High-Assurance High-Speed Cryptography Implementations in Jasmin

Vincent Laporte; Benjamin Grégoire
Manuel Barbosa; François Dupressoir; Pierre-Yves Strub; Tiago Oliveira

2022-07-07, Cyber in Nancy
Cryptography Implementations

High-level specification of protocols

Implementation

Hardware-level security
Example: symmetric encryption scheme

▶ k : key
▶ n : nonce
▶ m : plain-text message
▶ c : cyphertext

c := Enc(k, n, m)
m' := Dec(k, n, c)
Example: symmetric encryption from a PRF

\[ \text{Enc}(k, n, m) = m \oplus f(k, n) \]
\[ \text{Dec}(k, n, c) = c \oplus f(k, n) \]
### Requirements

1. **Efficiency**
2. **Correctness**
3. **Safety**
4. **Confidentiality**
   - against a PPT adversary (cryptographic security)
   - even in presence of side-channels (implementation security)
Requirements

- Efficiency
- Correctness
- Safety
- Confidentiality
  - against a PPT adversary (*cryptographic* security)
  - even in presence of side-channels (*implementation* security)
Efficiency

- CPU cycles matter
- This can be assessed experimentally (through measurements)
- No *formal* efficiency in this lecture
Correctness

Knowing the secret key allows to recover the plaintext:

$$\text{Dec}(k, n, \text{Enc}(k, n, m)) = m$$

Classical functional verification

Relies on the (formal) **semantics** of the programming language.
Running the program:

- terminates
- does not crash (division by zero...)
- does not access arrays out of bounds, uninitialized variables
- i.e., has a properly defined behavior

Programs are usually not safe. Only under some precondition.
Cryptographic security (IND$\text{-}\text{CPA})$

\[ \text{Game IND$\text{-}\text{CPA-Real}_A(\cdot)$} \]
\begin{align*}
&k \leftarrow K \\
b \leftarrow A^{\text{RealEnc}(\cdot,\cdot)}(\cdot) \\
\text{Return } b \\
\text{proc RealEnc}(n, m) \\
\text{Return } \text{Enc}(k, n, m)
\end{align*}

\[ \text{Game IND$\text{-}\text{CPA-Ideal}_A(\cdot)$} \]
\begin{align*}
b \leftarrow A^{\text{IdealEnc}(\cdot,\cdot)}(\cdot) \\
\text{Return } b \\
\text{proc IdealEnc}(n, m) \\
c \leftarrow C \\
\text{Return } c
\end{align*}

Security requires the following advantage measure to be small

\[ | \Pr \left[ \text{IND$\text{-}\text{CPA-Real}_A(\cdot)$} \Rightarrow \text{true} \right] - \Pr \left[ \text{IND$\text{-}\text{CPA-Ideal}_A(\cdot)$} \Rightarrow \text{true} \right] | \]

- Can be done using relational verification of probabilistic programs (e.g., using the EasyCrypt proof assistant).
- Not covered in this lecture.
Implementation security

An adversary can observe some effects of the program execution beyond its result:

- execution time
- electromagnetic emissions
- noise
- effect on the branch predictor
- effect on the memory cache

Can any sensitive information be learned by means of these side-channels?
Example Encryption Scheme in Jasmin

Look at nbaesenc.jazz
The Jasmin tool-box

- Unit test
- Safety checker
- Security against side-channel attacks
- No crash
- Termination
- Constant-time checker
- No crash
- Reference interpreter
- Functional correctness
- Cryptographic security
- Extraction of EasyCrypt models
- Compiler
- Constant-time security
- x86-64 assembly

Jasmin source

- Compiler
- Extraction of EasyCrypt models
- Compiler
- Compiler
Executable semantics

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

$$f : (\bar{a}, m) \downarrow_p (\bar{r}, m')$$

This is the definition of the program behaviors (formalized in Coq).

All proofs are made relative to this definition.

We gain trust by using it (execute & verify programs, verify static analyses, verify program transformations, ... )
Safety: \texttt{jasminc -checksafety} \ldots

- Programs are usually \textbf{not} safe
  - Restrictions on the initial state
- Returns a sufficient pre-condition for safety (a predicate)
- Overapproximation because of undecidability
- The design of the programming language encourages the use of high-level features that make safety verification doable automatically

When the safety checker infers precondition $P$ (for a function $f$ in program $p$), then for all initial state satisfying this precondition, there exists a corresponding final state:

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

No formal proof of this property.
Compiler
Compiler correctness

Forward simulation

If the compilation of source program $S$ succeeds and produces target program $T$, if from the initial state $i$, $S$ terminates with final result $r$, then from the same initial state $i$, $T$ also terminates with final result $r$.

Overlooked details

- Initial states may not be the same
  - Global data must be in the target memory
  - The “stack pointer” (RSP) must point to a valid region of memory
- The target stack must be large enough
  - i.e., the compiler does not enforce the absence of “stack overflow”
Preservation of functional properties

Compiler correctness implies

If a property holds for all source behaviors, then it holds for all target behaviors.

When the source program is a \textit{function} (deterministic, terminating)

Then the target program is the \textit{same} function.

- Cryptographic primitives are usually functions
- even PRNGs!
Probabilities

When a function consumes random data

Reasoning about the *distribution* of the results in terms of the distribution of the inputs can be done at the source level.

Probabilistic properties of functions are preserved (example: IND$-CPA)

Given a secret key, an adversary cannot distinguish (with non-negligible probability) the encryption function from random sampling.

This property is independent of the implementation.

▶ Unless the adversary has access to non-functional properties of the implementation
Non-preservation

Non-deterministic programs

A correct compiler may not preserve distributions. For instance, a source program that tosses a coin may be correctly compiled to the constant program that always returns heads.

Changing the representation of values

E.g., booleans implemented as 63-bit machine integers.

\[ S : b \mapsto \neg b \quad T : n \mapsto 1 - n \]

How to map invalid target values to source values?
There is no way to express at the source level the target behavior.

Non-functional properties

The theorem does not say anything about things that cannot be described by behaviors.
Constant-time: `jasminc -checkCT` ...

The compiler (always) preserves the constant-time property.

Formal (machine-checked) proof of this statement for version 21.0 of the compiler (Barthe et al. 2021).

This is a stronger property than compiler-correctness.
Reasoning about semantics of source programs is better done in a dedicated proof assistant.

Extract an EasyCrypt model from a Jasmin source program.

For safe inputs to the Jasmin program, the EasyCrypt program computes the same outputs (as the Jasmin program).

No formal proof of this statement.
Summary

- A language
- A compiler
- Safety is a key property
- Reasoning at the source level is valid
  - about safety; functional correctness (cryptographic security); constant-time security
  - Coq proofs justify this claim
Overview of the Jasmin programming language

Look at aes.jinc.
### Low-level control

<table>
<thead>
<tr>
<th>Function inlining</th>
<th>Vector (SIMD) instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>inline fn or #inline calls</td>
<td>No automatic vectorization</td>
</tr>
<tr>
<td></td>
<td>Convenient syntax for most operations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loop unrolling</th>
<th>Intrinsic instructions &amp; flag registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>for loops: unrolled</td>
<td>jasmnc -help-intrinsics to get the list</td>
</tr>
<tr>
<td>while loops: preserved</td>
<td>Flags are plain variables</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Storage class</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>param, inline: compile-time use only</td>
<td></td>
</tr>
<tr>
<td>global, stack: memory</td>
<td></td>
</tr>
<tr>
<td>reg: registers</td>
<td></td>
</tr>
</tbody>
</table>
Common uses of intrinsics & flags

- Initialize to zero using a XOR: \#set0
- Branch on the result of an arithmetic operation
- A single comparison with more than two outcomes

See src/low-level.jazz
Quizz (see src/array.jazz)

Can we tell something about the first returned value?

```jazz
1 // Defines fn f(reg u8 x y) −→ reg u8
2 require "array.jinc"
3
4 inline
5 fn quizz0(reg u8 x) −→ reg u8, reg u8 {
6   reg u8 r, y;
7
8   r = 0;
9   y = f(r, x);
10  return r, y;
11 }
```
Can we tell something about the first returned value?

```jasm
1 // Defines fn f(reg u8 x y) → reg u8
2 require "array.jinc"
3
4 inline
5 fn quizz0(reg u8 x) → reg u8, reg u8 {
6    reg u8 r, y;
7
8    r = 0;
9    y = f(r, x);
10   return r, y;
11 }
```

```jasm
1 // Defines fn g(stack u8[1] x, reg u8 y) → stack u8[1]
2 require "array.jinc"
3
4 inline
5 fn quizz1(reg u8 x) → stack u8[1], stack u8[1] {
6    stack u8[1] r, y;
7
8    r[0] = 0;
9    y = g(r, x);
10   return r, y;
11 }
```
Arrays: an explicit and powerful way to structure memory

Things made easier

- Modular reasoning is possible
- Sizes are explicit
  - Useful for proving safety
- Alias analysis is trivial
  - Array may overlap only when they have the same name

Caveat

Ensuring call-by-value semantics without copy is tricky (the compiler rejects programs)
Practice tomorrow

- Expand the encryption scheme to messages of 256 bits
- Prove constant-time security of small low-level programs
Constant-Time Security?

Threat: any shared component is a communication channel

Aim: protect against remote cache-based timing side-channel attacks

**Attack scenarios**

- A distant server takes time to answer
- Two clients of a (trusted) cloud provider
  - Processes are isolated (virtual memory, hypervisor, ...)
- Two websites in a browser
- Two apps running on the same device
Adversary model aka leakage model

The adversary learns from the victim program (whose code is public):

- the sequence of executed instructions (*program counter security*)
- the sequence of accessed memory addresses
  - sometimes only the cache-line (i.e., the least significant bits are kept secret)

and also:

- (size of) operands to some operations (division, floating-point arithmetic, ...)
- values of local variables on function return
- ...

Protection

Ensure that the leakage is independent of secrets
Constant-time select

In order to replace a conditional expression \( r = c ? t : f \):

- compute operations from both branches
- chose the right result depending on the condition

**Using arithmetic**

\[
r = (t \times m) + (f \times (1 - m))
\]

**Using a conditional move instruction**

\[
r = f; \quad r = t \text{ if } c;
\]

See `src/ctselect.jazz`
Enforcement by typing

See src/max.jazz

**Security type annotations**

Tell which inputs are public (and which are not). Security types are ordered (aka two-point lattice)

**Typing rules**

Result of operations is not lower than the level of arguments
Array indices and branch/loop conditions must be public
Note: usually points-to analysis is hard
This can become costly

- Load the whole array
- Rewrite the whole array
- See src/copyMAC.jazz

To avoid table lookups:

- Bit-slicing
- Hardware support (e.g., AES-NI)
- Take constant-time security into account during design
Declassification

Look at nbaesenc.jazz again.

- Sometimes, we need to argue that a tainted value is public
A programming language and associated tools for writing & verifying low-level implementations.

https://formosa-crypto.org/

Thanks for your attention.
Literature