Jasmin: a Certified Workbench for High-Assurance and High-Speed Cryptography

Vincent Laporte
& the many Jasmin contributors

The Coq Workshop 2021-07-02

Once upon a time

How to write **high-assurance implementations of cryptography primitives**?

Conflicting goals

- Fast
- Correct
- Secure

Dilemma

- Write assembly (runs fast)
- Use higher-level abstractions (may be proved correct)

Today

Many different[1] answers to this question: Fiat Cryptography, Hacl*, Jasmin, Vale...

Jasmin

- Programming language friendly to both practitioners and tools
- Compiler to assembly (proved correct and constant-time preserving in Coq)
- Verification tools: interactive (EasyCrypt) or (semi-)automated

[1] Diversity is good

Implement a first (reference) version

```
fn gimli(stack u32[12] state) → stack u32[12] {
    inline int round, column;
    for round = 24 downto 0 {
        for column = 0 to 4 {
            state = SP(state, column);
        }
        if round % 4 == 0 {
            state = swap(state, 0, 1);
            state = swap(state, 2, 3);
        }
        if round % 4 == 2 {
            state = swap(state, 0, 2);
            state = swap(state, 1, 3);
        }
        if round % 4 == 0 {
            state[0] ^= 0x9e377900 + round;
        }
    }
    return state;
}
```

```
fn SP(stack u32[12] st, inline int col) → stack u32[12] {
    reg u32 x, y, z, a, b, c;
    x = st[0 + col]; x = rotate(x, 24);
    y = st[4 + col]; y = rotate(y, 9);
    z = st[8 + col];
    a = x;
    b = z; b <<= 1;
    c = y; c &= z; c <<= 2;
    a ^= b; a ^= c;
    st[8 + col] = a;
    a = y;
    b = x; b |= z; b <<= 1;
    a ^= x; a ^= b;
    st[4 + col] = a;
    a = z;
    b = x; b &= y; b <<= 3;
    a ^= y; a ^= b;
    st[col] = a;
    return st;
}
```

Does it make any sense?

`jasminc -checksafety gimli.jazz`

- No safety violation
- Memory ranges: `state: [0; 48]` (bytes)
- Alignment: `state: 32` (bits)

Static analysis by abstract interpretation

- Infers linear relations between initial values of the arguments and values of variables
- Proves termination
- Proves absence of run-time errors
  - out-of-bound array accesses
  - undefined arithmetic (division by zero, etc.)
  - badly aligned memory accesses
- Returns a sufficient pre-condition on the initial memory

Verify it!

- Extract to EasyCrypt[2]

```
jasminc -ec gimli ./gimli.jazz
```

- and prove
  - functional correctness (wrt. HACSPEC specification)
  - bijectivity of the permutation
  - ...

EasyCrypt

- A time-tested interactive prover for cryptography primitives
- Probabilistic imperative programming language pWHILE
- Program logics, e.g., pRHL (probabilistic relational Hoare logic)
- Tactics for proof automation: `wp`, `sim`, `smt`...

[2] https://easycrypt.info

Optimize it (make it run fast)!

Programmer has control over low-level details

- wide registers & SIMD instructions
- instruction scheduling
- spilling (what to spill and when)

Correctness justified by program equivalence

- good support in EasyCrypt for relational reasoning
- high-level features of Jasmin make the proof relatively easy
- the most difficult is to specify x86 instructions

Prove it secure (Constant-Time)

Cryptographic Constant-Time (CT)

- An efficient counter-measure against remote (cache-based) time-channel attacks
- No branching on sensitive data
- Memory access at public addresses only

- EasyCrypt model with explicit leakage: `jasminc -CT -ec swap gimli.jazz`
- Prove non-interference by self-composition

$$i_1 = i_2 \land j_1 = j_2 \implies \ell_1 = \ell_2$$

```
var leakages : leakages_t
proc swap (state:W32.t Array12.t, i:int, j:int)
: W32.t Array12.t = {
    var aux, x, y: W32.t;
    leakages ← LeakAddr([i]) :: leakages;
    aux ← state.[i]; x ← aux;
    leakages ← LeakAddr([j]) :: leakages;
    aux ← state.[j]; y ← aux;
    leakages ← LeakAddr([]) :: leakages;
    aux ← y;
    leakages ← LeakAddr([i]) :: leakages;
    state.[i] ← aux;
    leakages ← LeakAddr([]) :: leakages;
    aux ← x;
    leakages ← LeakAddr([j]) :: leakages;
    state.[j] ← aux;
    return (state); }
```

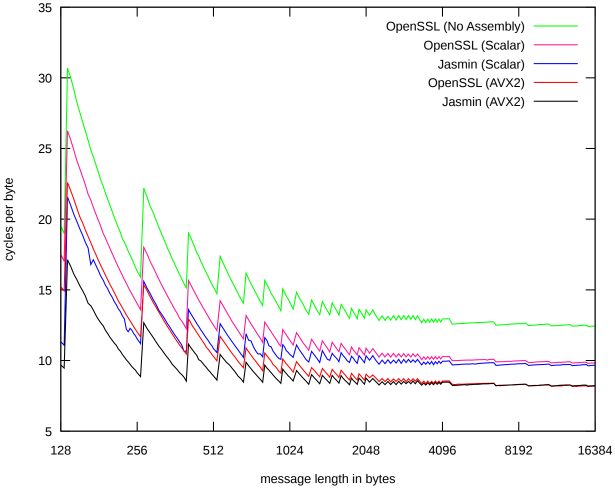
Example: SHA3 hash functions (Almeida et al. 2019)

Formal, machine-checked proofs of:

- Correctness (wrt standard)
- Security (indifferentiability)
- Constant-Time security

for fast assembly implementations.

Throughput of SHAKE256:



A programming language suitable for formal verification

- Good methodology is important to develop efficient correct implementations
- Language (incl. semantics) design is key for enabling machine-checked formal proofs
- Let's avoid the usual difficulties of formal verification of low-level programs

A high-level programming language (semantics)

Values

- Mathematical integers
- Booleans
- Machine words (8-bit – 256-bit)
- Arrays of words
  (static size, applicative)

Storage

- Local variables
- Immutable global values
- Global unstructured memory
  (shared with the environment)

Structure

- Structured control-flow
- Functions
  - Call-by-value
  - Unrestricted signatures

fn **swap**(stack u32[12] state, inline int i j) ⟶ stack u32[12]

Low-level control

- Support for SIMD
  - Vector values (incl. literal)
  - Vectorized instructions
- Intrinsics (access to assembly instructions)
  - With a pure semantics
  - Explicit manipulation of flags if needed
- Explicit spilling (reg ↔ stack)
- Alignment of code blocks
- ...

```
fn keccakf1600_avx2(reg u256[7] state, reg u64 ...)
⟶ reg u256[7] {
   reg u256[9] t;
   reg u256 c00 c14;
   reg u32 r;
   reg bool zf;
...
   align while {
...
      c00 = c00 ^ state[2];
      t[0] = #VPERMQ(c00, (4u2)[1, 0, 3, 2]);
      t[1] = c14 >>4u64 63;
      t[2] = c14 +4u64 c14;
      t[1] = t[1] | t[2];
...
      _, _, _, zf, r = #DEC_32(r);
   } (! zf)
   return state;
}
```

Applicative arrays

```
param int N = 4;

fn f(stack u64[N] x y) ⟶ stack u64[N] {
    inline int i;
    reg u64 v;
    for i = 0 to N {
        v = y[N - i];
        x[i] += v; // this writes to x, not to y
    }
    return x;
}
// This function returns zero, whatever h does
fn g() ⟶ reg u64 {
    stack u64[1] t;
    reg u64 r;
    t[0] = 0;
    h(t); // t is local, cannot be modified by h
    r = t[0];
    return r;
}
```

- Arrays are values, stored in variables
- No pointer arithmetic involving arrays
- Trivial alias analysis (based on names)
- Allows modular reasoning w/o separation logic

Formal reasoning on safe programs

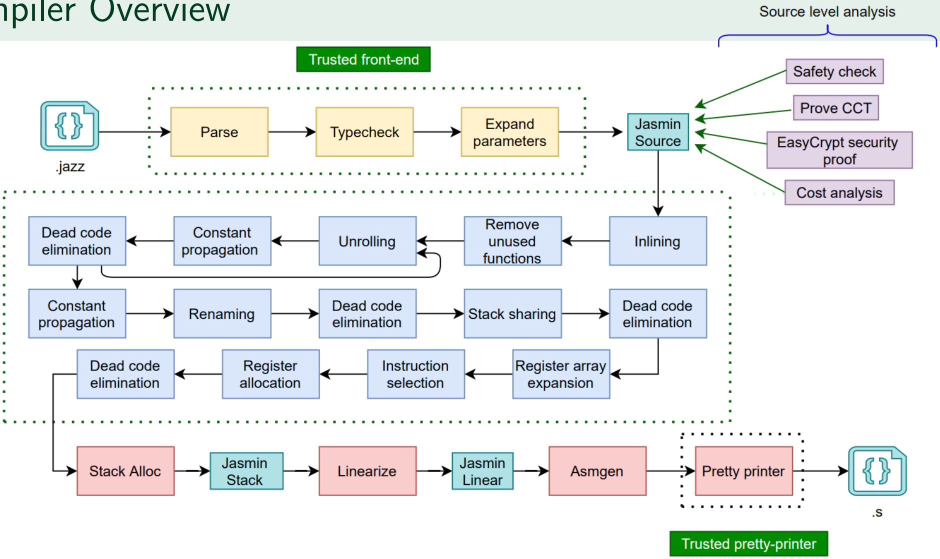
In EasyCrypt, only safe programs are considered

The semantics is thus simplified:

- arrays are unbounded (no bound checks)
- arrays are indexed by mathematical integers (no worries about overflow)
- all pointers are valid (for read and write access)

Such separation of concern ensures that low-level details about safety do not pollute the other proofs.

Compiler Overview



Translation-Validation

- Implement in OCaml
- Validate in Coq
- Only prove the checker, not the crazy heuristics
  - (there are no crazy heuristics)
- For a few passes, it is not worth the trouble
- Make it easy to experiment
- Make it harder to write friendly error messages

A few validated passes

Renaming

- Remove redundant copies (`x = y`)
- Done heuristically

Register array expansion

- Replace each array cell (`x[1]`) by a fresh variable (`x1`)
- A kind of renaming

Register allocation

- Enforce micro-architectural constraints
- Enforce standard calling-conventions for export functions
- Try to eliminate redundant copies
- Favor caller-saved registers

Stack allocation

- Choose the layout of local variables in the stack frame
- Use pointer arithmetic instead of variable names

Open questions (but within reach)

- Model “stateful” instructions (e.g., `RDTSC`, `RDRAND`)
- Several target architectures (ARM, RISC-V)
- Security against Spectre and co. (preservation of “speculative constant-time”)
- Reduce the TCB (validate the semantics)

Beyond High-Assurance Cryptography

Your own research on top of Jasmin: formal methods (Coq proofs!), verified compilation, security...

- Small language
- Clean semantics
- Formal semantics in the best proof assistant ever
- Realistic formally-verified compiler
- Exciting applications

Ongoing projects

- Low-level (pipeline-aware) cost analysis
- Security of implementations against Spectre attacks
- Secure compilation of speculative-constant-time
- Formal verification of an information-flow checker
- Certainly a few more

Thanks

None of this would be possible without Coq, the tool and the community.

Coq pain points

- Rose trees (useless induction principles, picky guard condition)
- No library for machine words (now there are too many of them)
- Build system (mixing hand-written and Coq-extracted OCaml)

Thanks

Jasmin contributors

Adrien Koutsos, Alley Stoughton, Arthur Blot, Benedikt Schmidt, Benjamin Grégoire, Cécile Baritel-Ruet, François Dupressoir, Gilles Barthe, Hugo Pacheco, Jean-Christophe Léchenet, José Bacelar Almeida, Manuel Barbosa, Pierre-Yves Strub, Swarn Priya, Tiago Oliveira, Vincent Laporte and many users

Join the Jasmin community

Get started https://github.com/jasmin-lang/jasmin

Zulip chat https://zulip.mpi-sp.org

Bibliography

Almeida, José Bacelar, 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 *CCS'17*.

Almeida, José Bacelar, 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 (S&P)*. https://doi.org/10.1109/SP40000.2020.00028.

Almeida, José Bacelar, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. 2019. "Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3." In *CCS'19*. https://doi.org/10.1145/3319535.3363211.