Definitions

- **Microarchitecture**: The implementation of an instruction set, including things like ALUs, caches, branch predictors, &c.

- **Spectre**: Attack which tricks the processor into speculatively executing instructions which will later be discarded, leading to observable microarchitecture changes.

Figure 1: Nowadays all security vulnerabilities need a mascot.
## Meltdown vs. Spectre

<table>
<thead>
<tr>
<th>Property</th>
<th>Meltdown</th>
<th>Spectre</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Mostly Intel</td>
<td>Mostly Everyone</td>
</tr>
<tr>
<td>Prerequisite</td>
<td>Limited code execution</td>
<td>Limited code execution</td>
</tr>
<tr>
<td>Cause</td>
<td>O3 execution</td>
<td>O3+Speculative execution</td>
</tr>
<tr>
<td>Impact</td>
<td>Physical memory read</td>
<td>Variant-dependent</td>
</tr>
<tr>
<td>Fix</td>
<td>Software patches</td>
<td>Harder software patches</td>
</tr>
</tbody>
</table>
if (condition) do_foo();
else do_bar();

- Branches make pipelining slow! Can’t execute instructions until branch destination is known
- Idea: predict output of the branch based on history
- If predicted branch was correct: retire speculated instructions, got a head start, changes become architecturally visible
- If predicted branch was wrong: discard speculated instructions, lose performance, changes never become architecturally visible
- We can use (a truncated) virtual address of the branch as an index for a history table
  - Actually more complicated: hashing and global state is involved... but let’s keep it simple
Variant 1: Conditional Branch Misprediction

Consider the following code:

```c
if (idx1 < array1_size) {
    size_t idx2 = array1[idx1] * 4096;
    y = array2[idx2];
}
```

- Clearly this code prevents reading out-of-bounds of `array1`.
- We will see what happens if `idx1 = array1_size` and speculative execution occurs.
With Speculative Execution

```c
if (idx1 < array1_size) {
    size_t idx2 = array1[idx1] * 4096;
    y = array2[idx2];
}
```

1. We have \( \text{idx1} = \text{array1\_size} \)
2. Processor cannot immediately check \( \text{idx1} < \text{array1\_size} \),
   \( \implies \) speculative execution
3. Processor branch predicts that \( \text{idx1} < \text{array1\_size} \)
4. Processor computes \( \text{idx2} = \text{array1[\text{idx1}] \times 4096} \)
5. Processor gets cache miss on memory access \( \text{array2[\text{idx2}]} \),
   queries memory and adds it to cache
6. Processor realizes speculative execution was wrong, squashes instructions
7. We use memory access time to figure out the value of \( \text{idx2} \)
   \( \implies \text{array1[\text{idx1}]} \)
So many questions

```c
if (idx1 < array1_size) {
    size_t idx2 = array1[idx1] * 4096;
    y = array2[idx2];
}
```

▶ How do we find memory access time with high precision?
   ▶ Easiest way: instructions like rdtscp
▶ How do we use the timing to determine the secret?
   ▶ Ask for array2[4096 * x] for x = 0, 1, .... Each access falls in a different cache line. The x taking the shortest time has the secret value.¹
▶ Why not immediately check idx1 < array1_size?
   ▶ Cache miss on array1_size, e.g. because attacker has done setup to fill the caches with garbage
▶ Why predict that idx1 < array1_size?
   ▶ We can fool the branch predictor by engineering it so that typically idx1 < array1_size, except this one time.

¹Might mix order to prevent prefetching.
Demo

- (Show demo here.)
Impact

- If an attacker can make code that looks like this run, and can
time cache accesses, then they can exploit this to read any
memory in that address space
  - Which never happens, except always when using Javascript.
  - Modern Javascript code is JIT’d to machine code, which makes
this even easier
  - Some mitigations by browser vendors to reduce this vector
- Extended Berkeley Packet Filters programs (go bears!) which
run in Linux kernel
  - Allows userspace programs to filter which packets they want to
receive on a kernel level
  - Includes things like conditionals, arrays...
  - Spectre 🔵 reading kernel memory!!
- It is possible (but harder) to exploit this using existing code
when attacker controls idx1.
Mitigations

- Hardware mitigations seem difficult – fix in software?
- Just add serializing instructions (lfence on x86\(^2\), csdb on ARM\(^3\)) before every branch destination!
  - Estimated 60% slowdown.\(^4\)
- Add serializing instruction if Spectre would occur
  - Microsoft Compiler does this with /Qspectre\(^4\)
  - Intel Compiler does this with conditional-branch flag\(^5\)
  - Only works in simple cases!
- Never (even speculatively) have an out-of-bounds array access
  - Could use idx1 \% array1_size instead
  - Can do this quicker using some bitwise tricks, Linux does this.\(^6\)

---

\(^2\)“Intel Analysis of Speculative Execution Side Channels”
\(^3\)“Consumption of Speculative Data Barrier” in ARMv8 ISA Documentation
\(^4\)https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html
\(^5\)“Intel Compilers to Mitigate Speculative Execution Side-Channel Issues”
\(^6\)include/linux/nospec.h as of commit 7876320f88802b22
Indirect Branch Prediction

```c
int (*fp)(int, int) = is_minimizing_player ? &min : &max;
fp(3, 7);
```

- **Indirect branch**: when we don’t know where we will be branching to until the instruction is executed
- Occurs a lot in practice
  1. Function pointers (like above)
  2. Virtual functions in C++ (and generally runtime polymorphism)
  3. `ret` in x86 ISA, although the indirect branch predictor is not actually used for this
- Again, typically use (a truncated) virtual address of the branch as an index for a history table
Variant 2: Poisoning Indirect Branches

Branch predictor is trained by attacker in “Context A” to jump to a certain virtual address.
In victim “Context B”, branch predictor speculatively jumps to “Spectre Gadget”.
Attacker controls the virtual address destination of this branch.
As in Variant 1, the results of speculation can be observed architecturally through timing.

Figure 2: Taken from Spectre Paper
Variant 2 Is Super Powerful

- Say that the following appears in the code:

```plaintext
; from ntdll.dll on Win32
adc edi, DWORD PTR [ebx + edi + 0x13BE13BD]
; edi += (load 4 bytes from (ebx+edi+0x13BE13BD)) + CF
adc dl, BYTE PTR [edi]
; dl += (load a byte from edi) + CF
```

- And we have an indirect branch in the victim code where the attacker controls ebx and edi
- Then an attacker can read arbitrary memory!
- Might be possible to detect cache accesses from another process running on the same core. How?
  - Process shares a physical page with you (e.g., dynamic library)
  - Can probe the cache using your copy of this physical page
“Training” the indirect branch predictor can be done:
- Between processes on the same CPU
- Between user and kernel code on the same CPU
- Between VM guests and their hypervisors!!
- On some microarchitectures, across CPUs

These “gadgets” are very common
- Like return-oriented programming gadgets, they can even appear in the middle of other x86 instructions...
- and even if they don’t appear in your code, they probably appear in the code of some dynamic library you use
Mitigations

- Disable indirect branch predictor completely (sad)
- Flush branch predictor on context switch
  - Microcode updates available to do this
- Separate branch prediction by process
  - Already done for TLB with PCIDs/ASIDs, not a huge addition
- Retpolines!\(^7\)
  - Use `ret` instruction to do indirect branches
  - Return Stack Buffer (RSB) can't be poisoned by other processes

\(^7\)https://support.google.com/faqs/answer/7625886
Further Directions

- Not limited to cache-timing and branch prediction, so many possibilities!
- Other Spectre and Spectre-like vulnerabilities: Variant 3a, Variant 4, Foreshadow, LazyFP, Spectre 1.1, Spectre 1.2, Spectre v5, Spectre RSB, NetSpectre, TLBleed ...
- NetSpectre is particularly cool: shows that Spectre-like attacks are possible to detect over the network (albeit very slowly)
- Microarchitecture largely proprietary, not well understood
  - Current information mostly from reverse engineering
  - “Technical details published by microprocessor vendors is often superficial, incomplete, selective and sometimes misleading.” - Agner Fog on microarchitecture

---

9 https://mlq.me/download/netspectre.pdf
10 https://www.agner.org/optimize/microarchitecture.pdf
Questions & Feedback Welcome