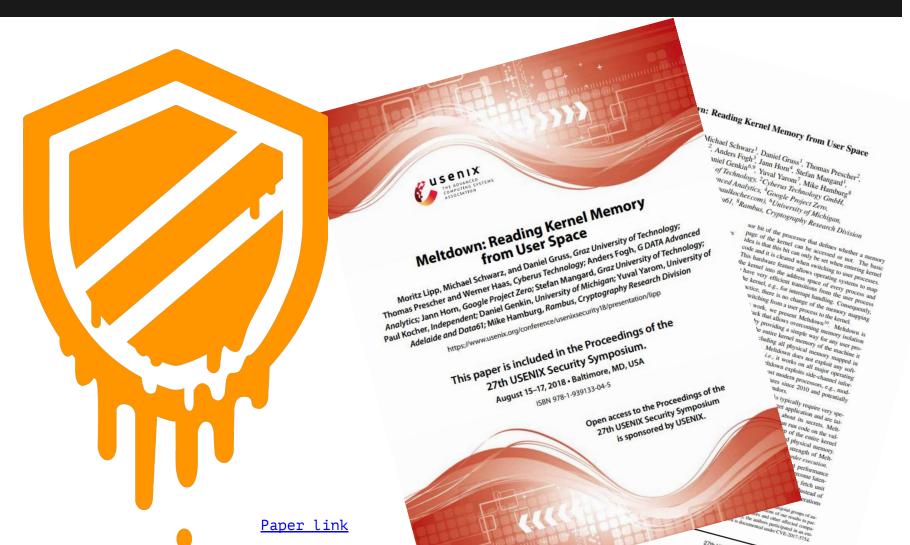
Assignment 3: Meltdown

Hardware Security

Meltdown aka CVE-2017-5754



What?

- An attack which exploits hardware optimizations.
- Breaks the isolation between
 - User and OS
 - User and User (e.g., JS sandbox)
- Allows a User program to read inaccessible kernel memory, and thus leak secrets of the OS and other User programs.

What again?

- A User reads arbitrary kernel memory addresses, mapped in his own address space.
 - Leaks browsers data, passwords, ssh keys etc.
- Exploiting instruction streams that are executed:
 - Out-of-order
 - Speculatively
- ... then rolled back in the CPU pipeline.
 - E.g. due to wrong access privilege, a condition misprediction etc.
- \bullet Leaving observable side effects on the $\mu arch$.
 - E.g. cache accesses based on the leaked data.

The idea

- Instructions are executed Out-of-Order.
- Even logically-after an exception has been raised.
- How can we observe access(buff[0x72]) ?
- 500 400 300 200 0 50 100 150 200 250 buff positions

1 raise_exception();
2 access(buff[0x72]);

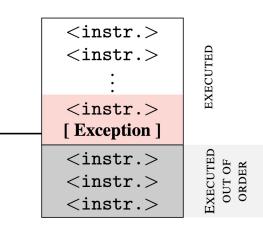
EXCEPTION

HANDLER

<instr.>

<instr.>

Terminate



Meltdown Core

```
1 ;rcx = kernel address
2 ;rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- Load the privileged address to a register.
- The next instrs are executed out of order.
- Check the probe array afterwards for cache hits.

Easy, right?

Complication 1: Noise

- The CPU µarch is so complex that it can act in what seems like random and unpredictable
- Aka noisy
- One measurement will be unreliable and makes a broken meltdown harder to debug

```
1 ;rcx = kernel address
2 ;rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- Repeat the measurements over and over for the same address
- Keep the cache-hits scores and pick the best scoring result

Complication 2: Exception

- The exception will kill your meltdown
- You can suppress it in different ways:
 - TSX transaction
 - SEGV handler

```
1 ;rcx = kernel address
2 ;rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Complication 3: Caching

- Meltdown is sensitive to what is in the cache
- The cache is your only observable resource
- Get the target in L1D (simply access the address)
- CLFLUSH the probe buffer every address

```
1 ;rcx = kernel address
2 ;rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

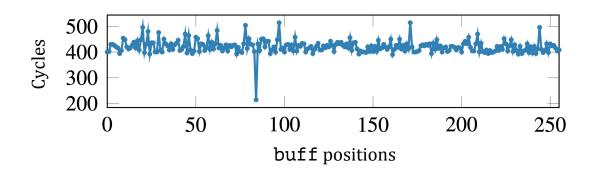
Complication 4: Prefetcher

- The hardware prefetcher will try to help you by caching stuff
- This is terrible for the probe buffer
- You have to fool the prefetcher
- Test the probe buffer in a random order
- Or at least random enough to fool the prefetcher

```
1 ;rcx = kernel address
2 ;rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Complication 5: Threshold

- When you probe an address and measure with RDTSC, when is it cached?
- Figure out the threshold (you did it F+R)
- Hardcode the threshold with max separation
- Use CLFLUSH to get uncached memory



Complication 6: µarch

- Different micro-architectures behave differently
- Meltdown doesn't exist on every one
- Intel is the 'safest'
 - Meldown paper table 1

Environment	CPU Model	Cores
Lab	Celeron G540	2
Lab	Core i5-3230M	2
Lab	Core i5-3320M	2
Lab	Core i7-4790	4
Lab	Core i5-6200U	2
Lab	Core i7-6600U	2
Lab	Core i7-6700K	4
Lab	Core i7-8700K	12
Lab	Xeon E5-1630 v3	8
Cloud	Xeon E5-2676 v3	12
Cloud	Xeon E5-2650 v4	12
Phone	Exynos 8890	8

Complication 7: Host conditions

Note: Make sure your exploit work on our cluster of CPUs.

On other machines:

- RDTSC has to be reliable, so disable power saving mode and frequency scaling
- Use the 'performance' governor
- Disabled KPTI
- Plug AC in for laptops

Assignment: 0x3 Tasks

Different ways to suppress the faults

Task #1:

Implement Meltdown with SEGV

Task #2:

• Implement Meltdown with TSX

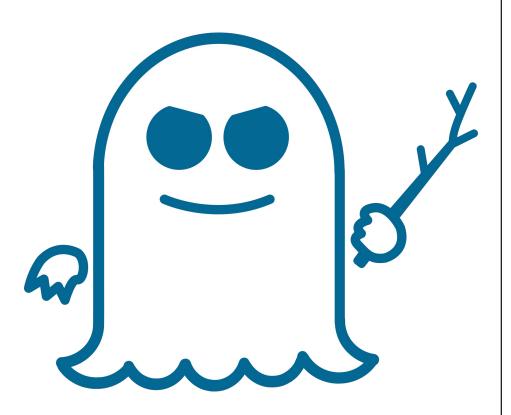
Task #3:

Meltdown using speculative execution

Assignment

- The assignment comes with a kernel module
- WOM: Write-only memory
- You can write data into it
- ioctl the device /dev/wom from C
- Returns privileged buffer address
- This is what you must leak with your Meltdown
- See template code in main-meltdown.c
- See also README for more info

Spectre aka CVE-2017-5753 & CVE-2017-5715



Spectre Attacks: Exploiting Speculative Execution

Paul Kocher¹, Jann Horn², Anders Fogh³, Daniel Genkin⁴, Daniel Gruss⁵, Werner Haas⁶, Mike Hamburg⁷, Moritz Lipp⁵, Stefan Mangard⁵, Thomas Prescher⁶, Michael Schwarz⁵, Yuval Yarom⁸ ¹ Independent (www.paulkocher.com), ² Google Project Zero, 3 G DATA Advanced Analytics, 4 University of Pennsylvania and University of Maryland, ⁵ Graz University of Technology, ⁶ Cyberus Technology, ⁷ Rambus, Cryptography Research Division, ⁸ University of Adelaide and Data61

Abstract-Modern processors use branch prediction and speculative execution to maximize performance. For example, if the destination of a branch depends on a memory value that is in the ocess of being read, CPUs will try to guess the destination and attempt to execute ahead. When the memory value finally arrives, the CPU either discards or commits the speculative computation Speculative logic is unfaithful in how it executes, can access the victim's memory and registers, and can perform operations with measurable side effects.

Spectre attacks involve inducing a victim to speculatively perform operations that would not occur during correct program execution and which leak the victim's confidential information via a side channel to the adversary. This paper describes practical attacks that combine methodology from side channel attacks. fault attacks, and return-oriented programming that can read arbitrary memory from the victim's process. More broadly, the paper shows that speculative execution implementations violate the security assumptions underpinning numerous software secu-rity mechanisms, including operating system process separation, containerization, just-in-time (JIT) compilation, and countermeasures to cache timing and side-channel attacks. These attacks represent a serious threat to actual systems since vulnerable speculative execution capabilities are found in microprocessors from Intel, AMD, and ARM that are used in billions of devices.

While makeshift processor-specific countermeasures are possi-ble in some cases, sound solutions will require fixes to processor designs as well as updates to instruction set architectures (ISAs) to give hardware architects and software developers a common understanding as to what computation state CPU implementations are (and are not) permitted to leak.

Computations performed by physical devices often leave observable side effects beyond the computation's nominal outputs. Side-channel attacks focus on exploiting these side effects to extract otherwise-unavailable secret information. Since their introduction in the late 90's [43], many physical radiation [58], or acoustic noise [20] have been leveraged to safe. extract cryptographic keys as well as other secrets.

Physical side-channel attacks can also be used to extract secret information from complex devices such as PCs and mobile phones [21, 22]. However, because these devices often execute code from a potentially unknown origin, they face additional threats in the form of software-based attacks, which do not require external measurement equipment. While some attacks exploit software vulnerabilities (such as buffer overflows [5] or double-free errors [12]), other software attacks these instructions on the nominal CPU state are eventually

leverage hardware vulnerabilities to leak sensitive information. Attacks of the latter type include microarchitectural attacks exploiting cache timing [8, 30, 48, 52, 55, 69, 74], branch prediction history [1, 2], branch target buffers [14, 44] or open DRAM rows [56]. Software-based techniques have also been used to mount fault attacks that alter physical memory [39] or internal CPU values [65].

Several microarchitectural design techniques have facilitated the increase in processor speed over the past decades. One such advancement is speculative execution, which is widely used to increase performance and involves having the CPU guess likely future execution directions and prematurely execute instructions on these paths. More specifically, consider an example where the program's control flow depends on an uncached value located in external physical memory. As this memory is much slower than the CPU, it often takes several hundred clock cycles before the value becomes known. Rather than wasting these cycles by idling, the CPU attempts to guess the direction of control flow, saves a checkpoint of its register state, and proceeds to speculatively execute the program on the guessed path. When the value eventually arrives from memory, the CPU checks the correctness of its initial guess. If the guess was wrong, the CPU discards the incorrect speculative execution by reverting the register state back to the stored checkpoint, resulting in performance comparable to idling. However, if the guess was correct, the speculative execution results are committed, yielding a significant performance gain as useful work was accomplished during the delay.

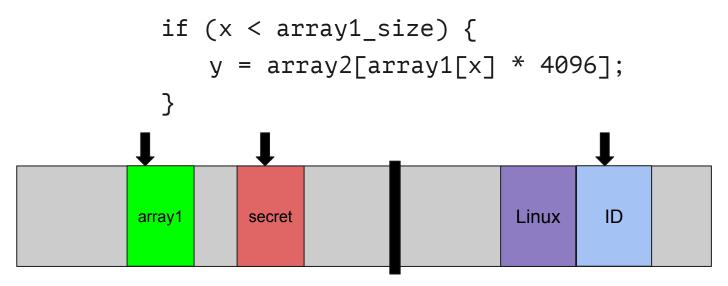
From a security perspective, speculative execution involves executing a program in possibly incorrect ways. However, because CPUs are designed to maintain functional correctness by reverting the results of incorrect speculative executions to effects such as power consumption [41, 42], electromagnetic their prior states, these errors were previously assumed to be

In this paper, we analyze the security implications of such incorrect speculative execution. We present a class of microarchitectural attacks which we call Spectre attacks. At a high level, Spectre attacks trick the processor into speculatively executing instruction sequences that should not have been executed under correct program execution. As the effects of

Assignment: Task #3

- Implement a spectre version in a 3rd binary
- This version executes Meltdown in a speculative branch
- This suppresses the page fault

Spectre v1: Bounds Check Bypass



- array2 can be a FLUSH+RELOAD buffer or any other buffer (PRIME+PROBE)
- Can be used to leak with Meltdown (speculation for exception suppression)

Assignment: Practicalities

- Change the Makefile so that it produces a
 - ./{vunetID}-meltdown
- Optionally it also creates a
 - ./{vunetID}-spectre
- It should do this both for the "all" target

Built-in Test

- test.sh does an end-to-end test for you
- Builds your meltdown
- Generates a random 32-byte secret
- Writes it to /dev/wom
- Invokes your meltdown to leak it
- Compares results
- Prints pass or fail
 - Your exploit passes even with a tiny %
- Your Meltdown must work with this test

Built-in Test

% export VUNETID=bgs137

```
% bash test.sh
        bgs137-meltdown - given threshold of 100
   Generated secret: 45d5d8c70fc8c92cb38a85c075e58f63
   address: 0xffff88012565e000.
   45d5d8c70fc8c92cb38a85c075e58f63
   pass
\lceil \ldots \rceil
     * bgs137-spectre - given threshold of 100
   Generated secret: 8bd34da530af683a0711640d768adb2f
   address: 0xffff88012565e000.
   8bd34da5?0af683a0711640d768adb2f
   fail
```

Deliverable

Source of 3 binaries:

- <u>Task#1</u>
 - o ./{vunetID}-meltdown_segv with SEGV
- Task#2
 - o ./{vunetID}-meltdown_tsx with TSX
- Task#3
 - ./{vunetID}-spectre
 - o A plot showing:
 - The rounds (x-axis)
 - Leakage accuracy (y-axis) and round duration (y-axis)
- Base this on the templates provided.
- Check the README.
- test.sh must work at least 1%, time limit: 0.2s
- You can assume the secret is exactly 32 hex bytes
- Assume that we will not debug your program if it doesn't work or build "just to check what's wrong", your submission has to be reliable.

 No build with make or test.sh = fail.

Grading & Deadline

- Deadline:
 - Deadline Tuesday Nov 12, 23:59
 Delays: -1pt per late day
- Grading:
 - 7 ⇒ Meltdown with SEGV or TSX
 - 9 ⇒ Meltdown with SEGV <u>and</u> TSX
 - **11** ⇒ Meltdown in speculative execution

Note:

- Leakage accuracy and time will be taken into account (1% vs. 90%) and (1sec vs 10min)

Questions?

- Discussion board on Canvas
 - Help each other
 - Don't give away your solution

