Meltdown & Spectre: Microarchitectural security bugs

https://meltdownattack.com/
Meltdown & Spectre

- Intro
- Introduction to Processors
- Fast processors
- Side-channel attacks
- Meltdown
- Spectre 1
- Spectre 2
- Workarounds
Intro

- Allow unprivileged programmes to read kernel memory
- Demo of it being done via Javascript in a browser through a VM!
  
  But pretty hairy
- Not a software bug
- Hardware meets specification
- Most modern fast processors
  
  Spectre more common, Meltdown rarer (Intel some ARM)
  Some low end ARMs/Atoms are immune
Processors (aka CPUs, Cores)

- **CPUs execute a stream of instructions, reading/writing memory and registers**

- **Instructions:**
  - Add, Multiply Load, Store, Compare, Branch; all stored in memory

- **Memory:**
  - Slow, but big; think of as having a Single large address (e.g. 0...4 billion)

- **Registers:**
  - Fast, but not many, e.g. 16
Programs

- **Sequences of instructions**
  At a series of memory locations
  All just numbers

- **Lets ignore how they get there, how they start running or how they stop**

- **Some special instructions**
  e.g. syscalls to enter OS, return, etc etc

```
0 5 1 0 Set r1= 0
1 1 2 3 Load r2 ← [r3]
2 a 33 1 Add r3,r3,#1
3 c 2 0 Compare r2,#END
4 b 0 7 Branch if-equal 7
5 a 11 1 Add r1,r1,#1
6 b 7 1 Branch always 1
7 7 55 1 Store result ← r1
8 f 00 0 exit
```

Simple string length
**Fetch, Decode, Execute (Simple)**

- **For every instruction:**
  - Fetch it from memory
  - Decode it (e.g. 5/1/0 means Set r1=0)
  - Execute it (store 0 in r1)

- **At least 3 clocks/instruction**
  - Some times ‘executes’ take longer (e.g. a multiply)

---

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
</tr>
<tr>
<td>D1</td>
</tr>
<tr>
<td>E1</td>
</tr>
<tr>
<td>D1</td>
</tr>
<tr>
<td>D2</td>
</tr>
<tr>
<td>E2</td>
</tr>
<tr>
<td>F3</td>
</tr>
<tr>
<td>D3</td>
</tr>
<tr>
<td>E3</td>
</tr>
<tr>
<td>F4</td>
</tr>
<tr>
<td>D4</td>
</tr>
<tr>
<td>E4</td>
</tr>
<tr>
<td>F5</td>
</tr>
<tr>
<td>D5</td>
</tr>
<tr>
<td>E5</td>
</tr>
<tr>
<td>F6</td>
</tr>
<tr>
<td>D6</td>
</tr>
<tr>
<td>E6</td>
</tr>
<tr>
<td>F1</td>
</tr>
<tr>
<td>D1</td>
</tr>
<tr>
<td>E1</td>
</tr>
</tbody>
</table>

Branch!
Pipelines

- **Overlap instructions**
  Upto 1 instruction/cycle

- **Branches get complex**
  throw away stuff

<table>
<thead>
<tr>
<th>0</th>
<th>5</th>
<th>1</th>
<th>0</th>
<th>Set r1= 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>Load r2 ← [r3]</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>33</td>
<td>1</td>
<td>Add r3,r3,#1</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>2</td>
<td>0</td>
<td>Compare r2,#END</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>0</td>
<td>7</td>
<td>Branch if-equal 7</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>11</td>
<td>1</td>
<td>Add r1,r1,#1</td>
</tr>
<tr>
<td>6</td>
<td>b</td>
<td>7</td>
<td>1</td>
<td>Branch always 1</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>55</td>
<td>1</td>
<td>Store result ← r1</td>
</tr>
<tr>
<td>8</td>
<td>f</td>
<td>00</td>
<td>0</td>
<td>exit</td>
</tr>
</tbody>
</table>

7  Meltdown+Spectre/2018/dave@treblig.org
Caches: A fast small chunk of memory

- **CPU clocks got faster**
  - But memory latency lagged
  - e.g. 0.3ns CPU clock
  - 10ns RAM latency
  - **30 cycles to fetch data**

- **Caches**
  - Between core and memory
  - Often multiple levels
  - Shared between cores
Caches: Allocation & Replacement

- **Multiple ‘lines’**
  Hold one chunk of data from RAM

- **Easiest way to find slot**
  e.g. address/lines remainder

- **Replace when reused**
  Read 3008 – goes in slot 8 - SLOW
  Read 3008 – read from slot 8 - **FAST**
  Read 4008 – replaces slot 8 - SLOW
  Read 3008 – read RAM again - SLOW

<table>
<thead>
<tr>
<th>Lines</th>
<th>Actual address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Free</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Free</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2002</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2003</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Free</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Free</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Free</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2007</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3008</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5109</td>
<td></td>
</tr>
</tbody>
</table>
Pipelines: Got more complex

- Split up more: Faster clocks
- Multiple execution units
- Lots of instructions in flight
  Maybe over 100
- Instructions out-of-order
  Skip an instruction while waiting for a result
  Put them back at the end
- Branches even more costly
  Must try and **predict** branches based on what they did before
Hyperthreading/SMT

- One complex pipeline running 2 or more sets of code
  Looks like multiple cores

- Shares complex things like
  Branch prediction
  All the execution units
Virtual memory & permissions

- **User processes can’t access each others memory**
  
  One ‘page table’ per process

- **User processes can’t access kernel memory**
  
  ‘privilege flag’ in pages

- **You can swap**
  
  Page tables can have gaps that cause errors when accessed, to cause disk to be read
Sidechannel attacks

- If CPU does what it is supposed to
  Rely on the timing of instructions
- e.g. if a read is fast it’s in the cache
  Which probably means someone else read it first!
- Often slow exploits
  Having to time things and wait
- Originally demonstrated in special cases
  e.g. finding out if another process was using crypto tables
  Previously felt pretty obscure
Meltdown!

- **Read kernel memory!**
- (a) means the rest is just prediction
- (b) loads from the kernel memory – still happens on some systems, but protection guarantees thrown away **eventually**
- But not before (c) that uses it to choose an address to access
- (d) Accesses memory later – at base+.... - finds which one is fast : The one that is fast corresponds to (b)’s value
- Relies on CPU implementation getting data in (b) before it notices permission error; some CPUs do, some don’t.
• Get kernel to do it for you
  Find somewhere in kernel that accesses table based on user input
  Of course kernel range checks it first

• Speculation means load happens and contaminates cache

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Cmp r1&lt;table-size ?</td>
</tr>
<tr>
<td>b</td>
<td>Branch if too big</td>
</tr>
<tr>
<td>c</td>
<td>Load r2 ← [...r1]</td>
</tr>
<tr>
<td>d</td>
<td>Load r3 ← [ r2...]</td>
</tr>
<tr>
<td>e</td>
<td>Time memory access</td>
</tr>
</tbody>
</table>
Spectre v2

- **Mistrain branch predictor to go where you want it!**
  
  Very hard, branch predictor algorithms are complex; always assumed to be unpredictable – they reverse engineered it!

  Only *indirect* branches normally

- **Find somewhere in kernel that has the ‘loads’**

- **Hardest to exploit**
  Also hardest to fix!
  But with public proof-of-concept

- **Almost all complex processors (i.e. with complex branch predictors)**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Misuse branch in kernel to...</td>
</tr>
<tr>
<td>b</td>
<td>Load r2 ← [...r1]</td>
</tr>
<tr>
<td>c</td>
<td>Load r3 ← [ r2...]</td>
</tr>
<tr>
<td>d</td>
<td>Time memory access</td>
</tr>
</tbody>
</table>
**Workaround: Meltdown**

- **‘Page Table Isolation’**

  - Keep two page tables
  - One for user-space has no kernel pages at all
  - Switch everytime we go in and out of kernel

  **Expensive**

    A bit better on newer CPUs with ‘PCID’

  **dmesg|grep isolation**

<table>
<thead>
<tr>
<th>User</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3005</td>
</tr>
<tr>
<td>1</td>
<td>Disk</td>
</tr>
<tr>
<td>2</td>
<td>4000</td>
</tr>
<tr>
<td>3</td>
<td>3001</td>
</tr>
<tr>
<td>4</td>
<td>Disk</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8000</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Workaround: Spectre v1

- **Pointer sanitization**
  
  Any user pointer that’s checked needs an extra barrier to stop the pipeline.

  Special instruction (new?)

  Only some architectures have them

  Some masking tricks

- **Need to find **every** use in kernel!**

  Compiler tools

  Beware closed source modules

  a Cmp r1<table-size ?
  b Branch if too big
  c Load r2 ← [...r1]
  d Load r3 ← [ r2...]
  e Time memory access
Workaround: Spectre v2 [Retpoline]

- **Avoid all indirect branches!**
  
  Find an instruction that doesn’t get branch prediction
  
  ‘ret’ - return from function

- **Compiler changes**

- **Manual code needs checking**

- **Slower**

- **Doesn’t work on latest CPUs**
  
  Because they predicted returns
  
  Needed ways to stop that
Workaround: Spectre v2  
[Branch speculation flush/restriction]

- **Before retpoline solution**
- **New microcode**
  
  Gives your CPU new instructions!
  
  Changes the way branch predictor works
  
  Stops predictions from userspace influencing kernel
  
  Protects hyperthreads from each other

- **Might be faster on future chips**
  
  When they design them in rather than bodge on later

- **Got to wait for microcode for your chip**

- **No microcode (or equivalent) on some RISC chips**
Summary

- **A new type of attack**
  Probably more with similar idea coming

- **Not currently easy to actually use Spectre**
  A few KB/s read
  Hard to exploit remotely, but demonstrable

- **Fixes are all pretty messy**
  All slow things down a bit
  How much varies vastly depending on task and CPU