Meltdown and Spectre

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

Usenix LISA 2018

MELTDOWN

SPECTRE

MELTDOWN

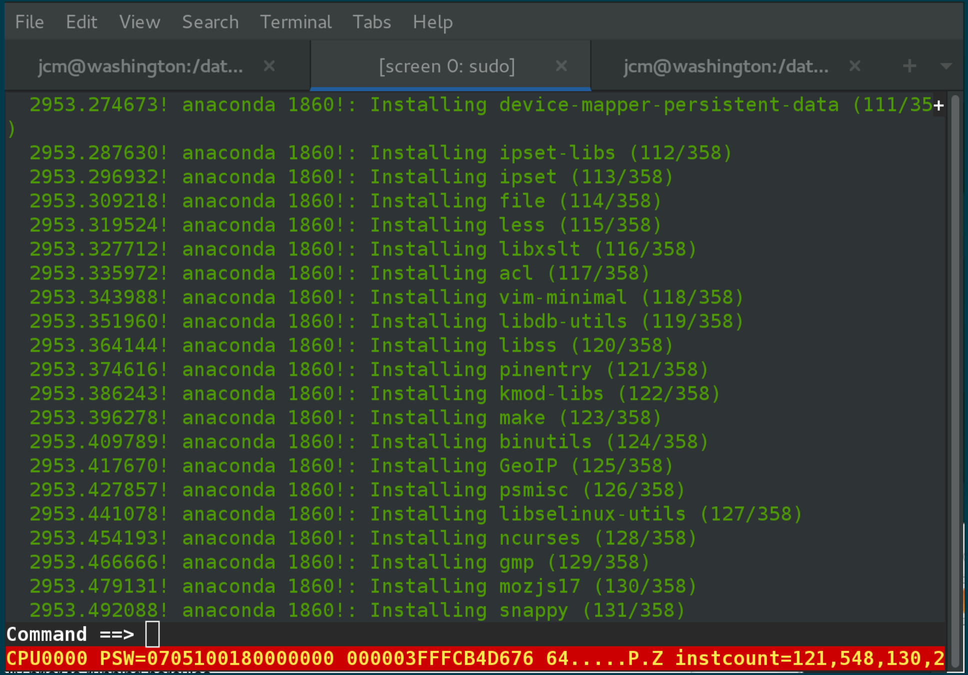
SPECTRE

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)

These issues impact everyone (not just “Intel”)



```
 2953.274673! anaconda 1860!: Installing device-mapper-persistent-data (111/35+
)
 2953.287630! anaconda 1860!: Installing ipset-libs (112/358)
 2953.296932! anaconda 1860!: Installing ipset (113/358)
 2953.309218! anaconda 1860!: Installing file (114/358)
 2953.319524! anaconda 1860!: Installing less (115/358)
 2953.327712! anaconda 1860!: Installing libxslt (116/358)
 2953.335972! anaconda 1860!: Installing acl (117/358)
 2953.343988! anaconda 1860!: Installing vim-minimal (118/358)
 2953.351960! anaconda 1860!: Installing libdb-utils (119/358)
 2953.364144! anaconda 1860!: Installing libss (120/358)
 2953.374616! anaconda 1860!: Installing pinentry (121/358)
 2953.386243! anaconda 1860!: Installing kmod-libs (122/358)
 2953.396278! anaconda 1860!: Installing make (123/358)
 2953.409789! anaconda 1860!: Installing binutils (124/358)
 2953.417670! anaconda 1860!: Installing GeoIP (125/358)
 2953.427857! anaconda 1860!: Installing psmisc (126/358)
 2953.441078! anaconda 1860!: Installing libselinux-utils (127/358)
 2953.454193! anaconda 1860!: Installing ncurses (128/358)
 2953.466666! anaconda 1860!: Installing gmp (129/358)
 2953.479131! anaconda 1860!: Installing mozjs17 (130/358)
 2953.492088! anaconda 1860!: Installing snappy (131/358)
Command ==>
CPU0000 PSW=0705100180000000 000003FFFCB4D676 64.....P.Z instcount=121,548,130,2
```

How did we get here?

How did we get here?

- **“Hardware” and “software” people don’t talk**
  - We created an “us” vs “them” collective mentality
  - We enjoy explicitly ignoring one another with intent
  - Software folks find hardware boring (and ignore it)
  - Hardware folks don’t have ways to engage ahead
  - Exceptions exist, but they are **NOT** the norm

- **“software” people: make friends with “hardware” people**
- **“hardware” people: make friends with “software” people**

Computer Architecture Refresher

Architecture

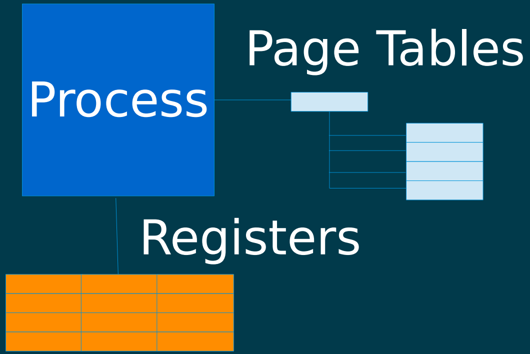
- **Instruction Set Architecture (ISA) describes contract between hw and sw**
  - Any machine implementing the same ISA is (potentially) compatible at a binary level
  - Defines instructions that all machines implementing the architecture must support
    - Load/Store from memory, architectural registers, stack, branches/control flow
    - Arithmetic, floating point, vector operations, and various possible extensions
  - Defines user (unprivileged, problem state) and supervisor (privileged) execution states
    - Exception levels used for software exceptions and hardware interrupts
    - Privileged registers used by the Operating System for system management
    - Mechanisms for application task context management and switching
  - Defines the memory model used by machines compliant with the ISA

Architecture (cont.)

- **The lowest level targeted by a programmer or (more often) compiler**
  - e.g. C code → assembly code → machine code
- **Most programmers assume simple sequential execution of their program**
  - Assume each line of code is executed in the sequence written
    - Compiler might re-order program instructions (build time)
    - Hardware might re-order program instructions (run time)
- **ISA traditionally timing-independent model of computer behavior**
  - Ignored timing-dependent optimizations present in many implementations
  - ...ISA was insufficiently defined in terms of additional behaviors (for both hw/sw)

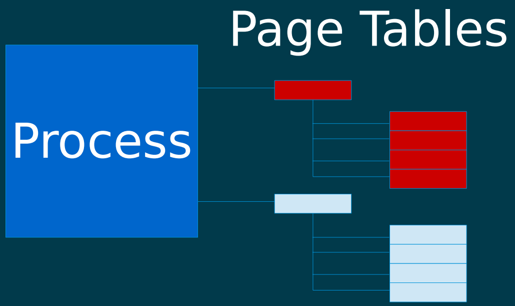
Application programs

- **Use non-privileged ISA instructions**
  - Execute at a lower privilege level than OS/Hypervisor
  - Often referred to as “rings”, “exception levels”, etc.
- **Have hardware context (state) when running**
  - General Purpose Registers (GPRs), Control Registers
  - e.g. “rax”, “rcx”, “rdx”, “rbx”, “rsp”, “rbp”…flags…
- **Execute within virtual memory environment**
  - Memory is divided into 4K (or larger) “pages”
  - OS managed using page tables (“address spaces”)
  - Memory Management Unit (MMU) translates all memory accesses using page tables (hardware)



Application programs

- **Request services from OS using system calls**
  - OS system calls also run in virtual memory
  - Linux (used to) map all of physical memory at the “top” of every process – kernel linked to run there
  - Low address (application), high address (kernel)
  - Page table protections (normally) prevent applications from seeing high mappings
- **Some applications have “sub-contexts”**
  - e.g. sandboxed browser JITs
  - Running at same privilege but different context



Operating Systems

- **OS uses additional privileged set of ISA instructions**
  - These include instructions to manage application context (registers, MMU state, etc.)
  - e.g. on x86 this includes being able to set the CR3 (page table base) control register
    - Page table base points to the top level of the page table structures
- **OS is responsible for switching between applications**
  - Save the process state (including registers), update the control registers
  - **Implication is that state is separated between contexts at process level**
- **OS maintains per-application page tables**
  - Hardware triggers a “page fault” whenever a virtual address is inaccessible
  - This could be because an application has been partially “swapped” (paged) out to disk
  - Or is being demand loaded, or does not have permission to access that address

Examples of computer architectures

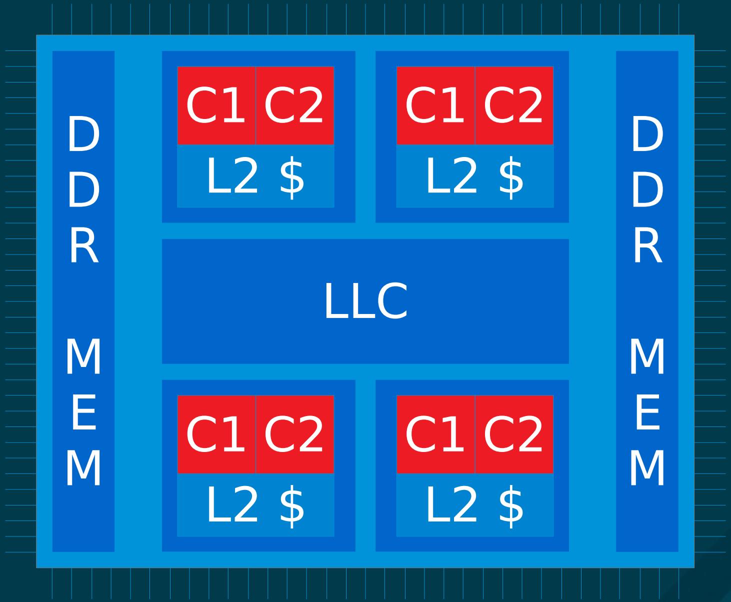
- Intel “x86” (Intel x64/AMD64)
  - CISC (Complex Instruction Set Computer)
  - Variable width instructions (up to 15 bytes)
  - 16 GPRs (General Purpose Registers)
  - Can operate “directly” on memory
  - 64-bit flat virtual address space
    - “Canonical” 48/56-bit addressing
    - Upper half kernel, Lower half user
    - Removal of older segmentation registers (except FS/GS)

- ARM ARMv8 (AArch64)
  - RISC (Reduced Instruction Set Computer)
  - Fixed width instructions (4 bytes fixed)
    - Clean uniform decode table
  - 31 GPRs (General Purpose Registers)
  - Classical RISC load/store using registers for all operations (first load from memory)
  - 64-bit flat virtual address space
    - Split into lower and upper halves

Computer Microarchitecture

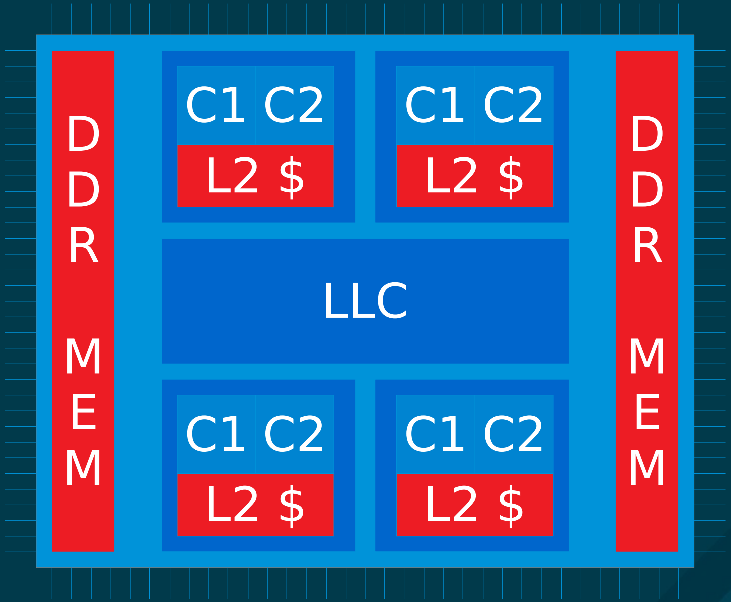
Elements of a modern System-on-Chip (SoC)

- Programmers often think in terms of “processors”
  - By which they usually mean “cores”
  - Some cores are “multi-threaded” (SMT) sharing execution resources between two threads
  - Minimal context separation is maintained through some (lightweight) duplication (e.g. arch registers)
- Many cores are integrated into today's SoCs
  - These are connected using interconnect(ion) networks and cache coherency protocols
  - Provides a hardware managed coherent view of system memory shared between cores
  - **Cores typically operate upon data in the L1 D$**
  - Cache is organized into “lines” of (e.g.) 64 bytes



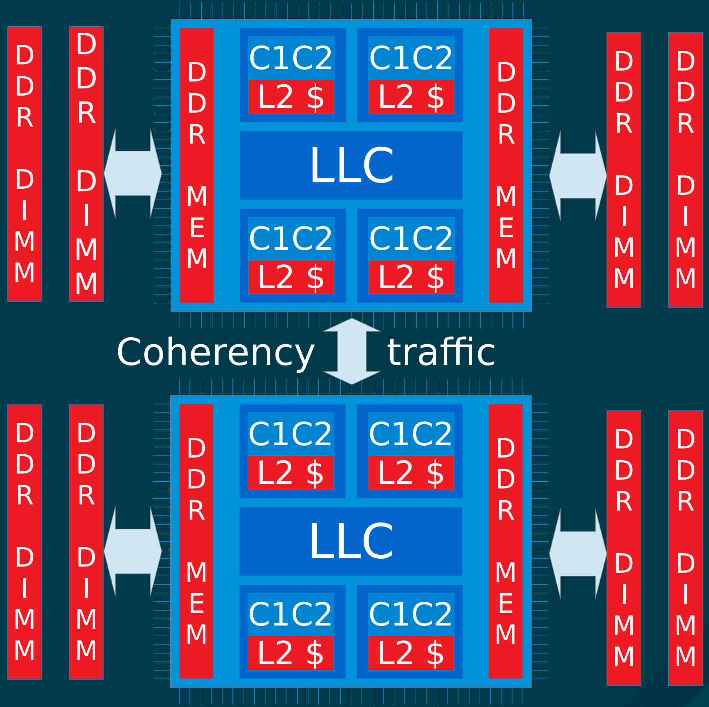
Elements of a modern System-on-Chip (SoC)

- Cache hierarchy sits between external (slow) RAM and (much faster) processor cores
  - Progressively tighter coupling from LLC (L3) through to L1 running at core speed (where compute happens)
  - Measurable performance difference between different levels of the cache hierarchy
  - Can use CPU cycle counting to measure access time and determine which level has data
- Memory controllers handle load/store of program instructions and data to/from RAM
  - Manage scheduling of DDR (or other memory) and sometimes leverage hardware access hints
  - Prefetchers will fetch “nearby” data, scheduler will balance data and instruction loads, as well as speculative and non-speculative data loads



The flow of data through a modern platform

- Caches exist to hide horrifying memory latencies
- Applications and Operating System code run from (DDR) memory attached to the SoC via its PHYs
- Current server chips have up to 8 “channels”
    - We are bound by the number of pins available on an SoC, which is now over 1,500 (up to 2,800+)
- Data is filled from memory through the DDR controller logic and loads into the caches
- Coherency protocols manage ownership of the (cached copies) of the data as it bounces around
    - This is why sharing data structures is expensive
- Processors taking a cache miss will track the load (Miss Status Holding Registers) and may continue

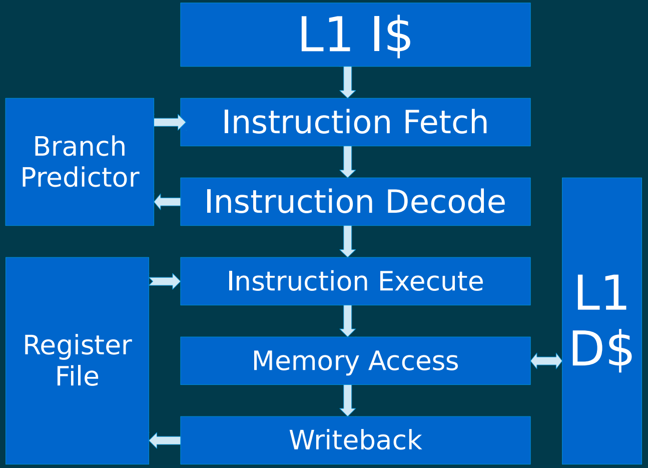


Microarchitecture (overview)

- Microarchitecture (“uarch”) refers to a specific implementation of an architecture
  - Compatible with the architecture defined ISA at a programmer visible level
  - Implies various design choices about the SoC platform upon which the core uarch relies
- Cores may be simpler “in-order” (similar to the classical 5-stage RISC pipeline)
  - Common in embedded microprocessors and those targeting low power points
  - Many Open Source processor designs leverage this design philosophy
  - Pipelining lends some parallelism without duplicating core resources
- Cores may be “out-of-order” similar to a dataflow machine inside
  - Programmer sees (implicitly assumed) sequential program order
  - Core uses an dataflow model with dynamic data dependency tracking
  - Results complete (retire) in-order to preserve sequential model

Elements of a modern in-order core

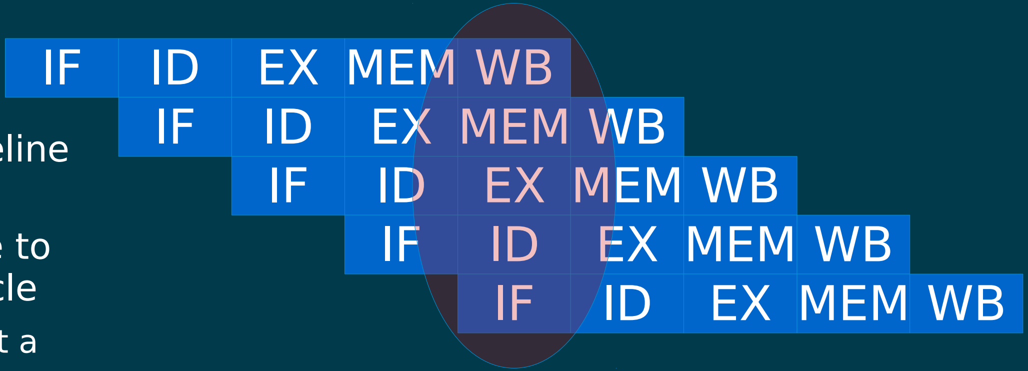
- Classical “RISC” pipeline taught first in CS courses
  - Pipelining splits processing into multiple cycles
  - Instructions may be at different pipeline “stages”
- 1. Instruction fetch from L1 Instruction Cache (I$)
  - L1$ automatically fills $ lines from “unified” L2/LLC
- 2. Instructions are then decoded according to ISA
  - defined in set of “encodings” (e.g. “add r3, r1, r2”)
- 3. Instructions are executed by the execution units
- 4. Memory access is performed to/from the L1 D$
- 5. The architectural register file is updated
  - e.g. r3 becomes the result of r1 + r2



* Intentionally simplified. Missing L2 Interface, load/store miss handling, etc.

In-order microarchitectures (continued)

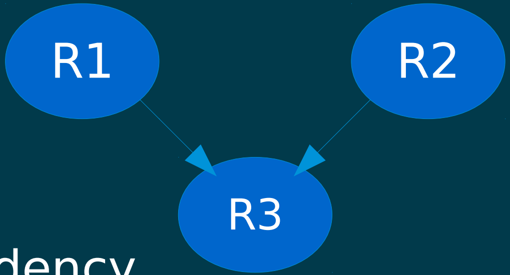
- In-order machines suffer from pipeline stalls when stages are not ready
- Memory access stage may be able to load from the L1 D$ in a single cycle
  - But if not in the L1 D$ then insert a pipeline “bubble” while waiting
- Limited capability to hide latency
  - Future instructions may not depend upon stalling earlier instructions
- Limited branch prediction (if at all)
  - Typically squash a few pipeline stages

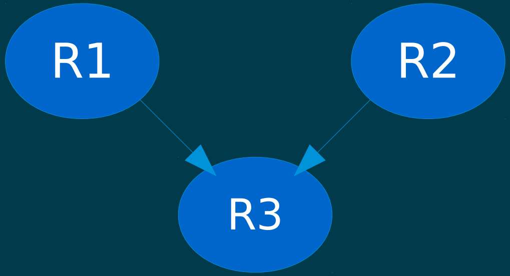


Out-of-Order (OoO) microarchitectures

R1 = LOAD A
R2 = LOAD B
R3 = R1 + R2
R1 = 1
R2 = 1
R3 = R1 + R2

No data dependency





Out-of-Order (OoO) microarchitectures

R1 = LOAD A
R2 = LOAD B
R3 = R1 + R2
R1 = 1
R2 = 1
R3 = R1 + R2

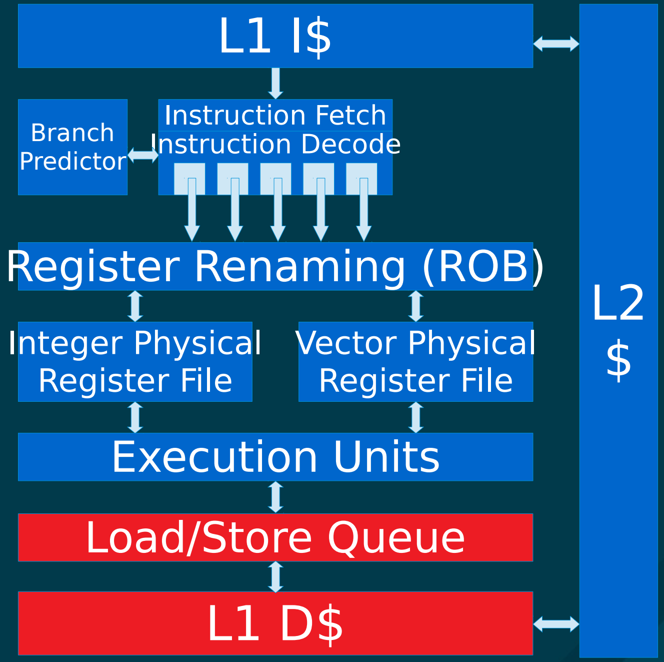
Program Order

Entry	RegRename	Instruction	Deps	Ready?
1	P1 = R1	P1 = LOAD A	X	Y
2	P2 = R2	P2 = LOAD B	X	Y
3	P3 = R3	P3 = R1 + R2	1,2	N
4	P4 = R1	P4 = 1	X	Y
5	P5 = R2	P5 = 1	X	Y
6	P6 = R3	P6 = P4 + P5	4,5	N

Re-Order Buffer (ROB)

Elements of a modern out-of-order core

- Modern performance is achieved via Out-of-Order
  - Backbone of last several decades (+Speculation)
- In-order “front end” issue, out-of-order “backend”
- Instructions are fetched and decoded
  - Decoder might be variable throughput (x86)
- Renamed and scheduled onto execution units
- “Load” or “Store” instructions operate on L1D$
  - Handled by a load/store execution unit (port)
- Load/Store Queue handles outstanding load/store
  - Also known as “load buffer”/ “store buffer”
  - Outstanding loads must search for recent stores



Out-of-Order (OoO) microarchitectures

- This type of design is common in aggressive high performance microprocessors
  - Also known as “dynamic execution” because it can change at runtime
  - Invented by Robert Tomasulo (used in System/360 Model 91 Floating Point Unit)
- Instructions are fetched and decoded by an in-order “front end” similar to before
- Instructions are dispatched to an out-of-order “backend”
  - Allocated an entry in a ROB (Re-Order Buffer), Reservation Stations
  - May use a Re-Order Buffer and separate Retirement (Architectural) Register File or single physical register file and a Register Alias Table (RAT)
- Re-Order Buffer defines an execution window of out-of-order processing
  - These can be quite large – over 200 entries in contemporary designs
- Resource sharing occurs between sibling SMT threads (e.g. Intel “hyper-threads”)
  - Share the same ROB and other structures, interference possible between threads

Out-of-Order (OoO) microarchitectures (cont.)

- Instructions wait only until their dependencies are available
  - Later instructions may execute prior to earlier instructions
  - Re-Order Buffer allows for more physical registers than defined by the ISA
  - Removes some so-called data “hazards”
    - WAR (Write-After-Read) and WAW (Write-After-Write)
- Instructions complete (“retire”) in-order
  - When an instruction is the oldest in the machine, it is “retired”
  - State becomes architecturally visible (updates the architectural register file)

Microarchitecture (summary)

- **The term microarchitecture (“uarch”) refers to a specific implementation**
    - Implies various design choices about the SoC platform upon which the core uarch relies
- Questions we can ask about a given implementation include the following:
    - What's the design point for an implementation – Power vs Performance vs Area (cost)
        - Low power simple in-order design vs Fully Out-of-Order high performance design
    - How are individual instructions implemented? How many cycles do they take?
        - How many pipelines are there? Which instructions can issue to a given pipe?
    - How many microarchitectural registers are implemented?
        - How many ports in the register file?
    - How big is the Re-Order Buffer (ROB) and the execution window?

Examples of computer microarchitectures

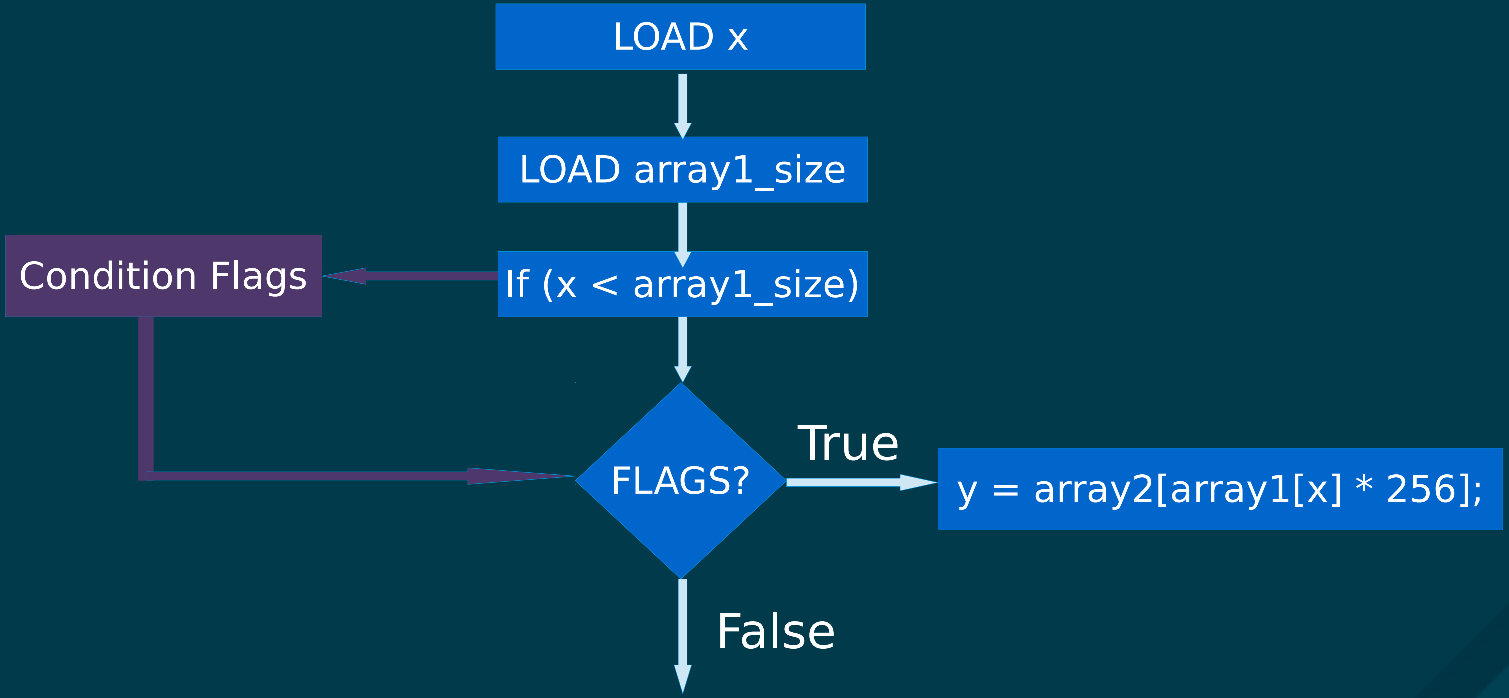
- Intel Core i7-6560U (“Skylake” uarch)
  - 2 SMT threads per core (configurable)
  - 32KB L1I$, 32KB L1D$, 256KB L2$
  - 4-8* uops instruction issue per cycle
  - 8 execution ports (14-19 stage pipeline)
  - 224 entry ROB (Re-Order Buffer)
  - 14nm FinFET with 13 metal layers

  * Typical is 4uops with rare exception

- IBM POWER8E (POWER8 uarch)
  - Up to 8 SMT threads per core (configurable)
  - 32KB L1I$, 64KB L1D$, 512KB L2$
  - 8-10 wide instruction issue per cycle
  - 16 execution piplines (15-23 stage pipeline)
  - 224 entry Global Completion Table (GCT)
  - 22nm SOI with 15 metal layers

Branch Prediction and Speculation

Branch prediction



Branch prediction

- **Applications frequently use program control flow instructions (branches)**
  - Conditionals such as “if then” are implemented as conditional direct branches
  - e.g. “if (raining) pack_umbrella();” depends upon the value of “raining”
- Branch condition evaluation is known as “resolving” the branch condition
  - This might require (slow) loads from memory (e.g. not immediately in the L1 D$)
- **Rather than wait for branch resolution, predict outcome of the branch**
  - This keeps the pipeline(s) filled with (hopefully) useful instructions
- Some ISAs allow compile-time “hints” to be provided for branches
  - These are encoded into the branch instruction, but may not be used
  - “if (likely(condition))” sequences in Operating System kernels

Speculative Execution

```
R1 = LOAD A
TEST R1
IF R1 ZERO {
R1 = 1
R2 = 1
R3 = R1 + R2
```

Entry	RegRename	Instruction	Deps	Ready?	Spec?
1	P1 = R1	P1 = LOAD A	X	Y	N
2		TEST R1	1	Y	N
3		IF R1 ZERO {	1	N	N
4	P2 = R1	P4 = 1	X	Y	Y*
5	P3 = R2	P5 = 1	X	Y	Y*
6	P4 = R3	P4 = P2 + P3	4,5	Y	Y*

* Speculatively execute the branch before the condition is known (“resolved”)

Speculative Execution

- Speculative Execution is implemented as a variation of Out-of-Order Execution
  - Uses the same underlying structures already present such as ROB, etc.
- Instructions that are speculative are specially tagged in the Re-Order Buffer
  - They **must not** have an architecturally visible effect on the machine state
  - Do not update the architectural register file until speculation is committed
  - Stores to memory are tagged in the ROB and will not hit the store buffers
- Exceptions caused by instructions will not be raised until instruction retirement
  - Tag the ROB to indicate an exception (e.g. privilege check violation on load)
  - If the instruction never retires, then no exception handling is invoked

Branch prediction and speculation

- Once the branch condition is successfully resolved:
  1) **If predicted branch correct, speculated instructions can be retired**
     - Once instructions are the oldest in the machine, they can retire normally
     - They become architecturally visible and stores ultimately reach memory
     - Exceptions are handled for instructions failing an access privilege check
     - Significant performance benefit from executing the speculated path
  2) **If predicted branch incorrect, speculated instructions MUST be discarded**
     - They exist only in the ROB, remove/fix, and discard store buffer entries
     - They do not become architecturally visible
     - Performance hit incurred from flushing the pipeline/undoing speculation
     - **For decades we assumed that this was a magic "black box"**

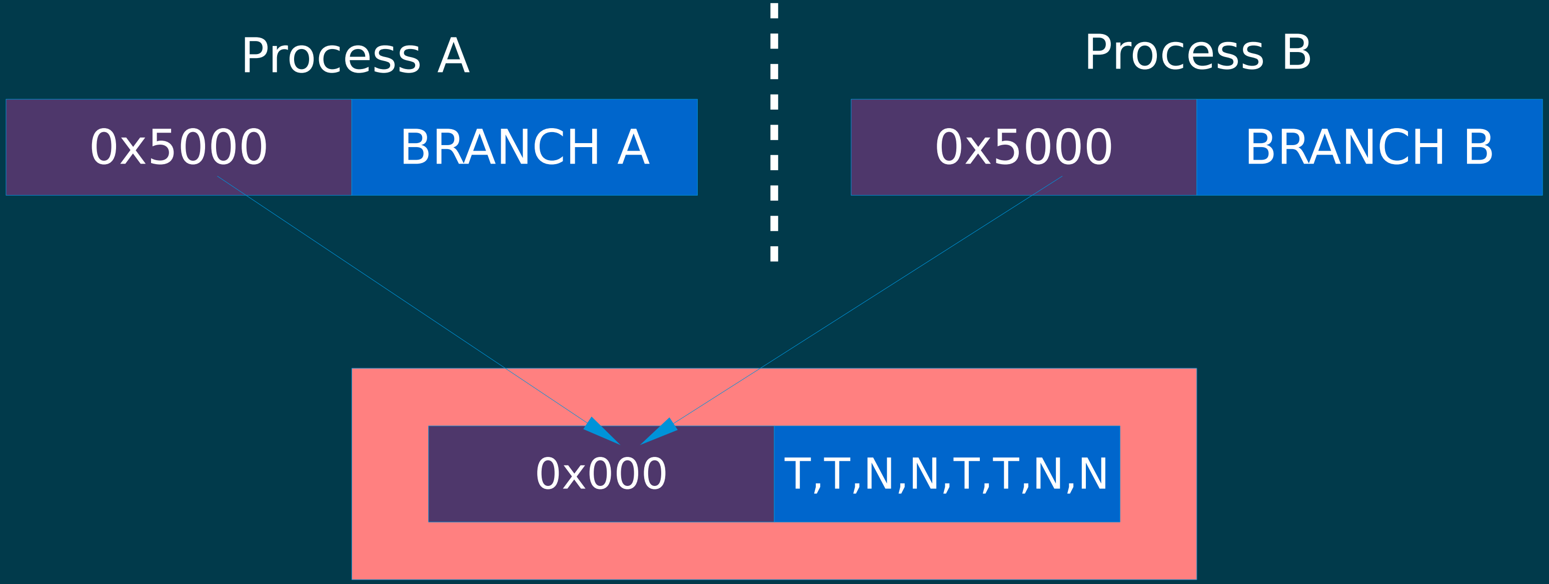
Conditional branches

- A conditional branch will be performed based upon the state of the condition flags
  - Condition flags are commonly implemented in modern ISAs and set by certain instructions
  - Some ISAs are optimized to set the condition flags only in specific instruction variants
- Most loops are implemented as a conditional backward jump following a test:

```
       movq $0, %rax
loop:
       incq %rax
       cmpq $10, %rax
       jle loop
```

Predict the jump
(in reality would use loop predictor)

Conditional branch prediction



Branch predictor (uses history buffer)

Conditional branch prediction

- Branch behavior is rarely random and can usually be predicted with high accuracy
  - Branch predictor is first “trained” using historical direction to predict future
  - Over 99% accuracy is possible depending upon the branch predictor sophistication
- Branches are identified based upon the (virtual) address of the branch instruction
  - Index into branch prediction structure containing pattern history e.g. T,T,N,N,T,T,N,N
  - These may be tagged during instruction fetch/decode using extra bits in the I$
- Most contemporary high performance branch predictors combine local/global history
  - Recognizing that branches are rarely independent and usually have some correlation
  - A Global History Register is combined with saturating counters for each history entry
  - May also hash GHR with address of the branch instruction (e.g. “Gshare “ predictor)

Conditional branch prediction

- Modern designs combine various different branch predictors
  - Simple loop predictors include BTFN (Backward Taken Forward Not)
  - Contemporary processors would identify the previous example as early as decode
  - May directly issue repeated loop instructions from pre-decoded instruction cache
- Optimize size of predictor internal structures by hashing/indexing on address
  - Common not to use the full address of a branch instruction in the history table
  - This causes some level of (known) interference between unrelated branches

Indirect branch prediction

- More complex branch types include “indirect branches”
  - Target address stored in register/memory location (e.g. function pointer, virtual method)
  - The destination of the branch is not known at compile time (known as “dynamic”)
- Indirect predictor attempts to guess the location of an indirect branch
  - Recognizes the branch based upon the (virtual) address of the instruction
  - Uses historical data from previous branches to guess the next time
- Speculation occurs beyond indirect branch into predicted target address
  - If the predicted target address is incorrect, discard speculative instructions

Branch predictor optimization

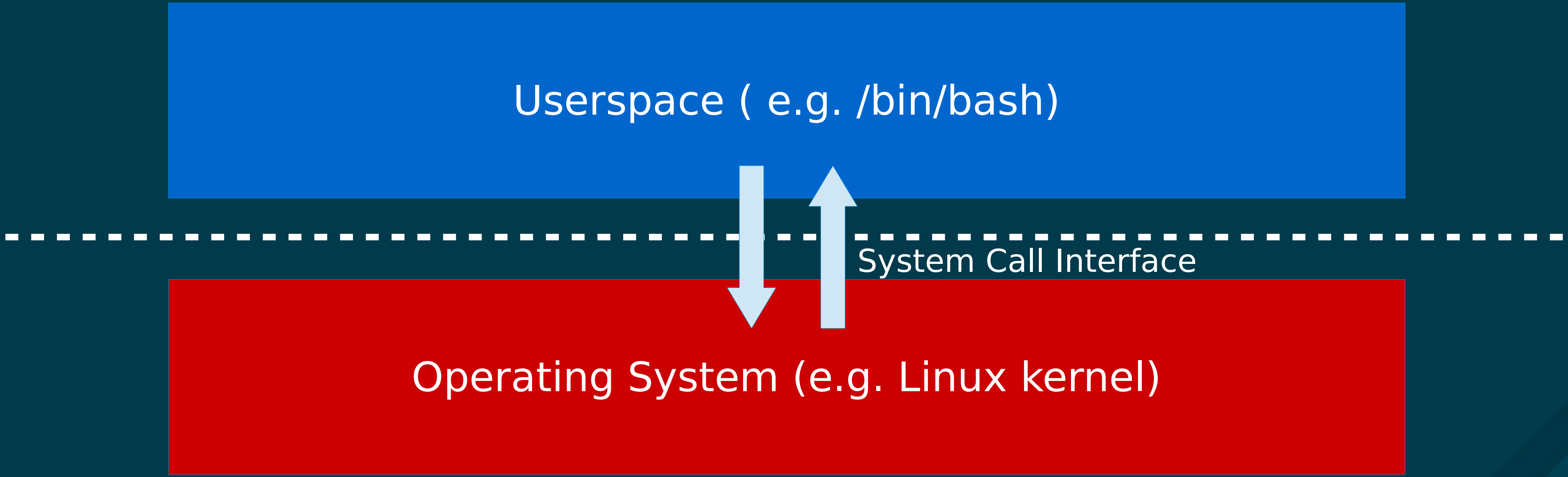
- Branch prediction is vital to modern microprocessor performance
  - Significant research has gone into optimization of prediction algorithms
  - Many different predictors may be in use simultaneously with voting arbitration
  - Accuracy rates of over 99% are possible depending upon the workload
- Predictors are in the critical path for instruction fetch/decode
  - Must operate quickly to prevent adding delays to instruction dispatch
  - Common industry optimizations aimed at reducing predictor storage
  - Optimizations include indexing on low order address bits of branches

Speculation in modern processors

- Modern microprocessors heavily leverage speculative execution of instructions
  - This achieves significant performance benefit at the cost of complexity and power
  - Required in order to maintain the level of single thread performance gains anticipated
- Speculation may cross contexts and privilege domains (even hypervisor entry/exit)
- Conventional wisdom holds that speculation is invisible to programmer/applications
  - Speculatively executed instructions are discarded and results flushed from store buffers
  - However speculation may result in cache loads (allocation) for values being processed
- **It is now realized that certain side effects of speculation may be observable**
  - This can be used in various exploits against popular implementations

Virtual Memory and Caches

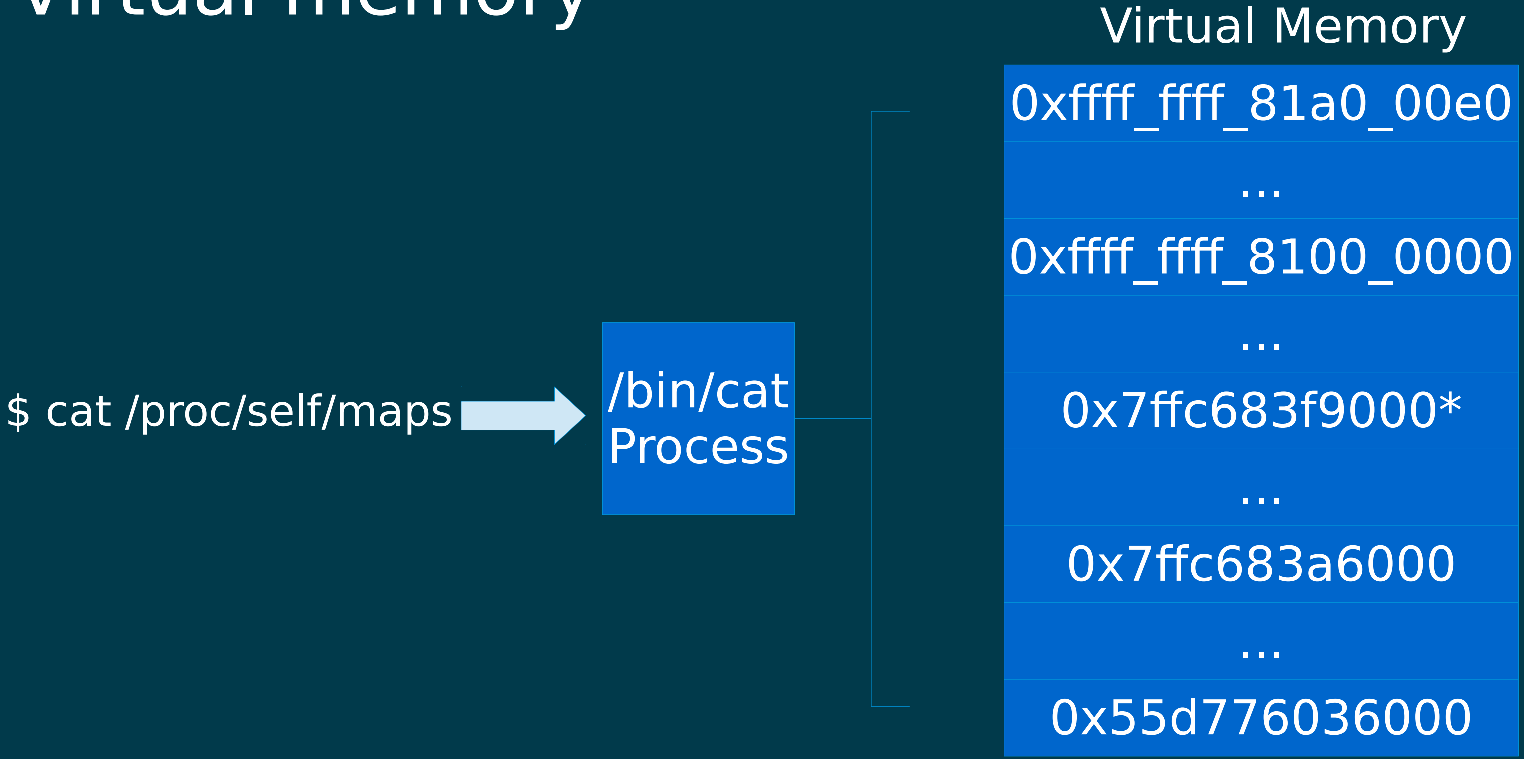
Userspace vs. Kernelspace


Userspace (e.g. /bin/bash)
System Call Interface
Operating System (e.g. Linux kernel)

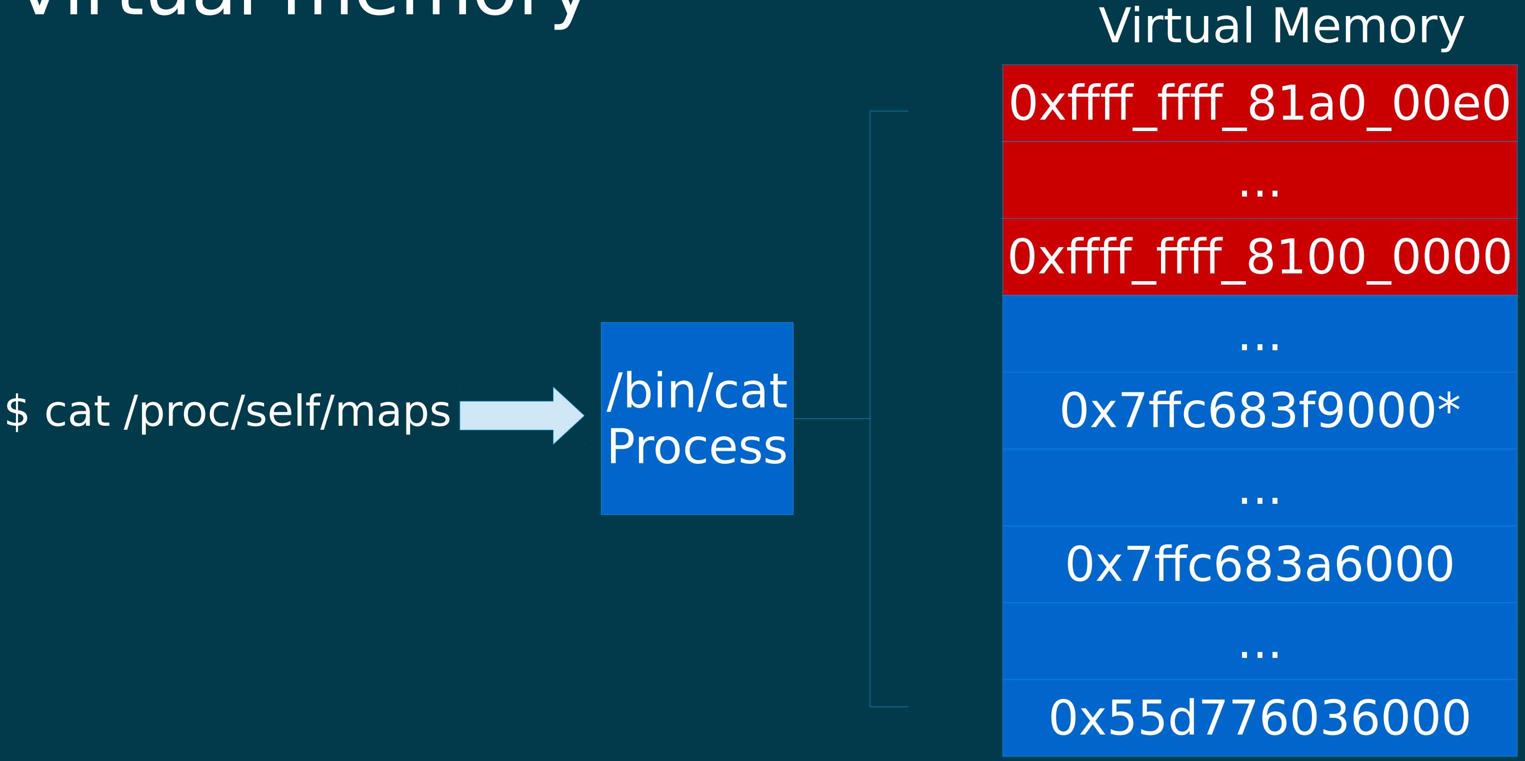
Userspace vs. Kernelspace

- User applications are known as “processes” (or “tasks”) when they are running
- They run in “userspace”, a less privileged context with many restrictions imposed
  - Managed through special hardware interfaces (registers) as well as other structures
  - We will look at an example of how “page tables” isolate kernel and userspace shortly
- Applications make “system calls” into the kernel to request services
  - For example “open” a file or “read” some bytes from an open file
  - Enter the kernel briefly using a hardware provided mechanism (syscall interface)
  - A great amount of optimization has gone into making this a lightweight entry/exit
- Special optimizations exist for some frequently used kernel services
  - VDSO (Virtual Dynamic Shared Object) looks like a shared library but provided by kernel
  - When you do a gettimeofday (GTOD) call you actually won't need to enter the kernel

Virtual memory

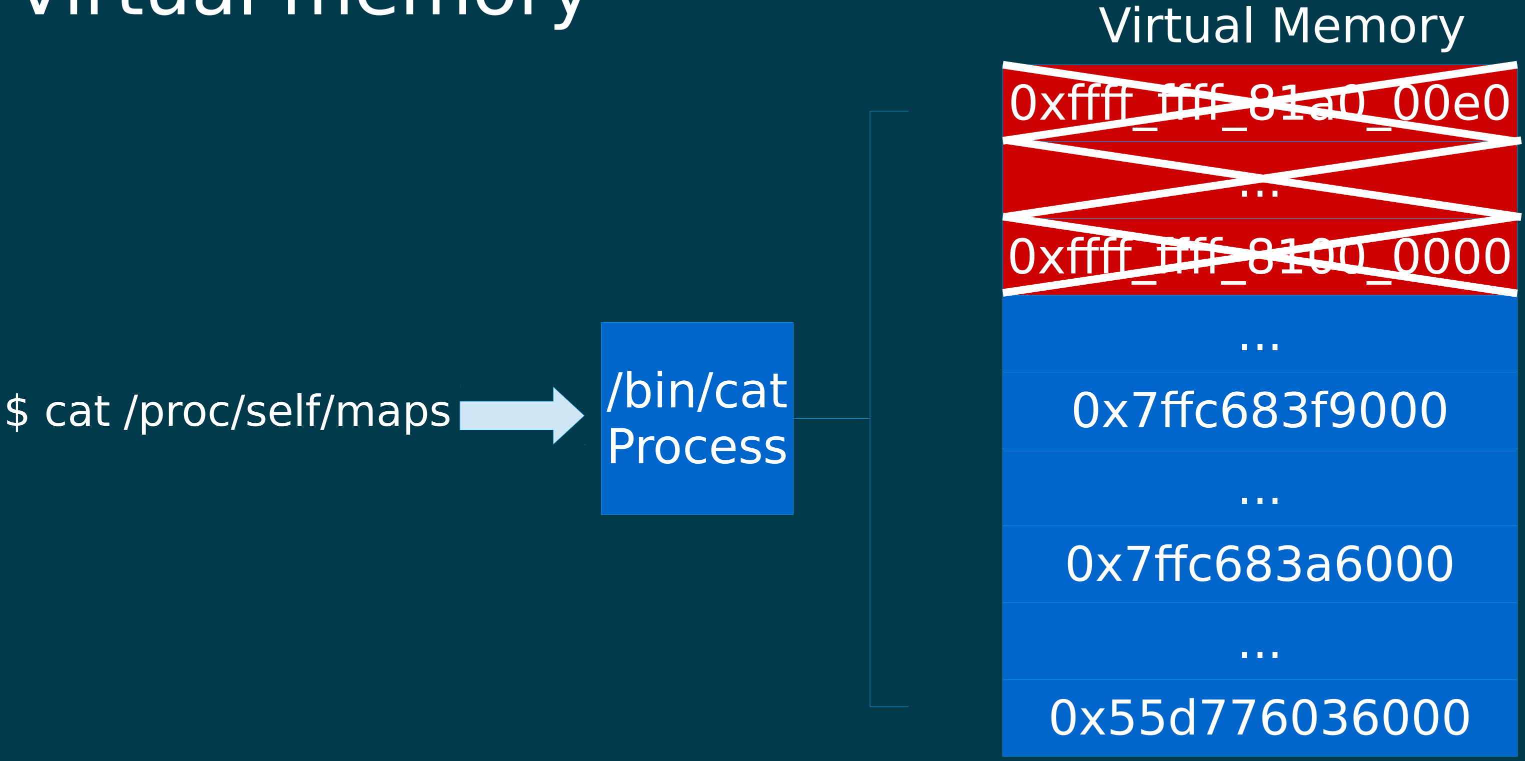


Virtual memory

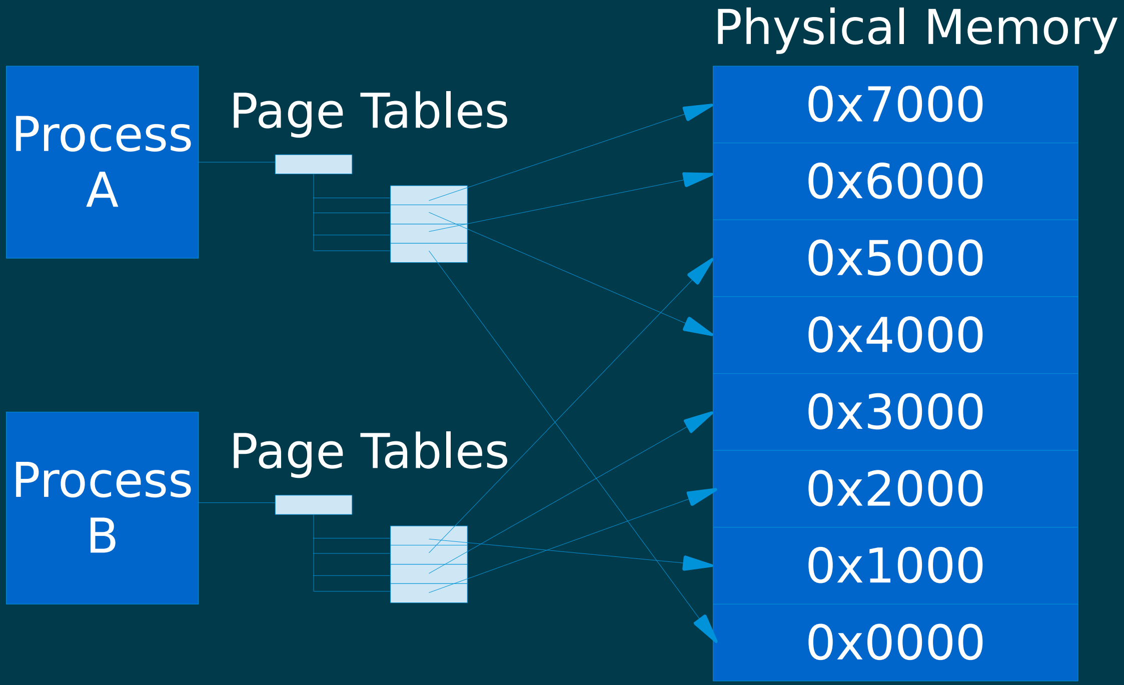


* Special case kernel VDSO (Virtual Dynamic Shared Object)

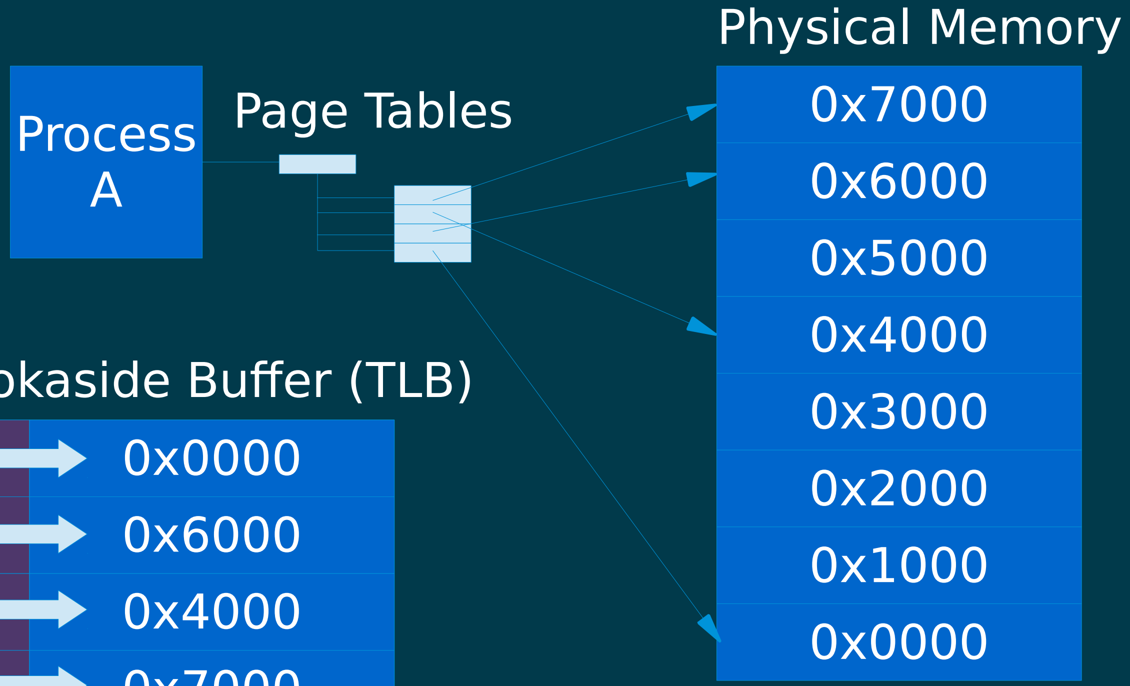
Virtual memory



Virtual memory



Virtual memory



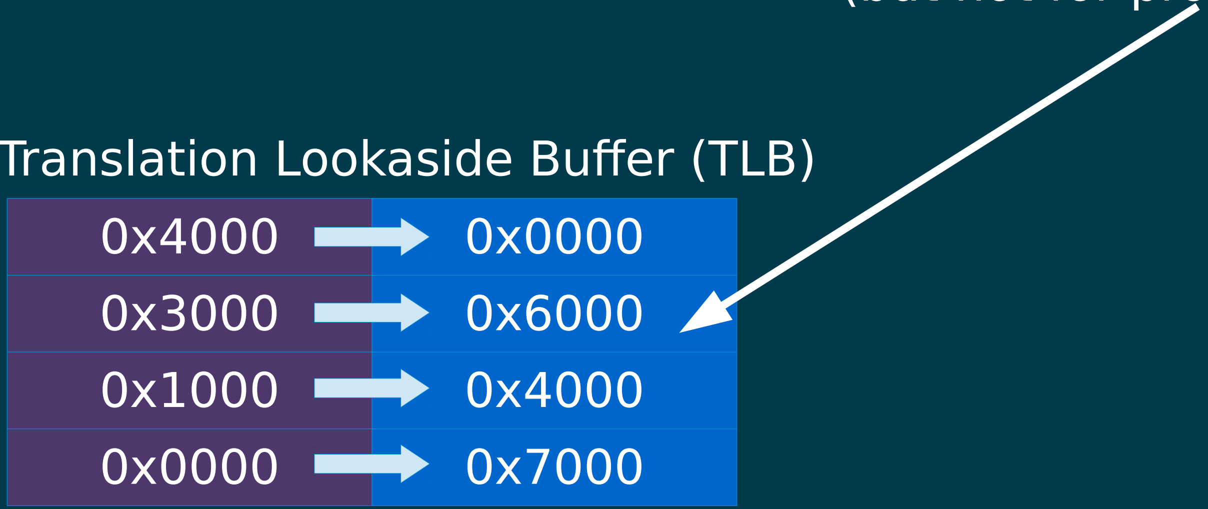
Translation Lookaside Buffer (TLB)

0x4000 ➡	0x0000
0x3000 ➡	0x6000
0x1000 ➡	0x4000
0x0000 ➡	0x7000

Virtual memory

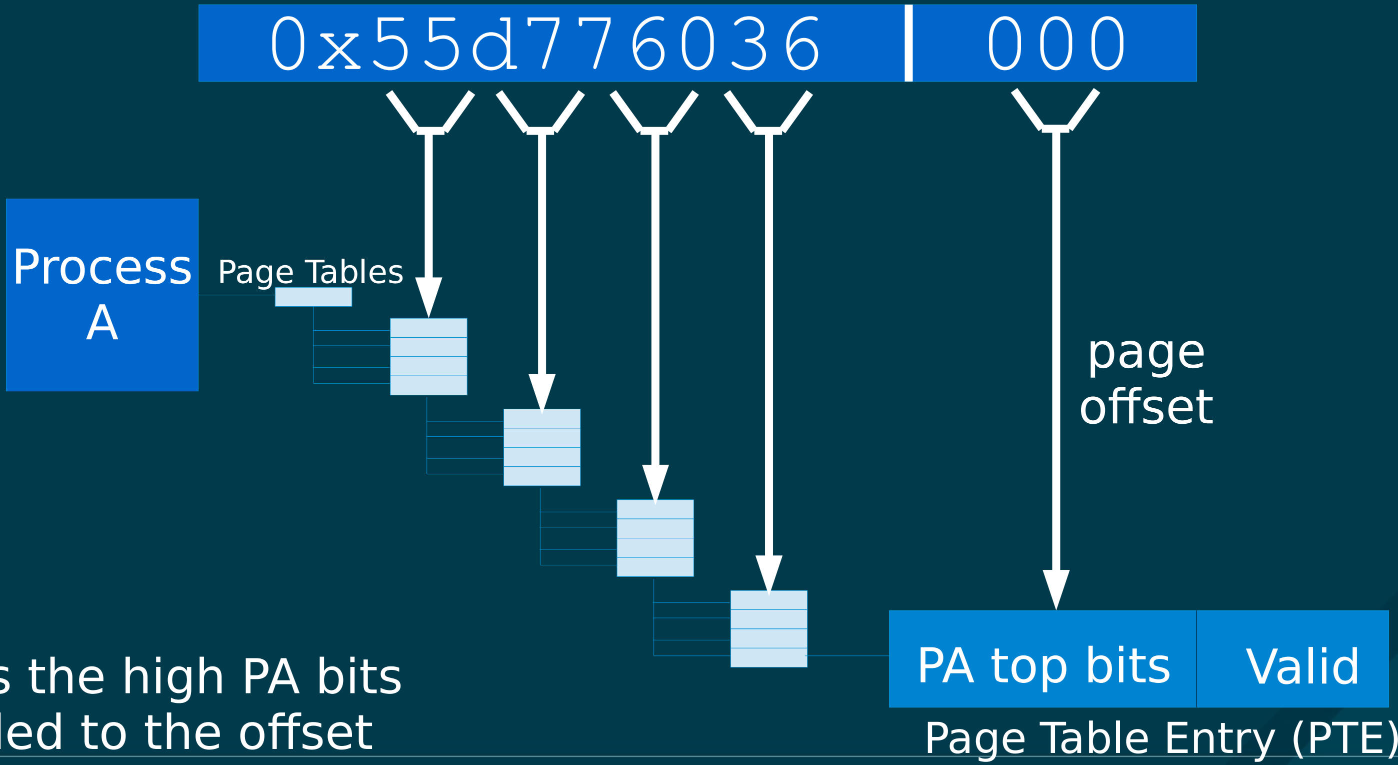
TLB is just another form of cache
(but not for program data)

Translation Lookaside Buffer (TLB)



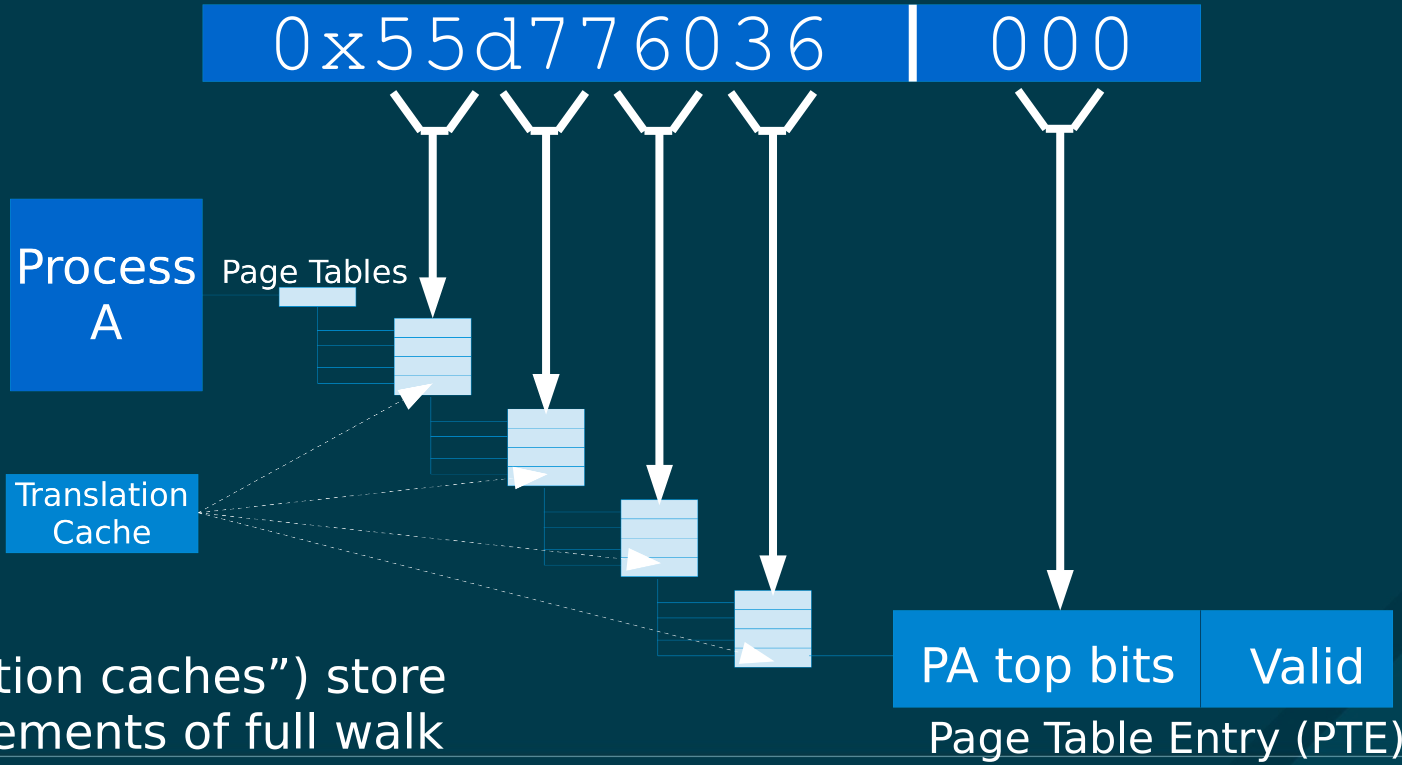
0x4000	→	0x0000
0x3000	→	0x6000
0x1000	→	0x4000
0x0000	→	0x7000

Virtual memory (VA to PA translation)



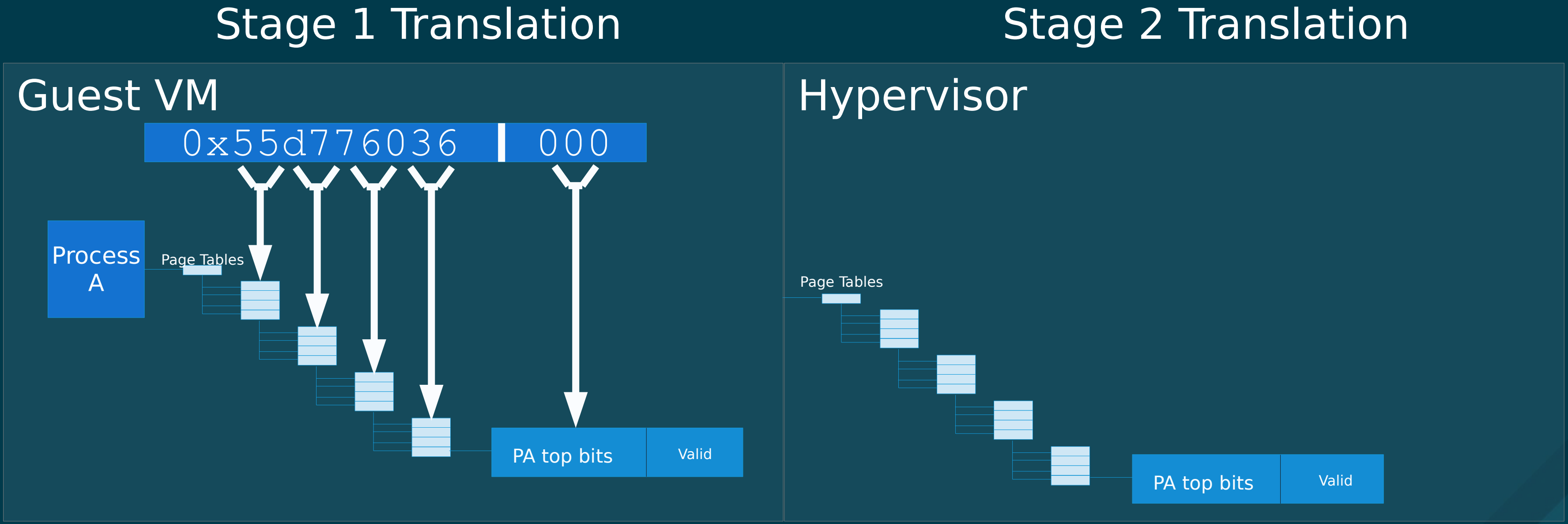
The PTE contains the high PA bits that are prepended to the offset

Virtual memory (VA to PA translation)



Walker (“translation caches”) store recently used elements of full walk

Virtual memory and Hypervisors (Second Level Address Translation)



Virtual memory (summary)

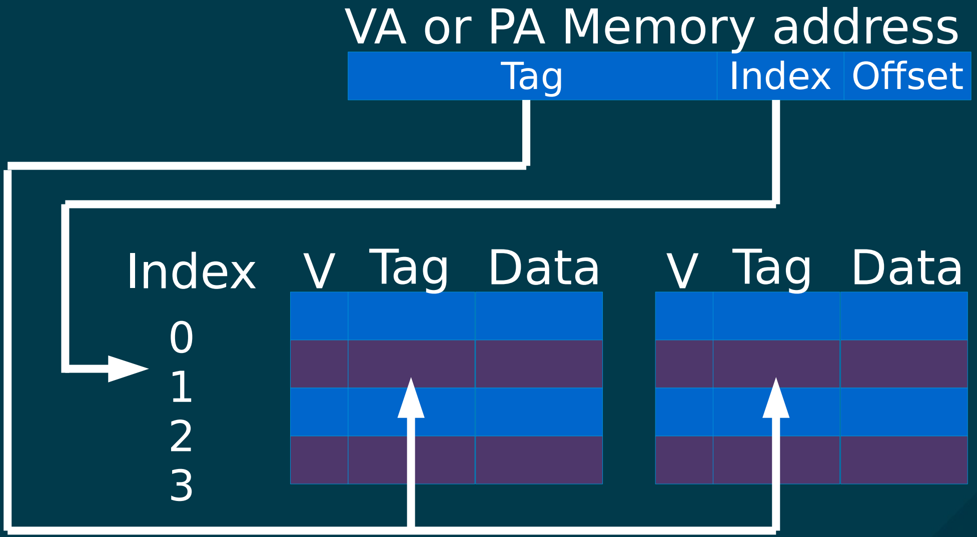
- **Accesses translated (possibly multiple times) before reaching memory**
  - Applications use virtual addresses (VAs) that are managed at page-sized granularity
  - A VA may be mapped to an intermediate address if a Hypervisor is in use
  - Either the Hypervisor or Operating System kernel manages physical translations
- **Translations use hardware-assisted page table “walkers” (traverse tables)**
  - The Operating System creates and manages the page tables for each application
  - Hardware manages TLBs (Translation Lookaside Buffers) filled with recent translations
  - TLBs cache multiple levels including stage 1 and stage 2 translations per VM instance
- The collection of currently valid addresses is known as a (virtual) address space
  - On “context switch” from one process to another, page table base pointers are swapped, and existing TLB entries are invalidated. Cache flushing may be required depending upon the use of address space IDs (ASIDs, PCIDs, etc.) in the architecture and the OS

Virtual memory (summary)

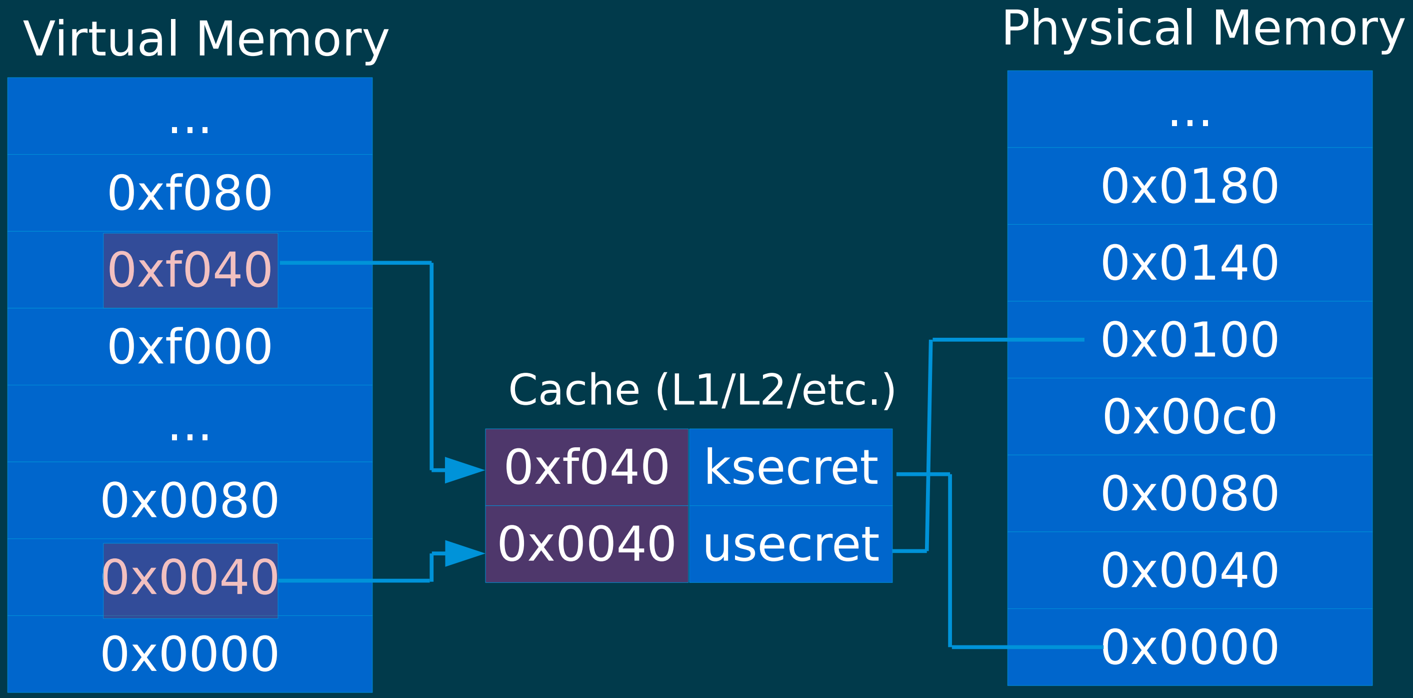
- **Applications have a large flat Virtual Address space mostly to themselves**
  - Text (code) and data are dynamically linked into the virtual address space at application load automatically using metadata from the ELF (Executable Linkng Format) binary
  - Dynamic libraries are mapped into the Virtual Address space and may be shared
- Operating Systems may map some OS kernel data into application address space
  - Limited examples intended for deliberate use by applications (e.g. Linux VDSO)
    - For performance. Data read from Virtual Dynamic Shared Object without system call
  - The rest is explicitly protected by marking it as inaccessible in the application page tables
- **Linux (used to) maps all of physical memory into every running application**
  - Allows for system calls into the OS without performing a full context switch on entry
  - The kernel is linked with high virtual addresses and mapped into every process

Caches

- Caches store “blocks” of memory
  - Copy of same data held in RAM
  - Aligned to “line size” (e.g. 64B)
- Uses physical or virtual addressing
  - L1$ is usually “virtually indexed” and “physically tagged”
- Implementation aka “organization”
  - e.g. “fully associative”
  - e.g. “8-way set associative”
- Cache replacement “policy”
  - e.g. LRU for conflicts (predictable)
- Intel uses VIPT L1D$ (32KB, 8-way)
  - Common parallel optimization

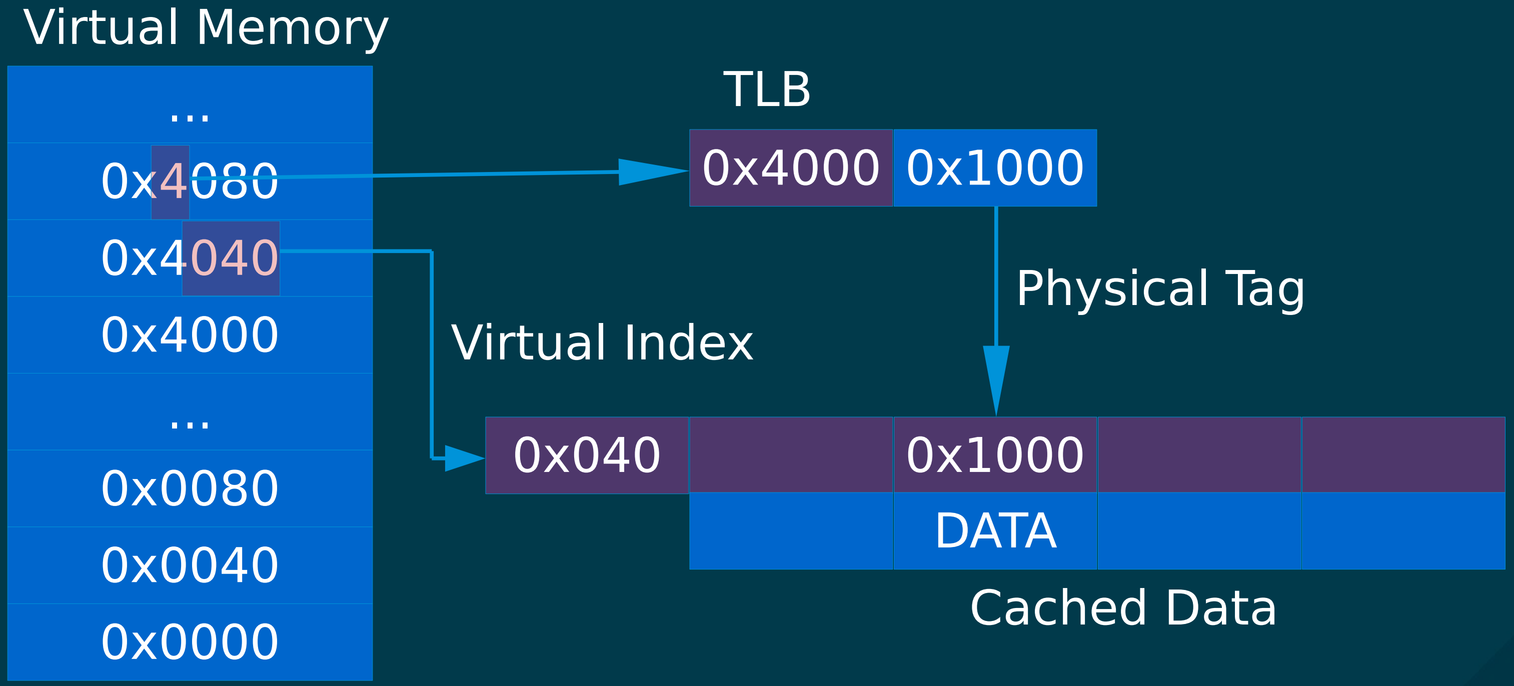


Caches (Virtually Indexed L1 D$)



* For readability privileged kernel addresses are shortened to begin 0xf instead of 0xffffffff...

Caches (VIPT parallel lookup mechanism)



A common L1 cache optimization – split Index and Tag lookup (for TLB lookup)

Caches (summary)

- Caches exist because principal of locality says recently used data likely used again
- Unfortunately we have a choice between “small and fast” and “large and slow”
  - Levels of cache provide the best of both, replacement policies handle cache eviction
- Caches are organized into sets where each set can contain multiple cache lines
  - A typical cache line is 64 or 128 bytes and represents a block of memory
  - A typical memory block will map to single cache set, but can be in any “way” of a set
- Caches may be direct mapped or (fully) associative depending upon complexity
  - Direct mapped allows one memory location to exist only in a specific cache location
  - Associative caches allow one memory location to map to one of N cache locations

Caches (summary)

- Cache entries are located using a combination of indexes and tags
    - Index and tag pages are formed from a given address address
    - The index locates the set that may contain blocks for an address
    - Each entry of the set is checked using the tag for an address match
- Caches may use virtual or physical memory addresses, or a combination
    - Fully virtual caches can result in homonyms for identical physical addresses
    - Fully physical caches can be much slower as they must use translated addresses
- A common optimization is to use VIPT (Virtually Indexed, Physically Tagged)
    - VIPT caches search index using the low order (page offset, e.g. 12) bits of a VA
        - In Intel x86 this means 6 bits for block, 6 bits for index, rest is the tag
    - Meanwhile the core finds the PA from the MMU/TLB and supplies to tag compare

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.

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

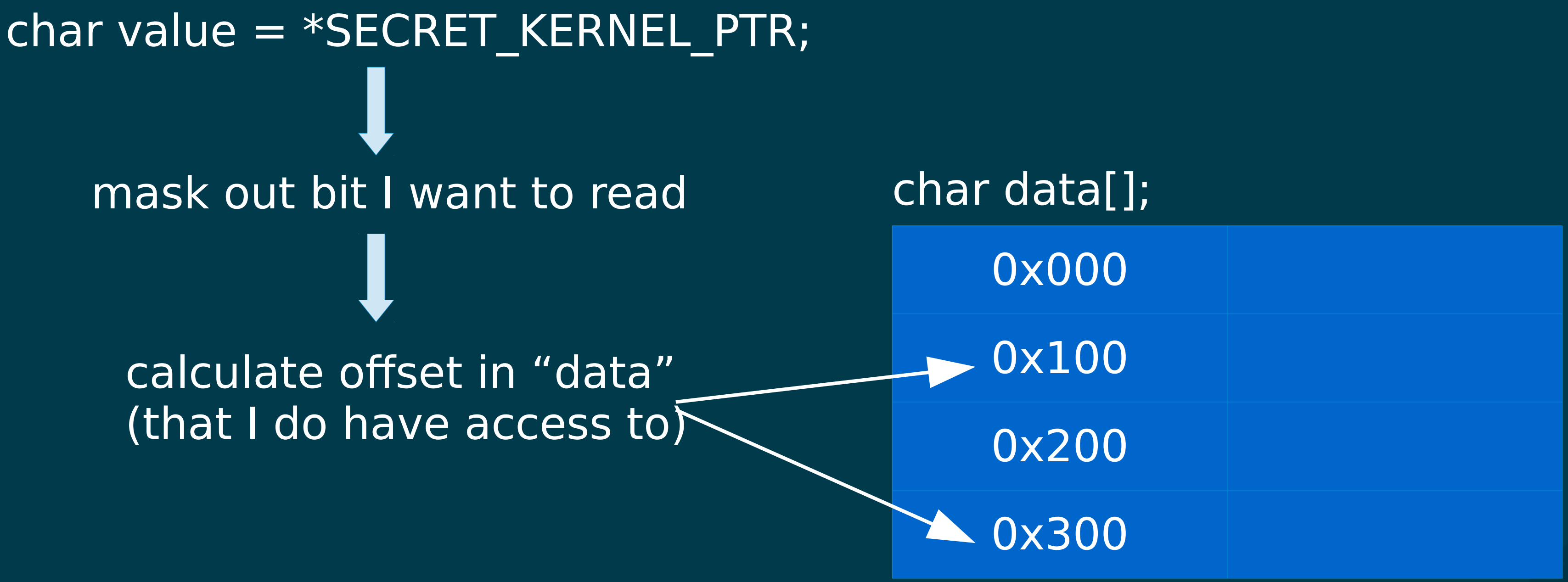
Meltdown (continued)

- 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)

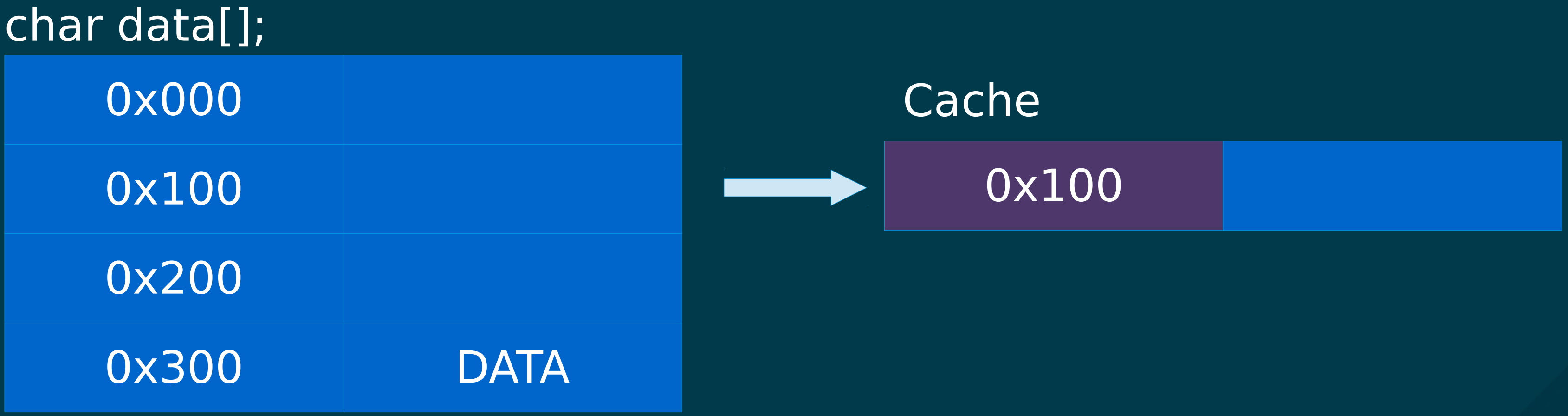


```
char value = *SECRET_KERNEL_PTR;
char data[];
```

mask out bit I want to read
calculate offset in “data” (that I do have access to)
0x000
0x100
0x200
0x300

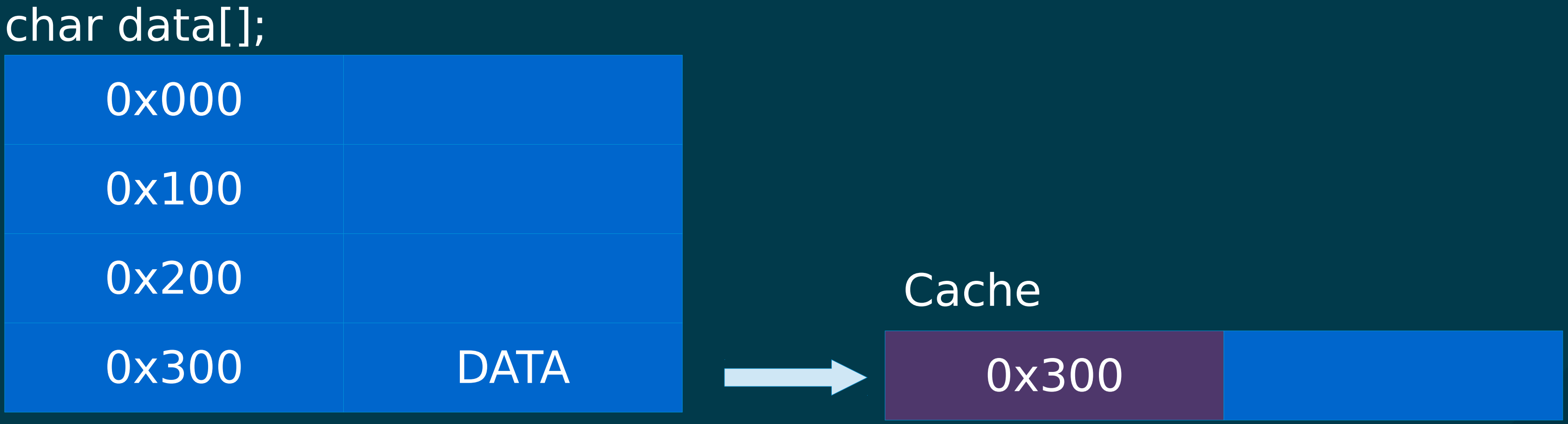
Meltdown (continued)

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



Meltdown (continued)

- Access to “data” element 0x300 pulls the corresponding entry into the 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 (TM)

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

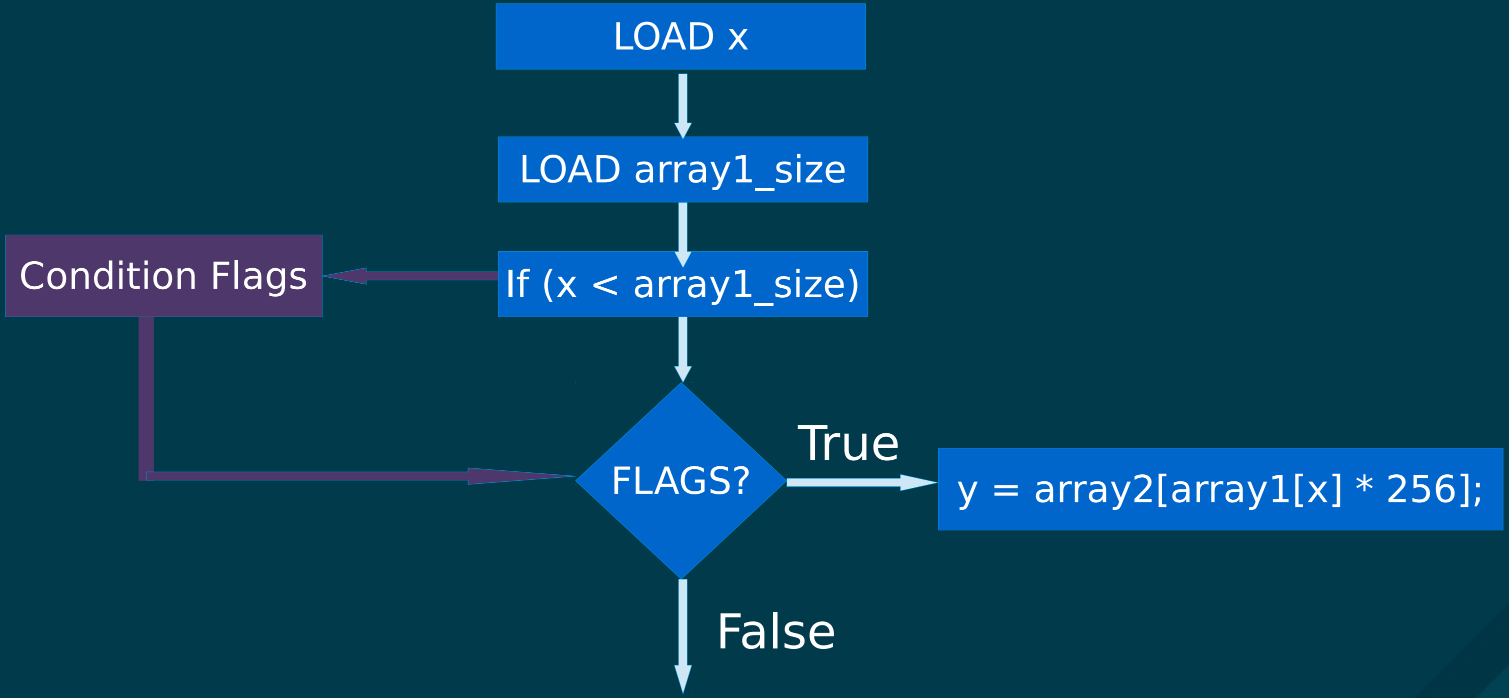
Spectre-v1: Bounds Check Bypass (CVE-2017-2573)

- 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
  - L1D$ 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

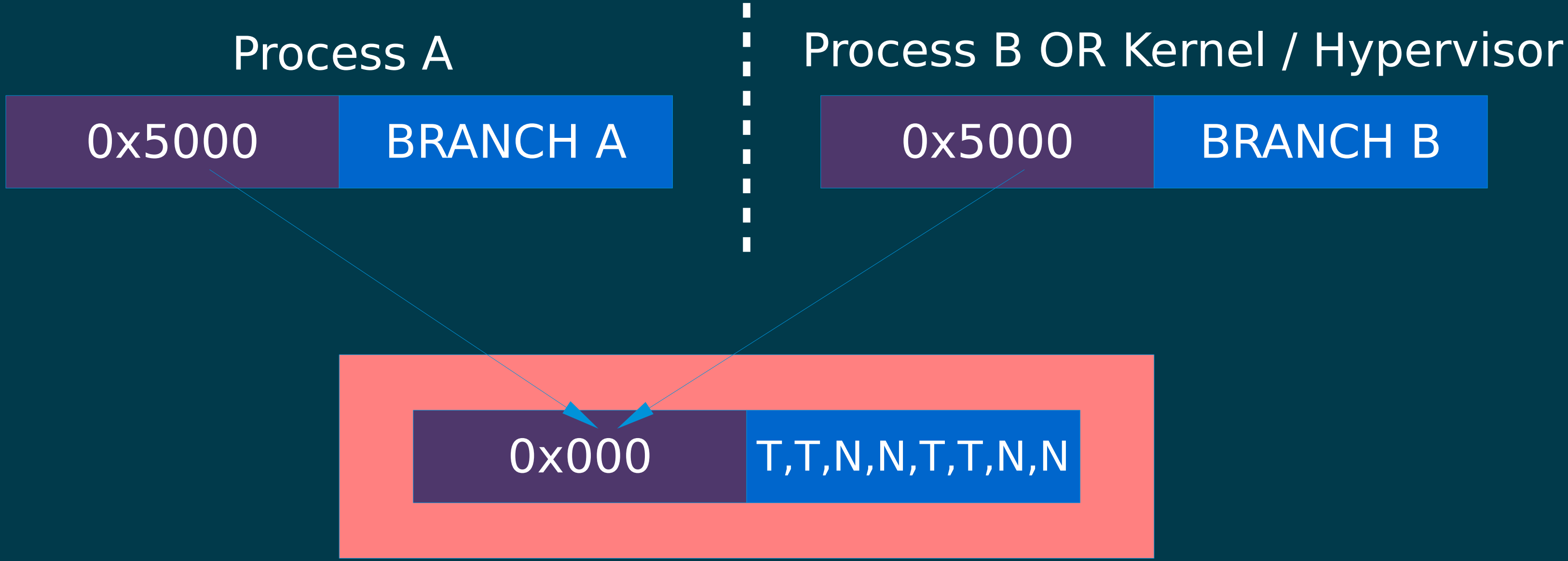
Mitigating Spectre-v1: Bounds Check Bypass (cont.)

- 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

redhat.

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

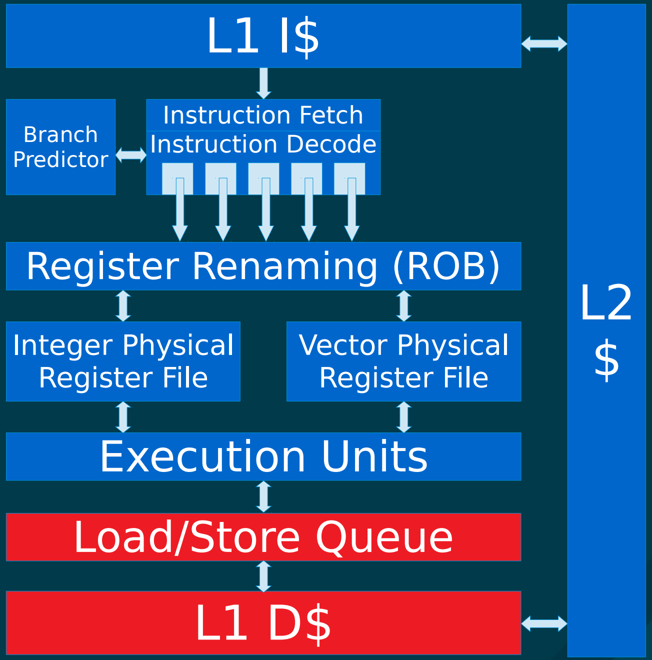
see SpectreRSB,“ret2spec”, and other RSB vulnerabilties

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:

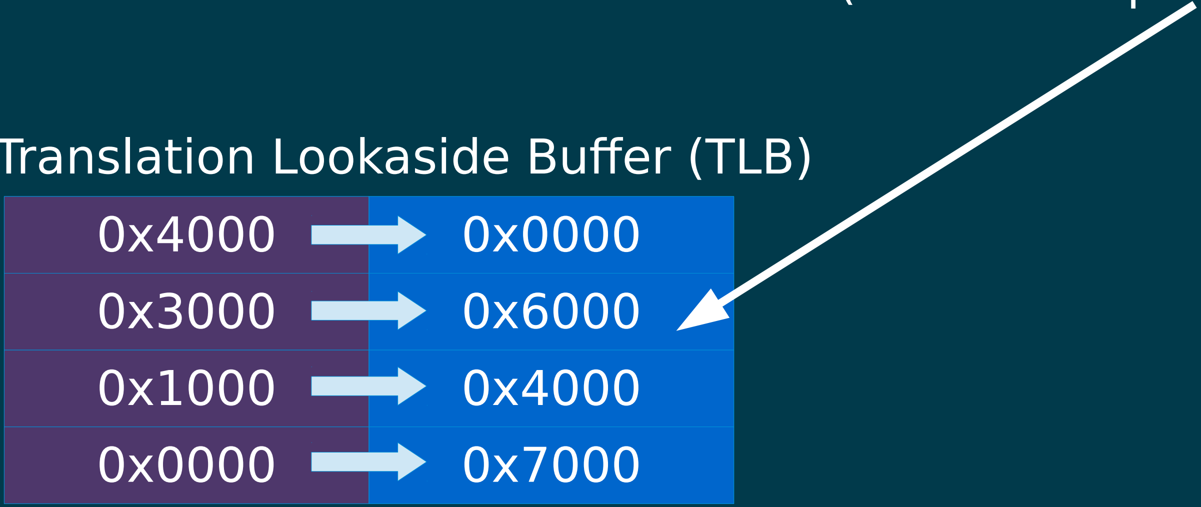
```
if (x < array1_size)
    y = array2[array1[x] * 256];
```

- 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)

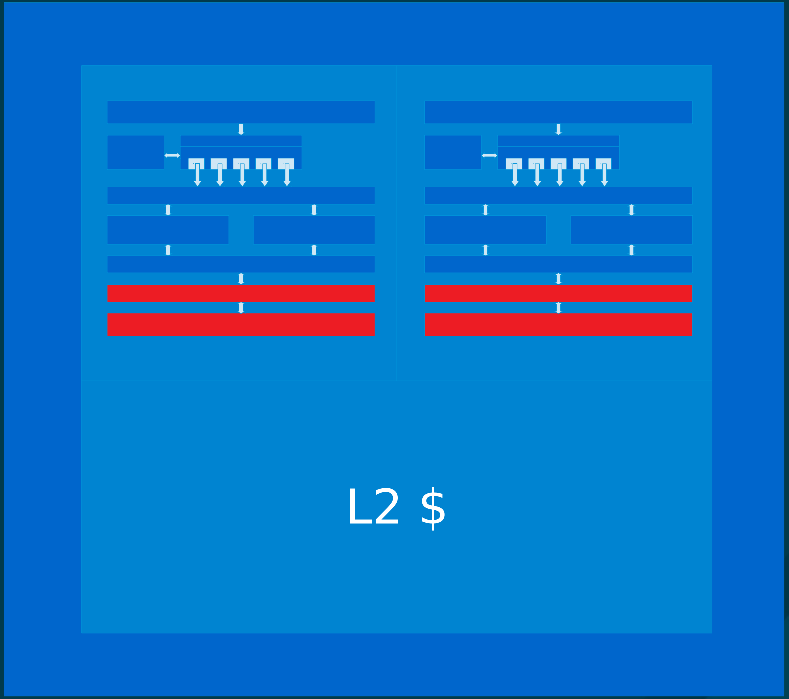
Translation Lookaside Buffer (TLB)



0x4000	0x0000
0x3000	0x6000
0x1000	0x4000
0x0000	0x7000

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

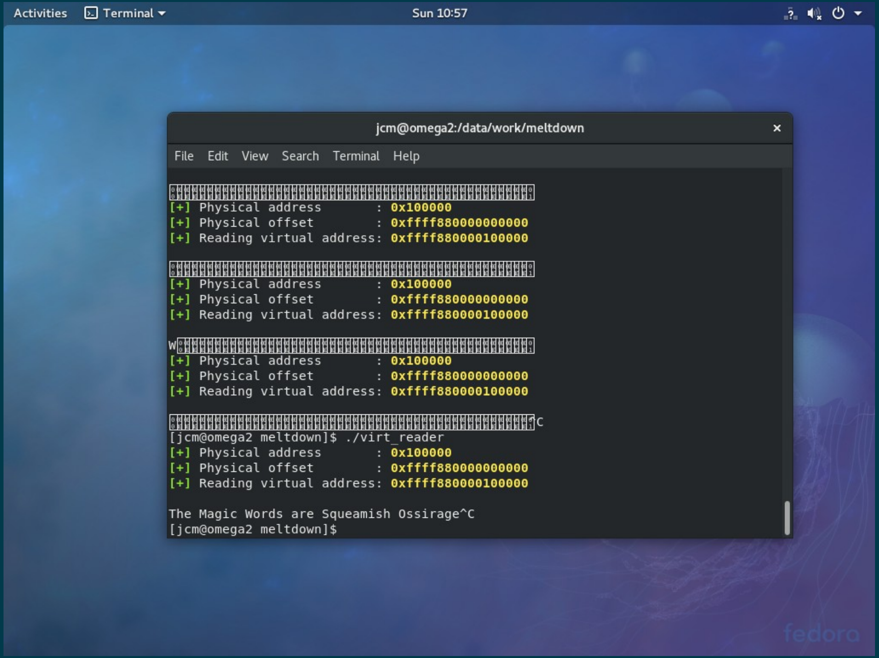
- Spectre attacks can be performed over the network by using two combined gadgets
  - A “leak” gadget sets some flag or state during speculative out of bounds access:

    ```
    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

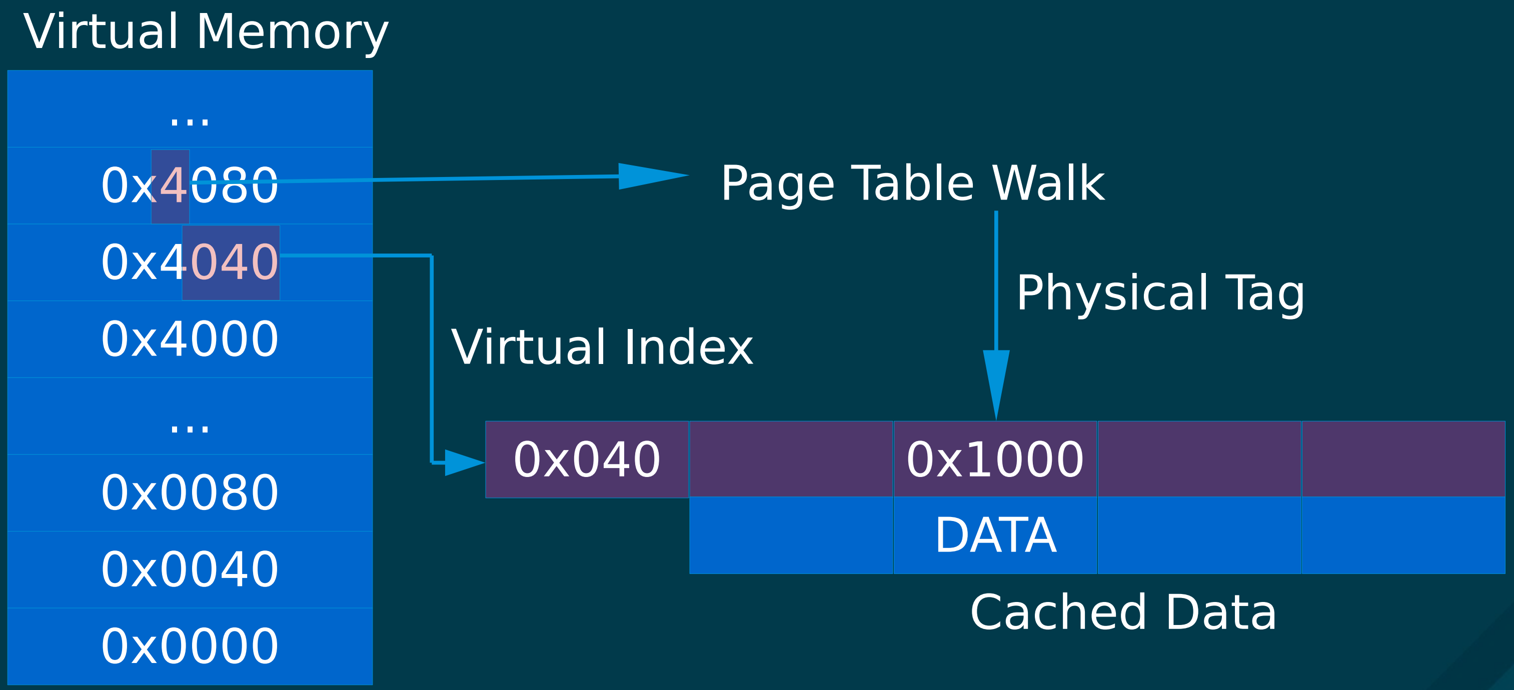
- An attacker trains the leak gadget then extracts data with the transmit gadget
  - Rate limited to bits per hour over the internet, detectable even among noise
  - Acceleration possible using e.g. an AVX2 power-down side-channel (Intel)

L1 Terminal Fault (aka “Foreshadow”)

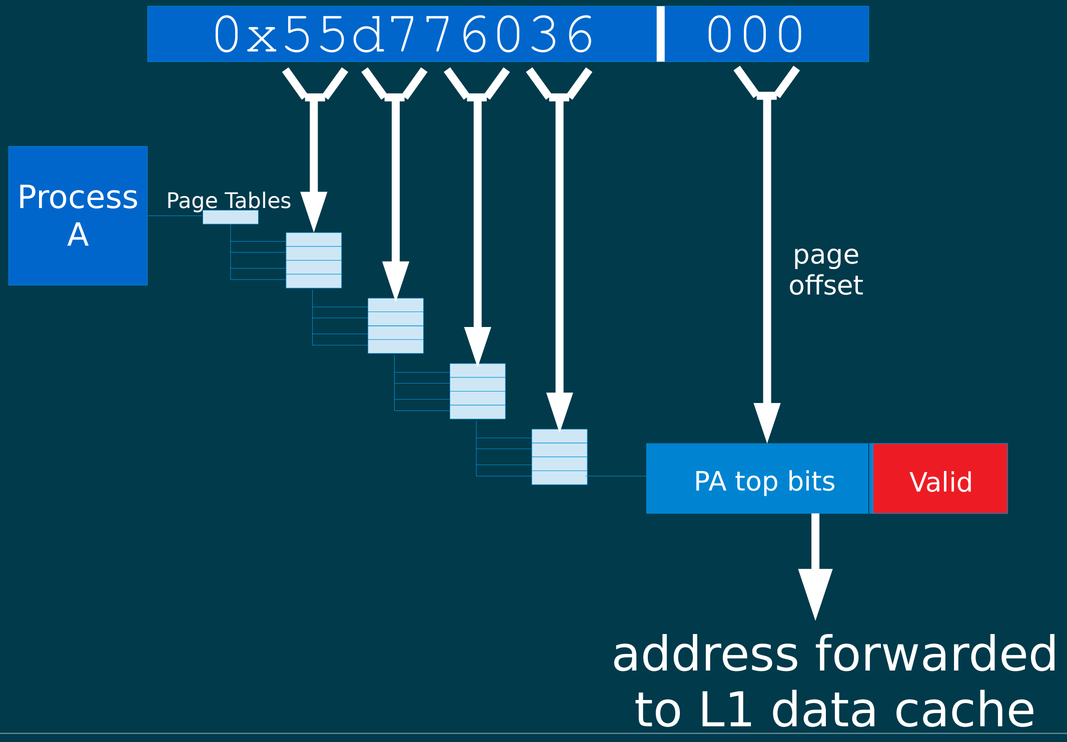


```

[+] Physical address       : 0x100000
[+] Physical offset        : 0xffff880000000000
[+] Reading virtual address: 0xffff880000100000


[+] Physical address       : 0x100000
[+] Physical offset        : 0xffff880000000000
[+] Reading virtual address: 0xffff880000100000

W
[+] Physical address       : 0x100000
[+] Physical offset        : 0xffff880000000000
[+] Reading virtual address: 0xffff880000100000

^C
[jcm@omega2 meltdown]$ ./virt_reader
[+] Physical address       : 0x100000
[+] Physical offset        : 0xffff880000000000
[+] Reading virtual address: 0xffff880000100000

The Magic Words are Squeamish Ossirage^C
[jcm@omega2 meltdown]$
```

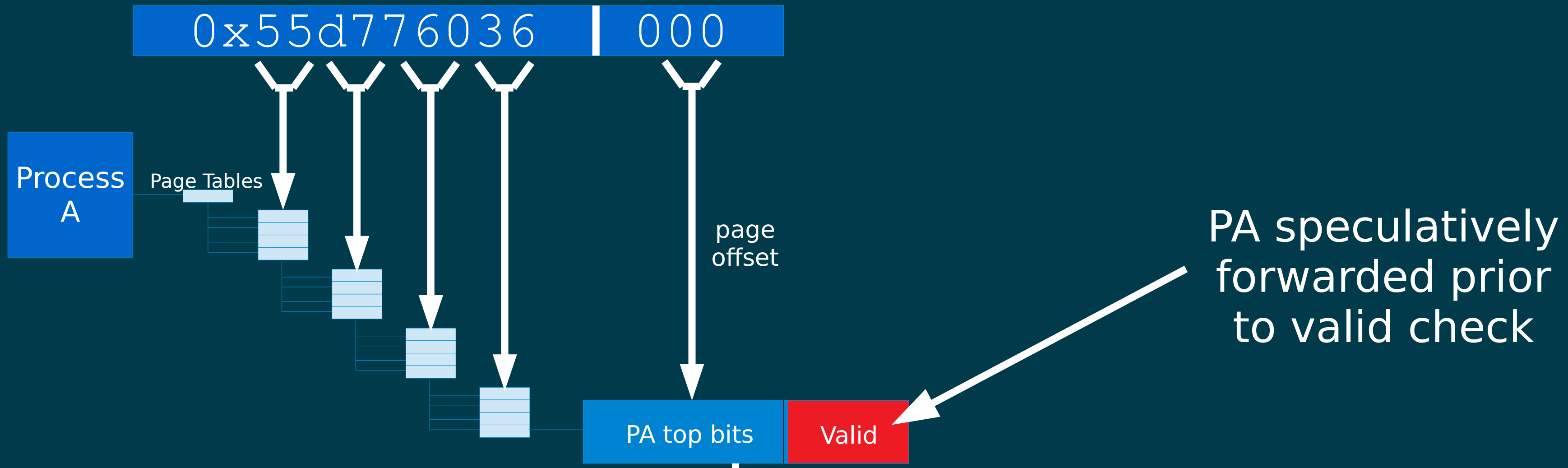
L1 Terminal Fault (aka “Foreshadow”)



L1 Terminal Fault (aka “Foreshadow”)

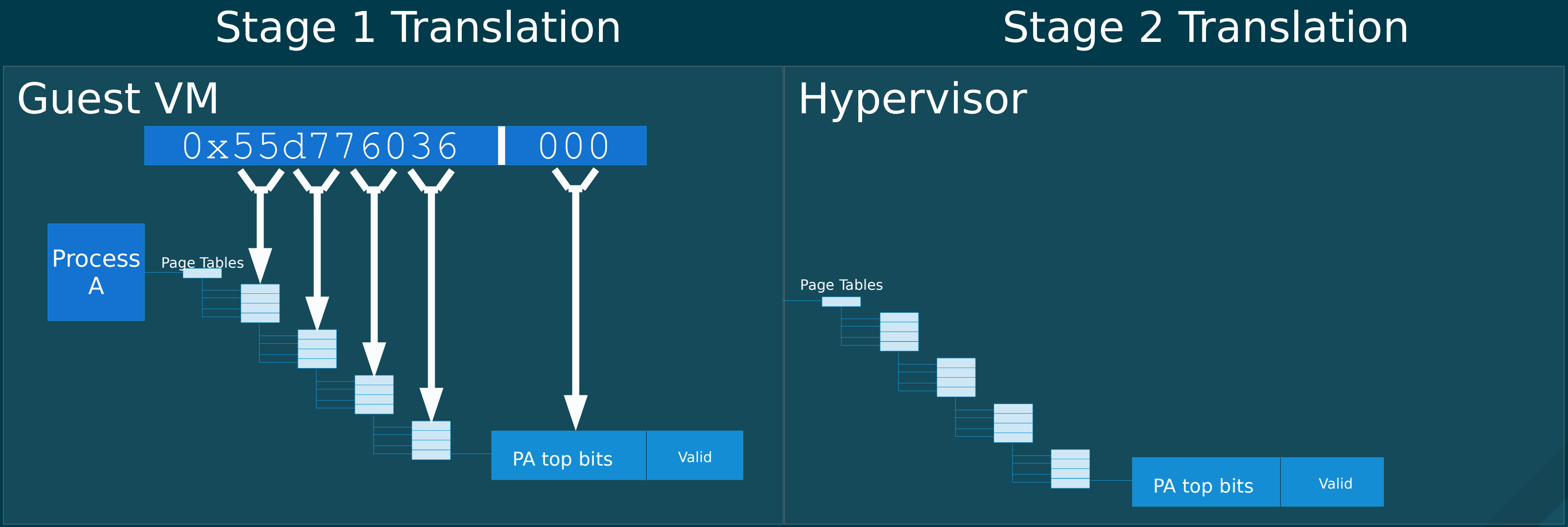


L1 Terminal Fault (aka “Foreshadow”)



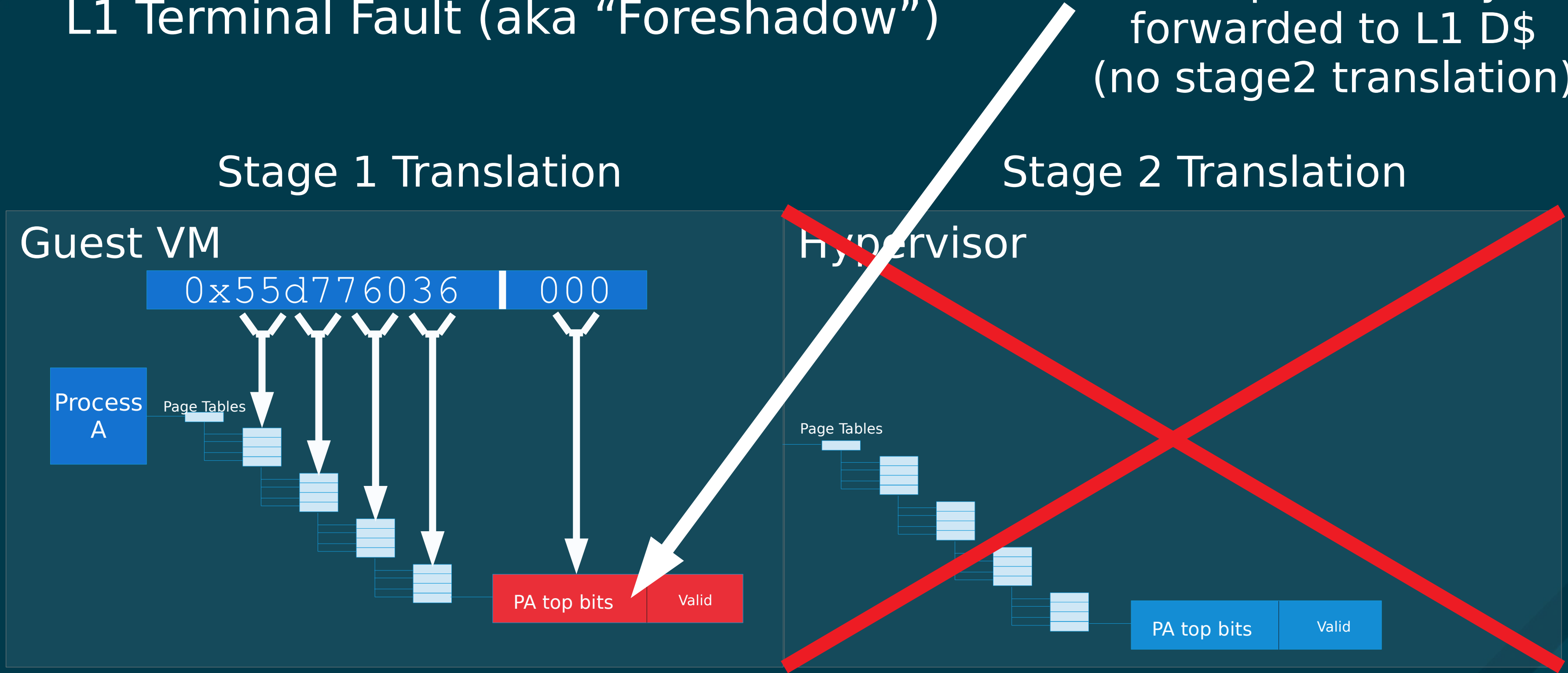
address forwarded to L1 data cache

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?

redhat.

THANK YOU

plus.google.com/+RedHat

facebook.com/redhatinc

linkedin.com/company/red-hat

twitter.com/RedHatNews

youtube.com/user/RedHatVideos

redhat®