Formal Verification of a Constant-Time Preserving C Compiler

Sandrine Blazy

joint work with Gilles Barthe, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie and Alix Trieu
The CompCert formally verified compiler

Compiler + proof that the compiler does not introduce bugs

CompCert, a moderately optimising C compiler usable for critical embedded software
• Fly-by-wire software, Airbus A380, FCGU

We prove 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, and S has a safe behaviour, then «C behaves like S».

Behaviour = termination / divergence / undefined («going wrong») + (finite or infinite) trace of I/O operations performed
CompCert: 1 compiler, 10 languages and 17 semantic-preservation proofs

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

- RTL
  - register allocation (IRC)
  - CFG construction expr. decomp.

- LTL
  - linearisation of the CFG

- CminorSel
  - instruction selection

- Cminor
  - type elimination
  - stack allocation of «&»variables
  - (instruction scheduling)

- Clight
  - spilling, reloading calling conventions

- LTLin
  - asm code generation

- Linear
  - layout of stack frames

- ASM
  - Mach
CompCert: 1 compiler, 10 languages and 17 semantic-preservation proofs

Operational semantics

\[ S \xrightarrow{t} S' \quad S \xrightarrow{t^n} S' \quad S \xrightarrow{t^+} S' \quad S \xrightarrow{t} \infty \]
Proof methodology: forward simulation

Ingredients

- simulation relation \( \approx \) between source and target states
- measure \( m \) from source states to a well-founded set

\[
\begin{align*}
s_1 \approx s_2 & \quad \sigma_1 \approx \epsilon \\
\end{align*}
\]

or

\[
\begin{align*}
s_1 \approx s_2 & \quad \sigma_1 \approx \epsilon \\
\end{align*}
\]

with \( 0 \leq m(s_2) < m(s_1) \)
The cryptographic constant-time discipline
Cryptographic constant-time programming

• Protect implementations against timing and cache side-channel attacks

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

```
unsigned not_constant_time (unsigned x, unsigned y, bool secret)
{ if (secret) return y; else return x; }
```

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

- Protect implementations against timing and cache side-channel attacks

- Cryptographic constant-time programs do not:
  - branch on secrets
  - perform memory accesses that depend on secrets

```c
unsigned not_constant_time (unsigned x, unsigned y, bool secret)
{
    if (secret) return y;
    else return x;
}
```

```c
unsigned constant_time1 (unsigned x, unsigned y, bool secret)
{
    return x + (y - x) * secret;
}
```

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

• Protect implementations against timing and cache side-channel attacks

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

```c
unsigned not_constant_time (unsigned x, unsigned y, bool secret)
{ if (secret) return y; else return x; }
```

```c
unsigned constant_time1 (unsigned x, unsigned y, bool secret)
{ return x + (y - x) * secret; }
```

```c
unsigned constant_time2 (unsigned x, unsigned y, bool secret)
{ return x ^ ((y ^ x) & (-((unsigned)secret))); }
```

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

• Several verification tools have been built and used for checking that popular libraries follow the cryptographic constant-time discipline.

• But checking low-level implementations is tricky. It makes:
  • the analysis work harder (e.g. alias analysis),
  • the results of the analysis difficult to understand for programmers.

• Verification at source level is achievable\(^1\), but it needs to be combined with a secure compiler.

\[
\forall P, \text{constantTime}(P) \xrightarrow{?} \text{constantTime(compile}(P)\text{)}
\]

Compilers vs. cryptographic constant-time

```c
unsigned not_constant_time(unsigned x, unsigned y, bool b)
{
    if (b) return y;
    else return x;
}

unsigned constant_time_1(unsigned x, unsigned y, bool b)
{
    return x + (y - x) * b;
}

unsigned constant_time_2(unsigned x, unsigned y, bool b)
{
    return x ^ ((y ^ x) & (-(unsigned)b));
}
```

```assembly
not_constant_time:
    cmpb $0, 12(%esp)
    jne .LBB0_1
    leal 4(%esp), %eax
    movl (%eax), %eax
    retl

.LBB0_1:
    leal 8(%esp), %eax
    movl (%eax), %eax
    retl
constant_time_1:
    cmpb $0, 12(%esp)
    jne .LBB1_1
    leal 4(%esp), %eax
    movl (%eax), %eax
    retl

.LBB1_1:
    leal 8(%esp), %eax
    movl (%eax), %eax
    retl
constant_time_2:
    movl 4(%esp), %ecx
    cmpb $0, 12(%esp)
    jne .LBB2_1
```

Output (0/0) x86-64 clang (trunk) i - 978ms (14804B)
Compilers vs. cryptographic constant-time

```c
int main() {
    unsigned long long x;
    double y;
    x = (unsigned long long)y;
    return 0;
}
```

```
main:
    push    rbp
    mov     rbp, rsp
    movsd   xmm0, QWORD PTR [rbp-8]
    comisd  xmm0, QWORD PTR .LC0[rip]
    jnb     .L2
    movsd   xmm0, QWORD PTR [rbp-8]
    cvttsd2si  rax, xmm0
    mov     QWORD PTR [rbp-16], rax
    jmp     .L3

.L2:
    movsd   xmm0, QWORD PTR [rbp-8]
    movsd   xmm1, QWORD PTR .LC0[rip]
    subsd   xmm0, xmm1
    cvttsd2si  rax, xmm0
    mov     QWORD PTR [rbp-16], rax
    movabs  rax, -9223372036854775808
    xor     QWORD PTR [rbp-16], rax

.L3:
    mov     rax, QWORD PTR [rbp-16]
    mov     QWORD PTR [rbp-16], rax
    mov     eax, 0
    pop     rbp
    ret
```
Lucky Thirteen: Breaking the TLS and DTLS Record Protocols

Nadhem J. AlFardan and Kenneth G. Paterson*
Information Security Group
Royal Holloway, University of London
{nadhem.alfardan.2009, kgp}@rhul.ac.uk
27th February 2013

Abstract

The Transport Layer Security (TLS) protocol aims to provide confidentiality and integrity of data in transit across untrusted networks. TLS has become the de facto secure protocol of choice for Internet and mobile applications. DTLS is a variant of TLS that is growing in importance. In this paper, we present distinguishing and plaintext recovery attacks against TLS and DTLS. The attacks are based on a delicate timing analysis of decryption processing in the two protocols. We include experimental results demonstrating the feasibility of the attacks in realistic network environments for several different implementations of TLS and DTLS, including the leading OpenSSL implementations. We provide countermeasures for the attacks. Finally, we discuss the wider implications of our attacks for the cryptographic design used by TLS and DTLS.

Lucky Microseconds: A Timing Attack on Amazon’s s2n Implementation of TLS

Martin R. Albrecht* and Kenneth G. Paterson**
Information Security Group
Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK
{martin.albrecht, kenny.paterson}@rhul.ac.uk
17th May 2016

Abstract. s2n is an implementation of the TLS protocol that was released in late June 2015 by Amazon. It is implemented in around 6,000 lines of C99 code. By comparison, OpenSSL needs around 70,000 lines of code to implement the protocol. At the time of its release, Amazon announced that s2n had undergone three external security evaluations and penetration tests. We show that, despite this, s2n — as initially released — was vulnerable to a timing attack in the case of CBC-mode ciphersuites, which could be extended to complete plaintext recovery in some settings. Our attack has two components. The first part is a novel variant of the Lucky 13 attack that works even though protections against Lucky 13 were implemented in s2n. The second part deals with the randomised delays that were put in place in s2n as an additional countermeasure to Lucky 13. Our work highlights the challenges of protecting implementations against sophisticated timing attacks. It also illustrates that standard code audits are insufficient to uncover all cryptographic attack vectors.

Keywords TLS, CBC-mode encryption, timing attack, plaintext recovery, Lucky 13, s2n.
A CompCert compiler that preserves cryptographic constant-time
Our contributions

• A machine-checked proof that a mildly modified version of the CompCert compiler preserves cryptographic constant-time

• Proof-engineering challenge: how to turn an existing formally-verified compiler into a formally-verified secure compiler? (CompCert: 100,000 lines of Coq)

• A proof toolkit for proving security preservation
Methodology and challenges

\[ \forall P, \text{constantTime}(P) \rightarrow \text{constantTime(compile}(P)) \]

- Smooth proof methodology to prove that CompCert preserves cryptographic constant-time (CT)
  - Reuse as much as possible existing CompCert simulation proof scripts
  - Follow the motto
    « simple transformations should be easy to prove CT-preserving »
Security property: cryptographic constant-time

- We enrich the CompCert traces of events with two kinds of leakages:
  - the truth value of a condition,
  - a pointer representing the address of
    - either a memory access
    - or a called function.
- We adapt consistently the semantics and still note $S \xrightarrow{t} S'$ the new judgement.
- Event erasure: from $S \xrightarrow{t} S'$ we can extract
  - the compile-only judgement $S \xrightarrow{t}_{\text{comp}} S'$ and
  - the leak-only judgement $S \xrightarrow{t}_{\text{leak}} S'$.
- Program leakage is observed by the $\rightarrow_{\text{leak}}$ semantics.
Security property: cryptographic constant-time

• Involves two executions of a program $P$: need to adapt CompCert simulations diagrams

• $\varphi(s_i, s'_i) = \text{two initial states share the same values for public inputs of } P, \text{ but differ on the values of secret inputs of } P$.

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

• We also provide alternative definitions (avoiding reasoning on infinite executions) and prove their equivalence with the previous property when languages are equipped with a well-formed same-point relation $\equiv$ (where control flow is explicit).
Modelling the same-point relation $s \equiv s'$

- The relation captures the fact that program positions match in both states (including stack pointers).
- We also capture that memory-block allocation histories match.
- In the CompCert languages, the relation satisfies the 4 following properties.

<table>
<thead>
<tr>
<th>$a, a'$ initial states of $P$</th>
<th>$a \equiv a'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \xrightarrow{t} b$</td>
<td>$b \equiv b'$</td>
</tr>
<tr>
<td>$a' \xrightarrow{t} b'$</td>
<td>$a \equiv a'$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$a$ final state of $P$</th>
<th>$a'$ final state of $P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \equiv a'$</td>
<td>$a' \equiv a'$</td>
</tr>
<tr>
<td>$a \xrightarrow{t} b$</td>
<td>$b \equiv b'$</td>
</tr>
<tr>
<td>$a' \xrightarrow{t} b'$</td>
<td>$</td>
</tr>
<tr>
<td>$a \equiv a'$</td>
<td>$a' \equiv a'$</td>
</tr>
</tbody>
</table>

- These properties are useful to prove security property equivalences and soundness of the forthcoming proof methods.
Method #1: leakage preservation

- Simplest situation: a program transformation preserves leakage.
- Traditional CompCert forward-simulation diagram
- Forward simulation implies behaviour preservation (in this setting)

\[
s_1 \approx s_2 \\
\sigma_1 \approx \sigma_2 \approx t + \sigma_2 \\
\text{or} \\
\sigma_1 \approx \varepsilon \rightarrow s_2 \\
\text{with } 0 \leq m(s2) < m(s1)
\]
# 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>Cshmggen</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></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></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>
Method #2: leakage erasing simulation

• Some optimisations erase leakages (e.g. a memory load is replaced by a load from a register).
• They are still constant-time preserving as long as their decision to erase this information does not depend on secret values.
• We slightly adapt the forward-simulation diagram.

\[ s_1 \approx_n t \approx_n s_2 \]
\[ \sigma_1 \xrightarrow{\tau} n \xrightarrow{\sigma_2} \]
\[ s_1 \approx_n \tau \approx_n s_2 \]
\[ s_1 \xrightarrow{\varepsilon} s_2 \]

\[ \tau = t \text{ or } (\tau = \varepsilon \text{ and } t \text{ is leak only}) \]

The previous proof script requires very few changes!
## 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><strong>Leakage erasing</strong></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><strong>Leakage erasing</strong></td>
<td>Common subexpression elimination</td>
</tr>
<tr>
<td>Deadcode</td>
<td><strong>Leakage erasing</strong></td>
<td>Redundancy elimination</td>
</tr>
<tr>
<td>Allocation</td>
<td><strong>Leakage erasing</strong></td>
<td>Register allocation</td>
</tr>
<tr>
<td>Tunneling</td>
<td><strong>Leakage erasing</strong></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>

Method #2 used 5 times among 17 proofs

---

Table 1. Compilation passes and how we prove them constant-time preserving trace transformations. This table shows that thanks to our proof techniques, most transformations were relatively convenient to adapt, and heavily benefit from existing proof scripts.

5.2 Leakage Preservation

The first case we consider is one where leakages are preserved by compilation. This case is actually the one covered by compiler correctness and can thus leverage directly the many already existing lemmas and theorems of CompCert. It is directly stated and proved on the instrumented semantics. Let \( S \) be a safe source program and \( T \) be the target compiled program. If \( S \) is constant-time w.r.t. \( \Delta \) then \( T \) is constant-time w.r.t. \( \Delta \), provided that the relation \( \mathcal{P} \) between program states is a trace preserving simulation (w.r.t. to \( \Delta \)) as defined in Figure 2.

While this simulation is quite basic and restrictive, it is still satisfied by a few transformation passes (6 among 17, see Table 1), including passes that do not optimize conditional branches, nor memory accesses. These passes are then especially easy to adapt to constant-time preservation since we can reuse the same proof script as for the original simulation proof. This is a great time saver for a pass like RTLgen (which transforms a structured Cminor program into a RTL control-flow graph) because its soundness proof is quite verbose. However, many optimizations do not preserve leakages. Indeed, optimizations such as common subexpression elimination (CSE) or constant
Step-counting simulation $\approx_n$

- We make sure that the prediction of $n$ does not depend on secrets by requiring it will only depend on the control states.
- Given a same-point relation $\equiv$, we define a notion $\approx_n$ of same-point congruence.

\[
\begin{array}{c}
\begin{array}{c}
s \\ \equiv_n \\ s'
\end{array} \\
\begin{array}{c}
\approx_n \\ \rightarrow \\
\begin{array}{c}
\sigma \\ \equiv_n \\
\sigma'
\end{array}
\end{array}
\end{array}
\Rightarrow n = n'
\]
Method #3: Leak-transforming by memory-injection simulation

- Some transformations alter the memory layout.
- Leaky pointers are not preserved.
- Still, there exists a leakage transformation that maps the source leakage trace to the target leakage trace.

Our solution:
- Use of step-counting simulations (with more advanced counting)
- and explicit memory injections
  (tracking how leaky pointers are transformed)
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>Cshmggen</td>
<td>Trace preservation</td>
<td>Type elaboration, simplification of control</td>
</tr>
<tr>
<td>Cminorgen</td>
<td>Memory injection</td>
<td>Stack allocation</td>
</tr>
<tr>
<td>Selection</td>
<td>Leakage 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>Leakage erasing</td>
<td>Common subexpression elimination</td>
</tr>
<tr>
<td>Deadcode</td>
<td>Leakage erasing</td>
<td>Redundancy elimination</td>
</tr>
<tr>
<td>Allocation</td>
<td>Leakage erasing</td>
<td>Register allocation</td>
</tr>
<tr>
<td>Tunneling</td>
<td>Leakage 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>Memory injection</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>Memory injection</td>
<td>Stack allocation</td>
</tr>
<tr>
<td>Selection</td>
<td>Leakage 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>Leakage erasing</td>
<td>Common subexpression elimination</td>
</tr>
<tr>
<td>Deadcode</td>
<td>Leakage erasing</td>
<td>Redundancy elimination</td>
</tr>
<tr>
<td>Allocation</td>
<td>Leakage erasing</td>
<td>Register allocation</td>
</tr>
<tr>
<td>Tunneling</td>
<td>Leakage 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>Memory injection</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>

+ 3 times with a slight generalisation…
## 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>Memory injection</td>
<td>Stack allocation</td>
</tr>
<tr>
<td>Selection</td>
<td>Leakage 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>Leakage erasing</td>
<td>Common subexpression elimination</td>
</tr>
<tr>
<td>Deadcode</td>
<td>Leakage erasing</td>
<td>Redundancy elimination</td>
</tr>
<tr>
<td>Allocation</td>
<td>Leakage erasing</td>
<td>Register allocation</td>
</tr>
<tr>
<td>Tunneling</td>
<td>Leakage erasing</td>
<td>Branch tunneling</td>
</tr>
<tr>
<td>Linearize</td>
<td><strong>CT-simulation</strong></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>Memory injection</td>
<td>Laying out stack frames</td>
</tr>
<tr>
<td>Asmgen</td>
<td>Trace transformation</td>
<td></td>
</tr>
</tbody>
</table>

---

Experiments
Conclusion and perspectives

• 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
  • Combine CT-CompCert with verified C crypto programs
  • Explore other observational information-flow policies and adapt CompCert