black hat USA 2016

DrK: Breaking Kernel Address Space Layout Randomization with Intel TSX

Yeongjin Jang, Sangho Lee, and Taesoo Kim

Georgia Institute of Technology, August 3, 2016

KASLR: A Practical Barrier for Exploits

Pod2g Finds Exploits for iOS 5.1 Jailbreak, Working On Bypassing ASLR

by Gary Ng on Wednesday, April 18th, 2012 - 1:08am PDT

Pod2g is back at work on a new iOS 5.1 jailbreak, as he noted last month. It was during this time he also confirmed he was searching for vulnerabilities within iOS 5.1.

Now, it looks like some hard work has paid off. He just tweeted that he (along with the Chronic Dev Team) has found exploits for a new iOS 5.1 jailbreak and is currently working on bypassing ASLR during bootup:

> "News: we have all exploits required to do a new jailbreak. I'm working on bypassing ASLR at bootup."

pod2g
@pod2g
Following

News: we have all exploits required to do a new jailbreak. I'm working on bypassing ASLR at bootup.

iPhoneinCanada.ca

ASLR BYPASS APOCALYPSE IN RECENT ZERO-DAY EXPLOITS

October 15, 2013 | by Xiaobo Chen | Vulnerabilities, Exploits, Threat Research, Targeted Attack

ASLR (Address Space Layout Randomization) is one of the most effective protection mechanisms in modern operation systems. But it's not perfect. Many recent APT attacks have used innovative techniques to bypass ASLR.

Here are just a few interesting bypass techniques that we have tracked in the past year:

- **Using non-ASLR modules**
- **Modifying the BSTR length/null terminator**
- **Modifying the Array object**

The following sections explain each of these techniques in detail.

Example: Linux

- To escalate privilege to root through a kernel exploit, attackers want to call commit_creds(prepare_kernel_creds(0)).

```
// full-nelson.c
static int __attribute__((regparm(3)))
getroot(void * file, void * vma)
{
    commit_creds(prepare_kernel_cred(0));
    return -1;
}
// https://blog.plenz.com/2013-02/privilege-escalation-kernel-exploit.html
int privesc(struct sk_buff *skb, struct nlmsghdr *nlh)
{
    commit_creds(prepare_kernel_cred(0));
    return 0;
}
```

Example: Linux

- Kernel symbols are hidden to non-root users.

```
[blue9057@pt ~$] cat /proc/kallsyms | grep ' commit_creds\| prepare_kernel'
0000000000000000 T commit_creds
0000000000000000 T prepare_kernel_cred
```

- KASLR changes kernel symbol addresses every boot.

```
[blue9057@pt ~$] sudo cat /proc/kallsyms | grep ' commit_creds\| prepare_kernel'
ffffffffaa0a3bd0 T commit_creds
ffffffffaa0a3fc0 T prepare_kernel_cred
```

1st Boot

```
[blue9057@pt ~$] sudo cat /proc/kallsyms | grep ' commit_creds\| prepare_kernel'
ffffffffbd0a3bd0 T commit_creds
ffffffffbd0a3fc0 T prepare_kernel_cred
```

2nd Boot

Example: tpwn - OS X 10.10.5 Kernel Privilege Escalation Vulnerability

- [CVE-2015-5864] IOAudioFamailiy allows a local user to obtain sensitive kernel memory-layout information via unspecified vectors.

```
char found = 0;
DO_TIMES(ALLOCS) {
    char* data = read_kern_data(heap_info[ctr].port);
    if (!found && memcmp(data,vz,1024 - 0x58)) {
        kslide = (*(uint64_t*)((1024-0x58+(char*)data))) - kslide ;
        found=1;
    }
}
if (!found) {
    exit(-3);
}
printf("leaked kaslr slide, @ 0x%016llx\n", kslide);
```

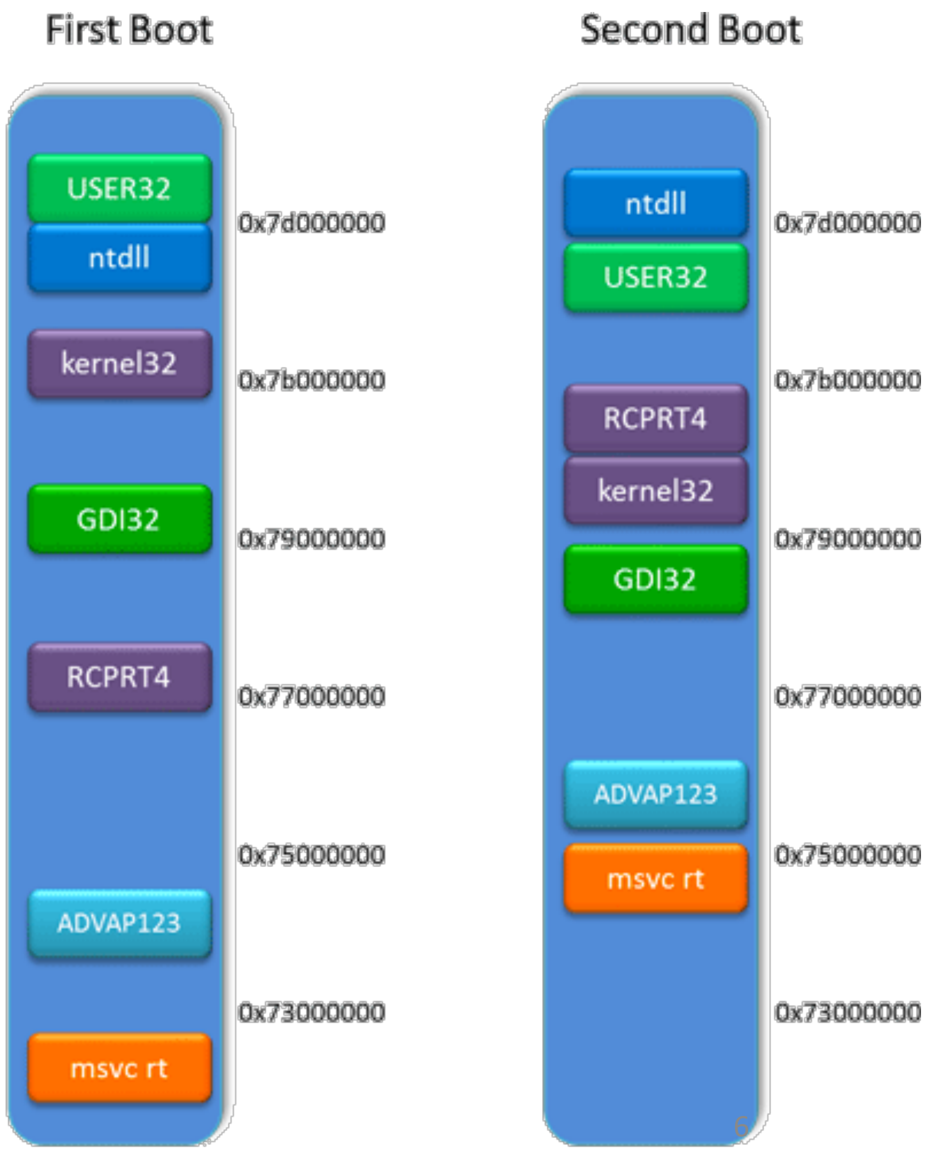
Bypassing KASLR is required…

Kernel Address Space Layout Randomization (KASLR)

- A statistical mitigation for memory corruption exploits

- Randomize address layout per each boot
  - Efficient (<5% overhead)

- Attacker should guess where code/data are located for exploit.
  - In Windows, a successful guess rate is 1/8192.



KASLR Makes Attacks Harder

- KASLR introduces an additional bar to exploits
  - Finding an information leak vulnerability

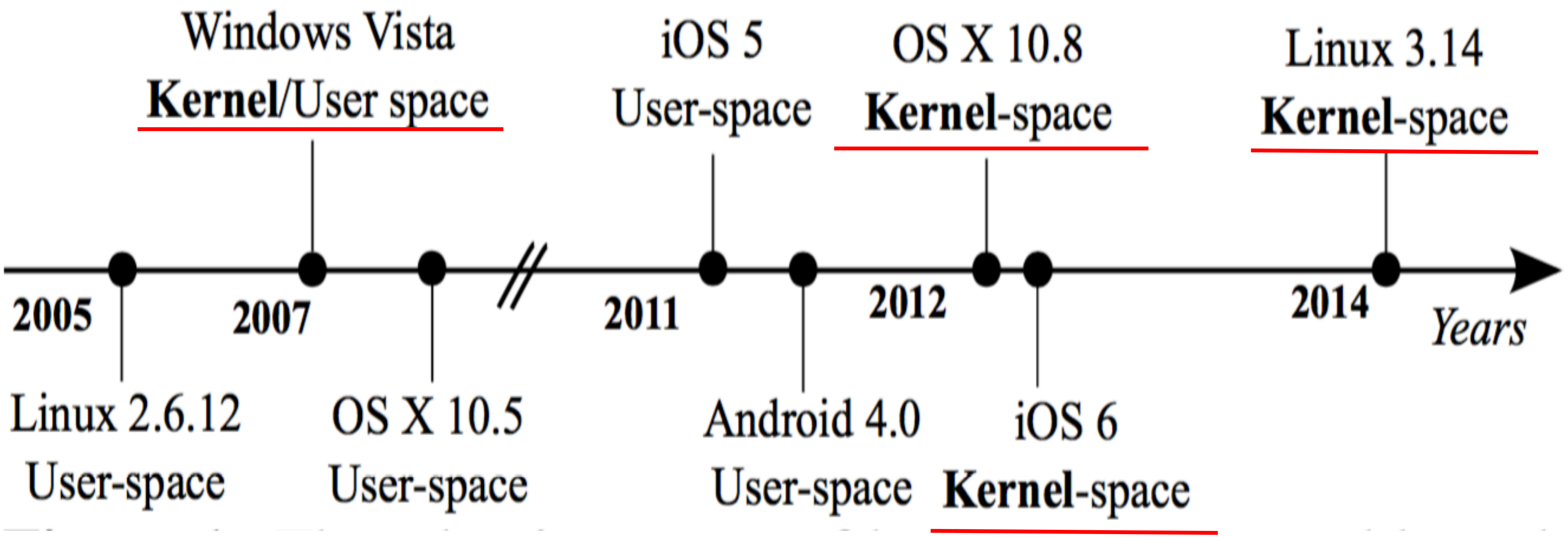
$$\Pr[\ \exists\ \text{Memory Corruption Vuln}\ ]$$

⬇

$$\Pr[\ \exists\ \text{information\_leak}\ ] \times \Pr[\ \exists\ \text{Memory Corruption Vuln}]$$

- Both attackers and defenders aim to detect info leak vulnerabilities.

Popular OSes Adopted KASLR



Is there any other way than info leak?

- Practical Timing Side Channel Attacks Against Kernel Space ASLR (Hund et al., Oakland 2013)
  - A **hardware-level** side channel attack against KASLR
  - **No** information leak vulnerability in OS is required

TLB Timing Side Channel

- If accessed a kernel address from the user space

```
blue9057@pt ~ $ ./access_address 0xffffffff80000000
Accessing address 0xffffffff80000000
[1]    15990 segmentation fault (core dumped)  ./access_address 0xffffffff80000000
```

Unmapped address

```
blue9057@pt ~ $ sudo cat /proc/kallsyms | grep \ commit_creds
ffffffffaa0a3bd0 T commit_creds
blue9057@pt ~ $ ./access_address 0xffffffffaa0a3bd0
Accessing address 0xffffffffaa0a3bd0
[1]    16025 segmentation fault (core dumped)  ./access_address 0xffffffffaa0a3bd0
```

Mapped address

- Regardless of its mapping status, it generates page fault.

TLB Timing Side Channel

- If an unmapped kernel address is accessed

Invalid address -> Page Fault

1. Try to get page table entry through page table walk

2. There is no page table entry found, generate page fault!

TLB Timing Side Channel

- If a mapped kernel address is accessed

Access Violation -> Page Fault

1. Try to get page table entry through page table walk

2. Cache the entry to TLB

3. Check page privilege level (3<0), generate page fault!

TLB Timing Side Channel

Virtual Address

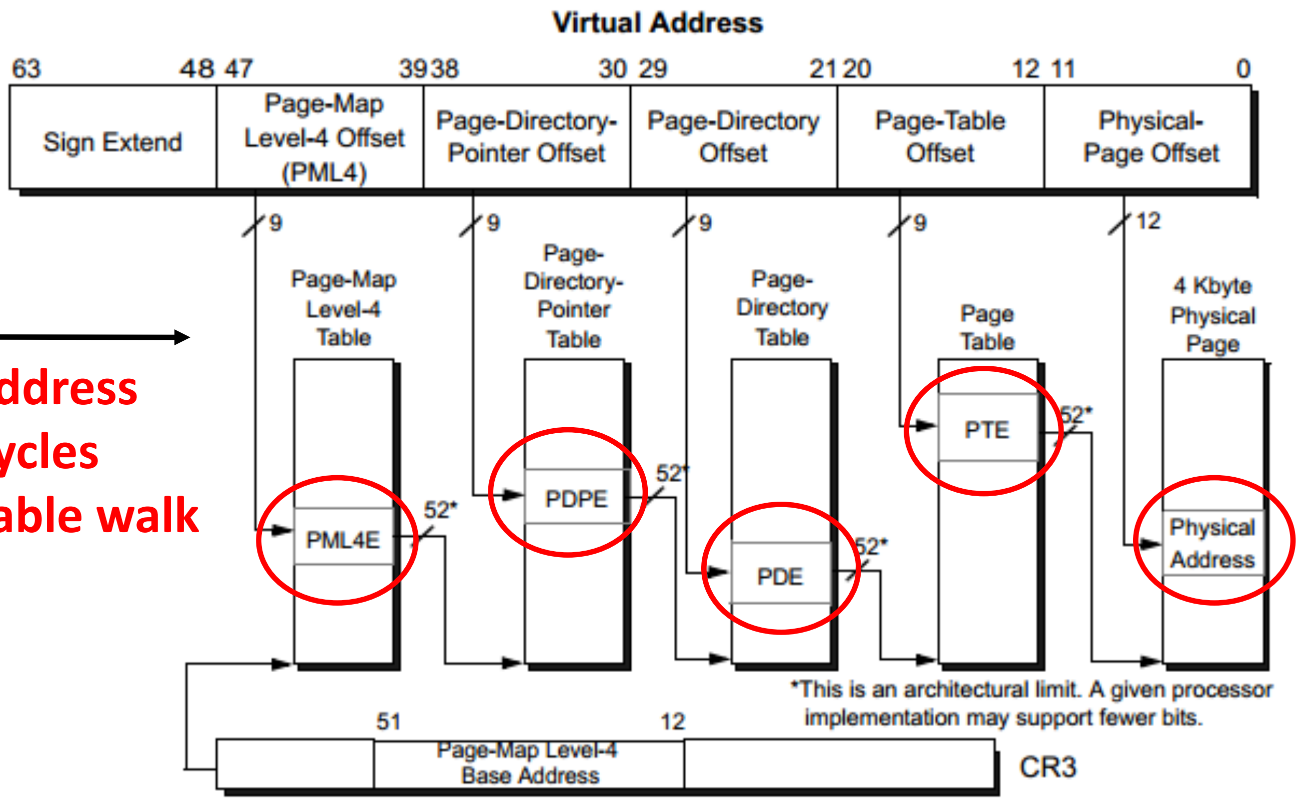
TLB

Miss

Hit

Unmapped address takes ~40 cycles more for page table walk

Mapped address returns quicker!



TLB Timing Side Channel

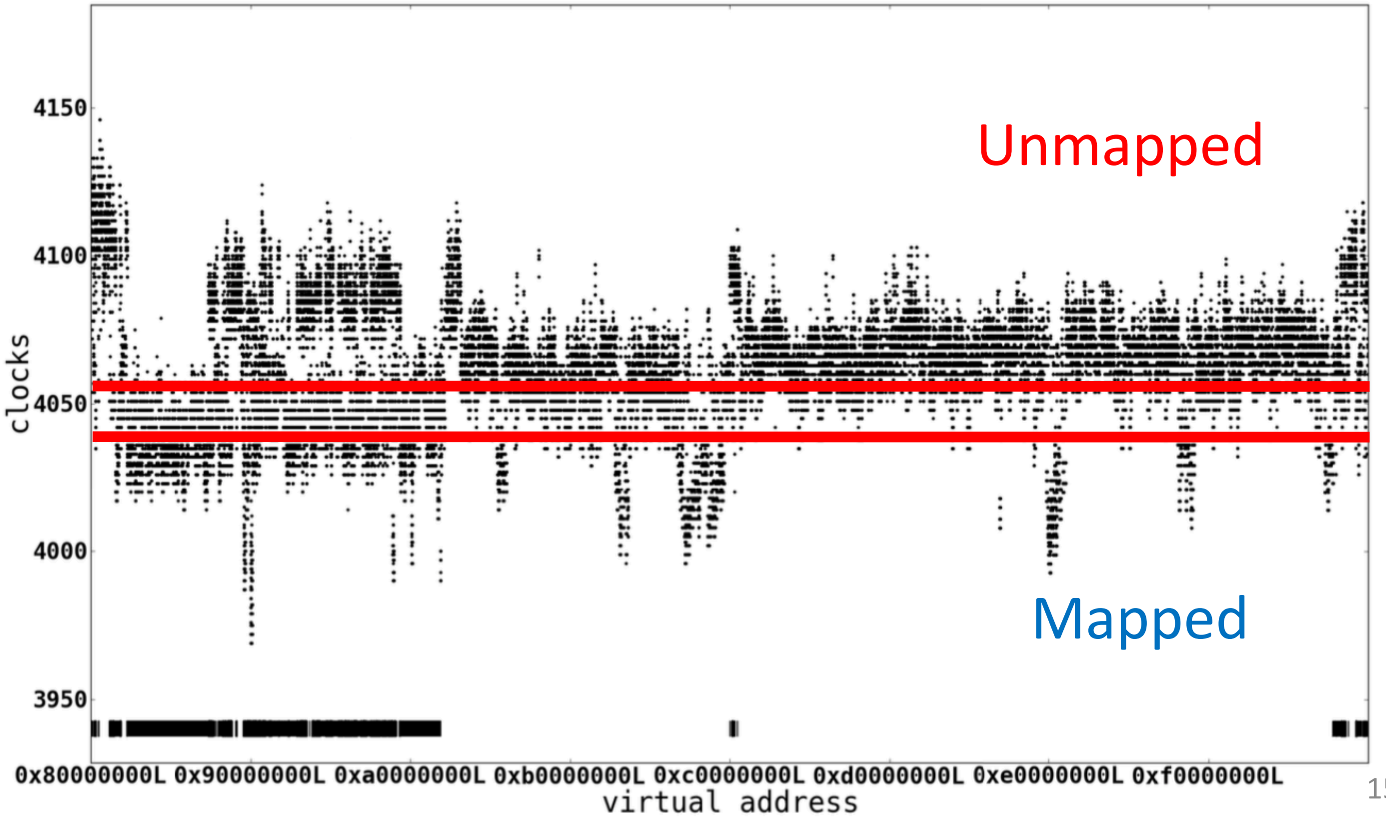
- Measuring the time in an exception handler

```
__try
{
    // a kernel address
    int *p = (int*)0xffffffff80000000;
    time_begin=__rdtscp();
    *p = 0;
}
__except(EXCEPTION_EXECUTE_HANDLER)
{
    time_diff = __rdtscp() - time_begin;
    // if time_diff < 4050, it is a mapped address
}
```

1. Generates Page Fault
2. CPU generates Page Fault
3. OS handles Page Fault
4. OS calls exception handler

TLB Timing Side Channel

- Result: Fault with TLB hit took less than 4050 cycles
  - While TLB miss took more than that…
- Limitation: Too noisy
  - Why????



TLB Timing Side Channel

Measured Time (~4000 cycles)

User | CPU | TL | OS Exception Handling | OS Noise

If we can *eliminate* the noise at OS, then the timing channel will be more stable.

...on

...on

TLB Side Channel

OS Execution

OS Handling Noise

OS Noise (~100 cycles)

OS Noise

Fault Handling Noise is *too much*!

A More Practical TLB Side Channel Attack on KASLR

- DrK Attack: We present a very practical side channel attack on KASLR
  - De-randomizing Kernel ASLR (this is where DrK comes from)
- Exploit Intel TSX for eliminate the noise from OS

	DrK	Hund et. al.
Channel Noise	Negligible	A lot of noise from OS
Speed	**5** sec for **100%** accuracy **0.1** sec for Linux	**65** seconds for **94.92%**
Covertness	OS do not know	Page fault handler is called at OS
Precision	**U** / **NX** / **X**	**U** / **M**
Tested OSes	Linux/Windows/OS X (**64bit**)	Windows 7 **32bit**

Starting From a PoC Example in the Wild

TSX to the rescue

Less noisy

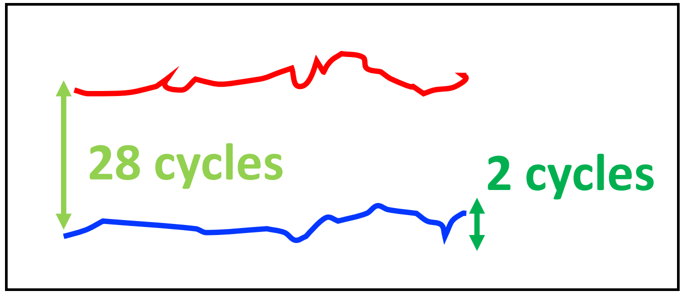
TSX makes kernel address probing much faster and less noisy. If an instruction executed within XBEGIN/XEND block (in usermode) tries to access kernel memory, then no page fault is raised – instead transaction abort happens, so execution never leaves usermode. On my i7-4800MQ CPU, the relevant timings, in CPU cycles, are (minimal/average/variance, 2000 probes, top half of results discarded):

1. access in TSX block to mapped kernel memory: 172 175 2
2. access in TSX block to unmapped kernel memory: 200 200 0
3. access in __try block to mapped kernel memory: 2172 2187 35
4. access in __try block to unmapped kernel memory: 2192 2213 57

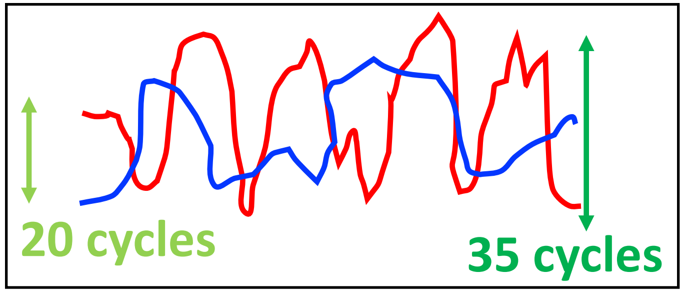
Rafal Wojtczuk, https://labs.bromium.com/2014/10/27/tsx-improves-timing-attacks-against-kaslr/

TSX Gives Better Precision on Timing Attack

- Access to mapped address in TSX: 172 clk
- Access to unmapped address in TSX : 200 clk
  - 28 clk in timing difference, with stddev **0~2**



- Access to mapped address in __try: 2172 clk
- Access to unmapped address in _try: 2192 clk
  - 20 clk in timing difference, with stddev **35~57**



Transactional Synchronization Extension (Intel TSX)

- Traditional Lock

```
pthread_mutex_t *mutex;
pthread_mutex_lock(mutex);

// atomic region
do_atomic_operation();

pthread_mutex_unlock(mutex);
// atomic region end
```

1. Block until acquires the lock

2. Atomic region (Guaranteed!)

3. Release the lock (finishes atomic region)

Transactional Synchronization Extension (Intel TSX)

- TSX: relaxed but faster way of handling synchronization

```
int status = 0;
if( (status = _xbegin()) == _XBEGIN_STARTED) {

    // atomic region
    try_atomic_operation();

    _xend();
    // atomic region end
}
else {

    // if failed,
    handle_abort();

}
```

1. Do not block, do not use lock

2. Try atomic operation (can fail)

3. If failed, handle failure with abort handler (retry, get back to traditional lock, etc.)

Transaction Aborts If Exist any of a Conflict

```
int status = 0;
if( (status = _xbegin()) == _XBEGIN_STARTED) {

    // atomic region
    try_atomic_operation();

    _xend();
    // atomic region end
}
else {

    // if failed,
    handle_abort();

}
```

Run If Transaction Aborts

- Condition of Conflict
  - Thread races
  - Cache eviction (L1 write/L3 read)
  - Interrupt
    - Context Switch (timer)
    - Syscalls
  - Exceptions
    - **Page Fault**
    - General Protection
    - Debugging
    - …

Abort Handler Suppresses Exceptions

```
int status = 0;
if( (status = _xbegin()) == _XBEGIN_STARTED) {

    // atomic region
    try_atomic_operation();

    _xend();
    // atomic region end
}
else {

    // if failed,
    handle_abort();

}
```

Run If Transaction Aborts

- Abort Handler of TSX
  - Suppress all sync. exceptions
    - E.g., page fault
  - **Do not notify OS**
    - Just jump into abort_handler()

No Exception delivery to the OS! (returns quicker, so less noisy than __try __except)

Exploiting TSX as an Exception Handler

- How to use TSX as an exception handler?

```
uint64_t time_begin, time_diff;
int status = 0;
int *p = (int*)0xffffffff80000000; // kernel addresss
time_begin = __rdtscp();
if((status = _xbegin()) == _XBEGIN_STARTED) {
    // TSX transaction
    *p; // read access
    // or,
    ((int(*)())p)(); // exec access
}
else {
    // abort handler
    time_diff = __rdtscp() - time_begin;
}
```

1. Timestamp at the beginning

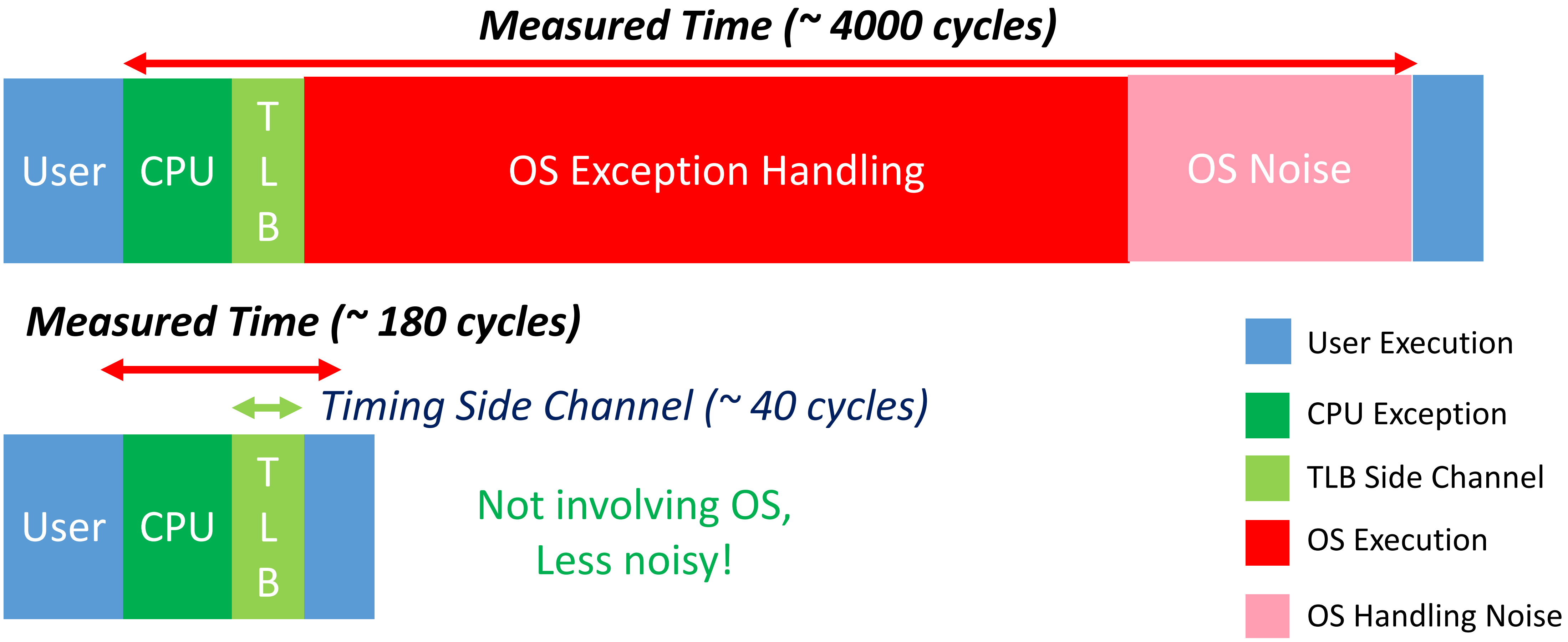
2. Access kernel memory within the TSX region (always aborts)

Processor directly calls the handler

OS handling path is *not* involved

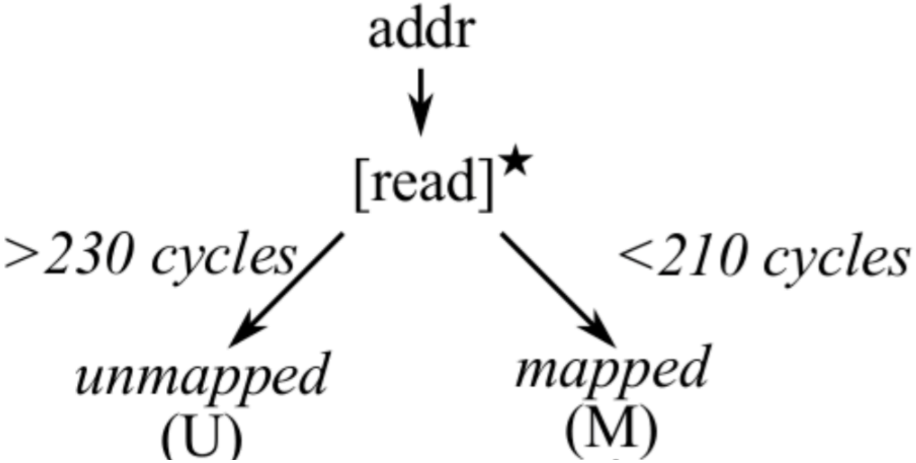
3. Measure timing at abort handler

Reducing Noise with Intel TSX



Measuring Timing Side Channel

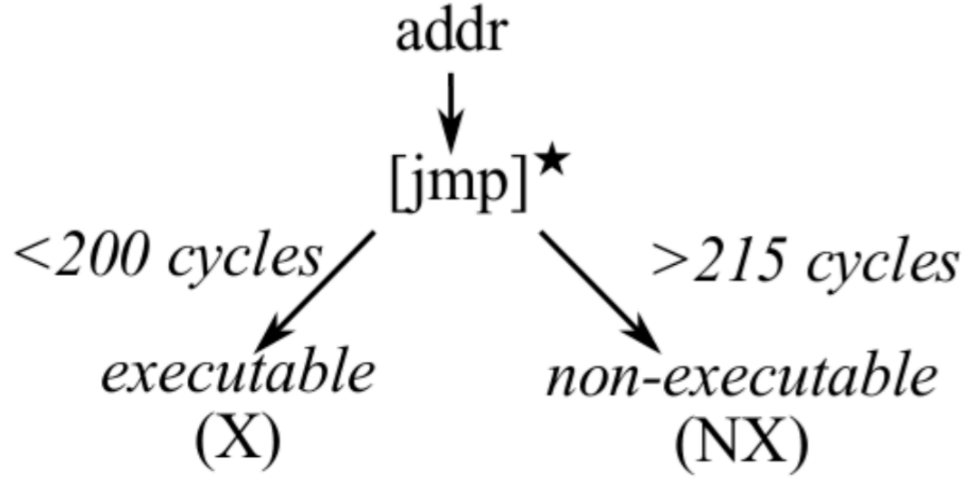
- Access Mapped / Unmapped kernel addresses
  - Attempt READ access within the TSX region
    - `mov [rax], 1`




```
def probe(addr):
    beg = rdtsc()
    if _xbegin():
        [mode]★
    else
        end = rdtsc()
    return end - beg
```

Measuring Timing Side Channel

- Access Executable / Non-executable address
  - Attempt JUMP access within the TSX region
    - **`jmp rax`**




```
def probe(addr):
    beg = rdtsc()
    if _xbegin():
        [mode]★
    else
        end = rdtsc()
    return end - beg
```

Demo 1: Timing Difference on M/U and X/NX

- Video Link
  - https://www.youtube.com/watch?v=NdndV_cMJ8k

Measuring Timing Side Channel

- Mapped / Unmapped kernel addresses
  - Ran 1000 iterations for the probing, minimum clock on 10 runs

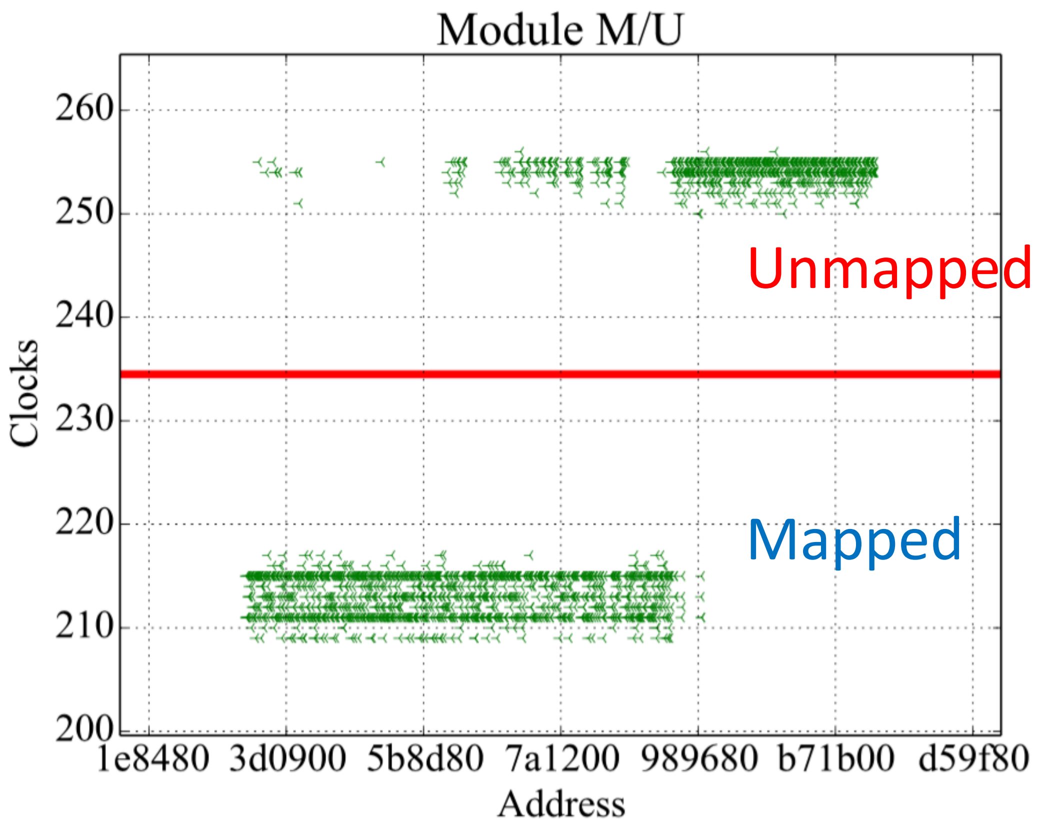
Processor	Mapped Page	Unmapped Page
i7-6700K (4.0Ghz)	209	240 (+31)
i5-6300HQ (2.3Ghz)	164	188 (+24)
i7-5600U (2.6Ghz)	149	173 (+24)
E3-1271v3 (3.6Ghz)	177	195 (+18)

Measuring Timing Side Channel

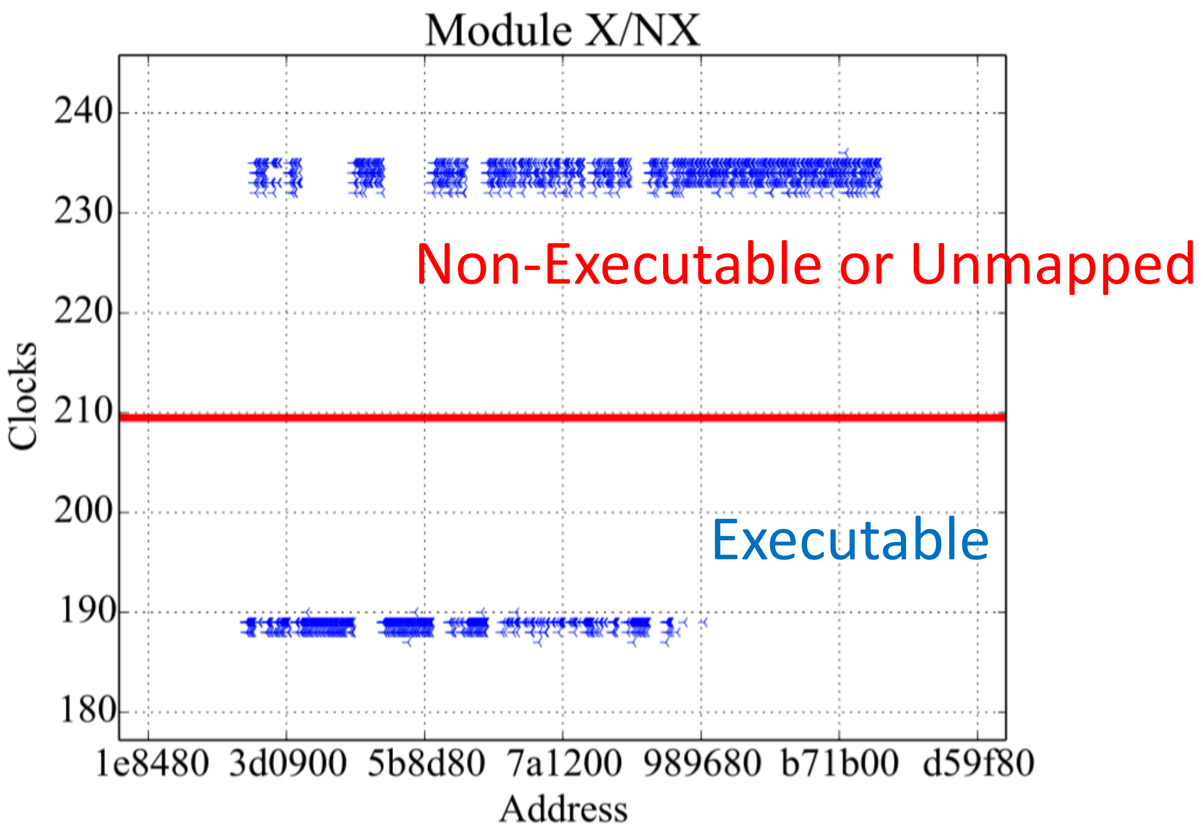
- Executable / Non-executable kernel addresses
  - Ran 1000 iterations for the probing, minimum clock on 10 runs

Processor	Executable Page	Non-exec Page
i7-6700K (4.0Ghz)	181	226 (+45)
i5-6300HQ (2.3Ghz)	142	178 (+36)
i7-5600U (2.6Ghz)	134	164 (+30)
E3-1271v3 (3.6Ghz)	159	189 (+30)

Clear Timing Channel



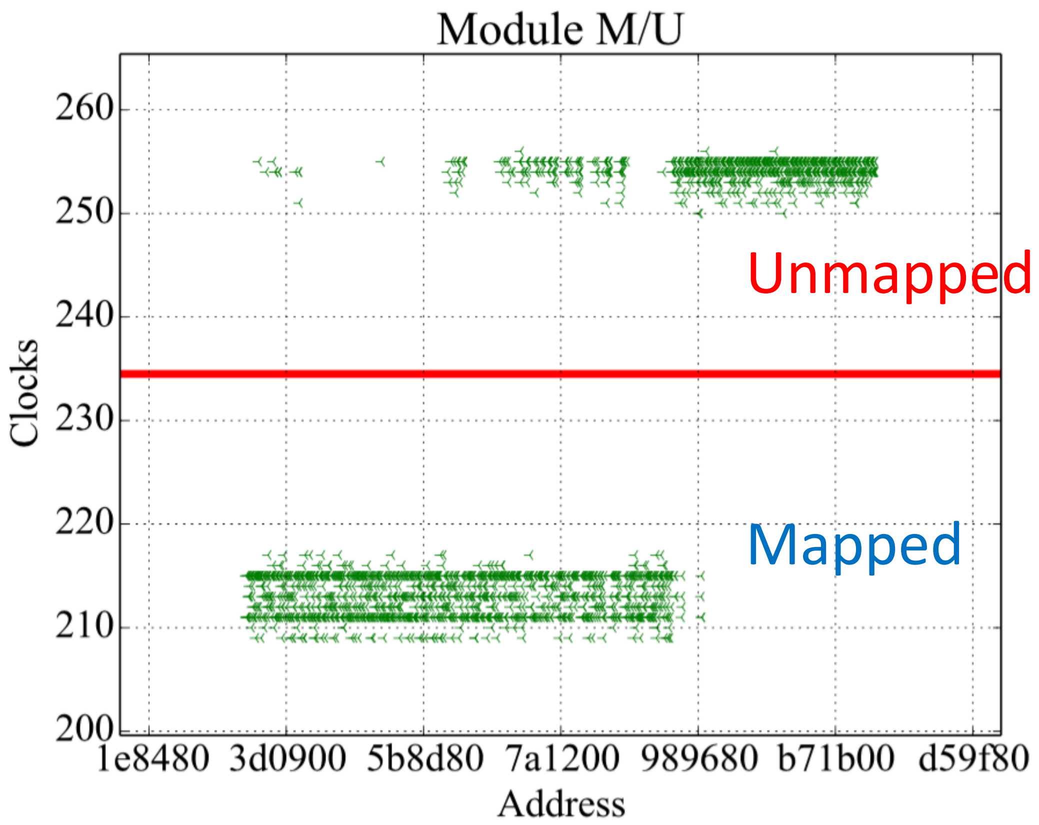
(a) Mapped vs. Unmapped



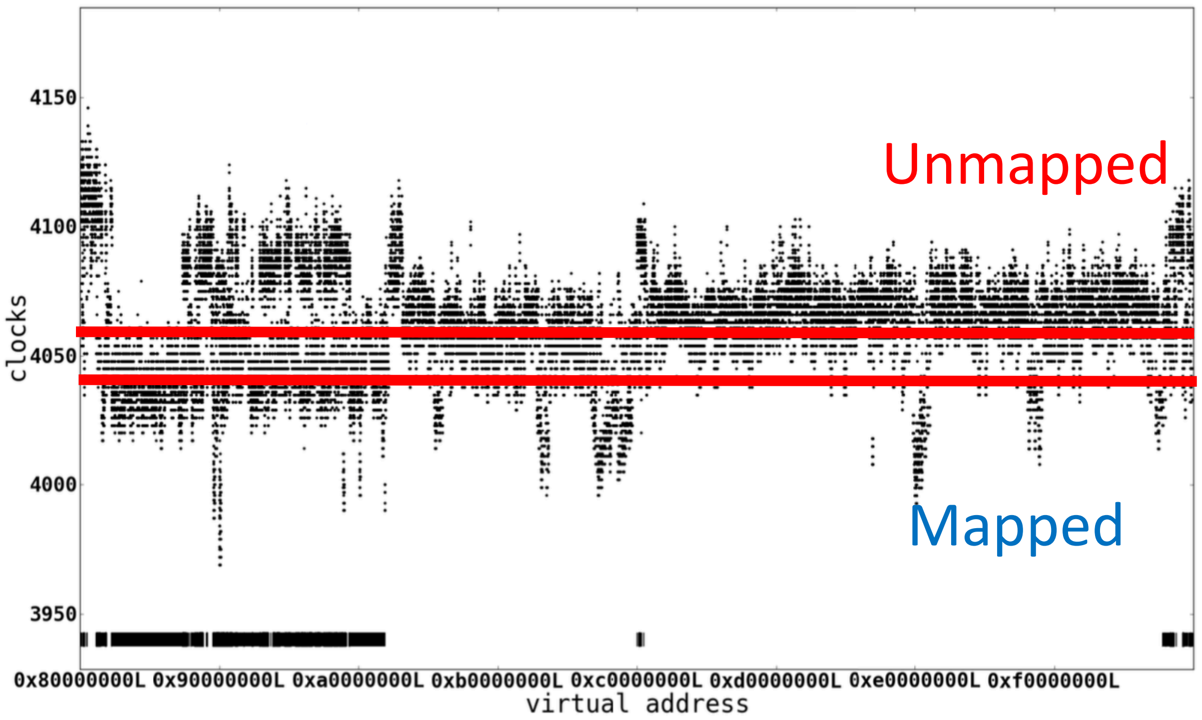
(b) Executable vs. Non-executable

Clear separation between different mapping status!

TSX vs SEH



(a) Mapped vs. Unmapped



Clear separation between different mapping status!

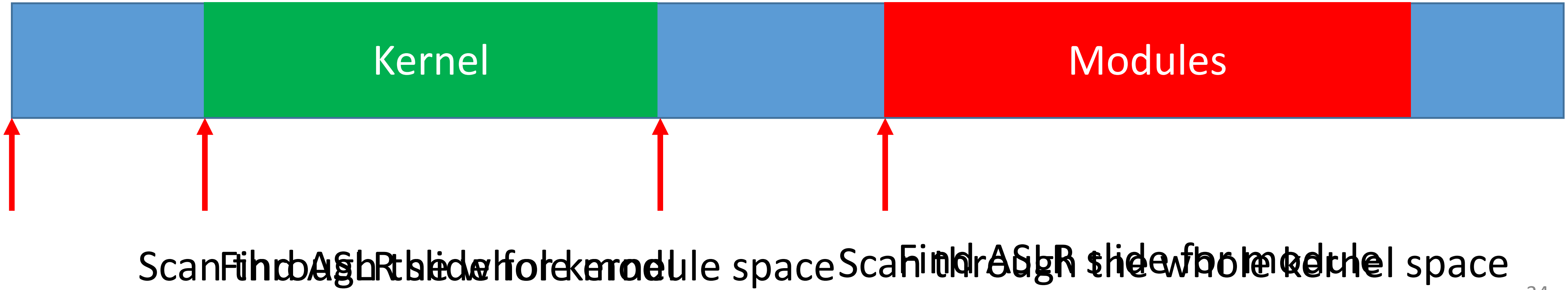
Attack on Various OSes

- Attack Targets
  - DrK is hardware side-channel attack
    - The mechanism is independent to OS
  - We target popular OSes: Linux, Windows, and OS X

- Attack Types
  - Type 1: Revealing mapping status of each page
  - Type 2: Finer-grained module detection

Attack on Various OSes

- Type 1: Revealing mapping status of each page
  - Find the start location of Kernel / Module (ASLR slide)
    - Mostly they are located contiguously in a chunk



Attack on Various OSes

- Type 1: Revealing mapping status of each page
  - Try to reveal the mapping status per each page in the area
    - X (executable) / NX (Non-executable) / U (unmapped)

```
0xffffffffc0278000-0xffffffffc027d000 U
0xffffffffc027d000-0xffffffffc0281000 X
0xffffffffc0281000-0xffffffffc0285000 NX
0xffffffffc0285000-0xffffffffc0289000 U
0xffffffffc0289000-0xffffffffc028b000 X
0xffffffffc028b000-0xffffffffc028e000 NX
0xffffffffc028e000-0xffffffffc0293000 U
0xffffffffc0293000-0xffffffffc02b7000 X
0xffffffffc02b7000-0xffffffffc02e9000 NX
0xffffffffc02e9000-0xffffffffc02ea000 U
0xffffffffc02ea000-0xffffffffc02f0000 X
```

Compute the accuracy by comparing this with ground-truth page table entry data

Attack on Various OSes

- Type 2: Finer-grained module detection
  - Section-size Signature
    - Modules are allocated in fixed size of X/NX sections if the attacker knows the binary file
  - Example
    - If the size of executable map is 0x4000, and the size of non-executable section is 0x4000, then it is libahci!

```
//  BASE_ADDR      -      END_ADDR     PERM   NAME      SIZE
0xffffffffc035b000-0xffffffffc0360000 U
0xffffffffc0360000-0xffffffffc0364000 X  libahci    4000
0xffffffffc0364000-0xffffffffc0368000 NX libahci    4000
0xffffffffc0368000-0xffffffffc036c000 U
0xffffffffc036c000-0xffffffffc036e000 X  i2c_hid    2000
0xffffffffc036e000-0xffffffffc0371000 NX i2c_hid    3000
0xffffffffc0371000-0xffffffffc0376000 U
0xffffffffc0376000-0xffffffffc039a000 X  drm        24000
0xffffffffc039a000-0xffffffffc03cc000 NX drm        32000
0xffffffffc03cc000-0xffffffffc03cd000 U
```

Attack on Linux

- Processor
  - Intel Core i5-6300HQ (Skylake)
- OS Settings
  - Kernel 4.4.0, running with Ubuntu 16.04 LTS
  - Available Slots
    - Kernel: 64 slots
      - 0xffffffff80000000 – 0xffffffffc0000000 (2MB page)
    - Module: 1,024 slots
      - 0xffffffffc0000000 – 0xffffffffc0400000 (4KB page)

Demo 2: Full Attack on Linux

- Video Link
  - https://www.youtube.com/watch?v=WXGCylmAZkA

Result

- Achieved 100% accuracy across 3 different CPUs
  - Took 0.1-0.67s for probing 6,147 pages.
- Detecting Modules
  - From size signature, detected 38 modules among 105 modules.

Attack on Windows

- OS Settings
  - Windows 10, 10.0.10586
  - Available Slots
    - Kernel: 8,192 slots
      - 0xfffff80000000000 - 0xfffff80400000000 (2 MB pages)
    - Drivers: 8,192 slots
      - 0xfffff80000000000 - 0xfffff80400000000 (4 KB pages, aligned with 2 MB)

Result

- 100% of accuracy for the kernel (ntoskrnl.exe)
- 100% of accuracy for detecting M/U for the drivers (5 sec.)
- 99.28% of accuracy for detecting X/NX for drivers (45 sec.)
  - Some areas in driver are dynamically deallocated
  - Misses some ‘inactive’ pages
- Detecting Modules
  - From size signature, detected 97 drivers among 141 drivers

Attack on OS X

- OS Settings
  - OS X El Capitan 10.11.4
  - Available Slots
    - Kernel: 256 slots
      - 0xffffff8000000000 - 0xffffff8020000000 (2 MB pages)
  - Result
    - Took 31 ms on finding ASLR slide (100% accuracy for 10 times)

Attack on Amazon EC2

- X1 Instance of Amazon EC2
  - Processor: Intel Xeon E7-8880 v3 (Haswell)
- OS Settings
  - Kernel 4.4.0, running with Ubuntu 14.04 LTS
  - Available Slots
    - Kernel: 64 slots
      - 0xffffffff80000000 – 0xffffffffc0000000 (2MB page)
    - Module: 1,024 slots
      - 0xffffffffc0000000 – 0xffffffffc0400000 (4KB page)

Result Summary

- Linux: 100% of accuracy around 0.5 second
- Windows: 100% for M/U in 5 sec, 99.28% for X/NX for 45 sec
- OS X: 100% for detecting ASLR slide, in 31ms
- Linux on Amazon EC2: 100% of accuracy in 3 seconds

Timing Side Channel (M/U)

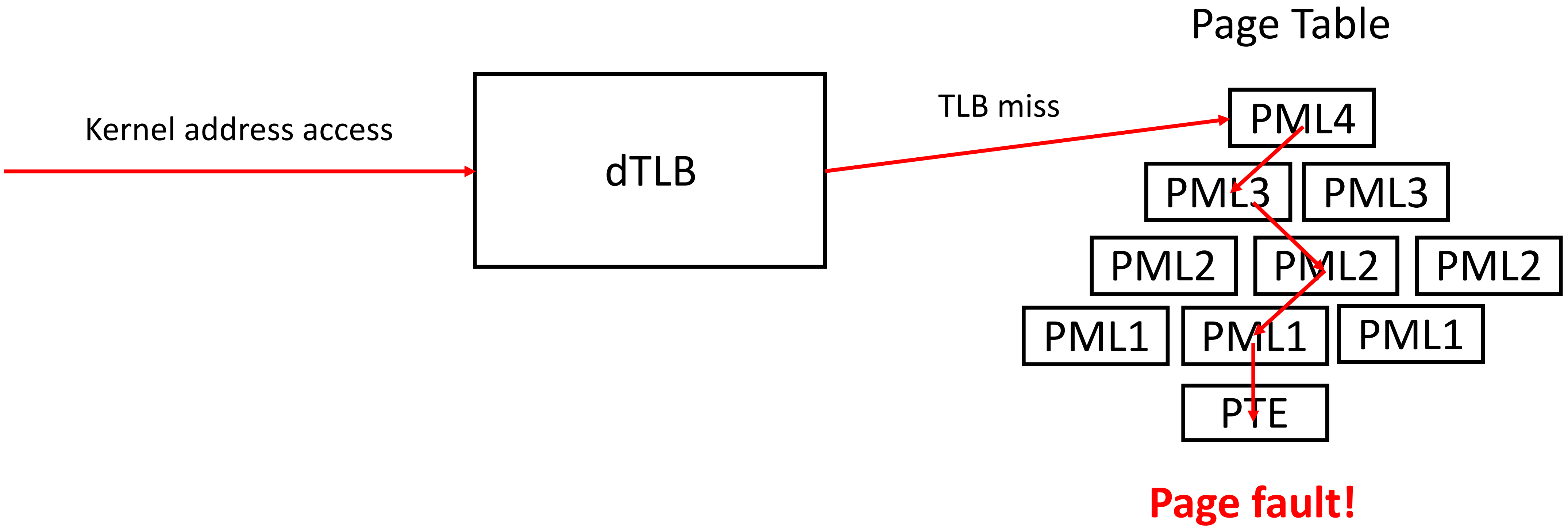
- For Mapped / Unmapped addresses
  - Measured performance counters (on 1,000,000 probing)

Perf. Counter	Mapped Page	Unmapped Page	Description
dTLB-loads	3,021,847	3,020,243	
dTLB-load-misses	84	2,000,086	TLB-miss on U
Observed Timing	209 (fast)	240 (slow)	

  - dTLB hit on mapped pages, but not for unmapped pages.
    - Timing channel is generated by dTLB hit/miss

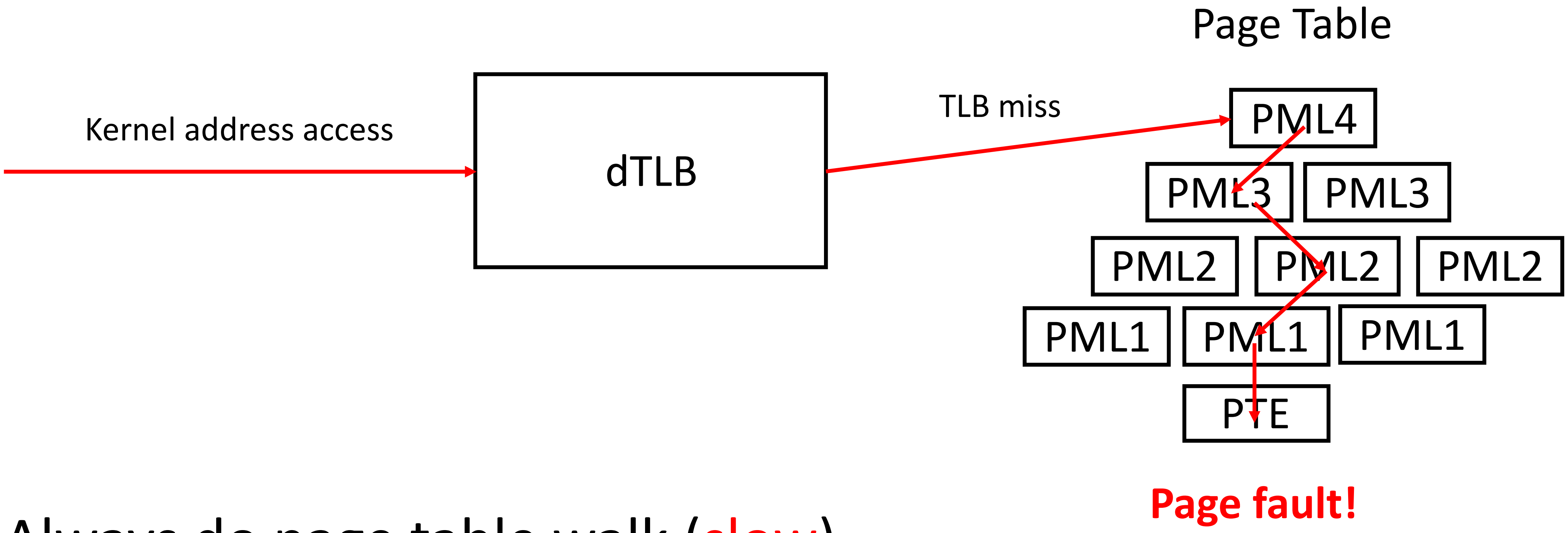
Path for an Unmapped Page

On the first access



Path for an Unmapped Page

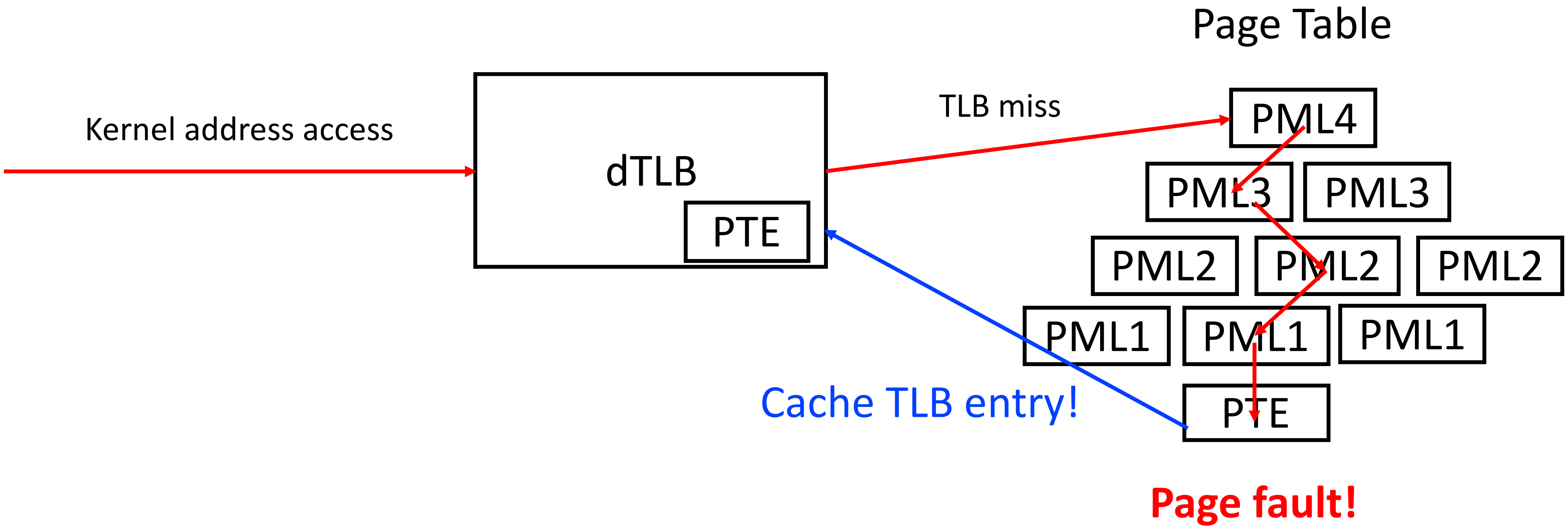
On the Second access



Always do page table walk (slow)

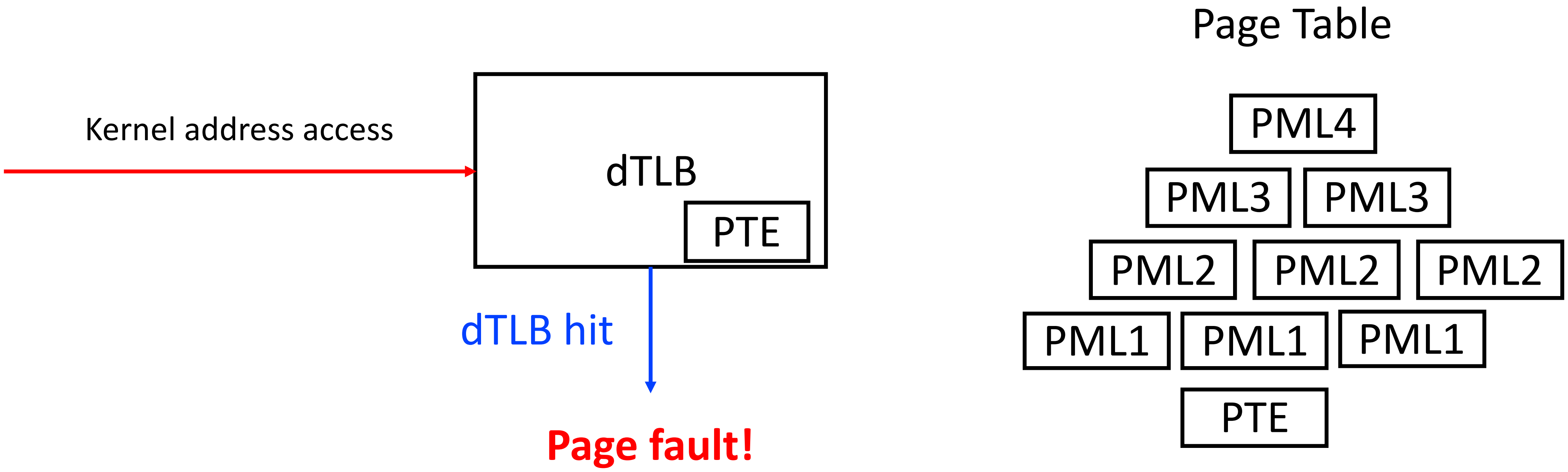
Path for a mapped Page

On the first access



Path for a mapped Page

On the second access



No page table walk on the second access (fast)

Root-cause of Timing Side Channels (M/U)

- For Mapped / Unmapped addresses

Fast Path (Mapped)	Slow Path (Unmapped)
1. Access a Kernel address 2. dTLB hits 3. Page fault!	1. Access a Kernel address 2. dTLB misses 3. Walks through page table 4. Page fault!
Elapsed cycles: 209	Elapsed cycles: 240

- Caching at dTLB generates timing side channel

Timing Side Channel (X/NX)

- For Executable / Non-executable addresses
  - Measured performance counters (on 1,000,000 probing)

Perf. Counter	Exec Page	Non-exec Page	Unmapped Page
iTLB-loads (hit)	590	1,000,247	272
iTLB-load-misses	31	12	1,000,175
Observed Timing	181 (fast)	226 (slow)	226 (slow)

  - Point #1: iTLB hit on Non-exec, but it is slow (226) why?
  - iTLB is not the origin of the side channel

Timing Side Channel (X/NX)

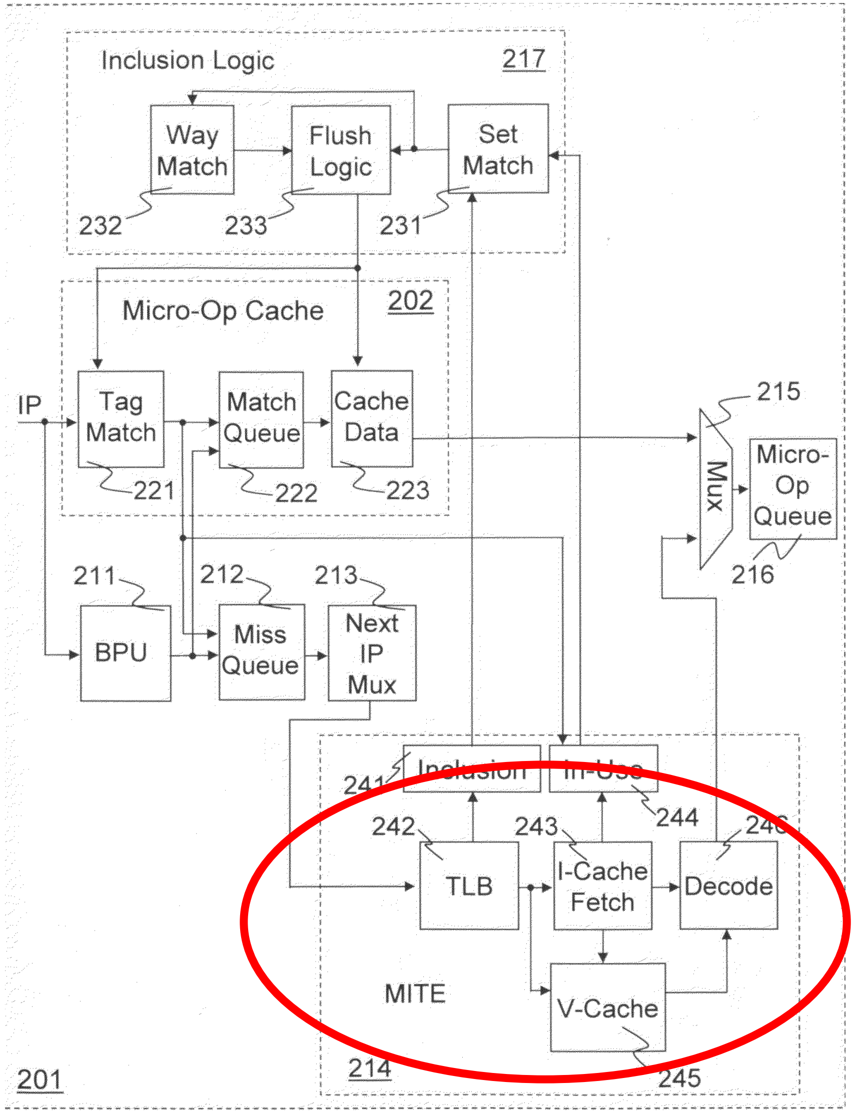
- For Executable / Non-executable addresses
  - Measured performance counters (on 1,000,000 probing)

Perf. Counter	Exec Page	Non-exec Page	Unmapped Page
iTLB-loads (hit)	590	1,000,247	272
iTLB-load-misses	31	12	1,000,175
Observed Timing	181 (fast)	226 (slow)	226 (slow)

  - Point #2: iTLB does not even hit on Exec page, while NX page hits iTLB
  - iTLB did not involve in the fast path

Intel Cache Architecture

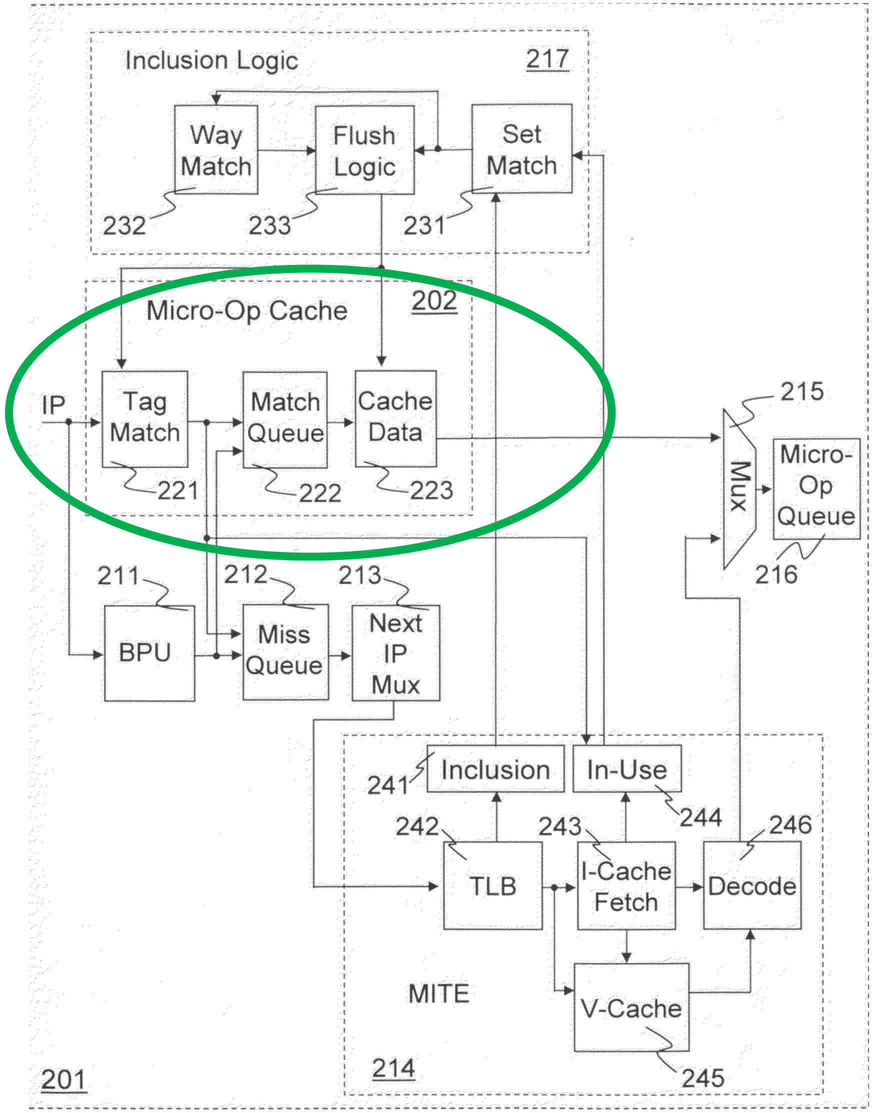
- L1 instruction cache
  - Virtually-indexed, Physically-tagged cache (requires TLB access)
  - Caches actual x86/amd64 opcode



From the patent **US 20100138608 A1**, registered by Intel Corporation

Intel Cache Architecture

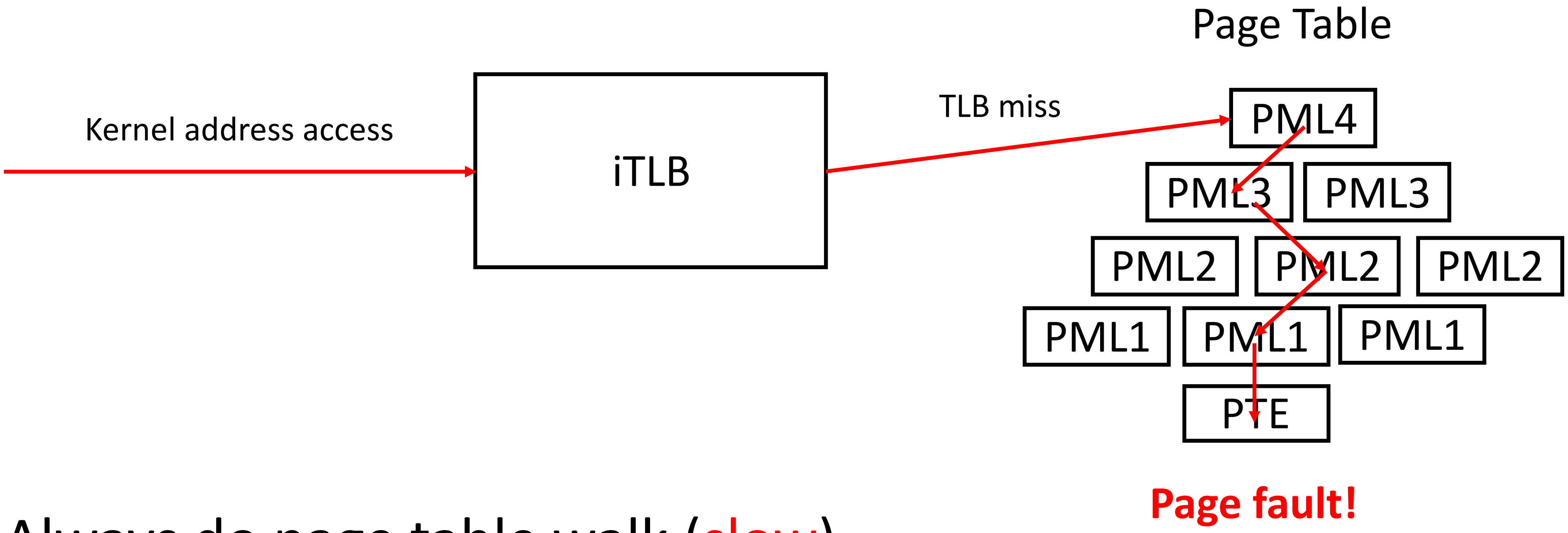
- Decoded i-cache
  - An instruction will be decoded as micro-ops (RISC-like instruction)
  - Decoded i-cache stores micro-ops
  - Virtually-indexed, Virtually-tagged cache (no TLB access)



From the patent **US 20100138608 A1**, registered by Intel Corporation

Path for an Unmapped Page

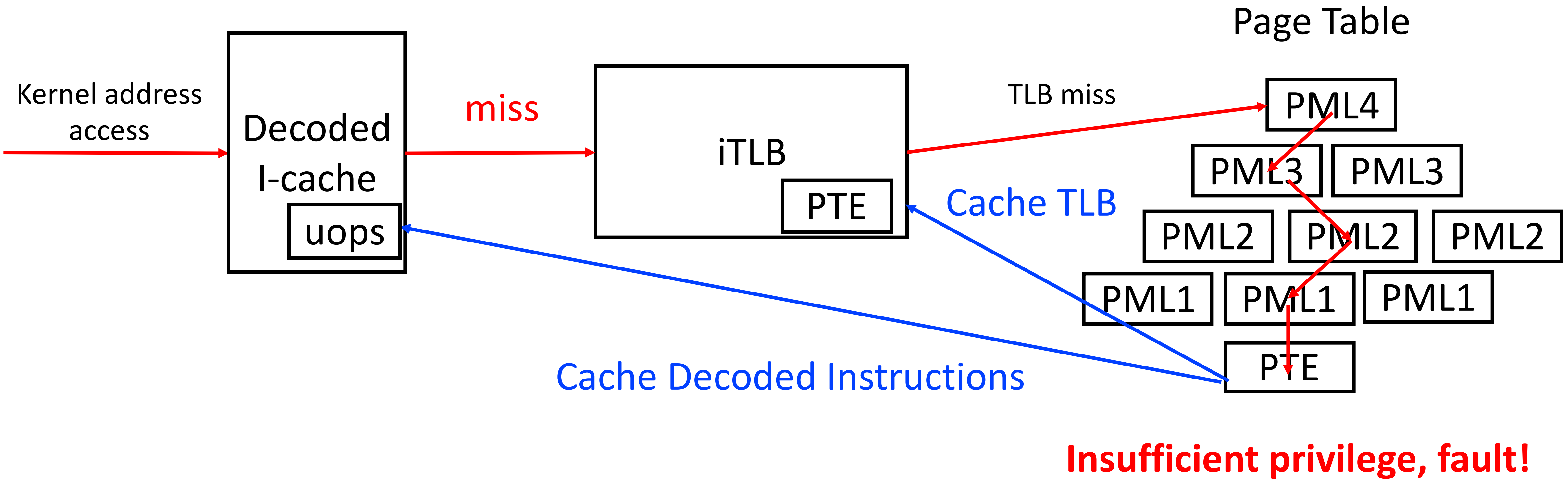
On the second access, 226 cycles



Always do page table walk (slow)

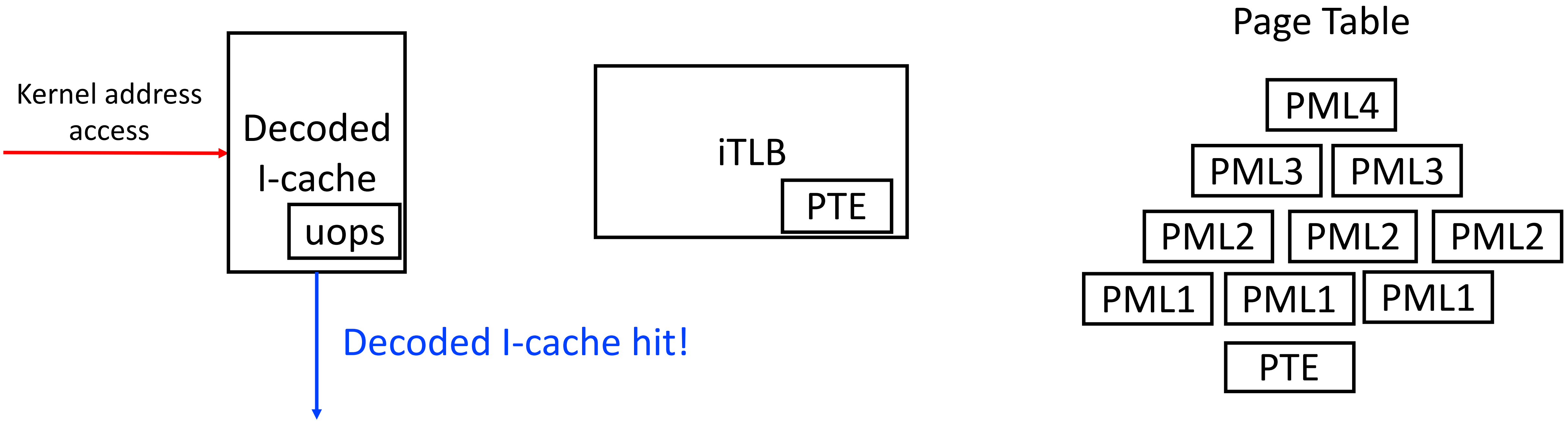
Path for an Executable Page

On the first access



Path for an Executable Page

On the second access, 181 cycles

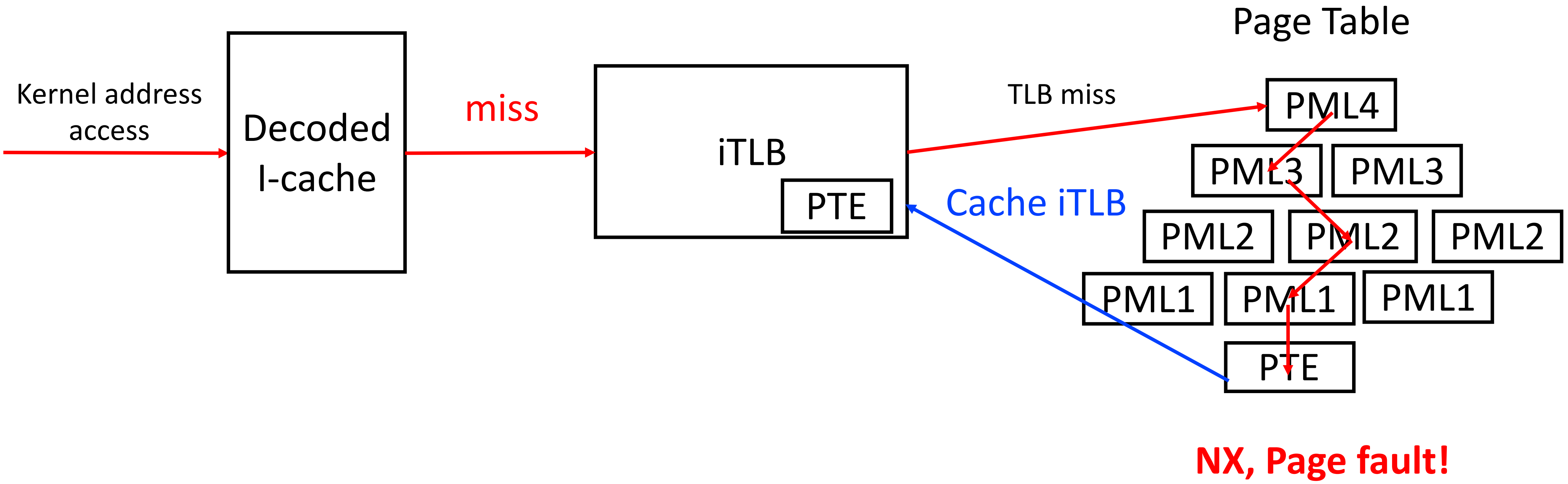


Insufficient privilege, fault!

No TLB access, No page table walk (fast)

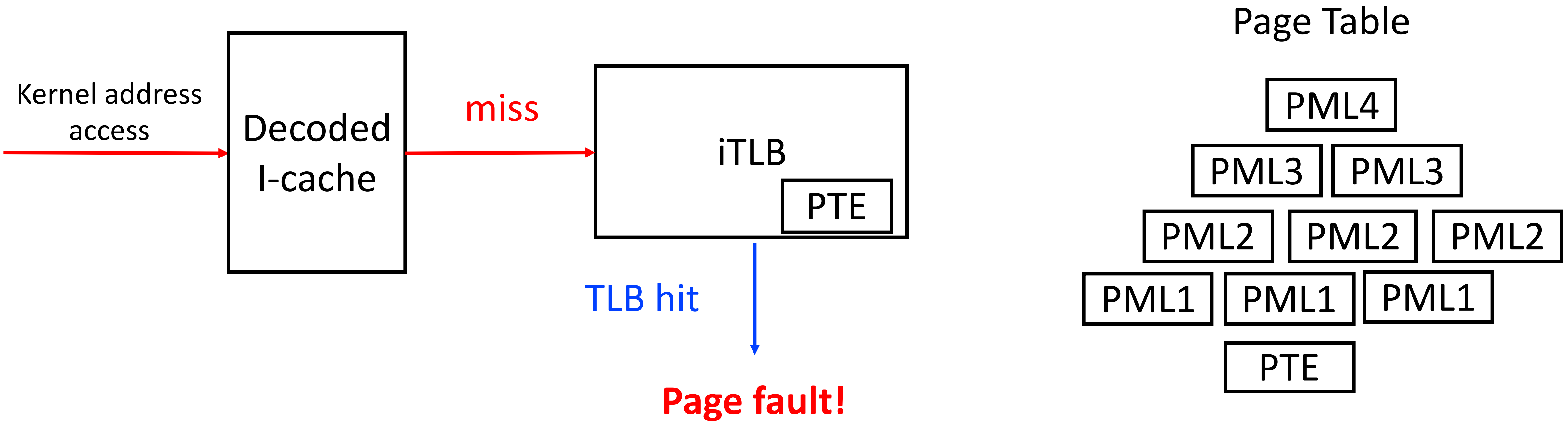
Path for a non-executable, but mapped Page

On the first access



Path for a Non-executable, but mapped Page

On the second access, 226 cycles



If no page table walk, it should be faster than unmapped (but not!)

Cache Coherence and TLB

- TLB is not a coherent cache in Intel Architecture

Core 1

TLB

Execute

0xff01->0x0010, NX

Core 2

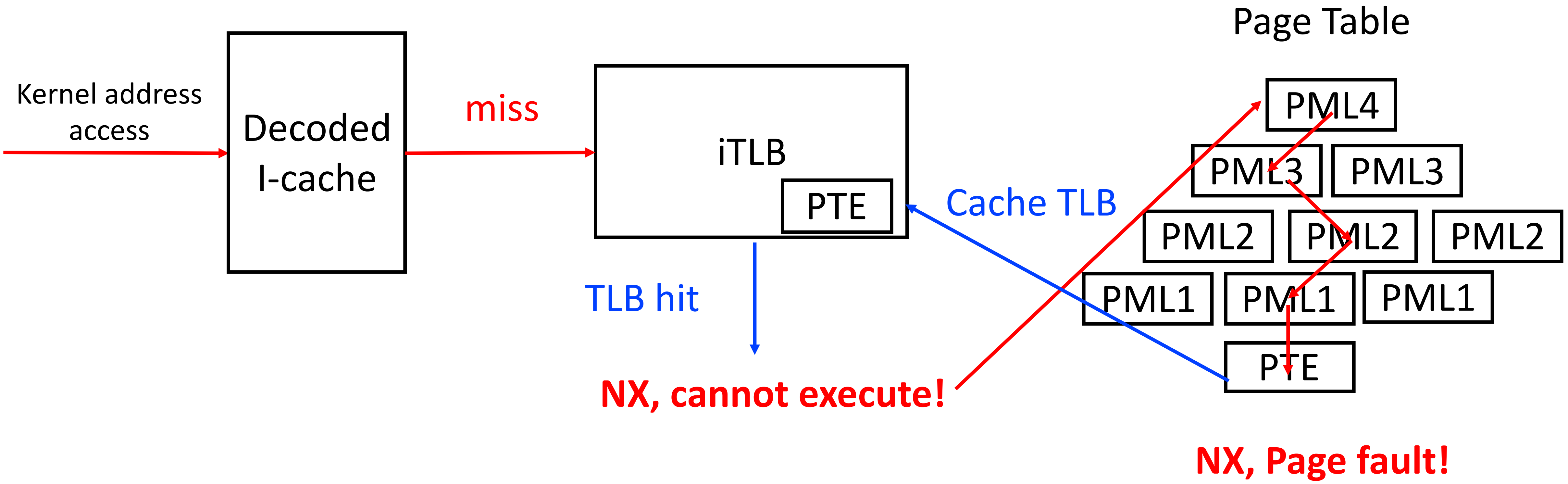
TLB

0xff01->0x0010, X

1. Core 1 sets 0xff01 as Non-executable memory
2. Core 2 sets 0xff01 as Executable memory
   No coherency, do not update/invalidate TLB in Core 1
3. Core 1 try to execute on 0xff01 -> fault by NX
4. Core 1 must walk through the page table
   The page table entry is X, update TLB, then execute!

Path for a Non-executable, but mapped Page

On the second access, 226 cycles



Root-cause of Timing Side Channel (X/NX)

- For executable / non-executable addresses

Fast Path (X)	Slow Path (NX)	Slow Path (U)
1. Jmp into the Kernel addr 2. Decoded I-cache hits 3. Page fault!	1. Jmp into the kernel addr 2. iTLB hit 3. Protection check fails, page table walk. 4. Page fault!	1. Jmp into the kernel addr 2. iTLB miss 3. Walks through page table 4. Page fault!
Cycles: 181	Cycles: 226	Cycles: 226

- Decoded i-cache generates timing side channel

Discussions: Controlling Noise

- Dynamic frequency scaling (SpeedStep, TurboBoost) changes the return value of rdtscp()
  - Run busy loops (while(1);) to make CPU run as full-throttle
- Hardware interrupts and cache conflicts also abort TSX
  - Probe multiple times (e.g., 2-100) and take the minimum

Discussions: Increasing Covertness

- OS never sees page faults
  - TSX suppresses the exception
- Possible traces: performance counters
  - High count on dTLB/iTLB-miss
    - Normal programs sequentially accessing huge memory could behave similarly
  - High count on tx-aborts or CPU time
    - Attackers could slow down the probing rate (e.g., 5 min, still fast)

Discussions: Countermeasures?

- Modifying CPU to eliminate timing channels
  - Difficult to be realized ☹
- Turning off TSX
  - Cannot be turned off in software manner (neither from MSR nor from BIOS)
- Coarse-grained timer?
  - Always suggested for timing side channel, but no one adopts it.

Discussions: Countermeasures?

- Using separated page tables for kernel and user processes
  - High performance overhead (~30%) due to frequent TLB flush
- Fine-grained randomization
  - Difficult to implement and performance degradation
- Inserting fake mapped / executable pages between the maps

Conclusion

- TSX can break KASLR of commodity OSes
  - Ensure accuracy, speed, and covertness
- Timing side channel is caused by hardware, independent to OS
  - dTLB (for Mapped & Unmapped)
  - Decoded i-cache (for eXecutable / non-executable)
- Current KASLR is not as secure as expected

Any Question?

- Q&A

TSX Support in Intel Processors

Grade/Generation	Skylake	Broadwell	Haswell
Server/Workstation	17/17 (100%)	19/19 (100%)	37/85 (43.5%)
High-end Consumer	23/38 (60.1%)	11/22 (50.0%)	2/92 (2.2%)
Low-end Consumer	4/32 (12.5%)	2/16 (12.5%)	0/79 (0%)

Prohibited Access to Kernel Address Space Layout

- OS X/iOS
  - Even root user has no access (**rootless**).
- Windows (**`NtQuerySystemInformation`**)
  - Sandbox process has no access (**low/untrusted integrity level**).
- Linux (kallsyms)
  - Non-root user has no access.