

On the Meltdown & Spectre Design Flaws

Mark D. Hill

Computer Sciences Dept.
Univ. of Wisconsin-Madison

February 2018

Computer Architect, Not Security Expert

Prepared while on a sabbatical visit to Google with public information only and
representing the author's views only, not necessarily Google's.

Talk Info (Hidden Slide)

Title: On the Meltdown & Spectre Design Flaws

Speaker: Mark D. Hill, Computer Sciences Department, University of Wisconsin-Madison

Abstract: Two major hardware security design flaws--dubbed Meltdown and Spectre--were broadly revealed to the public in early January 2018 in research papers and blog posts that require considerable expertise and effort to understand. To complement these, this talk seeks to give a general computer science audience the gist of these security flaws and their implications. The goal is to enable the audience can either stop there or have a framework to learn more. A non-goal is exploring many details of flaw exploitation and patch status, in part, because the speaker is a computer architect, not a security expert.

In particular, this talk reviews that Computer Architecture 1.0 (the version number is new) specifies the timing-independent functional behavior of a computer and micro-architecture that is the set of implementation techniques that improve performance by more than 100x. It then asks, "What if a computer that is completely correct by Architecture 1.0 can be made to leak protected information via timing, a.k.a., micro-architecture?" The answer is that this exactly what is done by the Meltdown and Spectre design flaws. Meltdown leaks kernel memory, but software & hardware fixes exist. Spectre leaks memory outside of sandboxes and bounds check, and it is scary. An implication is that the definition of Architecture 1.0--the most important interface between software and hardware--is inadequate to protect information. It is time for experts from multiple viewpoints to come together to create Architecture 2.0).

Bio: Mark D. Hill (<http://www.cs.wisc.edu/~markhill>) is John P. Morgridge Professor and Gene M. Amdahl Professor of Computer Sciences at the University of Wisconsin-Madison. Hill has a PhD in computer science from the University of California, Berkeley. Hill's research targets computer design and evaluation. He has made contributions to parallel computer system design (e.g., memory consistency models and cache coherence), memory system design (caches and translation buffers), computer simulation (parallel systems and memory systems), software (e.g., page tables and cache-conscious optimizations), deterministic replay and transactional memory. For example, he is the inventor of the widely-used 3C model of cache behavior (compulsory, capacity, and conflict misses) and co-inventor of the cornerstone for the C++ and Java multi-threaded memory specifications (sequential consistency for data-race-free programs). He is a fellow of IEEE and the ACM. He serves as Vice Chair of the Computer Community Consortium (2016-18) and served as Wisconsin Computer Sciences Department Chair 2014-2017.

Executive Summary

Architecture 1.0: the timing-independent functional behavior of a computer

Micro-architecture: the implementation techniques to improve performance

Question: What if a computer that is completely correct by **Architecture 1.0** can be made to leak protected information via timing, a.k.a., **Micro-Architecture**?

Meltdown leaks kernel memory, but software & hardware fixes exist



Spectre leaks memory outside of bounds checks or sandboxes, and is **scary**

Implication: **The definition of Architecture 1.0 is inadequate to protect information**

Outline

Computer Architecture & Micro-Architecture Background

Timing Side-Channel Attack

Meltdown

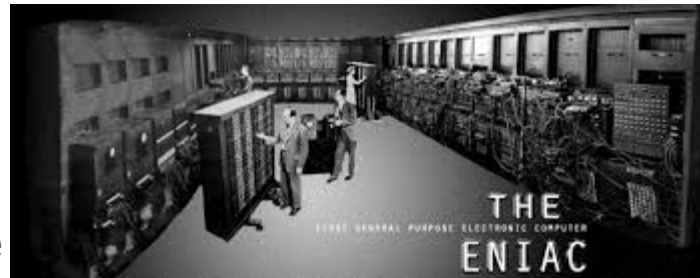
Spectre

Wrap-Up

Computer Architecture 0.0 -- Pre-1964

Each Computer was New

- Implemented machine (has mass) → hardware
- Instructions for hardware (no mass) → software



Software Lagged Hardware

- Each new machine design was different
- Software needed to be rewritten in assembly/machine language
- Unimaginable today

Going forward: Need to separate HW interface from implementation

Computer Architecture 1.0 -- Born 1964

IBM System 360 defined an **instruction set architecture**

```
1 branch (R1 >= bound) goto error
2 load R2 ← memory[train+R1]
3 and R3 ← R2 && 0xffff
4 load R4 ← memory[save+SIZE+R3]
```



- Stable interface across a family of implementations
- Software did NOT have to be rewritten

Architecture ^{μArch} 1.0 ^{API} the timing-independent functional behavior of a computer

Micro-architecture: implementation techniques that change timing to go fast

Note: The code is not IBM 360 assembly, but is the example used later.

μArch

Micro-architecture Harvested Moore's Law Bounty

For decades, every ~2 years: 2x transistors, 1.4x faster & 1x chip power possible;
2300 transistors for Intel 4004 → millions per core & billions for caches

(Micro-)architects took this ever doubling budget to make each processor core execute > 100x than what it would otherwise.

Key techniques w/ tutorial next:

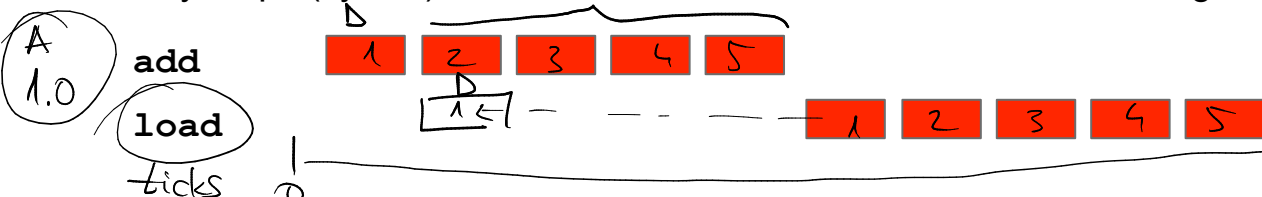
- Instruction Speculation
- Hardware Caching

Hidden by Architecture 1.0: timing-independent functional behavior unchanged

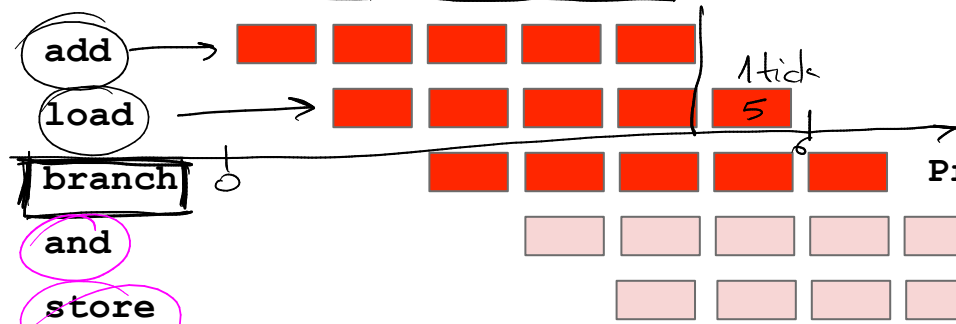


Instruction Speculation Tutorial

Many steps (cycles) to execute one instruction; time flows left to right →



Go Faster: Pipelining, branch prediction, & instruction speculation



Predict direction: target or fall thru

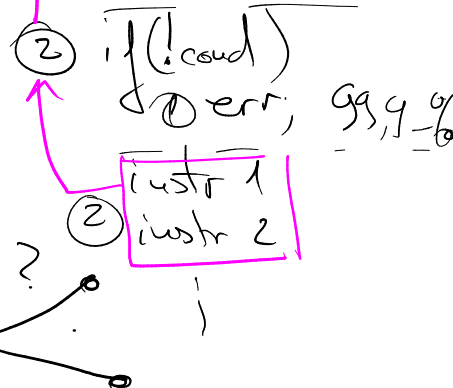
Speculate!

Speculate more!

Speculation correct: Commit architectural changes of and (register) & store (memory) go fast!

Mis-speculate: Abort architectural changes (registers, memory); go in other branch direction

pipeline



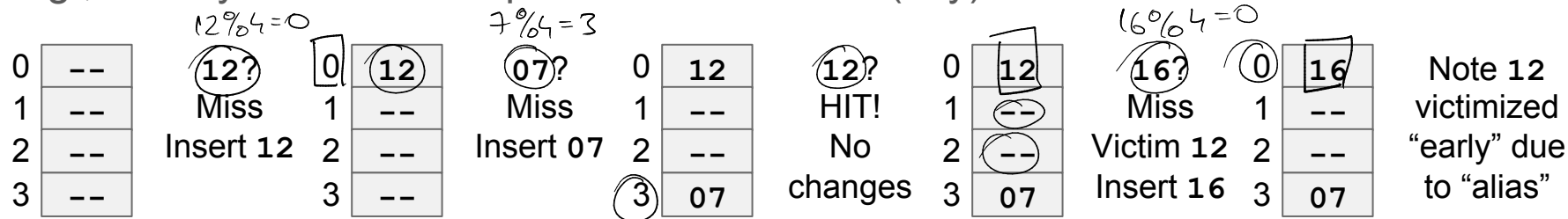
Hardware Caching Tutorial

Main Memory (DRAM) 1000x too slow

Add Hardware Cache(s): small, transparent hardware memory

- Like a software cache: speculate near-term reuse (locality) is common
- Like a hash table: an item (block or line) can go in one or few slots

E.g., 4-entry cache w/ slot picked with address (key) modulo 4



Micro-architecture Harvested Moore's Law Bounty

For decades, every ~2 years: 2x transistors, 1.4x faster & 1x chip power possible;
2300 transistors for Intel 4004 → millions per core & billions for caches

(Micro-)architects took this ever doubling budget to make each processor core
execute > 100x what it would otherwise

```
branch (R1 >= bound) goto error ; Speculate branch not taken
load R2 ← memory[train+R1]      ; Speculate load & speculate cache hit
and R3 ← R2 && 0xffff           ; Speculate AND
load R4 ← memory[save+SIZE+R3]  ; Speculate load & speculate cache hit
```

Hidden by Architecture 1.0: timing-independent functional behavior unchanged

Whither Computer Architecture 1.0?

Architecture 1.0: timing-independent functional behavior

Question: What if a computer that is completely correct by **Architecture 1.0** can be made to leak protected information via timing, a.k.a., **micro-architecture**?

Implication: **The definition of Architecture 1.0 is inadequate to protect information**

This is what Meltdown and Spectre do. Let's see why and explore implications.

Side-Channel Attack: **SAVE** Secret in Micro-Arch

1. Prime micro-architectural state

- Repeatedly access array `train[]` to train branch predictor to expect access `< bound`
- Access all of array `save[]` to put it completely in a cache of size `SIZE`

2. Coerce processor into **speculatively executing** instructions that will be nullified to (a) find a secret & (b) save it in micro-architecture

```
branch (R1 >= bound) goto error ; Speculate not taken even if R1 >= bound  
load R2 ← memory[train+R1]      ; Speculate to find SECRET outside of train[]  
and R3 ← R2 && 0xffff           ; Speculate to convert SECRET bits into index  
load R4 ← memory[save+SIZE+R3] ; Speculate to save SECRET by victimizing  
memory[save+R3] since it aliases in cache with new access memory[save+SIZE+R3]
```

3. HW detects **mis-speculation**

Undoes **architectural** changes

Leaves cache (**micro-architecture**) changes (correct by **Architecture 1.0**)

Side-Channel Attack: **RECALL** Secret from Micro-Arch

4: Probe **time** to access each element of `save[]` --**micro-architectural** property;
If accessing `save[foo]` slow due to cache miss, then SECRET is `foo`. A leak!

5: Repeat many times to obtain secret information at some bandwidth. (More shifting/masking needed to get all SECRET bits victimizing 64B cache lines)

Well-known in 1983/85 DoD “Orange Book”

Covert timing channels include all vehicles that would allow one process to signal information to another process by modulating its own use of system resources in such a way that the change in response time observed by the second process would provide information. --TRUSTED COMPUTER SYSTEM EVALUATION CRITERIA

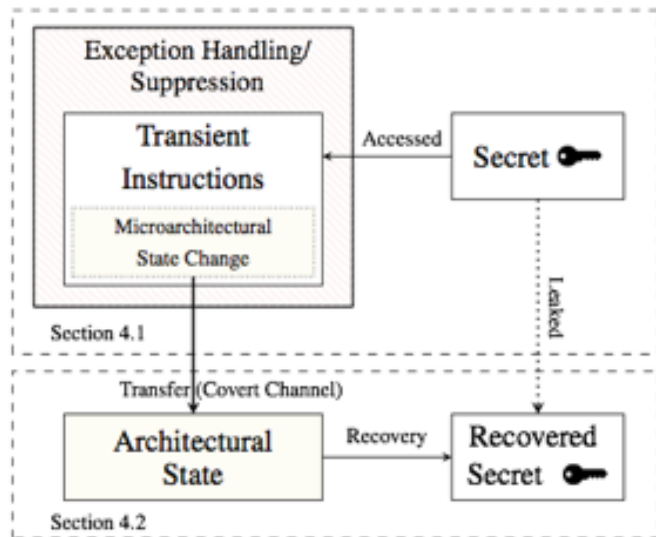
With roots back to 1974 TENEX password attack
But seemed fanciful



Spy vs. Spy, Mad Magazine, 1960

Meltdown (<https://meltdownattack.com/meltdown.pdf>)

Can leak the contents of kernel memory at up to 500KB/s



```
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]
```

TRAP!! (not branch)
Under mis-speculation

Listing 2: The core instruction sequence of Meltdown. An inaccessible kernel address is moved to a register, raising an exception. The subsequent instructions are already executed out of order before the exception is raised, leaking the content of the kernel address through the indirect memory access.

Meltdown & Hardware

Demonstrated for many Intel x86-64 cores; NOT demonstrated for AMD

Key: When to suppress load with protection violation (user load to kernel memory)

- EARLY: AMD appears to suppress early, e.g., at TLB access
- LATE: Intel appears to suppress at end after micro-arch state changes

My SWAG (Scientific Wild A** Guess) Why

- Both are correct by Architecture 1.0
- Performance shouldn't matter as this case is supposed to be rare
- Do what's easiest & have luck that is good (AMD) or bad (Intel)

Meltdown & Software

Bad: Meltdown operates with bug-free OS software (by [Architecture 1.0](#))

Good: Major commercial OSs patched for Meltdown ~January 2018

Idea: Don't map (much) of protected kernel address space in user process

- Offending load now fails address translation & does nothing
- Patches quickly derived from KAISER developed for side-channel attacks of Kernel Address Space Layout Randomization (KASLR)
- Performance impact 0-30% syscall frequency & core model.

Future hardware can fix Meltdown (like AMD) so maybe we dodged a bullet

Spectre (<https://spectreattack.com/spectre.pdf>)

Classic side-channel attack w/ deep micro-arch info



- 1. Attacker primes micro-architecture
 - E.g, branch predictor or branch target buffer for saving secret
 - E.g., cache for recalling secret
- 2: Victim loads secret under mis-speculation
 - Load should NOT trap (unlike Meltdown)
 - Still inappropriate if managed language or sandbox
- 3: Victim saves secret in micro-arch state, e.g., cache
- 4: Attacker recalls secret from micro-arch state; 4: repeat.

Spectre Applicability (Paper Sections 4, 5, & 6)

4. Exploit branch mis-prediction to let Javascript steal from Chrome browser
 - Demonstrated Intel Haswell/Skylake, AMD Ryzen, & several ARM cores
 - Many other existing designs vulnerable
5. Used indirect branches & return-oriented programming to mis-train branch target buffer to obtain information from different hyper-thread on same core
6. Many other known timing-channel exist, e.g., register file contention, functional unit occupancy, **but what about unknown timing channels?**

Spectre Mitigation (Section 7)

Branch prediction

- SW: Suppress branch prediction “when important” with **mfence**, etc.
- Not defined to work but appears to work--at a performance cost
- HW could auto-magically suppress branch prediction when appropriate (???)

Branch Target Buffer

- SW: Not clear. Disable hyper-threading, etc.?
- HW: Make micro-architecture state private to thread (not core or processor)

More generally: Hard to mitigate threats NOT YET DEFINED.

Need Computer Architecture 2.0?

With Meltdown & Spectre, [Architecture 1.0](#) is inadequate to protect information

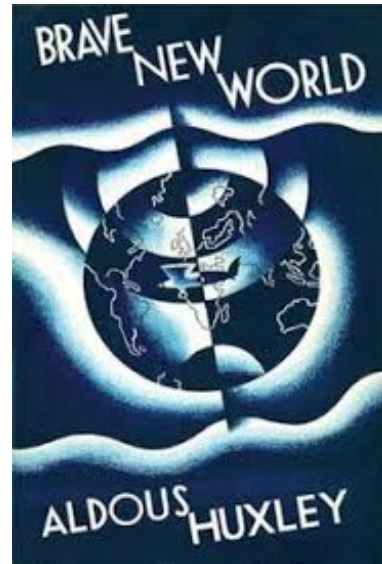
Augment [Architecture 1.0](#) with [Architecture 2.0](#) specification of

- (Abstraction of) time-visible micro-architecture?
- Bandwidth of known (unknown?) timing channels?
- Enforced limits on user software behavior? (c.f., KAISER)

Change [Microarchitecture](#) to mitigate timing channel bandwidth

- Suppress some speculation
- Undo most changes on mis-speculation

Can this be (formally) solved or must it be managed like crime?



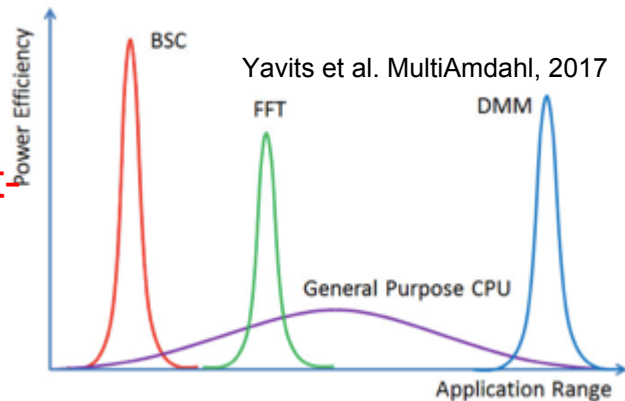
Need Computer Architecture 2.0?

More generally, can we reduce our dependence on SPECULATION?

Accelerators!! GPU, DSP, IPU, TPU, ... [Hennessy & Patterson 2018 Taxonomy]

- Dedicated Memories
- More ALUs
- Easy Parallelism
- Lower precision data
- Domain Specific Language

Speculation NOT a first order feature!



In 2005, Arvind said Speculation (w/ von Neumann model) killed Dataflow

After 2018, Dataflow-like Renaissance w/ Sea of Accelerators?

Executive Summary

Architecture 1.0: the timing-independent functional behavior of a computer

Micro-architecture: the implementation techniques to improve performance

Question: What if a computer that is completely correct by **Architecture 1.0** can be made to leak protected information via timing, a.k.a., **Micro-Architecture**?

Meltdown leaks kernel memory, but software & hardware fixes exist



Spectre leaks memory outside of bounds checks or sandboxes, and is **scary**

Implication: **The definition** of Architecture 1.0 is inadequate to protect information

Some References

New York Times: <https://www.nytimes.com/2018/01/03/business/computer-flaws.html>

Meltdown paper: <https://meltdownattack.com/meltdown.pdf>

Spectre paper: <https://spectreattack.com/spectre.pdf>

A blog separating the two bugs: <https://danielmiessler.com/blog/simple-explanation-difference-meltdown-spectre/>

Google Blog: <https://security.googleblog.com/2018/01/todays-cpu-vulnerability-what-you-need.html> and <https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html>

Industry News Sources:

<https://arstechnica.com/gadgets/2018/01/whats-behind-the-intel-design-flaw-forcing-numerous-patches/> and https://www.theregister.co.uk/2018/01/02/intel_cpu_design_flaw/

Backup Slides

Spectre Code Example



Listing 2: Exploiting Speculative Execution via JavaScript

```
1 if (index < simpleByteArray.length) {  
    2 index = simpleByteArray[index | 0];  
    3 index = (((index * TABLE1_STRIDE)|0) & (TABLE1_BYTES-1))|0;  
    4 localJunk ^= probeTable[index|0]|0;  
5}
```

Listing 3: Disassembly of Speculative Execution in Listing 2 JavaScript

```
1 cmpl r15,[rbp-0xe0] ; Compare index (r15) against simpleByteArray.length  
2 jnc 0x24dd099bb870 ; If index >= length, branch to instruction after move below  
3 REX.W leaq rsi,[r12+rdx*1] ; Set rsi=r12+rdx=addr of first byte in simpleByteArray  
4 movzxb rsi,[rsi+r15*1] ; Read byte from address rsi+r15 (= base address+index)  
5 shll rsi, 12 ; Multiply rsi by 4096 by shifting left 12 bits\%\%\  
6 andl rsi,0x1ffffff ; AND reassures JIT that next operation is in-bounds  
7 movzxb rsi,[rsi+r8*1] ; Read from probeTable  
8 xorl rsi,rdi ; XOR the read result onto localJunk  
9 REX.W movq rdi,rsi ; Copy localJunk into rdi
```

Meltdown v. Spectre

	 MELTDOWN	 SPECTRE
<i>Architecture</i>	Intel, Apple	Intel, Apple, ARM, AMD
<i>Entry</i>	Must have code execution on the system	Must have code execution on the system
<i>Method</i>	Intel Privilege Escalation + Speculative Execution	Branch prediction + Speculative Execution
<i>Impact</i>	Read kernel memory from user space	Read contents of memory from other users' running programs
<i>Action</i>	Software patching	Software patching (more nuanced)

Daniel Miessler 2018

Miessler Blog (<https://danielmiessler.com/blog/simple-explanation-difference-meltdown-spectre/>)