High-Assurance Cryptography in Jasmin & Spectre Security

Vincent Laporte;
and the Formosa Crypto team members

2022-09-26, Cambium, Inria, Paris

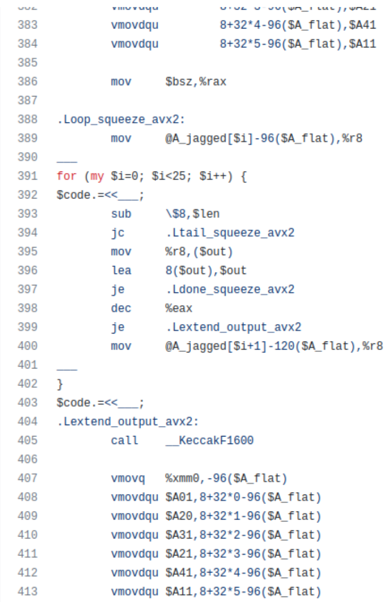
Challenges for a (post-quantum) cryptography library

Ambitious goals

- Execution speed
- Functional correctness
- Safety
- Security against:
    - quantum computers
    - side-channel attacks
    - speculative execution attacks
    - fault attacks

Everything is broken

Illustration: crypto/sha/asm/keccak1600-avx2.pl (OpenSSL)



```
        vmovdqu     8+32*4-96($A_flat),$A41
        vmovdqu     8+32*5-96($A_flat),$A11

        mov     $bsz,%rax

.Loop_squeeze_avx2:
        mov     @A_jagged[$i]-96($A_flat),%r8
___
for (my $i=0; $i<25; $i++) {
$code.=<<___;
        sub     \$8,$len
        jc      .Ltail_squeeze_avx2
        mov     %r8,($out)
        lea     8($out),$out
        je      .Ldone_squeeze_avx2
        dec     %eax
        je      .Lextend_output_avx2
        mov     @A_jagged[$i+1]-120($A_flat),%r8
___
}
$code.=<<___;
.Lextend_output_avx2:
        call    __KeccakF1600

        vmovq   %xmm0,-96($A_flat)
        vmovdqu $A01,8+32*0-96($A_flat)
        vmovdqu $A20,8+32*1-96($A_flat)
        vmovdqu $A31,8+32*2-96($A_flat)
        vmovdqu $A21,8+32*3-96($A_flat)
        vmovdqu $A41,8+32*4-96($A_flat)
        vmovdqu $A11,8+32*5-96($A_flat)
```

Formosa Crypto https://formosa-crypto.org/

Jasmin

A programming language that enables both:

- crypto practitioners to write optimized implementations
- formal method enthusiasts to verify these implementations

A tool-box

- Certified compiler: allows reasoning at source level
- Automatic checkers (safety, constant-time)
- EasyCrypt support for semi-automatic verification

LibJade: work in progress https://github.com/formosa-crypto/libjade

Aim: comprehensive library of (post-quantum) cryptography primitives

- efficient
- verified

Jasmin Hello World

```
export
fn lehmer(reg u64 state) ⟶ reg u64 {
    reg u64[2] s, m;
    stack u64[2] t;
    inline int i;
    reg u64 j, result;
    for i = 0 to 2 {
        s[i] = [state + i * 8];
    }
    m[0] = 0x261fd0407a968add;
    m[1] = 0x45a31efc5a35d971;
    t = mul128(s, m);
    result = t[1];
    j = 0;
    while (j < 2) {
        [state + j * 8] = t[(int) j];
        j += 1;
    }
    return result;
}
```

```
inline
fn mul128(reg u64[2] x, y) ⟶ stack u64[2] {
    reg u64 xhi, ylo, lo, hi, tmp;
    stack u64[2] r;
    xhi = x[1];
    ylo = y[0];
    hi, lo = #MULX(ylo, x[0]);
    tmp = xhi * y[0];
    hi += tmp;
    y[1] *= x[0];
    y[1] += hi;
    r[0] = lo;
    r[1] = y[1];
    return r;
}
```

Formal Semantics

Semantics judgment, defined in Coq

In program p,
calling function f with arguments $\vec{a}$ from initial memory m
terminates in final memory m' and returns values $\vec{r}$:

$$f : (\vec{a}, m) \Downarrow_p (\vec{r}, m')$$

Automatic Checker, implemented in OCaml

Infers a sufficient precondition P (for a function f in program p) such that:

$$\forall \vec{a}\ m, P(\vec{a}, m) \implies \exists \vec{r}\ m', f : (\vec{a}, m) \Downarrow_p (\vec{r}, m')$$

- polyhedra for numerical arguments
- range and alignment for pointer arguments

Formal Verification of Jasmin Programs, using EasyCrypt

Jasmin programs are translated into pWhile programs

For functional correctness

- Using (probabilistic) Hoare logic; or
- by proving program equivalence.

For semantic security (e.g., IND$-CPA)

- This is where EasyCrypt shines

Question to the audience

Which logic can express the correctness of the translation?

Compiler Correctness

Semantics Preservation (forward simulation)

If the compilation of program p produces a program p',
then its safe behaviors are preserved:

$$\forall \vec{a}\ m\ \vec{r}\ m', \qquad f : (\vec{a}, m) \Downarrow_p (\vec{r}, m') \qquad \Longrightarrow \qquad f : (\vec{a}, m) \Downarrow_{p'} (\vec{r}, m').$$

Hidden Details

- Source and target languages are different
- Initial states are not the same (but tightly related)
- The target stack must be large enough
  - i.e., the compiler does not enforce the absence of “stack overflow”

Consequences of Compiler Correctness

Source-level reasoning is **correct**

- Functional properties carry down to the assembly code
- including semantic security

Limits

- Non-determinism (caveat: next Jasmin release will have #randombytes)
- Changing representation of values
- Non-functional properties

Verification of Constant-Time Security

Instrumented Semantics

The adversary observes:

- control flow
- memory accesses:

$$f : (\vec{a}, m) \Downarrow_p^{\ell} (\vec{r}, m')$$

Security property (φ-CT)

Given a "low-equivalence" relation φ between initial states:

$$\forall \vec{a}_1\ m_1\ \vec{a}_2\ m_2\ \ell_1\ \ell_2\ \vec{r}_1\ \vec{r}_2\ m'_1\ m'_2,$$

$$(\vec{a}_1, m_1)\varphi(\vec{a}_2, m_2) \implies \begin{cases} f : (\vec{a}_1, m_1) \Downarrow_p^{\ell_1} (\vec{r}_1, m'_1) \\ f : (\vec{a}_2, m_2) \Downarrow_p^{\ell_2} (\vec{r}_2, m'_2) \end{cases} \implies \ell_1 = \ell_2.$$

Automatic checker

Programs can be annotated with security level annotations.

Able to deal with most cases

Extraction to EasyCrypt

EC has relational program logics

Mostly automatic

Instrumented Correctness

Leakage Transformers

If the compilation of program p produces a program p', there exists a leakage transformer F such that:

$$\forall \vec{a}\ m\ \ell\ \vec{r}\ m', \qquad f : (\vec{a}, m) \Downarrow_{p}^{\ell} (\vec{r}, m') \qquad \Longrightarrow \qquad f : (\vec{a}, m) \Downarrow_{p'}^{F(\ell)} (\vec{r}, m').$$

- Stronger correctness theorem
- Proved for Jasmin v21 only
- Implies preservation of constant-time

Summary

- Language & Tools & Theorems
- Case studies: Curve25519, ChaCha20/Poly1305, SHA3, Kyber, . . .
- Ongoing work:
  - reduce the TCB
  - more target architectures
  - better programming environment
  - LibJade, a comprehensive (post-quantum) crypto library

Speculative Execution: branch prediction and Spectre v1

- Do not wait
  - the end of an instruction before starting to execute the next one
- Speculate
  - what is the next instruction to execute

Example

```
⋮
mov     rax, 0
cmp     rdi, 2
jnb    ...
mov     rax, qword ptr[rcx + rdi * 8]
mov     rax, qword ptr[rdx + rax * 8]
⋮
```

Many processors feature branch prediction:

Figure 1: IBM Stretch (by Rama, CC BY-SA)

Spectre Vulnerabilities (2018) & Speculative Load Hardening (SLH)

```
u64[2] offsets = { 1, 0 };
u64[2] data = { 0xcafe, 0xbeef };

export fn archetype(#public reg u64 n) ⟶ reg u64 {
    reg u64 p r;
    reg bool c;


    r = 0;
    c = n <u 2;
    if c { // Branch trained with in-bounds n
        // Speculatively, out-of-bounds array access
        p = offsets[(int) n];
        // Speculatively, load from secret address → Cache attack!
        r = data[(int) p];
    }
    return r;
}
```

Spectre Vulnerabilities (2018) & Speculative Load Hardening (SLH)

```
u64[2] offsets = { 1, 0 };
u64[2] data = { 0xcafe, 0xbeef };

export fn archetype(#public reg u64 n) ⟶ reg u64 {
    reg u64 p r;
    reg bool c;


    r = 0;
    c = n <u 2;
    if c { // Branch trained with in-bounds n

        p = offsets[(int) n];
        ? #LFENCE;                      // Secure but inefficient
        r = data[(int) p];
    }
    return r;
}
```

SPECTRE

Spectre Vulnerabilities (2018) & Speculative Load Hardening (SLH)

```
u64[2] offsets = { 1, 0 };
u64[2] data = { 0xcafe, 0xbeef };

export fn archetype(#public reg u64 n) ⟶ reg u64 {
    reg u64 p r;
    reg bool c;


    r = 0;
    c = n <u 2;
    if c {

        p = offsets[(int) n];
        p = #protect(p);      // Wish: only prevent insecure flows
        r = data[(int) p];
    }
    return r;
}
```

Spectre Vulnerabilities (2018) & Speculative Load Hardening (SLH)

```
u64[2] offsets = { 1, 0 };
u64[2] data = { 0xcafe, 0xbeef };

export fn archetype(#transient reg u64 n) ⟶ reg u64 {
    reg u64 p r;
    reg bool c;
    #msf reg u64 mask;       // Is the hardware misspeculating?
    mask = #init_msf();                           // Initial fence
    r = 0;
    c = n <u 2;
    if c {
        mask = #set_msf(c, mask);     // Detect misspeculation
        p = offsets[(int) n];
        p = #protect(p, mask);            // Mask loaded value
        r = data[(int) p];
    }
    return r;
}
```

SPECTRE

Primitives for SLH

Primitive	Semantics	Implementation
m = init-msf()	m = 0	fence; m = 0
m = set-msf(c, m)	assert(c)	m = c ? m : -1
v = protect(v, m)	assert(m == 0)	v |= m

Speculative semantics for Jasmin

- The adversary has full control over the speculation through **directives**:

$$d \in \mathrm{Dir} ::= \mathsf{step} \mid \mathsf{force} \mid \mathsf{load}\ a, i \mid \mathsf{store}\ a, i$$

- Side-channel leakage is modeled through **observations**:

$$o \in \mathrm{Obs} ::= \bullet \mid \mathsf{read}\ a, v \mid \mathsf{write}\ a, v \mid \mathsf{branch}\ b$$

- Small steps relate states with an explicit mispeculation bit:

$$\frac{}{x := e; c,\ \langle \rho, \mu, b\rangle \xrightarrow[\mathsf{step}]{\bullet} c,\ \langle \rho\{x \leftarrow [\![e]\!]_\rho\}, \mu, b\rangle}\ \textsc{Assign}$$

Small Step Speculative Semantics

$$\frac{e = \text{if } (d = \text{force}) \text{ then } \neg[\![t]\!]_\rho \text{ else } [\![t]\!]_\rho}{\text{if } t \text{ then } c_\top \text{ else } c_\bot; c,\ \langle\rho, \mu, b\rangle \xrightarrow[d]{\text{branch}(e)} c_e; c,\ \langle\rho, \mu, b \vee d = \text{force}\rangle}\ \textsc{Cond}$$

$$\frac{[\![e]\!]_\rho = n \in |a| \qquad \mu(a, n) = v}{x := a[e]; c,\ \langle\rho, \mu, b\rangle \xrightarrow[\text{step}]{\text{addr}(n)} c,\ \langle\rho\{x \leftarrow v\}, \mu, b\rangle}\ \textsc{Load}$$

$$\frac{[\![e]\!]_\rho = n \notin |a| \qquad m \in |\alpha|, \mu(\alpha, m) = v}{x := a[e]; c,\ \langle\rho, \mu, \top\rangle \xrightarrow[\text{load}(\alpha, m)]{\text{addr}(n)} c,\ \langle\rho\{x \leftarrow v\}, \mu, \top\rangle}\ \textsc{U-Load}$$

Security goal

Speculative Constant-Time

Given an equivalence relation φ on states, a program c is φ-SCT when:

$$\forall D\ s_1\ s_2\ O_1\ O_2\ \ldots, s_1\ \varphi\ s_2 \implies$$

$$c, \langle s_1, \bot \rangle \xrightarrow[D]{O_1}\!\!\!\twoheadrightarrow c_1', \langle s_1', b_1' \rangle \implies c, \langle s_2, \bot \rangle \xrightarrow[D]{O_2}\!\!\!\twoheadrightarrow c_2', \langle s_2', b_2' \rangle \implies$$

$$O_1 = O_2.$$

Type System with Constraints

Security Types

- Two security levels: $\mathsf{L} \leqslant \mathsf{H}$
- A security type is a pair $\tau = (\tau_n, \tau_s)$ of security levels
  - τ_n: security level under normal executions
  - τ_s: security level under all executions, including misspeculated ones

MSF-tracking

Flow-sensitive typing state Σ:

- unknown: *top* state
- ms: register ms accurately models the (semantic) misspeculation bit
- $\mathsf{ms}_{|e}$: register ms and expression e model the MSF

Theorem

If a program c is well-typed: $\Sigma, \Gamma \vdash c : \Sigma', \Gamma' \mid C$
then for all valuation θ such that $\theta \models C$, c is $=^{\Sigma}_{\theta\Gamma}$-SCT.

Selected Rules

$$\frac{\Gamma \vdash e : \tau \mid C}{\Sigma, \Gamma \vdash x = e : \Sigma^{|x}, \Gamma\{x \leftarrow \tau\} \mid C}\textsc{Assign}$$

$$\frac{\Gamma \vdash b : \sigma \mid C_b \qquad \Sigma_{|b}, \Gamma \vdash c_1 : \Sigma_1, \Gamma_1 \mid C_1 \qquad \Sigma_{|!b}, \Gamma \vdash c_2 : \Sigma_2, \Gamma_2 \mid C_2 \qquad \Gamma'\text{fresh}}{\Sigma, \Gamma \vdash \text{if } b \text{ then } c_1 \text{ else } c_2 : \Sigma_1 \cap \Sigma_2, \Gamma' \mid C_b \cup C_1 \cup C_2 \cup \{\sigma \leqslant \mathsf{L}; \Gamma_1 \leqslant \Gamma'; \Gamma_2 \leqslant \Gamma'\}}\textsc{Cond}$$

$$\frac{\Gamma \vdash i : \sigma \mid C_i \qquad \tau \text{ fresh}}{\Sigma, \Gamma \vdash x = a[i] : \Sigma^{|x}, \Gamma\{x \leftarrow \tau\} \mid C_i \cup \{\sigma \leqslant \mathsf{L}, \Gamma_n(a) \leqslant \tau_n, \mathsf{H} \leqslant \tau_s\}}\textsc{Load}$$

$$\frac{\Gamma \vdash i : \sigma \mid C_i \qquad \tau \text{ fresh}}{\Sigma, \Gamma \vdash \textbf{ssafe } x = a[i] : \Sigma^{|x}, \Gamma\{x \leftarrow \tau\} \mid C_i \cup \{\sigma \leqslant \mathsf{L}, \Gamma(a) \leqslant \tau\}}\textsc{Safe Load}$$

Case Study: LibJade

Selective SLH

A fence at the beginning of each export function

Not all loads need to be protected:

- secret loads are fine
- speculatively safe loads too

A few implementation tricks

Reordering can help: load (public data) early, store (secret data) late

Can use MMX for public spills

When no protect are needed, tracking the MSF is not necessary

Example: ChaCha20

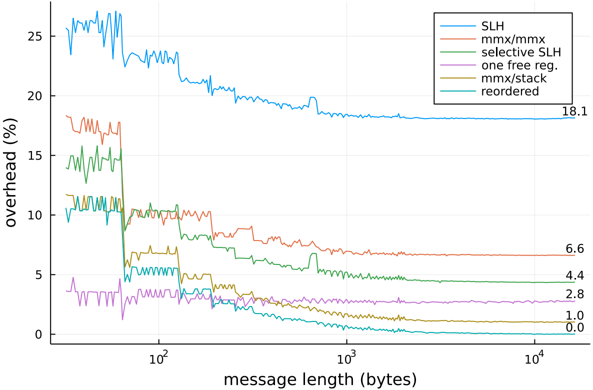


Figure 2: Run-time overhead for various implementations of ChaCha20

Summary

Selective SLH

- One can use the type-checker to insert protect instructions systematically
- Security against Spectre v1 can be achieved at a very little cost and library scale

Formal Verification

- Reasoning about speculative executions is possible
- Static analysis of optimized low-level programs is feasible

Conclusion

Formosa: High-speed, high-assurance cryptography

Fast, safe, correct, secure, . . .

Jasmin: a great environment for research

- Secure & correct compilation
- Language-based security
- . . .

Open questions

- SCT preservation
- Adoption (by crypto practitioners)
- Your questions. . .