Spectre/Meltdown &
What it means for future design

Hotchips Plenary Keynote Session
August 20th 1:45pm to 3:30PM

Session chair: Partha Ranganathan, Google

Spectre/Meltdown & what it means for future design

Plenary Keynote Session, Session chair: Partha Ranganathan, Google

The Era of Security, John Hennessy, Chairman BoD Alphabet; Stanford

Spectre/Meltdown: the project Zero journey, Paul Turner, Google

Exploiting modern µarchitectures: software implications, Jon Masters, Red Hat

Exploiting modern µarchitectures: hardware implications, Mark Hill, UWisc-Mad

Panel Q&A

The Era of Security

John Hennessy

Introduction: Security is more important than ever!

- **Lots more personal information available online.**
- **Cloud world means that strangers (even adversaries) are sharing HW**
- **State actors and organized cyber criminals means attacks are**
  - **Increasing**
  - **Better organized and more technically capable**
- **Complexity of modern systems means more surface area to attack**
  - **Growth in complexity has outstripped growth in security mechanisms**
  - **Most attacks are still SW-focused (e.g. buffer overflow)**

Side channel Attacks

- **Side channel attacks are not new**
- **1970s exploit holes in OS security to**
  - **Crack passwords: guess passwords one letter at a time by placing a page break between known password letters and unknown**
  - **Transfer information across a secure domain by changing VM behavior**
- **What's new: HW (rather then SW) security hole**
  - **Meltdown and Spectre (multiple versions) are just first of possible architectural holes**
    - **L1TF and Foreshadow: break the VM protection (including SGX).**
  - **Painful to fix in SW and significant performance losses possible**

Leveraging speculation in side channel attack

- **Speculation**
  - **Execute instructions early before you know that are definitely needed**
  - **Buffer results until instruction execution is certain (commit) than change the state**
  - **Maintain precise exceptions: delay any encountered exception until instruction is for certain**
    - **Because instructions are speculated they may appear to cause an exception that doesn't really exist because the instruction will be discarded.**
- **The Hole that Meltdown and Spectre Exploit:**
  - **Speculation can change the microarchitectural state**
  - **Can be observed by via the side channel.**

Meltdown: an outline

Setup:

1. Clear 256 blocks in cache at address X.
2. Prime the branch predictor for B to NOT TAKEN.

Steal a byte:

```
beq   r0,r1,skip   ; branch will be taken this time
lb   r1, address of protected data   ; speculative instruction--cancelled
lw r3, (X + r1)      ; speculative instruction--cancelled, BUT cache miss generated
```

Extract the protected byte:

Examine 256 cache locations at X, one of them will hit = value of stolen byte.

NetSpectre

- **Exploit the Spectre v1 hole without running any code!**
  - **Break-in from a remote machine via LAN or within the cloud**
- **NetSpectre: leak gadget and transmit gadget**
- **Leak gadget:**
  - **Send “valid” packets to train predictor in the networking code**
  - **Send “invalid” packet & leak data in shadow**
- **Transmit gadget: cannot observe state directly.**
  - **Use network request to change/measure cache state.**
  - **Can also leak through memory, I/O, and AVX instructions.**
- **“Noisy” chanel: leaks 0.25-1 bit/minute but attack is completely remote**

Security Challenges

1. **Software flaws:**
    a. HW's job: reduce these and minimize damage
    b. Support function, which works if programmers take advantage
        i. Must be effective
        ii. Must be efficient

2. **Hardware flaws**
    a. Cannot allow this--no matter how much performance could be gained!
    b. Hard to fix and fixes may cost more than the "HW optimization" gained.
        i. Next generation Intel processors will probably not fix Spectre v1.
    c. Lots of us missed this problem and for about 10-15 years.

Spectre/Meltdown: The Project Zero Journey

Paul Turner

Google Project Zero (GPZ)

Internal Google security team founded in 2014.
Goal: *Reducing the harm caused by attacks on the internet; focus on ‘zero-days’.*

During 2017, Jann Horn, a GPZ researcher, co-discovered a new class of vulnerabilities that allowed data escape from within speculative execution.

These vulnerabilities internally became known as, **“SpeckHammer”**.

“Numbers everyone should know”

L1 cache reference 0.5 ns
Branch mispredict 5 ns
L2 cache reference 7 ns
Mutex lock/unlock 100 ns
Main memory reference 100 ns
Compress 1K bytes with Zippy 10,000 ns
Send 2K bytes over 1 Gbps network 20,000 ns
Read 1 MB sequentially from memory 250,000 ns
Round trip within same datacenter 500,000 ns
Disk seek 10,000,000 ns
Read 1 MB sequentially from network 10,000,000 ns
Read 1 MB sequentially from disk 30,000,000 ns
Send packet CA->Netherlands->CA 150,000,000 ns

“Numbers everyone should know” ... *that the CPU tries to hide*

L1 cache reference 0.5 ns
Branch mispredict 5 ns
L2 cache reference 7 ns
Mutex lock/unlock 100 ns
Main memory reference 100 ns

- **Overlap** work while waiting for **slower** accesses and operations.
- Use predictors. Speculate. Increase overlap.

“Speck” is for Speculation

Almost all high-performance processors optimize the execution of conditional branches and memory stalls by “speculating” and predictively pre-executing the expected subsequent instructions.

If the prediction **is** correct:

- The speculative execution becomes visible and is immediately retired.

If the prediction is **not** correct:

- The speculative execution is discarded.
- It cannot be observed and has no side-effects.

“Speck” is for Speculation

Almost all high-performance processors optimize the execution of conditional branches and memory stalls by “speculating” and predictively pre-executing the expected subsequent instructions.

If the prediction **is** correct:

- The speculative execution becomes visible and is immediately retired.

If the prediction is **not** correct:

- The speculative execution is discarded.
- ~~*It cannot be observed and has no side effects.*~~

Our journey begins: The New Now

- Speculative side-effects previously thought to be “data”-free.
- New results fundamentally break these assumptions.

- **Worse, they’ve existed for ~20 years!**

Variant 1: Bounds Check Bypass

```
if (arbitrary_offset < array_len) {
    val = private_memory[arbitrary_offset]
    x = accessible_memory[(val & 1)*cache_line_size]
}
```

While our conditional is resolved, we may **speculatively** execute the inner block.

- This execution will be discarded when the conditional is resolved …
- **… But,** timing subsequent loads on accessible_memory[0] exposes bits of private_memory.

- **Variant 1: Caller attacks gadgets using accesible_memory side channels**

Variant 1: The building block

This turns out to be a generic idea

- Misspeculate on ***some*** boundary, intended to ***restrict*** execution
- Speculative execution now proceeds within a ***restricted*** space
- Extract information **from** a restricted space, into a shared one

Variant 1: In Practice

Two key examples

- **Browsers:** JITs run in a shared address space. Here, private_memory could be another tab or website; it can potentially use its own execution environment for an **accesible_memory** side-channel.
  - *Resolution: Site isolation, all tabs in their own process*
- **Kernels and Hypervisors:** User-space (or a guest) is usually directly mapped, providing a convenient **accesible_memory** side-channel. private_memory typically includes all host memory!

Variant 3: Rogue Data Cache Load (*aka Meltdown*)

```
if (0 < 1) {    // We'd fault if we actually retired below.
   val = kernel_memory[arbitrary_offset]
   x = user_memory[(val & 1)*64]
}
```

- Operating systems use page-table protections to isolate user memory…
- **… these protections are also speculatively evaluated!**
- **Allows inline variant1-attack on kernel memory, from user-space.**
- Meltdown….

Variant 2: Branch Target Injection

```
4148c0:   object->Foo();

...

9812ab:   val = private_memory[untrusted_large_offset]
           x = accessible_memory[(val & 1)*64]
```

- For **indirect calls** the processor must speculate on Foo()'s address
- It uses the instruction address (`4148c0`). This prediction can be **poisoned**, allowing us to speculatively execute code that would leak private_memory.

- Predictors shared between host/guest, **crosses VM boundary**!
- Predictors shared between Hyper Threads -- shared core structures

Where does mitigation start?

Despite GPZ discovery, Google's mitigation does not get "head start".

We're faced with two fundamental classes of challenge:

- How can we **restore** hardware "boundaries" where possible?

- How can we **introduce** boundaries that the hardware can understand, where they did **not previously exist**?

Mitigation: Meltdown/Variant 3

KPTI: Kernel Page Table Isolation

- Instead of depending on page-table protection boundaries for user/kernel separation; separate the page tables.
  - Now, no translation exists to misspeculate against.
- Hardware features such as PCIDs turn out to be useful again (x86 case)!

Here, we were able to change the software implementation to **restore** our original protections.

Mitigation: Spectre/Variant 2

Returns are a special type of indirect branch. As **ret** always pairs with ***call***, the hardware implements a dedicated predictor: the *Return Stack Buffer (RSB).*
When non-empty, the RSB always controls ret speculation.

- RSB shadow and on-stack values are separately maintained → we can use shadow to control speculative execution ***and guarantee it's not exploitable*** While using the on-stack execution to control retired execution.
- *RSB is per hyper-thread sibling, also prevents cross-core attack!*
- Translate indirect branches to use **ret** instead of **call** or **jmp**.

Intel validates this prevents all known attacks (provided RSB is non-empty)

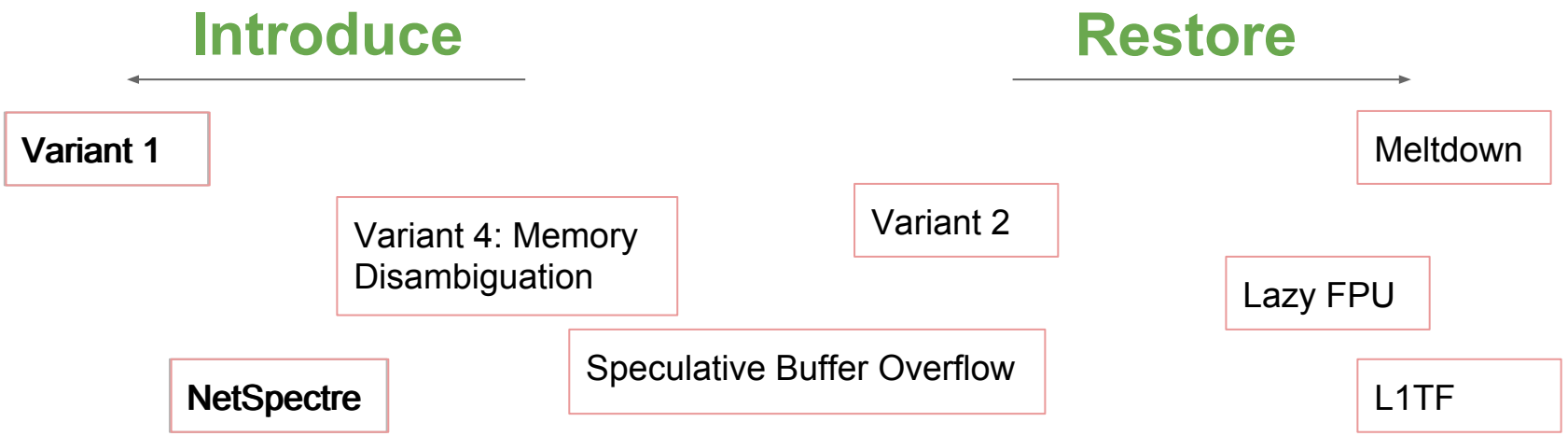
Mitigation: Spectre/Variant 2

- By introducing alternate software constructs; we were again able to **restore** the isolation we previously assumed exists between execution context and SMT sibling pairs. But this can also be considered constructive, we effectively **introduced** a new “restricted-speculation” branch type.

- Intel microcode mitigation **introduced** new restricted speculation execution mode.

Important distinction here is reconfiguring processor itself has a transition cost in addition to run-time overhead.

And the list keeps growing …



But what about variant 1?

- Where in the cases prior; there was a boundary that was previously assumed to exist. For variant 1, in-address space attacks. **We don’t have this luxury.**
- No existing constructs which allow software to **advertise** to the hardware the different **principals** (e.g. security “roles”) which might exist within a single address space.
    - Examples: Different websites within a browser, JIT vs JVM,
      Hypervisor vs encapsulating kernel
- Maybe we can use ideas similar to KPTI (e.g. site isolation). But what else can we do? How can we start to **introduce** these boundaries so that the hardware can understand?

Exploiting Modern µarchitectures: Software Implications

Jon Masters

How did we get here?

- “Hardware” and “software” people don’t talk
  - We created this almost comedic world over the past few decades in which the “us” vs “them” is pervasive
  - The two camps are distinct. We take some perverse pleasure in explicitly never communicating together
  - We build hardware or software before discussing whether what we are building is good for each other
  - Exceptions exist, but they are **NOT** the norm.

How did we get here?

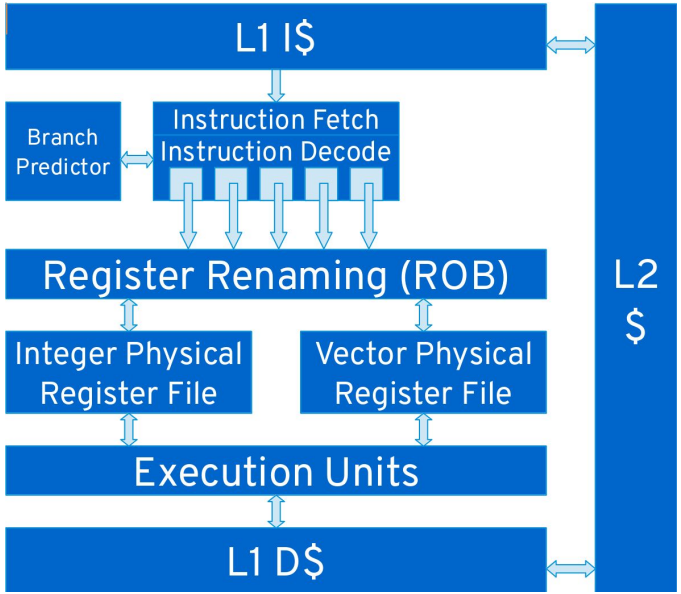
- Early hw implementations were simpler
  - This created implicit assumptions on both sides
  - The ISA contract between hardware and software…
    - ...was insufficiently defined
  - Programmers assume simple sequential execution
  - Assumptions were created but never codified
  - We kept building more layers upon layers...

How did we get here?

- Programming became much more abstract
  - In the early days, it was necessary to program much closer to the metal (and to understand it)
  - Most programmers today use higher level languages
    - Some have no idea what a stack or branch is
    - This helped build the amazing world we live in
    - But it means many (untrue) assumptions exist
  - Shared tenancy increased attack surface
    - We built the world in a much simpler time
    - Most of our isolation efforts came later

How did we get here?

- We demanded continual IPC gains every year
  - Many of the cheap wins in 52% YoY growth phase
  - Implementations became much more aggressive
    - Added speculation (e.g. branch prediction)
    - Further optimizations (valid/access bits)
  - Speculation treated as magic black box
    - Do everything at retirement or in parallel
  - Users and customers didn't ask (any) questions
    - "Nobody" cared where the gains came from

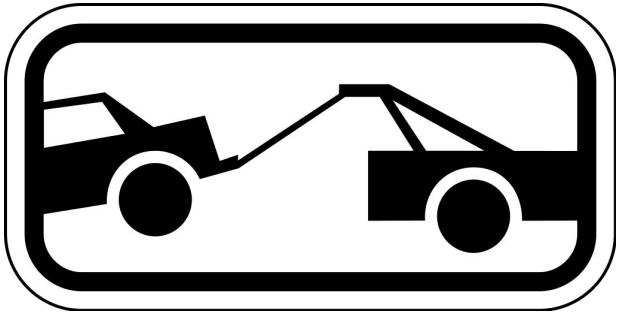


How did we get here?

- Implementations became incredibly complex
  - ...vs. programmer assumed sequential model
  - Many of today's programmers have no idea about…
    - ...instruction re-ordering or speculation
    - ...memory consistency models
    - ...
  - Machine microarchitects had implicit notions…
    - ...what software "should" do
    - ...but they didn't bother to enforce this either

How did we get here?

- Programmers don’t understand hardware
  - ...hardware is not magic, it has many limitations
  - Microcode does not fix every problem
    - Can patch some instructions, not critical path
    - Can’t change page walker or cache, or….
    - Same goes for millicode, or chicken bits
  - We are too used to how good we have had it
    - Reality is very far from this, hard limits exist

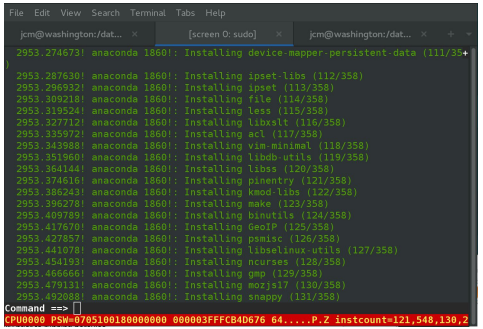


Summary: How did we get here?

- The “us” vs “them” became so ingrained we forgot how to collaborate
  - Most programmers negatively care about hardware, which is seen as a boring commodity
  - Software architects and hardware microarchitects don’t talk ahead of implementing new features, but instead build their view of the world and (maybe) reconcile it afterward
- Previous vertical system model gave way to separate hw/sw companies
  - Hardware folks design processors (and interconnects, and other platform pieces)
  - Platform-level capability was gradually eroded from outside processor vendors
  - The focus on security has actually been a positive from this perspective
- Renaissance in computer architecture brings us a new hope
  - Increasing need to understand a vertical stack from hardware to software
  - Focus on security has proven the need to understand how hw works

“Concerning modern microarchitectures”...

- Spectre and Meltdown unnoticed for several decades
  - These were multi-architectural issues impacting every vendor
  - Meltdown is often labeled as an “Intel” issue, but others impacted
  - Collaboration involved every major vendor across many arches
- Industry proactively tracks various security research
  - e.g. PRIME+PROBE cache side-channel attacks (e.g. crypto PoV)
  - Anders Fogh blog post last summer set off few alarm bells
  - Most people **didn’t believe** Meltdown could be possible



Screenshot: Installing a mainframe emulator over the holidays to test Spectre mitigations on s390x (z/Arch)

Example vendor response strategy

- We were on a specific timeline for public disclosure (which was good)
  - Limited amount of time to create, test, and prepare to deploy mitigations
  - Focus on mitigating the most egregious impact first, enhance later
  - Report/Warn the level of mitigation to the user/admin
- Created “Omega” Team for microarchitecture vulnerabilities
  - Collaborate with others across industry and upstream on mitigations
  - Backport those mitigations (with tweaks as needed) to Linux distros
    - Example: RH did 15 kernel backports, back to Linux 2.6.18
    - Other companies/vendors did similar numbers of patches
- Produce materials for use during disclosure
  - Blogs, whitepapers, performance webinars, etc.
  - The “X in 3 minutes” videos intended to be informative
- Meltdown and Spectre alone cost 10,000+ hours Red Hat engineering time

In the field - Microcode, Millicode, Chicken Bits...

- Modern processors are designed to be able to handle (some) in-field issues
- Microcoded processors leverage “ucode” assists to handle certain operations
  - Ucode has existed for decades, adopted heavily by Intel following (infamous) “FDIV” bug
  - Not a magic bullet. It only handles certain instructions, doesn’t do page table walks, cache loads, and other critical path operations, or simple instructions (e.g. an “add”)
  - OS can vendors ship signed blobs provided by e.g. Intel and AMD and loaded by the OS
- Millicode is similar in concept to Microcode (but specific to IBM)
  - We secretly deployed updates internally during the preparation for disclosure
- Chicken bits are used to control certain processor logic, and (de)features
  - RISC-based machines traditionally don’t use ucode but can disable (broken) features
  - Contemporary x86 processors also have on the order of 10,000 individual chicken bits
- Everything else needs to be done in software (kernel, firmware, app…)

Mitigation - Meltdown

- Two elements needed for a successful attack:
  - Valid page table (VA->PA) translation for the virtual memory area
  - Secret data must be present in caches, speculatively forwarded
- Two paths exist for mitigation of Meltdown:
  - Remove VA translations (Page Table Isolation, formerly “KAISER”)
  - Flush the data cache(s) of secrets so there are none to steal
  - The precise path chosen varies by microarchitecture
- Linux generally uses PTI (Page Table Isolation)
  - Added trampoline code for kernel entry/exit mode transitions
  - We added the ability to tune this at runtime (on/off) for sysadmins to manage perf hit, some don’t want it

Mitigation - Spectre-v2

- Requires (indirect) branch predictor mistrained into speculatively executing existing gadgets
  - Ind. predictors use insufficient address hash/tag bits
  - Future hw may use additional bits (PCID/ASID...)
- Two paths exist for mitigation of Spectre-v2:
  - Prevent speculation based upon the indirect predictor
    - e.g. x86: “IBRS” (Indirect Branch Restrict Speculation), “STIBP”, “IBPB”
    - e.g. IBM: “nop” assists via millicode traps or Arm: SMC/PSCI interface extensions
  - Prevent the use of indirect branches in otherwise vulnerable code (“Retpolines”, “Expolines”...)
- Retpolines required toolchain changes

```
.macro __IBRS_ENTRY
        movl IBRS_HI32_PCP, %edx
        movl IBRS_ENTRY_PCP, %eax
        GET_THREAD_INFO(%rcx)
        bt   $TIF_SSBD, TI_flags(%rcx)
        jnc  .Lno_ssbd_\@
        orl  $FEATURE_ENABLE_SSBD, %eax
.Lno_ssbd_\@:
        movl $MSR_IA32_SPEC_CTRL, %ecx
        wrmsr
.endm
```

Source: Linux Vendor x86 kernel

Mitigation - Spectre-v2

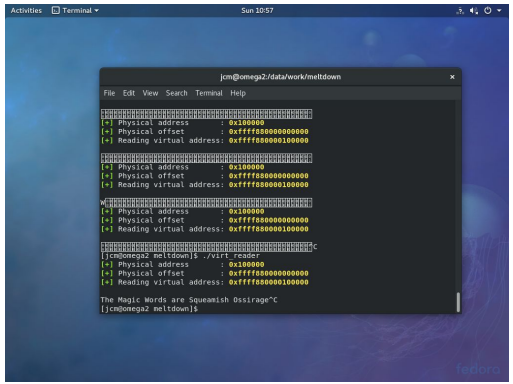
- Focus of mitigation was on protecting cross-privilege boundary attacks
  - Initially protect the kernel on entry with assembly macros
    - Frobbling IBRS on x86 (shipped initially, still not upstream)
  - Protect user process to user process attacks across (certain) context switches
    - IBPB (Indirect Branch Predictor Barrier) e.g. based on “dumpable”/traceable state
  - Later we switched to “Retpolines” for many processors (but not all, upstream default)
  - We did not rebuild all of userspace (including proprietary software) with e.g. retpolines
- Protecting against all possible attack types adds additional perf impact
  - Special kernel boot parameter (spectre_v2=ibrs_always) to guarantee userspace
  - Many users want to turn these mitigations off (e.g. closed lab environment)
  - Defaults even now are a tradeoff between performance and total security
- Future hardware will advertise fixes/automatic mitigations
  - e.g. ARCH_CAPABILITIES on x86, SMC interface on Arm, DT on IBM

Mitigation - Spectre-v1

- Focus of mitigation was on protecting cross-privilege boundary attacks
  - Protect the kernel against attacks e.g. from malicious eBPF or other bounds check bypass
  - Initially used context serializing instructions (e.g. “lfence”, nop encodings w/millicode assist)
  - Later transitioned to speculative “clamping” to in-bounds through a mask operation
  - All these sites were addressed individually through scanning and inspection
- New tools were created or modified
  - e.g. Coverity, Smatch, LLVM SLH (Google), MSVC /Qspectre (Microsoft)
  - Many false positives using early tools during the embargo period
- Focus was not on mitigating against all possible attacks
  - We could not rebuild the entire world (in time) with a “magic” compiler
  - “NetSpectre” extends realistic remote attack to (vastly larger) userspace
  - Spectre-v1 is very difficult to comprehensively mitigate in software

Mitigation - L1TF (“Foreshadow”)

- Just the latest vulnerability to be disclosed (Aug 14th)
  - This vulnerability is specific to Intel x86 processors
  - Similar cause to “Meltdown” (defer valid/access checks)
- Targets page table present (valid) bit speculation,
  - Requires exploitable page table entry and data in L1 data cache
  - A (detectable) page fault taken will be taken at retirement
  - Bare metal/OS attack mitigated with page table inversion
    - Mask OS-generated “not present” PTEs to high PA space
  - Hypervisor attack mitigated with secret scrubbing/scheduling
    - Some cases require SMT (Hyper-threading) disable



Screenshot: The first reproducer I created for L1TF analysis within Red Hat (with missing “f” in “Ossifrage”)

Testing and performance

- Testing these mitigations was uniquely challenging
  - Extensive need to investigate third party drivers and other code for interactions
    - e.g. Windows AV drivers with syscall table hooks required new registry setting
    - e.g. Some Linux vendors use a “stable” kernel ABI (similar to Windows) for drivers
  - Required a cover story in many cases to prevent unauthorized disclosure
    - e.g. kernels provided to third parties with (only) KAISER/PTI discreetly enabled
    - Those patches were already public, discussed under guise of KASLR bypass
      - “We just really care a lot about ASLR” and other interesting fiction
    - Allowed us to test the most impactful piece without breaking any embargo
- Invalidated many existing benchmarks and system tunings
  - Required new performance tuning profiles, guidelines, whitepapers, etc.
  - Will have a lasting impact on industry standard performance numbers

Ongoing Research

- Additional exploits will keep coming (TLBleed, NetSpectre, L1TF, RSB*...)
  - NetSpectre was an interesting extension to remote kernel/userspace attacks
  - TLBleed reminds us that TLBs are not the only uarch side-channel available
  - All of these require additional investment in tooling, expertise.
  - Need to avoid “Spectre Fatigue” (“Omega fatigue”)
- Industry collaboration is important
  - We built up some great contacts during the previous exploits
  - Neutral entities (such as Linux Foundation) good places to develop tools, etc.
  - We need to handle these together across all architectures, not just one
- Funding research is important
  - Academics get paper published when they show high profile real world examples
  - But others exist, many more topics than cycles available today

Where do we go from here?

- Changes to how we design hardware are required
  - Addressing Meltdown, and Spectre-v2 in future hardware is relatively straightforward
  - Addressing Spectre-v1 and v4 (SSB) may be possible through register tagging/tainting
  - A fundamental re-adjustment in focus on security vs. performance is required
- Changes to how we design software are required
  - All self-respecting software engineers should have some notion of how processors behave
  - A professional race car driver or pilot is expected to know a lot about the machine
  - We must communicate. No more “hardware” and “software” people. No more “us” and “them”.
- Open Source can help
  - Open Architectures won’t magically solve our security problems (implementation vs spec)
  - However they can be used to investigate and understand, and collaborate on solutions
    - Many/most security researchers using RISC-V already, makes a lot of sense
  - Opening up processor microcode/designs. Can you fully trust what you can’t see?

Exploiting Modern μarchitectures: Hardware Implications

Mark Hill

Micro-architectural & Architectural Implications of Meltdown & Spectre

Joint Hot Chips Keynote, August 2018

Mark D. Hill, Wisconsin, CCC, Google Sabbatical

1. Background, Meltdown, & Spectre
2. Repair Micro-Architecture
3. Change Architecture & Methods?

Computer Architect, Not Security Expert

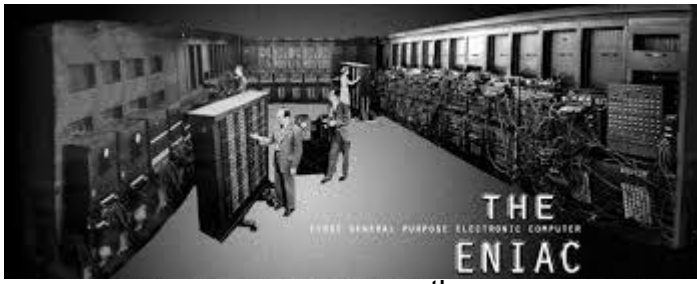
The Set Up

Architecture 0.0: Each computer implementation was new, requiring all software to be rewritten (in machine/assembly language)

Architecture 1.0: the timing-independent functional behavior of a computer
Micro-architecture: the implementation techniques to improve performance

Covert timing channels … one process to signal ... another … by modulating its own use of system resources [to be] observed by the 2nd process.

--1983/85 DoD “Orange Book”



ENIAC, etc. mid-20th Century

IBM 360 ISA, 1964

Spy vs. Spy, Mad Magazine, 1960

Executive Summary

Speculation leaks protected information but is essential for performance

Not bugs: **Micro-Architecture** correct to **Architecture 1.0** spec

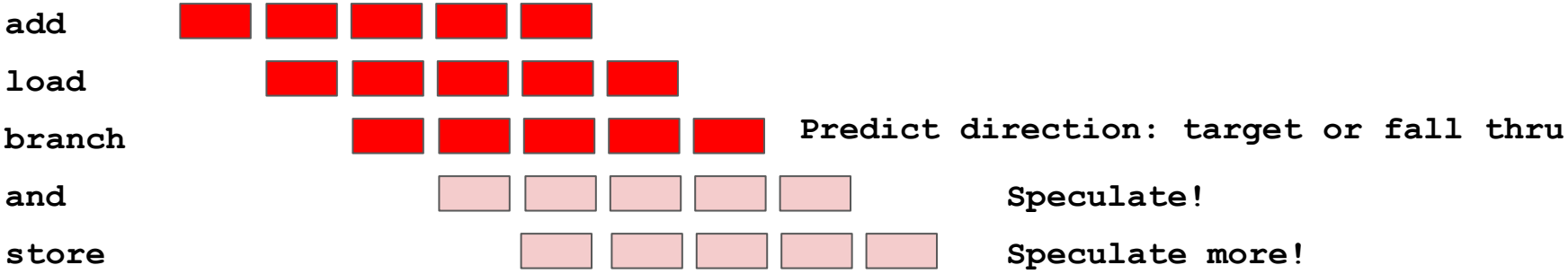
Flaws in the half-century-old timing-independent definition of **Architecture 1.0**

What TO-DO, since it can't be "correct" to leak protected information?

- We will repair **Micro-Architecture**: Manage, not fix, like crime
- We should define **Architecture 2.0** and/or change methods

Instruction Speculation

Go Faster: Pipelining, branch prediction, & instruction speculation



Fundamental Assumption: **Okay to check speculation at the end!**

- Speculation correct: Commit architectural changes of **`and`** (register) & **`store`** (memory) go fast!
- Mis-speculate: Abort architectural changes (registers, memory); go in other branch direction

To Over-Simply: SAVE Secret in Micro-Arch

```
branch (R1 >= bound) goto error
load R2 ← memory[train+R1]      ; Get SECRET
and R3 ← R2 && 0xffff
load R4 ← memory[save+SIZE+R3]  ; Put SECRET in address
                                  to perturb cache
```

To Over Simplify: Just Eliminate Speculation? No

Modern Processors (Intel Skylake example numbers)

- 224-entry reorder buffer w/ 14-19-stage pipeline
- 3 cache levels: speculate hit for 0.25ns cycle vs. ~100ns DRAM
- Interactions among 4-28 cores (speculate coherence good, no bank conflict, …)

Straightforward speculation elimination would cost >> 20X (like 200 MHz?)

Regardless on exact number

⇒ Not viable for a general-purpose processor product

⇒ Must more creatively mitigate timing channels

Meltdown (a.k.a. Google Variant 3)

Leaks kernel memory at up to 500KB/s on Intel x86-64 cores

Intel appears to suppress trap LATE (after micro-arch changes)
➔ Ok by Architecture 1.0 w/ High performance but Meltdown!

Others appear to trap EARLY (e.g., at address translation)
➔ Ok by Architecture 1.0 ➔ No Meltdown

Solutions
SW: Don't map kernel (KAISER) ➔ performance loss on syscalls
HW: Trap early (as done by many vendors)

In retrospect, uncommon case where easy to stop speculation early

Spectre (Google Variants ≠ 3)

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

- Most—if not all—cores & vendors
- Load does NOT trap (Meltdown traps)
- Still inappropriate if managed language or sandbox

Variants

1. Use branch mis-prediction to let Javascript steal from Chrome browser
2. Uses indirect branches (returns) & return-oriented programming
3. Meltdown
4. Re write buffer bypass

Performance ➔ Speculation ➔ Spectre

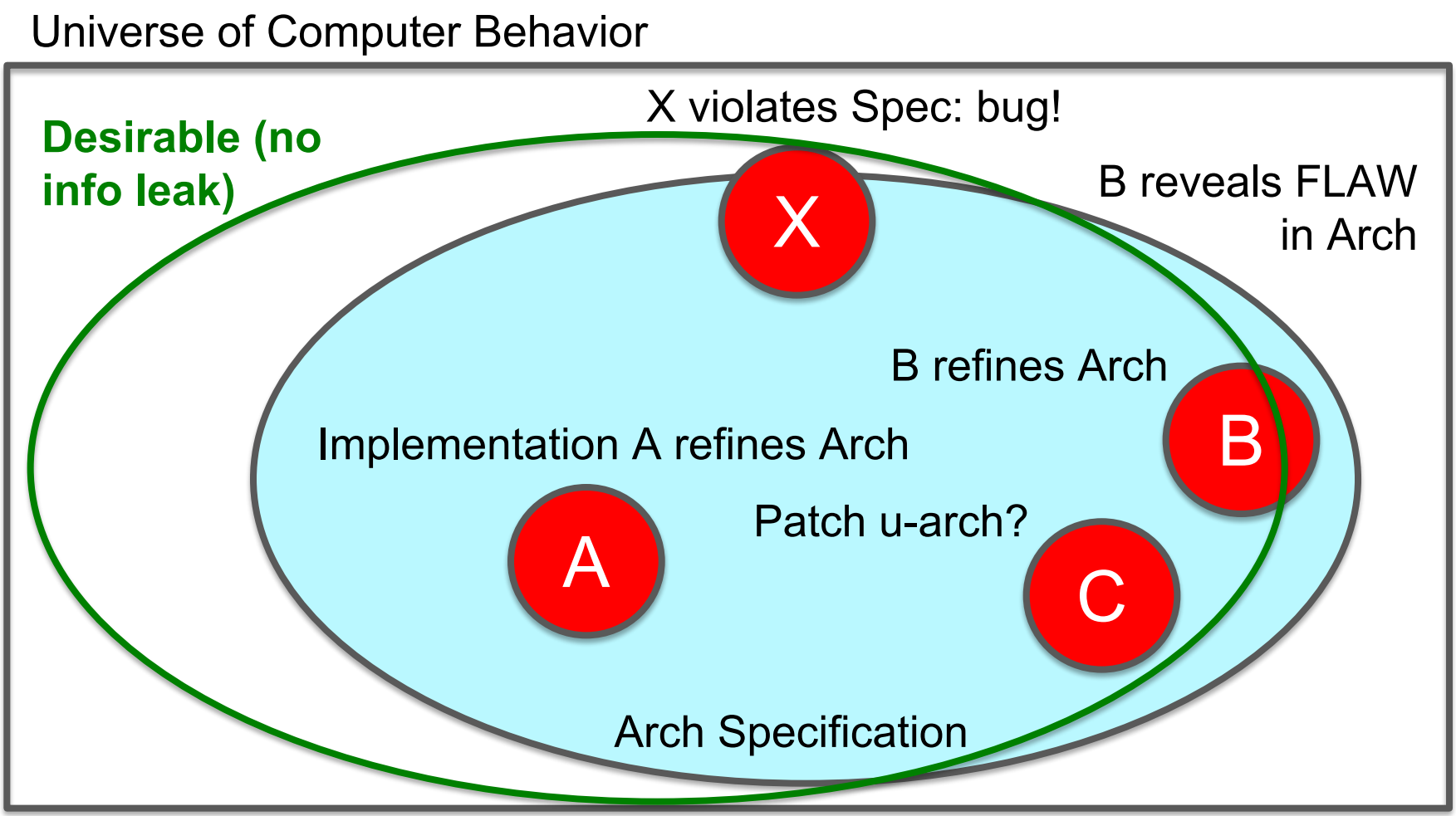
What to do?

... Coherence, functional units, TBD ☹

Page tables (L1TF 08/14/2018 ~ Meltdown)

Outline

1. Background, Meltdown, & Spectre
2. **Repair Micro-Architecture**
3. Change Architecture & Methods?


Universe of Computer Behavior
Desirable (no info leak)
X violates Spec: bug!
X
B reveals FLAW in Arch
B refines Arch
B
Implementation A refines Arch
Patch u-arch?
A
C
Arch Specification

Repair Microarchitecture

W/o speculation/caches >>20x performance loss

While (1)

1. Find timing channel with concerning bandwidth
2. Fix it with performance and/or complexity cost

Not easy

- Does shared cache way-partitioning cut timing channels (e.g., Intel CAT)?
- No, need changes to replacement algorithm & “shared” hits (see DAWG)

Treat timing channels like crime: Manage without solving

Goal: MIN(“security/police/etc. cost” + “crime cost”)

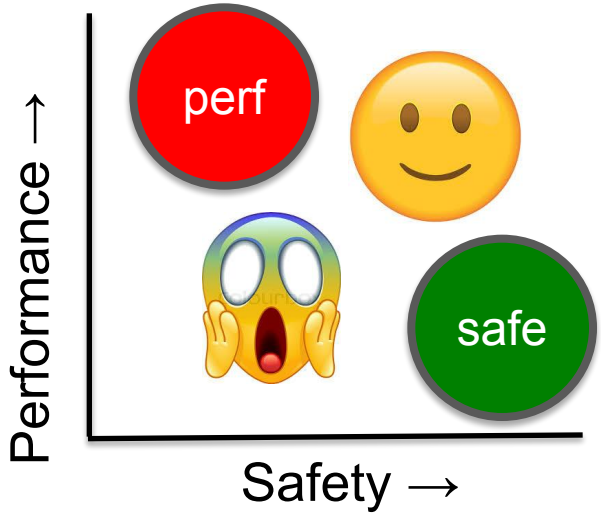
Micro-Architectural Ideas

- **Isolate** branch predictor, BTB, TLBs, etc. & flush/restore on context switch
- **Partition** caches among trusted processes (& flushed on context switch?)
- **Reduce aliasing** information, e.g., fully-associative caches or fancy indexing
- **Randomize** to lower BW; degrade/hide “timers”
- **HW Protection** w/i user address space
  e.g., trap if javascript accesses protected browser
- **Undo** speculation (as much as possible)

- **Constant-time execution?**
  (at some granularity: instrn, function, program)
  "He treats us all the same -- like dogs.”
  --Henry Jordan on Vince Lombardi


Vince Lombardi. © Vernon J. Biever

Whither High Performance & Timing-Channel “Free”?



Happy knee with good performance & good safety?

I fear not & arrives now as Moore’s Law bounty slows

Bifurcate! How?

In Time: Modes for fast(er) & safe(r)

- Disable some speculation & partition more
- Dynamically flexible but limited

In Space: Fast Cores, Safe Cores, etc.

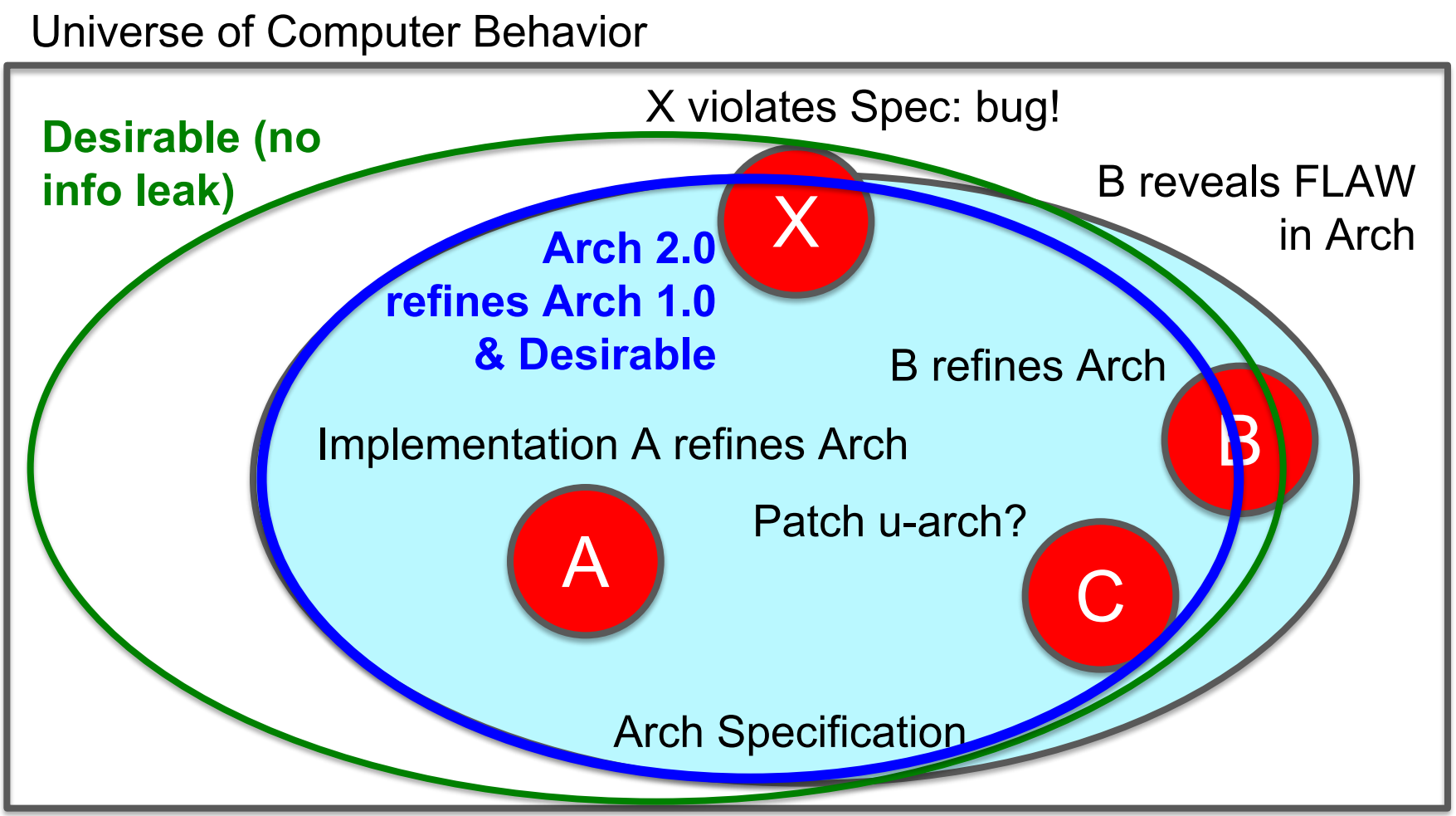
- Extension of what is being done for security
- Allows extremes; plays well w/ dark silicon

In Use:

- Cloud provider charges more for exclusive VMs
- Don’t execute downloaded code

Outline

1. Background, Meltdown, & Spectre
2. Repair Micro-Architecture
3. **Change Architecture & Methods?**


Universe of Computer Behavior
X violates Spec: bug!
Desirable (no info leak)
B reveals FLAW in Arch
X
Arch 2.0 refines Arch 1.0 & Desirable
B refines Arch
B
Implementation A refines Arch
Patch u-arch?
A
C
Arch Specification

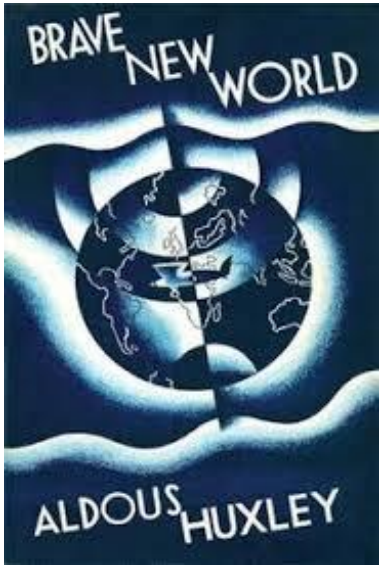
Need Computer Architecture 2.0?

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)
- Protect user-space regions & suppress speculation

None seem good enough to me (yet)



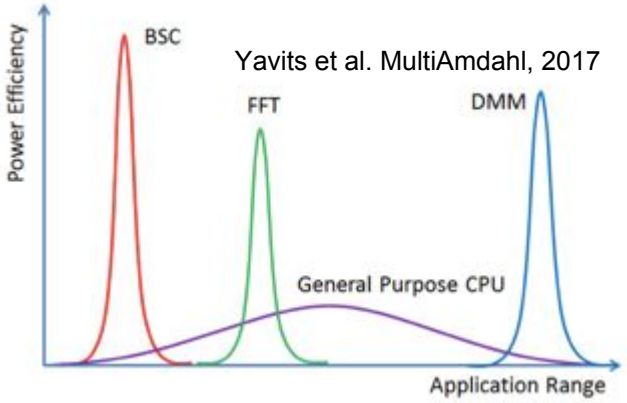
Computer Architecture 2.0: More Accelerators?

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!



But accelerators have timing channels
E.g., branch & memory divergence or bus & memory controller conflicts

Formal Methods but Hard

Tools:

- GLIFT [Tiwari ASPLOS'09] (follow-ons)
- SecVerilog [Zhang, ASPLOS'15]

Can't easily dynamically check for information exfiltration

- See **hyperproperties** of set of executions [Clarkson & Schneider, '10]

- Presumes a spec to check against (Architecture 2.0)
- Spatial bifurcation helps as methods may be easier to apply to safe cores

Open Architecture & Micro-Architectures?

Security Experts

- Disdain “security by obscurity”
- In favor of many “eyeballs”

Open-source SW can help security

- More eyeballs but bad implementation is still bad

Whither open-source HW?

- Interfaces: Instruction Set Architecture
- Implementations: libraries for low- to medium-end

“Most future HW security ideas with be tried with RISC V first.” – D. Patterson

We Should Talk

"Computer Architect, Not Security Expert"

➔ I am part of the problem

20th Century

- Layers worked: Roman *dīvide et imperā*
- Low BW among SW/HW/Security/Formal

21st Century needs

- Cross-layer, end-to-end solutions
- High BW inclusive discussions in public and confidential

Executive Summary

Speculation leaks protected information but is essential for performance

Not bugs: **Micro-Architecture** correct to **Architecture 1.0** spec

Flaws in the half-century-old timing-independent definition of **Architecture 1.0**

What TO-DO, since it can't be "correct" to leak protected information?

- We will repair **Micro-Architecture**: Manage, not fix, like crime
- We should define **Architecture 2.0** and/or change methods

Panel Discussion & Q&A