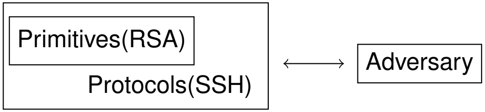
Jasmin: A Compiler and Framework for High-Assurance and High-Speed Cryptography

Benjamin Grégoire

Previous work: Provable crytography

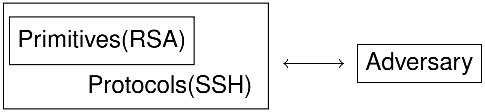
Algorithms:



Provable security: $Pr[A \text{ breaks } P] \leq Pr[B(A) \text{ breaks assumption}] + \epsilon$

Previous work: Provable crytography

Algorithms:



Provable security: $Pr[A \text{ breaks } P] \leq Pr[B(A) \text{ breaks assumption}] + \epsilon$

- Foundation: Probabilistic Relational Hoare Logic (pRHL)
- Tools : EasyCrypt / AutoG&P
- Case studies:
  encryption, signatures, hash designs (SHA3), key exchange protocols, zero knowledge protocols, garbled circuits, voting, key management service (AWS)

Implementation details are important

Most algorithms (widely) deployed today are provably secure

Still we frequently hear of crypto implementations being broken

What is the problem ?

Let us loop at some examples

Example 1: Bleichenbacher’s attack

in 1998 SSLv3 and PKCS#1 v1.5 were widely used

- Session key transported using RSA with random padding
- Bleichenbacher’s attack was published (CRYPTO’98)

- Some SSL servers would signal padding errors in decryption
- The revealed if exponentiation result was in known range

- Vulnerability allowed decrypting arbitrary messages
- Using adaptive chosen-ciphertext attack (IND-CCA)

“ The SSL documentation does not clearly specify the error conditions and corresponding alerts”
Error messages left as “implementation details”

Example 2: MEE-CBC and timing

- TLSv1.0 uses an authenticated encryption scheme called MEE-CBC
- This is a MAC-then-Pad-Then-Encrypt construction
- MEE-CBC is known to be vulnerable to padding oracle attacks using error messages
- In TLSv1.0 error messages for MEE-CBC decryption failures are sent over an encrypted channel
- So security analysis took this possibility into consideration and all is OK
- Not quite ...

Example 2: MEE-CBC and timing

- Canvel et al. [CRYPTO'03] break TLSv1.0
- Padding oracle attack can recover encrypted passwords

Attack strategy:

- Measure the time that a server takes to abort: checks padding only or padding and MAC
- If short time, guess it was padding error
- if long time, guess it was not a padding error

Simple timing attack brings padding oracle back

Lesson learned

Real-world attackers can observe more than the I/O values of crypto algorithms:

- Timing
- Power
- Sound emmitted by fan (!)

All that the adversary can somehow measure and use as a side-channel.

Constant-time programming

Software-based countermeasure against **timing** attacks and **cache** attacks.

Guideline: control-flow and memory accesses should not depend on secret data.

Rationale: crypto implementations without this property are vulnerable.

Can we trust our compiler ?

gcc/clang:

- Their intensive use provides some guaranties on their correctness
- This is not the case when we require a high level of optimisations

CompCert:

- It is certified in Coq (semantic preservation)
- Certified compilers do not perform aggressive optimisation

Do compilers preserve “constant-time”-ness?

Counter-example: emulation of conditional-move

Before

```
int cmove(int x, int y, bool b) {
  return x + (y−x) * b;
}
```

Counter-example: emulation of conditional-move

Before

```
int cmove(int x, int y, bool b) {
 return x + (y−x) * b;
}
```

After

```
int cmove(int x, int y, bool b) {
 if (b) {
  return y;
 } else {
  return x;
 }
}
```

Counter-example: double-word multiplication

Before

```
long long llmul(long long x, long long y) {
  return x * y;
}
```

$$x = aN + b \qquad y = cN + d \qquad xy = (ad + cb)N + bd \pmod{N^2}$$

Counter-example: double-word multiplication

Before

```
long long llmul(long long x, long long y) {
  return x * y;
}
```

$$x = aN + b \qquad y = cN + d \qquad xy = (ad + cb)N + bd \pmod{N^2}$$

After

```
long long llmul(long long x, long long y) {
  long a = High(x);
  long c = High(y);
  if (a | c) {
     . . .
  } else {
    return Low(x) * Low(y);
  }
}
```

Counter-example: tabulation

Before

```
char rot13(char x) {
  return `a` + (x − `a` + 13) % 26;
}
```

Counter-example: tabulation

Before

```
char rot13(char x) {
  return `a` + (x — `a` + 13) % 26;
}
```

After

```
char rot13(char x) {
  static char table[26] = ``nopqrstuvwxyzabcdefghijklm";
  return table[x — `a`];
}
```

Where we are?

High-assurance and high-speed cryptography

- We need cryptographic libraries that we can trust
- Correct implementation frequently remain vulnerable to side channel attacks
- Efficiency considerations often force developers to carry out very aggressive optimizations

Implementing “crypto”: a subtle equilibrium

- Fast
- Correct (functional correctness, safety)
- Side-channel resistant (constant-time)
- Provably secure

A gap between source and assembly languages

Source

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

A gap between source and assembly languages

Source

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

Assembly

- Efficiency
- Control (instruction selection and scheduling)
- Predictable

A gap between source and assembly languages

- **Assembly** is not programmer/verifier friendly
  - The code is obfuscated
  - More error prone
  - Harder to prove / analyze

- **Source** is not security/efficiency friendly
  - Trust compiler
  - Certified compilers are less efficient
  - Optimizing compilers can break side channel resistance

What are the possible solutions

- Fast
- Correct (functional correctness, safety)
- Side-channel resistant (constant-time)
- Provably secure

Vale: Assembly level

- Prove functional correctness and constant time at the assembly level
- the code can be manually written or generated

Frama-C, Hacl*, ...: Source level

- functional correctness is proved at the source level
- Should trust the compiler or use a certified compiler

Easycrypt

- Provable secure
- But at the algorithm level (no executable code)

Jasmin: last mile of high assurance cryptography

Fast and **formally verified** assembly code

- Source language: assembly in the head with formal semantics
  $\Longrightarrow$ programmer & verification friendly
- Compiler: predictable & formally verified (in Coq)
  $\Longrightarrow$ programmer has control and no compiler security bug
- Verification toolchain (based on EasyCrypt):
  - provable security
  - safety
  - side channel resistance (constant-time)
  - functional correctness

World speed record + full verification

TLS 1.3 components : ChaCha20, Poly1305, Curve25519, Sha3(keccak)

Jasmin by example

Initialization of Chacha20 state

```
inline fn init(reg u64 key nonce, reg u32 counter) → stack u32[16]
{
 inline int i;
 stack u32[16] st;
 reg u32[8] k;
 reg u32[3] n;

 st[0] = 0x61707865;
 st[1] = 0x3320646e;
 st[2] = 0x79622d32;
 st[3] = 0x6b206574;

 for i=0 to 8 {
  k[i] = (u32)[key + 4*i];
  st[4+i] = k[i];
 }

 st[12] = counter;

 for i=0 to 3 {
  n[i] = (u32)[nonce + 4*i];
  st[13+i] = n[i];
 }

 return st;
}
```

Zero-cost abstractions

- Variable names
- Arrays
- Loops
- Inline functions

User control: loop unrolling

```
for i=0 to 15 {
  k[i] = st[i];
}
```

- *For* loops are fully unrolled
- The value of the counter is propagated
- The source code still readable and compact

```
while(i < 15) {
  k[i] = st[i];  i += 1;
}
```

- *While* loops are untouched

User control: register or stack

- Jasmin has two kind of variables:
  - register variables (**reg**)
  - stack variables (**stack**)
- Array can be register array or stack array
- Spilling is done manually (by the user)

```
inline fn sum_states(reg u32[16] k, stack u32 k15, stack u32[16] st) → reg u32[16], stack u32
{
 inline int i;
 stack u32 k14;

 for i=0 to 15 {
  k[i] += st[i];
 }

 k14    = k[14];  // Spilling
 k[15]  = k15;    // Spilling
 k[15] += st[15];
 k15    = k[15];  // Spilling
 k[14]  = k14;    // Spilling

 return k, k15;
}
```

User control: instruction-set architecture level

- Direct memory access

```
reg u64 output, plain;
for i=0 to 12
{ k[i] ^= (u32)[plain + 4*i];
  (u32)[output + 4*i] = k[i]; }
```

- Conditional move

```
x = y if ¬cf;
```

- The carry flag is an ordinary boolean variable

```
reg u64[3] h;
reg bool cf0 cf1;
reg u64 h2rx4 h2r;
      h2r  += h2rx4;
cf0 , h[0] += h2r;
cf1 , h[1] += 0 + cf0;
 _   , h[2] += 0 + cf1;
```

User control : instruction-set architecture level

- Most of assembly instructions are accessible in the source language

```
of, cf ,sf, pf, zf, z = x86_ADC(x, y, cf);
of, cf, x = x86_ROL_32(x, bits);
```

- SIMD instructions

```
k[0] +8u32= k[1];  // vectorized addition of 8 32−bits word;
k[1] = x86_VPSHUFD_256(k[1], (4u2)[0,3,2,1]);
```

The Jasmin compiler

Goals / Features

- Predictability and control of generated assembly
- Preserves semantics (Coq proofs, machine-checked)
- Preserves side-channel resistance

Compilation passes

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

Formal verification:

- Maximal use of validation
- Reuse of a single checker (core) for various passes

Semantic preservation

Theorem

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

Preservation of constant-time’?

Good news . . .
Many compiler optimizations do preserve “constant-time”-ness:

- Constant folding
- Constant propagation
- Variable spilling / Register allocation
- Expression flattening
- Loop peeling
- Pull common instructions out of branches
- Swap independent instructions
- Linearization

Formally verified for a toy language: CSF18

Future work: Jasmin, CompCert

Constant-time: a non-interference property

Decorate the small-step relation with a *leakage*: $\mathrm{a} \xrightarrow{\ell} \mathrm{b}$

Constant-time: a non-interference property

Decorate the small-step relation with a *leakage*: $\mathrm{a} \xrightarrow{\ \ell\ } \mathrm{b}$

Definition (Constant-time):

For every two execution prefixes

$$\begin{tikzcd} i \arrow[r, "\ell_0"] & s_0 \arrow[r, "\ell_1"] & s_1 \arrow[r, "\ell_2"] & s_2 & \cdots \\ i' \arrow[r, "\ell'_0"] & s'_0 \arrow[r, "\ell'_1"] & s'_1 \arrow[r, "\ell'_2"] & s'_2 & \cdots \end{tikzcd}$$

the leakages agree whenever the inputs agree:

$$\psi(i, i') \Rightarrow \ell_0 \cdot \ell_1 \cdot \ell_2 = \ell'_0 \cdot \ell'_1 \cdot \ell'_2$$

Preservation of constant time

$$\begin{tikzcd}
i \arrow[r, "\ell_0"] & s_0 \arrow[r, "\ell_1"] & s_1 \arrow[r, "\ell_2"] & s_2 & \cdots \\
i' \arrow[r, "\ell'_0"] & s'_0 \arrow[r, "\ell'_1"] & s'_1 \arrow[r, "\ell'_2"] & s'_2 & \cdots
\end{tikzcd}$$

Intuition, all leakages at the target level correspond to:

- Correspond to an observation at the source level
- Or can be computed using previous leakage + some constant

$$\begin{tikzcd}
i \arrow[r, "\ell_0"] & s_0 \arrow[r, "\ell_1"] & s_1 \arrow[r, "\ell_2"] & s_2 & \cdots \\
i' \arrow[r, "\ell'_0"] & s'_0 \arrow[r, "\ell'_1"] & s'_1 \arrow[r, "\ell'_2"] & s'_2 & \cdots
\end{tikzcd}$$

Where we are ?

Two correctness theorems:

- Functional correctness can be proved at the source level
  $\Longrightarrow$ certified compiler
- Side channel resistance can be proved at the source level
  $\Longrightarrow$ constant-time preservation

Both theorems assume that programs are safe $\Longrightarrow$ we need to prove safety

We also need to establish at source level:

- Functional correctness
- Side channel resistance

Functional correctness

Using Hoare Logic: $c : P \Rrightarrow Q$

- P is the precondition
- the post-condition Q is a mathematical description

Using Relational Hoare Logic:

- c_1 is the reference implementation (the specification)
- c_2 is the optimized implementation

$$c_1 \sim c_2 : =_{\{\text{args}\}} \Rrightarrow =_{\{\text{res}\}}$$

Combine both of them

EasyCrypt already provides Hoare Logic and Relational Hoare Logic

Functional correctness by game hopping

We have built an EasyCrypt model for Jasmin

Jasmin compiler is able to translate programs into the EasyCrypt syntax

We perform functional correctness proofs by game hopping:

$$c_{\mathrm{ref}} \sim c_1 \sim \ldots \sim c_{\mathrm{opt}}$$

Function correctness by game hopping

Computing sum:

- p is a pointer,
- len the number of 64-bits words to sum

We want to compute : $\sum_{i<len}$ (**u64**)[p+8 * i]

S_0

```
s = 0;
i = 0;
while (i < len) {
  s += (u64)[p + 8 * i];
  i += i
}
return s;
```

Function correctness by game hopping

Computing sum:

- p is a pointer,
- len the number of 64-bits words to sum

We want to compute : $\sum_{i<len}$ (**u64**)[p+8 * i]

S_0

```
s = 0;
i = 0;
while (i < len) {
  s += (u64)[p + 8 * i];
  i += i
}
return s;
```

S_1

```
s = 0;
plen = p + 8 * len;
while (p < plen) {
  s += (u64)[p];
  p += 8;
}
return s;
```

Function correctness by game hopping

Computing sum:

- p is a pointer,
- len the number of 64-bits words to sum

We want to compute : $\sum_{i<len}$ (**u64**)[p+8 * i]

S_0

```
s = 0;
i = 0;
while (i < len) {
  s += (u64)[p + 8 * i];
  i += i
}
return s;
```

S_1

```
s = 0;
plen = p + 8 * len;
while (p < plen) {
  s += (u64)[p];
  p += 8;
}
return s;
```

$$S_0 \sim S_1 \ : \ =_{\{p, len, mem\}} \ \Rrightarrow \ =_{\{s\}}$$

Loop transformations

S_1

```
s = 0;
plen = p + 8 * len;
while (p < plen) {
  s += (u64)[p];
  p += 8;
}
return s;
```

Loop transformations

S_1

```
s = 0;
plen = p + 8 * len;
while (p < plen) {
  s += (u64)[p];
  p += 8;
}
return s;
```

S_2

```
s = 0;
plen = p + 8 * len;
len4 = 4 * (len / 4);
plen4 = p + 8 * len4;
while (p < plen4) {
  s += (u64)[p]; p += 8;
  s += (u64)[p]; p += 8;
  s += (u64)[p]; p += 8;
  s += (u64)[p]; p += 8;
}
while (p < plen) {
  s += (u64)[p];
  p += 8;
}
return s;
```

Loop transformations

S_1

```
s = 0;
plen = p + 8 * len;
while (p < plen) {
  s += (u64)[p];
  p += 8;
}
return s;
```

S_2

```
s = 0;
plen = p + 8 * len;
len4 = 4 * (len / 4);
plen4 = p + 8 * len4;
while (p < plen4) {
  s += (u64)[p]; p += 8;
  s += (u64)[p]; p += 8;
  s += (u64)[p]; p += 8;
  s += (u64)[p]; p += 8;
}
while (p < plen) {
  s += (u64)[p];
  p += 8;
}
return s;
```

S_3

```
s = 0;
plen = p + 8 * len;

if (4 ≤ len) {
  len4 = 4 * (len / 4);
  plen4 = p + 8 * len4;
  s0 = 0; s1 = 0; s2 = 0; s3 = 0;
  while (p < plen4) {
    s0 += (u64)[p]; p += 8;
    s1 += (u64)[p]; p += 8;
    s2 += (u64)[p]; p += 8;
    s3 += (u64)[p]; p += 8;
  }
  s0 += s2;
  s1 += s3;
  s  = s0 + s1;
}
while (p < plen) {
  s += (u64)[p];
  p += 8;
}
return s;
```

Use vector instructions

S_3

```
s = 0;
plen = p + 8 * len;

if (4 ≤ len) {
  len4 = 4 * (len / 4);
  plen4 = p + 8 * len4;
  s0 = 0; s1 = 0; s2 = 0; s3 = 0;
  while (p < plen4) {
    s0 += (u64)[p]; p += 8;
    s1 += (u64)[p]; p += 8;
    s2 += (u64)[p]; p += 8;
    s3 += (u64)[p]; p += 8;
  }
  s0 += s2;
  s1 += s3;
  s  = s0 + s1;
}
while (p < plen) {
  s += (u64)[p];
  p += 8;
}
return s;
```

Use vector instructions

S_3

```
s = 0;
plen = p + 8 * len;

if (4 ≤ len) {
  len4 = 4 * (len / 4);
  plen4 = p + 8 * len4;
  s0 = 0; s1 = 0; s2 = 0; s3 = 0;
  while (p < plen4) {
    s0 += (u64)[p]; p += 8;
    s1 += (u64)[p]; p += 8;
    s2 += (u64)[p]; p += 8;
    s3 += (u64)[p]; p += 8;
  }
  s0 += s2;
  s1 += s3;
  s  = s0 + s1;
}
while (p < plen) {
  s += (u64)[p];
  p += 8;
}
return s;
```

S_4

```
s = 0;
plen = p + 8 * len;

if (4 ≤ len) {
  len4 = 4 * (len / 4);
  plen4 = p + 8 * len4;
  ss0 = 0;  // u256
  while (p < plen4) {
    ss0 +4u64= (u256)[p];
    p += 32;
  }
  ss1  = x86_VPSHUFD_256(ss0,(4u2)[2,3,0,1]); /* s2, s3, s0, s1 */
  ss0 +4u64= ss1;                             /* s0+s2, s1+s3, s2+s0, s3+s1 */
  ss01 = x86_VEXTRACTI128(ss0, 0);  /* s0+s2, s1+s3 */
  s    = x86_VPEXTR_64(ss0, 0);     /* s0+s2 */
  s1   = x86_VPEXTR_64(ss0, 1);     /* s1+s3 */
  s   += s1;                        /* s0 + s2 + s1 + s3 */
}
while (p < plen) {
  s += (u64)[p];
  p += 4;
}
```

Where we are?

Implementing “crypto”: a subtle equilibrium

- Fast
- Safety + functional correctness (✓)
- Side-channel resistant
- Provably secure (EasyCrypt)

Side channel resistance

Constant time is a non-interference property:

$$\begin{tikzcd} i \arrow[r, "\ell_0"] & s_0 \arrow[r, "\ell_1"] & s_1 \arrow[r, "\ell_2"] & s_2 & \cdots \end{tikzcd}$$

$$\begin{tikzcd} i' \arrow[r, "\ell'_0"] & s'_0 \arrow[r, "\ell'_1"] & s'_1 \arrow[r, "\ell'_2"] & s'_2 & \cdots \end{tikzcd}$$

Constant time can be checked using RHL
Idea: introduce ghost variable to model leakage

if (t) c_1 **else** c_2

leaks = t :: leaks;
if (t) $\overline{c_1}$ **else** $\overline{c_2}$

Side channel resistance

Constant time is a non-interference property:

$$\begin{tikzcd} i \arrow[r, "\ell_0"] & s_0 \arrow[r, "\ell_1"] & s_1 \arrow[r, "\ell_2"] & s_2 & \cdots \\ i' \arrow[r, "\ell'_0"] & s'_0 \arrow[r, "\ell'_1"] & s'_1 \arrow[r, "\ell'_2"] & s'_2 & \cdots \end{tikzcd}$$

Constant time can be checked using RHL
Idea: introduce ghost variable to model leakage

```
if (t) c₁ else c₂
```

```
leaks = t :: leaks;
if (t) c̅₁ else c̅₂
```

c is constant time if its model $\overline{c}$ verifies the relational property:

$$\overline{c} \sim \overline{c} : =_{\{\textit{public inputs}\}} \Rrightarrow =_{\{\text{leaks}\}}$$

Verification of constant time

Jasmin compiler can generate a EasyCryptmodel for constant-time

Good news:

$$\overline{c} \sim \overline{c} \; : \; =_{\{\textit{public inputs}\}} \Rightarrow =_{\{\text{leaks}\}}$$

- Can be automatically decided (dependency analysis)
- the **sim** tactic do it for you

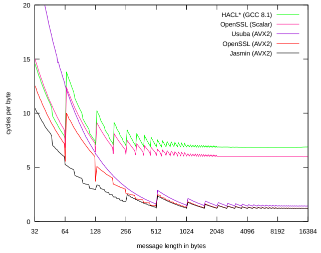
Where we are?

Implementing “crypto”: a subtle equilibrium

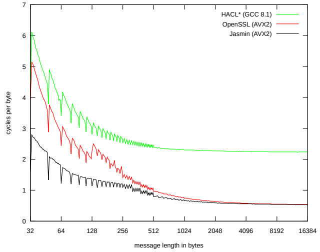
- Fast
- Safety + functional correctness ($\checkmark$)
- Side-channel resistant ($\checkmark$)
- Provably secure (EasyCrypt)

Benchmarks: Comparison to non-verified code

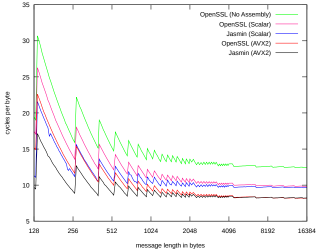
ChaCha20:



Poly1305:

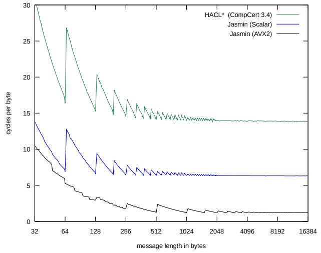


Shake256(SHA3):

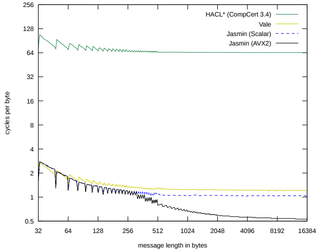


Benchmarks: Comparison to fully verified code

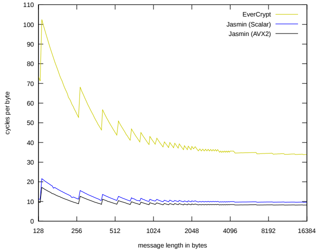
ChaCha20:



Poly1305:



Shake256(SHA3):



Conclusion

If P is safe, functionally correct, constant-time and

$$Adv_A(P) \leq \epsilon$$

then $[\![P]\!]$ is safe, functionally correct, constant-time and

$$Adv_B([\![P]\!]) \leq \epsilon$$

and $[\![P]\!]$ is fast!

Furthermore, safety, functional correctness, “constant-time”-ness can be *easily* checked at the source level

Future works

- Extension of Jasmin: More architectures (ARM)
- Preservation of constant time (Jasmin + CompCert)
- TLS 1.3 primitives
- Automation of equivalence proofs