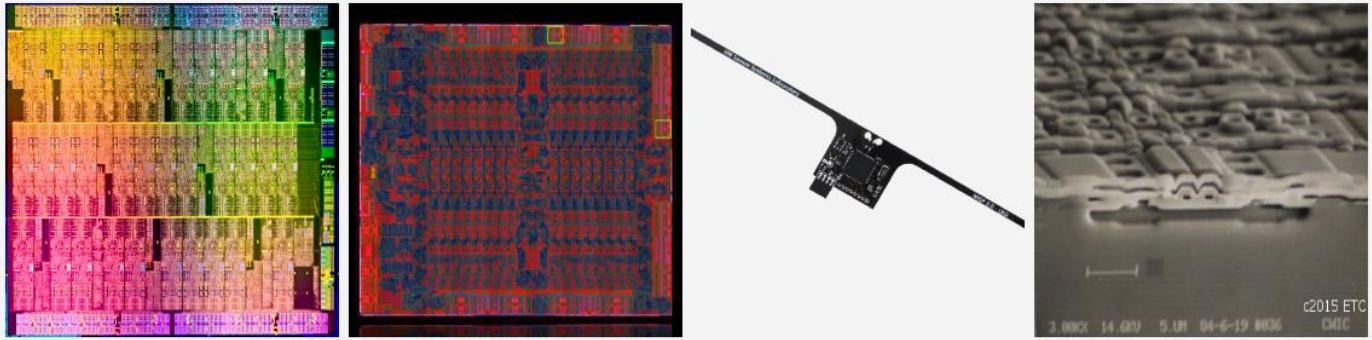
18-344: Computer Systems and the Hardware-Software Interface

Fall 2025



Course Description

Lecture 9: Meltdown and Spectre

This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course will have learned operating system fundamentals, computer architecture fundamentals, compilation to hardware abstractions, and how software actually executes from the perspective of the hardware software/boundary. The course will focus especially on understanding the relationships between software and hardware, and how those relationships influence the design of a computer system's software and hardware. The course will convey these topics through a series of practical, implementation-oriented lab assignments.

Credit: John Masers, RedHat, *Meltdown and Spectre,* USENIX Lisa 2018

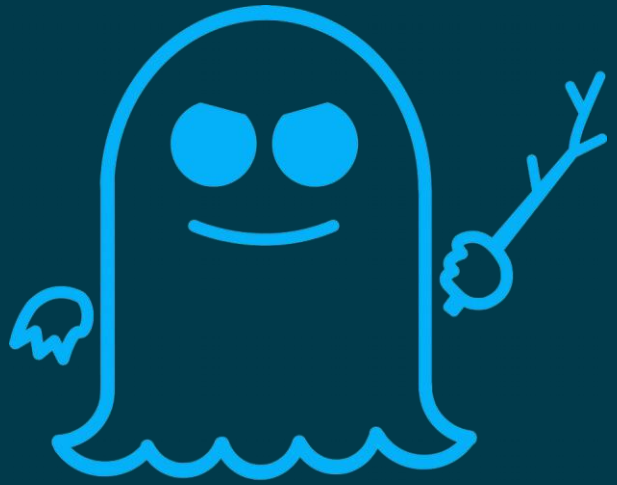
redhat.

Meltdown and Spectre

Jon Masters, Computer Architect, Red Hat, Inc.
jcm@redhat.com | @jonmasters

Usenix LISA 2018

MELTDOWN



SPECTRE

Side-channel attacks

Side-channel attacks

- “In computer security, a side-channel attack is any attack based on information gained from the physical implementation of a computer system, rather than weaknesses in the implemented algorithm itself (e.g. cryptanalysis and software bugs).” – from the Wikipedia definition

- Examples of side channels include
  - Monitoring a machine's electromagnetic emissions (“TEMPEST”-like remote attacks)
  - Measuring a machine's power consumption (differential power analysis)
  - Timing the length of operations to derive machine state
  - ...

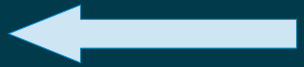
Caches as side channels

- **Caches exist because they provide faster access to frequently used data**
  - The closer data is to the compute cores, the less time is required to load it when needed
- **This difference in access time for an address can be measured by software**
  - Data closer to the cores will take fewer cycles to access
  - Data further away from the cores will take more cycles to access
- **Consequently it is possible to determine whether an address is cached**
  - Calibrate by measuring access time for known cached/not cached data
  - Time access to a memory location and compare with calibration

Caches as side channels

- Consequently it is possible to determine whether a specific address is in the cache
  - Calibrate by measuring access time for known cached/not cached data
  - Time access to a memory location and compare with calibration

```
time = rdtsc();
maccess(&data[0x300]);
delta3 = rdtsc() - time;

time = rdtsc();
maccess(&data[0x200]);
delta2 = rdtsc() - time;
```



Execution time taken for instruction is proportional to whether it is in cache(s)

Caches as side channels (continued)

- **Many arches provide convenient high resolution cycle-accurate timers**
  - e.g. x86 provides RDTSC (Read Time Stamp Counter) and RDTSCP instructions
- **But there are other ways to measure on arches without unprivileged TSC**
- Some arches (e.g. x86) also provide convenient unprivileged cache flush instructions
  - CLFLUSH guarantees that a given (virtual) address is not present in any level of cache
- **But possible to also flush using a "displacement" approach on other arches**
  - Create data structure the size of cache and access entry mapping to desired cache line
- On x86 the time for a flush is proportionate to whether the data was in the cache
  - flush+flush attack determines whether an entry was cached without doing a load
  - Harder to detect using CPU performance counter hardware (measuring cache misses)

Caches as side channels (continued)

- Some processors provide a means to prefetch data that will be needed soon
  - Usually encoded as “hint” or “nop space” instructions that may have no effect
  - x86 processors provide several variants of PREFETCH with a temporal hint
  - This may result in a prefetched address being allocated into a cache
- Processors will perform page table walks and populate TLBs on prefetch
  - This may happen even if the address is not actually fetched into the cache

```
asm volatile ("prefetcht0 (%0)" : : "r" (p));
asm volatile ("prefetcht1 (%0)" : : "r" (p));
asm volatile ("prefetcht2 (%0)" : : "r" (p));
asm volatile ("prefetchnta (%0)" : : "r" (p));
```


Meltdown, Spectre, TLBleed,
NetSpectre, Foreshadow, etc. etc.
etc.
redhat.

2018: year of uarch side-channel vulnerabilities

The list of publicly disclosed vulnerabilities so far this year includes:

- Spectre-v1 (Bounds Check Bypass)
- Spectre-v2 (Branch Target Injection)
- Meltdown (Rogue Data Cache Load)
- “Variant 3a” (Rogue System Register Read)
- “Variant 4” (Speculative Store Bypass)
- BranchScope (directional predictor attack)
- Lazy FPU save/restore
- Spectre-v1.1 (Bounds Check Bypass Store)
- Spectre-v1.2 (Read-only Protection Bypass)
- “TLBleed” (TLB side-channel introduced)
- SpectreRSB / ret2spec (return predictor attack)
- “NetSpectre” (Spectre over the network)
- “Foreshadow” (L1 Terminal Fault)

Example vendor response strategy

- **We were on a specific timeline for public disclosure (a good thing!)**
  - 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

Example vendor response strategy (cont.)

- Produce materials for use during disclosure
  - Blogs, whitepapers, performance webinars, etc.
  - The "X in 3 minutes" videos intended to be informative
- Run performance analysis and document best tuning practices
  - Goal is to be "safe by default" but to give customers flexibility to choose
  - Your risk assessment may differ from another environment
    - Threat model may be different for public/private facing

- **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 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 order of 10,000 individual chicken bits

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

- **Everything else needs to be done in software (kernel, firmware, app...)**

- In reality we leverage a combination of hardware interfaces and software fixes
- Remember: we can't change hardware but we can tweak its behavior+software

Deploying and validating mitigations (e.g. Linux)

- Operating System vendors provide tools to determine vulnerability and mitigation
  - The specific mitigations vary from one architecture and Operating System to another
- Windows includes new PowerShell scripts, various Linux tools have been created
- Very recent (upstream) Linux kernels include the following new “sysfs” entries:

```
$ grep . /sys/devices/system/cpu/vulnerabilities/*
/sys/devices/system/cpu/vulnerabilities/meltdown:Mitigation: PTI
/sys/devices/system/cpu/vulnerabilities/spectre_v1:Vulnerable
/sys/devices/system/cpu/vulnerabilities/spectre_v2:Vulnerable: Minimal
generic ASM retpoline
```

Meltdown

- Implementations of Out-of-Order execution that strictly follow Tomasulo algorthim handle exceptions arising from speculated instructions at instruction retirement
- **Speculated instructions do not trigger (synchronous) exceptions**
  - Loads that are not permitted will not be reported until they are no longer speculative
  - At that time, the application will likely receive a “segmentation fault” or other error
- **Some implementations may perform load permission checks in parallel**
  - This improves performance since we don’t wait to perform the load
  - Rationale is that the load is only speculative (“not observable”)
- A race condition may thus exist allowing access to privileged data

Meltdown (single bit example)

- A malicious attacker arranges for exploit code similar to the following to speculatively execute:

```
if (spec_cond) {
        unsigned char value = *(unsigned char *)ptr;
        unsigned long index2 = (((value>>bit)&1)*0x100)+0x200;
        maccess(&data[index2]);
}
```

- “data” is a user controlled array to which the attacker has access, “ptr” contains privileged data

Meltdown (continued)

- A malicious attacker arranges for exploit code similar to the following to speculatively execute:

```
if (spec_cond) {
        unsigned char value = *(unsigned char *)ptr;
        unsigned long index2 = (((value>>bit)&1)*0x100)+0x200;
        maccess(&data[index2]);
}
```

load a pointer to which we don't have access

Meltdown (continued)

- A malicious attacker arranges for exploit code similar to the following to speculatively execute:

```
if (spec_cond) {
        unsigned char value = *(unsigned char *)ptr;
        unsigned long index2 = (((value>>bit)&1)*0x100)+0x200;
        maccess(&data[index2]);
}
```

bit shift extracts
a single bit of data

Meltdown (continued)

- A malicious attacker arranges for exploit code similar to the following to speculatively execute:

```
if (spec_cond) {
        unsigned char value = *(unsigned char *)ptr;
        unsigned long index2 = (((value>>bit)&1)*0x100)+0x200;
        maccess(&data[index2]);
}
```

generate address
from data value

Meltdown (continued)

- A malicious attacker arranges for exploit code similar to the following to speculatively execute:

```
if (spec_cond) {
        unsigned char value = *(unsigned char *)ptr;
        unsigned long index2 = (((value>>bit)&1)*0x100)+0x200;
        maccess(&data[index2]);
}
```

use address as offset to pull in cache line that we control

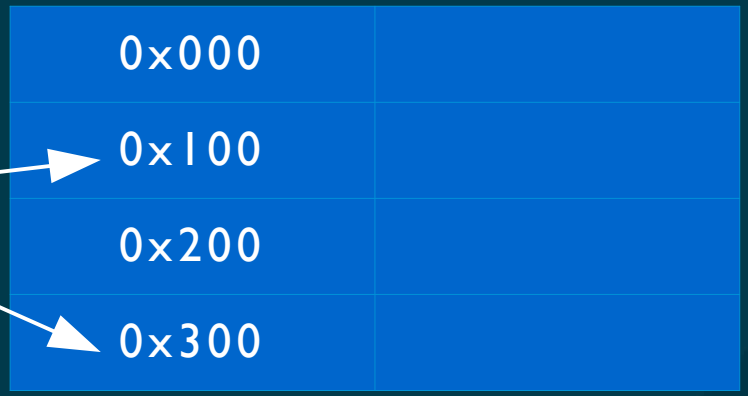
Meltdown (continued)

char value = *SECRET_KERNEL_PTR;

mask out bit I want to read

calculate offset in “data”
(that I do have access to)

char data[];

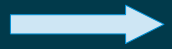


Meltdown (continued)

- Access to “data” element 0x100 pulls the corresponding entry into the cache

char data[];

0x000	
0x100	
0x200	
0x300	DATA



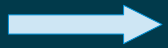
Cache

Meltdown (continued)

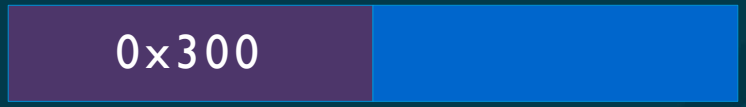
- Access to “data” element 0x300 pulls the corresponding entry into the cache

char data[];

0x000	
0x100	
0x200	
0x300	DATA



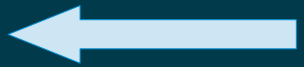
Cache



Meltdown (continued)

- We use the cache as a side channel to determine which element of “data” is in the cache
  - Access both elements and time the difference in access (we previously flushed them)

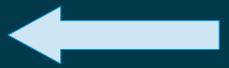
```
time = rdtsc();
maccess(&data[0x300]);
delta3 = rdtsc() - time;

time = rdtsc();
maccess(&data[0x200]);
delta2 = rdtsc() - time;
```



Execution time taken for instruction is proportional to whether it is in cache(s)

Meltdown: Speculative Execution

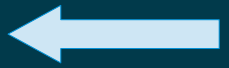
Entry	RegRename	Instruction	Deps	Ready?	Spec?
1	P1 = R1	R1 = LOAD SPEC_CONDITION	X	Y	N
2		TEST SPEC_CONDITION	1	Y	N
3		IF (SPEC_CONDITION) {	1	N	N
4	P2 = R1	R2 = LOAD KERNEL_ADDRESS	X	Y	Y*
5	P3 = R2	R3 = (((R2&1)*0x100)+0x200)	2	Y	Y*
6	P4 = R4	R4 = LOAD USER_BUFFER[R3]	3	Y	Y*



flags for future exception

Meltdown: Speculative Execution

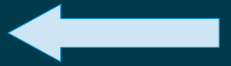
Entry	RegRename	Instruction	Deps	Ready?	Spec?
1	P1 = R1	R1 = LOAD SPEC_CONDITION	X	Y	N
2		TEST SPEC_CONDITION	1	Y	N
3		IF (SPEC_CONDITION) {	1	N	N
4	P2 = R1	R2 = LOAD KERNEL_ADDRESS	X	Y	Y*
5	P3 = R2	R3 = (((R2&1)*0x100)+0x200)	2	Y	Y*
6	P4 = R4	R4 = LOAD USER_BUFFER[R3]	3	Y	Y*



should kill speculation here

Meltdown: Speculative Execution

Entry	RegRename	Instruction	Deps	Ready?	Spec?
1	P1 = R1	R1 = LOAD SPEC_CONDITION	X	Y	N
2		TEST SPEC_CONDITION	1	Y	N
3		IF (SPEC_CONDITION) {	1	N	N
4	P2 = R1	R2 = LOAD KERNEL_ADDRESS	X	Y	Y*
5	P3 = R2	R3 = (((R2&1)*0x100)+0x200)	2	Y	Y*
6	P4 = R4	R4 = LOAD USER_BUFFER[R3]	3	Y	Y*



really bad thing (™)

Meltdown (continued)

- When the right conditions exist, this branch of code will run speculatively
  - **Privilege check for "value" will fail, but only result in an entry tag in the ROB**
  - The access will occur although "value" will be discarded when speculation is undone
- The offset in the "data" user array is dependent upon the value of privileged data
  - We can use this as a counter between several possible entries of the user data array
- Cache side channel timing analysis used to determine "data" location accessed
  - Time access to "data" locations 0x200 and 0x300 to infer value of desired bit
  - Access is done in reverse in my code to account for cache line prefetcher

Mitigating Meltdown

- The “Meltdown” vulnerability requires several conditions:
  - **Privileged data must reside in memory for which active translations exist**
  - **On some processor designs the data must also be in the L1 data cache**
- Primary Mitigation: separate application and Operating System page tables
  - Each application continues to have its own page tables as before
  - The kernel has separate page tables not shared with applications
  - Limited shared pages exist only for entry/exit trampolines and exceptions

Mitigating Meltdown

- Linux calls this page table separation “PTI”: Page Table Isolation
  - Requires an expensive write to core control registers on every entry/exit from OS kernel
  - e.g. TTBR write on impacted ARMv8, CR3 on impacted x86 processors
- Only enabled by default on known-vulnerable microprocessors
  - An enumeration is defined to discover future non-impacted silicon
- Address Space IDentifiers (ASIDs) can significantly improve performance
  - ASIDs on ARMv8, PCIDs (Process Context IDs) on x86 processors
  - TLB entries are tagged with address space so a full invalidation isn't required
  - Significant performance delta between older (pre-2010 x86) cores and newer ones

Spectre: A primer on exploiting “gadgets” (gadget code)

- A “gadget” is a piece of existing code in an (unmodified) existing program binary
  - For example code contained within the Linux kernel, or in another “victim” application
- A malicious actor influences program control flow to cause gadget code to run
- Gadget code performs some action of interest to the attacker
  - For example loading sensitive secrets from privileged memory
- Commonly used in “Return Oriented Programming” (ROP) attacks

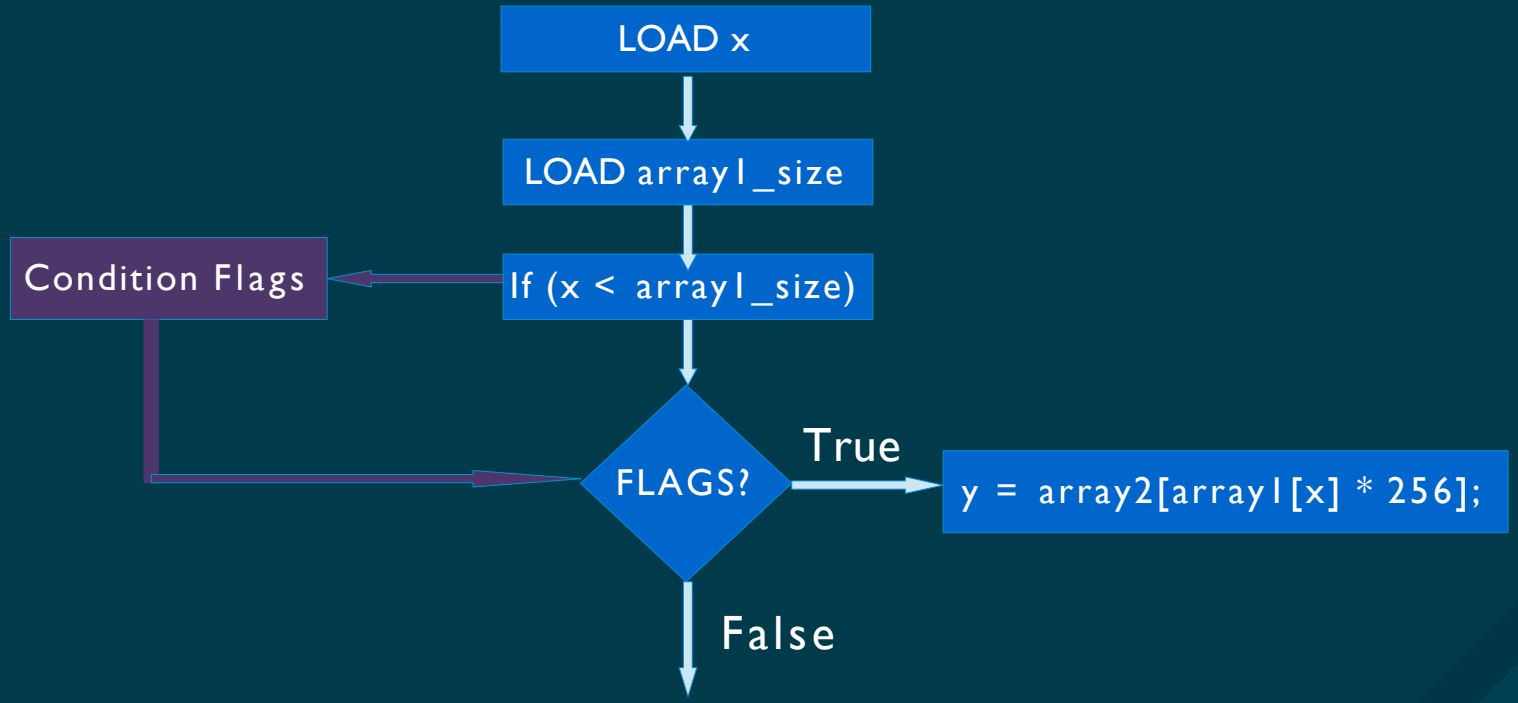
Spectre-v1: Bounds Check Bypass (CVE-2017-2573)

- Modern microprocessors may speculate beyond a bounds check condition
- What's wrong with the following code?

```
If (untrusted_offset < limit) {
    trusted_value = trusted_data[untrusted_offset];
    tmp = other_data[(trusted_value)&mask];
    ...
}
```

A bit “mask” extracts part of a word (memory location)

Branch prediction (Speculation)



Spectre-v1: Bounds Check Bypass (cont.)

- The code following the bounds check is known as a "gadget" (see ROP attacks)
  - Existing code contained within a different victim context (e.g. OS/Hypervisor)
- Code following the untrusted_offset bounds check may be executed speculatively
  - Resulting in the speculative loading of trusted data into a local variable
  - This trusted data is used to calculate an offset into another structure
- Relative offset of other_data accessed can be used to infer trusted_value
  - LID$ cache load will occur for other_data at an offset correlated with trusted_value
  - Measure which cache location was loaded speculatively to infer the secret value

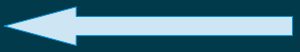
Mitigating Spectre-v1: Bounds Check Bypass

- Existing hardware lacks the capability to limit speculation in this instance
- Mitigation: modify software programs in order to prevent the speculative load
  - On most architectures this requires the insertion of a serializing instruction (e.g. “lfence”)
  - Some architectures can use a conditional masking of the untrusted_offset
    - Prevent it from ever (even speculatively) having an out-of-bounds value
  - Linux adds new “nospec” accessor macros to prevent speculative loads
- Tooling exists to scan source and binary files for offending sequences
  - Much more work is required to make this a less painful experience

Mitigating Spectre-v1: Bounds Check Bypass (cont.)

- Example of mitigated code sequence:

```
If (untrusted_offset < limit) {
    serializing_instruction();
    trusted_value = trusted_data[untrusted_offset];
    tmp = other_data[(trusted_value)&mask];
    ...
}
```



prevent load speculation

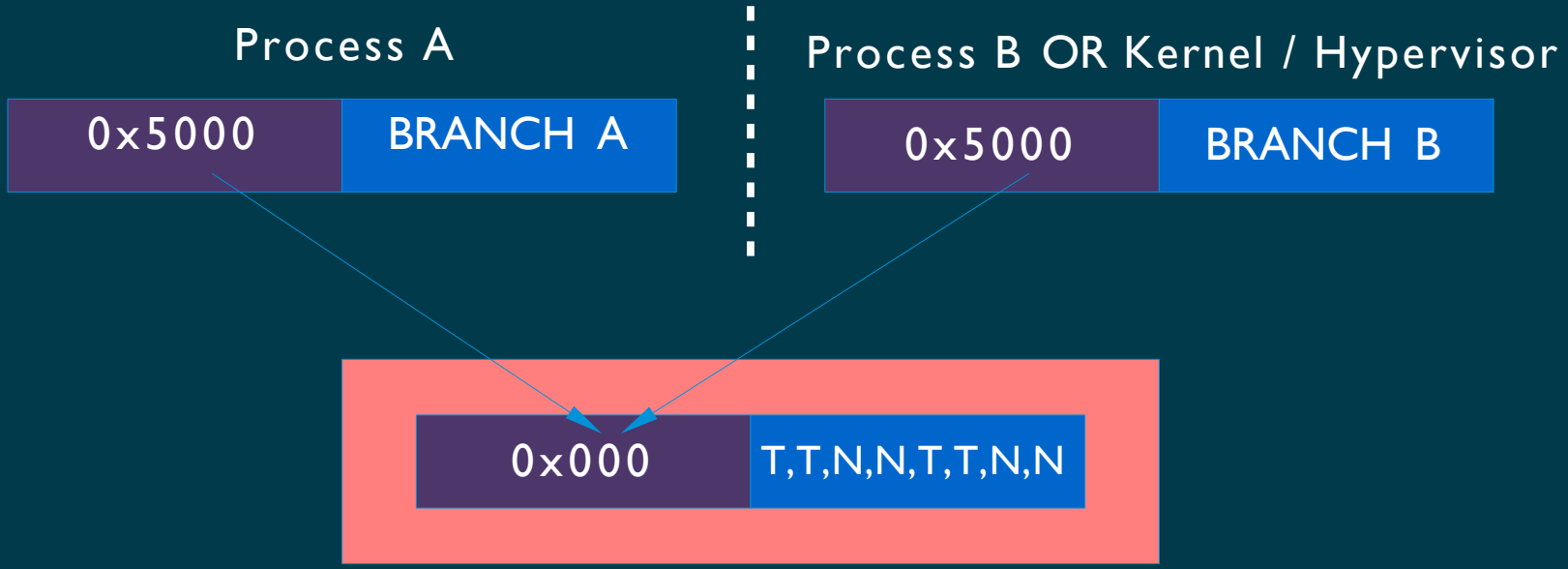
Mitigating Spectre-v1: Bounds Check Bypass (cont.)

- Another example of mitigated code sequence (e.g. Linux kernel):

```
If (untrusted_offset < limit) {
    untrusted_offset = array_index_nospec(untrusted_offset, limit);
    trusted_value = trusted_data[untrusted_offset];
    tmp = other_data[(trusted_value)&mask];
    ...
}
```

clamps value of untrusted_offset

Spectre-v2: Reminder on branch predictors



Spectre-v2: Branch Predictor Poisoning (CVE-2017-5715)

- Modern microprocessors may be susceptible to “poisoning” of the branch predictors
- Rogue application “trains” the indirect predictor to predict branch to “gadget” code
  - Processor incorrectly speculates down indirect branch into existing code but offset of the branch is under malicious user control – repurpose existing privileged code as a “gadget”
- Relies upon the branch prediction hardware not fully disambiguating branch targets
  - Virtual address of branch in malicious user code constructed to use same predictor entry as a branch in another application or the OS kernel running at higher privilege
- Privileged data is extracted using a similar cache access pattern to Spectre-v1

Mitigating Spectre-v2: Big hammer approach

- Existing branch prediction hardware lacks capability to disambiguate contexts
  - Relatively easy to add this in future cores (e.g. using ASID/PCID tagging in branches)
- Initial mitigation is to disable the indirect branch predictor hardware (sometimes)
  - Completely disabling indirect prediction would seriously harm core performance
  - Instead disable indirect branch prediction when it is most vulnerable to exploit
  - e.g. on entry to kernel or Hypervisor from less privileged application context
- Flush the predictor state on context switch to a new application (process)
  - Prevents application-to-application attacks across a new context
- A fine grained solution may not be possible on existing processors

Mitigating Spectre-v2: Big hammer (cont)

- Microcode can be used on some microprocessors to alter instruction behavior
- ...also used to add new "instructions" or system registers that exhibit side effects
- On Spectre-v2 impacted x86 microprocessors, microcode adds new SPEC_CTRL MSRs
  - Model Specific Registers are special memory addresses that control core behavior
  - Identified using the x86 "CPUID" instruction which enumerates available capabilities
  - IBRS (Indirect Branch Restrict Speculation)
    - Used on entry to more privileged context to restrict branch speculation
  - STIBP (Single Threaded Indirect Branch Predictor)
    - Use to force an SMT ("Hyperthreaded") core to predict on only one thread
  - IBPB (Indirect Branch Predictor Barrier)
    - Used on context switch into a new process to flush predictor entries
- **What are the problems with using microcode interfaces?**

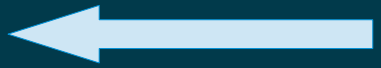
Mitigating Spectre-v2 with Retpolines

- Microcoded mitigations are effective but expensive due to their implementation
  - Many cores do not have convenient logic to disable predictors so “IBRS” must also disable independent logic within the core. It may take many thousands of cycles on kernel entry
- Google decided to try an alternative solution using a **pure software approach**
  - If indirect branches are the problem, then the solution is to avoid using them
  - “Retpolines” stand for “Return Trampolines” which replace indirect branches
  - Setup a fake function call stack and “return” in place of the indirect call

Mitigating Spectre with Retpolines (cont)

- Example retpoline call sequence on x86 (source: https://support.google.com/faqs/answer/7625886)

```
  call set_up_target;
capture_spec:
  pause;
  jmp capture_spec;
set_up_target:
  mov %r11, (%rsp);
  ret;
```

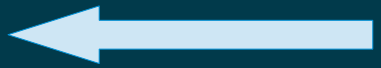


modify return stack to force “return” to target

Mitigating Spectre with Retpolines (cont)

- Example retpoline call sequence on x86 (source: https://support.google.com/faqs/answer/7625886)

```
  call set_up_target;
capture_spec:
  pause;
  jmp capture_spec;
set_up_target:
  mov %r11, (%rsp);
  ret;
```



harmless infinite loop for the CPU to speculate :)

* We might replace “pause” with “lfence” depending upon power/uarch

Mitigating Spectre-v2 with Retpolines (cont)

- Retpolines are a novel solution to an industry-wide problem with indirect branches
  - Credit to Google for releasing these freely without patent claims/encouraging adoption
- However they present a number of challenges for Operating Systems and users
  - Requires recompilation of software, possibly dynamic patching to disable on future cores
    - Mitigation should be temporary in nature, automatically disabled on future silicon
  - **Cores speculate return path from functions using an RSB (Return Stack Buffer)**
    - Need to explicitly manage (stuff) the RSB to avoid malicious interference
  - Certain cores will use alternative predictors when RSB underflow occurs

Mitigating Spectre-v2 with Retpolines (cont)

- Retpolines are a novel solution to an industry-wide problem with indirect branches
  - Credit to Google for releasing these freely without patent claims/encouraging adoption
- However they present a number of challenges for Operating Systems and users
  - Requires recompilation of software, possibly dynamic patching to disable on future cores
    - Mitigation should be temporary in nature, automatically disabled on future silicon
  - **Cores speculate return path from functions using an RSB (Return Stack Buffer)**
    - Need to explicitly manage (stuff) the RSB to avoid malicious interference
  - Certain cores will use alternative predictors when RSB underflow occurs

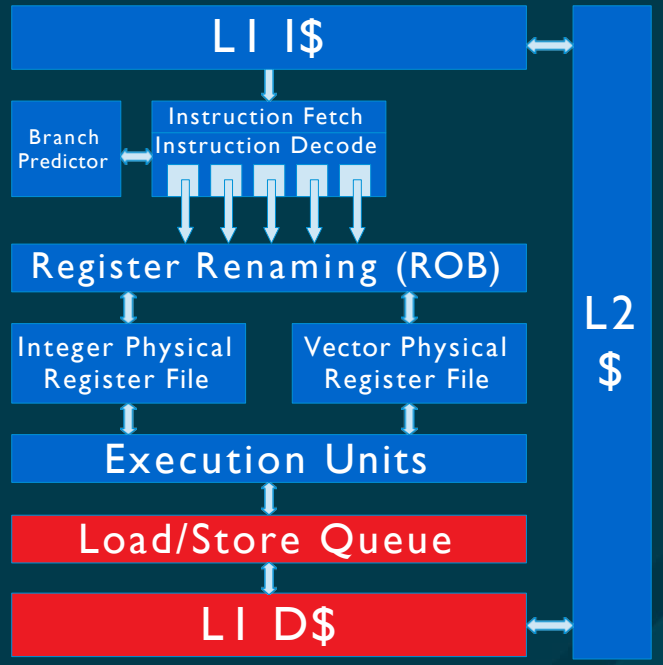
see SpectreRSB,"ret2spec", and other RSB vulnerabilties

Variations on a theme: variant 3a (Sysreg read)

- Variations of these microarchitecture attacks are likely to be found for many years
- An example is known as “variant 3a”. Some microprocessors will allow speculative read of privileged system registers to which an application should not have access
  - Can be used to determine the address of key structures such as page table base registers
- Sequence similar to meltdown but instead of data, access system registers
  - Extract the value by crossing the uarch/arch boundary in same way as in “Meltdown”

Variant 4: "Speculative Store Buffer Bypass"

- Recall that processors use "load/store" queues
  - These sit between the core and its cache hierarchy
- Recent stores may be to addresses we later read
  - The store might be obvious (e.g. to stack pointer)
  - But store may use register containing any address
  - Dynamically determine memory dependency
- Searching Load/Store queue takes some time
  - CAM (Content Addressable Memory)
  - Different alignments and sub-word
- Processor speculatively bypasses the store queue
  - Speculates there are no conflicting recent stores
  - May speculatively use older values of variables
  - Detect at retirement/unwind ("Disambiguation")



Variant 4: “Speculative Store Buffer Bypass” (cont.)

- Variant 4 targets same-context (e.g. JIT or scripted code in browser sandbox)
  - Also web servers hosting untrusted third-party code (e.g. Java)
- Can be creatively used to steer speculation to extract sandbox runtime secrets
- Mitigation is to disable speculative store bypassing in some cases
  - “Speculative Store Bypass Disable” (SSBD) is a new microcode interface on e.g. x86
  - We can tell the processor to disable this feature when needed (also on other arches)
  - Performance hit is typically a few percent, but worst case is 10+ percent hit
- Linux provides a global knob or a per-process “prctl”
  - The “prctl” is automatically used to enable the “SSBD” mitigation
  - e.g. Red Hat ship OpenJDK in a default-disable SSB configuration

Variations on a theme: LazyFPU save/restore

- Linux (and other OSes) used to perform “lazy” floating point “save/restore”
  - Floating point unit used to be a separate physical chip (once upon a time)
  - Hence we have a means to mark it “not present” and trap whenever it is used
  - On context switch from one process to another, don’t bother stashing the FP state
  - Many applications don’t use the FP, mark it unavailable and wait to see if they use it
- The “floating point” registers are used for many vectorized crypto operations
- Modern processors integrate the FPU and perform speculation including FP/AVX/etc.
  - It is possible to speculatively read the floating point registers from another process
  - Can be used to extract cryptographic secrets by monitoring the register state
- Mitigation is to disable lazy save/restore of the FPU
  - Which Linux has done by default for some time anyway (mostly vendor kernel issue)

Variations on a theme: Bounds Check Bypass Store (variant 1.x)

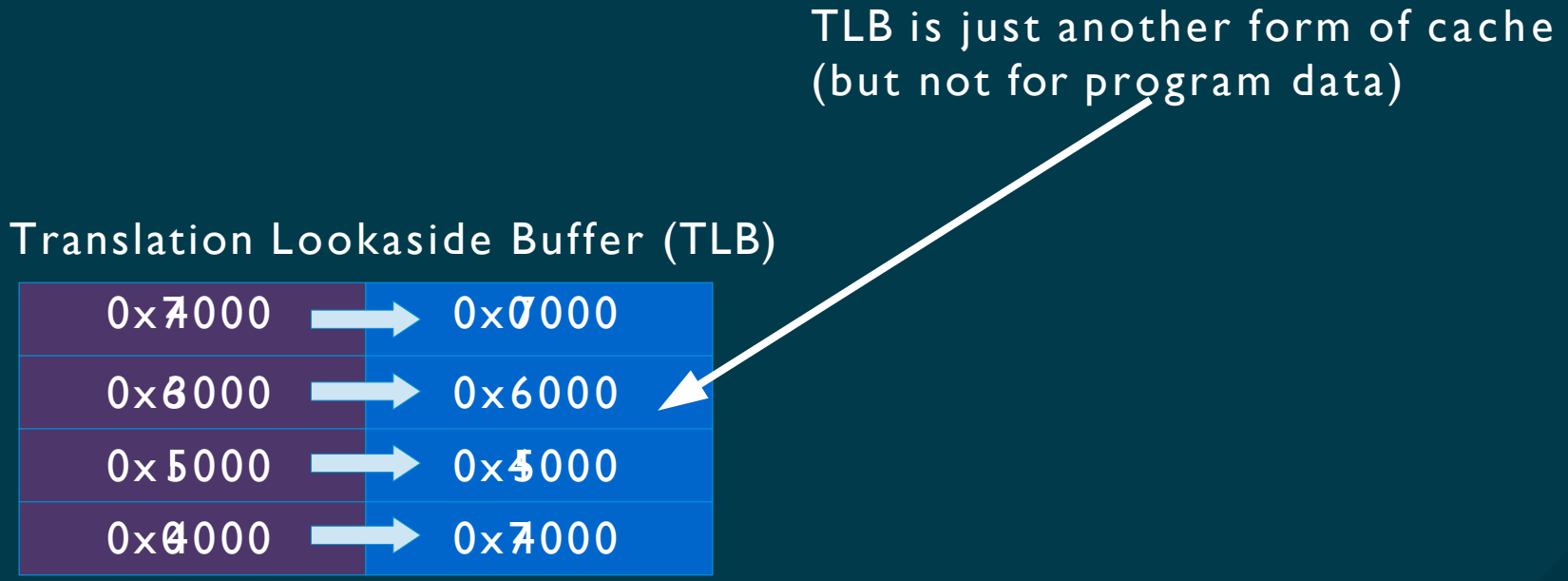
- The original Spectre-v1 disclosure gadget assumed a load following bounds check:

```
if (x < array1_size)
    y = array2[array1[x] * 256];
```

- It is possible to use a store following the bounds instead of a load
  - e.g. set a variable based upon some value within the array
- Mitigation is similar to load case but must locate+patch stores
  - Scanners such as “smatch” updated to account for “BCBS”

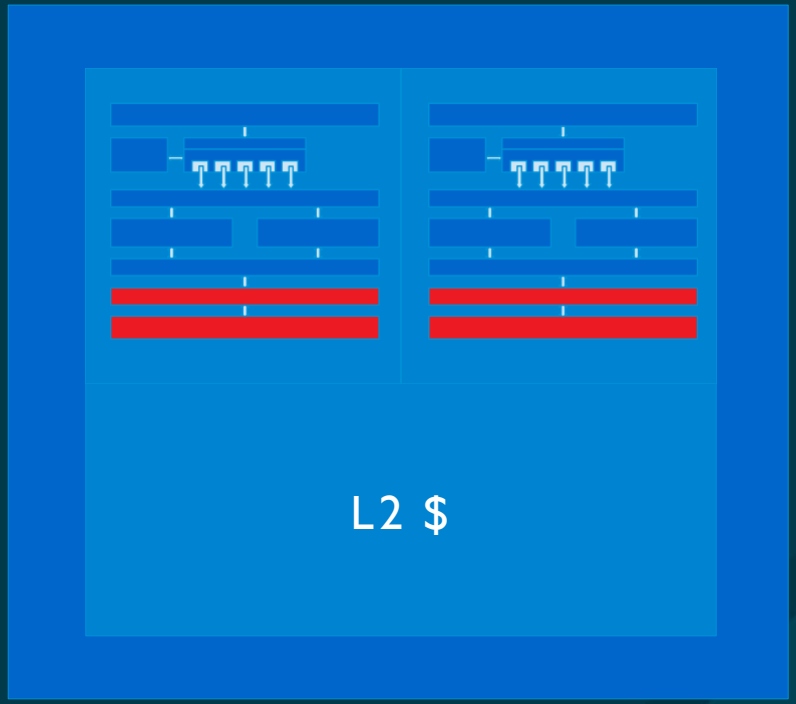
TLBleed – TLBs as a side-channel

TLB is just another form of cache
(but not for program data)



SMT (Simultaneous Multi-Threading)

- Recall that most “processors” are multi-core
- Cores may be partitioned into hw threads
  - Increases overall throughput by up to 30%
  - Can decrease perf. due to competition
- SMT productized into many designs including Intel’s “Hyper-threading” (HT) technology
  - This is what you see in “/proc/cpuinfo” as “sibling” threads of the same core
- Lightweight per-thread duplicated resources
  - Shared L1 cache, shared ROB, shared…
  - Separate context registers (e.g. arch GPRs)
  - Partitioning of some resources
- TLB partially competitively shared



TLBleed – TLBs as a side-channel

- TLBs similar to other caches (but cache memory translations, not their contents)
  - Formed from an (undocumented) hierarchy of levels, similar to caches
  - L1 i-side and d-side TLBs with a shared L2 sTLB
- Intel Hyper-threaded cores share data-side TLB resources between sibling threads
  - TLB not fully associative, possible to cause evictions in the peer
  - => Can observe the TLB activity of a peer thread
- TLBleed relies upon temporal access to data being measured
  - Requires co-resident hyper-threads between victim and attacker
  - Requires vulnerable software (e.g. some builds of libgcrypt)
  - Uses a novel machine learning approach to monitor TLBs
- Mitigation requires careful analysis of e.g. vulnerable crypto code
  - Can disable HT or apply process pinning strategies as well

NetSpectre – Spectre over the network

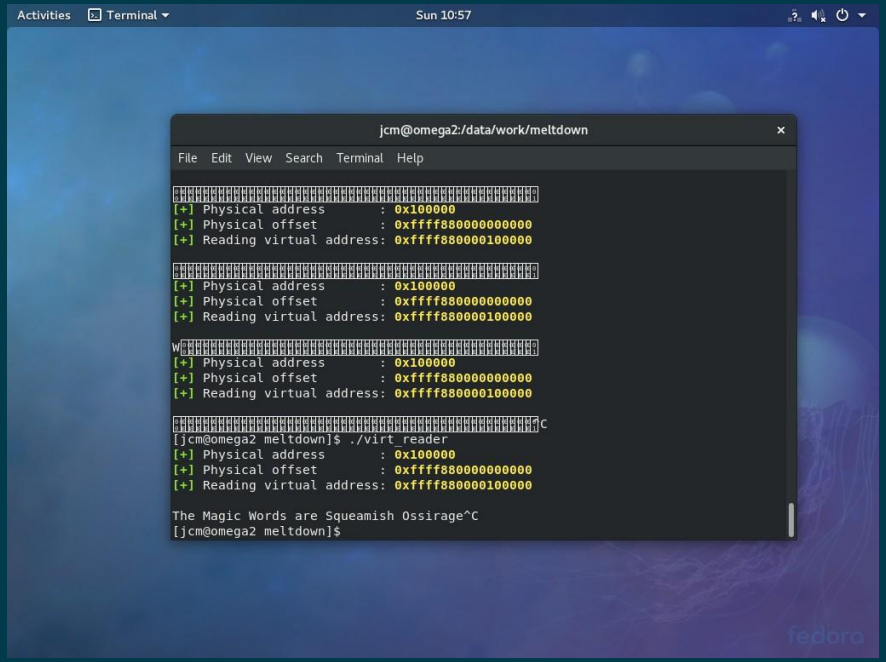
- Spectre attacks can be performed over the network by using two combined gadgets
  - A "leak" gadget sets some flag or state during speculative out of bounds access:

```
if (x < bitstream_length)
    if(bitstream[x])
        flag = true
```

  - A "transmit" gadget uses the flag during arbitrary operation that is remotely observable
    - e.g. during the transmission of some packet, check the flag value

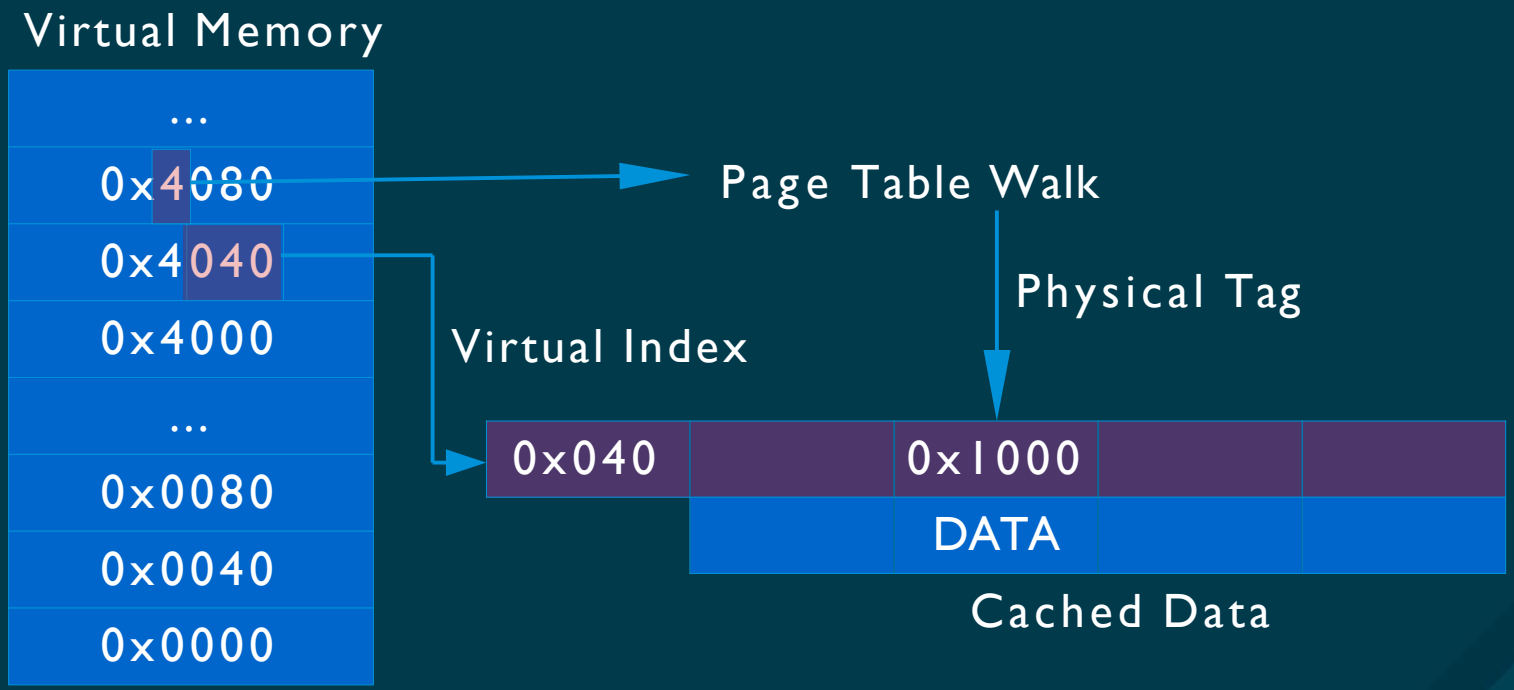
- An attacker trains the leak gadget then extracts data with the transmit gadget
  - Rate limited to bits per hour over the internet, detectable even among noise
  - Acceleration possible using e.g. an AVX2 power-down side-channel (Intel)

L1 Terminal Fault (aka “Foreshadow”)

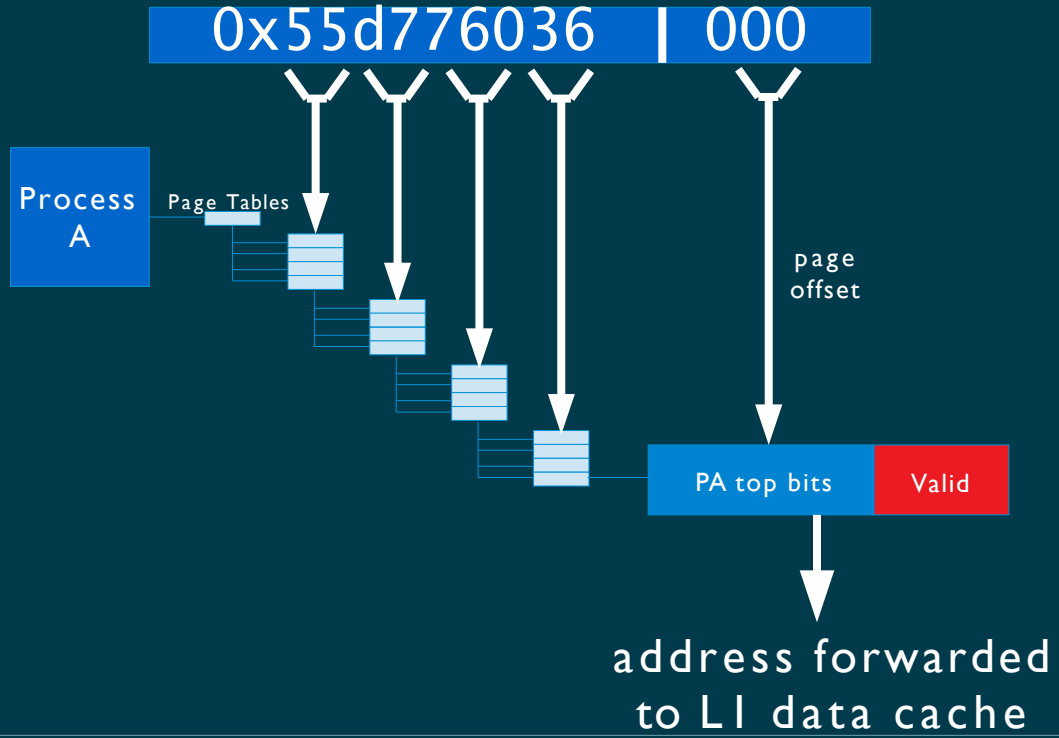


redhat.

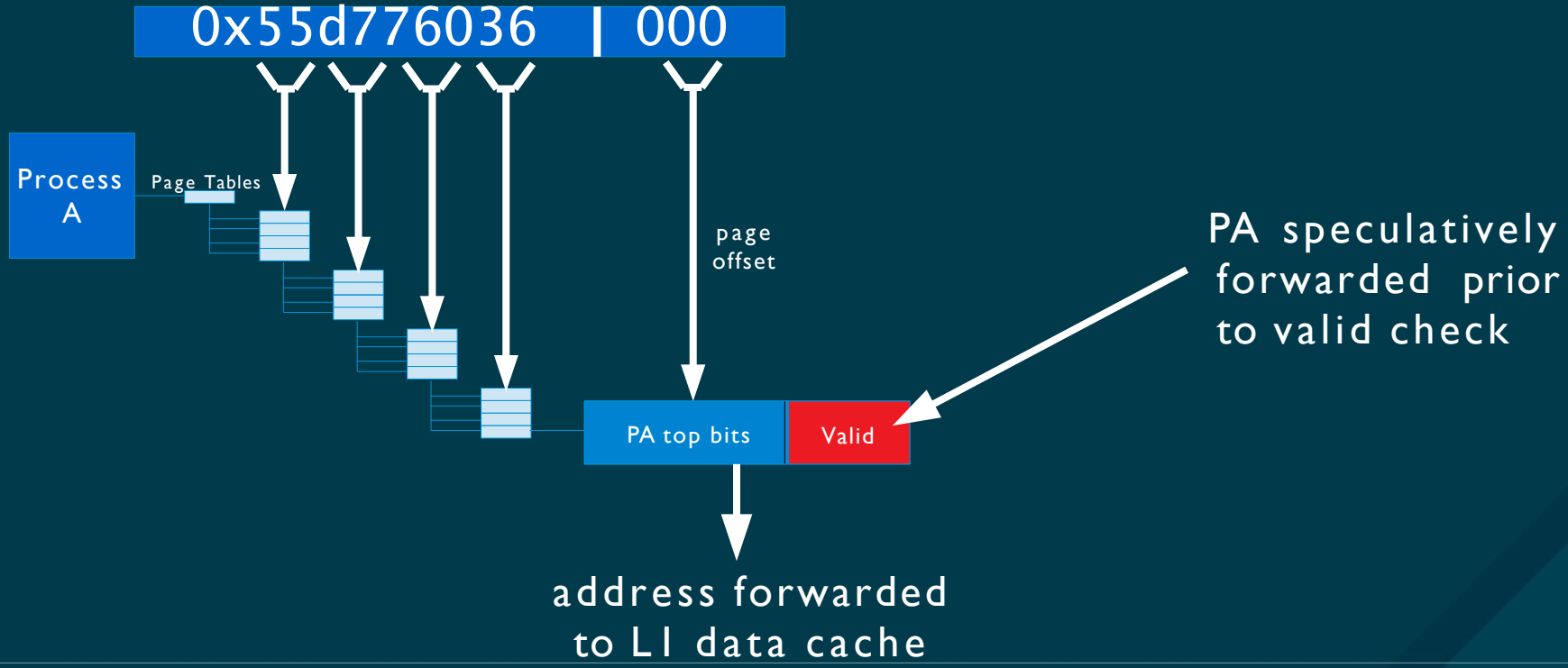
L1 Terminal Fault (aka “Foreshadow”)



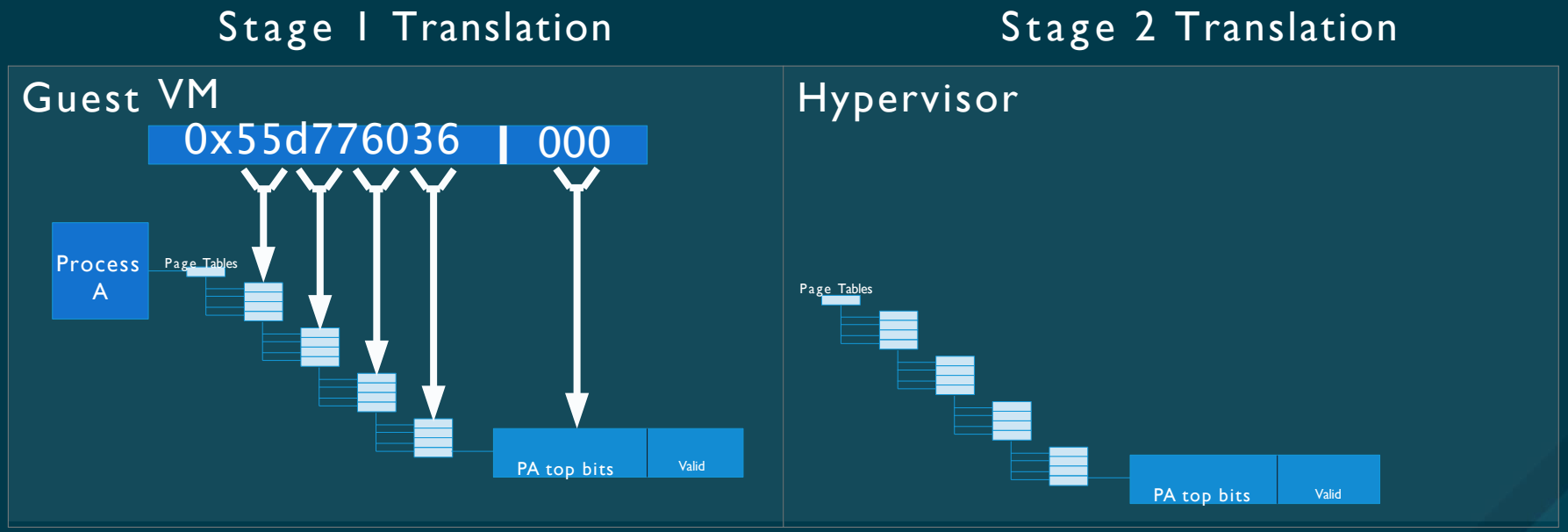
L1 Terminal Fault (aka “Foreshadow”)



L1 Terminal Fault (aka "Foreshadow")

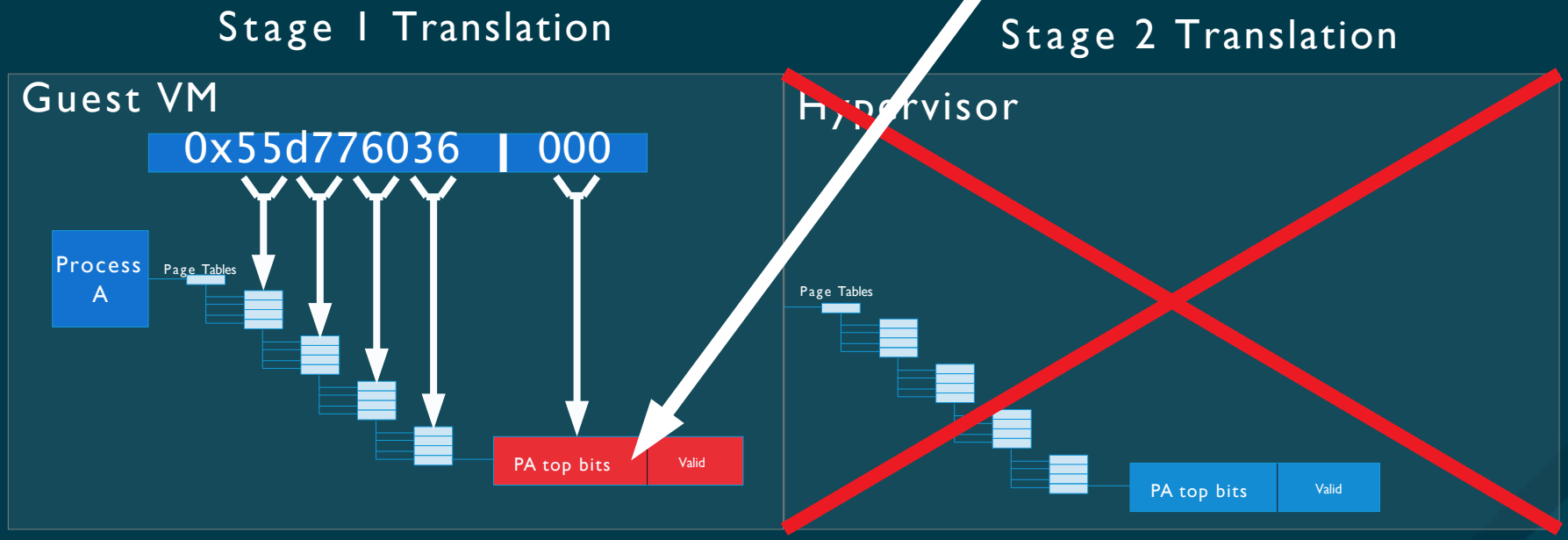


L1 Terminal Fault (aka "Foreshadow")



L1 Terminal Fault (aka "Foreshadow")

PA speculatively forwarded to L1 D$ (no stage2 translation)



L1 Terminal Fault (aka "Foreshadow")

- Operating Systems use PTE (Page Table Entry) valid bit for management
  - "paging" (aka "swapping") implemented by marking PTEs "not present"
  - Not present PTEs can be used to store OS metadata (disk addresses)
    - Specification says that all bits are ignored when not present
    - Linux stores the address on disk we swapped the page to
- Intel processors will speculate on validity of PTEs (Page Table Entries)
  - Forward the PA to the L1D$ prior to completing valid ("present") check
  - Common case (fast path) is that the PTE is "present" (valid)
  - "Terminal Fault" tagged in ROB for at-retirement handling
    - Similar to "Meltdown" and a similar underlying fix

L1 Terminal Fault (aka “Foreshadow”)

- A “not present” PTE can be used to speculatively read secrets
  - Contrive a not present PTE in the Operating System (bare metal attack)
- Mitigate this by ensuring OS never generates suitable PTEs
  - All swapped out pages are masked to generate PAs outside RAM
- **Terminating page walks do not undergo second stage translations**
  - Intel second stage (EPT) is ignored for not “present” PTEs
  - PA treated as a host address and forwarded to the cache
  - Can extract cached data from other Vms/Hypervisor
- Mitigate this by keeping secrets away from reach
  - Flush the L1D$ on entry into VM code (Linux)
  - Scrub secrets from the cache (e.g. Hyper-V)
  - Linux full mitigation may require HT disable

Related Research

- Meltdown and Spectre are only recent examples of microarchitecture attack
- A memorable attack known as “Rowhammer” was discovered previously
  - Exploit the implementation of (especially non-ECC) DDR memory
  - Possible to perturb bits in adjacent memory lines with frequent access
  - Can use this approach to flip bits in sensitive memory and bypass access restrictions
  - For example change page access permissions in the system page tables
- Another recent attack known as “MAGIC” exploits NBTI in silicon
  - Negative-bias temperature instability impacts reliability of MOSFETs (“transistors”)
  - Can be exploited to artificially age silicon devices and decrease longevity
  - Proof of concept demonstrated with code running on OpenSPARC core

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
  - Communication. No more “hardware” and “software” people. No more “us” and “them”.

Where do we go from here?

- 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
    - We can collaborate to explore novel solutions (yes, even us "software" people)
  - Opening up processor microcode/designs. Can you fully trust what you can't see?