Spectre Through the Looking Glass

Stefan Brunthaler

SPECTRE • CODE • BUM

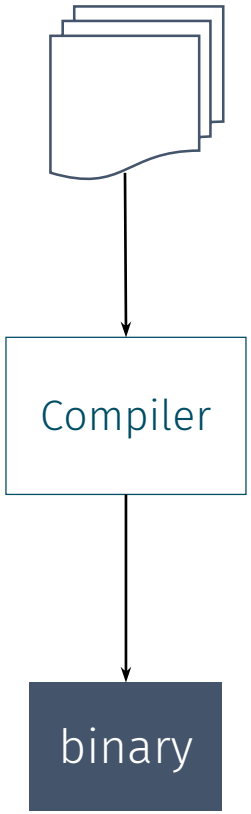
Security, privacy & Performance Enhancing Compilation Techniques Research lab

National Cyber Defense Research Institute

Bundeswehr University Munich

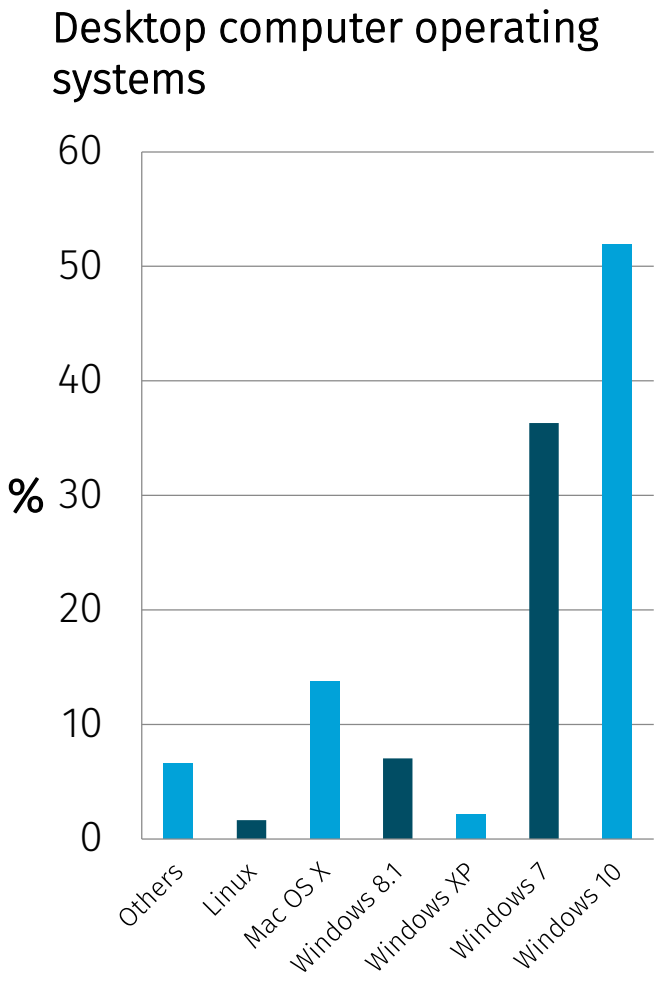
Software diversity

Software Monoculture



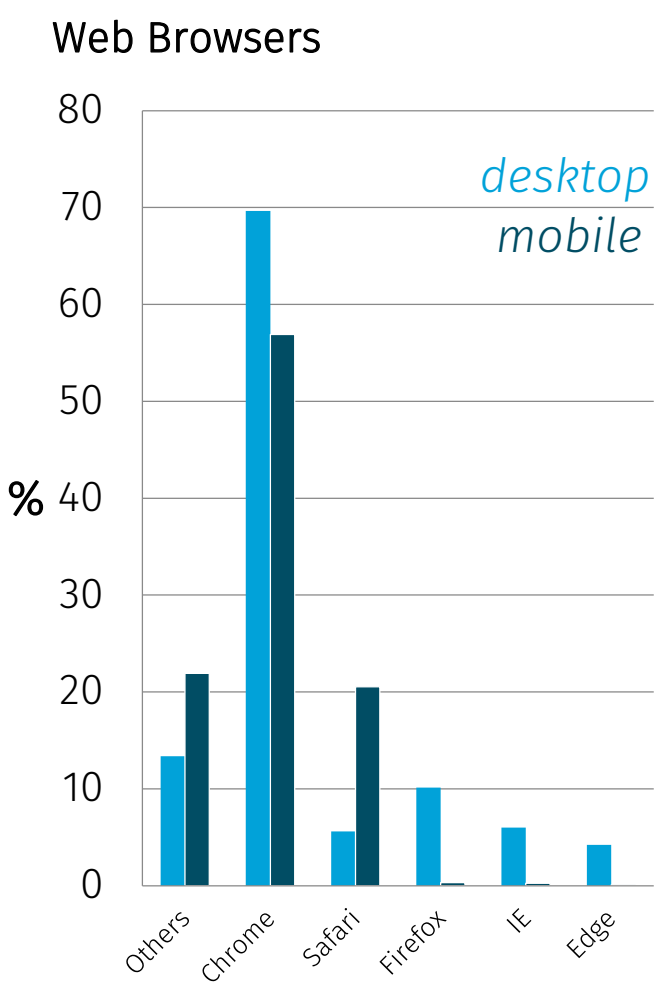
All users run the *same* binary
(incl. attackers)

Software Monoculture



- Win7 + Win10 ≈ 88%
- 100m+ machines

Software Monoculture



Cross platform threats:

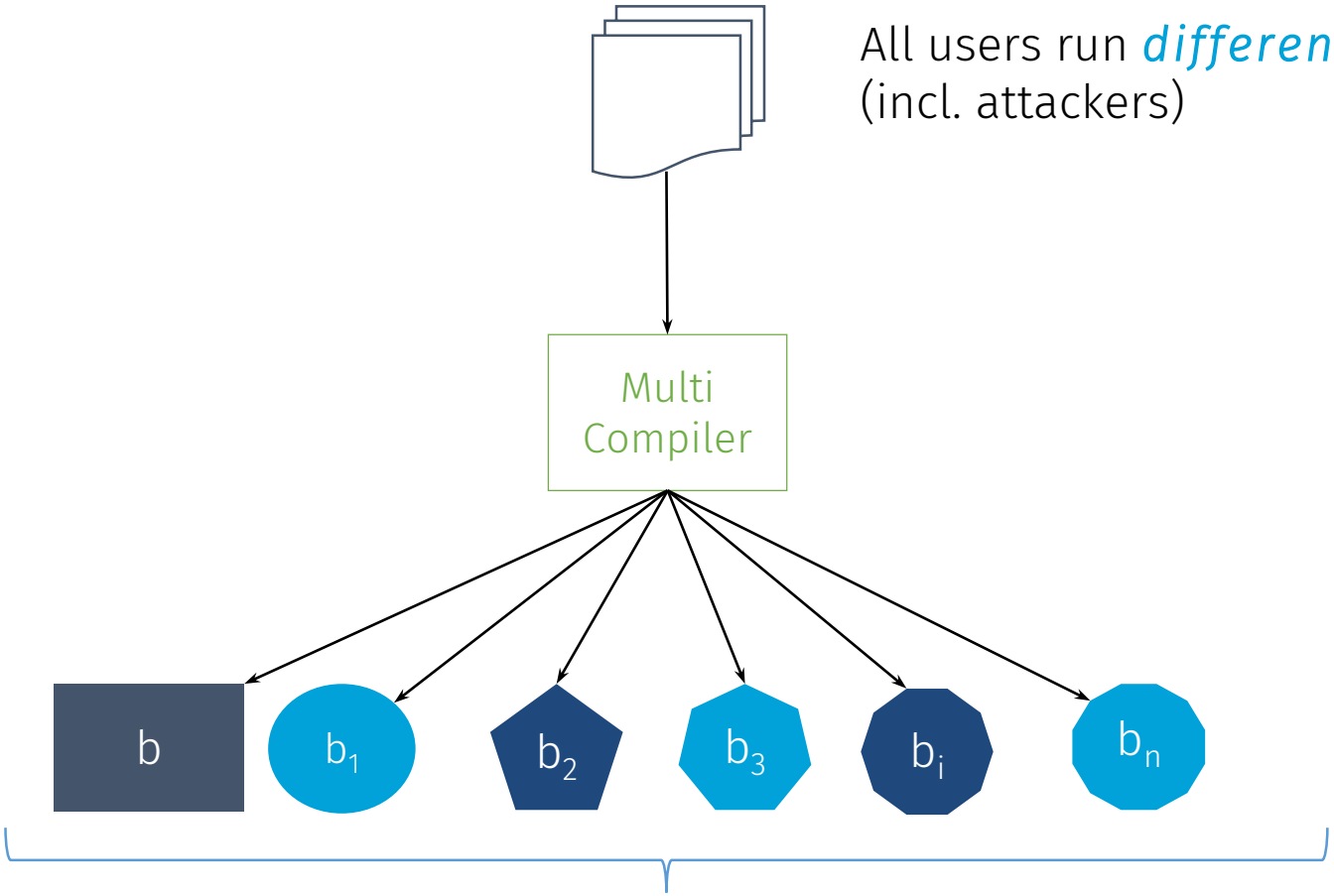
- Adobe Flash Player
- Oracle's Java VM
- Microsoft Office
- Browsers
  - Internet Explorer
  - Google Chrome
  - Mozilla Firefox
  - Apple Safari

IoT devices:

Mirai (Deutsche Telekom)

➡ enables large-scale exploitation

Software Diversity

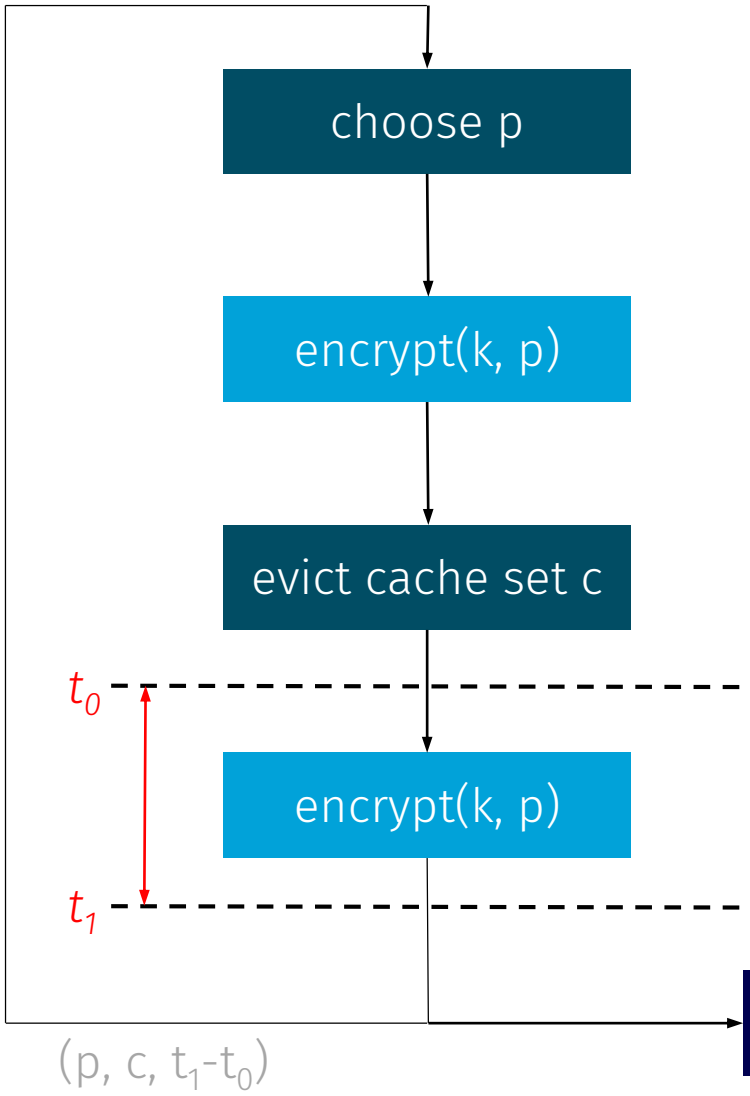


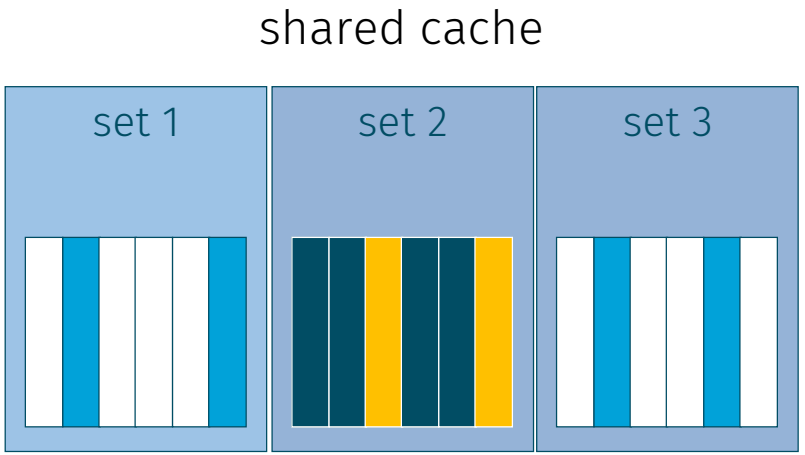
All users run *different* binaries
(incl. attackers)

diversified code
same semantics

Timing Channel in AES

EVICT+TIME AES Cache-based Side Channel Attack:





p..... plaintext
k..... key
c..... cache set

Tromer et al., Efficient cache attacks on AES, and countermeasures, *Journal of Cryptology*, 2010.

The Spectre Family of Attacks

- Spectre is used as an umbrella term
- several variants known:
  - Variant 1: Bounds check bypass
  - Variant 2: Branch Target Injection

  } a.k.a. Spectre

  - Variant 3: Speculative Load (a.k.a. Meltdown)
  - Variant 4: Speculative Store Bypass
  - Variant 5: L1 Terminal Fault (a.k.a. Foreshadow)
- situation resembling cancer

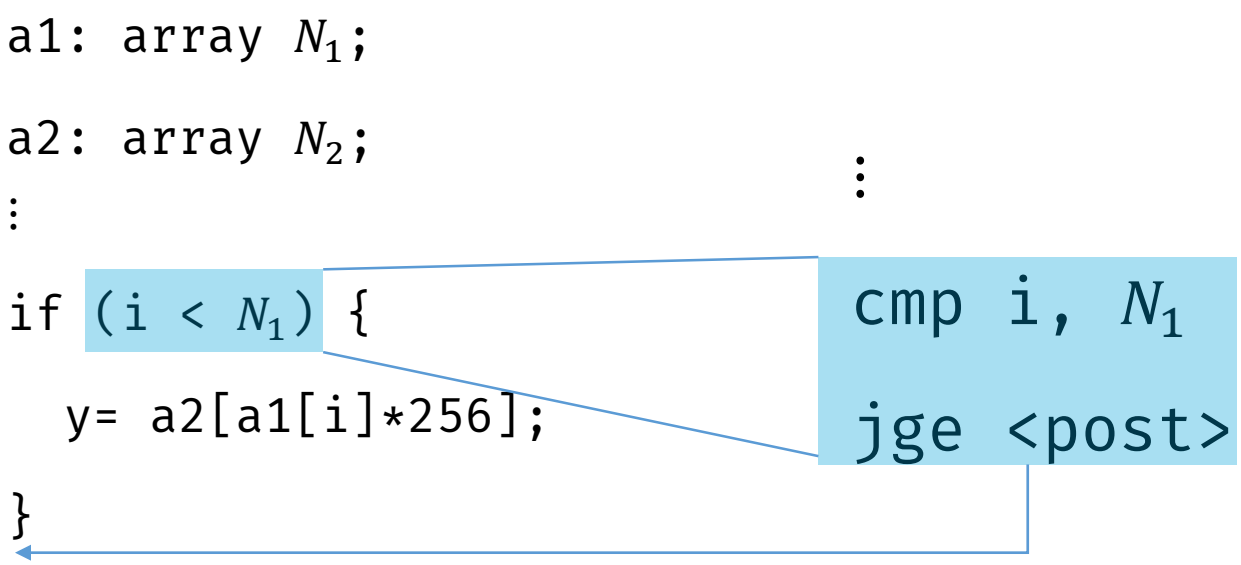
The Spectre Family of Attacks

- Spectre is used as an umbrella term
- several variants known:
  - *Variant 1: Bounds check bypass*
  - *Variant 2: Branch Target Injection*

  } a.k.a. Spectre

  - Variant 3: Speculative Load (a.k.a. Meltdown)
  - *Variant 4: Speculative Store Bypass*
  - Variant 5: L1 Terminal Fault (a.k.a. Foreshadow)
- situation resembling cancer

Spectre V1: Bounds-Check Bypass



Spectre V1: Bounds-Check Bypass

```
a1: array N1;
a2: array N2;
⋮
if (i < N1) {
  y= a2[a1[i]*256];
}
```

```
⋮
cmp i, N1
jge <post>
```

predicted as either

- taken
- not-taken

Spectre V1: Bounds-Check Bypass

```
a1: array N1;
a2: array N2;
⋮
if (i < N1) {
  y= a2[a1[i]*256];
}
```

```
⋮
cmp i, N1
jge <post>
```

predicted as either

- taken
- **not-taken**

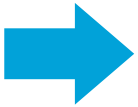
→ allows us to “widen” window of speculative execution

Spectre V1: Bounds-Check Bypass

```
a1: array N1;
a2: array N2;
⋮
if (i < N1) {
  y= a2[a1[i]*256];
}
```

⇝ "will not be executed"

⇝ $i \geq N_1$



can read arbitrary memory during speculative execution

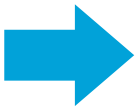
Spectre V1: Bounds-Check Bypass

```
for (i=0; i<10; i++) {
    x= a[i];

}
```

⇝ "will not be executed"

⇝ `i = 11`



can read past array bounds

Spectre V1: Bounds-Check Bypass

```
a1: array N1;
a2: array N2;
⋮
if (i < N1) {
  y= a2[a1[i]*256];
}
```

- idea is to *force* execution of a bounds-check operation

Spectre V1: Bounds-Check Bypass

```
a1: array N1;
a2: array N2;
⋮
if (i < N1) {
  i = i % N1;
  y= a2[a1[i]*256];
}
```

- idea is to *force* execution of a bounds-check operation

pros:

- will always be executed
- trivially generated by a compiler
- manual insertion by Linux kernel developers :(

Spectre V1: Bounds-Check Bypass

```
for (i=0; i<10; i++) {
    i= i % 10;
    x= a[i];

}
```

Q: What if this modulo operation is expensive?

Spectre V1: Bounds-Check Bypass

```
for (i=0; i<8; i++) {
    i= i % 8;
    x= a[i];
    …
}
x= a[8];
…
x= a[9];
…
```

unrolled twice

Use compiler optimization knowledge to our advantage!

- decrease upper bound to nearest power of two
- postfix unrolling missing iterations

Spectre V1: Bounds-Check Bypass

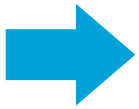
```
for (i=0; i<8; i++) {
    i= i % 8;
    x= a[i];
    …
}
```

```
x= a[8];
…
x= a[9];
…
```

unrolled twice

Use compiler optimization knowledge to our advantage!

- decrease upper bound to nearest power of two
- postfix unrolling missing iterations



great new opportunities for compiler research!

Spectre V1: Bounds-Check Bypass

Observations:

- relatively simple for bounds check
- much harder to secure:
  - checking pointer tags (e.g., type tags)
  - checking reference counts
  - more complicated data-structures

Spectre V2: Branch Target Injection/Poisoning

```
                ⋮
<rip> jmpq *%rax
```

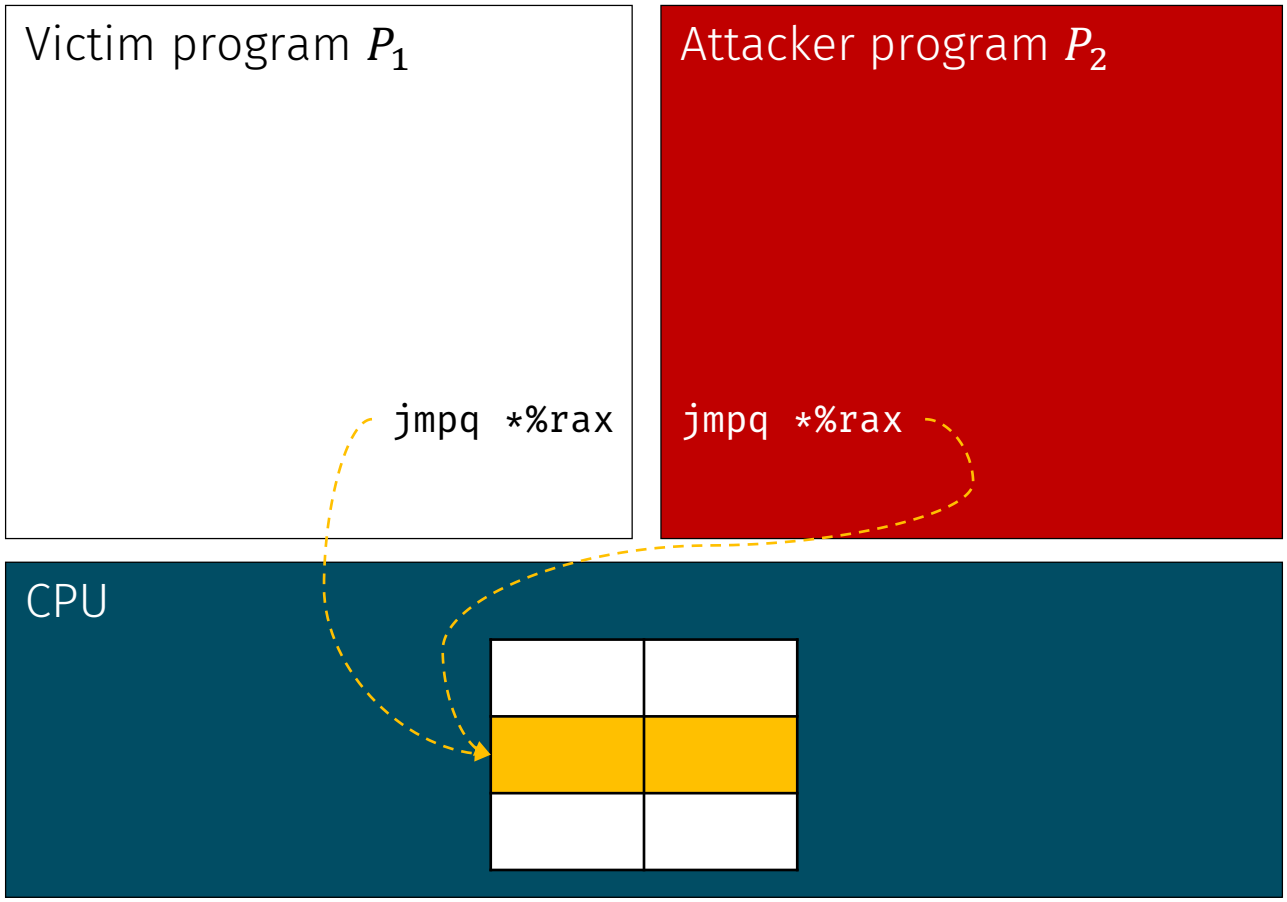
- leverages branch target prediction to improve speculation
- uses `rip` to index into the branch target buffer (BTB, a.k.a. branch target table, branch target address cache)

Spectre V2: Branch Target Injection/Poisoning

```
                ⋮
<rip> jmpq *%rax
```

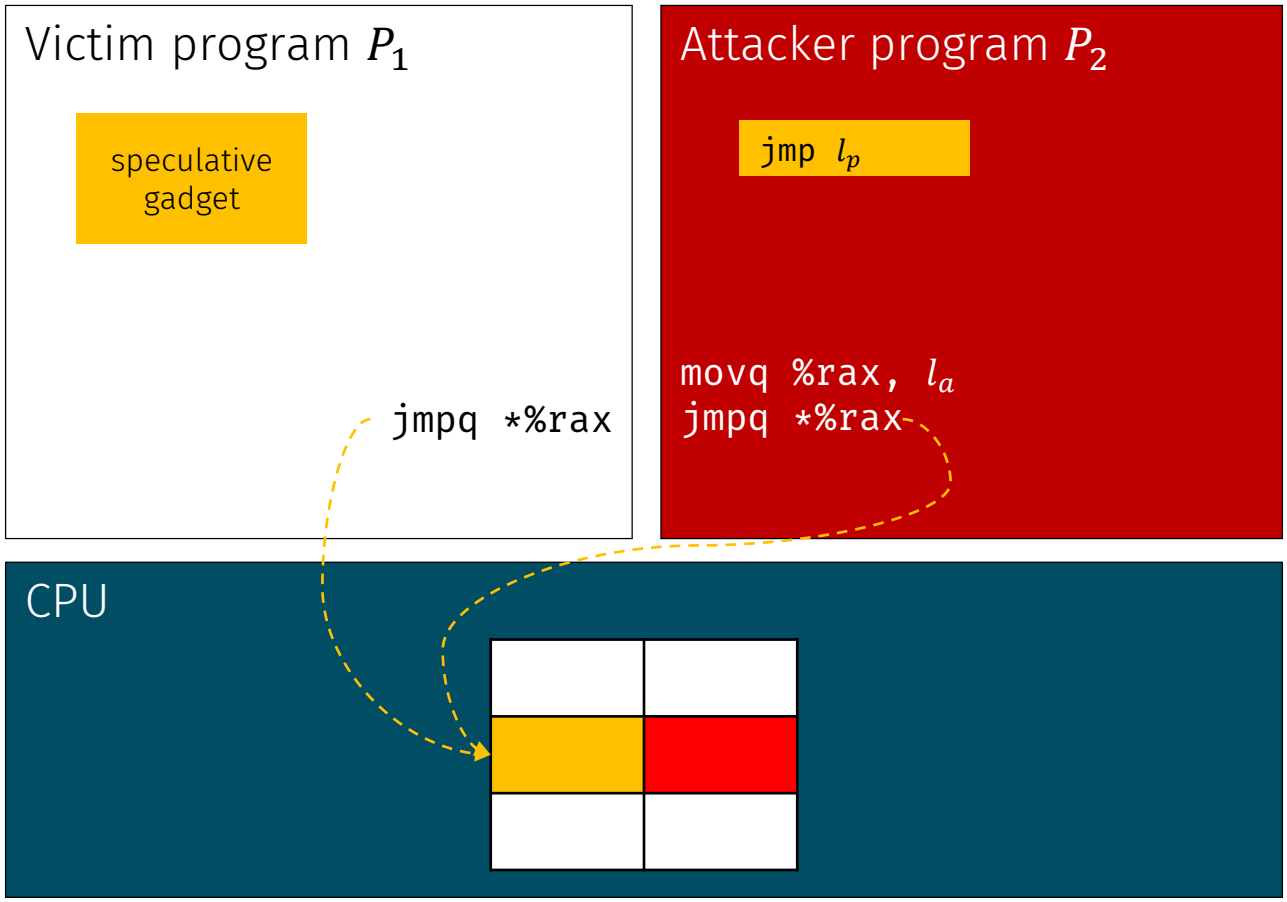
- BTB[`rip`] = $\{l_0, \ldots, l_n\}$
  - contains multiple labels l
  - selects one among those (using some heuristic)
  - cannot possibly hold *all* potential labels → uses optimization similar to n-way cache associativity
    - indexing uses last 12 bits of `rip`

Spectre V2: Branch Target Injection/Poisoning



both programs, P_1 and P_2, index into the ***same*** *BTB entry*

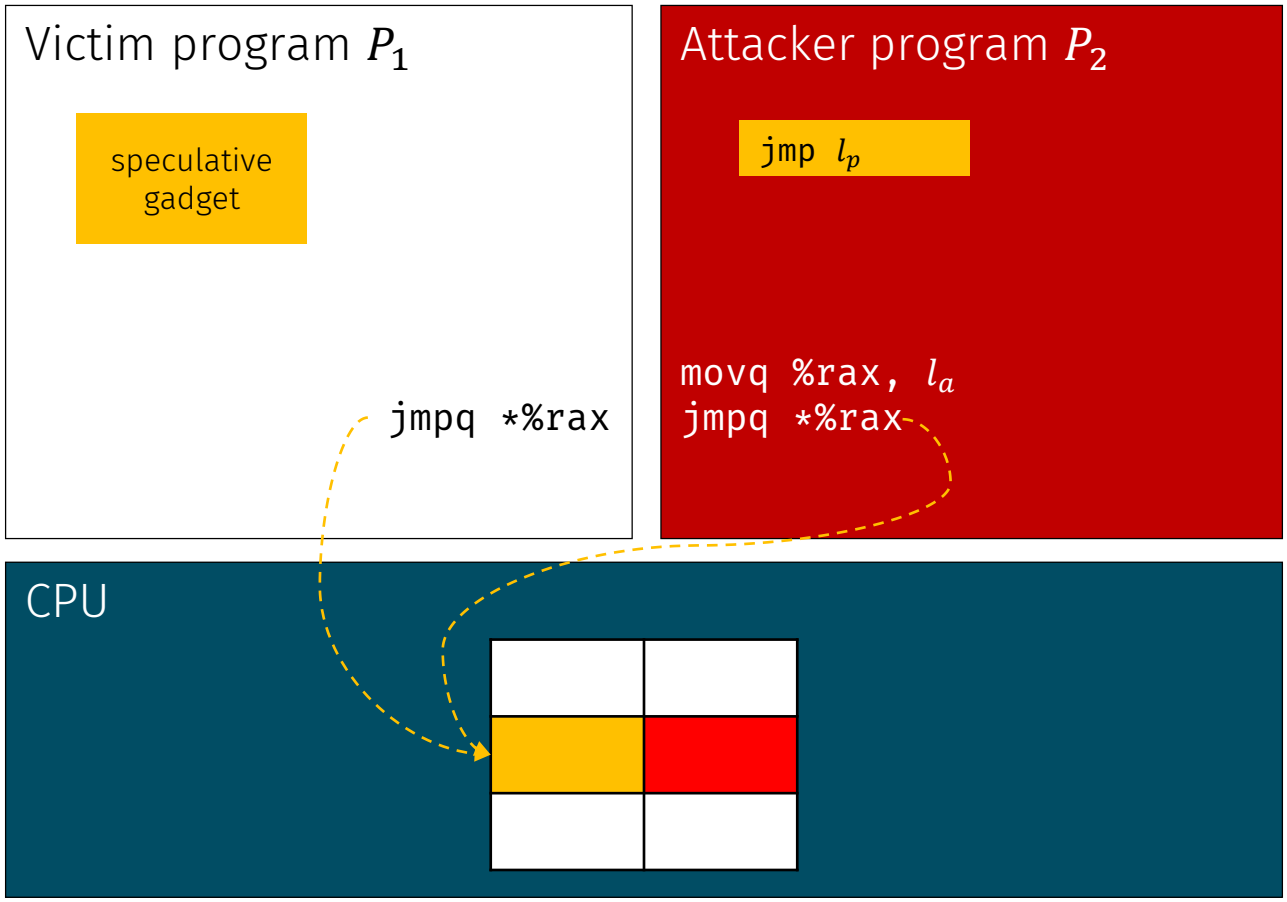
Spectre V2: Branch Target Injection/Poisoning



basically, overloading BTB:

$$\text{BTB}[\texttt{rip}] = \{\cancel{l_0, \ldots, l_n}\}$$

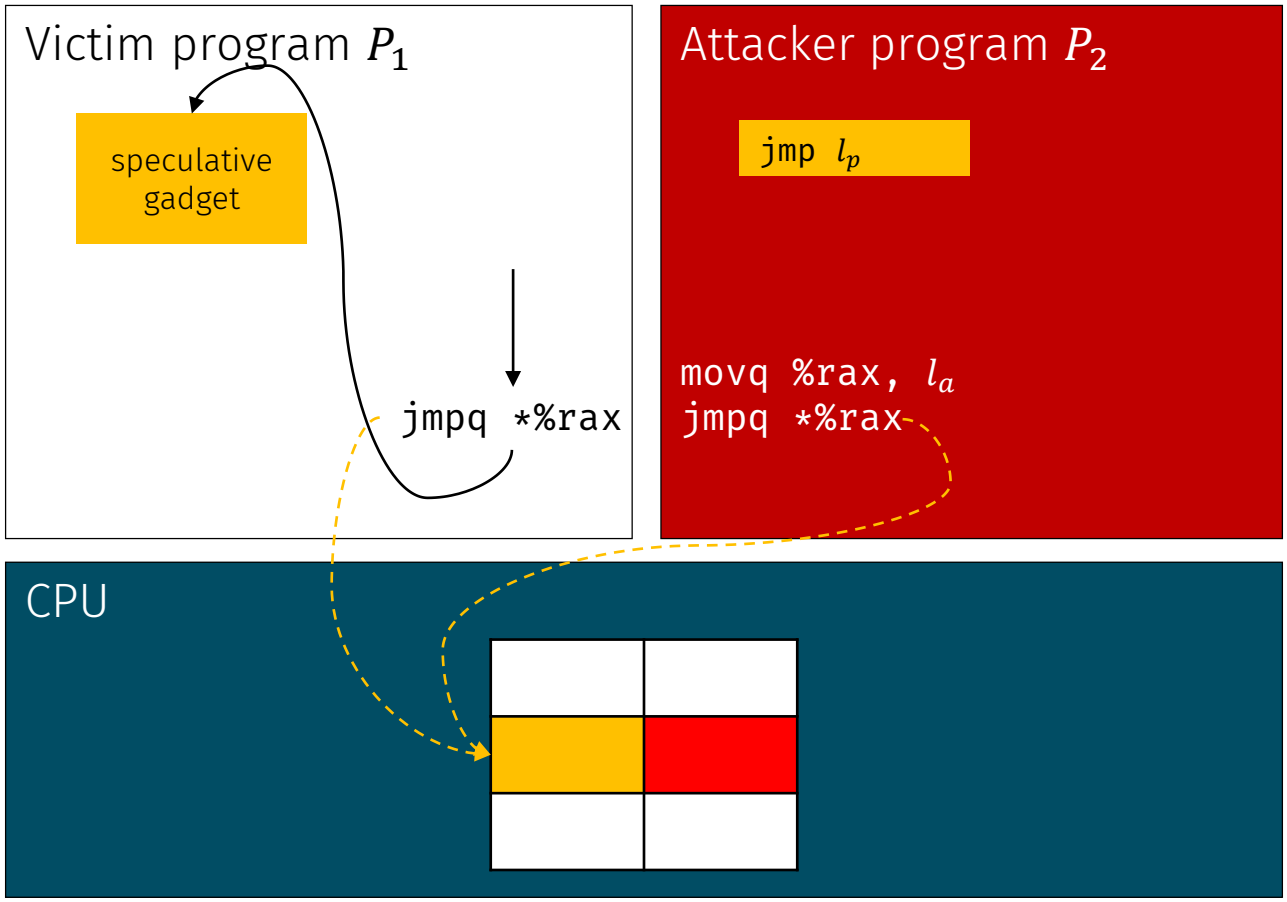
Spectre V2: Branch Target Injection/Poisoning



basically, overloading BTB:

$$\text{BTB}[\texttt{rip}] = \{l_a, l_a, l_a, l_a, \ldots\}$$

Spectre V2: Branch Target Injection/Poisoning



basically, overloading BTB:

$$\text{BTB}[\texttt{rip}] = \{l_a, l_a, l_a, l_a, \ldots\}$$

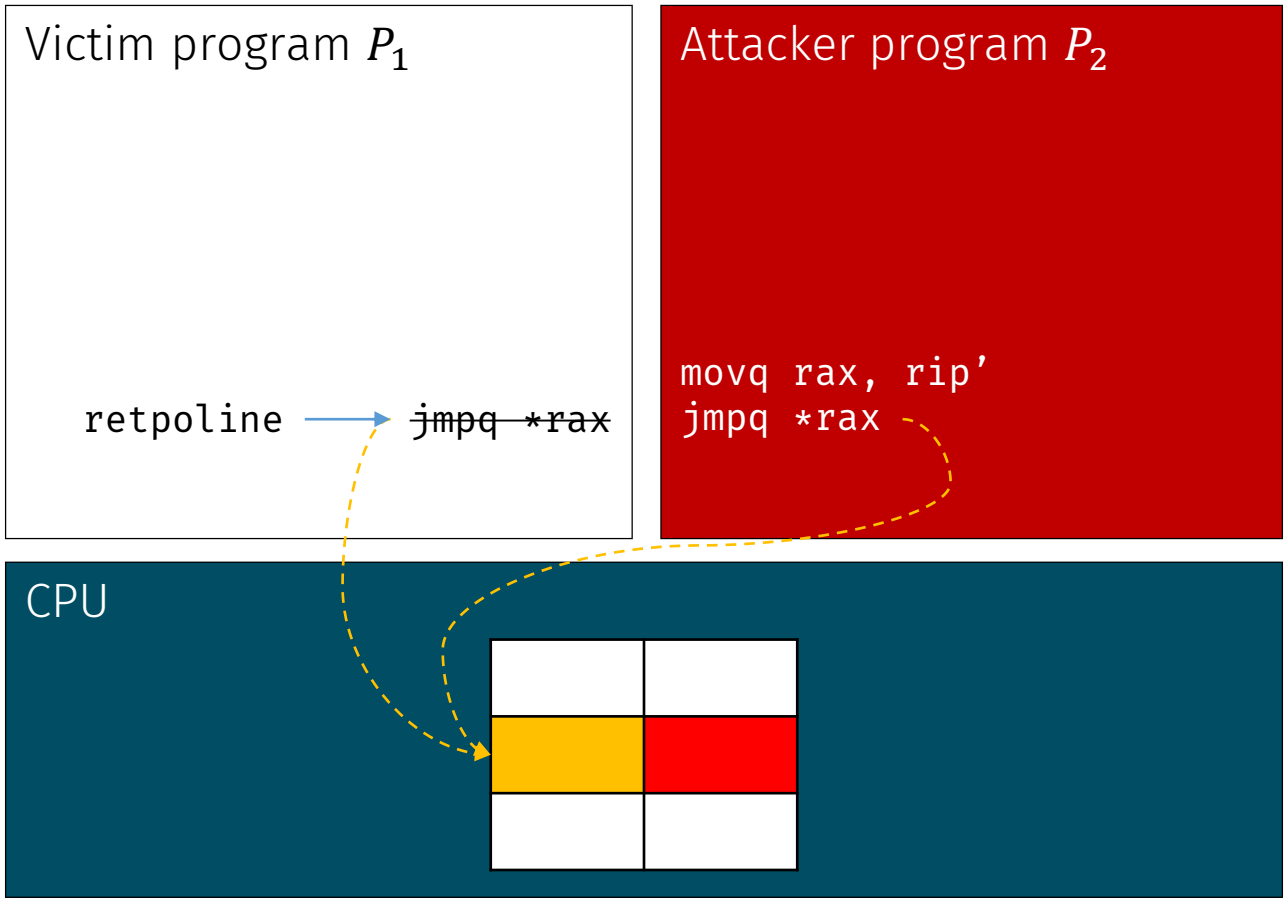
Spectre V2 defenses

- hardware level:
  - isolation in hardware for BTBs
  - DAWG from MIT, ISCA'18
- software level:
  - retpoline: *break speculation!*
  - software diversity

Spectre V2: retpoline

- mangled version of return trampoline
- replace affected sequences with code sequence that breaks speculation

Spectre V2: Branch Target Injection/Poisoning



basically, overloading BTAC:

$$\text{BTAC}[\texttt{rip}] = \{l_a, l_a, l_a, l_a, \ldots\}$$

retpoline

original code

```
…

jmpq *%rax
```

secured retpoline code

```
…
1:     call load
  capture:
2:     pause; lfence
3:     jmp capture
  load:
4:     mov %rax, (%rsp)
5:     ret
```

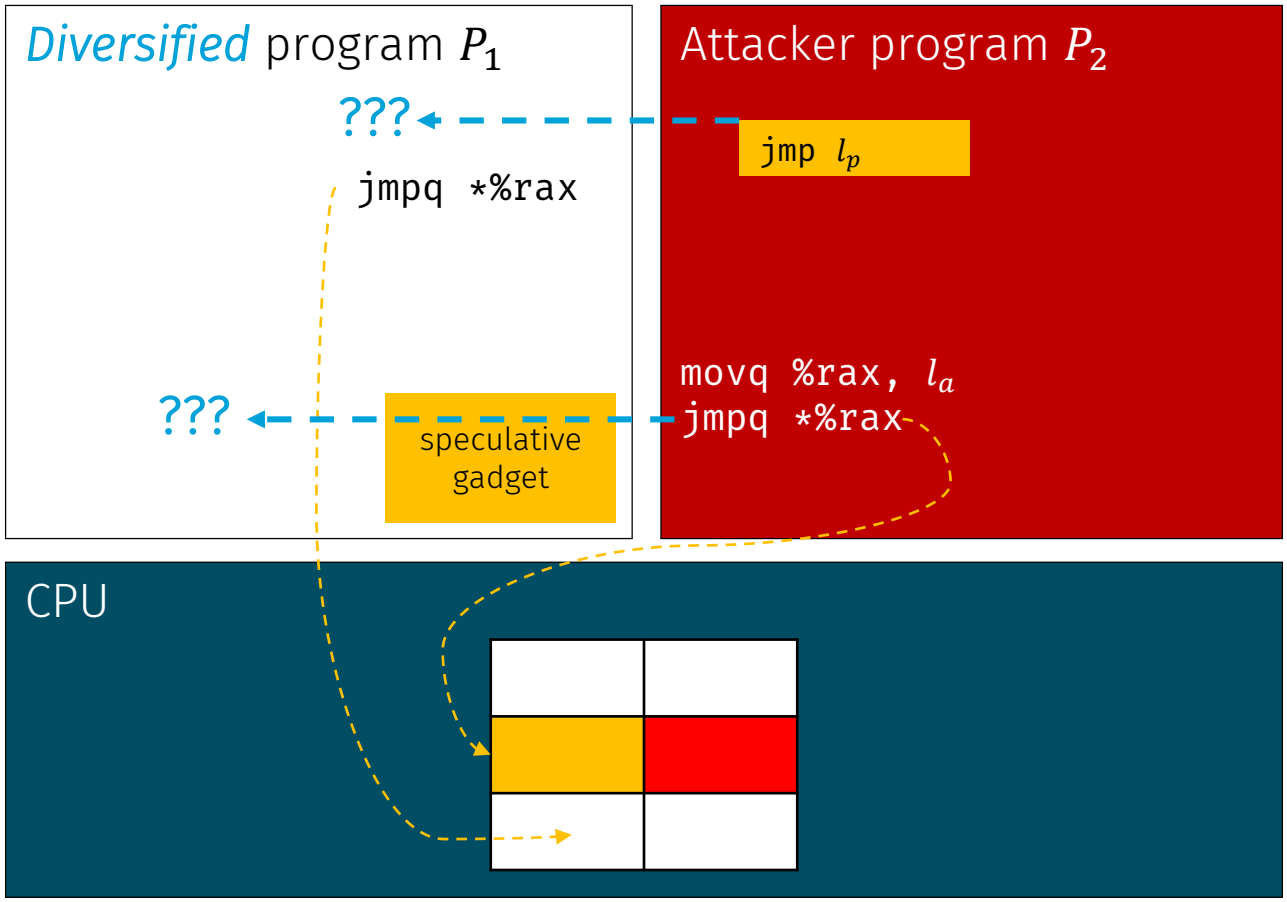
Managing speculation with fences

- `lfence`
  - serializes load operations
  - all load-from-memory operations prior to `lfence` instruction become globally visible
- `mfence`
  - serializes load *and* store instructions
  - all load-from-memory and store-to-memory instructions prior to `mfence` become globally visible
- `pause`
  - indicates a spin-wait loop to the processor, helps it to avoid memory order violation

Fencing popular with compilers

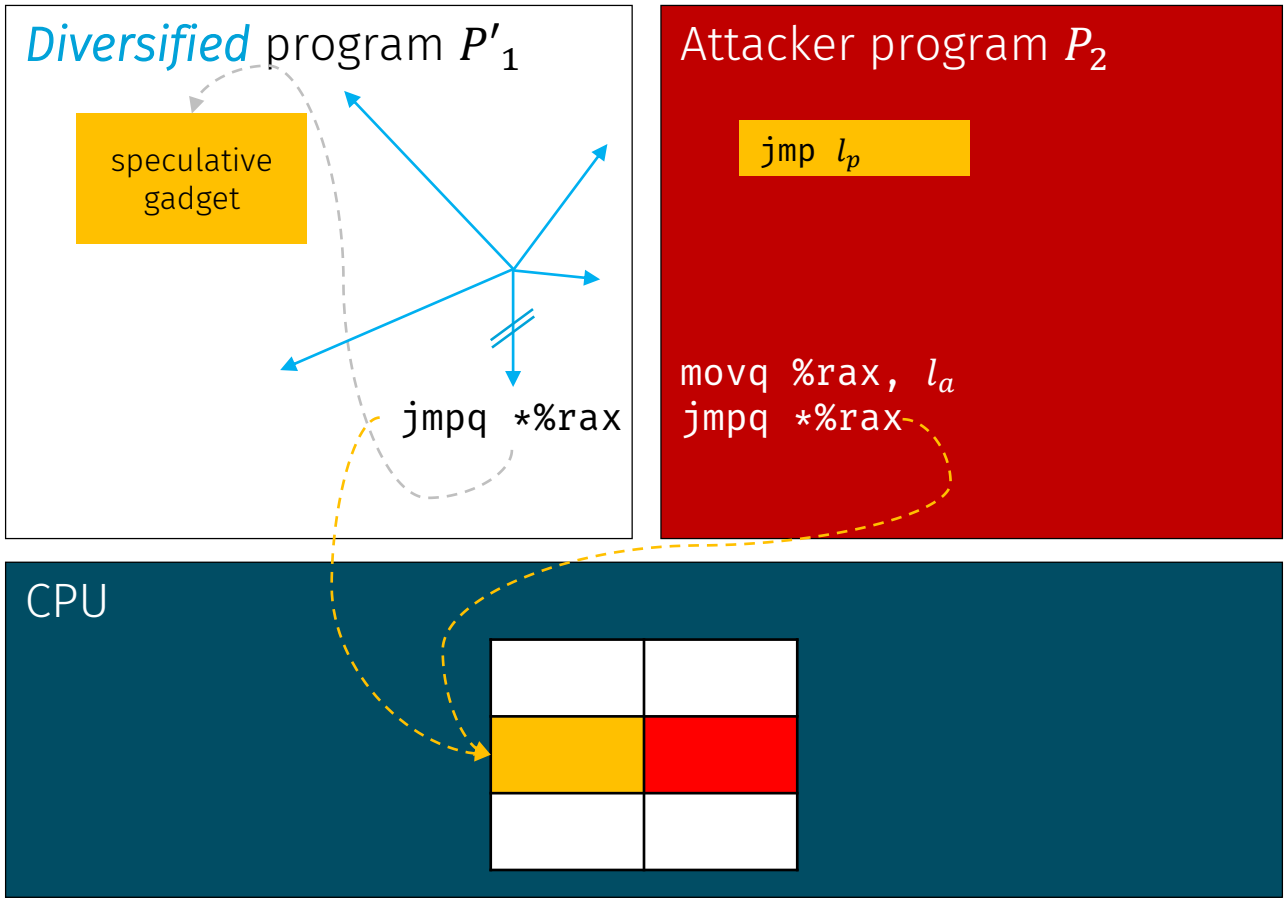
- Visual Studion C/C++ compiler
  - /Qspectre
- LLVM (+ retpolines)
- JIT Compilers
  - V8 (+ retpolines)
  - JSC/WebKit/Safari

Spectre V2 vs. Software Diversity, pt. 1



overloading will ***likely*** not mask intended instruction in the victim program!

Spectre V2 vs. Software Diversity, pt. 2



Use ***dynamic diversity*** to break assumption of single control-flow transfer!

(see NDSS'15: "Thwarting Timing-Based Cache Side Channel Attacks")

Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

```
                ⋮
mov   [x], al                 store contents of al in x
movzx r8,  byte [y]           load byte from y into r8
shl   r8,  byte 0xc           shift contents of r8 by 12 bits
mov   eax, [rdx+r8]           load contents into eax
                ⋮
```

dst *src*

Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

```
                ⋮
mov   [x], al
movzx r8,  byte [y]
shl   r8,  byte 0xc
mov   eax, [rdx+r8]
           ⋮
```

dst *src*

store contents of `al` in x

speculatively load y, iff $x \neq y$

Q: Why?

Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

```
                ⋮
mov    [x], al
movzx  r8,  byte [y]
shl    r8,  byte 0xc
mov    eax, [rdx+r8]
                ⋮
```

dst *src*

store contents of `al` in x

speculatively load y, iff $x \neq y$

Q: Why?

A: Would be a pipeline hazard, viz. read-after write (RAW)

Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

```
                ⋮
mov   [x], al
movzx r8,  byte [y]
shl   r8,  byte 0xc
mov   eax, [rdx+r8]
           ⋮
```

dst *src*

store contents of `al` in x

speculatively load y, iff $x \neq y$

Q: Why?

A: Would be a pipeline hazard, viz. read-after write (RAW)

→ Hardware, therefore, needs way to disambiguate memory references to speculate on load instruction

Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

```
        ⋮
mov   [x], al
movzx r8,  byte [x]
shl   r8,  byte 0xc
mov   eax, [rdx+r8]
        ⋮
```

dst *src*

store contents of `al` in x

speculatively load y, iff $x \neq y$

Q: Why?

A: Would be a pipeline hazard, viz. read-after write (RAW)

→ Hardware, therefore, needs way to disambiguate memory references to speculate on load instruction

→ attack fools this HW and reads stale value

Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

```
                ⋮
mov    [x], al
movzx  r8,  byte [y]
shl    r8,  byte 0xc
mov    eax, [rdx+r8]
                ⋮
```

dst *src*

Q: How to fool memory disambiguation?

A: Break heuristics implemented therein.

For example:

$x \coloneqq$ `[rdi+rcx]`

$y \coloneqq$ `[rsi+rcx]`

where, `rdi` = `rsi`

Why does *that* work?

- syntactically different
- unlikely to be identical in real-world programs
- = good heuristic

Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

```
                ⋮
mov   [x], al
movzx r8,  byte [x]
shl   r8,  byte 0xc
mov   eax, [rdx+r8]
           ⋮
```

dst *src*

store into x

load from x, speculatively

→ if store takes longer, then load will be executed *before*

→ enables the attacker to read a value that he wouldn't normally have access to

Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

```
                ⋮
mov    [x], al
movzx  r8,  byte [x]
shl    r8,  byte 0xc
mov    eax, [rdx+r8]
       ↑       ⋮  ↑
      dst         src
```

What about these?

Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

```
                ⋮
mov   [x], al
movzx r8,  byte [x]
shl   r8,  byte 0xc
mov   eax, [rdx+r8]
                ⋮
```

dst *src*

- eventually, mis-speculation will be detected, and values thrown away
- attackers need a subterfuge to succeed
- idea: use a traditional cache side-channel

Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

```
                ⋮
mov   [x], al
movzx r8,  byte [x]
shl   r8,  byte 0xc
mov   eax, [rdx+r8]
                ⋮
```

dst *src*

shift “stolen” value in `r8` by 12 bits

load a data-reference depending upon `r8` into `eax`

- address will be loaded in data cache
- subsequent read will be much faster
- detectable afterward by another process

Spectre V4 defenses

- primarily focused on placing `lfence`s in the code
- `retpoline`s not directly applicable, because not per se reliant on control-flow
- diversity offers a potent alternative

V4 vs. Software Diversity

Attack anatomy & assumptions:

- exact location of target data known
- data-type specifics known
- being able to fool disambiguation
- classical side-channel subterfuge

V4 vs. Software Diversity

Attack anatomy & assumptions:

- exact location of target data known
- data-type specifics known
- being able to fool disambiguation ← for HW people
- classical side-channel subterfuge

V4 vs. Software Diversity

Attack anatomy & assumptions:

- exact location of target data known
- data-type specifics known
- being able to fool disambiguation ← for HW people
- classical side-channel subterfuge

→ *can be diversified!*

Other side channel attacks mitigated by diversity

- AnC
  - trivially mitigated, because single address leak is insufficient (not ASLR!)
  - page table walk timing
- TLBleed
  - trains an SVM to recognize TLB walks
  - recognizes specific pattern through SVM
  - in a diversified ecosystem, single SVM most likely insufficient, since many more patterns can occur
    - worst possible case, SVM does not recognize all TLB walks

Spectre Summary

- V1 does not even need diversity ;)
- V2 diversity offers protective quality *for free*
  - requires that the attackers launch a multi-stage attack, with an information disclosure/memory leak to relocate attack:
    1. leak target program binary (analysis)
    2. regenerate attack binary (compile)
    3. launch attack
- V3 has kernel patches (KPTI, aka. KAISER)
- V4 less impact through software diversity

A bird's eye view of Spectre

- during speculative execution, some things are not checked (privileges, bounds, etc.)
- use this "magic" power and write speculatively read data to the cache
- use classical side-channel attack to read confidential data *after* speculation
  - EVICT+TIME, PRIME+PROBE, etc.

Thwarting Spectre – in general

- during speculative execution, some things are not checked (privileges, bounds, etc.)
- use this “magic” power and write speculatively read data to the cache
- use classical side-channel attack to read confidential data *after* speculation
  - EVICT+TIME, PRIME+PROBE, etc.

- fix things such that it can’t be speculated on (SW)
- fix HW to do speculate “properly”
- prevent such attacks, either HW or SW

➡ *may not be possible in general!*

Nothing is stronger than an idea whose time has come.

-Victor Hugo

Questions?

brunthaler@unibw.de