SOSCON

# Micro-architectural attack and defense on Linux kernel

## Spectre and Meltdown

Samsung Research | Security team | Jinbum Park | jinb-park.github.io
Samsung Research | Security team | Joonwon Kang

16-Oct-2019

# Index



# Overview

SOSCON 2019

SAMSUNG OPEN SOURCE CONFERENCE 2019

# Overview



**Meltdown?**

**Spectre?**

**Intel hardware vulnerabilities?**

**Micro-architecture attacks?**

# Software - Architecture - Micro-architecture



## Software

```
int *ptr = …;

void func(void)
{
    int a = *ptr;
}
```

## Architecture (Intel / ARM …)

```
mov %ebx, (%eax)

// eax == ptr
```

## Micro-architecture (i7-1, i7-2, i5-3, A12, …)

```
pa = convert_va_to_pa(%eax);
read pa;
```

# Software - Architecture - Micro-architecture



## Software

```
int *ptr = NULL;

void func(void)
{
    int a = *ptr;
}
```

## Architecture (Intel / ARM …)

```
mov %ebx, (%eax)

// eax == ptr (NULL)
```

## Micro-architecture (i7-1, i7-2, i5-3, A12, …)

```
pa = convert_va_to_pa(%eax);
if (is_valid_address(pa))
    read pa;
else
    error;
```

Segmentation fault (core dumped)

# Software - Architecture - Micro-architecture



## Software

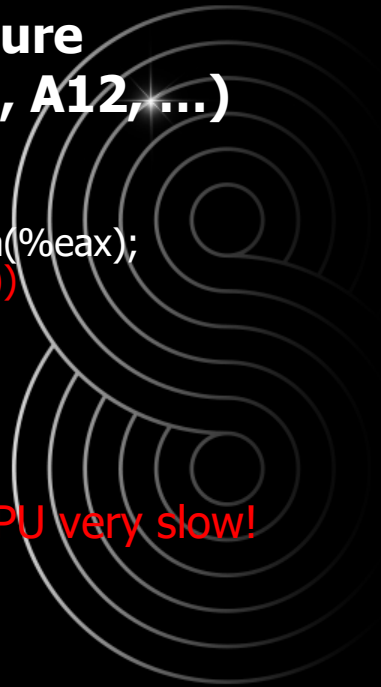
```
int *ptr = &obj;

void func(void)
{
    int a = *ptr;
}
```

## Architecture (Intel / ARM …)

```
mov %ebx, (%eax)

// eax == ptr (NULL)
```

## Micro-architecture (i7-1, i7-2, i5-3, A12, …)

```
pa = convert_va_to_pa(%eax);
if (is_valid_address(pa))
    read pa;
else
    error;
```



Checking on every memory access makes CPU very slow!

How can we make it better?

# Software - Architecture - Micro-architecture



## Software

```
int *ptr = &obj;

void func(void)
{
    int a = *ptr;
}
```

## Architecture (Intel / ARM …)

```
mov %ebx, (%eax)

// eax == ptr (NULL)
```

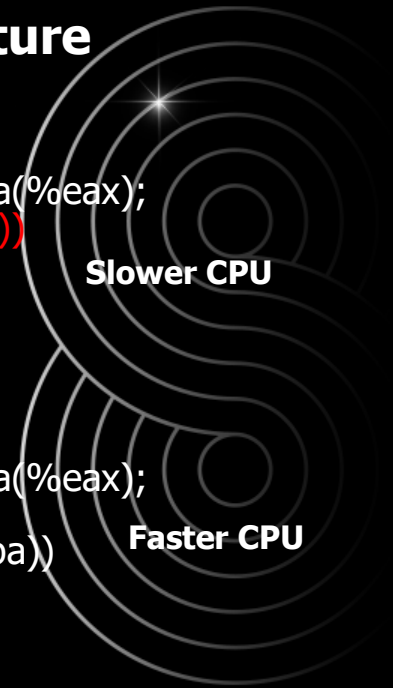
## Micro-architecture

```
pa = convert_va_to_pa(%eax);
if (is_valid_address(pa))
    read pa;
else
    error;
```

Slower CPU

```
pa = convert_va_to_pa(%eax);
read pa;
if (is_invalid_address(pa))
    error;
```

Faster CPU



Speculate “pa” is not NULL
Read “pa” first!
It called “Speculative execution”

# Software - Architecture - Micro-architecture



## Software

```
int *ptr = &obj;

void func(void)
{
    int a = *ptr;
}
```

## Architecture (Intel / ARM …)

```
mov %ebx, (%eax)

// eax == ptr (NULL)
```

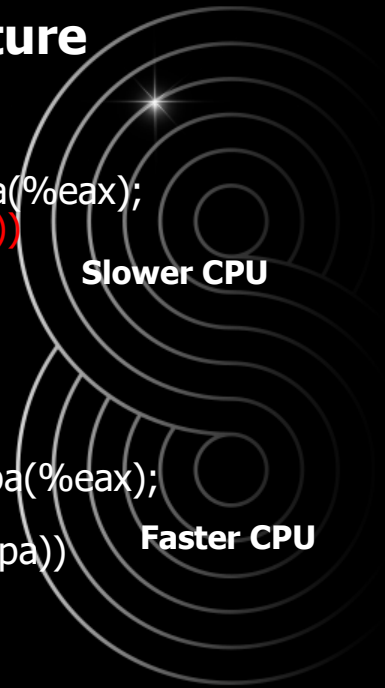
## Micro-architecture

```
pa = convert_va_to_pa(%eax);
if (is_valid_address(pa))
    read pa;
else
    error;
```

Slower CPU

```
pa = convert_va_to_pa(%eax);
read pa;
if (is_invalid_address(pa))
    error;
```

Faster CPU



**Security implication in it**
**:: Even though it occurs a fault,**
**We can read an invalid memory before the fault!**

# Software - Architecture - Micro-architecture



## Software

```
int *ptr = &obj;

void func(void)
{
    int a = *ptr;
}
```

## Architecture (Intel / ARM …)

```
mov %ebx, (%eax)

// eax == ptr (NULL)
```

## Micro-architecture

```
pa = convert_va_to_pa(%eax);
if (is_valid_address(pa))
    read pa;
else
    error;
```

Slower CPU

```
pa = convert_va_to_pa(%eax);
read pa;
if (is_invalid_address(pa))
    error;
```

Faster CPU

If you can exploit this implementation,
we call this "Hardware Vulnerability"
Call the attack on this "Micro-architecture attack"

# Vulnerable Micro-architectures



**Intel Desktop CPUs Affected By Meltdown + Spectre**

## Intel Desktop CPUs Vulnerable To Meltdown + Spectre

**Affected Variants :** These Intel CPUs are affected by **all three variants** of the speculative execution CPU bug. They are vulnerable to the Meltdown and both Spectre exploits.

**Contents**



**Intel Coffee Lake-S (2017)**

- Intel Core i7-8700K
- Intel Core i7-8700
- Intel Core i5-8600K
- Intel Core i5-8400
- Intel Core i3-8350K
- Intel Core i3-8100

**Intel Gemini Lake (2017)**

- Intel Pentium Silver J5005
- Intel Celeron J4105
- Intel Celeron J4005

**Intel Denverton (2017)**

- Intel Celeron C3958
- Intel Celeron C3955

# What we'll discuss



**4 Hardware Vulnerabilities**

- **Spectre Variant1, Variant2, Meltdown, L1TF**

**How to exploit them for user to read kernel-memory?**

**How does Linux kernel defend against them?**

# Spectre Attack & Defense

SOSCON 2019

SAMSUNG OPEN SOURCE CONFERENCE 2019

# 2. Attack 1: Out-of-bounds attack



## Intuition

```c
#include <stdio.h>
#include <stdlib.h>

char *secret = "This is victim's secret.";
char ary1[10];

int main(int argc, char **argv)
{
    int idx = atoi(argv[1]);

    char var = ary1[idx];

    return 0;
}
```

| address | value |
|---|---|
| 100 | 'T' |
| 101 | 'h' |
| ... | ... |
| 200 | ary1[0] |

**Q. What if execute it with idx = -100 ?**

A. var == 'T'

**Can access out of ary with the attacker chosen index without crash!**

# 2. Attack 1: Out-of-bounds attack (Cont.)



## Attack steps

```c
#include <stdio.h>
#include <stdlib.h>

extern char ary2[];

char *secret = "This is victim's secret.";
char ary1[10];

int main(int argc, char **argv)
{
    int idx = atoi(argv[1]);

    char var = ary2[ary1[idx]];

    return 0;
}
```

Shared memory between processes.

**Goal: What is in victim's ary1[-100] ?**

# 2. Attack 1: Out-of-bounds attack (Cont.)



**Attacker**

1. Flush ary2[0...255].
2. Set idx = -100 and Execute the victim.
5. Access ary2[0...255].
6. Find idx which takes the shortest time.
**Found! ary1[-100] == 'T'.**

Called "Flush+Reload" analysis.

**Cache**

| |
|---|
| ... |
| ary2['T'] |
| ... |

4. Cached!

**Shared memory**

| |
|---|
| ary2[0] |
| ... |
| ary2['T'] |
| ... |
| ary2[255] |

**Victim**

3. **var = ary2[ary1[-100]]**

| address | variable |
|---|---|
| **100** | **'T'** |
| 101 | 'h' |
| ... | ... |
| 200 | ary1[0] |

# 2. Attack 1: Out-of-bounds attack (Cont.)



**How can we defend against this attack?**

```c
#include <stdio.h>
#include <stdlib.h>

extern char ary2[];

char *secret = "This is victim's secret.";
char ary1[10];

int main(int argc, char **argv)
{
    int idx = atoi(argv[1]);

    char var = ary2[ary1[idx]];

    return 0;
}
```

**Let’s check bounds here!**

# 3. Defense 1: Bounds check



**Check bounds before the gadget.**

```c
#include <stdio.h>
#include <stdlib.h>

extern char ary2[];

char *secret = "This is victim's secret.";
char ary1[10];
int size = 10;

int main(int argc, char **argv)
{
    int idx = atoi(argv[1]);
    char var;

    if (idx < size)
        var = ary2[ary1[idx]];

    return 0;
}
```

**Out-of-bounds attack is blocked!**

**Really..?**

# 4. Attack 2: Bounds check bypass (Spectre v1)



## Can we bypass bounds check?

### Software

```
if (idx < size)
    var = ary2[ary1[idx]];
```

**Nope…** →

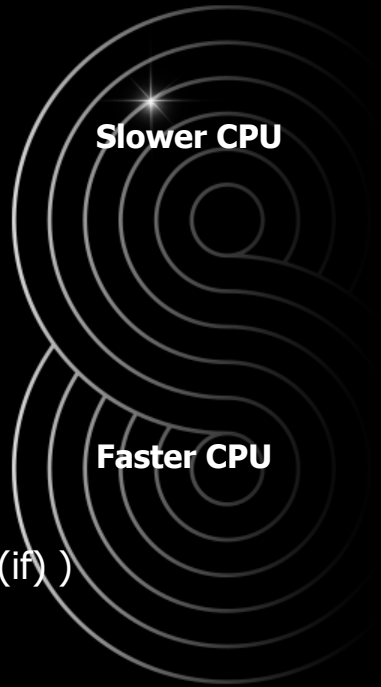
```
read idx
read size
res = idx < size
if ( res )
    read ary2[ary1[idx]]
```

**Yes! but we have to make** →

**likely_taken(if) == true.**

```
if ( likely_taken(if) )
    read ary2[ary1[idx]]
read idx
read size
res = idx < size
if ( res == likely_taken(if) )
    commit
else
    revert
...
```

### Micro-architecture



# 4. Attack 2: Bounds check bypass (Cont.)



## Attack steps

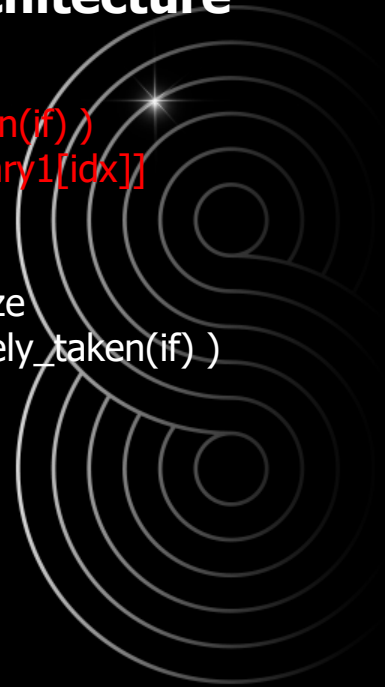
## Software

```
if (idx < size)
    var = ary2[ary1[idx]];
```

**Goal: What is in victim's ary1[-100] ?**

## Micro-architecture

```
if ( likely_taken(if) )
    read ary2[ary1[idx]]
read idx
read size
res = idx < size
if ( res == likely_taken(if) )
    commit
else
    revert
...
```



# 4. Attack 2: Bounds check bypass (Cont.)



## Attack steps

### Software

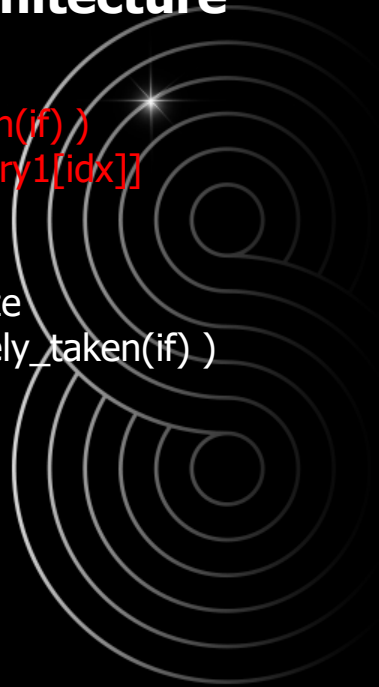
```
if (idx < size)
    var = ary2[ary1[idx]];
```

1. Execute victim with legitimate idx multiple times.

⟶ **Conditional branch predictor will learn that likely_taken(if) == true.**

### Micro-architecture

```
if ( likely_taken(if) )
    read ary2[ary1[idx]]
read idx
read size
res = idx < size
if ( res == likely_taken(if) )
    commit
else
    revert
...
```



# 4. Attack 2: Bounds check bypass (Cont.)



## Attack steps

### Software

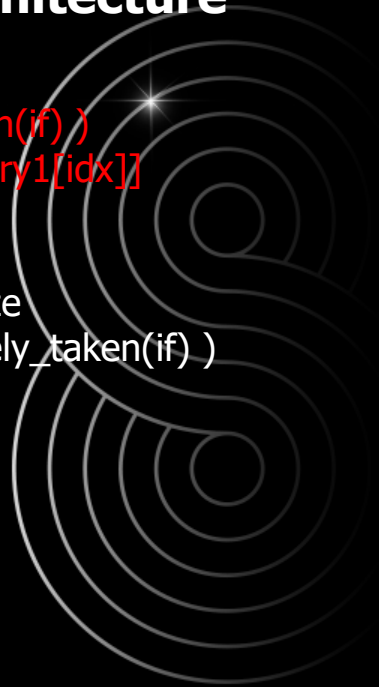
```
if (idx < size)
    var = ary2[ary1[idx]];
```

2. Execute victim with idx = -100.

→ **ary2[ary1[-100]] will be read and cached!**

### Micro-architecture

```
if ( likely_taken(if) )
    read ary2[ary1[idx]]
read idx
read size
res = idx < size
if ( res == likely_taken(if) )
    commit
else
    revert
...
```



# 4. Attack 2: Bounds check bypass (Cont.)



## Attack steps

## Software

```
if (idx < size)
    var = ary2[ary1[idx]];
```

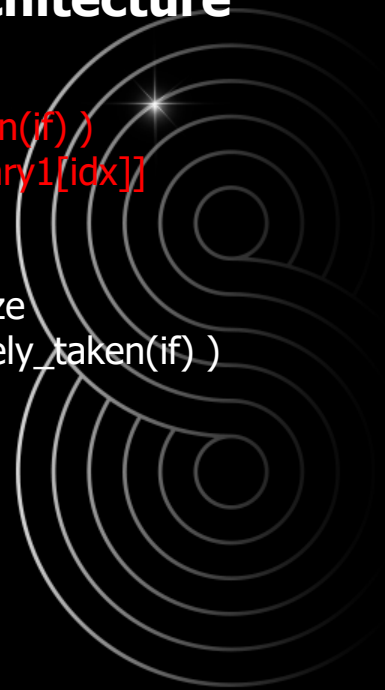
3. Do Flush+Reload analysis on ary2.

⟶ **ary2['T'] will take the shortest time.**

**Thus, ary1[-100] == 'T' !!**

## Micro-architecture

```
if ( likely_taken(if) )
    read ary2[ary1[idx]]
read idx
read size
res = idx < size
if ( res == likely_taken(if) )
    commit
else
    revert
...
```



# 4. Attack 2: Bounds check bypass (Cont.)



**How can we defend against this attack?**

## Software

```
if (idx < size)
    var = ary2[ary1[idx]];
```

**Let's sanitize idx here!**

## Micro-architecture

```
if ( likely_taken(if) )
    read ary2[ary1[idx]]
read idx
read size
res = idx < size
if ( res == likely_taken(if) )
    commit
else
    revert
...
```

# 5. Defense 2: Index sanitization



**How can we defend against this attack?**

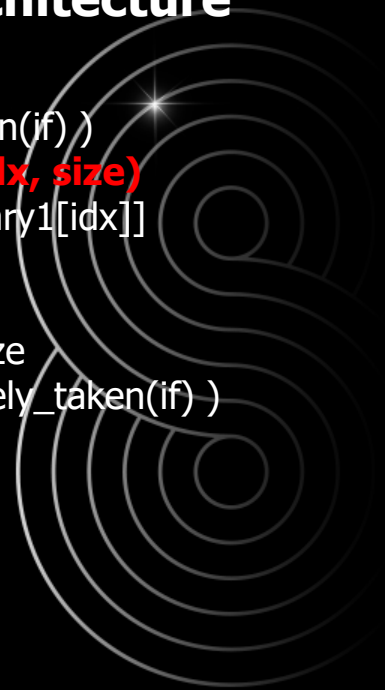
## Software

```
if (idx < size)
    sanitize(idx, size)
    var = ary2[ary1[idx]];
```

**Clamp idx in [0, size).**

## Micro-architecture

```
if ( likely_taken(if) )
    sanitize(idx, size)
    read ary2[ary1[idx]]
read idx
read size
res = idx < size
if ( res == likely_taken(if) )
    commit
else
    revert
...
```



# 5. Defense 2: Index sanitization



## Example: idx sanitization in Linux kernel.

```c
static inline struct file *__fcheck_files(struct files_struct *files, unsigned int fd)
{
    struct fdtable *fdt = rcu_dereference_raw(files->fdt);

    if (fd < fdt->max_fds) {
        fd = array_index_nospec(fd, fdt->max_fds);
        return rcu_dereference_raw(fdt->fd[fd]);
    }
    return NULL;
}
```

```c
static struct perf_event *ptrace_hbp_get_event(unsigned int note_type,
                                               struct task_struct *tsk,
                                               unsigned long idx)
{
    struct perf_event *bp = ERR_PTR(-EINVAL);

    switch (note_type) {
    case NT_ARM_HW_BREAK:
        if (idx >= ARM_MAX_BRP)
            goto out;
        idx = array_index_nospec(idx, ARM_MAX_BRP);
        bp = tsk->thread.debug.hbp_break[idx];
        break;
    case NT_ARM_HW_WATCH:
        if (idx >= ARM_MAX_WRP)
            goto out;
        idx = array_index_nospec(idx, ARM_MAX_WRP);
        bp = tsk->thread.debug.hbp_watch[idx];
        break;
```

```c
/*
 * array_index_nospec - sanitize an array index after a bounds check
 */
#define array_index_nospec(index, size)                 \
({                                                      \
    typeof(index) _i = (index);                         \
    typeof(size) _s = (size);                           \
    unsigned long _mask = array_index_mask_nospec(_i, _s);      \
                                                        \
    BUILD_BUG_ON(sizeof(_i) > sizeof(long));            \
    BUILD_BUG_ON(sizeof(_s) > sizeof(long));            \
                                                        \
    (typeof(_i)) (_i & _mask);                          \
})
```

linux/include/linux/nospec.h

# 5. Defense 2: Index sanitization



**How can we defense against this attack?**

## Software

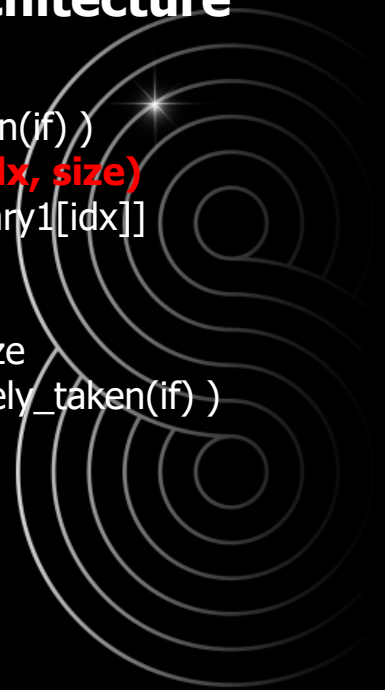
```
if (idx < size)
    sanitize(idx, size)
    var = ary2[ary1[idx]];
```

**Out-of-bounds attack is blocked!**

**You sure…?**

## Micro-architecture

```
if ( likely_taken(if) )
    sanitize(idx, size)
    read ary2[ary1[idx]]
read idx
read size
res = idx < size
if ( res == likely_taken(if) )
    commit
else
    revert
...
```



# 6. Attack 3: Indirect branch poisoning (Spectre v2)



**Can we execute the gadget only?**

## Software

```
if (idx < size)
    sanitize(idx, size)
    var = ary2[ary1[idx]];
```

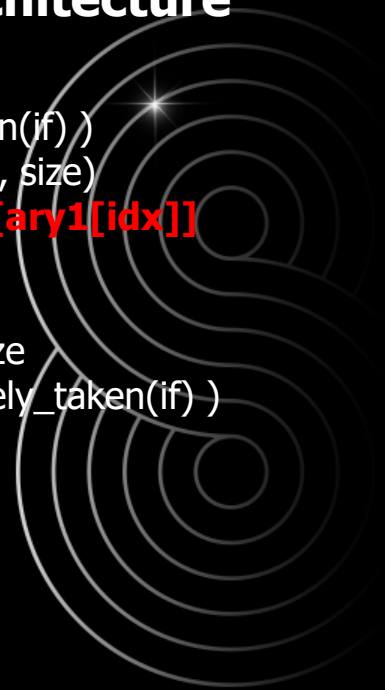
**Yes, if we can jump to the gadget like ROP.**

**But.. How can we make jump?**

**Maybe through function pointer?**

## Micro-architecture

```
if ( likely_taken(if) )
    sanitize(idx, size)
    read ary2[ary1[idx]]
read idx
read size
res = idx < size
if ( res == likely_taken(if) )
    commit
else
    revert
...
```



# 6. Attack 3: Indirect branch poisoning (Cont.)



## Can we jump to the gadget?

### Software

```
void (*fp)(void);

void func(void)
{
    fp();          ← VA: 10
}
```

**Nope.** →

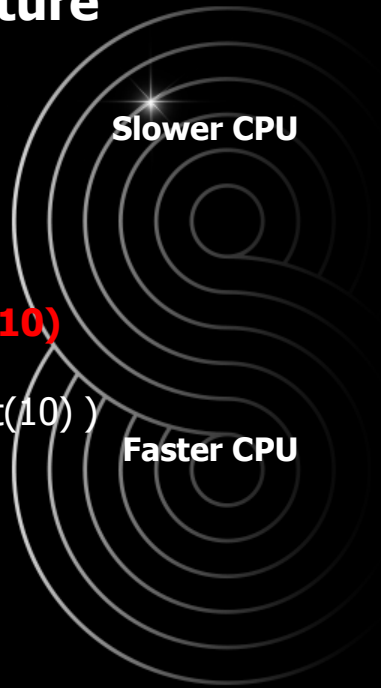
### Micro-architecture

```
read fp
jump fp
```

**Yes! but we have to make**

**likely_target(10) == gadget addr.**

```
jump likely_target(10)
read fp
if ( fp == likely_target(10) )
    commit
else
    revert
    jump fp
```



# 6. Attack 3: Indirect branch poisoning (Cont.)



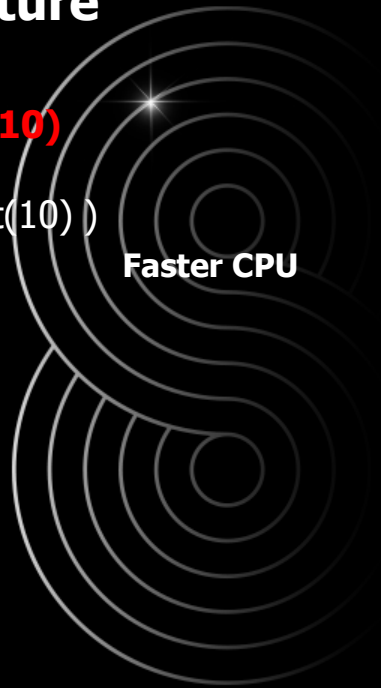
## Attack steps

### Software

```
void (*fp)(void);

void func(void)
{
    fp();            ← VA: 10
}
```

### Micro-architecture

```
jump likely_target(10)
read fp
if ( fp == likely_target(10) )
    commit
else
    revert
    jump fp
```




**Goal: What is in victim's ary1[-100] ?**

# 6. Attack 3: Indirect branch poisoning (Cont.)



## Attack steps

### Victim

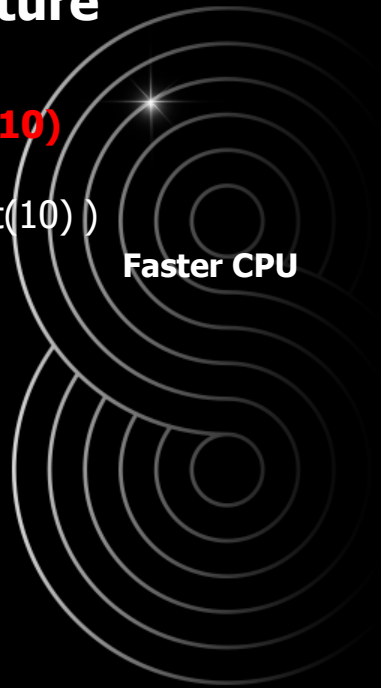
```
void (*fp)(void);

void func(void)
{
    fp();       ← VA: 10
}
```

### Attacker

```
void (*fp)(void);

void func(void)
{
    jump gadget;  ← VA: 10
}
```

### Micro-architecture

```
jump likely_target(10)
read fp
if ( fp == likely_target(10) )
    commit
else
    revert
    jump fp
```




1. Allocate the same memory as victim's and
   Modify fp() to **jump gadget**.

# 6. Attack 3: Indirect branch poisoning (Cont.)



## Attack steps

### Victim

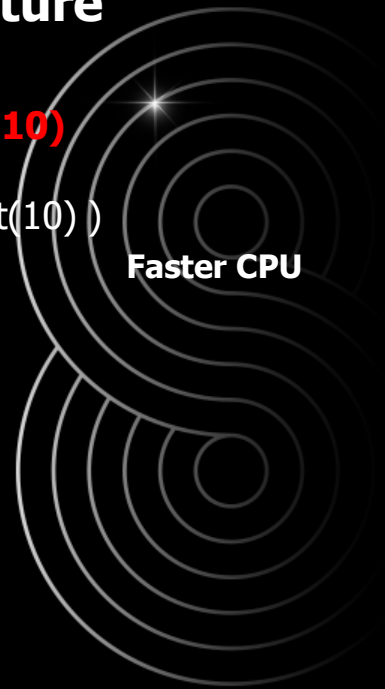
```
void (*fp)(void);

void func(void)
{
    fp();       ← VA: 10
}
```

### Attacker

```
void (*fp)(void);

void func(void)
{
    jump gadget;  ← VA: 10
}
```

### Micro-architecture

```
jump likely_target(10)
read fp
if ( fp == likely_target(10) )
    commit
else
    revert
    jump fp
```




2. Execute the attacker's program multiple times.

⟶ **Indirect branch predictor will learn that**

**likely_target(10) == gadget addr.**

# 6. Attack 3: Indirect branch poisoning (Cont.)



## Attack steps

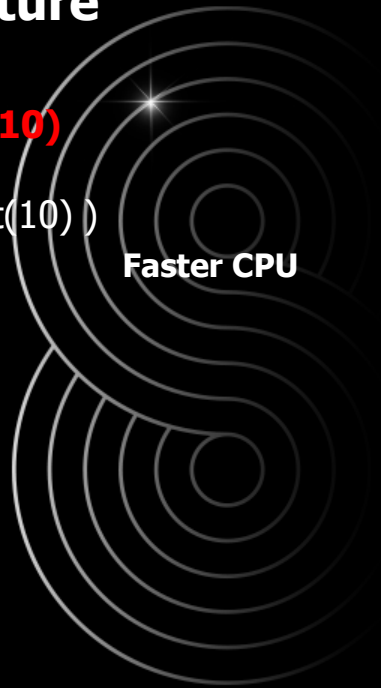
### Victim

**ary2[ary1[-100]] is executed!**

```
void (*fp)(void);

void func(void)
{
    fp();          ← VA: 10
}
```

### Micro-architecture

```
jump likely_target(10)
read fp
if ( fp == likely_target(10) )
    commit
else
    revert
    jump fp
```




3. Execute the victim program with idx = -100.

→ **ary2[ary1[-100]] will be read and cached!**

# 6. Attack 3: Indirect branch poisoning (Cont.)



## Attack steps

### Victim

```
void (*fp)(void);

void func(void)
{
    fp();            ← VA: 10
}
```

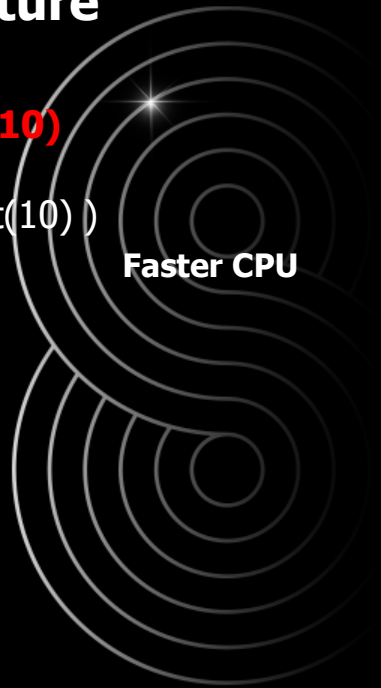
4. Do Flush+Reload analysis on ary2.

⟶ **ary2['T'] will take the shortest time.**

**Thus, ary1[-100] == 'T' !!**

### Micro-architecture

```
jump likely_target(10)
read fp
if ( fp == likely_target(10) )
    commit
else
    revert
    jump fp
```



# 6. Attack 3: Indirect branch poisoning (Cont.)



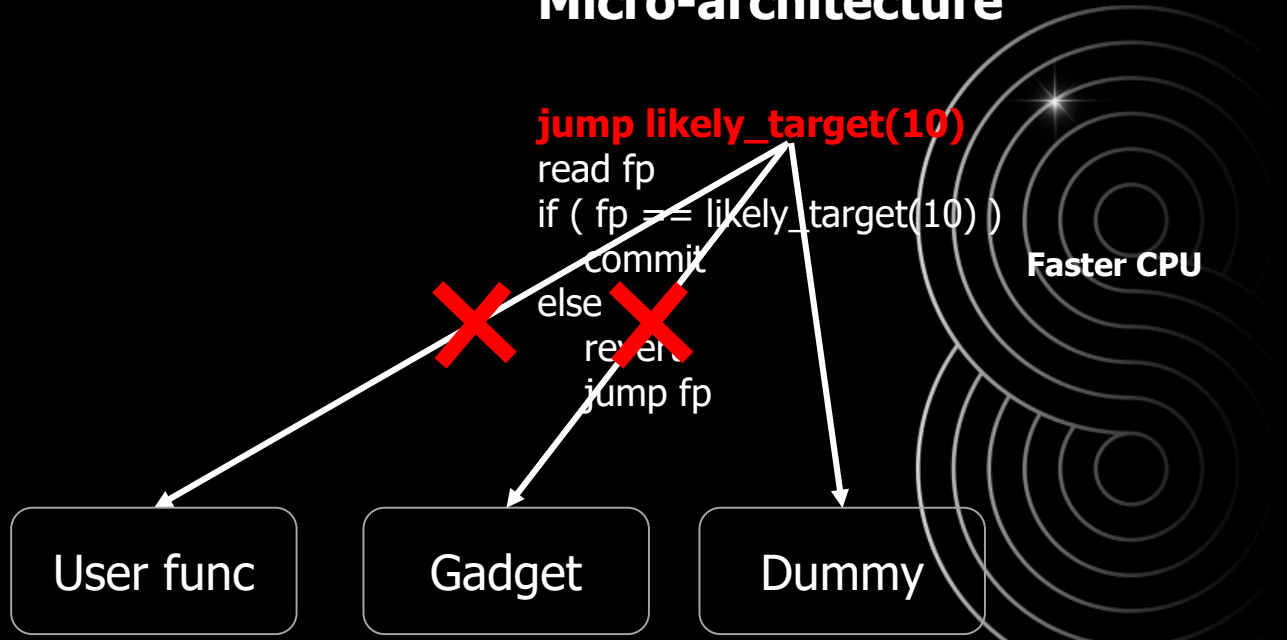
**How can we defend against this attack?**

## Victim

```
void (*fp)(void);

void func(void)
{
    fp();          ← VA: 10
}
```

## Micro-architecture

**jump likely_target(10)**
read fp
if ( fp == likely_target(10) )
    commit
else
    re:ei
    jump fp




**How about making it like this?**

# 7. Defense 3: Retpoline



**Indirect call replacement with retpoline.**

```
jmp *%rax  ⟶   call load_label
capture_ret_spec:
               pause ; lfence
               jmp capture_ret_spec
load_label:
               mov %rax, (%rsp)
               ret
```

# 7. Defense 3: Retpoline (Cont.)



**Goal: Capture speculative execution for indirect branch into a dummy loop.**

Details are hard to understand for beginner.

To get more details, please refer to

- Appendix-D: Retpoline details
- intel's guidance

# 7. Defense 3: Retpoline (Cont.)



**Example: Retpoline in Linux kernel.**

```
config RETPOLINE
    bool "Avoid speculative indirect branches in kernel"
    default y
    select STACK_VALIDATION if HAVE_STACK_VALIDATION
    help
      Compile kernel with the retpoline compiler options to guard against
      kernel-to-user data leaks by avoiding speculative indirect
      branches. Requires a compiler with -mindirect-branch=thunk-extern
      support for full protection. The kernel may run slower.
```

- linux/arch/x86/Kconfig

```
/*
 * JMP_NOSPEC and CALL_NOSPEC macros can be used instead of a simple
 * indirect jmp/call which may be susceptible to the Spectre variant 2
 * attack.
 */
.macro JMP_NOSPEC reg:req
#ifdef CONFIG_RETPOLINE
  ANNOTATE_NOSPEC_ALTERNATIVE
  ALTERNATIVE_2 __stringify(ANNOTATE_RETPOLINE_SAFE; jmp *\reg),  \
    __stringify(RETPOLINE_JMP \reg), X86_FEATURE_RETPOLINE, \
    __stringify(lfence; ANNOTATE_RETPOLINE_SAFE; jmp *\reg), X86_FEATURE_RETPOLINE_AMD
#else
  jmp *\reg
#endif
.endm
```

- linux/arch/x86/include/asm/nospec-branch.h

# Wrap up Spectre attacks



**For actual v1, v2 attacks, there are more prerequisites.**

- Additional cache miss for successful speculative execution.
- Control over specific registers.
- Run programs on the same core to share BTB.
- etc..

**To get more details, please refer to**

- Spectre Attacks: Exploiting Speculative Execution
- Google project zero blog

# Attack4: Meltdown

SOSCON 2019

SAMSUNG OPEN SOURCE CONFERENCE 2019

# Two phases of Meltdown



Goal : Reading kernel memory from user-space

1. Speculatively read kernel memory into register.
2. Stick the register to the leak gadget.

# Speculatively read kernel memory



## Software (User)

```
int *ptr = Kernel_Addr;

void func(void)
{
    int a = *ptr;
}
```

Segmentation fault!

What we expect

1. Segmentation Fault
2. Die or
   Run signal handler

Actually happens

1. int a = *ptr; (a == kernel value)
2. Segmentation Fault
3. a = 0;  (clear kernel value)
4. Die or Run signal handler

**We can read kernel-memory!!**

# Speculatively read kernel memory (Zoom in)



## Software

```
int *ptr = Kernel_Addr;

void func(void)
{
    int a = *ptr;
}
```

Vulnerable-to-Meltdown

## Micro-architecture

```
pa = convert_va_to_pa(ptr);
if (is_kernel_address(pa))
    error;
else
    read pa;
```

```
pa = convert_va_to_pa(ptr);
read pa;
if (is_kernel_address(pa))
    error;
```

**Let's turn our focus to this!**

# Speculatively read kernel memory (Zoom in more)



## Software

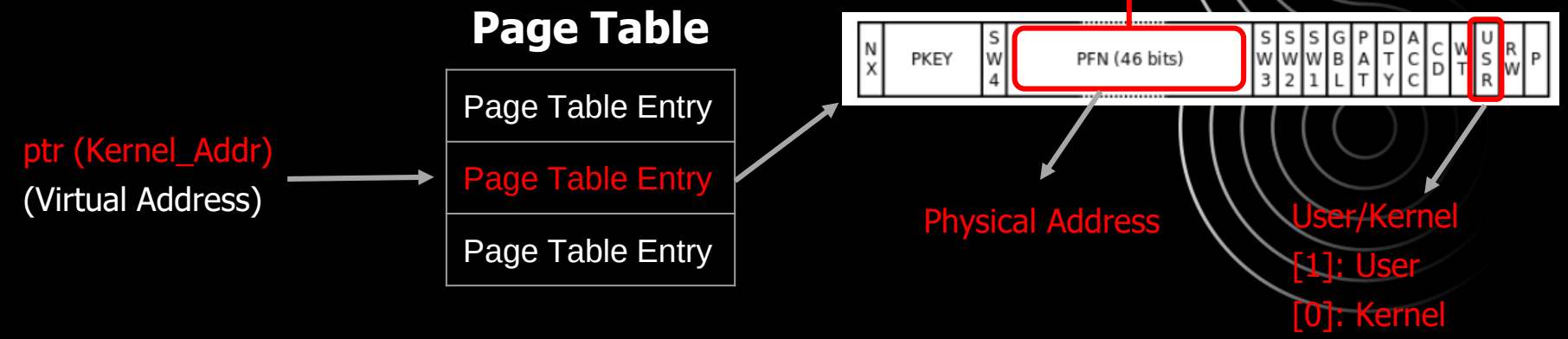
```
int *ptr = Kernel_Addr;

void func(void)
{
    int a = *ptr;
}
```

## Micro-architecture

(1) pa = convert_va_to_pa(ptr);

(2) read pa;

(3) if (is_kernel_address(pa))
    error;



# Stick it to the leak gadget



## Leak gadget + FLUSH-RELOAD + Meltdown = Kernel memory leak!

```
int *ptr = Kernel_Addr;

void func(void)
{
    int a = *ptr;        // (1) a = kernel-value
    var = ary1[a];       // (2) [leak gadget] kernel-value dependent access → Cache loading
    // (3) invalid access! fault!
}

void fault_handler(....)
{
    for (i=0; i<SIZE; i++) {
        if (time_to_access(ary[i]) < CONSTANT)
            // (4) i is kernel-value!;
    }
}
```

# Defense4: Page Table Isolation

SOSCON 2019

SAMSUNG OPEN SOURCE CONFERENCE 2019

# Micro-architecural fix



## Micro-architecture

### Software

```
int *ptr = Kernel_Addr;

void func(void)
{
    int a = *ptr;
}
```

Fix

```
pa = convert_va_to_pa(ptr);
if (is_kernel_address(pa))
    read pa;
else
    error;
```

```
pa = convert_va_to_pa(ptr);
read pa;
if (is_kernel_address(pa))
    error;
```

**(1) It's infeasible to update a lot of deployed CPUs..**

**(2) then.. what is the root cause in the view of software?**

**(3) Figure out solution to fix Meltdown with software fix! (Linux kernel)**

# Software root cause



## Software

```
int *ptr = Kernel_Addr;

void func(void)
{
    int a = *ptr;
}
```

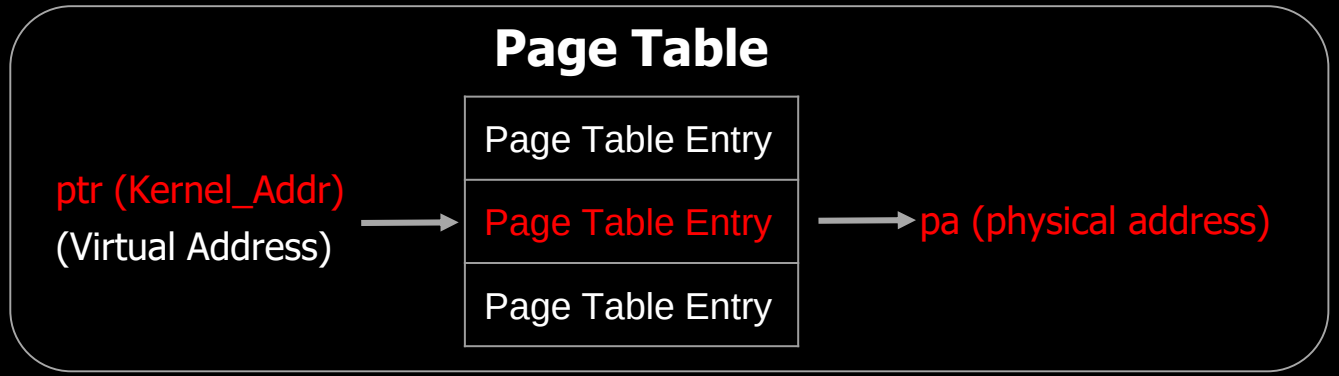
## Micro-architecture

(1) pa = convert_va_to_pa(ptr);

(2) read pa;

(3) if (is_kernel_address(pa))
    error;

**Why does it allow User to access Kernel??**




# Software root cause



## Software

```
int *ptr = Kernel_Addr;

void func(void)
{
    int a = *ptr;
}
```

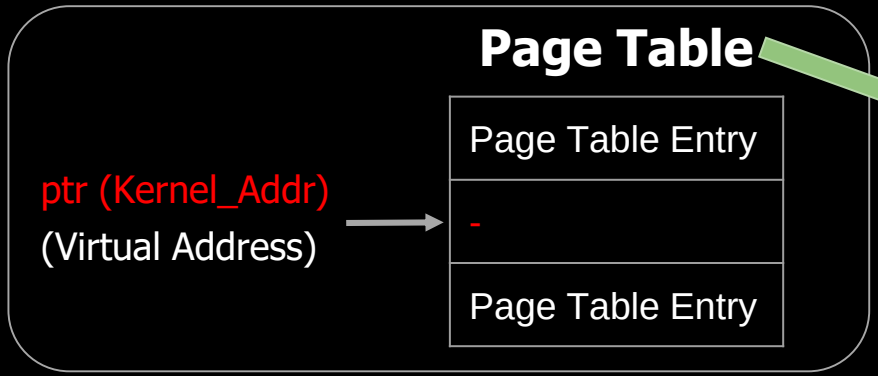
## Micro-architecture

(1) pa = convert_va_to_pa(ptr); ⇒ Fail!

~~(2) read pa;~~

~~(3) if (is_kernel_address(pa))~~
~~error;~~

**If no Page Table Entry for Kernel?**

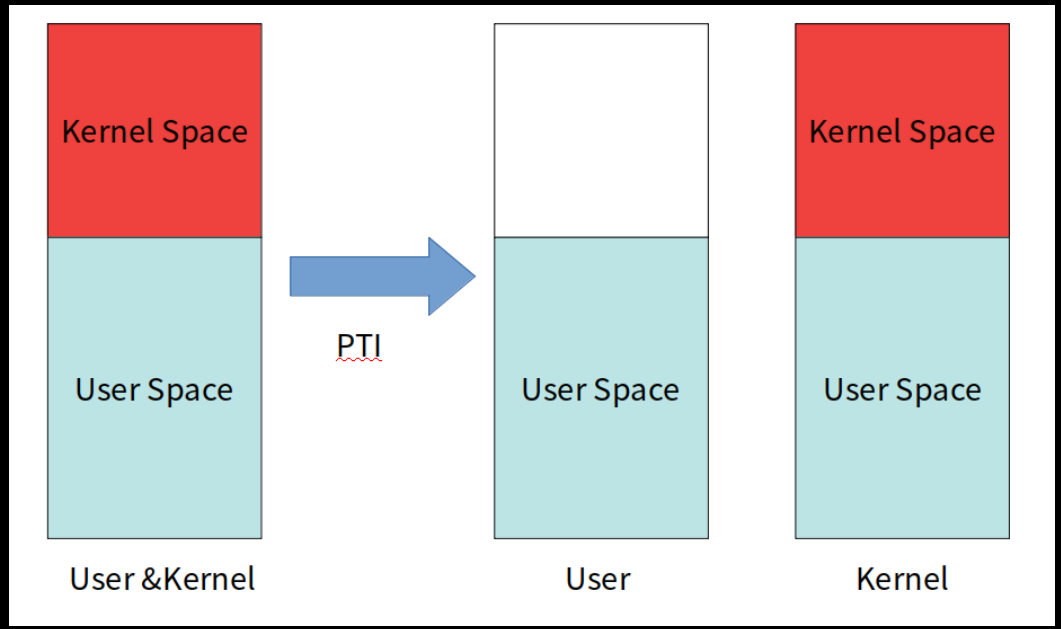



**Managed by Software (Linux)!**

# Page Table Isolation



## Unmapping Kernel Space in User Mode



# Implementation of Page Table Isolation



- Too difficult details for a beginner to understand...
- Links to get more details
  - https://gruss.cc/files/kaiser.pdf
  - https://jinb-park.blogspot.com/2019/06/deep-dive-into-page-table-isolation.html

# Attack5: L1TF (L1 Terminal Fault)

SOSCON 2019

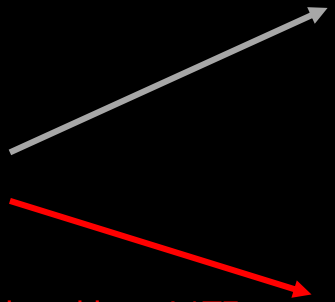
SAMSUNG OPEN SOURCE CONFERENCE 2019

# Revisit Micro-architecture code



## Software

```
int *ptr = Not_Exist_Addr;

void func(void)
{
    int a = *ptr;
}
```




## Micro-architecture

```
pa = convert_va_to_pa(ptr);

if (!is_present(pa))
   error;

read pa;
```

```
pa = convert_va_to_pa(ptr);

read pa;

if (!is_present(pa))
   error;
```

# From Zero Address



## Software

```
int *ptr = Not_Exist_Addr;

void func(void)
{
    int a = *ptr;
}
```

## Micro-architecture

(1) pa = convert_va_to_pa(ptr);

(2) read pa; → From address Zero?
We can't make sure that what's in it.
Secret? Dummy value?

(3) if (!is_present(pa))
        error;

## Page Table

ptr (Not_Exist_Addr)
(Virtual Address) →

| Page Table Entry |
| --- |
| Page Table Entry |
| Page Table Entry |


Normally Zero
(Physical Address)

Present Bit
[1]: Present
[0]: Not ...

# From Kernel Address



## Software

```
int *ptr = Not_Exist_Addr;

void func(void)
{
    int a = *ptr;
}
```

## Micro-architecture

(1) pa = convert_va_to_pa(ptr);

(2) read pa;

(3) if (!is_present(pa))
        error;

**Kernel Memory**

**Attacker can control!**

## Page Table

| Page Table Entry |
| --- |
| Page Table Entry |
| Page Table Entry |

ptr (Not_Exist_Addr)
(Virtual Address)


Present Bit
[1]: Present
[0]: Not ...

# Feasible?



**Software**

**Virtual Address**

```
int *ptr = Not_Exist_Addr;

void func(void)
{
    int a = *ptr;
}
```

**Page Table**

| Page Table Entry |
| --- |
| Page Table Entry (Kernel Address) |
| Page Table Entry |

**Physical Address**

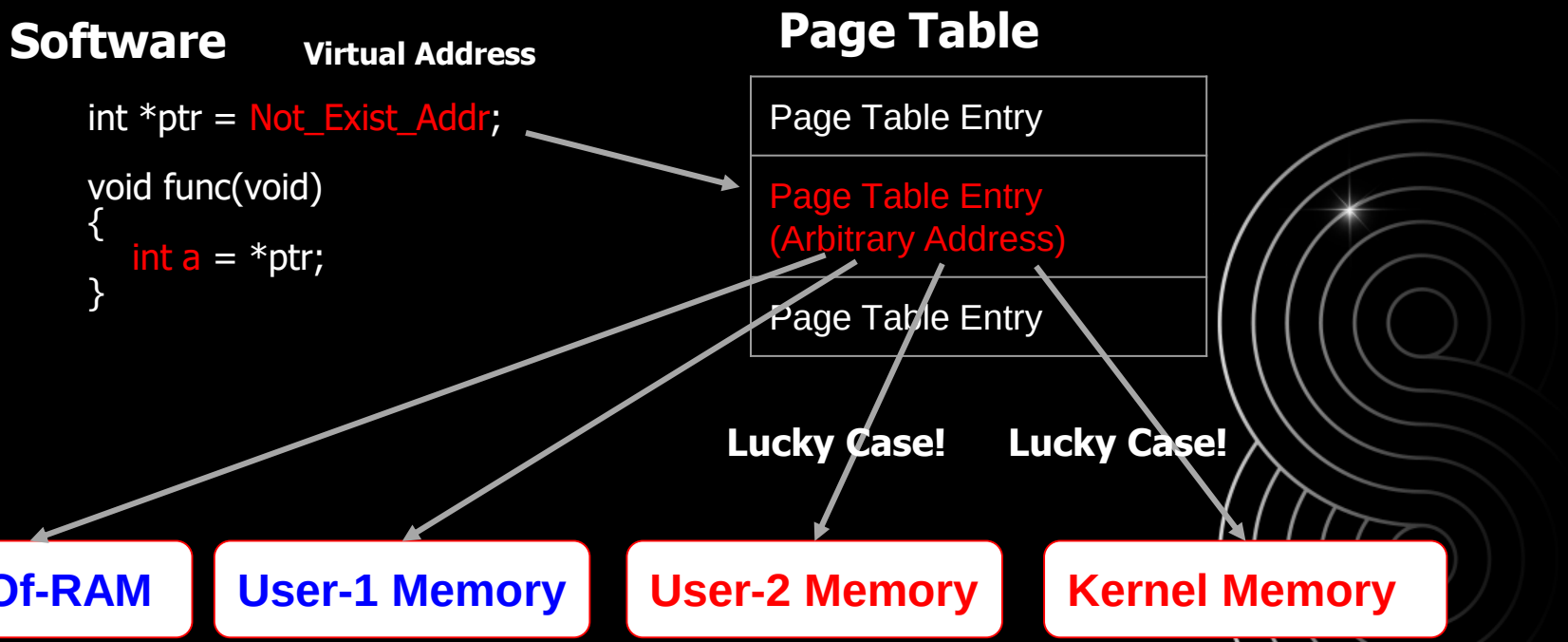
**: If attackers control the physical address of page table, they can read arbitrary kernel memory!**

**: But we are user, not kernel.  It's infeasible for user to control page table.**

**: That's why L1TF is not as popular as Meltdown.**

# Relying on Luck?






**: There might still be possibility relying on luck! How can we eliminate such possibility?**

# Stick it to the leak gadget



## Leak gadget + FLUSH-RELOAD + L1TF = Kernel memory leak!

```
int *ptr = Not_Exist_Addr;  // Physical address ==> Kernel address

void func(void)
{
    int a = *ptr;        // (1) a = kernel-value
    var = ary1[a];       // (2) [leak gadget] kernel-value dependent access → Cache loading
    // (3) invalid access! fault!
}

void fault_handler(....)
{
    for (i=0; i<SIZE; i++) {
        if (time_to_access(ary[i]) < CONSTANT)
            // (4) i is kernel-value!;
    }
}
```

# Additional Requirements



- L1TF has some prerequisites,
  -- L1 cache must contain entry for target physical address.

- Links to get more details
  - https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-van_bulck.pdf
  - https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault

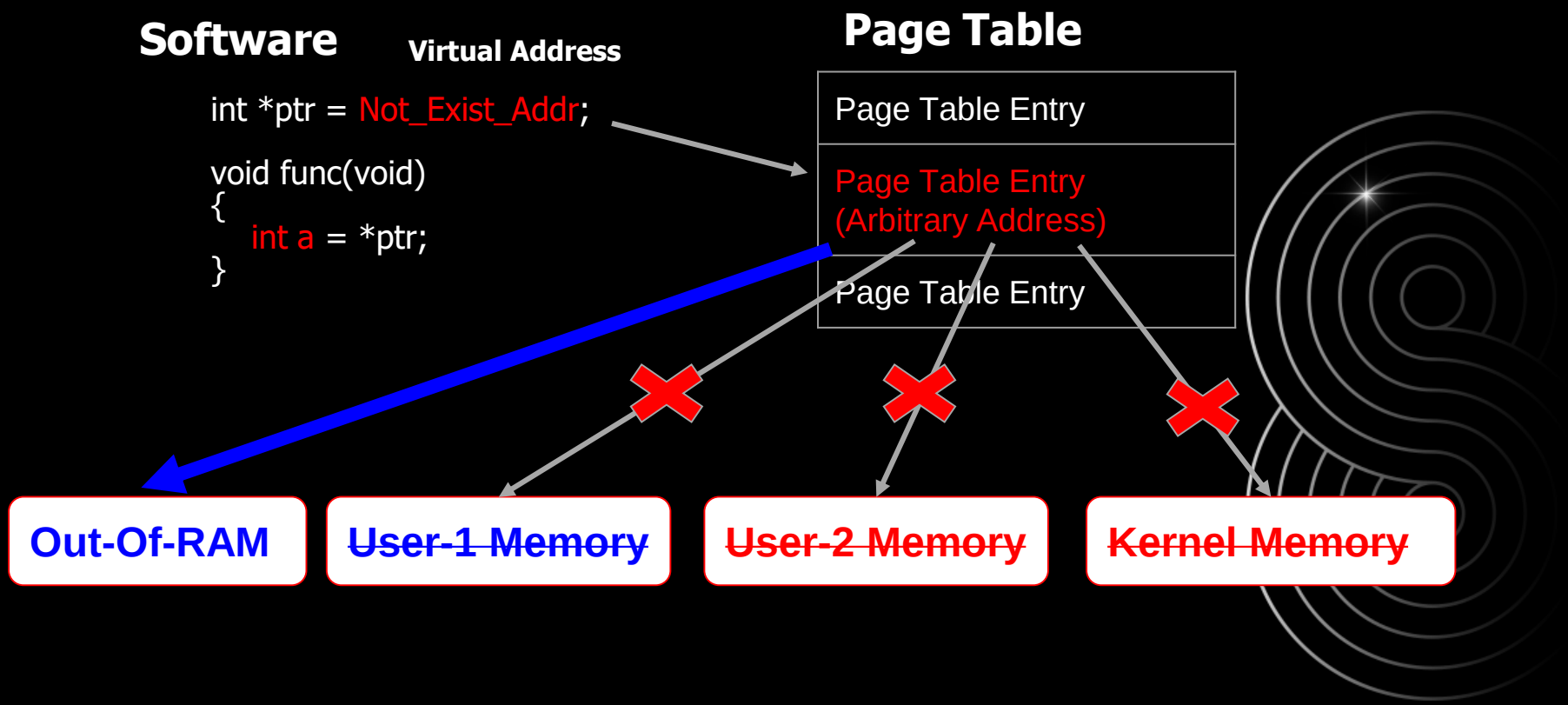
# Defense5: PTE Inversion

SOSCON 2019

SAMSUNG OPEN SOURCE CONFERENCE 2019

# Forcing PTE to point to Out-Of-RAM





# Making an out-of-RAM address



**Page Table**

Out-Of-RAM

Page Table Entry
(Physical Address)

**: Assume the size of RAM is 16GB. → 0x00 ~ 0x400000000**

**: Setting one to the top bit. → 0x00 ~ 0x8000000400000000**
**→ Over 1024*1024 TBs (Out-Of-All-Kinds-of-RAM)**

**: Performing an Inversion!**
**→ Original: 0x0000000000001000**
**→ Inversion: 0x1111111111110111**

# Kernel code of PTE Inversion



**Page Table**

Page Table Entry
(Physical Address)

**For all accesses on PTE, pte_pfn running!**

```
// PTE inversion

static inline bool __pte_needs_invert(u64 val)
{
    // NB. PTE is exist, but invalid.
    return val && !(val & _PAGE_PRESENT);
}

static inline u64 protnone_mask(u64 val)
{
    // NB. If PTE inversion needed,
    // return the mask for PTE to point to invalid memory.
    return __pte_needs_invert(val) ?  ~0ull : 0;
}

static inline unsigned long pte_pfn(pte_t pte)
{
    phys_addr_t pfn = pte_val(pte);
    // NB. Masking PFN (physical address)
    // after masking, it's pointing to invalid memory.
    pfn ^= protnone_mask(pfn);
    return (pfn & PTE_PFN_MASK) >> PAGE_SHIFT;
}
```

# THANK YOU

SOSCON 2019

SAMSUNG OPEN SOURCE CONFERENCE 2019

# Appendix-A

SOSCON 2019

SAMSUNG OPEN SOURCE CONFERENCE 2019

# All attacks so far



| Vulnerability | N | CVE | Public Vulnerability Name (codename) | Affected CPU architectures and mitigations | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | Intel[5] | | | | AMD[6] | |
| | | | | Ice Lake[7] | Cascade Lake | Whiskey Lake, Coffee Lake (9th gen)[8] | Coffee Lake (8th gen)* | Zen 1 / Zen 1+ | Zen 2[9] |
| Spectre | 1 | 2017-5753 | Bounds Check Bypass | OS/VMM | | | | Firmware + OS/VMM | Hardware + OS/VMM |
| Spectre | 2 | 2017-5715 | Branch Target Injection | Hardware + OS | | Firmware + OS | Firmware + OS | | |
| SpectreRSB[10]/ret2spec[11] | 2 | 2018-15572 | Return Mispredict | | | | | | |
| Meltdown | 3 | 2017-5754 | Rogue Data Cache Load | Not affected | | | Firmware | Not affected | |
| Spectre-NG | 3a | 2018-3640 | Rogue System Register Read | Not affected[12] | | Firmware | | | |
| Spectre-NG | 4 | 2018-3639 | Speculative Store Bypass | Hardware + OS/VMM[12] | | Firmware + OS | | OS/VMM | Hardware + OS/VMM |
| Foreshadow | 5 | 2018-3615 | L1 Terminal Fault (L1TF) | Not affected | | | Firmware | Not affected | |
| Spectre-NG | | 2018-3665 | Lazy FP State Restore | OS/VMM[13] | | | | | |
| Spectre-NG | 1.1 | 2018-3693 | Bounds Check Bypass Store | OS/VMM[14] | | | | | |
| Spectre-NG | 1.2 | | Read-only Protection Bypass (RPB) | | | | | | |
| Foreshadow-OS | | 2018-3620 | L1 Terminal Fault (L1TF) | Not affected | | | Firmware + OS | Not affected | |
| Foreshadow-VMM | | 2018-3646 | L1 Terminal Fault (L1TF) | | | | | | |
| ZombieLoad | | 2018-12130 | Microarchitectural Fill Buffer Data Sampling (MFBDS) | | | | | | |
| RIDL | | 2018-12127<br>2019-11091 | Microarchitectural Data Sampling Uncacheable Memory<br>MLPDS/MDSUM | | | | | | |
| Fallout | | 2018-12126 | Microarchitectural Store Buffer Data Sampling (MSBDS) | | | | | | |
| Spectre SWAPGS | 1 | 2019-1125 | None yet[15][16][17][18] | Same as Spectre 1 | | | | | |

: From https://en.wikipedia.org/wiki/Transient_execution_CPU_vulnerabilities

Discussed here

Worth to look at

# All attacks so far



- The most useful link : A dynamic Tree

- Intel Sandybridge Microarchitecture

# Appendix-B

SOSCON 2019

SAMSUNG OPEN SOURCE CONFERENCE 2019

# Branch Target Buffer



Used to predict indirect branch target.

Stores recent branch history (source & destination address pairs).

Typically, stores partial src addr and dst addr.

Shared per cpu core.

Ref: Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR

# Branch Target Buffer (Cont.)



## BTB

| src | dst |
|---|---|
| 0x80100 | 0x56780A00 |
| 0x80200 | 0x56780B00 |
| 0x80300 | 0x56780C00 |

Only low 20 bits of src addr are stored!
(It depends on the CPU design.)

## Code

0x56780100: jmp [0x56781010]
…
0x56780200: jmp [0x56781020]
…
0x56780300: jmp [0x56781030]

## Data

0x56781010: 0x56780A00

0x56781020: 0x56780B00

0x56781030: 0x56780C00

# Appendix-C

SOSCON 2019

SAMSUNG OPEN SOURCE CONFERENCE 2019

# Spectre v2 detailed steps



## Victim

### Memory

| | |
|---|---|
| | ... |
| 0x56780100 | jmp [0x56781010] |
| | ... |
| 0x56780200 | jmp [0x56781020] |
| | ... |
| 0x56780300 | jmp [0x56781030] |
| | ... |
| 0x56780F00 | **var = ary2[ary1[idx]]** |
| | ... |

### BTB

| src | dst |
|---|---|
| 0x80100 | 0x56780A00 |
| 0x80200 | 0x56780B00 |
| 0x80300 | **0x56780C00** |

Attacker will change this address to **0x56780F00**.

Gadget address!

### Data

| addr | value |
|---|---|
| 0x56781010 | 0x56780A00 |
| 0x56781020 | 0x56780B00 |
| **0x56781030** | 0x56780C00 |

Attacker will flush this memory for speculative execution.

# Spectre v2 detailed steps (Cont.)



## Attacker: 1. Mistrain BTB.

**Memory**

| | |
|---|---|
| | ... |
| 0x56780100 | ret |
| | ... |
| 0x56780200 | ret |
| | ... |
| 0x56780300 | ret |
| | ... |
| 0x56780F00 | **ret** |
| | ... |

**BTB**

| src | dst |
|---|---|
| 0x80100 | 0x56780A00 |
| 0x80200 | 0x56780B00 |
| 0x80300 | 0x56780C00 |

1) Allocate memory same as victim's.

2) Fill up with ret instructions.

# Spectre v2 detailed steps (Cont.)



## Attacker: 1. Mistrain BTB.

Before: 0x56780300 → 0x56780C00
Now: 0x56780300 → **0x56780F00**

| | Memory |
|---|---|
| | ... |
| 0x56780100 | ret |
| | ... |
| 0x56780200 | ret |
| | ... |
| 0x56780300 | ret |
| | ... |
| 0x56780F00 | ret |
| | ... |

| Stack |
|---|
| **0x56780A00** |
| next code addr 1 |
| **0x56780B00** |
| next code addr 2 |
| **0x56780F00** |
| next code addr 3 |

3) Push original dst addrs and the gadget addr.

**BTB**

| src | dst |
|---|---|
| 0x80100 | 0x56780A00 |
| 0x80200 | 0x56780B00 |
| 0x80300 | **0x56780F00** |

**Changed!!**

4) Jump to original src addrs.

*jmp 0x56780100*
*jmp 0x56780200*
*jmp 0x56780300*

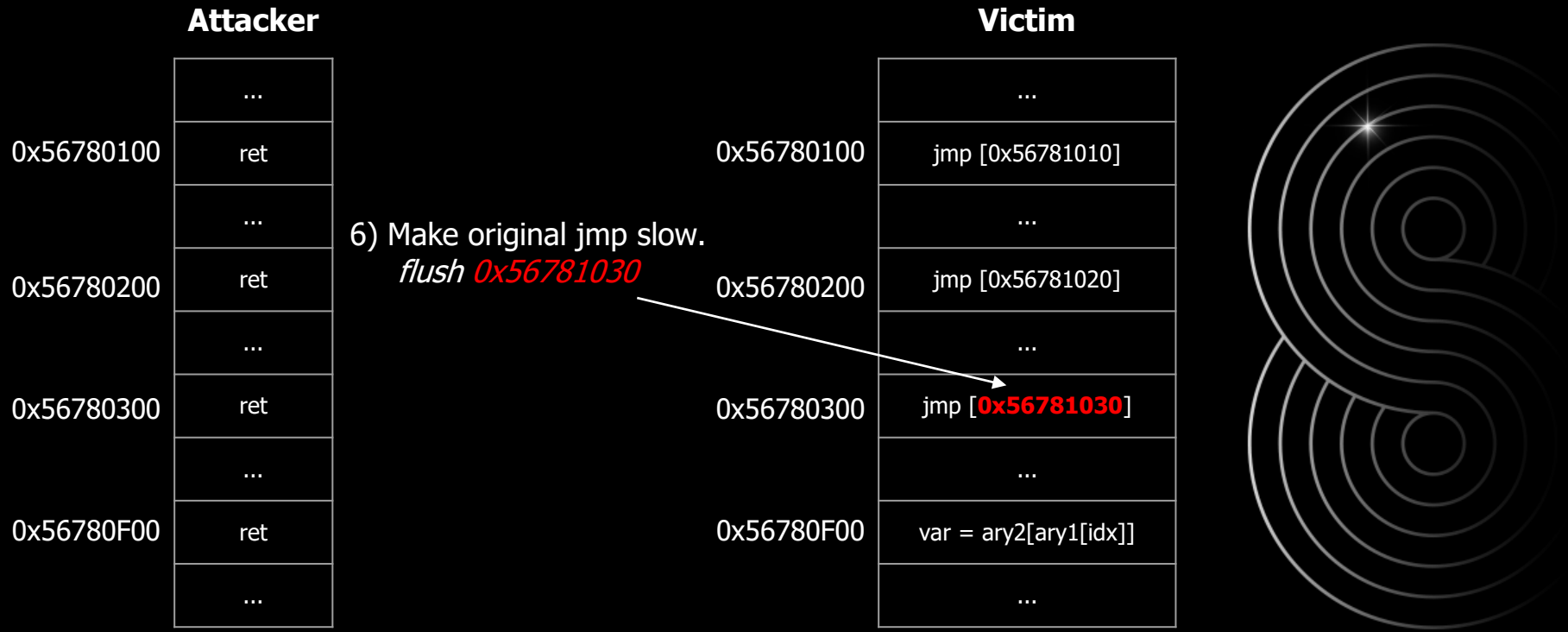
5) Iterate 3) and 4).

# Spectre v2 detailed steps (Cont.)



## Attacker: 1. Mistrain BTB.

Before: 0x56780300 → 0x56780C00
Now: 0x56780300 → **0x56780F00**



# Spectre v2 detailed steps (Cont.)



## Attacker: 2. Execute victim with malicious idx.

**Victim**

| | |
|---|---|
| | ... |
| 0x56780100 | jmp [0x56781010] |
| | ... |
| 0x56780200 | jmp [0x56781020] |
| | ... |
| 0x56780300 | jmp [0x56781030] |
| | ... |
| 0x56780F00 | **var = ary2[ary1[idx]]** |
| | ... |

**Flushed!**

**BTB**

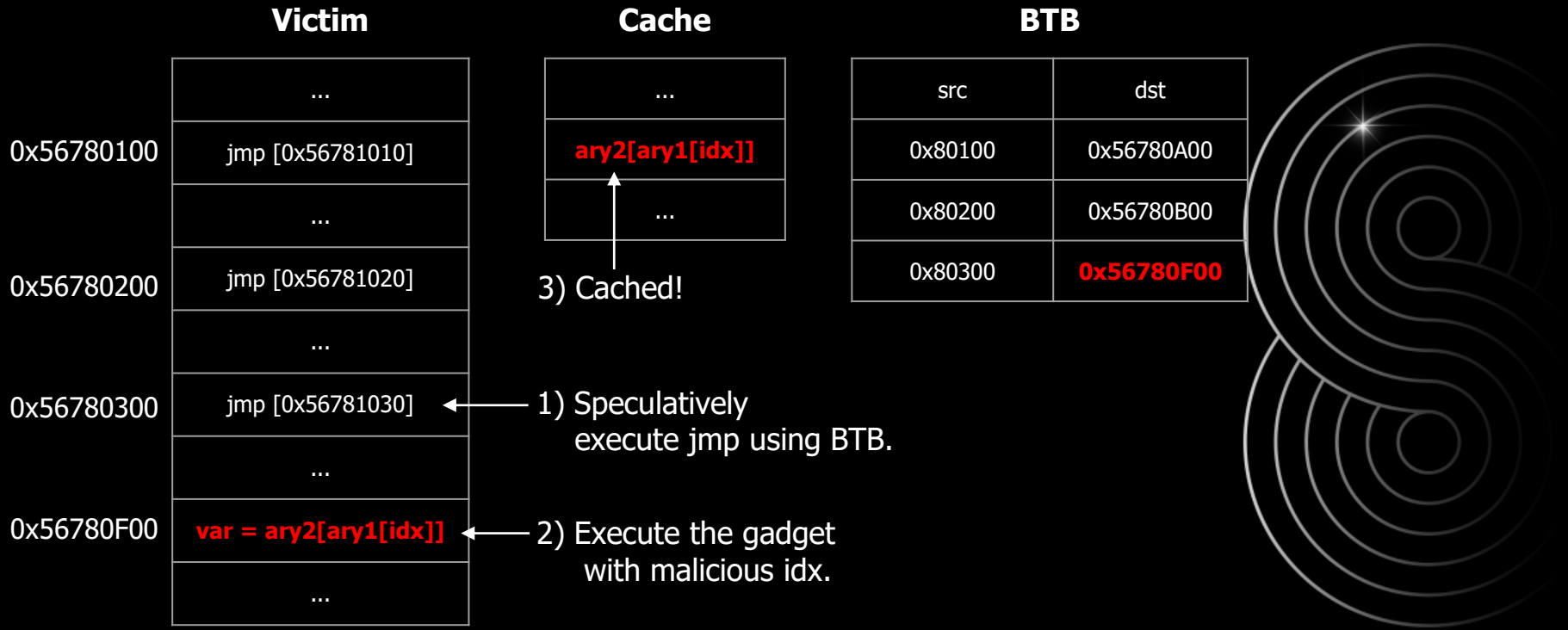
| src | dst |
|---|---|
| 0x80100 | 0x56780A00 |
| 0x80200 | 0x56780B00 |
| 0x80300 | **0x56780F00** |

**Mistrained to the gadget.**

# Spectre v2 detailed steps (Cont.)



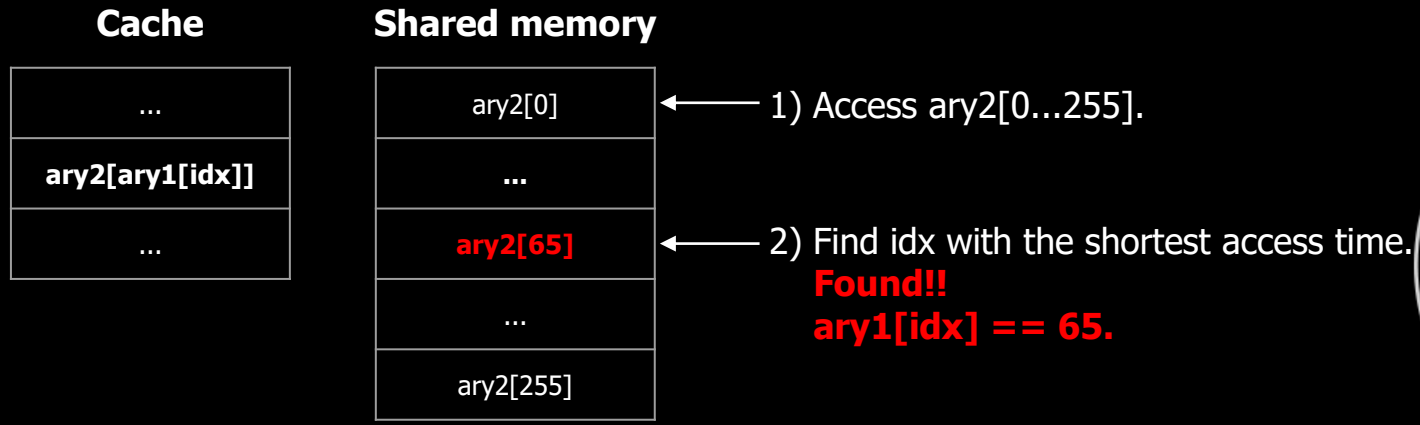
## Attacker: 2. Execute victim with malicious idx.



# Spectre v2 detailed steps (Cont.)



**Attacker: 3. Retrieve value via Flush + Reload analysis.**



# Appendix-D

SOSCON 2019

SAMSUNG OPEN SOURCE CONFERENCE 2019

# Retpoline details



**Goal: Avoid predictions which use the BTB.**

## Steps

1. Change indirect jmp to ret.

   When predicting ret's destination, RSB(Return Stack Buffer) is used ahead of BTB.

2. Capture speculative execution into a fake infinite loop.

# Retpoline details (Cont.)



**Indirect call replacement with retpoline.**

```
jmp *%rax  ⟶   call load_label
capture_ret_spec:
               pause ; lfence
               jmp capture_ret_spec
load_label:
               mov %rax, (%rsp)
               ret
```

# Retpoline details (Cont.)



## Original path

```
                call load_label        <--
capture_ret_spec:
                pause ; lfence
                jmp capture_ret_spec
load_label:
                mov %rax, (%rsp)
                ret
```

**rax register**

| Original jmp dst |
|---|

**Stack**

| |
|---|
| |
| |
| **capture_ret_spec** |

# Retpoline details (Cont.)



## Original path

```
                call load_label
capture_ret_spec:
                pause ; lfence
                jmp capture_ret_spec
load_label:
                mov %rax, (%rsp)   <--
                ret
```

**rax register**

| Original jmp dst |
|---|

**Stack**

| |
|---|
| |
| |
| **Original jmp dst** |

# Retpoline details (Cont.)



## Original path

```
                call load_label
capture_ret_spec:
                pause ; lfence
                jmp capture_ret_spec
load_label:
                mov %rax, (%rsp)
                ret
```

**rax register**

| Original jmp dst |
|---|

**Stack**

| |
|---|
| |
| |
| ~~Original jmp dst~~ |

<-- **Jump to the original dst.**

# Retpoline details (Cont.)



## Speculative path

```
                call load_label        <--
capture_ret_spec:
                pause ; lfence
                jmp capture_ret_spec
load_label:
                mov %rax, (%rsp)
                ret
```

**rax register**

| Original jmp dst |
|---|

**RSB (Return Stack Buffer)**

| |
|---|
| |
| **capture_ret_spec** |

# Retpoline details (Cont.)



## Speculative path

```
                call load_label
capture_ret_spec:
                pause ; lfence
                jmp capture_ret_spec
load_label:
                mov %rax, (%rsp)   <--
                ret
```

**rax register**

| Original jmp dst |
|---|

**RSB (Return Stack Buffer)**

| |
|---|
| |
| |
| capture_ret_spec |

# Retpoline details (Cont.)



## Speculative path

```
                call load_label
capture_ret_spec:
                pause ; lfence
                jmp capture_ret_spec
load_label:
                mov %rax, (%rsp)
                ret
```

**rax register**

| Original jmp dst |
|---|

**RSB (Return Stack Buffer)**

| |
|---|
| |
| |
| ~~capture_ret_spec~~ |

**<-- Jump to capture_ret_spec.**

# Retpoline details (Cont.)



## Speculative path

```
                call load_label
capture_ret_spec:
                pause ; lfence          <--
                jmp capture_ret_spec
load_label:
                mov %rax, (%rsp)
                ret
```

**rax register**

| Original jmp dst |
|---|

**RSB (Return Stack Buffer)**

| |
|---|
| |
| |
| |

**Captured in the infinite loop!**