

LANGUAGE BASED SECURITY (LBT)

SECURE COMPILATION

Chiara Bodei, Gian-Luigi Ferrari

Lecture May, 20 2024



Outline



Hardware-based approaches

Other approaches

Many possible alternative approaches to compiler security:

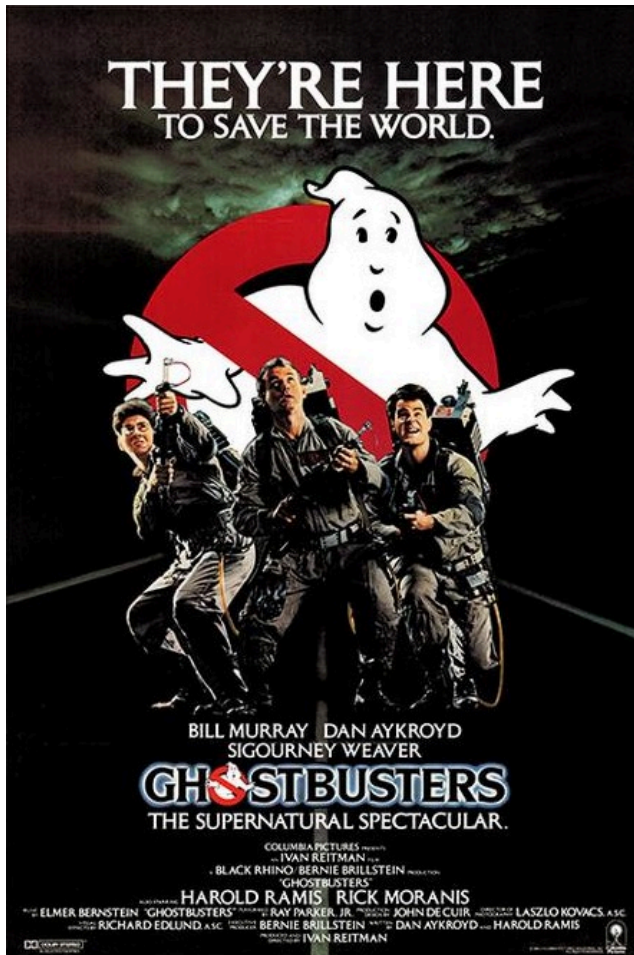
- Secure translation validation
- Hardware-based approaches:
 - Enclaves: Intel SGX, Sancus, ...
 - Micro-policies-based architectures

Becoming a Ghostbuster*

Chiara Bodei, Gian-Luigi Ferrari

We thank Matteo Busi for the possibility of using and modifying these nice slides

*actually, a Spectrebuster



Outline



Intro
Mechanisms, attacks and mitigations
More formally
Conclusions

Let's try to understand first

- Side-channel attacks
- Constant-time programs
- Speculative execution
- Speculative leaks
- Fixes and mitigations
- ...

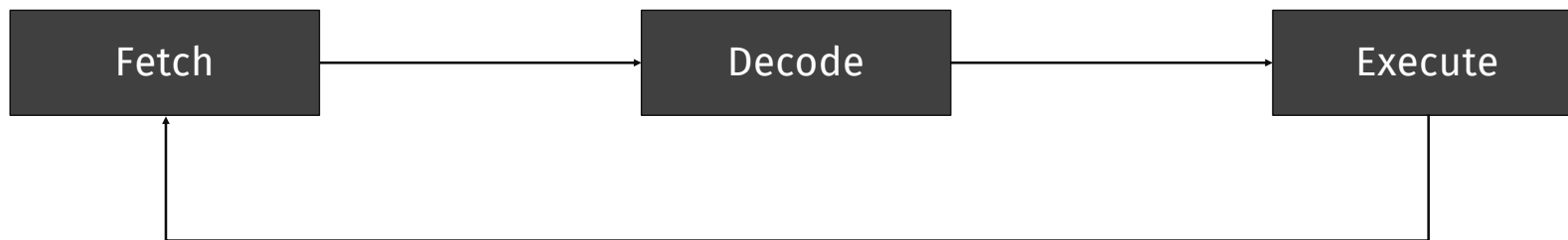
Outline



Intro
Mechanisms, attacks and mitigations
More formally
Conclusions

Computer Architecture

Do you remember how processors work?



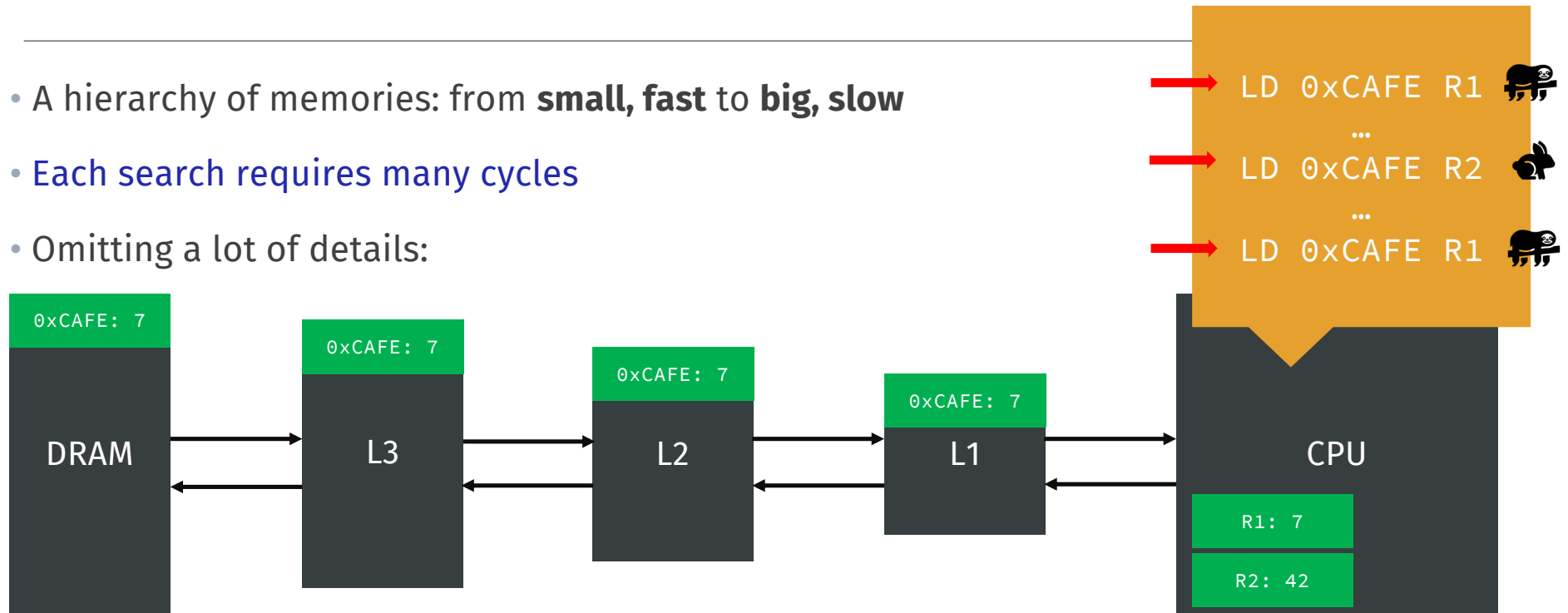
Problem: not very efficient (≥ 1 cycle per stage)

How to make processors faster?

- **First idea:** make cycles “shorter”, i.e., **increase** the processor’s clock
 - We can reach about 4 GHz (4 billion cycles per second)
 - **Very good**, but ~2004 we reached the peak: the clock speed has not increased
 - Also: power-consumption issues, too much heat, ...
- People started thinking about **alternatives!**
 - **Idea:** make the average case faster
 - **Solutions:** pipelines, [caches](#), [speculation](#), multi-core, multi-thread, ...

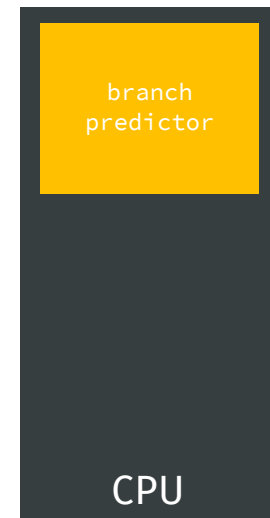
Caches

- A hierarchy of memories: from **small, fast** to **big, slow**
- Each search requires many cycles
- Omitting a lot of details:



Speculation

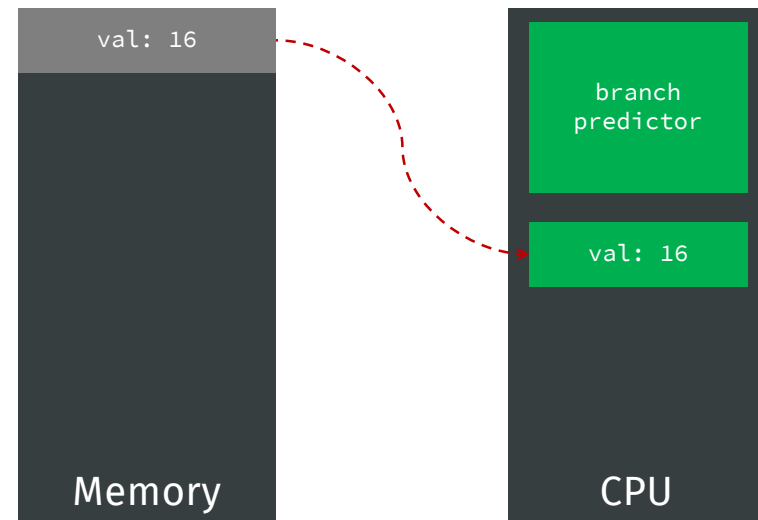
- Caches are **fast** when data is already there, **otherwise** we need to wait for the DRAM 🥲
- **Idea:**
 - Upon branch on a value in memory, the CPU speculates on what's **probably right** to do and does that
 - When the value arrives, if it was the **wrong thing: rollback**
- For that, the CPU is equipped with a **branch predictor**
 - **Roughly:** should the CPU take the next branch?
 - The CPU **trains** its branch predictor by **observing** what happened before
 - Modern branch predictors are complex, but for our purposes **just one bit!**



Speculation: when the CPU is **right**

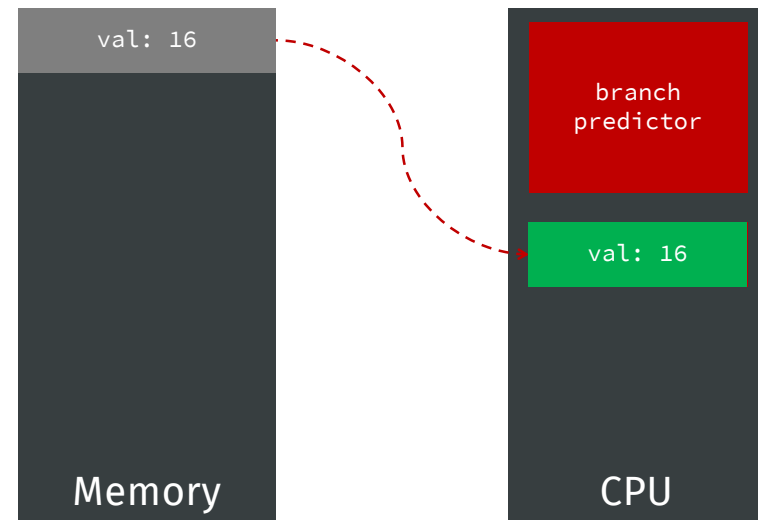
```
→ for (...)  
  → if val >= 0  
    → foo ();  
    else  
      bar ();
```

The CPU was **right**: commit the changes!



Speculation: when the CPU is **wrong**

```
→ for (...  
  → if val >= 0  
    → foo ();  
    else  
    → bar (); rollback
```



Side channels in this setting...

Designers are trying to optimize the average case!

- In the **worst** case the execution will be **slow**
- In the **best** case the execution will be **very fast**

Consider an attacker that just **observes** the execution of a program

- If it can learn some information from its observation, then there is a side channel!
- **Informally:** programs that resist against attackers able to measure time are said **constant-time programs**
 - **Actually:** wait till the end for a better (i.e., implementation-independent) definition

More precisely:

“A **side channel** is any observable side effect of computation that an attacker could measure and possibly influence.” [Lawson, 2009]*

*[Lawson, 2009] Lawson, Nate. "Side-channel attacks on cryptographic software." *IEEE Security & Privacy* 7, no. 6 (2009): 65-68.

More concretely:

- Assume an attacker can measure the execution time of a program
- Can it attack a program on a system with **caches**?
- **Yes!** 😎 Roughly because caches introduce *timing variations*, based on the memory accesses
- Find a line from the cache shared between the attacker and the victim
 - **Flush/Evict** it from the cache
 - e.g., using `clflush` on x86 or by loading attacker's data ending up in the same cache line
 - Let the victim run for a while
 - Measure the time it takes to perform a memory read at the address corresponding to the evicted cache line (**Reload**)
 - If the victim accessed the shared line, the access will be fast (data was cached!) 🐰
 - Otherwise, the read will be slow 🐢
- Attacks are also possible without sharing memory

Flash + Reload Attack

```
1 if (secret == 1) {  
2 x = x + 1;  
3 }  
4 else {  
  ...
```

Executed only
when secret is 1

Suppose there is an encryption algorithm **shared** between the attacker and the victim

- The attacker performs a **clflush(line 2)** and lets the victim execute
- The attacker try to reload the line:
 - In case of **cache miss**: the victim **did not** access line 2 🐼
 - In case of **cache hit**: the victim **did** access the line 2 🐼

Evict + Reload Attack

Evict+Reload forces contention on the cache set that stores the line, causing the processor to discard the contents of that cache line by loading something else instead of flushing it

The attacker **evicts**



Victim accesses/does not access

Attacker reloads

- **Fast access time** -> victim accessed 🐘
- **Slow access time** -> victim did not access 🐇

Prime + Probe Attack

No shared memory between the attacker and the victim

- The victim has access to a variable `a` and
- the attacker has access to a variable `c` s.t. `a` and `c` map to the same cache line
- The attacker evicts the address `&a` from the cache by accessing the address `&c` and lets the victim execute
- If the attacker try to access `&c`:
 - In case of cache miss: the victim **did** access `&a` 
 - In case of cache hit: the victim **did not** access `&a` 

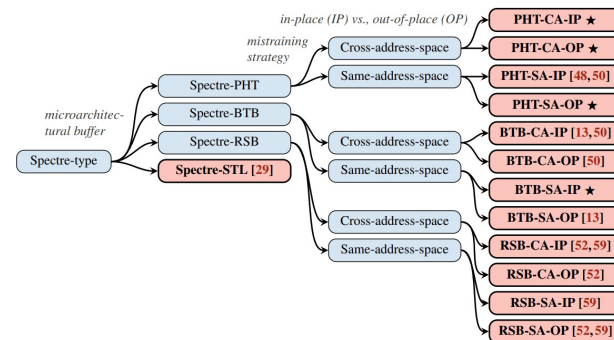
```
if (secret == 1) {  
    maccess(&a);  
}  
else {  
    ...  
}
```

We now are ready for



Spectre is an attack exploiting **speculative execution** to **leak secrets**

- It is part of a recent wave of **micro-architectural attacks**, i.e., attacks using side-channels induced by the micro-architecture of a processor (e.g., cache, timers, virtual memory, ...)
- “**Spectre** [...] transiently bypasses software-defined security policies (e.g., bounds checking, function call/return abstractions, memory stores) to leak secrets out of the program’s intended code/data paths.” [Canella et al., 2019] (← bit outdated, but a good survey on Spectre!)
- Lot of variants:



[Canella et al., 2019] Canella, Claudio, et al. "A systematic evaluation of transient execution attacks and defenses." *28th USENIX Security Symposium (USENIX Security 19)*. 2019. URL: <https://arxiv.org/abs/1811.05441v3>

Spectre v1/v1.1 (special-cases of PHT)

We focus on **Spectre v1 and v1.1**

- Also called **Spectre-PHT**: the attacker **mistrains** the branch predictor, by poisoning the **Pattern History Table**
- How? **For example, as follows**
 - The attacker:
 1. Looks for a piece of “vulnerable code” (**code gadget**) including a condition (next slide)
 2. Runs that code multiple times with an input such that the condition always hold (so training the branch predictor to take the “true” branch)
 3. Then, runs the code with a specially-crafted input making the condition false: now the CPU mis-speculates and executes the “true” branch with wrong data (!!!)
 4. Finally, it looks for information left behind after the CPU discovered it mis-speculated and rolled-back the computation (e.g., part of the contents of caches)

Not in
cache

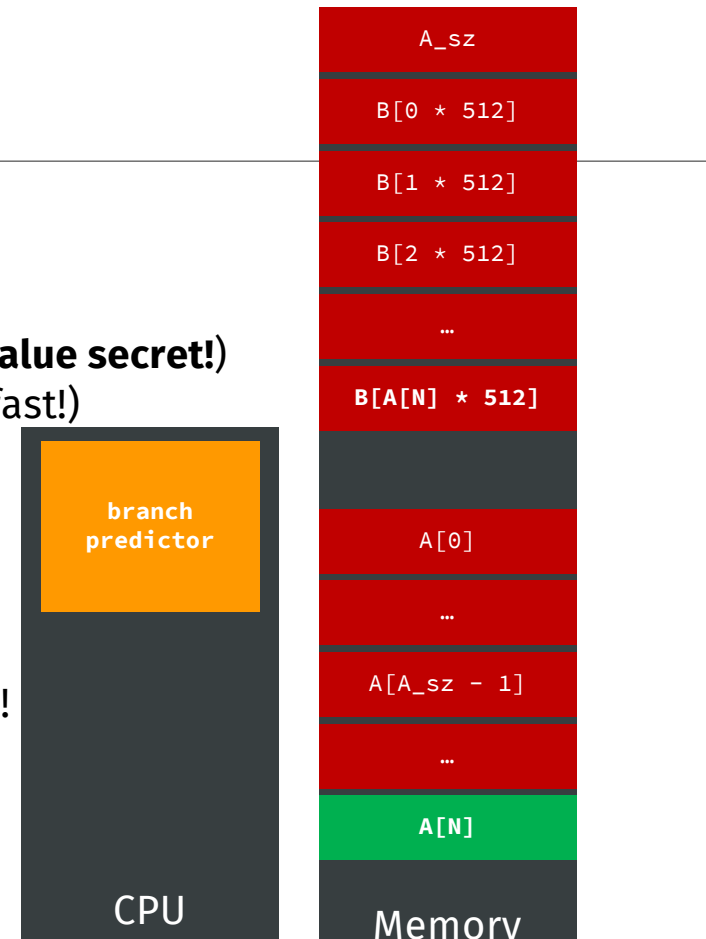
In cache

Spectre v1

1. Find a **code gadget** (i.e., a vulnerable piece of code) in victim
2. Mis-train the branch predictor
3. Suppose B and A_sz not in cache
4. Call the gadget with $x = N$ with $N > A_sz$ ($A[N]$ cached, **value secret!**)
5. CPU mis-speculates on the bound check and accesses $A[N]$ (fast!)
6. At that point, $B[A[N] * 512]$ is loaded in the cache
7. CPU discovers that the condition was false: **rollback!**
8. When control returns to caller (attacker) it just goes over $B[i * 512]$ and measures access time:
 - for $i \neq A[N]$ the access will be slow
 - for $i = A[N]$ the access will be fast, the secret is leaked!

```
if (x < A_sz)
    y = B[A[x] * 512];
```

The attacker can successfully read past the end of the array A

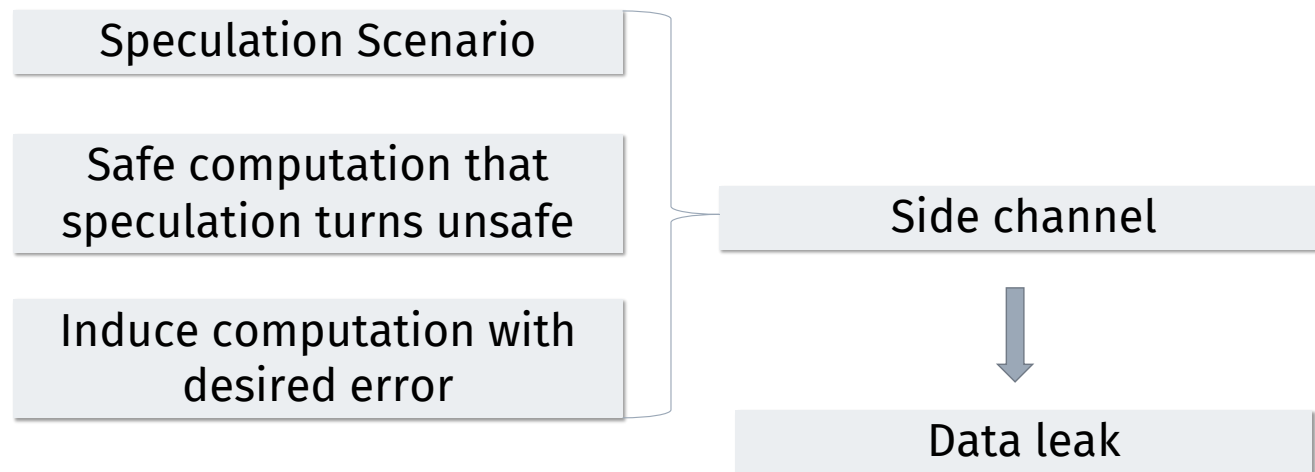


Spectre v1.1

1. Find a **code gadget** in victim
2. Mis-train the branch predictor
3. Let:
 - A, A_sz, B as above
 - A[N] contains a **secret**
 - y (supplied by the attacker) s.t. A[y] points on the return address on the stack
 - z (supplied by the attacker) to point to an instruction accessing B[A[N] * 512]
4. If function returns during mis-speculation, the CPU loads B[A[N] * 512] in cache
5. Despite rollback, when control returns to caller (attacker) it just goes over B[i * 512] and measures access time:
 - for $i \neq A[N]$ the access will be slow
 - for $i = A[N]$ the access will be fast, the secret is leaked!

```
if (y < A_sz)
    A[y] = z;
```

Spectre pattern



Mitigating Spectre

- Mitigations for Spectre are hard
- Also, the number of different variants does not help
- One simple idea is to **stop speculation** from accessing to data it should not:



Fences

Speculative Load Hardening (SLH)

Mitigating Spectre: fences

A **fence** or speculation barrier is a processor instruction that inhibits speculation

- On x86_64, we use `lfence` [[load fence](#)] on all branches to inhibit speculation;
- Despite the name, `lfence` serializes execution completely, and not just loads
- In pseudo-C:

```
if (x < A_sz) {  
    __asm__("lfence");  
    y = B[A[x] * 512];  
}  
else {  
    __asm__("lfence");  
    /* ... */  
}
```

Mitigating Spectre: fences

- **The Good:**

- It works (mostly :)
- Probably nothing else!

- **The Bad:**

- **Highly** inefficient (it stops speculation **completely**)
- Requires re-compilation

```
if (x < A_sz) {  
    __asm__("lfence");  
    y = B[A[x] * 512];  
}  
else {  
    __asm__("lfence");  
    /* ... */  
}
```

- To make this a bit better one could use [static analysis](#) to decide where fences should go
 - E.g., **Microsoft MSVC** does that by detecting known problematic patterns

Mitigating Spectre: SLH

- Speculative Load Hardening (SLH) is a bit more sophisticated than fences
- **Idea:**
 - It may be more efficient to introduce an **artificial data dependency** between the condition of a jump and the pointer
 - Simplifying, for Spectre v1 (assuming “ ? : ” to be implemented without branching, e.g., using a `cmov` in x86)

```
uint mask = ALL_ONES; /* all bits set to 1 */
if (cond) {
    mask = !cond ? ALL_ZEROES : mask;
    // ...
    y = B[(A[x] * 512) & mask];
}
else {
    mask = cond ? ALL_ZEROES : mask;
    /* ... */
}
```

Mitigating Spectre: SLH

- **The Good:**
 - It works (with some changes also for other Spectre variants)
 - More efficient than fences
 - No need of static analysis code (but still could help)
- **The Bad:**
 - Still slow (mask must be known when accessing to B, also when speculating!)
 - Must be carefully implemented
 - Still requires re-compilation

```
if (cond) {  
    mask = !cond ? ALL_ZEROES : mask;  
    // ...  
    y = B[(A[x] * 512) & mask];  
}  
else {  
    mask = cond ? ALL_ZEROES : mask;  
    /* ... */  
}
```

Fixing Spectre: other ideas

Other possible fixes:

- Make secrets non reachable:
 - Just like **Chrome** and **Firefox**: isolate secrets in different processes
- Reduce the bandwidth of side-channels:
 - e.g., via shadow micro-arch state that can be discarded, less accurate timers, adding noise, ...

Is Spectre is fixed?

Only if you are willing to re-compile/re-design your system and make it slow!

Outline



Intro
Mechanisms, attacks and mitigations
More formally
Conclusions

Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade: a short guide

[Vassena et al., 2021] M. Vassena, C. Disselkoen, K. von Gleissenthall, S. Cauligi, R. Gökhan Kıcı, R. Jhala, D. Tullsen, and D. Stefan. "Automatically eliminating speculative leaks from cryptographic code with blade." POPL 2021. URL: <https://arxiv.org/abs/2005.00294>

Modelling Spectre

- What we just saw is full of a lot of **details**
- Are they **really** needed to reason about Spectre? Nope
 - We can choose a relatively simple language. Why?
 1. Simplifies the reasoning
 2. Allows to reason directly at source level
 - We do not need too many details about the architecture
 - A Just In Time (JIT) semantics taking source programs, translating them on-the-fly to processor instructions and executing them
 - Does that in three stages (**fetch, exec, retire**)
 - Attacker chooses on what to fetch/exec/retire
 - A **static analysis** + a **consistency** and a **soundness theorem** guarantee the **correctness and security of the JIT compilation!**

The “source” language

Values $v ::= n \mid b \mid a$

Expr. $e ::= v \mid x \mid e + e \mid e < e$

$\mid e \otimes e \mid e ? e : e$

$\mid \text{length}(e) \mid \text{base}(e)$

Rhs. $r ::= e \mid *e \mid a[e]$

Cmd. $c ::= \text{skip} \mid x := r \mid *e = e$

$\mid a[e] := e \mid \text{fail} \mid c; c$

$\mid \text{if } e \text{ then } c \text{ else } c$

$\mid \text{while } e \text{ do } c$

$\mid x := \text{protect}(r)$

- This is **easy**, just the WHILE-language!
- Sure?
 - **Question:** how do you think **protect** works?
 - **Hint:** SLH prevent “wrong” accesses. Can you “abstract” SLH in a single command?
 - **Answer:** completes the assignment only when the evaluation of r is **stable**, i.e., no more rollbacks are possible, and r is the **final, non speculative value**

The “target” language

move

Instructions $i ::= \text{nop} \mid x := e \mid x := \text{load}(e)$
 $\mid \text{store}(e, e) \mid x := \text{protect}(e)$
 $\mid \text{guard}(e^b, cs, p) \mid \text{fail}(p)$

- Again, we have protect!
- **guard** (e^b, cs, p):
 - expresses a pending jump
 - e is the condition, b is its predicted outcome, cs is what to do if a rollback is needed (ignore p , an identifier useful for the security analysis)

The source language can be translated on-the-fly to a target one, using a JIT semantics that models speculation

The «Compiler»: A JIT semantics for speculation

The semantics formalizes the execution of source programs on a pipelined processor

$$C \rightarrow_o^d C'$$

Read as: “under the directive d , the configuration C moves to C' with observation o ” (what the attacker can see), where

- $d ::= \text{fetch} \mid \text{fetch } b \mid \text{exec } n \mid \text{retire}$, attacker-supplied directives (also each “class” of directives is a **stage** of the processor’s pipeline)
- $o ::= \epsilon \mid \text{read } (n, ps) \mid \text{write } (n, ps) \mid \text{fail} \mid \text{rollback } (p)$, attacker’s observations
- C, C' are of the form $\langle is, cs, \mu, \rho \rangle$, where:
 - is is a reorder buffer, i.e., a queue of in-flight (target) instructions (directives give the order)
 - cs represents the stack of (source) commands (the current execution path), i.e., the continuation of the program
 - μ is a memory and ρ is a map from variables to values

JIT semantics for speculation: the fetch stage

$$\langle is, cs, \mu, \rho \rangle \xrightarrow{o}^{\text{fetch}} \langle is', cs', \mu, \rho \rangle \quad \langle is, cs, \mu, \rho \rangle \xrightarrow{o}^{\text{fetch } b} \langle is', cs', \mu, \rho \rangle$$

- **Idea:** take a command from the top of cs , flatten it into a sequence of instructions and store them in is

FETCH-SEQ

$$\langle is, (c_1; c_2) : cs, \mu, \rho \rangle \xrightarrow{\text{fetch}}_{\epsilon} \langle is, c_1 : c_2 : cs, \mu, \rho \rangle$$

FETCH-ASGN

$$\langle is, x := e : cs, \mu, \rho \rangle \xrightarrow{\text{fetch}}_{\epsilon} \langle is \uparrow [x := e], cs, \mu, \rho \rangle$$

FETCH-PTR-LOAD

$$\langle is, x := *e : cs, \mu, \rho \rangle \xrightarrow{\text{fetch}}_{\epsilon} \langle is \uparrow [x := \text{load}(e)], cs, \mu, \rho \rangle$$

FETCH-ARRAY-LOAD

$$\frac{c = x := a[e] \quad e_1 = e < \text{length}(a) \quad e_2 = \text{base}(a) + e \quad c' = \text{if } e_1 \text{ then } x := *e_2 \text{ else fail}}{\langle is, c : cs, \mu, \rho \rangle \xrightarrow{\text{fetch}}_{\epsilon} \langle is, c' : cs, \mu, \rho \rangle}$$

$$\langle is, c : cs, \mu, \rho \rangle \xrightarrow{\text{fetch}}_{\epsilon} \langle is, c' : cs, \mu, \rho \rangle$$

FETCH-IF-TRUE

$$\frac{c = \text{if } e \text{ then } c_1 \text{ else } c_2 \quad \text{fresh}(p) \quad i = \text{guard}(e^{\text{true}}, c_2 : cs, p)}{\langle is, c : cs, \mu, \rho \rangle \xrightarrow{\text{fetch true}}_{\epsilon} \langle is \uparrow [i], c_1 : cs, \mu, \rho \rangle}$$

$$\langle is, c : cs, \mu, \rho \rangle \xrightarrow{\text{fetch true}}_{\epsilon} \langle is \uparrow [i], c_1 : cs, \mu, \rho \rangle$$

**The attacker decides
which branch**

JIT semantics for speculation: fetch-if-true

FETCH-IF-TRUE

$$\frac{c = \text{if } e \text{ then } c_1 \text{ else } c_2 \quad \text{fresh}(p) \quad i = \text{guard}(e^{\text{true}}, c_2 : cs, p)}{\langle is, c : cs, \mu, \rho \rangle \xrightarrow{\text{fetch true}}_e \langle is \uparrow [i], c_1 : cs, \mu, \rho \rangle}$$

The attacker forces the speculation on the true branch, so

- the **guard true instruction** is put in *is* and on the top of the stack and
- the **then-branch c_1** is put onto the stack *cs*
- The guard command **the else-branch together with a copy of the current commands stack** (i.e., $c_2 : cs$) as a rollback stack to restart the execution in case of misprediction

JIT semantics for speculation: the execute stage

$$\langle is, cs, \mu, \rho \rangle \rightarrow_o^{\text{exec } n} \langle is', cs', \mu, \rho \rangle$$

- **Idea:** take the n -th instruction from is (provided by the attacker) and execute it **speculatively**
 - Lots of details here, but intuitively **operands of instruction** are evaluated just using **non-transient information** (computed by an ad hoc function ϕ that marks variables from pending assignments)

EXECUTE

$$\frac{|is_1| = n - 1 \quad \rho' = \phi(is_1, \rho) \quad \langle is_1, i, is_2, cs \rangle \xrightarrow{(\mu, \rho', o)} \langle is', cs' \rangle}{\langle is_1 \uparrow [i] \uparrow is_2, cs, \mu, \rho \rangle \xrightarrow{o} \langle is', cs', \mu, \rho \rangle}$$

EXEC-LOAD

$$\frac{i = (x := \text{load}(e)) \quad \text{store}(_, _) \notin is_1 \quad n = \llbracket e \rrbracket^\rho \quad ps = \langle is_1 \rangle \quad i' = (x := \mu(n))}{\langle is_1, i, is_2, cs \rangle \xrightarrow{(\mu, \rho, \text{read}(n, ps))} \langle is_1 \uparrow [i'] \uparrow is_2, cs \rangle}$$

EXEC-BRANCH-OK

$$\frac{i = \text{guard}(e^b, cs', p) \quad \llbracket e \rrbracket^\rho = b}{\langle is_1, i, is_2, cs \rangle \xrightarrow{(\mu, \rho, \epsilon)} \langle is_1 \uparrow [\text{nop}] \uparrow is_2, cs \rangle}$$

EXEC-BRANCH-MISPREDICT

$$\frac{i = \text{guard}(e^b, cs', p) \quad b' = \llbracket e \rrbracket^\rho \quad b' \neq b}{\langle is_1, i, is_2, cs \rangle \xrightarrow{(\mu, \rho, \text{rollback}(p))} \langle is_1 \uparrow [\text{nop}], cs' \rangle}$$

JIT semantics for speculation: the retire stage

$$\langle is, cs, \mu, \rho \rangle \xrightarrow{o}^{\text{retire}} \langle is', cs', \mu', \rho' \rangle$$

- **Idea:** remove complete instructions from is and commit changes to memory and variable map
 - Do that in-order, so giving the illusion of no instruction-level parallelism!

RETIRE-NOP

$$\langle \text{nop} : is, cs, \mu, \rho \rangle \xrightarrow{\epsilon}^{\text{retire}} \langle is, cs, \mu, \rho \rangle$$

RETIRE-ASGN

$$\langle x := v : is, cs, \mu, \rho \rangle \xrightarrow{\epsilon}^{\text{retire}} \langle is, cs, \mu, \rho[x \mapsto v] \rangle$$

RETIRE-STORE

$$\langle \text{store}(n, v) : is, cs, \mu, \rho \rangle \xrightarrow{\epsilon}^{\text{retire}} \langle is, cs, \mu[n \mapsto v], \rho \rangle$$

RETIRE-FAIL

$$\langle \text{fail}(p) : is, cs, \mu, \rho \rangle \xrightarrow{\text{fail}(p)}^{\text{retire}} \langle [], [], \mu, \rho \rangle$$

A few words about protect

- protect can be implemented both in software and in hardware
- **Software implementation:** no support from hardware needed
 - exactly as SLH for array access (i.e., computes a **mask** depending on the array's **bound check**, and then executes the access depending using the mask)
- **Hardware implementation:** Can you guess? What needs to be added?
 - **Fetch stage:** translates protect into protect, easy
 - **Exec aux. relation:** reduces the parameter to a value, then if no guard is pending removes the protect
 - Instruction $x := \text{protect}(r)$ assigns the value of r , only after all previous guard instructions have been executed, i.e., when the value has become stable and no more rollbacks are possible

Protect: software implementation

FETCH-PROTECT-SLH

$$\frac{\begin{array}{llll} c = x := \text{protect}(a[e]) & e_1 = e < \text{length}(a) & e_2 = \text{base}(a) + e & \\ c_1 = m := e_1 & c_2 = m := m ? 1 : 0 & c_3 = x := *(e_2 \otimes m) & c' = c_1; \text{if } m \text{ then } c_2; c_3 \text{ else fail} \end{array}}{\langle is, c : cs, \mu, \rho \rangle \xrightarrow{\text{fetch}}_{\epsilon} \langle is, c' : cs, \mu, \rho \rangle}$$

(b) Software implementation of `protect(a[e])`.

```
uint mask = ALL_ONES; /* all bits set to 1 */
if (cond) {
    mask = !cond ? ALL_ZEROES : mask;
    // ...
    y = B[(A[x] * 512) & mask];
}
else {
    mask = cond ? ALL_ZEROES : mask;
    /* ... */
}
```

Example: the source

```
if (i1 < length(a))  
{x := a[i1];}  
if (i2 < length(a))  
{y := a[i2];}  
  z := x + y  
if (z < length(b))  
{w := b[z]}
```

Example: the target

```
if (i1 < length(a))
{x := a[i1];}
if (i2 < length(a))
{y := a[i2];}
  z := x + y
if (z < length(b))
{w := b[z]}
```

x := load(e)
guard(eb, cs, p)

Attacker's directives

1	guard (($i_1 < \text{length}(a)$) ^{true} , [fail], 1)
2	x := load (base(a) + i_1)
3	guard (($i_2 < \text{length}(a)$) ^{true} , [fail], 2)
4	y := load (base(a) + i_2)
5	z := x + y
6	guard (($z < \text{length}(b)$) ^{true} , [fail], 3)
7	w := load (base(b) + z)

The attacker instructs the processor to speculatively execute the load instructions (2,4,5,7) and the assignment, but not the guards

Example of leaking execution

Evolution of the buffer is after each attacker directive (execute 2,4,5,7); the memory μ and variable map ρ are shown above

$C \rightarrow_o^d C'$
 C, C' are of the form
 $\langle is, cs, \mu, \rho \rangle$

Memory Layout

$\mu(0) = 0$	$b[0]$
$\mu(1) = 0$	$a[0]$
$\mu(2) = 0$	$a[1]$
$\mu(3) = 42$	$s[0]$
\dots	\dots

Variable Map

$\rho(i_1) = 1$
 $\rho(i_2) = 2$
 \dots

	Reorder Buffer	is
1	$\text{guard}((i_1 < \text{length}(a))^{\text{true}}, [\text{fail}], 1)$	
2	$x := \text{load}(\text{base}(a) + i_1)$	
3	$\text{guard}((i_2 < \text{length}(a))^{\text{true}}, [\text{fail}], 2)$	
4	$y := \text{load}(\text{base}(a) + i_2)$	
5	$z := x + y$	
6	$\text{guard}((z < \text{length}(b))^{\text{true}}, [\text{fail}], 3)$	
7	$w := \text{load}(\text{base}(b) + z)$	
Observations: o		1

Example: exec 2

Directive **exec 2** executes the first load by computing the memory address 2 and replacing the instruction with the assignment $x := \mu(2)$ containing the loaded value 0

$C \rightarrow_o^d C'$
 C, C' are of the form
 $\langle is, cs, \mu, \rho \rangle$

Memory Layout

$\mu(0) = 0$	$b[0]$
$\mu(1) = 0$	$a[0]$
$\mu(2) = 0$	$a[1]$
$\mu(3) = 42$	$s[0]$
...	...

Variable Map

$\rho(i_1) = 1$
 $\rho(i_2) = 2$
 ...

Reorder Buffer	iS	exec 2
1	$\text{guard}((i_1 < \text{length}(a))^{\text{true}}, [\text{fail}], 1)$	
2	$x := \text{load}(\text{base}(a) + i_1)$	$x := \mu(2)$
3	$\text{guard}((i_2 < \text{length}(a))^{\text{true}}, [\text{fail}], 2)$	
4	$y := \text{load}(\text{base}(a) + i_2)$	
5	$z := x + y$	
6	$\text{guard}((z < \text{length}(b))^{\text{true}}, [\text{fail}], 3)$	
7	$w := \text{load}(\text{base}(b) + z)$	
Observations: o		$\text{read}(2, [1])$

Example: exec 4

Directive **exec 4** transiently reads public array *a* past its bound, at index 2, reading into the memory ($\mu(3) = 42$) of secret array *s* [0] and generates the corresponding observation

Memory Layout		Variable Map	
$\mu(0) = 0$	<i>b</i> [0]	$\rho(i_1) = 1$	
$\mu(1) = 0$	<i>a</i> [0]	$\rho(i_2) = 2$	
$\mu(2) = 0$	<i>a</i> [1]	...	
$\mu(3) = 42$	<i>s</i> [0]		
...	...		

Reorder Buffer	<i>i</i> <i>s</i>	exec 2	exec 4
1 guard($(i_1 < \text{length}(a))^{\text{true}}$, [fail], 1)			
2 $x := \text{load}(\text{base}(a) + i_1)$		$x := \mu(2)$	
3 guard($(i_2 < \text{length}(a))^{\text{true}}$, [fail], 2)			
4 $y := \text{load}(\text{base}(a) + i_2)$			$y := \mu(3)$
5 $z := x + y$			
6 guard($(z < \text{length}(b))^{\text{true}}$, [fail], 3)			
7 $w := \text{load}(\text{base}(b) + z)$			
Observations: <i>o</i>		read(2, [1])	read(3, [1, 2])

Example: exec 5

The processor forwards the values of x and y through the transient variable map ρ [$x \mapsto \mu(2)$, $y \mapsto \mu(3)$] to compute their sum in the fifth instruction, ($z := 42$)

Memory Layout		Variable Map		
$\mu(0) = 0$	$b[0]$	$\rho(i_1) = 1$		
$\mu(1) = 0$	$a[0]$	$\rho(i_2) = 2$		
$\mu(2) = 0$	$a[1]$	\dots		
$\mu(3) = 42$	$s[0]$			
\dots	\dots			

Reorder Buffer	iS	exec 2	exec 4	exec 5
1	$\text{guard}((i_1 < \text{length}(a))^{\text{true}}, [\text{fail}], 1)$			
2	$x := \text{load}(\text{base}(a) + i_1)$	$x := \mu(2)$		
3	$\text{guard}((i_2 < \text{length}(a))^{\text{true}}, [\text{fail}], 2)$			
4	$y := \text{load}(\text{base}(a) + i_2)$		$y := \mu(3)$	
5	$z := x + y$			$z := 42$
6	$\text{guard}((z < \text{length}(b))^{\text{true}}, [\text{fail}], 3)$			
7	$w := \text{load}(\text{base}(b) + z)$			
Observations: o		$\text{read}(2, [1])$	$\text{read}(3, [1, 2])$	ϵ

Example: exec 7

The value of z is then used as an index in the last instruction and leaked to the attacker via observation $\text{read}(42, [1, 2, 3])$

Memory Layout		Variable Map			
$\mu(0) = 0$	$b[0]$	$\rho(i_1) = 1$			
$\mu(1) = 0$	$a[0]$	$\rho(i_2) = 2$			
$\mu(2) = 0$	$a[1]$	\dots			
$\mu(3) = 42$	$s[0]$				
\dots	\dots				
Reorder Buffer	iS	exec 2	exec 4	exec 5	exec 7
1	$\text{guard}((i_1 < \text{length}(a))^{\text{true}}, [\text{fail}], 1)$				
2	$x := \text{load}(\text{base}(a) + i_1)$	$x := \mu(2)$			
3	$\text{guard}((i_2 < \text{length}(a))^{\text{true}}, [\text{fail}], 2)$				
4	$y := \text{load}(\text{base}(a) + i_2)$		$y := \mu(3)$		
5	$z := x + y$			$z := 42$	
6	$\text{guard}((z < \text{length}(b))^{\text{true}}, [\text{fail}], 3)$				
7	$w := \text{load}(\text{base}(b) + z)$				$w := \mu(42)$
Observations: o		$\text{read}(2, [1])$	$\text{read}(3, [1, 2])$	ϵ	$\text{read}(42, [1, 2, 3])$

Almost there...

The language(s) we just saw are nice, but

- Can we be sure that the programs are *speculatively secure*?

Two possibilities



We prove **manually** that a program is not vulnerable to Spectre

We **let the machine** to prove that a program is not vulnerable to Spectre

Let the machine prove things...

... via **type checking**!

Transient-flow type system that statically rejects programs that can potentially leak through transient execution attacks: i.e. that exhibit any source-to-sink data flows

Expressions:

$$\Gamma \vdash e : \tau$$

Reads as: “expression e has type τ under Γ ”

- Γ maps variables to types
- τ is a type, can be either
 - **Stable S**, i.e., contains no transient values during executions
 - **Transient T**, otherwise
 - $S \sqsubseteq T$ (can-flow-to relation) not viceversa

Commands:

$$\Gamma, \text{Prot} \vdash c$$

Reads as: “command c is well-typed under Γ and Prot ”

- Γ as before, Prot a set of implicitly protected vars
- **Idea:** assume that all assignments to variables in Prot are protected, then c does not leak

Let the machine prove things... (cont)

... via a type **inference** (type constraints restrict the types)!

Expressions:

$$\Gamma \vdash e : \tau \Rightarrow k$$

Commands:

We record which variables are protected

$$\Gamma, \text{Prot} \vdash c \Rightarrow k$$

In both cases, the algorithm

- i. Generates a set of type constraints k , involving concrete types and atoms (i.e., program variables and type variables)
- ii. Builds a **def-use** graph: nodes are types and atoms, edges their relations; If there is a path from **T** to **S** the program is leaky (Thm: k is satisfiable iff no path from **T** to **S** exists) and reconstructs type information
- iii. Cuts all the **T** to **S** paths so to make k satisfiable, and protects each program variable in the cut (note that type variables are never in cuts; Why?)

Let the machine prove things... (cont)

This rule disallows assignment from **T** to **S**, since **T** \sqsubseteq **S** is not true

$$\text{ASGN} \quad \frac{\Gamma \vdash r : \tau \Rightarrow k \quad \tau \sqsubseteq \Gamma(x)}{\Gamma, \text{Prot} \vdash x := r \Rightarrow k \cup (r \sqsubseteq x)}$$

The rule generates the constraint **r \sqsubseteq x** disallowing transient to stable assignments

Let the machine prove things... (cont)

In case of operation with an expression of type **T** and one of type **S**, since $\mathbf{T} \sqsubseteq \mathbf{S}$, the overall type will be **T**

$$\begin{array}{c} \text{BOP} \\ \frac{\Gamma \vdash e_1 : \tau_1 \Rightarrow k_1 \quad \Gamma \vdash e_2 : \tau_2 \Rightarrow k_2 \quad \tau_1 \sqsubseteq \tau \quad \tau_2 \sqsubseteq \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau \Rightarrow k_1 \cup k_2 \cup (e_1 \sqsubseteq e_1 \oplus e_2) \cup (e_2 \sqsubseteq e_1 \oplus e_2)} \end{array}$$

the unknown type of $e_1 \oplus e_2$ should be at least as transient as the (unknown) type of e_1 and e_2

Let the machine prove things... (cont)

Array can be indexed out of bounds during speculation, hence the rule assigns the transient type **T** to array reads. E must be of type to avoid leaks

ARRAY-READ

$$\frac{\Gamma \vdash e : \mathbf{S} \Rightarrow k}{\Gamma \vdash a[e] : \mathbf{T} \Rightarrow k \cup (e \sqsubseteq \mathbf{S}) \cup (\mathbf{T} \sqsubseteq a[e])}$$

(a) Typing rules for expressions and arrays.

The rule generates constraint $e \sqsubseteq \mathbf{S}$ for the unknown type of the array index, thus forcing it to typed be **S** and forces the type of $a[e]$ to be **T**

Let the machine prove things... (cont)

The indexed expression used in array writes must be **S**, while the stored value can have any type

$$\begin{array}{c} \text{ARRAY-WRITE} \\ \frac{\Gamma \vdash e_1 : \mathbf{S} \Rightarrow k_1 \quad \Gamma \vdash e_2 : \tau \Rightarrow k_2}{\Gamma, \text{Prot} \vdash a[e_1] := e_2 \Rightarrow k_1 \cup k_2 \cup (e_1 \sqsubseteq \mathbf{S})} \end{array}$$

Let the machine prove things... (cont)

To prevent leaks, branch condition must be **S**

IF-THEN-ELSE

$$\frac{\Gamma \vdash e : \mathbf{S} \Rightarrow k \quad \Gamma, \text{Prot} \vdash c_1 \Rightarrow k_1 \quad \Gamma, \text{Prot} \vdash c_2 \Rightarrow k_2}{\Gamma, \text{Prot} \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 \Rightarrow k \cup k_1 \cup k_2 \cup (e \sqsubseteq \mathbf{S})}$$

Example: constraint generation

EXAMPLE

$x := a[i_1]$

$y := a[i_2]$

$z := x + y$

$w := b[z]$

Example: constraint generation

EXAMPLE

$x := a[i_1]$
 $y := a[i_2]$
 $z := x + y$
 $w := b[z]$

ARRAY-READ

$$\frac{\Gamma \vdash e : \mathbf{S} \Rightarrow k}{\Gamma \vdash a[e] : \mathbf{T} \Rightarrow k \cup (e \sqsubseteq \mathbf{S}) \cup (\mathbf{T} \sqsubseteq a[e])}$$

ASGN

$$\frac{\Gamma \vdash r : \tau \Rightarrow k \quad \tau \sqsubseteq \Gamma(x)}{\Gamma, \text{Prot} \vdash x := r \Rightarrow k \cup (r \sqsubseteq x)}$$

Example: constraint generation

EXAMPLE

$x := a[i_1]$
 $y := a[i_2]$
 $z := x + y$
 $w := b[z]$

CONSTRAINTS

$T \sqsubseteq a[i_1]$

$a[i_1] \sqsubseteq x$

- Reading memory creates T value
- Types are propagated
- Constant values like a_1 do not produce constraints

ARRAY-READ

$$\frac{\Gamma \vdash e : S \Rightarrow k}{\Gamma \vdash a[e] : T \Rightarrow k \cup (e \sqsubseteq S) \cup (T \sqsubseteq a[e])}$$

ASGN

$$\frac{\Gamma \vdash r : \tau \Rightarrow k \quad \tau \sqsubseteq \Gamma(x)}{\Gamma, \text{Prot} \vdash x := r \Rightarrow k \cup (r \sqsubseteq x)}$$

propagation

Example: constraint generation

EXAMPLE

$x := a[i_1]$
 $y := a[i_2]$
 $z := x + y$
 $w := b[z]$

CONSTRAINTS
$T \sqsubseteq a[i_1]$
$a[i_1] \sqsubseteq x$
$T \sqsubseteq a[i_2]$
$a[i_2] \sqsubseteq y$

ARRAY-READ

$$\frac{\Gamma \vdash e : S \Rightarrow k}{\Gamma \vdash a[e] : T \Rightarrow k \cup (e \sqsubseteq S) \cup (T \sqsubseteq a[e])}$$

ASGN

$$\frac{\Gamma \vdash r : \tau \Rightarrow k \quad \tau \sqsubseteq \Gamma(x)}{\Gamma, \text{Prot} \vdash x := r \Rightarrow k \cup (r \sqsubseteq x)}$$

Example: constraint generation

EXAMPLE

$x := a[i_1]$
 $y := a[i_2]$
 $z := x + y$
 $w := b[z]$

CONSTRAINTS
$\mathbf{T} \sqsubseteq a[i_1]$
$a[i_1] \sqsubseteq x$
$\mathbf{T} \sqsubseteq a[i_2]$
$a[i_2] \sqsubseteq y$
$x \sqsubseteq x+y$
$y \sqsubseteq x+y$
$x+y \sqsubseteq z$

BOP

$$\frac{\Gamma \vdash e_1 : \tau_1 \Rightarrow k_1 \quad \Gamma \vdash e_2 : \tau_2 \Rightarrow k_2 \quad \tau_1 \sqsubseteq \tau \quad \tau_2 \sqsubseteq \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau \Rightarrow k_1 \cup k_2 \cup (e_1 \sqsubseteq e_1 \oplus e_2) \cup (e_2 \sqsubseteq e_1 \oplus e_2)}$$

ASGN

$$\frac{\Gamma \vdash r : \tau \Rightarrow k \quad \tau \sqsubseteq \Gamma(x)}{\Gamma, \text{Prot} \vdash x := r \Rightarrow k \cup (r \sqsubseteq x)}$$

x and y used to compute x+y

Example: constraint generation

EXAMPLE

$x := a[i_1]$
 $y := a[i_2]$
 $z := x + y$
 $w := b[z]$

CONSTRAINTS
$T \sqsubseteq a[i_1]$
$a[i_1] \sqsubseteq x$
$T \sqsubseteq a[i_2]$
$a[i_2] \sqsubseteq y$
$x \sqsubseteq x+y$
$y \sqsubseteq x+y$
$x+y \sqsubseteq z$
$z \sqsubseteq S$

ARRAY-READ

$$\frac{\Gamma \vdash e : S \Rightarrow k}{\Gamma \vdash a[e] : T \Rightarrow k \cup (e \sqsubseteq S) \cup (T \sqsubseteq a[e])}$$

ASGN

$$\frac{\Gamma \vdash r : \tau \Rightarrow k \quad \tau \sqsubseteq \Gamma(x)}{\Gamma, \text{Prot} \vdash x := r \Rightarrow k \cup (r \sqsubseteq x)}$$

The dataflow

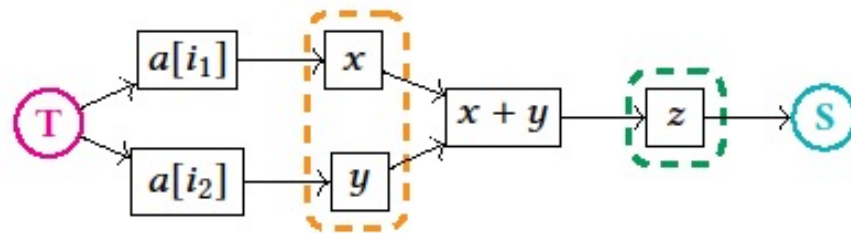
EXAMPLE

$x := a[i_1]$
 $y := a[i_2]$
 $z := x + y$
 $w := b[z]$

CONSTRAINTS

$T \sqsubseteq a[i_1]$
$a[i_1] \sqsubseteq x$
$T \sqsubseteq a[i_2]$
$a[i_2] \sqsubseteq y$
$x \sqsubseteq x+y$
$y \sqsubseteq x+y$
$x+y \sqsubseteq z$
$z \sqsubseteq S$

Def-Use Graph



- A set of constraints k is **satisfiable** if and only if there is no T - S path in k
- Here we have T - S paths

Cut the dataflow

EXAMPLE

$x := a[i_1]$

$y := a[i_2]$

$z := x + y$

$w := b[z]$

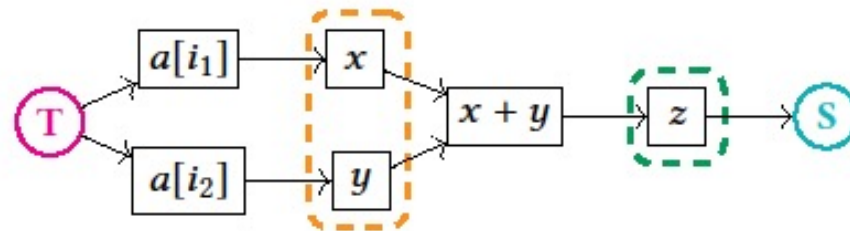
CUT-SETS

$\{x, y\}$

$\{z\}$

$\{z\}$ is a **minimal** cut

- If a **set of constraints is unsatisfiable**, as here, we can make it satisfiable by removing some of the nodes or equivalently protecting some of the variables
- A is a **cut-set** for a set of constraints k , if A cuts all T-S paths in k .



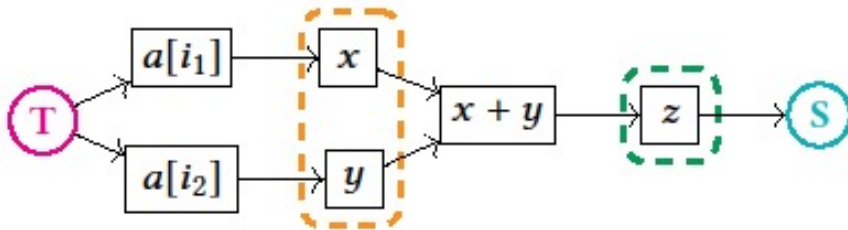
Program repair

Program Repair

As a final step, our repair algorithm repair traverses program c and inserts a **protect** for each variable in the cut-set

EXAMPLE

```
 $x := a[i_1]$   
 $y := a[i_2]$   
 $z := x + y$   
 $w := b[z]$ 
```



EXAMPLE PATCHED

```
 $x := \text{protect}(a[i_1])$   
 $y := \text{protect}(a[i_2])$   
 $z := \text{protect}(x + y)$   
 $w := b[z]$ 
```

The **orange** patch is sub-optimal because it requires more protect statements than the optimal **green** patch

Putting everything together

- **Theorem 1 (Consistency):** Programs executed speculatively produce the same result as if they were executed sequentially
- **Constant-time (CT) program** if and only if its branches and memory accesses do not depend on secrets (e.g., crypto keys) – I know, it's strange
- **Speculative constant-time (SCT) program** if and only if its observables and final configuration do not depend on secrets (assume the sequence of attacker-provided directives as fixed before execution)
- **Theorem 2 (Soundness):** If a program c is CT and well-typed $\Gamma, \emptyset \vdash c$ then it is also SCT
 - If you want: when compiling c from the source to the target using the JIT “compiler”, a secure source program (CT & well-typed) is mapped into a secure target program (SCT)

Outline

Intro

Mechanisms, attacks and mitigations

More formally

Conclusions



Summing up

- We worked with the “bare-metal”: caches, speculation, side-channels
- We saw how that can be abstracted elegantly
 - A source language translated on-the-fly to a target one, using a JIT semantics
 - The JIT semantics has nice properties: programs deemed secure at the source are such also at the target
 - Also: it is possible to make programs secure by synthesizing protections
 - Nice theoretical framework 😊

Bibliography

- M. Vassena, C. Disselkoen, K. von Gleissenthall, S. Cauligi, R. Gökhan Kıcı, R. Jhala, D. Tullsen, and D. Stefan. *Automatically eliminating speculative leaks from cryptographic code with blade*. POPL 2021. URL: <https://arxiv.org/abs/2005.00294>
<https://www.youtube.com/watch?v=mCzbjGRIVeQ>
- Canella, Claudio, et al. *A systematic evaluation of transient execution attacks and defenses*. 28th USENIX Security Symposium (USENIX Security 19). 2019. URL: <https://arxiv.org/abs/1811.05441v3>

End