Verification of constant-time implementation in the Compcert 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
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 -> Clight -> C#minor

RTL -> CminorSel -> Cminor

LTL -> LTLin -> Linear

ASM -> Mach

side-effects out of expressions

type elimination loop simplifications

stack allocation of «&»variables

instruction selection

spilling, reloading calling conventions

layout of stack frames

asm code generation

register allocation (IRC)

linearization of the CFG
CompCert: 1 compiler, 11 languages

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

where should we perform the constant time analysis?
This talk: 3 approaches

1. Analysis at (almost) assembly level

   G. Barthe, G. Betarte, J. D. Campo, C. Luna and D. Pichardie.  
   *System-level non-interference for constant-time cryptography.*  
   CCS 2014.

2. Analysis at (almost) assembly level, with help from an analysis at source level

   *Lightweight, Verified Translation Validation of Static Analyses.*  
   CSF 2017.

3. Analysis at source level

   Sandrine Blazy, David Pichardie, Alix Trieu.  
   *Verifying Constant-Time Implementations by Abstract Interpretation.*  
   ESORICS 2017.
First approach
Performing the analysis at (pre)-assembly level

Good place for proving constant-time on actual implementation

- Compcert Mach level is the last IR before full assembly
- Compcert does no introduce new memory operations after that level

But the place is challenging for static analysis tool

- no more memory abstraction: memory is one single big array
- all memory accesses handle some kind of arithmetic on addresses
First approach
Performing the analysis at (pre)-assembly level

Strong points
• verified static alias analysis at Mach level
• verified taint analysis using the alias information
• several experiments on real crypto C programs: Salsa20, Sha256, TEA

Weak points
• several manual rewriting of the source programs are required
• efficiency is bad because of function full inlining

Performing the analysis at (pre)-assembly level

Technical details

Low level memory model
  • registers + one memory block for each global variable + one memory block for the whole stack

Pre-analysis
  • we perform a points-to analysis to tracks the set of memory blocks manipulated by each memory instruction

Taint analysis
  • one taint for each global variable
  • one taint for each register, at each program point
  • one taint for each stack slot (byte), at each program point
Performing the analysis at (pre)-assembly level
Constraint based specification (excerpt)

\[ X_h \vdash n : \tau_1 \Rightarrow \tau_2 \]
Performing the analysis at (pre)-assembly level
Constraint based specification (excerpt)

\[ X_h \vdash n : \tau_1 \Rightarrow \tau_2 \]

- taint of each global variable
- program point
- local types (registers + stack slot)
Performing the analysis at (pre)-assembly level
Constraint based specification (excerpt)

\[
X_h \vdash n : \tau_1 \Rightarrow \tau_2
\]

- Taint of each global variable
- Program point
- Local types (registers + stack slot)

\[
p(n) = \text{store}_s(\text{addr}, \vec{r}, r, n')
\]

\[
\text{PointsTo}(n, \text{addr}, \vec{r}) = \text{Symb}(S)
\]

\[
\tau(\vec{r}) = \text{Low} \quad \tau(r) \subseteq X_h(S)
\]

\[
X_h \vdash n : \tau \Rightarrow \tau
\]

\[
p(n) = \text{store}_s(\text{addr}, \vec{r}, r, n')
\]

\[
\text{PointsTo}(n, \text{addr}, \vec{r}) = \text{Stack}(\delta)
\]

\[
X_h \vdash n : \tau \Rightarrow \tau[\delta \mapsto \tau(r), \ldots, \delta + \varsigma - 1 \mapsto \tau(r)]
\]
Performing the analysis at (pre)-assembly level
Constraint based specification (excerpt)

\[
X_h \vdash n : \tau_1 \Rightarrow \tau_2
\]

- taint of each global variable
- program point
- local types (registers + stack slot)
- instruction at program point \( p \)
- symbolic memory address
- stored value
- size of the accessed memory block
- next program point

\[
\begin{align*}
p(n) &= \text{store}_s(\text{addr}, \vec{r}, r, n') \\
\text{PointsTo}(n, \text{addr}, \vec{r}) &= \text{Symb}(S) \\
\tau(\vec{r}) &= \text{Low} \\
\tau(r) &\subseteq X_h(S) \\
X_h \vdash n : \tau \Rightarrow \tau
\end{align*}
\]

\[
\begin{align*}
p(n) &= \text{store}_s(\text{addr}, \vec{r}, r, n') \\
\text{PointsTo}(n, \text{addr}, \vec{r}) &= \text{Stack}(\delta) \\
X_h \vdash n : \tau \Rightarrow \tau[\delta \mapsto \tau(r), \ldots, \delta + \varsigma - 1 \mapsto \tau(r)]
\end{align*}
\]
Performing the analysis at (pre)-assembly level
Constraint based specification (excerpt)

\[ X_h \vdash n : \tau_1 \Rightarrow \tau_2 \]

- taint of each global variable
- program point
- local types (registers + stack slot)
- instruction at program point \( p \)
- symbolic memory address
- stored value
- points-to pre-analysis
- size of the accessed memory block
- next program point

\[
\begin{align*}
\text{PointsTo}(n, \text{addr}, \vec{r}) &= \text{Symb}(S) \\
\tau(\vec{r}) &= \text{Low} \\
\tau(r) &\subseteq X_h(S) \\
\end{align*}
\]

\[
\begin{align*}
\text{PointsTo}(n, \text{addr}, \vec{r}) &= \text{Stack}(\delta) \\
p(n) &= \text{store}_s(\text{addr}, \vec{r}, r, n') \\
\end{align*}
\]

\[
X_h \vdash n : \tau \Rightarrow \tau
\]

\[
X_h \vdash n : \tau \Rightarrow \tau[\delta \mapsto \tau(r), \ldots, \delta + \varsigma - 1 \mapsto \tau(r)]
\]
Performing the analysis at (pre)-assembly level
Constraint based specification (excerpt)

\[ X_h \vdash n : \tau_1 \Rightarrow \tau_2 \]

- taint of each global variable
- program point
- local types (registers + stack slot)
- instruction at program point \( p \)
- symbolic memory address
- stored value
- points-to pre-analysis
- size of the accessed memory block
- next program point

\[
p(n) = \text{store}_s(addr, \vec{r}, r, n') \\
\text{PointsTo}(n, addr, \vec{r}) = \text{Symb}(S) \\
\tau(\vec{r}) = \text{Low} \quad \tau(r) \subseteq X_h(S) \\
X_h \vdash n : \tau \Rightarrow \tau
\]

\[
p(n) = \text{store}_s(addr, \vec{r}, r, n') \\
\text{PointsTo}(n, addr, \vec{r}) = \text{Stack}(\delta) \\
X_h \vdash n : \tau \Rightarrow \tau[\delta \mapsto \tau(r), \ldots, \delta + \varsigma - 1 \mapsto \tau(r)]
\]

we forbid high taints on address computation
Performing the analysis at (pre)-assembly level
Constraint based specification (excerpt)

\[ X_h \vdash n : \tau_1 \Rightarrow \tau_2 \]

- taint of each global variable
- program point
- local types (registers + stack slot)
- instruction at program point \( p \)
- symbolic memory address
- stored value
- points-to pre-analysis
- size of the accessed memory block
- next program point

\[ p(n) = \text{store}_s(addr, \bar{r}, r, n') \]
\[ \text{PointsTo}(n, addr, \bar{r}) = \text{Symb}(S) \]
\[ \tau(\bar{r}) = \text{Low} \]
\[ \tau(r) \subseteq X_h(S) \]

We forbid high taints on address computation
If register \( r \) is high, global variable \( S \) must be high

\[ X_h \vdash n : \tau \Rightarrow \tau \]
\[ X_h \vdash n : \tau \Rightarrow \tau[\delta \mapsto \tau(r), \ldots, \delta + \varsigma - 1 \mapsto \tau(r)] \]
Performing the analysis at (pre)-assembly level
Constraint based specification (excerpt)

\[
X_h \vdash n : \tau_1 \Rightarrow \tau_2
\]

taint of each global variable
program point
local types (registers + stack slot)

instruction at program point \( p \)
symbolic memory address
stored value
points-to pre-analysis
size of the accessed memory block
next program point

\[
p(n) = \text{store}_s(\text{addr}, \vec{r}, r, n')
\]

\[
\text{PointsTo}(n, \text{addr}, \vec{r}) = \text{Symb}(S)
\]

\[
\tau(\vec{r}) = \text{Low}
\]

\[
\tau(r) \subseteq X_h(S)
\]

\[
X_h \vdash n : \tau \Rightarrow \tau
\]

we forbid high taints on address computation
if register \( r \) is high, global variable \( S \) must be high
we taint each stack position

\[
p(n) = \text{store}_s(\text{addr}, \vec{r}, r, n')
\]

\[
\text{PointsTo}(n, \text{addr}, \vec{r}) = \text{Stack}(\delta)
\]

\[
X_h \vdash n : \tau \Rightarrow \tau[\delta \mapsto \tau(r), \ldots, \delta + \varsigma - 1 \mapsto \tau(r)]
\]
Performing the analysis at (pre)-assembly level

Limitations

Engineering simplification
- no function call (we require full inlining)
- no dynamic allocation

Analysis precision limitation
- no array in stack (we only track constant addresses in stack)
- no fine grained struct tainting for structures in global variables

Manual rewriting
- every local arrays must be put as global!

But the analyser
- is proved correct and extracted from Coq formalisation
- runs on three real C programs

<table>
<thead>
<tr>
<th>Program</th>
<th>LoC</th>
<th>Analysis time</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEA</td>
<td>70</td>
<td>0.08s</td>
</tr>
<tr>
<td>SHA256</td>
<td>419</td>
<td>68.14s</td>
</tr>
<tr>
<td>Salsa20</td>
<td>1077</td>
<td>0.68s</td>
</tr>
</tbody>
</table>
Second approach
Some help from higher level representations…

*Lightweight, Verified Translation Validation of Static Analyses.*
CSF 2017.

Improvements

- no more manual rewriting
- better performance

How?

- The Verasco static analyser transmits strong alias informations through the compiler toolchain

Extensibility

- soundness of the translation is independent of compiler optimisation passes
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 »

http://verasco.imag.fr
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

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

numbers

CompCert compiler
Verasco
Abstract numerical domains

- CompCert C → Clight → C#minor → ... → CompCert compiler
- Abstract interpreter
- Memory & value domain
  - Z → int
    - Convex polyhedra
      - VERIMAG work
    - Symbolic equalities
    - Nonrel → Rel
      - Integer congruences
    - Nonrel → Rel
      - 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

VERIMAG work

Transforms any rel. domain over Z into a rel. domain over machine integers with modulo arithmetic.
Verasco
Abstract numerical domains

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

Alarms ← Abstract interpreter → control flow

Memory & value domain

Nonrel → Rel

Nonrel → Rel

Z → int

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

[SAS’13]

VERIMAG work

Convex polyhedra

Symbolic equalities

Nonrel→ Rel

Integer congruences

Nonrel→ Rel

Integer & F.P. intervals
Verasco
Abstract numerical domains

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

Alarms ← Abstract interpreter → control flow

Memory & value domain

symbolic conditional expressions (improve precision of assume commands)

Z → int

Convex polyhedra
Symbolic equalities

Nonrel → Rel

Integer congruences

Nonrel → Rel

Integer & F.P. intervals

VERIMAG work
Verasco
Abstract numerical domains

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

Abstract interpreter
Alarms
Control flow

Memory & value domain
States

Z → int

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

Verimag work

Transforms any non-rel. domain into a (reduced) rel. domain
Verasco
Abstract numerical domains

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

Alarms ← Abstract interpreter / control flow

Memory & value domain / states

Z → int

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

Convex polyhedra
Symbolic equalities
VERIMAG work

Nonrel → Rel
Nonrel → Rel

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 \rightarrow \text{int}

Convex polyhedra ← Symbolic equalities ← Nonrel \rightarrow \text{Rel} ← Nonrel \rightarrow \text{Rel}

requires reasoning on double-precision floating-point numbers (IEEE754)

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 → numbers

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

Nonrel → Rel

VERIMAG work

custom reduced product
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

VERIMAG work

• 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.
How to translate Verasco results down to assembly?

The Coq proof assistant
Translation validation of Verasco results
Translation validation of Verasco results
Translation validation of Verasco results
Translation validation of Verasco results

CompCert

C source

C#minor

Lowering
Optimization

Annotated RTL

Annotated Asm

Verasco

High-Level Annotations

Lowering
Optimization

Low-Level Annotations

Defensive C#minor
Translation validation of Verasco results

Diagram:
- CompCert
  - C source
  - C#minor
    - Lowering
    - Optimization
    - Annotated RTL
      - Annotation correctness proof
      - Annotated Asm
      - ?
  - Verasco
    - High-Level Annotations
    - Low-Level Annotations
      - Annotation correctness proof
      - Defensive RTL
  - Defensive C#minor
    - Lowering
    - Optimization
Translation validation of Verasco results
Translation validation of Verasco results

- CompCert
  - C source
    - C#minor
      - Lowering
      - Optimization
        - Annotated RTL
          - Low-Level Annotations
            - Defensive RTL
              - Defensive C#minor

- Verasco
  - High-Level Annotations
    - Low-Level Annotations
      - Defensive RTL

- RTL

- Safety proof
Translation validation of Verasco results

CompCert

C source

C#minor

Lowering
Optimization

Annotated RTL

Annotated Asm

Verasco

High-Level Annotations

Low-Level Annotations

Defensive RTL

Defensive C#minor

Verasco

Safety proof

RTL

Relative-safety checker

Equivalence proof

Annotation correctness proof

Annotation correctness proof
Third approach
Constant-time analysis at source level

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

Improvements
- Inform the programmer at source level
- Deeper interaction with Verasco

How?
- We mix Verasco memory abstract domain with fine-grained tainting
Constant-time analysis at source level
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

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

We design an abstract functor

- takes as input an abstract memory domain

\[ [e]^\# : M^\# \to V^\# \]
\[ [x \to e]^\# : M^\# \to M^\# \]
\[ [*e_1 \to e_2]^\# : M^\# \to M^\# \]
\[ [x \to *e]^\# : M^\# \to M^\# \]
\[ \text{assert}(e)^\# : M^\# \to M^\# \]
\[ \text{concretize}^\# : V^\# \to \mathcal{P}(L) \]
Constant-time analysis at source level

We design an abstract functor

- takes as input an abstract memory domain
  - abstract memory
  - abstract value

- returns an abstract domain that taints every memory cells

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

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

Set of concrete memory locations

Value taints \{MustBeLow, MayBeHigh\}

Tainting of each memory cell
Constant-time analysis at source level

We design an abstract functor

• takes as input an abstract memory domain

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

• returns an abstract domain that taints every memory cell

  \[
  \begin{align*}
  \mathcal{T}[\llbracket e \rrbracket^\#] : & \quad M_{\text{taint}}^\# \rightarrow V_{\text{taint}}^\# \\
  \mathcal{T}[\llbracket x \rightarrow e \rrbracket^\#] : & \quad M^\# \rightarrow M_{\text{taint}}^\# ightarrow M_{\text{taint}}^\# \\
  \mathcal{T}[\llbracket \ast e_1 \rightarrow e_2 \rrbracket^\#] : & \quad M^\# \rightarrow M_{\text{taint}}^\# ightarrow M_{\text{taint}}^\# \\
  \mathcal{T}[\llbracket x \rightarrow \ast e \rrbracket^\#] : & \quad M^\# \rightarrow M_{\text{taint}}^\# ightarrow M_{\text{taint}}^\#
  \end{align*}
  \]

Example:

\[
\mathcal{T}[\llbracket \ast e_1 \rightarrow e_2 \rrbracket^\#](m^\#, t^\#) = t^\#[l \mapsto \mathcal{T}[\llbracket e_2 \rrbracket^\#]]_{t^\#}
\]

\[
\forall l \in \text{concretize}^\# \circ \llbracket e_1 \rrbracket^\#(m^\#)
\]
### Table 1: Verification of cryptographic primitives

<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>

The first block of lines gathers test cases for the implementations of a representative set of cryptographic primitives including TEA [36], an implementation of sampling in a discrete Gaussian distribution by Bos et al. [10] (rlwe_sample) from the Open Quantum Safety library [30], a NaCl implementation of elliptic curve arithmetic operations over Curve25519 [6], and various primitives such as AES, DES, etc. The second block reports on different implementations from the NaCl library [7]. The third block reports on implementations from the mbedTLS [26] library. Finally, the last result corresponds to an implementation of MAC-then-Encode-then-CBC-Encrypt (MEE-CBC). All these examples are proven constant time, except for AES and DES. Our prototype rightfully reports memory accesses depending on secrets, so these two programs are not constant time. Similarly to [2], rlwe_sample is only proven constant time, provided that the core random generator is also constant time, thus showing that it is the only possible source of leakage. The last example mee-cbc is a full implementation of the MEE-CBC construction using low-level primitives taken from the NaCl library. Our prototype is able to verify the constant-time property of this example, showing that it scales to large code bases (939 loc).
## Experiments at source level (ESORICS’17)

We report in Table 1 our results on a set of cryptographic algorithms, all executions times reported were obtained on a 3.1GHz Intel i7 with 16GB of RAM. Sizes are reported in terms of numbers of C\# major statements (i.e., close to C statements), lines of code are measured with cloc and execution times are reported in seconds.

<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>nbedtls_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>

The first block of lines gathers test cases for the implementations of a representative set of cryptographic primitives including TEA [36], an implementation of sampling in a discrete Gaussian distribution by Bos et al. [10] (rlwe_sample) taken from the Open Quantum Safety library [30], an implementation of elliptic curve arithmetic operations over Curve25519 [6] by Langley [16] (curve25519-donna), and various primitives such as AES, DES, etc. The second block reports on different implementations from the NaCl library [7]. The third block reports on implementations from the mbedTLS [26] library. Finally, the last result corresponds to an implementation of MAC-then-Encode-then-CBC-Encrypt (MEE-CBC).

All these examples are proven constant time, except for AES and DES. Our prototype rightfully reports memory accesses depending on secrets, so these two programs are not constant time. Similarly to [2], rlwe_sample is only proven constant time, provided that the core random generator is also constant time, thus showing that it is the only possible source of leakage.

The last example mee-cbc is a full implementation of the MEE-CBC construction using low-level primitives taken from the NaCl library. Our prototype is able to verify the constant-time property of this example, showing that it scales to large code bases (939 loc).

Same benchmarks than Almeida et al.


Not handled by Almeida et al. because LLVM alias analysis limitations
## Comparing the three approaches

<table>
<thead>
<tr>
<th><strong>Approach</strong></th>
<th><strong>Pro</strong></th>
<th><strong>Cons</strong></th>
<th><strong>Current state of proof mechanization</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct analysis at pre-assembly level</td>
<td>property established at the expected level</td>
<td>engineering a static analysis at assembly level is hard</td>
<td>fully verified in Coq</td>
</tr>
<tr>
<td>Translation of Verasco results</td>
<td>the translation mechanism may be useful outside security analysis</td>
<td>the validation technique may be incomplete with respect to state-of-the-art compiler optimizations</td>
<td>only the annotation validation is currently verified</td>
</tr>
<tr>
<td>Analysis at source level</td>
<td>1) reuse the Verasco effort feedback for crypto programmers</td>
<td>we need to trust (or prove) that the compiler will not break the security property</td>
<td>only a paper proof</td>
</tr>
</tbody>
</table>
Conclusions

Constant-time
• simpler than full non-interference but still challenging security property
• hard to obtain at assembly level without control on the compiler
• further work: cover more side-channels (e.g. floating point computations)

Verified C compiler toolchain for security
• strong soundness guaranties
• allow experimentation with real crypto programs
• further work: enforce other folklore protections