Spectre(v1/v2/v4) V.S. Meltdown(v3)

Gavin Guo
Engineering Technical Lead

CANONICAL

Spectre and Meltdown

Spectre

- Variant 1: bounds check bypass (CVE-2017-5753)
  - PoC of V1
  - Mitigation of V1(Dan William/OSB)
- Variant 2: branch target injection (CVE-2017-5715)
  - PoC of V2
  - Mitigation of V2(retpoline/IBRS/IBPB)
- Variant 4: speculative store bypass (CVE 2018-3639)

Meltdown

- Variant 3: rogue data cache load (CVE-2017-5754)

Spectre V1: Bounds Check Bypass (CVE-2017-5753)

PoC of Variant 1/3

```
struct array {
        unsigned long length;
         unsigned char data[];
};

struct array *arr1 = ...; /* small array, arr1->length = 10 */
struct array *arr2 = ...; /* array of size 0x400 */

/* >0x400 (OUT OF BOUNDS!) */
unsigned long untrusted_offset_from_caller = 0, i = 0;
clflush(arr2->data);
While (i < 30) { // branch prediction buffer size
    if (untrusted_offset_from_caller < arr1->length) {  // Where Speculation happens if (i == 29)
            unsigned char value = arr1->data[untrusted_offset_from_caller];

            unsigned long index2 = ((value & 1) * 0x100) + 0x200;
            if (index2 < arr2->length) {
                    unsigned char value2 = arr2->data[index2];
            }
    }
    If (i == 28) { untrusted_offset_from_caller = ((void *)addrof(proc_banner) - (void *)addr->data); }
         i++;
}
probe_read_time(arr2->data[0x200], arr2->data[0x300]); // rdtsc assembly
```

```
%eax = Rdtsc
arr2->data[0x300]
ld %ecx, arr2, 0x300
%ebx = Rdtsc
```

Ref: [5]

Mitigations of Spectre V1

Variant 1: bounds check bypass (CVE-2017-5753)

[PATCH v4.1 00/10] spectre variant1 mitigations for tip/x86/pti

For example, in the presence of branch prediction, it is possible for bounds checks to be ignored by code which is speculatively executed. Consider the following code:

```
int load_array(int *array, unsigned int idx)
{
        if (idx >= MAX_ARRAY_ELEMS)
                return 0;
        else
                return array[idx];
}
```

==>

Which, on arm64, may be compiled to an assembly sequence such as:

Branch predict taken

```
        CMP  <idx>, #MAX_ARRAY_ELEMS
        B.LT less
        MOV  <returnval>, #0
        RET
less:
        LDR  <returnval>, [<array>, <idx>]
        RET
```

Mitigations of Spectre V1

- **Variant 1: bounds check bypass (CVE-2017-5753)**
  - Mitigated: [PATCH v4.1 00/10] spectre variant1 mitigations for tip/x86/pti

```
+/*
+ * If idx is negative or if idx > size then bit 63 is set in the mask,
+ * and the value of ~(-1L) is zero. When the mask is zero, bounds check
+ * failed, array_ptr will return NULL.
+ */
+#ifndef array_ptr_mask
+static inline unsigned long array_ptr_mask(unsigned long idx, unsigned
long sz)
+{
+        return ~(long)(idx | (sz - 1 - idx)) >> (BITS_PER_LONG - 1);
+}
+#endif
+
```

idx = 33, sz = 32
32 - 1 - 33 = -1 = 0xFFFFFFFF
idx | 0xFFFFFFFF = 0xFFFFFFFF
~0xFFFFFFFF = 0x0
0x0 >> 31 = 0x0

idx = 1, sz = 32
32 - 1 - 1 = 30 = 0x1E
idx | 0x1E = 0x1F
~0x1F = 0xFFFFFFE0
0xFFFFFFE0 >> 31 =
0xFFFFFFFF

```
+/**
+ * array_ptr - Generate a pointer to an array element, ensuring
+ * the pointer is bounded under speculation to NULL.
+ *
+ * @base: the base of the array
+ * @idx: the index of the element, must be less than LONG_MAX
+ * @sz: the number of elements in the array, must be less than LONG_MAX
+ *
+ * If @idx falls in the interval [0, @sz), returns the pointer to
+ * @arr[@idx], otherwise returns NULL.
+ */
+#define array_ptr(base, idx, sz)                                      \
+({                                                                    \
+        union { typeof(*(base)) *_ptr; unsigned long _bit; } __u;     \
+        typeof(*(base)) *_arr = (base);                               \
+        unsigned long _i = (idx);                                     \
+        unsigned long _mask = array_ptr_mask(_i, (sz));               \
+                                                                      \
+        __u._ptr = _arr + _i;                                         \
+        __u._bit &= _mask;                                            \
+        __u._ptr;                                                     \
+})
+#endif /* __NOSPEC_H__ */
```

Spectre V1 mitigation: Ubuntu adopts the patches

cb11ef4db1d0 UBUNTU: SAUCE: arm: no osb() implementation yet
1e738b18f256 UBUNTU: SAUCE: arm64: no osb() implementation yet
2fc7fab69765 UBUNTU: SAUCE: s390/spinlock: add osb memory barrier
c5a6edd430fc UBUNTU: SAUCE: powerpc: add osb barrier
71585422c5ed UBUNTU: SAUCE: claim mitigation via observable speculation barrier
3627e88783ac userns: prevent speculative execution
9780ac7b9211 udf: prevent speculative execution
798c6525821d net: mpls: prevent speculative execution
8661e7a75b2a fs: prevent speculative execution
5603071f433b ipv6: prevent speculative execution
98f3f8c322fe ipv4: prevent speculative execution
ce20b02844d1 Thermal/int340x: prevent speculative execution
7895a746d217 qla2xxx: prevent speculative execution
8e8527e8a64a carl9170: prevent speculative execution
06c867e7e4a5 UBUNTU: SAUCE: FIX: x86, bpf, jit: prevent speculative execution when JIT is enabled
3cc56bbbc676 x86, bpf, jit: prevent speculative execution when JIT is enabled
81774d484f26 bpf: prevent speculative execution in eBPF interpreter
1fed0ab0bd69 locking/barriers: introduce new observable speculation barrier

The patch set was made by elena.reshetova@intel.com

OVMSA-2018-0015 - Unbreakable Enterprise kernel security update(Oracle)

- KVM: x86: Add memory barrier on vmcs field lookup (Andrew Honig) {CVE-2017-5753}

[4.1.12-61.60.1]
- userns: prevent speculative execution (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}
- udf: prevent speculative execution (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}
- net: mpls: prevent speculative execution (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}
- fs: prevent speculative execution (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}
- ipv6: prevent speculative execution (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}
- ipv4: prevent speculative execution (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}
- Thermal/int340x: prevent speculative execution (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}
- cw1200: prevent speculative execution (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}
- qla2xxx: prevent speculative execution (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}
- p54: prevent speculative execution (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}
- carl9170: prevent speculative execution (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}
- uvcvideo: prevent speculative execution (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}
- bpf: prevent speculative execution in eBPF interpreter (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}
- locking/barriers: introduce new observable speculation barrier (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}
- x86/cpu/AMD: Remove now unused definition of MFENCE_RDTSC feature (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}
- x86/cpu/AMD: Make the LFENCE instruction serialized (Elena Reshetova) [Orabug: 27345857] {CVE-2017-5753}

Spectre V1 mitigation

The mitigation of Spectre v1 is backporting from the upstream to Ubuntu Kernel on 2018/6/1:

[SRU][Xenial][PATCH 0/5] Prevent speculation on user controlled

pointer

https://lists.ubuntu.com/archives/kernel-team/2018-June/093096.html

Spectre V2 - Variant 2: Branch Target Injection (CVE-2017-5715)

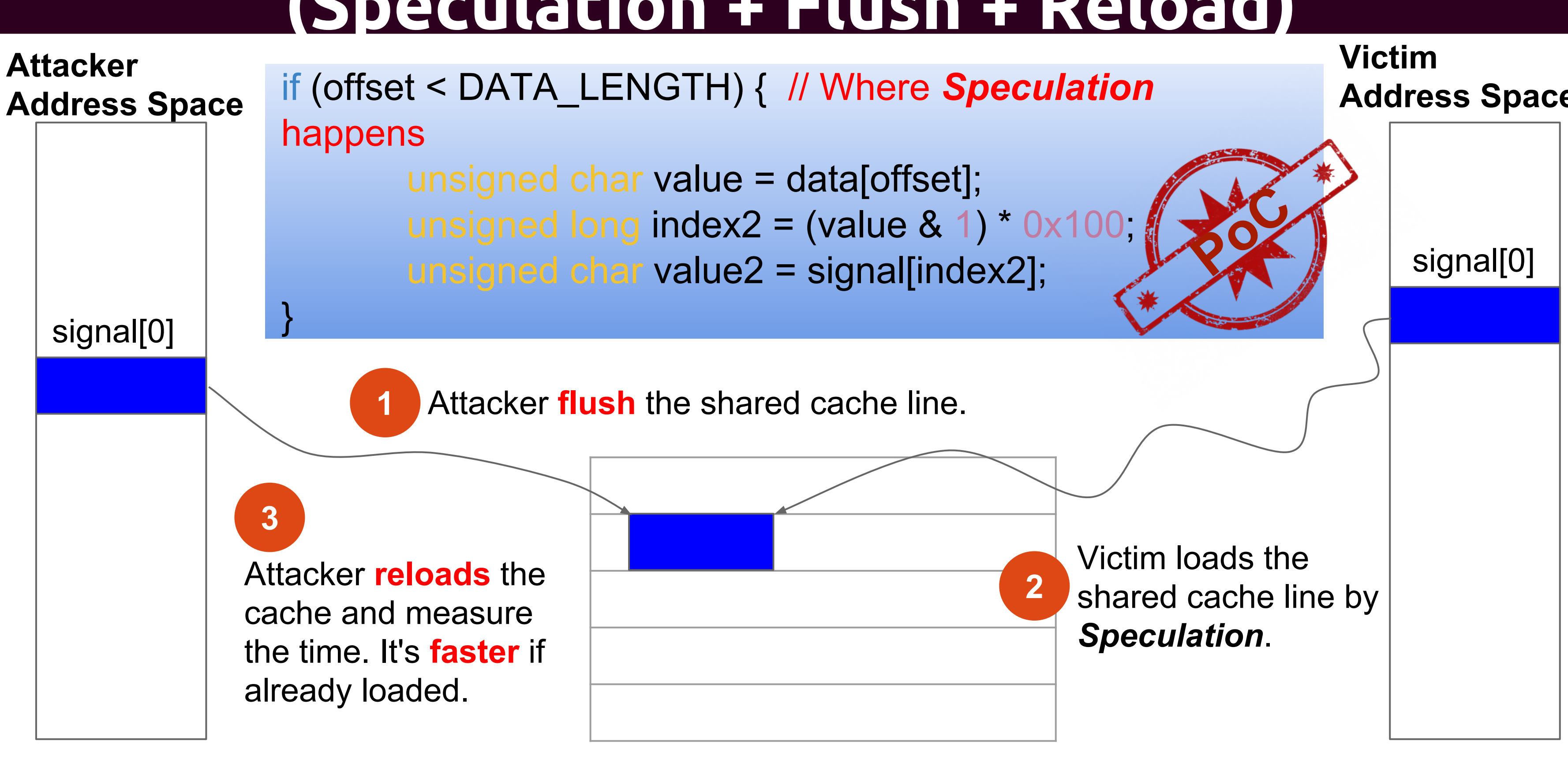
Agenda

- Introduction
- What's the ***Side-Channel*** Attack?
- **Three kinds** of branch prediction mechanisms.
  - Generic Branch Predictor
  - Indirect Branch Predictor
  - Return Stack Predictor
- Spectre V2 - PoC Summary
- The exploit relies on the knowledge of the symbol locations of certain functions
  - The jump table technique to inject branch target.
  - How to get the base address of kvm_intel.ko/kvm.ko/vmlinux loading address/physical address of user-mapped page/page_offset?

Introduction

- The slide is based on ***Jann Horn***'s work[5][9]. Kudos to ***Jann Horn***!
- The **PoC** can be downloaded from chromium bugs checking website[9].
- The magic number(jump table or gadget offset) in the slide can be found in the kernel[10]. Using objdump against the kvm-intel.ko/kvm.ko/vmlinux can find the magic number(The vmlinux needs to be extracted first by "scripts/extract-vmlinux boot/vmlinuz-4.9.0-3-amd64 > vmlinux-4.9.0-3").
- As the detail process is too _**COMPLICATED**_, I had an detail presentation in Chinese at Taiwan Linux Kernel Hackers Club on 2018/06/12:
  - 主題分享 : 深入 Spectre v2 - VM 如何攻擊 Host(Understanding Spectre v2 - How VM can attack the Host?)
    - https://www.youtube.com/watch?v=zYRI60uAwYc.
    - https://v.youku.com/v_show/id_XMzY2MTQzMzg5Mg
- The Spectre v2 is presented in Beijing Linux Conference 2018.

What's *Side-Channel* Attack? (Speculation + Flush + Reload)



Ref: [5][7][8]

Three kinds of prediction mechanisms

- **Generic Branch Predictor**
  - The **Branch Target Buffer** which is part of the CPU to predict the target address of the branch instruction before the execution unit calculate the result.
- **Indirect Branch Predictor**
  - Includes **Branch History Buffer** and **Indirect Branch Target Buffer**. Use the previous branch history as the footprint to index the BHB/IBTB.
- **Return Predictor** (A.K.A **Return Stack Buffer**)
  - We haven't analyzed that in detail yet.

Ref: [5]

What's Return Stack Buffer(RSB)?

Call Func

Instruction length. i.e. 4

+

Indirect Branch Predictor

Return Stack Buffer

Return

Indirect Branch Predictor

Return Stack Buffer

Mux

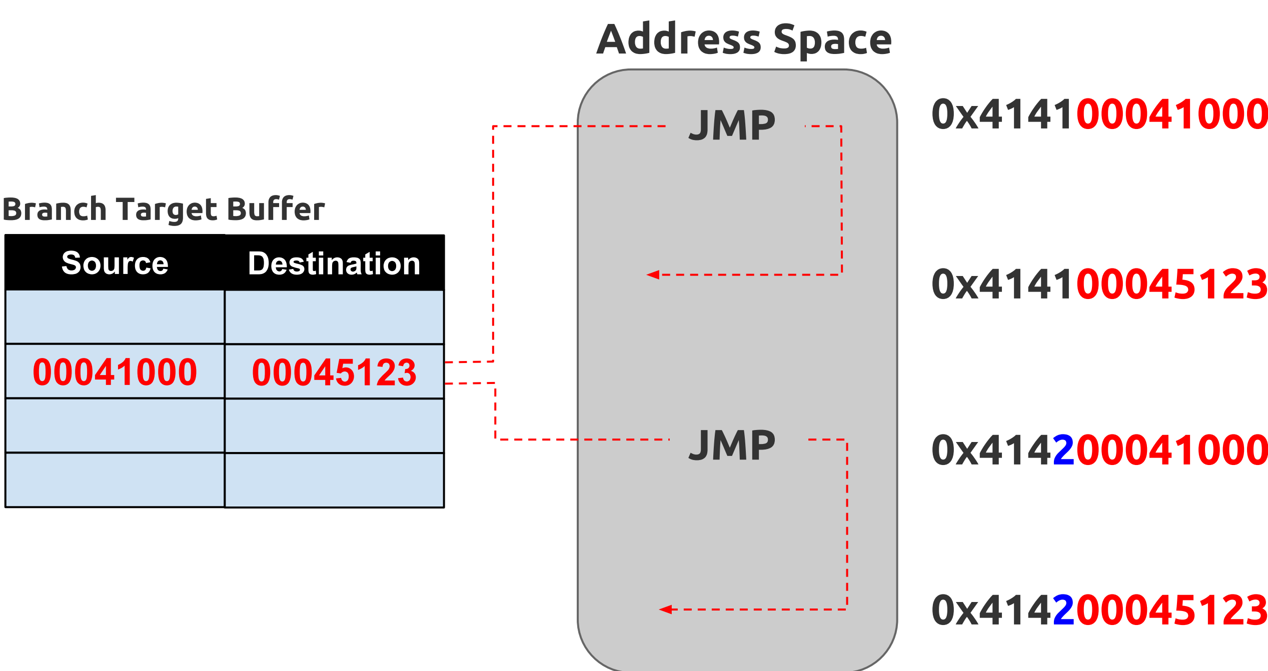
Select

Ref: [3:p.28]

The Return Stack Buffer (RSB) is a microarchitectural structure that holds **predictions** for execution of near **RET** instructions. Each execution of a near CALL instruction adds an entry to the RSB that contains the address of the instruction sequentially following that CALL instruction. The RSB is **not used** or updated by **far** CALL, **far** RET, or **IRET** instructions.

What's Generic Branch Predictor?

The Generic Branch Predictor is the Branch Target Buffer which is part of the CPU to predict the target address of the branch instruction before the execution unit calculate the result.



Ref: [2][3][19:p.19]

What's Indirect Branch Predictor?

The **branch history buffer (BHB)** stores information about the **last 29 taken branches** - basically a fingerprint of recent control flow - and is used to allow better prediction of indirect calls that can have multiple targets. Only the **lower 20 bits** of the **source** and **target** addresses have an influence on the **branch history buffer**.

Ref: [5]

What's Indirect Branch Predictor?

Ref: [4:p.35][15:p.22,p.34][18:p.20][20:p.15][21:p.18][22:p.22][23:p.19][24:p.41][25:p.18]

What's the Indirect Branch History Buffer XOR Math?

Source

Destination

addr[0,19]

addr[0,5]

XOR

58 bits

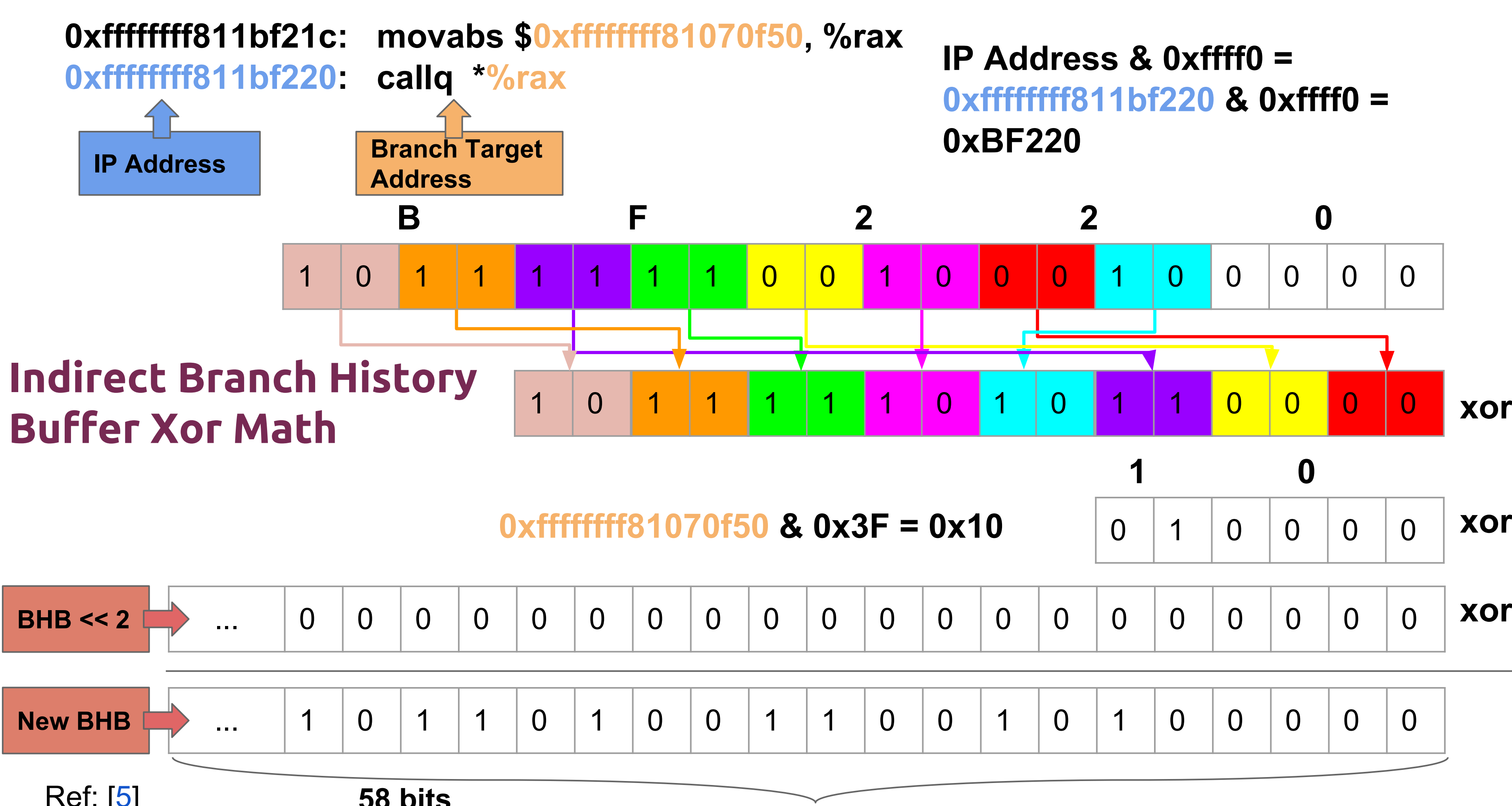
BHB << 2

Some of the bits of the BHB state seem to be folded together further using XOR when used for a BTB access, but the precise folding function **hasn't been understood** yet.

The only fact is that the BHB is shifted **two bits** at a time. We can guess the BHB by the trick.

```
void bhb_update(uint58_t *bhb_state, unsigned long src, unsigned long dst) {
 *bhb_state <<= 2;
 *bhb_state ^= (dst & 0x3f);
 *bhb_state ^= (src & 0xc0) >> 6;
 *bhb_state ^= (src & 0xc00) >> (10 - 2);
 *bhb_state ^= (src & 0xc000) >> (14 - 4);
 *bhb_state ^= (src & 0x30) << (6 - 4);
 *bhb_state ^= (src & 0x300) << (8 - 8);
 *bhb_state ^= (src & 0x3000) >> (12 - 10);
 *bhb_state ^= (src & 0x30000) >> (16 - 12);
 *bhb_state ^= (src & 0xc0000) >> (18 - 14);
}
```

Ref: [5][15:p.22,p.34]



0xffffffff811bf21c: movabs $0xffffffff81070f50, %rax
0xffffffff811bf220: callq *%rax
IP Address
Branch Target Address
IP Address & 0xffff0 =
0xffffffff811bf220 & 0xffff0 =
0xBF220
B
F
2
2
0
1 0 1 1 1 1 1 1 0 0 1 0 0 0 1 0 0 0 0 0
Indirect Branch History Buffer Xor Math
1 0 1 1 1 1 1 0 1 0 1 1 0 0 0 0
xor
1
0
0xffffffff81070f50 & 0x3F = 0x10
0 1 0 0 0 0
xor
BHB << 2
... 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
xor
New BHB
... 1 0 1 1 0 1 0 0 1 1 0 0 1 0 1 0 0 0 0 0
58 bits
Ref: [5]

What's Indirect Branch Predictor?(Fetch Stage)

IP address

XOR

branch history register

Bi-modal selector between Local and Global predictor

Generic Branch Predictor Buffer

Tag | 2bc

...

Taken or Not-Taken

Branch Target Buffer

...

Tag | Target

...

Hash Function

Indirect Branch Predictor Buffer

Tag | 2bc

...

Taken or Not-Taken

Indirect Branch Target Buffer

...

Tag | Target

...

Global prediction target address

Generic Prediction Target Address

Mux

Final prediction target address

T or NT

T

NT

Next Instruction

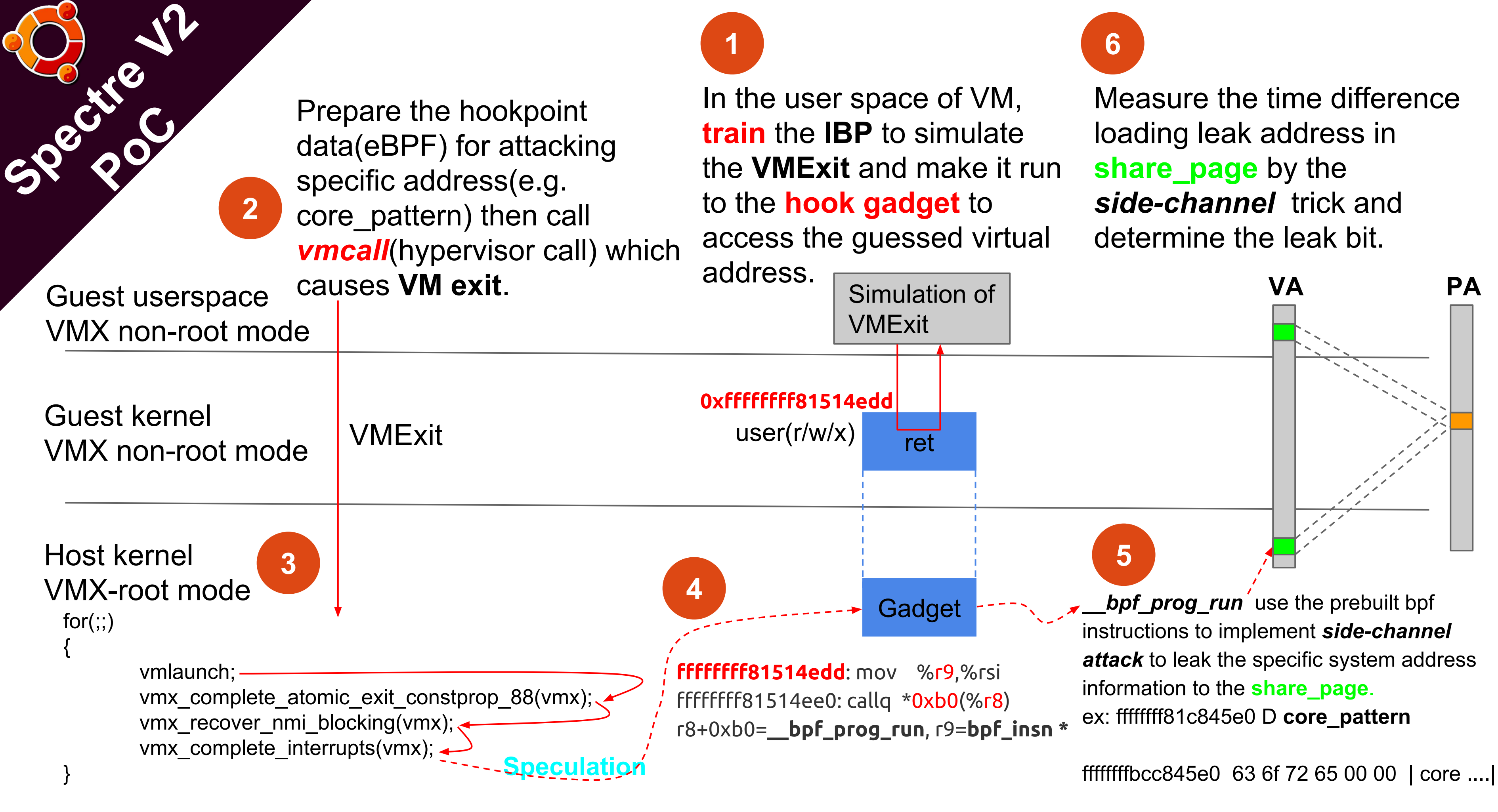
Mux

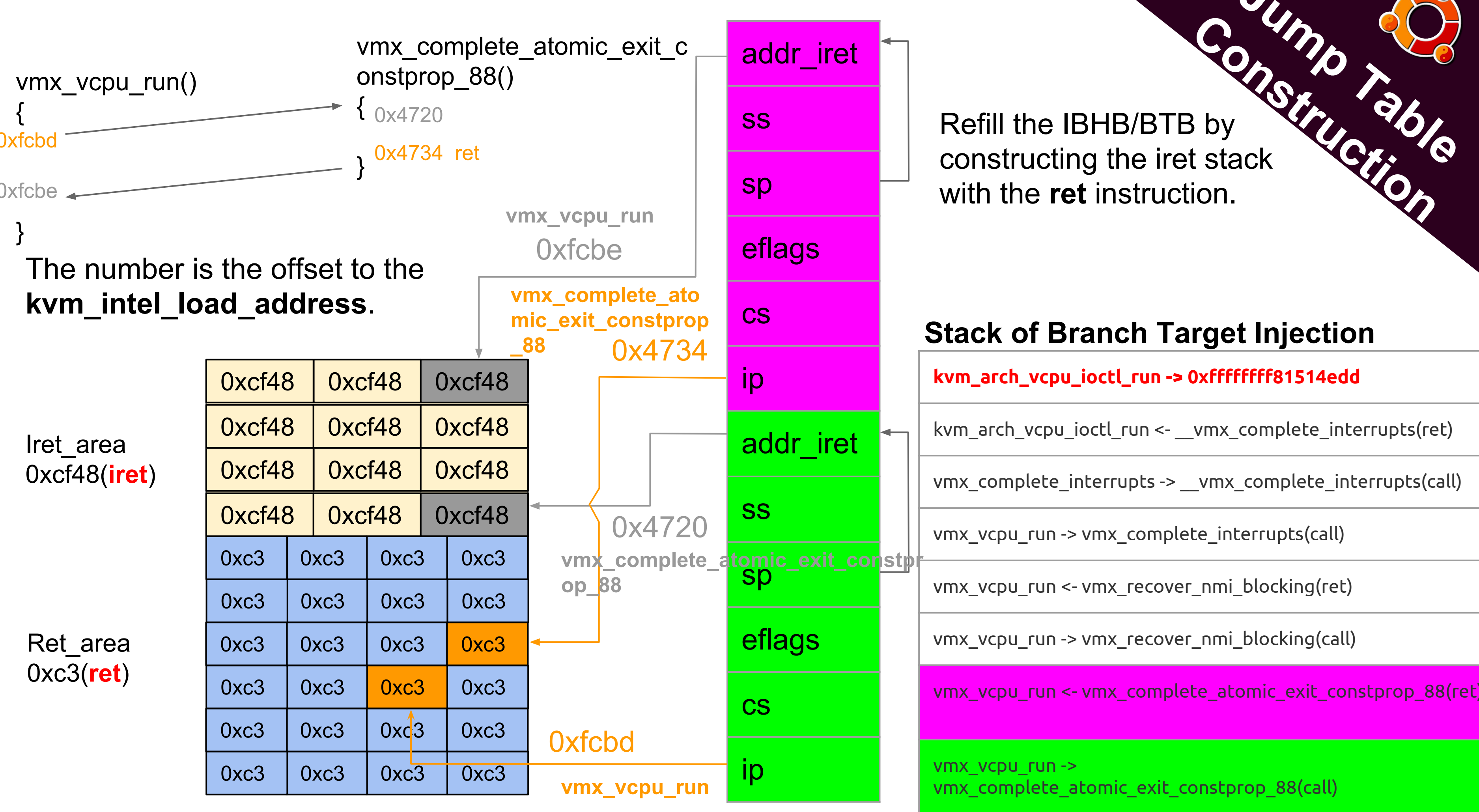
Spectre V2 - PoC Summary

- Take advantage of the ***Branch Target Injection*** and ***Side-Channel*** attack to get the base address of **kvm-intel.ko**, **vmlinux loading address**, and **physical address** of user-mapped pages, **page_offset**.

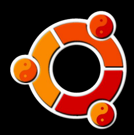
- Once we get the base address, we can train the **VMExit path** for the **CPU Speculation** to make the **Indirect/Generic Branch Predictor** jump to the designated **eBPF program** and peep the privilege data.

- Once the privilege data is fetched by the ***Speculation***, use the ***Side-Channel*** attack to access the **mmap user space page** to cache specific data for the malicious attacker program to **measure** the **time difference** and identify the specific bit of the privilege data.


Spectre V2 PoC
2
Prepare the hookpoint data(eBPF) for attacking specific address(e.g. core_pattern) then call vmcall(hypervisor call) which causes VM exit.
1
In the user space of VM, train the IBP to simulate the VMExit and make it run to the hook gadget to access the guessed virtual address.
6
Measure the time difference loading leak address in share_page by the side-channel trick and determine the leak bit.
Guest userspace
VMX non-root mode
Simulation of VMExit
VA
PA
0xffffffff81514edd
user(r/w/x)
ret
Guest kernel
VMX non-root mode
VMExit
Host kernel
VMX-root mode
3
4
5
Gadget
for(;;)
{
vmlaunch;
vmx_complete_atomic_exit_constprop_88(vmx);
vmx_recover_nmi_blocking(vmx);
vmx_complete_interrupts(vmx);
}
Speculation
ffffffff81514edd: mov %r9,%rsi
ffffffff81514ee0: callq *0xb0(%r8)
r8+0xb0=__bpf_prog_run, r9=bpf_insn *
__bpf_prog_run use the prebuilt bpf instructions to implement side-channel attack to leak the specific system address information to the share_page.
ex: ffffffff81c845e0 D core_pattern
ffffffffbcc845e0 63 6f 72 65 00 00 | core ....|


Jump Table Construction
vmx_vcpu_run()
{
0xfcbd
0xfcbe
}
vmx_complete_atomic_exit_constprop_88()
{ 0x4720
} 0x4734 ret
vmx_vcpu_run
0xfcbe
The number is the offset to the kvm_intel_load_address.
vmx_complete_atomic_exit_constprop_88
0x4734
0x4720
vmx_complete_atomic_exit_constprop_88
0xfcbd
vmx_vcpu_run
Iret_area
0xcf48(iret)
Ret_area
0xc3(ret)
0xcf48
0xc3
addr_iret
ss
sp
eflags
cs
ip
Refill the IBHB/BTB by constructing the iret stack with the ret instruction.
Stack of Branch Target Injection
kvm_arch_vcpu_ioctl_run -> 0xffffffff81514edd
kvm_arch_vcpu_ioctl_run <- __vmx_complete_interrupts(ret)
vmx_complete_interrupts -> __vmx_complete_interrupts(call)
vmx_vcpu_run -> vmx_complete_interrupts(call)
vmx_vcpu_run <- vmx_recover_nmi_blocking(ret)
vmx_vcpu_run -> vmx_recover_nmi_blocking(call)
vmx_vcpu_run <- vmx_complete_atomic_exit_constprop_88(ret)
vmx_vcpu_run -> vmx_complete_atomic_exit_constprop_88(call)

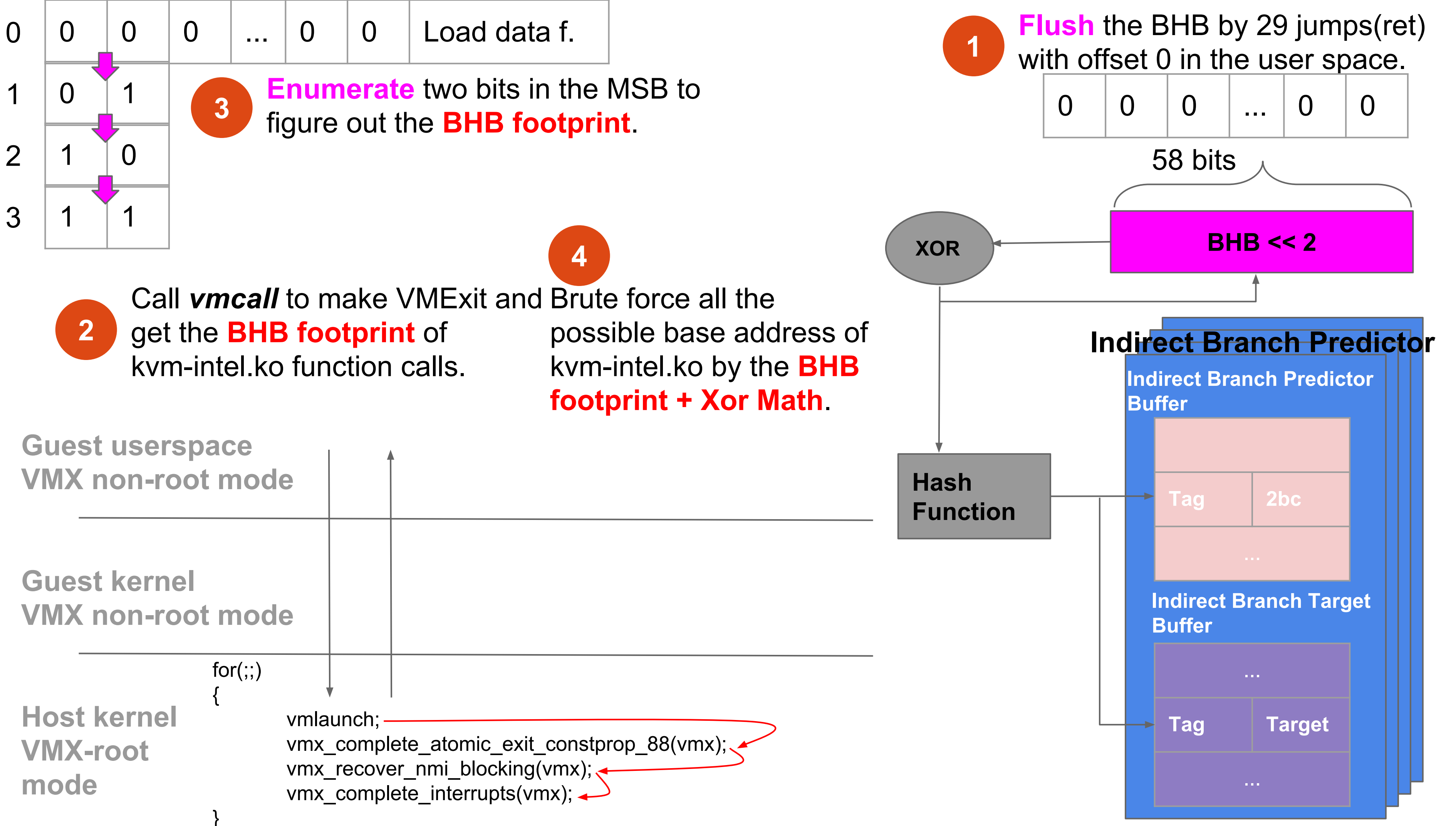
Attack KASLR and get the base address



- The exploit relies on the knowledge of the **symbol locations** of certain functions.
- With the base address of kvm-intel.ko, kvm.ko, and vmlinux, the Spectre v2 PoC can leverage the BTB and make the exploit more reliable.
- With the **vmlinux** base address, all the kernel's **global variables** address get.
- With the virtual address of the **guest share page**, the ***Side-Channel*** attack can notify the guest about the result.

How to get the base address of kvm_intel.ko?

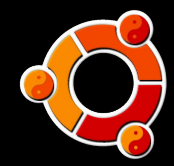
- Guess and train the **Branch History Buffer**(58 bits) by two bits at a time with Jump table setup for specific offset.
- Make the ***vmcall*** which will VMExit and go through the VMExit calling path to fill the **Branch Prediction Buffer** with **BHB footprint**.
- The BHB footprint can be figured out by the branch target injection skill.
- After the BHB footprint is figured out, the base address of kvm_intel.ko can be calculated by **comparing** BHB footprint with the emulated BHB result by brute-force trying all the possible base address and apply XOR Math operation to the sequential jumps.

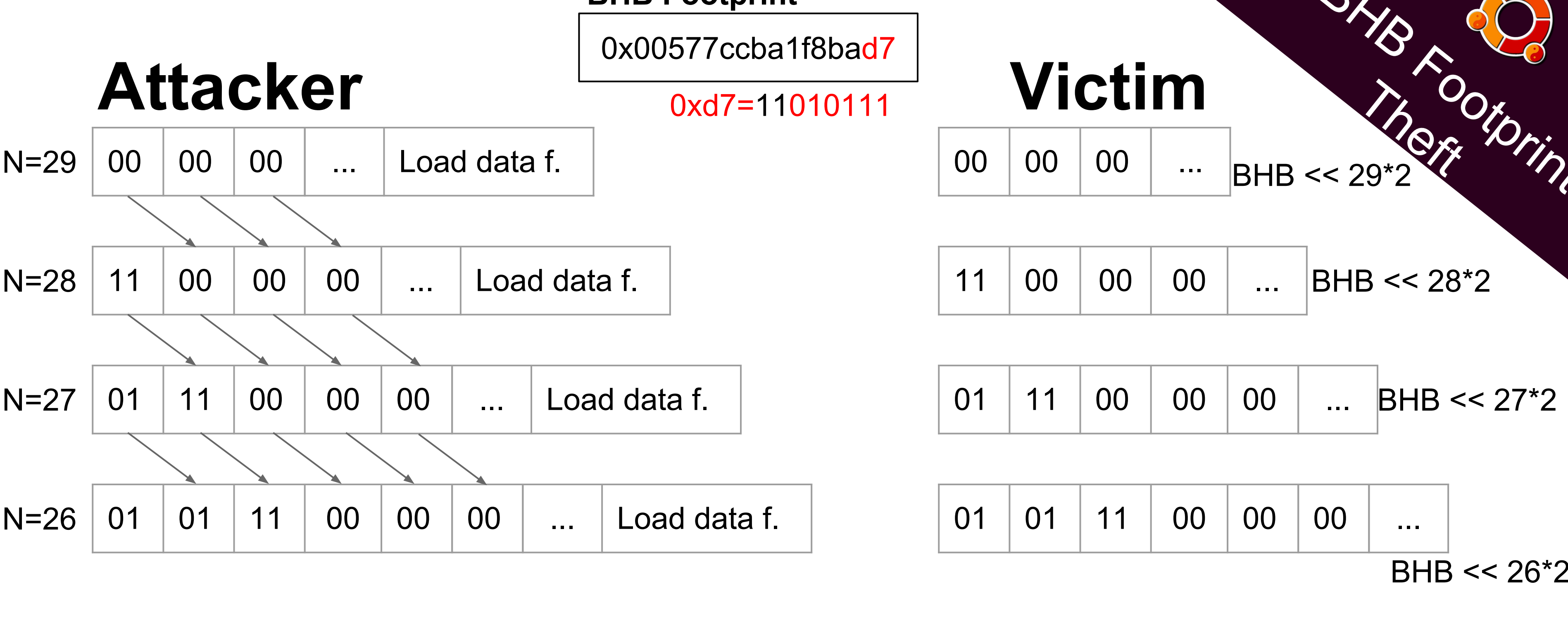


0 0 0 0 ... 0 0 Load data f.
1 0 1
2 1 0
3 1 1
1 Flush the BHB by 29 jumps(ret) with offset 0 in the user space.
0 0 0 ... 0 0
58 bits
3 Enumerate two bits in the MSB to figure out the BHB footprint.
XOR
BHB << 2
4
2 Call vmcall to make VMExit and get the BHB footprint of kvm-intel.ko function calls.
Brute force all the possible base address of kvm-intel.ko by the BHB footprint + Xor Math.
Indirect Branch Predictor
Indirect Branch Predictor Buffer
Tag
2bc
...
Guest userspace
VMX non-root mode
Hash Function
Guest kernel
VMX non-root mode
Indirect Branch Target Buffer
...
Tag
Target
...
for(;;)
{
vmlaunch;
vmx_complete_atomic_exit_constprop_88(vmx);
vmx_recover_nmi_blocking(vmx);
vmx_complete_interrupts(vmx);
}
Host kernel
VMX-root mode

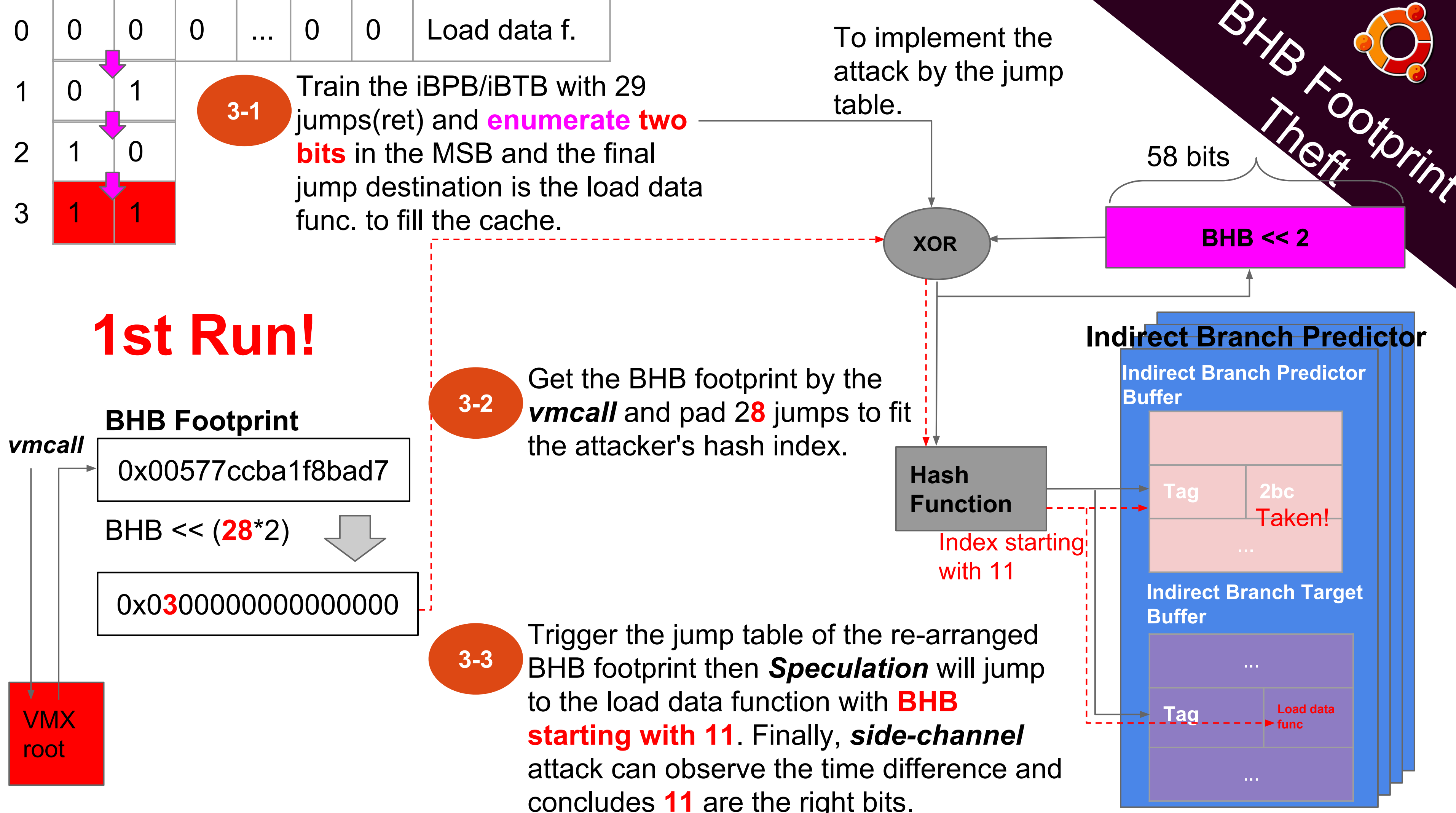
More elaborate on the Step 3

BHB Footprint Theft

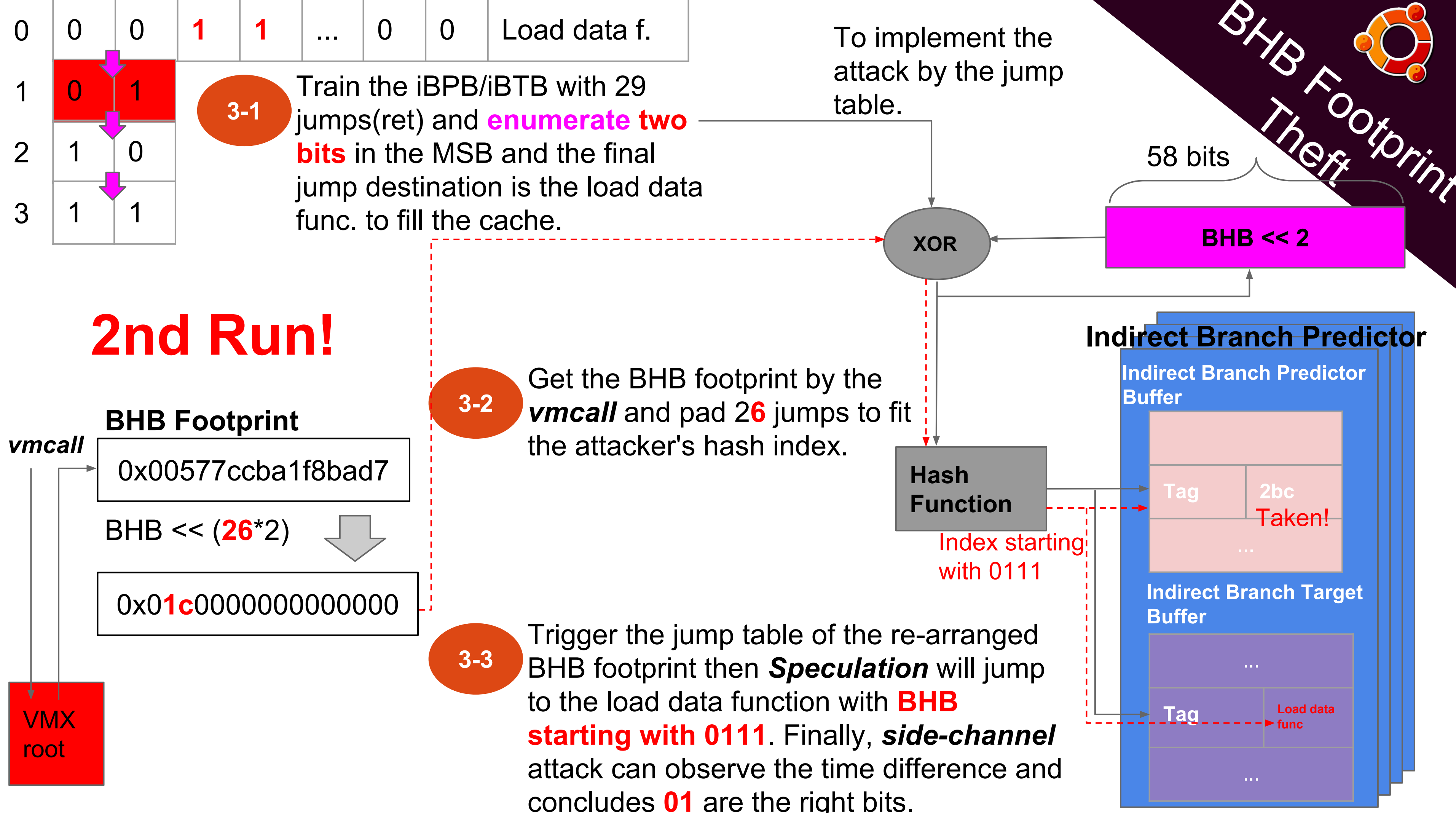




The BHB Footprint can be figured out by 29 runs. The first two runs' detail attacking process will be illustrated in the next two slides.



0
1
2
3
0 0 0 ... 0 0 Load data f.
0 1
1 0
1 1
3-1
Train the iBPB/iBTB with 29 jumps(ret) and enumerate two bits in the MSB and the final jump destination is the load data func. to fill the cache.
To implement the attack by the jump table.
BHB Footprint Theft
58 bits
XOR
BHB << 2
1st Run!
3-2
Get the BHB footprint by the vmcall and pad 28 jumps to fit the attacker's hash index.
Indirect Branch Predictor
Indirect Branch Predictor Buffer
Hash Function
Tag
2bc
Taken!
...
Index starting with 11
BHB Footprint
vmcall
0x00577ccba1f8bad7
BHB << (28*2)
0x0300000000000000
Indirect Branch Target Buffer
...
Tag
Load data func
...
3-3
Trigger the jump table of the re-arranged BHB footprint then Speculation will jump to the load data function with BHB starting with 11. Finally, side-channel attack can observe the time difference and concludes 11 are the right bits.
VMX root



BHB Footprint Theft
0
1
2
3
0
0
1
1
...
0
0
Load data f.
0
1
1
0
1
1
3-1
Train the iBPB/iBTB with 29 jumps(ret) and enumerate two bits in the MSB and the final jump destination is the load data func. to fill the cache.
To implement the attack by the jump table.
58 bits
XOR
BHB << 2
2nd Run!
3-2
Get the BHB footprint by the vmcall and pad 26 jumps to fit the attacker's hash index.
BHB Footprint
vmcall
0x00577ccba1f8bad7
BHB << (26*2)
0x01c000000000000
VMX root
Hash Function
Index starting with 0111
Indirect Branch Predictor
Indirect Branch Predictor Buffer
Tag
2bc
Taken!
...
Indirect Branch Target Buffer
...
Tag
Load data func
...
3-3
Trigger the jump table of the re-arranged BHB footprint then Speculation will jump to the load data function with BHB starting with 0111. Finally, side-channel attack can observe the time difference and concludes 01 are the right bits.

More elaborate on the Step 4

Brute Force kvm-intel.ko address

Brute Force kvm-intel.ko address

Manually construct the BHB by exclusive-or the src, dst, and the emulated BHB. Then **compare** the **emulated BHB** with the **BHB footprint** from step 3. If it's equal, we figure out the right kvm_intel_base address.

0 <= i <= 7

jumps[i].src —addr[0,19]→ XOR
jumps[i].dst —addr[0,5]→ XOR
Emulated BHB << 2 (58 bits) → XOR → Emulated BHB << 2

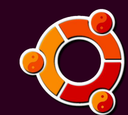
kvm_intel_base [0, 0x100000), kvm_intel_base += 0x1000

```
/* The jump table offset was collected from the objdump for the
 * record of branch sequence after VMExit
 */
struct jump jumps[] = {
  { .src = kvm_intel_base+0x0029, .dst = 0xffffffff81059620    },
  { .src = 0xffffffff81059630   , .dst = kvm_intel_base+0x002a },
  { .src = kvm_intel_base+0x002c, .dst = kvm_intel_base+0xfaf8 },
  { .src = kvm_intel_base+0xfb05, .dst = kvm_intel_base+0xf9b0 },
  { .src = kvm_intel_base+0xf9f3, .dst = kvm_intel_base+0xfb06 },
  { .src = kvm_intel_base+0xfb2d, .dst = kvm_intel_base+0xfb38 },
  { .src = kvm_intel_base+0xfb46, .dst = kvm_intel_base+0xfb4a },
  { .src = kvm_intel_base+0xfbbb, .dst = kvm_intel_base+0xfbc1 }
};
```

How to get the base address of kvm.ko and vmlinux?

- Find out the gadget point where kvm.ko calls kvm-intel.ko and vmlinux calls kvm.ko.
- Construct the instruction table in guess **source** address page and populate with **nop**.
- Construct the instruction table in guess **destination** address page and the instruction table content is to **touch the leak address for side-channel attack**.
- Invoke the ***vmcall*** and the instructions which causes ***VMExit*** to trigger the calling path and fill the BTB.
- Call guess source address and it will ***speculate*** to the destination according to the BTB entry populated in the previous step if the guess address is **matched**. (Even nop instruction is in the source address).

How to get the base address of kvm.ko?



RANGE_START
0x00001000c0000000

Jump table **source**

	0x90	
	...	
	0x90	
	0x90	

4K
0x00001000c0683390
4K
4K
0x1f390

RANGE_END
0x00001000c4000000

Speculation

Jump table **destination**

	load leak_arr	
	...	
	load leak_arr	
	load leak_arr	

0x00001000c072904a
0x4a

The code for loading leak_arr in the destination will be ***speculated*** according to the BTB if the offset is matched(even the source instruction is **nop/0x90**.) and ***side-channel*** attack can detect the cache existence of leak_arr. Otherwise, go to Step 2.

2 Dereferencing(***source**) the next source address(+4k).

1 Call ***vmcall*** to trigger VMExit and then fill the gadget to BTB entry.

Guest userspace
VMX non-root mode

Guest kernel
VMX non-root mode

Host kernel
VMX-root mode

callq *0x2b0(%rax)

kvm_arch_vcpu_ioctl_run
0xffffffffc0683390

vmx_handle_external_intr
0xffffffffc072904a

Fill the BTB entry

Branch Target Buffer

c0683390	c072904a

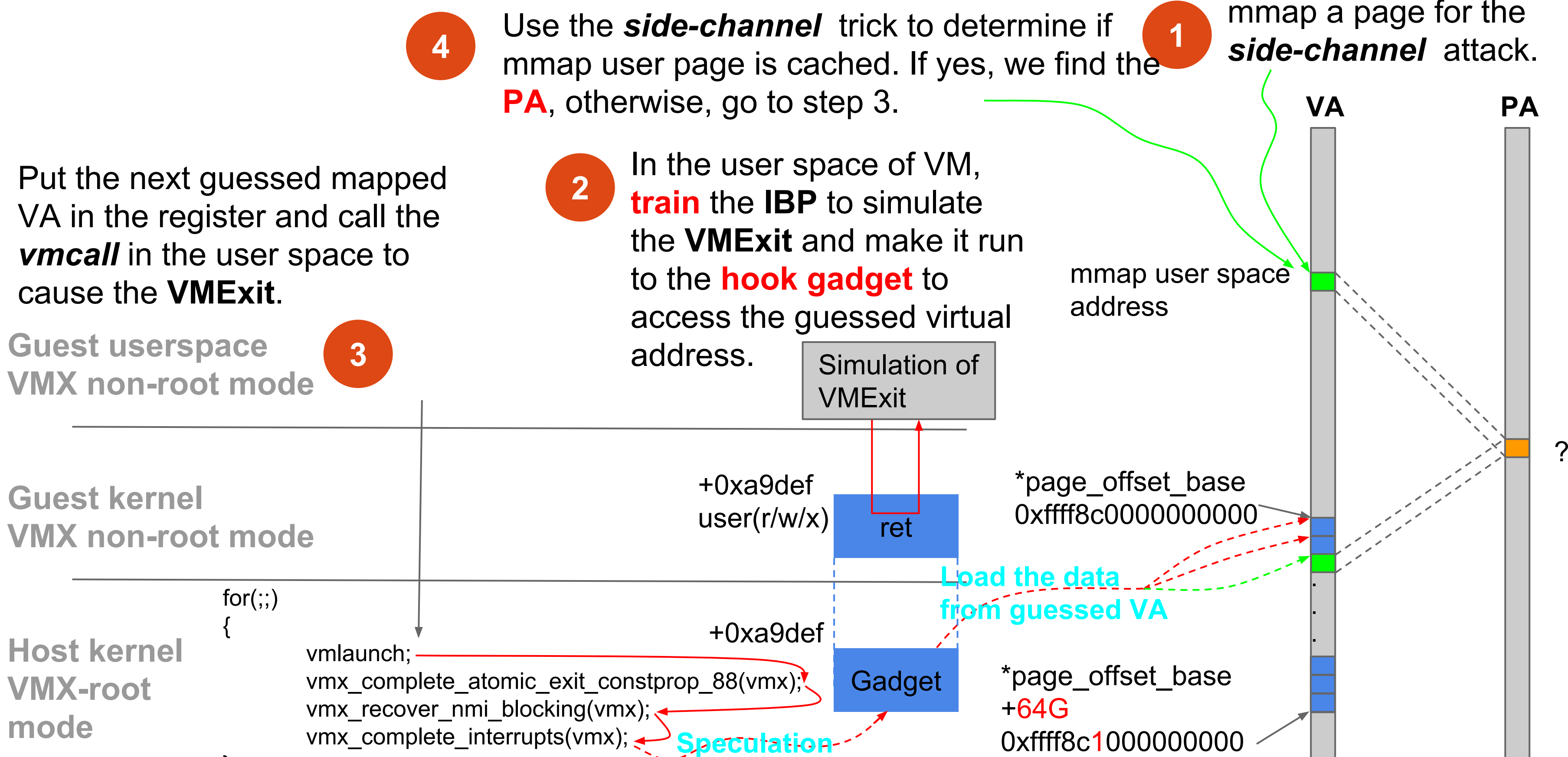
How to get the physical address of mmap user space virtual address?

- mmap a page and the page will be used in the ***side-channel*** attack for recording the result of the bit identification of victim's data.

- Try to find the **physical address** mapping of the page. As the physical address will be useful to find the PAGE_OFFSET in the next attack.

- After we get the PAGE_OFFSET, we can figure out the virtual address(**PAGE_OFFSET + PA**) for the ***side-channel*** attack in the kernel to access the page.

How to get the physical address of mmap user space virtual address?



How to get the physical address of mmap user space virtual address?

- What's the **_GADGET_** for accessing the kernel VA of specific **guess** physical address?

```
$ grep -i a6b3c0 boot/System.map-4.9.0-3-amd64
ffffffff81a6b3c0 R __per_cpu_offset(PER_CPU_OFFSET_ADDRESS;)

0 - ffffffff81a6b3c0 = 0x7e594c40
```

```
unsigned long r8val = phys_addr - 0xf8UL;
unsigned long rop_relocated_immediate = 0UL - (unsigned long)PER_CPU_OFFSET_ADDRESS;
unsigned long r9val = ((rop_relocated_immediate + PAGE_OFFSET_BASE_ADDRESS) / 8UL);
train_mispredict_and_hypercall(r8val, r9val, PHYS_LOAD_ADDRESS); # ffffffff810a9def
  /*
   * ffffffff810a9def:      4c 89 c0          mov    %r8,%rax # r8=phys_addr - 0xf8UL
   * ffffffff810a9df2:      4d 63 f9          movslq %r9d,%r15
                                            # (rop_relocated_immediat(0x7e594c40) + PAGE_OFFSET_BASE_ADDRESS) / 8UL
   * ffffffff810a9df5:      4e 8b 04 fd c0 b3 a6  mov -0x7e594c40(,%r15,8),%r8
                                                  # %r8 = %r15 * 8 - 0x7e594c40 = *(PAGE_OFFSET_BASE_ADDRESS)
   * ffffffff810a9dfc:      81
   * ffffffff810a9dfd:      4a 8d 3c 00           lea (%rax,%r8,1),%rdi
   * ffffffff810a9e01:      4d 8b a4 00 f8 00 00  mov 0xf8(%r8,%rax,1),%r12
                                                  # %r12 = *(phys_addr + PAGE_OFFSET_BASE_ADDRESS)
   */
```

How to get the page_offset virtual address?

1. The **PA(physical address)** of the mmap page is known from the previous step.
2. In the user space of VM, **train** the **IBP** to simulate the **VMExit** and make it run to the **hook gadget** to access the guessed virtual address.
3. Put the next **guess** mapped VA(**Old PAGE_OFFSET+1GB+PA**) in the register and call the ***vmcall*** in the user space to cause the **VMExit**.
4. Use the ***side-channel*** attack to determine if mmap user page is cached. If yes, we find the **PAGE_OFFSET**, otherwise, go to step 3.

Guest userspace
VMX non-root mode

Guest kernel
VMX non-root mode

Host kernel
VMX-root
mode

VMExit

Simulation of VMExit

+0x2c9d3
user(r/w/x)

ret

+0x2c9d3

Gadget

```
for(;;)
{
    vmlaunch;
    vmx_complete_atomic_exit_constprop_88(vmx);
    vmx_recover_nmi_blocking(vmx);
    vmx_complete_interrupts(vmx);
}
```

Speculation

Load the data from guessed VA

VA

PA

mmap user space address

0xFFFF880000000000

Shift 1GB

1GB

1GB

0xFFFFFFFFFFFFFFFF

PA

Indirect Branch Prediction and Intel® Hyper-Threading Technology (Intel® HT Technology)

Physical Core 0

Indirect Branch Predictor

Logic Core 0

Logic Core 1

Physical Core 1

Indirect Branch Predictor

Logic Core 0

Logic Core 1

X

- In a processor supporting Intel ® Hyper-Threading Technology, **a core** (or physical processor) may include **multiple logical processors**.
- In such a processor, the **logical processors** sharing a core may **share indirect branch predictors**. As a result of this sharing, software on one of a core's logical processors may be able to **control** the predicted target of an indirect branch executed on **another logical processor** of the **same core.** This sharing occurs only within a core.
- Software executing on a **logical processor** of one core **cannot** control the predicted target of an indirect branch by a logical processor of a **different core**.

Indirect Branch Prediction

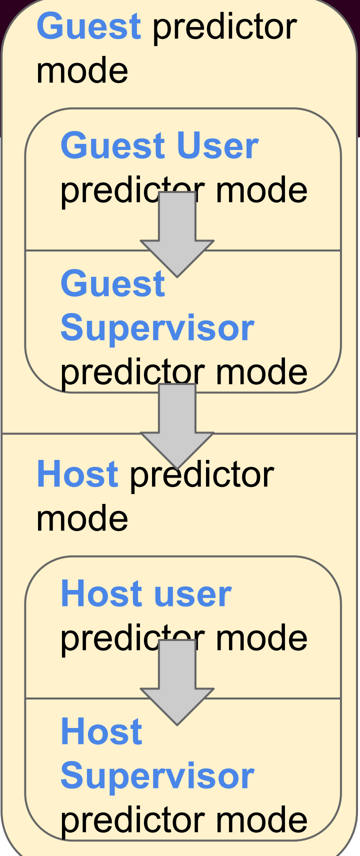
The processor uses indirect branch predictors to control only the operation of the branch instructions enumerated in the table below.

Table 2-1. Instructions that use Indirect Branch Predictors

Branch Type	Instruction	Opcode
Near Call Indirect	CALL r/m16, CALL r/m32, CALL r/m64	FF /2
Near Jump Indirect	JMP r/m16, JMP r/m32, JMP r/m64	FF /4
Near Return	RET, RET Imm16	C3, C2 Iw

Predictor Mode

- To prevent attacks based on branch target injection, it can be important to ensure that less privileged software **cannot** control use of the branch predictors by more privileged software. For this reason, it is useful to introduce the concept of predictor mode.
- There are **four** predictor modes:
  - 1). host-supervisor, 2). host-user, 3). guest-supervisor, and 4). guest-user.
- The **guest predictor** modes are considered **less privileged** than the **host predictor** modes. Similarly, the user predictor modes are considered less privileged than the supervisor predictor modes.



Mitigation of Spectre V2

1. **Retpoline**
   a. Kernel patch(inline assembly with volatile flag cannot be affected by the recompilation): [PATCH v6 00/10] Retpoline: Avoid speculative indirect calls in kernel
   b. Kernel space/user space recompilation with retpoline enabled GCC compiler.
   c. 2018 Feb 21: gcc packages supporting retpoline options for x86 published to Ubuntu 17.10, Ubuntu 16.04 LTS, and Ubuntu 14.04 LTS.[KB]
2. **Microcode + IBRS patch**
3. **Microcode + IBPB patch**

Mitigation of Spectre V2

Retpoline Theory

<table>
<tr><th colspan="2">Indirect branch construction</th></tr>
<tr><td>origin</td><td>retpoline</td></tr>
<tr><td>jmp *%r11</td><td>call set_up_target; (1)
capture_spec: (4)
pause;
jmp capture_spec;
set_up_target:
mov %r11, (%rsp); (2)
ret; (3)</td></tr>
</table>

Mitigation of Spectre V2

Retpoline cost

- To compile the whole system with the retpoline enabled GCC compiler[pjt].

- And obviously, if you **do not** have the **sources** to the target **you are trying** to protect, **IBRS** allows you to run it in a protected fashion --while it cannot easily be retpolined or the software **cannot be rebuilt** for some reasons[pjt].

Mitigation of Spectre V2 - IBRS V.S. IBPB

The CPUID instruction enumerates support for the indirect branch control mechanisms using three feature flags in CPUID.(EAX=7H,ECX=0):EDX:[tim][intel]

Table 2-2. CPUID Leaf 07H, Sub-leaf 0: Updated EDX Register Details

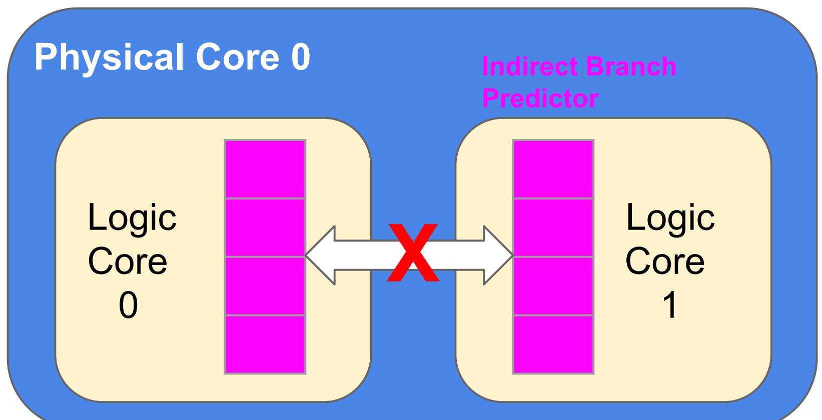
Initial EAX Value	Information Provided About the Processor	
Structured Extended Feature Flags Enumeration Leaf (Output depends on ECX input value)		
07H		**NOTES:** Leaf 07H main leaf (ECX = 0). If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0.
	EDX	Bits 25-00: Reserved Bit 26: IBRS and IBPB supported Bit 27: STIBP supported Bit 28: Reserved Bit 29: IA32_ARCH_CAPABILITIES supported Bits 31-30: Reserved

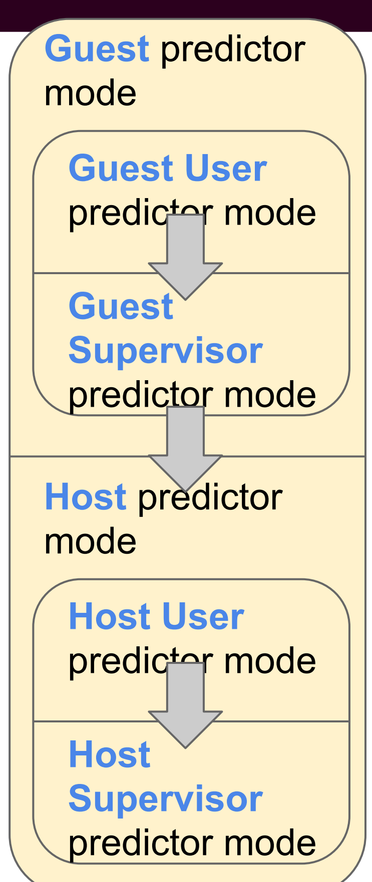
Mitigation of Spectre V2 - IBRS

If software sets **IA32_SPEC_CTRL.IBRS** to 1 after a transition to a more privileged predictor mode, predicted targets of indirect branches executed in that predictor mode with IA32_SPEC_CTRL.IBRS =1 **cannot** be controlled by software that was executed[intel]

1). In a **less privileged** predictor mode

2). On **another** logical processor.(imply **STIBP**)





Mitigation of Spectre V2 - IBRS

Support Based on Software Enabling[intel]

- If IA32_SPEC_CTRL.IBRS is **already** 1 before a transition to a more privileged predictor mode, **some processors** may allow the predicted targets of indirect branches executed in that predictor mode to be controlled by software that executed before the transition. It is **not** necessary to clear the bit first; **writing** it with a value of 1 after the transition suffices, regardless of the bit's original value.
- **Enhanced IBRS** doesn't need to write to the register every time when entering the mode from lower privilege to higher privilege. A processor supports enhanced IBRS if RDMSR returns a value of 1 for **bit 1** of the **IA32_ARCH_CAPABILITIES** MSR.

Mitigation of Spectre V2 - IBRS

IBRS limitation with RSB

- Setting IA32_SPEC_CTRL.IBRS to 1 **does not suffice** to prevent the predicted target of a **near return** from using an **RSB entry** created in a less privileged predictor mode.

- Software can avoid this by using:
  - An **RSB overwrite sequence** following a transition to a more privileged predictor mode.
  - if supervisor-mode execution prevention (**SMEP**) is enabled.

Mitigation of Spectre V2 - IBRS

IBRS(Indirect Branch Restricted Speculation) cost

“noibrs”/ibrs_enabled controls the IBRS feature in the SPEC_CTRL model-specific register (MSR) when SPEC_CTRL is present in cpuid (post microcode update). When ibrs_enabled is set to 1 the **kernel** runs with **indirect branch restricted speculation**, which **protects** the **kernel space** from attacks (even from hyperthreading/simultaneous multi-threading attacks). When IBRS is set to 2, **both userland and kernel** runs with indirect branch restricted speculation. This protects **userspace from hyperthreading/simultaneous multi-threading attacks** as well, and is also the default on AMD processors (family 10h, 12h and 16h). This feature addresses CVE-2017-5715, variant #2.

echo 0 > /sys/kernel/debug/ibrs_enabled will turn off IBRS
echo 1 > /sys/kernel/debug/ibrs_enabled will turn on IBRS in kernel
echo 2 > /sys/kernel/debug/ibrs_enabled will turn on IBRS in both userspace and kernel

Mitigation of Spectre V2 - IBRS

IBRS(Indirect Branch Restricted Speculation) cost

- The cost of IBRS performance **varies** with processor generation. **Skylake** incurs the **least overhead**. It is expected that future generations will be better still[pjt]. Behavior with exhausted return stack predictors is not well-specified, which means we must potentially avoid underflow. For example, in the case that the hardware chose to instead turn to another predictor.

- A naked **indirect call** (with **IBRS enabled**) on Skylake and a **retpolined call** have approximately **the same** cost.(I have not compared this cost for pre-Skylake uarchs.)[pjt]

- The transition cost for enabling/disabling the feature as we schedule into (and out of) **protected code**[pjt].

Mitigation of Spectre V2 - IBPB

Indirect branch predictor barrier (IBPB)[intel]

- Enabling IBRS **does not** prevent software from controlling the predicted targets of indirect branches of unrelated software executed later at **the same predictor mode** (for example, between **two different user applications, or two different virtual machines**). Such isolation can be ensured through use of the IBPB command.
- The **indirect branch predictor barrier (IBPB)** is an indirect branch control mechanism that establishes a barrier, **preventing** software that executed **before** the barrier from controlling the predicted targets of indirect branches executed after the barrier.

Guest predictor mode

Guest User predictor mode

Guest Supervisor predictor mode

Host predictor mode

Hsot User predictor mode

Host Supervisor predictor mode

Mitigation of Spectre V2 - IBPB

- IBPB can be used in conjunction with IBRS to account for cases that IBRS does not cover:[intel]
    - IBRS does not prevent software from controlling the predicted target of an indirect branch of unrelated software (e.g., a different user application or a different virtual machine) executed at **the same predictor** mode. Software can prevent such control by executing an IBPB command when changing the identity of software operating at a particular predictor mode (e.g., when **changing user applications** or **virtual machines**).
    - Software may choose to **clear** **IA32_SPEC_CTRL.IBRS** in certain situations (e.g., for executionwith **CPL = 3 in VMX root operation)**. In such cases, software can use an **IBPB** command on certain transitions (e.g., after running an untrusted virtual machine) to prevent software that executed earlier from controlling the predicted targets of indirect branches executed subsequently with IBRS disabled.

Mitigation of Spectre V2 - IBPB use cases

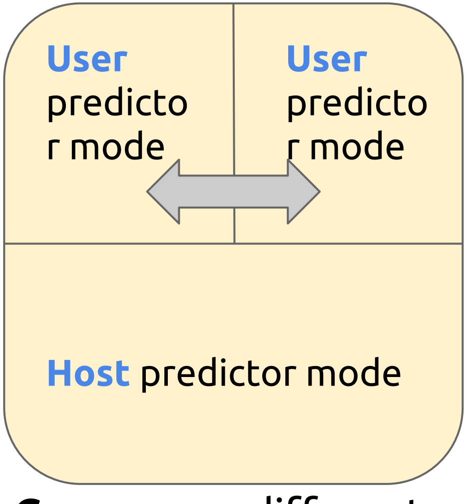
Guest predictor mode

Host predictor mode

Guest predictor mode

Host predictor mode

Case two: a different virtual machine switch.



Case one: a different user application switch.

Indirect Branch Prediction Barriers (ibpb)

“noibpb”/ibpb_enabled controls the IBPB feature in the PRED_CMD model-specific register (MSR) if either IBPB_SUPPORT or SPEC_CTRL is present in cpuid (post microcode update). When ibpb_enabled is set to 1, an IBPB barrier that flushes the contents of the indirect branch prediction is run across user mode or guest mode context switches to prevent user and guest mode from attacking other applications or virtual machines on the same host. In order to protect virtual machines from other virtual machines, ibpb_enabled=1 is needed even if ibrs_enabled is set to 2. If ibpb_enabled is set to 2, indirect branch prediction barriers are used instead of IBRS at all kernel and hypervisor entry points (in fact, this setting also forces ibrs_enabled to 0). ibpb_enabled=2 is the default on CPUs that don’t have the SPEC_CTRL feature but only IBPB_SUPPORT. ibpb_enabled=2 doesn’t protect the kernel against attacks based on simultaneous multi-threading (SMT, also known as hyperthreading); therefore, ibpb_enabled=2 provides less complete protection unless SMT is also disabled. This feature addresses CVE-2017-5715, variant #2.

Customer and vendors can disable the ibpb implementation in microcode by passing "noibpb" to the kernel command line at boot, or dynamically with the debugfs control below:

```
# echo 0 > /sys/kernel/debug/x86/ibpb_enabled
```

Mitigation of Spectre V2

IBRS V.S. Retpoline on Skylake platform

- The problem with **Skylake+** is that an **RSB underflow** falls back to using a **BTB prediction**, which allows the attacker to take control of speculation[A.C].

- The concern is that the attacker could **poison** the **BTB** for a 'ret' instruction, as in the general case of the SP2 (conditional branch misprediction) attack, so that it predicts a branch to an address of the attacker's choice. Now *most* of the time, one might expect the target for that 'ret' to come from the **RSB**. But if there is a way to **force** the **RSB** to **empty**, or the attacker is just happy to keep trying, and wait for things like SMI to make it work every now and then, then it *might* be exploitable.
- Also remember that **sibling threads** share a **BTB**, so you **can't rely** on isolated straight-line codepath on the current cpu for safety. (e.g. by **issuing an IBPB** on every entry to supervisor mode).[A.C] **<-- IBRS is the only way to mitigate Spectre V2 on Skylake?**

Mitigation of Spectre V2

Skylake limitation on retpoline underflow - refill the RSB
d1c99108af3c Revert "x86/retpoline: Simplify vmexit_fill_RSB()"

- To prevent this, in some cases it may be necessary to “**refill**” the return stack to guarantee that underflow cannot occur[pjt]. Cases which this applies to include:
  - When we transfer control into **protected execution** (so that we do not perturb the steps that it may have taken to preserve the integrity of their hardware return prediction).
  - Guest to **hypervisor transitions**, context-switches into a protected process, interrupt delivery (and return).
  - When we resume from a hardware sleep state which may not have preserved this cache (e.g., **mwait**).
  - When natural execution potentially **exhausts the return stack** in a protected application. (Note that this is a particular corner case, with more limited exploitability -- we expect that most binaries deploying retpoline protections will not require this specific mitigation.)

Mitigation of Spectre V2

IBRS V.S. Retpoline on Skylake platform

- On Skylake the target for a 'ret' instruction may also come from the BTB. So if you ever let the RSB (which remembers where the 'call's came from get empty, you end up vulnerable.) Other than the obvious call stack of more than **16 calls in depth**, there's also a big list of other things which can empty the RSB, including an **SMI**. Which basically makes **retpoline** on Skylake+ ***very* hard to use** reliably. The plan is to use **IBRS** there and not retpoline[woodhouse].

- A **naked indirect call** (with **IBRS enabled**) on Skylake and **a retpolined call** have approximately **the same cost**[pjt].

Spectre V2 - Microcode

- **Intel Microcode fix (#intel-microcode 3.20180108.0~ubuntu16.04.2)**
  - sudo apt install -y intel-microcode
    1. install the initramfs-tools, iucode-tool and intel-microcode packages;
    2. configure the system to use a initramfs created by initramfs-tools during boot (Debian kernels do this by default);
    3. make sure the initramfs for the kernel you will use was updated (it should have been done automatically for the default boot kernel, at least for Debian kernels), using "update-initramfs -u", and possibly "update-initramfs -u -k <kernel version>"
    4. reboot.
  - Where is the Intel microcode being installed?
    - /lib/firmware/intel-ucode/
  - How to disable the intel microcode loading at boot time:
    - "IUCODE_TOOL_INITRAMFS=no" in /etc/default/intel-microcode
    - Or purge the package.

Spectre V2 - Microcode

- **Intel Microcode fix (#intel-microcode 3.20180108.0~ubuntu16.04.2)**
  - By default, the “iucool_tool --scan-system” scan the underlying CPU architecture to load specific microcode. “IUCODE_TOOL_SCANCPUS=no” will install all micro codes or installed specific micro code by ‘IUCODE_TOOL_EXTRA_OPTIONS=""’.
  - By default, the microcode module is disabled in modprobe.d/intel-microcode-blacklist.conf:
    - The microcode module attempts to apply a microcode update when it autoloads. This is not always safe, so we block it by default.
  - In v4.4-rc1, the config name changed from MICROCODE_INTEL_EARLY to MICROCODE_INTEL
    - $ git describe --contains fe055896c040df571e4ff56fb196d6845130057b v4.4-rc1~154^2~6

Intel: We now won't ever patch Spectre variant 2 flaw in these chips

Microcode Revision Guidance - April 2 2018

After a comprehensive investigation of the microarchitectures and microcode capabilities for these products, Intel has determined to **not release** microcode updates for these products for one or more reasons including, but not limited to the following:

- **Micro-architectural characteristics** that **preclude** a **practical implementation** of features mitigating Variant 2 (CVE-2017-5715)
- Limited Commercially Available System Software support
- Based on customer inputs, most of these products are implemented as “closed systems” and therefore are expected to have a lower likelihood of exposure to these vulnerabilities.

The list of CPU families Intel **won't** patch are:

Bloomfield, Bloomfield Xeon, Clarksfield, Gulftown, Harpertown Xeon C0, Harpertown Xeon E0, Jasper Forest, Penryn/QC, SoFIA 3GR, Wolfdale C0, Wolfdale M0, Wolfdale E0, Wolfdale R0, Wolfdale Xeon C0, Wolfdale Xeon E0, Yorkfield, Yorkfield Xeon.

Variant 3: rogue data cache load A.K.A MeltDown(CVE-2017-5754)

Meltdown PoC

CPUs: information leak using speculative execution - Comment 2 is the PoC provided by Jann Horn.

https://github.com/paboldin/meltdown-exploit The meltdown PoC summarized from the spectre paper.

Meltdown mitigation

- **Variant 3: rogue data cache load (CVE-2017-5754)**
  - Mitigated by kernel **pti** patch set(previous name is **KAISER**)
  - [patch 00/60] x86/kpti: Kernel Page Table Isolation (was KAISER)
- All of the **stable kernel backports** are based on **KAISER**
  - Dave Hansen backported to v4.14
    - [PATCH 00/23] KAISER: unmap most of the kernel from userspace page tables
    - KAISER: hiding the kernel from user space
  - GregKH to 4.{4,9,14}
    - [PATCH 4.4 00/37] 4.4.110-stable review
  - Sasha Levin to 4.1
  - Hugh Dickins to 3.18 (git tree cannot be found?)
  - Ben Hutchings to 3.{2,16}
  - Juerg Haefliger to 3.13

Meltdown mitigation

Page Table Isolation (pti)

“nopti”/pti_enabled controls the Kernel Page Table Isolation feature,which isolates kernel pagetables when running in userland. This feature addresses CVE-2017-5754, also called variant #3, or Meltdown.

Customers and vendors can disable the PTI feature by passing “nopti” to the kernel command line at boot, or dynamically with the runtime debugfs control below:

```
# echo 0 > /sys/kernel/debug/x86/pti_enabled
```

Performance impact

- Facts About the New Security Research Findings and Intel® Products
- Firmware Updates and Initial Performance Data for Data Center Systems
- Intel Security Issue Update: Initial Performance Data Results for Client Systems
- Speculative Execution Exploit Performance Impacts - Describing the performance impacts to security patches for CVE-2017-5754 CVE-2017-5753 and CVE-2017-5715

Variant 4: Speculative Store Bypass (CVE-2018-3639)

Overview of Speculative Store Bypass

Assume that a key K exists. The attacker is allowed to know the value of M, but not the value

of key K. X is a variable in memory.

1. X = &K; // Attacker manages to get variable with address of K stored into pointer X

<at some later point>

2. X = &M; // Does a store of address of M to pointer X

3. Y = Array[*X & 0xFFFF]; // Dereferences address of M which is in pointer X in order to

// load from array at index specified by M[15:0]

When the above code runs, the load from address X that occurs as part of step 3 may execute speculatively and, due to memory disambiguation, initially receive a value of address of K instead of the address of M. When this value of address of K is dereferenced, the array is speculatively accessed with an index of K[15:0] instead of M[15:0]. The CPU will later reexecute the load from address X and use M[15:0] as the index into the array. However, the cache movement caused by the earlier speculative access to the array may be analyzed by the attacker to infer information about K[15:0].

Ref: [11][13][14] PoC:[12]

Supplementary Information

Spectre

(Spectre / Variant 2 Mitigation with IBRS)

Spectre / Variant 2 Mitigation with IBRS

CPU	Microcode update	Use retpoline	IBRS Dynamic	IBRS Always On
non-Intel	?	Yes, default to conservative?	?	?
All Current Intel CPUs	Yes	Yes, **Default**	If IBRS available, users can opt-in	Opt-in if more paranoid
Future Work	N/A	Not needed, but not harmful	Not recommended	**Default** if CPU enumerates “cheaper” IBRS

Affected ARM Cores of Spectre & Meltdown

Processor	Variant 1	Variant 2	Variant 3	Variant 3a
Cortex-R7	Yes*	Yes*	No	No
Cortex-R8	Yes*	Yes*	No	No
Cortex-A8	Yes (under review)	Yes	No	No
Cortex-A9	Yes	Yes	No	No
Cortex-A15	Yes (under review)	Yes	No	Yes
Cortex-A17	Yes	Yes	No	No
Cortex-A57	Yes	Yes	No	Yes
Cortex-A72	Yes	Yes	No	Yes
Cortex-A73	Yes	Yes	No	No
Cortex-A75	Yes	Yes	Yes	No

https://developer.arm.com/support/security-update

References

[1]. Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR

[2]. MAMAS - Computer Architecture - Branch Prediction - by Dr. Avi Mendelson

[3]. Computer Structure - Advanced Branch Prediction - by Lihu Rappoport and Adi Yoaz

[4]. ECE 4100/6100 Advanced computer architecture - Prof. Hsien Hsin Lee - L12 P6 and NetBurst Microarchitecture

[5]. Reading privileged memory with a side-channel - by Jann Horn

[6]. Cache Side-Channel Attacks and the case of Rowhammer Daniel Gruss IAIK, Graz University of Technology

[7]. Last-Level Cache Side-Channel Attacks are Practical - paper

[8]. Last-level cache side channel attacks are practical - slide - Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser and Ruby B. Lee

[9]. CPUs: information leak using speculative execution by Jann Horn - Comment 5

[10]. http://snapshot.debian.org/archive/debian/20170701T224614Z/pool/main/l/linux/linux-image-4.9.0-3-amd64_4.9.30-2%2Bdeb9u2_amd64.deb

[11]. Speculative Execution Side Channel Mitigations. Revision 2.0 - Intel

[12]. speculative execution, variant 4: speculative store bypass

[13]. Intel Analysis of Speculative Execution Side Channels. Revision 3.0 - Intel

[14]. Ubuntu KB - Spectre v4

Branch prediction Related Reference

[15]. Com 506 Computer Design - Lecture 3 Branch Prediction - Prof. Taeweon Suh

Branch Prediction Related Reference

[16]. Lecture 9 - Branch Prediction

[17]. CPE 631: Branch Prediction - University of Alabama in HuntsVille - Aleksandar Milenkovic

[18]. Computer Architecture - Advanced Branch Prediction - by Dan Tsafrir - 2012/05/21

[19]. Dynamic Branch Predictor - EE524 - Cpts561 Computer Architecture

[20]. Determining Branch Direction

[21]. 18-447 Computer Architecture - Lecture 10 - Branch Prediction II - Prof. Onur Mutlu - Carnegie Mellon University

[22]. ECE 4100/6100 Advanced computer architecture - Lecture 5 Branch Prediction - Prof. Hsien Hsin Lee - Georgia Institute of Technology

Branch Prediction Related Reference

[23]. Microbenchmark and Mechanisms for reverse engineering of branch predictor structure - LaCASA laboratory - Alabama Huntsville

[24]. Prediction and Speculation

[25]. Advanced Microarchitecture - Lecture 4 Branch Predictor - Georgia Technology Institute

[26]. Advanced Branch Predictors - Guang Pan, Ming Lu

[27]. Computer Architecture CS6354 - Branch Prediction - Samira Khan - University of Virginia

[28]. Bimode cascading: Adaptive Rehashing for ITTAGE Indirect Branch Predictor - The university of Tokyo