Constant-Time Programming: Formal Verification & Secure Compilation

Vincent Laporte

2019-09-25 — VeriSiCC Seminar

Side channels

What this talk is not about

Physical side-channels

- Power consumption
- Electromagnetic emissions
- Acoustic emissions

Micro-architecture issues

- Spectre
- Meltdown

Side-channel attacks (examples)

Lucky Thirteen: man-in-the-middle attack against TLS (2013)

> *The attacks involve detecting small differences in the time at which TLS error messages appear on the network in response to attacker-generated ciphertexts.*

Side-channel attacks (examples)

Lucky Thirteen: man-in-the-middle attack against TLS (2013)

> *The attacks involve detecting small differences in the time at which TLS error messages appear on the network in response to attacker-generated ciphertexts.*

Cache attacks against AES (2009)

> *The attacks allow an unprivileged process to attack other processes running in parallel on the same processor, despite partitioning methods such as memory protection, sandboxing, and virtualization.*

More generally

Any shared component:

- branch predictor
- stack memory
- event loop in a browser (shared between tabs)
- ...

may create a communication channel.

How to prevent sensitive data to leak through these channels?

A counter-measure: constant-time programming

Thou shalt not branch on secret data

- Bad example (musl-libc): compare sensitive buffers of public size

```
int memcmp(char* a, char* b, int size) {
 for (; size && *a == *b; size--, a++, b++);
 return size ? *a-*b : 0;
}
```

- Good example (openVNC)

```
 int ret = 0;
 for (int i = 0; i < size; i++) {
    ret |= *a++ ^ *b++;
 }
 return ret;
```

A counter-measure: constant-time programming (2)

Thou shalt not use secret data to compute memory addresses

- Bad implementations of AES use pre-computed in-memory S-boxes

```
static const uint32_t Ssm0[] = {
   0xC66363A5, 0xF87C7C84, ...
...
s0 ^= skey[0];
...
v0 = Ssm0[s0 >> 24] ^ ...
```

- Good implementations of AES use
  - a different algorithm, e.g., bitslicing; or
  - dedicated hardware, e.g., AES-NI.

Make your own constant-time policy

Depending on what the adversary can observe, forbid some operations on sensitive data.

- Avoid floating-point operations on secrets
- Don't keep secret data in freed memory
- Don't use SIMD registers with a lazy FPU
- ...

Pros and cons of "constant-time"

Pros:

- Effective counter-measure
- Weaker counter-measures are not as effective (e.g., CacheBleed)
- Can be implemented at the source level
- Praised by the NIST[1]: *"optimized implementations that address side-channel attacks (e.g., constant-time implementations) are more meaningful than those which do not"*

Cons:

- May be tricky to implement correctly
- Might be broken by program transformations (e.g., compilation)

[1] Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process

Outline

Shared components create **side channels** and **leak information**

Constant-time programming ensures that the **observable** use of these shared components cannot be influenced by **sensitive data**.

1. How to formally prove that a program is constant-time?
2. Do compilers preserve the *constant-time* property?

Is my program *constant-time*?

FaCT: all well-typed programs are constant-time [PLDI'19, Cauligi *et alii*]

- A C-like DSL, where variable declarations are annotated as public or secret
- The compiler will produce constant-time LLVM IR (or reject the program).

Automatic verification by static analysis

Taint analysis:

- Track which data is sensitive
- Check that each branching condition, accessed memory address, etc is public

Remarks

- There are no implicit flows in constant-time programs
- A precise points-to analysis is required to locate secrets in memory

Formal definitions of constant-time

Small steps operational semantics with *leakage*: $a \xrightarrow{\ell} b$

Formal definitions of constant-time

Small steps operational semantics with *leakage*: $a \xrightarrow{\ell} b$

A hierarchy of leakage models, to specify a wide range of adversaries:

0. No leak: no adversary
1. Execution time of each instruction
2. Current program point, targets of conditional jumps
3. Memory addresses
4. Arguments to (some) arithmetic operators
5. Contents of freed memory
6. …

Security as a non-interference property

Definition (No sensitive leak through side channels)

For any pair of prefixes of executions

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

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

if initial states are equivalent
(same public inputs, i.e., only differ on sensitive data),
then leakage traces are equal:

$$i \equiv i' \implies \ell_0 \cdot \ell_1 \cdot \ell_2 = \ell'_0 \cdot \ell'_1 \cdot \ell'_2$$

Non-interference proof in relational Hoare logic

- Judgments: $\{P\}\ c_1 \sim c_2\ \{Q\}$

Non-interference proof in relational Hoare logic

- Judgments: $\{P\}\ c_1 \sim c_2\ \{Q\}$
- Interpretation:

$$\forall\ m_1\ m_2\ m_1'\ m_2' \cdot \begin{cases} P\ m_1\ m_2 \\ m_1 \Downarrow^{c_1}\ m_1' \\ m_2 \Downarrow^{c_2}\ m_2' \end{cases} \implies Q\ m_1'\ m_2'$$

Non-interference proof in relational Hoare logic

- Judgments: $\{P\}\ c_1 \sim c_2\ \{Q\}$
- Interpretation:

$$\forall\ m_1\ m_2\ m_1'\ m_2' \cdot \begin{cases} P\ m_1\ m_2 \\ m_1 \Downarrow^{c_1}\ m_1' \\ m_2 \Downarrow^{c_2}\ m_2' \end{cases} \implies Q\ m_1'\ m_2'$$

- Functional equivalence (c_1 specification for c_2):

$$\{inputs\langle 1\rangle = inputs\langle 2\rangle\}\ c_1 \sim c_2\ \{result\langle 1\rangle = result\langle 2\rangle\}$$

Non-interference proof in relational Hoare logic

- Judgments: $\{P\}\ c_1 \sim c_2\ \{Q\}$
- Interpretation:

$$\forall\ m_1\ m_2\ m_1'\ m_2' \cdot \begin{cases} P\ m_1\ m_2 \\ m_1 \Downarrow^{c_1} m_1' \\ m_2 \Downarrow^{c_2} m_2' \end{cases} \implies Q\ m_1'\ m_2'$$

- Functional equivalence (c_1 specification for c_2):

$$\{\mathit{inputs}\langle 1\rangle = \mathit{inputs}\langle 2\rangle\}\ c_1 \sim c_2\ \{\mathit{result}\langle 1\rangle = \mathit{result}\langle 2\rangle\}$$

- Non-interference:

$$\{\text{public-inputs}\langle 1\rangle = \text{public-inputs}\langle 2\rangle\}\ c \sim c\ \{\mathit{leaks}\langle 1\rangle = \mathit{leaks}\langle 2\rangle\}$$

Issue: how to reason about the leakage?

Make leakage explicit in the program

- A global (ghost) variable “leaks” models the leakage
- Instrument each critical operation to explicitly update this variable:

```
if b then p else q   ⟶   leaks = b :: leaks; if b then p' else q'

x = t[i]             ⟶   leaks = i :: leaks; x = t[i]
```

Example: constant-time verification of Jasmin programs [S&P 2019]

Jasmin source

```
fn store2(reg u64 p, reg u64[2] x) {
    [p + 0] = x[0];
    [p + 8] = x[1];
}
```

Proof obligation:

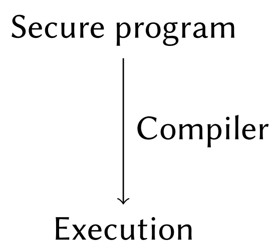
$p_1 = p_2 \implies$
$[0; p_1; 1; p_1 + 8] = [0; p_2; 1; p_2 + 8]$

EasyCrypt model with explicit leakage

```
proc store2 (p:u64, x:u64 array2) : unit = {
  var aux: u64;
  leaks ← LeakAddr [0] :: leaks;
  aux ← x.[0];
  leaks ← LeakAddr [to_uint (p + 0)] :: leaks;
  Glob.mem ← storeW64 Glob.mem (to_uint (p + 0)) aux;
  leaks ← LeakAddr [1] :: leaks;
  aux ← x.[1];
  leaks ← LeakAddr [to_uint (p + 8)] :: leaks;
  Glob.mem ← storeW64 Glob.mem (to_uint (p + 8)) aux;
}
```

Secure compilation

Secure compilation



- Given secure (source) code, do we get secure execution?
- Is the attacker model relevant at source level?

Compilers may introduce branches (examples)

- Implementation of uint32⟶float32 when micro-architecture only has int32⟶float32

Source

```
float floatofintu(unsigned int x) {
  return x;
}
```

CompCert 3.4

Target

```
float floatofintu(unsigned int x) {
  if (x < 0x80000000) // 2³¹
    return floatofints(x);
  else
    return 0x1p31 + floatofints(x − 0x80000000);
}
```

Compilers may introduce branches (examples)

- Implementation of uint32⟶float32 when micro-architecture only has int32⟶float32

Source

```
float floatofintu(unsigned int x) {
   return x;
}
```

CompCert 3.4

Target

```
float floatofintu(unsigned int x) {
   if (x < 0x80000000) // 2³¹
      return floatofints(x);
   else
      return 0x1p31 + floatofints(x − 0x80000000);
}
```

- "Optimization" of branchless selection on a micro-architecture without conditional move

Source

```
unsigned select(unsigned x, unsigned y, bool b) {
   return x + (y − x) × b;
}
```

clang -O1 -m32 -march=pentium

Target

```
unsigned select(unsigned x, unsigned y, bool b) {
   if (b = 0)
      return x;
   else
      return y;
}
```

Compilers may introduce memory accesses (examples)

- Register spilling

This should not be an issue: these new *memory accesses do not depend on secret data…*

Compilers may introduce memory accesses (examples)

- Register spilling

This should not be an issue: these new *memory accesses do not depend on secret data…*

- Loop hoisting

Source

```
while (condition) {
  x = *ptr;
  …
}
```

What if the condition never holds?

Target

```
x = *ptr;
while (condition) {
  …
}
```

Compilers may be proved

- Compilation may transform the leakage trace
- Compilation may nonetheless preserve “constant-time” security

Compilers may be proved

- Compilation may transform the leakage trace
- Compilation may nonetheless preserve “constant-time” security

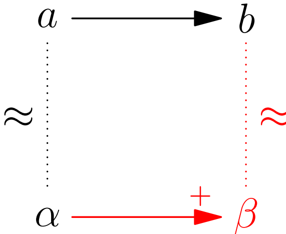
Rest of this talk:

- Formal proof that the CompCert compiler — slightly modified — preserves “constant-time” security from Clight to x86 assembly.

Correctness proof by simulation

The correctness of a compilation pass can be proved by:

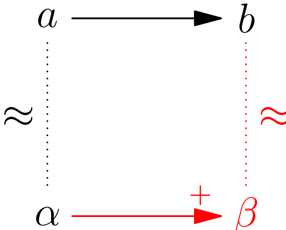
1. finding an invariant linking source states and target states (≈);
2. proving a diagram like the following:



Correctness proof by simulation

The correctness of a compilation pass can be proved by:

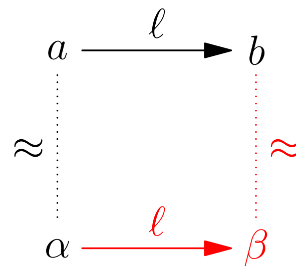
1. finding an invariant linking source states and target states (≈);
2. proving a diagram like the following:



Can we turn the simulation argument into a proof of security preservation?

Easy case: lock-step leakage preservation

Same simulation diagram, applied to the semantics with leakage:



Theorem: a compilation pass that satisfies this enhanced simulation diagram:

- is correct;
- preserves constant-time security.

The proof scripts from the original correctness proof can be reused.

Same-point relations

Compilation passes *transform* the leakage: insert, remove, …
This is secure provided the transformation cannot be influenced by sensitive data

For each programming language, we define the *control-flow state*: a view of the execution state that cannot depend on sensitive data (in constant-time programs):

- program-point
- stack pointer
- stack of return addresses and saved stack pointers

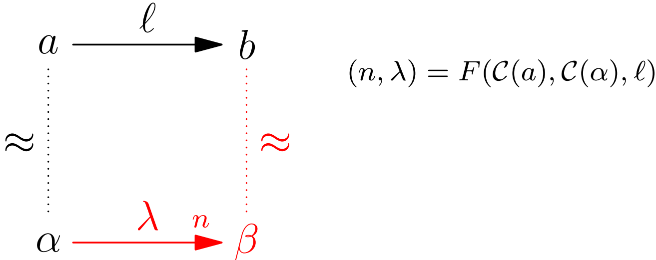
Two execution states a and a' with equal control-flow are in the *same-point relation*, written $a \equiv a'$.

All these relations satisfy well-formedness conditions:

- initial states are related; final states are only related to final states;
- $a \equiv a' \implies a \xrightarrow{\ell} b \implies a' \xrightarrow{\ell} b' \implies b \equiv b'$
- $a \equiv a' \implies a \xrightarrow{\ell} b \implies a' \xrightarrow{\ell'} b' \implies (\ell = \varepsilon \iff \ell' = \varepsilon)$

Static trace transformation

For a program transformation, if there is a *static leakage transformation function F* such that the following diagram holds where $\mathcal{C}$ is the control-flow view of the state:

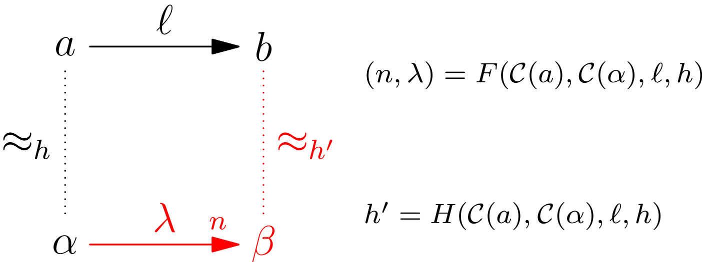


then this transformation preserves constant-time.

Special case leakage erasure: λ is either ℓ or ε.

More general proof schemes

- Dynamic leakage transformation: we need to remember some history of the simulation proof
  - and to prove that this history is not tainted by sensitive information



- Constant-time simulation [CSF 2018]: two instances of the simulation diagram
  - last resort proof technique
  - convenient mostly to prove the soundness of the other techniques

CT-CompCert preserves “constant-time” security

Pass	Description	Modif.	Proof
Cshmgen	Type elaboration, simplification of control		Preservation
Cminorgen	Stack allocation		Dynamic
Selection	Recognition of operators and addr. modes	Cmove	Erasure
RTLgen	Generation of CFG and 3-address code	No switch	Preservation
Tailcall	Tailcall recognition		Preservation
Inlining	Function inlining		Dynamic
Renumber	Renumbering CFG nodes		Preservation
ConstProp	Constant propagation		Dynamic
CSE	Common subexpression elimination		Erasure
Deadcode	Redundancy elimination		Erasure
UnusedGlob	Elimination of unreferenced static defs.	Disabled	
Allocation	Register allocation		Erasure
Tunneling	Branch tunneling		Erasure
Linearize	Linearization of CFG		CT-simulation
CleanLabels	Removal of unreferenced labels		Preservation
Debugvar	Synthesis of debugging information		Preservation
Stacking	Laying out stack frames		Dynamic
Asmgen	Emission of assembly code	Ghost	Static

Conclusions

- Many compilation passes preserve “constant-time” security
- Because they transform leakage traces in simple ways
- This can be proved reusing correctness proofs
- This enables to reason about security and implement counter-measures at the source level

Conclusions

- Many compilation passes preserve “constant-time” security
- Because they transform leakage traces in simple ways
- This can be proved reusing correctness proofs
- This enables to reason about security and implement counter-measures at the source level

Thanks