Certified Compilation for High-Assurance Cryptography

Vincent Laporte (Inria Nancy)

68NQRT webinar – 2020-12-03
Outline

High-Assurance Cryptography

Certified compilation

Preservation of non-functional properties

High-performance vs. certified compilation

Concluding remarks
High-Assurance Cryptography
Cryptography

An exciting field

> Building block for security mechanisms
> Real life applications (telecommunication, privacy, voting, currencies, etc.)
> Interesting mathematics
> Unusual to program
Cryptography

An exciting field

- Building block for security mechanisms
- Real life applications (telecommunication, privacy, voting, currencies, etc.)
- Interesting mathematics
- Unusual to program

Good crypto implementations achieve:

- correctness
- semantic security
- efficiency
- protection against side-channel attacks
- ...
Correctness

Interoperability, usefulness

- **Signature** the verification of a legit signature succeeds (always)
- **Encryption** the recipient should be able to decrypt
- **Poly1305** evaluates a polynomial (the message) in the field $\mathbb{F}_{2^{130} - 5}$

- Standard program verification
- Non-elementary mathematics
Semantic security

The actual purpose of these cryptographic routines

**Signature.** the verification of a forged signature fails (with high probability)

**Encryption** given a ciphertext, an adversary cannot learn anything about the corresponding plaintext (IND-CCA etc.)

**Poly1305** no adversary can forge an authenticated message, except by guessing (SUF-CMA)

- Probabilities
- Proofs on open programs
Efficiency

CPU cycles matter

- 300,000 (signed) transactions per day in bitcoin
- Live video streaming (encrypted & decrypted)
- Full-disk encryption
  - A program (e.g., a web browser) must be decrypted before it is launched...
  - ...in order to prevent a theft to know whether one uses links or lynx
  - Thus decryption cost must be negligible
Implementation security

Any shared component creates a communication channel: information will leak through such a channel.

▶ Different clients of a cloud provider may share a CPU
▶ Different tabs in a web browser may share the *event loop*
▶ Different applications on a smartphone share *a lot*

To avoid side-channel attacks, leaked information should not be sensitive.
Constant-time

Programming guide-line: control flow and memory accesses should not depend on sensitive data.

Efficient counter-measure against remote timing and cache attacks.

A relational property: any two executions with similar public inputs run the same instructions and access the same memory addresses (whatever the secret inputs are).
Formal verification of cryptographic implementation

- Usual program verification
- Probabilistic properties
- Relational properties
- Properties should hold at the lowest level (the code that runs)
Certified compilation
A compiler, with a theorem
Bad things happen at the target level only if bad things can happen at the source level. Removes the need to reason on the target program.

Caveat
The only compiler that has this property for all kinds of bad things is the identity. Balance between allowed transformations and preserved properties.
Examples

**CompCert (X. Leroy et alii, ca. 2008)**
A family of compilers from CompCert-C to assembly, for a few architectures: PowerPC, ARM, RISC-V, and x86.

**Jasmin (B. Schmidt et alii, ca. 2017)**
A low-level programming language designed for implementing high-assurance cryptography.
Backwards simulation
If the compilation of source program \( S \) succeeds and produces target program \( T \), if \( T \) exhibits some behavior \( b \), then \( b \) is a possible behavior of \( S \).

Behavior (examples)
- Sequence (eventually infinite) of system calls
- Initial state and final state

Refinement
The target behavior may also *improve* upon a source behavior (in case the source program is *unsafe*).
A simpler theorem (Jasmin)

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 $r$.

Remarks
Forward simulation is stronger than backward simulation but does not hold (usually) for non-deterministic source languages.
Preservation of functional properties

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

When the source program is a function (deterministic, terminating)
Then the target program is the 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: SUF-CMA)
Given a secret key, the probability that an adversary, outputs a message $M$ with a valid authentication tag $t$ is negligible.
The adversary has access to an oracle that authenticates messages.
The adversary wins only if $t$ has not been produced by the oracle for $M$.

The oracle calls the function being compiled (and does some logging).
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.
Preservation of non-functional properties
Wish-list

Correct compilers preserve
▶ functional correctness
▶ semantic security

Can we use a certified compiler to reason on source code about:
▶ implementation security (side channel leaks)
▶ WCET (worst-case execution time)

(Credit: https://saint-nicolas.nancy.fr)
More precise behaviors

- Execution traces are extended with events that describe execution time, leakage through side-channels, etc.
- They enable to describe non-functional properties:
  - Execution time may be the length of the trace
  - Constant-time programs produce a trace that does not depend secret inputs
- These events are erased when talking about compilation correctness

- Can the compilation theorem describe how extended traces are transformed?
Preservation of constant-time

“Cube” lemma: sufficient condition for constant-time preservation (CSF 2018)
Preservation of constant-time

“Cube” lemma: sufficient condition for constant-time preservation (CSF 2018)

CompCert-CT (POPL 2020)

- There are simpler proof arguments; no need for *relational* reasoning
- Because many compilation passes satisfy stronger preservation properties
  - Leakage is preserved
  - Leakage is erased (or preserved)
  - Leakage is deterministically transformed
Deterministic transformation of trace

A strong property

- Implies correctness
  - if the transformation preserves external events
- Implies preservation of constant-time
  - if the transformation only depends on public data

A nice proof technique

- Compositional
- Not relational
- Works also with big-step semantics

Work in progress

Apply this proof technique to the Jasmin compiler
Application to preservation of execution time

The trace transformation describes how the cost of a source operation is mapped to the cost of corresponding target operations.

In theory, we can find linear bounds $k$ such that:

$$\forall i, cost(T, i) \leq k \cdot cost(S, i)$$

In practice, can we infer precise bounds?
High-performance vs. certified compilation
To appear

“Certified Compilation for Cryptography: Extended x86 Instructions and Constant-Time Verification”

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Vincent Laporte, and Tiago Oliveira

IndoCrypt, December 2020.
Common wisdom

It is said that certified compiler emit *inefficient* code.

*Performance of the generated code is decent but not outstanding: on PowerPC, about 90% of the performance of GCC version 4 at optimization level 1.*

(https://compcert.org/compcert-C.html, December 2020)

- Optimizations must be proved correct
- Proving them is hard and tedious
Experimental study on cryptographic implementations

“SUPERCOP is a toolkit developed by the VAMPIRE lab for measuring the performance of cryptographic software”

https://bench.cr.yp.to/supercop.html

- How much is the run-time cost of using CompCert for C implementation of cryptographic software?
- Where does it come from?

Long story short: coverage is key.
Extended instruction sets

Microprocessors have features that are not exposed in the CompCert-C language:

- Wide registers
  - 128 bits (SSE, 1999)
  - 256 bits (AVX, 2011)
- Vectorized instructions (SIMD)
- Cryptographic instructions
  - AES-NI
  - ARM cryptographic extensions
  - even in some RISC-V processors
A generic extension of CompCert, able to soundly compile programs using extended instruction sets.

- Wide registers
- Support for intrinsics
- Correctness theorem

The precise semantics of intrinsics is not needed for the correctness theorem; only to reason about particular programs.
Coverage

Table 2. SUPERCOP coverage statistics for various compilers.

<table>
<thead>
<tr>
<th>architecture operations</th>
<th>baseline</th>
<th>x86-32</th>
<th>ccomp-2.2</th>
<th>ccomp-ext</th>
<th>ccomp-3.0</th>
<th>amd64</th>
<th>baseline</th>
<th>ccomp-3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>aead</td>
<td>343</td>
<td>258</td>
<td>290</td>
<td>178</td>
<td></td>
<td>506</td>
<td>269</td>
<td></td>
</tr>
<tr>
<td>auth</td>
<td>16</td>
<td>10</td>
<td>10</td>
<td>8</td>
<td></td>
<td>19</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>box</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

- It’s not only about optimizing programs.
- It’s also about letting programmers write optimized programs.
Run-time performance

Execution time ratios, compared to the best available (non-certified) compiler:

| operation     | $|P|$ | ccomp-2.2 | ccomp-ext |
|---------------|-----|----------|-----------|
| aead.decrypt  | 166 | 22.03    | 5.39      |
| aead.encrypt  | 166 | 24.70    | 5.49      |
| hash          | 32  | 8.13     | 5.01      |

(Caveat: we use an exponential scale to make the results more impressive.)

- For AEAD, about half of the performance penalty is due to the (non-)support for extended instructions.
- There is room for more improvement.
Checking constant-time

Does CompCert-SIMD preserves the constant-time property?

(I don’t know, but I bet it doesn’t.)

CompCert-SIMD provides a static analyzer for constant-time at the Mach level (the intermediate representation just before assembly).
Concluding remarks
Certified compilers are a big opportunity for high-assurance cryptographic implementations:

- they legitimate the reasoning on source code about correctness & semantic security
- even when talking about probabilistic properties
- under some circumstances, we can also trust them for implementation security (i.e., they preserve counter-measures against side-channel attacks)

Certified compilers should allow the programmers to use all features of the target architecture.
Research directions

Can a certified compiler prove useful bounds on run-time cost increase?

What are the performance critical optimizations still missing in certified compilers?
Thanks