Preventing CPU side-channel attacks with kernel tracking

Marian Marinov
mm@siteground.com

Chief System Architect
Head of the DevOps department
❖ **Who am I?**

- Chief System Architect of SiteGround.com
- Sysadmin since 1996
- Organizer of OpenFest, BG Perl Workshops, LUG-BG and others
- Teaching Network Security and Linux System Administration courses in Sofia University and SoftUni
There are 4 different caches inside the CPU

- L1 instructions cache
- L1 data cache
- L2 cache
- L3 cache
- L1 I cache
- L1 D cache
- L2 cache
- L3 cache
- L1 I cache
- L1 D cache
- L2 cache
- L3 cache
- L1 I cache
- L1 D cache
- L2 cache
- L3 cache
There are 4* different caches inside the CPU.

* In some architectures, there is even L4 cache.
Sharing the cache

L1 and L2 caches are shared between hyper-threads in a single core.

L2 cache is shared between different execution engines inside the core (ALU, FMA, ADD, etc.).

L3 cache is shared between all cores.
Some CPU architecture intro :)
Cache Side-Channel Attacks

- Flush + Reload
- Flush + Flush
1. Find a shared library location in memory
2. Clear the cache
3. Check if it is accessed or not by comparing the time it takes to execute the code

<table>
<thead>
<tr>
<th>Single Core</th>
<th>Instruction decoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Instruction cache</td>
<td>Pipeline(s)</td>
</tr>
<tr>
<td>Isnt. Fetch</td>
<td>Branch Predict.</td>
</tr>
</tbody>
</table>

- Integer Cluster 1
  - L1 data cache
- Dispatch
- FPU
- W.C. Cache
- Integer Cluster 2
  - L1 data cache

Core Interface

L2 Data Cache shared

Shared L3 Cache (LLC)

Synchronization
Flush + Reload

Single Core

- L1 Instruction cache
- Isnt. Fetch
- Branch Predict.
- Pipeline(s)
- Instruction decoder
- Integer Cluster 1
  - L1 data cache
- Integer Cluster 2
  - L1 data cache
- Dispatch
- FPU
- W.C. Cache
- Core Interface
- L2 Data Cache shared
- Shared L3 Cache (LLC)

Synchronization
Flush + Reload

![Diagram of a computer system with various components including L1 Instruction cache, L1 data cache, Dispatch, Integer Cluster 1, Integer Cluster 2, W.C. Cache, L2 Data Cache, FPU, L1 data cache, Core Interface, L3 Cache, and Synchronization.]
Flush + Flush

1. Find a shared library location in memory
2. Clear the cache
3. Clear the cache again and observe the timing
Flush + Flush

Single Core

- L1 Instruction cache
- Branch Predict.
- Isnt. Fetch
- Pipeline(s)

Instruction decoder

- Integer Cluster 1
  - L1 Instruction cache
  - L1 data cache

- Integer Cluster 2
  - L1 data cache

- Dispatch
- FPU
- W.C. Cache

Core Interface

- L2 Data Cache shared

Shared L3 Cache (LLC)

Synchronization
Flush + Flush

Single Core
- L1 Instruction cache
- Isnt. Fetch
- Branch Predict.
- Pipeline(s)
- Instruction decoder
- Dispatch
- Integer Cluster 1
  - L1 data cache
  - Integer Cluster 2
  - L1 data cache
- FPU
- W.C. Cache
- Core Interface
- L2 Data Cache shared
- L2 Data Cache shared

Shared L3 Cache (LLC)

Synchronization
Cache Side-Channel Attacks

➢ Both attacks require CPU cache flushing
  ➢ clflush or clflushopt
  ➢ filling the cache with already known data (as in the Spectre attacks)
And than...

Meltdown

and

Spectre

came
More architecture...

Thread 1: floating point
Meltdown mitigation

➢ Protection
➢ Kernel Page Table Isolation (KPTI)
Meltdown

- exploits usually use
- Flush + Reload
- Flush + Flush
- TSX
- for better results the dying processes should be on the same core
Meltdown mitigation

➢ Protection
➢ Kernel Page Table Isolation (KPTI)

All memory

kernel space

user space
Spectre – Mitigations

Variant 1

➢ Retpoline
Spectre – Mitigations

Variant 2

- Indirect Branch Restricted Speculation (IBRS)
- Single Thread Indirect Branch Predictors (STIBP)
- Indirect Branch Predictor Barrier (IBPB)
Spectre - Mitigations

Variant 2

- Indirect Branch Restricted Speculation (IBRS)
  - disable branch prediction
- Single Thread Indirect Branch Predictors (STIBP)
  - isolate the predictions to hyper-thread
- Indirect Branch Predictor Barrier (IBPB)
  - prevent user space predictions to leak to kernel space (on every context switch)
All of the proposed mitigations cost performance in different scales
Apache Benchmark v2.4.7
Static Web Page Serving

<table>
<thead>
<tr>
<th>KPTI Off</th>
<th>SE +/- 315.70</th>
<th>Linux 4.4.110-rc1</th>
<th>49434.20</th>
</tr>
</thead>
<tbody>
<tr>
<td>SE +/- 140.80</td>
<td>44140.28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SE +/- 93.59</td>
<td>43564.16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SE +/- 203.52</td>
<td>47057.09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SE +/- 105.38</td>
<td>41715.29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SE +/- 64.13</td>
<td>41260.62</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KPTI On</th>
<th>SE +/- 203.52</th>
<th>Linux 4.9.75-rc1</th>
<th>47057.09</th>
</tr>
</thead>
<tbody>
<tr>
<td>SE +/- 105.38</td>
<td>41715.29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SE +/- 64.13</td>
<td>41260.62</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Requests Per Second, More Is Better

OpenBenchmarking.org

1. (CC) gcc options: -shared -fpic -O2 -pthread

What others did?

- Monitoring and analysis
- Capsule 8 – wrote on Jan 5
- EndGame – shared their research on Jan 08
- I found out about EndGame and Capsule 8 on Jan 11, after we had already started on our work
Capsule 8 approach

Kernel tracepoints and monitor for:

- exceptions/page_fault_user

Kernel perf counters

- PERF_COUNT_HW_CACHE_OP_READ
- PERF_COUNT_HW_CACHE_RESULT_ACCESS
- PERF_COUNT_HW_CACHE_RESULT_MISS

https://github.com/capsule8/capsule8/tree/master/examples
Capsule 8 approach

This is nice, but not enough...

since

Flush+Reload can be replaced by Flush+Flush to achieve the same result without actual page miss

https://github.com/capsule8/capsule8/tree/master/examples
They did not provide code examples...

However they explained a lot around the statistics and using the CPU performance counters.

Both examples by Capsule 8 and EndGame provide detection, but little to no countermeasures.
What is our setup...

First, our environment is shared hosting and Linux containers hosting, where users may have root access.
What is our setup...

A decrease of CPU performance by 5–15% for one machine is not much...
What is our setup...

A decrease of CPU performance by 5-15% for one machine is not much...

but if you have hundreds or thousands of machines...
the numbers become UGLY 😞
What we did?

Kernel module
What we did?

Kernel module

- detecting processes that had more than 1 child dying with SIGSEGV
What we did?

Kernel module

- detecting processes that had more than 1 child dying with SIGSEGV
- when such process is detected it is STOPPED, not KILLED
What we did?

Kernel module

- detecting processes that had more than 1 child dying with SIGSEGV
- when such process is detected it is STOPPED, not KILLED
- only the root on the host machine can continue or kill this process
What we had to change?

- Introduced a per process counter of its SIGSEGV children
- check the counter on every reschedule of the process
- cpu pinning is allowed only to the host root
Cache Line Flush

- Limiting clflush and clflushopt effectively stops Flush+Reload and Flush+Flush attacks
- cache flush can be indirectly called when invalid instruction is issued
- This greatly limits the options for executing Meltdown, by leaving only TSX instructions
Cache Line Flush

- Both clflush and clflushopt are unprivileged instructions. Trapping them is not directly possible.

- We discussed different approaches:
  - Inspecting the instructions of each binary, before it is executed and marking it clean.
  - Inspecting the binary in parallel while the program is executing.
  - Virtualizing the system and actually trapping the instructions after they have been evaluated by the guest kernel.
Cache Line Flush

- adding noclflush on the kernel cmdline does NOT disable clflush!!!

- Events you can monitor for clflush:
  
  - L2_LINES_OUT.DEMAND_CLEAN
  - MEM_LOAD_UOPS_RETIRED.L3_MISS
  - PERF_COUNT_HW_CACHE_LL
  - PERF_COUNT_HW_CACHE_OP_READ
  - PERF_COUNT_HW_CACHE_RESULT_ACCESS
  - PERF_COUNT_HW_CACHE_RESULT_MISS
**TSX**

Transactional Synchronization eXtensions (TSX)

- Because of issues with the implementation, TSX instructions should be disabled on Haswell CPUs.

- However if the microcode is not applied your Haswell CPUs support TSX :)

- TSX instructions are supported on Skylake...
One thing that EndGame showed us is that TSX instructions can be counted

- RTM_RETIRED.ABORTED
TSX

- I considered lying to userspace by reporting that TSX is not supported by the CPU...
  
  - but the cpuid instruction is unprivileged so trapping it is a non-trivial job 😞
TSX

- So relying on CPU counters is currently the only sensible way of detecting if TSX is being used for Meltdown exploits.
- There is almost no code that has a legitimate use of TSX instructions, and on shared hosting we will most likely not see such software.
TSX events

- **RTM_RETIRED.START** - Number of times we entered an RTM region. Does not count nested transactions

- **RTM_RETIRED.ABORTED** - Number of times an RTM execution aborted due to any reasons (multiple categories may count as one)

- **RTM_RETIRED.ABORTED_TIMER** - Number of times an RTM execution aborted due to uncommon conditions
you can find the list of Processor Monitor Unit (PMU) events by running:

```bash
# perf list
```

Perf can be build from the linux kernel source tree in tools/perf:

```bash
# make
# mv perf /usr/bin
```
Glossary

ALU - Arithmetic Logic Unit
AGU - Address Generation Unit
TLB - Translation Lookaside Buffer
BTP - Branch Target Predictor
BP - Branch Predictor
BTB - Branch Target Buffer
LLC - Last Level Cache
WB Cache - Writeback cache
W.C. Cache - Write combining cache
Trace Cache - execution trace cache
Links

Flush + Reload paper
Flush + Flush paper
Spectre & Meltdown attacks
EndGame research
Capsule 8 meltdown detection
Meltdown PoC
Collection of Speculation bugs info
Marian Marinov
mm@siteground.com