

# Micro-architectural & Architectural Implications of Meltdown & Spectre

@ Wisconsin, 04/2019

Mark D. Hill, U. Wisconsin-Madison, USA

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

Computer Architect, Not Security Expert



# 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**

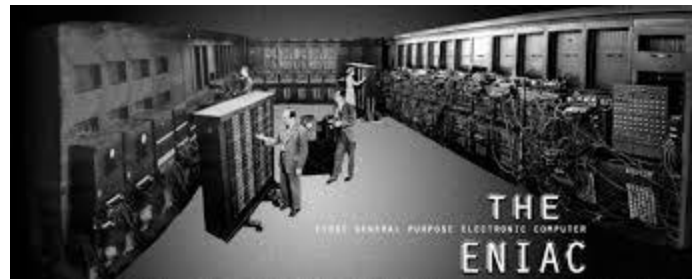
What TO-DO, since it can't be “correct” to leak protected information?

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

# 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**

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

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

**Architecture 1.0:** 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.



# 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 (including caches).

Key techniques w/ tutorial next:

- Instruction Speculation
- Hardware Caching

Hidden by Architecture 1.0: timing-independent functional behavior unchanged

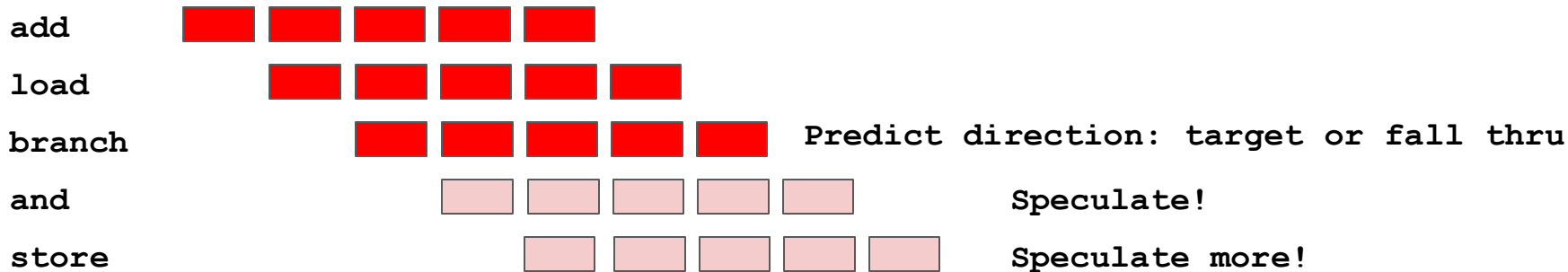
# Instruction Speculation Tutorial

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



E.g., Intel Skylake has  
224-entry reorder buffer  
w/ 14-19-stage pipeline

Go Faster: Pipelining, branch prediction, & instruction speculation



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

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

# Hardware Caching Tutorial

E.g., Intel Skylake has 3 cache levels  
w/ 0.25ns L1 hit vs. ~100ns DRAM  
Interactions among 4-28 cores  
(coherence good, no bank conflict, ...)

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

0	--	12?	0	12	07?	0	12	12?	0	12	16?	0	16	Note 12 victimized "early" due to "alias"
1	--	Miss	1	--	Miss	1	--	HIT!	1	--	Miss	1	--	
2	--	Insert 12	2	--	Insert 07	2	--	No	2	--	Victim 12	2	--	
3	--		3	--		3	07	changes	3	07	Insert 16	3	07	

# 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**)

## To Over-Simplify: **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
```

# 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

But seemed fanciful



Spy vs. Spy, Mad Magazine, 1960

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

Instead of branch/load: user load of kernel memory that traps  
Leaks kernel memory at up to 500KB/s on Intel x86-64 cores



Intel appears to start cache miss before trapping (allow **micro-arch** changes)

→ Ok by **Architecture 1.0** w/ High performance but Meltdown!

Others appear to trap without cache miss (e.g., at address translation)

→ Ok by **Architecture 1.0** → No Meltdown

## Solutions

SW: Don't map kernel → performance loss on syscalls

HW: Trap early (as done by many vendors & claimed for new Intel Cascade Lake)

**“Easy” to stop speculation early: SW protection boundary known to HW**

# 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)
- Violation of managed language or sandbox
- Hard as SW protection boundary unknown to HW



## 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

... Coherence, functional units, TBD ☹

What to do?

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

# Ref 1/2: Spectre Variants, 10/2018, for IEEE Micro

Variant Name and Gist	Gist of Mitigation Strategies
<b>V1 (Bounds Check Bypass).</b> Mistrained conditional branch predictor used to violate program semantics by speculatively accessing data beyond an array limit.	Either enforce instruction stream serialization with respect to later loads (e.g. through an “lfence” on x86) or use speculative load clamping to constrain bounds of array. LLVM calls this “speculative load hardening”.
<b>V1.1 (Bounds Check Bypass Store).</b> Similar to variant 1 but applies to stores, allowing e.g. speculative buffer overflow/stack overflow with restoring of returns.	Careful auditing for potentially risky stores, aided by automated tools (smatch, etc.) or compiler lift (e.g. LLVM speculative load hardening, MSVC). Enforce instruction stream serialization or use clamping.
<b>V1.2 (Read-only Protection Bypass).</b> Hardware may implement lazy enforcement of page table protections allowing speculative writes to read-only data.	Extension of Bounds Check Bypass Store. Relying on read only memory protections against e.g function pointer overwrite is not sufficient. It is necessary to protect against potential overwrites into RO memory.
<b>V2 (Branch Target Injection).</b> Mistrained indirect branch predictor Branch Target Buffer (BTB) speculatively executes attacker-controlled “gadgets”.	Limit speculation based upon the Branch Target Buffer when crossing privilege boundaries, flush predictor state when transitioning from one task to another, limit speculation based upon BTB between SMT threads.
<b>V3 (Rogue Data Cache Load, aka “Meltdown”).</b> User load that speculatively accesses kernel space. See [Melt]	Exploitation requires both a valid address translation as well as (typically) data present in the L1 data cache. Either separate address space between privileged and unprivileged execution states, and/or ensure data is not present in the cache and cannot be loaded by attacker. On some architectures, implement Page Table Isolation (PTI) between user/kernel, on others use an L1D flush.
<b>V3a (Rogue System Register Read).</b> Speculative reads to normally inaccessible system registers may be used to infer information, such as page table base address used to point to all active page tables	In some cases, updated microcode (etc.) can be used to make such reads serializing and not execute speculatively. In other cases, it may not be possible to prevent certain information leakage - such as the location in memory of page table base address.
<b>V4 (Speculative Store Bypass).</b> Speculative reads may proceed prior to determining whether a conflicting store exists in the store buffer (memory disambiguation)	Disabling speculative store buffer bypassing (aka “memory disambiguation”) either globally, or on a per-application basis, is one mitigation path. Another is aggressive use of process-level isolation (separating contexts of execution), but this is difficult for some cases. Linux eBPF and Java runtimes are examples where a per-process control to disable speculative bypassing of the store buffer is typically employed.

# Ref 2/2: Spectre Variants, 10/2018, for IEEE Micro

Variant Name and Gist	Gist of Mitigation Strategies
BranchScope. Conditional directional branch predictor attack used to infer direction of branches taken in vulnerable code (for example, cryptographic libraries)	Two broad strategies exist - careful removal of branch dependencies upon secret data, and "if-conversion" in which branches are converted to sequential code using conditional instructions. The latter can be implemented at the compiler level, while the former is purely manual.
LazyFPU save/restore. Processor implementation may be optimized to avoid saving Floating Point Unit (and vector) context when switching tasks until the new task performs an FPU operation. Vulnerable hardware still allows speculative reads of the disabled FPU state.	Disabling lazy save/restore of Floating Point Unit state. In many cases, this actually improves performance on contemporary processors, particularly those which have hardware assisted save/restore FPU instructions.
TLBleed. A temporal attack against Simultaneous Multithreading (SMT) implementations with tightly shared TLBs (Translation Lookaside Buffers) allowing precise measurement of TLB state on one thread to infer execution of process running on the peer thread.	The simplest solution is to carefully schedule sensitive (e.g.) cryptographic operations such that they do not occur on a sibling SMT thread (e.g. "Hyperthread") at the same time as an untrusted workload is running on its peer. Other mitigations include refactoring libraries to use only constant time operations, which may not be feasible or possible (hardware may not support this).
SpectreRSB/ret2spec. Return Stack Buffer manipulated in order to divert speculative execution of a function return into an attacker-determine leak gadget.	RSB "stuffing" is employed to ensure the RSB is filled with a benign delay gadget. This RSB stuffing approach is also used as part of the mitigation for Spectre-v2 on certain process (e.g. Intel Skylake+) wherein an underfill in the RSB causes speculation from the BTB. Thus, it is preferable to reuse the existing mitigation.
NetSpectre. Similar to Spectre but performed over a network using a combination of a leak gadget (used to alter microarchitectural state) and transmission gadget (used to transmit this altered state across a network).	Mitigation is similar to Spectre-v1 however the impacted code is potentially very significant. As a result, other solutions at the network layer may be employed, or the impact of leakage may be reduced through careful application of rekeying during transactions. Very sensitive deployments may choose to recompile significant portions of applications using speculative load hardening techniques e.g. as found in LLVM.
L1TF (L1 Terminal Fault, "Foreshadow" - SGX). Speculative loads to virtual addresses translated by Page Table Entries (PTEs) with "present" bit not set may result in the processor forwarding the incorrect physical address to the L1 data cache (L1D), allowing reads of attacker-controlled addresses if in the cache.	L1TF requires that data be present in the L1 Data cache of impacted Intel processors, and that it be possible to construct a vulnerable page table entry. For the "bare metal" use case of an OS on hardware, it is possible to protect against malicious applications by ensuring that all "not present" OS PTEs are masked such that the address is outside of populated physical memory. For virtual machines, it is necessary to employ an L1D cache flush via microcode assist on VM entry.
Spectre variants after October 2018?	To be determined.

# Outline

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



# 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, ...)

Naïve elimination of speculation & caches would slow by >> 10X

Regardless on exact number

- ➔ Not viable for a general-purpose processor product
- ➔ Must more creatively mitigate timing channels

# Repair Microarchitecture

**W/o speculation/caches >> large 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:  $\text{MIN}(\text{“security/police/etc. cost”} + \text{“crime cost”})$



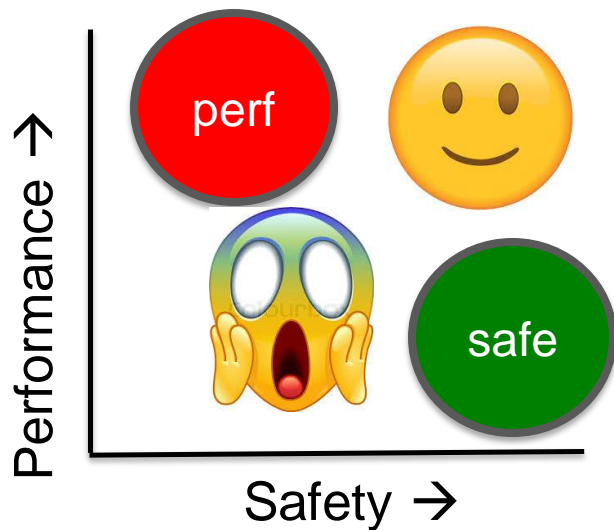
# Micro-Architectural Ideas

- **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:**

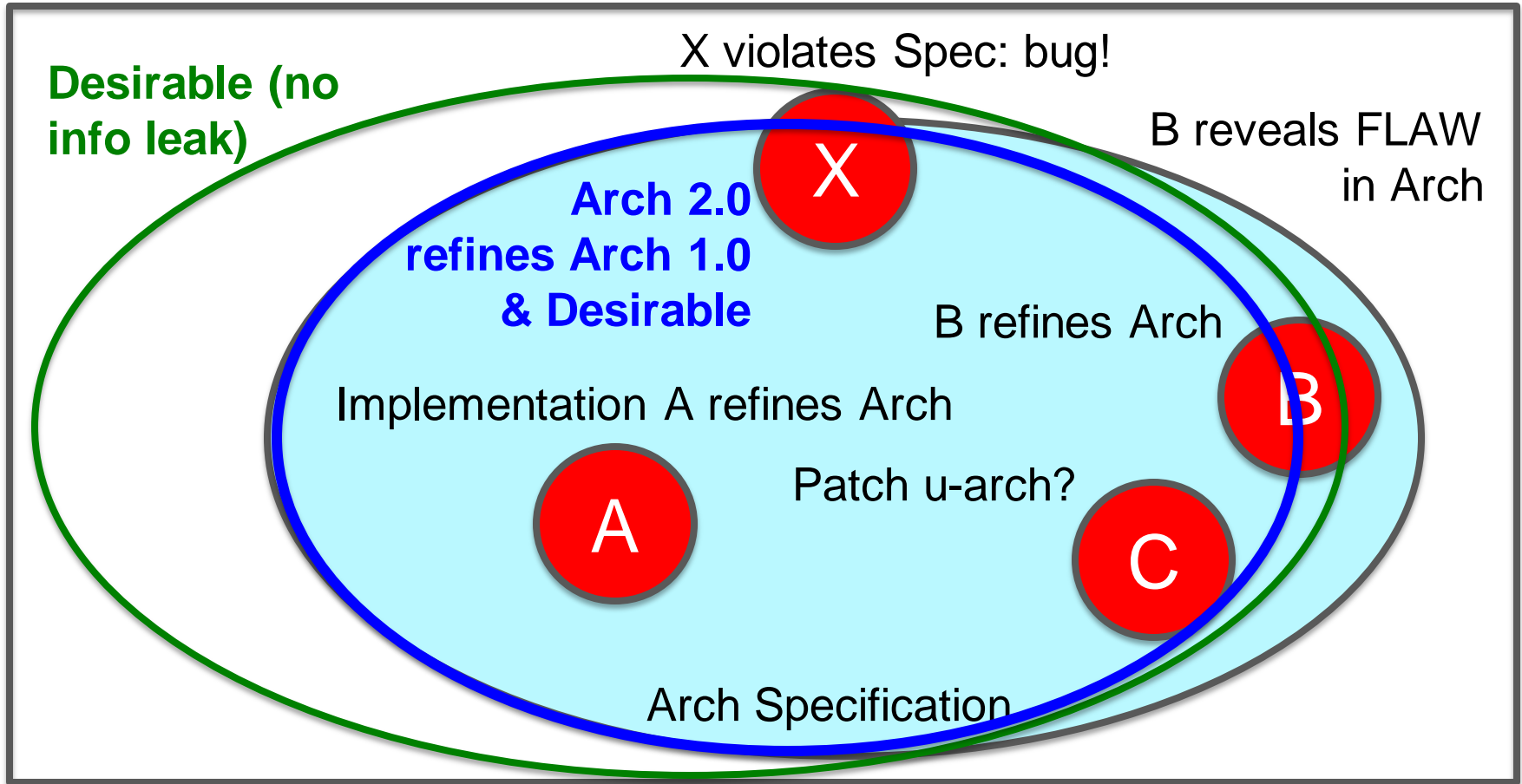
- Cloud provider charges more for exclusive VMs
- Don't execute downloaded code



# Outline

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

# Universe of Computer Behavior



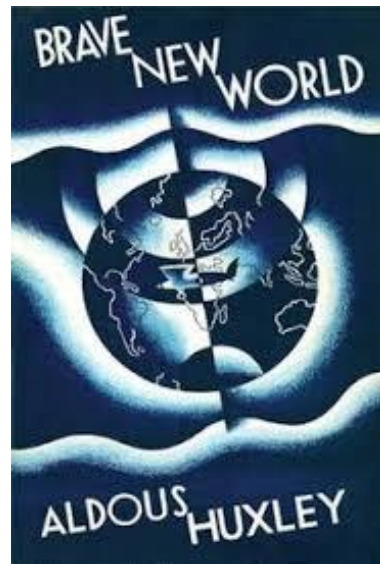
# Need Computer Architecture 2.0?

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)





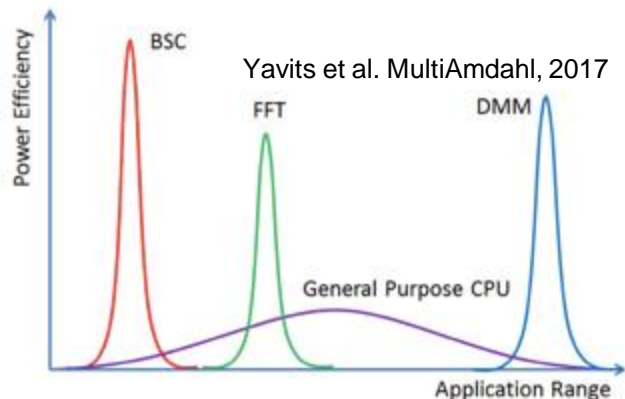
# Computer Architecture 2.0: More Accelerators?

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

$$e^{i\pi} + 1 = 0$$

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 will be tried with RISC V first.” – D. Patterson*

# We Should Talk

“Computer Architect, Not Security Expert”

➔ I am part of the problem

20<sup>th</sup> Century

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

21<sup>st</sup> 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