

CSE 127 Computer Security

Stefan Savage, Spring 2018, Lecture 12

System Security II:
Process Isolation, Cache Side Channels

Process Isolation

- Process boundary is a trust boundary
 - Any inter-process interface is part of the attack surface
- How are individual processes isolated from each other?
 - Which mechanism ensures that one process cannot affect another, without proper authorization?

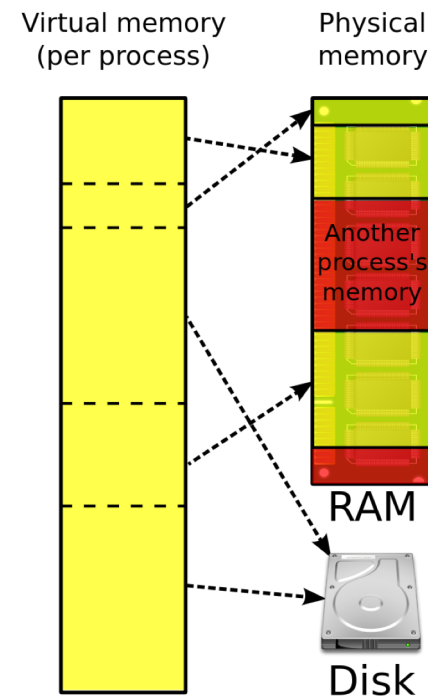
Virtual Memory

- Each process gets its own ***virtual address space***, managed by the operating system
 - A “personalized” view of the entire addressable memory space
 - As if this is the only process on the system
- Primary security mechanism for isolating processes from each other



Virtual Memory

- Memory addresses used by processes are ***virtual addresses***
- Virtual addresses are mapped by the operating system into ***physical addresses***, corresponding to actual storage locations
- ***Address translation*** is the mechanism for mapping virtual to physical addresses



Address Translation

- Isolation
 - Provides (to a process, an operating system, a peripheral) a virtualized view of memory with limited visibility/access to the underlying memory space
- Memory Access Polymorphism
 - Different access implementations for different memory regions/types
 - Rules for speculative access, out of order access, caching, backing store, etc.

Address Translation

- Using 64-bit ARM architecture as an example...
- How to practically map arbitrary 64bit addresses?
 - 64 bits * 2^{64} (128 exabytes) to store any possible mapping

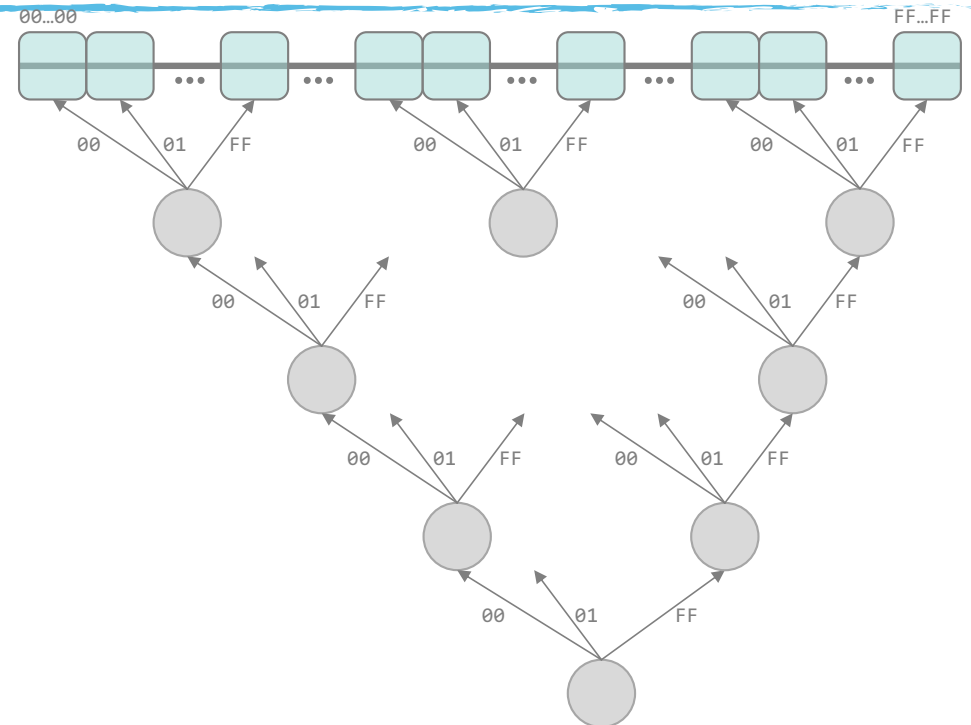
Address Translation

- **Page**

- Basic unit of mapping granularity
- Usually 4KB (or multiple thereof)
- Still $52 \text{ bits} * 2^{52}$ (208 petabytes) to store any possible page mapping

- **Multi-level *Page Table***

- Sparse tree of page mappings
- Use virtual address as path through tree
- Leaf node stores corresponding physical address
- Each process gets its own tree
- Root kept in a dedicated register: Translation Table Base Register



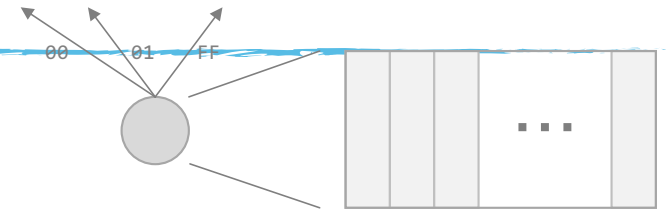
Pages

- *"In our terminology we made a strict distinction between memory units (we called them **"pages"** and had "core pages" and "drum pages") and corresponding information units (for lack of a better word we called them "segments")"*
 - Dijkstra, *The Structure of the "THE"-Multiprogramming System*, 1968



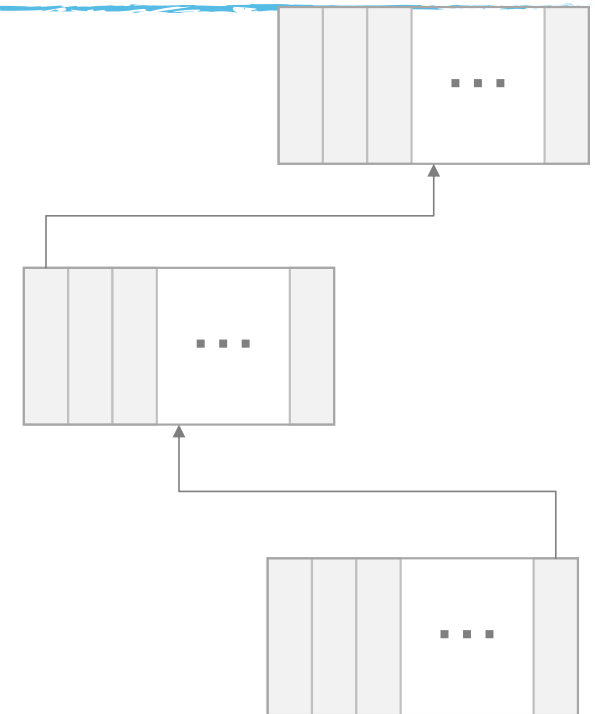
Page Tables

- Data structures used to store address mapping
 - Nodes of the tree
- Each table/node is:
 - Array of translation descriptors
 - Same size as a memory page



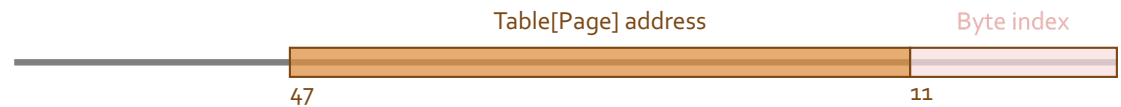
Page Tables

- Data structures used to store address mapping
 - Nodes of the tree
- Each table/node is:
 - Array of translation descriptors
 - Same size as a memory page
- Organized into a tree
 - Iteratively resolve n bits of address at a time
 - Each descriptor is either
 - Page descriptor (leaf node)
 - Table descriptor (internal node)

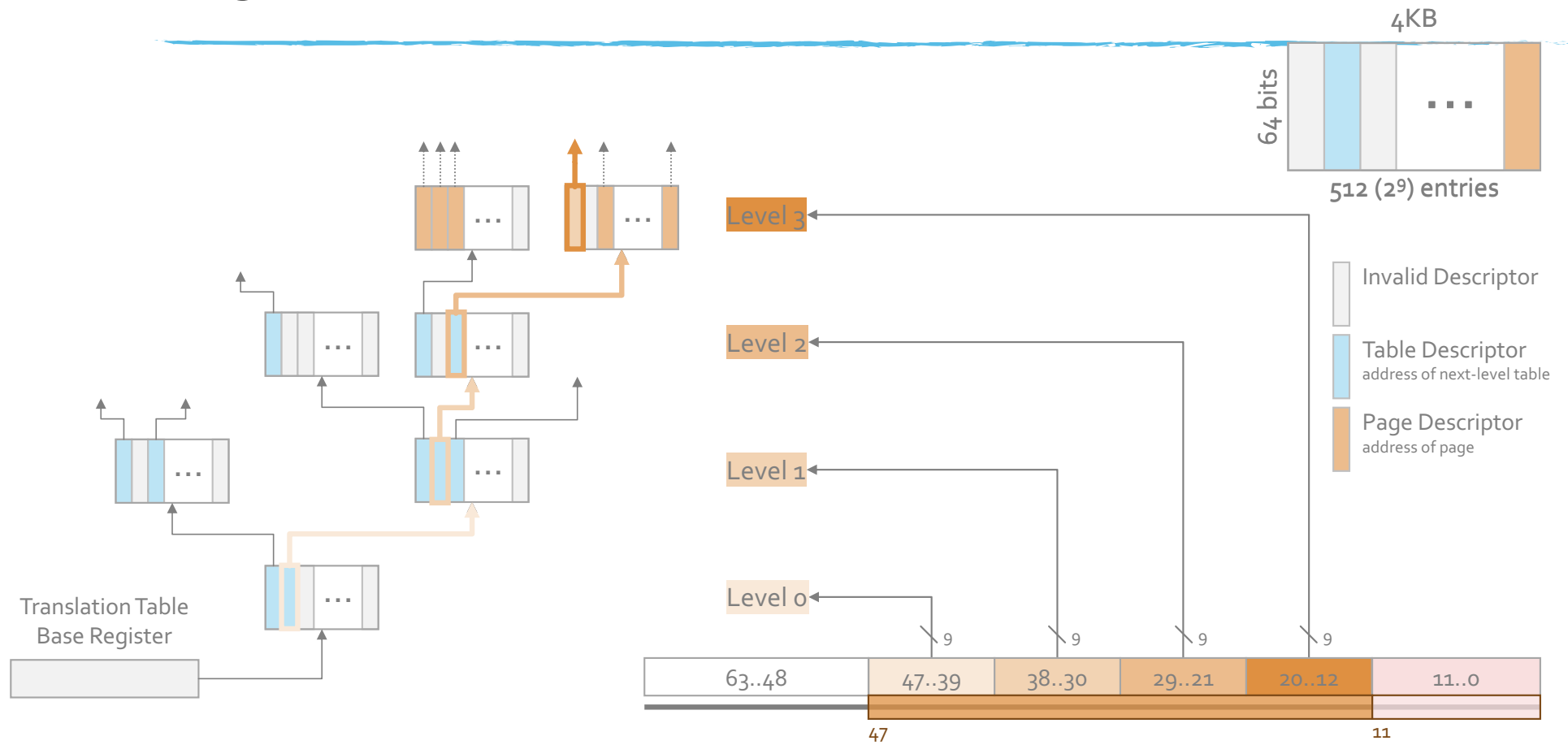


Page Tables

- In reality, the full 64bit address space is not used.
 - Working assumption: 48bit addresses



Page Table Walk



Address Translation

- Every memory access a process performs goes through address translation*
 - Load, store, instruction fetch
 - “complete mediation”
- That’s a very expensive operation to perform several times for each instruction
 - Multiple optimizations in the page table structure (not covered here)
 - Translation Lookaside Buffer (TLB)
- *Assuming the system supports virtual memory. May not be available on low-end embedded systems
 - **May not apply to every cache access. Architecture-dependent.

Translation Lookaside Buffer (TLB)

- Small cache of recently translated page addresses
 - Before translating a referenced address, the processor checks the TLB
 - Typically done in parallel with memory cache lookup
 - Identifies if:
 - Physical page corresponding to virtual page (or that page isn't present)
 - If page mapping allows the mode of access (access control)

Access Control

- Not everything within a processes' virtual address space is equally accessible
- Page (and table) descriptors contain additional access control information
 - Read, Write, eXecute permissions
 - Set by the operating system
- Is access to physical memory protected by an ACL or capabilities?

Hierarchy of Trust

- Another common security pattern is *hierarchy of trust*
- **Root of Trust**: most trusted level
 - Security of all other levels depends on it
- Hierarchy of increasingly less trusted, less privileged levels
- Examples: certificate hierarchy, regular users and superusers

Hierarchy of Trust in Operating Systems

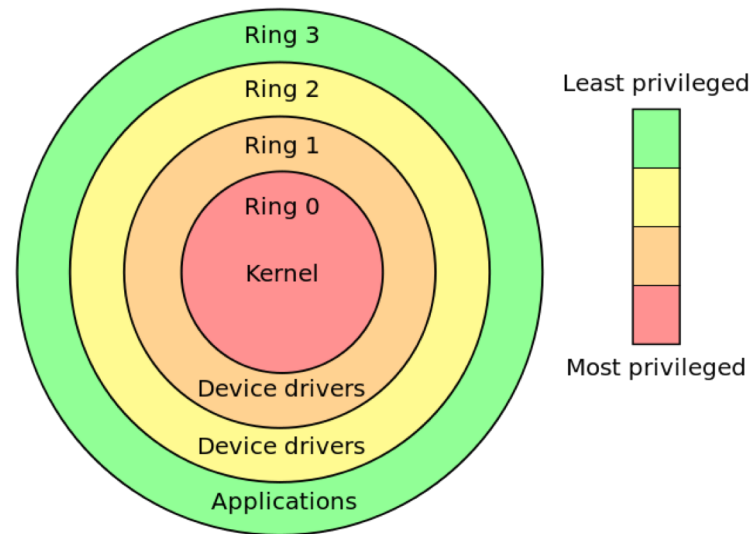
- Kernel vs User
- Hypervisors and Virtual Machines
- ARM TrustZone
- Enclaves (Intel SGX)
 - What if we don't want to trust every layer below

Privilege Levels

- OS trust hierarchy depends on hardware support
- Multiple privilege levels
 - Processor states
 - Typically, just two used by the operating system:
 - Privileged and Non-privileged
 - Kernel Mode and User Mode
 - Supervisor and Normal
- Processor operates at some privilege level
 - Protected system register holds value of current privilege level
- Sensitive system operations require certain minimum privilege level

Intel Privilege Levels

- 4 *rings* (2 used by OS)
 - Ring 0 most privileged
 - “Nothing will ever be more privileged than the kernel”
 - -1, -2, ...



ARM Privilege Levels

- 2 *worlds*
 - Secure and Non-Secure
- 4 *exception levels* (2 used by OS)
 - EL0 least privileged
 - “Nothing will ever be less trusted than a user mode application”
 - Sandboxing, ...

	Non-Secure				Secure	
EL0	App X	App Y	App X'	App Y'	App X''	App Y''
EL1	Guest OS A		Guest OS B		Secure OS	
EL2	Hypervisor					
EL3			Secure Monitor			

Privilege Levels

- Boundary between privilege levels is a trust boundary
 - Any cross-privilege interface is part of the attack surface
- Dedicated mechanisms for safely changing privilege level are needed
 - Anyone can drop privileges, elevating is harder

Privilege Levels

- To enter more privileged state the process:
 - Prepares arguments, including id of the desired entry point and
 - Executes a special instruction that initiates the transfer
- Each privilege level defines a set of entry points for less privileged callers
 - Also, specific registers for passing arguments
 - Typically pointers to more data in less privileged memory
 - These are the only valid entry points when calling from less privileged state
 - The higher-privileged callee is in control of what code is executed
- Details vary by architectures, but core concept is consistent

System Calls

- User mode code may need frequent assistance from kernel
 - I/O operations (files, network, devices, etc.)
 - System information (time, environment, etc.)
 - Process control (fork, signals, mutex, etc.)
- Kernel has its own page table
 - Also maintains page tables for all other processes
- Normally, switching between a usermode process and the kernel would require switching between the respective process' address space and the kernel's
 - This potentially requires flushing TLBs, etc.
- But, system calls need to be fast and efficient

Kernel Mapping

- To make system calls fast, kernel's virtual memory space is mapped into every process, but made inaccessible in usermode
 - This way, system calls are fast, just switch into Kernel mode and go
- Separate permission bits in page tables
 - Unprivileged (usermode): UR, UW, UX
 - Privileged (kernel): PR, PW, PX



Kernel Mapping

- When a process makes a system call and transfers control to the kernel:
 - Kernel's memory space is already mapped
 - Calling process' memory space remains mapped and accessible
- On a process switch, the userland page table is swapped
 - Translation Table Base Register updated
- All processes share the same kernel mapping



Kernel Mapping

- What kind of access should kernel have to usermode memory?



Kernel Security

- Threat model:
 - Confidentiality and integrity of kernel memory and control flow must be protected from compromise by usermode processes
 - All usermode processes are untrusted and potentially malicious
- Operating model:
 - Usermode processes make frequent calls into the kernel, with data passing back and forth
 - Example: network packets, file contents, etc.

Kernel Security

- Kernel must be careful to keep track of whether it is operating on kernel data or usermode data
 - Avoid becoming a ***confused deputy***, i.e. being manipulated into abusing its privileges
- A usermode process may trick the kernel into writing attacker-controlled data into kernel memory or leaking kernel memory to the attacker

Kernel Security

- Separate mechanisms for operating on usermode and kernel data
- Hardware:
 - Special load and store instructions that operate as if in usermode, even when processor is executing in kernel mode.
 - To prevent inadvertently overwriting sensitive data
 - Privileged Access Never (PAN) processor state that prevents kernel mode access to usermode data
 - To prevent inadvertently leaking sensitive data
 - Privileged eXecute Never (PXN) page permission
 - To mark usermode pages non-executable in kernel mode

Kernel Security

- Separate mechanisms for operating on usermode and kernel data
- Software:
 - `copy_to_user()` and `copy_from_user()`
 - Special care needs to be taken with:
 - *Time Of Check vs Time Of Use (TOCTOU)*
 - Nested pointers
 - Uninitialized variables
 - Structure padding

Virtualization

- How to support isolation between the *hypervisor* and *virtual machines*?

	Non-Secure				Secure	
EL0	App X	App Y	App X'	App Y'	App X''	App Y''
EL1	Guest OS A		Guest OS B		Secure OS	
EL2	Hypervisor					
EL3			Secure Monitor			

Virtualization

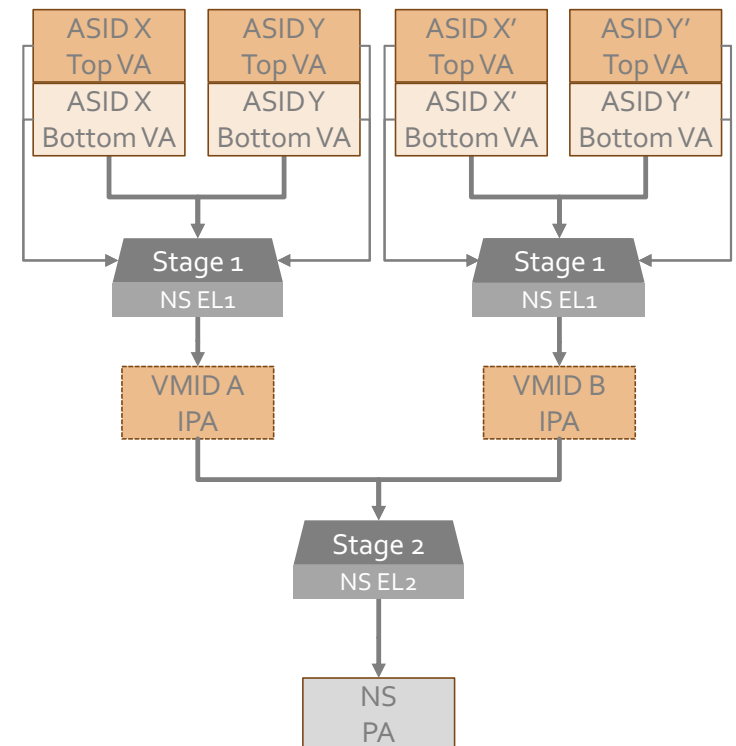
- Multiple stages of address translation to support virtualization
 - Nested



Address Translation Security

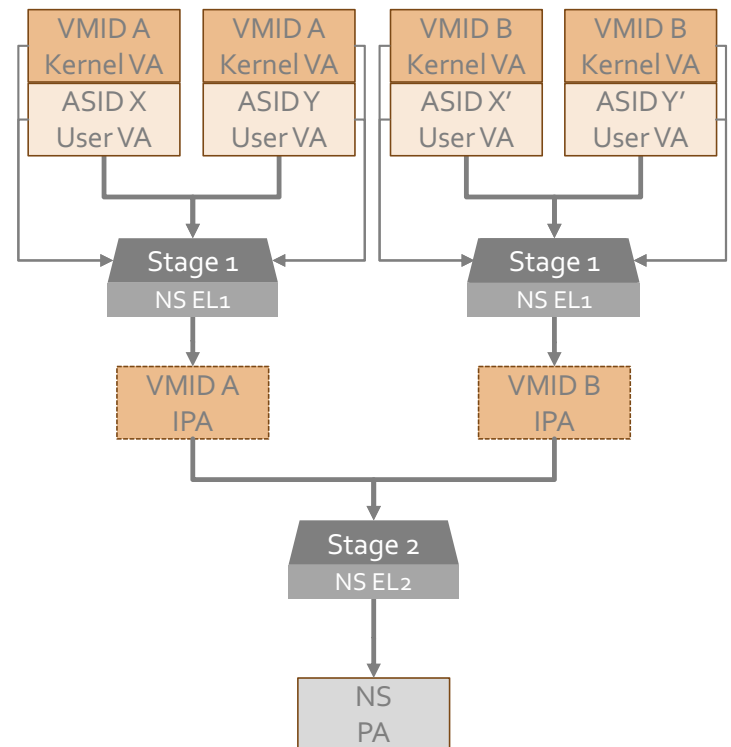
- Non-Secure EL1&0 Translation Regime
- Stage 1 translation
 - Managed by Non-secure EL1
 - Support for dual VA ranges
- Stage 2 translation available
 - Managed by EL2
- Access only to Non-Secure Physical Address Space

	Non-Secure				Secure	
EL0	App X	App Y	App X'	App Y'	App X*	App Y*
EL1	Guest OS A		Guest OS B		Secure OS	
EL2	Hypervisor					
EL3			Secure Monitor			



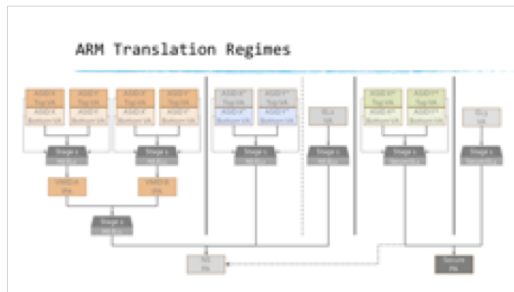
Address Translation Security

- Non-Secure EL1&0 Translation Regime
- Each process has its own virtual address space
 - By convention
 - Top stage-1 range for kernel (global)
 - Bottom stage-1 range for user
 - Actual privileged/unprivileged access managed at individual page level
 - Application Space Identifier (ASID)
 - managed by the guest OS (8 or 16 bits)
 - Virtual Machine Identifier (VMID)
 - managed by the hypervisor (8 or 16 bits)
- Translation Table Base Register updated on process switch



Virtual Memory Security

- Details vary a lot between processor architectures and operating system kernels
 - Even within an architectural family, details may vary a lot between specific processors
 - Even within an operating system, details may vary a lot between specific kernel versions

[illegible]

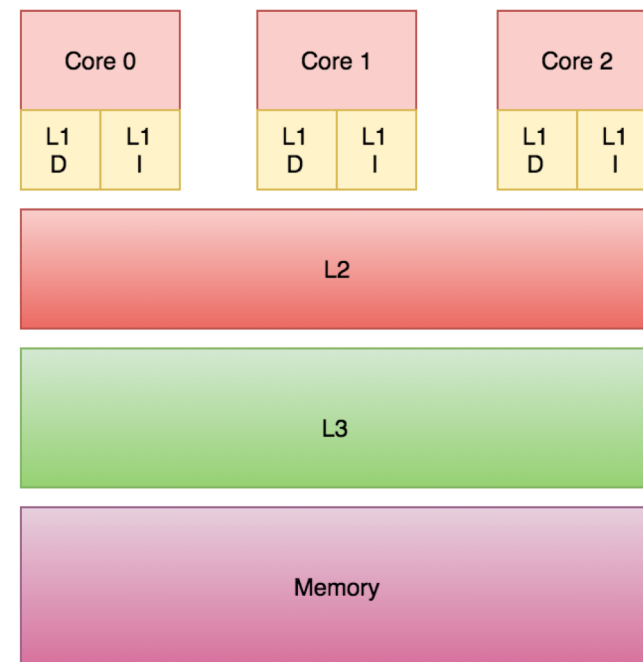
Cache Side Channels

Cache

- Main memory is dense (high capacity) ... but slow
- Processors try to “**cache**” recently used memory in faster, but smaller capacity, memory cells closer to the actual processing core

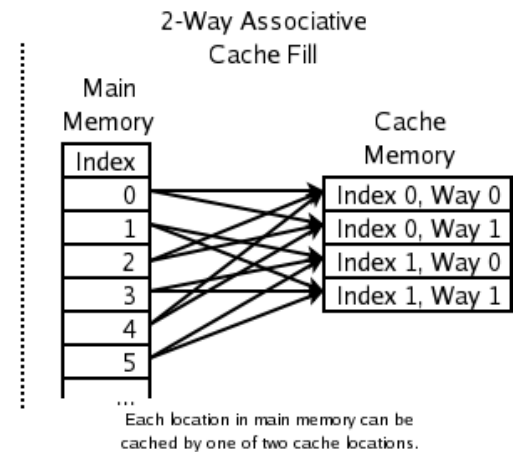
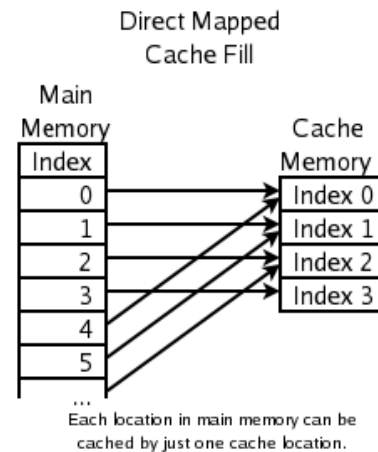
Cache Hierarchy

- Caches are such a great idea, let's have caches for caches!
- Level closest to the processing core is fastest but smallest capacity
- Successive levels have increasing capacity, but slower speeds



Cache Organization

- Cache line
 - Unit of cache granularity
 - Example: 64 bytes
- Set Associativity
 - Cache lines grouped into sets
 - Each memory address is mapped to a set of cache lines
 - Ex: $\text{address} \% \text{number_of_sets}$



Cache Side Channels

- Cache is a shared system resource
 - Not isolated by process, VM, or privilege level
 - “Just a performance optimization”

Cache Side Channels

- Threat model:
 - Attacker and victim are two different execution domains (processes or privilege levels) on the same physical system
 - Attacker is able to invoke (directly or indirectly) functionality exposed by the victim
 - Potentially, with attacker-supplied parameters
 - Example: using API, making a system call, running a setuid binary, etc.
 - Attacker should not be able to infer anything about the contents of victim memory
- Many algorithms, including cryptographic algorithms, have memory access patterns that are dependent on sensitive memory contents
 - Therefore memory access patterns may leak memory contents

Cache Side Channels

- Two basic mechanisms for mounting cache side channel attacks
- Mechanism 1. Attacker:
 - Arranges to put the cache into a known state
 - Lets victim run
 - Checks to see what changed: what addresses have appeared or disappeared from cache
- Mechanism 2. Attacker:
 - Times normal victim operation (baseline)
 - Makes controlled changes to cache contents: evicting or fetching specific addresses
 - Times victim operation again
- How to tell if something is in cache?
 - Time access to it

Evict & Time

- Run the victim code several times and time it
- Evict (portions of) the cache
 - How?
 - Access many memory locations so that previous cache contents are replaced with this recently-accessed data
- Run the victim code again and time it
- If it is slower than before, then cache lines evicted by the attacker must've been used by the victim
 - We now know something about the addresses accessed by victim code

Prime & Probe

- Prime the cache
 - Access many memory locations so that previous cache contents are replaced with this recently-accessed data
- Let victim code run
- Time access to different memory locations to see which ones got slower
 - Must have been evicted by victim code.
 - We now know something about the addresses accessed by victim code

Flush & Reload

- Only for shared memory
- Flush the cache
- Let victim code run
- Time access to specific memory locations to see which ones got faster
 - Must have been used by victim code.
 - We now know something about the addresses accessed by victim code

Cache Side Channels

- Ok, if I suspend all disbelief I may be able to imagine how something like this is possible in theory, but how practical is it really?
- *"Our error-correcting and error-handling high-throughput covert channel can sustain transmission rates of more than 45 KBps on Amazon EC2, which is 3 orders of magnitude higher than previous covert channels demonstrated on Amazon EC2. Our robust and error-free channel even allows us to build an SSH connection between two virtual machines, where all existing covert channels fail."*
 - *Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud*
 - by Clementine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Kay Romer, Stefan Mangard

Homework

- For 5/15:
 - Read Chapter 12 from *The Craft of System Security*
- No class on 5/17

Next Lecture...

Web Security I