Verification of constant-time implementation in a verified compiler toolchain

David Pichardie
Cache timing attacks

- Common side-channel: Cache timing attacks
- Exploit the latency between cache hits and misses
- Attackers can recover cryptographic keys
  - Tromer et al (2010), Gullasch et al (2011) show efficient attacks on AES implementations
- Based on the use of look-up tables
  - Access to memory addresses that depend on the key
Constant-time programs

Characterization

• Constant-time programs do not:
  • branch on secrets
  • perform memory accesses that depend on secrets

• There are constant-time implementations of many cryptographic algorithms: AES, DES, RSA, etc
Constant-time programs

Example
Constant-time programs

Example

```java
boolean testPIN(int code[]) {
    for (int i=0; i<N; i++) {
        if (code[i] != secret[i]) return false;
    }
    return true;
}
```
Constant-time programs

Example

```java
boolean testPIN(int code[]) {
    for (int i=0; i<N; i++) {
        if (code[i] != secret[i]) return false;
    }
    return true;
}
```

Not constant-time
Constant-time programs

Example

```java
boolean testPIN(int code[]) {
    for (int i=0; i<N; i++) {
        if (code[i] != secret[i]) return false;
    }
    return true;
}
```

```java
boolean testPIN(int code[]) {
    int diff = 0;
    for (int i=0; i<N; i++) {
        diff = diff | (code[i] ^ secret[i]);
    }
    return (diff == 0);
}
```

Not constant-time
Constant-time programs

Example

```java
boolean testPIN(int code[]) {
    for (int i=0; i<N; i++) {
        if (code[i] != secret[i]) return false;
    }
    return true;
}
```

Not constant-time

```java
boolean testPIN(int code[]) {
    int diff = 0;
    for (int i=0; i<N; i++) {
        diff = diff | (code[i] ^ secret[i]);
    }
    return (diff == 0);
}
```

Constant-time
Verification of constant-time programs

Challenges

• Provide a mechanism to formally check that a program is constant-time
  • static tainting analysis for implementations of cryptographic algorithms

• At low level implementation (C, assembly), advanced static analysis is required
  • secrets depends on data, data depends on control flow, control flow depends on data…

• A high level of reliability is required
  • semantic justifications, Coq mechanizations…

• Attackers exploit executable code, not source code
  • we need guaranties at the assembly level using a compiler toolchain
Background: verifying a compiler

CompCert, a moderately optimizing C compiler usable for critical embedded software

= compiler + proof that the compiler does not introduce bugs

Using the Coq proof assistant, X. Leroy proves the following semantic preservation property:

For all source programs $S$ and compiler-generated code $C$, if the compiler generates machine code $C$ from source $S$, without reporting a compilation error, then «$C$ behaves like $S$». 
Background: verifying a compiler

CompCert, a moderately optimizing C compiler usable for critical embedded software

= compiler + proof that the compiler does not introduce bugs

Using the Coq proof assistant, X. Leroy proves the following semantic preservation property:

For all source programs S and compiler-generated code C, if the compiler generates machine code C from source S, without reporting a compilation error, then «C behaves like S».

does not deal with the constant-time security property!
CompCert: 1 compiler, 11 languages

Optimizations: constant prop., CSE, tail calls, (LCM), (software pipelining)

CompCert C  \rightarrow  Clight  \rightarrow  C#minor

RTL  \rightarrow  CminorSel  \leftarrow  Cminor

LTL  \rightarrow  LTLin  \rightarrow  Linear

ASM  \rightarrow  Mach
CompCert: 1 compiler, 11 languages

Optimizations: constant prop., CSE, tail calls, (LCM), (software pipelining)

Where should we perform the constant time analysis?
Our approach

1. Analyse the program at source level

   Sandrine Blazy, David Pichardie, Alix Trieu.  
   *Verifying Constant-Time Implementations by Abstract Interpretation.*  
   ESORICS 2017.

2. Make the compiler preserve the property

   *Formal verification of a constant-time preserving C compiler.*  
   POPL 2020.
Constant-time analysis at source level

Sandrine Blazy, David Pichardie, Alix Trieu.
*Verifying Constant-Time Implementations by Abstract Interpretation.*
ESORICS 2017.

We perform static analysis at (almost) C level
- Based on previous work with a value analyser, Verasco
- We mix Verasco memory tracking with fine-grained tainting
The Verasco project
INRIA Celtique, Gallium, Antique, Toccata + VERIMAG + Airbus
ANR 2012-2016

Goal: develop and verify in Coq a realistic static analyzer by abstract interpretation

- Language analyzed: the CompCert subset of C
- Nontrivial abstract domains, including relational domains
- Modular architecture inspired from Astrée’s
- To prove the absence of undefined behaviors in C source programs

Slogan:

- if « CompCert \(\approx 1/10\)th of GCC but formally verified »,
- likewise « Verasco \(\approx 1/10\)th of Astrée but formally verified »
Verified Static Analysis
Verified Static Analysis

Logical Framework
(Coq)
Verified Static Analysis

- Logical Framework (Coq)
- Language Semantics (CompCert C)
Verified Static Analysis

Analyzer Implementation (manual)

Language Semantics (CompCert C)

Logical Framework (Coq)
Verified Static Analysis

- Analyzer Implementation (manual)
- Analyzer Spec. (abstract interp. methodology)
- Language Semantics (CompCert C)

Logical Framework (Coq)
Verified Static Analysis

- Analyzer Implementation (manual)
- Analyzer Spec. (abstract interpr. methodology)
- Language Semantics (CompCert C)
- Soundness Proof (mostly interactive)
- Logical Framework (Coq)
Verified Static Analysis

Analyzer Implementation (manual)
Analyzer Spec. (abstract interpr. methodology)
Language Semantics (CompCert C)

Soundness Proof (mostly interactive)

Logical Framework (Coq)

extraction

analyzer .exe
Verasco
A Formally-Verified C Static Analyzer


Verasco
Abstract numerical domains

- CompCert C ➔ Clight ➔ C#minor ➔ ... ➔ CompCert compiler

- Alarms ➔ Abstract interpreter ➔ control flow

- Memory & value domain ➔ states

- Z → int ➔ numbers
  - Convex polyhedra
  - Symbolic equalities
  - Nonrel → Rel
  - Integer congruences
  - Nonrel → Rel
  - Integer & F.P. intervals

- CompCert compiler
Verasco
Abstract numerical domains

CompCert C → Clight → C#minor → ... → CompCert compiler

Alarms ← Abstract interpreter

Memory & value domain

Control flow

Z → int

Convex polyhedra
Symbolic equalities
Nonrel → Rel
Nonrel → Rel

Integer congruences
Integer & F.P. intervals

VERIMAG work

VERIMAG work
Verasco
Abstract numerical domains

CompCert C → Clight → C#minor → ... → CompCert compiler

Alarms → Abstract interpreter → control flow

Memory & value domain → states

\[ Z \rightarrow \text{int} \]

numbers

Convex polyhedra
Symbolic equalities
Nonrel → Rel
Nonrel → Rel

VERIMAG work
Integer congruences
Integer & F.P. intervals
Verasco
Abstract numerical domains

CompCert C → Clight → C#minor → ... → CompCert compiler

Alarms → Abstract interpreter → control flow

Memory & value domain → states

Z → int →

Convex polyhedra

Symbolic equalities

Nonrel→ Rel

Integer congruences

Nonrel→ Rel

Integer & F.P. intervals

conjunctions of linear inequalities $\sum a_i x_i \leq c$

[SAS’13]

VERIMAG work

CompCert compiler
Verasco
Abstract numerical domains

- Symbolic conditional expressions (improve precision of assume commands)
- Convex polyhedra
- Symbolic equalities
- Integer congruences
- Integer & F.P. intervals

VERIMAG work

CompCert C → Clight → C#minor → ... → CompCert compiler

Alarms → Abstract interpreter → control flow

Memory & value domain → states

Z → int

Nonrel → Rel

Nonrel → Rel

C#minor

Clight

CompCert C

VERIMAG work
Verasco
Abstract numerical domains

CompCert C → Clight → C#minor → ... → CompCert compiler

Alarms → Abstract interpreter → control flow

Memory & value domain → states

Z → int ➜ transforms any non-rel. domain into a (reduced) rel. domain

Nonrel → Rel

Convex polyhedra ➜ Symbolic equalities ➜ Integer congruences ➜ Integer & F.P. intervals

Nonrel → Rel

VERIMAG work
Verasco
Abstract numerical domains

CompCert C → Clight → C#minor → ... → CompCert compiler

Abstract interpreter

Alarms

control flow

Memory & value domain

states

Z → int

crucial to analyze the safety of memory accesses (memory alignment)

VERIMAG work

Convex polyhedra

Symbolic equalities

Nonrel→Rel

Integer congruences

Nonrel→Rel

Integer & F.P. intervals

C#minor...
Verasco
Abstract numerical domains

CompCert C → Clight → C#minor → ... → CompCert compiler

Alarms → Abstract interpreter (control flow)
Memory & value domain (states)

Z \rightarrow \text{int}

Convex polyhedra, Symbolic equalities, Nonrel \rightarrow \text{Rel}, Nonrel \rightarrow \text{Rel}, Integer congruences, Integer & F.P. intervals

VERIMAG work

requires reasoning on double-precision floating-point numbers (IEEE754)
Verasco
Abstract numerical domains

- CompCert C → Clight → C#minor → ...
  CompCert compiler

- Alarms → Abstract interpreter → control flow

- Memory & value domain → states

- Z → int
  custom reduced product

- Convex polyhedra
- Symbolic equalities

- Nonrel → Rel
- Integer congruences
- Integer & F.P. intervals

VERIMAG work
Verasco
Abstract numerical domains

VERIMAG work

Convex polyhedra
Symbolic equalities
Nonrel → Rel
Integer congruences
Nonrel → Rel
Integer & F.P. intervals
Verasco
Implementation

34 000 lines of Coq, excluding blanks and comments
- half proof, half code & specs
- plus parts reused from CompCert

Bulk of the development: abstract domains for states and for numbers (involve large case analyses and difficult proofs over integer and floating points arithmetic)

Except for the operations over polyhedra, the algorithms are implemented directly in Coq’s specification language.
Constant-time analysis at source level

... → C#minor → ... → CompCert compiler

Abstract interpreter → control flow

Memory & value domain → states

Numerical domain → numbers
Constant-time analysis at source level

We design an abstract functor
Constant-time analysis at source level

We design an abstract functor

- takes as input an abstract memory domain

\[
\begin{align*}
[e]^\# : & M^\# \rightarrow V^\# \\
[x \rightarrow e]^\# : & M^\# \rightarrow M^\# \\
[*e_1 \rightarrow e_2]^\# : & M^\# \rightarrow M^\# \\
[x \rightarrow *e]^\# : & M^\# \rightarrow M^\# \\
\text{assert}(e)^\# : & V^\# \rightarrow M^\# \\
\text{concretize}^\# : & \forall^\# \rightarrow P(L)
\end{align*}
\]
Constant-time analysis at source level

We design an abstract functor

- takes as input an abstract memory domain

\[
\begin{align*}
[e] : & M^\# \to V^\#
\\
[x \to e] : & M^\# \to M^\#
\\
[*e_1 \to e_2] : & M^\# \to M^\#
\\
[x \to *e] : & M^\# \to M^\#
\\
assert(e) : & M^\# \to M^\#
\\
concretize : & \forall^\# \to P(\mathcal{L})
\end{align*}
\]

Abstract interpreter

Taint domain

Memory & value domain

Numerical domain

CompCert compiler

control flow

taints

states

numbers
Constant-time analysis at source level

We design an abstract functor

- takes as input an abstract memory domain
- returns an abstract domain that taints every memory cells

\[
\begin{align*}
[e]^\#: & \quad M^\# \rightarrow V^\# \\
[x \rightarrow e]^\#: & \quad M^\# \rightarrow M^\# \\
[*e_1 \rightarrow e_2]^\#: & \quad M^\# \rightarrow M^\# \\
[x \rightarrow *e]^\#: & \quad M^\# \rightarrow M^\# \\
\text{assert}(e)^\#: & \quad M^\# \rightarrow M^\# \\
\text{concretize}^\#: & \quad V^\# \rightarrow P(L)
\end{align*}
\]

\[
\begin{align*}
\mathcal{T}[e]^\#: & \quad M_{\text{taint}}^\# \rightarrow V_{\text{taint}}^\# \\
\mathcal{T}[x \rightarrow e]^\#: & \quad M^\# \rightarrow M_{\text{taint}}^\# \rightarrow M_{\text{taint}}^\# \\
\mathcal{T}[*e_1 \rightarrow e_2]^\#: & \quad M^\# \rightarrow M_{\text{taint}}^\# \rightarrow M_{\text{taint}}^\# \\
\mathcal{T}[x \rightarrow *e]^\#: & \quad M^\# \rightarrow M_{\text{taint}}^\# \rightarrow M_{\text{taint}}^\#
\end{align*}
\]
Constant-time analysis at source level

We design an *abstract functor*

- takes as input an abstract memory domain
- returns an abstract domain that taints every memory cell

\[
\begin{align*}
[e] : & \quad M \rightarrow V \\
[x \rightarrow e] : & \quad M \rightarrow M \\
[*e_1 \rightarrow e_2] : & \quad M \rightarrow M \\
[x \rightarrow *e] : & \quad M \rightarrow M \\
assert(e) : & \quad M \rightarrow M \\
concretize : & \quad V \rightarrow P(L)
\end{align*}
\]

- set of concrete memory locations

\[
\begin{align*}
\mathcal{T}[e] : & \quad M_{taint} \rightarrow V_{taint} \\
\mathcal{T}[x \rightarrow e] : & \quad M_{taint} \rightarrow M_{taint} \rightarrow M_{taint} \\
\mathcal{T}[*e_1 \rightarrow e_2] : & \quad M_{taint} \rightarrow M_{taint} \rightarrow M_{taint} \\
\mathcal{T}[x \rightarrow *e] : & \quad M_{taint} \rightarrow M_{taint} \rightarrow M_{taint}
\end{align*}
\]

- tainting of each memory cell

- value taints \{MustBeLow, MayBeHigh\}
Constant-time analysis at source level

We design an abstract functor

- takes as input an abstract memory domain
- returns an abstract domain that taints every memory cell

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>value</td>
</tr>
<tr>
<td>x</td>
<td>memory cell</td>
</tr>
<tr>
<td>e₁</td>
<td>value</td>
</tr>
<tr>
<td>e₂</td>
<td>value</td>
</tr>
<tr>
<td>m</td>
<td>memory</td>
</tr>
<tr>
<td>T</td>
<td>tainting</td>
</tr>
</tbody>
</table>

Example:

\[
T[*e_1 \rightarrow e_2](m, t) = t[l \mapsto T[e_2](l)]
\]

∀l ∈ concretize \circ [e_1](m)

Diagram:

- Abstract interpreter
- Control flow
- Taint domain
- States
- Memory & value domain
- Numbers

Set of concrete memory locations

Value taints {MustBeLow, MaybeHigh}

Tainting of each memory cell
Experiments at source level (ESORICS’17)

<table>
<thead>
<tr>
<th>Example</th>
<th>Size</th>
<th>Loc</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes</td>
<td>1171</td>
<td>1399</td>
<td>41.39</td>
</tr>
<tr>
<td>curve25519-donna</td>
<td>1210</td>
<td>608</td>
<td>586.20</td>
</tr>
<tr>
<td>des</td>
<td>229</td>
<td>436</td>
<td>2.28</td>
</tr>
<tr>
<td>rlwe_sample</td>
<td>145</td>
<td>1142</td>
<td>30.76</td>
</tr>
<tr>
<td>salsa20</td>
<td>341</td>
<td>652</td>
<td>0.04</td>
</tr>
<tr>
<td>sha3</td>
<td>531</td>
<td>251</td>
<td>57.62</td>
</tr>
<tr>
<td>snow</td>
<td>871</td>
<td>460</td>
<td>3.37</td>
</tr>
<tr>
<td>tea</td>
<td>121</td>
<td>109</td>
<td>3.47</td>
</tr>
<tr>
<td>nacl_chacha20</td>
<td>384</td>
<td>307</td>
<td>0.34</td>
</tr>
<tr>
<td>nacl_sha256</td>
<td>368</td>
<td>287</td>
<td>0.04</td>
</tr>
<tr>
<td>nacl_sha512</td>
<td>437</td>
<td>314</td>
<td>1.02</td>
</tr>
<tr>
<td>mbedtls_sha1</td>
<td>544</td>
<td>354</td>
<td>0.19</td>
</tr>
<tr>
<td>mbedtls_sha256</td>
<td>346</td>
<td>346</td>
<td>0.38</td>
</tr>
<tr>
<td>mbedtls_sha512</td>
<td>310</td>
<td>399</td>
<td>0.26</td>
</tr>
<tr>
<td>mee-cbc</td>
<td>1959</td>
<td>939</td>
<td>933.37</td>
</tr>
</tbody>
</table>
## Experiments at source level (ESORICS’17)

<table>
<thead>
<tr>
<th>Example</th>
<th>Size</th>
<th>Loc</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes</td>
<td>1171</td>
<td>1399</td>
<td>41.39</td>
</tr>
<tr>
<td>curve25519-donna</td>
<td>1210</td>
<td>608</td>
<td>586.20</td>
</tr>
<tr>
<td>des</td>
<td>229</td>
<td>436</td>
<td>2.28</td>
</tr>
<tr>
<td>rlwe_sample</td>
<td>145</td>
<td>1142</td>
<td>30.76</td>
</tr>
<tr>
<td>salsa20</td>
<td>341</td>
<td>652</td>
<td>0.04</td>
</tr>
<tr>
<td>sha3</td>
<td>531</td>
<td>251</td>
<td>57.62</td>
</tr>
<tr>
<td>snow</td>
<td>871</td>
<td>460</td>
<td>3.37</td>
</tr>
<tr>
<td>tea</td>
<td>121</td>
<td>109</td>
<td>3.47</td>
</tr>
<tr>
<td>nacl_chacha20</td>
<td>384</td>
<td>307</td>
<td>0.34</td>
</tr>
<tr>
<td>nacl_sha256</td>
<td>368</td>
<td>287</td>
<td>0.04</td>
</tr>
<tr>
<td>nacl_sha512</td>
<td>437</td>
<td>314</td>
<td>1.02</td>
</tr>
<tr>
<td>mbedtls_sha1</td>
<td>544</td>
<td>354</td>
<td>0.19</td>
</tr>
<tr>
<td>mbedtls_sha256</td>
<td>346</td>
<td>346</td>
<td>0.38</td>
</tr>
<tr>
<td>mbedtls_sha512</td>
<td>310</td>
<td>399</td>
<td>0.26</td>
</tr>
<tr>
<td>mee-cbc</td>
<td>1959</td>
<td>939</td>
<td>933.37</td>
</tr>
</tbody>
</table>

- Same benchmarks than Almeida et al.
- Not handled by Almeida et al. because LLVM alias analysis limitations

Preserving the property through compilation


- Makes precise what secure compilation means for cryptographic constant-time
- Provides a machine checked-proof that a mildly modified version of the CompCert compiler preserves cryptographic constant-time
- Explains how to turn a pre-existing formally-verified compiler into a formally-verified secure compiler
- Provides a proof toolkit for proving security preservation with simulation diagrams
CompCert: 1 compiler, 11 languages
CompCert preservation proof methodology

• Each language is given an operational semantics $s \xrightarrow{t} s'$ that models a small step transition from a state $s$ to a state $s'$ by emitting a trace of external events $t$.

• From this stems a notion of program behavior (event trace) for complete (possibly infinite) executions.

• Behavior preservation is proved via backward and forward simulation, but thanks to language determinism, forward simulation is enough.
CompCert preservation proof methodology

• Each language is given an operational semantics $s \xrightarrow{t} s'$ that models a small step transition from a state $s$ to a state $s'$ by emitting a trace of external events $t$.

• From this stems a notion of program behavior (event trace) for complete (possibly infinite) executions.

• Behavior preservation is proved via backward and forward simulation, but thanks to language determinism, forward simulation is enough.
CompCert preservation proof methodology

• Each language is given an operational semantics \( s \xrightarrow{t} s' \) that models a small step transition from a state \( s \) to a state \( s' \) by emitting a trace of external events \( t \).

• From this stems a notion of program behavior (event trace) for complete (possibly infinite) executions.

• Behavior preservation is proved via backward and forward simulation, but thanks to language determinism, forward simulation is enough.
CompCert preservation proof methodology

- Each language is given an **operational semantics** $s \xrightarrow{t} s'$ that models a small step transition from a state $s$ to a state $s'$ by emitting a trace of external events $t$.

- From this stems a notion of **program behavior** (event trace) for complete (possibly infinite) executions.

- Behavior preservation is proved via backward and forward simulation, but thanks to language determinism, **forward simulation** is enough.

\[
s_1 \xrightarrow{\sigma_1} t \approx s_2 \xrightarrow{\sigma_2}
\]

or

\[
s_1 \xrightarrow{t} s_2
\]

with $t = \varepsilon$

and $m(s_2) < m(s_1)$

well founded measure
CompCert: 17 preservations proofs

<table>
<thead>
<tr>
<th>Compiler pass</th>
<th>Explanation on the pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cshmggen</td>
<td>Type elaboration, simplification of control</td>
</tr>
<tr>
<td>Cminorgen</td>
<td>Stack allocation</td>
</tr>
<tr>
<td>Selection</td>
<td>Recognition of operators and addr. modes</td>
</tr>
<tr>
<td>RTLgen</td>
<td>Generation of CFG and 3-address code</td>
</tr>
<tr>
<td>Tailcall</td>
<td>Tailcall recognition</td>
</tr>
<tr>
<td>Inlining</td>
<td>Function inlining</td>
</tr>
<tr>
<td>Renumber</td>
<td>Renumbering CFG nodes</td>
</tr>
<tr>
<td>ConstProp</td>
<td>Constant propagation</td>
</tr>
<tr>
<td>CSE</td>
<td>Common subexpression elimination</td>
</tr>
<tr>
<td>Deadcode</td>
<td>Redundancy elimination</td>
</tr>
<tr>
<td>Allocation</td>
<td>Register allocation</td>
</tr>
<tr>
<td>Tunneling</td>
<td>Branch tunneling</td>
</tr>
<tr>
<td>Linearize</td>
<td>Linearization of CFG</td>
</tr>
<tr>
<td>CleanupLabels</td>
<td>Removal of unreferenced labels</td>
</tr>
<tr>
<td>Debugvar</td>
<td>Synthesis of debugging information</td>
</tr>
<tr>
<td>Stacking</td>
<td>Laying out stack frames</td>
</tr>
<tr>
<td>Asmgen</td>
<td>Emission of assembly code</td>
</tr>
</tbody>
</table>
Cryptographic constant-time property: defining leakages

- We enrich the CompCert traces of events with leakages of two types
  - either the truth value of a condition,
  - or a pointer representing the address of
    - either a memory access (i.e., a load or a store)
    - or a called function
- Using event erasure, from $s \xrightarrow{t} s'$ we can extract
  - the compile-only judgment $s \xrightarrow{t \text{comp}} s'$
  - the leak-only judgment $s \xrightarrow{t \text{leak}} s'$
- Program leakage is defined as the behavior of the $\xrightarrow{\text{leak}}$ semantics
Cryptographic constant-time property: preservation

• We note \( \varphi(s, s') \) the fact that two initial states \( s \) and \( s' \) share the same values for public inputs, but may differ on the values of secret inputs.

• A program is **constant-time secure w.r.t.** \( \varphi \) if for two initial states \( s \) and \( s' \) such that \( \varphi(s, s') \) holds, then both leak-only executions starting from \( s \) and \( s' \) observe the same leakage.

\[ \varphi(s, s') \]

\[ \text{implies} \quad t = t' \]
Cryptographic constant-time property: preservation

• We note $\varphi(s, s')$ the fact that two initial states $s$ and $s'$ share the same values for public inputs, but may differ on the values of secret inputs.

• A program is **constant-time secure w.r.t. $\varphi$** if for two initial states $s$ and $s'$ such that $\varphi(s, s')$ holds, then both leak-only executions starting from $s$ and $s'$ observe the same leakage.

**Main Theorem (Constant-Time security preservation):** Let $P$ be a safe Clight source program that is compiled into an x86 assembly program $P'$. If $P$ is constant-time w.r.t. $\varphi$, then so is $P'$. 
Conclusion
Conclusion

- We can build secure programming abstractions at source level (C-like)

This talk focused on Crypto-Constant-Time security property
Conclusion

- We can build secure programming abstractions at source level (C-like).
- We make sure the compiler will generate executables that are as secure.

This talk focused on Crypto-Constant-Time security property.
Conclusion

• We can build secure programming abstractions at source level (C-like)

• We make sure the compiler will generate executables that are as secure

• We reduce as much as possible the TCB (Trusted Computing Base) with formal proofs