Semantic preservation of constant-time policies during compilation

Sandrine Blazy

joint work with Gilles Barthe, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, Alix Trieu

GT MFS, Fréjus, 2022-03-21

Information-flow policies for side-channel leakages

- **Observational non-interference**: observing program leakage during execution does not reveal any information about secrets

$$P, \sigma \xrightarrow{\ell} \sigma'$$

- **Indistinguishability property** $\varphi(\sigma, \sigma')$: share public values, but may differ on secret values

$$P, \sigma_1 \xrightarrow{\ell_1} \sigma_1'$$

$$P, \sigma_2 \xrightarrow{\ell_2} \sigma_2'$$

with $\varphi(\sigma_1, \sigma_2)$

implies $\ell_1 = \ell_2$

Information-flow policies for side-channel leakages

$$i, \sigma \xrightarrow{\ell} i', \sigma'$$

Cryptographic constant-time

Leakages: boolean guards and memory accesses

CompCert C compiler

Challenges:

- reuse of correctness proofs
- proof scalability

Constant-resource

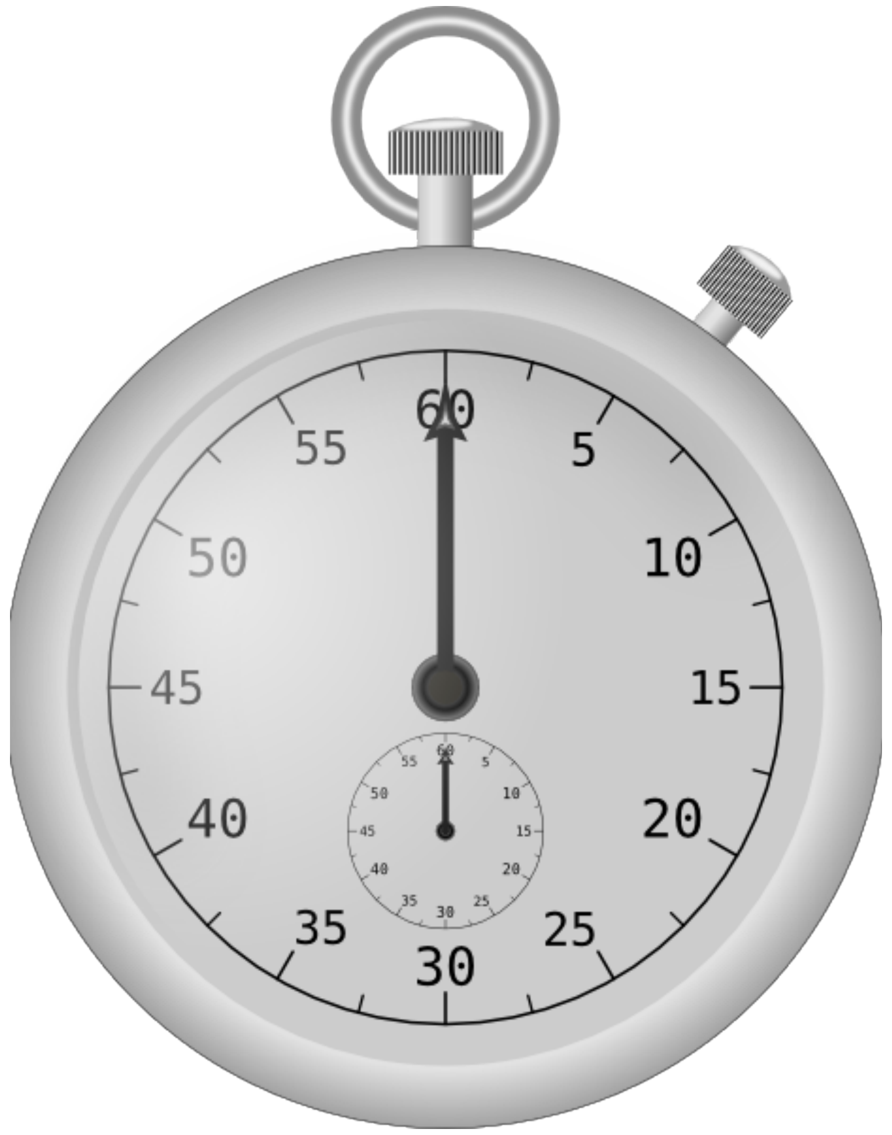
Leakages: amount of resources consumed

Toy language, basic optimisations

Challenges:

- find the relevant security policy
- need to modify the compiler

Part one: cryptographic constant-time



Cryptographic constant-time programming

- Leakage: ℓ ::= ε | guard b | read a v | write a v

```
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 cryptographic 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, but it needs to be combined with a **secure compiler**.

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

Compilers vs. cryptographic constant-time policy

```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
```

Output (0/0) x86-64 clang (trunk) - 978ms (14804B)

Compilers vs. cryptographic constant-time policy

```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
```

Output (0/0) x86-64 gcc 8.3 - 849ms (12804B)

Cryptographic constant-time attacks

S&P'2013

Lucky Thirteen: Breaking the TLS and DTLS Record Protocols

Nadhem J. AlFardan and Kenneth G. Paterson*
Information Security Group
Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK
{nadhem.alfardan.2009, kenny.paterson}@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.*

1.0 [31], which roughly matches TLS 1.1 and DTLS 1.2 [32
which aligns with TLS 1.2.

Both TLS and DTLS are actually protocol suites, rather tha
single protocols. The main component of (D)TLS that co
cerns us here is the Record Protocol, which uses symmetr
key cryptography (block ciphers, stream ciphers and MAC a
gorithms) in combination with sequence numbers to build a s
cure channel for transporting application-layer data. Other m
jor components are the (D)TLS Handshake Protocol, which
responsible for authentication, session key establishment a
ciphersuite negotiation, and the TLS Alert Protocol, which ca
ries error messages and management traffic. Setting aside de
icated authenticated encryption algorithms (which are yet
see widespread support in TLS or DTLS implementations), t
(D)TLS Record Protocol uses a MAC-Encode-Encrypt (ME
construction. Here, the plaintext data to be transported is fi

EuroCrypt 2016

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

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.

CompCert


continuations
formal-semantics
induction
data-flow
small-step
error-monad
ordered-types
intermediate-language
abstract-syntax
graph
integers
maps
interpreter
floats
RTL
run-time-values
verified-compiler
solver
control-flow
simulation-proof
observable-events
reference
memory-model

The CompCert formally verified compiler

Proving semantic properties on realistic compilers requires a machine-checked proof

CompCert

- a moderately optimizing C compiler
- programmed and verified using the Coq proof assistant
- used in commercial settings and for software certification

CompCert's main theorem states that the compiler

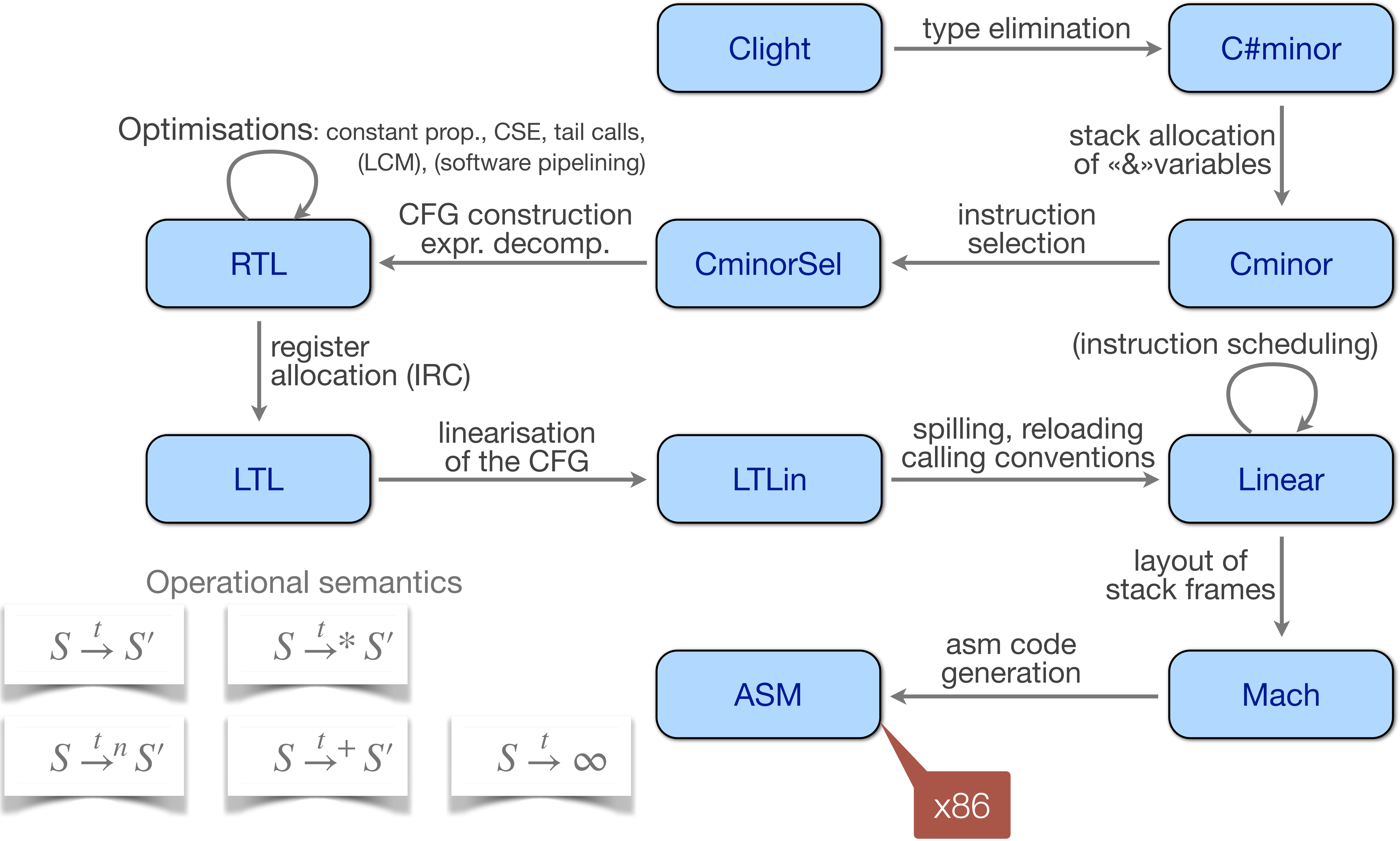
- preserves observational behaviors
- preserves memory safety

`Nothing about side-channels attacks`

Our contributions

- Make precise what preservation through compilation means for cryptographic constant-time
- Provide a machine-checked proof that a mildly modified version of the CompCert compiler preserves the cryptographic constant-time policy
- Explain how to turn an existing formally-verified compiler into a formally-verified secure compiler
- Provide a proof toolkit for proving security preservation with simulation diagrams

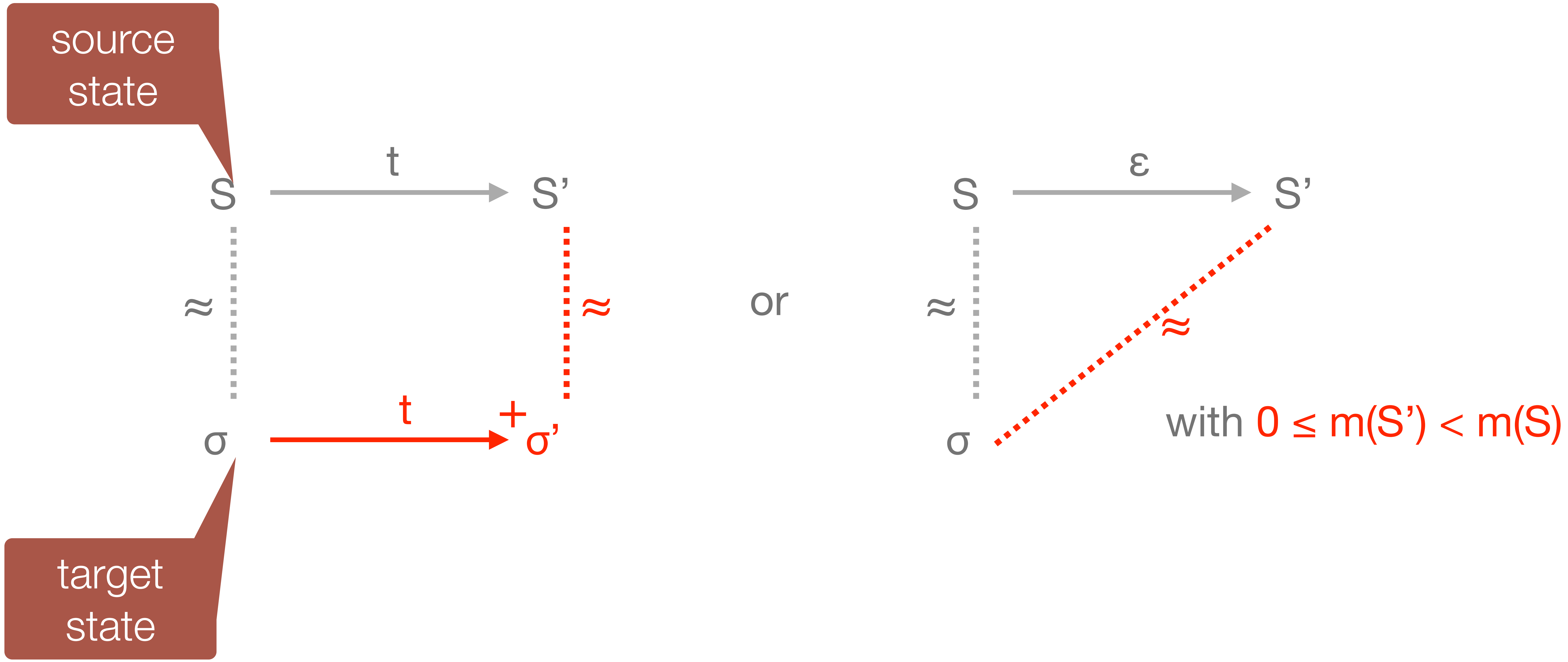
CompCert compiler: 10 languages, 17 passes



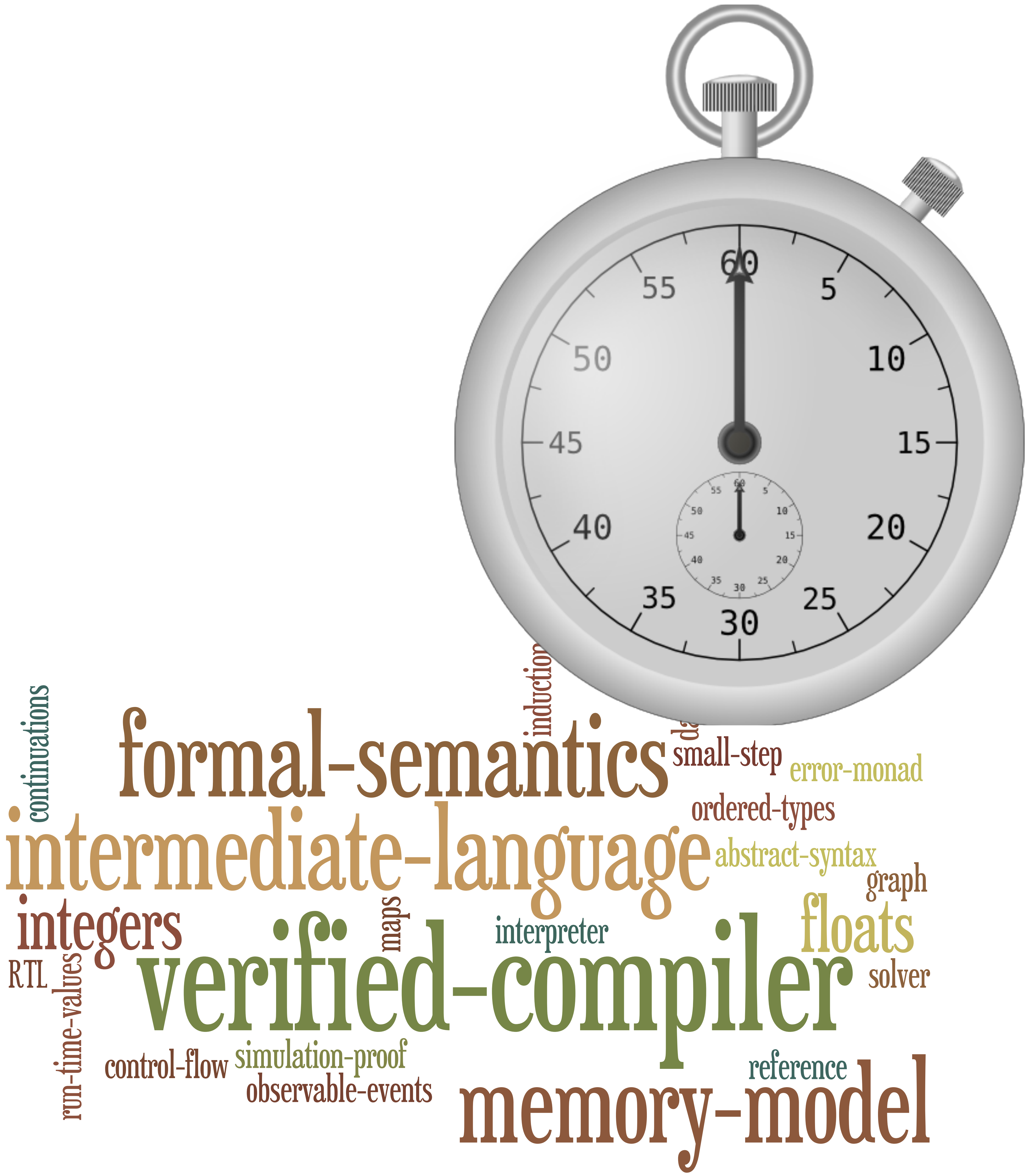
Proof methodology: forward simulation

Ingredients

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



A CompCert compiler that preserves cryptographic constant-time



Security policy: cryptographic constant-time
Defining leakages

$$\frac{\langle e, \sigma \rangle \overset{\ell_0}{\Downarrow} \mathit{true} \qquad \langle p_1, \sigma \rangle \overset{\ell_1}{\Downarrow} \sigma'}{\langle \mathbf{\mathit{if}}\ (e)\ \{p_1\}\ \{p_2\}, \sigma \rangle \overset{\ell_0 \cdot \mathit{true} \cdot \ell_1}{\Downarrow} \sigma'}$$

- We enrich the CompCert traces of events with **leakages**:
  - truth value of a condition,
  - pointer representing the address of either a memory access or a called function.
- We adapt the CompCert 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 $S \xrightarrow{t}_{\text{leak}} S'$.

Security policy: cryptographic constant-time
Semantic preservation

- Indistinguishability property $\varphi(S, S')$: two initial states share the same values for public inputs, but differ on the values of secret inputs

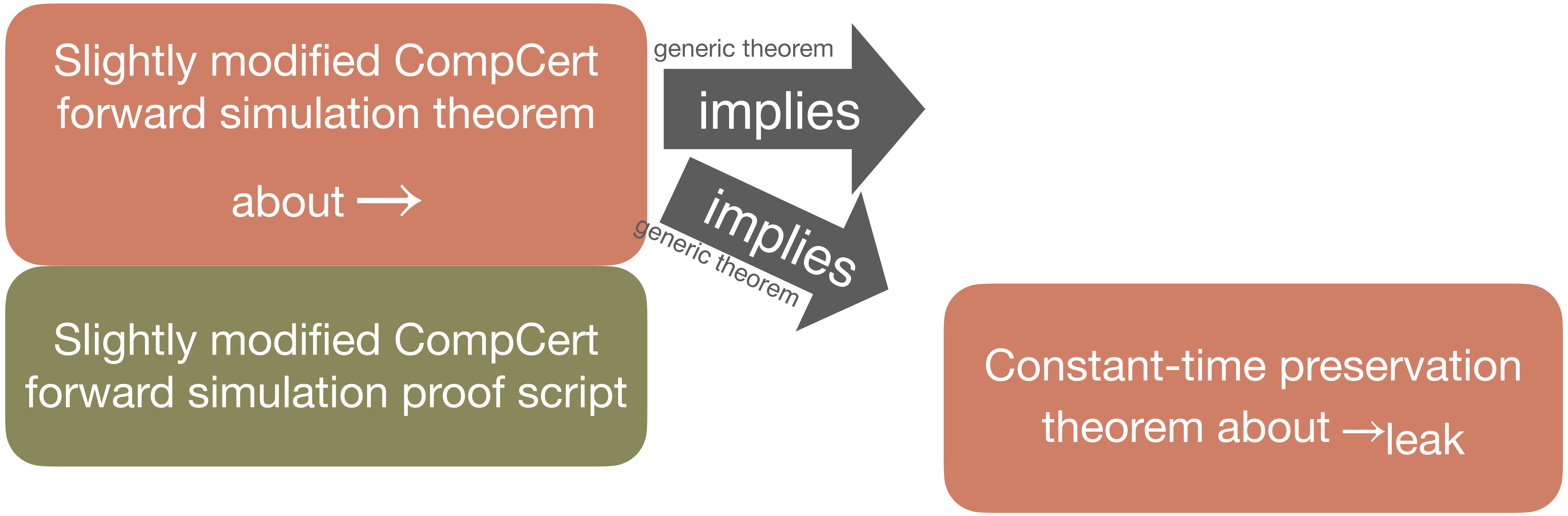
- P is **constant-time secure w.r.t.** φ

Main theorem (preservation of constant-time security)
Let P be a safe Clight source program that is compiled into an x86 assembly program P'.
If P is constant-time secure w.r.t. φ, then so is P'.

$$
\begin{array}{ll}
s \xrightarrow{t}{}^{*} \text{leak} & \\
\varphi & \text{implies } t = t' \\
s' \xrightarrow{t'}{}^{*} \text{leak} &
\end{array}
$$

Proving cryptographic constant-time preservation

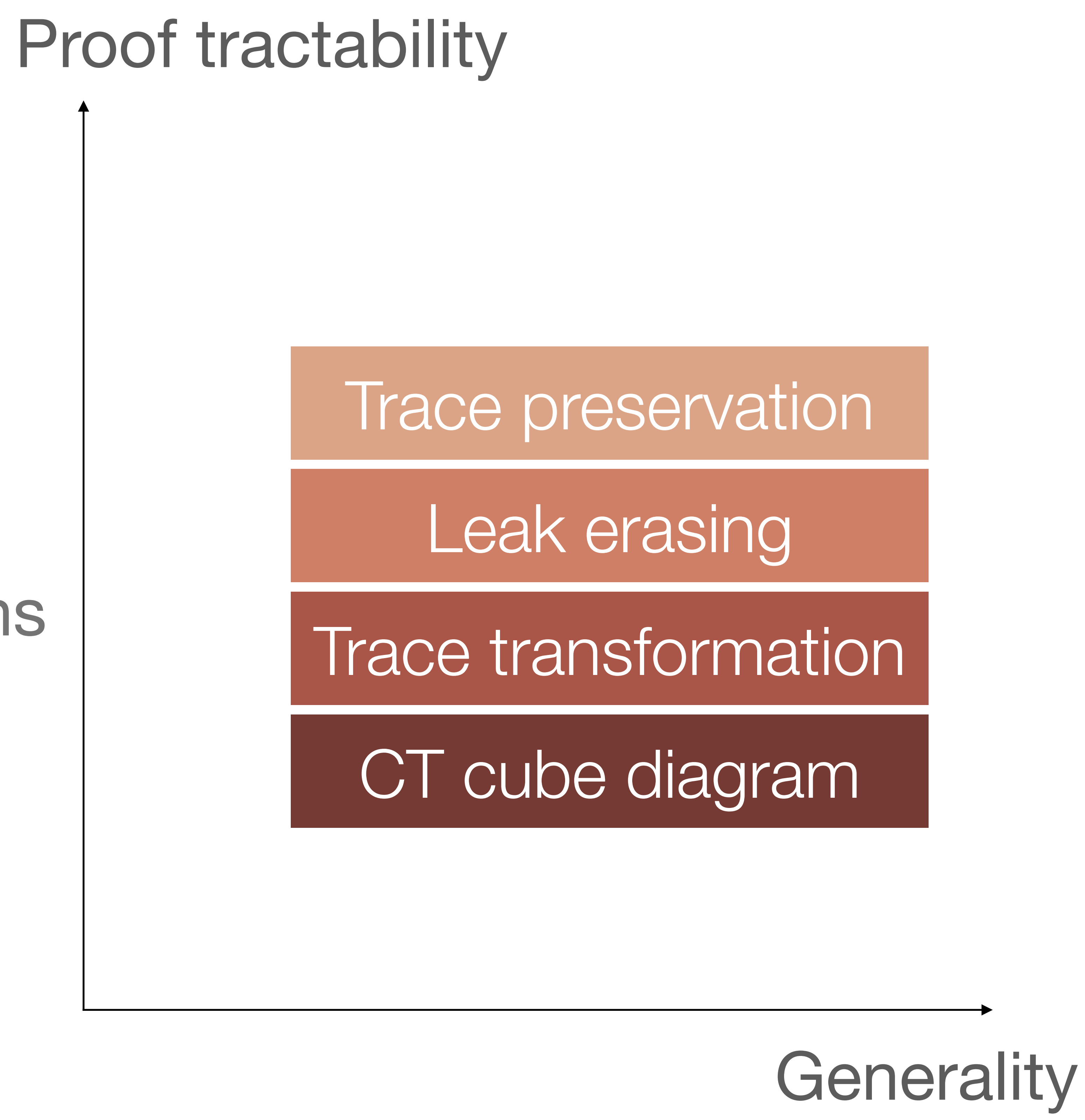
- The involved semantics is the leak-only semantics $\rightarrow_{\text{leak}}$.
- Existing CompCert simulation diagrams deal with the compile-only semantics $\rightarrow_{\text{comp}}$.
- Our proof-engineering strategy is to benefit as much as possible from the **proof scripts** of these diagrams.



Four proof techniques

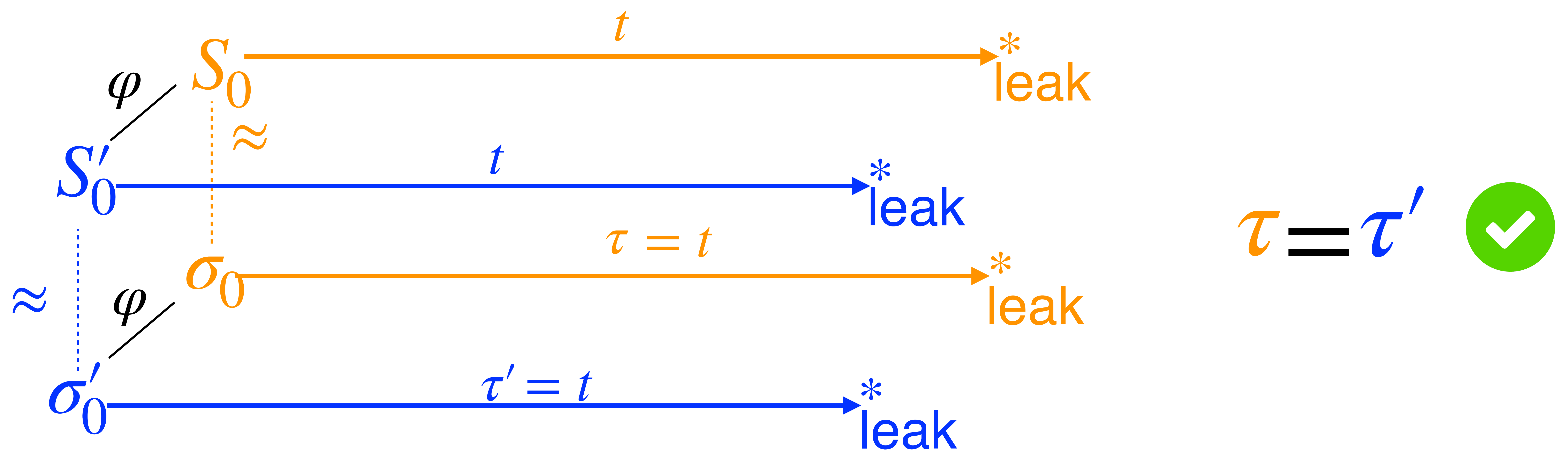
- Trade-off between generality and proof tractability

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



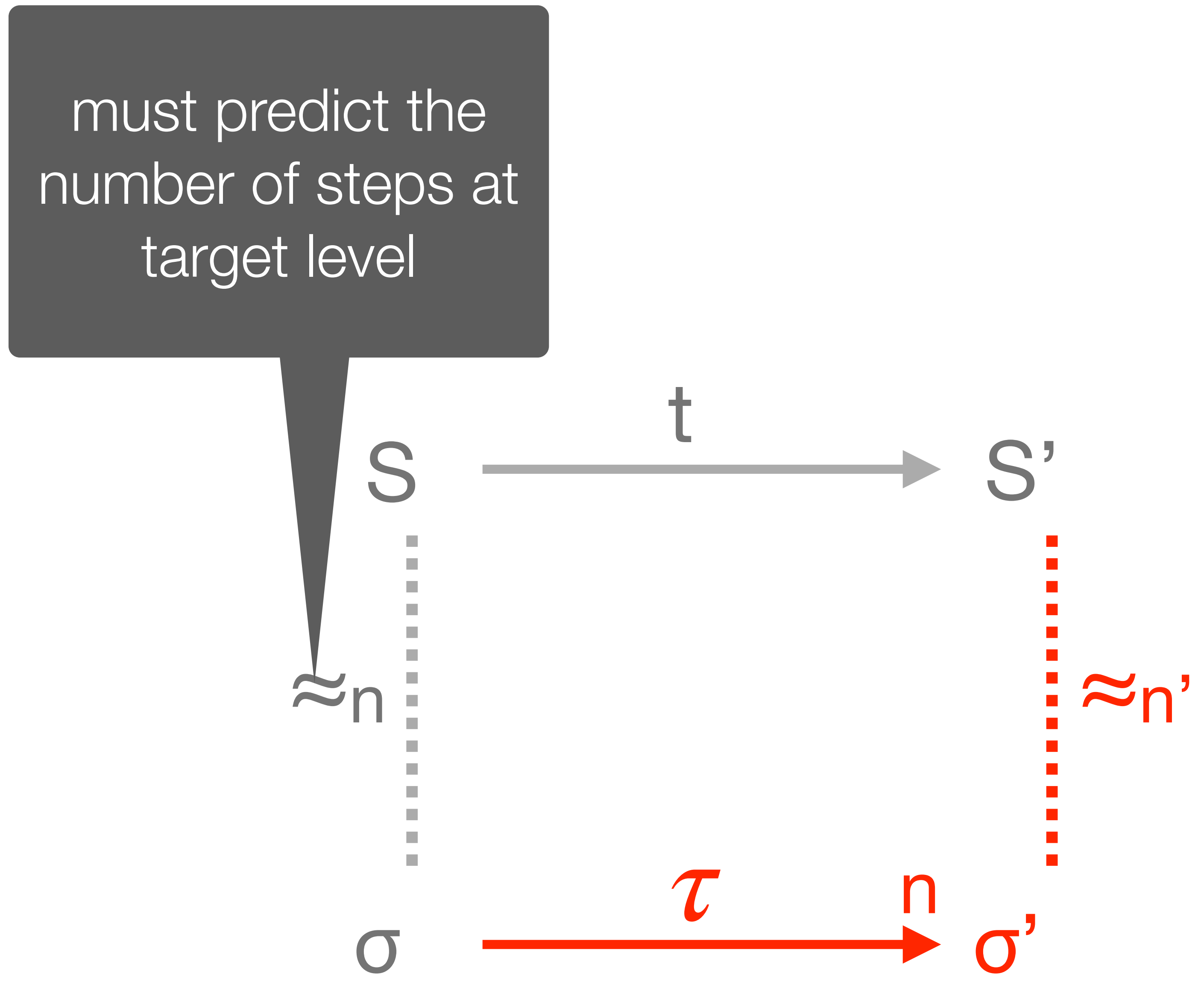
Proving cryptographic constant-time preservation
Method #1: leakage preservation

- Simplest situation: a program transformation preserves the trace of leakages
- Only need to prove the traditional CompCert's forward simulation diagram on $\rightarrow$
- This forward simulation implies behaviour preservation.



Proving cryptographic constant-time preservation Method #2: leakage erasing simulation

- Some optimisations erase leakages.
- 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.



The previous proof script requires very few changes!

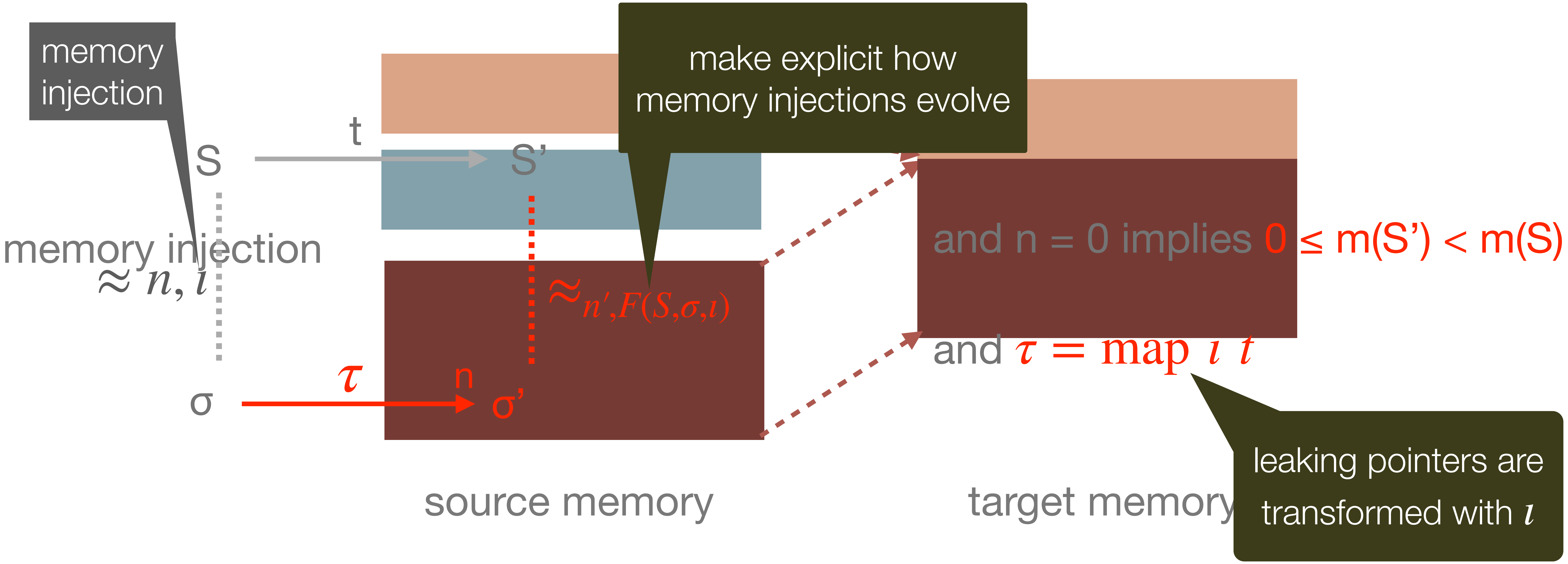
and n = 0 implies $0 \leq m(S') < m(S)$

and $\tau = t$

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

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.



A palette of proof methods

Trace preservation

Leak erasing

Trace transformation

CT cube diagram

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

A palette of proof methods

- Trace preservation 6/17
- Leak erasing
- Trace transformation
- CT cube diagram

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

A palette of proof methods

Trace preservation 6/17

Leak erasing 5/17

Trace transformation

CT cube diagram

Compiler pass	Diagram used	Explanation on the pass
Cshmgen	Trace preservation	Type elaboration, simplification of control
Cminorgen		Stack allocation
Selection	**Leak 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	**Leak erasing**	Common subexpression elimination
Deadcode	**Leak erasing**	Redundancy elimination
Allocation	**Leak erasing**	Register allocation
Tunneling	**Leak 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

A palette of proof methods

Trace preservation	6/17
Leak erasing	5/17
Trace transformation	5/17
CT cube diagram	

Compiler pass	Diagram used	Explanation on the pass
Cshmgen	Trace preservation	Type elaboration, simplification of control
Cminorgen	Trace transformation	Stack allocation
Selection	Leak 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	Leak erasing	Common subexpression elimination
Deadcode	Leak erasing	Redundancy elimination
Allocation	Leak erasing	Register allocation
Tunneling	Leak erasing	Branch tunneling
Linearize		Linearization of CFG
CleanupLabels	Trace preservation	Removal of unreferenced labels
Debugvar	Trace preservation	Synthesis of debugging information
Stacking	Trace transformation	Laying out stack frames
Asmgen	Trace transformation	Emission of assembly code

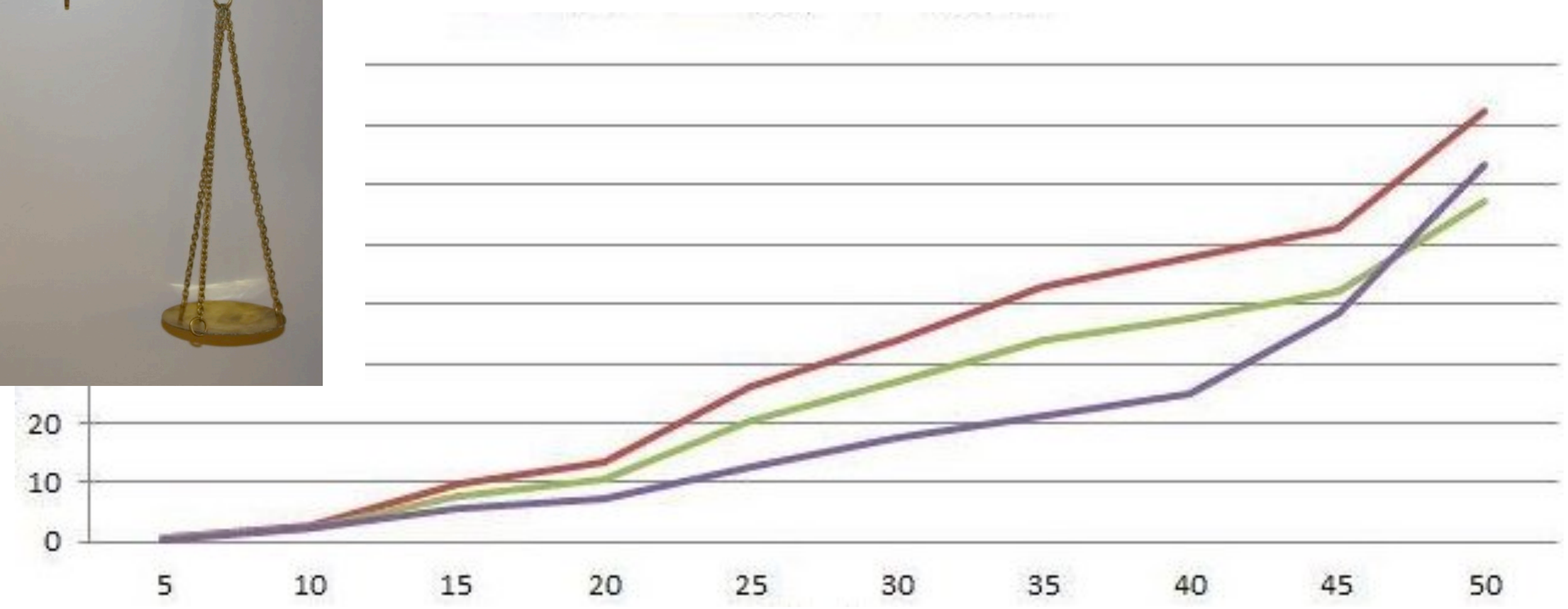
A palette of proof methods

Trace preservation	6/17
Leak erasing	5/17
Trace transformation	5/17
CT cube diagram	1/17

Compiler pass	Diagram used	Explanation on the pass
Cshmgen	Trace preservation	Type elaboration, simplification of control
Cminorgen	Trace transformation	Stack allocation
Selection	Leak 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	Leak erasing	Common subexpression elimination
Deadcode	Leak erasing	Redundancy elimination
Allocation	Leak erasing	Register allocation
Tunneling	Leak erasing	Branch tunneling
Linearize	**CT cube diagram**	Linearization of CFG
CleanupLabels	Trace preservation	Removal of unreferenced labels
Debugvar	Trace preservation	Synthesis of debugging information
Stacking	Trace transformation	Laying out stack frames
Asmgen	Trace transformation	Emission of assembly code

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

Part two: constant resource



The constant-resource (CR) policy

Relax the cryptographic contant-time policy to allow balanced branches

Leakage: amount of resources consumed during an execution

Every construct of the language consumes a constant amount of resources.

Example of CR-secure program

```
if (secret)
   { x = a*b;
     y = (a*b)+c+d; }
else
   { x = a+b;
     y = (a+b)*c*d; }
```

6 accesses to variables,
2 additions,
2 multiplications,
2 assignments

The constant-resource policy

$$i_a; i_b, \sigma_1 \xrightarrow{\ell_1 ++ \ell_1'} \sigma_1'$$

$$i_a; i_b, \sigma_2 \xrightarrow{\ell_2 ++ \ell_2'} \sigma_2'$$

with $\varphi(\sigma_1, \sigma_2)$

implies
$\ell_1 + + \ell_1' = \ell_2 + + \ell_2'$

The non-cancelling property of leakages fails for constant-resource programs.

$\ell_1 + + \ell_1' = \ell_2 + + \ell_2' \implies \ell_1 = \ell_2 \wedge \ell_1' = \ell_2'$ when $|\ell_1| = |\ell_2|$ and $|l_1'| = |l_2'|$

Constant-resource policy and compilation

Preserving the constant-resource policy requires to define some security-enhancing program transformations

```
if (secret)
   { x = a*b;
     y = (a*b)+c+d; }
else
   { x = a+b;
     y = (a+b)*c*d; }
```

optimisation

```
if (secret)
   { x = a*b;
     y = x+c+d; }
else
   { x = a+b;
     y = x*c*d; }
```

padding insertion

$t_1 = K^{mult} + K^{var}$
and
$t_2 = K^{add} + K^{var}$

```
if (secret)
   { δ(t1);
     x = a*b;
     y = x+c+d; }
else
   { δ(t2);
     x = a+b;
     y = x*c*d; }
```

padding minimisation

$t = max(t_1, t_2)$

```
if (secret)
   { δ(t1-t);
     x = a*b;
     y = x+c+d; }
else
   { δ(t2-t);
     x = a+b;
     y = x*c*d; }
```

Constant-resource policy and compilation

Balancing all branches is not realistic

Use of an **atomic** annotation

- introduced by a previous static analysis
- allows for padding instructions

```
if (public) {
    { … }
else
    { … }
atomic {
    if (secret)
        { … }
    else
    { … }
}
…
```

Conclusion

Reducing security to safety for

- expressing two policies to protect a program against timing attacks
- proving using Coq that the policies are preserved through compilation

Follows previous uses of instrumented semantics

- CompCertSFI: a sandboxing transformation ensures that an untrusted module cannot escape its dedicated isolated address space

Perspectives

Cryptographic constant-time

Extend CompCert with support for vectorization instructions

Combine CT-CompCert with verified C cryptographic programs

- VST, HACL*

Constant-resource

Experimental validation

Allow memory accesses in the atomic parts

Towards a more realistic language with loops, functions, and instructions with variable resource consumption

Further reading

- G.Barthe, S.Blazy, R.Hutin, D.Pichardie. **Secure compilation of Constant-Resource Programs**. CSF 2021.
- G.Barthe, S.Blazy, B.Grégoire, R.Hutin, V.Laporte, D.Pichardie, A.Trieu. **Formal Verification of a Constant-Time Preserving C Compiler**. POPL 2020.
- S.Blazy, D.Pichardie, A.Trieu. **Verifying Constant-Time Implementations by Abstract Interpretation**. Journal of Computer Security, 27(1), 2019.
- F.Besson, S.Blazy, A.Dang, T.Jensen, P.Wilke. **Compiling Sandboxes: Formally Verified Software Fault Isolation**. ESOP 2019.
- G.Barthe, B. Grégoire, and V. Laporte. **Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic Constant-Time**. CSF 2018.

Questions ?