Spectre and Meltdown: Data leaks during speculative execution

Speaker: Jann Horn (Google Project Zero)
Paul Kocher (independent)
Daniel Genkin (University of Pennsylvania and University of Maryland)
Yuval Yarom (University of Adelaide and Data61)
Involved researchers

- **Meltdown:**
  - Werner Haas, Thomas Prescher (Cyberus Technology)
  - Daniel Gruss, Moritz Lipp, Stefan Mangard, Michael Schwarz (Graz University of Technology)
  - Jann Horn (Google Project Zero)
  - Anders Fogh (GData) [credited for contributing ideas]

- **Spectre:**
  - Paul Kocher in collaboration with, in alphabetical order, Daniel Genkin (University of Pennsylvania and University of Maryland), Mike Hamburg (Rambus), Moritz Lipp (Graz University of Technology), and Yuval Yarom (University of Adelaide and Data61)
  - Jann Horn (Google Project Zero)
Outline

- Shared concepts for the attack variants
- Variants overview
- Spectre / variant 1
- Spectre / variant 2
- Meltdown / variant 3
Cache-based covert channel

- Memory access patterns affect data cache state
- Cache state affects memory access timing
- Measuring access timings reveals information about memory access patterns
  - here: FLUSH+RELOAD
- Normally used as side channel
- Other covert channels exist
Speculative and Out-of-Order Execution, Branch Prediction

- Instructions can be executed in a different order and in parallel
- Branches are predicted before the target is known

```java
1 if (foo_array[index1] ^ foo_array[index2] == 0) {
2    result = bar_array[100];
3 } else {
4    result = bar_array[200];
5 }
```
Misspeculation

- Exceptions and incorrect branch prediction can cause “rollback” of *transient instructions*
- Old register states are preserved, can be restored
- Memory writes are buffered, can be discarded
- **Cache modifications are not restored!**
Covert channel out of misspeculation

- Sending via cache-based covert channel works from transient instructions
## Variants overview

<table>
<thead>
<tr>
<th>Spectre</th>
<th>Meltdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>● CVE-2017-5753</td>
<td>● CVE-2017-5754</td>
</tr>
<tr>
<td>● &quot;Variant 1&quot;</td>
<td>● &quot;Variant 3&quot;</td>
</tr>
<tr>
<td>● &quot;Bounds Check Bypass&quot;</td>
<td>● &quot;Rogue Data Cache Load&quot;</td>
</tr>
<tr>
<td>● Primarily affects interpreters/JITs</td>
<td>● Affects kernels (and architecturally equivalent software)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>● CVE-2017-5715</td>
<td></td>
</tr>
<tr>
<td>● &quot;Variant 2&quot;</td>
<td></td>
</tr>
<tr>
<td>● &quot;Branch Target Injection&quot;</td>
<td></td>
</tr>
<tr>
<td>● Primarily affects kernels/hypervisors</td>
<td></td>
</tr>
</tbody>
</table>
Variant 1: Conditional Branch Example

```c
if (x < array1_size)
    y = array2[array1[x] * 256];
```

- Execution without speculation is safe
  - CPU will never read array1[x] for any $x \geq array1\_size$
- Execution with speculation can be exploited
  - Attacker sets up some conditions
    - train branch predictor to assume ‘if’ is likely true
    - make array1\_size and array2[] uncached
  - Invokes code with out-of-bounds $x$ such that array1[x] is a secret
    - NOTE: This read changes the cache state in a way that depends on the value of array1[x]
    - ... recognizes its error when array1\_size arrives, restores its architectural state, and proceeds with ‘if’ false
- Attacker detects cache change (e.g. basic FLUSH+RELOAD or EVICT+RELOAD)
  - E.g. next read to array2[i*256] will be fast $i=\text{array}[x]$ since this got cached

Note: Only need a few instructions to run speculatively, but CPUs can run many more (e.g. ~200 on Haswell)
Variant 1: Violating the JavaScript Sandbox

JavaScript code runs in a sandbox
- Not permitted to read arbitrary memory
- No pointers, array accesses are bounds checked

Browser runs JavaScript from untrusted websites
- JavaScript engine can interpret code (slow) or compile it (JIT) to run faster
- In all cases, engine must is required to ensure sandbox (e.g. apply bounds checks)

Speculative execution can blast through safety checks…
- Can we write JavaScript that compiles into machine code that leaks memory contents?
Variant 1: Violating JavaScript’s Sandbox

index will be in-bounds on training passes, and out-of-bounds on attack passes

```javascript
if (index < simpleByteArray.length) {
    index = simpleByteArray[index | 0];
    index = (((index * TABLE1_STRIDE)|0) & (TABLE1_BYTES-1))|0;
    localJunk ^= probeTable[index|0]|0;
}
```

Teach JIT that index is in bounds for `simpleByteArray[]` so it can omit bounds check in next line. Want length uncached for attack passes

Do the out-of-bounds read on attack passes!

Need to use the result so the operations aren’t optimized away

Leak out-of-bounds read result into cache state!

4096 bytes (= page size)

This AND keeps the JIT from adding unwanted bounds checks on the next line

“|0” is a JS optimizer trick (makes result an integer)
Variant 2: Basics

- Branch predictor state is stored in a Branch Target Buffer (BTB)
  - Indexed and tagged by (on Intel Haswell):
    - partial virtual address
    - recent branch history fingerprint
- Branch prediction is expected to sometimes be wrong
- Unique tagging in the BTB is unnecessary for correctness
- Many BTB implementations do not tag by security domain
- Prior research: Break Address Space Layout Randomization (ASLR) across security domains
- Inject misspeculation to controlled addresses across security domains
Variant 2: Exploitation against KVM

- break hypervisor ASLR using branch prediction
- misdirect first indirect call with memory operand after guest exit
- flush cache line containing memory operand
- guest register state stays across VM exit
- guest memory is mapped
- abuse eBPF bytecode interpreter; call through register-loading gadget

```
ffffffff81514edd: mov rsi,r9
ffffffff81514ee0: call QWORD PTR [r8+0xb0]
```

```
static unsigned int _bpf_prog_run(void *ctx, const struct bpf_insn *insn)
```
Meltdown / Variant 3

```
i = *pointer;
y = i * 256;
z = array2[y];
```
Meltdown / Variant 3

```c
i = *pointer;
y = i * 256;
z = array2[y];
```
Meltdown / Variant 3

```
i = *pointer;
y = i * 256;
z = array2[y];
```

Diagram:
- Pointer
- Array2
- Cache
- Architectural code
- Virtual memory
- Kernel space
- Physical memory
Meltdown / Variant 3

\[
i = \star \text{pointer};\\
y = i \times 256;\\
z = \text{array2}[y];\\
\]

![Diagram of memory access with architectural attack code](image)

- Pointer
- Array2
- Kernel Space
- Physical Memory
- Virtual Memory
- Cache
Meltdown / Variant 3

```c
i = *pointer;
y = i * 256;
z = array2[y];
```

---

**Diagram:**
- **Virtual Memory:**
  - Pointer
  - Array 2

- **Kernel Space:**
  - Physical Memory

- **Cache:**
  - Architectural attack code

---

**Architecture:**
- **Pointer:**
- **Array 2:**
- **Kernel Space:**
  - **Physical Memory**
Meltdown / Variant 3

```c
i = *pointer;
y = i * 256;
z = array2[y];
```

Cache

```plaintext
data == 3
```

virtual memory

pointer

array2

Kernel Space

Physical Memory

architectural attack code
Meltdown / Variant 3

- Privilege checks for memory access based on pagetable entries
- Privilege checks can be performed asynchronously
- **Dependent instructions can execute before execution is aborted!**
- Race condition in the privilege check
- Straightforward attack: Leak cached data
- TU Graz result: Uncached data can also be leaked
- Suppression of architectural pagefault:
  - signal handler
  - TSX
  - mispredicted branch
Conclusion

- Covert channels in CPUs are useful for more than transferring secrets between isolated processes
- Not all security issues are correctness issues
References

Papers / Blogposts on Meltdown / Spectre:

- Spectre: https://spectreattack.com/spectre.pdf
- https://blog.cyberus-technology.de/posts/2018-01-03-meltdown.html

Prior research mentioned in this talk:

- Yuval Yarom, Katrina Falker: "FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack"
- Dmitry Evtyushkin, Dmitry Ponomarev and Nael Abu-Ghazaleh: "Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR"
- Felix Wilhelm: https://github.com/felixwilhelm/mario_baslr "PoC for breaking hypervisor ASLR using branch target buffer collisions"