High-Assurance and High-Speed
Cryptographic Implementations Using the
Jasmin Language

J.B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, P-Y. Strub
Octobre 9th, 2019
Connection Encrypted (TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256, 256 bit keys, TLS 1.2)

The page you are viewing has been encrypted before being transmitted over the Internet.

Encryption makes it difficult for unauthorized people to view information traveling between computers. It is therefore unlikely that anyone read this page as it traveled across the network.
Developing cryptographic libraries is hard, as the code must be:

- **efficient**: pervasive usage, on large amount of data.
- **functionally correct**: the specification must be respected.
- **protected against side-channel attacks**: constant-time implementation.
Side-Channel Attacks

Exploit auxilliary information to break a cryptographic primitive.
**Context**

**Side-Channel Attacks**
Exploit auxilliary information to break a cryptographic primitive.

**Constant-Time Programming**
- Countermeasure against timing and cache attacks.
- Control-flow and memory accesses should not depend on secret data.
- Crypto implementations without this property are vulnerable.
## Constraints

- **Efficiency**: *low-level* operations and *vectorized* instructions.
- **Functional Correctness**: *readable* code, with *high-level abstractions*.
- **Side-Channel Attacks Protection**: *control* over the executed code.
## Gap Between Source and Assembly

### Source

- High-level abstractions.
- **Readable** code.
## Gap Between Source and Assembly

### Source
- **High-level** abstractions.
- **Readable** code.

### Source is not Security/Efficiency Friendly
- Trust compiler (GCC or Clang).
- Certified compilers are less efficient (CompCert).
- Optimizing can break side channel resistance.
Preservation of Constant-Timerness?

**Before**

```c
int cmove(int x, int y, bool b) {
    return x + (y-x) * b;
}
```
Preservation of Constant-Timeness?

**Before**

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

**After**

```c
int cmove(int x, int y, bool b) {
    if (b) {
        return y;
    } else {
        return x;
    }
}
```
### Gap Between Source and Assembly

<table>
<thead>
<tr>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>Efficient</strong> code.</td>
</tr>
<tr>
<td>• <strong>Control</strong> over the program execution.</td>
</tr>
</tbody>
</table>
## Gap Between Source and Assembly

<table>
<thead>
<tr>
<th><strong>Assembly</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- <strong>Efficient</strong> code.</td>
</tr>
<tr>
<td>- <strong>Control</strong> over the program execution.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Assembly is not Programmer/Verifier Friendly</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- The code is obfuscated.</td>
</tr>
<tr>
<td>- More error prone.</td>
</tr>
<tr>
<td>- Harder to prove/analyze.</td>
</tr>
</tbody>
</table>
## Fast and Formally Verified Assembly Code

- **Source language**: assembly in the head with formal semantics
  \[ \Rightarrow \] programmer & verification friendly
- **Compiler**: predictable & formally verified (in Coq)
  \[ \Rightarrow \] programmer has control and no compiler security bug
- **Verification tool-chain**:  
  - Functional correctness.
  - Side-channel resistance (constant-time).
  - Safety.

### Implementations in Jasmin

TLS 1.3 components: ChaCha20, Poly1305, Curve25519.
The Jasmin Language
Initialization of ChaCha20 State

```c
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];
}

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

### For Loops
- Fully unrolled.
- The value of the counter is propagated.
- The source code still readable and compact.

### While Loops
- Untouched.
Jasmin has three kinds of variables:
- register variables (reg).
- stack variables (stack).
- global variables (global).

Arrays can be register arrays or stack arrays.
Spilling is done manually (by the user).

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

    return k, k15;
}
```
• Direct memory access.

```plaintext
reg u64 output, plain;

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

• The carry flag is an ordinary boolean variable.

```plaintext
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;
```
• Most assembly instructions are available.

\[
of, cf, sf, pf, zf, z = x86\_ADC(x, y, cf);
\]
\[
of, cf, x = x86\_ROL\_32(x, bits);
\]

• Vectorized instructions (SIMD).

\[
k[0] +8u32 = k[1]; \quad // \text{vectorized addition of 8 32-bits words;}
\]
\[
k[1] = x86\_VPSHUF D\_256(k[1], (4u2)[0,3,2,1]);
\]
The Jasmin Compiler
The Compiler

Goals And Features

- Predictability and control of generated assembly.
- Preserves semantics (machine-checked in Coq).
- Preserves side-channel resistance
Compilation

Passes and Optimizations

- For loop unrolling.
- Function inlining.
- Constant-propagation.
- Sharing of stack variables.
- Register array expansion.
- Lowering.
- Register allocation.
- Linearisation.
- Assembly generation.
Semantic Preservation

Compilation Theorem (Coq)

\[ \forall p, p'. \text{ compile}(p) = \text{ok}(p') \Rightarrow \]
\[ \forall v_a, m, v_r, m'. \text{enough-stack-space}(p', m) \Rightarrow \]
\[ v_a, m \downarrow^p v_r, m' \Rightarrow v_a, m \downarrow^{p'} v_r, m' \]

Remarks

- The compiler uses validation.
- We may need some extra memory space for \( p' \): 
  \[ \text{enough-stack-space}(p', m) \]
- If \( p \) is not safe, i.e. \( v_a, m \downarrow^p \bot \), then we have no guarantees.
Functional Correctness
## Methodology

- We start from a **readable reference implementation**:
  - Using a mathematical specification (e.g. in $\mathbb{Z}/p\mathbb{Z}$).
  - Or a simple imperative specifications.
- We gradually transform the **reference implem.** into an **optimized implem.**:
  - We prove that each transformation **preserves functional correctness** by equivalence (game-hoping).
- We prove additional properties of the final implementation:
  - **Constant-time** by program equivalence.
  - **Safety** by static analysis.
Functional Correctness

Gradual Transformation

We perform functional correctness proofs by game hopping:

\[ \mathcal{C}_{\text{ref}} \sim \mathcal{C}_1 \sim \ldots \sim \mathcal{C}_n \sim \mathcal{C}_{\text{opt}} \]

EasyCrypt

- Jasmin programs are translated into EasyCrypt programs.
- EasyCrypt model for Jasmin (memory model + instructions).
- Equivalences are proved in EasyCrypt.
Functional Correctness

Relational Hoare Logic

A judgment \( \{ P \} \ c_1 \sim c_2 \ \{ Q \} \) is valid if:

\[
(m_1, m_2) \in P \Rightarrow m_1 \downarrow^{c_1} m'_1 \Rightarrow m_2 \downarrow^{c_2} m'_2 \Rightarrow (m'_1, m'_2) \in Q
\]

Relational Hoare Logic is provided in EasyCrypt.

Example

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

\[
\{ \text{args} \langle m_1 \rangle = \text{args} \langle m_2 \rangle \} \ c_1 \sim c_2 \ \{ \text{res} \langle m_1 \rangle = \text{res} \langle m_2 \rangle \}
\]
Example: ChaCha20

Stream cipher that iterates a *body* on all the blocks of a message.

Reference

```c
while (i < len) {
    chacha_body;
    i += 1;
}
```

Loop tiling

```c
while (i + 4 ≤ len) {
    chacha_body;
    chacha_body;
    chacha_body;
    chacha_body;
    i += 4;
}
```

Scheduling

```c
while (i + 4 ≤ len) {
    chacha_body4_swapped;
    i += 4;
}
```

Vectorization

```c
while (i + 4 ≤ len) {
    chacha_body4_vectorized;
    i += 4;
}
```
Safety
Safety

Definition
A program $p$ is safe under precondition $\phi$ if and only if:

$$\forall (v, m) \in \phi. \; v, m \Downarrow^p \perp$$

Why do we Need Safety?

- If $p$ is safe, its execution never crashes.
- The compilation theorem gives no guarantees if $p$ is not safe.
- Jasmin semantics in Easycrypt assumes that $p$ is safe.
Properties to Check

- Division by zero.
- Variable and array initialization.
- Out-of-bound array access.
- Termination.
- Valid memory access.

Jasmin

Safety is checked automatically by static analysis.
Soundness: $\mathbf{♯}$ over-approximates $\mathbf{X}$ if and only if $\mathbf{X} \subseteq \mathbf{♯}(\mathbf{X})$.
Abstract Interpretation: Abstract Values

Soundness

\[ X^\# \text{ over-approximates } X \text{ if and only if } X \subseteq \gamma(X^\#) \]
Abstract Interpretation: Abstract Values

Soundness

$X^\#$ over-approximates $X$ if and only if $X \subseteq \gamma(X^\#)$
Soundness

\[ X^\# \text{ over-approximates } X \text{ if and only if } X \subseteq \gamma(X^\#) \]
Soundness

\[ X^\# \text{ over-approximates } X \text{ if and only if } X \subseteq \gamma(X^\#) \]
Soundness

\( f^\# \) over-approximates \( f \) if and only if:

\[ \forall X^\#. f \circ \gamma(X^\#) \subseteq \gamma \circ f^\#(X^\#) \]
Abstract Interpretation: Abstract Transformers

\[ y \leftarrow y + 1.5 \]

**Soundness**

\( f^\# \) over-approximates \( f \) if and only if:

\[ \forall X^\#. f \circ \gamma(X^\#) \subseteq \gamma \circ f^\#(X^\#) \]
Abstract Interpretation: Abstract Transformers

Soundness

$f^\#$ over-approximates $f$ if and only if:

$$\forall X^\#. f \circ \gamma(X^\#) \subseteq \gamma \circ f^\#(X^\#)$$
### Features of the Language

Jasmin is a simple language for static analysis:

- No recursion.
- Arrays size are statically known.
- No dynamic memory allocation.
fn load(reg u64 in, reg u64 len) {
    inline int i;
    reg u8 tmp;

    tmp = 0;
    while (len >= 16) {
        for i = 0 to 16 {
            tmp = (u8)[in + i];
        }
        in += 16;
        len -= 16;
    }

    for i = 0 to 16 {
        if i < len {
            tmp = (u8)[in + i];
        }
    }

    return tmp;
}
fn load(reg u64 in, reg u64 len) {
    inline int i;
    reg u8 tmp;
    tmp = 0;
    while (len >= 16) {
        for i = 0 to 16 {
            tmp = (u8)[in + i];
        }
        in += 16;
        len -= 16;
    }
    for i = 0 to 16 {
        if i < len {
            tmp = (u8)[in + i];
        }
    }
    return tmp;
}

Memory Calling Contract

valid-mem_{load}(in_0, len_0) = [in_0; in_0 + len_0]
## Variables in the Abstract Domain

Let $\mathcal{P}$ be a set of pointers. To a variable $x \in \mathcal{V}$, we associate:

- $x \in \mathcal{V}^\#$: its abstract value.
- $x_0 \in \mathcal{V}^\#$: its abstract initial value.
- $\text{pt}_x \subseteq \mathcal{P}$: points-to information.
- $\text{offset}_x \in \mathcal{V}^\#$: its abstract offset.
Variables in the Abstract Domain

Let $\mathcal{P}$ be a set of pointers. To a variable $x \in \mathcal{V}$, we associate:

- $x \in \mathcal{V}^\#$: its abstract value.
- $x_0 \in \mathcal{V}^\#$: its abstract initial value.
- $\text{pt}_x \subseteq \mathcal{P}$: points-to information.
- $\text{offset}_x \in \mathcal{V}^\#$: its abstract offset.

Moreover, for every $p \in \mathcal{P}$, we have:

- $\text{mem}_p \in \mathcal{V}^\#$: memory accesses at $p$ (plus an offset).
Concretization Function

We decompose $x$ into a base pointer $p$ and an offset $\text{offset}_x$:

$$\gamma(\text{pt}_x = \{p\} \land \text{offset}_x = S^\#) = x \mapsto \{p + o \mid o \in \gamma(S^\#)\}$$
**Concretization Function**

We decompose $x$ into a base pointer $p$ and an offset $\text{offset}_x$:

$$\gamma(pt_x = \{p\} \land \text{offset}_x = S\#) = x \mapsto \{p + o \mid o \in \gamma(S\#)\}$$

**Example**

- $\gamma(pt_x = \{p\} \land \text{offset}_x = [32; 63]) = x \mapsto [p + 32; p + 63]$
Concretization Function

We decompose \( x \) into a base pointer \( p \) and an offset \( \text{offset}_x \):

\[
\gamma(\text{pt}_x = \{p\} \land \text{offset}_x = \mathcal{S}^\#) = x \mapsto \{p + o \mid o \in \gamma(\mathcal{S}^\#)\}
\]

Example

- \( \gamma(\text{pt}_x = \{p\} \land \text{offset}_x = [32; 63]) = x \mapsto [p + 32; p + 63] \)
- Abstract transformer:
  - \( \mathcal{S}^\# : \text{pt}_x = \{p\} \land \text{offset}_x = [32; 63] \)
    \[
y \leftarrow x + 16
    \]
  - \( \mathcal{S'}^\# : \text{pt}_y = \{p\} \land \text{offset}_y = [48; 79] \)
Static Analysis

Remark

- In $y \leftarrow x + z$, we can either use $x$ or $z$ as a base pointer.
- In practice, it is never a problem (assembly coding style).
Memory Calling Contract

Let $f$ be a procedure with pointers $\mathcal{P}$. If:

$$\llbracket f \rrbracket^\#(S_{\text{init}}) \triangleq \bigwedge_{p \in \mathcal{P}} \text{mem}_p = S_p^\# \land \ldots$$

Then for every $S_{\text{init}} \subseteq \gamma(S_{\text{init}})$:

$$\text{valid-mem}_f(S_{\text{init}}) \subseteq \bigcup_{p \in \mathcal{P}} \gamma(S_p^\#)$$
Example

- $S^\# : \text{pt}_x = \{p\} \land \text{mem}_p = [0; 127] \land \text{offset}_x = [128; 128 + 16]$

  \[\text{tmp} \leftarrow (u8)[x + 16]\]

- $S'^\# : \text{mem}_p = [0; 127] \cup^\# [128; 128 + 32] = [0; 160]$
fn load(reg u64 in, reg u64 len) {
    inline int i;
    reg u8 tmp;

    tmp = 0;
    while (len >= 16) {
        for i = 0 to 16 {
            tmp = (u8)[in + i];
        }
        in += 16;
        len -= 16;
    }

    for i = 0 to 16 {
        if i < len {
            tmp = (u8)[in + i];
        }
    }

    return tmp;
}
Static Analysis

The Analyzer

- Intervals + Relational domain (polyhedra).
- Basic syntactic pre-analysis.
- Disjunctive domain (using the control flow).
- Simple non-relational boolean abstractions (for bools and initialization).
- Brutal handling of function calls.
### Result

For Poly1305, with signature:

```rust
export fn poly1305_avx2(reg u64 out, reg u64 in, reg u64 len, reg u64 k)
```

We infer the ranges:

<table>
<thead>
<tr>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem_out</td>
<td><code>out + [0; 16]</code></td>
</tr>
<tr>
<td>mem_k</td>
<td><code>k + [0; 32]</code></td>
</tr>
<tr>
<td>mem_len</td>
<td><code>∅</code></td>
</tr>
<tr>
<td>mem_in</td>
<td><code>in + [0; len]</code></td>
</tr>
</tbody>
</table>
Caveat

We manually provide some information to the analyser:

- pointers (input) variables: k, in and out in Poly1305.
- relational (input) variables: len in Poly1305.
Conclusion
Conclusion

Contributions

A framework to build high-speed certified implementations of cryptographic primitives.

- Code is manually optimized.
- Functional correctness is obtained by game hopping.
- Safety and security against timing attacks are proved automatically.
- Efficient implementation of Poly1305, ChaCha20 and Gimli.
Future Works

• More TLS 1.3 primitives.
• More architectures, more general purpose language.
  • procedure calls.
  • register allocation/spilling.
• Certification for safety proofs.