Breaking Kernel Address Space Layout Randomization (KASLR) with Intel TSX

Yeongjin Jang, Sangho Lee, and Taesoo Kim
Georgia Institute of Technology
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.
Example: Linux

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

```c
// full-nelson.c
static int __attribute__((regparm(3)))
getroot(void * file, void * vma)
{
    commit_creds(prepare_kernel_cred(0));
    return -1;
}
int privesc(struct sk_buff *skb, struct nlmsghdr *nlh)
{
    commit_creds(prepare_kernel_cred(0));
    return 0;
}
```
Example: Linux

- KASLR changes kernel symbol addresses every boot.
Example: Linux

• KASLR changes kernel symbol addresses every boot.

```
[blue9057@pt ~]$ sudo cat /proc/kallsyms | grep ' commit_creds\| prepare_kernel'
fffffffffaa0a3bd0 T commit_creds
fffffffffaa0a3fc0 T prepare_kernel_creds
```

1st Boot
Example: Linux

- KASLR changes kernel symbol addresses every boot.

```
[blue9057@pt ~]$ sudo cat /proc/kallsyms | grep ' commit_creds\| prepare_kernel'
fffffffffffa0a3bd0 T commit_creds
fffffffffffa0a3fc0 T prepare_kernel_cred
```

1st Boot

```
[blue9057@pt ~]$ sudo cat /proc/kallsyms | grep ' commit_creds\| prepare_kernel'
fffffffffbd0a3bd0 T commit_creds
fffffffffbd0a3fc0 T prepare_kernel_cred
```

2nd Boot
KASLR Makes Attacks Harder

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

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

• Both attackers and defenders aim to detect info leak vulnerabilities.
KASLR Makes Attacks Harder

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

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

\[ \Pr[ \exists \text{ information}_\text{leak} ] \times \Pr[ \exists \text{ Memory Corruption Vuln} ] \]

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

Virtual Address → TLB

Hit

Miss
TLB Timing Side Channel

Virtual Address → TLB

- Hit: Mapped address generate page fault quicker!
- Miss:
TLB Timing Side Channel

Virtual Address → TLB

Hit

Unmapped address takes ~40 cycles more for page table walk

Mapped address generate page fault quicker!

Unmapped address

Miss

Unmapped address takes ~40 cycles more for page table walk

Mapped address generate page fault quicker!
TLB Timing Side Channel

Virtual Address \(\rightarrow\) TLB \(\rightarrow\) Hit

Miss

Unmapped address takes ~40 cycles more for page table walk

Mapped address generate page fault quicker!
TLB Timing Side Channel

- **Virtual Address**
- **TLB**
- **Hit**
- **Miss**

Unmapped address takes ~40 cycles more for page table walk

- Mapped address generate page fault quicker!
TLB Timing Side Channel

Virtual Address \(\rightarrow\) TLB

Hit

Unmapped address takes \(~40\) cycles more for page table walk

Mapped address generate page fault quicker!
TLB Timing Side Channel

Virtual Address → TLB

Miss → Unmapped address takes ~40 cycles more for page table walk

Hit → Mapped address generate page fault quicker!
TLB Timing Side Channel

Virtual Address → TLB

Hit

Unmapped address takes ~40 cycles more for page table walk

Mapped address generate page fault quicker!
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

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

[Diagram showing mapped and unmapped virtual addresses with a horizontal line at 4050 cycles]
TLB Timing Side Channel

Measured Time (~4000 cycles)

Timing Side Channel (~40 cycles)

OS Noise (~100 cycles)

Fault Handling Noise is too much!
If we can *eliminate* the noise at OS, then the timing channel will be *more stable*.

Measured Time (~4000 cycles)

OS Noise (~100 cycles)

Fault Handling Noise is *too much*!
A More Practical Side Channel Attack on KASLR

• The DrK Attack: We present a 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
  • Distinguish mapped and unmapped pages
  • Distinguish executable and non-executable pages
Transactional Synchronization Extension (Intel TSX)

• TSX: relaxed but faster way of handling synchronization

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

    // atomic region
    try_atomic_operation();

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

    // if failed,
    handle_abort();
}
```
Transactional Synchronization Extension (Intel TSX)

- TSX: relaxed but faster way of handling synchronization

```c
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
Transactional Synchronization Extension (Intel TSX)

• TSX: relaxed but faster way of handling synchronization

```c
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)
Transaction Synchronization Extension (Intel TSX)

• TSX: relaxed but faster way of handling synchronization

```c
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

```c
int status = 0;
if (status == XBEGIN_STARTED) {
    // atomic region
    try_atomic_operation();
    _xend();
    // atomic region end
} else {
    // if failed,
    handle_abort();
}
```

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

*Run If Transaction Aborts*
Transaction Aborts If Exist any of a Conflict

```c
int status = 0;
if ( (status = _xbegin()) == _XBEGIN_STARTED) {
    // atomic region
    try_atomic_operation();
    _xend();
    // atomic region end
} else {
    // if failed,
    handle_abort();
}
```

- **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 OS exception handler)

Run If Transaction Aborts
Reducing Noise with Intel TSX

- User Execution
- CPU Exception
- TLB Side Channel
- OS Execution
- OS Handling Noise

Measured Time (~ 4000 cycles)
Reducing Noise with Intel TSX

**Measured Time (~ 4000 cycles)**

- User Execution
- CPU Exception
- TLB Side Channel (~40 cycles)
- OS Execution
- OS Handling Noise

**Measured Time (~ 180 cycles)**

- User Execution
- CPU Exception
- TLB Side Channel
- TLB Side Channel (~40 cycles)
- Not involving OS, Less noisy!
Exploiting TSX as an Exception Handler

• How to use TSX as an exception handler?

```c
uint64_t time_begin, time_diff;
int status = 0;
int *p = (int*)0xfffffffff8000000; // 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;
}
```
Exploiting TSX as an Exception Handler

• How to use TSX as an exception handler?

```c
uint64_t time_begin, time_diff;
int status = 0;
int *p = (int*)0xffffffff80000000; // kernel address

time_begin = __rdtscp();
if((status = __xbegin()) == XBEGIN_STARTED) {
    // TSX transaction
    *p; // read access
    // or,
    ((int(*)(void))p)(); // exec access
}
else {
    // abort handler
    time_diff = __rdtscp() - time_begin;
}
```
Exploiting TSX as an Exception Handler

• How to use TSX as an exception handler?

```c
uint64_t time_begin, time_diff;
int status = 0;
int *p = (int*)0xfffffffff80000000; // kernel address

time_begin = __rdtscp();
if((status = __xbegin()) == __XBEGIN_STARTED) {
    // TSX transaction
    *p; // read access
    // or,
    ((int(*)(void))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)
Exploiting TSX as an Exception Handler

• How to use TSX as an exception handler?

```c
uint64_t time_begin, time_diff;
int status = 0;
int *p = (int*)0xffffffff80000000; // kernel address

// Timestamp at the beginning
time_begin = __rdtscp();

if((status = __xbegin()) == __XBEGIN_STARTED) {
    // TSX transaction
    *p; // read access
    // or,
    ((int(*)(void))p)(0); // 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)
3. Measure timing at abort handler
Exploiting TSX as an Exception Handler

• How to use TSX as an exception handler?

```c
uint64_t time_begin, time_diff;
int status = 0;
int *p = (int*)0xffffffff80000000; // kernel address

time_begin = __rdtscp();
if((status = __xbegin()) == __XBEGIN_STARTED) {
    // TSX transaction
    *p; // read access
    // or,
    ((int (*)(void))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)
3. Measure timing at abort handler

Processor directly calls the handler
OS handling path is *not* involved
Measuring Timing Side Channel

• Mapped / Unmapped kernel addresses (across 4 CPUs)
  • Ran 1000 iterations for the probing, minimum clock on 10 runs

<table>
<thead>
<tr>
<th>Processor</th>
<th>Mapped Page</th>
<th>Unmapped Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-6700K (4.0Ghz)</td>
<td>209</td>
<td>240 (+31)</td>
</tr>
<tr>
<td>i5-6300HQ (2.3Ghz)</td>
<td>164</td>
<td>188 (+24)</td>
</tr>
<tr>
<td>i7-5600U (2.6Ghz)</td>
<td>149</td>
<td>173 (+24)</td>
</tr>
<tr>
<td>E3-1271v3 (3.6Ghz)</td>
<td>177</td>
<td>195 (+18)</td>
</tr>
</tbody>
</table>

• Mapped page always faults faster
Measuring Timing Side Channel

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

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

• Executable page always faults faster
Clear Timing Channel

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 macOS

• Attack Types
  • Type 1: Revealing mapping status of each page (X / NX / U)
  • Type 2: Finer-grained module detection
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
```
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!
Result Summary

- Linux: 100% of accuracy around 0.1 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)

<table>
<thead>
<tr>
<th>Perf. Counter</th>
<th>Mapped Page</th>
<th>Unmapped Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dTLB-loads</td>
<td>3,021,847</td>
<td>3,020,243</td>
<td></td>
</tr>
<tr>
<td>dTLB-load-misses</td>
<td>84</td>
<td>2,000,086</td>
<td>TLB-miss on U</td>
</tr>
<tr>
<td>Observed Timing</td>
<td>209 (fast)</td>
<td>240 (slow)</td>
<td></td>
</tr>
</tbody>
</table>

• dTLB hit on mapped pages, but not for unmapped pages.
  • Timing channel is generated by dTLB hit/miss
Timing Side Channel (M/U)

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

<table>
<thead>
<tr>
<th>Perf. Counter</th>
<th>Mapped Page</th>
<th>Unmapped Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dTLB-loads</td>
<td>3,021,847</td>
<td>3,020,243</td>
<td></td>
</tr>
<tr>
<td>dTLB-load-misses</td>
<td></td>
<td>84</td>
<td>2,000,086 TLB-miss on U</td>
</tr>
<tr>
<td>Observed Timing</td>
<td>209 (fast)</td>
<td>240 (slow)</td>
<td></td>
</tr>
</tbody>
</table>

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

Probing an unmapped page took 240 cycles
Path for an Unmapped Page

Probing an unmapped page took 240 cycles

Kernel address access

Page Table

PML4
PML3 PML3
PML2 PML2 PML2
PML1 PML1 PML1
PTE
Path for an Unmapped Page

Probing an unmapped page took **240** cycles
Path for an Unmapped Page

Probing an unmapped page took 240 cycles
Path for an Unmapped Page

Probing an unmapped page took 240 cycles

Always do page table walk (slow)
Path for a mapped Page

On the first access, 240 cycles
Path for a mapped Page

On the first access, 240 cycles
Path for a mapped Page

On the first access, 240 cycles
Path for a mapped Page

On the first access, **240 cycles**

Kernel address access → dTLB

Page Table

PML4
PML3
PML3
PML2
PML2
PML2
PML1
PML1
PML1
PTE

TLB miss

Page fault!
Path for a mapped Page

On the first access, 240 cycles

Kernel address access → dTLB → Page Table

PTE

PML4
PML3 PML3
PML2 PML2 PML2
PML1 PML1 PML1
PTE

TLB miss → Cache TLB entry!

Page fault!
Path for a mapped Page

On the second access, 209 cycles
Path for a mapped Page

On the second access, 209 cycles
Path for a mapped Page

On the second access, 209 cycles

Kernel address access → dTLB

PML4 → PML3 → PML3 → PML2 → PML2 → PML2 → PML1 → PML1 → PML1 → PTE

Page Table

dTLB hit → dTLB

Page fault!
Path for a mapped Page

On the second access, 209 cycles

No page table walk on the second access (fast)
Timing Side Channel (X/NX)

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

<table>
<thead>
<tr>
<th>Perf. Counter</th>
<th>Exec Page</th>
<th>Non-exec Page</th>
<th>Unmapped Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>iTLB-loads (hit)</td>
<td>590</td>
<td>1,000,247</td>
<td>272</td>
</tr>
<tr>
<td>iTLB-load-misses</td>
<td>31</td>
<td>12</td>
<td>1,000,175</td>
</tr>
<tr>
<td>Observed Timing</td>
<td>181 (fast)</td>
<td>226 (slow)</td>
<td>226 (slow)</td>
</tr>
</tbody>
</table>

• 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)

<table>
<thead>
<tr>
<th>Perf. Counter</th>
<th>Exec Page</th>
<th>Non-exec Page</th>
<th>Unmapped Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>iTLB-loads (hit)</td>
<td>590</td>
<td>1,000,247</td>
<td>272</td>
</tr>
<tr>
<td>iTLB-load-misses</td>
<td>31</td>
<td>12</td>
<td>1,000,175</td>
</tr>
<tr>
<td>Observed Timing</td>
<td>181 (fast)</td>
<td>226 (slow)</td>
<td>226 (slow)</td>
</tr>
</tbody>
</table>

• 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)

<table>
<thead>
<tr>
<th>Perf. Counter</th>
<th>Exec Page</th>
<th>Non-exec Page</th>
<th>Unmapped Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>iTLB-loads (hit)</td>
<td>590</td>
<td>1,000,247</td>
<td>272</td>
</tr>
<tr>
<td>iTLB-load-misses</td>
<td>31</td>
<td>12</td>
<td>1,000,175</td>
</tr>
<tr>
<td>Observed Timing</td>
<td>181 (fast)</td>
<td>226 (slow)</td>
<td>226 (slow)</td>
</tr>
</tbody>
</table>

- Point #2: iTLB does not even hit on Exec page, while NX page hits iTLB
- iTLB did not involve in the fast path
  - Is there any cache that does not require address translation?
Intel Cache Architecture

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

- L1 instruction cache
  - Virtually-indexed, Physically-tagged cache (requires TLB access)
  - Caches actual x86/x64 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
Path for an Unmapped Page

On the second access, **226** cycles

![Diagram of the Path for an Unmapped Page]

- Kernel address access
- iTLB

Page Table:
- PML4
- PML3
- PML3
- PML2
- PML2
- PML2
- PML1
- PML1
- PML1
- PTE
Path for an Unmapped Page

On the second access, 226 cycles
Path for an Unmapped Page

On the second access, 226 cycles
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

Decoded I-cache

iTLB

Page Table

PML4
PML3
PML3
PML2
PML2
PML2
PML1
PML1
PML1
PTE
Path for an Executable Page

On the first access

Kernel address access

Decoded I-cache

iTLB

Page Table

PML4
PML3  PML3
PML2  PML2  PML2
PML1  PML1  PML1
PTE
Path for an Executable Page

On the first access

Kernel address access → Decoded I-cache → miss → iTLB

Page Table

PML4
PML3  PML3
PML2  PML2  PML2
PML1  PML1  PML1
PTE
Path for an Executable Page

On the first access

Kernel address access

Decoded I-cache

miss

iTLB

TLB miss

Page Table

PML4

PML3  PML3

PML2  PML2  PML2

PML1  PML1  PML1

PTE

access

TLB miss

Decoded

I-cache

miss

iTLB

TLB miss

Page Table

PML4

PML3  PML3

PML2  PML2  PML2

PML1  PML1  PML1

PTE
Path for an Executable Page

On the first access

Kernel address access → Decoded I-cache (miss) → iTLB → Page Table

Insufficient privilege, fault!
Path for an Executable Page

On the first access

Kernel address access → Decoded I-cache → miss → iTLB → miss → TLB miss → Cache TLB → Insufficient privilege, fault!

Page Table

PML4 → PML3 → PML3 → PML2 → PML2 → PML2 → PML1 → PML1 → PML1 → PTE
Path for an Executable Page

On the first access

Kernel address access → Decoded I-cache uops → miss

iTLB → PTE

TLB miss → Page Table

PML4 → PML3 → PML3 → PML2 → PML2 → PML2 → PML1 → PML1 → PML1 → PTE

Insufficient privilege, fault!
Path for an Executable Page

On the second access, **181** cycles

Decoded I-cache uops

iTLB

Page Table

PML4

PML3  PML3

PML2  PML2  PML2

PML1  PML1  PML1

PTE
Path for an Executable Page

On the second access, **181** cycles

Kernel address access

Decoded I-cache uops

iTLB

Page Table

PML4

PML3   PML3

PML2   PML2   PML2

PML1   PML1   PML1

PTE
Path for an Executable Page

On the second access, 181 cycles

Kernel address access

Decoded I-cache uops

Decoded I-cache hit!

Insufficient privilege, fault!

Page Table

- PML4
- PML3
- PML3
- PML2
- PML2
- PML2
- PML1
- PML1
- PML1
- PTE

iTLB

PTE
Path for an Executable Page

On the second access, 181 cycles

Decoded I-cache uops

Decoded I-cache hit!

Insufficient privilege, fault!

No TLB access, No page table walk (fast)
Path for a non-executable, but mapped Page

On the second access, 226 cycles

Decoded I-cache

iTLB

PTE

Page Table

PML4

PML3

PML3

PML2

PML2

PML2

PML1

PML1

PML1

PTE
Path for a non-executable, but mapped Page

On the second access, **226** cycles
Path for a non-executable, but mapped Page

On the second access, 226 cycles

Kernel address access → Decoded I-cache → miss → iTLB → PTE

Page Table

PML4
PML3 PML3
PML2 PML2 PML2
PML1 PML1 PML1
PTE
Path for a non-executable, but mapped Page

On the second access, 226 cycles

Kernel address access → Decoded I-cache → miss → iTLB → TLB hit → Page fault!

Page Table:
- PML4
- PML3
- PML3
- PML2
- PML2
- PML2
- PML1
- PML1
- PML1
- PTE
Path for a non-executable, but mapped Page

On the second access, **226** cycles

[Diagram showing the path through the page table structure:]

- Kernel address access
- Decoded I-cache (miss)
- iTLB (miss)
- TLB hit
- Page fault!

Page Table:
- PML4
- PML3 PML3
- PML2 PML2 PML2
- PML1 PML1 PML1
- PTE

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
Cache Coherence and TLB

- TLB is not a coherent cache in Intel Architecture

<table>
<thead>
<tr>
<th>Core 1</th>
<th>1. Core 1 sets 0xff01 as Non-executable memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>0xff01→0x0010, NX</td>
</tr>
</tbody>
</table>
Cache Coherence and TLB

• TLB is not a coherent cache in Intel Architecture

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
Cache Coherence and TLB

- TLB is not a coherent cache in Intel Architecture

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
Cache Coherence and TLB

• TLB is not a coherent cache in Intel Architecture

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

Decoded I-cache

iTLB

Page Table

PML4

PML3  PML3

PML2  PML2  PML2

PML1  PML1  PML1

PTE

PTE
Path for a Non-executable, but mapped Page

On the second access, 226 cycles

Kernel address access → Decoded I-cache → iTLB → Page Table

Page Table:
- PML4
- PML3
- PML3
- PML2
- PML2
- PML2
- PML1
- PML1
- PML1
- PTE

93
Path for a Non-executable, but mapped Page

On the second access, **226 cycles**
Path for a Non-executable, but mapped Page

On the second access, 226 cycles

Kernel address access → Decoded I-cache → miss → iTLB → miss → TLB hit → NX, cannot execute!

Page Table

PML4

PML3  PML3

PML2  PML2  PML2

PML1  PML1  PML1

PTE
Path for a Non-executable, but mapped Page

On the second access, **226** cycles

Kernel address access → Decoded I-cache → miss

Decoded I-cache → iTLB

iTLB → TLB hit

NX, cannot execute!

Page Table

PML4 → PML3 → PML3 → PML2 → PML2 → PML2 → PML1 → PML1 → PML1 → PTE
Path for a Non-executable, but mapped Page

On the second access, 226 cycles

Kernel address access → Decoded I-cache → iTLB → TLB hit → NX, cannot execute!

Page Table:
- PML4
- PML3
- PML2
- PML1
- PTE

Cache TLB:
- NX, Page fault!
Root-cause of Timing Side Channel (X/NX)

- For executable / non-executable addresses

<table>
<thead>
<tr>
<th>Fast Path (X)</th>
<th>Slow Path (NX)</th>
<th>Slow Path (U)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Jmp into the Kernel addr</td>
<td>1. Jmp into the kernel addr</td>
<td>1. Jmp into the kernel addr</td>
</tr>
<tr>
<td>2. Decoded I-cache hits</td>
<td>2. iTLB hit</td>
<td>2. iTLB miss</td>
</tr>
</tbody>
</table>

Cycles: 181 | Cycles: 226 | Cycles: 226

- **Decoded i-cache** generates timing side channel
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?
  • A workaround could be having another thread to measure the timing indirectly (e.g., counting i++;)
Countermeasures?

• Using separated page tables for kernel and user processes
  • High performance overhead (~30%) due to frequent TLB flush
    • TLB flush on every copy_to_user()

• Fine-grained randomization
  • Compatibility issues on memory alignment, etc.

• Inserting fake mapped / executable pages between the maps
  • Adds some false positives to the DrK Attack
Conclusion

• Intel TSX makes cache side-channel less noisy
  • Suppress OS Exception

• Timing side channel can distinguish X / NX / U pages
  • dTLB (for Mapped & Unmapped)
  • Decoded i-cache (for executable / non-executable)
  • Work across 3 different architectures, commodity OSes, and Amazon EC2

• Current KASLR is not as secure as expected
Any Questions?

• Try DrK at
  • [https://github.com/sslab-gatech/DrK](https://github.com/sslab-gatech/DrK)