input value $n : \mathbb{N}$, the `match` function either returns `ZERO` or `SUCC`(`pred` n), which allows one to use Rust pattern matching, with inner types still being $\mathbb{N}$. Given the $\mathbb{N}$ type and its constructors and destructors, we can now implement all the operations over $\mathbb{N}$, e.g., addition, multiplication, etc.

Next we define positive binary numbers $\mathbb{N}_+$ from the constructors for `XH` (one), `XO` (times two plus zero), and `XI` (times two plus one). Again, these are defined from `mul2` and `div2` operations on the underlying byte array. The destructor again uses an enum type. Using the binary positives, we can define binary $\mathbb{N}$ as `ZERO` or `POS`, where `POS` is given a $\mathbb{N}_+$. We can furthermore define $\mathbb{Z}$ from constructors `NEG`, `ZERO`, and `POS`, with `POS` and `NEG` given a $\mathbb{N}_+$. Having the binary positives allows us to somewhat efficiently implement addition and multiplication but further allows us to implement integer division and modulo using the greatest common divisor (gcd) operation. Thus, we can actually run tests using these mathematical definitions, where unary numbers would be too slow.

Implementing, e.g., `u8` using $\mathbb{N}$ is an instance of a more general problem of defining bounded integers. We therefore define an intermediate type for both signed and unsigned bounded integers, which can be instantiated for the current (and possibly future) Rust integer types. We have a generalized constant bound for the type and define operations using the operations on the internal binary $\mathbb{N}$ or $\mathbb{Z}$ types. This usually boils down to calling the operations and then doing a modulo on the result. Thus, we get a full library for generalized machine integers that can be translated to all the backends of Hax, with a common naming scheme. This greatly *simplifies the renaming and redirection* efforts, but at the cost of using translated structures instead of simple or native ones.

Using a more efficient representation or a library with more proofs requires manually replacing more of the definitions and comes with some maintenance cost. Alternatively, we can show the translated types are equivalent to the ones in the library and then translate proofs over the equivalence.

We translate most of the primitive Rust types this way, i.e., the never type, arrays and vectors (as linked lists/cons lists), and the option and result types.

Further extensions

We have some experimental and partial support for iterators. However, since these are such a central part of the translation, some backends translate common patterns to specific language constructs (e.g., for-loops). There are also some complexity and efficiency concerns with translating iterators without manually replacing parts of them.

Another thing that is common among the backends is a proof library. Currently, we only have the definitions of the primitive types and some proof statements and unit tests for ensuring correctness. However, we could define a common proof language that is translated into proofs in each of the backends and thereby share a common formalization of the core library. Having automated proofs of statements translate into the backends would be an interesting thing to explore.

Other than these considerations, implementing more of the core library would be very useful, as this is a large part of the work required to add a new backend. Having more tools and backends connected to Hax would be very useful, as the strength of the tool lies in the combination of the strengths of each backend.

External libraries

This process of reimplementing external code and creating models for code that cannot be implemented in the Hax subset ensures a common representation, which can be tested against the original. When using Hax, as with many tools, the main restriction for adaptation is the library support, the robustness of backends, and general documentation. Having a methodology to extend both documentation and library support allows the tool to grow and enables external contributions. As specifying libraries in Hax allows one to translate to all the backends, this *ensures* that all *the backends are robust*, not just the most used ones.

3.6 Creusot

Creusot is a framework for annotating and validating properties of Rust code. The framework allows one to write models and generate proofs from annotations. These proofs are then checked using SMT solvers (Why3).

We have added this annotation to an earlier version of Hax, as the supported subset of Hax is still just Rust code. Translating these annotations to, e.g., Rocq, we can directly reprove the properties. One interesting project in this regard is “A Formalization of Core Why3 in Coq” [30], which would allow us to extract Why3 code with Creusot and then run Why3 in Rocq, getting the SMT proof within Rocq.

Pearlite

Pearlite is the annotation language used by Creusot. The Hax engine did at one point support writing Pearlite annotations and getting proof statements translated into Rocq. In the current version of Hax, more general contracts[1] are used. Having the annotation between the tools aligned will allow us to run the static analysis tools or SMT solvers before translation and formally proof the results after translation. This can help ensure the correctness of the properties that would be a target of verification.

[1]`https://doc.rust-lang.org/core/contracts/index.html`

3.7 QuickCheck/QuickChick

QuickCheck is a property-based testing framework based on the original library with the same name in Haskell [27]. It allows one to test properties by generating a lot of tests. Here, we need to remember the following wise words.

Testing shows the presence, not the absence, of bugs.
— Edsger Wybe Dijkstra

However, it is still useful to find issues before starting on formal proofs. Furthermore, we can translate the statements of QuickCheck into Rocq. This allows us to rerun the tests using QuickChick, the Rocq implementation of QuickCheck. We can also proof the statements once and for all, after having shown it holds for, e.g., 10,000 examples.

This process has already helped capture issues in an elliptic curve implementation, as the generator in Rust did not find the case, which was found in Rocq. This also highlights the importance of writing good generators when working with property-based testing.

3.8 Automation: SMT solvers, Hammers, and Large Language Models (LLMs)

Extending the selection of tools like Creusot and QuickCheck, which Hax can collaborate with, emboldens the multiprover/multitool paradigm of Hax. Many static analysis tools can give a lot of strong properties with little extra effort. Starting the formalization with these guarantees can lessen the complexity of the formalization and the amount of annotation needed to express it.

The focus of Hax is on verification of complex properties or protocols. Thus, automation has been less of a priority, as, e.g., security of TLS cannot be proven automatically. However, having a common representation, using proof transfer, and offloading simpler proof statements to SMT solvers, hammers, or LLMs could greatly improve the verification process. This also lowers the requirement for verification by allowing some simple code to be verified without the need for expertise in interactive theorem proving.

Chapter 4

The Last Yard: Foundational End-to-End Verification of High-Speed Cryptography

We start by giving some extra background theory relevant for the paper. First we present Hacspec a functional subset of Rust for writing (cryptographic) specifications. Next, we explain relevant properties of the Advanced Encryption Standard (AES). Then the paper is presented, followed by a summary (see §4.4) framing the work in relation to the rest of the thesis. We then discuss the SSProve backend of Hax and its dual translation, which can be seen as translation validation or a simple form of realizability. Finally, we discuss the possibility for a more mathematical specification of AES.

4.1 Hacspec

One of the issues with the internet standards (like IETF or NIST) is that they use ambiguous or imprecise descriptions written in pseudocode or just plain English. To fix this specification issue, we want to have a simple and executable programming language, which allows one to specify all the standards without the use of complex features, which would reintroduce some of the interpretation issues. Thus, we use the high-assurance cryptographic specification language, Hacspec [7, 58, 74], for writing specifications of protocols. The Hacspec language is a functional subset of Rust; this enables both high- and low-level specification. One of the important parts of Hacspec is the library, which implements prime fields and secret integers, as these are common structures used in cryptographic specifications. The language also allows the use of annotations for defining pre- and post-conditions on functions to make some of the requirements in specifications explicit.

The Hacspec subset of Rust allows one to use records, enums, traits, pattern matching, functions, and macros and only allows bounded recursion. We can define Hacspec as a specific set of features in Hax. Any code needing transformations to fit into the feature set should be rejected, as the specification language should remain simple! As this feature set is well within what the backends of Hax usually support, we should be able to translate Hacspec to each backend. Thus, Hacspec is an executable language with a (small) library of common and useful definitions, making specifications simple, readable, and precise. Furthermore, we can proof general properties of a specification and use Hax to write a more efficient implementation and then prove that it adheres to the specification.

4.2 Advanced Encryption Standard (AES)

We will present the mathematical specification of the Advanced Encryption Standard (AES) [75] to give insights into the inner workings. This mirrors the executable specification implemented in Hacspec that is used in the paper; see §4.3.

AES is a block-based encryption standard, which works by doing a series of mixing and shifting. If one does only mixing or only shifting, then it is not secure, but doing both seems to be enough to get cryptographic security.

AES first instantiates a list of round constants and the substitution box (S-box). The round constants are precomputations of the coefficients of x^{i-1} for $i \in [0,14]$ in the polynomial field

$$GF(2)[x]/(x^8+x^4+x^3+x+1).$$

for multiplying numbers using their representation in the above polynomial field. The computations of the round constants are then done as follows

$$\begin{aligned}
x^{0-1} &= x^{-1} = x^7+x^3+x^2+1\\
&= \texttt{0b10001101} = \texttt{0x8d}\\
x^{1-1} &= x^0 = 1\\
&= \texttt{0b00000001} = \texttt{0x01}\\
x^{2-1} &= x^1 = x\\
&= \texttt{0b00000010} = \texttt{0x02}\\
x^{3-1} &= x^2 = x^2\\
&= \texttt{0b00000100} = \texttt{0x04}\\
x^{4-1} &= x^3 = x^3\\
&= \texttt{0b00001000} = \texttt{0x08}\\
x^{5-1} &= x^4 = x^4\\
&= \texttt{0b00010000} = \texttt{0x10}\\
x^{6-1} &= x^5 = x^5\\
&= \texttt{0b00100000} = \texttt{0x20}\\
x^{7-1} &= x^6 = x^6\\
&= \texttt{0b01000000} = \texttt{0x40}
\end{aligned}$$

$$\begin{aligned}
x^{8-1} &= x^7 = x^7 \\
&= \texttt{0b10000000} = \texttt{0x80} \\
x^{9-1} &= x^8 = x^8 + (x^8 + x^4 + x^3 + x + 1) = x^4 + x^3 + x + 1 \\
&= \texttt{0b00011011} = \texttt{0x1b} \\
x^{10-1} &= x^9 = x^8 \cdot x = (x^4 + x^3 + x + 1) \cdot x = x^5 + x^4 + x^2 + x \\
&= \texttt{0b00110110} = \texttt{0x36} \\
x^{11-1} &= x^{10} = x^9 \cdot x = (x^5 + x^4 + x^2 + x) \cdot x = x^6 + x^5 + x^3 + x^2 \\
&= \texttt{0b01101100} = \texttt{0x6c} \\
x^{12-1} &= x^{11} = x^{10} \cdot x = (x^6 + x^5 + x^3 + x^2) \cdot x = x^7 + x^6 + x^4 + x^3 \\
&= \texttt{0b11011000} = \texttt{0xd8} \\
x^{13-1} &= x^{12} = x^{11} \cdot x = (x^7 + x^6 + x^4 + x^3) \cdot x = x^8 + x^7 + x^5 + x^4 \\
&= x^8 + x^7 + x^5 + x^4 + (x^8 + x^4 + x^3 + x + 1) \\
&= x^7 + x^5 + x^3 + x + 1 \\
&= \texttt{0b10101011} = \texttt{0xab} \\
x^{14-1} &= x^{13} = x^{12} \cdot x = (x^7 + x^5 + x^3 + x + 1) \cdot x = x^8 + x^6 + x^4 + x^2 + x \\
&= x^8 + x^6 + x^4 + x^2 + x + (x^8 + x^4 + x^3 + x + 1) \\
&= x^6 + x^3 + x^2 + 1 \\
&= \texttt{0b01001101} = \texttt{0x4d};
\end{aligned}$$

thus, the round constants are

$$[\texttt{8d}, \texttt{01}, \texttt{02}, \texttt{04}, \texttt{08}, \texttt{10}, \texttt{20}, \texttt{40}, \texttt{80}, \texttt{1b}, \texttt{36}, \texttt{6c}, \texttt{d8}, \texttt{ab}, \texttt{4d}].$$

In the presentation of `MixColumns`, we show how to do this computation using numbers, instead of abstract field multiplications. The S-box is given as

$$\begin{pmatrix}
\texttt{63} & \texttt{7C} & \texttt{77} & \texttt{7B} & \texttt{F2} & \texttt{6B} & \texttt{6F} & \texttt{C5} & \texttt{30} & \texttt{01} & \texttt{67} & \texttt{2B} & \texttt{FE} & \texttt{D7} & \texttt{AB} & \texttt{76} \\
\texttt{CA} & \texttt{82} & \texttt{C9} & \texttt{7D} & \texttt{FA} & \texttt{59} & \texttt{47} & \texttt{F0} & \texttt{AD} & \texttt{D4} & \texttt{A2} & \texttt{AF} & \texttt{9C} & \texttt{A4} & \texttt{72} & \texttt{C0} \\
\texttt{B7} & \texttt{FD} & \texttt{93} & \texttt{26} & \texttt{36} & \texttt{3F} & \texttt{F7} & \texttt{CC} & \texttt{34} & \texttt{A5} & \texttt{E5} & \texttt{F1} & \texttt{71} & \texttt{D8} & \texttt{31} & \texttt{15} \\
\texttt{04} & \texttt{C7} & \texttt{23} & \texttt{C3} & \texttt{18} & \texttt{96} & \texttt{05} & \texttt{9A} & \texttt{07} & \texttt{12} & \texttt{80} & \texttt{E2} & \texttt{EB} & \texttt{27} & \texttt{B2} & \texttt{75} \\
\texttt{09} & \texttt{83} & \texttt{2C} & \texttt{1A} & \texttt{1B} & \texttt{6E} & \texttt{5A} & \texttt{A0} & \texttt{52} & \texttt{3B} & \texttt{D6} & \texttt{B3} & \texttt{29} & \texttt{E3} & \texttt{2F} & \texttt{84} \\
\texttt{53} & \texttt{D1} & \texttt{00} & \texttt{ED} & \texttt{20} & \texttt{FC} & \texttt{B1} & \texttt{5B} & \texttt{6A} & \texttt{CB} & \texttt{BE} & \texttt{39} & \texttt{4A} & \texttt{4C} & \texttt{58} & \texttt{CF} \\
\texttt{D0} & \texttt{EF} & \texttt{AA} & \texttt{FB} & \texttt{43} & \texttt{4D} & \texttt{33} & \texttt{85} & \texttt{45} & \texttt{F9} & \texttt{02} & \texttt{7F} & \texttt{50} & \texttt{3C} & \texttt{9F} & \texttt{A8} \\
\texttt{51} & \texttt{A3} & \texttt{40} & \texttt{8F} & \texttt{92} & \texttt{9D} & \texttt{38} & \texttt{F5} & \texttt{BC} & \texttt{B6} & \texttt{DA} & \texttt{21} & \texttt{10} & \texttt{FF} & \texttt{F3} & \texttt{D2} \\
\texttt{CD} & \texttt{0C} & \texttt{13} & \texttt{EC} & \texttt{5F} & \texttt{97} & \texttt{44} & \texttt{17} & \texttt{C4} & \texttt{A7} & \texttt{7E} & \texttt{3D} & \texttt{64} & \texttt{5D} & \texttt{19} & \texttt{73} \\
\texttt{60} & \texttt{81} & \texttt{4F} & \texttt{DC} & \texttt{22} & \texttt{2A} & \texttt{90} & \texttt{88} & \texttt{46} & \texttt{EE} & \texttt{B8} & \texttt{14} & \texttt{DE} & \texttt{5E} & \texttt{0B} & \texttt{DB} \\
\texttt{E0} & \texttt{32} & \texttt{3A} & \texttt{0A} & \texttt{49} & \texttt{06} & \texttt{24} & \texttt{5C} & \texttt{C2} & \texttt{D3} & \texttt{AC} & \texttt{62} & \texttt{91} & \texttt{95} & \texttt{E4} & \texttt{79} \\
\texttt{E7} & \texttt{C8} & \texttt{37} & \texttt{6D} & \texttt{8D} & \texttt{D5} & \texttt{4E} & \texttt{A9} & \texttt{6C} & \texttt{56} & \texttt{F4} & \texttt{EA} & \texttt{65} & \texttt{7A} & \texttt{AE} & \texttt{08} \\
\texttt{BA} & \texttt{78} & \texttt{25} & \texttt{2E} & \texttt{1C} & \texttt{A6} & \texttt{B4} & \texttt{C6} & \texttt{E8} & \texttt{DD} & \texttt{74} & \texttt{1F} & \texttt{4B} & \texttt{BD} & \texttt{8B} & \texttt{8A} \\
\texttt{70} & \texttt{3E} & \texttt{B5} & \texttt{66} & \texttt{48} & \texttt{03} & \texttt{F6} & \texttt{0E} & \texttt{61} & \texttt{35} & \texttt{57} & \texttt{B9} & \texttt{86} & \texttt{C1} & \texttt{1D} & \texttt{9E} \\
\texttt{E1} & \texttt{F8} & \texttt{98} & \texttt{11} & \texttt{69} & \texttt{D9} & \texttt{8E} & \texttt{94} & \texttt{9B} & \texttt{1E} & \texttt{87} & \texttt{E9} & \texttt{CE} & \texttt{55} & \texttt{28} & \texttt{DF} \\
\texttt{8C} & \texttt{A1} & \texttt{89} & \texttt{0D} & \texttt{BF} & \texttt{E6} & \texttt{42} & \texttt{68} & \texttt{41} & \texttt{99} & \texttt{2D} & \texttt{0F} & \texttt{B0} & \texttt{54} & \texttt{BB} & \texttt{16}
\end{pmatrix},$$

which is computed from

$$\begin{aligned} p_i &= (x+1)^i \mod (x^8+x^4+x^3+x+1) \\ q_i &= ((x+1)^{-1})^i \\ &= (x^7+x^6+x^5+x^4+x^2+x)^i \mod (x^8+x^4+x^3+x+1). \end{aligned}$$

We define the rotation of a word as

$$\texttt{ROTL}_8(v,s) = ((v \ll s) \,\texttt{mod}\, 2^8) \mid (v \gg 8-s).$$

The indexes of the S-box is then defined as

$$\texttt{SBOX}[p_i] = q_i \oplus \texttt{ROTL}_8(q_i,1) \oplus \texttt{ROTL}_8(q_i,2) \oplus \texttt{ROTL}_8(q_i,3) \oplus \texttt{ROTL}_8(q_i,4) \oplus \texttt{0x63}.$$

The two main operations for AES are `MixColumns` and `ShiftRows`. A round consists of applying the S-box as a substitution table, then doing `ShiftRows` and `MixColumns`, and finally exclusive OR the result with the round key. This process is reversible, but here we are only looking at the encryption part of the protocol. Verification of decryption should not be too hard, as we already have all the components and a proof that they follow the specification. Furthermore, we have also tested the implementations on the test vectors given in [75].

Matrix Representation

In the following we assume that we are working with a big-endian representation; that is, the number

```
0x3c4fcf098815f7aba6d2ae2816157e2b
```

is stored as

```
3c 4f cf 09   88 15 f7 ab   a6 d2 ae 28   16 15 7e 2b,
```

whereas the NIST document is working in little endian, i.e.,

```
2b 7e 15 16   28 ae d2 a6   ab f7 15 88   09 cf 4f 3c.
```

We can map one to the other by reversing the order of the bytes. Many of the operations in AES represent the `u128` numbers as a matrix. This is done by first splitting the input `u128` into four `u32` words, i.e.,

$$a = a_3\, a_2\, a_1\, a_0,$$

and then each word is split into four

$$a_i = a_{i,3}\, a_{i,2}\, a_{i,1}\, a_{i,0}.$$

The computation is done by

$$a_i = (a \gg i \cdot 32) \bmod 2^{32}$$

and

$$a_{j,i} = (a_j \gg i \cdot 8) \bmod 2^8.$$

This gives us the following matrix of u8s from a u128

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}.$$

The u128 is rebuilt by combining four u32s rebuilt from the u8s, i.e.,

$$a = a_{3,3}\ a_{3,2}\ a_{3,1}\ a_{3,0} \quad a_{2,3}\ a_{2,2}\ a_{2,1}\ a_{2,0} \quad a_{1,3}\ a_{1,2}\ a_{1,1}\ a_{1,0} \quad a_{0,3}\ a_{0,2}\ a_{0,1}\ a_{0,0}.$$

This is the inverse of the split and can be computed by

$$a_i = (a_{i,3} \ll 3 \cdot 8) \mid (a_{i,2} \ll 2 \cdot 8) \mid (a_{i,1} \ll 1 \cdot 8) \mid a_{i,0}$$

and

$$a = (a_3 \ll 3 \cdot 32) \mid (a_2 \ll 2 \cdot 32) \mid (a_1 \ll 1 \cdot 32) \mid a_0.$$

SubBytes

We define helper functions

$$\texttt{RotWord}(v) = (v \gg 8) \mid ((v \ll 3 \cdot 8) \bmod 2^{32})$$

and

$$\texttt{SubWord}(v) = \texttt{SBOX}[v_3] \quad \texttt{SBOX}[v_2] \quad \texttt{SBOX}[v_1] \quad \texttt{SBOX}[v_0].$$

We can then define the substitution phase as

$$\texttt{SubBytes}(a) = \texttt{SubWord}(a_3) \quad \texttt{SubWord}(a_2) \quad \texttt{SubWord}(a_1) \quad \texttt{SubWord}(a_0).$$

Thus, the SubBytes function indexes into the SBOX for each u8 in the matrix representation of the input u128 word. Indexing into the SBOX with a u8 can be seen as using the 4 most significant bits as an x-coordinate (row) and the 4 less significant bits as the y-coordinate (column).

MixColumns

The mix columns algorithm computes a mixing of a column c by

$$r_{c,i} = a_{c,i} \oplus (a_{c,0} \oplus a_{c,1} \oplus a_{c,2} \oplus a_{c,3}) \oplus \texttt{xtime}(a_{c,i} \oplus a_{c,(i+1) \bmod 4}),$$

where xtime is defined as

$$\texttt{xtime}(v) = ((v \ll 1) \bmod 2^8) \oplus (((v \gg 7)\ \&\ \texttt{0x01}) \cdot \texttt{0x1b}).$$

If we use the notation

$$a \bullet b = a \cdot b \; \texttt{mod} \; (x^8 + x^4 + x^3 + x + 1)$$

to represent multiplication in the field, then $\texttt{xtime}(v)$ represents $v \bullet x$. Thus, each of the round constants can also be computed by repeated application of `xtime`, i.e., by $rc_i = \texttt{xtime}^i(1)$, as $v \bullet x^i$ can be computed by $\texttt{xtime}^i(v)$. More generally we can define multiplication with a polynomial as

$$a \bullet b = \bigoplus_{i=0}^{7} \texttt{xtime}^{b_i \cdot i}(x),$$

where $\texttt{xtime}^0(a) = a$. We can compute p_i by

$$p_{i+1} = \texttt{xtime}(p_i) \oplus p_i,$$

as this computes $p_i \bullet x + p_i = p_i \bullet (x+1)$ and q_i as

$$q_{i+1} = \texttt{xtime}^7(q_i) \oplus \texttt{xtime}^6(q_i) \oplus \texttt{xtime}^5(q_i) \oplus \texttt{xtime}^4(q_i) \oplus \texttt{xtime}^2(q_i) \oplus \texttt{xtime}(q_i).$$

We can compute the round constants and S-box using the following Python code:

```python
from functools import reduce

def rotl8(x,shift):
  return ((x << shift) % 2**8) | (x >> (8-(shift)))
def xtime(v):
  return ((v<<1)%2**8) ^ (((v >> 7) & 0x01) * 0x1b)
def xtime_i(v,i):
  return reduce(lambda x, y: xtime(x), range(i), v)
def mult_poly(p,v):
  return reduce(
    lambda i, j: i ^ j,
    [xtime_i(v,i) for i in range(8) if p & 2**i])

def sbox():
  sbox = [0 for i in range(256)]
  sbox[0] = 0x63
  p = 1; q = 1
  while True:
    p = mult_poly(0b00000011, p)
    q = mult_poly(0b11110110, q)
    sbox[p] = q ^ rotl8(q,1) ^ rotl8(q,2) ^ rotl8(q,3) ^
    rotl8(q,4) ^ 0x63
    if (p == 1): break
  return sbox

def rcon():
  return [0x8d] + [xtime_i(1,i) for i in range(14)]
```

To complete the construction of the mix columns, we rebuild each column

$$r_c = r_{c,3}\ r_{c,2}\ r_{c,1}\ r_{c,0}$$

and finally the full number

$$r = r_3\ r_2\ r_1\ r_0.$$

ShiftRows

We take the matrix representation of the `u128`. We then shift each row by its index, starting at zero. That is, we leave the first row as is, shift the second row by one, shift the third row by two, and shift the fourth row by three. This can be illustrated as

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \overset{shift}{\Longrightarrow} \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,2} & a_{2,3} & a_{2,0} & a_{2,1} \\ a_{3,3} & a_{3,0} & a_{3,1} & a_{3,2} \end{pmatrix}.$$

Verifying the implementation of shift rows can be done directly by computation, as we are comparing the observable behavior of two pieces of executable code, which is one of the advantages of having an executable specification. This check caught an index error[1] in the Jasmin implementation of AES, where $a_{2,3}$ was incorrectly given the value of $a_{2,2}$.

AES encryption

Encryption is done by taking a message and a key

$$\texttt{enc}_{\texttt{AES}}(m,k) = \texttt{MixColumns}(\texttt{SubBytes}(\texttt{ShiftRows}(m))) \oplus k,$$

except for the last round of AES, where we do not apply `MixColumns`

$$\texttt{encLast}_{\texttt{AES}}(m,k) = \texttt{SubBytes}(\texttt{ShiftRows}(m)) \oplus k.$$

For the full AES computation, the encryption function is then called in 10 rounds (for AES-128), finishing with the `encLast` function

$$\texttt{aes}(m, k_{0..11}) = \texttt{encLast}(\texttt{enc}(\cdots \texttt{enc}(\texttt{enc}(m \oplus k_0, k_1), k_2) \cdots, k_{10}), k_{11}).$$

Key Expansion

The goal of key expansion is to generate a sequence of keys from the original. The key expand function first splits the key into bytes, then computes

$$k_{i+1,0} = k_{i,0} \oplus \texttt{SubWord}(\texttt{RotWord}(k_{i,3})) \oplus \texttt{RCON}_i$$

[1]`https://github.com/jasmin-lang/jasmin/pull/429`

$$\begin{aligned} k_{i+1,1} &= k_{i,1} \oplus k_{i+1,0} \\ k_{i+1,2} &= k_{i,2} \oplus k_{i+1,1} \\ k_{i+1,3} &= k_{i,3} \oplus k_{i+1,2}, \end{aligned}$$

which are then combined by

$$k_{i+1} = k_{i+1,3}\ k_{i+1,2}\ k_{i+1,1}\ k_{i+1,0}.$$

The round keys are computed from the key expansion function, iteratively starting with $k_0 = key$, i.e., the users private key.

Intel AES-NI Instructions

An alternative definition comes from the x86 instruction set in the Intel architecture [47] with

$$\texttt{KeyExpand}(rc, k, t) = \texttt{KeyCombine}(k, \texttt{aeskeygenassist}(k, rc), t).$$

Here, the returned value is the next k and t, starting with $k_0 = key$ and $t_0 = 0$. The AES key generation assistance (aeskeygenassist) function computes part of the AES key. There exist x86 instructions for these[2] [47]. Given 128-bit input x and 8-bit input b we compute

$$\begin{aligned} y_0 &= \texttt{SubWord}(x_1) \\ y_1 &= \texttt{RotWord}(y_0) \oplus b \\ y_2 &= \texttt{SubWord}(x_3) \\ y_3 &= \texttt{RotWord}(y_2) \oplus b, \end{aligned}$$

and then the aeskeygenassist function returns the recombination of these $y = y_3\ y_2\ y_1\ y_0$. We define some helper functions available in the Intel architecture

$$\texttt{vpshufd1}(s, o)_i = (s \gg (32 \cdot ((o \gg (2 \cdot i)) \bmod 4)))_0,$$

$$\texttt{vpshufd}(s, o) = \{\texttt{vpshufd1}(s, o)\}_{i \in 0..3},$$

and

$$\texttt{vshufps}(s_1 s_2, o) = \{\texttt{vpshufd1}(s_2, o)\}_{i \in 2..3} \quad \{\texttt{vpshufd1}(s_1, o)\}_{i \in 0..1}.$$

The KeyCombine function takes the round key k and two values a and b. It then computes a couple intermediate values:

$$a' = \texttt{vpshufd}(a, \texttt{0xFF}),$$

[2]https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

$$b' = \texttt{vshufps}(b, k, 16),$$
$$k' = k \oplus b',$$

and

$$b'' = \texttt{vshufps}(b', k', 140).$$

Then we can define

$$\texttt{KeyCombine}(k, a, b) = (k' \oplus a' \oplus b'', b'').$$

This is intended to compute the same value, but can now use specialized hardware to compute the aeskeygenassist function.

4.3 The Paper

We will now present the paper "The Last Yard: Foundational End-to-End Verification of High-Speed Cryptography" [48] published at the Conference on Certified Programs and Proofs (CPP'24). The paper introduces our framework for validating cryptographic primitives. That is writing an executable specification, proving security about it, and proving an implementation adheres to the specification. The tools we use here are Hacspec/Hax, Jasmin, and SSProve. A summary and a framing of the paper can be found in §4.4.

The Last Yard: Foundational End-to-End Verification of High-Speed Cryptography

Philipp G. Haselwarter*
Aarhus University
Denmark
philipp@haselwarter.org

Benjamin Salling Hvass*
Aarhus University
Denmark
bsh@cs.au.dk

Lasse Letager Hansen*
Aarhus University
Denmark
letager@cs.au.dk

Théo Winterhalter
Inria
France
theo.winterhalter@inria.fr

Cătălin Hriţcu
MPI-SP
Germany
catalin.hritcu@mpi-sp.org

Bas Spitters
Aarhus University
Denmark
spitters@cs.au.dk

Abstract

The field of high-assurance cryptography is quickly maturing, yet a unified foundational framework for end-to-end formal verification of efficient cryptographic implementations is still missing. To address this gap, we use the Coq proof assistant to formally connect three existing tools: (1) the Hacspec emergent cryptographic specification language; (2) the Jasmin language for efficient, high-assurance cryptographic implementations; and (3) the SSProve foundational verification framework for modular cryptographic proofs. We first connect Hacspec with SSProve by devising a new translation from Hacspec specifications to imperative SSProve code. We validate this translation by considering a second, more standard translation from Hacspec to purely functional Coq code and generate a proof of the equivalence between the code produced by the two translations. We further define a translation from Jasmin to SSProve, which allows us to formally reason in SSProve about efficient cryptographic implementations in Jasmin. We prove this translation correct in Coq with respect to Jasmin's operational semantics. Finally, we demonstrate the usefulness of our approach by giving a foundational end-to-end Coq proof of an efficient AES implementation. For this case study, we start from an existing Jasmin implementation of AES that makes use of hardware acceleration and prove that it conforms to a specification of the AES standard written in Hacspec. We use SSProve to formalize the security of the encryption scheme based on the Jasmin implementation of AES.

*Equal Contribution.

CCS Concepts: • **Theory of computation** → **Program verification**; **Program specifications**; • **Security and privacy** → *Symmetric cryptography and hash functions*; **Logic and verification**;

Keywords: high-assurance cryptography, formal verification, computer-aided cryptography, AES, Coq

ACM Reference Format:
Philipp G. Haselwarter, Benjamin Salling Hvass, Lasse Letager Hansen, Théo Winterhalter, Cătălin Hriţcu, and Bas Spitters. 2024. The Last Yard: Foundational End-to-End Verification of High-Speed Cryptography. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '24), January 15–16, 2024, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3636501.3636961

1 Introduction

Research on high-assurance cryptography recently led to significant practical success, with formally verified cryptographic code making its way into mainstream libraries and software products [7, 14, 16, 19, 21, 34, 37, 41, 42]. Since in this area missing any bugs can have a serious security impact, some additionally try to reduce the trusted computing base of their verification tools for cryptographic code and construct foundational proofs [5, 14, 21, 25, 29, 32]. Such foundational proofs rely on strong logical foundations—usually by working in a proof assistant like Coq or Isabelle/HOL—and only on standard, clearly stated assumptions. Yet despite good progress in this direction, a couple of important gaps remain for foundational end-to-end cryptographic verification.

First, there is a specification gap. Currently, cryptographic primitives and protocols are specified only using informal pseudo-code in the standards (e.g., in IETF RFCs). The Hacspec language [15, 31] aims to improve this, by making the code of these cryptographic specifications executable, which allows them to also serve as reference implementations that can be used as oracles for testing more efficient implementations. Hacspec is a simple subset of the Rust programming language, which aims to be understandable for both ordinary developers and cryptographers. Hacspec can be translated

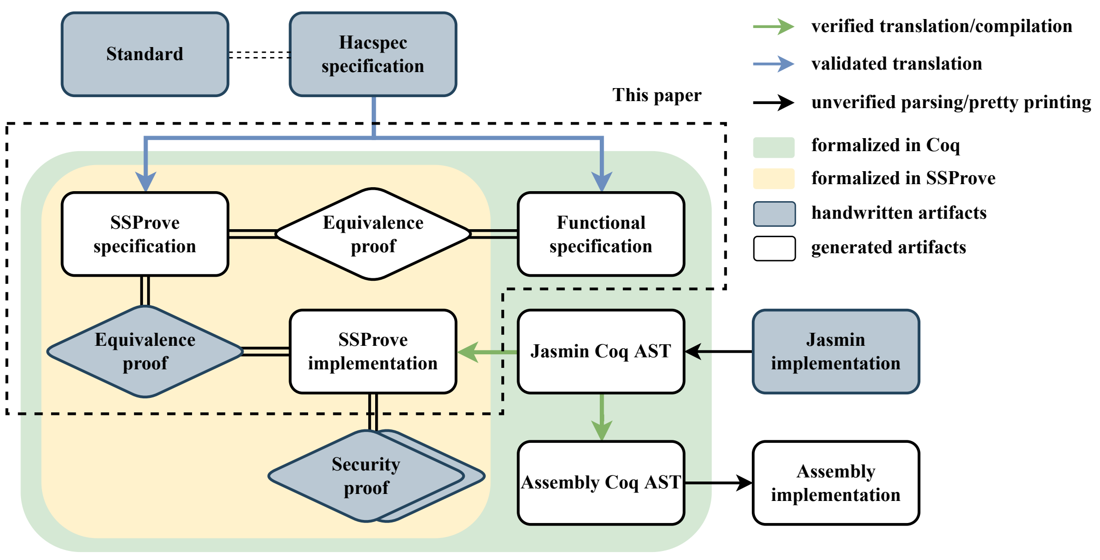


Figure 1. Proposed workflow for foundational end-to-end verification of high-speed cryptography

to the typed, purely functional language of proof assistants such as Coq, EasyCrypt, or F*, which allows sharing cryptographic specifications across these proof assistants.

Such translations from Hacspec to a proof assistant produce a functional specification that can be used for verifying cryptographic code. In such a verification one often starts by proving the equivalence of the functional specification with an imperative specification, which is closer to the code of an implementation to be verified [5]. We automate this step by devising a new translation from Hacspec to imperative programs in SSProve, which is a recent foundational verification framework for modular cryptographic proofs in Coq [1, 25]. Moreover, we provide translation validation infrastructure for automatically proving the equivalence of the code produced by these two translations.

Second, there is an implementation gap. Implementing cryptography in C has pitfalls: (1) unverified C compilers cannot be trusted to be always correct and secure [39], and (2) the CompCert verified C compiler does not perform aggressive optimizations and generates code with efficiency comparable only to GCC at optimization level 1 [2, 28]. Moreover, even aggressively optimized C programs are sometimes not fast enough since they cannot make use of special instructions providing hardware acceleration for cryptographic primitives (e.g., Intel AES-NI [23]). So cryptographic primitives are often implemented directly in assembly, at the cost of loss of abstraction, clarity, and convenience. The Jasmin language [4] was proposed as a solution to this problem. It is a language for implementing cryptographic primitives combining structured control flow with assembly instructions, which allows one to produce efficient code for x86 and ARM. Moreover, the Jasmin compiler comes with Coq proofs that it preserves the semantics of the source code [4, 5] and that it does not introduce timing side-channel attacks [6].

In the fundamental 'Last Mile' paper [5], Jasmin programs are given semantics in Coq and compiled with a compiler verified in Coq, but reasoning about the security and correctness of Jasmin programs is done only after an *unverified* translation to EasyCrypt. In this paper, we close this gap by providing a *verified* translation from Jasmin to SSProve. Staying in Coq not only allows us to reduce the trusted computing base, but it also facilitates reusing existing mathematical Coq libraries [3, 30] to verify Jasmin implementations.

Contributions. We formally connect three existing tools, Hacspec, Jasmin, and SSProve, into a unified foundational Coq framework for the end-to-end verification of high-speed cryptography (Figure 1). This includes the following novel contributions:

- We devise a new translation from Hacspec specifications to imperative SSProve code. In contrast to the existing functional translations, it allows us to reason about the *stateful* behavior of Hacspec.
- We provide a translation validation infrastructure, which automatically produces Coq proofs of program equivalence between the results of this imperative translation and those of a more standard functional translation. We do this by performing a *compositional* symbolic evaluation, relating imperative code to its mathematical model.
- We connect the Jasmin language and verified compiler to SSProve, by providing a translation of Jasmin source code to SSProve. We overcome the challenge created by the fact that SSProve only supports global state while Jasmin programs can use local state.

- We give a mechanized proof in Coq that this translation from Jasmin to SSProve preserves Jasmin's operational semantics.
- We demonstrate the usefulness of our approach on a case study by producing a foundational end-to-end Coq proof of an efficient AES implementation. We start from an existing Jasmin implementation of AES using the Intel AES-NI instructions for hardware acceleration [23] and prove (in ~2500 lines of Coq code) that it conforms to a Hacspec specification of the AES standard [20]. Finally, we instantiate a PRF-based symmetric encryption scheme with the implementation of AES, and use SSProve to prove IND-CPA security of this scheme under the (standard) assumption that AES is a pseudo-random function (PRF).

Outline. We start by giving an overview of our methodology and illustrating it on a very simple one-time pad example (Section 2). We then discuss necessary background (Section 3), before diving in the two formal connections we establish: the one between Hacspec and SSProve (Section 4), the other between Jasmin and SSProve (Section 5). We finally present the AES case study (Section 6), before discussing related (Section 7) and future work (Section 8).

2 Foundational End-to-End Verification, from Specification to Efficient Implementation

In this section, we first give an overview of our methodology following Figure 1 and then demonstrate its workings on the very simple example of a one-time pad. At a high level, we provide a foundational framework for proving the equivalence between a specification in Hacspec and an efficient, low-level implementation in Jasmin, by translating both to imperative SSProve programs. Once translated, we relate the programs and prove properties about them in Coq using SSProve's probabilistic relational Hoare logic.

2.1 Workflow

The workflow is illustrated in Figure 1. Starting from an informal description, such as an official ***standard*** (e.g., published by NIST or IETF) for a cryptographic primitive or protocol, one uses a subset of Rust with a simple, well-defined semantics to develop a ***Hacspec specification***.[1] We then automatically translate this specification in two ways:

- once to the purely functional language of Coq; this translation produces a ***functional specification***; and
- once to the imperative language of SSProve; this translation produces an ***SSProve specification***.

The functional translation [35] targets Coq's mathematical language, and is similar to the usual functional semantics of Hacspec in F* and EasyCrypt. The imperative translation serves as a stepping stone towards a Jasmin implementation, which is inherently imperative.

We then perform translation validation [33] to automatically construct an ***equivalence proof*** in Coq, which formally shows that the functional and imperative Hacspec translations produce equivalent SSProve code from a given Hacspec program. More specifically, we prove that in a clean state, the program produced by the imperative translation will return the same value as the one produced by the functional translation. The proof is conducted in SSProve's relational Hoare logic (see Section 3.3.4).

The second part of our framework concerns efficient cryptographic implementations written in Jasmin. We implemented a translation from Jasmin to the imperative language of SSProve and proved that it preserves semantics. This proof is entirely mechanized in Coq, which is possible because both SSProve and Jasmin already have formal semantics in Coq [4–6, 25]. So from the same ***Jasmin implementation*** (1) we can produce an ***assembly implementation*** using the existing Jasmin compiler, which was proved in Coq to preserve the source language semantics [4, 25]; and (2) we can obtain the ***Jasmin Coq AST*** of the Jasmin implementation, which we then translate to an ***SSProve implementation*** in a way that we proved to preserve semantics.

We are now in a position to reason about the SSProve implementation using the relational probabilistic Hoare logic of SSProve. On the one hand, we can conduct an ***equivalence proof*** between the SSProve implementation obtained from Jasmin and the SSProve specification obtained from Hacspec. On the other hand, we can connect the translated Hacspec specification with ***security proofs*** done in the SSProve framework. These proofs use the standard security games from the cryptographic literature [13, 24, 36, 38].

Formal Guarantees. By combining the correctness theorems of the Jasmin compiler and our translation to SSProve, we get the following corollary: for any function in a Jasmin program with well-defined semantics, there exists a corresponding compiled assembly function and translated SSProve function with the same semantics, i.e., which maps equal arguments to equal results and which modifies memory in an equivalent manner. In particular, the semantics of the SSProve and assembly functions agree and we can prove the properties of the assembly program by analyzing the corresponding SSProve program; probabilistic properties cannot however be carried to the assembly level, since the semantics there are deterministic. Note that we inherit some assumptions from the compiler proof (e.g., assuming sufficient stack-space) and introduce some in the translation proof (e.g., functions cannot use while-loops), see also Section 5.

[1] In fact, Hacspec is directly used in the upcoming hash-to-curve IETF standard [22] for writing a reference implementation.

2.2 One-time Pad Example

We now illustrate this methodology using a very simple example: We construct a one-time pad (OTP) from exclusive or (XOR). This toy example should convey intuition on the methodology. A more interesting case study for the AES encryption scheme is presented in Section 6. This section is to get an idea of the workflow and the ideas, however, we will introduce the background theory in more detail in Section 3.

2.2.1 Specification. The Hacspec specification for `xor` takes two 64-bit words as input, puts them into mutable variables, and computes their XOR (`^` in Hacspec). The result is stored in a mutable variable[2], which is then returned.

```
fn xor(w1 : u64, w2 : u64) -> u64 {
  let mut x : u64 = w1;
  let mut y : u64 = w2;
  let mut r : u64 = x ^ y;
  r
}
```

Our framework produces an automatic translation of this code to the following Coq function of type `both`.

```
Definition hacspec_xor (w1 : int64) (w2 : int64) :=
  letbm x_0 : int64 loc( x_0_loc ) := w1 in
  letbm y_1 : int64 loc( y_1_loc ) := w2 in
  letbm r_2 : int64 loc( r_2_loc ) := x_0 .^ y_1 in
  r_2.
```

Here `letbm` stands for "let bind mutable". The type `both` can be projected both to pure Coq and to SSProve code (see Section 4.3), resulting in the following two functions:

```
Definition hacspec_xor_pure x y := x .^ y.

Definition hacspec_xor_state (x y : int64) :=
  put x_loc := x ;;
  temp_x ← get x_loc ;;
  put y_loc := y ;;
  temp_y ← get y_loc ;;
  put r_loc := int_xor temp_x temp_y ;;
  temp_r ← get r_loc ;;
  ret temp_r.
```

For achieving translation validation, the `both` type also carries an equivalence proof between these two functions:

```
∀ x y, ⊢ { λ '(h₀, h₁), ⊤ }
           hacspec_xor_state x y ≈
           ret (hacspec_xor_pure x y)
         { λ '(v₀, h₀) '(v₁, h₁), v₀ = v₁ }.
```

2.2.2 Jasmin Implementation. A Jasmin implementation of `xor` could look as follows.

```
export fn xor(reg u64 x, reg u64 y) -> reg u64 {
  reg u64 r;
  r = x;
  r ^= y;
  return r;
}
```

It takes two register-allocated arguments `x` and `y` (as indicated by the `reg` keyword) and writes the XOR of `x` and `y` into the return register `r`.

2.2.3 SSProve Implementation. The next step is to translate the Jasmin code to the following SSProve function.

```
Definition JXOR id0 w1 w2 :=
  put x := w1 ;;
  put y := w2 ;;
  put r := w1 ⊕ w2 ;;
  r1 ← get r ;;
  ret r1.
```

While this readable code is not the literal output of the translation, it is the result of some careful (but semi-automated and verified) unfolding and simplification. The produced code also takes an "identifier", `id0`, as input: this determines which locations on the heap it will use for its local memory. This technical detail will be explained in Section 5 and can safely be ignored for now.

2.2.4 Equivalence of Implementation and Specification. Now that we have both translations to SSProve, we can prove that they are equivalent in our program logic.

```
Theorem xor_equiv : ∀ id0 w1 w2,
  ⊢ { λ '(h₀, h₁), ⊤ }
    JXOR id0 w1 w2 ≈ hacspec_xor_state w1 w2
  { λ '(v₀, h₀) '(v₁, h₁), v₀ = v₁ }.
```

The precondition is a predicate over the two initial heap states and the postcondition is a predicate over the two final heaps and values. The notion of equivalence we use here to relate the two functions only requires the return values v_0, v_1 of the two programs to be equal, provided we run them both on the same inputs. In particular, we do not make assumptions or restrict how the two programs use the heaps h_0, h_1. The programs are thus allowed to use different locations to store their intermediate values. This theorem is proved using the rules of the relational program logic of SSProve [1].

2.2.5 Security Proof for the OTP Implementation. We now prove perfect cryptographic security of the Jasmin implementation of OTP using XOR. To this end, we first need to define some terminology. In SSProve a *package* is a finite set of procedures that might contain calls to external procedures. The set it implements is called its *export interface* and the set on which it depends its *import interface*. A *game* is a package with no imports and a *game pair* is a pair of games that export the same procedures. These can be used to model cryptographic games, e.g., a game pair might consist of a

[2]This use of mutability is for illustrative purposes only.

real encryption scheme and an oracle: these have the same interfaces but different implementations.

For OTP we define the game pair consisting of an implementation of OTP using the Jasmin code and an implementation which is obviously secure. The Jasmin game is the package JOTP_real exporting the single procedure:

```
Definition JOTP id0 m :
  k_val ⟵ sample uniform ('word n) ;;
  JXOR id0 m k_val.
```

We already have a security proof for the package OTP_real exporting the single procedure:

```
Definition OTP m :
  k_val ⟵ sample uniform ('word n) ;;
  ret m ⊕ k_val.
```

This game is already proven to be indistinguishable under chosen plaintext attack from an implementation where the message is chosen at random. The statement and proof are in the SSProve library. This is done by proving that, when the input is disregarded and a random message is encrypted, the advantage of an attacker in distinguishing between OTP_real and a game OTP_ideal is zero.

If we can prove that JOTP_real is perfectly indistinguishable from OTP_real, then we can combine the two results using the triangle inequality for advantages of games (Lemma 1 in the SSProve paper [1]) and prove that an adversary also cannot distinguish between JOTP_real and OTP_ideal, i.e., the Jasmin implementation is IND-CPA. That is, we only need to prove the following theorem.

```
Lemma JOTP_OTP_perf_ind id : JOTP_real id ≈₀ OTP_real.
```

Here $\approx_0$ means that the advantage of an adversary trying to distinguish between the two games is zero. To prove this lemma we use Theorem 1 from the SSProve paper [1], which allows us to conclude if we can prove the following code equivalence for all *m* and some *stable invariant* inv:

```
⊢ { λ '(s₀, s₁), inv (s₀, s₁) }
  JOTP id0 m ≈  OTP m
{ λ '(b₀, s₀) '(b₁, s₁), b₀ = b₁ ∧ inv (s₀, s₁) }.
```

For the precise definition of stable invariant see Section 4.2 of the SSProve paper [1]. In our case, we can use the invariant heap_ignore, which asserts that both heaps are preserved during execution if the locations used by JXOR are ignored.

Combining this result with the already established security of OTP_real we get security of JOTP_real.

```
Theorem unconditional_secrecy_jas : ∀ LA A,
  fdisjoint LA xor_locs → ValidPackage LA
    [interface #val #[i1] : 'word → 'word ]
    A_export A →
  Advantage IND_CPA_jasmin A = 0.
```

That is, for all valid adversaries A with a matching interface, and all regions of adversarial memory LA, if the adversary cannot use the same locations as JXOR then their advantage in distinguishing between JOTP_real and OTP_ideal is zero.

3 Background & Technical Preliminaries

3.1 Hacspec

Hacspec is a High Assurance Cryptography SPECification language [15, 27, 31] aiming to provide a common language to programmers, cryptographers and proof engineers. It proposes to make future internet standards, such as those published by IETF and NIST, machine-readable. Hacspec is a subset of Rust which makes it executable and accessible to cryptographic engineers.

The Hacspec language was carefully crafted to have a functional semantics, in which assignments are translated to let-expressions. The Hacspec tool comes with functional translations to the purely functional languages of several proof assistants, currently F⋆, Coq, and EasyCrypt. As such it is a convenient tool to share specifications across proof assistants.[3] Hacspec also comes with an operational semantics [31], but since the semantics is not formalized in the backends, the functional translation cannot be verified against it. Instead, this translation constitutes the authoritative semantics. This motivates our choice to relate our imperative translation via translation-validation[4].

Currently, all Hacspec backends use a functional semantics. However, both in EasyCrypt and in Coq/SSProve, one could also choose to use a translation to an embedded imperative language. This can be seen as one of the benefits of Hacspec, as anyone familiar with either functional or imperative coding paradigms will understand the Hacspec specification. We will explain how to do so in Section 4.

3.2 Jasmin

Jasmin [4] is a low-level language designed for implementing high-speed cryptography, with a verified compiler backend supporting the x86 and ARM architectures. The language has a formal big-step operational semantics in Coq. The Jasmin compiler is also implemented and verified in Coq, in the sense that it preserves the semantics of the Jasmin source [4, 5] and also that it does not introduce timing side-channel attacks [6]. We give a condensed overview of Jasmin, focusing on the aspects that are interesting for the sake of our discussion, and limiting the explanation to a few representative examples. For more details please see the Jasmin paper [4].

[3]This also allows one to combine code generated from different proof assistants. For example, one could combine a hash function from F⋆ and an elliptic curve implementation from Coq, both of which would be specified in Hacspec, verified, and then extracted to C (or Rust, or ASM). This is the methodology proposed in the libcrux library [27].

[4]The operational semantics of Hacspec would be a good target for future formalization.

3.2.1 The Language. Jasmin is an imperative language with structured control flow in the form of loops, conditionals, and procedure calls. Jasmin has types for booleans, integers, bit-words of various sizes, and arrays. Despite these high-level features, the Jasmin compiler produces predictable assembly code, which enables efficient and secure cryptographic implementations. For instance, the programmer can use architecture-specific assembly instructions and can specify whether procedure-local variables should be stored in registers (using the `reg` keyword) or on the stack (using the `stack` keyword). Jasmin's operational semantics was carefully crafted to hide low-level details such as the distinction between the storage types `reg` and `stack`. Our correctness theorem for the translation from Jasmin to SSProve, like Jasmin's compiler correctness theorem, is proven with respect to this operational semantics, and we can thus safely ignore such distinctions.

A Jasmin program P consists of a list of non-recursive function definitions, associating to each function name f a list of variables used for arguments $P(f)_{param}$, variables used for returning results $P(f)_{res}$, and a command, i.e., a sequence of instructions $P(f)_{body}$ for the body of the function.

Instructions include assignments, operators, conditionals, for and while loops, and function calls. Expressions occurring in instructions include variable and array access, arithmetic and logical operators, as well as assembly operations such as shifts, increments, *etc.*

3.2.2 Jasmin State. Jasmin features both global and local state, denoted by a pair (m, ρ) of a *global memory* m and local *variable map* ρ. A variable is local when it is declared within a function, and global when declared at the top level. We will write $\rho[\,\cdot\,]$ and $\rho[\,\cdot \leftarrow \cdot\,]$ respectively for local variable map lookup and update. For global state, we will write $m[\,\cdot\,]_i$ and $m[\,\cdot \leftarrow \cdot\,]_i$ for lookup and storage of *size* i, given in bits (possible values are 8, 16, 32, 64, 128, 256). Global state is indexed by integers (pointers) and local state by variables (strings). Note that looking up memory in Jasmin can fail, so we will abuse notation by denoting by $m[p]_i = v$ that v is stored at p in m *and* that it is valid to make a read of size i at p in m. We will do the same for writes.

3.2.3 Jasmin Operational Semantics. The operational semantics of Jasmin is mostly standard. A judgment of the form $\langle\, c \mid (m, \rho) \,\rangle \Downarrow (m', \rho')$ means that for an initial state (m, ρ), execution of the command c terminates in the final state (m', ρ'), and $\langle\, e \mid (m, \rho) \,\rangle \Downarrow_{exp} v$ means that the expression e evaluates to the value v under state (m, ρ) (expressions can only read, not modify the state). All judgments are implicitly parametrized by an ambient program (i.e., list of function definitions), which will not be mentioned explicitly unless required. For instance, in the rule for assigning a local variable in Figure 2 we start by evaluating the expression e to v. We then look up the type α of the variable x, and perform a truncation[5] of v at type α, yielding v' compatible with the type of x. Finally, we update the local state to $\rho[x \leftarrow v']$, while the global state remains unchanged.

ASSGN

$$\frac{\langle\, e \mid (m, \rho) \,\rangle \Downarrow_{exp} v \qquad \alpha = ty(x) \qquad v' = \|v\|^{\alpha}}{\langle\, x = e \mid (m, \rho) \,\rangle \Downarrow (m, \rho[x \leftarrow v'])}$$

FUNCALL

$$\frac{\begin{array}{c}\langle\, e_i \mid (m, \rho_0) \,\rangle \Downarrow_{exp} v_i \qquad \text{for } i = 1, \ldots, k \\ \langle\, f(v_1, \ldots, v_k) \mid m \,\rangle \Downarrow_{call} \langle\, (w_1, \ldots, w_n) \mid m' \,\rangle \\ \langle\, x_j = w_j \mid (m', \rho_{j-1}) \,\rangle \Downarrow (m', \rho_j) \qquad \text{for } j = 1, \ldots, n\end{array}}{\langle\, x_1, \ldots, x_n = f(e_1, \ldots, e_k) \mid (m, \rho_0) \,\rangle \Downarrow (m', \rho_n)}$$

CALLRUN

$$\frac{\begin{array}{c}\text{let } \rho_0 = \emptyset \text{ and } c = P(f)_{body} \\ \text{and let } y_i = (P(f)_{param})_i \text{ and } x_j = (P(f)_{res})_j \\ \langle\, y_i = v_i \mid (m, \rho_{i-1}) \,\rangle \Downarrow (m, \rho_i) \qquad \text{for } i = 1, \ldots, k \\ \langle\, c \mid (m, \rho_k) \,\rangle \Downarrow (m', \rho') \\ w_j = \|\rho'[x_j]\|^{ty(x_j)} \qquad \text{for } j = 1, \ldots, n\end{array}}{\langle\, f(v_1, \ldots, v_k) \mid m \,\rangle \Downarrow_{call} \langle\, (w_1, \ldots, w_n) \mid m' \,\rangle}$$

Figure 2. Excerpt of Jasmin operational semantics

The main subtlety for translating Jasmin to SSProve arises from function calls and their treatment of local state. The execution of function calls in Jasmin is split into two rules. The perspective of the caller is captured by FUNCALL: We evaluate the arguments e_i and perform the call to the function f according to the callee's perspective. We obtain a new global state m' and store the resulting values w_j in the caller-local variables x_j. Jasmin's type checker guarantees that the number of returned values equals the number of variables. Crucially, when switching from caller to callee, we *retain the local state* ρ_0 and pass only the global state m to CALLRUN as witnessed by the use of $\Downarrow_{call}$ relating pairs of instructions and global memories and values and global memories.

To describe the callee perspective, we write ρ_0 for the empty local state, and c, y_i, and x_j for the body, parameter-, and result-variables of f respectively. Each argument v_i is stored in the local variable y_i according to the definition of parameters of $P(f)_{param}$. We then execute c from state (m, ρ_k), yielding (m', ρ'). We obtain the values $w_1, \ldots, w_n$ by reading the result variables x_j from the local state ρ' and truncating as necessary. Finally, the local state ρ' is discarded, and the result values and updated global state m' are returned.

[5]This truncation only exists at the high level to mimic the implicit truncations happening at the assembly level. In practice, the types of v and x mostly agree and the truncation can be simplified away.

3.3 SSProve

SSProve is a Coq library for modular cryptographic proofs introduced by Abate et al. [1]. We only review the concepts needed to understand the current paper. More details can be found in the extended version of the SSProve paper [25].

3.3.1 Code. In this paper, we de-emphasize the probabilistic capabilities of SSProve, as they are not currently reflected in Jasmin. Thus, for our purposes, SSProve essentially embeds a stateful language inside Coq using a monad called `raw_code`. In `raw_code A` one can (1) embed any pure value `x` of type `A` using `ret x`, (2) read from a memory location ℓ to a variable `x`, and use `x` in a continuation `k`, written `x ← get ℓ ;; k x`, (3) write a value `v` to a memory location ℓ and then continue with `k`, written `put ℓ ;; k`, (4) sequentially combine `u : raw_code X` and `k : X → raw_code A` using the bind operator that we write `x ← u ;; k x`. It is also possible to sample from a distribution `D` in this monad using `x ← sample D ;; k x` as shown in Section 2.

3.3.2 Memory Model. Memory locations consist of a natural number and a type that together serve as an index in a global shared memory. This global state is represented as a map from locations to values. We say that a state is valid for a set of (typed) locations when all locations point to values of the matching type. Note that to be able to use the type in the key of the memory, we must in fact use codes of types; since SSProve is built for probabilistic programs, these codes represent types on which one may build (discrete) distributions. In type-theoretic terms, they encode a universe of datatypes `choice_type` which represents a subset of mathcomp's `choiceType` [30, §8.3]. For the purposes of our translation, we use a modified version of SSProve where `choice_type` is extended to include sums, words and lists. This allows us to encode all the types needed to represent Hacspec and Jasmin programs. Memory is simulated using a structure we call `heap`, essentially a map from locations to values. We would like to stress the fact that in SSProve the memory is *global*, in contrast to Jasmin's function local state. Thus, one must take care to generate code without overlapping locations. We address this in Section 5.

For a heap h, location ℓ and value v, we will write $h[l]$ and $h[\ell \leftarrow v]$ for heap lookup and storage (as for Jasmin state).

3.3.3 Packages. Another defining feature of SSProve is that of packages. Packages are used extensively to compose modular security games in the style of state-separating proofs [17]. Since our methodology allows us to reuse existing security proofs [25], we will not get into the details of security proofs, so we only introduce packages briefly. Packages are collections of procedures that can all refer to the same set of locations and invoke certain procedures that are part of an *import interface*. The signature of this collection defines the *export interface* of the package. Packages can thus be combined modularly to create bigger packages. For instance, a package can be linked to another that implements its import interface or they can be composed in parallel to export the union of their respective export interfaces.

3.3.4 Relational Hoare Logic. Finally, SSProve features a (probabilistic) relational Hoare logic that allows us to prove the relational properties of programs. Once again, we will focus on the stateful but deterministic fragment. In this program logic, we prove judgments of the form

$$\vdash \{\phi\}\ c_0 \ \sim\ c_1\ \{\psi\}$$

where c_0 and c_1 are two code pieces we compare and ϕ and ψ are respectively a pre- and a postcondition relating (1) the initial heaps (for ϕ); (2) the final heaps and final return values of both code pieces (for ψ). For deterministic code, this is equivalent to: for all initial memory states m_0 and m_1 such that $\phi(m_0, m_1)$ holds, running c_i in state m_i will yield final state m_i' and final value v_i such that $\psi(v_0, m_0')(v_1, m_1')$ holds.

SSProve comes with a number of rules for this logic and provides tactics to facilitate writing proofs. Moreover, one can fall back on the semantics above to prove judgments [25].

3.4 Interoperability

For the sake of getting Hacspec, Jasmin, and SSProve to interact smoothly, we had to extend each of them in a minor way. We did not, however, make any modifications to the core projects that would change the interpretation of any of the statements that can be found in the published literature.

Specifically, besides the translations which constitute the core contributions of this work, we made the following additions. For SSProve, we added sum types to represent the result types of Hacspec. We also added bitwidth-indexed machine words as well as lists. For Jasmin, we added the ability to pretty-print the Coq abstract syntax tree of a parsed Jasmin program, and we added the definition of the Intel AES-NI instructions [23] to Jasmin's x86 semantics, since the AES-NI instructions are used in the AES case study. Hacspec remained unchanged.

4 Hacspec & SSProve

Hacspec facilitates proving the correctness of efficient implementations with respect to a specification by translating it to multiple proof assistants. We further this goal by adding a translation from any valid Hacspec specification to SSProve. This imperative translation is accompanied by a pure translation, which adds a wrapper around the existing Coq translation to embed it into SSProve. We can thus compare the imperative and pure translations using SSProve's relational logic and automatically generate a proof stating that they return the same values.

4.1 The Functional Translation

The pure translation constitutes a minor modification of Hacspec's existing Coq backend [35] that we undertook to facilitate the connection to Jasmin. Coq does not provide a standard library for machine integers, so the existing backend chose the CompCert library to model machine integers [28]. Jasmin uses its own word library. In the long run, we would hope for a unified word library in the Coq ecosystem. Meanwhile, we changed the backend to use Jasmin words.

We translate for-loops as a fixed point with an accumulator of all the mutable variables changed inside the loop. Hacspec has support for early return of option or result types. We model these early returns using the option and error monad. We thus need a fold operation that respects the monadic operations to allow early returns in for-loops.

4.2 The Imperative Translation

Since we provide the first translation from Hacspec to an imperative programming language, we need to extend the information gathered in the translation from Hacspec to the various backends. SSProve needs information about what memory locations and functions are used in a given scope. To compute this, we add static dependency analysis to the Hacspec pipeline. This is done by walking the AST for every block of code and adding a unique memory location for each mutable variable. In a second pass, we then unify the memory locations used by all the local function calls, to get the total set of memory locations a function might change.

The translation evaluates arguments passed to function calls or operators before evaluating the function or operator. This is done by binding the arguments to temporary values, which are then passed to the function. This makes it easier to prove equality to another SSProve implementation, as we can first prove that all the arguments are equal, and then show that the functions agree on equal input.

A subtlety arises from the fact that Hacspec supports early return statements: `x = e?` is operationally equivalent to

```
x = match e { Some(v) => v, None => return None }
```

In particular, if `e` evaluates to `None`, the ambient function in which the statement `x = e?` occurs returns early with the result `None`. Since SSProve's `raw_code` does not support control effects, we cannot directly represent this `return`. We instead embed Rust code with early returns into the option monad. To ensure that this encoding interacts well with the effectful operations of SSProve which manipulate state, we define a special bind operation, combining the two monads.

```
Definition obind (x : raw_code (option A))
 (f : A → raw_code (option B)) : raw_code (option B)
:= t_x ← x ;;
   match t_x with Some s ⇒ f s | None ⇒ ret None end.
```

The Hacspec code we translate carries sufficient typing information to determine whether a function may return early. We leverage this information to select between this custom bind operator and SSProve's standard bind. For example:

```
x = f(v)? ; y = g(x) ; y + 2
```

is translated to the following SSProve code:

```
temp_x ← f(v) ;;
obind temp_x (λ x, temp_y ← g(x) ;; temp_y .+ 2)
```

4.3 Equivalence Between the Hacspec Translations

On the one hand, it is often easier to define and prove properties for a functional specification. On the other hand, it is easier to show an efficient imperative implementation equivalent to an imperative specification. So, it is desirable to derive an equality between the imperative and functional translations. We automatically generate such a proof, as part of the translation from the Hacspec specification. To achieve this we first define a record `both`, which has projections to a piece of code for the functional translation and for the imperative translation. It also contains the proof of equivalence for the two pieces of code. We traverse the AST building the functional translation, the imperative translation and their equivalence at the same time. This is achieved by using compositional blocks for the control structures of Hacspec.

An example of such block is the one used for `let` expression in Hacspec, where the functional translation is a functional let binding in Coq, while the imperative translation uses bind in SSProve. The equivalence can be proven using the bind rule in SSProve, since we have a proof of equality of the arguments and a proof of equality of the rest of the code bodies. Other blocks are loops, mutable let bindings (where a location is used, as shown in Section 2.2.1), early returns, operator calls, lifting pure values, etc. We can therefore get the full translation to the imperative and functional code, together with the equality between them, by chaining these compositional blocks. This also requires us to define all the library functions in Hacspec in the `both` type. Using this combined type, we can write elements in a style where the translation looks close to the original specification and can be made more readable by the notation engine of Coq.

5 Jasmin & SSProve

5.1 Memory

A major difference between the Jasmin and SSProve semantics is how memory is handled: SSProve only has a global notion of memory and Jasmin supports both global and local variables. To model local variables in SSProve, we parameterize all translated code over a "base stack frame ID" which reserves an (*a priori* unbounded) region of SSProve's global memory for local variables. Then instantiating translated code with a concrete base stack frame ID correctly assigns new stack frame IDs to all its called functions. In particular, we prove that variables translated with different stack frame IDs never overlap, i.e., translation of variables is injective w.r.t. IDs. We store the Jasmin global memory in a map (from integers to bytes) at a static location called `MEM`.

5.2 Program Translations

We now describe our translation from Jasmin to SSProve, meaning the translation of programs, but also of types, values, expressions and commands. As a first step, we use the Jasmin compiler to pretty-print the internal AST corresponding to a Jasmin source program to Coq syntax. Since this AST datatype was extracted from Coq in the first place, it amounts to 'de-extracting' it back to Coq. Our translation thus translates Coq's datatype of Jasmin programs to SSProve programs (i.e., the `raw_code` monad).

5.2.1 Types and Values. The only base types missing from SSProve's `choice_type` (the restricted set of types which a `raw_code` can return; see Section 3.3.2) were words and arrays. Following Jasmin, we use the `coqword` library's type of words, which is based on the mathcomp library [30]. We represent arrays as maps from integers to bytes. The only minor difference is our implementation of maps differs from Jasmin. Using similar types makes it easy to embed Jasmin values into SSProve values (via the identity) for all except array values. We denote the function taking Jasmin values to SSProve values by `translate_value`.

5.2.2 Expressions. For the translation of expressions (denoted `translate_pexpr`) we have to be careful and do the right casts and truncations, as dictated by the semantics of Jasmin: e.g., when looking up in an array, the index is always cast to an integer type. For the translation of function applications in expressions (additions, subtractions, *etc.*), we reused the semantics from Jasmin expressions, by transporting values back to Jasmin types, applying the operations, and then transporting back to SSProve types. Note that this transport is only non-trivial for arrays. This simplifies the proof significantly, only requiring us to prove that all operations are invariant under this transport.

5.2.3 Instructions. The main difficulty in translating instructions is translating function calls; for calls to operations we could mostly use the same solution as for expressions and for for-loops we simply iterate the translated body. To be able to call functions, we choose to let our translation keep track of previously translated functions, and only allow these to be called; this avoids cyclic calls and recursion (which are always rejected by the Jasmin compiler). Furthermore, we make sure to call these translated functions with a fresh stack frame ID to avoid collisions between local variables.

Note that we currently do not translate Jasmin while loops, as they do not have a correspondent in SSProve. This does not constitute a conceptual problem in practice, since for-loops are sufficient for most cryptographic routines.

5.2.4 Programs. We translate Jasmin programs, which map function names to function declarations (Section 3.2.1), to maps from function names to SSProve functions taking an ID and a list of inputs to SSProve code.

5.3 Unary Deterministic Judgments

SSProve originally supported only relational judgments of the form $\vdash \{\phi\}\ c_0 \sim c_1\ \{\psi\}$, as presented in Section 3.3. For the sake of our correctness theorem, we want to relate a translated Jasmin term c_0 to the value v it evaluates to, i.e., c_1 is always of the form $\texttt{ret}\ v$. Since Jasmin's semantics is deterministic, we do not need the full power of a probabilistic judgment. We thus extend SSProve and build a new unary judgment on top of the relational logic, to deal with the special case where we relate a `raw_code` with a return value: $\vdash \{\phi\}\ c \Downarrow v\ \{\psi\}$. Here ϕ is a precondition on the initial state of c, while ψ is a postcondition on the final state after running c. The postcondition no longer mentions a final state or return value for the right hand side, instead the return value v is part of the judgment. We define $\vdash \{\phi\}\ c \Downarrow v\ \{\psi\}$ as the following judgment relating c to $\texttt{ret}\ v$:

$$\vdash \{(m_0, m_1).\ \phi\ m_0\}\ c \sim \texttt{ret}\ v\ \{(a_0, m_0'), (a_1, m_1').\ \psi\ m_0' \wedge a_0 = a_1 \wedge a_1 = v\}$$

The precondition only considers the memory of the left-hand side, while the postcondition also states that both sides must produce the value v.

While this unary judgment is conceptually simpler than the relational logic, we have found it beneficial to reuse the existing theory instead of starting from scratch. An advantage of this is that we can easily leverage the rules of the relational program logic and the tactics provided by SSProve to prove unary judgments. Moreover, we establish a precise connection between the two logics by proving that whenever c is free of sampling operations, the judgment above is equivalent to saying that running c on any initial state m such that $\phi\ m$ will yield return value v and final state m' such that $\psi\ m'$. For instance, we obtain the expected rules for values, sequential composition, and writing to the heap.

$$\frac{\forall m.\ \phi\ m \Longrightarrow \psi\ m \wedge v = v'}{\vdash \{\phi\}\ \texttt{ret}\ v \Downarrow v'\ \{\psi\}}$$

$$\frac{\vdash \{\phi\}\ c \Downarrow u\ \{\xi\} \qquad \vdash \{\xi\}\ k\ u \Downarrow v\ \{\psi\}}{\vdash \{\phi\}\ x \leftarrow c\ ;;\ k\ x \Downarrow v\ \{\psi\}}$$

$$\frac{\vdash \{\lambda\, m, \exists\, m', \phi(m') \wedge m = m'[\ell \leftarrow v]\}\ r \Downarrow w\ \{\psi\}}{\vdash \{\phi\}\ \texttt{put}\ \ell\ v\ ;;\ r \Downarrow w\ \{\psi\}}$$

Other rules can also be derived straightforwardly from the definition of the unary judgment as analogues of the relational rules, which are detailed by Haselwarter et al. [25].

5.4 Correctness Theorem

We prove that our translation preserves the semantics of well-defined programs. To do this we define a relation between Jasmin memory states and SSProve memory states. First, we relate the global Jasmin memory to the "global memory map" stored on the heap in SSProve. We say that the global Jasmin state m is related to the heap h when, if one can successfully read a single byte at an address from the Jasmin memory, then one can look up the corresponding value in the "global memory map" stored at MEM on the SSProve heap:

$$m \sim h := \forall p\ v.\ m[p]_8 = v \Rightarrow h[\mathsf{MEM}][p] = v$$

To relate the local memory of Jasmin and our encoding of local memory in SSProve, we define a relation between a variable map ρ and a heap h relative to a ***stack frame ID*** ι. We write $h[x]^\iota$ for the lookup of the variable x on the heap relative to ID ι. A variable map ρ is related to the heap h w.r.t. ι if successfully looking up a variable x in ρ implies that looking up x on h relative to ι yields the same value:

$$\rho \sim_\iota h := \forall x\ v.\ \rho[x] = v \Rightarrow h[x]^\iota = v$$

Now, the relation between a Jasmin memory pair (m, ρ) (of global and local state) and an SSProve heap is not just the conjunction over all these relations, since we need to know that a function can make an arbitrary number of function calls, each with their own local state, and not run out of space on the heap. To state this we need some terminology: We say that a stack frame ID ι is ***fresh*** w.r.t. a heap h when $\rho_0 \sim_\iota h$ holds, where ρ_0 is the empty variable map. We assume that we have a prefix order $\preceq$ on stack frame IDs and say that a stack frame ID s is ***valid*** w.r.t. a heap h when all strict successors of s are fresh w.r.t. h, i.e., for all $s' \succ s$, $\rho_0 \sim_{s'} h$. Furthermore, we say that two IDs s_1 and s_2 are ***disjoint***, when there is no ID which they are both a prefix of. Concretely, we require for all IDs s that $s_1 \preceq s$ and $s_2 \preceq s$ do not both hold simultaneously. We assume that storing at disjoint ID locations preserves values: if s_1 and s_2 are disjoint then $\forall x, y,\ h[y \leftarrow v]^{s_2}[x]^{s_1} = h[x]^{s_1}$.

For a variable map ρ, two stack frame IDs ι, σ (main and *sub*-ID) and a set I of IDs we say that the tuple (ρ, ι, σ, I) is a ***stack frame***. We say that a stack frame (ρ, ι, σ, I) is ***valid*** w.r.t. a heap h when the following conditions hold: (1) σ is valid w.r.t. h, (2) $\rho \sim_\iota h$, (3) $\sigma \notin I$, (4) for all $\sigma' \in I$, $\iota \prec \sigma'$, σ' is disjoint from σ and σ' is valid w.r.t. h, (5) for all $\sigma', \sigma'' \in I$, σ' and σ'' are disjoint.

The intuition for a valid stack frame (ρ, ι, σ, I) is that ρ should be related to the main stack frame ID ι, and the sub stack frame ID σ should be a valid ID from which the current function can spawn new functions with fresh memory; I is there to keep track of which IDs are currently in use and to which variable maps they relate. Note that the set I is only needed for the proof of correctness, and is not actually used in the translation of a given program.

A ***stack*** is then a list of stack frames. The empty stack is denoted by S_0. A stack frame (ρ, ι, σ, I) is ***disjoint*** from a stack S when ι is disjoint from all sub IDs and IDs occurring in sets of the stack frames on S. A stack S is ***valid*** w.r.t. a heap h when either S is empty or $S = F :: S'$ where S' is a valid stack and F is a valid stack frame disjoint from S'.

Using these constructions we can finally define our relation on Jasmin and SSProve states. A Jasmin state pair (m, ρ) is related to the heap h w.r.t. the stack S, which we write $(m, \rho) \sim_S h$, when the following conditions hold: (1) S is valid w.r.t. h, (2) $m \sim h$, (3) ρ is the variable map at the top of the stack, i.e., the top of the stack is of the form (ρ, ι, σ, I). This relation satisfies two key lemmas, which are needed to prove the correctness of our translation.

Lemma 1 (Push empty stack frame). *If $(m, \rho) \sim_{(\rho,\iota,\sigma,I)::S} h$ and σ_1, σ_2 are two disjoint IDs with $\sigma \prec \sigma_1, \sigma_2$, then*

$$(m, \rho_0) \sim_{(\rho_0,\sigma_1,\sigma_1,\emptyset)::(\rho,\iota,\sigma_2,I)::S} h.$$

Lemma 2 (Pop stack frame). *Let $F_i = (\rho_i, \iota_i, \sigma_i, I_i)$, then if $(m, \rho_2) \sim_{F_2::F_1::S} h$ then $(m, \rho_1) \sim_{F_1::S} h$.*

These two lemmas correspond to (1) calling a function and assigning it a fresh region of memory for local state and (2) returning from a function call to its caller, accounting for the operational semantics of Jasmin function calls according to Figure 2. Note in Lemma 1 that the sub-ID of the calling stack frame, σ, is updated to a fresh ID σ_2, and that we initialize the callee frame with the same main and sub ID σ_1, since when the frame gets pushed in a function call, the callee has not invoked any further functions yet.

Using this relation, we show how our translation of Jasmin code relates to its source. For example, if we consider the function `translate_pexpr`, which translates Jasmin expressions to `raw_code`, we get the following correctness lemma.

Lemma 3. *Let v be a value, e an expression, s a Jasmin state pair and S a stack. If $\langle\, e \mid s \,\rangle \Downarrow_{\text{exp}} v$ then*

$$\vdash \{h.\ s \sim_S h\}\ \ \mathtt{translate\_pexpr}\ S\ e \Downarrow \\ \mathtt{translate\_value}\ v\ \{h.\ s \sim_S h\}$$

As evaluating expressions does not have memory side effects, the relation between Jasmin and SSProve states is preserved under expression translation.

We now prove the main theorem, which establishes the connection between *function calls* in Jasmin and in SSProve:

Theorem 1. *Let P be a Jasmin program, (m, ρ) a Jasmin state-pair, f a function name, and v_i, w_i values for $i = 1, \ldots, k$. Furthermore, let $\iota, \sigma, \sigma_1, \sigma_2$ be IDs such that σ_1 and σ_2 are disjoint and strict successors of σ. If P' is the result of translating P and $\langle\, f(v_1, \ldots, v_k) \mid m \,\rangle \Downarrow_{\text{call}} \langle\, (w_1, \ldots, w_n) \mid m' \,\rangle$ then*

$$\begin{aligned} \vdash\ & \{h.\ (m, \rho) \sim_{(\rho,\iota,\sigma,I)} h\} \\ & P'\ f\ \sigma_1\ \mathtt{translate\_values}\ (v_1, \ldots, v_k) \\ \Downarrow\ & \mathtt{translate\_values}\ (w_1, \ldots, w_n) \\ & \{h.\ (m', \rho) \sim_{(\rho,\iota,\sigma_2,I)} h\} \end{aligned}$$

The theorem states that if calling the function f in the Jasmin program P and global memory m with arguments $\vec{v}$ results in the new global memory m' and returns the values $\vec{w}$, then we can conclude two things:

1. Calling the function at a fresh ID (σ_1) and with the translation of the arguments $\vec{v}$ evaluates to the translation of the return values $\vec{w}$.
2. After calling the translated function, the global memory m' is related to heap where we have updated the sub-ID to a fresh one (from σ to σ_2).

This is the expected behavior: calling a function can change the global but not the local state. We have to update our sub-ID because the previous one is no longer fresh, as we might have stored local state inside the function call.

6 AES Example

As a larger case study of our framework, we verify the security of a Jasmin implementation of a PRF-based encryption scheme using AES and prove it equivalent to a Hacspec reference implementation. The Jasmin implementation and the general methodology for proving security are similar to the presentation in EasyCrypt [8], but we use our toolchain based on SSProve to conduct the formalization.

The workflow for proving security of our AES implementation is as follows:

1. Implement the encryption scheme in Hacspec and Jasmin.
2. Translate the two implementations to SSProve code.
3. Prove the two translations equivalent and prove security properties of the Jasmin translation.[6]

We skip implementing the Jasmin code by reusing the implementation from the EasyCrypt and Jasmin tutorial [8], which relies on the Intel AES-NI hardware acceleration instructions [23]. Our reference implementation in Hacspec is based on the NIST standard [20], and it successfully passes the corresponding public test vectors [20, 23].

For the security analysis, we prove indistinguishability under chosen plaintext attack (IND-CPA) of the AES implementation of the PRF-based symmetric encryption scheme described below. Concretely, we prove that the advantage of an adversary in distinguishing the encryption of a message from the encryption of a random message is (linearly) bounded by the advantage of the same adversary in distinguishing AES from a PRF. For details on the concrete bounds, see the SSProve journal paper [25, §2.3].

As was the case in Section 2, we do not have to write a security proof of the abstract encryption scheme from scratch, since such a proof, for an abstract PRF, is already present in the SSProve library [25, §2.3]. To connect this with our efficient implementation, we need to prove that an adversary cannot distinguish between the efficient implementation and the abstract implementation given in SSProve [25, §2.3] instantiated with a Coq implementation of AES.

As in *loc. cit.*, our definitions follow SSP methodology [17]. The PRF-based encryption scheme is given by the code:

```
Definition PRF_ENC f m :=
  k_val ⟵ kgen ;;  enc m k_val.
```

Here, `kgen` is a key generation code that uniformly samples a key on its first invocation and returns a fixed key on subsequent calls. The `enc` function is given by the code:

```
Definition enc m k :=
  r ⟵ sample uniform N ;;
  let pad := f r k in let c := m ⊕ pad in
  ret (r, c).
```

Here `f` is the function which we assume to be a PRF and which we will instantiate with AES. The PRF is used to generate a `pad` from a uniformly sampled nonce `r`; the ciphertext is computed as the `xor` of the message and the `pad`. For all functions `f : word → word → word` we denote the game consisting of the single export `PRF_ENC f` by `PRF_real f`.

We reuse the SSProve proof [1, §2.3] by showing that `PRF_real aes` is perfectly indistinguishable from the same scheme with `enc` replaced by the translated Jasmin code.

The high-level structure of the security analysis of the implementation is as follows:

1. Write an intermediate imperative implementation directly in SSProve code.
2. Write a functional implementation directly in Coq.
3. Prove the equivalence between the intermediate implementation and the functional implementation.
4. Prove the equivalence between the translated implementation and the intermediate implementation.
5. Connect the equivalences to the existing security proof of the abstract encryption scheme.

Steps (1) and (2) can also be copied almost verbatim from the EasyCrypt development: the syntactic similarities of the EasyCrypt and SSProve codes make the translation very straightforward. For the proofs in steps (3) and (4) we can reuse some parts, e.g., the loop invariants, but in general the differences in the programming languages and the underlying proof assistants require new proofs.

6.1 Translation

As mentioned in Section 2, we start by printing the Coq ASTs of all the involved functions during Jasmin compilation. Then we use the translation described in Section 5 to obtain SSProve code for each function used in the implementation.

[6]Here we deviate slightly from the intended workflow from Section 2 by doing the security proof on the implementation instead of the specification. The reason for this is simply that parts of security proof about the Jasmin implementation were already completed when the Hacspec specification was added to the project.

6.2 Specification

Next, we write intermediate specifications for the Jasmin functions. Compared to the example in Section 2, these correspond to the pure Coq XOR function. As mentioned, we take inspiration from the specifications in the EasyCrypt and Jasmin tutorial [8]. This step removes translation artefacts (e.g., compiler-generated memory locations) and allows us to focus on proving the underlying logical statements.

6.3 Equivalences for Intermediate Code

Then we prove that our intermediate implementations are equivalent to functional (stateless) Coq functions. The statements we prove are generally of the form:

$$\vdash \{(m_0, m_1).\ \phi\ (m_0, m_1)\}\ c\ i\ \sim \\ \texttt{ret}\ (f\ i)\ \{(a_0, m_0'), (a_1, m_1').\ \phi\ (m_0', m_1') \land a_0 = a_1\}$$

where i is arbitrary input, c is the intermediate SSProve code and f is the pure Coq function. Note that we also prove that these equivalences preserve the precondition ϕ; for the equivalences to hold we usually have to assume that ϕ is stable w.r.t. memory locations used by c.

Even though f is usually stateless, we have to keep the heap of the right-hand side in mind, since it might be relevant in certain contexts; otherwise we could have used the unary judgments of Section 5.3.

6.4 Equivalences for Translated Code

When reasoning about the code generated by our translation from Jasmin to SSProve, we have to prove equivalences of the following form:

$$\vdash \{(m_0, m_1).\ \phi\ (m_0, m_1)\}\ P'\ F\ id\ i\ \sim \\ c\ i\ \{(a_0, m_0'), (a_1, m_1').\ \phi\ (m_0', m_1')\ \land\ a_0 = a_1\}$$

where P' is the translated Jasmin program, i is an arbitrary input, id is a stack frame ID, F is the function name in the Jasmin program and c is the intermediate code.

Once we have proven such an equivalence, we can reuse it in proofs where F appears as a called function. It is therefore important that the equivalences are parametric in id. We also want to preserve the precondition ϕ and again we have to assume that ϕ is stable w.r.t. the locations of F and c. However, there is one issue here: the locations set of F is not straightforward to compute and might also be rather large. Instead we require that ϕ is stable w.r.t. *all possible* locations used by ϕ, i.e., locations stored using an id' with prefix id ($id \preceq id'$). This turns out to be a sufficient and reasonably manageable invariant to preserve.

6.5 Connecting AES to the PRF Security Proof

The encryption function of which we want to prove the security can be implemented in Jasmin as:

```
fn enc(reg u128 n,reg u128 k,reg u128 p) -> reg u128 {
  reg u128 mask, c;
  mask = aes(n, k);
  c = xor(mask, p);
  return(c);
}
```

We translate it into SSProve as `JENC` and use it in the following security game, supplying the random nonce `r`:

```
Definition JPRF_real id0 m :=
  k_val ⟵ kgen ;;
  r ⟵ sample uniform N ;;
  res ⟵ JENC id0 k_val r m ;;
  ret (r, res)
```

We then prove it perfectly indistinguishability from a similar scheme `CPRF_real` which uses an intermediate, simplified SSProve encryption function, `ENC`, in place of `JENC`.

We establish the indistinguishability by applying Theorem 1 of the SSProve paper [1]. We thus have to find a stable invariant that is preserved by a run of each of these schemes and prove that they return equal values. We prove a slight generalization of the version that theorem. Before, the invariant was required to be stable w.r.t. the *finite sets* of locations used by the program. Moreover, these sets were assumed to be disjoint from the state of the adversary. We now only require the invariant to be stable w.r.t. some *arbitrary* sets of locations assumed to be disjoint from the state of the adversary. In particular, the sets can be infinite.

Thanks to this generalization we can apply the theorem when one of the programs is the output of our translation, since we do not have to provide the concrete set of locations used by the program, but instead we can use an infinite over-approximation. We thus obtain the following.

Theorem `JPRF_perf_ind id` : `JPRF_real id` $\approx_0$ `CPRF_real`.

We prove that `CPRF_real` is perfectly indistinguishable from `PRF_real aes` using the original SSProve Theorem 1 as we have better control over which locations are used.

Theorem `CPRF_perf_ind` : `CPRF_real` $\approx_0$ `PRF_real aes`.

Combining these two theorems, we get the following: the advantage of any adversary, which uses locations disjoint from `JENC` and from the intermediate encryption schemes, in distinguishing between `JPRF_real` and `PRF_ENC` is 0. This we can then combine with the result from the SSProve paper [1, Section 2.3] which states that `PRF_ENC` is IND-CPA secure up to the advantage of an adversary against `aes` as a PRF.

7 Related Work

The use of formal verification for cryptography has been intensely investigated, and Barbosa et al. [7] give an overview. More narrowly, work related to SSProve can be found in the extended version of the SSProve paper [25]. In this section, we survey the closest related work to ours in this space.

CertiCrypt [11] is the earliest framework for reasoning about cryptographic code in Coq, but is no longer maintained. FCF [32] is a more recent foundational Coq framework for cryptographic proofs. It was used together with VST to verify the C implementations of HMAC in OpenSSL [14] and mbedTLS [41]. Our work is similar in that we prove the security and correctness of the Jasmin implementation of AES. While FCF could have been a reasonable option for us, we chose SSProve because it is under active development, uses the well-developed mathcomp [30] and mathcomp-analysis libraries [3], and supports modular proofs.

EasyCrypt [9, 10] is a proof assistant and verification tool specifically designed for game-based cryptographic proofs. Its good integration with automatic theorem provers (e.g., SMT solvers) is helpful for large proofs, even though it comes at a cost in terms of trusted computing base. The program logics of CertiCrypt and EasyCrypt come with native support for reasoning about function calls. This was not available in SSProve before and addressing this is one of the contributions of the present work (see Section 5.1).

In the fundamental 'Last Mile' paper [5] Jasmin programs are given semantics in Coq and the correctness of the Jasmin compiler is proved in Coq with respect to this semantics. As a realistic case study, they use EasyCrypt to prove the security and correctness of a Jasmin implementation of SHA3, relying on an unverified translation from Jasmin to EasyCrypt. In the present work, we bridge this gap by providing a verified translation from Jasmin to SSProve.

CryptHOL [12] is a foundational framework for game-based proofs that uses the theory of relational parametricity to achieve automation in Isabelle/HOL. However, unlike FCF and EasyCrypt, CryptHOL has so far not been used for the verification of efficient programs, as far as we are aware.

Schwabe et al. [37] prove the correctness of the C implementation of X25519 in TweetNaCl using VST. Protzenko et al. [34] verify an impressive library of cryptographic code in F*. Fiat-Crypto [21] is a foundational tool that can *generate* verified efficient implementations of finite field arithmetic. These works are focused on correctness though and do not consider cryptographic security.

Currently, there is no formal specification for the complete Rust language. The Hacspec semantics can be seen as a precise semantics for a non-controversial subset of Rust. Similar proposals, but for much larger subsets of Rust, include those of Ho and Protzenko [26] and Denis et al. [18].

8 Future Work

Jasminify [40] is a python tool that simplifies the process of calling Jasmin code from Rust. After compiling a program, the Rust object file is replaced with the Jasmin object file. However, Jasminify does not come with any correctness guarantees. We have shown how to prove the equivalence of a Rust (Hacspec) implementation for AES with a Jasmin program. Hacspec is expressive enough to implement high-level cryptographic protocols. For such protocols, we now have a safe way to replace its cryptographic primitives by optimized Jasmin ones, as we know that their source-level semantics agree. As future work, one could try to test this toolchain, by using Jasminify, proving equivalence between the Hacspec and Jasmin implementations and then benchmarking to see what kind of performance gains one can achieve.

In concurrent work, libcrux [27] provides a library of verified implementations from different frameworks; and combines them with a safe Rust API. For example, it starts with a Hacspec reference implementation of HMAC and HKDH, and replaces their hash functions with optimized Jasmin implementations. It was proved [5] in EasyCrypt that the SHA3 implementation indeed implements a hash-function, but a formal connection with Hacspec is still missing. It would be exciting to use our framework to formally verify some of the replacements done in libcrux.

The Jasmin language is still under active development. In the present work, we devised a verified translation for the published version of the language [4]. It would be interesting to extend our work with language features that were added to Jasmin concurrently to our work.

Acknowledgements

We are very grateful to François Dupressoir for feedback on an earlier version of this article.

This work was in part supported by the Concordium Blockchain Research Center at Aarhus University, by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation, by the German Federal Ministry of Education and Research BMBF (grant 16KISK038, project 6GEM), and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972.

References

[1] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. 2021. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. (2021). https://eprint.iacr.org/2021/397

[2] AbsInt. [n. d.]. Factsheet: CompCert C Compiler. Available at https://www.absint.com/factsheets/factsheet_compcert_c_web.pdf. https://www.absint.com/factsheets/factsheet_compcert_c_web.pdf

[3] Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Kazuhiko Sakaguchi, and Pierre-Yves Strub. 2021. mathcomp-analysis. Analysis library compatible with Mathematical Components. https://github.com/math-comp/analysis

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1807–1823. https://doi.org/10.1145/3133956.3134078

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves

Strub. 2020. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 965–982.

[6] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2022. Enforcing Fine-Grained Constant-Time Policies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 83–96. https://doi.org/10.1145/3548606.3560689

[7] Manuel Barbosa, Gilles Barthe, Karthikeyan Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2019. SoK: Computer-Aided Cryptography. *IACR Cryptol. ePrint Arch.* 2019 (2019), 1393. https://eprint.iacr.org/2019/1393

[8] Manuel Barbossa, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Pierre-Yves Strub, and Tiago Oliveira. 2022. EasyCrypt and Jasmin Tutorial. https://formosa-crypto.gitlab.io/news/2022-06-07/sibenik Šibenik.

[9] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures (Lecture Notes in Computer Science, Vol. 8604)*. Springer, 146–166. https://doi.org/10.1007/978-3-319-10082-1_6

[10] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *CRYPTO (Lecture Notes in Computer Science, Vol. 6841)*. Springer, 71–90. https://doi.org/10.1007/978-3-642-22792-9_5

[11] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *POPL*. 90–101.

[12] David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. 2020. CryptHOL: Game-Based Proofs in Higher-Order Logic. *J. Cryptol.* 33, 2 (2020), 494–566. https://doi.org/10.1007/s00145-019-09341-z

[13] Mihir Bellare and Phillip Rogaway. 2004. Code-Based Game-Playing Proofs and the Security of Triple Encryption. *IACR Cryptol. ePrint Arch.* (2004), 331. http://eprint.iacr.org/2004/331

[14] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *24th USENIX Security Symposium*. USENIX Association, 207–221. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer

[15] Karthikeyan Bhargavan, Franziskus Kiefer, and Pierre-Yves Strub. 2018. hacspec: Towards Verifiable Crypto Standards. In *Security Standardisation Research - 4th International Conference, SSR 2018, Darmstadt, Germany, November 26-27, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11322)*, Cas Cremers and Anja Lehmann (Eds.). Springer, 1–20. https://doi.org/10.1007/978-3-030-04762-7_1

[16] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 917–934. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond

[17] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. 2018. State Separation for Code-Based Game-Playing Proofs. In *ASIACRYPT*. Springer International Publishing, Cham, 222–249. https://eprint.iacr.org/2018/306

[18] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering*, Adrian Riesco and Min Zhang (Eds.). Springer, 90–105.

[19] Jason A. Donenfeld. [n. d.]. WireGuard: Formal Verification. Available at https://www.wireguard.com/formal-verification/. https://www.wireguard.com/formal-verification/

[20] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. 2001. Advanced Encryption Standard (AES). https://doi.org/10.6028/NIST.FIPS.197

[21] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *IEEE S&P*. https://doi.org/10.1109/SP.2019.00005

[22] Armando Faz-Hernandez, Sam Scott, Nick Sullivan, Riad S. Wahby, and Christopher A. Wood. 2022. *Hashing to Elliptic Curves*. Internet-Draft draft-irtf-cfrg-hash-to-curve-16. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-irtf-cfrg-hash-to-curve/16/ Work in Progress.

[23] Shay Gueron. 2012. White Paper: Intel® Advanced Encryption Standard (AES) New Instructions Set. https://www.intel.com/content/www/us/en/developer/articles/tool/intel-advanced-encryption-standard-aes-instructions-set.html

[24] Shai Halevi. 2005. A plausible approach to computer-aided cryptographic proofs. *IACR Cryptol. ePrint Arch.* (2005), 181. http://eprint.iacr.org/2005/181

[25] Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Cătălin Hrițcu, Kenji Maillard, and Bas Spitters. 2023. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. *ACM Trans. Program. Lang. Syst.* 45, 3, Article 15 (jul 2023), 61 pages. https://doi.org/10.1145/3594735

[26] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust Verification by Functional Translation. *Proc. ACM Program. Lang.* 6, ICFP, Article 116 (2022), 31 pages. https://doi.org/10.1145/3547647

[27] Franziskus Kiefer, Karthikeyan Bhargavan, Lucas Franceschino, Denis Merigoux, Lasse Letager Hansen, Bas Spitters, Manuel Barbosa, Antoine Séré, and Pierre-Yves Strub. 2023. HACSPEC: a gateway to high-assurance cryptography. In *RWC23*.

[28] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert – a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

[29] Andreas Lochbihler, S. Reza Sefidgar, David A. Basin, and Ueli Maurer. 2019. Formalizing Constructive Cryptography using CryptHOL. In *CSF*. IEEE, 152–166. https://doi.org/10.1109/CSF.2019.00018

[30] Assia Mahboubi and Enrico Tassi. 2021. Mathematical components. Online book. https://math-comp.github.io/mcb/

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

[32] Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9036)*, Riccardo Focardi and Andrew C. Myers (Eds.). Springer, 53–72. https://doi.org/10.1007/978-3-662-46666-7_4

[33] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1384)*, Bernhard Steffen (Ed.). Springer, 151–166. https://doi.org/10.1007/BFb0054170

[34] Jonathan Protzenko and Bryan Parno. 2019. EverCrypt cryptographic provider offers developers greater security assurances. Microsoft Research Blog. https://www.microsoft.com/en-

us/research/blog/evercrypt-cryptographic-provider-offers-developers-greater-security-assurances/

[35] Mikkel Milo Rasmus Holdsbjerg-Larsen, Bas Spitters. 2022. A Verified Pipeline from a Specification Language to Optimized, Safe Rust. CoqPL. https://cs.au.dk/~spitters/CoqPL22.pdf

[36] Mike Rosulek. 2021. The Joy of Cryptography. Online textbook. http://web.engr.oregonstate.edu/~rosulekm/crypto/

[37] Peter Schwabe, Benoît Viguier, Timmy Weerwag, and Freek Wiedijk. 2021. A Coq proof of the correctness of X25519 in TweetNaCl. In *2021 34th CSF*. 1–16. https://doi.org/10.1109/CSF51468.2021.00023

[38] Victor Shoup. 2004. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptol. ePrint Arch.* (2004), 332. http://eprint.iacr.org/2004/332

[39] Laurent Simon, David Chisnall, and Ross Anderson. 2018. What you get is what you C: Controlling side effects in mainstream C compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 1–15.

[40] Juriaan van Drunen. 2021. Calling Jasmin from Rust. https://gitlab.com/Jur/jasminify

[41] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. 2017. Verified Correctness and Security of mbedTLS HMAC-DRBG. In *CCS'17*. ACM, 2007–2020. https://doi.org/10.1145/3133956.3133974

[42] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *ACM Conference on Computer and Communications Security*. ACM, 1789–1806. http://eprint.iacr.org/2017/536

Received 2023-09-19; accepted 2023-11-25

4.4 Summary

In this paper, we presented how to verify efficient implementation written in Jasmin against specifications written in the Hacspec subset. This is achieved by translating both to SSProve and verifying their (functional) equivalence. Using the SSProve library in Rocq, we can prove security properties about the specification, which then hold for the implementation by transitivity.

4.5 The SSProve Backend of Hax

SSProve is built over the *choice type* from mathematical components (MathComp) [66, Section 8.3]. These are types with a choice operator. As these are not the native types of Rocq, we cannot, e.g., use the inductive type construction mechanism of Rocq, as the types generated are not directly choice types, though possibly equivalent to one.

We translate enums as indexed types $\prod_{n \in \mathbb{N}_{fin}} (args_n \to T)$; projections give us the constructors. We can do dependent matching for pattern matching on the enum type. Records are defined as telescoped tuples. We define the function with implicit arguments, thus allowing the arguments to be given in any order. We define notation for updating the records; this could be replaced by existing solutions from coq-record-update [22]. Primitives are mapped to their choice type counterpart (unit, bool, (machine) words). Functions and items are defined top-level as Rocq definitions.

Instead of doing this remapping, we could use some meta programming (e.g., ELPI [87], MetaRocq [1]) or directly encode the equivalence in the translation to automatically build or derive the type constructs as choice types. This is exactly what Trocq [28, 29] is being developed to do.

Since SSProve has an imperative language, we can actually translate mutable variables directly. This currently requires that we state which variables each function might affect. This is done by collecting all variables that are directly used and then walking the dependency graph (a DAG), taking the union of all variables from dependent functions. However, most of the time reasoning about locations and mutability is not necessary, and doing the functionalization is okay.

4.6 The Dual Translation of SSProve

The SSProve backend of Hax is doing two translations simultaneously. This dual translation can be explained in a couple different styles. It can be seen as *translation validation* (as explained in the paper), where the equivalence proof between the two shows the correctness of the imperative translation. The translation can alternatively be interpreted as a *realization* of the imperative translation. We will explain these interpretations of the translations in the following subsections.

Translation Validation

To look at the dual translation as translation validation [79], we let the imperative translation into SSProve represent the generated target code, and let the functional translation represent the source code. Then the equivalence can be seen as a refinement relation validating the correctness of the translation. The generation of the equivalence proof verifies that the imperative code has the same semantics as the functional translation.

Realizability

Since the dual translation is defined per language construct, we can see the equivalence as a realizability interpretation [10, 12, 61, 77]. The equivalence proof is then simply a statement realizing the relation between the real and imperative translation. If we formalize the semantics or convert the two translations into deep embeddings, then we can formalize this realizability.

Monadic Verification Condition Generator (mvcgen)

An alternative way of handling the dual translation is using a monadic verification condition generator (mvcgen) [43, 44, 83], which is done for the L∀∃N backend of Hax. Here imperative code is constructed using `do` notation, and then all goals are generated and automatically discharged. As this method is more general, it could make sense to encode it into Hax and allow translation into the imperative monad directly. This could also be an alternative phase in the Hax engine, as the generation of goals and reduction to functional code could happen as part of the translation instead of having to do it after translation. However, this would break the formalization, as the reduction would happen in Hax, where it is not formalized.

Chapter 5

Formal Security and Functional Verification of Cryptographic Protocol Implementations in Rust

We will start by giving a general introduction to TLS 1.3; a more detailed description can be found in the paper. Then we introduce the paper, followed by a technical deep dive into the key schedule security proofs with some hints about how to apply the formalization to related projects, i.e., MLS.

5.1 Transfer Layer Security (TLS 1.3)

The goal of the transfer layer security (TLS 1.3) [80] protocol is to securely exchange a set of cryptographic keys, which is used to encrypt further communication. TLS 1.3 forms the basis for most networking and is a large part of hypertext transfer protocol secure (HTTPS), which is used by most websites. There have been *many attacks* on older versions of TLS, and post-quantum attacks are becoming more realistic; thus, the construction of TLS 1.3 is focusing on using *post-quantum* secure primitives and *verifying the security*. These factors make TLS 1.3 a very interesting formalization target, as bugs and possible attacks have a very big impact.

The effort to formalize TLS is also part of Project Everest [14, 88], which tries to make a formalized version of the full HTTPS stack.

5.2 The Paper

A summary of the paper is made in §5.3 for convenience. This is the cryptology ePrint archive version [17] of the paper, as the paper is accepted at Computer and Communications Security (CCS'25), but not yet published.

Formal Security and Functional Verification of Cryptographic Protocol Implementations in Rust

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

[1]Cryspen, Paris, France
[2]Aarhus University, Aarhus, Denmark

Abstract

We present an effective methodology for the formal verification of practical cryptographic protocol implementations written in Rust. Within a single proof framework, we show how to develop machine-checked proofs of diverse properties like runtime safety, parsing correctness, and cryptographic protocol security. All analysis tasks are driven by the software developer who writes annotations in the Rust source code and chooses a backend prover for each task, ranging from a generic proof assistant like F⋆ to dedicated crypto-oriented provers like ProVerif and SSProve. Our main contribution is a demonstration of this methodology on `Bert13`, a portable, post-quantum implementation of TLS 1.3 written in Rust and verified both for security and functional correctness. To our knowledge, this is the first security verification result for a protocol implementation written in Rust, and the first verified post-quantum TLS 1.3 library.

1 High-Assurance Cryptographic Protocols

The last decade has been a fertile time for the design and deployment of advanced cryptographic schemes and protocols, motivated by a variety of reasons ranging from the Snowden revelations to the popularity of cryptocurrencies. This trend promises to continue with new standards for post-quantum cryptography and new efforts around privacy-preserving machine learning, which will undoubtedly require novel protocol designs and fresh implementations.

Cryptographic protocol libraries, like OpenSSL[1], libsignal[2], and Bitcoin Core[3], have come to occupy an increasingly large part of the trusted computing base of modern computer systems, and are consequently held to a high standard. Any bug in these codebases is treated as a potentially costly vulnerability. Hence, the current period of rapid change raises concerns about the quality and security of all the new protocol code that is being developed and deployed.

In this work, we demonstrate a methodology for building high-assurance implementations of cryptographic protocols, where different core components can be formally verified for the desired security and correctness guarantees, using some of the most practical, state-of-the-art verification techniques available today.

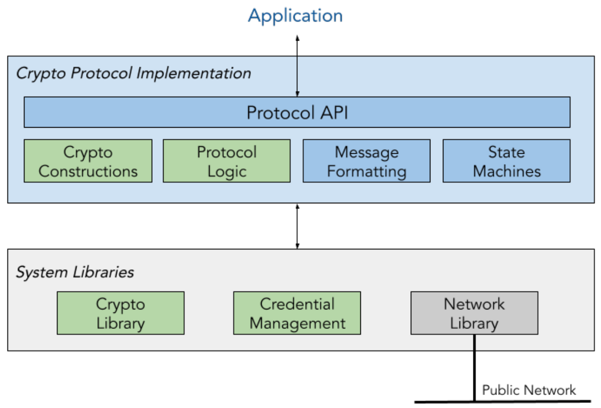


Figure 1: Crypto Protocol Implementation Components

Key Components of Protocol Implementations. Figure 1 depicts the high-level structure of a crypto protocol implementation.

The protocol relies on several system libraries: a *cryptographic library* that implements standard crypto algorithms; a *credential management library* that handles the retrieval, validation, and storage of long-term keys and credentials, such as X.509 certificates, private keys, and pre-shared keys; and a *networking library* that sends and receives messages over the untrusted network.

The protocol implementation itself consists of: protocol-specific *cryptographic constructions* that may compose multiple cryptographic algorithms; the core *protocol logic* that handles protocol message construction and processing; one or more *state machines* to keep track of protocol progress; and *message formatting* code to serialize and deserialize both public messages and internal cryptographic inputs. The protocol implementation combines these components to provide an API that can be used by the Application.

[1]`https://openssl-library.org/`
[2]`https://signal.org/docs/`
[3]`https://bitcoin.org/en/bitcoin-core/`

Bugs and Attacks. Each of these protocol components is security-critical and has a long history of attacks and vulnerabilities.

For example, consider implementations of the Transport Layer Security (TLS) protocol [54]. Prior works have found attacks on the specialized cryptographic constructions implemented in TLS [3, 7], allowing attackers to decrypt application messages. Other works have found flaws in the design and implementation of the protocol logic [15, 2], which weakened the expected authentication or confidentiality properties. Devastating state machine bugs found in TLS implementations allowed for all the protocol guarantees to be bypassed [12]. Ambiguities in the TLS message formats resulted in attacks on the authentication guarantees of the protocol [48, 19]. Of course, bugs are also frequently found in the libraries TLS depends on, e.g. in X.509 validation [36], and in the crypto library [35].

This wide variety of bugs and attacks is not restricted to standard protocols like TLS. Recent papers have found such attacks also on modern implementations of secure messengers [50], encrypted cloud storage [8], and multi-party computation [47]. Consequently, a methodology for developing high-assurance cryptographic protocol designs and implementations is an urgent necessity.

Formally Verification of Protocol Components. A growing field of research, sometimes called Computer-Aided Cryptography [9], is concerned with the formal analysis of and machine-checked proofs for the design and implementation of cryptographic mechanisms and protocols. Many of the tools and techniques developed in this field can be used to verify protocol components.

Domain-specific software verification tools have been developed to analyze the correctness and security of message formatting code [53, 61], the security of cryptographic constructions [10, 32], the formal analysis of protocol logic and state machines [16, 11, 42], and the formal verification of entire protocol implementations up to high-level APIs [17, 40]. A separate line of work has focused on developing formally verified cryptographic libraries in C and assembly (see e.g. [62, 30, 52, 5]). Recent work has also addressed verified implementations of X.509 public key certificate validation [25].

A common feature of most of these tools is that they address implementations written either in domain-specific languages (DSLs) or in highly-stylized subsets of mainstream languages. Consequently, these results are mainly applied to verification-oriented research code and do not consider idiomatic implementations written by protocol developers in C or Rust. Furthermore, the literature shows that different tools are better at different verification tasks. In particular, targeted security-oriented tools are better at analyzing cryptographic components (shown in green in Figure 1) while standard software verification tools are effective on the rest. Combining these tools to verify a full protocol implementation remains a challenge.

hax: Verifying Rust Code with Multiple Provers. In this paper, we target protocol implementations written in idiomatic Rust, and we aim to drive proofs for all the protocol components from a single framework, while still using the best tool for each task. To this end, we use and build upon hax [14], a generic formal verification framework for Rust programs that translates the source code into the input languages of multiple backend provers, including F⋆, Rocq, ProVerif, and SSProve.

The programmer controls which tools are used to verify each module, and provides annotations in the Rust code that serve as proof goals and hints. Hence, the same code can be verified for different properties using different tools. hax supports safe Rust, which already guarantees memory-safety and type safety. This is a great improvement over C code. However, Rust code can still raise runtime exceptions ('panic'), e.g. when by an integer overflow or index out of bounds access of a vector. For `Bert13`, we use the F⋆ backend to prove runtime safety (the program does not crash/-panic) and to prove the correctness of message formatting; we use ProVerif to analyze the symbolic security of the core protocol logic and state machine code; we use SSProve to prove the computational security of protocol-specific cryptographic constructions.

Case Study: Formally Verifying `Bert13`. We demonstrate this methodology on `Bert13`, an implementation of TLS 1.3 that is written in Rust and supports both classical and post-quantum ciphersuites.[4]

`Bert13` uses formally verified cryptography from the libcrux library [41] and is practical on low-end devices with sub-10ms handshake completion, depending on the choice of ciphersuite. The core protocol code in `Bert13` is formally verified for the expected authenticity and confidentiality guarantees of TLS using ProVerif. The ProVerif model assumes the security of the key schedule, which we separately prove using SSProve. The model also assumes the correctness of the parsing code, which we verify using F⋆. Finally, we prove that the implementation does not panic at runtime, by verifying it for runtime safety using F⋆. In addition, we use the strong typing of the Rust type system to enforce coding disciplines such as secret independence and state machine linearity.

Contributions. In combination, these results are the first of their kind for cryptographic protocol implementations written in Rust, and `Bert13` is the first high-assurance implementation for a post-quantum variant of TLS. Ours is also the first machine-checked proof of the TLS 1.3 key schedule. We believe that the wide range of techniques we demonstrate in this paper will be independently useful as a guide to the formal analysis of other protocol libraries.

Outline. Section 2 outlines our multi-prover methodology for verifying Rust code. Section 3 describes the TLS 1.3 protocol and sets out verification goals for its implemen-

[4]We use an anonymous name for `Bert13`, which is developed as an open-source project, and will be de-anonymized before publication.

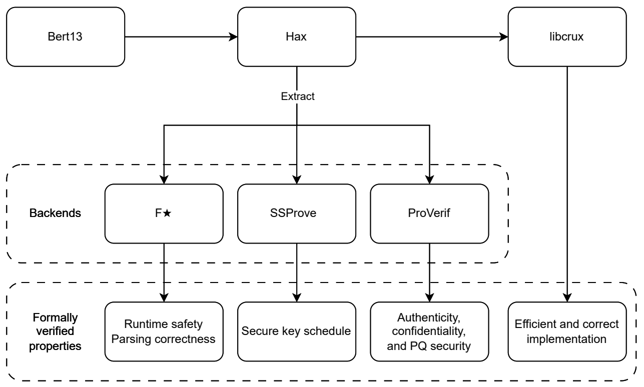

Bert13
Hax
libcrux
Extract
Backends
F⋆
SSProve
ProVerif
Formally verified properties
Runtime safety Parsing correctness
Secure key schedule
Authenticity, confidentiality, and PQ security
Efficient and correct implementation

Figure 2: Verifying Protocol Implementations with hax

tations. Section 4 describes the Bert13 implementation of post-quantum TLS 1.3. Section 5 proves security of the key schedule in the computational model with SSProve. Section 6 proves the main confidentiality and authentication guarantees for the Bert13 code using the symbolic prover ProVerif. Section 7 uses F⋆ to prove runtime safety and message formatting properties for Bert13. Section 7.2 concludes with some discussion.

2 Methodology: Verifying Rust Code with hax

Our methodology is based on hax [14], a framework for Rust verification that supports multiple proof backends. The way we use hax in this paper is depicted in Figure 2. We begin with a Rust implementation of some cryptographic protocol (here Bert13). The implementation is written in idiomatic Rust but is annotated by the Rust developer with verification goals and proof hints. The hax toolchain takes the Rust code along with the annotations and translates it into the input language of different provers, where they can be verified for security or functional properties. Notably, the developer can choose which source modules are analyzed with which tools and for which properties. The cryptography underlying the protocol implementation is provided by libcrux, a formally verified cryptographic library.

The hax toolchain has been used before for security proofs of cryptographic constructions [37] and for the correctness proofs in libcrux itself, but this is the first work to apply hax to protocol implementations. The main advantage of hax for our work is that it allows us to use multiple provers while allowing the Rust developer to drive the verification. There are many other Rust verification tools under active development [39, 6, 27, 46, 59, 45, 34, 31, 64]. We chose hax for this project primarily for its support of both security and functional verification tools. On the other hand, hax itself does not support all of Rust, and so the developer has to stay within the supported subset to use the toolchain.

For each input Rust crate, hax parses it using the rustc compiler, performs a series of transformations to facilitate translation to the functional languages in proof assistants like F⋆ and Rocq, and then generates models for various backends.

F⋆ Backend. The first backend we consider is the F⋆ [57] proof assistant, which has been used in a number of verification projects including the HACL⋆ cryptographic library [62] and libcrux; see also Section 4.6. Verification in F⋆ proceeds with the aid of assertions, refinement types and invariants. This could be used to show that e.g. that the reverse function on lists preserves its length, or that QuickSort is indeed a correct sorting function. Such properties are proven with the help of the Z3 SMT-solver.

The Rust developer can write contracts in the form of pre- and post-conditions, assertions, and loop invariants, that are translated by hax into the appropriate verification conditions in F⋆. In particular, we typically use pre-conditions to provide constraints for runtime safety, and we use post-conditions to specify correctness properties. Once all functions are annotated with contracts, they can be verified by F⋆, for the most part automatically, using its SMT solvers, although some proofs may require some additional hints. (See Section 7 for how this works in Bert13).

This is in line with the very recent (*experimental*) addition of Contracts to the Rust language.[5] It envisions a unified language for static and dynamic checks, with the ultimate goal that:

> All unsafe functions in Rust should have their safety conditions specified using contracts, and verified that those conditions are enough to guarantee absence of undefined behavior. We provide an example in Section 4.4.
>
> Rust users should be able to check that their code do not violate the safety contracts of unsafe functions, which would rule out the possibility that their applications could have a safety bug.

In this work we show that hax already facilitates this for a realistic project such as Bert13. We recommend that the hax team aligns their contracts with the experimental contracts supported by the rust language, once their design stabilizes.

Rocq and SSProve backends. Rocq [58], like F⋆, is a general proof assistant build on dependent type theory. It is a foundational proof assistant in the LCF tradition. It does not use SMT-solvers, so we expect more user interaction would be required to prove runtime safety.

Our main use of Rocq is via the SSProve [38] library in Rocq which includes syntax, semantics and programming logic of a probabilistic imperative programming language, as commonly used by cryptographers in the computational model. It also provides a program logic in the spirit of EasyCrypt [10]. On top of this, it builds an interpretation of the State-Separating Proof [23] modular style of reasoning, also used in the Joy of Cryptography book [55]. The main ingredient of SSP is a calculus for program fragments (packages) in the aforementioned programming language.

[5]`https://rust-lang.github.io/rust-project-goals/2025h1/std-contracts.html`

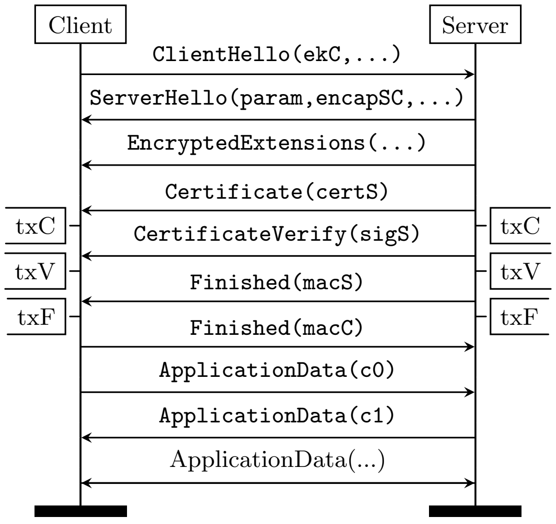
hax supports special annotations for cryptographic properties used in SSProve. These properties can then be proven in dialogue with a proof engineer. We will see in Section 5.4 that the SSP structure can help to improve the structure of the rust code.

By limiting the number of transformations in the hax toolchain, one can translate the hax subset of Rust to the simple imperative language used by SSProve in Rocq. hax even provides a proof that the functional and imperative translation agree. This can be seen as a partial correctness proof of the hax transformations.

ProVerif backend. In addition to proof assistants, hax also supports security verification of Rust code using dedicated protocol verifiers. Currently, it supports the ProVerif [21] tool, but others can be added similarly.

ProVerif is an automated tool for checking security protocols in the symbolic model (or 'Dolev-Yao'), which is codified using the applied π-calculus. Given security properties, such as confidentiality, integrity and authenticity, ProVerif will try to automatically verify these properties for a protocol model written in terms of message-passing processes. The symbolic model is less precise than the computational model (used in SSProve). It treats cryptographic primitives, such as encryption, as perfect black boxes. However, this has the advantage of much better automation. The two models aid each other, in the sense that one can prove in the computational model that the primitives have the assumed security properties.

Symbolic analyses, using tools like ProVerif, Tamarin, and DY*, have proved effective for the comprehensive formal analysis of real-world protocols like TLS, Messaging Layer Security, Noise, and Signal [42, 24, 60, 40]. We use ProVerif (in Section 6) to formally analyze our Rust implementation of TLS 1.3.



The main cryptographic computations in the protocol are:

`(k,encapSC)` =	`KEM-Encap(ekC)`
`sigS` =	`Sign(skS, txC)`
`macS` =	`MAC(mkS, txV)`
`macC` =	`MAC(mkC, txF)`
`c0` =	`AEAD(akC, m0)`
`c1` =	`AEAD(akS, m1)`

and the symmetric keys `mkC`, `mkS`, `akC`, `akS` are derived from the encapsulated key `k` via the key-schedule, as described in Section 5.

Figure 3: The TLS 1.3 protocol: main elements of the server-authenticated handshake and application data exchange

3 The TLS 1.3 Protocol

The Transport Layer Security (TLS) protocol is the IETF standard that that underlies all secure Web connections. In 2018, partly in response to some weaknesses in the previous protocol version, it was completely redesigned as TLS 1.3 [54].

3.1 Protocol Flow

Figure 3 shows the main protocol flow commonly used on the Web. The protocol is initiated by a *client* when it wishes to establish a connection with a *server*. The protocol starts with a key exchange, called the *handshake*, which authenticates the server (and potentially the client) and establishes a sequence of keys shared between them, via a novel cryptographic construction called the *key schedule*. Once the handshake is complete, the client and server can use the established keys to exchange encrypted application data with each other, using the *record* sub-protocol.

In Figure 3, the server is authenticated with an X.509 certificate, but the client remains unauthenticated. There is an alternate flow, not shown here, where the client also provides an X.509 certificate, and yet another without certificates, where both client and server authenticate each other via a pre-shared key. We purposely choose a KEM-based presentation of the protocol to make it possible to uniformly account for both Diffie-Hellman and Post-Quantum KEM-based key exchange modes.

The client first sends a `ClientHello` message containing an ephemeral KEM public key `ekC`. In response, the server sends a `ServerHello` containing a fresh key `k` encapsulated under `ekC`. After the `ServerHello`, both the client and server initialize the key schedule with the key `k`, and then use it to derive a sequence of keys as the protocol proceeds. For example, the key schedule produces handshake encryption keys, which are used to protect all subsequent handshake messages (a detail elided in the figure.)

The two hello messages also implement the negotiation phase of the protocol: the client offers a choice of versions, ciphersuites, and other extensions, and the server chooses one set of parameters in its response and in the subsequent `EncryptedExtensions` message.

The server then sends its X.509 certificate in a `Certificate` message and proves that it knows the corresponding private key by providing a signature over the current protocol transcript in a subsequent `CertificateVerify` message. The server then ends its side of the handshake by sending a `Finished` message containing a MAC over the current transcript using a MAC key `mkS` derived from the key schedule.

The client processes this stream of handshake messages from the server, decapsulates the key `k` and derives the same sequence of keys from the key schedule to decrypt the handshake messages. It then validates the server's X.509 certificate using its local certificate validation library, and verifies the server's signature and MAC. It then sends its own `Finished` message to complete the handshake.

At this point, the client and server can start exchanging application data messages that are encrypted using AEAD keys for the two directions derived from the key schedule.

3.2 Formal Analyses of TLS 1.3

Given its importance to the Web ecosystem, TLS has been comprehensively analyzed against a variety of threats in a number of security models. For TLS 1.3, there are many pen-and-paper proofs of security (see e.g. [29, 22]), mostly focused on the core protocol logic and crypto constructions. There are also several machine-checked proofs of the protocol: proofs using symbolic provers like ProVerif [42] and Tamarin [24] that treat the cryptographic primitives abstractly using equational theories, and proofs using computational provers like CryptoVerif [42] and Computational F⋆ [26] that precisely model cryptographic algorithms as probabilistic functions over bit-strings.

All the proofs above are for abstract *models* of the protocol; they do not consider the precise cryptographic formats specified in the standard, or account for multiple ciphersuites running in parallel. Consequently, it is possible that they miss some attacks. Conversely, modeling and analyzing a large protocol like TLS 1.3 is not an easy task, and the risk that the model itself will have mistakes is non-trivial.

Consequently, we advocate that protocol security analysis must be performed, where possible, directly on the protocol implementation. In this way, one can be sure of not missing some low-level formatting detail, or some protocol feature that is needed for the normal functioning of the protocol. In the past, some works have analyzed reference implementations of TLS 1.3, such as a proof-of-concept JavaScript implementation [42] of the full protocol, and a verification-oriented F⋆ implementation [26] of the record layer. Neither of these are practical implementations; they were written primarily by researchers to exercise verification tools.

3.3 Goals for our TLS 1.3 Implementation

In this paper, our goal is to verify a practical Rust implementation of TLS 1.3. Our implementation must run efficiently on a variety of platforms, ranging from IoT devices, phones, desktops, to servers. It must interoperate with other TLS implementations including popular web browsers and web servers. Furthermore, it should support both classical Elliptic-Curve Diffie-Hellman key exchanges and post-quantum key exchanges (based on post-quantum KEMs).

Importantly, we would like to formally verify that the protocol implementation achieves the confidentiality and authentication guarantees expected of TLS. To achieve this proof, we need to make some assumptions about the underlying cryptography. TLS 1.3 mainly uses well-understood cryptographic constructions (signatures, MACs, AEAD encryptions) for which we can make standard assumptions. The only novel construction in the protocol is its *key schedule*, which needs new analysis. Furthermore, most of the cryptographic operations in the protocol rely on using the protocol *transcript* to encode all the session content, and so we must prove that the transcript is *unambiguous*: if a client and server have the same transcript, their view of the session (parameters, certificates, public keys, etc.) should be the same.

We summarize these classic protocol security requirements for TLS 1.3 implementations as follows:

- **Protocol Security Guarantees**: the protocol implementation must ensure that the server (and optionally client) is authenticated and that the application data sent between honest clients and servers is confidential. This, in turn, relies on two sub-goals.
  - **Key Schedule Security**: the cryptographic construction implemented in the key schedule implementation must be provably secure.
  - **Unambiguous Transcripts**: the transcripts maintained in the protocol implementation must be injective with respect to session data.

Beyond these core cryptographic security guarantees, a cryptographic protocol implementation must satisfy certain other functional properties that are also important for the security of the user. The implementation must be memory safe, i.e. it must not read or write data out of bounds, which might leak secrets (e.g. see HeartBleed[6]). It must not crash with an unexpected error, even if an adversary were to send a maliciously crafted message, otherwise it may enable denial-of-service attacks. It must implement the protocol state machine correctly and not accept or reject messages out of turn, or else it might open up state machine attacks [12]. And it must safely handle the ephemeral session secrets generated during the run of the protocol and not accidentally reveal them to the adversary.

We summarize these additional requirements for TLS 1.3 implementations as follows:

[6]`https://heartbleed.com`

- **Implementation Security Guarantees**: the implementation must not break the security invariants expected by the protocol application. In particular:
  - **Runtime Safety**: the protocol implementation must be memory safe and must not crash with an unexpected error.
  - **Session Secret Management**: the short-term secrets generated during a session must not be revealed to the attacker via some public channel.
  - **State Machine Correctness**: the implementation must correctly implement the protocol state machine

Of course, this list of properties is not complete. One may, for example, also wish to prove full functional conformance for the protocol implementation against a formal specification of the protocol. Here, we restrict our ambitions to proving properties we deem to be essential for security, based on known attacks on TLS implementations, and leave other properties for future work.

3.4 Implementation and Proofs

In Section 4, we present Bert13, our portable post-quantum TLS 1.3 implementation in Rust. Via interoperability testing, we experimentally verify that this implementation conforms to the TLS standard. In the implementation, we use the strong type system of Rust to enforce disciplines such as secret independence (for session secret management) and for state machine correctness.

In Section 5, we prove cryptographic Security for the key schedule implementation in Bert13 using the SSProve tool. In Section 6, we prove the main confidentiality and authentication guarantees for the protocol code in Bert13 using the symbolic prover ProVerif. In Section 7, we first use the F⋆ framework to prove the runtime safety for the entire protocol implementation. We then use F⋆ to also prove the transcript unambiguity for our implementation.

4 Bert13: Post-Quantum TLS 1.3 in Rust

Bert13 is an implementation of the TLS 1.3 protocol intended for real-world usage. It is not intended to be a research artifact. As such, we have different requirements and approach development and verification as equally important goals. Hence, instead of writing the protocol in a proof-oriented language, which requires verification experts, Bert13 is written in Rust, by Rust engineers. This illustrates our methodology of enabling domain experts and software engineers to work together towards a verified implementation.

4.1 Code Structure

The Bert13 source code is separated into the core TLS 1.3 protocol and the necessary networking APIs. The repository also defines example client and server applications and provides utilities for interoperability testing.

The main components of the protocol implementation are as follows: the `formats` module implements the parsing and generation of TLS 1.3 messages; the `keyschedule` module and its submodules implement the key schedule; the `handshake` module implements the TLS 1.3 state machine and the main messaging functions for the handshake protocol; the `record` module implements the record layer encryption and decryption functions; and the `api` module provides a protocol API to applications.

The implementation relies on a few external libraries. The cryptography module provides a wrapper around the libcrux library, which provides verified implementations of all the necessary cryptographic primitives. One difference to classical TLS implementations is that the crypto module provides a Key encapsulation mechanism (KEM) API instead of Elliptic Curve Diffie Hellman (ECDH), to facilitate a uniform interface which captures both classical and Post-Quantum cipher suites.

Finally, the certificate module implements the minimal functionality required for parsing certificates as part of the TLS 1.3 handshake, and is considered an untrusted module here. The client application is expected to take the certificate, server name, and public key provided by the protocol API and validate them using an external PKI implementation. On the server, application needs to provide the protocol implementation with the appropriate certificate and private key.

4.2 Rust Types for Secret Independence

The protocol implementation uses the strong typing discipline of Rust to enforce several security and functional invariants.

Although Bert13 relies on libcrux for all its cryptography, it must still carefully handle several secret values, such as the certificate private key (on the server) and various symmetric keys derived by the key schedule. To ensure that we do not inadvertently leak these values to the adversary, we use the Rust type system to enforce *secret independence*. When the feature `secret-integers` is set, all the byte-strings in Bert13 are treated as potentially secret values. This means that their contents cannot be inspected, compared, written on public channels, or used as indices into arrays. Everything handled by the protocol is secret by default, and if the programmer wishes to look into a byte-string (because they know its contents are public) they must call the `declassify` function.

For example, after record encryption a ciphertext needs to be declassified before it can be sent on the network, and we can decide that this is safe because of the protocol security guarantees. Conversely, when decrypting a record, if we wish to inspect any part of the message, we must first declassify it, hence declaring that we are consciously willing to leak these contents. We enforce this strict discipline throughout the Bert13 implementation.

Table 1: `Bert13` performance measurements across 1000 iterations.

Ciphersuite			Client Handshake Performance	
Cipher	Signatures	KEM	Time/Handshake [μs]	Throughput [per second]
Chacha20Poly1305	P-256 ECDSA	P-256 ECDH	8872	112.70
Chacha20Poly1305	P-256 ECDSA	X25519	5287	189.11
Chacha20Poly1305	P-256 ECDSA	X25519Kyber768Draft00	6275	159.34
Chacha20Poly1305	P-256 ECDSA	X25519MlKem768	6185	161.67

4.3 Rust Types for State Machines

We also rely on the linearity guarantees of Rust types to implement the TLS 1.3 handshake state machine. When each message is sent or received, the client or server retrieves its previous state and generates a new state. By using the Rust type system, we can enforce that the previous state has been *consumed* and hence cannot be used again. For example, the `put_server_hello` function which processes a server hello message has the following structure:

```
fn put_server_hello(
    handshake: &HandshakeData,
    state: ClientPostClientHello,
    ks: &mut TLSkeyscheduler,
) -> Result<(DuplexCipherStateH,
  ClientPostServerHello), TLSError> {
    let ClientPostClientHello(...) = state;
    ...
    Ok((
      DuplexCipherStateH::new(...),
      ClientPostServerHello(...)))
}
```

In Rust, the argument `state` is not a pointer, the caller is transferring ownership of the state to this function which is consuming the old state and creating a new one. The caller cannot use the old `state` after calling this function. This style of implementing state machines is sometimes called *type state* and is usable in any language that provides *affine types* like Rust does. We implement the entire handshake state machine in this style.

4.4 Developer-driven Proof Annotations

The software engineers writing the Rust code can also add pre-conditions to help with the verification. In some areas this enforces some safe engineering practices that are otherwise only enforced by reviews. Take for example the length check in the listing below. The default way of comparing the lengths would panic, which will most likely not be caught in tests. Fuzzing may catch bugs like this. But the verification statically *ensures* that this check does not over- or underflow. The software engineer can make sure of this by adding the "requires" before the function and use the correct way of comparing the length.

```
#[requires(self.len() >= start)]
pub(crate) fn find_handshake_message(
  &self,
  handshake_type: HandshakeType,
  start: usize,
) -> bool {
  // self.len() < start + 4 would panic
  if self.len() - start < 4 {
      return false;
  }
```

4.5 Implementing Post-Quantum TLS 1.3

As mentioned before, `Bert13` supports both classical cipher suites and Post-Quantum cipher suites. Since `Bert13` uses a KEM based crypto API, supporting Post-Quantum cipher suites does not require changes to the protocol implementation.

`Bert13` implements the hybrid ciphersuite X25519MLKEM768 $0x11ec$[7] defined in [43]. Note that the exact hybrid specification for TLS 1.3 is still in progress. However, this ciphersuite is currently implemented by Firefox, Chrome, Cloudflare and others, and is compatible with the draft RFC Hybrid key exchange in TLS 1.3 [56]. The shared secret that is used to compute the TLS 1.3 master secret is defined as the 64 bytes concatenation of the X25519 shared secret and the ML-KEM shared secret shared_secret = X25519.shared_secret || ML-KEM.shared_secret.

4.6 libcrux: Formally Verified Cryptography

libcrux is a formally verified cryptographic library that provides all the primitives necessary for TLS 1.3 in `Bert13`. It contains code written in Rust and proven with hax [14], as well as verified Rust code generated from the HACL⋆ project [63, 33]. It provides, in particular, its own verified Rust implementation of ML-KEM, that is used to provided support for hybrid post-quantum KEMs in `Bert13`.

Each algorithm implemented in libcrux is formally verified for runtime safety (memory safety and crash freedom), for functional correctness with respect to a high-level mathematical specification of the algorithm written in F⋆, and

[7] `https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml`

for secret independence, a discipline that prevents certain classes of side channels. Despite including only verified implementations, code from libcrux is often as fast as or faster than unverified cryptographic implementations.

4.7 Performance and Interoperability

`Bert13` is portable across all `std` targets supported by the Rust compiler, and the libcrux library. Bare metal `no_std` environments are supported in the presence of a global allocator. The implementation is compatible with Chrome (134), Firefox (137), and Cloudflare on all implemented ciphersuites.

The `Bert13` library supports the following algorithms and protocols

as signature schemes: RSA-PSS-SHA256, ECDSA-P256-SHA256, Ed25519

as KEM: X25519, P-256 ECDH, X25519Kyber768-Draft00, X25519MlKem768

as session cipher: Chacha20Poly1305

as digest: SHA256, SHA384, SHA512

Other ciphersuites such as AES-GCM can be supported when using for example HACL⋆-backed C bindings instead of the pure Rust implementations used here. While there may be faster cryptographic implementations out there, the performance numbers in Table 1 show that `Bert13` is a usable implementation with performance comparable to the most popular TLS libraries.

See Table 1 for `Bert13` client benchmark results obtained on a Raspberry Pi 3 Model B Rev 1.2, with 900 MB of RAM and a Broadcom BCM2835 CPU running at 1.2 GHz. On this device, to establish a connection, the client 55.8 KB of stack memory using X25519 as KEM and 85.1 KB using a post-quantum hybrid KEM, at a binary size of 2980 KB.

5 Key Schedule Security with SSProve

One of the essential parts of the TLS protocol is the key scheduler. It is responsible for generating secure keys used throughout the communication between the client and server, and for eliminating incorrect or invalid invocation of key generation. An example of an attack on the key schedule is tricking the key schedule to generate the same key for two different parts of the protocol. Another type of attack is not including enough randomness or new information into the key generation. Thus making the newly generated keys weaker than required. To mitigate these attacks, we ensure that the implementation in `Bert13` is covered by the security proof from [22].

5.1 State-Separating Proofs (SSP)

The core theorem in [22] is a security proof bounding the advantage of an adversary to distinguish between a key generated by invoking the key schedule and a uniformly random key.

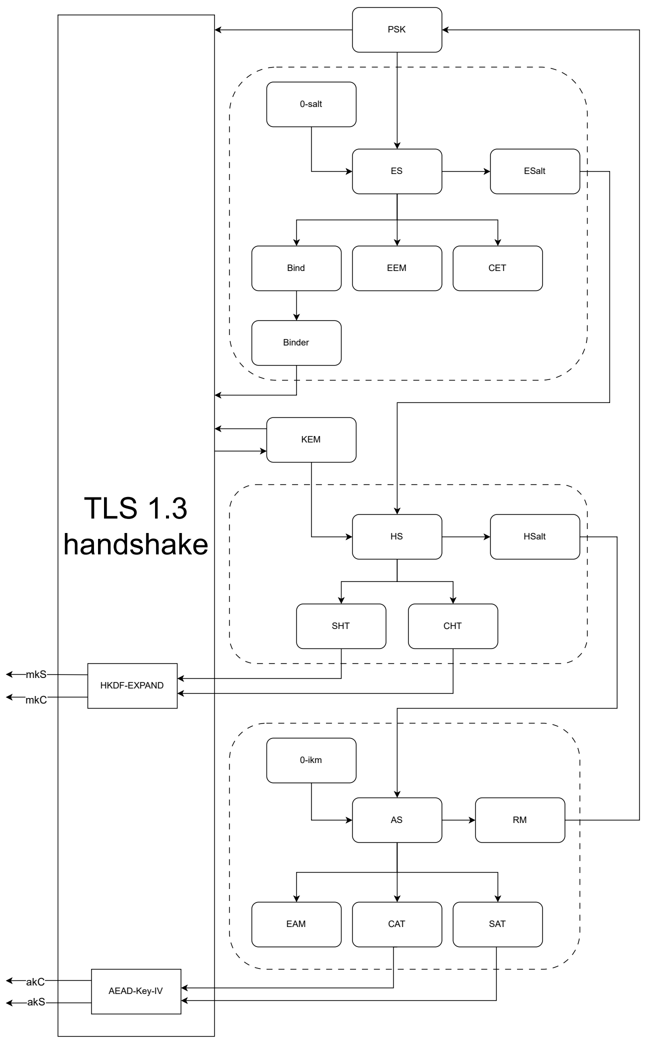


Figure 4: Calls to key schedule in the handshake protocol

The paper also proves two other theorems. The modular theorem states that one can introduce a mapping of keys. This allows a more abstract treatment, thus simplifying the arguments in the core theorem. The main theorem states the security of the composition of the modular games is bounded by a more classical monolithic version of the game, thus ensuring we can reason about the parts and still get a security statement for the protocol as a whole.

All the theorems have a pen-and-paper proof [22] in the state-separating proof (SSP) style [23]. The benefits of using SSP are that it enables modular reasoning, which is helpful when trying to scale to a large protocol like key schedule for TLS 1.3. This is achieved by providing a clear interface for each module (or 'package'). These modules can be composed in serial or parallel to create larger and more advanced packages. Security is shown by using security games: Given two packages, without any imports, one shows that an adversary cannot distinguish between them (up to negligible probability). One package describes the real behavior of the protocol and the other describes the ideal behavior. A 'game hop' replaces the real package with the ideal package. The entire protocol is defined as the composition of such packages. By a sequence of game hops, one idealizes the protocol step by step.

5.2 Mechanizing SSP in SSProve

In this paper we focus on formalizing the core theorem in SSProve. SSProve is a foundational framework in Rocq for modular cryptographic proofs in the SSP style [1].

We write the key schedule in Rust and translate the code into SSProve using hax. This guarantees that not only that the abstract Key schedule protocol is secure, but also its Rust *implementation*. We prove this by showing functional equivalence between the implementation and the package specifying the real behavior of the protocol. The equivalence is another game. We obtain the security guarantees of the implementation by transitivity.

5.3 The Formalization

The overall structure for the proof of the core theorem is given by two hybrid arguments. These come naturally from the package composition structure. The key schedule protocol is defined in a number of rounds. One of the hybrid arguments shows that one can idealize one round at a time. More concretely, we define a package for a single round of the key schedule parameterized by the round number. The key schedule package is then given by the composition of the rounds in serial, since we have a dependence on the keys of the previous round. The second hybridization argument comes from the structure of the round itself. The idealization order from [22] ensures that we can split the round into groups of packages. Each group has no dependence internally and only depends on packages earlier in the idealization order. This closely mirrors the steps in the handshake protocol, as no extra communication is needed to generate all keys in a group. We only need additional information when generating the next group in the order. The hybridization argument states that: from a bound on the advantage of idealizing each type of package, we obtain a bound on idealizing the entire round.

The proof [22] first uses the hybrid argument for proving a bound on the rounds (horizontal) and then the hybrid argument for the full protocol as a bound on the round number (vertical). However, during the formalization, we realized that we can swap the order of the hybrid arguments — do the vertical proof first for each of the smaller key packages, and then do the horizontal proof. One reason to do this is that the vertical proofs are simpler, though more plentiful. In the last round of the protocol, we do not generate the pre-share key (PSK) for the next round, so there is some difference in the interface description for the horizontal package. By swapping the order, we can handle this misalignment directly, as the horizontal proof only needs to align with the package interface of the full protocol when it is the outer hybrid argument.

5.4 The Implementation

The implementation of the key schedule is written in Rust. To facilitate the equivalence proof, we modified the implementation of the key schedule to follow the modular structure in the proof. That is, we wrote functions and interfaces based on the description in the state-separating proof (SSP) packages. This facilitates equivalence proofs, as we just have to bundle the functions into packages and then show equivalence to the SSProve package line-by-line. Moreover, it clarifies and modularizes the code base. This use of SSP for structuring implementations is one of our contributions. The rewrite of the key schedule made the handshake protocol improved readability and highlighted some shortcomings of the initial `Bert13` implementation.

Echoing the Curry-Howard correspondence, (cryptographic) proofs are programs, thus they need to be modular and parametric. Moreover, to maintain verified code, we should ensure that the code is close to the specification used by the proofs; thus, the structure of the proof guides the structure of the code and visa versa. Working with SSP is beneficial to this process, as SSP ensures modularity of proofs and code in the specification, which can be mirrored by the implementation.

The proof suggest implementing the four functions: `PrntN`, which encodes the transition graph as a map to the parent keys needed to produce a given key; `label` maps a key to its label and is used in `XTR` and `XPD` to ensure correctness of the key state; `XTR` runs key extraction (e.g. `HKDF-EXTRACT`), used when there are two parent keys; `XPD` runs key expansion (e.g. `HKDF-EXPAND`), used when there is only one parent key.

These functions together complete the graph in Fig. 4, thus implementing the key schedule for TLS. Some computations can be bundled together, so we compute/derive their values in rounds. This more or less follows the groupings generated by the idealization order. For XTR we combine two keys and a label, while XPD takes one key and some data.

5.5 Formalization Effort

The following gives a crude overview of the formalization/implementation effort.

- The paper proof in [22] ($\sim$ 1600 LoT)
- The security proof ($\sim$ 7500 LoC)
- The Rust implementation ($\sim$ 700 LoC)
- The translation ($\sim$ 1200 LoC)

The formalization is a little more than 4 times the length of the informal proof, which is reasonable, given that the formalization is more detailed.

We conjure that it is possible to facilitate the proof process by automation. The composition proofs are especially well suited for automation, as most of the proofs are boilerplate based on the structure of the composition. We also spend some effort to argue about disjointedness of package. This could possibly be simplified using Nominal SSProve [44] saving about 500 LoC.

The translation of the code is quite close to the original code, so the size difference is quite small, which is one of the benefits of using hax over other tools.

5.6 Security Reduction

The security proof in SSProve follows the pen-and-paper proof [22], which uses Diffie-Hellman for key exchange. Instead Bert13 uses a KEM based version of TLS, which is suitable for agile cryptography as it generalizes both DH and ML-KEM.

We prove security of Bert13 assuming an IND-CCA secure KEM, such a KEM is provided by a DHKEM or ML-KEM [4]. The TLS key schedule paper [22, Sec. 6], already suggest this is possible. For both DHKEM and ML-KEM libcrux provides verified rust implementations. We assume that the ML-KEM implementation in libcrux[8] agrees with the ML-KEM specification in EasyCrypt. The latter has been verified to be cryptographically secure [4].

We proved the Core Key Schedule Theorem [22, Appendix. D], which guarantees that the generated keys remain private. This theorem follows from six lemmas, D2-7 [22, Fig. 17]. We prove the main lemma D6. The others are direct consequence of the correct implementation of the cryptographic primitives which we inherit from libcrux.

To sum up, we have reduced the security of Bert13 to the existence of a secure hash function, such as provided by libcrux. We also rely on libcrux for secure cryptographic primitives such as HKDF-EXTRACT and HKDF-EXPAND.

[8]https://cryspen.com/post/ml-kem-implementation/

6 Verifying the Protocol Code with ProVerif

ProVerif [21] is an automated tool for protocol verification in the symbolic model, also known as the Dolev-Yao model [28, 49]. In conventional use of the tool, designers model by hand their protocol in a process calculus, where cryptographic primitives are treated in an idealized fashion as constructors and destructors on terms.

ProVerif then allows protocol designers to formulate queries on trace properties that should hold on all protocol runs, e.g. the occurrence of a certain event in the trace should imply previous occurrence in the trace of another event, or certain events should be ruled out for all traces. This allows, among others, a natural formulation of authentication and confidentiality guarantees as properties off the set of possible protocol traces.

Namely, if the trace contains an event indicating that a server has concluded a handshake with a client, obtaining a session key in the process, we can ask ProVerif to verify that in all traces this event is preceded by another event indicating that the client has initiated a handshake with the server and that in no trace will the session key be revealed to the attacker. Such properties can be strengthened by adding expected failure modes, e.g. explicitly allowing the attacker to learn the server's longterm secret keys.

We use the hax toolchain to automatically extract a ProVerif model of the TLS 1.3 handshake from Bert13. We then write, by hand, the top-level processes that define the protocol scenario and the security queries that encode the verification goals.

6.1 Generated Protocol Model

For each protocol function in the source Rust code, hax generates a ProVerif function modeling its behavior. For example, the Rust function put_server_hello is used by the client to process the server's hello message. It gets translated to a ProVerif function:

```
letfun put_server_hello(
   msg : t_HandshakeData,
   state : t_ClientPostClientHello,
   ks : t_TLSkeyscheduler)
=
   let ClientPostClientHello(
       client_random, ciphersuite, sk, psk, tx)
       = state in
   let (sr: t_Bytes, ct: t_Bytes) =
       parse_server_hello(ciphersuite, msg) in
   let shared_secret =
       kem_decap(ciphersuite, ct, sk) in
   let tx = transcript_add(tx, msg) in
   let th = transcript_hash(tx) in
   let shared_secret_handle = key_schedule(...)
```

This handshake function takes three arguments, an input handshake message `msg`, an input `state`, and a handle to the key schedule `ks`. It first opens up the input state (which must be the state after sending the client hello) to extract the current session parameters; it then calls the `parse_server_hello` function to parse the incoming message as a server hello. If parsing succeeds, it computes the `shared_secret` by calling `kem_decap`, updates the transcript hash and starts deriving keys with the key schedule.

The main thing to note here is that the ProVerif model captures the flow of the Rust code, including the state management, cryptographic calls, and calls to the message formatting and key schedule functions. We model exactly what the Rust code does, and do not miss any branch or coding detail.

In total, we translate 104 Rust types to ProVerif types and 119 functions from Rust to ProVerif constructors, destructors or process macros, resulting in a generated model of 5980 lines. This mainly covers the handshake and record protocols.

However, the underlying libraries are abstracted in our ProVerif model: the cryptographic library models KEM encapsulation and decapsulation using symbolic constructors and destructors; the messaging formatting model treats serialization functions as constructors and parsing functions as destructors, without modeling the precise bit-level formats of these messages; and the key schedule model uses expand and extract as opaque constructors. These abstractions are standard for symbolic analysis, but in this paper, we justify these assumptions wherever possible, by developing proofs in SSProve and F$\star$, and by relying on the correctness of the underlying libcrux crypto library.

6.2 Hand-written Verification Scenario

To complete our protocol model, we write by hand a top-level process that composes several sub-processes:

- `CreateServer` sets up server long term secrets corresponding to the ciphersuite given as an argument.
- `Client` models a client that connects to a server using a specified ciphersuite; it models the client handshake state-machine by calling the generated protocol functions (like `put_server_hello`) in sequence.
- `Server` which accepts connections from clients; it reads long-terms secrets from a table populated by `CreateServer` and then calls a sequence of server-side functions generated from the Rust code.
- `CompromiseServer` which allows the attacker to compromise server long-term secrets based on the server name, thereby emitting a `LeakServerCertSK` event in the trace.

```
process
      !CreateServer(SHA256_Chacha20Poly1305...)
      (* ... *)
    | !Client(SHA256_Chacha20Poly1305...)
      (* ... *)
    | !Server() | !CompromiseServerCertSK()
```

Each of these sub-processes is replicated, which means that we model an unbounded number of client and server sessions, and an unbounded number of compromises. We also allow clients and servers to run any non-PSK ciphersuite. The attacker is not specifically modeled; instead the ProVerif attacker is any process running in parallel to the protocol which can read and write on public channels and make use of its own keys as well as compromised keys, and can interfere with any number of sessions to try and break the security goals of the protocol. This sets up our verification scenario.

6.3 Protocol Analysis

Now that we have the protocol model, we can ask ProVerif to prove that the model provides *server authentication*, as well as *session key forward secrecy* for authenticated sessions.

At the end of the handshake, the client and server construct a duplex cipher state **`cipher_state`**, which contains among others the choice of AEAD algorithm, the client-to-server key **`akC`** as well as the server-to-client key pair **`akS`**. We write **`cipher_state(akC)`** to denote that **`akC`** is part of a cipher state. We state our security goals for the TLS handshake in terms of these cipherstates.

Server Authentication. We show that whenever a client finishes the handshake with a given server, the server must have finished as well, deriving the same cipherstate. This holds unless the server's long term certificate private key was compromised.

```
query
  server_name:  t_Bytes,
  cipher_state: t_DuplexCipherState,
  client_state: t_ClientPostClientFinished,
  server_state: t_ServerPostServerFinished;

  event(ClientFinished(server_name,
                       cipher_state,
                       client_state))
  ==>
   event(ServerFinished(server_name,
                        cipher_state,
                        server_state))
|| event(LeakServerCertSK(server_name)).
```

ProVerif shows verifies this query in under 2s.

If we ask ProVerif to prove that this query holds without the clause for server compromise, ProVerif finds an attack within 3 seconds that uses the compromised server key.

Session Key (Forward) Secrecy. We show that if the attacker learns a session key, then the server's long term certificate private key was compromised before the client was finished.

```
query
  i: time, j:time,
  server_name:  t_Bytes,
  cipher_state: t_DuplexCipherState,
  client_state: t_ClientPostClientFinished;

  event(ClientFinished(server_name,
                       cipher_state(akS),
                       client_state))@i
  && attacker(akS)
  ==>
   event(LeakServerCertSK(server_name))@j
   && i > j.
```

Hence, if the attacker learns the server's private key and uses it to impersonate the server, it may then learn the session key akS established in the session. In all other cases, the session keys are confidential. In particular, session keys established before the server compromise remain confidential.

Message integrity and confidentiality As corollaries of the handshake security goals above, we can also ask ProVerif to prove the integrity and confidentiality of each application data message sent or received in either direction.

6.4 Post-Quantum Security against Harvest-Now-Decrypt-Later Attacks

Bert13 implements post-quantum ciphersuites for TLS 1.3 and so we also analyze whether the protocol is secure against a class of quantum adversaries. In particular, we model Harvest-Now-Decrypt-Later attackers, in the same way as prior work on symbolic analysis of post-quantum protocols [18].

We include in our model the possibility that at some time, marked by an event, the attacker is able to compromise all Diffie-Hellman constructions and signature algorithms. After this time, the attacker can obtain Diffie-Hellman private keys and forge signatures.

We then ask if our protocol model is still secure, if the KEM constrution is unaffected. ProVerif is able to prove that all the queries above still hold, as long as the quantum apocalypse occurs after the session is completed. In other words, as long as we use a PQ-KEM, a passive attacker today who records all messages cannot break the TLS 1.3 guarantees using a quantum computer in the future.

7 Verifying Runtime Safety and Unambiguous Message Formats with F⋆

Using the hax toolchain, we translate the full protocol implementation to purely functional code in F⋆. This includes all the key schedule code, the message formatting modules, the protocol state machine, and the core handshake and record protocol code, all the way up to the protocol API. The total amount of Rust code we process is 3264 lines (without comments) in 8 modules, which translate to 10964 lines of F⋆.

7.1 Runtime Safety

Rust is a memory-safe language equipped with a strong type system. The Rust borrow-checker enforces that mutable variables cannot be aliased, and is able to impose a strong discipline over the use of memory in a program. The hax toolchain relies on this discipline to translate Rust code with side-effects into purely functional F⋆.

However, although the Rust compiler ensures that safe Rust cannot access memory out of bounds, programs can still try, and this will result in a panic, an unrecoverable exception where the program essentially crashes. Other language features can also panic: for example, arithmetic over a machine integer that results in an out-of-bounds value is undefined behavior and will panic in debug builds, and so can calls to unwrap on a Result or Option.

When hax translates a potentially-panicking Rust function to F⋆, it requires the programmer to prove that the function is *total*, that is, it can never panic. For example, consider the funcion find_handshake_message excerpted in Section 4. When translated to F⋆, it has the following implementation:

```
let rec impl_HandshakeData__find_handshake_message
      (self: t_HandshakeData)
      (handshake_type: t_HandshakeType)
      (start: usize)
      =
    if ((impl_HandshakeData__len self <: usize) -!
          start <: usize) <. mk_usize 4
    then false
    else ...
```

Here, the function uses the strict subtraction operator -! which requires that the result of the subtraction must be within the bounds of the usize type, and hence cannot be negative. When we try to type-check this function in F⋆, F⋆ immediately flags an error saying that it found a situation when this subtraction might underflow.

However, when we add the relevant pre-condition to the Rust code, the generated F⋆ function has a type declaration as follows:

```
val impl_HandshakeData__find_handshake_message
      (self: t_HandshakeData)
      (handshake_type: t_HandshakeType)
      (start: usize)
    : Prims.Pure bool
      (requires (impl_HandshakeData__len self <: usize)
                 ≥  start)
```

With this pre-condition, F⋆ is able to automatically prove that the subtraction is safe and that the full function is panic-free.

In this case, the function was correct, we just needed a type annotation, but during the course of our verification we found a number of cases, usually in message parsing functions, where the code was allowing for panics and we needed to change it to ensure panic-freedom. This is particularly important for code that runs on inputs taken from the untrusted network, since the attacker may have send us a maliciously crafted message to crash our software to trigger a denial-of-service, or worse.

One example of such a function is the `parse_client_hello` function, the very first function a TLS server calls on data it receives over a connection. The body of the function looks as follows:

```
let version = bytes2(3, 3);
let mut next = 0;
check_eq_with_slice(version.as_raw(),
                    client_hello.as_raw(),
                    next, next + 2)?;
next += 2;
check(client_hello.len() >= next + 32)?;
let client_random =
    client_hello.slice_range(next..next + 32);
...
```

This code uses the variable `next` as a pointer into the input `client_hello`. It starts by calling `check_eq_with_slice` to check that the first two bytes of the input matches the expected protocol version (this function returns an error if the input is too short or the match fails). It then increments `next` by 2 and extracts the client random value by slicing the next 32 bytes of the `client_hello`, after checking that the input has a sufficient number of bytes.

In an earlier version of this function, there was no call to `check` before the client random was extracted. Consequently, an attacker could have sent any message of size less than 34 and crashed the server (with a panic). Verification with F⋆ finds this bug, and adding the check suffices to prevent it.

By adding a combination of such checks (when needed) and pre-conditions, we are able to prove that all 3K+ lines of the protocol implementation are panic free.

7.2 Proving Transcript Unambiguity

As discussed in Section 3.3, the security of the TLS handshake relies crucially on the protocol transcript unambiguously representing the contents of the handshake. However, in the ProVerif analysis of Section 6, we abstract away from the low-level formatting details of the handshake messages and transcript, instead simply treating them as symbolic constructors.

Abstracting away from message formats is quite usual in protocol security analyses; indeed, other machine-checked proofs of TLS 1.3 [42, 24] also make the same assumption, and so do all pen-and-paper proofs. The main reason for this assumption is that handling the bit-level formatting details is annoying and seems irrelevant to the cryptographic analysis of the protocol.

In this paper, we seek to verify protocol *implementations*, not abstract models, and so we need to justify this abstraction. Furthermore, as many recent works show, ambiguity in important cryptographic inputs, like the TLS 1.3 transcript, can sometimes lead to serious attacks and should not be ignored [61].

Consider the function that serializes the client hello:

```
#[cfg_attr(feature = "hax-pv", pv_constructor)]
pub(crate) fn client_hello(
    algorithms: &Algorithms,
    client_random: Random,
    kem_pk: &KemPk,
    server_name: &Bytes,
    session_ticket: &Option<Bytes>,
) -> Result<(HandshakeData, usize), TLSError> {
  ...
}
```

The annotation above the function says that this function is treated as a constructor in the ProVerif analysis; in other words, we assume that given a serialized client hello, we can unambiguously parse from it the `algorithms` the client offered to the server, the client random, the client's public key, the name of the server the client wished to connect to, and the session ticket pointing to the pre-shared key (if any).

This serialized client hello is added to the transcript at both client and server, and hence after authenticating the transcript in the `Finished` messages, we know that the client and server have the same view of these fields, which is crucial for a key agreement and negotiation protocol like the TLS handshake.

To justify the assumption that the `client_hello` function operates like an injective constructor, we add a second annotation to the function, this time a post-condition for use in the F⋆ backend:

```
#[hax_lib::ensures(|result| match result {
    Result::Ok((ch,trunc_len)) => {
      trunc_len <= ch.len() &&
      match parse_client_hello(algorithms, &ch) {
        Result::Ok((cr,_,sn,pk,st,_,_)) =>
        cr == client_random &&
        &pk == kem_pk &&
        &sn == server_name &&
        &st == session_ticket,
        _ => false }},
    _ => true})]
pub(crate) fn client_hello(...) {...}
```

Table 2: Formal Verification Results for `Bert13`

Backend Prover	Rust Modules	Rust LoC	Translated LoC	Properties Proven	Time Taken for Proofs (s)
SSProve	1	425	815	Core Key Schedule Security	11m17s
ProVerif	3	1723	5980	Forward Secrecy, Authentication	20s
				HNDL Post-Quantum Security	
F⋆	8	3264	10964	Runtime Safety, Unambiguous Formats	1m21s

The `ensures` clause states that if the `client_hello` function succeeds and returns a serialized value `ch`, then if we parse this resulting value using the `parse_client_hello` function, we obtain the same values that were passed into `client_hello`. In other words, `parse_client_hello` works as an inverse of `client_hello`. So, if the client and server have the same transcript, and hence the same `client_hello`, they must also agree on all the inputs to the `client_hello` function. The post-condition also tracks other variables like `trunc_len` which we ignore here, but are needed for the panic-freedom proofs elsewhere in the protocol code.

This post-condition is then proved for the code of `client_hello` in F⋆. In a similar way, we annotate and prove unambiguity for all the message formats in TLS 1.3 and hence for the transcript.

8 Discussion

In this paper, we have demonstrated a verification methodology for cryptographic protocol implementations written in Rust. The key features of this methodology are that it targets code written by professional Rust developers (not verification researchers), and that we use multiple specialized provers to handle different parts of the proof, rather than rely on a single proof framework. In this way, we were able to prove both secrurity and functional properties for `Bert13`, our post-quantum TLS 1.3 implementation. Our implementation and all our proofs are provided in the submitted artifact.

The formal verification results for `Bert13` are summarized in Table 2. We used three tools: SSProve for cryptographic security of the key schedule code, ProVerif for the symbolic security of the protocol code, and F⋆ for runtime safety of the full protocol implementation and proofs about message formatting. Each tool is well-suited to its task, and this can be seen by the time and effort we spent on each task. Using a single framework for all proofs would have, we believe, suited one task but made the others much harder. Conversely, using a single framework has the advantage that the the properties proved for different parts of the coe can be formally connected with each other. We forego this benefit in favour of our pragmatic approach which makes it possible to effectively verify real-world Rust code.

Comparison with Other Approaches. We have already discussed a number of related works throughout the paper. Here, we focus on works that seek to verify cryptographic protocol implementations.

The miTLS project [17] developed a verified reference implementation of TLS 1.2 in a functional programming language, but this code was never considered a practical implementation.

Project Everest [13] was an umbrella project that sought to build a formally verified implementation of the entire HTTPS stack. The project produced verified cryptographic libraries [62, 52], message formatting libraries [53], and a TLS 1.3 implementation [26], all of which were written and verified in the F⋆ framework before being compiled to C. The generated C code was incorporated into many mainstream software projects and hence was used in production. However, the source code in F⋆ is arguably inscrutable to protocol developers, and the proofs for TLS 1.3 were incomplete, since they only covered the record layer, not the handshake.

RefTLS [42] used another compilation toolchain to compile a TLS 1.3 implementation written in JavaScript to models in ProVerif and CryptoVerif [20]. Hence, the authors were able to analyze the same protocol code in both the symbolic and computational models. However, the source code in JavaScript was not meant to be used in production, and the proofs did not include the message formatting code or guarantee runtime safety.

Implementations of protocols other than TLS have also been formally verified, including the Signal protocol [51], the Noise protocol framework [40], and messaging layer security [60]. All of these implementations are in functional languages, although some of them can be compiled to C or WebAssembly.

Future Work. We believe the methodology demonstrated in this paper is effective and flexible and can be extended with other verification tools and applied to other protocol implementations. In future work, we intend to explore the use of computational proof frameworks like EasyCrypt and CryptoVerif to verify stronger security properties for the protocol code than one can prove with ProVerif. We would also like to extend the functional verification guarantees beyond the proocol layer into the X.509 cerification library and the networking APIs. As the post-quantum transition gets into full swing, we believe formal verification techniques like the one presented in this paper will be essential for us to have confidence in the new set of protocols and their implementations.

References

[1] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Catalin Hritcu, Kenji Maillard, and Bas Spitters. SSProve: A foundational framework for modular cryptographic proofs in Coq. In *CSF*, pages 1–15. IEEE, 2021.

[2] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How diffie-hellman fails in practice. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 5–17. ACM, 2015.

[3] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 526–540. IEEE Computer Society, 2013.

[4] José Bacelar Almeida, Santiago Arranz Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, and Pierre-Yves Strub. Formally verifying Kyber. In *CRYPTO 2024*, pages 384–421. Springer, 2024.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *CCS*, pages 1807–1823. ACM, 2017.

[6] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 3, pages 147:1–147:30, 2019.

[7] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: breaking TLS using sslv2. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 689–706. USENIX Association, 2016.

[8] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. MEGA: malleable encryption goes awry. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 146–163. IEEE, 2023.

[9] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-Aided Cryptography. In *SP*, pages 777–795. IEEE, 2021.

[10] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In *FOSAD*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.

[11] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2):8:1–8:45, 2011.

[12] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: taming the composite state machines of TLS. *Commun. ACM*, 60(2):99–107, 2017.

[13] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, K. Rustan M. Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean Karim Zinzindohoue. Everest: Towards a verified, drop-in replacement of HTTPS. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, volume 71 of *LIPIcs*, pages 1:1–1:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[14] Karthikeyan Bhargavan, Maxime Buyse, Lucas Franceschino, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. hax: Verifying security-critical rust software using multiple provers. In *Verified Software. Theories, Tools and Experiments (VSTTE)*, 2024. `https://eprint.iacr.org/2025/142`.

[15] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 98–113. IEEE Computer Society, 2014.

[16] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. *ACM Trans. Program. Lang. Syst.*, 31(1):5:1–5:61, 2008.

[17] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 445–459. IEEE Computer Society, 2013.

[18] Karthikeyan Bhargavan, Charlie Jacomme, Franziskus Kiefer, and Rolfe Schmidt. Formal verification of the PQXDH post-quantum key agreement protocol for end-to-end secure messaging. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[19] Karthikeyan Bhargavan and Gaëtan Leurent. Transcript collision attacks: Breaking authentication in tls, IKE and SSH. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[20] Bruno Blanchet. CryptoVerif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied*, volume 117, page 156, 2007.

[21] Bruno Blanchet. Automatic verification of security protocols in the symbolic model: The verifier proverif. In *FOSAD*, volume 8604 of *Lecture Notes in Computer Science*, pages 54–87. Springer, 2013.

[22] Chris Brzuska, Antoine Delignat-Lavaud, Christoph Egger, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. Key-schedule security for the TLS 1.3 standard. Cryptology ePrint Archive, Paper 2021/467, 2021.

[23] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. In *Advances in Cryptology – ASIACRYPT 2018*, page 222–249. Springer, 2018.

[24] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1773–1788. ACM, 2017.

[25] Joyanta Debnath, Christa Jenkins, Yuteng Sun, Sze Yiu Chau, and Omar Chowdhury. ARMOR: A formally verified implementation of X.509 certificate chain validation. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 1462–1480. IEEE, 2024.

[26] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. 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*, pages 463–482. IEEE Computer Society, 2017.

[27] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: a foundry for the deductive verification of rust programs. In *International Conference on Formal Engineering Methods*, pages 90–105. Springer, 2022.

[28] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–207, 1983.

[29] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. *J. Cryptol.*, 34(4):37, 2021.

[30] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic: With proofs, without compromises. *ACM SIGOPS Oper. Syst. Rev.*, 54(1):23–30, 2020.

[31] Nima Rahimi Foroushaani and Bart Jacobs. Modular formal verification of rust programs with unsafe blocks, 2022.

[32] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 341–350. ACM, 2011.

[33] Aymeric Fromherz and Jonathan Protzenko. Compiling C to safe Rust, formalized, 2024.

[34] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. RefinedRust: A type system for high-assurance verification of rust programs. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1115–1139, 2024.

[35] Cesar Pereida García and Billy Bob Brumley. Constant-Time callees with Variable-Time callers. In

26th USENIX Security Symposium (USENIX Security 17), pages 83–98, Vancouver, BC, August 2017. USENIX Association.

[36] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 38–49. ACM, 2012.

[37] Philipp G. Haselwarter, Benjamin Salling Hvass, Lasse Letager Hansen, Théo Winterhalter, Catalin Hritcu, and Bas Spitters. The last yard: Foundational end-to-end verification of high-speed cryptography. In *CPP*, pages 30–44. ACM, 2024.

[38] Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Catalin Hritcu, Kenji Maillard, and Bas Spitters. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. *ACM Trans. Program. Lang. Syst.*, 45(3):15:1–15:61, 2023.

[39] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *PACM PL*, 6(ICFP), 2022.

[40] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. Noise*: A library of verified high-performance secure channel protocol implementations. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 107–124. IEEE, 2022.

[41] Franziskus Kiefer, Karthikeyan Bhargavan, Lucas Franceschino, Denis Merigoux, Lasse Letager Hansen, Bas Spitters, Manuel Barbosa, Antoine Séré, and Pierre-Yves Strub. HACSPEC: a gateway to high-assurance cryptography. *RealWorldCrypto*, 2023.

[42] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 435–450. IEEE, 2017.

[43] Kris Kwiatkowski, Panos Kampanakis, Bas Westerbaan, and Douglas Stebila. Post-quantum hybrid ECDHE-MLKEM Key Agreement for TLSv1.3. Internet-Draft draft-kwiatkowski-tls-ecdhe-mlkem-03, Internet Engineering Task Force, December 2024. Work in Progress.

[44] Markus Krabbe Larsen and Carsten Schürmann. Nominal state-separating proofs. Cryptology ePrint Archive, Paper 2025/598, 2025.

[45] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7, 2023.

[46] Nico Lehmann, Adam T Geller, Niki Vazou, and Ranjit Jhala. Flux: Liquid types for rust. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1533–1557, 2023.

[47] Nikolaos Makriyannis, Oren Yomtov, and Arik Galansky. Practical key-extraction attacks in leading MPC wallets. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 3053–3064. ACM, 2024.

[48] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. A cross-protocol attack on the TLS protocol. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 62–72. ACM, 2012.

[49] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, December 1978.

[50] Kenneth G. Paterson, Matteo Scarlata, and Kien Tuong Truong. Three lessons from threema: Analysis of a secure messenger. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 1289–1306. USENIX Association, 2023.

[51] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. 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*, pages 1256–1274. IEEE, 2019.

[52] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 983–1002. IEEE, 2020.

[53] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 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*, pages 1465–1482. USENIX Association, 2019.

[54] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[55] Mike Rosulek. *The Joy of Cryptography.* 2025. `https://joyofcryptography.com`.

[56] Douglas Stebila, Scott Fluhrer, and Shay Gueron. Hybrid key exchange in TLS 1.3. Internet-Draft draft-ietf-tls-hybrid-design-12, Internet Engineering Task Force, January 2025. Work in Progress.

[57] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F*. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016.

[58] The Coq Development Team. The Coq Proof Assistant. 2024.

[59] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. Verifying dynamic trait objects in rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 321–330, 2022.

[60] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. Treesync: Authenticated group management for messaging layer security. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 1217–1233. USENIX Association, 2023.

[61] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. Comparse: Provably secure formats for cryptographic protocols. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 564–578. ACM, 2023.

[62] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1789–1806. ACM, 2017.

[63] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, 2017.

[64] Sacha Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. A hybrid approach to semi-automated rust verification, 2025.

5.3 Summary

In this paper we present Bert13, a TLS 1.3 implementation, for which we prove runtime safety and correctness of the message formatting in F*; we show cryptographic security of the key schedule code in SSProve and the symbolic security of the protocol code in ProVerif. The implementation and verification of Bert13 is developer-driven in that it utilizes Hax; i.e., a Rust expert can write the implementation and add annotations to guide the verification. Then an automation or proof engineer can finish the proofs from the translation of the annotated code. The primary takeaway is that we prove properties about an actual implementation, which are usually assumed in proofs but are often a source for real attacks. We achieve this while also showing full protocol properties by using a multi-prover framework.

5.4 Details of the Security Proof for TLS 1.3 Key Scheduler

A full presentation of the proof can be found in the original TLS 1.3 key schedule paper [19, 20]. Here, we will present the technical details of the formalization based on that paper proof.

Building the Parts

First we define some more complex operations than just parallel and serial composition of packages. Then we introduce a way of decomposing and indexing the packages for the TLS 1.3 key schedule.

Hierarchies and Collections of Packages

A useful definition is a form of mapping [21, Section 5], defining a set of packages p in parallel indexed by some element of a set $i \in S$

$$\bigotimes_{i \in S} p_i,$$

and its dependent version

$$\bigotimes_{i \in S} \forall \mathbf{H}_{\mathtt{in}} : i \in S,\ p_{i,\mathbf{H}_{\mathtt{in}}}.$$

Setting $S = \mathbb{N}$ is particularly useful to parameterize the packages with the round number as

$$\bigotimes_{i \in \mathbb{N}} \forall \mathbf{H}_{\mathtt{lt}} : i < k,\ p_{i,\mathbf{H}_{\mathtt{lt}}}$$

or alternatively directly using ordinals

$$\bigotimes_{i \in \{x \in \mathbb{N} \mid x < k\}} p_i.$$

These hierarchies of collections of packages have some useful properties. Disjointness of hierarchies is equivalent to all elements being disjoint

$$\bigotimes_{i\in A} p_i \cap \bigotimes_{i\in B} q_i = \emptyset \iff \forall i \in A,\ p_i \cap \bigotimes_{i\in B} q_i = \emptyset,$$

$$x \cap \bigotimes_{i\in S} p_i = \emptyset \iff \forall i \in S,\ x \cap p_i = \emptyset,$$

and

$$x \cap y = \emptyset \iff \forall a \in x,\ \forall b \in y,\ a \neq b.$$

Thus, we can compute

$$\bigotimes_{i\in A} p_i \cap \bigotimes_{i\in B} q_i = \emptyset \iff \forall a \in \bigotimes_{i\in A} p_i,\ \forall b \in \bigotimes_{i\in B} q_i,\ a \neq b.$$

We have similar properties for subsets

$$\bigotimes_{i\in S} p_i \subseteq P \iff \forall a \in \bigotimes_{i\in A} p_i,\ a \subseteq P$$

and

$$\forall x \in S,\ p_x \subseteq \bigotimes_{i\in S} p_i.$$

There are also useful identity properties

$$\bigotimes_{\_\in S} p = p$$

when $S \neq \emptyset$ or $p = \emptyset$. Hierarchies can be split into subsets

$$\bigotimes_{i\in S} p_i = \bigotimes_{i\in S_1} p_i \otimes \bigotimes_{i\in S\setminus S_1} p_i$$

for $S_1 \subseteq S$. None of the above equations are package specific, so it also generalizes to the interfaces.

Disjoint Namespace

We define a disjoint allocation of function names (natural numbers) indexed by key name n, round index ℓ, and function type t as

$$\texttt{FnNumber}_{n,\ell,t} ::= KN_{\#n} + \ell \cdot |KN| + t \cdot |KN| \cdot (d+1) + \texttt{offset},$$

where $KN_{\#n}$ is the index of n, $|\cdot|$ is the size, d is a global upper bound on the number of rounds, and `offset` shifts the index to allow a number of predefined functions not following this scheme. For TLS we have

$$KN = [\bot, \texttt{ES}, \texttt{EEM}, \texttt{CET}, \texttt{BIND}, \texttt{BINDER}, \texttt{HS}, \texttt{SHT}, \texttt{CHT}, \texttt{H}_{\texttt{salt}},$$

$$\mathtt{AS}, \mathtt{RM}, \mathtt{CAT}, \mathtt{SAT}, \mathtt{EAM}, \mathtt{PSK}, \mathtt{O}_{\mathtt{salt}}, \mathtt{E}_{\mathtt{salt}}, \mathtt{ML\text{-}KEM}, \mathtt{O}_{\mathtt{IKM}}]$$

using the indexing notation we have, e.g., $KN_{\mathtt{CET}} = 3$. The definition of `FnNumber` has the disjointness properties we want, i.e.,

$$\begin{aligned}
\forall n_1\, n_2,\; n_1 \neq n_2 \Rightarrow \forall \ell\, t,\; \mathtt{FnNumber}_{n_1,\ell,t} \neq \mathtt{FnNumber}_{n_2,\ell,t}\\
\forall \ell_1\, \ell_2,\; \ell_1 \neq \ell_2 \Rightarrow \forall n\, t,\; \mathtt{FnNumber}_{n,\ell_1,t} \neq \mathtt{FnNumber}_{n,\ell_2,t}\\
\forall t_1\, t_2,\; t_1 \neq t_2 \Rightarrow \forall n\, \ell,\; \mathtt{FnNumber}_{n,\ell,t_1} \neq \mathtt{FnNumber}_{n,\ell,t_2}.
\end{aligned}$$

So if the key name, the index, or the function types differ, then the function names differ. Furthermore, the function name is never below offset

$$\forall n\, \ell\, t,\; k < \mathtt{offset} \Rightarrow \mathtt{FnNumber}_{n_1,\ell,t} \neq k.$$

We can combine this into a single statement,

$$\begin{aligned}
&\forall n_1\, n_2\, \ell_1\, \ell_2\, t_1\, t_2,\\
&\quad ((t_1 = t_2) \Rightarrow (\ell_1 \neq \ell_2 \vee n_1 \neq n_2)) \Rightarrow \mathtt{FnNumber}_{n_1,\ell_1,t_1} \neq \mathtt{FnNumber}_{n_2,\ell_2,t_2}.
\end{aligned}$$

Composition for Key Packages

We start by defining a construct for functions over a set of names and index

$$\mathscr{L}_{d,S}(p_{n,\ell}) = \bigotimes_{\ell \in \{x \in \mathbb{N} | x < d\}} \bigotimes_{n \in S} p_{n,\ell}.$$

We use $\mathscr{L}_S$ when using the globally defined round upper bound d. We now categorize keys into

$$\begin{aligned}
&\mathtt{XTR} ::= \{\mathtt{ES}, \mathtt{HS}, \mathtt{AS}\}\\
&\mathtt{XPD} ::= \{\mathtt{PSK}, \mathtt{E}_{\mathtt{salt}}, \mathtt{EEM}, \mathtt{CET}, \mathtt{BIND}, \mathtt{BINDER}, \mathtt{SHT}, \mathtt{CHT}, \mathtt{H}_{\mathtt{salt}}, \mathtt{RM}, \mathtt{CAT}, \mathtt{SAT}, \mathtt{EAM}\}
\end{aligned}$$

and define packages

$$P_{\mathtt{XTR}}^{f_b} = \mathscr{L}_{\mathtt{XTR}}(xtr_{n,\ell}^{f_b(n,\ell)})$$

and

$$P_{\mathtt{XPD}} = \mathscr{L}_{\mathtt{XPD} \setminus \{\mathtt{PSK}\}}(\mathtt{xpd}_{n,\ell}) \quad \otimes \bigotimes_{\ell \in \{x \in \mathbb{N} | x < d\}} \mathtt{xpd}_{\mathtt{PSK},\ell};$$

for package code, see Figure 5.1a and Figure 5.1b. We are using function names here, but they are defined as

$$\begin{aligned}
\mathtt{SET}_{n,\ell} &::= \mathtt{FnNumber}_{n,\ell,0},\\
\mathtt{GET}_{n,\ell} &::= \mathtt{FnNumber}_{n,\ell,1},
\end{aligned}$$

and

$$\mathtt{HASH} < \mathtt{offset}.$$

```
$\texttt{xpd}_{n,\ell}$
Imports :
$\texttt{GET}_{n_1,\ell}$ for $n_1 = \pi_1\ (\texttt{PrntN}_n)$
$\texttt{SET}_{n,\ell'}$ for $\ell' = \ell + (n \stackrel{?}{=} \texttt{PSK})$
HASH
Exports:
$\texttt{XPD}_{n,\ell}(h_1, r, args)$
$(n_1, \_) \leftarrow \texttt{PrntN}_n$
$label \leftarrow \texttt{Labels}_{n,r}$
$h \leftarrow \texttt{xpd}\langle n, label, h_1, args\rangle$
$(k_1, hon) \leftarrow \texttt{get}_{n_1,\ell,h_1}$
$h \leftarrow$ if $n = \texttt{PSK}$
    $k \leftarrow \texttt{xpd}(k_1, (label, args))$
    $\texttt{set}_{n,\ell+1}(h, hon, k)$
  else
    $digest \leftarrow \mathscr{H}(args)$
    $k \leftarrow \texttt{xpd}(k_1, (label, digest))$
    $\texttt{set}_{n,\ell}(h, hon, k)$
  fi
ret $h$
```

(a) xpd package

```
$\texttt{xtr}^{b}_{n,\ell}$
Imports :
$\texttt{GET}_{n_1,\ell}$ for $n_1 = \pi_1\ (\texttt{PrntN}_n)$
$\texttt{GET}_{n_2,\ell}$ for $n_2 = \pi_2\ (\texttt{PrntN}_n)$
$\texttt{SET}_{n,\ell}$
Exports:
$\texttt{XTR}_{n,\ell}(h_1, h_2)$
$(n_1, n_2) \leftarrow \texttt{PrntN}_n$
$h \leftarrow \texttt{xtr}\langle n, h_1, h_2\rangle$
$(k_1, hon_1) \leftarrow \texttt{get}_{n_1,\ell}(h_1)$
$(k_2, hon_2) \leftarrow \texttt{get}_{n_2,\ell}(h_2)$
$k \leftarrow \texttt{xtr}(k_1, k_2)$
$hon \leftarrow \texttt{ret}\ (hon_1 \parallel hon_2)$
$k \leftarrow$ if $b \parallel hon2$
    $k^* \in_R \mathbb{Z}_{|KN|}$
    ret $\texttt{tag}(alg\ k, k^*)$
  else
    ret $k$
  fi
$h \leftarrow \texttt{set}_{n,\ell}(h, hon, k)$
ret $h$
```

(b) xtr package

Figure 5.1: xtr and xpd package definitions

Next we build hierarchies for the key stores. We assume there exists a lookup table for all keys and for all logging. We define functions for applying different functions based on if a key exists. We also assume a decision function $\exists_? h^*, \_$ exists, which checks if any h^* fulfills the functions without failure. For the logging and key code, see Figure 5.2a and Figure 5.2b. We define the hierarchies of these packages as

$$P_{\mathrm{L},S,f_P} ::= \bigotimes_{n \in S} (\mathrm{L}_{n,f_P(n)})$$
$$P^{f_b}_{\mathrm{K},S} ::= \mathscr{L}_S(\mathrm{K}^{f_b(n,\ell)}_{n,\ell}).$$

Before we can define the core game, we are only missing the definition of $\texttt{check}_{n,\ell}$, which can be found in Figure 5.3. The set `separation_points` contains elements such that any path from PSK to an output key contains an element in `separation_points`. The same goes for any path from `ML-KEM`. This ensures a separation between the initial keys and the output keys. Furthermore, the set `early` represents keys not

```
$\mathtt{L}_{n,P}$
Exports:
$\mathtt{UNQ}_n(h, hon, k)$
if $\exists_?\ h^*$,
    $(h', hon', k) \leftarrow \mathtt{get?}(\mathtt{TABLE}_L[h^*])$
then
    $r \leftarrow \mathtt{level}(h)$
    $r' \leftarrow \mathtt{level}(h^*)$
    match $P$ with
     | $Z \Rightarrow \mathtt{ret\ tt}$
     | $A \Rightarrow$ if $hon \neq hon'$ && $r \neq r'$
         then fail
     | $F \Rightarrow$ fail
     | $D \Rightarrow$ if $hon \neq hon'$ then fail
     | $R \Rightarrow$ if $hon \neq hon'$
         then $\mathbf{fail}_{abort}$
         else $\mathbf{fail}_{win}$
fi
$\mathtt{set\_at}_{\mathtt{TABLE}_L\ h}(h, hon, k)$
$\mathtt{ret}\ h$
```

(a) Log package

```
$\mathtt{K}^b_{n,\ell}$
Imports :
$\mathtt{UNQ}_n$
Exports:
$\mathtt{SET}_{n,\ell}(h, hon, k^*)$
$\mathtt{get\_fn}(\mathtt{TABLE}_K[h])$
| $\mathtt{FAIL} \Rightarrow$
    $k \leftarrow \mathtt{ret}\ (\mathtt{untag}(k^*))$
    $k \leftarrow$ if $b$ && $hon$
        $x \in_R \mathbb{Z}_{|KN|}$
        $\mathtt{ret}\ x$
    else
        $\mathtt{ret}\ k$
    fi
    $\mathtt{unq}(h, hon, k)$
    $\mathtt{set\_at}(\mathtt{TABLE}_K[h])(k, hon)$
    $\mathtt{ret}\ h$
| $\mathtt{SUCCESS}\ \_ \Rightarrow$
    $\mathtt{ret}\ h$

$\mathtt{GET}_{n,\ell}(h)$
$(k^*, hon) \leftarrow \mathtt{get\_fail}(\mathtt{TABLE}_K[h])$
  | $\mathtt{FAIL} \Rightarrow \mathtt{fail}$
$k \leftarrow \mathtt{ret}\ (\mathtt{tag}(\pi_1(\pi_2 h), \pi_2 k^*))$
$\mathtt{ret}\ (k, hon)$
```

(b) Key package

Figure 5.2: Log and key store package definitions

influenced by ML-KEM. For the proof of the core theorem for Bert13, these checks can be ignored, as we do not use binders. We can now define the core game, but we will start by giving a generalized definition, which can be used for some of the intermediate steps in the proof of the core theorem. The generalized core package construction is

$$G_{\mathtt{core\text{-}package\text{-}construction}}(f_{b,\mathtt{xtr}}, f_{b,\mathrm{K}}, f_{P,\mathrm{L}}, b_{\mathtt{Hash}}) ::= \\ ((\mathscr{L}_{\mathtt{XPR}}(\mathtt{check}_{n,\ell}) \otimes \mathscr{L}_{\mathtt{XTR}}(\mathtt{ID}_{n,\ell}) \otimes \mathscr{L}_{\mathtt{O}^*}(\mathtt{ID}_{n,\ell})) \circ (\mathscr{L}_{\mathtt{O}^*}(\mathtt{ID}_{n,\ell}) \otimes P_{\mathtt{XPD}} \otimes P_{\mathtt{XTR}}^{f_{b,\mathtt{xtr}}})) \\ \otimes (G_{\mathtt{ml\text{-}kem}} \otimes \mathtt{ID}_{\mathtt{PSK},0}) \\ \otimes ((P_{\mathtt{K},KN}^{f_{b,\mathrm{K}}} \circ P_{\mathtt{L},KN,(\lambda\_,\mathtt{Z})}) \otimes (K_{\mathtt{PSK},d+1}^{f_{b,K}(\mathtt{PSK},d+1)} \circ L_{\mathtt{PSK},(\lambda\_,Z)}))$$

```
$\texttt{check}_{n,i}$
Exports:
$\texttt{XPD}_{n,\ell}(h_1, r, args)$
if $n = \texttt{BIND}$
  if $r = \texttt{false}$ then $\texttt{assert}(\texttt{level}(h_1) = 0)$ else $\texttt{ret tt}$
  if $r = \texttt{true}$ then $\texttt{assert}(\texttt{level}(h_1) > 0)$ else $\texttt{ret tt}$
else
  if $n \in \texttt{separation\_points} \cap \texttt{early}$
    $binder \leftarrow \textsc{BinderArgs}(args)$
    $h_{bndr} \leftarrow \textsc{BinderHand}(h_1, args)$
    $(k, \_) \leftarrow \texttt{get}_{\texttt{BINDER},\ell}(h_{bndr})$
    $\texttt{assert}(binder = k)$
  else
    if $n \in \texttt{separation\_points}$
      $(X, Y) \leftarrow \textsc{MLKemArgs}(args)$
      $h_{\texttt{ml-kem}} \leftarrow \textsc{MLKemHand}(h_1)$
      $\texttt{assert}(h_{\texttt{ml-kem}} = \texttt{ml-kem}\langle(sort(X, Y))\rangle)$
      $binder \leftarrow \textsc{BinderArgs}(args)$
      $h_{bndr} \leftarrow \textsc{BinderHand}(h_1, args)$
      $(k, \_) \leftarrow \texttt{get}_{\texttt{BINDER},\ell}(h_{bndr})$
      $\texttt{assert}(binder = k)$
    else
      $\texttt{ret tt}$
    fi
  fi
$h \leftarrow \texttt{xpd}(h_1, r, args)$
$\texttt{ret}\ h$
```

Figure 5.3: Check package

$\otimes\, \texttt{Hash}^{b_{\texttt{Hash}}}$.

The core game of the key schedule is then defined as

$$G^b_{\texttt{core}} ::= G_{\texttt{core-package-construction}}((\lambda\_, b), (\lambda\_, b), (\lambda\_, Z), b).$$

For a graphical overview of the core package, see [20, Figure. 11]. For a proof overview, see [20, Figure. 31]. We will use λ^v to represent constant functions returning v. We can then specify the intermediate steps by

$$G_{\texttt{core-hash}} ::= G_{\texttt{core-package-construction}}(\lambda^{\texttt{false}}, \lambda^{\texttt{false}}, \lambda^Z, \texttt{true})$$

$$\begin{aligned}
G_{\texttt{core-D}} &::= G_{\texttt{core-package-construction}}(\lambda^{\texttt{false}}, \lambda^{\texttt{false}}, \lambda^{D}, \texttt{true}) \\
f_{P,\texttt{E}_{\texttt{salt}}} &::= \lambda n, \begin{cases} R & n = \texttt{E}_{\texttt{salt}} \\ D & o.w. \end{cases} \\
G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}} &::= G_{\texttt{core-package-construction}}(\lambda^{\texttt{false}}, \lambda^{\texttt{false}}, f_{P,\texttt{E}_{\texttt{salt}}}, \texttt{true}) \\
f_{b,\texttt{HS}} &::= \lambda n, \begin{cases} \texttt{true} & n = \texttt{HS} \\ \texttt{false} & o.w. \end{cases} \\
G_{\texttt{core-SO-KEM}} &::= G_{\texttt{core-package-construction}}(f_{b,\texttt{HS}}, \lambda^{\texttt{false}}, f_{P,\texttt{E}_{\texttt{salt}}}, \texttt{true}) \\
N^{*} &::= [\texttt{ES}, \texttt{EEM}, \texttt{CET}, \texttt{BIND}, \texttt{BINDER}, \texttt{HS}, \texttt{SHT}, \texttt{CHT}, \\
&\qquad \texttt{H}_{\texttt{salt}}, \texttt{AS}, \texttt{RM}, \texttt{CAT}, \texttt{SAT}, \texttt{EAM}, \texttt{0}_{\texttt{salt}}, \texttt{E}_{\texttt{salt}}, \texttt{0}_{\texttt{IKM}}] \\
f_{b,N^{*},\ell} &::= \lambda n, \ ((n \in N^{*}) \parallel (n = \texttt{PSK})) \ \&\& \ \neg(i \leq \ell) \\
G_{\texttt{core-hyb},\ell} &::= G_{\texttt{core-package-construction}}(f_{b,\texttt{HS}}, f_{b,N^{*},\ell}, f_{P,\texttt{E}_{\texttt{salt}}}, \texttt{true}) \\
f_{b,N^{*},\ell,C} &::= \lambda n, \ ((n \in N^{*}) \parallel (n = \texttt{PSK})) \ \&\& \ \neg(i + (n \in C) \leq \ell) \\
G_{\texttt{core-hyb-pred},\ell,c} &::= G_{\texttt{core-package-construction}}(f_{b,\texttt{HS}}, f_{b,N^{*},\ell,C}, f_{P,\texttt{E}_{\texttt{salt}}}, \texttt{true}) \\
G_{\texttt{core-ki}} &::= G_{\texttt{core-package-construction}}(f_{b,\texttt{HS}}, \lambda^{\texttt{true}}, \lambda^{D}, \texttt{true}).
\end{aligned}$$

Now the proof of the core theorem is a sequence of equivalences in order of the above definitions; see [20, Appendix. D] in general and page 72 in specific. We first introduce a couple lemmas (without proof):

$$\begin{aligned}
\mathscr{A}(G_{\texttt{core}}, A) &\leq \mathscr{A}(G_{acr}(f_{hash}), A \circ R_{cr}) + \\
&\quad \mathscr{A}(G_{\texttt{core-hash}}, G^{1}_{\texttt{core}}, A) && (5.1) \\
\mathscr{A}(G_{\texttt{core-hash}}, G_{\texttt{core-D}}, A) &\leq \mathscr{A}(G_{acr}(f_{xtr}), A \circ R_{Z}\ f_{xtr}) + && (5.2) \\
&\quad \mathscr{A}(G_{acr}(f_{xpd}), A \circ R_{Z}\ f_{xpd}) + \\
&\quad \mathscr{A}(G_{acr}(f_{xtr}), A \circ R_{D}\ f_{xtr}) + \\
&\quad \mathscr{A}(G_{acr}(f_{xpd}), A \circ R_{D}\ f_{xpd}) \\
\mathscr{A}(G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}}, G^{1}_{\texttt{core}}, A) &\leq \max_{i} \mathscr{A}(G_{\texttt{core-D}}, G^{1}_{\texttt{core}}, A_{i}) && (5.3) \\
\mathscr{A}(G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}}, G_{\texttt{core-SO-KEM}}, A) &= \mathscr{A}(G_{\texttt{core-SO-KEM}}, A_{i} \circ R_{\texttt{so-kem}}) && (5.4) \\
\mathscr{A}(G_{\texttt{core-ki}}, G^{1}_{\texttt{core}}, A_{i}) &\leq \mathscr{A}(G_{pi,[\texttt{E}_{\texttt{salt}}],R}, A_{i} \circ R_{pi,[\texttt{E}_{\texttt{salt}}]}) + && (5.5) \\
&\quad \mathscr{A}(G_{pi,\texttt{0}^{*},R}, A_{i} \circ R_{pi,\texttt{0}^{*}}),
\end{aligned}$$

and the equivalence for $\texttt{D} \to \texttt{R}$

$$\mathscr{A}(G_{\texttt{core-D}}, G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}}, A) = 0. \tag{5.6}$$

The Proof of the Important Lemma

First we compute the order in which we idealize the named key stores for each round (see Figure 4 in the paper above). The order is computed by maintaining a set of key names and then, in each round, adding any names for keys that could be produced with information from previous rounds. This makes it so that the set is monotonically increasing over the number of rounds. We start from the keys $\mathtt{PSK}, 0_{\mathtt{salt}}, \mathtt{KEM}, 0_{\mathtt{IKM}}$ and continue the process until the set contains all the names of the keys, i.e., at most $|KN|$. To simplify the proof, we precompute the set of each round for the TLS 1.3 keys. Written as a list of additions, it computes to

$$\begin{aligned}&[[\mathtt{PSK}, 0_{\mathtt{salt}}, \mathtt{ML\text{-}KEM}, 0_{\mathtt{IKM}}],\\ &[\mathtt{ES}],\\ &[\mathtt{EEM}, \mathtt{CET}, \mathtt{BIND}, \mathtt{E}_{\mathtt{salt}}],\\ &[\mathtt{BINDER}],\\ &[\mathtt{HS}],\\ &[\mathtt{SHT}, \mathtt{CHT}, \mathtt{H}_{\mathtt{salt}}],\\ &[\mathtt{AS}],\\ &[\mathtt{RM}, \mathtt{CAT}, \mathtt{SAT}, \mathtt{EAM}]].\end{aligned}$$

This aligns well with communication rounds of TLS 1.3, as it encapsulates what we can do before needing more information. We will use Ord_{idl} to refer to the full accumulated list of the idealization order. This is reflected by the lemma

$$\forall i\ j,\ i < j \Rightarrow (j \leq |Ord_{idl}|) \Rightarrow \forall x,\ x \in Ord_{idl}[i] \Rightarrow x \in Ord_{idl}[j]$$

Next we introduce some lemmas (without proof):

$$\begin{aligned}&((c = 0)\ \|\ (\mathtt{PSK} \notin Ord_{idl}[c-1]))\\ &\Rightarrow \mathtt{PSK} \in Ord_{idl}[c]\\ &\Rightarrow C = Ord_{idl}[c] \wedge C' = Ord_{idl}[c+1]\\ &\wedge \mathscr{A}(G_{\mathtt{core\text{-}hyb\text{-}pred},\ell,C}, G_{\mathtt{core\text{-}hyb\text{-}pred},\ell,C'}, A) \leq \mathscr{A}(G_{\mathtt{xtr},\mathtt{ES},\ell}, A \circ R_{\mathtt{xtr},\mathtt{ES},\ell})\end{aligned} \tag{5.7}$$

$$\begin{aligned}&((c = 0)\ \|\ (\mathtt{E}_{\mathtt{salt}} \notin Ord_{idl}[c-1]))\\ &\Rightarrow \mathtt{E}_{\mathtt{salt}} \in Ord_{idl}[c]\\ &\Rightarrow C = Ord_{idl}[c] \wedge C' = Ord_{idl}[c+1]\\ &\wedge \mathscr{A}(G_{\mathtt{core\text{-}hyb\text{-}pred},\ell,C}, G_{\mathtt{core\text{-}hyb\text{-}pred},\ell,C'}, A) \leq \mathscr{A}(G_{\mathtt{xtr},\mathtt{HS},\ell}, A \circ R_{\mathtt{xtr},\mathtt{HS},\ell})\end{aligned} \tag{5.8}$$

$$\begin{aligned}&((c = 0)\ \|\ (\mathtt{H}_{\mathtt{salt}} \notin Ord_{idl}[c-1]))\\ &\Rightarrow \mathtt{H}_{\mathtt{salt}} \in Ord_{idl}[c]\\ &\Rightarrow C = Ord_{idl}[c] \wedge C' = Ord_{idl}[c+1]\\ &\wedge \mathscr{A}(G_{\mathtt{core\text{-}hyb\text{-}pred},\ell,C}, G_{\mathtt{core\text{-}hyb\text{-}pred},\ell,C'}, A) \leq \mathscr{A}(G_{\mathtt{xtr},\mathtt{AS},\ell}, A \circ R_{\mathtt{xtr},\mathtt{AS},\ell})\end{aligned} \tag{5.9}$$

$$\begin{aligned}&\mathtt{ChldrOp}(\pi_1(\mathtt{PrntN}\ n)) = \mathtt{xpdOp}\\ &\Rightarrow n \in \mathtt{XPD}\end{aligned} \tag{5.10}$$

$$\begin{aligned}
&\Rightarrow \pi_1(\texttt{PrntN}\ n) \notin Ord_{idl}[c-1] \\
&\Rightarrow ((c=0) \parallel (\pi_1(\texttt{PrntN}\ n) \in Ord_{idl}[c+1])) \\
&\Rightarrow C = Ord_{idl}[c] \wedge C' = Ord_{idl}[c+1] \\
&\wedge \mathscr{A}(G_{\texttt{core-hyb-pred},\ell,C}, G_{\texttt{core-hyb-pred},\ell,C'}, A) \leq \mathscr{A}(G_{\texttt{xpd},n,\ell}, A \circ R_{\texttt{xpd},n,\ell}).
\end{aligned}$$

Now we can state and prove the important lemma, a hybridization argument, which states

$$\begin{aligned}
&\mathscr{A}(G_{\texttt{core-hyb},\ell}, G_{\texttt{core-hyb},\ell+1}, A) \\
&\leq \mathscr{A}(G_{\texttt{xtr},\texttt{ES},\ell}, A \circ R_{\texttt{xtr},\texttt{ES},\ell}) + \\
&\quad \mathscr{A}(G_{\texttt{xtr},\texttt{HS},\ell}, A \circ R_{\texttt{xtr},\texttt{HS},\ell}) + \\
&\quad \mathscr{A}(G_{\texttt{xtr},\texttt{AS},\ell}, A \circ R_{\texttt{xtr},\texttt{AS},\ell}) + \\
&\quad \sum_{n \in \texttt{XPD}} \mathscr{A}(G_{\texttt{xpd},n,\ell}, A \circ R_{\texttt{xpd},n,\ell})).
\end{aligned} \tag{5.11}$$

We apply the triangle inequality to setup for the hybridization argument

$$\begin{aligned}
&\mathscr{A}(G_{\texttt{core-hyb},\ell}, G_{\texttt{core-hyb},\ell+1}, A) \\
&\overset{\Delta}{\leq} \mathscr{A}(G_{\texttt{core-hyb},\ell}, G_{\texttt{core-hyb-pred},\ell,[\,]}, A) + \\
&\quad \mathscr{A}(G_{\texttt{core-hyb-pred},\ell,[\,]}, G_{\texttt{core-hyb},\ell+1}, A).
\end{aligned}$$

Then we argue that the first part disappears because

$$\mathscr{A}(G_{\texttt{core-hyb},\ell}, G_{\texttt{core-hyb-pred},\ell,[\,]}, A) = 0.$$

To show this, we unfold the definitions,

$$\begin{aligned}
&\mathscr{A}(G_{\texttt{core-hyb},\ell}, G_{\texttt{core-hyb-pred},\ell,[\,]}, A) \\
&= \mathscr{A}(G_{\texttt{core-package-construction}}(f_{b,\texttt{HS}}, f_{b,N^*,\ell}, f_{P,\texttt{E}_{\texttt{salt}}}, \texttt{true}), \\
&\qquad G_{\texttt{core-hyb-pred},\ell,[\,]}, A) \\
&= \mathscr{A}(G_{\texttt{core-package-construction}}(f_{b,\texttt{HS}}, f_{b,N^*,\ell}, f_{P,\texttt{E}_{\texttt{salt}}}, \texttt{true}), \\
&\qquad G_{\texttt{core-package-construction}}(f_{b,\texttt{HS}}, f_{b,N^*,\ell,[\,]}, f_{P,\texttt{E}_{\texttt{salt}}}, \texttt{true}), A).
\end{aligned}$$

It is obvious that $f_{b,N^*,\ell,[\,]} = f_{b,N^*,\ell}$, thus,

$$\begin{aligned}
&= \mathscr{A}(G_{\texttt{core-package-construction}}(f_{b,\texttt{HS}}, f_{b,N^*,\ell}, f_{P,\texttt{E}_{\texttt{salt}}}, \texttt{true}), \\
&\qquad G_{\texttt{core-package-construction}}(f_{b,\texttt{HS}}, f_{b,N^*,\ell}, f_{P,\texttt{E}_{\texttt{salt}}}, \texttt{true}), A) \\
&= 0.
\end{aligned}$$

We also step closer to the hybridization argument from the other side by again using the triangle inequality

$$\mathscr{A}(G_{\texttt{core-hyb-pred},\ell,[\,]}, G_{\texttt{core-hyb},\ell+1}, A)$$

$$\overset{\Delta}{\leq} \mathscr{A}(G_{\texttt{core-hyb-pred},\ell,[\,]}, G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[|Ord_{idl}|-1]}, A) + \\ \mathscr{A}(G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[|Ord_{idl}|-1]}, G_{\texttt{core-hyb},\ell+1}, A).$$

Again we argue that

$$\mathscr{A}(G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[|Ord_{idl}|-1]}, G_{\texttt{core-hyb},\ell+1}, A) = 0.$$

We unfold the definitions to get

$$\begin{aligned}
&\mathscr{A}(G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[|Ord_{idl}|-1]}, G_{\texttt{core-hyb},\ell+1}, A)\\
&= \mathscr{A}(G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[|Ord_{idl}|-1]},\\
&\qquad G_{\texttt{core-package-construction}}(f_{b,\texttt{HS}}, f_{b,N^*,\ell+1}, f_{P,\texttt{E}_{\texttt{salt}}}, \texttt{true}), A)\\
&= \mathscr{A}(G_{\texttt{core-package-construction}}(f_{b,\texttt{HS}}, f_{b,N^*,\ell,Ord_{idl}[|Ord_{idl}|-1]}, f_{P,\texttt{E}_{\texttt{salt}}}, \texttt{true}),\\
&\qquad G_{\texttt{core-package-construction}}(f_{b,\texttt{HS}}, f_{b,N^*,\ell+1}, f_{P,\texttt{E}_{\texttt{salt}}}, \texttt{true}), A).
\end{aligned}$$

It is, again, obvious that

$$f_{b,N^*,\ell,Ord_{idl}[|Ord_{idl}|-1]} = f_{b,N^*,\ell+1}$$

as $Ord_{idl}[|Ord_{idl}|-1] = KN$, thus,

$$\mathscr{A}(G_{\texttt{core-package-construction}}(f_{b,\texttt{HS}}, f_{b,N^*,\ell+1}, f_{P,\texttt{E}_{\texttt{salt}}}, \texttt{true}),\\ G_{\texttt{core-package-construction}}(f_{b,\texttt{HS}}, f_{b,N^*,\ell+1}, f_{P,\texttt{E}_{\texttt{salt}}}, \texttt{true}), A) = 0.$$

Now we can finally do the hybridization argument

$$\begin{aligned}
&\mathscr{A}(G_{\texttt{core-hyb-pred},\ell,[\,]}, G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[|Ord_{idl}|-1]}, A)\\
&\leq \sum_{c=0}^{|Ord_{idl}|-1} \mathscr{A}(G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[c]}, G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[c+1]}, A)
\end{aligned}$$

as an induction proof over the size of Ord_{idl}. For the base case of length 0 we have a trivial equivalence. For the inductive case where $n = |Ord_{idl}| - 1$, we can apply the triangle inequality

$$\begin{aligned}
&\mathscr{A}(G_{\texttt{core-hyb-pred},\ell,[\,]}, G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[n+1]}, A)\\
&\overset{\Delta}{\leq} \mathscr{A}(G_{\texttt{core-hyb-pred},\ell,[\,]}, G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[n]}, A) +\\
&\quad \mathscr{A}(G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[n]}, G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[n+1]}, A),
\end{aligned}$$

and we apply the induction hypothesis to get

$$\sum_{c=0}^{n} \mathscr{A}(G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[c]}, G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[c+1]}, A) +\\ \mathscr{A}(G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[n]}, G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[n+1]}, A)$$

$$\leq \sum_{c=0}^{n+1} \mathscr{A}(G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[c]}, G_{\texttt{core-hyb-pred},\ell,Ord_{idl}[c+1]}, A).$$

Unfolding the full sum, we can split everything into cases given by Equations (5.7) to (5.10). There are three cases for `xtr` (i.e., `ES`, `HS`, and `AS`) where Equations (5.7) to (5.9) apply; the rest falls into `xpd` where Equation (5.10) applies. The sum thereby becomes

$$\begin{aligned}&\mathscr{A}(G_{\texttt{xtr},\texttt{ES},\ell}, A \circ R_{\texttt{xtr},\texttt{ES},\ell}) + \\ &\quad \mathscr{A}(G_{\texttt{xtr},\texttt{HS},\ell}, A \circ R_{\texttt{xtr},\texttt{HS},\ell}) + \\ &\quad \mathscr{A}(G_{\texttt{xtr},\texttt{AS},\ell}, A \circ R_{\texttt{xtr},\texttt{AS},\ell}) + \\ &\quad \sum_{n \in \texttt{XPD}} \mathscr{A}(G_{\texttt{xpd},n,\ell}, A \circ R_{\texttt{xpd},n,\ell}),\end{aligned}$$

concluding the proof of Equation (5.11). This is used in combination with

$$\begin{aligned}\mathscr{A}(G_{\texttt{core-SO-KEM}}, G^1_{\texttt{core}}, A_i) \leq \mathscr{A}(G_{\texttt{core-ki}}, G^1_{\texttt{core}}, A_i) + \\ \sum_{\ell=0}^{d} \mathscr{A}(G_{\texttt{core-hyb},\ell}, G_{\texttt{core-hyb},\ell+1}, A_i).\end{aligned} \tag{5.12}$$

The Proof of the Core Theorem

We now have all the prerequisites to prove the core theorem proof without any further interruptions. It proceeds by a stepping the inequality using the lemmas, or splitting it into parts, which can be shown independently. We start with

$$\begin{aligned}\mathscr{A}(G_{\texttt{core}}, A) &\overset{(5.1)}{\leq} \mathscr{A}(G_{acr}(f_{hash}), A \circ R_{cr}) + \mathscr{A}(G_{\texttt{core-hash}}, G^1_{\texttt{core}}, A) \\ &\overset{(5.2),\Delta}{\leq} \mathscr{A}(G_{acr}(f_{hash}), A \circ R_{cr}) + \\ &\quad (\mathscr{A}(G_{acr}(f_{xtr}), A \circ R_Z\ f_{xtr}) + \\ &\quad \mathscr{A}(G_{acr}(f_{xpd}), A \circ R_Z\ f_{xpd}) + \\ &\quad \mathscr{A}(G_{acr}(f_{xtr}), A \circ R_D\ f_{xtr}) + \\ &\quad \mathscr{A}(G_{acr}(f_{xpd}), A \circ R_D\ f_{xpd})) + \\ &\quad \mathscr{A}(G_{\texttt{core-D}}, G^1_{\texttt{core}}, A).\end{aligned}$$

Focusing on the last term, we get the inequality

$$\begin{aligned}\mathscr{A}(G_{\texttt{core-D}}, G^1_{\texttt{core}}, A) &\overset{\Delta}{\leq} \mathscr{A}(G_{\texttt{core-D}}, G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}}, A) + \mathscr{A}(G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}}, G^1_{\texttt{core}}, A) \\ &\overset{\texttt{D}\to\texttt{R}}{\leq} 0 + \mathscr{A}(G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}}, G^1_{\texttt{core}}, A) \\ &\leq \mathscr{A}(G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}}, G^1_{\texttt{core}}, A) \\ &\overset{(5.3)}{\leq} \max_i \mathscr{A}(G_{\texttt{core-D}}, G^1_{\texttt{core}}, A_i).\end{aligned}$$

Next we can look at each case of i; thus, we get

$$\begin{aligned}
&\mathscr{A}(G_{\texttt{core-D}}, G^1_{\texttt{core}}, A_i)\\
&\overset{\Delta}{\leq} \mathscr{A}(G_{\texttt{core-D}}, G_{\texttt{core-SO-KEM}}, A_i) + \mathscr{A}(G_{\texttt{core-SO-KEM}}, G^1_{\texttt{core}}, A_i)\\
&\overset{\Delta}{\leq} \mathscr{A}(G_{\texttt{core-D}}, G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}}, A_i) +\\
&\quad \mathscr{A}(G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}}, G_{\texttt{core-SO-KEM}}, A_i)) +\\
&\quad \mathscr{A}(G_{\texttt{core-SO-KEM}}, G^1_{\texttt{core}}, A_i)\\
&\overset{(5.4)}{\leq} \mathscr{A}(G_{\texttt{core-D}}, G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}}, A_i) +\\
&\quad \mathscr{A}(G_{\texttt{core-SO-KEM}}, A_i \circ R_{\texttt{so-kem}}) +\\
&\quad \mathscr{A}(G_{\texttt{core-SO-KEM}}, G^1_{\texttt{core}}, A_i).
\end{aligned}$$

Again, we focus on first and last terms, which gives us the following inequality

$$\begin{aligned}
&\mathscr{A}(G_{\texttt{core-D}}, G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}}, A_i) + \mathscr{A}(G_{\texttt{core-SO-KEM}}, G^1_{\texttt{core}}, A_i)\\
&\overset{(5.12)}{\leq} \mathscr{A}(G_{\texttt{core-D}}, G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}}, A_i) +\\
&\quad \mathscr{A}(G_{\texttt{core-ki}}, G^1_{\texttt{core}}, A_i) +\\
&\quad \sum_{0\leq\ell\leq d} \mathscr{A}(G_{\texttt{core-hyb},\ell}, G_{\texttt{core-hyb},\ell+1}, A_i)\\
&\overset{(5.11)}{\leq} \mathscr{A}(G_{\texttt{core-D}}, G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}}, A_i) +\\
&\quad \mathscr{A}(G_{\texttt{core-ki}}, G^1_{\texttt{core}}, A_i) +\\
&\quad \sum_{0\leq\ell\leq d} (\mathscr{A}(G_{\texttt{xtr},\texttt{ES},\ell}, A_i \circ R_{\texttt{xtr},\texttt{ES},\ell}) +\\
&\qquad \mathscr{A}(G_{\texttt{xtr},\texttt{HS},\ell}, A_i \circ R_{\texttt{xtr},\texttt{HS},\ell}) +\\
&\qquad \mathscr{A}(G_{\texttt{xtr},\texttt{AS},\ell}, A_i \circ R_{\texttt{xtr},\texttt{AS},\ell}) +\\
&\qquad \sum_{n\in\texttt{XPD}} \mathscr{A}(G_{\texttt{xpd},n,\ell}, A_i \circ R_{\texttt{xpd},n,\ell})).
\end{aligned}$$

We can apply Equation (5.5) to get

$$\mathscr{A}(G_{\texttt{core-ki}}, G^1_{\texttt{core}}, A_i) \leq \mathscr{A}(G_{pi,[\texttt{E}_{\texttt{salt}}],R}, A_i \circ R_{pi,[\texttt{E}_{\texttt{salt}}]}) + \mathscr{A}(G_{pi,O^*,R}, A_i \circ R_{pi,O^*})$$

and Equation (5.6) for

$$\mathscr{A}(G_{\texttt{core-D}}, G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}}, A_i) = 0.$$

Thus, reducing the inequality to

$$\begin{aligned}
&\mathscr{A}(G_{\texttt{core-D}}, G_{\texttt{core-R}_{\texttt{E}_{\texttt{salt}}}}, A_i) + \mathscr{A}(G_{\texttt{core-SO-KEM}}, G^1_{\texttt{core}}, A_i)\\
&\leq \mathscr{A}(G_{pi,[\texttt{E}_{\texttt{salt}}],R}, A_i \circ R_{pi,[\texttt{E}_{\texttt{salt}}]}) + \mathscr{A}(G_{pi,O^*,R}, A_i \circ R_{pi,O^*}) +
\end{aligned}$$

$$
\begin{aligned}
&\sum_{0\le\ell\le d}(\mathscr{A}(G_{\texttt{xtr},\texttt{ES},\ell}, A_i \circ R_{\texttt{xtr},\texttt{ES},\ell}) + \\
&\quad \mathscr{A}(G_{\texttt{xtr},\texttt{HS},\ell}, A_i \circ R_{\texttt{xtr},\texttt{HS},\ell}) + \\
&\quad \mathscr{A}(G_{\texttt{xtr},\texttt{AS},\ell}, A_i \circ R_{\texttt{xtr},\texttt{AS},\ell}) + \\
&\quad \sum_{n\in\texttt{XPD}} \mathscr{A}(G_{\texttt{xpd},n,\ell}, A_i \circ R_{\texttt{xpd},n,\ell})).
\end{aligned}
$$

Combining the parts, we get a final security bound of

$$
\begin{aligned}
\mathscr{A}(G_{\texttt{core}}, A) \le\ & \mathscr{A}(G_{acr}(f_{hash}), A \circ R_{cr}\ f_{hash}) + \\
& \mathscr{A}(G_{acr}(f_{xtr}), A \circ R_Z\ f_{xtr}) + \mathscr{A}(G_{acr}(f_{xpd}), A \circ R_Z\ f_{xpd}) + \\
& \mathscr{A}(G_{acr}(f_{xtr}), A \circ R_D\ f_{xtr}) + \mathscr{A}(G_{acr}(f_{xpd}), A \circ R_D\ f_{xpd}) + \\
& \max_i(\mathscr{A}(G_{\texttt{core-SO-KEM}}, A_i \circ R_{\texttt{so-kem}}) + \\
& \quad \mathscr{A}(G_{pi,[\texttt{E}_{\texttt{salt}}],R}, A_i \circ R_{pi,[\texttt{E}_{\texttt{salt}}]}) + \mathscr{A}(G_{pi,O^*,R}, A_i \circ R_{pi,O^*}) \\
& \quad \sum_{0\le\ell\le d}(\mathscr{A}(G_{\texttt{xtr},\texttt{ES},\ell}, A_i \circ R_{\texttt{xtr},\texttt{ES},\ell}) + \\
& \quad\quad \mathscr{A}(G_{\texttt{xtr},\texttt{HS},\ell}, A_i \circ R_{\texttt{xtr},\texttt{HS},\ell}) + \\
& \quad\quad \mathscr{A}(G_{\texttt{xtr},\texttt{AS},\ell}, A_i \circ R_{\texttt{xtr},\texttt{AS},\ell}) + \\
& \quad\quad \sum_{n\in\texttt{XPD}} \mathscr{A}(G_{\texttt{xpd},n,\ell}, A_i \circ R_{\texttt{xpd},n,\ell})),
\end{aligned}
$$

concluding the proof of the core theorem.

Instantiating the Proof

We have a (partial) proof that a TLS-like key scheduler is secure. To apply this proof, we implement such a protocol and show that it fulfills all the initial requirements. That is all the data structures used in the implementation maps to the one used in the proof. Furthermore, the implementation needs to have at least the required set of keys. Given this, we show the initial package and functions (`xtr`, `xpd`, `prntN`, `label`) can be instantiated using the implementation.

Chapter 6

CryptoConCert: A Framework for Secure Rust Smart Contract Verification, with an Application to Voting

First we introduce the Schnorr (§6.2) and CDS (§6.3) *zero-knowledge proofs* used in the CryptoConCert paper in full detail. Then we present the paper, followed by an alternative (paper) proof for maximum ballot secrecy. We give both a general outline of the proof and a technical description of the proof. Finally, we describe the possible improvements and modifications that can be made to the implementation of the open vote network protocol.

6.1 Properties of Σ-protocols

We will give a general description of the cryptographic constructions used in Subsection 2.4 of the paper. This will serve as an introduction to the structure and properties of Σ-protocols [36]. We then apply it in the two following sections for the Schnorr and CDS protocols.

To define a Σ-protocol, we first state the zero-knowledge *proof statement* we want to show and the witness that exhibits it. Next we define the protocol by describing the procedure for the *committer* and the *validator*. To follow the Σ-protocol template, we define the protocol based on the *three messages* (initial message, challenge, and response). The committer tries to convince the validator that the statement is true using their knowledge of the witness. Correctness can be checked by showing the validators' checks after the response never fails if the protocol is followed.

We show the conversation is secure by defining a *simulator*, which needs to produce a valid transcript having the same distribution. The simulator is only given the public input and a random challenge. Showing the existence of a simulator ensures that any conversation is *refutable*, since an actual conversation and a simulated one cannot be distinguished; thus, no information should be gained from seeing the transcript.

Finally, we define a *witness extractor*, which, given two different challenge and response pairs for the same initial message, produces a witness of the statement. The extractor shows that anyone who can convince the validator must know have *knowledge of the witness*, as being able to produce a response to two different challenges is enough to extract the witness.

6.2 The Schnorr Protocol

The goal with the Schnorr protocol [84] is to show we know the private key m for a public key h, without leaking any extra information about the private key. Thus the zero-knowledge (ZK) proof statement is

$$h = g^m,$$

where m is a witness of the statement.

Correctness of the Committer and the Validator

The definitions of the committer and validator for the Schnorr protocol are in Figure 6.1. We check correctness by ensuring the final check is valid if both parties

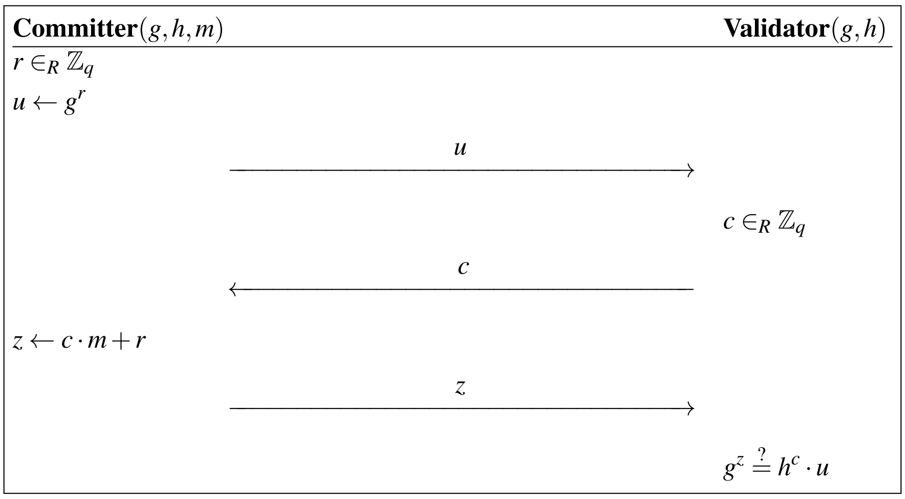


Figure 6.1: The Schnorr protocol committer and validator

adhere to the protocol

$$\begin{aligned} h^c \cdot u &= (g^m)^c \cdot g^r \\ &= g^{m \cdot c + r} \\ &= g^z. \end{aligned}$$

The Simulator and Special Honest Verifier Zero-Knowledge (SHVZK)

The simulator for the Schnorr protocol needs to produce a transcript (u,c,z) given public input h and the random challenge c. The simulator is defined in Figure 6.2. We

```
**Simulator**$(h,c)$
$z \in_R \mathbb{Z}_q$
$u \leftarrow \frac{g^z}{h^c}$
ret $(u,c,z)$
```

Figure 6.2: The Schnorr protocol simulator

check that all the definitions from the prover are equal to those given by the simulator

$$\begin{aligned} h^c \cdot u &= h^c \cdot \frac{g^z}{h^c} \\ &= g^z. \end{aligned}$$

We also need to ensure that the checks still hold

$$\begin{aligned} u &= \frac{g^z}{h^c} \\ &= g^{z-c\cdot m} \\ &= g^{(c\cdot m+r)-c\cdot m} \\ &= g^r. \end{aligned}$$

The Extractor and Special Soundness

The extractor should find a witness to the ZK statement given the two transcripts, with the same initial message but different challenges and responses, e.g., (u,c,z) and (u,c',z'). We define the extractor in Figure 6.3. The extractor computes a valid

```
**Extractor**$((c,z),(c',z'))$
$m \leftarrow \frac{z-z'}{c-c'}$
ret $m$
```

Figure 6.3: The Schnorr protocol extractor

witness as

$$\begin{aligned} \frac{z-z'}{c-c'} &= \frac{(m\cdot c+r)-(m\cdot c'+r')}{c-c'} \\ &= \frac{m\cdot(c-c')}{c-c'} \end{aligned}$$

since $c \neq c'$ we get

$$= m.$$

This concludes the proof of special soundness.

6.3 The Cramer-Damård-Shoenmaker (CDS) Construction

The OR proof, also known as the Cramer, Damgård, Schoenmakers (CDS) construction [33] or a specialized case of Disjunctive Chaum-Pedersen (DCP) [23], is a zero-knowledge (ZK) protocol. The idea is to show a value y is encoding v which is either a 0 or a 1, without leaking which. The ZK statement is

$$x = g^m \quad \wedge \quad y = h^m \cdot g^v \quad \wedge \quad v \in \{0,1\},$$

where a witness of the statement is (m, v). To define the Σ-protocol and show zero-knowledge, we need to define a committer, validator, and extractor, for which we need to show correctness, special honest verifier zero-knowledge, and special soundness.

Correctness of the Committer and the Validator

The committer will generate a pair of values, one based on the real values, and the other sampled at random. To determine which part of the pair uses the real values and which uses the sampled once, we look at the value of v. The Σ-protocol can be seen in Figure 6.4. To prove the correctness of the protocol, each check at the end of the protocol needs to be proven to succeed, given the protocol is followed honestly. Checking $c = d_1 + d_2$ when $v = 1$

$$d_1 + d_2 = d_1 + (c - d_1) = c$$

and when $v = 0$

$$d_1 + d_2 = (c - d_2) + d_2 = c$$

follows directly from the definitions. The check $g^{r_1} \cdot x^{d_1} = a_1$ is true by definition for $v = 1$, so we compute for $v = 0$ as follows

$$\begin{aligned} g^{r_1} \cdot x^{d_1} &= g^{r_1} \cdot g^{m \cdot d_1} \\ &= g^{r_1 + m \cdot d_1} \\ &= g^{w - m \cdot d_1 + m \cdot d_1} \\ &= g^{w} \\ &= a_1. \end{aligned}$$

Conversely, $g^{r_2} \cdot x^{d_2} = a_2$ is true for $v = 0$ by definition, and the computation for $v = 1$ is similar to the above

$$g^{r_2} \cdot x^{d_2} = g^{r_2} \cdot g^{m \cdot d_2}$$

Committer(g,h,v,m)		**Validator**(g,h)
v = 1	**v = 0**	
$w, r_1, d_1 \in_R \mathbb{Z}_q$	$w, r_2, d_2 \in_R \mathbb{Z}_q$	
$x \leftarrow g^m$	$x \leftarrow g^m$	
$y \leftarrow h^m \cdot g$	$y \leftarrow h^m$	
$a_1 \leftarrow g^{r_1} \cdot x^{d_1}$	$a_1 \leftarrow g^w$	
$b_1 \leftarrow h^{r_1} \cdot y^{d_1}$	$b_1 \leftarrow h^w$	
$a_2 \leftarrow g^w$	$a_2 \leftarrow g^{r_2} \cdot x^{d_2}$	
$b_2 \leftarrow h^w$	$b_2 \leftarrow h^{r_2} \cdot \left(\frac{y}{g}\right)^{d_2}$	
	$\xrightarrow{(x,y,a_1,b_1,a_2,b_2)}$	
		$c \in_R \mathbb{Z}_q$
	$\xleftarrow{c}$	
$d_2 \leftarrow c - d_1$	$d_1 \leftarrow c - d_2$	
$r_2 \leftarrow w - m \cdot d_2$	$r_1 \leftarrow w - m \cdot d_1$	
	$\xrightarrow{(r_1,d_1,r_2,d_2)}$	
		$c \stackrel{?}{=} d_1 + d_2$
		$a_1 \stackrel{?}{=} g^{r_1} \cdot x^{d_1}$
		$b_1 \stackrel{?}{=} h^{r_1} \cdot y^{d_1}$
		$a_2 \stackrel{?}{=} g^{r_2} \cdot x^{d_2}$
		$b_2 \stackrel{?}{=} h^{r_2} \cdot \left(\frac{y}{g}\right)^{d_2}$

Figure 6.4: Definition of committer and validator

$$\begin{aligned}
&= g^{r_2+d_2\cdot m}\\
&= g^{w-m\cdot d_2+m\cdot d_2}\\
&= g^{w}\\
&= a_2.
\end{aligned}$$

The check $h^{r_1}\cdot y^{d_1}=b_1$ again follows by definition for $v=0$, so we check $v=1$ by

$$\begin{aligned}
h^{r_1}\cdot y^{d_1} &= h^{r_1}\cdot h^{m\cdot d_1}\\
&= h^{r_1+m\cdot d_1}\\
&= h^{w-m\cdot d_1+m\cdot d_1}\\
&= h^{w}\\
&= b_1.
\end{aligned}$$

Finally, we check $h^{r_2}\cdot\left(\frac{y}{g}\right)^{d_2}=b_2$, which breaks the symmetry a bit but is still true by definition for $v=1$. Checking for $v=0$ by

$$\begin{aligned}
h^{r_2}\cdot\left(\frac{y}{g}\right)^{d_2} &= h^{r_2}\cdot\left(\frac{h^m\cdot g}{g}\right)^{d_2}\\
&= h^{r_2}\cdot(h^m)^{d_2}\\
&= h^{r_2}\cdot h^{m\cdot d_2}\\
&= h^{r_2+m\cdot d_2}\\
&= h^{w-m\cdot d_2+m\cdot d_2}\\
&= h^{w}\\
&= b_2.
\end{aligned}$$

Thus, all the checks are correct for both values of v.

The Simulator and Special Honest Verifier Zero-Knowledge (SHVZK)

The goal of the simulator is to produce a transcript of the protocol without interacting with the prover. The transcript still needs to have the correct distribution. For CDS the simulator has to produce

$$((x,y),(a_1,b_1,a_2,b_2),c,(r_1,d_1,r_2,d_2))$$

given (x,y) and c. This is achieved by the procedure in Figure 6.5. We need to check the correctness of the generated protocol. That is, we need to check the definitions are equal to once defined by a prover. We are given (a_1,b_1,a_2,b_2), and define

$$d_2=c-d_1$$

and

$$r_2=w-m\cdot d_2$$

```
**Simulator**$(g, h, (x, y), c)$
$d_2, r_1, r_2 \in_R \mathbb{Z}_q$
$d_1 \leftarrow c - d_2$
$a_1 \leftarrow g^{r_1} \cdot x^{d_1}$
$b_1 \leftarrow h^{r_1} \cdot y^{d_1}$
$a_2 \leftarrow g^{r_2} \cdot x^{d_2}$
$b_2 \leftarrow h^{r_2} \cdot \left(\frac{y}{g}\right)^{d_2}$
ret $((x,y), (a_1, b_1, a_2, b_2), c, (r_1, d_1, r_2, d_2))$
```

Figure 6.5: CDS simulator

when $v = 1$, and for $v = 0$ we define

$$d_1 = c - d_2$$

and

$$r_1 = w - m \cdot d_1.$$

The correctness argument for the validator, for the $v = 1$ case, is

$$\begin{aligned} c - d_1 &= c - (c - d_2) \\ &= d_2 \end{aligned}$$

and

$$\begin{aligned} w - m \cdot d_2 &= (r_2 + m \cdot d_2) - m \cdot d_2 \\ &= r_2. \end{aligned}$$

For $v = 0$, the value of d_1 is defined exactly as the prover does; furthermore, the check for r_1 is correct by

$$\begin{aligned} w - m \cdot d_1 &= w - m \cdot (c - d_2) \\ &= (r_1 + m \cdot (c - d_2)) - m \cdot (c - d_2) \\ &= r_1. \end{aligned}$$

Thus, all the simulated definitions have equal distributions.

The Extractor and Special Soundness

In a Σ-protocol, the extractor is given two transcripts with the same initial message

$$(x, y), (a_1, b_1, a_2, b_2)$$

```
**Extractor**$((r_1,d_1,r_2,d_2),c,(r'_1,d'_1,r'_2,d'_2),c')$
$v \leftarrow (d_1 \overset{?}{=} d'_1)$
if $v = 0$
    $m \leftarrow \frac{r'_1 - r_1}{d_1 - d'_1}$
else
    $m \leftarrow \frac{r'_2 - r_2}{d_2 - d'_2}$
ret $(m,v)$
```

Figure 6.6: CDS extractor

but two different challenges,

$$(r_1,d_1,r_2,d_2),c$$

and

$$(r'_1,d'_1,r'_2,d'_2),c'.$$

It then needs to extract a valid witness for the statement. We do this by the checks in Figure 6.6. If both runs validate and the challenges are different

$$c \neq c'$$

$$c = d_1 + d_2 \qquad c' = d'_1 + d'_2 \tag{6.1}$$

$$a_1 = g^{r_1} \cdot x^{d_1} \qquad a_1 = g^{r'_1} \cdot x^{d'_1} \tag{6.2}$$

$$b_1 = h^{r_1} \cdot y^{d_1} \qquad b_1 = h^{r'_1} \cdot y^{d'_1} \tag{6.3}$$

$$a_2 = g^{r_2} \cdot x^{d_2} \qquad a_2 = g^{r'_2} \cdot x^{d'_2} \tag{6.4}$$

$$b_2 = h^{r_2} \cdot \left(\frac{y}{g}\right)^{d_2} \qquad b_2 = h^{r'_2} \cdot \left(\frac{y}{g}\right)^{d'_2}, \tag{6.5}$$

then we need to find a witness for the statement of the protocol

$$x = g^m \quad \wedge \quad y = h^m \cdot g^v \quad \wedge \quad v \in \{0,1\}.$$

We can conclude $(d_1 \neq d'_1) \vee (d_2 \neq d'_2)$ from $c \neq c'$. We insert the definitions into the statement to get

$$\begin{cases} x = g^{\frac{(r'_1 - r_1)}{(d_1 - d'_1)}} \quad \wedge \quad y = h^{\frac{(r'_1 - r_1)}{(d_1 - d'_1)}} \quad (d_1 \neq d'_1) \\ x = g^{\frac{(r'_2 - r_2)}{(d_2 - d'_2)}} \quad \wedge \quad y = h^{\frac{(r'_2 - r_2)}{(d_2 - d'_2)}} \cdot g \quad (d_2 \neq d'_2) \end{cases}. \tag{6.6}$$

We know v is a Boolean; it is either 0 or 1. We can therefore remove the last condition of the ZK statement. We equate the two different definitions of Equations (6.2) to (6.5) by

$$\begin{aligned}
&c \neq c' \\
&g^{r'_1 - r_1} = x^{d_1 - d'_1} \\
&h^{r'_1 - r_1} = y^{d_1 - d'_1} \\
&g^{r'_2 - r_2} = x^{d_2 - d'_2} \\
&h^{r'_2 - r_2} = \left(\frac{y}{g}\right)^{d_2 - d'_2} .
\end{aligned}$$

We then plug these definitions into Equation (6.6) to get

$$\begin{cases}
x = x^{\frac{(d_1 - d'_1)}{(d_1 - d'_1)}} & \land \quad y = y^{\frac{(d_1 - d'_1)}{(d_1 - d'_1)}} \qquad (d_1 \neq d'_1) \\
x = x^{\frac{(d_2 - d'_2)}{(d_2 - d'_2)}} & \land \quad y = \left(\frac{y}{g}\right)^{\frac{(d_2 - d'_2)}{(d_2 - d'_2)}} \cdot g \quad (d_2 \neq d'_2)
\end{cases},$$

which simplifies into

$$\begin{cases}
x = x & \land \quad y = y \qquad (d_1 \neq d'_1) \\
x = x & \land \quad y = \left(\frac{y}{g}\right) \cdot g \quad (d_2 \neq d'_2)
\end{cases}.$$

Thus, we have shown that the extractor of Figure 6.6 correctly extracts the witness!

6.4 The Paper

The paper "CryptoConCert: A Framework for Secure Rust Smart Contract Verification, with an Application to Voting" collects the work on formalizing the OVN smart contract. It is currently under submission; thus, the deanonymized version of the paper follows in full. A summary of the paper is made in §6.5 for convenience.

CryptoConCert: A Framework for Secure Rust Smart Contract Verification, with an Application to Voting

LASSE LETAGER HANSEN, Aarhus University , Denmark
ESKE HOY NIELSEN, Aarhus University , Denmark
NIKOLAJ SIDORENCO
BAS SPITTERS, Aarhus University , Denmark

Smart contracts carry large amounts of monetary value, are immutable, but of surveyable size. They thus form a welcome target for formal verification. However, until now, the cryptographic aspects of smart contracts have not been formally scrutinized. In this work, we describe a general framework for formally verifying the cryptographic security and (trace-based) correctness of smart contracts. We apply program verification techniques to prove security by using the SSProve library in Rocq, which provides a probabilistic relational program logic built on Dijkstra monads. We use Rust as a smart contract language to produce WebAssembly (Wasm), as this combination is gaining traction in the blockchain space. We use the Hax toolchain to embed our Rust smart contract into Rocq. For added assurance, we use ConCert/CertiRocq-Wasm to compile this in a verified way to Wasm, which can be run on chain. As a case study, we will apply this to a voting protocol based on homomorphic encryption. Specifically, we analyze the Open Vote Network (OVN) smart contract in a safe subset of Rust and extract it in a verified way to on-chain Wasm.

ACM Reference Format:
Lasse Letager Hansen, Eske Hoy Nielsen, Nikolaj Sidorenco, and Bas Spitters. 2026. CryptoConCert: A Framework for Secure Rust Smart Contract Verification, with an Application to Voting. In *Proceedings of Principles of Programming Languages (POPL'26).* ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Due to its critical place in society, cryptographic software has been a popular application area for formal methods. This includes the internet stack, voting software, payments, and blockchain systems; see [6] for an overview. The Everest project [66], which aims to verify HTTPS, is one of the highlights in the field of high-assurance cryptographic software (HACS). HACS started with the verification of primitives and abstract protocols. For primitives one verifies the functional correctness of highly optimized implementations. One proves the security properties of primitives and simple protocols in the computational model. This is usually treated by reasoning in a probabilistic Hoare logic, most prominently in EasyCrypt [7]. For larger protocols, one tends to use the symbolic ('Dolev-Yao') model, where one assumes perfect behavior of cryptographic primitives. The symbolic model lends itself well to automatic verification. More recently, the emphasis is shifting to the modular verification of realistic *implementations* of bigger protocols. The verification [16] of a Rust implementation of TLS 1.3 is one example of this.

Authors' addresses: Lasse Letager Hansen Aarhus University, Denmark, letager@cs.au.dk; Eske Hoy Nielsen Aarhus University, Denmark, eske@cs.au.dk; Nikolaj Sidorenco; Bas Spitters, spitters@cs.au.dk Aarhus University, Denmark.

POPL'26, 11-17th of January, Rennes, France
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Smart contracts are a similar attractive topic for formal verification. They are an important part of the decentralized economy, which is currently valued at several trillions. Moreover, the decentralized economy is closely intertwined with the traditional economy. However, bugs in smart contracts have been exploited for hundreds of millions[1]. Smart contracts are immutable, so vulnerabilities need to be caught before deployment. Fortunately, one can use the cryptographer's view of a blockchain as a secure append-only log to treat smart contracts as regular programs appending to this log. With this view, smart contracts are moderate in size. This combination of factors makes them an ideal target for formal verification. Some initial steps in this direction have been taken; see Subsection 8.4. Most of these are automated methods for simple properties. Some more complex correctness properties, such as trace properties, have been proved using interactive theorem proving. For example, the total number of coins in a decentralized exchange stays constant when trading [58]. However, important smart contracts also include cryptographic aspects implemented in smart contracts. Merkle trees or zero-knowledge-based contracts, such as the layer-2 contracts, which address the performance issues of (layer-1) blockchains, are an example of this. Another example is on-chain voting, used for minor elections such as boardroom voting or communal elections; see Subsection 8.1.

To verify properties of cryptographic smart contracts, one needs to combine probabilistic reasoning from HACS with trace-based smart contract verification. To do so, we contribute a general framework, CryptoConCert, to reason about such cryptographic smart contracts. Our approach is similar to the one for the verification of cryptographic protocols, where one proves the security of primitives in the (probabilistic) computational model, and then proves the security of the protocol in the (trace based) symbolic model. Usually, these two models are not formally connected, and both are treated with dedicated tools. Hax provides a way to make sure they work from the same Rust code [16]. ConCert is more precise than the symbolic verifiers for cryptography, in that it verifies concrete implementations. So, we improve on the state of the art in cryptographic verification, by formally proving that the code in our probabilistic model is equivalent to the one in the trace model.

To evaluate CryptoConCert, we consider a specific voting contract, the Open Vote Network (OVN) protocol [43], as a case study. This is an application of blockchains to voting. More generally, the topics of voting and blockchains are intertwined: secure voting is used for internal mechanisms of the blockchain, such as for consensus, governance, and decentralized autonomous organizations.

Blockchains generally come in two flavors: 1. account-based (like Ethereum), where each party has a "bank account"; or 2. UTXO-based (like Bitcoin), where the blockchain records the transactions between the parties. The two flavors are equivalent [19, 21]. CryptoConCert, like ConCert, works for general account-based blockchains. Here, we instantiate it with the (Rust/Wasm)-combination of smart contract languages, as this is gaining popularity; see Subsection 2.2. We do so by extending the Hax Rust verification framework to smart contracts. As all Rust based blockchains have their own glue code for smart contracts, we specialize this to the Concordium blockchain to ensure it actually runs in practice.

Proving the security and correctness of smart contracts is important, but one also needs to ensure that they are correctly compiled to on-chain code. For comparison, a number of critical bugs have been found in Ethereum's Solidity compiler. To avoid this in the future, the Solidity compiler includes a built-in model checker, SMTChecker. The compiler has also been an active subject for formal verification, but no fully verified compiler for Solidity exists; see Section 8.4. Rust is a popular industry strength language which has been designed with clear PL-concepts

[1]https://go.chainalysis.com/2025-Crypto-Crime-Report.html

in mind. However, the full Rust language is still in the process of being specified[2] and a verified compiler, like CompCert [53] for C, does not exist. So, for added certainty, we connect Hax with CertiRocq-Wasm, thus providing a verified compiler-step that can replace the steps from (T)HIR to MIR to LLVM to Wasm in the Rust compiler. The performance penalty for this is moderate; see Section 7.

1.1 Contributions

In this work, we contribute a framework for the verification of the (probabilistic) security and (trace-based) correctness of cryptographic smart contracts, focusing on the Rust/Wasm languages. We do this by the following contributions:

- We extend the industry strength Hax [14] framework for Rust verification to support smart contracts. This is by analogue to the use of Hax for protocol verifiers, but we provide a tighter connection between the probabilistic aspects and the trace aspects; see Section 3.
- We provide the first smart contract to be formally proven cryptographically secure (see Subsection 5.1), and we prove it to be correct (see section 6). This is also the first realistic verified Rust smart contract. To prove the security of the implementation, we extend the modular SSP-style for reasoning about cryptographic *protocols* to *implementations*.
- We evaluate this framework on an OVN Rust *implementation*, and provide a verified Wasm implementation. In the process, we found an incompleteness in the security argument of the OVN protocol and provide a mitigation; see Subsection 5.1.1.

1.2 Structure

Section 2 introduces relevant background material. Section 3 explains the general structure and use of our framework, CryptoConCert. In Section 4, we introduce the Open Vote Network as a case study of our framework. Section 5 and Section 6 prove security and correctness of our OVN implementation. In Section 7, we evaluate the implementation of OVN. Finally, we discuss related work on the verification of voting, smart contracts, Rust, and cryptography in Section 8.

2 BACKGROUND

2.1 ConCert

A challenge when working with permissionless blockchains is that the adversary is very strong since it has complete knowledge of the system and full access to the network. In particular, an adversary has access to the smart contracts code and state and can interact with any smart contract on the network. Working with smart contracts is further complicated due to the complex execution model in which smart contracts can call arbitrary smart contracts. For example, some smart contracts make calls to other smart contracts based on user input, meaning that the code needs to handle interactions with arbitrary code.

To verify correctness of smart contracts we use ConCert [4] which is a framework for developing, testing, and verifying smart contracts in Rocq. An advantage of ConCert is that it models the full execution model of the blockchain abstractly as a totally ordered broadcast, in the spirit of the cryptographer's ideal functionality. This means that it is possible to both reason about the functional correctness of smart contract implementations and prove more complex properties about blockchain traces and smart contract interactions. Furthermore, since the model is of an abstract blockchain, it means that several blockchains can be targeted using the same smart contract implementation without having to prove correctness of the contract for each blockchain individually. ConCert's model follows Ethereum's account-based model. In this model, smart contracts are special addresses

[2]E.g. https://github.com/minirust/minirust

with an additional state and code. This code is executed when transactions are sent to the address and the state is updated according to the computation. In ConCert smart contracts are modeled as pure state transforming functions.

ConCert implements verified program extraction of smart contracts to several blockchain and smart contract languages such as Rust and CameLIGO through MetaRocq's typed erasure [3] and Wasm through CertiRocq-Wasm [57].

2.2 Rust, Hax, and Wasm Smart Contracts

Rust is a type- and memory-safe imperative programming language that avoids the use of a garbage collector by using a borrow checker to enforce an ownership model. Rust supports most programming idioms with a zero-cost abstraction. Some design principles of Rust are memory safety without garbage collection, and abstraction without overhead. Rust is very performant, comparable with C. Our main interest in Rust is its popularity in security-critical applications and its clear semantics for this application domain.

Traits in Rust provide ad hoc polymorphism, inspired by Haskell's type classes. A trait is a collection of functions and type definitions that can be generalized over some type parameters. Similar to type classes, a type can instantiate a trait with an implementation, and functions can be defined over any type implementing a trait.

2.2.1 Hax. Hax is a framework for writing cryptographic primitives, specifications, and protocols in a subset of Rust, which we call Rust$_{Hax}$. Hax translates Rust programs to programs that are more amenable to verification. The translation happens in well-defined phases [14].

Rust is a popular language for writing cryptographic specifications (=reference implementations) and implementations. For cryptographic reference implementations, only a limited fragment of safe Rust is needed. It turns out that for efficient implementations of protocols, it is often enough to write a reference implementation and replace the cryptographic primitives with highly optimized implementations. Hax allows us to translate such reference implementations into a selection of tools and backends (F*, Rocq, Lean, SSProve, ProVerif). Hax supports safe Rust (e.g. no raw pointers, async, static) with partial support for mutable borrows, nested matching, and bounded iteration.

Each backend specifies a set of features it allows. Hax then runs a sequence of transformation phases to reduce to the requested feature set. Finally, Hax can print the AST of the transformed code to the syntax of (a DSL) in the backend.

For SSProve (see Subsection 2.4), there is a shallow embedding into its imperative language. There is also a shallow functional embedding into Rocq with an equivalence proof to the imperative embedding. This proof is built during translation and can be seen as translation validation or a simple form of realizability. Language constructs like loops, early returns, let bindings, matching, and if-statements are translated into the code monad of SSProve by using Rocq features (if, match, fold, let) or by combining the code monad with another monad, such as the result monad [44].

2.2.2 Rust$_{Hax}$. The Hax framework is built to verify the Rust$_{Hax}$ subset of safe Rust. This subset is designed for the verification of security-critical protocols. These typically have a straightforward functional semantics, which can be embedded in proof assistants by shadowing lets. It supports basic data types (bool, u8, ..., u128, string, float), structs, enums (and simple pattern matching), collections (vectors and arrays), and traits. Hax only has limited support for some of the more advanced features. It does not currently support mutual borrows as return types. Generic types, similar to parametric polymorphism, are allowed in most types and definitions. The support for explicit lifetimes is limited by the functional semantics. The lifetime annotations are not translated, since the borrow checker is simple for Rust$_{Hax}$ and treated by Hax. Macros are supported by evaluation and unfolding. However, they should be used with care, as this can clutter the embedding. Most control flow structures are supported. The Hax frontend allows non-wellfounded recursive definitions and unbounded iteration. However, some backends do not allow those general definitions. So, the user will often be careful to avoid those.

The use of similar restricted subsets of Rust is common among analysis and verification tools; see Subsection 8.2. The semantics, where present, for these tools agree on the Rust_{Hax} subset, and efforts are underway to fully specify the semantics of a larger subset of Rust.

Like other security critical protocols, most smart contracts can naturally be captured in Rust_{Hax}. Many smart contracts can naturally be written as pure programs transforming the (blockchain) state. Often, they implement a state transition machine.

2.2.3 Wasm smart contracts. The Ethereum blockchain uses its own EVM virtual machine, and it provides Solidity, a JavaScript-inspired language, to compile to it. A number of blockchains have chosen Wasm as their virtual machine. These include Polkadot, NEAR, Cosmos, Solana, Dfinity, Hyperledger, and Concordium. Wasm is also used in Ethereum's sidechains/rollups.

We name a few motivations for the choice of Wasm as on-chain VM. It is a popular compilation target for traditional programming languages. Wasm is very fast, which is important for blockchain applications, as each computation needs to be carried out by all nodes. Moreover, Wasm is formally specified, deterministic (if one removes floats), and has no undefined behavior. Wasm provides memory safety, control flow integrity, and capability based isolation, as it does not provide *e.g.* disk or network access.

Wasm does not have a garbage collector (GC). This gives predictable performance, which is advantageous on-chain. So, Rust is a natural choice for a smart contract language, as it does not have a GC, it has efficiency close to C, is popular with developers, and has an expressive type system that helps one catch bugs early.

Aside, RISC-V has been proposed as a replacement for Ethereum's EVM. This would make it possible to use Rust as a smart contract language for Ethereum as well. Moreover, our verified compilation with CertiRocq in Section 7, could easily be retargeted to RISC-V, as this is already provided by the CompCert backend.

2.3 Cryptography

In Section 2.4, we will construct *modular* proofs of protocol security using reduction-style proofs. In this section, we first introduce the cryptographic primitives, from which we construct the larger procedures and protocols. The security and correctness proofs for the primitives follow from standard cryptographic assumptions, which we will explain in the following subsections.

2.3.1 Commitment schemes. The first primitive we will look at is a commitment, where a party commits to using a specific value for a computation. This primitive ensures that if the party publishes their value used in the computation, one can check the commitment to ensure it is the same value. Thus, a commitment scheme requires a function for obtaining a commitment to the value, and a function for validating correctness of the commitment. The security properties are as follows.

- Hiding: private inputs used are not leaked.
- Binding: After choosing a value, it cannot be changed.

Here, correctness entails that committing to a value will always validate correctly; binding forces the value to stay consistent throughout the protocol; and hiding secures the value from being disclosed before the party itself publishes it.

One could implement such a commitment scheme using a secure one-way hash function (e.g. SHA-3). Validation of the commitment is done by checking if the published commitment is equal to the hash of the published value. The correctness follows from the hash function being deterministic.

Because the function used is a secure one-way hash, where collisions are hard to find, the binding property follows, as we cannot choose another value with the same hash without breaking the guarantees of the hash function [31, 34]. Finally, the hiding property follows from the hash function being hard to invert [31, 34].

2.3.2 Zero-knowledge (ZK). The primary tool we use to ensure security in this protocol builds on zero-knowledge (ZK) protocols [38]. These provide proof of a statement without leaking any information. E.g. the Schnorr zero-knowledge protocol shows that the prover knows the value of the private key without leaking any information. We use the ZK-protocols to bind the secret key to the public key using the Schnorr protocol [63] and show the vote cast has the correct format (either 0 or 1) using the Cramer-Damgård-Schoenmakers (CDS) protocol [28]. For these protocols, we want to show the security statement: the adversary learns nothing except that the statement is true.

2.3.3 Σ-protocols. A Σ-protocol [27] is a three-message (commit, response, verify) protocol between two parties, a prover and a verifier, with the prover going first. The protocol guarantees that the prover can make the verifier accept if the prover knows a witness to some relation for a common input. An example relation could be proving knowledge of the secret key for some public key. Here, the common input would be the public key, while the witness to the relation would be the secret key. Thus, if the prover knows the secret key, they can convince the verifier of that fact.

We prove and use the following properties of the Σ-protocols in this paper.

- Completeness: for an honest prover and honest verifier, the protocol always accepts if the private input to the prover is a witness of the relation.
- Special Soundness: Given a pair of accepting conversations with the same inputs, but with a different challenge response, we can compute a witness for the relation.

For Σ-protocols with zero-knowledge, the verifier wants to ensure that the prover knows the witness, while the prover does not want to disclose any additional information about the witness. To model this behavior, one must be able to construct a simulator that, given the common input to the prover and verifier, produces accepting conversations with the same distribution as real conversations. *Honest verifier zero-knowledge* (HVZK) is the property that making a simulated conversation with an honest verifier is computationally easy. Here, simulating the conversation means producing a transcript of a conversation without actually interacting with a prover. Furthermore, this transcript should have the same distribution as one with a real prover. *Special honest verifier zero-knowledge* (SHVZK) is a simulator that uses a specific challenge response for HVZK. For the Σ-protocols in this paper, a proof of SHVZK can be turned into a proof of zero-knowledge [30, Sec. 8 and Ex. 3].

A *witness extractor* is a procedure that can produce a witness for an input given a number of honest provers for other input and witness pairs. A protocol is *witness hiding* if such an extractor succeeds with the same probability as the verifier. This is usually shown using witness indistinguishability, which states that given two honest provers with the same input but different witnesses, one cannot distinguish the conversations of one from the other.

The Schnorr and CDS protocols are instances of Σ-protocols.

2.3.4 Fiat-Shamir heuristic / Non-interactive zero-knowledge (NIZK) proofs. We use the Fiat-Shamir heuristic [36] to replace the challenge message in the Σ-protocol with a hash [30, Sec. 10]. This allows us to send the entire zero-knowledge proof without having a round of communication in between. This is also very useful in protocols with many participants and not just two parties, as it is not obvious who should send the challenge.

2.4 Security Games, SSP, SSProve

When using all phases of the Hax translation, one ends up with an AST of a functional program, which can then be printed to F* or Rocq. When only using some phases, one ends up with a simple imperative program, which one can then print to an embedded DSL in Rocq. In particular, to the embedded imperative language in SSProve, a library in Rocq, which allows one to reason about security games and proofs in the state separating (SSP) style.

The focus on having well-defined and modular cryptographic constructions has given rise to the constructive cryptography [55] paradigm. Here, security is a property that can be constructed from smaller components with their security properties. That is, applying the ideas of constructive mathematics to the cryptographic reduction style of proofs.

2.4.1 Advantages and security Games. To measure security, we consider the advantage an adversary has in distinguishing between two different packages. That is, how much better than random the adversary is at guessing which of the two packages we are using. Here, a package is a set of functions; thus, the adversary needs to figure out what code is being used to produce a given result. Thus, the two packages need to have the same interface.

We represent the advantage of a game between packages X and Y as $\mathcal{A}(X, Y)$. We write $X \approx_\varepsilon Y$ to represent $\mathcal{A}(X, Y) \leq \varepsilon$ and call two such packages indistinguishable if $\mathcal{A}(X, Y) = 0$, e.g. $X \approx_0 Y$. A *game hop* replaces X with Y. For a sequence of such hops, the advantage between the first and the last is bounded by the sum of advantages in each step.

2.4.2 State separating proofs (SSP). The state separating proofs framework [20] defines a calculus for packages consisting of a collection of import/export interfaces and functions. This calculus allows one to combine packages in series (inlining function definitions) and in parallel to make larger packages, provided that their interfaces match. SSP allows one to isolate the shared state of packages and reason about game hops by isolating the changes and, thus, work with smaller packages. This facilitates modularity and scalability of security proofs.

A simple example of a security game in SSP is the one-time pad (OTP) encryption. This is presented as two programs, corresponding to the cryptographer's real world (reference implementation) and ideal world (specification) paradigm.

```
$\mathtt{OTP_{enc}}^0(x)$            $\mathtt{OTP_{enc}}^1(\_)$
$y \leftarrow\$ \{0,1\}^n$          $z \leftarrow\$ \{0,1\}^n$
ret $x \oplus y$              ret $z$
```

The real package $\mathtt{OTP}^0_{enc}$ encrypts its input by sampling a uniformly random bit string and xor-ing it with the input message. The ideal behavior is a uniformly random bit string. Hence, the ideal package $\mathtt{OTP}^1_{enc}$ samples such a string and returns it. By proving (perfectly) indistinguishability of the two packages using code equivalence, we obtain

$$\mathtt{OTP}^0_{enc} \approx_0 \mathtt{OTP}^1_{enc}.$$

This allows one to replace any use of the real OTP encryption with the idealized version.

2.4.3 SSProve. SSProve [45] is a foundational library in Rocq for performing cryptographic proofs in the SSP style. It has an imperative language with state and probability embedded using a monadic presentation. From this encoding, a relational program logic based on the Dijkstra monad framework is derived. SSProve formalizes the SSP framework and facilitates modularity and reuse for security proofs built on top of a probabilistic language. SSProve works in the computational model, which is more precise than the symbolic ('Dolev-Yao') model. As SSProve is embedded in Rocq, one can use tools and libraries developed for Rocq. Especially having a large mathematical library, like Mathematical Components (MathComp) [54], eases the formalization effort.

To reason about implementations of cryptographic protocols, one needs a tool for translating the implementation into SSProve. Hax [14] provides such a translation into SSProve for the Rust_{Hax} subset of Rust; see Subsection 2.2.1. Thus, we can implement cryptographic primitives using more advanced language features of Rust, such as traits. Starting from Rust enables us to integrate with other tools and to use Hax for translating to other backends.

Formulating this process for the OTP example, we can implement OTP in Rust using the xor function. Hax translates this into SSProve as $\mathtt{OTP}_{enc,impl}$. We then prove the equivalence of the implementation to the specification $\mathtt{OTP}^0_{enc}$ as an SSP security game.

$\mathtt{OTP}_{\mathtt{enc,impl}}(x)$	$\mathtt{OTP}_{\mathtt{enc}}{}^0(x)$
$y \leftarrow\$ \{0,1\}^n$	$y \leftarrow\$ \{0,1\}^n$
$\mathtt{ret}\ x \oplus_{Rust} y$	$\mathtt{ret}\ x \oplus y$

Thus, we use the SSP proof style and its modularity, together with the stateful and probabilistic imperative language of SSProve, for software verification. The proof of $\mathtt{OTP}^0_{enc} \approx_0 \mathtt{OTP}^1_{enc}$ in SSProve uses of the probabilistic reasoning of SSProve. More advanced proofs can make use of the relation between the program logic and its semantics in Rocq, which builds on the MathComp library.

3 CRYPTOCONCERT

We introduce a framework for verifying both the correctness and security of smart contracts; see Figure 1. We implement the smart contract in Rust_{Hax} and use the Hax framework for translating

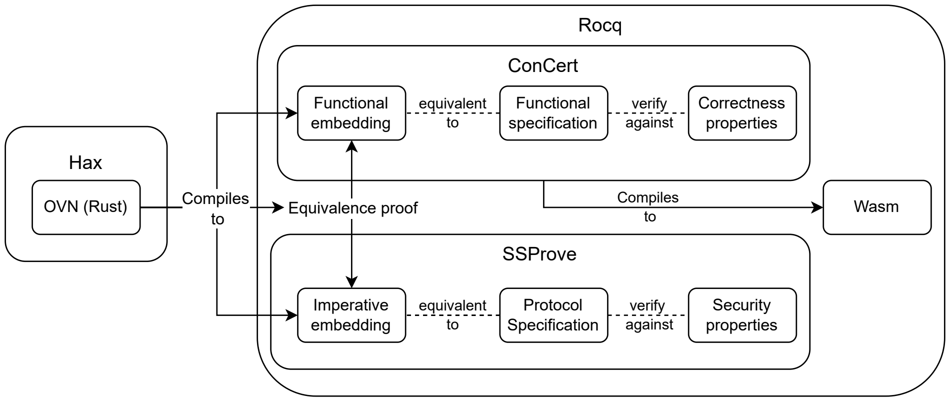


Fig. 1. Verification pipeline

the implementation into Rocq, specifically SSProve and ConCert. We verify the security (using SSProve) and correctness (using ConCert) of the translation and extract it to Wasm, which can be run by most of the blockchains using Rust as a smart contract language. Thus, defining the translation as the specification of the smart contract, we have a fully verified process.

3.1 SSProve for Program Verification

The SSProve framework has mostly been used for security proofs. In this work, we have used SSProve as a framework for proving code equivalence between a specification and a protocol. To aid this process, we have shown properties of any code translated from Hax and made Ltac macros for extracting code and facilitating equivalence proofs by following the structure of the code. That is, we can show two function calls are equal by showing each argument is equal, and then by code equality of the function, the results will be equal.

3.2 Extending Hax to Support Smart Contracts

Hax can be used as a frontend for protocol verifiers in the symbolic model of cryptography, such as ProVerif [17] which is based on the applied π-calculus. Since ProVerif abstracts from details in a Rust implementation, the Hax ProVerif-backend [16] uses extra annotations, to specify how certain parts of the code should be modelled more abstractly.

We use a similar approach to extend Hax to support smart contracts. We have a concrete model of the blockchain in Rust, which we abstract in ConCert. Morally, this is the inverse of the final extraction step when ConCert's smart contracts are instantiated with a specific (Rust) blockchain. Concretely, our OVN Rust contract and the extracted Rust code from the Hax/ConCert embedding can both run on Concordium. We do not prove the correspondence between the two formally.

More concretely, in Rust, blockchain traits are defined using attributes to tag

- the type of the smart contract state,
- which function initializes the contract,
- and what functions can be called in the contract.

Furthermore, the attributes contain the name of the smart contract that they are part of and other meta-information about additional parameters that are parsed to the different functions. In Hax, we gather all this information and construct a ConCert model for each smart contract together with a message type and function for delegating calls to the contract.

4 THE OPEN VOTE NETWORK PROTOCOL

The Open Vote Network (OVN) [43] is a decentralized voting protocol for small-scale elections. It uses a blockchain to eliminate the need for a trusted third-party and uses zero-knowledge proofs to detect malicious voters deviating from the protocol. A (n unverified) Solidity implementation of this protocol was deployed on Ethereum [56]. The OVN protocol allows a small group (of N voters) to vote anonymously. The protocol is defined by four rounds of communication (1) Register (2) Commit to vote (3) Cast vote (4) Tally result.

Before the first round, the election admin publishes a list on the blockchain of the N users allowed to participate in the election. The smart contract ensures that only these users can vote. The protocol is parametrized by a cyclic group G, with generator g, for which the discrete logarithm problem is hard. The discrete logarithm problem is: given a group element h, compute n such that $h = g^n$. Examples of such groups come from modular arithmetic or elliptic curves.

Round 1, Register. Each participant P_i selects a random secret key $x_i \in \mathbb{Z}_{|G|}$, computes their public key $y_i = g^{x_i}$, and a zero-knowledge proof that y_i and x_i are related using the Schnorr protocol. This proof is used to ensure that each participant knows their secret key. These zk-proofs and public keys are published on the blockchain.

Round 2, Commit to vote. Each participant validates all zero-knowledge proofs from the previous round, computes their reconstruction key $g^{y_i} = \prod_{j=1}^{i-1} g^{x_j} / \prod_{j=i+1}^{N} g^{x_j}$ and ballot $vote_i = (g^{y_i})^{x_i} g^{v_i}$ for their vote $v_i \in \{0, 1\}$. A commitment to the ballot is published on the blockchain.

The commitment round was not in the original OVN protocol and was added later to prevent an attack where the last voter could compute the tally before voting. Without this fix, there would be an adaptive issue: knowledge of the outcome of the tally could influence the voter's cast vote.

Round 3, Cast vote. Each P_i computes a zero-knowledge proof showing that their vote is either 0 or 1 using the CDS protocol. Both the proof and ballot $vote_i$ are published on the blockchain.

Round 4, Tally result. Everyone checks the validity of the commitments and the zero knowledge proofs from the previous round. Finally, all the votes are tallied, and the final result is computed. The reconstruction keys computed in round 2 are constructed such that the product of all ballots $\prod_{i=1}^{N} vote_i = g^{\sum_{i=1}^{N} v_i}$ is the product of all the votes without revealing any single vote. To recover the tally, the exponent is brute-forced. This requires creating a table of all multiples of the generator. The size of the table is limited by the number of participants. The smart contract is self-tallying, meaning that it computes the tally on-chain. However, the tally can also be computed off-chain by anyone since every vote is publicly available on the smart contract, and computing the tally requires no private information.

The OVN protocol has the following properties:

- Maximum Ballot Secrecy (see Subsection 5.1): Any party's secret inputs (secret key and vote) are indistinguishable from random noise and hence this information stays private.
- Universal Verifiablity (see Subsection 4.1): Anyone can audit the protocol, e.g. validate the correctness of the execution of the protocol.
- Self-tallying (see Section 6): Any valid run of the protocol will result in the correct tally.

None of the properties have previously been formally verified, although a paper proof of maximum ballot secrecy exists [43].

4.1 Universal Verifiablity

The property of universal verifiability states that anyone can audit the outcome of the voting process. This is a property of the construction of the protocol. A separate verifier can be constructed to check these properties [41]. Here, we ensure the underlying properties hold but do not construct a verifier. Universal verifiability can be split into the following parts [1].

- *Cast as intended*: The vote on the ballot corresponds to the voter's intended choice.
- *Recorded as cast*: The voter is sure that the ballot recorded is the one they cast.
- *Counted as recorded*: The tally is correctly counting the ballots on the public board.
- *Eligible voter verification*: only eligible voters can cast a ballot.

Where *cast as intended* and *recorded as cast* are individual properties, meaning that only the participants of the vote can validate these properties. The *counted as recorded* property is a universal verifiable property, meaning anyone can check that the tally is correct. The *eligible voter verification* property is handled in the setup phase of the OVN protocol and is thus not further discussed here.

4.1.1 Cast as intended and recorded as cast. All the public keys can be verified to be elements of the chosen secure group, and each participant computes their own vote and publishes it. Thus, any participant of the protocol can check that their vote is *cast as intended*. Likewise, participants can check that the information they submit corresponds to what is recorded on the blockchain, as all published information is public. This ensures *recorded as cast*.

4.1.2 Counted as recorded. The Schnorr proof has built-in validation, which is proven secure and correct in the Σ-protocol framework of SSProve [45, sec.7]. This ensures each party must know the secret used to generate their public key. The commitments can be validated against the vote, so everyone can check that the votes have not been changed after the commitment. Finally, we know that all the votes must be either 0 or 1 votes from the CDS proofs, which is again proven in SSProve. By the correctness of the protocol, we can check that the final tally is correct. Thus, all parts of the public transcript can be validated, even by people not participating in the protocol.

4.1.3 Universal verifiability. The reduction to the validation of the Σ-protocols (Schnorr and CDS) is part of their specification. The commitment scheme ensures it is impossible to change a value one has committed to. Meaning, we have shown that universal verifiability reduces to standard cryptographic assumptions. That is, the security of each part of universal verifiability has been reduced to the security of the cryptographic primitives and the discrete logarithm security assumption for the group. This style of reducing the universal verifiability to the properties of the primitives has been done more formally in [40] for the related Election Guard (EG) protocol; see Section 8.1 for a more thorough comparison. In this work, we focus on proving the security and correctness of the implementation of the voting protocol, not on properties of the external verifiers.

4.2 Rust Implementation of OVN

The OVN smart contract follows the structure laid out at the start of Section 4.

As a first step for verification, we show that the embedding of Rust implementation is functionally equivalent to a cleaner specification. We do this both for the functional implementation in ConCert and for the imperative implementation in SSProve.

Examples of this are the field and group traits. In Rust, we have a trait for the group operations. However, in Rocq, we want the group laws, so we also add these and package them up using MathComp's Hierarchy Builder.

Similarly, for fields, we provide a field in MathComp, but with a more proof friendly representation. This means that we will need to translate over this equivalence ~60 times in our proof. Instead, one could transport the properties over this equivalence, as we know they should hold for any equivalent definition. Trocq [22] aims to facilitate this, but it is still under development.

On the SSProve side, to prove this equivalence, we use the SSP style in a novel way. We use SSP reductions to show program equivalence, thus extending the SSP technique, which was designed only for security proofs.

The original OVN protocol [43] had a security issue. We follow the improved protocol [56]. This improved protocol adds an extra round in which all parties commit to their vote and publish this commitment before publishing their vote. This ensures that one chooses one's vote before knowing the other votes.

5 OVN SECURITY PROOFS

We will give a detailed account of our security proof for the OVN protocol using SSProve (as introduced in Subsection 2.4).

5.1 Maximum Ballot Secrecy

The security property for a voting protocol like OVN, as described in Section 4, is maximum ballot secrecy, which states: Any party's secret inputs (secret key and vote) are indistinguishable from random noise, and hence this information stays private. We show this property by a sequence of security games. Each game idealizes part of the protocol until it is fully idealized and thus no longer depends on the secret inputs. Given that each game hop is indistinguishable from random, the full transformation is also indistinguishable from random. Thus, the secret inputs are secure, as we can change them without it being detectable by the adversary.

The security proof uses the following games, which we will introduce below.

- Discrete logarithm
- the Schnorr protocol
- commitment scheme / hashing
- CDS-proof

OVN depends on a group for which the discrete logarithm problem is (computationally) hard.

The Schnorr proof uses the Σ-protocol framework of SSProve. Thus, we write the mathematical specification for the Schnorr proof (commit, response, verify, simulate, extractor), prove security (special honest verifier zero-knowledge, simulation sound extractability), and show it agrees with our translated Rust implementation. Similarly, for the CDS proof.

The commitment scheme is done by hashing. Thus, we assume the existence of a hashing function that takes group elements and produces a field element. Such a hash function is also needed for the Fiat-Shamir transformation of the CDS and Schnorr ZK-proofs.

Once we have all the security statements, we can combine them into a proof of maximum ballot secrecy. In Section 5.2 we prove that this property extends to the *implementation* of OVN.

Below is the pseudocode for the specification ($\texttt{OVN}^0$) and the idealization ($\texttt{OVN}^1$) of the OVN protocol.

```
$\texttt{OVN}^0(i, v_i)$
/ Register vote
$x_i \leftarrow\$ \mathbb{Z}_q$
$Schnorr_{zkp_i} \leftarrow \texttt{Schnorr}^0(g^{x_i}, x_i)$
$\texttt{Publish} : Schnorr_{zkp_i}, g^{x_i}$
/ Commit to vote
$\texttt{validate}(Schnorr_{zkp_j}) \quad \forall j \in (1, n)$
$g^{y_i} \leftarrow \dfrac{\prod_{j=1}^{i-1} g^{x_j}}{\prod_{j=i+1}^{n} g^{x_j}}$
$vote_i := (g^{y_i})^{x_i} \cdot g^{v_i}$
$commit_i := \mathcal{H}(vote_i) : \mathbb{Z}_q$
$\texttt{Publish} : commit_i$
/ Cast vote
$CDS_i = \texttt{CDS}(g^{y_i}, x_i, v_i)$
$\texttt{Publish} : vote_i, CDS_i$
/ Tally
$\texttt{CDSvalidate}(g^{y_j}, CDS_j) \quad \forall j \in (1, n)$
$\texttt{CheckCommit}(commit_j, vote_j) \quad \forall j \in (1, n)$
$g^{tally} = \prod_{j=1}^{n} vote_j \overset{bruteforce}{\Longrightarrow} tally$
```

```
$\texttt{OVN}^1(i)$
/ Register vote
$h, g^{y_i}, k \leftarrow \texttt{DDH}_{gen}$
$Schnorr_{zkp_i} \leftarrow \texttt{Schnorr}^1(h, \_)$
$\texttt{Publish} : Schnorr_{zkp_i}, h$
/ Commit to vote
$\texttt{validate}(Schnorr_{zkp_j}) \quad \forall j \in (1, n)$
$vote_i \leftarrow k \cdot g^{v_i}$
$commit_i \leftarrow\$ \mathbb{Z}_q$
$\texttt{Publish} : commit_i$
/ Cast vote
$CDS_i = \texttt{CDS}^1(g^{y_i}, \_, \_)$
$\texttt{Publish} : vote_i, CDS_i$
/ Tally
$\texttt{CDSvalidate}(g^{y_j}, CDS_j) \quad \forall j \in (1, n)$
$\texttt{CheckCommit}(commit_j, vote_j) \quad \forall j \in (1, n)$
$g^{tally} = \prod_{j=1}^{n} vote_j \overset{bruteforce}{\Longrightarrow} tally$
```

We prove the security of the specification by idealizing the protocol, thus obtaining a version that is independent of the secret inputs (vote and secret key). No adversary will be able to distinguish which input the user chose, ensuring maximum ballot secrecy.

5.1.1 *Security assumptions.*

We built the proof on some security assumptions. We state the assumptions as a security game being bound to a negligible advantage (see Subsection 2.4.1). The assumptions are as follows:

$$\begin{aligned}\texttt{dl}^0 &\approx_\varepsilon \texttt{dl}^1,\\ \texttt{ddh}^0 &\approx_\zeta \texttt{ddh}^1,\\ \texttt{commit}^0 &\approx_\psi \texttt{commit}^1,\\ \texttt{uniform}^0_{y_i} &\approx_\nu \texttt{uniform}^1_{y_i}.\end{aligned}$$

The value of y_i must differ from 0, otherwise the CDS proof does not work. We can ensure $y_i \neq 0$ by adding a check and aborting if $g^{y_i} = 1$ for any i. This can be checked by anyone, as the computation only depends on global information. This is done after the first round, before anyone publishes $g^{y_i x_i} \cdot g^{v_i}$, which would leak the vote.

Aside, one might think this enables a denial-of-service (DOS) attack, where we continue finding values such that $y_i = 0$ for some i. This is not the case, as this requires

- either guessing the correct random value, which is one over the order of the group, or
- to efficiently compute
$$g^{sk} = \frac{\prod_{j=0,j\neq k}^{i-1} g^{x_j}}{\prod_{j=i+1,j\neq k}^{n} g^{x_j}}.$$
However, the Schnorr proof forces one to know the secret key sk. Thus, this is as hard as discrete logarithm (DL).

By these arguments, there is only a negligible chance we will fail the check. Thus, a protocol without the check is still valid; however, the security bound is slightly weaker.

We generally need y_i to be uniformly random, even when people get to choose x_j for g^{x_j}, thus we introduce the uniformity as an assumption. But, we expect it to reduce to properties of the Schnorr protocol and DL.

5.1.2 *Discrete logarithm.* The first step in the protocol is to generate a private and public key and publish the public key. We build on the assumption that the discrete logarithm problem is hard

DL_{gen}	$\mathsf{DL}^0_{guess}(h)$	$\mathsf{DL}^1_{guess}(\_)$
$x_i \leftarrow\$ \mathbb{Z}_q$		
$\mathtt{store}_{\mathsf{DL}}(x_i)$	$x_i \leftarrow \mathtt{load}_{\mathsf{DL}}$	
$\mathtt{ret}\ g^{x_i}$	$\mathtt{ret}\ (g^{x_i} \overset{?}{=} h)$	$\mathtt{ret}\ false$

to ensure that publishing the public key is safe.

5.1.3 *The Schnorr proof.* The next step is the Schnorr ZK proof. Since this is a Σ-protocol, we can replace the code with a call to a simulator (see Subsection 2.3.3)

$\mathsf{Schnorr}^0$	$\mathsf{Schnorr}^1$
/ Register vote	/ Register vote
...	...
$x_i \leftarrow\$ \mathbb{Z}_q$	
$Schnorr_{zkp_i} \leftarrow \mathtt{Schnorr}_{committer}(h, x_i)$	$Schnorr_{zkp_i} \leftarrow \mathtt{Schnorr}_{simulator}(h)$
	$x_i \leftarrow\$ \mathbb{Z}_q$
...	...

by the zero-knowledge property; doing this ensures we do not depend on the secret input. Thus allowing us to push the sampling of the secret input later in the protocol.

5.1.4 Decisional Diffie-Hellman. The security of publishing the vote is based on the decisional Diffie-Hellman (DDH) assumption.

DDH^0	DDH^1
$x_i \leftarrow\$ \mathbb{Z}_q$	$x_i \leftarrow\$ \mathbb{Z}_q$
$\mathtt{store}_{\mathsf{DDH},0}(x_i)$	$\mathtt{store}_{\mathsf{DDH},0}(x_i)$
$y_i \leftarrow\$ \mathbb{Z}_q$	$y_i \leftarrow\$ \mathbb{Z}_q$
$\mathtt{store}_{\mathsf{DDH},1}(y_i)$	$\mathtt{store}_{\mathsf{DDH},1}(y_i)$
	$z_i \leftarrow\$ \mathbb{Z}_q$
$\mathtt{ret}\ (g^{x_i}, g^{y_i}, g^{x_i \cdot y_i})$	$\mathtt{ret}\ (g^{x_i}, g^{y_i}, g^{z_i})$

Applying the DDH assumption requires that the values g^{x_i}, g^{y_i} are randomly distributed. We therefore assume that at least one value used (i.e. g^{x_j}) to compute g^{y_i} is randomly distributed. This, combined with the Schnorr proof, ensures that g^{y_i} is randomly distributed, or equivalently that computing y_i is hard.

$\mathtt{uniform}^0_{y_i}$	$\mathtt{uniform}^1_{y_i}$
$g^{x_k} \leftarrow\$ \mathcal{G} \quad \exists k, k \neq i$	
$g^{y_i} \leftarrow \dfrac{\prod_{j=1}^{i-1} g^{x_j}}{\prod_{j=i+1}^{n} g^{x_j}}$	$y_i \leftarrow\$ \mathbb{Z}_q$
$\mathtt{ret}\ g^{y_i}$	$\mathtt{ret}\ g^{y_i}$

These two assumption combine to allow us to swap the publish vote for a random value.

5.1.5 Commitment scheme. We build on the assumption that the hash function is secure; thus, we can idealize the commitment scheme into a random oracle.

$\mathtt{commit}^0$	$\mathtt{commit}^1$
/ Commit to vote	/ Commit to vote
...	...
$x_i \leftarrow\$ \mathbb{Z}_q$	
$commit_i := \mathcal{H}(vote_i) : \mathbb{Z}_q$	$commit_i \leftarrow\$ \mathbb{Z}_q$
	$x_i \leftarrow\$ \mathbb{Z}_q$
...	...

This assumption is also used in the Fiat-Shamir transformation (see Subsection 2.3.4) for the Schnorr and CDS (see Subsection 2.3.2) ZK proofs.

5.1.6 CDS-proof. As for the Schnorr proof, we can idealize the CDS protocol.

CDS^0	CDS^1
/ Cast vote	/ Cast vote
...	...
$x_i \leftarrow\$ \mathbb{Z}_q$	
$CDS_i = \mathsf{CDS}_{committer}(g^{y_i}, x_i, v_i)$	$CDS_i = \mathsf{CDS}_{simulator}(g^{y_i}, \_, \_)$
	$x_i \leftarrow\$ \mathbb{Z}_q$
...	...

5.1.7 Security bound. We use the discrete logarithm (DL) assumption for publishing the public key without leaking the private key, this adds ε to the security bound. We use decisional Diffie-Hellman (DDH) for publishing our vote without leaking $y_i \cdot x_i + v_i$, adding ζ to the security bound. The argument for not leaking x_i and $y_i \cdot x_i + v_i$ is also based on those being chosen from a uniformly random distribution; however, this is only true if y_i is uniformly random, as v_i is not chosen uniformly at random. This assumption reduces to the DL assumption. This adds ν to the security bound, although we expect $\nu \approx \varepsilon$. The transition for the Schnorr and CDS proofs is done without adding anything extra to the security bound. Finally, we go from hashing to random sampling for our commitment scheme, which incurs an additional ψ; thus, our final security bound becomes

$$\mathcal{A}(\mathtt{OVN}^0(b), \mathtt{OVN}^1) \leq \varepsilon + \zeta + \nu + \psi.$$

This is the bound for idealizing the protocol given some specific vote. To show the equivalence between $\mathtt{OVN}^1(\mathtt{true})$ and $\mathtt{OVN}^1(\mathtt{false})$, we show there is a bijection between the distributions. Specifically by synchronizing the sampling with the bijective function $k_{\mathtt{true}} \equiv k_{\mathtt{false}} \cdot g$, we get the equivalence for the vote $k_{\mathtt{true}} \cdot g^{\mathtt{true}} \equiv (k_{\mathtt{false}} \cdot g) \cdot g^{\mathtt{false}}$. The rest of the transcript for $\mathtt{OVN}^1$ does not depend on any private input, thus, are equivalent. Now by transitivity, we can then show that an adversary can only distinguish the vote with negligible probability, which is the maximum ballot secrecy property

$$\mathcal{A}(\mathtt{OVN}^0(\mathtt{false}), \mathtt{OVN}^0(\mathtt{true})) \leq 2 \cdot (\varepsilon + \zeta + \nu + \psi).$$

Putting all these steps together, we have now proved maximum ballot secrecy for the OVN protocol.

5.2 Equivalence of $\mathtt{OVN}_{impl}$ and $\mathtt{OVN}^0$

As explained in Subsection 4.2, to prove the security of the implementation ($\mathtt{OVN}_{impl}$), we need to show that it is (probabilistically) equivalent to the protocol ($\mathtt{OVN}^0$).

Rust$_{Hax}$ is both deterministic and functional. We have built a library for reasoning about such code in SSProve. An important difference between Rust and SSProve is the sampling of randomness. In Rust$_{Hax}$, we pass all the randomness as an argument (eager sampling). Whereas, in SSProve we have a sampling primitive, so we can sample lazily. We prove the equivalence of eager and lazy code semi-automatically by pushing all sampling to the start of a function and showing that the sampling functions are equivalent (e.g. sampling a single value twice is the same as sampling a pair).

SSProve facilitates proving code equivalence modularly by replacing functions independently. In this way, we use the structure of imperative translation to guide the equivalence of the functional code. Moreover, we can use the pure translation to prove equivalence by computation, which is useful for simple code fragments.

6 SELF-TALLYING IN CONCERT

The self-tallying property of OVN states that once all ballots have been cast, anyone (both voters and outside observers of the protocol) can compute the correct tally without external help. Self-tallying of OVN follows from the use of smart contracts and blockchains to make ballots publicly available and the vanishing property of the ballots. The vanishing property comes from the construction of the reconstruction key such that they cancel out and leave just the votes. We ensure the availability of the ballots by storing them in the smart contract state, which is publicly available on the blockchain and can only be mutated by the smart contracts. Thus, the OVN implementation is self-tallying if it can compute the correct tally using only its public state.

We prove this using ConCert. The proofs are complete and available in the artifact (barring administrative lemmas, which we will complete before the rebuttal period).

6.1 Proofs

The main theorem that we want to prove for self-tallying is that when the tally field is set in the contract state, then it is equal to the sum of all votes. We formulate this as an invariant of the smart contract state over all valid blockchain traces. A valid trace is a chain of blocks starting from the empty block.

THEOREM 6.1. *Consider n parties with votes $v_1, \ldots, v_n$, and assume that all parties behave honestly. For any reachable block after OVN is deployed, let s be the contract state in the block and $s.t$ be the tally field, then if $s.t$ is set then $t = \sum_{i=1}^{n} v_i$.*

By universal verifiability and the zero-knowledge proofs that are checked by the smart contract, we ensure that deviation from the protocol is detected. We can thus assume that all parties behave honestly, that is, their inputs to the smart contract are computed according to the protocol specification.

To prove the theorem, we first show that the smart contract enforces the round structure of the protocol. This is important since some rounds require that all parties have finished the previous round, either for security or correctness reasons. Thus, the code should enforce that a party can only publish a round message while that round is active and that the next round does not start until all parties have published their messages.

It is imperative that we check this properties, as they are similar to the notorious state machine attacks [12], where an adversary can change the state of the protocol by publishing messages out of order or skipping rounds. ConCert's framework for reasoning about trace properties has some similarity with the reasoning used by automatic protocol verifiers like Tamarin [8] and ProVerif [17], where the trace properties are used to ensure that the protocol is executed correctly.

Aside, the requirement of active participation of all parties creates a denial of service problem with OVN. However, this is not specific to this implementation and is a general limitation of many boardroom voting protocols. Some implementations use timers to ensure the round structure, and if there aren't enough votes by the deadline, the election is declared void, and a new one is typically started without the parties that did not vote.

Next, we show that each round correctly stores the messages published by the parties and that these messages are immutable. Together with the previous property and the fact that all parties are honest, implies that in the last round, all data is available and is computed as specified by the protocol. Similarly, these properties are also formulated as trace properties and proven by induction over the trace.

By functional correctness of the tally and compute_reconstruction_key functions we see that the tally field t equals $\texttt{bruteforce}(\prod_i g^{x_i y_i} g^{v_i})$.

Using that $\sum_i x_i y_i = 0$ and the correctness of the bruteforce function we have

$$\begin{aligned} t &= \texttt{bruteforce}(\prod_i g^{x_i y_i} g^{v_i}) \\ &= \texttt{bruteforce}((\prod_i g^{x_i y_i}) \prod_i g^{v_i}) \\ &= \texttt{bruteforce}(g^{\sum_i x_i y_i} g^{\sum_i v_i}) \\ &= \texttt{bruteforce}(g^0 g^{\sum_i v_i}) \\ &= \texttt{bruteforce}(g^{\sum_i v_i}) \\ &= \sum_i v_i \end{aligned}$$

Additionally, we also prove other correctness guarantees and desirable properties about the OVN implementation.

6.2 Implementation Equivalence

The Hax embedding into Rocq is that it embeds both into Rocq and SSProve and gives an automatic equivalence. Hax embeds a Rust program as a pair (p, f, e), where e is a proof that the imperative program p and the functional program f are equivalent. This can be seen as a refinement of the functional program f to the imperative program p, or as a realizability relation between the two. For proving security in SSProve, this representation is useful. However, for correctness, we project out to the functional representation as this makes the embedding easier to work with. This step could be automated in the Hax translation.

The functional part is defined using the MathComp library, which is designed to be proof-friendly, but not well suited for computation. Thus, we first translate this to equivalent Rocq code that uses the more standard Rocq libraries. CoqEAL [23] is an experimental library to automate such refinements. It is currently being redeveloped using Trocq [22]. It is not yet fully production ready, so we needed to carry out these transformations manually. Moreover, this transformation was also needed to resolve universe conflicts between MathComp and Stdpp. Finally, we use this to clean up a non-optimal Hax-translation of Rust structs. One would like to translate them into records in Rocq. However, currently this is not possible due to a lack of support for universe polymorphism in Hierarchy Builder [24].

7 EVALUATION

We evaluated our Hax implementation of OVN on several group implementations and election sizes ranging up to 500 voters. The code was tested locally. This is a good predictor of on-chain computation (we will confirm this at the rebuttal stage). The OVN implementation abstracts over the group and hash implementations. We evaluated the implementation using groups $\mathbb{Z}_p^*$ with 64 and 256-bit primes and using a secp256k1 curve reference implementation. In practice, the code should be instantiated with verified and more efficient implementations, for example, using libcrux's [49] x25519 elliptic curve implementation.

Our experiment shows that the implementation can handle boardroom-sized voting, for which the protocol was designed. Moreover, elections of several hundred voters seem feasible in a reasonable time. This would allow much bigger elections to be run, since in large elections, the outcome of the election is usually published per polling station. For more on such elections see Section 8.1. Before we discuss this, we survey the costs of the individual phases. To evaluate the feasibility, one should evaluate the cost of running the contract. On blockchains, one pays for computation time and data storage.

Our evaluation shows that the election initialization time is negligible and most of the cost would come from the size of the OVN contract. The time spent per voter during registration is constant, while the per voter time spent in the commit and vote phases scales linearly with the number of participants. The costs of the commit and vote phases are due to the costs of computing the reconstruction key, which takes up the majority of the computations. It is possible to compute the reconstruction keys more efficiently by moving this computation to a separate phase, which eliminates some duplicate computation of shared subterms. The tallying round is the most expensive single smart contract call during the protocol and scales linearly in the amount of voters.

This is not a problem for boardroom voting. To scale to bigger elections, one could modify the smart contract and split the tally call into multiple calls, or compute the tally off-chain and only verify the tally on-chain.

Another possibility would be to use a closed blockchain; see Section 8.1.

7.1 Compilation

We use CertiRocq-Wasm's verified compilation to Wasm to produce a verified Wasm implementation as an alternative to the Rust/Hax implementation. We extract the functional implementation of OVN in ConCert, rather than the embedded implementation, because MetaRocq currently does not support sort polymorphism. Finally, we evaluate the extracted Wasm implementation using $\mathbb{Z}_p^*$ with a 63-bit prime implemented using Rocq's primitive integers and a 256-bit prime implemented using Rocq's binary integer representation. We find that the performance of the Rust and Wasm implementations is comparable with a factor ~3 performance cost. This is important evidence that CryptoConCert is practical.

More improvements are within reach: Some performance costs are due to the inefficient group and hash implementations; see Section 8.5. Moreover, the Rocq implementation uses inefficient data structures, such as lists instead of arrays. Rocq's native arrays are currently not supported by CertiRocq-Wasm. In an unverified extraction, such issues could be addressed by remappings. However, this is not supported by CertiRocq-Wasm.

8 RELATED AND FUTURE WORK

8.1 Verified Voting Software

There are still many obstacles for online voting for national elections [9]. However, online voting protocols are in use today for boardroom voting, school and university elections, minor elections, etc. . . Here, our methodology already provides an improvement to the state of the art. Online voting also has some benefits over voting by mail [67]. The use of permissionless blockchains provides transparency, but comes with a possible attack vector where the majority of the computational power, or stake, may be biased toward a certain outcome. However, the ConCert framework is flexible enough that it could be run on a permissioned ("closed") blockchain such as Hyperledger, which also supports Wasm.

ElectionGuard [9] is another protocol that is run on a blockchain[3]. It uses similar building blocks (homomorphic tallying), and our methods should apply, as both protocols are based on [29]. The ElectionGuard Rust implementation [9] already includes verified cryptographic primitives from the HACL⋆ library [69]. Compared to ElectionGuard, OVN is less dependent on an election committee, as it uses more of the blockchain. To do so, it bruteforces the discrete log for small numbers on-chain, this makes it less efficient.

In comparison to Subsection 4.1, the universal verifiability of the ElectionGuard *protocol* has been proved in Rocq [40]. They do this by verifying a *verifier* that checks the traces of the zk-proofs, but does not prove properties of the full protocol. Their work mentions clear limitations [40, 3.2]: 1. They do not verify the privacy of the vote. We do. 2. To avoid probabilistic reasoning, they use a non-standard treatment of Σ-protocols. We followed the standard treatment, including Fiat-Shamir (2.3.4), in the computational model. 3. They use the unverified extraction from Rocq to OCaml (with unverified remappings). In comparison, we use *verified* compilation to Wasm.

The strategy of 'Verifying the verifiers' [40] was first applied to the Helios protocol [41]. The privacy of the Helios protocol and many variants, was proved in the computational model in Easy-Crypt [25]. They emphasize that security of the implementation is an important open problem [25, V.C].

Both [41] and [42] verify mix-nets, which are important components of other voting protocols, but are not used in OVN or ElectionGuard.

Belenios is a voting protocol inspired by Helios. The security of the Belenios protocol has been verified in the computational model in Easycrypt [26]. The Belenios certification campaign [18]

[3]electis.com, concordium.com/article/concordiums-on-chain-voting-protocol

uses formal verification to aid ANSII certification and provides recommendations for a further integration of formal verification in such certification endeavors; see also [11, 62]. They emphasize the need for formal guarantees of the *implementation* of the protocol, not just the protocol itself. This is precisely a topic we contribute to: By showing that verifying cryptographic smart contracts is feasible, we hope to increase the general requirements for smart contracts, as e.g., the EU Data Act[4] already demands using best practices. Similarly, there is high demand for electronic voting. For example, the Swiss voting regulation already demands formal proofs of cryptographic security[5]. However, these voting regulations currently say little about software security.

8.2 Rust Verification

Rust is a multi-paradigm language and as such verification of Rust programs varies with the application domain one targets. In the security domain, the safe part of Rust with a functional semantics tends to suffice, as one can separate the verification of the main protocol from the implementation of the cryptographic primitives. The latter tend to be best generated from proof assistants; e.g. [33, 49, 69]. The main protocol can then be made parametric over this implementation by, for example, the Rust trait system. This separation of concerns also means that issues like side-channel resistance can be treated by dedicated toolchains such as the ones mentioned above.

There are several Rust verification tools in active development; see [66, 5.1] for a recent overview. None of them has been used on smart contracts, moreover Hax is the only one that has been used on larger cryptographic protocols.

We restrict our discussion here to Rust verification in proof assistants.

- RefinedRust [37] uses RustBelt [48] to reason about a variant of Rust's Vec implementation that involves intricate reasoning about unsafe pointer-manipulating code. Its target domain is small and very complex Rust code, whereas Hax focuses on bigger protocols in the security domain.
- Aeneas [46] provides a functional semantics for a larger subset of safe Rust than Hax. Hax and Aeneas share the frontend code. Aeneas starts from Rust's MIR intermediate language, whereas Hax starts from the higher level HIR, thus staying closer to the code the programmer wrote. Aeneas is being used to verify Microsoft's SymCrypt library of verified cryptographic primitives[6]. However, it does not include an embedding into a framework like SSProve to verify security protocols, nor has it been used for smart contract security. Its main backend is in Lean, which does not have a verified compiler to Wasm. However, Aeneas' Rocq backend could target Wasm similarly as we do.

8.3 Verification of Cryptographic Implementations

A careful comparison with related work on cryptographic verification is available in the SSProve paper [45]. An important contribution of the present work is that we verify Rust code and generate on-chain Wasm.

The closest work to ours is perhaps the verification of a post-quantum TLS implementation [16]. Like us, it uses Hax and SSProve. However, they use a more hybrid approach that combines multiple provers. Unlike us, they do not use verified compilation of Rust code.

Ho *et.al.* [47] provide verified C implementations for instances of the Noise protocol. Their framework provides verification in the symbolic ('Dolev-Yao') model of cryptography. The symbolic

[4]https://www.eu-data-act.com/Data_Act_Article_36.html
[5]https://www.bk.admin.ch/bk/en/home/politische-rechte/e-voting/sicherheit-beim-e-voting.html
[6]https://www.microsoft.com/en-us/research/blog/rewriting-symcrypt-in-rust-to-modernize-microsofts-cryptographic-library/

model can be used for Zero-knowledge proofs [5], but this would gloss over many more details of the implementation.

Protzenko *et.al.* [61] emphasize the need for providing a Wasm implementation of HACS. They provide efficient, but unverified, extraction from F* to Wasm, which they apply to the HACL* cryptographic library and libsignal.

OwlC [65], like us, emphasizes the need for secure *implementations*. They start from a symbolic protocol and emit Rust code. The generation of rust code is trusted, and, moreover, it still depends on the Rust compiler. Instead, we compile, in a verified way, to Wasm. Their key application domain consists of protocols that include a lot of communication, such as those traditionally analyzed in the symbolic model. It has not been applied to zero-knowledge applications yet, and it is unclear to us whether they would be supported by the Owl language.

8.4 Smart Contract Verification

As with other software, lightweight formal methods are in regular use for smart contracts. Here, we restrict our focus to formal verification using proof assistants.

Simplicity [59] is a smart contract language for Bitcoin that comes with formalized semantics.

The position paper [15] was the first to consider verified compilation from Solidity to EVM using F*. Grishchenko *et.al.* [39] provides a complete small-step semantics of EVM bytecode in the F* proof assistant.

Scilla [64] is a smart contract IR with a semantics in Rocq. It has been used to verify a Crowdfunding contract.

ConCert (Section 2.1) provides verified extraction from Rocq to a number of smart contract languages, including Rust and Tezos' CameLIGO. It has been used to prove key properties of interacting smart contracts [2, 4, 58], such as a decentralized exchange (DEX), a DAO, escrow, token standards, and a liquidity exchange protocol.

Mi-cho-Coq [10] formalizes the operational semantics of Tezos' Michelson on-chain language.

Cardano uses a UTXO-like model. They provide verified implementations of a multi-signature wallet and a DEX [35] in Agda, which they then extract to Haskell. Cardano uses a Haskell-like smart contract language, Plutus. Krijnen *et.al.* [51] present an ongoing effort to provide a *verifying* compiler, based on translation validation — that is, during the compilation of a program, the compiler provides independently-checkable evidence that each compiler step is carried out correctly. Djed [68] is a Plutus stable coin contract. Parts of its mathematical specification have been verified in Isabelle.

The MoveProver [32] is a dedicated prover for the Rust-like Move smart contract language.

8.5 Future Work

As discussed in Section 7 and 6, there are a number of issues in the Rocq eco-system that make it hard to combine libraries. The forthcoming integration of Sort polymorphism [60] in MetaRocq, algebraic universes [13] and Trocq, should facilitate developing large libraries like ours.

Some of our proofs can likely be simplified by using the nominal extension of SSProve [52], which is in the process of being merged into the main SSProve library.

The performance of our Rust implementation can be improved by using a verified Rust implementation of elliptic curves, such as is available in the libcrux library [49]. On the Wasm side, one could extend CertiRocq-Wasm with verified implementations of optimized cryptography. Here, one could follow the methodology of VeriFFI [50].

ACKNOWLEDGEMENT

We would like to thank Markus Krabbe Larsen who spotted an insecurity in the definition of the discrete log assumption and gave a reference for how to fix it. We would also like to thank DIREC for funding the project.

REFERENCES

[1] Ben Adida and C. Andrew Neff. 2006. Ballot casting assurance. In *Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop 2006 on Electronic Voting Technology Workshop (EVT'06)*. USENIX Association, USA, 7.

[2] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. 2021. Extracting Smart Contracts Tested and Verified in Coq. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2021)*. Association for Computing Machinery, New York, NY, USA, 105–121. https://doi.org/10.1145/3437992.3439934

[3] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. 2022. Extracting functional programs from Coq, in Coq. *Journal of Functional Programming* 32 (2022), e11. https://doi.org/10.1017/S0956796822000077

[4] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. 2020. ConCert: A Smart Contract Certification Framework in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 215–228. https://doi.org/10.1145/3372885.3373829

[5] Michael Backes, Matteo Maffei, and Dominique Unruh. 2008. Zero-Knowledge in the Applied Pi-calculus and Automated Verification of the Direct Anonymous Attestation Protocol. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE Computer Society, New York City, USA, 202–215. https://doi.org/10.1109/SP.2008.23

[6] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-Aided Cryptography. In *42nd IEEE Symposium on Security and Privacy, SP 2021*. IEEE, New York City, USA, 777–795.

[7] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *FOSAD (Lecture Notes in Computer Science, Vol. 8604)*. Springer, 146–166.

[8] David A. Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. 2022. Tamarin: Verification of Large-Scale, Real-World, Cryptographic Protocols. *IEEE Secur. Priv.* 20, 3 (2022), 24–32.

[9] Josh Benaloh, Michael Naehrig, Olivier Pereira, and Dan S. Wallach. 2024. ElectionGuard: a Cryptographic Toolkit to Enable Verifiable Elections. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, USA, 5485–5502. https://www.usenix.org/conference/usenixsecurity24/presentation/benaloh

[10] Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. 2019. Mi-Cho-Coq, a Framework for Certifying Tezos Smart Contracts. In *FM Workshops (1) (Lecture Notes in Computer Science, Vol. 12232)*. Springer, 368–379.

[11] Yves Bertot, Maxime Dénès, Arnaud Fontaine, Vincent Laporte, and Thomas Letan. 2020. Requirements on the Use of Coq in the Context of Common Criteria Evaluations. , 20 pages. https://inria.hal.science/hal-04452421

[12] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2017. A messy state of the union: taming the composite state machines of TLS. *Commun. ACM* 60, 2 (2017), 99–107.

[13] Marc Bezem, Thierry Coquand, Peter Dybjer, and Martín Escardó. 2022. Type Theory with Explicit Universe Polymorphism. In *TYPES (LIPIcs, Vol. 269)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:16.

[14] Karthikeyan Bhargavan, Maxime Buyse, Lucas Franceschino, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. 2025. hax: Verifying Security-Critical Rust Software Using Multiple Provers. In *Verified Software. Theories, Tools and Experiments: 16th International Conference, VSTTE 2024, Prague, Czech Republic, October 14–15, 2024, Revised Selected Papers*. Springer-Verlag, Berlin, Heidelberg, 96–119. https://doi.org/10.1007/978-3-031-86695-1_7

[15] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. Association for Computing Machinery, New York, NY, USA, 91–96. https://doi.org/10.1145/2993600.2993611

[16] Karthikeyan Bhargavan, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. 2025. Formal Security and Functional Verification of Cryptographic Protocol Implementations in Rust. Cryptology ePrint Archive, Paper 2025/980. https://eprint.iacr.org/2025/980

[17] Bruno Blanchet. 2016. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Found. Trends Priv. Secur.* 1, 1-2 (2016), 1–135.

[18] Angèle Bossuat, Eloïse Brocas, Véronique Cortier, Pierrick Gaudry, Stéphane Glondu, and Nicolas Kovacs. 2024. Belenios: the Certification Campaign. In *SSTIC 2024 - Symposium sur la sécurité des technologies de l'information et des*

communications. HAL CCSD, Rennes, France, 19 pages. https://inria.hal.science/hal-04578848

[19] Lars Brünjes and Murdoch James Gabbay. 2020. UTxO- vs Account-Based Smart Contract Blockchain Programming Paradigms. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 73–88. https://doi.org/10.1007/978-3-030-61467-6_6

[20] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. 2018. State Separation for Code-Based Game-Playing Proofs. In *Advances in Cryptology – ASIACRYPT 2018 (Lecture Notes in Computer Science, Vol. 11274)*, Thomas Peyrin and Steven Galbraith (Eds.). Springer International Publishing, Cham, 222–249. https://doi.org/10.1007/978-3-030-03332-3_9

[21] Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. 2020. The Extended UTXO Model. In *Financial Cryptography and Data Security*, Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala (Eds.). Springer International Publishing, Cham, 525–539.

[22] Cyril Cohen, Enzo Crance, and Assia Mahboubi. 2024. Trocq: Proof Transfer for Free, With or Without Univalence. In *Programming Languages and Systems (Lecture Notes in Computer Science, Vol. 14576)*, Stephanie Weirich (Ed.). Springer, Cham, 239–268.

[23] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. 2013. Refinements for Free!. In *CPP (Lecture Notes in Computer Science, Vol. 8307)*. Springer, 147–162.

[24] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. 2020. Hierarchy Builder: Algebraic hierarchies Made Easy in Coq with Elpi (System Description). In *FSCD (LIPIcs, Vol. 167)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 34:1–34:21.

[25] Véronique Cortier, Constantin Catalin Dragan, François Dupressoir, Benedikt Schmidt, Pierre-Yves Strub, and Bogdan Warinschi. 2017. Machine-Checked Proofs of Privacy for Electronic Voting Protocols. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, New York City, USA, 993–1008. https://doi.org/10.1109/SP.2017.28

[26] Véronique Cortier, Constantin Catalin Dragan, François Dupressoir, and Bogdan Warinschi. 2018. Machine-checked proofs for electronic voting: privacy and verifiability for Belenios. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, IEEE Computer Society, New York City, USA, 298–312. https://doi.org/10.1109/CSF.2018.00029

[27] Ronald Cramer. 1997. *Modular Design of Secure yet Practical Cryptographic Protocols*. Ph. D. Dissertation. Quantum Computing and Advanced System Research, Universiteit van Amsterdam.

[28] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. 1994. Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols. In *Advances in Cryptology — CRYPTO '94 (Lecture Notes in Computer Science, Vol. 839)*, Yvo Desmedt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 174–187. https://doi.org/10.1007/3-540-48658-5_19

[29] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. 1997. A secure and optimally efficient multi-authority election scheme. *Eur. Trans. Telecommun.* 8, 5 (1997), 481–490.

[30] Ivan Damgård. 2010. On Σ-protocols. https://www.cs.au.dk/~ivan/Sigma.pdf

[31] Ivan Damgård. 1998. Commitment Schemes and Zero-Knowledge Protocols. In *Lectures on Data Security, Modern Cryptology in Theory and Practice, Summer School, Aarhus, Denmark, July 1998 (Lecture Notes in Computer Science, Vol. 1561)*, Ivan Damgård (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 63–86. https://doi.org/10.1007/3-540-48969-X_3

[32] David L. Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Jingyi Emma Zhong. 2022. Fast and Reliable Formal Verification of Smart Contracts with the Move Prover. In *TACAS (1) (Lecture Notes in Computer Science, Vol. 13243)*. Springer, 183–200.

[33] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2020. Simple High-Level Code For Cryptographic Arithmetic: With Proofs, Without Compromises. *SIGOPS Oper. Syst. Rev.* 54, 1 (Aug. 2020), 23–30. https://doi.org/10.1145/3421473.3421477

[34] Uriel Feige and Adi Shamir. 1990. Witness Indistinguishable and Witness Hiding Protocols. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, Harriet Ortiz (Ed.). Association for Computing Machinery, New York, NY, USA, 416–426. https://doi.org/10.1145/100216.100272

[35] Tudor Ferariu, Philip Wadler, and Orestis Melkonian. 2025. Validity, Liquidity, and Fidelity: Formal Verification for Smart Contracts in Cardano. In *6th International Workshop on Formal Methods for Blockchains (FMBC 2025) (Open Access Series in Informatics (OASIcs), Vol. 129)*, Diego Marmsoler and Meng Xu (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 6:1–6:21. https://doi.org/10.4230/OASIcs.FMBC.2025.6

[36] Amos Fiat and Adi Shamir. 1987. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology — CRYPTO' 86*, Andrew M. Odlyzko (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 186–194.

[37] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1115–1139. https://doi.org/10.1145/3656422

[38] S Goldwasser, S Micali, and C Rackoff. 1985. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (STOC '85)*. Association for Computing Machinery, New York, NY, USA, 291–304. https://doi.org/10.1145/22145.22178

[39] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *POST (Lecture Notes in Computer Science, Vol. 10804)*. Springer, 243–269.

[40] Thomas Haines, Rajeev Goré, and Jack Stodart. 2020. Machine-Checking the Universal Verifiability of ElectionGuard. In *Secure IT Systems: 25th Nordic Conference, NordSec 2020, Virtual Event, November 23–24, 2020, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 57–73. https://doi.org/10.1007/978-3-030-70852-8_4

[41] Thomas Haines, Rajeev Goré, and Mukesh Tiwari. 2019. Verified Verifiers for Verifying Elections. In *CCS*. Association for Computing Machinery, New York, NY, USA, 685–702. https://doi.org/10.1145/3319535.3354247

[42] Thomas Haines, Rajeev Gore, and Mukesh Tiwari. 2023. Machine-checking Multi-Round Proofs of Shuffle: Terelius-Wikstrom and Bayer-Groth. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 6471–6488. https://www.usenix.org/conference/usenixsecurity23/presentation/haines

[43] F. Hao, P.Y.A. Ryan, and P. Zieliński. 2010. Anonymous Voting by Two-Round Public Discussion. *IET Information Security* 4 (2010), 62–67. Issue 2. https://doi.org/10.1049/iet-ifs.2008.0127 arXiv:https://digital-library.theiet.org/doi/pdf/10.1049/iet-ifs.2008.0127

[44] Philipp G. Haselwarter, Benjamin Salling Hvass, Lasse Letager Hansen, Théo Winterhalter, Cătălin Hriţcu, and Bas Spitters. 2024. The Last Yard: Foundational End-to-End Verification of High-Speed Cryptography. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2024)*. Association for Computing Machinery, New York, NY, USA, 30–44. https://doi.org/10.1145/3636501.3636961

[45] Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. 2023. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. *ACM Trans. Program. Lang. Syst.* 45, 3, Article 15 (jul 2023), 61 pages. https://doi.org/10.1145/3594735

[46] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP, Article 116 (aug 2022), 31 pages. https://doi.org/10.1145/3547647

[47] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. 2022. Noise: A Library of Verified High-Performance Secure Channel Protocol Implementations. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, New York City, USA, 107–124. https://doi.org/10.1109/SP46214.2022.9833621

[48] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

[49] Franziskus Kiefer, Karthikeyan Bhargavan, Lucas Franceschino, Denis Merigoux, Lasse Letager Hansen, Bas Spitters, Manuel Barbosa, Antoine Séré, and Pierre-Yves Strub. 2023. HACSPEC: a gateway to high-assurance cryptography. Video at https://youtu.be/lahO3de3k_0?t=7, abstract at https://github.com/hacspec/hacspec/blob/master/rwc2023-abstract.pdf.

[50] Joomy Korkut, Kathrin Stark, and Andrew W. Appel. 2025. A Verified Foreign Function Interface between Coq and C. *Proc. ACM Program. Lang.* 9, POPL, Article 24 (Jan. 2025), 31 pages. https://doi.org/10.1145/3704860

[51] Jacco O. G. Krijnen, Manuel M. T. Chakravarty, Gabriele Keller, and Wouter Swierstra. 2024. Translation certification for smart contracts. *Sci. Comput. Program.* 233 (2024), 103051.

[52] Markus Krabbe Larsen and Carsten Schürmann. 2025. An Induction Principle for Hybrid Arguments in Nominal-SSProve. , 1122 pages. https://eprint.iacr.org/2025/1122

[53] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[54] Assia Mahboubi and Enrico Tassi. 2022. *Mathematical Components*. Zenodo, Genève, Switzerland. https://doi.org/10.5281/zenodo.3999478

[55] Ueli Maurer. 2012. Constructive Cryptography – A New Paradigm for Security Definitions and Proofs. In *Theory of Security and Applications*, Sebastian Mödersheim and Catuscia Palamidessi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 33–56.

[56] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. 2017. A Smart Contract for Boardroom Voting with Maximum Voter Privacy. In *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10322)*, Aggelos Kiayias (Ed.). Springer International Publishing, Cham, 357–375. https://doi.org/10.1007/978-3-319-70972-7_20

[57] Wolfgang Meier, Martin Jensen, Jean Pichon-Pharabod, and Bas Spitters. 2025. CertiCoq-Wasm: A Verified WebAssembly Backend for CertiCoq. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Denver, CO, USA) *(CPP '25)*. Association for Computing Machinery, New York, NY, USA, 127–139. https://doi.org/10.1145/3703595.3705879

[58] Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. 2023. Formalising Decentralised Exchanges in Coq. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2023)*. Association for Computing Machinery, New York, NY, USA, 290–302. https://doi.org/10.1145/3573105.3575685

[59] Russell O'Connor. 2017. Simplicity: A New Language for Blockchains. In *PLAS@CCS*. ACM, 107–120.

[60] Josselin Poiret, Gaëtan Gilbert, Kenji Maillard, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau, and Éric Tanter. 2025. All Your Base Are Belong to Us: Sort Polymorphism for Proof Assistants. *Proc. ACM Program. Lang.* 9, POPL (2025), 2253–2281.

[61] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. 2019. Formally Verified Cryptographic Web Applications in WebAssembly. In *IEEE Symposium on Security and Privacy*. IEEE, New York City, USA, 1256–1274.

[62] Jonathan Protzenko and Bas Spitters. 2024. Modernizing FIPS for safe languages and verified libraries. NIST Workshop on Formal Methods within Certification Programs (FMCP 2024).

[63] Claus-Peter Schnorr. 1991. Efficient Signature Generation by Smart Cards. *J. Cryptol.* 4, 3 (1991), 161–174. https://doi.org/10.1007/BF00196725

[64] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018. Scilla: a Smart Contract Intermediate-Level LAnguage. *CoRR* abs/1801.00687 (2018).

[65] Pratap Singh, Joshua Gancher, and Bryan Parno. 2025. OwlC: Compiling Security Protocols to Verified, Secure, High-Performance Libraries. Cryptology ePrint Archive, Paper 2025/1092. https://eprint.iacr.org/2025/1092

[66] The Everest team. 2025. *Project Everest: Perspectives from Developing Industrial-grade High-Assurance Software*. Technical Report. Project Everest Team. https://project-everest.github.io/assets/everest-perspectives-2025.pdf

[67] Jelizaveta Vakarjuk, Nikita Snetkov, and Jan Willemson. 2025. Comparing security levels of postal and Internet voting. *Information Security Journal: A Global Perspective* 34, 4 (2025), 265–285. https://doi.org/10.1080/19393555.2024.2410332

[68] Joachim Zahnentferner, Dmytro Kaidalov, Jean-Frédéric Étienne, and Javier Díaz. 2021. Djed: A Formally Verified Crypto-Backed Pegged Algorithmic Stablecoin. *IACR Cryptol. ePrint Arch.* (2021), 1069.

[69] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). Association for Computing Machinery, New York, NY, USA, 1789–1806. https://doi.org/10.1145/3133956.3134043

6.5 Summary

In this paper, we show how to prove the security and correctness of a voting smart contract. We prove that the protocol keeps the vote hidden by showing maximum ballot secrecy in SSProve. The correctness property of *self-tallying* is shown using ConCert. The final voting property is *universal verifiability*, which states that a verifier can be constructed to validate runs of the protocol. This is, like the paper in the previous chapter (Chapter 5), an example of combining a *trace-based symbolic verification* tool with tools based on the *computational model* (e.g., SSProve), thereby extending the capabilities of both.

6.6 Maximum Ballot Secrecy - Alternative in SSP Style

The proof for maximum ballot secrecy in the paper uses code rewriting to show we can push the sampling of the private key after everything; thus, nothing will depend on it. However, this is not the state-separating proof (SSP) style (see §2.3) of doing proofs. In this section we will introduce how the proof would look if done in the SSP style [18, 21].

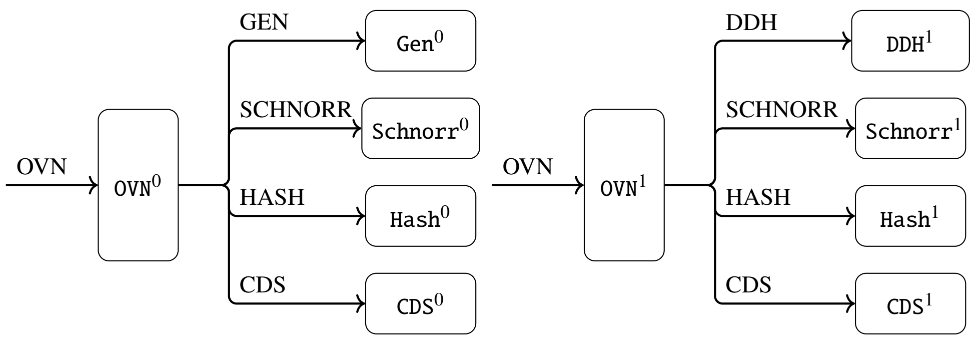
Overview of the Proof

The key insight to invoke the modularity of SSP is to keep common dependencies in a separate package. Thus, for OVN we would have a separate store for each part of the common information. E.g., the Schnorr protocol will produce a key and put it into K_{zkp}, instead of handling the shared information itself. This allows us to reason separately about the use of the information and the generation of the information. We will first look at how the proof in the paper would look in SSP style. Afterwards we will look at a formulation of the proof taking each party into account, thus allowing more arbitrary updates of the state.

Another generalization of the proof is to look at all parties communicating instead of looking at the state from a single party, as we did in the paper. The formalization proceeds in a similar manner, but more of the reasoning happens at the level of packages instead of reasoning of code. This helps increase the level of abstraction while maintaining precision and achieving the same security bound (per party).

To put the proof in perspective, we will show how the paper proof would look using the SSP games; see Figure 6.7. Thus, to go from the real game to the ideal game, we do the following sequence of game jumps:

- idealize the Schnorr protocol and use the discrete log assumption to then swap $\texttt{Schnorr}^1$ and $\texttt{Gen}^0$;
- use the discrete log assumption and the assumption that y_i is random to pack the vote in $\texttt{DDH}^0$, and idealize it;
- idealize the hash function and then swap $\texttt{Hash}^1$ and $\texttt{Gen}^0$;



(a) real OVN protocol (b) ideal OVN protocol

Figure 6.7: Maximum ballot secrecy proof from paper

- idealize the CDS protocol and then swap CDS^1 and Gen^0; and
- finally, idealize the private key generation as nothing depends on it.

The first step towards a more modular proof is to introduce the state as a package, instead of an input to the packages, and then separate OVN into the register, commit, vote and tally rounds. This is done in Figure 6.8. Now we can push the sampling

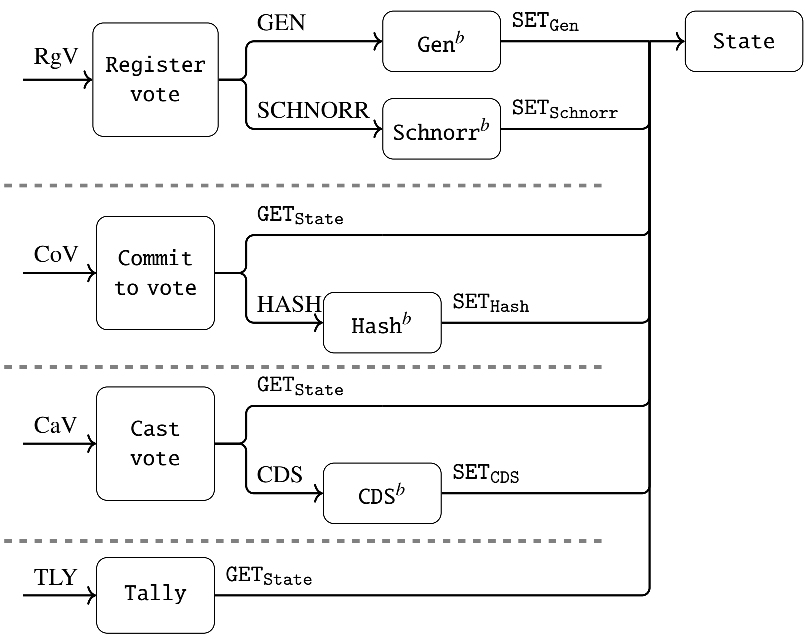


Figure 6.8: OVN with state as a package

to the bottom of each round separately and thus idealize by parts. This is closer to what actually happens in the SSProve formalization for maximum ballot secrecy in the paper.

The next step is then to split the state into parts by making a store for each value in the state; see Figure 6.9. Doing this allows us to look at a combination of parties running the protocols with different interleavings. Whereas we assumed the state was given before, thus generalizing over other parties' effect on the state. The proof here

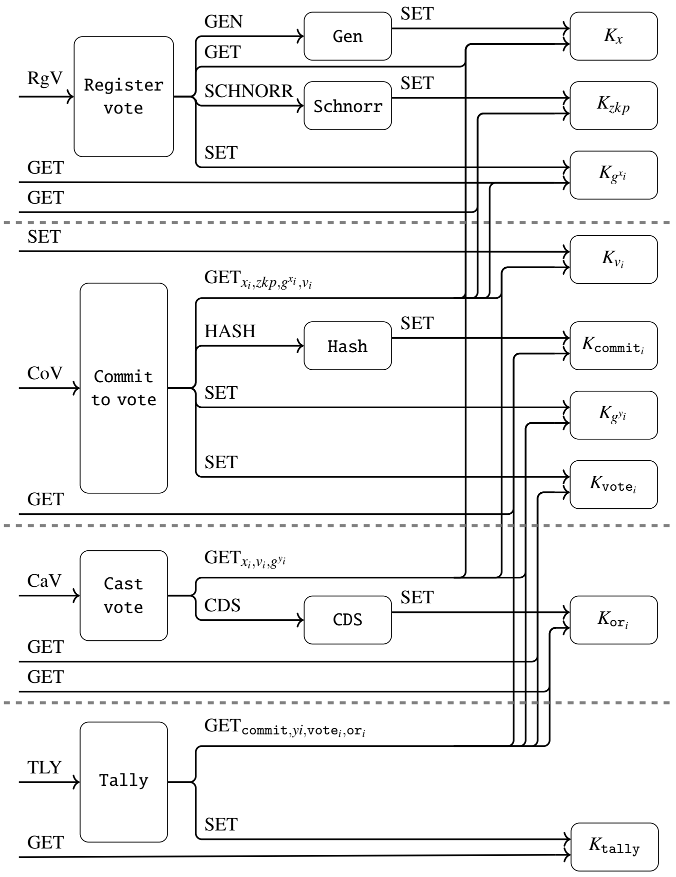


Figure 6.9: Maximum ballot secrecy

will then go by idealizing the composition of running the first round of each party, followed by running the second round of each party, and so on. Thus, we can focus on a round at a time. Furthermore, the idealization for any party is symmetric and independent (in a round), allowing us to reuse the argument for idealizing a single party. The modular SSP version of the OVN game is given in Figure 6.10, again split into the rounds. The ideal game just sets the values of the key stores at random, thereby requiring us to show the values stored are indistinguishable from randomly sampled ones.

The Technical Details

Since the game can be modularized by splitting it into rounds, we can reason about how to idealize each round separately.

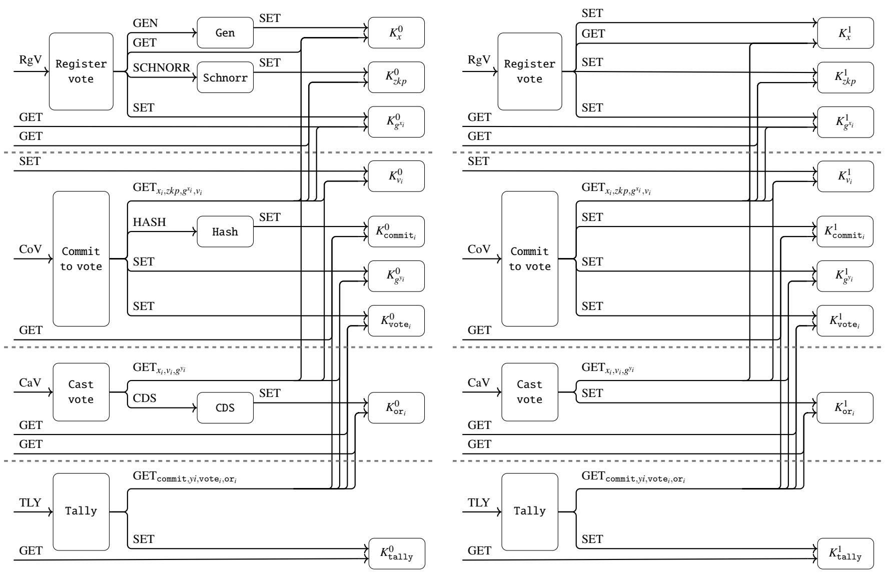


Figure 6.10: Modular version of OVN

Register Vote

We start with the register vote round and idealize it in steps; see Figure 6.11.

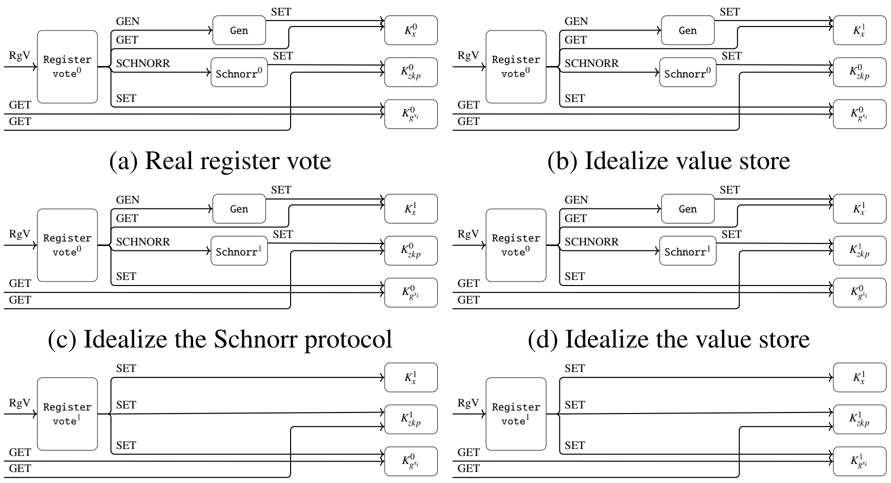


Figure 6.11: Maximum ballot secrecy - register vote

We start by idealizing the value store for key generation, then we idealize the Schnorr protocol, then finally the value store. We use the discrete log (DL) assumption to idealize the value store for g^x. But first, we define the value store $K^b_v : v \in V$, see Figure 6.12. For the Schnorr protocol (see Figure 6.1), the code uses the value stores; see Figure 6.13. We can now define a real and ideal package for all parties in the OVN

$\mathtt{K}_v^b : v \in V$

```
SET^0_v(x)
ov ← get_v
if is_none ov
  put_v x
fi
```

```
SET^1_v(_)
ov ← get_v
if is_none ov
  x ←$ V
  put_v x
fi
```

```
GET_v
ov ← get_v
match ov
  | none   ⇒ fail
  | some v ⇒ ret v
```

Figure 6.12: Value store

```
schnorr^0_i
h ← GET_{g^{x_i}}
m ← GET_{x_i}
r ←$ Z_q
u ← g^r
c ← H(g, h, u)
z ← c · m + r
SET_{zkp_i}(u, c, z)
```

```
schnorr^1_i
h ← GET_{g^{x_i}}
z ←$ Z_q
c ←$ Z_q
u ← g^z / h^c
SET_{zkp_i}(u, c, z)
```

Figure 6.13: The Schnorr protocol using the value stores

protocol. By indexing these two packages using a Boolean b, with 0 representing real and 1 representing ideal, resulting in the following security game

$$G^b_{\mathtt{schnorr}(x,zkp)} ::= \mathtt{schnorr}^b_i \circ (K^b_{x_i} \otimes K_{g^{x_i}} \otimes K^b_{zkp_i}).$$

We prove the security of the game by idealizing the parts step by step by

$$\begin{aligned} G^0_{\mathtt{schnorr}(x,zkp)} &= \mathtt{schnorr}^0_i \circ (K^0_{x_i} \otimes K_{g^{x_i}} \otimes K^0_{zkp_i}) \\ &\approx_0 \mathtt{schnorr}^0_i \circ (K^1_{x_i} \otimes K_{g^{x_i}} \otimes K^0_{zkp_i}) \\ &\approx_0 \mathtt{schnorr}^1_i \circ (K^1_{x_i} \otimes K_{g^{x_i}} \otimes K^0_{zkp_i}) \\ &\approx_0 \mathtt{schnorr}^1_i \circ (K^1_{x_i} \otimes K_{g^{x_i}} \otimes K^1_{zkp_i}) \\ &= G^1_{\mathtt{schnorr}(x,zkp)}. \end{aligned}$$

To idealize the Schnorr protocol, we do not need to idealize the public key. After idealizing the Schnorr protocol, we can idealize the value store for it, as the requirement of value being indistinguishable from randomness is now true. We assume the discrete log is hard; thus, we assume the existence of the security game in Figure 6.14. We can extend the definition to a game that includes the value store

$$G^b_{\mathtt{dl}(g^{x_i})} ::= \mathtt{dl}^b_i \circ (K_{x_i} \otimes K^b_{g^{x_i}}).$$

$\texttt{dl}_i^0$	$\texttt{dl}_i^1$
$x_i \xleftarrow{\$} \mathbb{Z}_q$	$x_i \xleftarrow{\$} \mathbb{Z}_q$
$\texttt{SET}_{x_i}(x_i)$	$\texttt{SET}_{x_i}(x_i)$
	$h \leftarrow\!\!\$\, \mathcal{G}$
$\texttt{SET}_{g^{x_i}}(g^{x_i})$	$\texttt{SET}_{g^{x_i}}(h)$

Figure 6.14: The discrete log game

Since the discrete log game is only assumed secure up to a (negligible) bound of ε, we get the same bound when including the value store

$$\begin{aligned} G^0_{\texttt{dl}(g^{x_i})} &= \texttt{dl}_i^0 \circ (K_{x_i} \otimes K^0_{g^{x_i}}) \\ &\approx_\varepsilon \texttt{dl}_i^1 \circ (K_{x_i} \otimes K^0_{g^{x_i}}) \\ &\approx_0 \texttt{dl}_i^1 \circ (K_{x_i} \otimes K^1_{g^{x_i}}) \\ &= G^1_{\texttt{dl}(g^{x_i})}. \end{aligned}$$

This enables us to define the registration round as the code in Figure 6.15. We combine

```
register_i
----------
dl_i
schnorr_i
```

Figure 6.15: The registration round code

the registration round with the above packages to get a security game for the first round as

$$G^b_{\texttt{register}_i} ::= \texttt{register}_i \circ (\texttt{schnorr}_i^b \otimes \texttt{dl}_i^b) \circ (K^b_{x_i} \otimes K^b_{g^{x_i}} \otimes K^b_{zkp_i}).$$

Reusing the security games from above, we can step-by-step prove the security of the round. Each application of the smaller security games requires us to isolate the game; however, this is a trivial exercise, using associativity and commutativity rules. We thereby get the security proof

$$\begin{aligned} G^0_{\texttt{register}_i} &= \texttt{register}_i \circ (\texttt{schnorr}_i^0 \otimes \texttt{dl}_i^0) \circ (K^0_{x_i} \otimes K^0_{g^{x_i}} \otimes K^0_{zkp_i}) \\ &\overset{G_{\texttt{schnorr}}}{\approx_0} \texttt{register}_i \circ (\texttt{schnorr}_i^1 \otimes \texttt{dl}_i^0) \circ (K^1_{x_i} \otimes K^0_{g^{x_i}} \otimes K^1_{zkp_i}) \\ &\overset{G_{\texttt{dl}}}{\approx_\varepsilon} \texttt{register}_i \circ (\texttt{schnorr}_i^1 \otimes \texttt{dl}_i^1) \circ (K^1_{x_i} \otimes K^1_{g^{x_i}} \otimes K^1_{zkp_i}) \\ &= G^1_{\texttt{register}_i}. \end{aligned}$$

Commit to Vote

Next we repeat this process for the commitment round. We start by making an illustration of the security proof; see Figure 6.16.

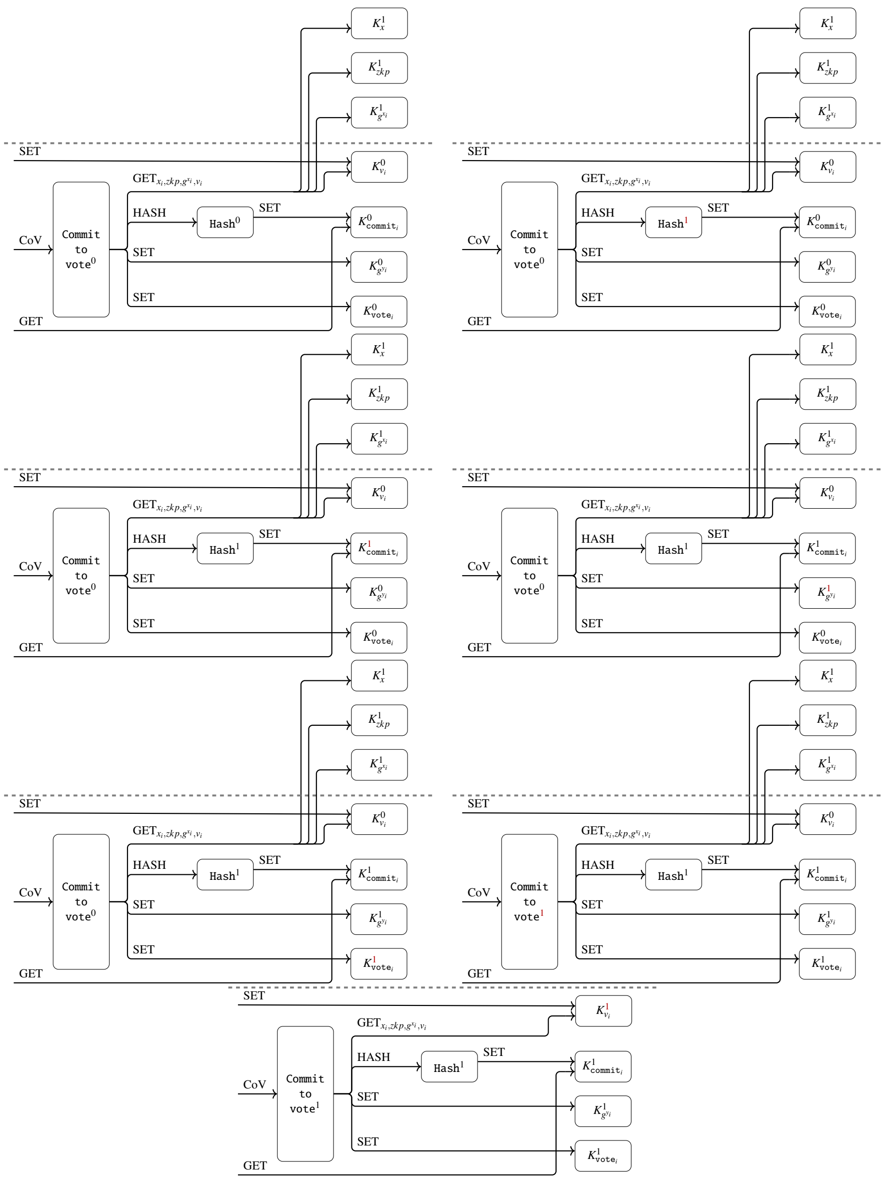


Figure 6.16: Maximum ballot secrecy - commit to vote

Now let us look at the game for each of the parts. We start by looking at g^y (see Figure 6.17), which assumes the value is randomly distributed, which is the case if there are two honest parties. It further assumes the value of y_i is hidden using the discrete log assumption as discussed for the single instance in the paper. We define

```
uniform^0_{y_i}
    $p_1 \leftarrow \prod_{j=0}^{i-1} \mathtt{GET}_{g^{x_j}}$
    $p_2 \leftarrow \prod_{j=i+1}^{n} \mathtt{GET}_{g^{x_j}}$
    $g^{y_i} \leftarrow \frac{p_1}{p_2}$
    $\mathtt{SET}_{g^{y_i}}(g^{y_i})$

uniform^1_{y_i}
    $h \leftarrow\$ \mathcal{G}$
    $\mathtt{SET}_{g^{y_i}}(h)$
```

Figure 6.17: Code for g^{y_i}, similar to DL

the security game for g^{y_i} for each party as

$$G^b_{g^{y_i}} ::= \mathtt{uniform}^b_{y_i} \circ \left(\bigotimes_{j=0}^{i-1} K_{g^{x_j}} \otimes \bigotimes_{j=i+1}^{n} K_{g^{x_j}} \otimes K^b_{g^{y_i}} \right).$$

This introduces the extra dependence of ν in the bound, as we assume the randomness of y_i. This assumption could possibly be proven by requiring two of the key stores to be idealized, thus ensuring the randomness of the value by DL. The proof goes as follows:

$$\begin{aligned} G^0_{g^{y_i}} &= \mathtt{uniform}^0_{y_i} \circ \left(\bigotimes_{j=0}^{i-1} K_{g^{x_j}} \otimes \bigotimes_{j=i+1}^{n} K_{g^{x_j}} \otimes K^0_{g^{y_i}} \right) \\ &\approx_\nu \mathtt{uniform}^1_{y_i} \circ \left(\bigotimes_{j=0}^{i-1} K_{g^{x_j}} \otimes \bigotimes_{j=i+1}^{n} K_{g^{x_j}} \otimes K^0_{g^{y_i}} \right) \\ &\approx_0 \mathtt{uniform}^1_{y_i} \circ \left(\bigotimes_{j=0}^{i-1} K_{g^{x_j}} \otimes \bigotimes_{j=i+1}^{n} K_{g^{x_j}} \otimes K^1_{g^{y_i}} \right) \\ &= G^1_{g^{y_i}}. \end{aligned}$$

For `vote` we use the Decisional Diffie-Hellman assumption; see Figure 6.18 for the code. The game for this code is as expected,

$$G^b_{\mathtt{vote}} ::= \mathtt{vote}^b_i \circ \left(K^1_{g^{y_i}} \otimes K^1_{x_i} \otimes K_{v_i} \otimes K^b_{\mathtt{vote}_i} \right).$$

The proof is straightforward, but it adds a factor to the security bound for applying Decisional Diffie-Hellman (DDH). It proceeds by

$$\begin{aligned} G^0_{\mathtt{vote}} &= \mathtt{vote}^0_i \circ \left(K^1_{g^{y_i}} \otimes K^1_{x_i} \otimes K_{v_i} \otimes K^0_{\mathtt{vote}_i} \right) \\ &\approx_\zeta \mathtt{vote}^1_i \circ \left(K^1_{g^{y_i}} \otimes K^1_{x_i} \otimes K_{v_i} \otimes K^0_{\mathtt{vote}_i} \right) \\ &\approx_0 \mathtt{vote}^1_i \circ \left(K^1_{g^{y_i}} \otimes K^1_{x_i} \otimes K_{v_i} \otimes K^1_{\mathtt{vote}_i} \right) \end{aligned}$$

$\texttt{vote}_i^0$	$\texttt{vote}_i^1$
$g^{y_i} \leftarrow \mathsf{GET}_{g^{y_i}}$	$z \leftarrow\!\!\$\ \mathbb{Z}_q$
$x_i \leftarrow \mathsf{GET}_{x_i}$	$\mathsf{SET}_{\texttt{vote}_i}(g^z)$
$v_i \leftarrow \mathsf{GET}_{v_i}$	
$\texttt{vote}_i \leftarrow g^{y_i x_i} \cdot g^{v_i}$	
$\mathsf{SET}_{\texttt{vote}_i}(\texttt{vote}_i)$	

Figure 6.18: Code for computing the vote

$$= G^1_{\texttt{vote}}.$$

The vote package could be split into two packages, namely a DDH package and a package showing that multiplying a random group element by the vote and publishing it does not reveal the vote, as shifting the distribution by one is still a uniform distribution. However, for simplicity we just combine the two packages here. The code for committing to the vote is in Figure 6.19. It assumes we are working in the

$\texttt{commit}_i^0$	$\texttt{commit}_i^1$
$\texttt{vote}_i \leftarrow \mathsf{GET}_{\texttt{vote}_i}$	$z \leftarrow\!\!\$\ \mathbb{Z}_q$
$\texttt{commit}_i \leftarrow \mathscr{H}(\texttt{vote}_i)$	$\mathsf{SET}_{\texttt{commit}_i}(z)$
$\mathsf{SET}_{\texttt{commit}_i}(\texttt{commit}_i)$	

Figure 6.19: Commit code

random oracle model. Thus, we also introduce an additional bound ψ to account for this assumption. The game for the commit code is

$$G^b_{\texttt{commit}} ::= \texttt{commit}_i^b \circ (K_{\texttt{vote}_i} \otimes K^b_{\texttt{commit}_i}),$$

and the security proof is

$$\begin{aligned} G^0_{\texttt{commit}} &= \texttt{commit}_i^0 \circ (K_{\texttt{vote}_i} \otimes K^0_{\texttt{commit}_i}) \\ &\approx_\psi \texttt{commit}_i^1 \circ (K_{\texttt{vote}_i} \otimes K^0_{\texttt{commit}_i}) \\ &\approx_0 \texttt{commit}_i^1 \circ (K_{\texttt{vote}_i} \otimes K^1_{\texttt{commit}_i}) \\ &= G^1_{\texttt{commit}}. \end{aligned}$$

We can now write up the full commit to vote round; see Figure 6.20. The security game is again each part and a composition of the dependencies defined as

$$\begin{aligned} G^b_{\texttt{commit\_vote}_i} ::= \texttt{commit\_vote}_i \circ (\texttt{uniform}^b_{y_i} \otimes \texttt{vote}^b_i \otimes \texttt{commit}^b_i) \circ \\ \left(\bigotimes_{j=0}^{i-1} K_{g^{x_j}} \otimes \bigotimes_{j=i+1}^{n} K_{g^{x_j}} \otimes K^b_{g^{y_i}} \otimes K_{x_i} \otimes K_{v_i} \otimes K^b_{\texttt{vote}_i} \otimes K^b_{\texttt{commit}_i} \right). \end{aligned}$$

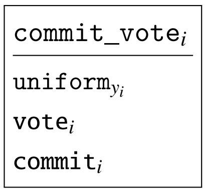


Figure 6.20: Commit to vote round

The proof again builds on the security proof for the parts. Applying the modular proofs requires doing rewrite operations to isolate the games as described for the register vote round. The proof for the commitment round follows

$$
\begin{aligned}
G^0_{\texttt{commit\_vote}_i} &= \texttt{commit\_vote}_i \circ (\texttt{uniform}^0_{y_i} \otimes \texttt{vote}^0_i \otimes \texttt{commit}^0_i) \circ \\
&\quad \left(\bigotimes_{j=0}^{i-1} K_{g^{x_j}} \otimes \bigotimes_{j=i+1}^{n} K_{g^{x_j}} \otimes K^0_{g^{y_i}} \otimes K_{x_i} \otimes K_{v_i} \otimes K^0_{\texttt{vote}_i} \otimes K^0_{\texttt{commit}_i} \right) \\
&\overset{G_{g^{y_i}}}{\approx_{\nu}} \texttt{commit\_vote}_i \circ (\texttt{uniform}^1_{y_i} \otimes \texttt{vote}^0_i \otimes \texttt{commit}^0_i) \circ \\
&\quad \left(\bigotimes_{j=0}^{i-1} K_{g^{x_j}} \otimes \bigotimes_{j=i+1}^{n} K_{g^{x_j}} \otimes K^1_{g^{y_i}} \otimes K_{x_i} \otimes K_{v_i} \otimes K^0_{\texttt{vote}_i} \otimes K^0_{\texttt{commit}_i} \right) \\
&\overset{G_{\texttt{vote}}}{\approx_{\zeta}} \texttt{commit\_vote}_i \circ (\texttt{uniform}^1_{y_i} \otimes \texttt{vote}^1_i \otimes \texttt{commit}^0_i) \circ \\
&\quad \left(\bigotimes_{j=0}^{i-1} K_{g^{x_j}} \otimes \bigotimes_{j=i+1}^{n} K_{g^{x_j}} \otimes K^1_{g^{y_i}} \otimes K_{x_i} \otimes K_{v_i} \otimes K^1_{\texttt{vote}_i} \otimes K^0_{\texttt{commit}_i} \right) \\
&\overset{G_{\texttt{commit}}}{\approx_{\psi}} \texttt{commit\_vote}_i \circ (\texttt{uniform}^1_{y_i} \otimes \texttt{vote}^1_i \otimes \texttt{commit}^1_i) \circ \\
&\quad \left(\bigotimes_{j=0}^{i-1} K_{g^{x_j}} \otimes \bigotimes_{j=i+1}^{n} K_{g^{x_j}} \otimes K^1_{g^{y_i}} \otimes K_{x_i} \otimes K_{v_i} \otimes K^1_{\texttt{vote}_i} \otimes K^1_{\texttt{commit}_i} \right) \\
&= G^1_{\texttt{commit\_vote}_i}.
\end{aligned}
$$

Cast Vote

We present the construction and idealization of the cast vote round. Again, we start with the illustration for the proof; see Figure 6.21. In this round we publish the vote and we do the CDS proof. Thus, we are only missing the definition and security game for CDS, as we can simply move the vote to a public value store. The code for CDS is in Figure 6.22. The security game is straightforward to define

$$G^b_{\texttt{CDS}} ::= \texttt{CDS}^b_i \circ (K^0_{x_i} \otimes K^0_{g^{x_i}} \otimes K_{v_i} \otimes K^b_{\texttt{CDS}_i}).$$

The proof builds on the proofs for CDS, but otherwise it progresses without any surprise,

$$G^0_{\texttt{CDS}} = \texttt{CDS}^0_i \circ (K^0_{x_i} \otimes K^0_{g^{x_i}} \otimes K_{v_i} \otimes K^0_{\texttt{CDS}_i})$$

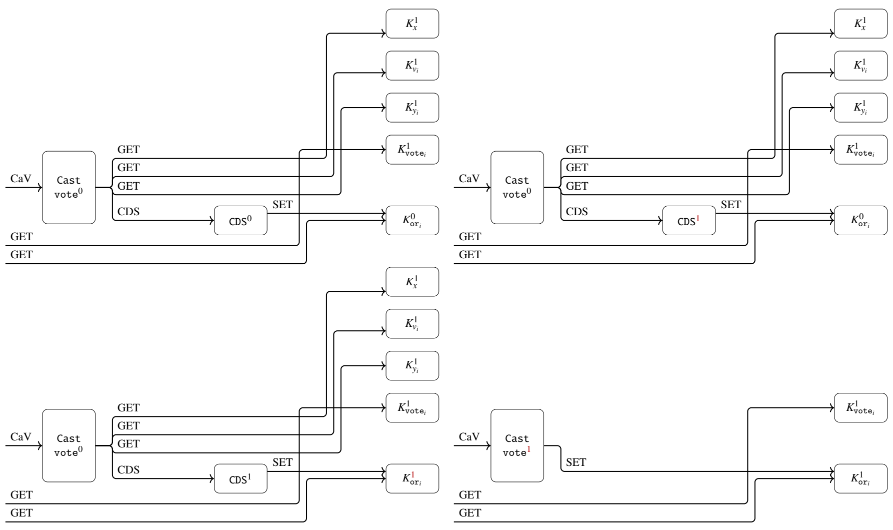


Figure 6.21: Maximum ballot secrecy - Cast vote

```
$\mathtt{CDS}^0_i(x,y)$
$x_i \leftarrow \mathtt{GET}_{x_i}$
$h \leftarrow \mathtt{GET}_{g^{x_i}}$
$v_i \leftarrow \mathtt{GET}_{v_i}$
$\mathtt{assert}\ x = g^{x_i} \wedge y = h^{x_i} \cdot g^{v_i}$
$v_i = 0$
    $w, r_2, d_2 \leftarrow\$ \mathbb{Z}_q$
    $x \leftarrow g^{x_i}$
    $y \leftarrow h^{x_i}$
    $a_1 \leftarrow g^w$
    $b_1 \leftarrow h^w$
    $a_2 \leftarrow g^{r_2} \cdot x^{d_2}$
    $b_2 \leftarrow h^{r_2} \cdot \left(\frac{y}{g}\right)^{d_2}$
    $c \leftarrow \mathscr{H}(..)$
    $d_1 \leftarrow c - d_2$
    $r_1 \leftarrow w - m \cdot d_1$
$v_i = 1$
    $w, r_1, d_1 \leftarrow\$ \mathbb{Z}_q$
    $x \leftarrow g^m$
    $y \leftarrow h^m \cdot g$
    $a_1 \leftarrow g^{r_1} \cdot x^{d_1}$
    $b_1 \leftarrow h^{r_1} \cdot y^{d_1}$
    $a_2 \leftarrow g^w$
    $b_2 \leftarrow h^w$
    $c \leftarrow \mathscr{H}(..)$
    $d_2 \leftarrow c - d_1$
    $r_2 \leftarrow w - m \cdot d_2$
$\mathtt{SET}_{\mathtt{CDS}}((a_1,b_1,a_2,b_2),c,(r_1,d_1,r_2,d_2))$
```

```
$\mathtt{CDS}^1_i(x,y)$
$h \leftarrow \mathtt{GET}_{g^{x_i}}$
$c \leftarrow\$ \mathbb{Z}_q$
$d, r, r' \leftarrow\$ \mathbb{Z}_q$
$d_2 \leftarrow d$
$r_2 \leftarrow r$
$r_1 \leftarrow r'$
$d_1 \leftarrow c - d_2$
$a_1 \leftarrow g^{r_1} \cdot x^{d_1}$
$b_1 \leftarrow h^{r_1} \cdot y^{d_1}$
$a_2 \leftarrow g^{r_2} \cdot x^{d_2}$
$b_2 \leftarrow h^{r_2} \cdot \left(\frac{y}{g}\right)^{d_2}$
$\mathtt{SET}_{\mathtt{CDS}}\ ((a_1,b_1,a_2,b_2),c,(r_1,d_1,r_2,d_2))$
```

Figure 6.22: CDS code

$$\approx_0 \mathtt{CDS}^1_i \circ (K^0_{x_i} \otimes K^0_{g^{x_i}} \otimes K_{v_i} \otimes K^0_{\mathtt{CDS}_i})$$
$$\approx_0 \mathtt{CDS}^1_i \circ (K^0_{x_i} \otimes K^0_{g^{x_i}} \otimes K_{v_i} \otimes K^1_{\mathtt{CDS}_i})$$

$$= G^1_{\mathtt{CDS}}.$$

We now have everything to define the cast vote round, as it is just running CDS, the definition is

$$G^b_{\mathtt{cast}_i} ::= G^b_{\mathtt{CDS}_i}.$$

The proof of the security game is just the security proof for CDS.

Tally

We can now define the tally round, which is only computation. The SSP security game illustration for the tally round is in Figure 6.23. The code for the tally round can be

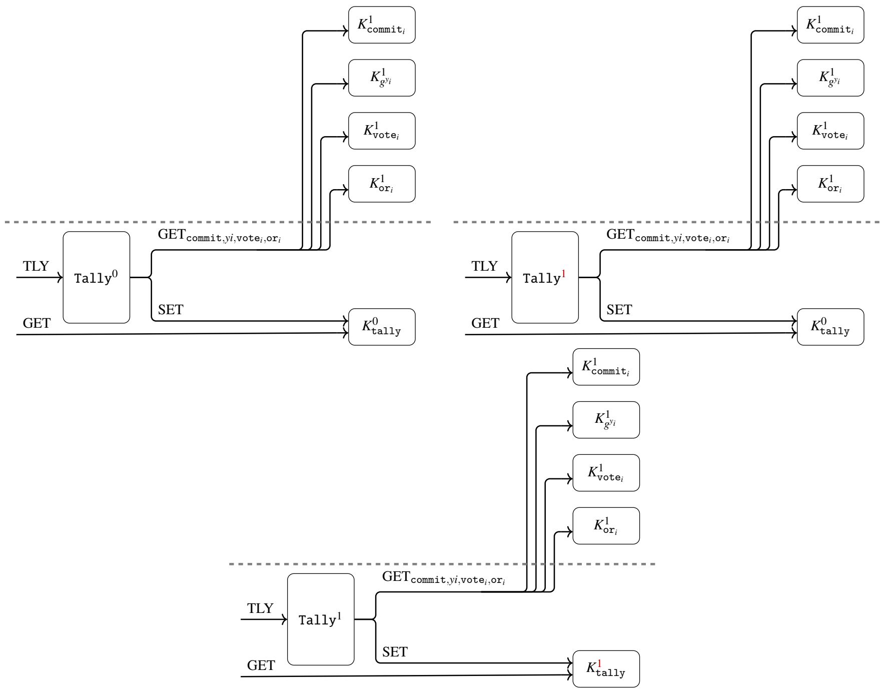


Figure 6.23: Maximum ballot secrecy - tally

found in Figure 6.24.

The Full Proof - Combination of the Parts

Now we have the definitions for each of the rounds. This allows us to define the game for the full OVN protocol. First we define the code in Figure 6.25. The security game is

$$G^b_{\mathtt{OVN}} ::= \mathtt{OVN} \circ \bigotimes_{i=0}^{n} \Big((\mathtt{register}_i \circ (\mathtt{schnorr}^b_i \otimes \mathtt{dl}^b_i) \\ \otimes \mathtt{commit\_vote}_i \circ (\mathtt{uniform}^b_{y_i} \otimes \mathtt{vote}^b_i \otimes \mathtt{commit}^b_i) \otimes \mathtt{CDS}^b_i)$$

```
tally
$g^{\texttt{tally}} \leftarrow \prod_{j=0}^{n} \texttt{GET}_{\texttt{vote}_j}$
$\texttt{tally} = \prod_{j=0}^{n} j \cdot (g^j = g^{\texttt{tally}})$
$\texttt{SET}_{\texttt{tally}}(\texttt{tally})$
```

Figure 6.24: Tally round code

$$\circ \left(K^b_{x_i} \otimes K^b_{zkp_i} \otimes \bigotimes_{j=0}^{n} K^b_{g^{x_j}} \otimes K^b_{g^{y_i}} \otimes K^b_{v_i} \otimes K^b_{\texttt{vote}_i} \otimes K^b_{\texttt{commit}_i} \otimes K^b_{\texttt{CDS}_i} \right) \Bigg),$$

and finally, we get the full security proof for maximum ballot secrecy

$$\begin{aligned}
G^0_{\texttt{OVN}} = \texttt{OVN} \circ \bigotimes_{i=0}^{n} \Bigg(&(\texttt{register}_i \circ (\texttt{schnorr}^0_i \otimes \texttt{dl}^0_i) \\
&\otimes \texttt{commit\_vote}_i \circ (\texttt{uniform}^0_{y_i} \otimes \texttt{vote}^0_i \otimes \texttt{commit}^0_i) \otimes \texttt{CDS}^0_i) \\
&\circ \left(K^0_{x_i} \otimes K^0_{zkp_i} \otimes \bigotimes_{j=0}^{n} K^0_{g^{x_j}} \otimes K^0_{g^{y_i}} \otimes K^0_{v_i} \otimes K^0_{\texttt{vote}_i} \otimes K^0_{\texttt{commit}_i} \otimes K^0_{\texttt{CDS}_i} \right) \Bigg) \\
\overset{G_{\texttt{cast}}}{\approx_0} \texttt{OVN} \circ \bigotimes_{i=0}^{n} \Bigg(&(\texttt{register}_i \circ (\texttt{schnorr}^0_i \otimes \texttt{dl}^0_i) \\
&\otimes \texttt{commit\_vote}_i \circ (\texttt{uniform}^0_{y_i} \otimes \texttt{vote}^0_i \otimes \texttt{commit}^0_i) \otimes \texttt{CDS}^1_i) \\
&\circ \left(K^0_{x_i} \otimes K^0_{zkp_i} \otimes \bigotimes_{j=0}^{n} K^0_{g^{x_j}} \otimes K^0_{g^{y_i}} \otimes K^0_{v_i} \otimes K^0_{\texttt{vote}_i} \otimes K^0_{\texttt{commit}_i} \otimes K^1_{\texttt{CDS}_i} \right) \Bigg) \\
\overset{G_{\texttt{register}}}{\approx_{n \cdot \varepsilon}} \texttt{OVN} \circ \bigotimes_{i=0}^{n} \Bigg(&(\texttt{register}_i \circ (\texttt{schnorr}^1_i \otimes \texttt{dl}^1_i) \\
&\otimes \texttt{commit\_vote}_i \circ (\texttt{uniform}^0_{y_i} \otimes \texttt{vote}^0_i \otimes \texttt{commit}^0_i) \otimes \texttt{CDS}^1_i)
\end{aligned}$$

```
OVN
$\texttt{register}_i \quad \forall i \in 0..n$
$\texttt{commit\_vote}_i \quad \forall i \in 0..n$
$\texttt{cast}_i \quad \forall i \in 0..n$
tally
```

Figure 6.25: Full protocol code

$$
\begin{aligned}
&\circ \left(K^{1}_{x_i} \otimes K^{1}_{zkp_i} \otimes \bigotimes_{j=0}^{n} K^{1}_{g^{x_j}} \otimes K^{0}_{g^{y_i}} \otimes K^{0}_{v_i} \otimes K^{0}_{\texttt{vote}_i} \otimes K^{0}_{\texttt{commit}_i} \otimes K^{1}_{\texttt{CDS}_i} \right) \Bigg) \\
\overset{G_{\texttt{commit\_vote}}}{\approx}_{n\cdot(\nu+\zeta+\psi)} \; & \texttt{OVN} \circ \bigotimes_{i=0}^{n} \Bigg((\texttt{register}_i \circ (\texttt{schnorr}^{1}_{i} \otimes \texttt{dl}^{1}_{i}) \\
&\otimes \texttt{commit\_vote}_i \circ (\texttt{uniform}^{1}_{y_i} \otimes \texttt{vote}^{1}_{i} \otimes \texttt{commit}^{1}_{i}) \otimes \texttt{CDS}^{1}_{i}) \\
&\circ \left(K^{1}_{x_i} \otimes K^{1}_{zkp_i} \otimes \bigotimes_{j=0}^{n} K^{1}_{g^{x_j}} \otimes K^{1}_{g^{y_i}} \otimes K^{1}_{v_i} \otimes K^{1}_{\texttt{vote}_i} \otimes K^{1}_{\texttt{commit}_i} \otimes K^{1}_{\texttt{CDS}_i} \right) \Bigg) \\
= \; & G^{1}_{\texttt{OVN}}.
\end{aligned}
$$

This gives us the same security bound of $\varepsilon + \nu + \zeta + \psi$ per party as in the paper. As long as this is a negligible probability, then doing it any polynomial amount of times is still negligible and thus secure.

6.7 Modifications of Implementation

The OVN protocol can also be improved for use as a smart contract. Here, the focus is the cost per party to participate in the protocol. Currently, the validations are done by each party. However, since we trust the blockchain, each party can *prove their own public values* valid. This will save a factor of n. Furthermore, we can add a round to the protocol for computing the reconstruction key g^{y_i} for each party. Doing this as a linear scan, instead of having each party do it, also saves a factor of n. Alternatively, one can batch the computations in blocks of size k, and then each party only needs to do around $(n/k) + k - 1$ work instead of the current n for the reconstruction key. These two solutions do, however, have a tradeoff in that they increase the size of the contract state. The batching trick exploits the ordering of the computations on the blockchain, which might make internal optimizations of the blockchain less efficient, which could increase the cost. The computation of the *final tally* could also be done *off-chain*, and the result published to the blockchain, which checks that $\prod_{i=1}^{n} vote_i = g^{tally}$. These modifications to the OVN protocol could make it run in *constant time* for each party, though using a linear amount of space.

Chapter 7

Related Work

We will introduce some alternative projects and tools within the fields of formal guarantees for Rust code and verification of cryptography. We start by presenting some Rust projects, then we describe some other projects doing cryptographic formalization. Finally, we will talk about ongoing efforts to collaborate and widen the field.

7.1 Formal Verification in Rust

We will look at related work within high-assurance cryptography with a focus on Rust tools and frameworks. We will also introduce alternatives to the frameworks we have presented and highlight the differences and the reason we choose the tools we use for the type of verification we have done above.

Formal Semantics of Rust

Rust is a very promising language with strong guarantees with a focus on allowing multiple coding paradigms. Rust programmers, tools and projects take many of these guarantees as a given, however, Rust does not have a formal semantics. This means that these assumptions might not hold. To remedy this projects have been started to define or verify semantics of parts or the entirety of Rust. We will present some of these efforts here.

The aim with the *RustBelt* project [57] is to formally investigate the safety claims of Rust. Thus, trying to improve on the surface documentation of unsafe behavior found in the Rustonomicon [89].

The *RustHorn* paper [71] introduces a translation of Rust programs into constraint Horn clauses (CHC), where program verification can be done formally. The project relies on the strong invariants of the Rust type system to model the behavior of stateful Rust code with first-order logic.

RustHornBelt [70] is the combination of these two projects, ensuring the soundness of the RustHorn translation and models. This allows one to do machine-checked proofs of safety and correctness for Rust code.

The focus of this thesis is on a smaller safe subset of Rust. The reason we only want to support a smaller subset of Rust is to have obviously correct translations for simple implementation. Most cryptographic primitives and protocols can be implemented in the Hax subset, for which we can use simpler verification methods and ensure stronger guarantees. We still want a formal semantic of Rust (or at least the Hax subset). So ensuring we agree with this work on the Hax subset is also important.

A project that tries to capture the essence of (surface-level) Rust is *Oxide* [96, 97]. It makes a hierarchical description with increasing expressibility of the semantics of Rust. A representation of safe Rust is $\texttt{Oxide}_0$, which is then extended with abstractions implemented using `unsafe` code. New levels are defined when there are observationally equivalent programs in $\texttt{Oxide}_n$ that are not observationally equivalent in $\texttt{Oxide}_{n+1}$. This allows one to describe features and optimizations based on the abstraction level.

This approach closely mirrors the Hax approach of defining the language as a feature set. The current features of Hax seem equivalent to $\texttt{Oxide}_1$; however, no formal connection exists.

The *verify Rust std lib project*[1] has posed a number of challenges, which aims at verifying parts of the standard library for Rust. The tools currently being used for this are ESBMC (GOTO-Transcoder)[2], Flux [64], Kani [95], and VeriFast for Rust [39, 53].

The verification of the standard library should produce models for the data structures and functions, which can simplify proofs about Rust code. However, rewriting these models for all proof assistants and tools that want to proof things about Rust might introduce errors or complexity. Our work on the annotated core library is an attempt at fixing this problem but is less formal and only for the core library.

The goal with making the semantics explicit is to allow formalization of guarantees and trust in the correctness and soundness of Rust. Another part to this is testing and documentation, which ensures Rust qualifies to be used for critical systems. *Ferrocene*[3,4] is a toolchain for using Rust in safety-critical environments. It has qualifications in accordance with ISO 26262 (ASIL D), IEC 61508 (SIL 4), and IEC 62304. As part of the development, the Ferrocene language specification[5] (FLS) was made and later adopted by Rust.

These projects show that the community is putting a strong effort into fully formalizing the semantics of Rust, and we can see the translation of Hax as another semantic for the subset of Rust that Hax supports. Ensuring these semantics align will allow us to formalize the translation done by Hax and other similar tools.

[1]`https://github.com/model-checking/verify-rust-std`

[2]`https://github.com/rafaelsamenezes/goto-transcoder`

[3]`https://ferrocene.dev/en/`

[4]`https://github.com/ferrocene/ferrocene`

[5]`https://public-docs.ferrocene.dev/main/specification/index.html`

Verification of Unsafe Rust

Sometimes Rust is going to call external tools or break from the intended information flow or security guarantees. Usually these deviations are done to optimize performance or interact with a larger system, or simply implement functionality not directly possible in Rust. This is a necessity of system-level programming language; however, we still want to ensure the correctness of this possibly unsafe behavior.

Having a formal description of the language to be analyzed, like what is described in the previous section, is very useful. *MiniRust*[6] [55] tries to carve out the core language of Rust. It expresses all the unpleasant unsafe behavior in Rust but is executable, e.g., there is an interpreter; however, it does not come with the normal conveniences of Rust, like a concrete syntax. Thus, writing code directly in MiniRust is not intended. *Specr lang* exists as a meta language. MiniRust is still not complete; however, it is another direction for specifying the undefined behaviors of Rust by minimizing the formalization surface. The focus of MiniRust is the operational semantics of Rust; thus, the MiniRust spec is written as an interpreter. MiniRust could become a semantic used by the tools of this section, serving the same purpose as the projects of the last section, but with a focus on unsafe Rust.

A tool for verifying the behavior of Rust programs is *Kani* [95], which uses model checking to analyze Rust programs and focuses on verifying unsafe Rust programs. Kani can check memory safety properties and some runtime errors (index out of bounds, panic, etc.) and unexpected behavior (arithmetic overflow), as well as user-defined assertions.

Another tool is *Miri*[7] [56] a Rust interpreter, which can detect if code triggers undefined behavior. It has been used to find bugs in the Rust standard library. *MIRAI*[8] like Miri is an interpreter for the mid-level intermediate representation (MIR). It is intended to be used as a static analysis tool for Rust, i.e., to verify correctness properties without having to do anything extra.

Verus [63] is another tool for verifying correctness. It uses static checking to validate user annotations. Verus supports an ever-increasing subset of Rust, improving on feedback from the Rust type checker.

The frameworks we have presented, along with Hax, currently are not focusing that much on unsafe code as well as using model checking and static analysis. Thus, connecting Hax with these tools could speed up the verification process, though these tools would not help with verifying security properties, only correctness.

Verification Frameworks and Tools

Aeneas [51] has the same frontend [50] as Hax but goes to MIR instead of (T)HIR. Aeneas transforms Rust code into a pure λ-calculus and generates code in a similar selection of backends ($F^{\star}$, Rocq, L∀∃N, Viper, Why3) to Hax.

[6]`https://github.com/minirust/minirust`
[7]`https://github.com/rust-lang/miri`
[8]`https://github.com/endorlabs/MIRAI`

Gillian-Rust [5] is an approach to do end-to-end verification building on the Gillian platform [42, 67] that can reason about type safety and functional correctness of unsafe code. Gillian-Rust is also linked with Creusot to get a strong tool that can verify a lot of real-world code with minimal annotations while still allowing the use of a strong annotation system to guide the automated proofs.

Using *Verifast* [53] for Rust [39] allows one to verify unsafe code is not exhibiting undefined behavior. Verifast for Rust has also been used in formalizing the Rust standard library.

Prusti[9] [4] is a compiler plugin, which does a number of checks after the Rust compiler's type checking pass. It works for an expressive subset of safe Rust. Prusti is built on top of Viper.

For Hax, we do the verification separate from the compilation, but still after the Rust compiler's type check. Results shown by Prusti could also be shown by F* and Rocq using Hax, as the model we use requires proving, e.g., panic freedom. Adding Prusti to the Hax pipeline and ensuring obligations are generated for bugs caught with Prusti could help with showing the correctness of the Hax framework.

Viper[10] [38] is a static verification infrastructure based on separation logic. It allows one to build automated verifiers (like Prusti). and has been used for Go, Java, Python, Rust, and many others. It generates and solves verification conditions and validates a symbolic execution using an SMT solver.

Having an intermediate language with strong and generalized tools for verification is the same philosophy as Hax. However, the goal of the intermediate language of Hax is to serve as a common input for multiple tools instead of the common target.

StableMIR[11,12] aims to become the public interface of the Rust compiler for analysis tools. It is intended to be a stable target, whereas the internal representation of the Rust compiler might fluctuate more.

The work we have done has been focusing on surface Rust or THIR. However, the goal of stabilizing an input for tools is important for ensuring multiple tools can collaborate.

Flux [64] is an implementation of liquid types for Rust. Flux adds refinement types to allow for verification of imperative safe Rust.

In Hax we achieve something similar by adding refinement annotations, which get translated by the backends; thus, the refinements are handled by the proof assistants. However, this is very limited, whereas full liquid types can push feedback earlier in the pipeline while possibly simplifying the readability over using annotations.

[9] `http://prusti.org`
[10] `https://www.pm.inf.ethz.ch/research/viper.html`
[11] `https://github.com/rust-lang/project-stable-mir`
[12] `https://hackmd.io/jBRkZLqAQL2EVgwIIeNMHg`

7.2 Cryptographic Proof Frameworks

We have used F*, ProVerif and SSProve to verify security properties, however, other tools exists. We will present a few of these here.

Tamarin [9] is a symbolic verification tool that can find attacks on security protocols or show their absence. The tool has been successfully applied to prove properties about large protocols like TLS 1.3, 5G-AKA, Noise, EMV, and Apple iMessage.

Squirrel [6] is another proof assistant dedicated to cryptographic protocols. It follows the computationally complete symbolic attacker approach, i.e., it uses the symbolic model; however, it gives computational guarantees.

DY* [13] is a tool for doing symbolic security analysis of protocols written in F*. It can reason about unbounded loops, mutable recursive data structures, and low-level details.

EasyCrypt [8] is another proof assistant for doing cryptographic proofs in the computational model. It has many tools for doing cryptographic reasoning but lacks tools for doing general math, though efforts are being made to fix this. SSProve benefit from the extensive mathematical libraries already part of Rocq.

7.3 Crypto Proof Ladders

The Crypto Proof Ladders project[13] is an introduction to the use of formal methods for cryptography. This is done by building a collection of tutorials and examples in a wide range of tools. Another goal of the project is to find possible gaps and places to improve and allow cross community collaboration.

[13]`https://proof-ladders.github.io/`

Chapter 8

Discussion

In Figure 8.1, we have made an overview of how the three frameworks (primitive, protocol, and smart contract) look together. This is a merge of figures from the papers, so for more details about the parts, see the relevant papers (i.e. Figure 1 in §3.2, Figure 1 in §4.3, Figure 2 in §5.2, and Figure 1 in §6.4).

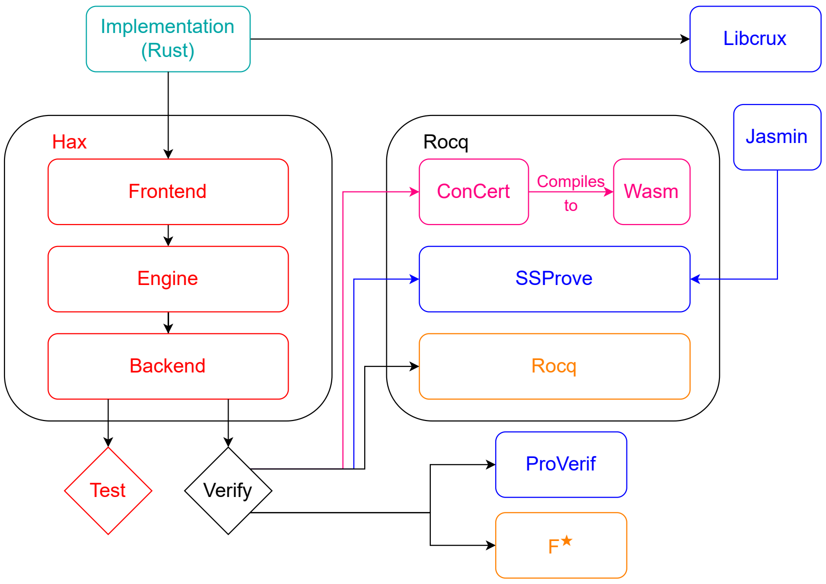


Figure 8.1: Combination of frameworks

In Chapter 3, we introduce the Hax framework (the red part of Figure 8.1), which enables us to translate primitives, protocols, and smart contracts written in Rust into a selection of backends.

In Chapter 4, we show how to show security and correctness of cryptographic primitives (the blue part of Figure 8.1 related to SSProve). We do this by writing an efficient implementation of the primitive in the Hax subset of Rust or in Jasmin and then translating them into SSProve. In SSProve, we can show that the implementation adheres to a specification written in Hacspec and again translated using Hax.

In Chapter 5, we show how to use multiple tools to verify different parts of a protocol, both for security guarantees (the blue part of Figure 8.1)and runtime safety (the orange part of Figure 8.1).

Finally, in Chapter 6, we show how to verify security (the blue part of Figure 8.1)and protocol correctness (the magenta part of Figure 8.1)of smart contracts written in Rust.

Continuing this process, we can extend the tools and frameworks connected to Hax to allow for more verification and automation. The work uses SSProve as state-separating proofs (SSP), which helps with the modularity of the verification process; however, the framework for doing the verification is general, and the concrete tools could be replaced.

An important part of the verification done is that we are doing both formally secure and also efficient implementations. We are not working on toy examples but on realistic real-world examples. Ensuring that the Jasmin implementation of AES actually adheres to the NIST specification using Intel instructions allows us to plug in the efficient implementation instead of using an AES implementation written in Rust. The use of TLS is also ever present, and having a secure implementation is important. Furthermore, we ensure the implementation can actually run a small internet of things (IoT) device using efficient primitives. The implementation and its security and correctness proofs are modular and extensible; thus, adding features does not require a re-formalization. The formalization of the paper proof for the key schedule theorem does also have separate interest, as it can (partially) be reused for other projects like MLS. Lastly, the formalization of the OVN protocol makes use of common components that can be reused for formalization of a full election protocol like ElectionGuard (EG). The protocol itself is also of use for doing boardroom elections, and it could be tweaked to work for larger elections. Showing that we can ensure the security and correctness of not only a voting protocol but also a smart contract can be used as an argument for having stricter requirements for such crucial software.

The work of this thesis helps to showcase the possibility to formalize efficient primitives, write executable specifications, and prove the security, correctness, and panic freedom of protocols. While we have been building these frameworks and formalizing the example, other projects have pushed the semantics of Rust along with tools for ensuring correctness of unsafe Rust code. We should incorporate these tools and theories into the frameworks to allow for verification of more advanced primitives and protocols and the use of external components. Furthermore, applying the frameworks to validate more protocols is important to further push verified software and executable specifications into standards and cryptographic libraries. Our work on TLS has been acknowledged in Project Everest [14, 88], showing that these formalizations do have an impact on the community.

Another direction we have pushed, in this work, is to use Hax as a collaborative tool, which allows multiple tools to contribute their individual strengths to the formalization of a common implementation. Making it easier to extend Hax by building a common proof language and/or standard library (like the annotated core) further encourages other tools to hook into Hax. This also aligns very well with the Crypto Proof Ladders, which was started at the latest HACS workshop.

Chapter 9

Future Work

In this chapter we will summarize some of the possibilities for future work observed paper sections. We will also give a broader description of future projects and what tools or techniques remain to be improved to make high assurance cryptographic software (HACS) the standard and not something only for very critical code.

9.1 Extensions to the Work of the Papers

Increasing the trust in Hax as a tool by using more static analysis, testing, verification and projects, both for internal code, but also to validate the translation. Stabilizing the backends and ensuring they all have (large) verification libraries for common libraries especially the core and standard libraries of Rust.

Expanding the example libraries of Hacspec and Libcrux, to allow for verification of more protocols using efficient primitives.

9.2 Larger Trends and the Field as a Whole

A general trend in the formalization community is not formally verifying their own formal verification tool. This usually boils down to getting a usable product faster, allows for earlier verification, which is the goal of the tool. However, this is usually the same argument everyone else use, for now using formal methods. Thus, we should employ all the tools and methods on the code bases we are using for the development as well.

I see many very useful tools being developed for Rust. However, combining these tools are not always trivial. The oxide project tries to classify surface Rust into a hierarchy of abstractions, however, not all tools fit into one of these, so having more fine grained definitions could also be very useful.

Especially for formalization of cryptographic primitives and protocols there have been many tools developed for solving specific problems. Having a way to dispatch the goals to these tools as part of a larger proof effort would be very useful. However, this requires not using contradicting assumptions, such that composing the proofs does not foundationally break the axiomatic system. A step towards this goal is the efforts of the cryptographic proof ladder, which tries to understand how different tools solve different tasks by making a large collection of example solutions to cryptogrpahic problems. The next step is to figure out how the tools can collaborate. This could require defining common languages or interfaces between tool.

Bibliography

[1] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards Certified Meta-Programming with Typed Template-Coq. In *LNCS*, volume 10895, pages 20–39. Springer, July 2018. `doi:10.1007/978-3-319-94821-8_2`. (pp. 47 and 77).

[2] Carlo Angiuli and Daniel Gratzer. *Principles of Dependent Type Theory*. `https://www.danielgratzer.com/papers/type-theory-book.pdf` or `https://carloangiuli.com/papers/type-theory-book.pdf`, 2025. (pp. 9 and 10).

[3] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. ConCert: A Smart Contract Certification Framework in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, January 2020. `doi:10.1145/3372885.3373829`. (p. 5).

[4] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging Rust Types for Modular Specification and Verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 3, pages 147:1–147:30, 2019. `doi:10.5281/zenodo.3363914`. (p. 166).

[5] Sacha-Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. A Hybrid Approach to Semi-automated Rust Verification. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025. `doi:10.1145/3729289`. (p. 166).

[6] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Joseph Lallemand. The Squirrel Prover and its Logic. *ACM SIGLOG News*, 11(2):62–83, 2024. `doi:10.1145/3665453.3665461`. (p. 167).

[7] Manuel Barbosa, Karthikeyan Bhargavan, Franziskus Kiefer, Peter Schwabe, Pierre-Yves Strub, and Bas E. Westerbaan. Formal Specifications for Certifiable Cryptography. In *NIST Workshop on Formal Methods within Certification Programs (FMCP 2024)*, 2024. (pp. 20 and 53).

[8] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume

8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013. doi:10.1007/978-3-319-10082-1_6. (p. 167).

[9] David Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. Tamarin: Verification of Large-Scale, Real-World, Cryptographic Protocols. *IEEE Security & Privacy*, 20(3):24–32, 2022. doi:10.1109/MSEC.2022.3154689. (pp. 5 and 167).

[10] Andrej Bauer. Realizability as Connection between Constructive and Computable Mathematics. In Tanja Grubba, Peter Hertling, Hideki Tsuiki, and Klaus Weihrauch, editors, *CCA 2005 - Second International Conference on Computability and Complexity in Analysis, August 25-29, 2005, Kyoto, Japan*, volume 326-7/2005 of *Informatik Berichte*, pages 378–379. FernUniversität Hagen, Germany, 2005. (p. 78).

[11] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, page 62–73, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/168588.168596. (p. 13).

[12] Jean-Philippe Bernardy and Marc Lasson. Realizability and Parametricity in Pure Type Systems. In *Foundations of Software Science and Computational Structures*, pages 108–122, 2011. doi:10.1007/978-3-642-19805-2_8. (p. 78).

[13] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 523–542. IEEE Computer Society, 2021. doi:10.1109/EuroSP51992.2021.00042. (p. 167).

[14] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Cătălin Hriţcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pang, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *2nd Summit on Advances in Programming Languages*, May 2017. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7119/pdf/LIPIcs-SNAPL-2017-1.pdf. (pp. 6, 81, and 170).

[15] Karthikeyan Bhargavan, Maxime Buyse, Lucas Franceschino, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. hax: Verifying Security-Critical Rust Software Using Multiple Provers. In Jonathan Protzenko and Azalea Raad, editors, *Verified Software. Theories, Tools and*

Experiments, pages 96–119, Cham, 2025. Springer Nature Switzerland. `doi:10.1007/978-3-031-86695-1_7`. (pp. 19 and 21).

[16] Karthikeyan Bhargavan, Lucas Franceschino, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. Hax - Enabling High Assurance Cryptographic Software. Talk given at RustVerify'24, 2024. Extended abstract at `https://github.com/hacspec/hacspec.github.io/blob/master/RustVerify24.pdf`. (p. 21).

[17] Karthikeyan Bhargavan, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. Formal Security and Functional Verification of Cryptographic Protocol Implementations in Rust. Cryptology ePrint Archive, Paper 2025/980, 2025. URL: `https://eprint.iacr.org/2025/980`. (p. 81).

[18] C. Brzuska, Marc Fischlin, Nigel P. Smart, Bogdan Warinschi, and Stephen C. Williams. Less is More: Relaxed yet Composable Security Notions for Key Exchange. In *International Journal of Information Security*, volume 12, pages 267–297, 2013. `doi:10.1007/s10207-013-0192-y`. (p. 148).

[19] Chris Brzuska, Antoine Delignat-Lavaud, Christoph Egger, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. Key-Schedule Security for the TLS 1.3 Standard. In *Advances in Cryptology – ASIACRYPT 2022*, page 621–650, Berlin, Heidelberg, 2022. Springer-Verlag. `doi:10.1007/978-3-031-22963-3_21`. (p. 100).

[20] Chris Brzuska, Antoine Delignat-Lavaud, Christoph Egger, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. Key-schedule Security for the TLS 1.3 Standard. Cryptology ePrint Archive, Paper 2021/467, 2021. URL: `https://eprint.iacr.org/2021/467`. (pp. 100, 105, and 106).

[21] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State Separation for Code-Based Game-Playing Proofs. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, pages 222–249. Springer International Publishing, 2018. `doi:10.1007/978-3-030-03332-3_9`. (pp. 11, 12, 100, and 148).

[22] Tej Chajed. Record Updates in Coq. In *CoqPL'21*, 2021. URL: `https://popl21.sigplan.org/details/CoqPL-2021-papers/3/Record-Updates-in-Coq`. (pp. 46 and 77).

[23] David Chaum and Torben Pryds Pedersen. Wallet Databases with Observers. In Ernest F. Brickell, editor, *Advances in Cryptology — CRYPTO' 92*, pages 89–105, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. `doi:10.1007/3-540-48071-4_7`. (p. 118).

[24] Alonzo Church. A Note on the Entscheidungsproblem. *J. Symb. Log.*, 1(1):40–41, 1936. doi:10.2307/2269326. (p. 9).

[25] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58:345–363, 1936. doi:10.2307/2268571. (pp.).

[26] Alonzo Church. Correction to *A Note on the Entscheidungsproblem. J. Symb. Log.*, 1(3):101–102, 1936. doi:10.2307/2269030. (p. 9).

[27] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/351240.351266. (pp. 5 and 50).

[28] Cyril Cohen, Enzo Crance, and Assia Mahboubi. Trocq: Proof Transfer for Free, With or Without Univalence. In Stéphanie Weirich, editor, *Lecture Notes in Computer Science*, volume 14576 of *Lecture Notes in Computer Science*, pages 239–268, Luxembourg, Luxembourg, April 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-57262-3_10. (p. 77).

[29] Cyril Cohen, Enzo Crance, and Assia Mahboubi. Trocq: Proof Transfer for Free, Beyond Equivalence and Univalence. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, May 2025. doi:10.1145/3737283. (p. 77).

[30] Joshua M. Cohen and Philip Johnson-Freyd. A Formalization of Core Why3 in Coq. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi:10.1145/3632902. (p. 49).

[31] Thierry Coquand and Gérard Huet. Constructions: A Higher Order Proof System for Mechanizing Mathematics. In Bruno Buchberger, editor, *EUROCAL '85*, pages 151–184, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg. doi:10.1007/3-540-15983-5_13. (p. 11).

[32] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2):95–120, 1988. doi:10.1016/0890-5401(88)90005-3. (p. 10).

[33] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols. In Yvo Desmedt, editor, *Advances in Cryptology — CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. doi:10.1007/3-540-48658-5_19. (p. 118).

[34] H. B. Curry. Functionality in Combinatory Logic*. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934. doi:10.1073/pnas.20.11.584. (p. 9).

[35] Haskell Brooks Curry and Robert M. Feys. *Combinatory Logic Vol. 1.* North-Holland Publishing Company, Amsterdam, Netherlands, 1958. (p. 9).

[36] Ivan Damgård. On Σ-protocols, 2010. URL: https://www.cs.au.dk/~ivan/Sigma.pdf. (p. 115).

[37] Danny Dolev and Andrew Chi-Chih Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983. doi:10.1109/TIT.1983.1056650. (pp. 11 and 12).

[38] Marco Eilers, Malte Schwerhoff, Alexander J. Summers, and Peter Müller. Fifteen Years of Viper. In Ruzica Piskac and Zvonimir Rakamarić, editors, *Computer Aided Verification (CAV)*, pages 107–123, Cham, 2025. Springer Nature Switzerland. doi:10.1007/978-3-031-98668-0_5. (p. 166).

[39] Nima Rahimi Foroushaani and Bart Jacobs. Modular Formal Verification of Rust Programs with Unsafe Blocks, 2022. arXiv:2212.12976. (pp. 164 and 166).

[40] Abraham A. Fraenkel. Zu den Grundlagen der Cantor-Zermeloschen Mengenlehre. *Mathematische Annalen*, 86:230–237, 1922. doi:10.1007/BF01457986. (p. 10).

[41] Abraham Adolf Fraenkel, Yehôsua' Bar-Hillel, and Azriel Levy. *Foundations of set theory, 2nd Edition*, volume 67 of *Studies in logic and the foundations of mathematics*. North-Holland Publ., 1973. URL: https://www.worldcat.org/oclc/185773710. (p. 10).

[42] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, Part I: A Multi-language Platform for Symbolic Execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 927–942, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3386014. (p. 166).

[43] Sebastian Graf. Verifying imperative programs using mvcgen. URL: https://hackmd.io/@sg-fro/BJRlurP_xg. (p. 78).

[44] Niklas Grimm, Kenji Maillard, Cédric Fournet, Cătălin Hriţcu, Matteo Maffei, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, and Santiago Zanella-Béguelin. A Monadic Framework for Relational Verification: Applied to Information Security, Program Equivalence, and Optimizations. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 130–145, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3167090. (p. 78).

[45] Rust Team Types Working Group. A-mir-formality. URL: `https://github.com/rust-lang/a-mir-formality`. (p. 46).

[46] The Rust Team Traits Working Group. *The Chalk book*. The Rust Team, 2025. URL: `https://doc.rust-lang.org/reference/`. (p. 46).

[47] Shay Gueron. White Paper: Intel® Advanced Encryption Standard (AES) New Instructions Set, 2012. URL: `https://www.intel.com/content/www/us/en/developer/articles/tool/intel-advanced-encryption-standard-aes-instructions-set.html`. (p. 60).

[48] Philipp G. Haselwarter, Benjamin Salling Hvass, Lasse Letager Hansen, Théo Winterhalter, Catalin Hritcu, and Bas Spitters. The Last Yard: Foundational End-to-End Verification of High-Speed Cryptography. In *CCP'24*, pages 30–44. ACM, 2024. `doi:10.1145/3636501.3636961`. (p. 61).

[49] Arend Heyting. *Intuitionism: An introduction, Studies in Logic and the Foundations of Mathematics*. North-Holland, 1956, 1971. (p. 9).

[50] Son Ho, Guillaume Boisseau, Lucas Franceschino, Yoann Prak, Aymeric Fromherz, and Jonathan Protzenko. Charon: An Analysis Framework for Rust, 2025. `arXiv:2410.18042`. (p. 165).

[51] Son Ho and Jonathan Protzenko. Aeneas: Rust Verification by Functional Translation. *Proc. ACM Program. Lang.*, 6(ICFP), 2022. `doi:10.1145/3547647`. (pp. 5 and 165).

[52] William Alvin Howard. The Formulae-as-Types Notion of Construction. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980. (p. 9).

[53] Bart Jacobs, Jan Smans, and Frank Piessens. The VeriFast Program Verifier: A Tutorial, August 2024. `doi:10.5281/zenodo.13380705`. (pp. 5, 164, and 166).

[54] Thomas Jech. *Set Theory: The Third Millennium Edition, revised and expanded*. Springer Monographs in Mathematics. Springer, Berlin, 3rd edition, 2003. (p. 10).

[55] Ralf Jung. Announcing: MiniRust, 2022. URL: `https://www.ralfj.de/blog/2022/08/08/minirust.html`. (p. 165).

[56] Ralf Jung. Miri: Practical undefined behavior detection for rust (keynote). In *Proceedings of the 19th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems*, ICOOOLPS 2024, page 1, New York, NY, USA, 2024. Association for Computing Machinery. `doi:10.1145/3679005.3695733`. (p. 165).

[57] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158154. (p. 163).

[58] Franziskus Kiefer, Karthikeyan Bhargavan, Bas Spitters, and Manuel Barbosa. HACSPEC: a gateway to high-assurance cryptography. Talk given at RWC 2023, 2023. Video at https://youtu.be/lahO3de3k_0?t=7, extended abstract at https://github.com/hacspec/hacspec/blob/master/rwc2023-abstract.pdf. (pp. 21 and 53).

[59] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018. (p. 13).

[60] Steve Klabnik, Carol Nichols, and Chris Krycho. *The Rust Programming Language*. The Rust Project Developers, 2025. URL: https://doc.rust-lang.org/book/. (p. 13).

[61] S. C. Kleene. On the Interpretation of Intuitionistic Number Theory. *Journal of Symbolic Logic*, 10(4):109–124, 1945. doi:10.2307/2269016. (p. 78).

[62] Andrey Nikolaevich Kolmogoroff. Zur Deutung der intuitionistischen Logik. *Mathematische Zeitschrift*, 1932. doi:10.1007/BF01186549. (p. 9).

[63] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023. doi:10.1145/3586037. (pp. 5 and 165).

[64] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi:10.1145/3591283. (pp. 164 and 166).

[65] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France, January 2016. SEE. URL: https://inria.hal.science/hal-01238879. (p. 5).

[66] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, September 2022. doi:10.5281/zenodo.7118596. (p. 77).

[67] Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, Part II: Real-World Verification for JavaScript and C. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 827–850, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-81688-9_38. (p. 166).

[68] Per Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. Elsevier, 1975. doi:10.1016/S0049-237X(08)71945-1. (p. 10).

[69] Nicholas D. Matsakis and Felix S. Klock. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2663171.2663188. (p. 13).

[70] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. Rusthornbelt: a semantic foundation for functional verification of rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 841–856, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519939.3523704. (p. 163).

[71] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. RustHorn: CHC-based Verification for Rust Programs. *ACM Trans. Program. Lang. Syst.*, 43(4), October 2021. doi:10.1145/3462205. (p. 163).

[72] Ueli Maurer. Abstract Models of Computation in Cryptography. In Nigel P. Smart, editor, *Cryptography and Coding*, pages 1–12, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11586821_1. (p. 11).

[73] Ueli Maurer. Constructive Cryptography - A New Paradigm for Security Definitions and Proofs. In Sebastian Mödersheim and Catuscia Palamidessi, editors, *Theory of Security and Applications*, volume 6993 of *Lecture Notes in Computer Science*, pages 33–56, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-27375-9_3. (pp. 11 and 12).

[74] Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan. hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust. Technical report, Inria, March 2021. URL: https://hal.inria.fr/hal-03176482. (pp. 20, 21, and 53).

[75] National Institute of Standards, Technology (NIST), Morris J. Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence E. Bassham, E. Roback, and James Dray Jr. Advanced Encryption Standard (AES), 2001-11-26 00:11:00 2001. doi:10.6028/NIST.FIPS.197. (pp. 54 and 56).

[76] Christine Paulin-Mohring. Inductive Definitions in the system Coq Rules and Properties. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, pages 328–345, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. doi:10.1007/BFb0037116. (p. 11).

[77] Rasmus Lerchedahl Petersen, Lars Birkedal, Aleksandar Nanevski, and Greg Morrisett. A Realizability Model for Impredicative Hoare Type Theory. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, volume 4960 of *Lecture Notes in Computer Science*, pages 337–352. Springer, 2008. `doi:10.1007/978-3-540-78739-6_26`. (p. 78).

[78] Frank Pfenning and Christine Paulin-Mohring. Inductively Defined Types in the Calculus of Constructions. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, pages 209–228, New York, NY, 1990. Springer-Verlag. `doi:10.1007/BFb0040259`. (p. 10).

[79] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166, 1998. `doi:10.1007/BFb0054170`. (p. 78).

[80] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018. `doi:10.17487/RFC8446`. (p. 81).

[81] Mike Rosulek. The Joy of Cryptography. Online textbook, 2021. URL: `http://web.engr.oregonstate.edu/~rosulekm/crypto/`. (pp. 11 and 12).

[82] Rust Project Developers. The Rust Reference, 2025. URL: `https://doc.rust-lang.org/reference/`. (pp. 13 and 15).

[83] Cole Schlesinger and Nikhil Swamy. Verification Condition Generation with the Dijkstra State Monad. Technical Report MSR-TR-2012-45, Microsoft Research, April 2012. URL: `https://www.microsoft.com/en-us/research/publication/verification-condition-generation-with-the-dijkstra-state-monad/`. (p. 78).

[84] Claus-Peter Schnorr. Efficient Signature Generation by Smart Cards. *J. Cryptol.*, 4(3):161–174, 1991. `doi:10.1007/BF00196725`. (p. 116).

[85] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure Distributed Programming with Value-Dependent Types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 266–278, New York, NY, USA, 2011. Association for Computing Machinery. `doi:10.1145/2034773.2034811`. (p. 11).

[86] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent Types and Multi-monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 256–270, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2837614.2837655`. (p. 11).

[87] Enrico Tassi. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect). In *The Fourth International Workshop on Coq for Programming Languages*, Los Angeles (CA), United States, January 2018. URL: `https://inria.hal.science/hal-01637063`. (p. 77).

[88] Project Everest Team. Project Everest: Perspectives from Developing Industrial-grade High-Assurance Software. `https://project-everest.github.io/assets/everest-perspectives-2025.pdf`, 2025. (pp. 6, 81, and 170).

[89] The Rust Team. *The Rustonomicon*. The Rust Team, 2024. URL: `https://doc.rust-lang.org/nomicon/`. (pp. 15 and 163).

[90] The F* Development Team. F$*$: A Verification-Oriented Programming Language. `https://fstar-lang.org`, 2025. (p. 11).

[91] The Rocq Team. The Rocq Prover. `https://rocq-prover.org`, 2025. Formerly known as the Coq proof assistant. (p. 11).

[92] A. S. Troelstra. *Principles of Intuitionism*. Springer Heidelberg, 1969. `doi:10.1007/BFb0080643`. (p. 9).

[93] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 42(1):230–265, 1937. `doi:10.1112/PLMS/S2-42.1.230`. (p. 9).

[94] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013. (pp. 9 and 10).

[95] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. Verifying Dynamic Trait Objects in Rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '22, pages 321–330, New York, NY, USA, 2022. Association for Computing Machinery. `doi:10.1145/3510457.3513031`. (pp. 5, 164, and 165).

[96] Aaron Weiss, Daniel Patterson, and Amal Ahmed. Rust Distilled: An Expressive Tower of Languages. *CoRR*, 2018. `arXiv:1806.02693`. (p. 164).

[97] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The Essence of Rust. *CoRR*, 2019. `arXiv:1903.00982`. (p. 164).

[98] Ernst Zermelo. Untersuchungen über die Grundlagen der Mengenlehre I. *Mathematische Annalen*, 65:261–281, 1908. `doi:10.1007/BF01449999`. (p. 10).