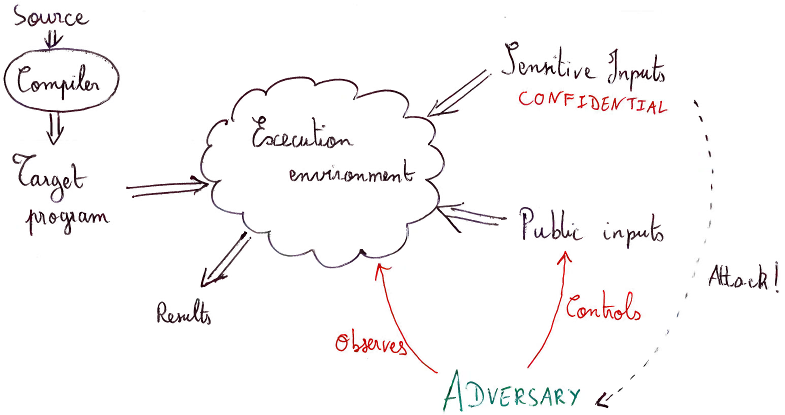
Secure Compilation of Counter-Measures to Spectre Attacks

Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter, Benjamin Grégoire, **Vincent Laporte**

2024-10-07 — Prosecco Seminar, Paris (XIII^{e})

Secure Compilation



Security of Software Implementations

- Small-step deterministic semantics with leakage: $s \xrightarrow{o} s'$.
  - Example: $\mathrm{Obs} ::= \bullet \mid \mathrm{branch}\ b \mid \mathrm{addr}\ a\ i$.
- Programs take inputs to initial states
  - Same language of inputs for all programming languages
  - Execution states and leakage may differ
  - Some states are *final*
- Relation φ over inputs characterizes their confidentiality
- Semantic security of program P w.r.t. φ:

$$\forall i_1\ i_2\ n\ O_1\ O_2\ s_1\ s_2,$$

$$i_1\ \varphi\ i_2 \implies P(i_1) \xrightarrow{O_1}{}^{n} s_1 \implies P(i_2) \xrightarrow{O_2}{}^{n} s_2 \implies O_1 = O_2.$$

The (non) issue of safety

Is the following program constant-time?

```
u64[1] t = { 42 };
export fn get(#secret reg u64 x) ⟶ #public reg u64 {
    reg u64 r;
    r = t[x];
    return r;
}
```

All possible executions:

1. $P(x) \xrightarrow{\epsilon}{}^{0} P(x)$
2. $P(0) \xrightarrow{\text{addr } t \ 0} \ldots$

Yes: function `get` is CT.

Solution: include all safety preconditions in φ.

Compilers vs. Constant-Time

```
param int N = 4;
export fn copy(reg ptr u64[N] dst src)
⟶ reg ptr u64[N] {
  reg u64 v i = 0;
  while (i <u N) {
    v = src[i];
    dst[i] = v;
    i += 1;
  }
  return dst;
}
```

```
copy:
    mov rax, 0
    jmp head
loop:
    mov rcx, [rsi + rax * 8]
    mov [rdi + rax * 8], rcx
    inc rax
head:
    cmp rax, 4
    jb loop
    ret
```

The compilers knows how to transform code; it also knows how to transform leakage.

CT-Simulations

The compilation of program P into program Q preserves CT if there is a function F from source leakage to target leakage such that:

$$\forall i\ n\ O\ s, P(i) \xrightarrow{O}^* s \implies \exists t, Q(i) \xrightarrow{F(O)}^* t \land (\text{final}(s) \iff \text{final}(t)).$$

- This whole-trace property can be proved by means of usual simulation diagrams.
  - Various examples by Barthe et al. (2021) and Barthe et al. (2019).

Speculative Execution: branch prediction and Spectre v1

- Do not wait
  - the end of an instruction before starting to execute the next one
- Speculate
  - what is the next instruction to execute

Example

```
mov   rax, 0
cmp   rdi, 2
jnb   ...
mov   rax, [rcx + rdi * 8]
mov   rax, [rdx + rax * 8]
⋮
```

Speculative Execution: branch prediction and Spectre v1

- Do not wait
  - the end of an instruction before starting to execute the next one
- Speculate
  - what is the next instruction to execute

Example

```
mov    rax, 0
cmp    rdi, 2
jnb    ...
mov    rax, [rcx + rdi * 8]
mov    rax, [rdx + rax * 8]
⋮
```

- Some transient effects may be observed
- Speculative bypass of safety checks may lead to security issues

Example vulnerability: iLeakage [CCS 2023]

We have actively discussed countermeasures with Safari's development team [...]. Our discussion has resulted in Apple refactoring Safari's multi-process architecture significantly.

Defenses: SLH & SCT

(Selective) Speculative Load Hardening

- Detect mis-speculated executions in software
- (Selectively) sanitize sensitive values before they leak

This is effective (see Ammanaghatta Shivakumar et al. (2023))

- Often cheap to implement
- Protection can be automated
- Security can be proved

Running example

```
param int N = 3;
export fn main(#secret reg u64 sec) {
    stack u64[N] spill p;
    spill[0] = sec;
    sec = spill[0];
    reg u64 i = 0;
    while (i <u N) {
        p[i] = 0;
        i += 1;
    }
    i = p[0];
    p[i] = 0;
}
```

Is this program secure?

Running example

```
param int N = 3;
export fn main(#secret reg u64 sec) {
    stack u64[N] spill p;
    spill[0] = sec;
    sec = spill[0];
    reg u64 i = 0;
    while (i <u N) {
        p[i] = 0;
        i += 1;
    }
    i = p[0];
    p[i] = 0;
}
```

Is this program secure?

- Speculative execution may bypass the initialization loop.
- Compiler may allocate spill and p at the same address.
- So the leaked value of i at the final line might be secret.

Speculative Semantics

- The adversary has full control over the speculation through **directives**:

$$\text{Dir} ::= \text{step} \mid \text{force } b \mid \text{mem } a\ i.$$

- No-backtrack theorem: no need for backtracking to reason about Spectre.
- Small steps relate states with an explicit mispeculation bit:

$$\frac{\textsc{Assign}}{\langle x = e;\ c,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\text{step}]{\bullet} \langle c,\ \rho[x \leftarrow [\![e]\!]_\rho],\ \mu,\ ms\rangle}$$

Speculative Semantics (selected rules)

Cond

$$\frac{[\![\, e \,]\!]_\rho = b'}{\langle \text{if } e \text{ then } c_\top \text{ else } c_\bot;\ c,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\text{force } b]{\text{branch } b'} \langle c_b;\ c,\ \rho,\ \mu,\ ms \vee (b \neq b')\rangle}$$

N-Load

$$\frac{[\![\, e \,]\!]_\rho = i \qquad i \in [0, |a|) \qquad \mu(a, i) = v}{\langle x = a[e];\ c,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\text{mem } b\ j]{\text{addr } a\ i} \langle c,\ \rho[x \leftarrow v],\ \mu,\ ms\rangle}$$

S-Load

$$\frac{[\![\, e \,]\!]_\rho = i \qquad i \notin [0, |a|) \vee \mu(a, i) = \bot \qquad j \in [0, |b|) \qquad \mu(b, j) = v}{\langle x = a[e];\ c,\ \rho,\ \mu,\ \top\rangle \xrightarrow[\text{mem } b\ j]{\text{addr } a\ i} \langle c,\ \rho[x \leftarrow v],\ \mu,\ \top\rangle}$$

Example

```
param int N = 3;
export fn main(#secret reg u64 sec) {
    stack u64[N] spill p;
    spill[0] = sec;
    sec = spill[0];
    reg u64 i = 0;
    while (i <u N) {
        p[i] = 0;
        i += 1;
    }
    i = p[0];
    p[i] = 0;
}
```

Directives

- mem *spill* 0
- mem *spill* 0
- step
- force $\bot$
- mem *spill* 0
- mem *spill* 0

Observations

- addr *spill* 0
- addr *spill* 0
- $\bullet$
- branch $\top$
- addr *p* 0
- addr *p* sec

Speculative Constant-Time

Given a relation φ on inputs, a program P is φ-SCT when:

$$\forall D\ i_1\ i_2\ O_1\ O_2\ s_1\ s_2,$$

$$i_1\ \varphi\ i_2 \implies P(i_1) \xrightarrow[D]{O_1}{}^* s_1 \implies P(i_2) \xrightarrow[D]{O_2}{}^* s_2 \implies O_1 = O_2.$$

Informally: no choice of directives can reveal any secret.

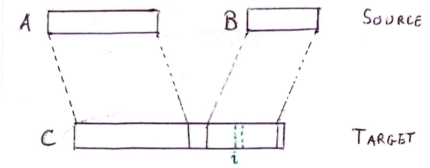
Nota bene:

- As usual, φ guarantees safety under *normal* executions
- Mis-speculated executions cannot be stuck due to a wrong choice of directive

Motivating Example: Array Concatenation

Let's merge two source arrays A and B into a single target array C.

- $A[e]$ becomes $C[e]$
- $B[e]$ becomes $C[e + |A|]$.



How to explain the value of the access $C[e]$ when e evaluates to i?

- when source accesses B?
- when it accesses A?

We need to extend the correctness proof to mis-speculated executions (of the target).

General Theorem

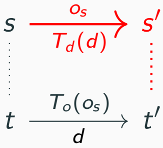
Secure compilation (backward case)

Let P and Q be programs. If there exist two functions T_d and T_o such that

$$\forall D\ i\ O_t\ t, Q(i) \xrightarrow[D]{O_t}{}^{*}\ t \implies \exists O_s\ s, P(i) \xrightarrow[T_d(D)]{O_s}{}^{*}\ s \land O_t = T_o(O_s)$$

then for any φ, if P is φ-SCT then Q is φ-SCT.

Lockstep simulation diagram



Proof Sketch for Array Concatenation

Transforming directives

$$T_d(\text{mem } C\ j) = \begin{cases} \text{mem } A\ j & \text{if } j < |A| \\ \text{mem } B\ (j - |A|) & \text{otherwise} \end{cases}$$

$$T_d(d) = d$$

Transforming observations

$$T_o(\text{addr } A\ i) = \text{addr } C\ i$$

$$T_o(\text{addr } B\ i) = \text{addr } C\ (i + |A|)$$

$$T_o(o) = o$$

Applications

The following passes have been proved to preserve security:

Lock-step, identity transformers

- Constant folding
- Constant propagation
- Loop peeling
- Register allocation (no spilling)

Introduce directives, eliminate leakage

- Dead assignment elimination
- Dead branch elimination

More complex transformers

- Array reuse
- Array concatenation
- Linearization

And their composition

Security preservation proofs can be composed.

A dilemma?

What is the right speculative semantics?

Previous work

- Adversaries cannot take advantage from uninitialized reads
- Fresh arrays can be seen as public
- More programs are SCT
- Array reuse is not secure

This work

- Adversaries can access any location on uninitialized reads
- Fresh arrays must be seen as speculatively secret (aka *transient*)
- Less programs are SCT
- Some compilers are secure

Experiments

Set-up

1. Amend the Jasmin SCT checker to treat uninitialized arrays as transient.
2. Typecheck a few programs deemed secure by the unmodified checker
    - test-suite of the SCT checker
    - the SCT implementations from LibJade

Results

- Only a couple of synthetic programs are now rejected.
- The 44 real cryptographic implementations are still (automatically) proved secure.

Questions ?

Thank you

Pre-print available online (Arranz Olmos et al. 2024).

See you soon. . .

https://formosa-crypto.org/

Ammanaghatta Shivakumar, Basavesh, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 2023. "Typing High-Speed Cryptography against Spectre v1." In *SP 2023- IEEE Symposium on Security and Privacy*, 1592–1609. San Francisco, United States: IEEE. https://doi.org/10.1109/SP46215.2023.10179418.

Arranz Olmos, Santiago, Gilles Barthe, Lionel Blatter, Benjamin Grégoire, and Vincent Laporte. 2024. "Preservation of Speculative Constant-time by Compilation." https://hal.univ-lorraine.fr/hal-04663857.

Barthe, Gilles, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2019. "Formal Verification of a Constant-Time Preserving C Compiler." *Proc. ACM Program. Lang.* 4 (POPL). https://doi.org/10.1145/3371075.

Barthe, Gilles, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2021. "Structured Leakage and Applications to Cryptographic Constant-Time and Cost." In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 462–76. CCS '21. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3460120.3484761.