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

Verified Software workshop, Cambridge, 2019-09-25

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

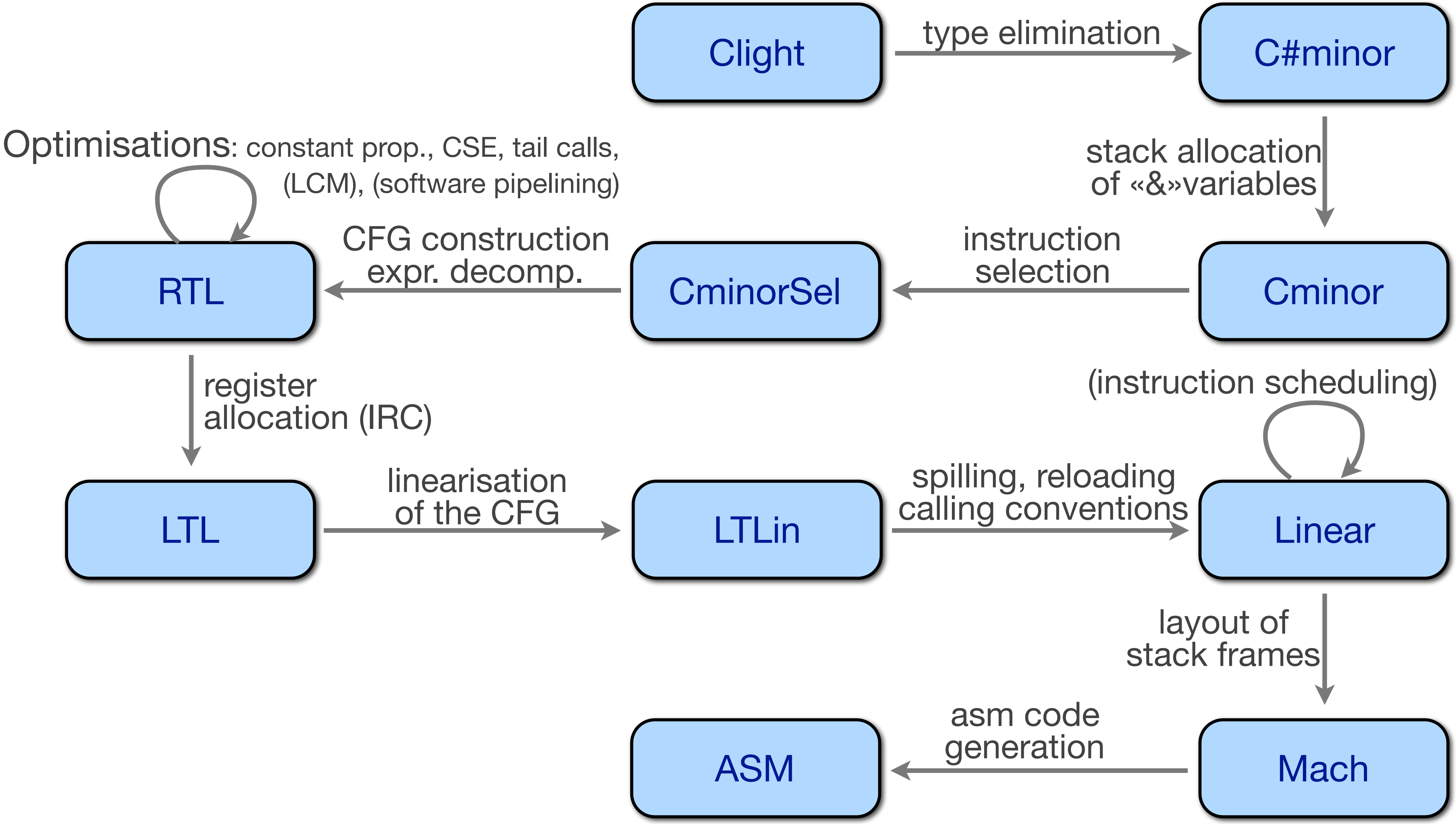
- Fly-by-wire software, Airbus A380, FCGU

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».

Behaviours = termination / divergence / undefined («going wrong»)
+ (finite or infinite) trace of I/O operations performed

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



Clight

type elimination

C#minor

stack allocation
of «&»variables

Cminor

instruction
selection

CminorSel

CFG construction
expr. decomp.

RTL

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

register
allocation (IRC)

LTL

linearisation
of the CFG

LTLin

spilling, reloading
calling conventions

Linear

(instruction scheduling)

layout of
stack frames

Mach

asm code
generation

ASM

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

Operational semantics

$S \xrightarrow{t} S'$

$S \xrightarrow{t}{}^{*} S'$

$S \xrightarrow{t}{}^{n} S'$

$S \xrightarrow{t}{}^{+} S'$

$S \xrightarrow{t} \infty$

Clight

C#minor

RTL

CminorSel

Cminor

LTL

LTLin

Linear

ASM

Mach

Proof methodology: forward simulation

Ingredients

- simulation relation ≈ between source and target states
- measure m from source states to a well-founded set

$$\begin{tikzcd}
s1 \arrow[r, "t"] \arrow[d, dash, dotted, "\approx"'] & s2 \arrow[d, dash, dotted, "\approx"] \\
\sigma1 \arrow[r, "t"', "+"] & \sigma2
\end{tikzcd}$$

or

$$\begin{tikzcd}
s1 \arrow[r, "\varepsilon"] \arrow[d, dash, dotted, "\approx"'] & s2 \arrow[dl, dash, dotted, "\approx"] \\
\sigma1 &
\end{tikzcd}$$

with $0 \leq m(s2) < m(s1)$

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; }
```

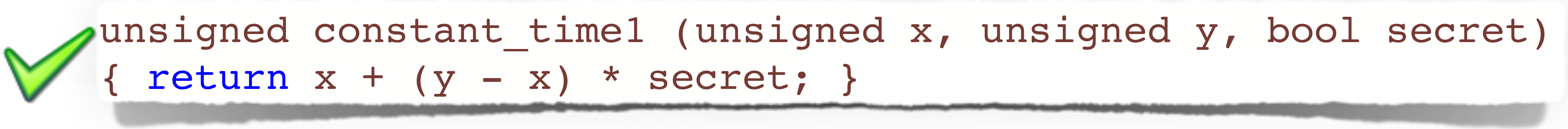
- There are constant-time implementations of many cryptographic algorithms: AES, DES, RSA, etc.

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

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



```
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.

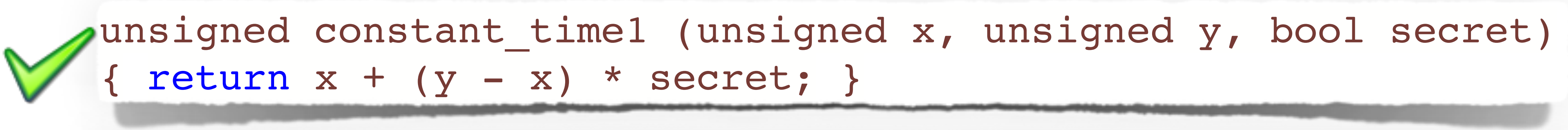
Cryptographic constant-time programming

- Protect implementations against timing and cache side-channel attacks

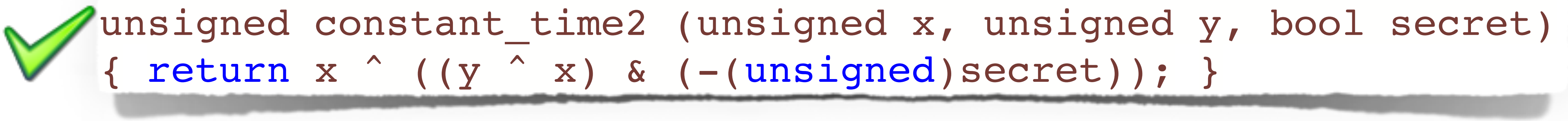
- Cryptographic constant-time programs do not:
  - branch on secrets
  - perform memory accesses that depend on secrets


✗

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


✓

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


✓

```
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.

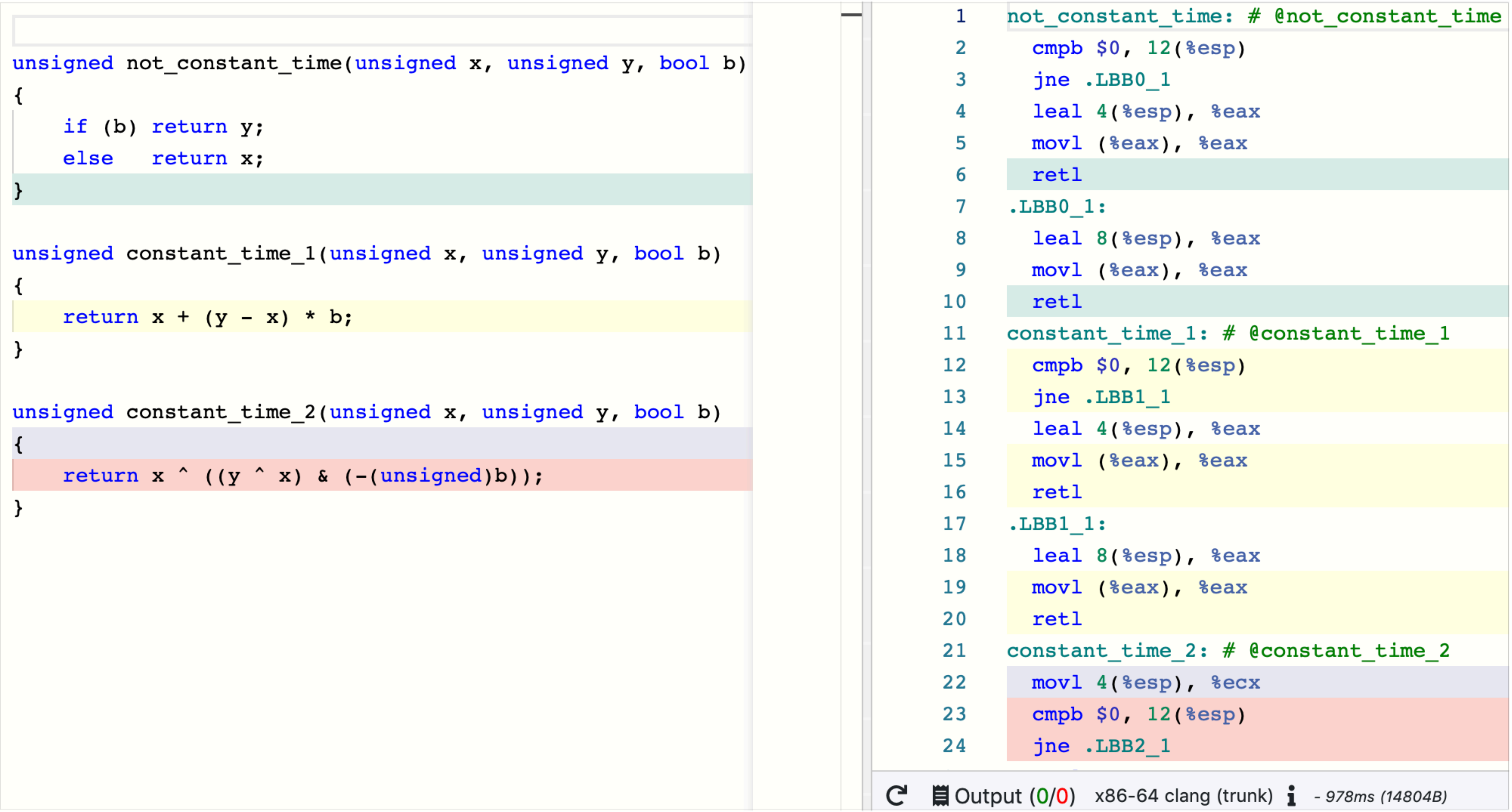
- Verification at source level is achievable[1], but it needs to be combined with a **secure compiler**.

$$\forall P, \mathsf{constantTime}(P) \overset{?}{\rightarrow} \mathsf{constantTime}(\mathsf{compile}(P))$$

---

[1] S.Blazy, D.Pichardie, A.Trieu. Verifying constant-time implementations by abstract interpretation. Journal of Computer Security. 01/2019.

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));
}
```

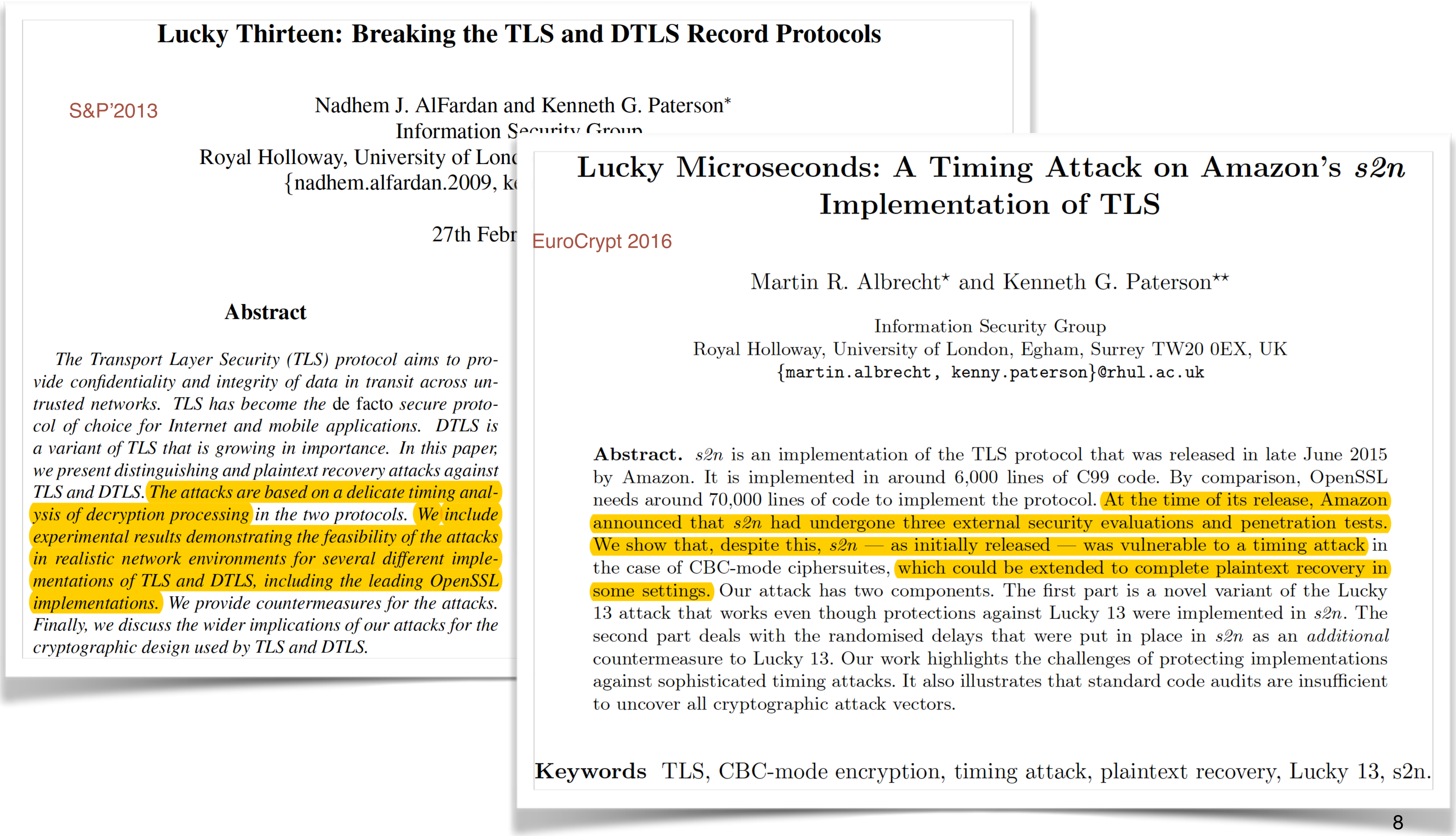
```asm
not_constant_time: # @not_constant_time
  cmpb $0, 12(%esp)
  jne .LBB0_1
  leal 4(%esp), %eax
  movl (%eax), %eax
  retl
.LBB0_1:
  leal 8(%esp), %eax
  movl (%eax), %eax
  retl
constant_time_1: # @constant_time_1
  cmpb $0, 12(%esp)
  jne .LBB1_1
  leal 4(%esp), %eax
  movl (%eax), %eax
  retl
.LBB1_1:
  leal 8(%esp), %eax
  movl (%eax), %eax
  retl
constant_time_2: # @constant_time_2
  movl 4(%esp), %ecx
  cmpb $0, 12(%esp)
  jne .LBB2_1
```

Compilers vs. cryptographic constant-time

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

```asm
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
```

Compilers vs. cryptographic constant-time



Lucky Thirteen: Breaking the TLS and DTLS Record Protocols

S&P'2013

Nadhem J. AlFardan and Kenneth G. Paterson*
Information Security Group
Royal Holloway, University of Lon
{nadhem.alfardan.2009, k

27th Febr

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

EuroCrypt 2016

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`

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

$$\forall P, \mathsf{constantTime}(P) \overset{?}{\to} \mathsf{constantTime}(\mathsf{compile}(P))$$

- 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
- $\varphi(s_i, s_i')$ = two initial states share the same values for public inputs of P, but differ on the values of secret inputs of P.
- A program P is **constant-time secure** w.r.t. φ 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.

$$\boxed{a\,,\ a' \text{ initial states of } P \quad \Longrightarrow \quad a \equiv a'}$$

$$\boxed{\begin{array}{c} a \text{ final state of } P \\ a \equiv a' \end{array} \quad \Longrightarrow \quad a' \text{ final state of } P}$$

$$\boxed{\begin{array}{c} a \xrightarrow{t} b \\ a' \xrightarrow{t} b' \\ a \equiv a' \end{array} \quad \Longrightarrow \quad b \equiv b'}$$

$$\boxed{\begin{array}{l} a \xrightarrow{t} b \\ a' \xrightarrow{t'} b' \\ a \equiv a' \end{array} \quad \Longrightarrow \quad |t| = |t'|}$$

- 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)

$$\begin{tikzcd}
s1 \arrow[r, "t"] \arrow[d, dash, dashed, "\approx"'] & s2 \arrow[d, dash, dotted, "\approx"] \\
\sigma1 \arrow[r, "t"{near start}, "+"{near end}] & \sigma2
\end{tikzcd}$$

or

$$\begin{tikzcd}
s1 \arrow[r, "\varepsilon"] \arrow[d, dash, dashed, "\approx"'] & s2 \arrow[dl, dash, dotted, "\approx"] \\
\sigma1 &
\end{tikzcd}$$

with $0 \le m(s2) < m(s1)$

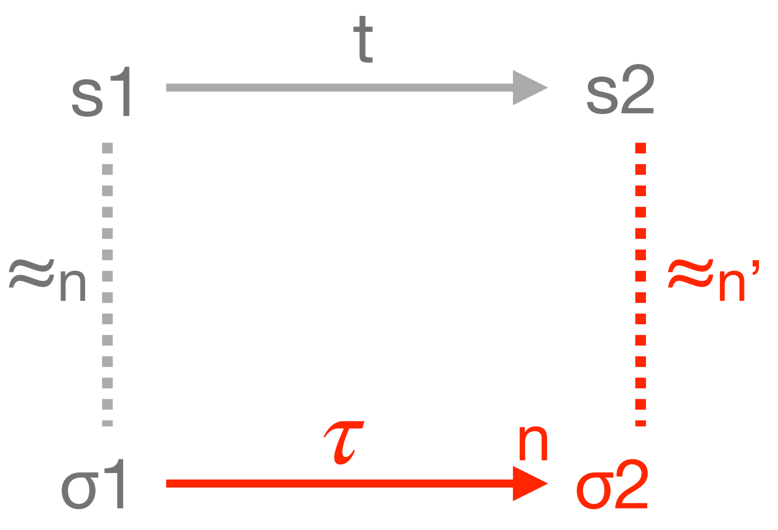
A palette of proof methods

Method #1 used 6 times among 17 proofs

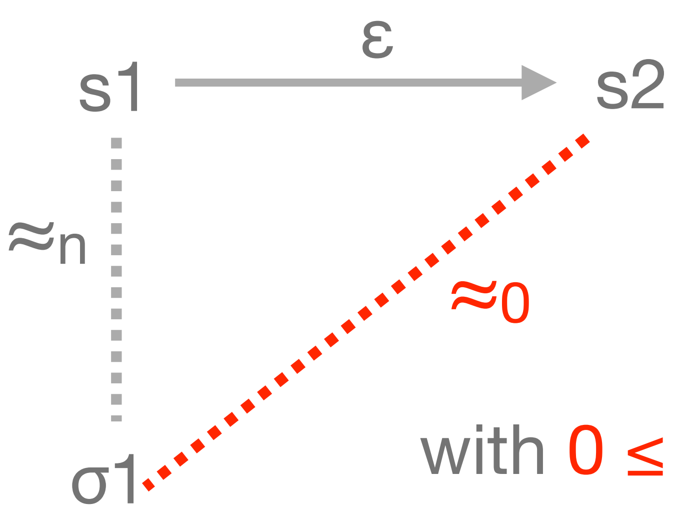
Compiler pass	Diagram used	Explanation on the pass
Cshmgen	Trace preservation	Type elaboration, simplification of control
Cminorgen		Stack allocation
Selection		Recognition of operators and addr. modes
RTLgen	Trace preservation	Generation of CFG and 3-address code
Tailcall	Trace preservation	Tailcall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

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.



or



with $0 \leq m(s2) < m(s1)$

$\tau = t$ or ($\tau = \varepsilon$ and t is leak only)

The previous proof script requires very few changes!

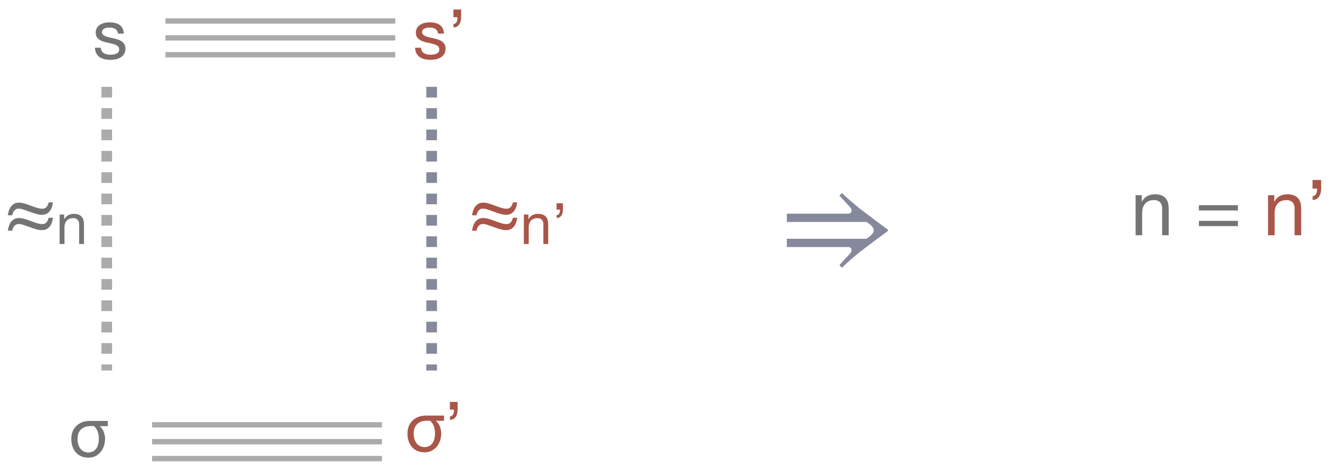
A palette of proof methods

Method #2 used 5 times among 17 proofs

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

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**.



$$\begin{tikzcd} s \arrow[r, phantom, "\equiv"] \arrow[d, dash, dotted, "\approx_n"'] & s' \arrow[d, dash, dotted, "\approx_{n'}"] & & \\ \sigma \arrow[r, phantom, "\equiv"] & \sigma' & \Rightarrow & n = n' \end{tikzcd}$$

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

Method #3 used 2 times
among 17 proofs

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

A palette of proof methods

+ 3 times with a slight generalisation…

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

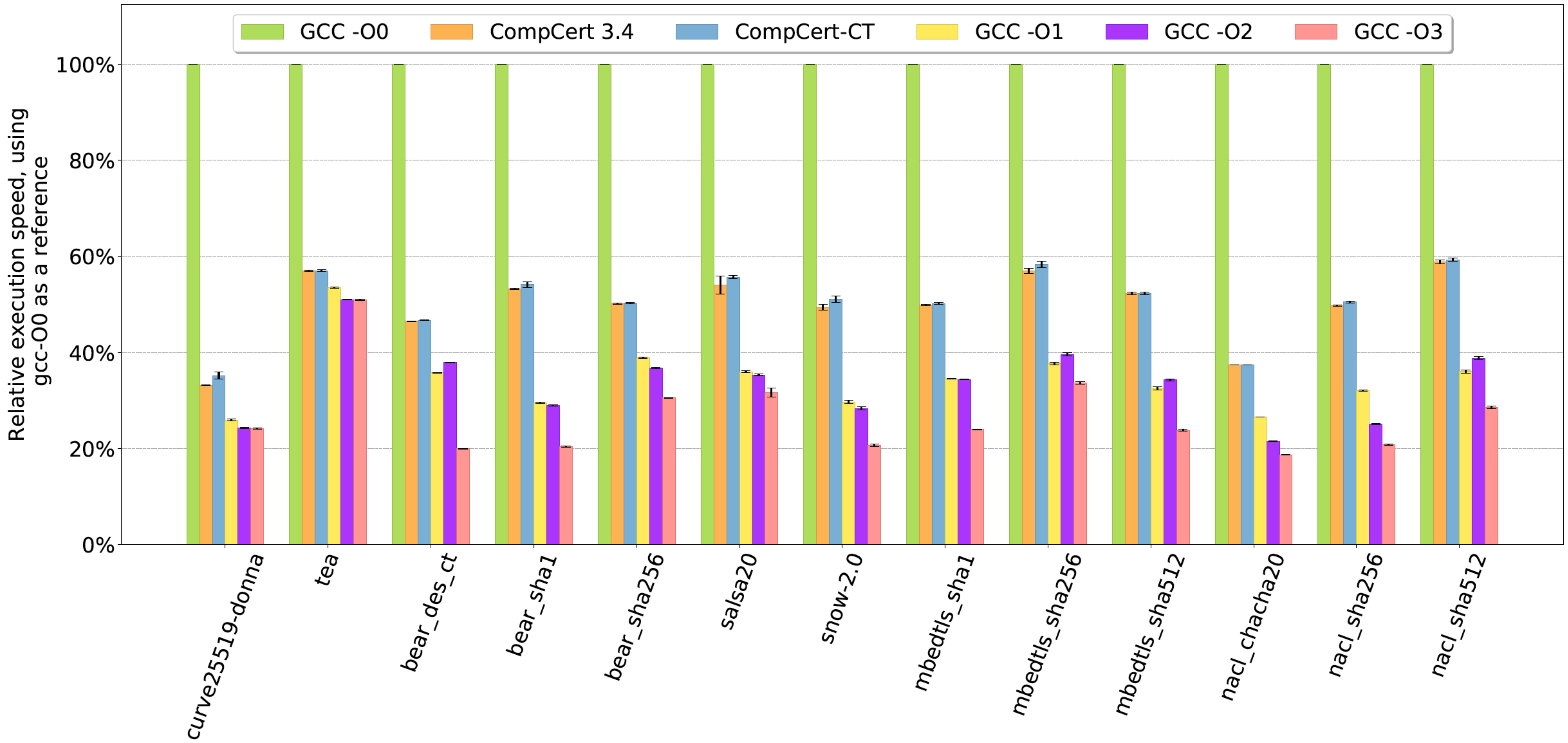
A palette of proof methods

Method[2] #4 used only 1 time

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

2. G.Barthe, B. Grégoire, and V. Laporte. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic Constant-Time. *CSF,* 2018.

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