Jasmin: High-Assurance and High-Speed Cryptography

José Bacelar Almeida Manuel Barbosa Gilles Barthe Arthur Blot Benjamin Grégoire
Vincent Laporte Tiago Oliveira Hugo Pacheco Benedick Schmidt Pierre-Yves Strub

CCS'17 — 2017-11-02

Implementing “crypto”: a subtle equilibrium

- Correct
- Fast
- Secure (side-channel free)

A gap between C and assembly

C

- Portable
- Convenient software-engineering abstractions
- Readable, maintainable

A gap between C and assembly

C

- Portable
- Convenient software-engineering abstractions
- Readable, maintainable

Assembly

- Efficiency
- Control (instruction selection and scheduling)
- Precise semantics

Jasmin: the blossoming programming language

A language

A formal semantics

A (correct) compiler

Tooling for proof of safety and leakage-freedom

Jasmin by example

Jasmin “Hello World!” (constant-time swapping)

```
param int n = 4;

inline
fn cswap(stack b64[n] x, stack b64[n] y, reg b64 swap)
-> stack b64[n], stack b64[n] {
  reg b64 tmp1, tmp2, mask;
  inline int i;
  mask = swap * 0xffffffffffffffff;
  for i = 0 to n {
    tmp1 = x[i];
    tmp1 ^= y[i];
    tmp1 &= mask;
    tmp2 = x[i];
    tmp2 ^= tmp1;
    x[i] = tmp2;
    tmp2 = y[i];
    tmp2 ^= tmp1;
    y[i] = tmp2;
  }
  return x, y;
}
```

Zero-cost abstractions

- Variable names
- Global parameters
- Arrays
- Loops
- Inline functions (with custom calling conventions)

Control down to the instruction-set architecture level

```
reg bool cf;
reg b64 addt0, addt1, t10, t11, t12, t13;
// ...
t10 = [workp + 4 * 8];
t11 = [workp + 5 * 8];
t12 = [workp + 6 * 8];
t13 = [workp + 7 * 8];
// ...
cf, t10 += [workp + 8 * 8];
cf, t11 += [workp + 9 * 8] + cf;
cf, t12 += [workp + 10 * 8] + cf;
cf, t13 += [workp + 11 * 8] + cf;
addt0 = 0;
addt1 = 38;
addt1 = addt0 if ! cf;
```

- Direct memory access
- The carry flag is an ordinary boolean variable

Control down to the instruction-set architecture level

```
reg bool cf;
reg b64 addt0, addt1, t10, t11, t12, t13;
// ...
t10 = [workp + 4 * 8];
t11 = [workp + 5 * 8];
t12 = [workp + 6 * 8];
t13 = [workp + 7 * 8];
// ...
cf, t10 += [workp + 8 * 8];
cf, t11 += [workp + 9 * 8] + cf;
cf, t12 += [workp + 10 * 8] + cf;
cf, t13 += [workp + 11 * 8] + cf;
addt0 = 0;
addt1 = 38;
addt1 = addt0 if ! cf;
```

- Direct memory access
- The carry flag is an ordinary boolean variable

```
reg b64 i, j;
stack b64 is, js;
// ...
j = 62;
i = 3;
while (i >=s 0) {
    is = i;
    // ...
    while (j >=s 0) {
        js = j;
        // ...
        j = js; j -= 1;
    }
    j = 63; i = is; i -= 1;
}
```

- Control over loop unrolling
- Control over spilling

The Jasmin compiler

Goals / Features

- Predictability and control of the generated assembly
- Zero-cost abstractions
- Correct (Coq proofs, machine-checked)
- Preserves constant-time

Compilation passes

- For loop unrolling
- Function inlining
- Constant-propagation
- Redundancy elimination
- Sharing of stack variables
- Register array expansion
- Lowering
- Register allocation
- Linearization

A correctness theorem

Each pass proved on its own

Maximal use of validation

Reuse of a single checker (core) for various passes

Suitable theorem for libraries (aka separate compilation)

Theorem (Jasmin compiler correctness)

$$
\begin{aligned}
&\forall\ p\ p'\ \cdot\ \mathit{compile}(p) = \mathit{ok}(p') \longrightarrow \\
&\forall\ f\ \cdot\ f \in \mathit{exports}(p) \longrightarrow \\
&\forall\ v_a\ m\ v_r\ m'\ \cdot\ \mathit{enough\text{-}stack\text{-}space}(f, p', m) \longrightarrow \\
&f, v_a, m \Downarrow^{p} v_r, m' \longrightarrow f, v_a, m \Downarrow^{p'} v_r, m'
\end{aligned}
$$

Beyond correctness

Automatic proof of memory safety

- Goal: prove that all memory accesses are valid.
- Require user annotations: function preconditions, loop invariants.
- Translate to Dafny
- Reuse the Dafny·Boogie infrastructure

```
fn pack(reg u64 rp, reg u64[4] xa)
//@ requires valid(rp, 0 * 8, 4 * 8 - 1);
{
  inline int i;

  xa = freeze(xa);

  for i = 0 to 3
  //@ invariant valid(rp, 0 * 8, 4 * 8 - 1);
  {
    [rp + (i * 8)] = xa[i];
  }
}
```

Automatic proof of constant-time

- Goal: prove that a function is "constant-time"
- Require user annotations: function preconditions, loop invariants.
- Translate to Dafny then Boogie
- Build a product program and check that the product is safe (technique adapted from the CT-verif tool)

```
fn mladder(stack u64[4] xr, reg u64 sp)
    ⟶ (stack u64[4], stack u64[4])
//@ security requires public(sp);
{
    ...
    while (i >=s 0)
    //@ security invariant public(i);
    {
        ...
```

Jasmin for secure programming

- Infrastructure for automatic checks
- The compiler preserves functional correctness, safety, constant-time
- Known verification techniques apply to Jasmin programs
- Jasmin high-level features make automatic verification easier: arbitrary while loops and pointer arithmetic are seldom used.

Running Jasmin programs

Jasmin programs as libraries

- Compliant with standard ABI
- Link with your own programs
  written in assembly, C, OCaml, Rust...

Experiment: from qhasm to Jasmin

qhasm

- A *high-level* assembly language by D. Bernstein
- "Included" in Jasmin
- ... hence easy automatic translation from qhasm to Jasmin

Experiment: from qhasm to Jasmin

qhasm

- A *high-level* assembly language by D. Bernstein
- "Included" in Jasmin
- ... hence easy automatic translation from qhasm to Jasmin

Supercop

- A cool infrastructure for testing and comparing crypto implementations
- Includes various examples written in qhasm
- A nice supply of Jasmin programs
  (since there are not many Jasmin programmers yet)

Jasmin in Supercop

Implementation	qhasm	Jasmin	Ratio
X25519-4limb-base	147 084	148 914	1.012
X25519-4limb	147 890	148 922	1.006
X25519-4limb-jasmin		143 982	
X25519-5limb-base	148 354	147 200	0.992
X25519-5limb	148 572	147 090	0.990
ed25519-5limb-keypair	55 364	56 594	1.022
ed25519-5limb-sign	83 430	85 038	1.019
ed25519-5limb-open	182 520	188 180	1.031
salsa20	12 322	12 460	1.011
salsa20-xor	12 208	12 252	1.004

Figure 1: Supercop cycle count

Conclusions

Present

- The Jasmin language
- Correct compiler for x64
- Automatic checkers for safety and constant-time
- Programs obtained from qhasm

Future

- More target architectures
- Vector instructions
- Logic and verification tools
- Write more programs

Thanks

¿ Questions ?

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

Extra

Jasmin vs. qhasm

qhasm

- Multi-platform
- Vector instructions

Jasmin

- Loops
- Arrays
- Functions
- Formal semantics
- Proof of correctness

Jasmin vs. Bedrock

Bedrock

- Very low-level IL: XCAP, code-labels as r-values
- Embedded into Gallina (Coq)
- Deep emphasis on specifications and functional correctness (pervasive annotations)
- No optimizing compiler
- Programmer must declare stack usage

Jasmin

- New language
- Optimizing compiler

Jasmin vs. HACL*

HACL*

- In Mozilla Firefox
- Formal proof of the programs

Jasmin

- Formal proof of the compiler
- Control over efficiency and low-level features
- For the crypto programmer (not for the type theorist)

Jasmin vs. Vale

Vale

- All proofs are done at the assembly level
- Embedded in Dafny

Jasmin

- Correct compiler to assembly
- New language