Formal Verification of a Constant-Time Preserving C Compiler

Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie and Alix Trieu
Cache timing attacks against cryptographic implementations

- 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 programming
A programming discipline for crypto programmers

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

• This is a strictly stronger property than « time execution does not depend on secrets »!

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

• Several verification tools have been built and used for checking that popular libraries are constant-time [Almeida16, Rodrigues16]

• But checking low-level implementations is not ideal
  • it makes the analysis work harder (e.g. alias analysis)
  • it makes the results of the analysis difficult to understand for programmers
Our Research Program
Our Research Program

• Build secure programming abstractions at source level (C-like)
Our Research Program

• Build secure programming abstractions at source level (C-like)

• Make sure the compiler will generate executables that are as secure
Our Research Program

• Build secure programming abstractions at source level (C-like)

• Make sure the compiler will generate executables that are as secure

• Reduce as much as possible the TCB (Trusted Computing Base) with formal proofs
Our Research Program

- Build secure programming abstractions at source level (C-like)
- Make sure the compiler will generate executables that are as secure
- Reduce as much as possible the TCB (Trusted Computing Base) with formal proofs
Our Research Program

• Build secure programming abstractions at source level (C-like)

• Make sure the compiler will generate executables that are as secure

• Reduce as much as possible the TCB (Trusted Computing Base) with formal proofs
Our Research Program

• Build secure programming abstractions at source level (C-like)

• Make sure the compiler will generate executables that are as secure

• Reduce as much as possible the TCB (Trusted Computing Base) with formal proofs
Our Research Program

• Build secure programming abstractions at source level (C-like)

• Make sure the compiler will generate executables that are as secure

• Reduce as much as possible the TCB (Trusted Computing Base) with formal proofs
• Build secure programming abstractions at source level (C-like)

• Make sure the compiler will generate executables that are as secure

• Reduce as much as possible the TCB (Trusted Computing Base) with formal proofs
Cryptographic constant-time verification

- In [ESORICS’17] we provide a verification tool at C source level
  - it tracks taints in memory and checks the constant-time property
  - it is based on the Verasco C abstract interpreter [POPL’15]

- In this work [POPL’20]
  - we prove the CompCert compiler preserves the constant-time property

∀p, ConstantTime(p) ⇒ ConstantTime(compile(p))
Verified Compilation

• Proving semantic properties on non-toy compilers requires a machine-checked proof

• CompCert [Leroy06] is a milestone in this area
  • a moderately optimizing compiler for C
  • programmed and verified with the Coq proof assistant
  • now being used in commercial settings and for software certification [Kästner18]

• CompCert theorems show
  • it preserves memory safety ✔ ✔
  • it preserves observable behaviors ✔ ✔
  • but they says nothing about side channels attacks ❓
This work

• 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
Some background on CompCert
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».

Compiler written from scratch, along with its proof; not trying to prove an existing compiler
Compcert meets the industrial world

Fly-by-wire software, for recent Airbus planes
- control-command code generated from block diagrams (3600 files, 3.96 MB of assembly code)
- minimalistic OS

Results
- Estimated WCET for each file
- Average improvement per file: 14%
- Compiled with CompCert 2.3, May 2014

Conformance to the certification process (DO-178)
- Trade-off between traceability guarantees and efficiency of the generated code
Compcert meets the industrial world

Fly-by-wire software, for recent Airbus planes

• control-command code generated from block diagrams (3600 files, 3.96 MB of assembly code)
• minimalistic OS

Results

• Estimated WCET
• Average improvement per file: 14%
• Compiled with CompCert 2.3, May 2014

Conformance to the certification process (DO-178)

• Trade-off between traceability guarantees and efficiency of the generated code

https://www.absint.com/compcert/
CompCert: 1 compiler, 11 languages…

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

- CompCertC
  - side-effect out of expressions
- Clight
  - type elimination
  - loop simplifications
- C#minor
  - stack allocation of «&»variables
  - instruction selection
  - (instruction scheduling)
- RTL
  - register allocation (IRC)
  - CFG construction
  - expr. decomp.
  - linearization of the CFG
- CminorSel
  - spilling, reloading
  - calling conventions
- Cminor
  - layout of stack frames
  - asm code generation
- LTL
  - asm code generation
- LTLin
  - asm code generation
- Linear
  - asm code generation
- Mach
CompCert: 1 compiler, 11 languages…

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

- RTL
  - CFG construction expr. decomp.
  - register allocation (IRC)
- LTL
  - linearization of the CFG
- CMINOR
  - CminorSel
  - instruction selection
  - spilling, reloading calling conventions
- CMinor
  - Linear
  - layout of stack frames
- ASm
  - asm code generation

We limit our work on deterministic semantics

side-effect out of expressions

type elimination loop simplifications

CompCert verification tools [Jourdan15,Blazy19] work here anyway
CompCert: ... and 17 preservations proofs

<table>
<thead>
<tr>
<th>Compiler pass</th>
<th>Explanation on the pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cshmgen</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>
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 langage 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 langage determinism, forward simulation is enough.
Verified Static Analysis meets CompCert
The verified C static analyzer Verasco [POPL’15]

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 1/10th of GCC but formally verified »,
- likewise « Verasco \approx 1/10th of Astrée but formally verified »
Defining Cryptographic Constant-Time Preservation
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 $s \xrightarrow{\text{\textup{comp}}} s'$
    • the leak-only judgment $s \xrightarrow{\text{\textup{leak}}} s'$
  • Program leakage is defined as the behavior of the $\xrightarrow{\text{\textup{leak}}}$ 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

$\varphi|_{s\to t} = \varphi|_{s'\to t'}$ implies $t = t'$
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 $P'$. 
Proving Cryptographic Constant-Time Preservation
Proving cryptographic constant-time preservation
A proof engineering perspective

• Cryptographic constant-time preservation is a property about the leak-only semantics $\rightarrow_{\text{leak}}$
• But existing CompCert simulation diagrams deal with the compile-only semantics $\rightarrow_{\text{comp}}$
• Our proof engineering strategy is to benefit as much as possible from the proof scripts of these diagrams
Proving cryptographic constant-time preservation
A proof engineering perspective

- Cryptographic constant-time preservation is a property about the leak-only semantics $\rightarrow_{\text{leak}}$
- But existing CompCert simulation diagrams deal with the compile-only semantics $\rightarrow_{\text{comp}}$
- Our proof engineering strategy is to benefit as much as possible from the proof scripts of these diagrams

Standard CompCert forward simulation theorem about $\rightarrow_{\text{comp}}$

Standard CompCert forward simulation proof script
Proving cryptographic constant-time preservation
A proof engineering perspective

• Cryptographic constant-time preservation is a property about the leak-only semantics $\rightarrow_{\text{leak}}$
• But existing CompCert simulation diagrams deal with the compile-only semantics $\rightarrow_{\text{comp}}$
• Our proof engineering strategy is to benefit as much as possible from the proof scripts of these diagrams

- Slightly modified CompCert forward simulation theorem about $\rightarrow$
- generic theorem $\implies$
- Standard CompCert forward simulation theorem about $\rightarrow_{\text{comp}}$
Proving cryptographic constant-time preservation
A proof engineering perspective

- Cryptographic constant-time preservation is a property about the leak-only semantics →leak
- But existing CompCert simulation diagrams deal with the compile-only semantics →comp
- Our proof engineering strategy is to benefit as much as possible from the proof scripts of these diagrams
Proving cryptographic constant-time preservation
A proof engineering perspective

• Cryptographic constant-time preservation is a property about the leak-only semantics $\rightarrow_{\text{leak}}$
• But existing CompCert simulation diagrams deal with the compile-only semantics $\rightarrow_{\text{comp}}$
• Our proof engineering strategy is to benefit as much as possible from the proof scripts of these diagrams

Slightly modified CompCert forward simulation theorem about $\rightarrow_{\text{comp}}$
Slightly modified CompCert forward simulation proof script

implies
implies
implies
implies

Standard CompCert forward simulation theorem about $\rightarrow_{\text{comp}}$
Constant-time preservation theorem about $\rightarrow_{\text{leak}}$

generic theorem
Four proof techniques

• Each technique provides a specific tradeoff between generality and proof tractability

• The first three are slight relaxations of the classical forward diagram and reuse existing scripts

Trace preservation
Leak erasing
Trace transformation
CT cube diagram
Four proof techniques

- Each technique provides a specific tradeoff between generality and proof tractability
- The first three are slight relaxations of the classical forward diagram and reuse existing scripts
A palette of proof methods

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

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

Trace preservation  
Leak erasing  
Trace transformation  
CT cube diagram
# A palette of proof methods

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

- **Trace preservation:** 6/17
- **Leak erasing:** 5/17
- **Trace transformation:**
- **CT cube diagram:**
## A palette of proof methods

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

**Categories:**
- Trace preservation
- Leak erasing
- Trace transformation
- CT cube diagram
A palette of proof methods

<table>
<thead>
<tr>
<th>Compiler pass</th>
<th>Diagram used</th>
<th>Explanation on the pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cshmgen</td>
<td>Trace preservation</td>
<td>Type elaboration, simplification of control</td>
</tr>
<tr>
<td>Cminorgen</td>
<td>Trace transformation</td>
<td>Stack allocation</td>
</tr>
<tr>
<td>Selection</td>
<td>Leak erasing</td>
<td>Recognition of operators and addr. modes</td>
</tr>
<tr>
<td>RTLgen</td>
<td>Trace preservation</td>
<td>Generation of CFG and 3-address code</td>
</tr>
<tr>
<td>Tailcall</td>
<td>Trace preservation</td>
<td>Tailcall recognition</td>
</tr>
<tr>
<td>Inlining</td>
<td>Trace transformation</td>
<td>Function inlining</td>
</tr>
<tr>
<td>Renumber</td>
<td>Trace preservation</td>
<td>Renumbering CFG nodes</td>
</tr>
<tr>
<td>ConstProp</td>
<td>Trace transformation</td>
<td>Constant propagation</td>
</tr>
<tr>
<td>CSE</td>
<td>Leak erasing</td>
<td>Common subexpression elimination</td>
</tr>
<tr>
<td>Deadcode</td>
<td>Leak erasing</td>
<td>Redundancy elimination</td>
</tr>
<tr>
<td>Allocation</td>
<td>Leak erasing</td>
<td>Register allocation</td>
</tr>
<tr>
<td>Tunneling</td>
<td>Leak erasing</td>
<td>Branch tunneling</td>
</tr>
<tr>
<td>Linearize</td>
<td>CT cube diagram</td>
<td>Linearization of CFG</td>
</tr>
<tr>
<td>CleanupLabels</td>
<td>Trace preservation</td>
<td>Removal of unreferenced labels</td>
</tr>
<tr>
<td>Debugvar</td>
<td>Trace preservation</td>
<td>Synthesis of debugging information</td>
</tr>
<tr>
<td>Stacking</td>
<td>Trace transformation</td>
<td>Laying out stack frames</td>
</tr>
<tr>
<td>Asmgen</td>
<td>Trace transformation</td>
<td>Emission of assembly code</td>
</tr>
</tbody>
</table>
Conclusion and perspectives

Conclusion

• A machine checked-proof that a mildly modified version of the CompCert compiler preserves cryptographic constant-time
• A carefully crafted methodology that maximises proof reuse

Perspectives

• Make CompCert generate more efficient code for crypto programs (e.g. using SIMD instructions)
• Explore other observational information-flow policies and adapt CompCert
Execution times of 23 benchmark programs compiled with gcc -00, CompCert, gcc -01, and gcc -02

https://www.absint.com/compcert/
6.1 Experimental Evaluation

We carry out our experimental evaluation on selected examples from the literature. We note that our experimental evaluation is primarily used to validate that our approach is reasonable. However, a systematic and extensive evaluation of the impact of our compiler, or more generally of the CompCert compiler, on cryptographic libraries (including widely deployed libraries such as OpenSSL 2019 or repositories such as SUPERCOP 2019) is left for future work.

We first compare our version of CompCert to the original CompCert (version 3.4), and to gcc, with and without optimizations. We test these compilers on a benchmark of common cryptographic programs that were shown to be constant-time in Almeida et al. 2016; Blazy et al. 2019. They include cryptographic primitives such as an implementation of elliptic curve arithmetic operations over Curve25519 Bernstein 2006; Langley 2015, and TEA Wheeler and Needham 1994, together with implementations from commonly used cryptographic libraries such as NaCl Bernstein et al. 2012 and mbedTLS ARM 2016. These are C implementations that we experiment with in order to evaluate our compiler, but it should be reminded that if performance is an issue, it is generally better to use hand-optimized assembly code at the cost of portability.

We first measured the execution times (using an Intel i7-8550U CPU 1.8GHz, with 16GB of RAM), which are shown in Figure 10. We compiled these programs using the original CompCert, our modified CompCert, and gcc from -O0 to -O3. We normalized the measured execution times with the execution times of gcc -O0. The error bars represent the 99% confidence intervals of our measurements.