Provably secure compilation of side-channel countermeasures

Gilles Barthe
Benjamin Grégoire
Vincent Laporte

2018-02-07

Introduction

Side-channels

"Computers are made of stuff"

Running a program:

- requires power, time, etc.
- produces heat, light, sound, etc.
- leaves traces (memory cache, branch predictor, etc.)

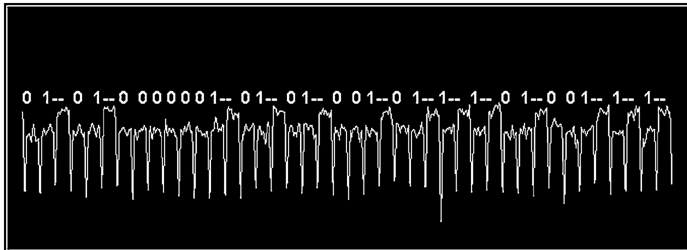
All these "side-channels" carry information about what is going on *inside* the machine.

Side-channel attacks

Power Analysis

Example: modular exponentiation ($r = x^k \pmod{p}$), as found in RSA

```
r = 1;
for(i = base - 1; 0 <= i; --i) {
    r = r² mod p;
    if ((k >> i) & 1) r = (r * x) mod p;
}
```



(Kocher et al., “Introduction to differential power analysis”, 2011)

Side-channel attacks

Cache attacks

Cache memory is shared among concurrent processes.

Many attacks, e.g.,:

- Percival, 2005, against RSA in OpenSSL
- Osvik, Shamir, Tromer, 2006, against AES
- Gras, Rasavi, et al., 2017, against ASLR
- Kocher et al., 2018, against OS-level isolation
- …

Constant-time programming

Software-based defense against side-channel attacks:

- control-flow (loop, if conditions)
- memory accesses (array indices)

should not depend on secret (sensitive) values.

Problem

- Do compilers preserve or break the constant-time property?
- Can a secure program be written in a high-level language?

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;
}
```

Counter-example B: double-word multiplication

Before

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

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'];
}
```

Contributions

- Which compiler passes do preserve constant-time?
- How to convince you that a compiler preserves constant-time?

Theorem (Constant-time preserving compiler)

compile(p) = p' $\longrightarrow$ constant-time(p) $\longrightarrow$ constant-time(p').

- Machine-checked proofs using Coq
- Tractable proofs
- A generic framework
- Illustrative instantiations on example languages and compilation passes

Compiler correctness à la Coq

Language semantics

- A type for programs
- A type for execution states
- An initial state for every program and every valuation of the input parameters
- Result extraction from final states
- A small-step execution (deterministic) relation between states $(\cdot \longrightarrow \cdot)$

Program behavior: set of input·result pairs related by $\longrightarrow^*$:

$$\text{input} \longrightarrow s_0 \longrightarrow s_1 \longrightarrow \ldots \longrightarrow s \longrightarrow \text{result}$$

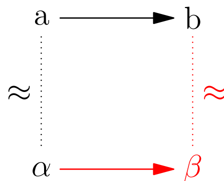
Proof technique: simulation diagrams

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).

Example: While language with arrays

Syntax:

- $e ::= n \mid x \mid a[e] \mid e \; o \; e$
- $c ::= x = e \mid a[e] = e \mid \text{if } e \; \vec{c} \; \vec{c} \mid \text{loop } \vec{c} \; e \; \vec{c}$

Semantics:

- State: $\{ \vec{c}, \rho \}$
- Evaluation of expressions in an environment: $[e]_\rho$
- Execution step, depending on the first instruction:
  - $\{ x = e; \vec{c}, \rho \} \longrightarrow \{ \vec{c}, \rho[x \leftarrow [e]_\rho] \}$
  - $\{ \text{if } e \; \vec{c}_1 \; \vec{c}_2; \vec{c}, \rho \} \longrightarrow \{ \vec{c}_i; \vec{c}, \rho \}$ where i is 1 if $[e]_\rho$ is true, 2 otherwise
  - $\{ \text{loop } \vec{c}_1 \; e \; \vec{c}_2; \vec{c}, \rho \} \longrightarrow \{ \vec{c}_1; \text{if } e \; (\vec{c}_2; \text{loop } \vec{c}_1 \; e \; \vec{c}_2) \; \epsilon; \vec{c}, \rho \}$

Example: constant-folding

- Replace constant sub-expressions by their values, e.g.:
  - $1 + 2 \longrightarrow 3$
  - $0 \times e \longrightarrow 0$
- Simulation relation: $\{ \vec{c}_1, \rho_1 \} \approx \{ \vec{c}_2, \rho_2 \}$ when:
  - $\vec{c}_2$ is the compilation of $\vec{c}_1$
  - $\rho_2 = \rho_1$

Constant-time, formally

Instrumented semantics

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

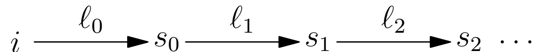
The leakage includes:

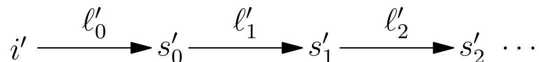
- Program counter (number of steps, direction of branches)
- Memory addresses, array offsets
- … anything you want, to model various adversaries

A relational property

Definition (Constant-time)

For every two execution prefixes





the leakages agree whenever the inputs agree:

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

Example: leakage of the while language

Evaluation of expressions:

- access to array cells
- values of dividend (second arguments of divisions)

Execution of instructions:

- leakage of the evaluated expressions
- written array cells
- boolean values of branching conditions

Remark

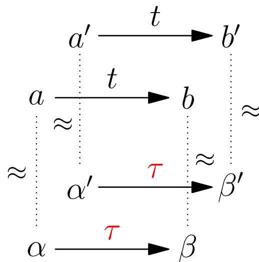
Leakage preservation entails constant-time preservation.

Corollary: focus on compilation passes which *do not* preserve leakage.

CT-simulations

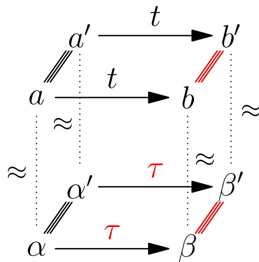
Lockstep CT-simulation

- 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 CT-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:



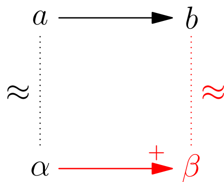
Use relations $\equiv$ between states to link the two executions.

Example: constant folding

- May remove leakage: “$0 \times t[i]$” $\longrightarrow$ “0”
- Synchronized executions ($\equiv$): the command is the same in both states.
- Equality of source leakages implies:
  1. equality of target leakages;
  2. both source executions stay synchronized;
  3. both target executions stay synchronized.

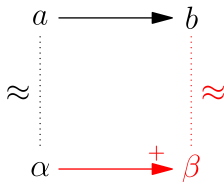
Many-steps simulation

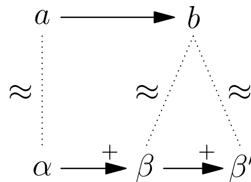
- Some compilation passes require a more general simulation diagram



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

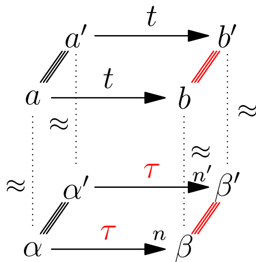
- Some compilation passes require a more general simulation diagram

- **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 CT-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)

More examples

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

More examples

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

Good news: constant-time is preserved

Conclusion

Summary

- A main theorem to easily build constant-time preservation proofs on top of semantics preservation proofs
- A handful of illustrative examples
- All proved using the Coq proof assistant
- Constant-time preservation proofs are generally *easier* that correctness proofs

Summary

- A main theorem to easily build constant-time preservation proofs on top of semantics preservation proofs
- A handful of illustrative examples
- All proved using the Coq proof assistant
- Constant-time preservation proofs are generally *easier* that correctness proofs

Thanks!