Spectre Through the Looking Glass

Stefan Brunthaler

S P E C T R E  •  C O D E  •  B U M

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

Desktop computer operating systems

- Win7 + Win10 ≈ 88%
- 100m+ machines
Software Monoculture

Web Browsers

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)

Multi Compiler

diversified code *same* semantics
**Timing Channel in AES**

**EVICT+TIME** AES Cache-based Side Channel Attack:

1. Choose $p$
2. Encrypt $k$, $p$
3. Evict cache set $c$
4. $t_0$ to $t_1$
5. Derive $k$

**Shared Cache**

- Set 1
- Set 2
- Set 3

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

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
  • 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
  • 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

```plaintext
a1: array \( N_1 \);
a2: array \( N_2 \);

if (i < \( N_1 \)) {
    y = a2[a1[i] \times 256];
} else {
    cmp i, \( N_1 \)
    jge <post>
}
```
Spectre V1: Bounds-Check Bypass

\begin{align*}
a1 & : \text{array } N_1; \\
a2 & : \text{array } N_2; \\
\vdots & \text{ array } \vdots; \\
\text{if } (i < N_1) \{ \\
\quad y & = a2[a1[i] * 256]; \\
\}
\end{align*}

\hspace{1cm} \text{cmp } i, N_1 \quad \text{jge } \text{<post>}

\text{predicted as either}
\begin{itemize}
  \item taken
  \item not-taken
\end{itemize}
Spectre V1: Bounds-Check Bypass

\[ \text{a1: array } N_1; \]
\[ \text{a2: array } N_2; \]
\[ \vdots \]
\[ \text{if } (i < N_1) \{ \]
\[ \quad y = a2[a1[i] \times 256]; \]
\[ \} \]

\[ \text{cmp } i, N_1 \]
\[ \text{jge } <\text{post}> \]

predicted as either
- taken
- not-taken

→ allows us to “widen” window of speculative execution
Spectre V1: Bounds-Check Bypass

\[ \begin{align*} 
\text{a1: array } N_1; \\
\text{a2: array } N_2; \\
\vdots \\
\text{if } (i < N_1) \{ \\
\quad y = a2[a1[i] \times 256]; \\
\} \\
\end{align*} \]

\( \Rightarrow \) “will not be executed”

\( \Rightarrow \) \( i \geq N_1 \)

can read arbitrary memory during speculative execution
```java
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 $N_1$

a2: array $N_2$

: 

if ($i < N_1$) {
    
y= a2[a1[i]*256];

} 

- idea is to force execution of a bounds-check operation
Spectre V1: Bounds-Check Bypass

```
a1: array \( N_1 \);
a2: array \( N_2 \);
:
if (i < \( N_1 \)) {
    i = i \% \( N_1 \);
    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];
...

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];
    ...
}

Use compiler optimization knowledge to our advantage!

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

unrolled twice

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

\[
\text{\textless \textit{rip} \ text{jmpq} \ *\%rax}
\]

- leverages branch target prediction to improve speculation
- uses \texttt{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

\[
\text{\texttt{<rip> jmpq *%rax}}
\]

- \(\text{BTB[rip]} = \{l_0, ..., l_n\}\)
- contains multiple labels \(l\)
- selects one among those (using some heuristic)
- cannot possibly hold all potential labels \(\rightarrow\) 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

Victim program $P_1$

speculative gadget

jmpq *%rax

Attacker program $P_2$

jmp $l_p$

movq %rax, $l_a$

jmpq *%rax

CPU

basically, overloading BTB:

$$\text{BTB}[\text{rip}] = \{l_0, \ldots, l_n\}$$
Spectre V2: Branch Target Injection/Poisoning

Victim program $P_1$
- speculative gadget

Attacker program $P_2$
- `jmp l_p`
- `movq %rax, l_a`
- `jmpq *%rax`

basically, overloading BTB:

$\text{BTB}[\text{rip}] = \{l_a, l_a, l_a, l_a, \ldots\}$
Spectre V2: Branch Target Injection/Poisoning

Victim program $P_1$

speculative gadget

jmpq *%rax

Attacker program $P_2$

jmp $l_p$

movq %rax, $l_a$

jmpq *%rax

basically, overloading BTB:

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

Victim program $P_1$

- `retpoline` → `jmpq *rax`

Attacker program $P_2$

- `movq rax, rip'`
- `jmpq *rax`

Basically, overloading BTAC:

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

Diversified program $P_1$

\[ \text{???} \]
\[ \text{jmpq} \ast \%rax \]

Attacker program $P_2$

\[ \text{jmp} \ l_p \]
\[ \text{movq} \ %rax, \ l_a \]
\[ \text{jmpq} \ \ast \%rax \]

overloading will \textit{likely} not mask intended instruction in the victim program!
Use *dynamic diversity* to break assumption of single control-flow transfer!
(see NDSS’15: “Thwarting Timing-Based Cache Side Channel Attacks”)

**Spectre V2 vs. Software Diversity, pt. 2**

**Diversified** program $P'_1$

- speculative gadget
- `jmpq *%rax`

**Attacker program $P_2$**

- `jmp \ell_p`
- `movq %rax, l_a`
- `jmpq *%rax`

**CPU**
Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

\[
\begin{align*}
&\text{mov} \ [x], \text{al} \quad \text{store contents of al in } x \\
&\text{movzx} \ r8, \text{ byte } [y] \quad \text{load byte from } y \text{ into } r8 \\
&\text{shl} \ r8, \text{ byte } 0xc \quad \text{shift contents of } r8 \text{ by 12 bits} \\
&\text{mov} \ eax, [rdx+r8] \quad \text{load contents into eax} \\
\end{align*}
\]
Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

\[
\begin{align*}
\text{mov} & \quad [x], \text{al} & \quad \text{store contents of al in } x \\
\text{movzx} & \quad r8, \text{ byte } [y] & \quad \text{speculatively load } y, \text{ iff } x \neq y \\
\text{shl} & \quad r8, \text{ byte } 0xc \\
\text{mov} & \quad \text{eax}, [rdx+r8] \\
\end{align*}
\]

Q: Why?
mov [x], al  
movzx r8, byte [y]  
shl r8, byte 0xc  
mov eax, [rdx+r8]

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

store contents of al in x
speculatively load y, iff x ≠ y
Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

: 

\[
\begin{align*}
\text{mov} & \quad [x], \text{al} \\
\text{movzx} & \quad \text{r8}, \quad \text{byte} \ [y] \\
\text{shl} & \quad \text{r8}, \quad \text{byte} \ 0xc \\
\text{mov} & \quad \text{eax}, \ [\text{rdx+r8}] \\
\end{align*}
\]

: 

\[
\begin{align*}
\text{dst} & \quad \text{src} \\
\end{align*}
\]

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)

\( \rightarrow \) Hardware, therefore, needs way to disambiguate memory references to speculate on load instruction
**Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics**

```
: mov [x], al                      : store contents of al in x
movzx r8, byte [x]                : speculatively load y, iff x ≠ y
shl r8, byte 0xc
mov eax, [rdx+r8]
```

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
Q: How to fool memory disambiguation?

A: Break heuristics implemented therein. For example:

\[ x := [\text{rdi}+\text{rcx}] \]

\[ y := [\text{rsi}+\text{rcx}] \]

where, \( \text{rdi} = \text{rsi} \)

Why does \textit{that} work?

- syntactically different
- unlikely to be identical in real-world programs
- = good heuristic
Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

: 

\[
\text{mov } [x], \text{ al} \quad \text{store into } x
\]

\[
\text{movzx r8, byte } [x] \quad \text{load from } x, \text{ speculatively}
\]

\[
\text{shl r8, byte } 0xc \quad \rightarrow \text{if store takes longer, then}
\]

\[
\text{mov eax, } [\text{rdx+r8}] \quad \text{load will be executed } \textbf{before}
\]

\[
\rightarrow \text{enables the attacker to read a value that he wouldn’t}
\]

\[
\text{normally have access to }
\]
Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

: 

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

: 

\[ \text{dst} \quad \text{src} \]

\{ What about these? \}
Spectre V4: Speculative Store Bypass – Deconstructing its Mechanics

- 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

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

:  

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 lfences in the code

• retpolines 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
• 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
• 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