Provably secure compilation of side-channel countermeasures: the case of cryptographic “constant-time”

Gilles Barthe
Benjamin Grégoire
Vincent Laporte

CSF’18, 2018-07-12

Side channels

Running a program of physical devices leak information through side channels.

- Light
- Heat
- Sound
- Power
- Time
- …

- Memory cache
- Branch predictor
- …

Constant-time programming

Software-based countermeasure against **timing** attacks and **cache** attacks.

Guideline: control-flow and memory accesses should not depend on sensitive data.

Rationale: crypto implementations without this property are vulnerable.

Caveat: wide range of attacker models.

Secure compilation

- Can we reason about “constant-time” at the source level?
- Do compilers preserve “constant-time”-ness?

Counter-example A: emulation of conditional-move

Before

```
int cmove(int x, int y, bool b) {
  return x + (y − x) * b;
}
```

Counter-example A: emulation of conditional-move

Before

```
int cmove(int x, int y, bool b) {
  return x + (y − x) * b;
}
```

After

```
int cmove(int x, int y, bool b) {
  if (b) {
    return y;
  } else {
    return x;
  }
}
```

Counter-example B: double-word multiplication

Before

```
long long llmul(long long x, long long y) {
  return x * y;
}
```

$$x = \overline{ab} = aN + b \qquad y = \overline{cd} = cN + d \qquad xy = (ad + cb)N + bd \pmod{N^2}$$

Counter-example B: double-word multiplication

Before

```
long long llmul(long long x, long long y) {
  return x * y;
}
```

$$x = \overline{ab} = aN + b \qquad y = \overline{cd} = cN + d \qquad xy = (ad + cb)N + bd \pmod{N^2}$$

After

```
long long llmul(long long x, long long y) {
  long a = High(x);
  long c = High(y);
  if (a | c) {
    /* ... */
  } else {
    return Low(x) * Low(y);
  }
}
```

Counter-example Γ: tabulation

Before

```
char rot13(char x) {
    return 'a' + ((x - 'a' + 13) % 26);
}
```

Counter-example Γ: tabulation

Before

```
char rot13(char x) {
  return 'a' + ((x - 'a' + 13) % 26);
}
```

After

```
char rot13(char x) {
  static char table[26] = "nopqrstuvwxyzabcdefghijklm";
  return table[x - 'a'];
}
```

Counter-example Δ: speculative load introduction

Before

```
if (false) {
  let x = *ptr;
  ... x ...
}
```

Counter-example Δ: speculative load introduction

Before

```
if (false) {
  let x = *ptr;
  ... x ...
}
```

After

```
let x = *ptr;
if (false) {
  ... x ...
}
```

Good news…

Some compilers do preserve “constant-time”-ness.

Let’s prove it (very formally)!

Case studies:

- Constant folding
- Constant propagation
- Variable spilling
- Expression flattening
- Loop peeling
- Pull common instructions out of branches
- Swap independent instructions
- Linearization

A non-interference property

Decorate the small-step relation with a *leakage*: $a \xrightarrow{\ell} b$

A non-interference property

Decorate the small-step relation with a *leakage*: $\mathrm{a} \xrightarrow{\ell} \mathrm{b}$

Definition (Constant-time)

For every two execution prefixes

$$i \xrightarrow{\ell_0} s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} s_2 \cdots$$

$$i' \xrightarrow{\ell'_0} s'_0 \xrightarrow{\ell'_1} s'_1 \xrightarrow{\ell'_2} s'_2 \cdots$$

the leakages agree whenever the inputs agree:

$$\varphi(i, i') \implies \ell_0 \cdot \ell_1 \cdot \ell_2 = \ell'_0 \cdot \ell'_1 \cdot \ell'_2$$

Leakage?

Any combination of:

- tick per step
- branching conditions
- dereferenced addresses
- arguments of arithmetic operators (division, shift, etc.)
- content of freed memory
- …

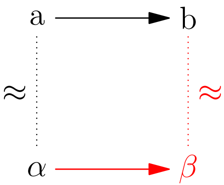
Compiler correctness & simulation diagrams

Given a relation $\approx$ between source and target execution states,

if initial states (for the same input values) are in relation

if related final states yield the same result

If the following diagram holds



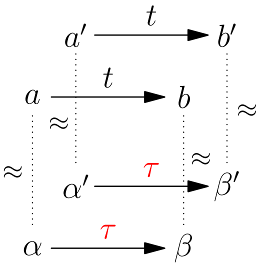
then the compiler is correct

(moreover, the $\approx$ relation is a relational invariant of any two related executions).

Lockstep 2-simulation

- Each target step is related by the simulation proof to a source step.
- Use this relation to justify that the target leakage is benign.
- Take two instances of the simulation diagram with equal source leakage;

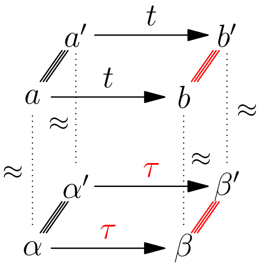
  and prove that target leakages are equal:



Lockstep 2-simulation

- Each target step is related by the simulation proof to a source step.
- Use this relation to justify that the target leakage is benign.
- Take two instances of the simulation diagram with equal source leakage;

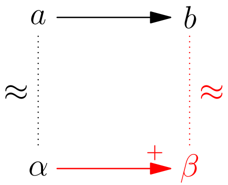
  and prove that target leakages are equal:



Use relations $\equiv$ between states to link the two executions.

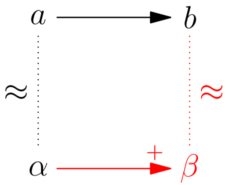
Many-steps simulation

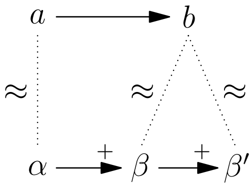
- Some compilation passes require a more general simulation diagram



Many-steps simulation

- Some compilation passes require a more general simulation diagram

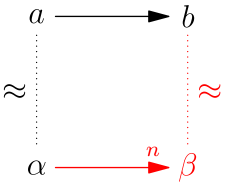


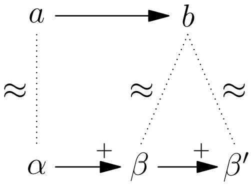


- **Issue**: how to (universally) quantify over instances of this diagram?
- Complying with hypotheses and conclusions is not enough

Many-steps simulation

- Some compilation passes require a more general simulation diagram



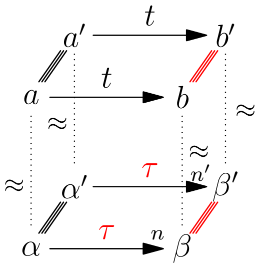


- **Issue**: how to (universally) quantify over instances of this diagram?
- Complying with hypotheses and conclusions is not enough
- Explicitly state the number of target steps: use a function "$n = \text{num-steps}(a, \alpha)$"

  and prove the simulation diagram for this number of steps

Many-steps 2-simulation

- The 2-diagram then generalizes to many-steps:



- **NB**: also works for $n, n' = 0$ (the *size* of the source state needs to strictly decrease)

Example: constant-propagation

1. Analysis: what variables have a statically known value
2. Simplify expressions, as in constant folding, using the analysis result
3. Remove (some) trivial branches (depending on heuristics), e.g.:
    - if 1 c_1 $c_2 \longrightarrow c_1$
    - loop c_1 0 $c_2 \longrightarrow c_1$

Correctness:

- Need to remember the analysis results (e.g., with annotations in the program)

Constant-time preservation:

- Need to remember which branches are simplified (with similar annotations)

Take-away

- A general theorem to reduce constant-time preservation to one diagram.
- Builds atop correctness proofs.
- Constant-time preservation is usually (much) simpler to prove.
- Can be instantiated to several leakage/adversary models.
- Many transformations are actually secure.
- Direct proof vs. translation validation is irrelevant

  (we prove that all correct runs of the transformation are secure).