The Jasmin Workbench for High-Assurance & High-Speed Cryptography

Benjamin Grégoire

2023-06-28

This Talk

1. Why & What is Jasmin
2. A few Case Studies
3. Side channels
4. We are not done yet!

Why & What is Jasmin

Challenges for a (post-quantum) cryptography library

Ambitious goals

- Security proof (algorithm, quantum adversaries, EasyCrypt)
- Execution speed
- Functional correctness
- Safety
- Security against:
  - side-channel attacks
  - speculative execution attacks
- All those guaranties should be provided at the assembly level

```
	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)
```

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

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

An illustrative Jasmin program

```
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

Compiler Correctness (Coq)

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-functional properties

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

For implementation security (e.g., constant-time)

- Using relational Hoare logic
- on an instrumented program with explicit information leakage.

A few Case Studies

Fast Implementations

Two example of implementations in Jasmin, as fast as the fastest available implementations:

Curve25519 [CCS17]

- Scalar multiplication on a standard elliptic curve
- Verified for safety and constant-time security

Chacha20/Poly1305 [SP20]

- Authenticated encryption scheme
- Verified for safety and constant-time security
- With formal proofs of functional correctness

Fast, Secure, and Correct

Secure High-Assurance Implementations of SHA-3 [CCS19]

- Fast (optimized for AVX2)
- Secure (constant-time)
- Correct (wrt. a reference implementation)

Indifferentiability proof of the Sponge construction

- Main theorem about security of SHA-3
- Bounds the probability for an adversary to break it:
  - in particular to find collisions, preimages, or second preimages
- Theorem applies to the optimized implementation!

Post Quantum implementations

Kyber [CHES23]

- KEM
- Reference + avx2 implementations
- Verified for safety, functional correctness (hard part)

Dilithium [CRYPTO23]

- Signature Scheme
- Reference implementation
- Formal security proof in Easycrypt

Side channels

Implementation Security

Adversaries may observe the machine running a victim program.

Is any sensitive information leaked into these observations?

Constant-Time

- A popular mitigation against timing (cache-based) side-channel attacks
- Two rules
  - No branching on secret data
  - No memory access at secret addresses
- Can be checked using a taint analysis
  - Propagate from entry-points "security labels" (low/high)

Preservation of Constant-Time (Swarn Priya Thesis) [CSF18, POPL19, CCS21]

- Source is CT implies target is CT

Fine-Grained Leakage Models for Constant-Time

Variation of the model [CCS22]

The base-line constant-time model is too coarse in practice:

- some arithmetic operations leak a function of their arguments (DIV/MOD)
- leaking only the cache line
- Bug found in OpenSSL, patch accepted.

Problem with rejection sampling

- Kyber, Dilithium and Falcon are all based on rejection sampling
- This is not constant time, but it can be (approximate) probabilistic constant time

Spectre Security

A serious vulnerability

- Leakage of sensitive information due to speculative execution (branch prediction)
- Not a hardware bug

SPECTRE

Efficiently mitigated [SP23]

- Manually protect (against V1) the whole LibJade library (using SSLH)
- Automatically prove the result secure (type system)
- Experimentally assess that the protection cost is *low*
- Don't know how to prove preservation !!!

We are not done yet!

Scalability

- Still hard to program in Jasmin
  - Lacking documentation
  - Register allocation error message are not helpful
  - Detailed knowledge of the target architecture is often needed
- Lack of modularity
  - No separate compilation
  - Existing libraries are difficult to reuse
- Checking safety of Kyber decapsulation takes 16 CPU-hours
- Functional correctness hard to establish

Portability

Target architectures

- Currently: x86_64
- Soon: ARMv7
- Later: ARMv8, RISC-V, Open-titan?

Open question

How to build & maintain a comprehensive, multi-platform, verified library ?

Security Proofs

- More than Spectre V1, zeroing the stack and register …
- Correctness proofs too hard (link with Cryptoline?)
- Post-quantum security proof (EasyPQC, [CCS21])
- Falcon needs floating point (dealing with error) and new notion (Rényi divergence)

Questions?

Thanks