Spectre Attacks: Exploiting Speculative Execution

David Bimmler
May 2, 2019
Agenda

Executive Summary
Background
Novelty
Key Approach and Ideas
Mechanisms (in some detail)
Key Results: Methodology and Evaluation
Summary
Strengths
Weaknesses
Thoughts and Ideas
Discussion
Executive Summary
Executive Summary

- Spectre is a security vulnerability violating memory isolation

Executive Summary

- Spectre is a security vulnerability violating memory isolation
- It abuses speculative execution to execute instructions which should never be executed
• Spectre is a security vulnerability violating memory isolation
• It abuses speculative execution to execute instructions which should never be executed
• It uses side-channels to leak microarchitectural state changed by erroneously executed instructions
Background
The *instruction set architecture* (ISA) is the contract between hardware and software.

A *microarchitecture* (µarch) is an implementation of an ISA in a given processor.
Background: Direct and Indirect Branches

<table>
<thead>
<tr>
<th>Direct branch</th>
<th>Indirect branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP 0x89AB</td>
<td>CALL EAX</td>
</tr>
<tr>
<td>JNE 0x90AB</td>
<td>JMP EAX</td>
</tr>
<tr>
<td>... many more</td>
<td>RET</td>
</tr>
</tbody>
</table>
• Superscalar processors predict branch outcomes
Background: Branch Prediction

- Superscalar processors predict branch outcomes
- Direction of direct branches (taken/not taken)
  - cached by Pattern History Table (PHT)/Branch History Buffer (BHB)
Background: Branch Prediction

- Superscalar processors predict branch outcomes
- Direction of direct branches (taken/not taken)
  - cached by Pattern History Table (PHT)/Branch History Buffer (BHB)
- Target address of indirect branches
  - cached by the Branch Target Buffer (BTB)
  - Return Stack Buffer (RSB) for CALL/RET pairs
Background: Speculative Execution

- Predicted path is executed speculatively
  - Processor keeps track of what is being executed speculatively
  - Prediction incorrect: discard effects
Background: Speculative Execution

- Predicted path is executed speculatively
  - Processor keeps track of what is being executed speculatively
  - Prediction incorrect: discard effects
- Instructions executed due to misprediction called transient instructions
Background: Microarchitectural Side-Channels

- μarch is stateful (e.g. PHT, BTB, RSB, caches, ...)

Yarom and Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack"
Background: Microarchitectural Side-Channels

- μarch is stateful (e.g. PHT, BTB, RSB, caches, ...)
- State shared between processes
Background: Microarchitectural Side-Channels

- µarch is stateful (e.g. PHT, BTB, RSB, caches, ...)
- State shared between processes
- Information leaks called *side-channels*
Background: Microarchitectural Side-Channels

- µarch is stateful (e.g. PHT, BTB, RSB, caches, ...)
- State shared between processes
- Information leaks called side-channels
- Example: Flush+Reload\(^1\) — a cache side-channel

\(^1\)Yarom and Falkner, “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”
Flush+Reload: Attack

- Flush+Reload can monitor access of memory lines in shared pages
Flush+Reload: Attack

- Flush+Reload can monitor access of memory lines in shared pages
- Access to monitored memory is fast if victim has accessed
Flush+Reload: Example

- **victim process**
  - core 0

- **evil process**
  - core 1

- **cache**

- **physical memory**
Flush+Reload: Example

- Victim process
  - Core 0

- Evil process
  - Core 1

- Cache

- libx.so
Flush+Reload: Example

- Victim process
- Evil process

- Core 0
- Core 1

- Cache

- Libx.so
Flush+Reload: Example

- Victim process
  - Core 0
- Evil process
  - Core 1

Cache

Libx

Physical memory
Flush+Reload: Example

- **victim process**
  - core 0
- **evil process**
  - core 1

![Diagram showing cache access by victim and evil processes](image)

- 'clflush'
- 'libx.so'
- 'physical memory'
Flush+Reload: Example

- Victim process
  - Core 0
- Evil process
  - Core 1
- Cache
- Libx.so
- Pseudo memory
Flush+Reload: Example

- Victim process
- Core 0
- Evil process
- Core 1
- Cache
- Libx
- Physical memory
Flush+Reload: Example

- Victim process
- Core 0
- Evil process
- Core 1
- Cache
- Libx.so
Novelty
Novelty

Spectre Attack

- Trick victim into speculatively performing operations which would not occur during correct program execution
Novelty

Spectre Attack

- Trick victim into speculatively performing operations which would not occur during correct program execution
- Leak sensitive information through microarchitectural side channel
Key Approach and Ideas
The following code constitutes a Spectre gadget, vulnerable when `unsigned int x` is attacker controlled

```c
if (x < array1_size)
    y = array2[array1[x] * 512];
```
Exploiting Conditional Branches

How is it vulnerable?

```java
if (x < array1_size)
    y = array2[array1[x] * 512];
```

Speculative execution if `array1_size` is not available
Exploiting Conditional Branches

How is it vulnerable?

```c
if (x < array1_size)
    y = array2[array1[x] * 512];

Speculative out of bounds read for a malicious x
```
How is it vulnerable?

```c
if (x < array1_size)
    y = array2[array1[x] * 512];
```

Encode value in µarch state using cache side-channel
Exploiting Conditional Branches

if x in bounds
  true
  prediction
    true
    fast
    slow
  false
    Spectre
    fast
  false
  false
Mechanisms (in some detail)
unsigned int array1_size = 16;
uint8_t array1[16] = {1, 2, ..., 15, 16};
uint8_t array2[256 * 512];
char *secret = "Squeamish Ossifrage";

void victim_function(size_t x) {
    if (x < array1_size) {
        y = array2[array1[x] * 512];
    }
}
A generic Spectre attack consists of three phases

1. Setup
2. Transient Execution
3. Data Exfiltration
Prepare exfiltration side-channel

1 /* Flush array2[(0..255)*512] from cache */
2 for (i = 0; i < 256; i++)
3   _mm_clflush(&array2[i * 512]);
Induce speculative execution by flushing `array1_size`

```c
1 for (j = 5; j >= 0; j--) {
2    _mm_clflush(&array1_size);
3    victim_function(training_x);
4 }
```
Train branch prediction to take branch using valid values for \( x \)

```c
for (j = 5; j >= 0; j--)
{
    _mm_clflush(&array1_size);
    victim_function(training_x);
}
```
A generic Spectre attack consists of three phases

1. Setup
2. Transient Execution
3. Data Exfiltration
Spectre Attack: Transient Execution Phase

Execute gadget with malicious $x$ results in a speculative out of bounds read

1. `_mm_clflush(&array1_size);`
2. `victim_function(malicious_x);`
Result of malicious read encoded in probe array

1. if (x < array1_size)
2. y = array2[array1[x] * 512];
A generic Spectre attack consists of three phases

1. Setup
2. Transient Execution
3. Data Exfiltration
Exfiltrate using Flush+Reload

```
for (i = 0; i < 256; i++) {
    addr = &array2[i * 512];
    time1 = __rdtscp(&junk);
    junk = *addr;
    time2 = __rdtscp(&junk) - time1;  // compute access time
    if (time2 <= CACHE_HIT_THRESHOLD)
        printf("found: %#x\n", i);
}
```
Variant 2: Poisoning Indirect Branches

- Destination address of indirect branch may be unknown
Variant 2: Poisoning Indirect Branches

- Destination address of indirect branch may be unknown
- Speculative execution at predicted target address
Variant 2: Poisoning Indirect Branches

• Destination address of indirect branch may be unknown
• Speculative execution at predicted target address
• Attack: mistrain branch target buffer in attacker controlled context
Variant 2: Poisoning Indirect Branches

- Destination address of indirect branch may be unknown
- Speculative execution at predicted target address
- Attack: mistrain branch target buffer in attacker controlled context
- Speculatively execute Spectre gadget for observable side effects
Mistraining Branch Predictors

• Attacker mimics pattern of branches in its own context
Mistraining Branch Predictors

- Attacker mimics pattern of branches in its own context
- Attacker-chosen target predicted in victim
Mistraining Branch Predictors

- Attacker mimics pattern of branches in its own context
- Attacker-chosen target predicted in victim
- Highly \( \mu \)-arch-specific — reverse-engineering necessary
Key Results: Methodology and Evaluation
The paper presents multiple exploits:

1. Variant 1 proof of concept in native code
2. Variant 1 attacks in JavaScript and eBPF
3. Variant 2 proof of concept in native code
4. Variant 2 attack to leak host memory from within a KVM VM
Evaluation

- Spectre works
Evaluation

- Spectre works
- Quite a few µarchs tested
Evaluation

• Spectre works
• Quite a few µarchs tested
  • Intel Ivy Bridge, Broadwell, Haswell, Sky Lake, Kaby Lake, AMD Ryzen, ...
• Spectre works
• Quite a few µarchs tested
  • Intel Ivy Bridge, Broadwell, Haswell, Sky Lake, Kaby Lake, AMD Ryzen, ...
• Bandwidth
Evaluation

• Spectre works
• Quite a few µarchs tested
  • Intel Ivy Bridge, Broadwell, Haswell, Sky Lake, Kaby Lake, AMD Ryzen, ...
• Bandwidth
• Error rate
<table>
<thead>
<tr>
<th>native PoC var. 1</th>
<th>Bandwidth</th>
<th>Error rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript var. 1</td>
<td>~10 kB/s</td>
<td>&lt; 0.01%</td>
</tr>
<tr>
<td>eBPF var. 1</td>
<td>2 kB/s to 5 kB/s</td>
<td>—</td>
</tr>
<tr>
<td>native PoC var. 2</td>
<td>0.041 kB/s</td>
<td>—</td>
</tr>
<tr>
<td>KVM var. 2</td>
<td>~1.8 kB/s</td>
<td>1.7%</td>
</tr>
</tbody>
</table>
Summary
• Transient instructions can violate security
• Transient instructions can violate security
  • in correct programs
• Transient instructions can violate security
  • in correct programs
• Through microarchitectural side-channels we can observe the effects
Summary

- Transient instructions can violate security
  - in correct programs
- Through microarchitectural side-channels we can observe the effects
- Multiple variants to cause misprediction
Strengths
Gigantic impact
Strengths

- Gigantic impact
- Complete mitigation in software seemingly impossible\(^2\)

\(^2\)McIlroy, Sevcík, Tebbi, Titzer, and Verwaest, “Spectre is here to stay: An analysis of side-channels and speculative execution”
• Gigantic impact
• Complete mitigation in software seemingly impossible\(^2\)
• Generality of attack

\(^2\)McIlroy, Sevcík, Tebbi, Titzer, and Verwaest, “Spectre is here to stay: An analysis of side-channels and speculative execution”
Strengths

- Gigantic impact
- Complete mitigation in software seemingly impossible\(^2\)
- Generality of attack
- *Many* papers discussing the attack

\(^2\)McIlroy, Sevcík, Tebbi, Titzer, and Verwaest, “Spectre is here to stay: An analysis of side-channels and speculative execution”
Weaknesses
Weaknesses

- μarch attack: finnicky, brittle
Weaknesses

- μarch attack: finnicky, brittle
- Local execution required

Schwarz, Schwarzl, Lipp, and Gruss, "NetSpectre: Read Arbitrary Memory over Network"
Weaknesses

• \(\mu\)arch attack: finnicky, brittle
• Local execution required
  • But: NetSpectre\(^3\)

\(^3\)Schwarz, Schwarzl, Lipp, and Gruss, “NetSpectre: Read Arbitrary Memory over Network”
Weaknesses

- \( \mu \)arch attack: finnicky, brittle
- Local execution required
  - But: NetSpectre\(^3\)
- Spotty evaluation

\(^3\)Schwarz, Schwarzl, Lipp, and Gruss, “NetSpectre: Read Arbitrary Memory over Network”
Weaknesses

- µarch attack: finnicky, brittle
- Local execution required
  - But: NetSpectre\(^3\)
- Spotty evaluation
- Generality of attack not explained well enough

\(^3\)Schwarz, Schwarzl, Lipp, and Gruss, “NetSpectre: Read Arbitrary Memory over Network”
Thoughts and Ideas
Performance versus Security

- Fundamental tradeoff: performance versus security
Performance versus Security

- Fundamental tradeoff: performance versus security
- Mitigations are stop gaps: ISA might have to change
Discussion
Discussion: Spectre Variations

1 if (x < array1_size) {
2     y = array1[x];
3     // do something using y that is observable
4     // when speculatively executed
5 }

Can you think of more observable effects?
Thank You
References


Backup
Background: Virtual Memory

- memory isolation between different processes
- typically provided by hardware via MMU
- page tables translate virtual to physical addresses
Meltdown\textsuperscript{4} is not Spectre

- also violates memory isolation
- exploits out of order execution
- privilege escalation - reads kernel memory
- race condition specific to Intel processors
- mitigated by the KAISER patches

\textsuperscript{4}Lipp, Schwarz, Gruss, Prescher, Haas, Fogh, Horn, Mangard, Kocher, Genkin, Yarom, and Hamburg, “Meltdown: Reading Kernel Memory from User Space”.
Background: Out-of-Order Execution

- performance optimisation for pipelined processors
- instructions executed out of order
- but retired (i.e. become visible) in order
- complex data dependency logic in hardware
Flush+Reload: Background

- Identical memory pages are shared between processes
  - e.g. for shared libraries
• Identical memory pages are shared between processes
  • e.g. for shared libraries
• Shared pages imply identical physical addresses
• Identical memory pages are shared between processes
  • e.g. for shared libraries
• Shared pages imply identical physical addresses
• L3 cache is physically tagged
Variant 2: Spectre Gadget

Attacker-controlled `ebx` and `edi` allows reading memory

1. `adc edi,dword ptr [ebx+edx+13BE13BDh]`
2. `adc dl,byte ptr [edi]`

Set `edi` to address of probe array (e.g. in shared library)

Set `ebx` to `m - 0x13BE13BD - edx`