Assignment 2: Flush + Reload

Meltdown & Spectre

Hardware Security 2020

Flush + Reload

FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack

Yuval Yarom and Katrina Falkner, *The University of Adelaide*

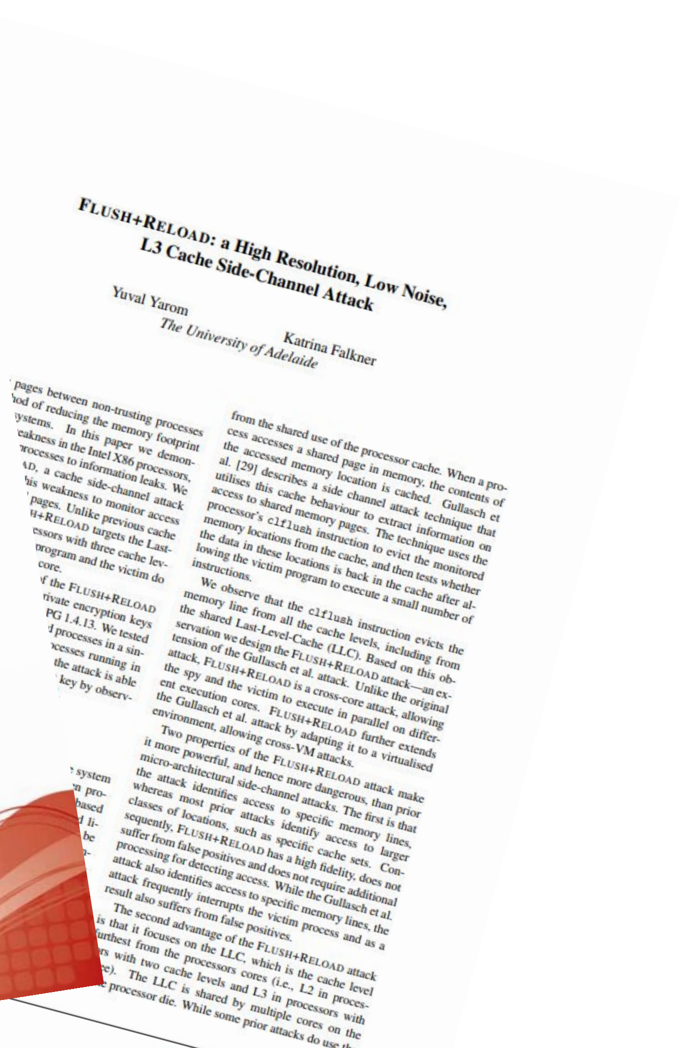
https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

This paper is included in the Proceedings of the 23rd USENIX Security Symposium.

August 20–22, 2014 • San Diego, CA

ISBN 978-1-931971-15-7

Open access to the Proceedings of the 23rd USENIX Security Symposium is sponsored by USENIX



FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack

Yuval Yarom Katrina Falkner
The University of Adelaide

pages between non-trusting processes
hod of reducing the memory footprint
ystems. In this paper we demon-
eakness in the Intel X86 processors,
rocesses to information leaks. We
AD, a cache side-channel attack
his weakness to monitor access
pages. Unlike previous cache
H+RELOAD targets the Last-
essors with three cache lev-
program and the victim do
core.
f the FLUSH+RELOAD
rivate encryption keys
PG 1.4.13. We tested
d processes in a sin-
ocesses running in
the attack is able
key by observ-

system
n pro-
based
d li-
be
n-

from the shared use of the processor cache. When a process accesses a shared page in memory, the contents of the accessed memory location is cached. Gullasch et al. [29] describes a side channel attack technique that utilises this cache behaviour to extract information on access to shared memory pages. The technique uses the processor's `clflush` instruction to evict the monitored memory locations from the cache, and then tests whether the data in these locations is back in the cache after allowing the victim program to execute a small number of instructions.

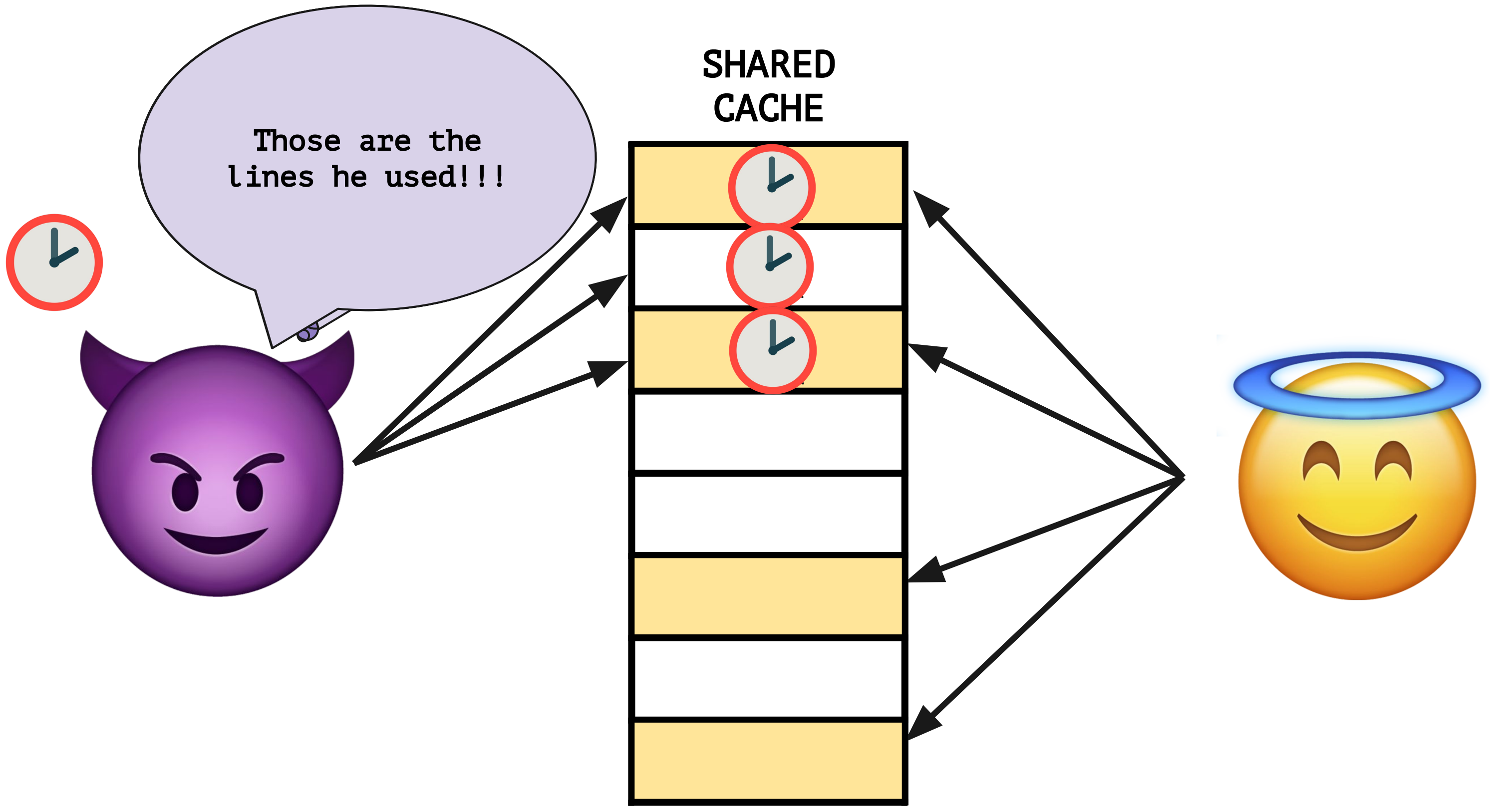
We observe that the `clflush` instruction evicts the memory line from all the cache levels, including from the shared Last-Level-Cache (LLC). Based on this observation we design the FLUSH+RELOAD attack—an extension of the Gullasch et al. attack. Unlike the original attack, FLUSH+RELOAD is a cross-core attack, allowing the spy and the victim to execute in parallel on different execution cores. FLUSH+RELOAD further extends the Gullasch et al. attack by adapting it to a virtualised environment, allowing cross-VM attacks.

Two properties of the FLUSH+RELOAD attack make it more powerful, and hence more dangerous, than prior micro-architectural side-channel attacks. The first is that the attack identifies access to specific memory lines, whereas most prior attacks identify access to larger classes of locations, such as specific cache sets. Consequently, FLUSH+RELOAD has a high fidelity, does not suffer from false positives and does not require additional processing for detecting access. While the Gullasch et al. attack also identifies access to specific memory lines, the attack frequently interrupts the victim process and as a result also suffers from false positives.

The second advantage of the FLUSH+RELOAD attack is that it focuses on the LLC, which is the cache level furthest from the processors cores (i.e., L2 in processors with two cache levels and L3 in processors with three). The LLC is shared by multiple cores on the processor die. While some prior attacks do use the

F+R paper

What



FLUSH + RELOAD

```
// remove the buff from the cache
for (i in 0..255)
     clflush(buff[i])

// bring back buff[secret[secret_offset]] to the cache
encode(buff, secret_offset)

// time every access
for (i in 0..255)
     time(buff[i])
```

Tells you which buff entry was loaded (cached) by encode()

Challenges

- **Prefetcher:** Optimizes memory accesses to hide latency. During Reload you will see cache hits not caused by encode()

- **Hint:** Check the lecture material

Bonus Task

You need to force the table entries out of the caches by means of eviction.

⇩

EVICT + RELOAD

Simply fill up the caches

Meltdown

aka CVE-2017-5754

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Meltdown: Reading Kernel Memory from User Space

Moritz Lipp, Michael Schwarz, and Daniel Gruss, *Graz University of Technology;* Thomas Prescher and Werner Haas, *Cyberus Technology;* Anders Fogh, *G DATA Advanced Analytics;* Jann Horn, *Google Project Zero;* Stefan Mangard, *Graz University of Technology;* Paul Kocher, *Independent;* Daniel Genkin, *University of Michigan;* Yuval Yarom, *University of Adelaide and Data61;* Mike Hamburg, *Rambus, Cryptography Research Division*

https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-939133-04-5

Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.

n: Reading Kernel Memory from User Space

Michael Schwarz[1], Daniel Gruss[1], Thomas Prescher[2], Anders Fogh[3], Jann Horn[4], Stefan Mangard[1], niel Genkin[6,9], Yuval Yarom[7], Mike Hamburg[8]
of Technology, [2]Cyberus Technology GmbH, nced Analytics, [4]Google Project Zero, aulkocher.com), [6]University of Michigan, a61, [8]Rambus, Cryptography Research Division

[Paper link](#)

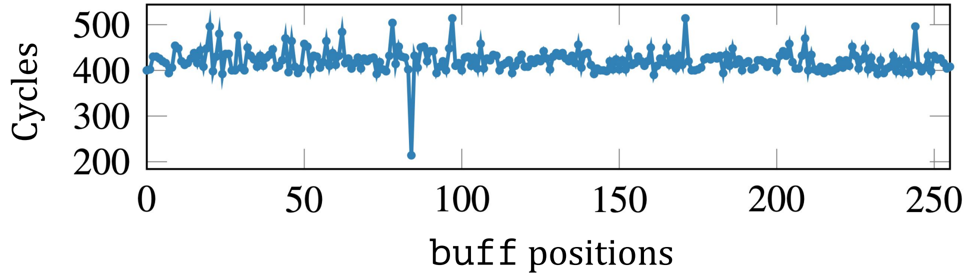
What?

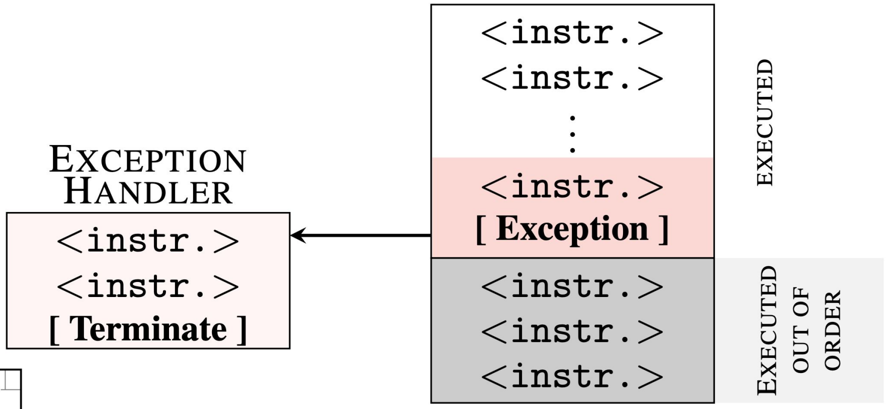
- An attack which exploits hardware optimizations.
- Breaks the isolation between
  - User and OS
  - User and User (e.g., JS sandbox)
- Allows a User program to read inaccessible kernel memory, and thus leak secrets of the OS and other User programs.

The idea

- Instructions are executed Out-of-Order.
- Even logically-after an exception has been raised.
- How can we observe **access(buff[0x72])** ?

```
raise_exception();
access(buff[0x72]);
```





Meltdown Core

```
;rcx = kernel address
;rbx = probe array
retry:
mov al, byte [rcx]
shl rax, 0xc
jz retry
mov rbx, qword [rbx + rax]
```

🟥 EXCEPTION
▨ OoO Execution

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

```
;rcx = kernel address
;rbx = probe array
retry:
mov al, byte [rcx]
shl rax, 0xc
jz retry
mov rbx, qword [rbx + rax]
```

Complication 2: Exception

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

```
;rcx = kernel address
;rbx = probe array
retry:
mov al, byte [rcx]
shl rax, 0xc
jz retry
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

```
;rcx = kernel address
;rbx = probe array
retry:
mov al, byte [rcx]
shl rax, 0xc
jz retry
mov rbx, qword [rbx + rax]
```

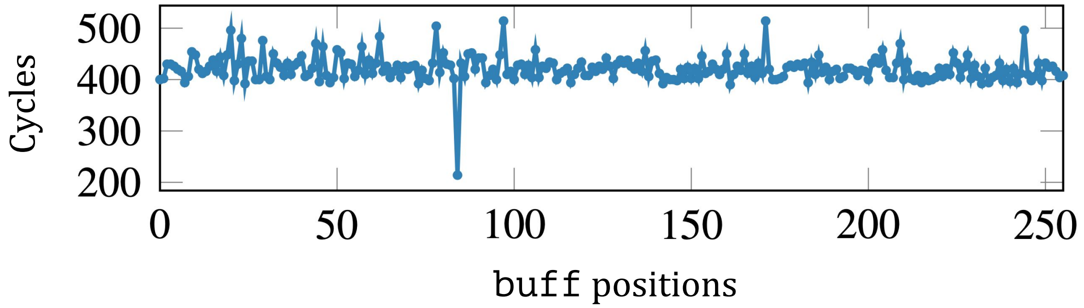
Complication 4: Prefetcher

- 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

```
;rcx = kernel address
;rbx = probe array
retry:
mov al, byte [rcx]
shl rax, 0xc
jz retry
mov rbx, qword [rbx + rax]
```

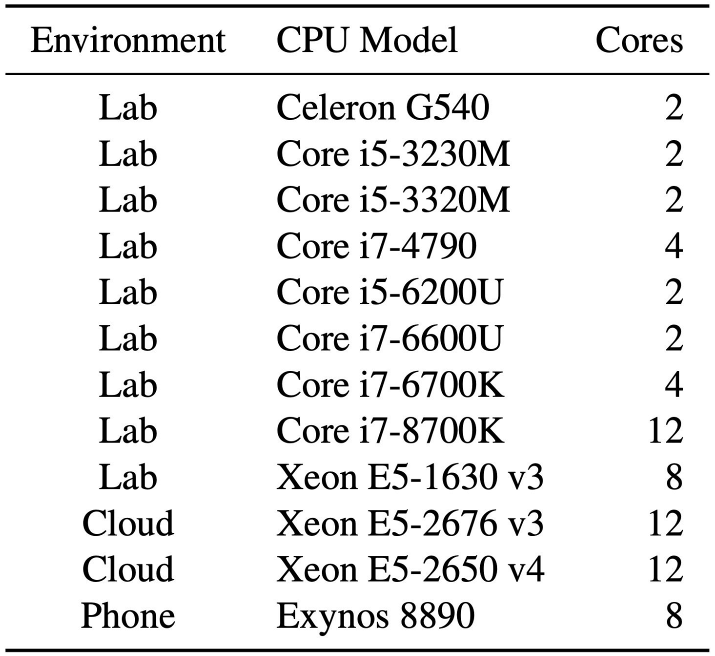
Complication 5: Threshold

- 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

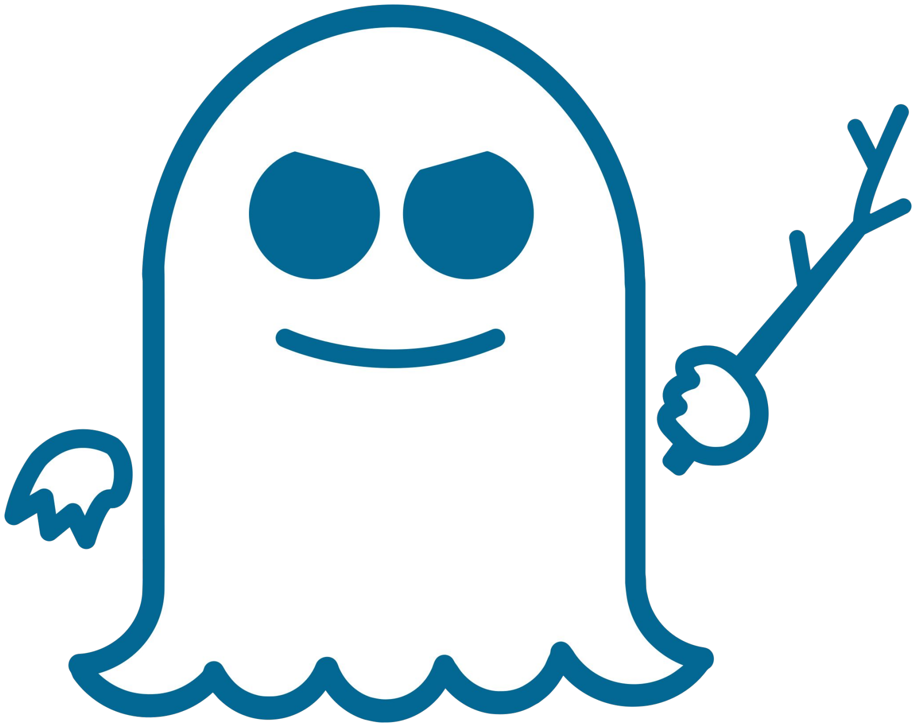
Note: Make sure your exploit work on our cluster of CPUs.

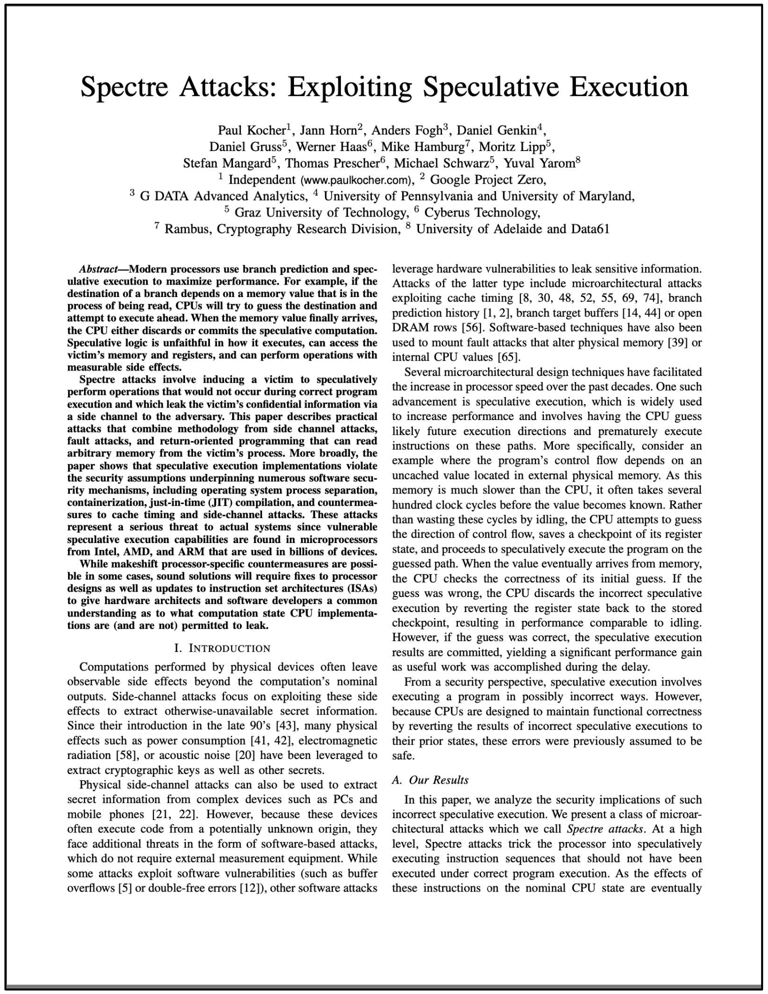
On other machines:

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

Spectre

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





Spectre Attacks: Exploiting Speculative Execution

Paul Kocher[1], Jann Horn[2], Anders Fogh[3], Daniel Genkin[4],
Daniel Gruss[5], Werner Haas[6], Mike Hamburg[7], Moritz Lipp[5],
Stefan Mangard[5], Thomas Prescher[6], Michael Schwarz[5], Yuval Yarom[8]
[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 process 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 security 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 possible 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.

I. Introduction

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 effects such as power consumption [41, 42], electromagnetic radiation [58], or acoustic noise [20] have been leveraged to 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 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 their prior states, these errors were previously assumed to be safe.

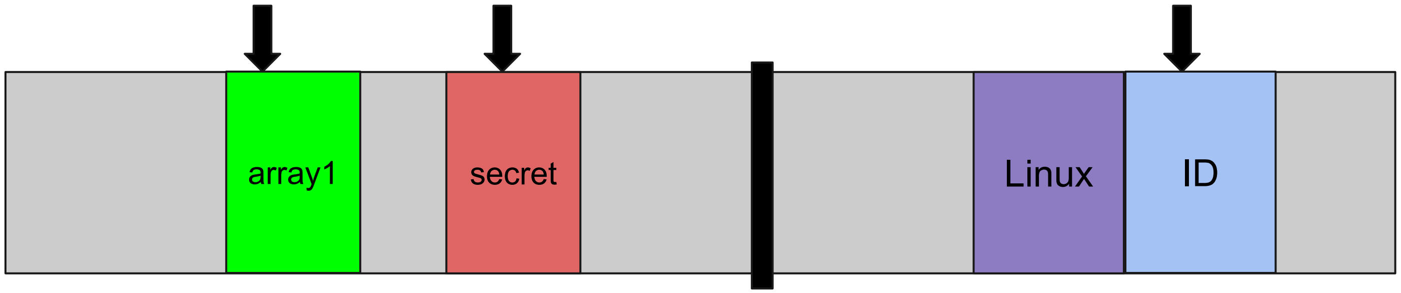
A. *Our Results*

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 these instructions on the nominal CPU state are eventually

Paper link

Spectre v1: Bounds Check Bypass

```
if (x < array1_size) {
    y = array2[array1[x] * 4096];
}
```



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

WOM Kernel Module

- 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/Spectre
- See template code in meltdown.c/spectre.c
- See also README for more info

Deliverable

- **Task #1**
  - Leak with Flush+Reload the secret encoded by the provided library
- **Task #2**
  - Implement Meltdown TSX and leak the secret contained in the WOM kernel module
- **Task #3:**
  - Implement Meltdown in a speculative branch and leak the same secret of Task #2
- **Bonus task:** Leak the same secret key of Task #1 with Evict+Reload

Grading & Deadline

- Deadline:
  - Deadline **Friday Nov 6th 2020 @ 23:59**
    Delays: -1pt per late day

- Grading:

5	⇒	Task	#1	Flush + Reload
9	⇒	Task	#2	Meltdown TSX
10	⇒	Task	#3	Meltdown Spec. Branch
11	⇒	Bonus		Evict + Reload

Questions?

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