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 Hrițcu[3] Bas Spitters[1]

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

January 15

[†] Equal Contributions

Specification gap

- current standards use informal pseudo-code.

Implementation gap

- unoptimized or unverified compilers
- cryptographic primitives are often implemented directly in assembly

We therefore want

> *a unified foundational framework for*
> *end-to-end formal verification of*
> *efficient cryptographic implementations*

thus

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

Hacspec

Outline

Hacspec

Background - What is it?

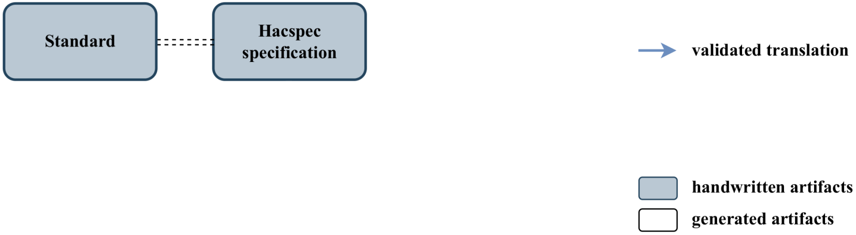
The High Assurance Cryptography SPECification (Hacspec) language

- provides a shared language
- makes internet standards (e.g. IETF and NIST) machine-readable.
- is a simple subset of Rust
- has translations ⚙ to proof assistants (e.g. Coq, EasyCrypt, or F*)

Starting from an official *standard* (e.g. NIST or IETF) produce Hacspec and generate

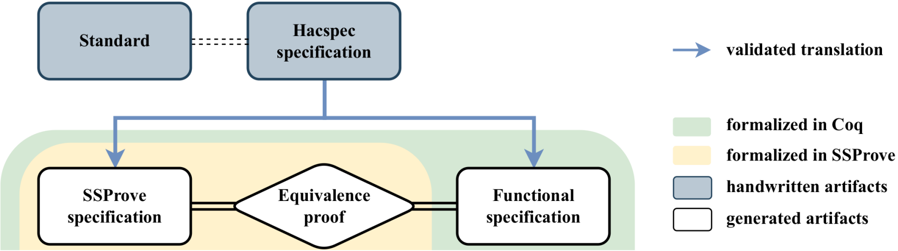
- *functional specification*
  - easier to define and prove properties
- *SSProve specification*
  - closer to efficient implementation
- Translation validation
  - build a proof of equality

Hacspec
Workflow



Hacspec

Workflow



SSProve

SSProve:

- is a foundational framework for modular cryptographic proofs in Coq
- essentially embeds a stateful language inside Coq
- the Dijkstra monad framework gives us a program logic to reason about this embedded language

We use the relational Hoare logic of SSProve for

- equivalence proof: implementation $\approx$ specification
- security proofs about specification

We build a modular syntactic translation ⚙ as triples

- the functional specification
- the SSProve specification
- a proof of equality between them

This can be seen as a binary logical relation

Equivalence between the Hacspec translations

An example of such triple

- Hacspec: `let x := y; k`

becomes

- Coq: `let x_fun := y_fun in k_fun`
- SSProve: `x_imp ← y_imp ;; k_imp`
- Equality proof: `ssprove_bind`

Other examples are

- loops, mutable let bindings, early returns, operator calls, lifting pure values, etc.

Jasmin

- is a low-level language designed for implementing high-speed cryptography,
- has a compiler implemented and verified in Coq supporting x86 and ARM
- has a formal big-step operational semantics in Coq.

Jasmin

Background

Jasmin

- is an imperative language with structured control flow
  - loops, conditionals, and procedure calls.
- has types for
  - booleans, integers, bit-words of various sizes, and arrays.
- compiler produces predictable assembly code

Jasmin

Workflow: Jasmin $\rightarrow$ SSProve

From Jasmin we get

- assembly implementation (from Jasmin compiler)

for which we

- pretty-print the internal AST (de-extracting) to Coq syntax.
- Jasmin Coq AST $\Rightarrow$ SSProve implementation
- get a mechanized proof that semantics are preserved

The main theorem, connecting *function calls* in Jasmin and in SSProve states that:

- if $f(\vec{v}) \rightsquigarrow \vec{w}$
- then $trans(f)(trans(\vec{v})) \rightsquigarrow trans(\vec{w})$

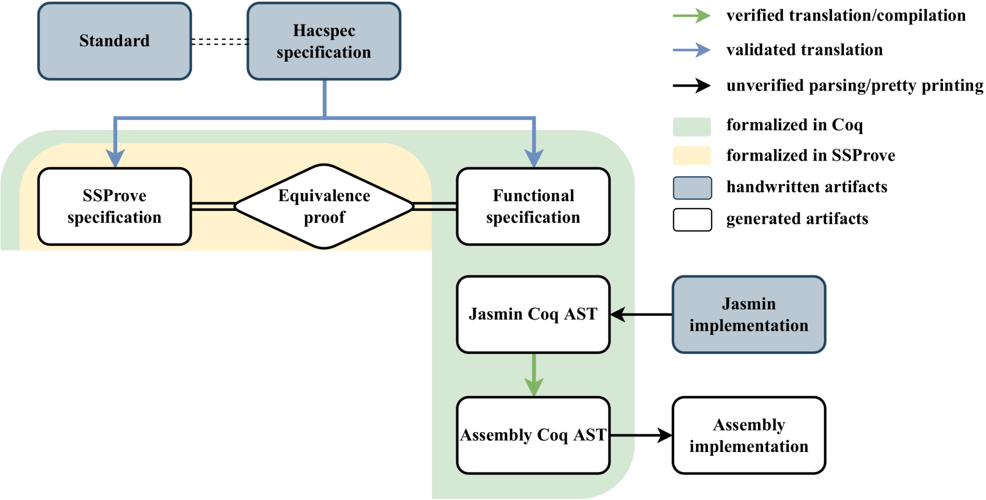
the translation modifies memory in an equivalent manner. Combined with

- the correctness theorems of the Jasmin compiler

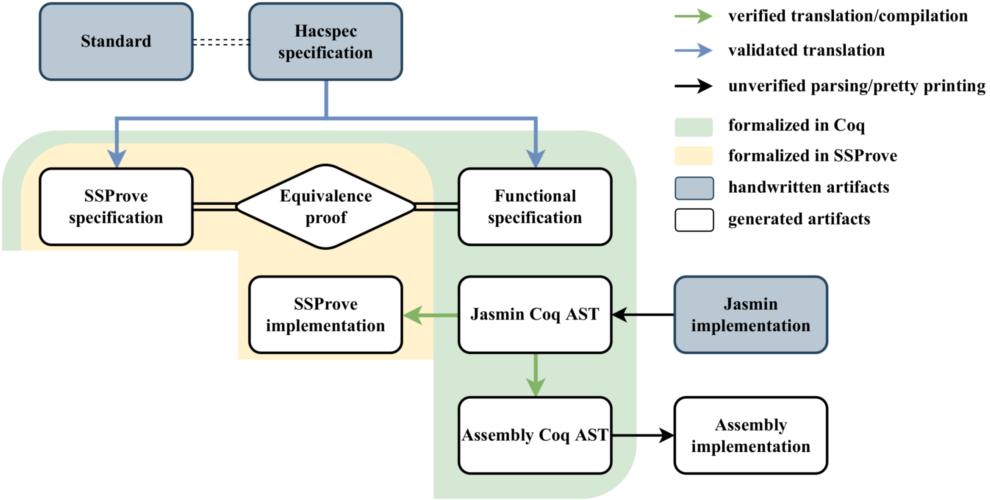
allows us to prove properties about Jasmin in SSProve

Workflow

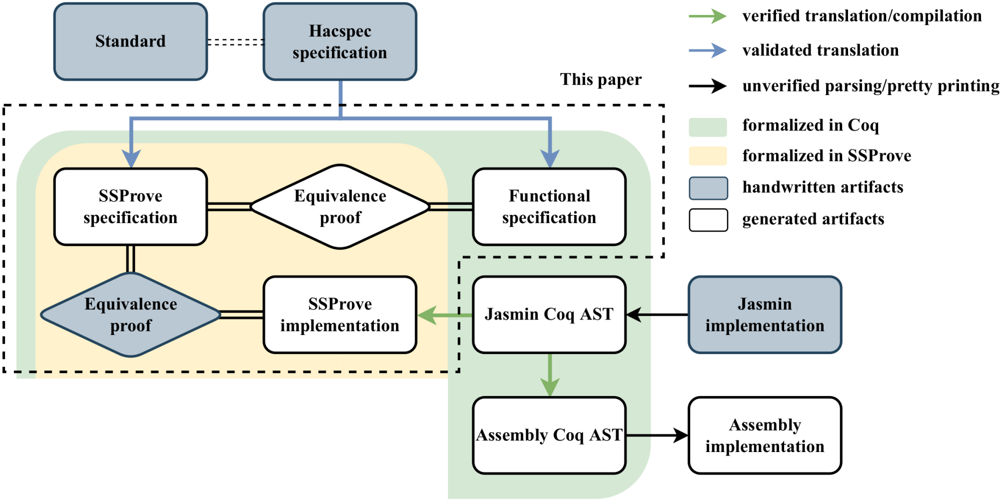
Jasmin

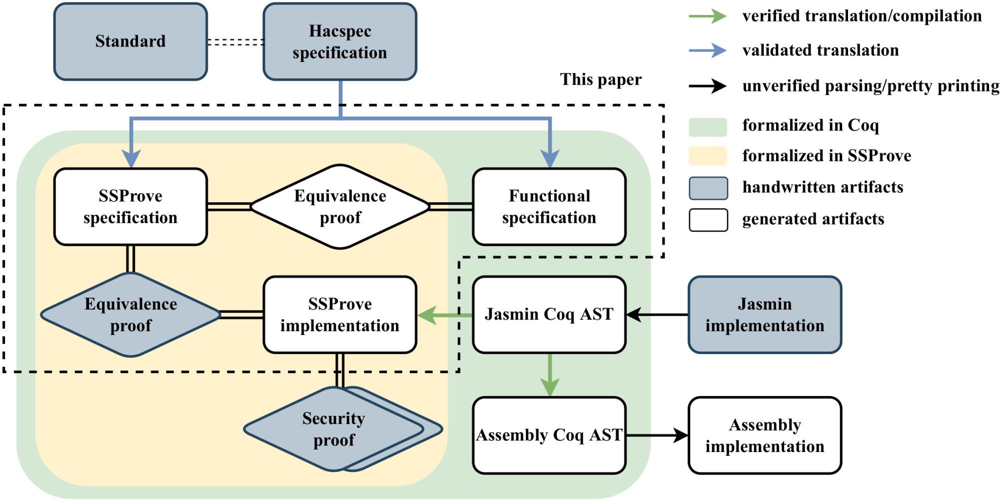


Introduction
Workflow - Jasmin



Introduction
Workflow - Jasmin



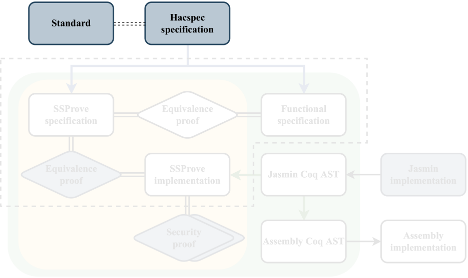


Standard
Hacspec specification
This paper
SSProve specification
Equivalence proof
Functional specification
Equivalence proof
SSProve implementation
Jasmin Coq AST
Jasmin implementation
Security proof
Assembly Coq AST
Assembly implementation
verified translation/compilation
validated translation
unverified parsing/pretty printing
formalized in Coq
formalized in SSProve
handwritten artifacts
generated artifacts

Example: One-time pad (OTP)

Example: One-time pad (OTP)

Specification

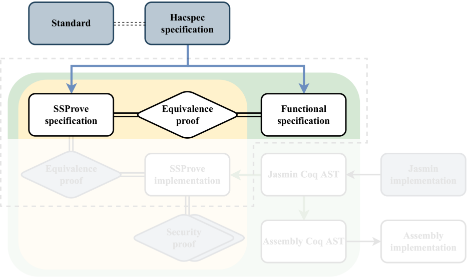


Hacspec definition

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

automatically translated to SSProve

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

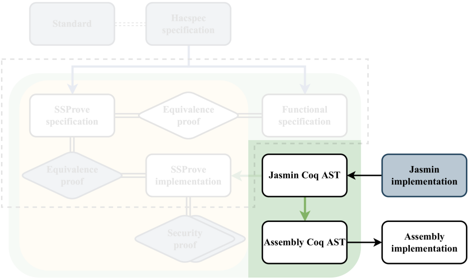


Example: One-time pad (OTP)

Jasmin implementation

Jasmin implementation

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

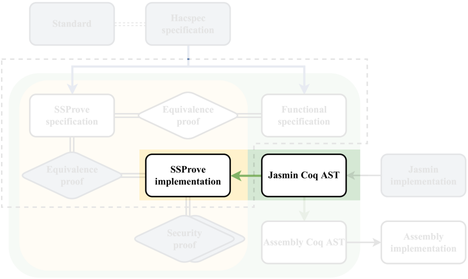


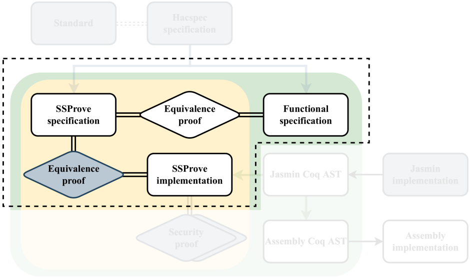
Example: One-time pad (OTP)

Jasmin implementation

automatically translated to SSProve.

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





Equivalence of translations

$$
\begin{aligned}
&\texttt{Theorem xor\_equiv} : \forall\ \texttt{id w1 w2},\\
&\quad \vdash \{\top\}\\
&\qquad \texttt{JXOR id w1 w2} \approx \texttt{hacspec\_xor\_state w1 w2}\\
&\quad \{\ \lambda\ (v_0,\ h_0)\ (v_1,\ h_1),\ v_0 = v_1\ \}.
\end{aligned}
$$

is proved using the rules of the relational program logic of SSProve.

Example: One-time pad (OTP)

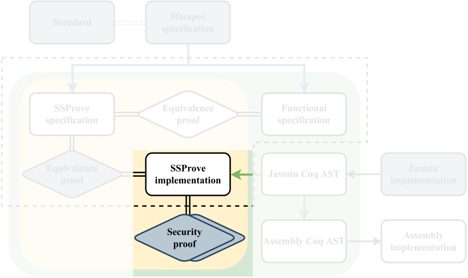
Security proof

SSProve define

- a *package*: collection of procedures with import and export interface
- a *game*: a package without imports
- a *game pair*: two games that export the same procedures.
    - e.g. a real encryption scheme and an oracle
- *game hopping*: chain of game pairs

The Jasmin game (`JOTP_real`) exports

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



Example: One-time pad (OTP)

Security proof

The SSProve game (`OTP_real`) exports

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

for which we have a security proof

Example: One-time pad (OTP)

Security proof

We now show the Jasmin implementation is secure (IND-CPA). Combining

Lemma JOTP_OTP_perf_ind id : JOTP_real id $\approx$ OTP_real.

with the already established security of OTP_real
we get security of JOTP_real

Framework

We now have a foundation framework!

- Specifying in Hacspec and implementing in Jasmin
- de-extracting and translating ⚙ into SSProve
- proving equivalence and security properties in SSProve

Evaluation: Advanced Encryption Standard (AES)

Evaluation: Advanced Encryption Standard (AES)

For a pseudo random function (PRF) one can build an encryption scheme

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

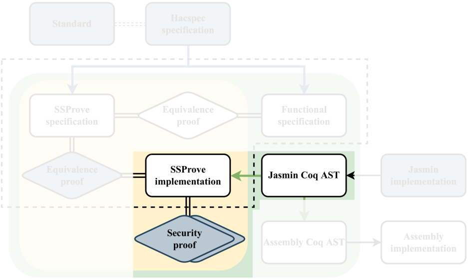
The enc function is given by

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

Evaluation: Advanced Encryption Standard (AES)

The high-level structure of the security analysis is

1. (imp.): intermediate impl. in SSProve
2. (fun.): functional impl. in Coq.
3. (imp.) $\approx$ (fun.).
4. (trans.) $\approx$ (imp.).
5. Connect to the existing security proof

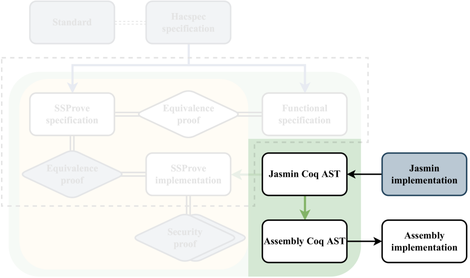


Evaluation: Advanced Encryption Standard (AES)

Connecting AES to the PRF security proof

The encryption function implemented in Jasmin:

```
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);
}
```

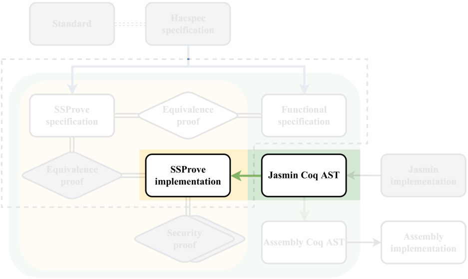


Evaluation: Advanced Encryption Standard (AES)

Connecting AES to the PRF security proof

We automatically translate it into SSProve as `JENC` and use it in the following security game:

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



prove it indistinguishability from a similar scheme `CPRF_real JENC`.

Evaluation: Advanced Encryption Standard (AES)

Connecting AES to the PRF security proof

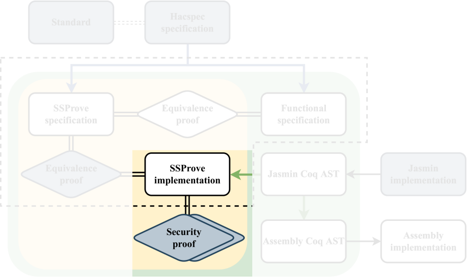
Now we show

```
Theorem JPRF_perf_ind id :
  JPRF_real id ≈ CPRF_real.
```

and

```
Theorem CPRF_perf_ind :
  CPRF_real ≈ PRF_real aes.
```

this combined with an equivalence to a Hacspec specification gives us end-to-end verification of AES.



Conclusion

We contribute

- a framework for end-to-end verification
- a monadic embedding of a simple subset of Rust into SSProve
  - with a refinement relation to a logical specification in Coq
- pritty-printing of Jasmin and automatic translation to SSProve

the framework has strong guarantees:

- Hacspec: translation validation
- SSProve: equivalence proof, security properties
- Jasmin: the preservation of operation semantics (robust compilation)