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

Philipp G. Haselwarter[†,1] Benjamin Salling Hvass[†,1] Lasse Letager Hansen[†,1]
Théo Winterhalter[2] Cătălin Hriţcu[3] Bas Spitters[1]

[1]Aarhus University, Denmark [2]Inria Saclay, France [3]MPI-SP, Bochum, Germany

***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 automatically proving the equivalence of 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, prove its security using SSProve, and also that it conforms to a specification of the AES standard written in Hacspec.**

1 Introduction

Research on high-assurance cryptography has recently achieved significant practical success, with formally verified cryptographic code making its way into mainstream libraries and software products [7, 13, 15, 18, 20, 32, 34, 37, 38]. Since in this area missing any bugs can have a serious security impact, the authors of some of the verification tools for cryptographic code additionally try to reduce the trusted computing base of their tools and construct foundational proofs [5, 13, 20, 23, 27, 30]. 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 in informal pseudo-code in the standards (e.g., in IETF RFCs). The Hacspec language [14, 29] 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 is understandable for both ordinary developers and cryptographers. Hacspec can be translated 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, 23]. 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 [35], 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, 26]. Moreover, even aggressively optimized C programs are sometimes not fast enough, for instance, since they cannot make use of special instructions providing hardware acceleration for cryptographic primitives (e.g., Intel AES-NI [22]). 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

[†]Equal Contribution

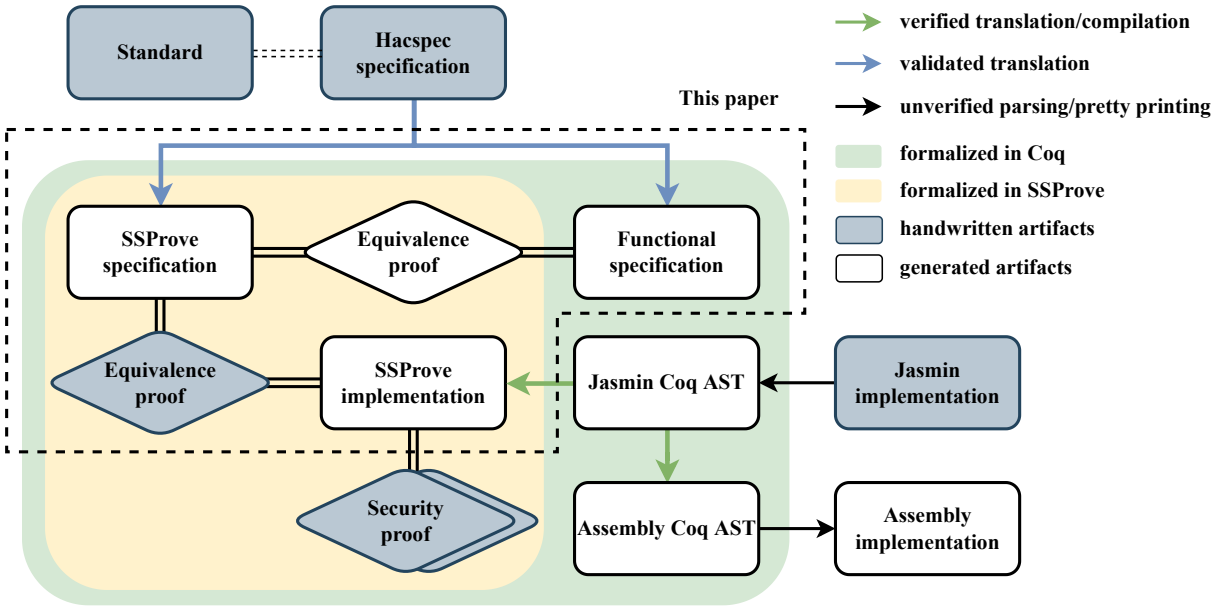

Fig. 1. Proposed workflow for foundational end-to-end verification of high-speed cryptography

not only allows us to reduce the trusted computing base, but it also facilitates reusing existing large mathematical Coq libraries [3, 28] for verifying 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, this allows us to observe and reason about the *stateful* behavior of Hacspec. One particular challenge was dealing with early returns in Rust.
- 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 is semantics-preserving with respect to 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 [22], prove its security using SSProve, and also that it conforms to a Hacspec specification of the AES standard [19].

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 more interesting 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 different ways:

[1]In fact, Hacspec is directly used in the upcoming hash-to-curve IETF standard [21] for writing a reference implementation.

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

The functional translation [33] 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 [31] to automatically construct an ***equivalence proof*** in Coq, which formally shows that the two 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 constructed 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 we proved that our translation preserves the semantics of programs. This proof is entirely mechanized in Coq, which is made possible by the fact that both SSProve and Jasmin already have formal semantics in Coq [4, 5, 6, 23]. 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, 23]; and (2) we can obtain the ***Jasmin Coq AST*** (i.e., abstract syntax tree) 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 Jasmin implementation with ***security proofs*** done in the SSProve framework.These proofs use the standard security games from the cryptographic literature.

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). While this obviously constitutes a toy example, we hope to convey intuition on the methodology. A more interesting case study for the AES encryption scheme is presented in Section 6.

2.2.1 Specification

The Hacspec specification for `xor` takes two 64-bit words as input, puts them into mutable variables, and computes the `^` operator of the Hacspec language, which implements XOR. The result is stored in a mutable variable, which is then returned. Even if using mutable variables is not the simplest possible way to write this specification, we chose it since it helps us illustrate more of our workflow.

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

We translate this 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 : raw_code 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 the two:

```
Lemma hacspec_xor_equiv : ∀ 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;
}
```

which 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 in our workflow 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 ;;
  r0 ← get r ;;
  ret r0.
```

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
    ≈
    XOR 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 two final 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 by applying the proofs of the relational program logic of SSProve [1].

2.2.5 Security proof for 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 [1] 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 real encryption scheme and an oracle: these have the same interfaces but different implementations.

Returning to the OTP example, 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.
```

The implementation for which we already have a security proof is 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 the advantage of an attacker in distinguishing between `OTP_real` and a game `OTP_ideal`, where the input is disregarded and a random message is encrypted, 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 id0 :
  JOTP_real id0 ≈₀ OTP_real.
```

where $\approx_0$ 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 adversaries `A` and 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

3.1 Hacspec

3.1.1 The Hacspec language and functional translations

Hacspec is a **H**igh **A**ssurance **C**ryptography **SPEC**ification language [14, 25, 29] aiming to provide a connection between programmers, cryptographers and proof engineers. It proposes to make future internet standards, such as those published by IETF and NIST, machine-readable. The Hacspec language was constructed as a subset of Rust (see Section A.1 for the syntax), 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.[2] Hacspec also comes with an operational semantics [29, appendix], but the functional translation is not formalized and proved sound w.r.t. the operational semantics, even if that would also be a good target for future formalization.

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. We will explain how to do so in Section 4.

3.1.2 The Hacspec library

Hacspec is aimed towards specifying cryptography, and therefore provides a builtin library that implements common functionalities needed by cryptographers. This includes:

- modular arithmetic integers;
- machine integers (`u8`, `u16`, `u32`, `u64`, `u128`);
- constant-/fixed-length arrays

These types have a special semantics in the backends. Hacspec uses constant-/fixed-length arrays that are a wrapper around the Rust vectors, restricting the use of vectors. This is to prevent the use of mutable or extensible vectors, which is a common source of confusion or bugs in specifications [29].

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 will 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].

3.2.1 The Jasmin 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 are sequences of 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 $\langle\, e | (m, \rho) \,\rangle \Downarrow_{exp} v$ 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[3] of v at type α to v' to ensure that v' is compatible with the type of the variable x. Finally, we update the local state to $\rho[x \leftarrow v']$, while the global state remains unchanged.

The main subtlety for translating Jasmin to SSProve will arise 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,

[2] 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 [25].

[3] 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.

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

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

CALLRUN

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

Fig. 2. Jasmin operational semantics of selected instructions

when switching from caller to callee, we *retain the local state* ρ_0 and pass only the global state m to CALLRUN. This is witnessed by the use of an auxiliary relation $\Downarrow_{call}$ between 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. We again assume here that the number of arguments supplied to f matches the number of parameters. Starting from an *empty local state* ρ_0, 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 $w_1,\dots,w_n$ 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.

3.3 SSProve

SSProve is a Coq library for modular cryptographic proofs introduced by Abate et al. [1]. Here we only review the concepts needed to understand the current paper. More details can be found in the extended version of the SSProve paper [23].

3.3.1 Code

In this paper, we will 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 two codes `u : raw_code X` and `k : X → raw_code A` using the bind operator that we write `x ← u ;; k x`. We will still mention that it is 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` [28, §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, say, Jasmin's function local state. This means that when generating code one must take care not to produce overlaps by ensuring disjoint locations. We will address this point when talking about the translation from Jasmin to SSProve 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 [16]. Since our methodology allows us to reuse existing security proofs [23], 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, one package can be linked to another one 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 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 wish to 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). In the case of 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 proves a number of rules for this logic and provides tactics to facilitate writing proofs. Moreover, one can also fall back on the semantics above to prove judgments as described by Haselwarter et al. [23].

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 Hacspec 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. This allows us to compare the imperative and pure translations using SSProve's relational logic. We use this to automatically generate a proof that the two translations return the same values.

4.1 The pure translation

The pure translation constitutes a minor modification of Hacspec's existing Coq backend [33], which 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 [26]. 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. As a result of these two features, we need fixed points that respect 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 all values passed to function calls or operators, before evaluating the function or operator. The evaluation 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 a limited form of control operator via early return statements. A Hacspec statement of the form `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, which we can readily encode in SSProve using sum types. 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 bind_code_option (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 the following Hacspec code:

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

Is translated to the following SSProve code:

```
temp_x ← f(v) ;;
bind_code_option temp_x (fun 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 then 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. This way the translation looks close to the original specification and can be made more readable by the notation engine in Coq.

5 Jasmin & SSProve

5.1 Memory

A major difference between the Jasmin and SSProve semantics, is how memory is handled. While SSProve only has a global notion of memory, Jasmin supports both global and local variables: global variables corresponding to pointers into memory and local variables representing values stored on the stack frame of a single function call. To model local variables in SSProve, we parameterize all translated code over a "process ID" which reserves a (*a priori* unbounded) region of SSProve's global memory for local variables. Then instantiating code with a process ID correctly assigns new process IDs to all its called functions. In particular, we prove that variables translated with different process 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 `MEM`.

5.2 Program translations

In the following section we describe our translation from Jasmin to SSProve. In order to translate programs, we need to consider the translation of types and values, expressions and commands. Before we can start the translation proper, 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 word and array types. Following Jasmin, we used the `coqword` library for a type of words, which is based on the Mathematical Components (mathcomp) library [28]. We represent arrays as maps from integers to bytes. The only minor difference is our implementation of maps differs from Jasmin. The benefit of using similar types made 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 we had to be careful to 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 transportation. We denote the function taking Jasmin expressions to SSProve code by `translate_pexpr`.

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 process ID, such that there are no collisions between local variables in separate function calls.

Note that we currently do not translate Jasmin while loops, as they do not have a correspondent in SSProve. The extended version of the SSProve paper mentions adding general recursion as future work [23].

5.2.4 Programs

We translate Jasmin programs, which map function names to function declarations (see 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 `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 $\texttt{ret}\ v$:

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

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. [23].

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 $\texttt{MEM}$ on the SSProve heap:

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

To relate the local memory of Jasmin and our implementation of local memory in SSProve, we define a relation between a variable map ρ and a heap h relative to a ***process ID*** ι. Process IDs are used to split the heap, i.e., SSProve state, into disjoint regions of memory, which can be used when a fresh piece of memory is needed for a function call. We denote by $h[x]^\iota$ the lookup of the variable x on the heap relative to the ID ι. A variable map ρ is related to the heap h w.r.t. ι if successfully looking up a variable in ρ implies that looking up the same variable 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 certain process can spawn arbitrarily many sub-processes and not run out of space on the heap. To state this we need some terminology.

We will say that a process 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 will assume that we have a prefix order, $\preceq$, on process IDs and say that a process 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 will say that two IDs s_1 and s_2 are ***disjoint***, when there is no ID which they are both a prefix of (i.e., for all IDs s not both $s_1 \preceq s$ and $s_2 \preceq s$ holds). We assume that if s_1 and s_2 are disjoint then for all locations x, y, $h[y \leftarrow v]^{s_2}[x]^{s_1} = h[x]^{s_1}$, i.e., storing at a disjoint ID locations preserves values.

For a variable map ρ, two process IDs ι, σ (main and *sub*-ID) and a set I of IDs we will say that the tuple (ρ, ι, σ, I) is a ***stack frame***. We will say that a stack frame (ρ, ι, σ, I) is ***valid*** w.r.t. a heap h when the following conditions hold:

- σ is valid w.r.t. h,
- $\rho \sim_\iota h$,
- $\sigma \notin I$,
- for all $\sigma' \in I$, $\iota \prec \sigma'$, σ' is disjoint from σ and σ' is valid w.r.t. h,
- 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 ι and σ should be a valid process ID, from which the process can spawn new processes 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.

We then define a ***stack*** simply as a list of stack frames. The empty stack is denoted by S_0. A stack frame (ρ, ι, σ, I) is said to be ***disjoint*** from a stack S when ι is disjoint from all sub IDs and IDs occurring in sets of the stack frames on S. We define a stack S to be ***valid*** w.r.t. a heap h when either S is empty or $S = F :: S'$ for some valid stack S' and some valid stack frame F which is disjoint from S' (here $F :: S'$ denotes pushing F to the stack 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 $(m, \rho) \sim_S h$, if the following conditions hold:

- S is valid w.r.t. h,
- $m \sim h$,
- ρ is the variable map of the top of the stack, i.e., the top of the stack is equal to (ρ, ι, σ, I) for some process IDs ι, σ and set 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):

If

$$(m, \rho_2) \sim_{(\rho_2,\iota_2,\sigma_2,I_2)::(\rho_1,\iota_1,\sigma_1,I_1)::S} h$$

then

$$(m, \rho_1) \sim_{(\rho_1,\iota_1,\sigma_1,I_1)::S} h.$$

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 subprocess ID of the calling stack frame, σ, is updated to a fresh ID and that we initialize processes with the same main and sub ID.

Using this relation, we can prove 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 theorem:

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

$$\begin{array}{ll} \vdash & \{h.\ s \sim_S h\} \\ & \texttt{translate\_pexpr}\ S\ e \\ \Downarrow & \texttt{translate\_value}\ v \\ & \{h.\ s \sim_S h\} \end{array}$$

This is the expected result, since evaluating expressions does not have memory side-effects, so the relation between Jasmin and SSProve states is preserved under expression translation.

We proved the following 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 $\langle\, f(v_1, \ldots, v_k) \mid m \,\rangle \Downarrow_{call} \langle\, (w_1, \ldots, w_n) \mid m' \,\rangle$ then

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

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

1) Calling the function at a fresh ID (σ_1) and with the translation of the arguments v evaluates to the translation of the return values r.
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).

Again this is the expected behavior: Calling a function should be able to change the global but not the local state. We have to update our sub-ID because the previous one is no longer fresh, since 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], except that we use SSProve instead.

We follow the same workflow as presented in Section 2:

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.

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 [22]. Our reference implementation in Hacspec is based on the Jasmin implementation (to simplify the equivalence proof) and the NIST standard [19].

For the security analysis, we prove indistinguishability under chosen plaintext attack (IND-CPA). As was the case in Section 2 we do not actually have to provide a security proof of the abstract encryption scheme, since such a proof, using a generic function as pseudo-random function (PRF) instead of AES, is already present in the SSProve library.

The PRF-based encryption scheme is given by the code:

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

where `kgen` is a key generation code (by uniform sampling) and `enc` 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 c.
```

Here f is the function which we assume to be a PRF and which we will instantiate with AES in our example. For all functions `f` : word → word → word we denote the game consisting of the single export `PRF_ENC f` by `PRF_real f`.

To connect to the existing proof, we have to prove that `PRF_real aes` is perfectly indistinguishable from the same scheme where `enc` has been substituted with the translated Jasmin code. For details on the security proof for PRF-based encryption see Section 2.3 of the SSProve paper [1].

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 operational semantics 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.

6.2 Specification

Next, we write intermediate specifications for the Jasmin functions. When comparing to the example in Section 2, these correspond to the pure Coq XOR function. As mentioned, we can reuse the specifications from the EasyCrypt and Jasmin tutorial [8], which simplifies this step considerably. Currently, we do not reuse the functional specification generated by the Hacspec translation, since it proved easier to write fresh ones by hand.

The reasoning for having the intermediate specification between the translated and the functional one; is that it is usually easier to remove the artefacts from the translation first (generated memory locations and generated operation specifications) and only then prove the underlying mathematical logical statements.

6.3 Equivalences for intermediate code

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

$$\begin{aligned} \vdash\ & \{(m_0, m_1).\ \phi\ (m_0, m_1)\} \\ & c\ i \\ \sim\ & \texttt{ret}\ (f\ i) \\ & \{(a_0, m_0'), (a_1, m_1').\ \phi\ (m_0', m_1') \wedge a_0 = a_1\} \end{aligned}$$

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 a context where we wish to apply the program equivalence. Note that we could have used the unary judgments from Section 5.3 if we could ignore the heap of the right-hand side.

6.4 Equivalences for translated code

Now we have to reason about the code generated by the Jasmin compiler and passed through our translation to SSProve. The general form of these equivalences is:

$$\begin{aligned} \vdash\ & \{(m_0, m_1).\ \phi\ (m_0, m_1)\} \\ & P'\ F\ id\ i \\ \sim\ & c\ i \\ & \{(a_0, m_0'), (a_1, m_1').\ \phi\ (m_0', m_1')\ \wedge\ a_0 = a_1\} \end{aligned}$$

where P' is the translated Jasmin program, i is an arbitrary input, id is a process ID (determining the locations used by the function), F is the function name in the Jasmin program and c is the intermediate code.

By proving an equivalence of this form, we can reuse it in proofs where F appears as a called function. It is therefore important that the equivalences are parametric in the id, since functions can call other functions at arbitrary ids.

Here 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 set locations 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 to 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 this into SSProve as `JENC` and use it in the following encryption scheme:

```
Definition JPRF_ENC id0 m :=
  k_val ← kgen ;;
  r ← sample uniform N ;;
  JENC id0 k_val r m
```

We will denote the game consisting of just `JPRF_ENC` by `JPRF_real`. Then we prove perfect indistinguishability between this scheme and a similar scheme `CPRF_real`, which simply uses an intermediate SSProve encryption function, `ENC`, in place of `JENC`.

To do this, we again use Theorem 1 from 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 their return values are equal. Here we prove a slight generalization of the version of Theorem 1 previously implemented in Coq. In the previous version of the theorem, 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 generalize this and 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 are no longer required to be finite.

This generalization facilitates applying the theorem to the case where 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 just give an infinite over approximation of locations used by the program. We obtain the following theorem.

```
Theorem JPRF_perf_indist id0 :
  JPRF_real id0 ≈₀ CPRF_real.
```

Now we prove that `CPRF_real` is perfectly indistinguishable from `PRF_real aes` where `aes` is the functional Coq specification of AES, i.e., we prove the theorem:

```
Theorem CPRF_perf_indist id0 :
  CPRF_real ≈₀ PRF_real aes.
```

Here we can apply the original version of Theorem 1 since we have better control over which locations are used.

Now we can use the triangle inequality for advantages similar to how we used them in Section 2 and derive that `JPRF_real` is IND-CPA with the same bounds as in the SSProve paper [1, Section 2.3].

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 [23]. In this section, we survey the closest related work to ours in this space.

CertiCrypt [9] is the earliest framework for reasoning about cryptographic code in Coq, but it is no longer maintained.

FCF [30] 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 [13] and mbedTLS [37]. 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 too, we choose to use SSProve because it is under active development, it uses the well-developed mathcomp [28] and mathcomp-analysis libraries [3], and supports modular proofs.

EasyCrypt [10, 11] is a proof assistant and verification tool specifically designed for game-based cryptographic proofs. EasyCrypt's good integration with automatic theorem provers (e.g., SMT solvers) is helpful for large proofs, even if it does come 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 the Isabelle/HOL proof assistant. 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. [34] prove the correctness of the C implementation of X25519 in TweetNaCl using VST. Protzenkoet al. [32] verify an impressive library of cryptographic code in F*. Fiat-Crypto [20] is a foundational tool that can *generate* verified efficient implementations of finite field arithmetic. These works are focused on correctness though and don't 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 [24] and Denis et al. [17].

8 Future work

Jasminify [36] is a python tool that simplifies the process of calling Jasmin code from Rust. After the compilation of a program, the Rust object file is replaced with the Jasmin object file. However, Jasminify does not come with any correctness guarantees. Above 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 agrees. For 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 [25] 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

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] C. Abate, P. G. Haselwarter, E. Rivas, A. V. Muylder, T. Winterhalter, C. Hriţcu, K. Maillard, and B. Spitters. SSProve: A foundational framework for modular cryptographic proofs in Coq. 2021.

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

[3] R. Affeldt, C. Cohen, M. Kerjean, A. Mahboubi, D. Rouhling, K. Sakaguchi, and P.-Y. Strub. mathcomp-analysis. Analysis library compatible with Mathematical Components, 2021.

[4] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub. Jasmin: High-assurance and high-speed cryptography. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2017.

[5] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P.-Y. Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[6] B. Ammanaghatta Shivakumar, G. Barthe, B. Grégoire, V. Laporte, and S. Priya. Enforcing fine-grained constant-time policies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022.

[7] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. Sok: Computer-aided cryptography. *IACR Cryptol. ePrint Arch.*, 2019, 2019.

[8] M. Barbossa, F. Dupressoir, B. Grégoire, V. Laporte, P. Strub, and T. Oliveira. Easycrypt and jasmin tutorial, 2022.

[9] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *POPL*, 2009.

[10] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *CRYPTO*. 2011.

[11] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. 2013.

[12] D. A. Basin, A. Lochbihler, and S. R. Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *J. Cryptol.*, 33(2), 2020.

[13] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium*. 2015.

[14] K. Bhargavan, F. Kiefer, and P. Strub. hacspec: Towards verifiable crypto standards. In C. Cremers and A. Lehmann, editors, *Security Standardisation Research - 4th International Conference, SSR 2018, Darmstadt, Germany, November 26-27, 2018, Proceedings*. 2018.

[15] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. T. V. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. *USENIX Security*. 2017.

[16] C. Brzuska, A. Delignat-Lavaud, C. Fournet, K. Kohbrok, and M. Kohlweiss. State separation for code-based game-playing proofs. In *ASIACRYPT*. 2018.

[17] X. Denis, J.-H. Jourdan, and C. Marché. Creusot: A foundry for the deductive verification of Rust programs. In A. Riesco and M. Zhang, editors, *Formal Methods and Software Engineering*. 2022.

[18] J. A. Donenfeld. Wireguard: Formal verification. Available at https://www.wireguard.com/formal-verification/.

[19] M. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. Bassham, E. Roback, and J. Dray. Advanced Encryption Standard (AES), 2001.

[20] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. *IEEE S&P*, 2019.

[21] A. Faz-Hernandez, S. Scott, N. Sullivan, R. S. Wahby, and C. A. Wood. Hashing to Elliptic Curves. Internet-Draft draft-irtf-cfrg-hash-to-curve-16, Internet Engineering Task Force, 2022. Work in Progress.

[22] S. Gueron. White Paper: Intel® Advanced Encryption Standard (AES) New Instructions Set, 2012.

[23] P. G. Haselwarter, E. Rivas, A. V. Muylder, T. Winterhalter, C. Abate, N. Sidorenco, C. Hriţcu, K. Maillard, and B. Spitters. SSProve: A foundational framework for modular cryptographic proofs in Coq. Cryptology ePrint Archive, Paper 2021/397, 2021. Extended version https://eprint.iacr.org/2021/397.

[24] S. Ho and J. Protzenko. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.*, 6 (ICFP), 2022.

[25] F. Kiefer, K. Bhargavan, L. Franceschino, D. Merigoux, L. L. Hansen, B. Spitters, M. Barbosa, A. Séré, and P.-Y. Strub. HACSPEC: a gateway to high-assurance cryptography. In *RWC23*, 2023.

[26] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. CompCert – a formally verified optimizing compiler. In *ERTS 2016: Embedded Real*

Time Software and Systems, 8th European Congress, 2016.

[27] A. Lochbihler, S. R. Sefidgar, D. A. Basin, and U. Maurer. Formalizing constructive cryptography using CryptHOL. In *CSF*. 2019.

[28] A. Mahboubi and E. Tassi. Mathematical components. Online book, 2021.

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

[30] A. Petcher and G. Morrisett. The foundational cryptography framework. *POST*. 2015.

[31] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *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*. 1998.

[32] J. Protzenko and B. Parno. EverCrypt cryptographic provider offers developers greater security assurances. Microsoft Research Blog, 2019.

[33] M. M. Rasmus Holdsbjerg-Larsen, Bas Spitters. A verified pipeline from a specification language to optimized, Safe Rust. CoqPL, 2022.

[34] P. Schwabe, B. Viguier, T. Weerwag, and F. Wiedijk. A Coq proof of the correctness of X25519 in TweetNaCl. In *2021 34th CSF*, 2021.

[35] L. Simon, D. Chisnall, and R. Anderson. 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, 2018.

[36] J. van Drunen. Calling Jasmin from Rust, 2021.

[37] K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher, and A. W. Appel. Verified correctness and security of mbedTLS HMAC-DRBG. In *CCS'17*. 2017.

[38] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. *CCS*. 2017.

Appendix A
Appendix

A.1 Hacspec language

```
Items i :=
  (* import modules *)
  | use u;
  (* constant variables *)
  | const x : τ = c;
  (* type aliases *)
  | type y = τ';
  (* fixed-length array declaration *)
  | array!(y, τ, c);
  (* fixed-length polynomials *)
  | poly!(y', y, a);
  (* abstract field integer declaration *)
  | field_integers!(y, c, c);
  (* function *)
  | fn f([x: (&)τ,]+) -> τ' { e }
  (* enum type *)
  | enum y { [z(τ),]+ }
  (* struct type *)
  | struct y { [f: τ,]+ }

Use path u :=
  (* sequence of nested modules*)
  | [m::]*m'

Type τ :=
  | bool | usize
  | u8 | u16 | u32 | u64 | u128 | U8
  | U16 | U32 | U64 | U128
  | i8 | i16 | i32 | i64 | i128 | I8
  | I16 | I32 | I64 | I128
  (* Unknown-length array *)
  | Seq<τ>
  (* type variable *)
  | y
  (* tuples *)
  | ([τ,]+)

Statement s :=
  (* let binding *)
  | let (mut) p (: τ) = e
  (* mutable variable reassignment *)
  | x = e
  (* if statement *)
  | if e1 { e2 } (else { e3 })
  (* for loop *)
  | for x in e1..e2 { s }
  (* sequencing *)
  | s1; s2
  (* array update *)
  | x[e1] = e2

Expression e :=
  (* literal *)
  | l
  (* variable *)
  | (u::)x
  (* function call *)
  | (u::)(y::)f([(&)e]+);
  (* method call *)
  | e.f([e']+)
  (* tuple *)
  | ([e])
  (* tuple member access *)
  | e1.n
  (* range *)
  | e1..e2
  (* arithmetic operations *)
  | e1 op e2
  (* unary op *)
  | unop e
  (* array indexing *)
  | x[e]
  (* Unsafe integer casting *)
  | e1 as e2

Operator op := + | - | * | / | ^ | &&
  | || | & | | | % | >> | <<
  | == | != | <= | >=
Unary operator unop := ~ | ! | -
```

```
Pattern p :=
  | x
  (* tuple destructing *)
  | ([p,]*)
  (* wildcard *)
  | _
```