The Devastation of Meltdown

J.Nider

2021-11

Background

Virtual Memory

OoO Execution

Linux Memory Management

The Exploit

The Fix

The Damage

Background

- Meltdown allows an unprivileged app to read ALL memory of a victim machine
- Official name: CVE-2017-5754 “Rogue Data Cache Load” (RDCL)
- Caused by a race condition in out-of-order CPU’s
- NSA potentially knew about this since 1995

Scope

Scope

AMD

Scope

Background

Virtual Memory

OoO Execution

Linux Memory Management

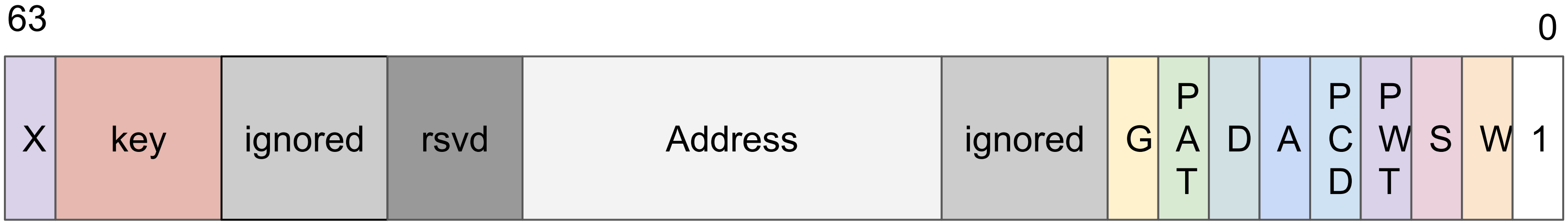
The Exploit

The Fix

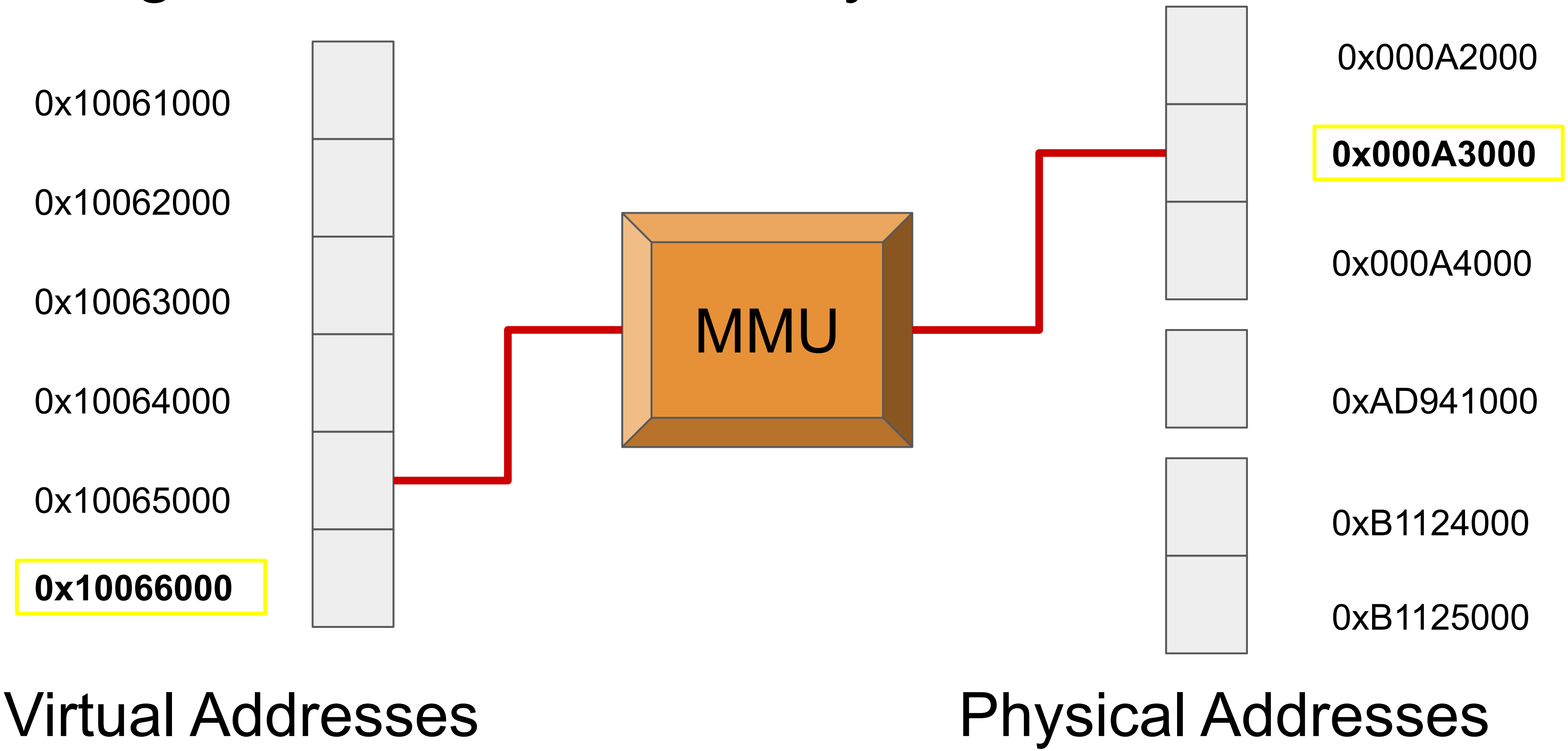
The Damage

Virtual Memory

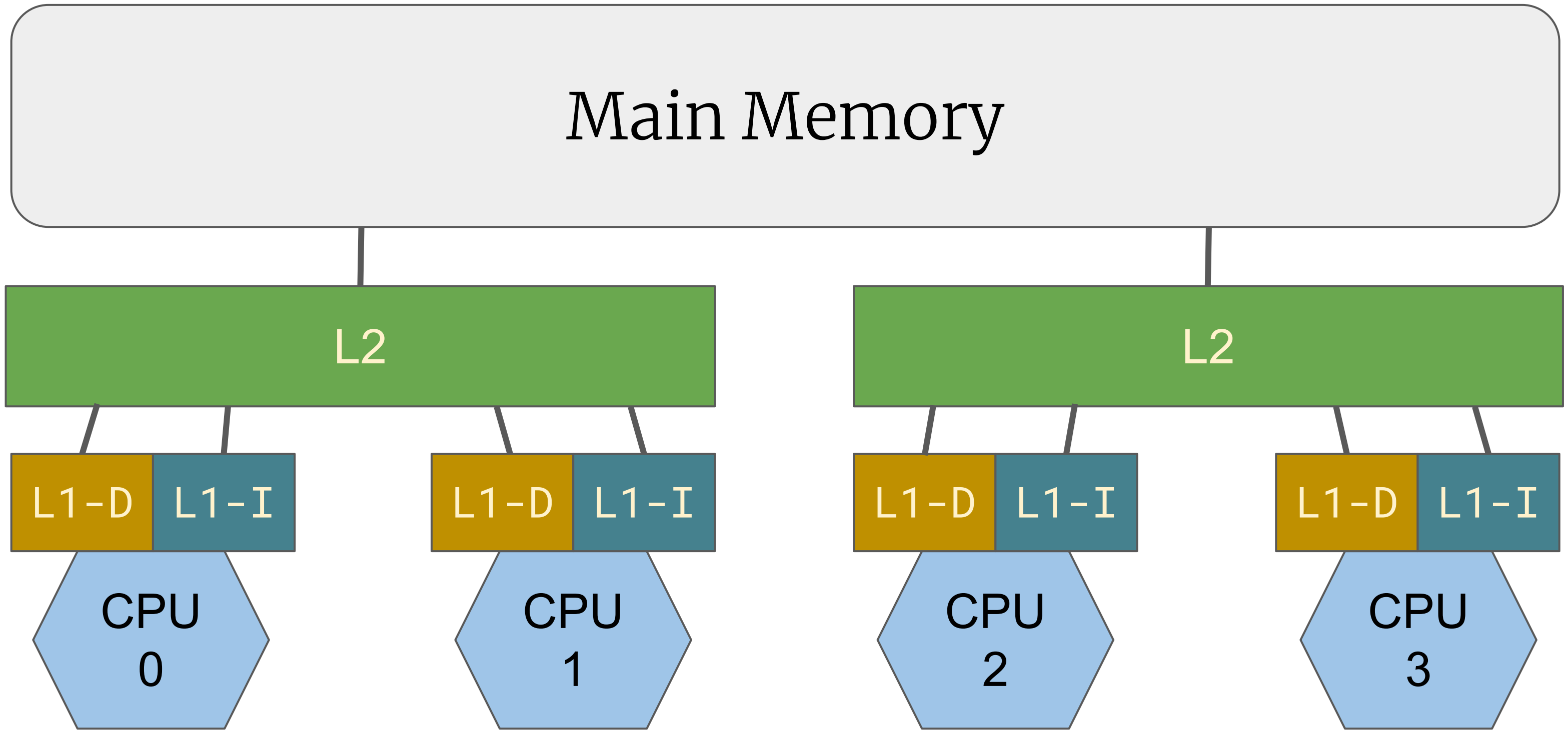
- Memory is organized into pages
  - Page sizes range from 4KB to up 1GB
- Virtual addresses are mapped to physical addresses
  - Usually through page tables, but there are other mechanisms (such as hashing)
- Each page has attributes
  - Describes permissions (WX), supervisor (S) and caching ©
- MMU performs virtual to physical translations
- Translations are cached in TLB (translation lookaside buffer)



Background: Virtual Memory



Cache Organization



Cache Organization

- Reading from main memory is slow!
  - In the range of 400-800ns
  - Therefore we want to avoid main memory as much as possible
- So we cache (make a copy) of any data in a smaller, faster memory
  - Much faster - in the range of 10-100ns
  - Faster memories are more expensive
- We can make a hierarchy with different attributes
  - Capacity
  - Access time
  - Mapping (Direct, Associativity)
  - Multiple ways
- **Cache is not part of the Instruction Set Architecture!**
  - **It is part of the microarchitecture**

Background

Virtual Memory

OoO Execution

Linux Memory Management

The Exploit

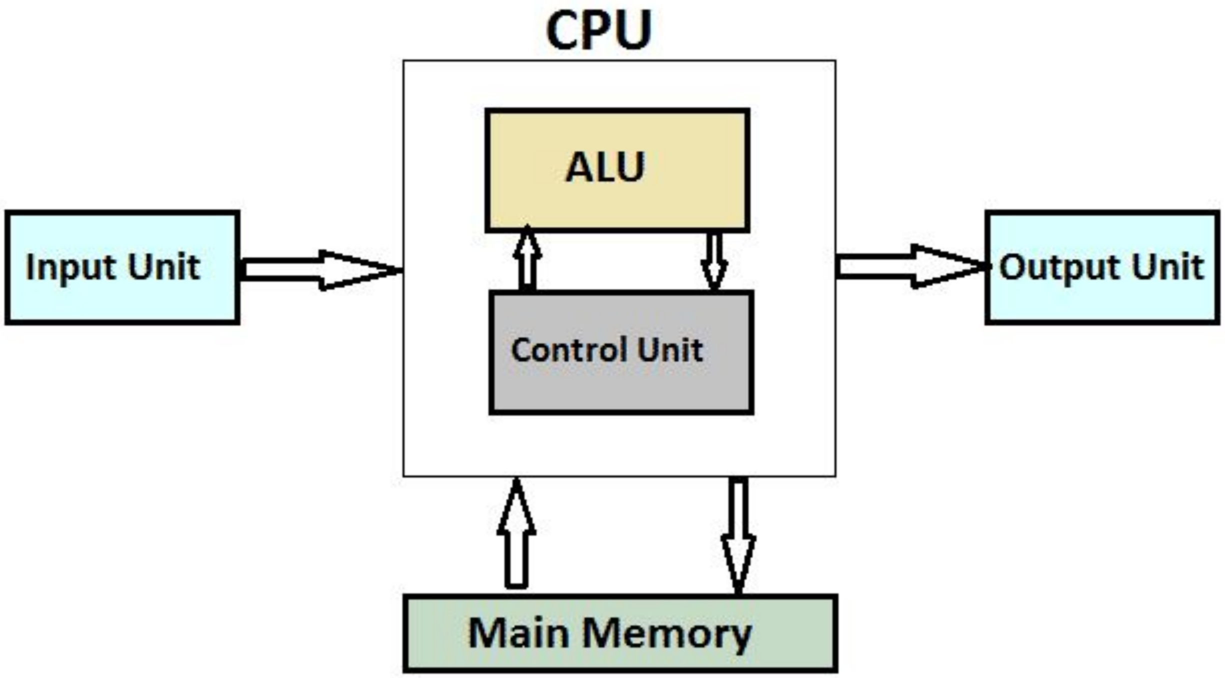
The Fix

The Damage

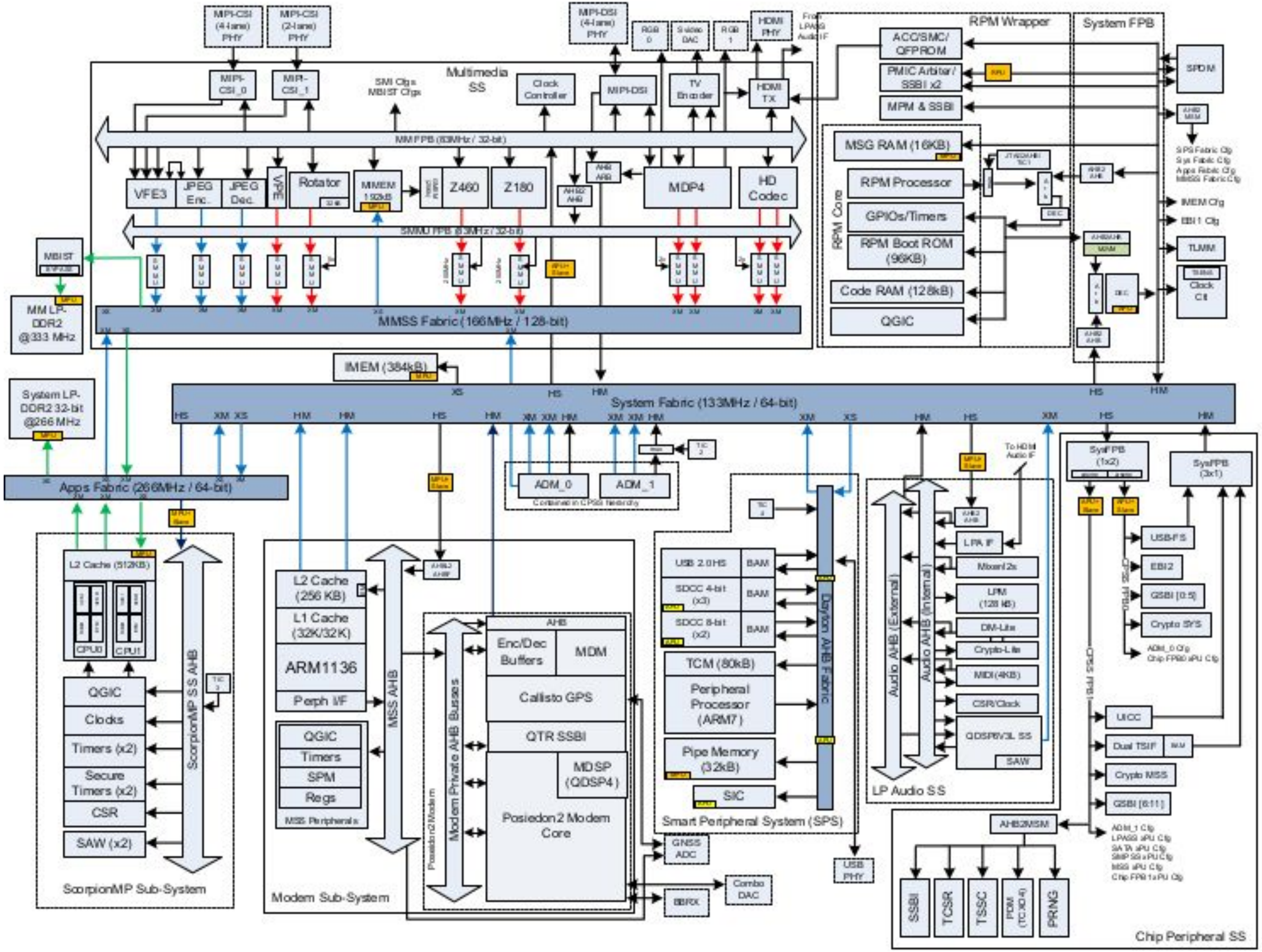
Out-of-order Execution

- CPUs are made up of many hardware blocks
  - Prefetchers
  - Decoders
  - Functional units (Integer units, floating point units, etc)
  - Register file
  - Reservation stations
  - Many more
- Not all hardware is used for each instruction
- Some instructions wait even though there are no direct dependencies
- Some instructions will execute before later instructions
- We want to work as fast as possible

CPU Architecture - Simplified

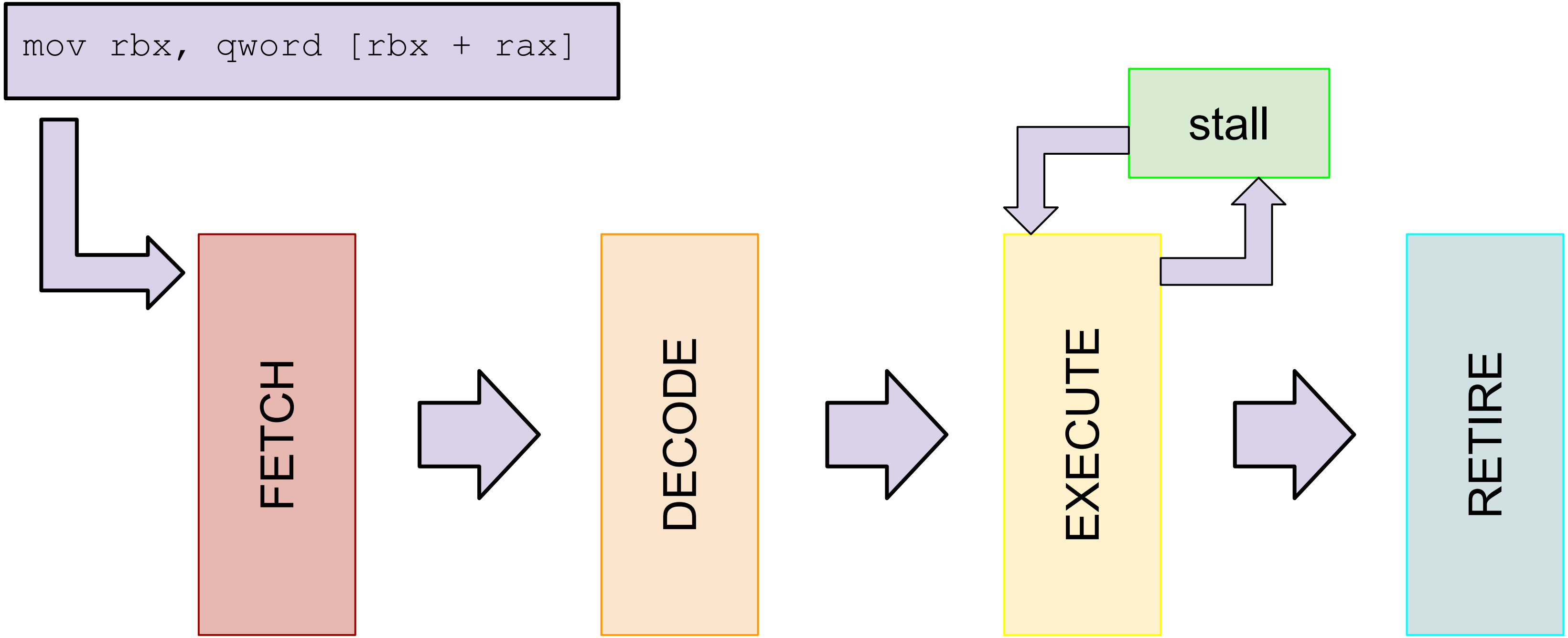


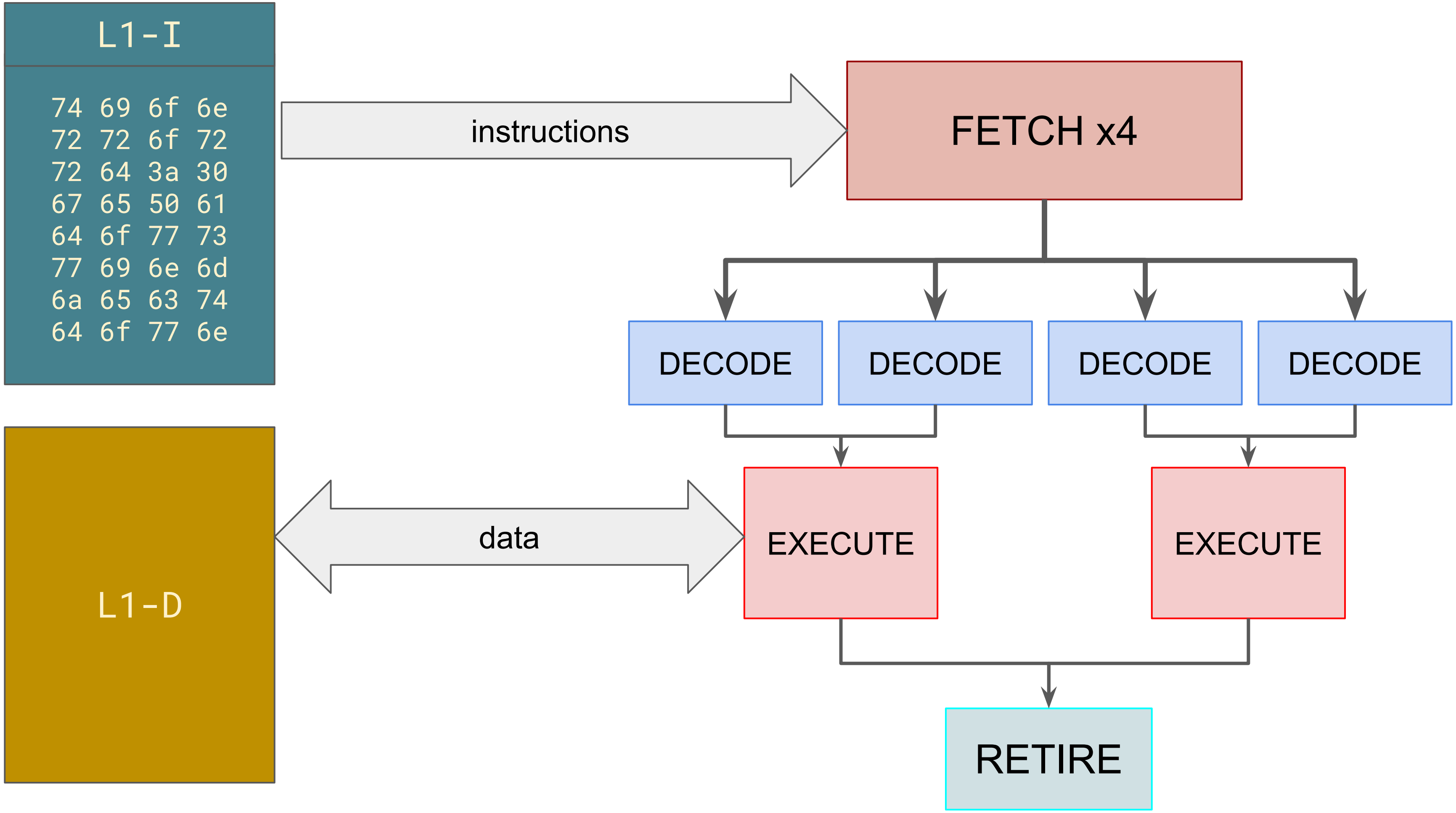
Real Life



Multimedia SS
MM FPB (83MHz / 32-bit)
SMMU FPB (83MHz / 32-bit)
MMSS Fabric (166MHz / 128-bit)
VFE3
JPEG Enc.
JPEG Dec.
Rotator
MMEM 192kB
Z460
Z180
MDP4
HD Codec
MIPI-CSI_0
MIPI-CSI_1
MIPI-DSI
TV Encoder
HDMI TX
Clock Controller
MBIST
MM LP-DDR2 @333 MHz
System LP-DDR2 32-bit @266 MHz
IMEM (384kB)
System Fabric (133MHz / 64-bit)
Apps Fabric (266MHz / 64-bit)
RPM Wrapper
ACC/SMC/ QFPROM
PMIC Arbiter/ SSBI x2
MPM & SSBI
MSG RAM (16KB)
RPM Processor
GPIOs/Timers
RPM Boot ROM (96KB)
Code RAM (128kB)
QGIC
RPM Core
System FPB
SPDM
TLMM
ADM_0
ADM_1
L2 Cache (512KB)
CPU0
CPU1
QGIC
Clocks
Timers (x2)
Secure Timers (x2)
CSR
SAW (x2)
ScorpionMP SS AHB
ScorpionMP Sub-System
L2 Cache (256 KB)
L1 Cache (32K/32K)
ARM1136
Perph I/F
QGIC
Timers
SPM
Regs
MSS Peripherals
MSS AHB
Modem Private AHB Busses
Enc/Dec Buffers
MDM
Callisto GPS
QTR SSBI
MDSP (QDSP4)
Posiedon2 Modem Core
Modem Sub-System
USB 2.0HS
SDCC 4-bit (x3)
SDCC 8-bit (x2)
BAM
TCM (80kB)
Peripheral Processor (ARM7)
Pipe Memory (32kB)
SIC
Dayton AHB Fabric
Smart Peripheral System (SPS)
GNSS ADC
BBRX
Combo DAC
USB PHY
Audio AHB (External)
Audio AHB (Internal)
LPA IF
Mixer/I2s
LPM (128 kB)
DM-Lite
Crypto-Lite
MIDI(4KB)
CSR/Clock
QDSP6V3L SS
SAW
LP Audio SS
SysFPB (1x2)
SysFPB (3x1)
USB-FS
EBI2
GSBI [0:5]
Crypto SYS
UICC
Dual TSIF
Crypto MSS
GSBI [6:11]
AHB2MSM
SSBI
TCSR
TSSC
PRNG
Chip Peripheral SS

Memory Read (sequential)




L1-I
74 69 6f 6e
72 72 6f 72
72 64 3a 30
67 65 50 61
64 6f 77 73
77 69 6e 6d
6a 65 63 74
64 6f 77 6e
instructions
FETCH x4
DECODE
DECODE
DECODE
DECODE
L1-D
data
EXECUTE
EXECUTE
RETIRE

Dependent Operations

```
uint32_t probe[256];

uint8_t A = *(uint_t*)ptr;
uint32_t val = probe[A];
```

How Far?

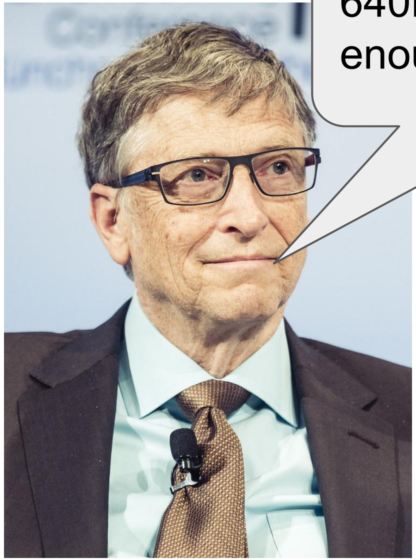
```
83 c4 02              add    $0x2,%sp
ff 46 fc              incw   -0x4(%bp)
83 7e fc 19           cmpw   $0x19,-0x4(%bp)
7c ed                 jl     1e <fn000010+0xe>
b8 38 00              mov    $0x38,%ax
50                    push   %ax
e8 00 00              call   38 <fn000010+0x28>
83 c4 02              add    $0x2,%sp
b8 61 00              mov    $0x61,%ax
50                    push   %ax
```

Background
Virtual Memory
OoO Execution
Linux Memory Management
The Exploit
The Fix
The Damage

Linux Memory Management

- The operating system abstracts the hardware
  - We can make assumptions without understanding the details of the hardware
- Makes sure we can get physical memory when we need it
- Use the hardware (MMU) to protect our memory from other programs

Start	**End**	**Size**	**Description**
`0000000000000000`	`00007fffffffffff`	128 TB	user-space virtual memory
`0000800000000000`	`ffff7fffffffffff`	~16 EB	empty
`ffff800000000000`	`ffffffffffffffff`	128 TB	Kernel-space virtual memory

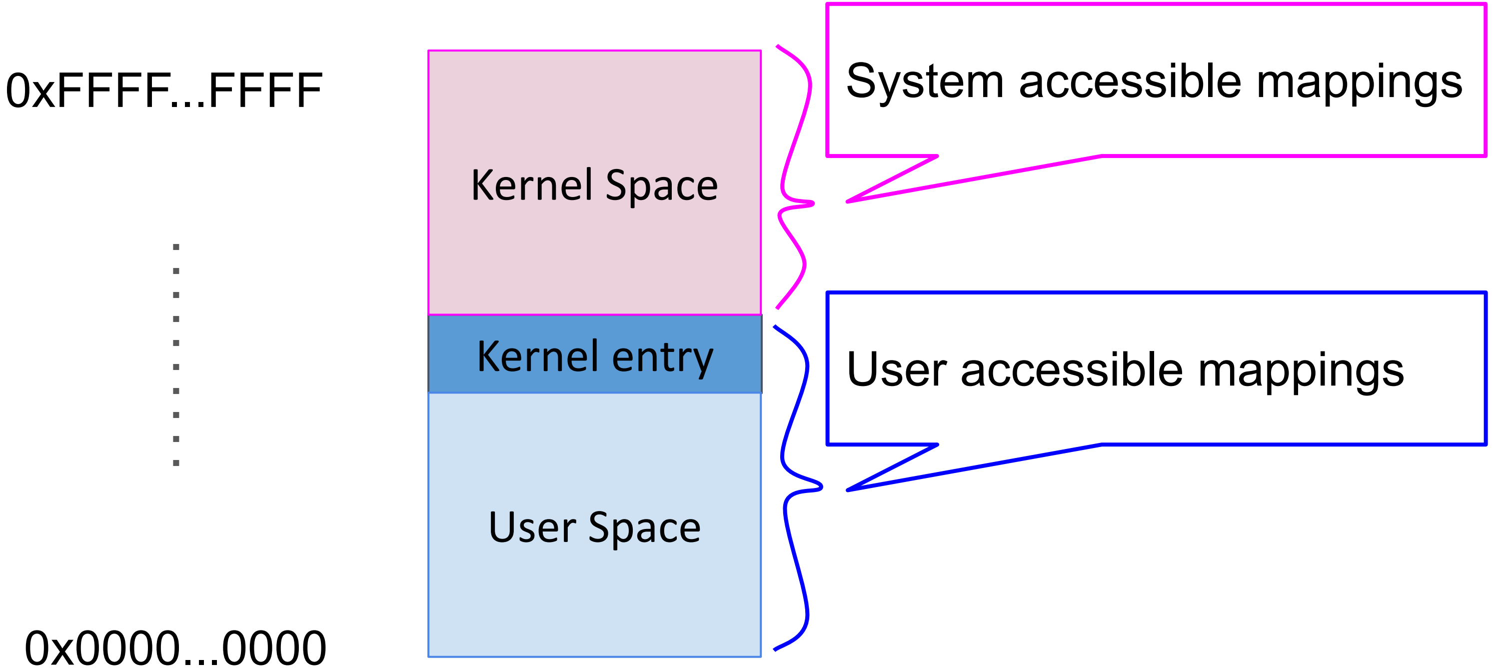


640K ought to be enough for anyone

Let’s use 64-bit addressing, just in case

*This conversation may not have actually taken place

Linux Virtual Address Space Layout (w/o KPTI)



Linux Direct Map

- All physical memory is directly mapped in kernel virtual memory space
- Basis for `phys2virt` and `virt2phys` macros
- Used primarily for drivers and ‘mm’ functions
- This makes memory manipulation code small, fast and efficient
- This is also a big security risk!

Start	**End**	**Size**	**Description**
`ffff888000000000`	`ffffc87fffffffff`	64 TB	Direct Map

Background

Virtual Memory

OoO Execution

Linux Memory Management

The Exploit

The Fix

The Damage

The Exploit

- We want to read kernel memory - how?

Two conditions must hold

1. Mapping of physical page in our virtual address space
2. Permission bit to allow unprivileged access to page

The Exploit

- We want to read kernel memory - how?

Two conditions must hold

1. Mapping of physical page in our virtual address space ✓
2. Permission bit to allow unprivileged access to page

The Exploit

- We want to read kernel memory - how?

Two conditions must hold

1. Mapping of physical page in our virtual address space ✓
2. Permission bit to allow unprivileged access to page ✗

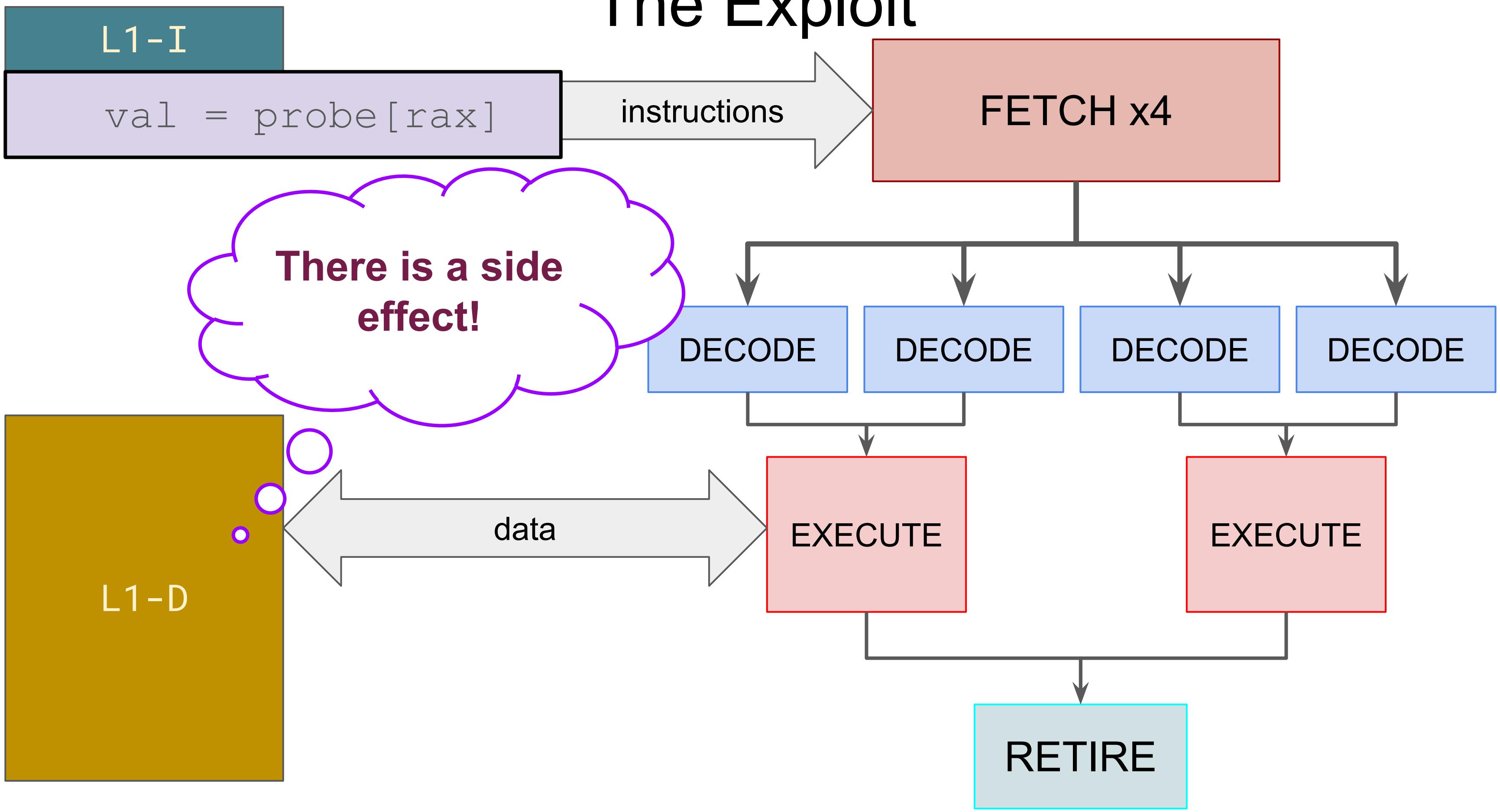
The Exploit in C

```
char val;
char probe[4096 * 256];        the probe array
unsigned long rcx = 0xffff800000000000;  pointer to a kernel address
unsigned long rax = 0;

rax = *(byte*)rcx;          (no permission!)
rax <<= 12;                 shift the secret value by the page size
val = probe[rax];           secret value becomes index into probe array
```

The Exploit

`xor rax, rax`	
`retry:`	
`mov al, byte [rcx]`	rcx is a pointer to a kernel address (no permission!)
`shl rax, 0xc`	shift the secret value by the page size secret value becomes index into probe array
`jz retry`	
`mov rbx, qword [rbx + rax]`	rbx is the base address of the probe array

The Exploit


L1-I
val = probe[rax]
instructions
FETCH x4
There is a side effect!
DECODE
DECODE
DECODE
DECODE
L1-D
data
EXECUTE
EXECUTE
RETIRE

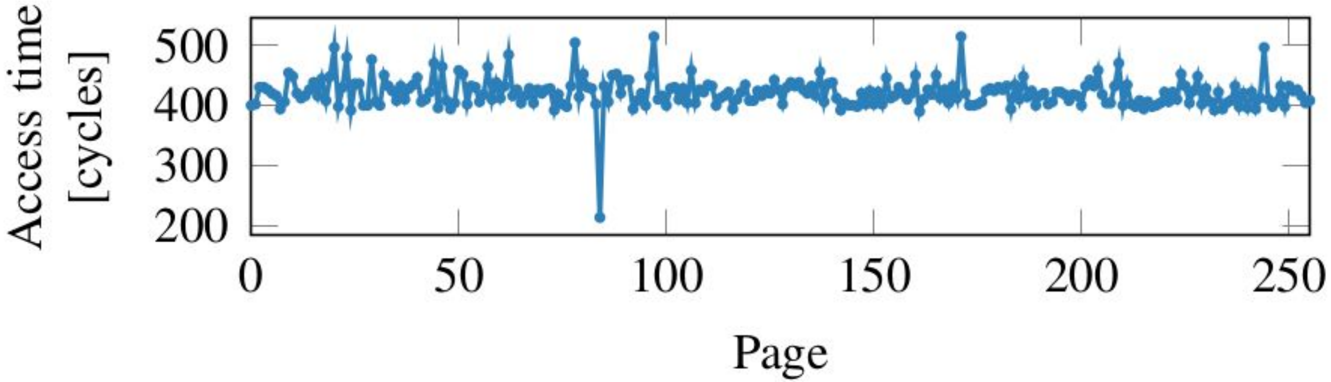
The Exploit

```
unsigned long rax = 0;
char probe[4096 * 256];                   the probe array
unsigned long rcx = 0xffff800000000000;   pointer to a kernel address
char val;

rax = *(byte*)rcx;   Exception!

rax <<= 12;
val = probe[rax];
```

Already scheduled and perhaps executed

Flush + Reload

- Make sure the cache is empty (clflush)
- Perform attack
- Read all entries in the probe array, and measure access time
- One measurement might stand out!
- Index of cached page is the value of the secret byte



Accessing All Memory

- Now we know how to access kernel memory!
  - Not very fast, but it works
- But how to access memory of another process?
  - Linux manages all processes (including their hierarchy) in a linked list
  - The head of this task list is stored in the init_task structure
- Use the direct memory map
  - Must find the page tables belonging to another process
  - Perform a page walk to find the physical page for a particular virtual address
  - Access that physical page through the direct map

Performance

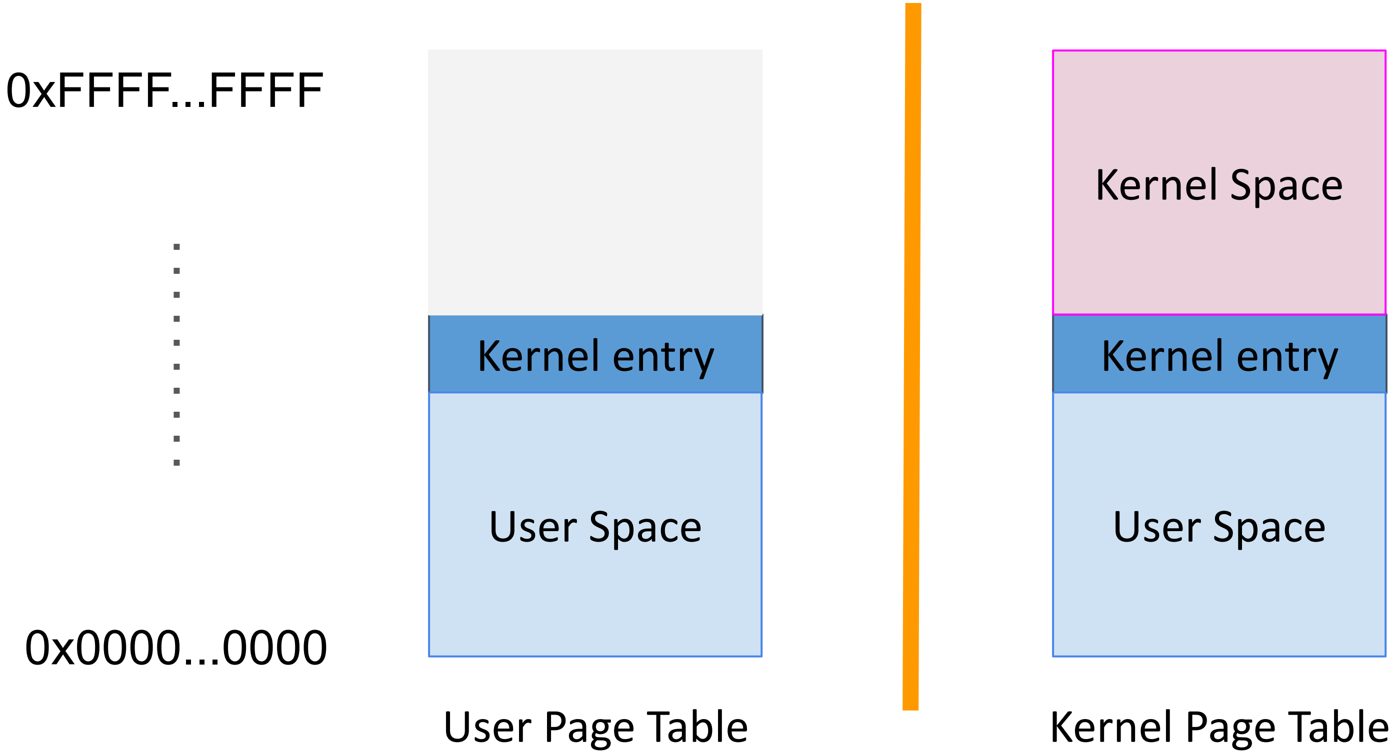
- Flush-Reload is the bottleneck of the attack
- Instead of 8 bits (=256 entries), send 1 bit (=2 entries) of information
  - Much faster
  - Less reliable (noise bias to ‘0’)
- Can read memory at rates between 4KB/s - 500KB/s

Background
Virtual Memory
OoO Execution
Linux Memory Management
The Exploit
The Fix
The Damage

The Fix

- KPTI - Kernel Page Table Isolation
- Based on KAISER patches
- Removes kernel mappings from user process virtual memory
- Requires a pair of page tables for each process
  - One for user space
  - One for kernel space
- Drastically increases overhead during context switch

Linux Virtual Address Space Layout (with KPTI)



Background

Virtual Memory

OoO Execution

Linux Memory Management

The Exploit

The Fix

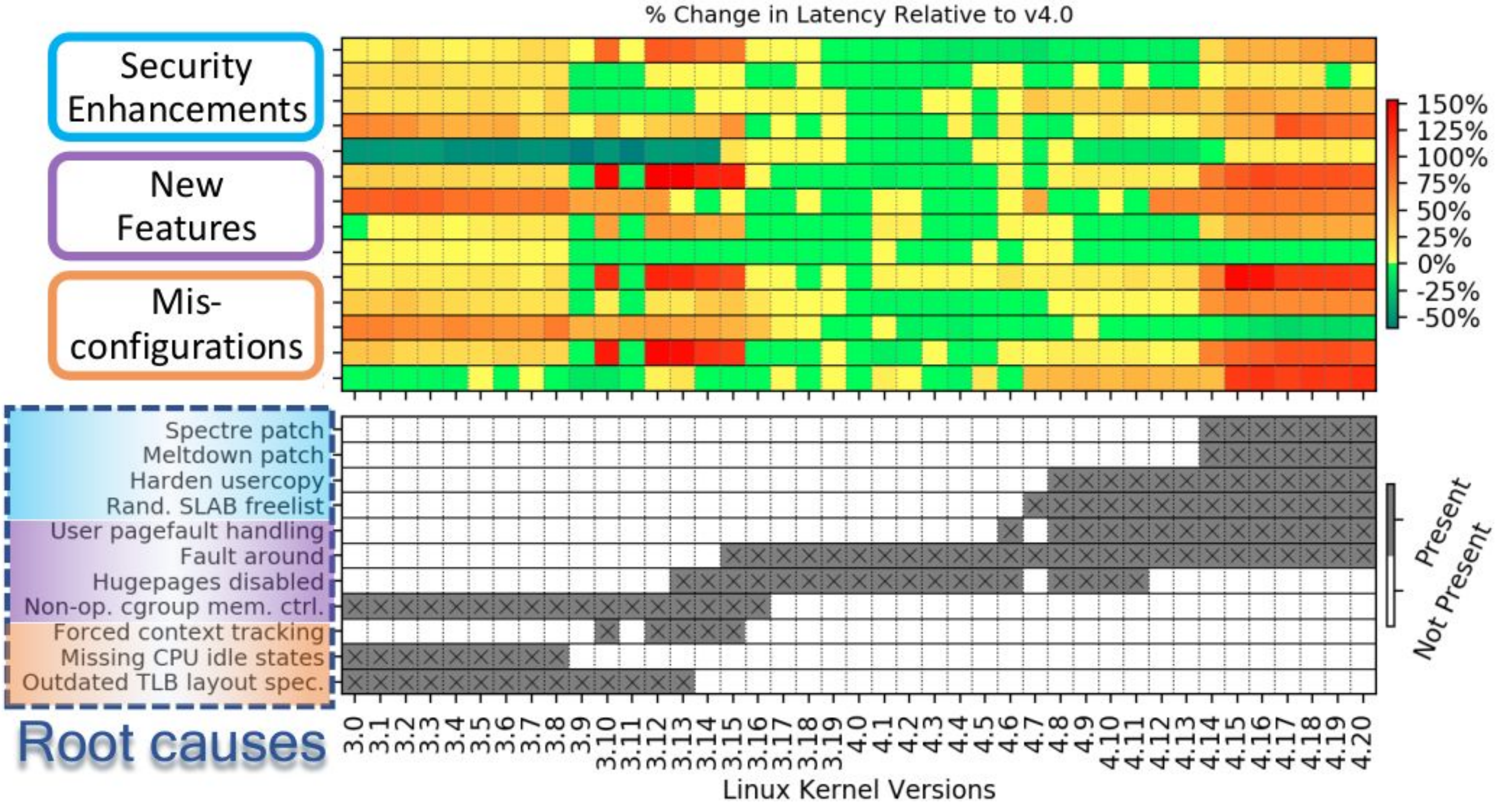
The Damage

The Damage

- Measurements are very dependent on the number of syscalls
- The overhead was measured to be 0.28% according to KAISER's original authors
- a Linux developer measured it to be roughly 5% for most workloads and up to 30% in some cases
- for database engine PostgreSQL the impact on read-only tests on an Intel Skylake processor was 16–23% (without PCID)
- Redis slowed by 6–7%

- Linux kernel compilation slowed down by 5% on Haswell

The Damage



% Change in Latency Relative to v4.0
Security Enhancements
New Features
Mis-configurations
150%
125%
100%
75%
50%
25%
0%
-25%
-50%
Spectre patch
Meltdown patch
Harden usercopy
Rand. SLAB freelist
User pagefault handling
Fault around
Hugepages disabled
Non-op. cgroup mem. ctrl.
Forced context tracking
Missing CPU idle states
Outdated TLB layout spec.
Root causes
Present
Not Present
3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 3.10 3.11 3.12 3.13 3.14 3.15 3.16 3.17 3.18 3.19 4.0 4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 4.10 4.11 4.12 4.13 4.14 4.15 4.16 4.17 4.18 4.19 4.20
Linux Kernel Versions

Making It Hurt Slightly Less

- PCIDs allow a logical processor to cache information for multiple linear-address spaces
- Allows us to bypass the TLB flush on syscall entry/exit

- PostgreSQL read-only tests on an Intel Skylake processor was 7–17% (or 16–23% without PCID)

Conclusions

- Even the most commonly used, professionally made chips have bugs
- Operating systems can (sometimes) be used to mask these bugs
- Even so, the bugs are costly!

Meltdown
Spectre
L1TF
RIDL
Fallout
MDS
More??

References

https://sosp19.rcs.uwaterloo.ca/slides/ren.pdf

https://meltdownattack.com/meltdown.pdf

https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

https://en.wikipedia.org/wiki/Kernel_page-table_isolation