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)
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
```plaintext
Jasmin Hello World

```
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
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?
Semantics Preservation (forward simulation)

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

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

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

<table>
<thead>
<tr>
<th>Instrumented Semantics</th>
<th>Security property (φ-CT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The adversary observes:</td>
<td>Given a “low-equivalence” relation φ between initial states:</td>
</tr>
<tr>
<td>• control flow</td>
<td>$\forall \vec{a}_1 \ m_1 \ \vec{a}_2 \ m_2 \ \ell_1 \ \ell_2 \ \vec{r}_1 \ \vec{r}_2 \ m'_1 \ m'_2,$</td>
</tr>
<tr>
<td>• memory accesses:</td>
<td>$(\vec{a}_1, m_1) \varphi (\vec{a}_2, m_2) \implies \begin{cases} f : (\vec{a}<em>1, m_1) \downarrow</em>{p}^{\ell_1} (\vec{r}_1, m'_1) \ f : (\vec{a}<em>2, m_2) \downarrow</em>{p}^{\ell_2} (\vec{r}_2, m'_2) \end{cases} \implies \ell_1 = \ell_2.$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Automatic checker</th>
<th>Extraction to EasyCrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programs can be annotated with security level annotations.</td>
<td>EC has relational program logics</td>
</tr>
<tr>
<td>Able to deal with most cases</td>
<td>Mostly automatic</td>
</tr>
</tbody>
</table>
If the compilation of program $p$ produces a program $p'$, there exists a leakage transformer $F$ such that:

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

- 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

```assembly
: 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)
1 u64[2] offsets = { 1, 0 };
2 u64[2] data = { 0xc02e, 0xbeef };
3
4 export fn archetype(#public reg u64 n) → reg u64 {
5    reg u64 p r;
6    reg bool c;
7
8    r = 0;
9    c = n <u 2;
10   if c { // Branch trained with in-bounds n
11      // Speculatively, out-of-bounds array access
12      p = offsets[(int) n];
13      // Speculatively, load from secret address → Cache attack!
14      r = data[(int) p];
15   }
16   return r;
17 }
18

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

```rust
1  u64[2] offsets = { 1, 0 };  
2  u64[2] data = { 0xcafe, 0xbeef };  
3  
4  export fn archetype(#public reg u64 n) \rightarrow reg u64 {  
5      reg u64 p r;  
6      reg bool c;  
7      
8      r = 0;  
9      c = n <u 2;  
10     if c { // Branch trained with in-bounds n  
11         p = offsets[(int) n];  
12         ? #LFENCE; // Secure but inefficient  
13         r = data[(int) p];  
14     }  
15     return r;  
16  }
```

---

SPECTRE

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

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

3 export fn archetype(#public reg u64 n) → reg u64 {  
4     reg u64 p r;  
5     reg bool c;  
6     r = 0;  
7     c = n <u 2;  
8     if c {  
9         p = offsets[(int) n];  
10        p = #protect(p);  // Wish: only prevent insecure flows  
11        r = data[(int) p];  
12    }  
13    return r;  
14 }
```
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;
}
Primitives for SLH

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Semantics</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = init-msf()</td>
<td>m = 0</td>
<td>fence; m = 0</td>
</tr>
<tr>
<td>m = set-msf(c, m)</td>
<td>assert(c)</td>
<td>m = c ? m : -1</td>
</tr>
<tr>
<td>v = protect(v, m)</td>
<td>assert(m == 0)</td>
<td>v</td>
</tr>
</tbody>
</table>
Speculative semantics for Jasmin

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

  \[ d \in \text{Dir} ::= \text{step} | \text{force} | \text{load } a, i | \text{store } a, i \]

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

  \[ o \in \text{Obs} ::= \bullet | \text{read } a, v | \text{write } a, v | \text{branch } b \]

- Small steps relate states with an explicit mispeculation bit:

  \[
  x := e; c, \langle \rho, \mu, b \rangle 
  \xrightarrow{\text{step}} 
  c, \langle \rho \{ x \leftarrow [e]_{\rho} \}, \mu, b \rangle
  \]

\[ \text{ASSIGN} \]
Small Step Speculative Semantics

\[
\begin{align*}
e &= \text{if } (d = \text{force}) \text{ then } \neg\llbracket t \rrbracket_\rho \text{ else } \llbracket t \rrbracket_\rho \\
\text{if } t \text{ then } c_T \text{ else } c_\perp &; c, \langle \rho, \mu, b \rangle \xrightarrow{\text{branch}(e)} c_e; c, \langle \rho, \mu, b \lor d = \text{force} \rangle
\end{align*}
\]

\[
\begin{align*}
\llbracket e \rrbracket_\rho &= n \in |a| \quad \mu(a, n) = v \\
x &:= a[e]; c, \langle \rho, \mu, b \rangle \xrightarrow{\text{addr}(n)} c, \langle \rho\{x \leftarrow v\}, \mu, b \rangle
\end{align*}
\]

\[
\begin{align*}
\llbracket e \rrbracket_\rho &= n \notin |a| \quad m \in |\alpha|, \mu(\alpha, m) = v \\
x &:= a[e]; c, \langle \rho, \mu, \top \rangle \xrightarrow{\text{load}(\alpha, m)} c, \langle \rho\{x \leftarrow v\}, \mu, \top \rangle
\end{align*}
\]
Security goal

Speculative Constant-Time

Given an equivalence relation $\varphi$ on states, a program $c$ is $\varphi$-SCT when:

$$\forall D s_1 s_2 O_1 O_2 \ldots, s_1 \varphi s_2 \implies c, \langle s_1, \bot \rangle \xrightarrow{O_1} c_1, \langle s_1', b_1' \rangle \implies c, \langle s_2, \bot \rangle \xrightarrow{O_2} c_2, \langle s_2', b_2' \rangle \implies O_1 = O_2.$$
Type System with Constraints

Security Types

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

MSF-tracking

Flow-sensitive typing state $\Sigma$:
- unknown: top state
- ms: register ms accurately models the (semantic) misspeculation bit
- $ms \mid e$: 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 $\theta$ such that $\theta \models C$, $c$ is $=\Sigma_{\theta\Gamma}$-SCT.
Selected Rules

\[ \Gamma \vdash e : \tau \mid C \]
\[ \Sigma, \Gamma \vdash x = e : \Sigma^x, \Gamma\{x \leftarrow \tau\} \mid C \] \hspace{1cm} \text{ASSIGN}

\[ \Gamma \vdash b : \sigma \mid C_b \quad \Sigma \mid_b, \Gamma \vdash c_1 : \Sigma_1, \Gamma_1 \mid C_1 \quad \Sigma \mid!_b, \Gamma \vdash c_2 : \Sigma_2, \Gamma_2 \mid C_2 \quad \Gamma' \text{fresh} \]
\[ \Sigma, \Gamma \vdash \text{if } b \text{ then } c_1 \text{ else } c_2 : \Sigma_1 \cap \Sigma_2, \Gamma' \mid C_b \cup C_1 \cup C_2 \cup \{\sigma \leq L; \Gamma_1 \leq \Gamma'; \Gamma_2 \leq \Gamma'\} \] \hspace{1cm} \text{COND}

\[ \Gamma \vdash i : \sigma \mid C_i \quad \tau \text{ fresh} \]
\[ \Sigma, \Gamma \vdash x = a[i] : \Sigma^x, \Gamma\{x \leftarrow \tau\} \mid C_i \cup \{\sigma \leq L, \Gamma_n(a) \leq \tau_n, H \leq \tau_s\} \] \hspace{1cm} \text{LOAD}

\[ \Gamma \vdash i : \sigma \mid C_i \quad \tau \text{ fresh} \]
\[ \Sigma, \Gamma \vdash \text{ssafe } x = a[i] : \Sigma^x, \Gamma\{x \leftarrow \tau\} \mid C_i \cup \{\sigma \leq L, \Gamma(a) \leq \tau\} \] \hspace{1cm} \text{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...