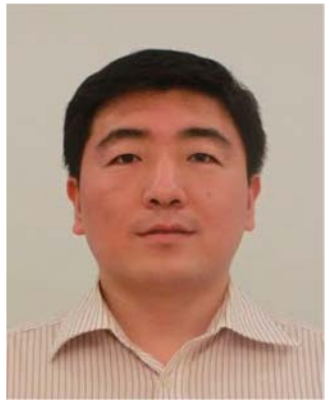
Oh No! KPTI Defeated Unauthorized Data Leakage is Still Possible

Yueqiang Cheng, Zhaofeng Chen, Yulong Zhang, Yu Ding, Tao Wei

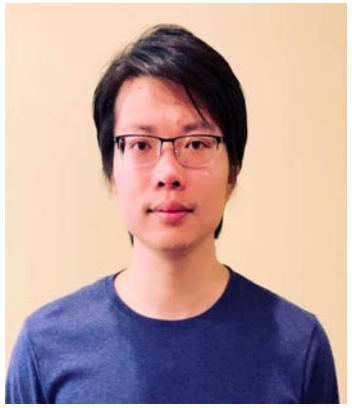
Baidu Security

About Speakers

Baidu X-Lab



Dr. Yueqiang Cheng



Mr. Zhaofeng Chen

Mr. Yulong Zhang

Dr. Yu Ding

Dr. Tao Wei

Our Security Projects:

MesaLock

MesaLink

MesaPy

MesaTEE

MesaArmor

How to Read Unauthorized Data From Unprivileged App?

Specific Settings

In kernel space, we have a
secret msg, e.g., **xlabsecretxlabsecret**,
location is at, e.g., **0xffffffffc0e7e0a0**

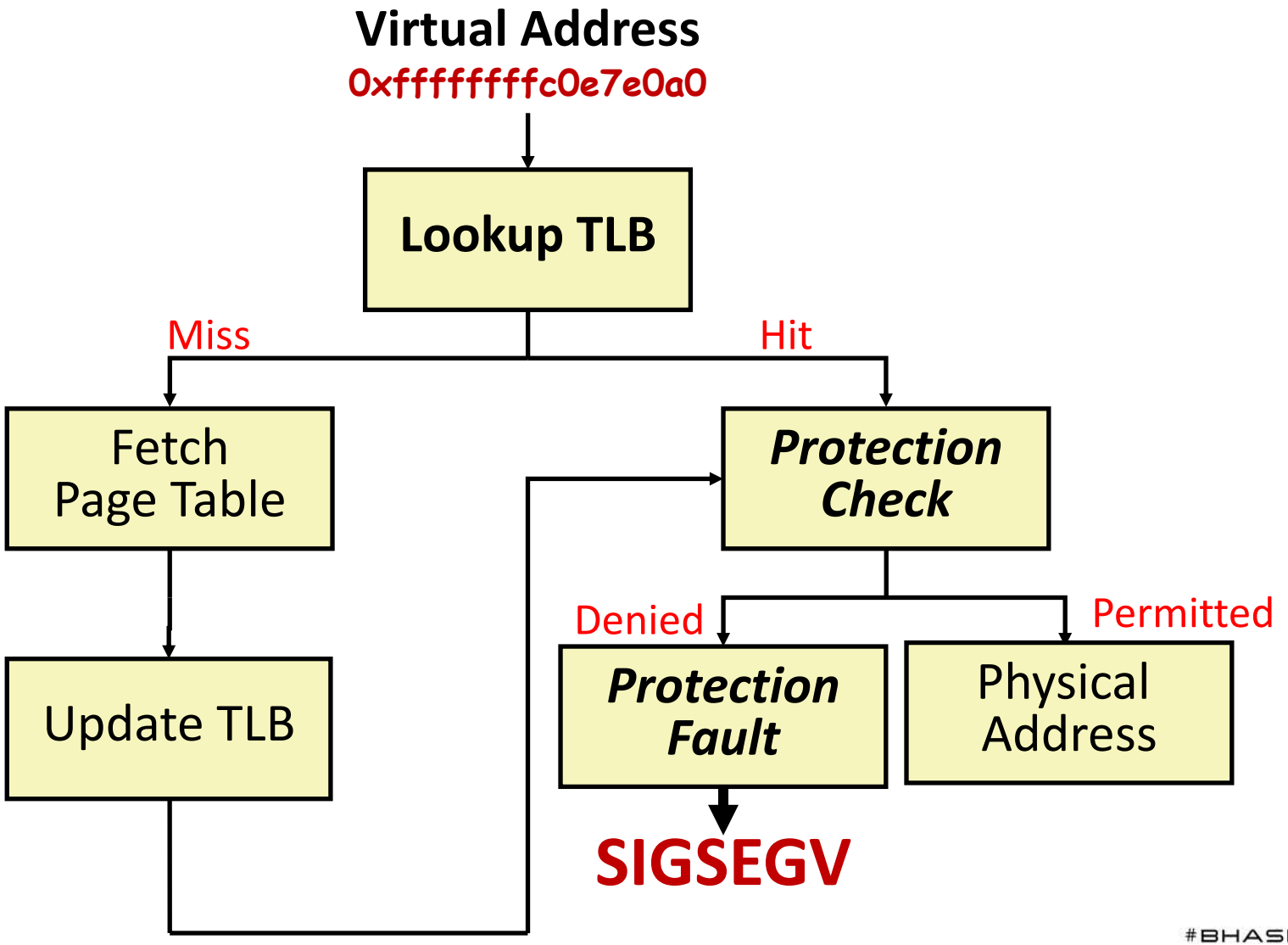
Kernel is bug-free:
there is no vulnerability for user application to arbitrarily read kernel space

A Rough Attempt

Simple C code:

```
char* ptr = (char*) 0xffffffffc0e7e0a0;
printf("%c\n", (*ptr));
```

What Really Happened



What Really Happened

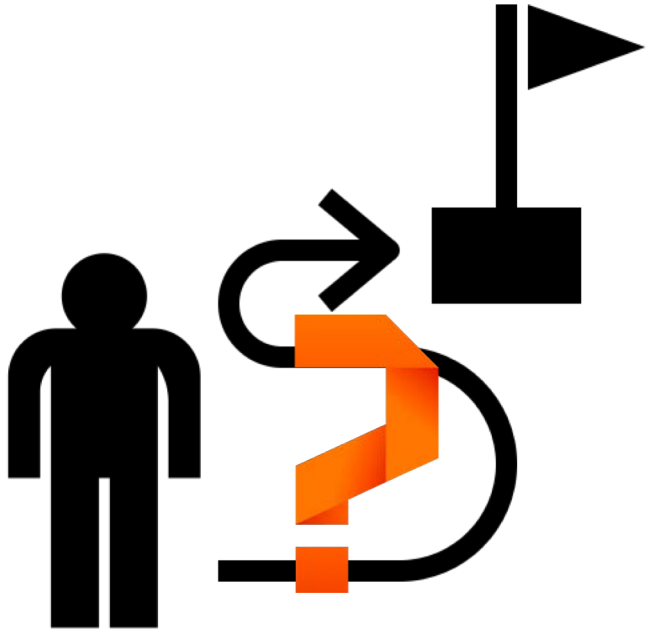
1: Page Table Permissions

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)						Bits 39:32 of address[2]			PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

2: Control Registers, e.g., SMAP in CR4

Image from Intel sdm

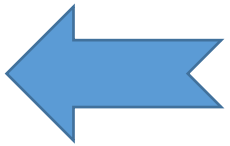
No Way to Go?



1. Unprivileged App +
2. Permission Checking +
3. Bug-free Kernel

However, in order to gain high performance, CPU …

1. Unprivileged App +
2. Permission Checking +
3. Bug-free Kernel

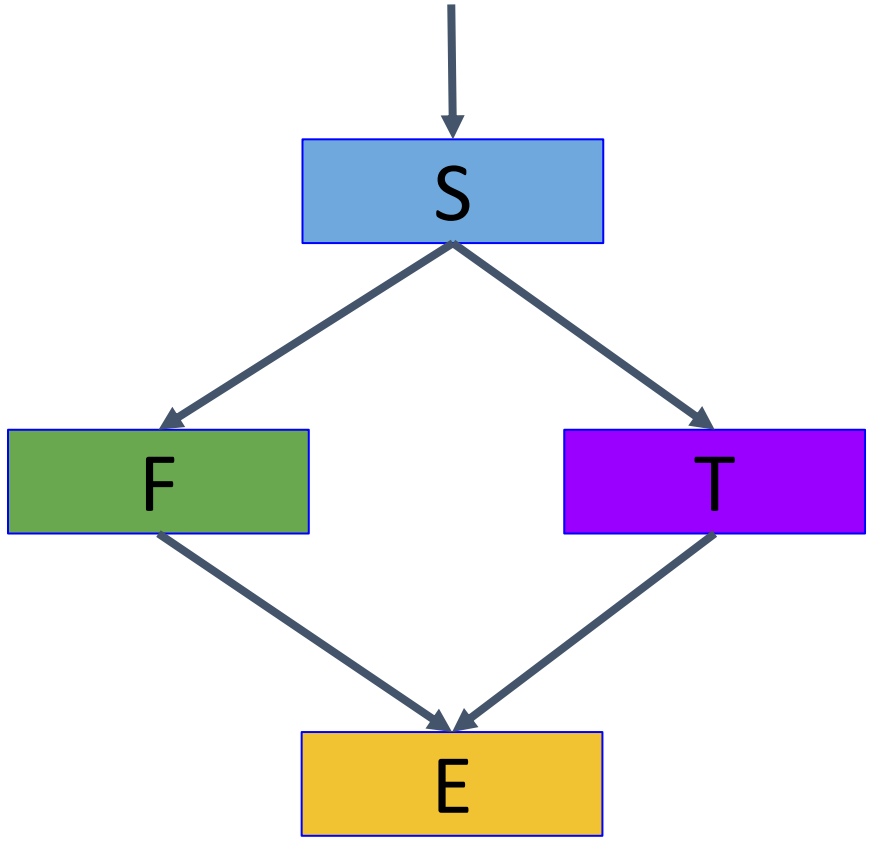


Meltdown (v3)

Microarchitecture

Speculative Execution + Out-of-order Execution

Speculative Execution



No Speculative Execution

Correct Prediction

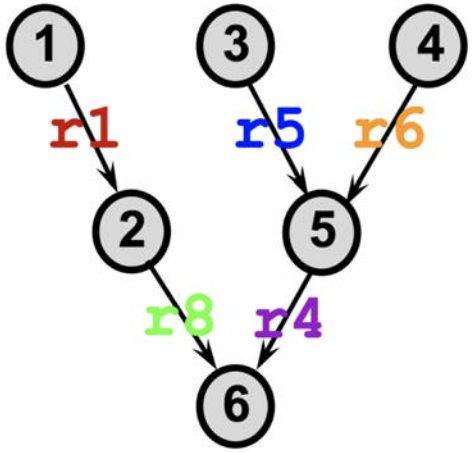
Misprediction

Out-of-order Execution

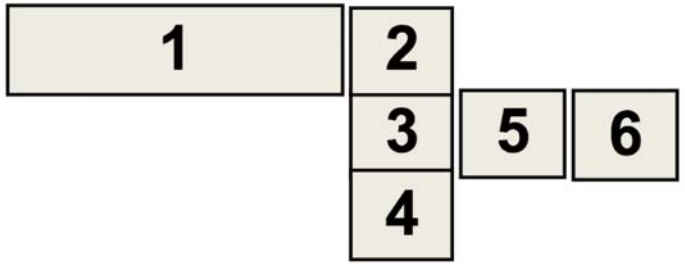
Example:

```
(1) r1 ← r4 / r7
(2) r8 ← r1 + r2
(3) r5 ← r5 + 1
(4) r6 ← r6 - r3
(5) r4 ← r5 + r6
(6) r7 ← r8 * r4
```

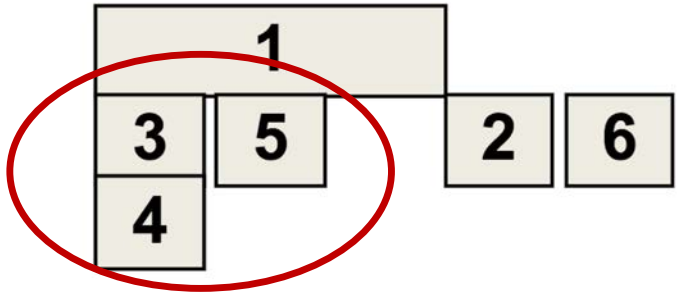
Data Flow Graph



In-order execution



Out-of-order execution



Images are from Dr. Lihu Rappoport

Speculative Execution + Out-of-order Execution

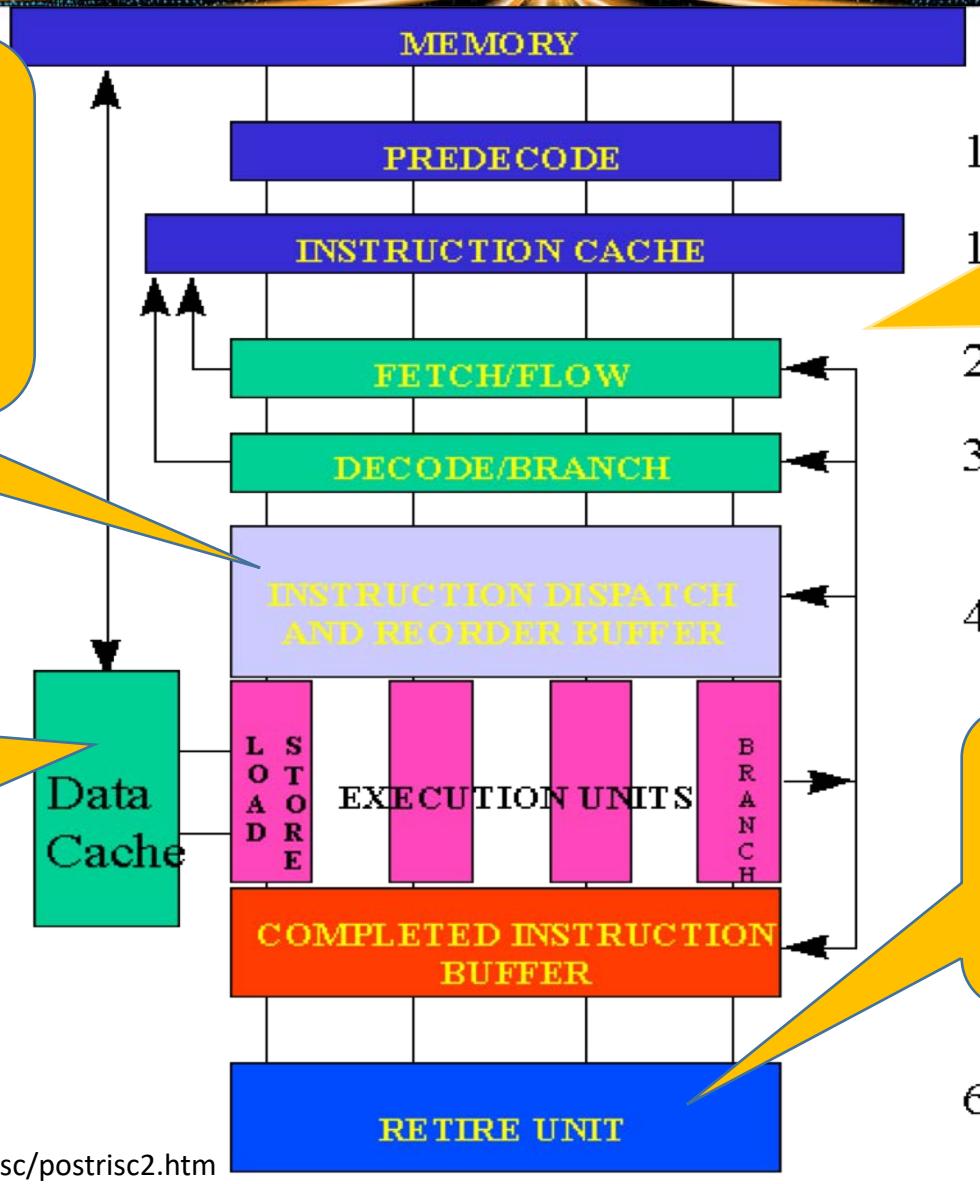
Enough?

Not Enough !!!

Delayed Permission Checking + Cache Side Effects

Execution Engine executes in a out-of-order way

Branch Predictor in Front End Serving Speculative Execution



Side effects in cache are still there!!!

Permission checking is delayed to Retire Unit

Image from https://www.cse.msu.edu/~enbody/postrisc/postrisc2.htm

How Meltdown (v3) Works

1. The content of an attacker-chosen memory location, which is inaccessible to the attacker, is loaded into a register.

```
asm __volatile__ (
    "%=:
    "xorq %%rax, %%rax                    \n"
    "movb (%[addr]), %%al                 \n"
    "shlq $0xc, %%rax                     \n"
    "jz %=b                               \n"
    "movq (%[dest], %%rax, 1), %%rbx      \n"
    :
    :  [addr] "r" (addr), [dest] "r" (dest)
    :  "%rax", "%rbx");
```

Point to the target kernel address
e.g., 0xffffffffc0e7e0a0

How Meltdown (v3) Works

2. A transient instruction accesses a cache line based on the secret content of the register.

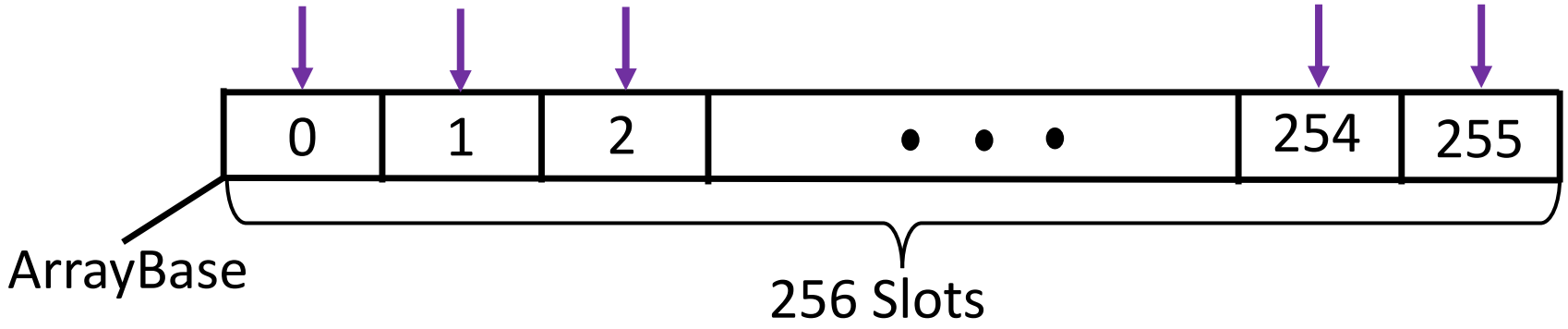
Bring data into cache

This number should >= 0x6

```
asm __volatile__ (
    "%=:                                    \n"
    "xorq %%rax, %%rax                      \n"
    "movb (%[addr]), %%al                   \n"
    "shlq $0xc, %%rax                       \n"
    "j%=b                                   \n"
    "movq (%[dest], %%rax, 1), %%rbx        \n"
    :
    : [addr] "r" (addr), [dest] "r" (dest)
    : "%rax", "%rbx");
```

How Meltdown (v3) Works

3. The attacker uses Flush+Reload to determine the accessed cache line and hence the secret stored at the chosen memory location.



The selected index is the value of the target byte

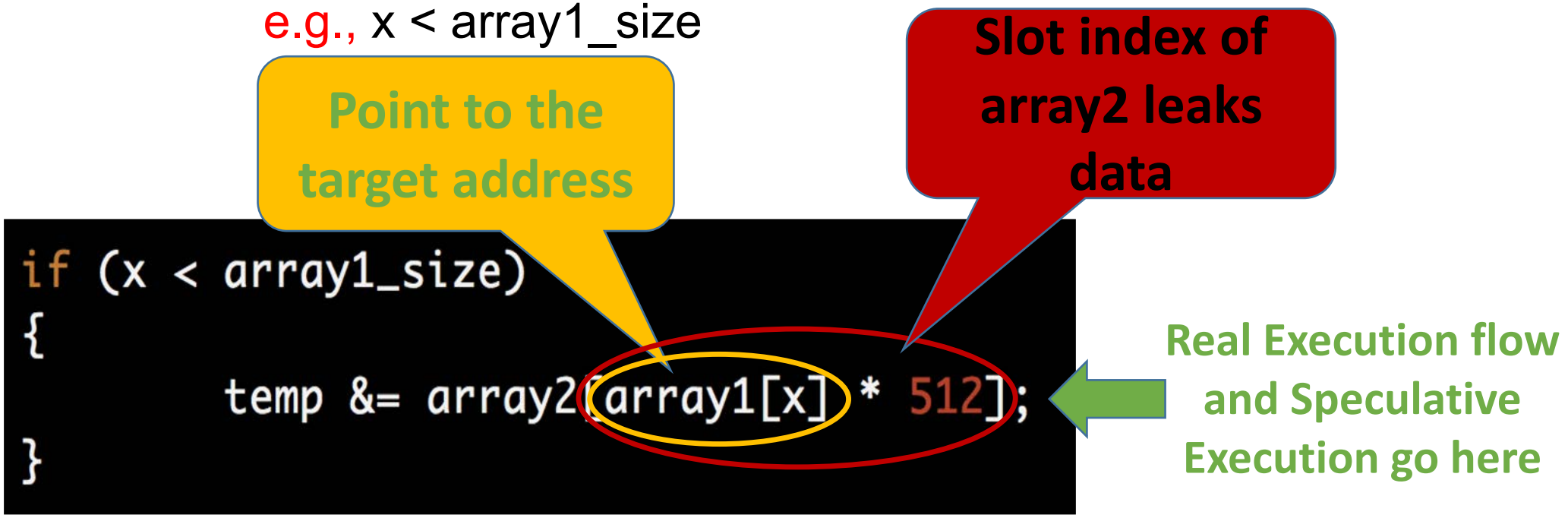
e.g., if the selected index is **0x65**, the value is **'A'**

How about Spectre (v1/v2)?

How Spectre Works

1. The setup phase, in which the processor is mistrained to make "an exploitable erroneous speculative prediction."

e.g., x < array1_size



How Spectre Works

2. The processor speculatively executes instructions from the target context into a microarchitectural covert channel.
   e.g., x > array1_size

Execution flow should go here

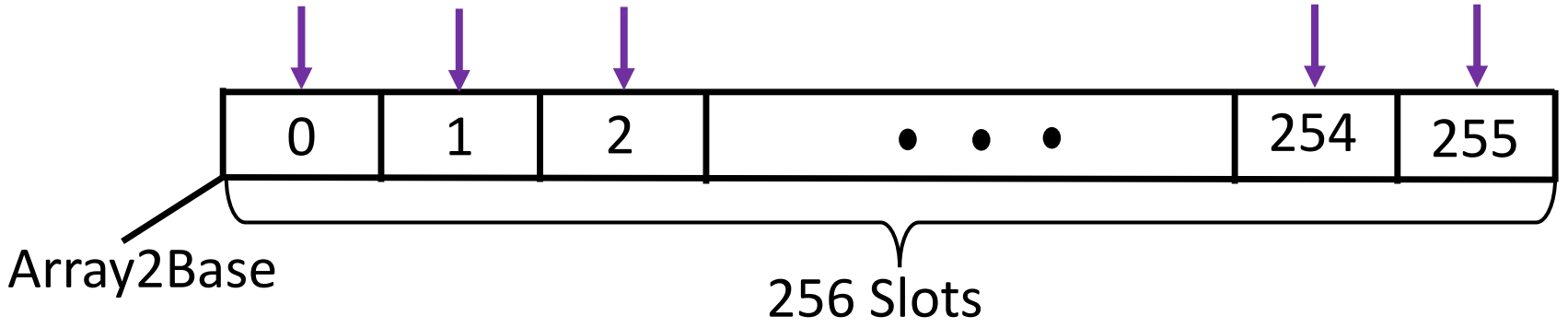
```
if (x < array1_size)
{
        temp &= array2[array1[x] * 512];
}
```

Speculative Execution goes here!

A slot of array2 is loaded into cache

How Spectre Works

3: The sensitive data is recovered. This can be done by timing access to memory addresses in the CPU cache.



The selected index is the value of the target byte

e.g., if the selected index is **0x66**, the value is **'B'**

How Spectre Read Kernel Data

- ✓ **array1** and **array2** are in user-space
- ✓ **x** is controlled by the adversary

array1+x points to
0xffffffffc0e7e0a0

Slot index of array2 leaks kernel data

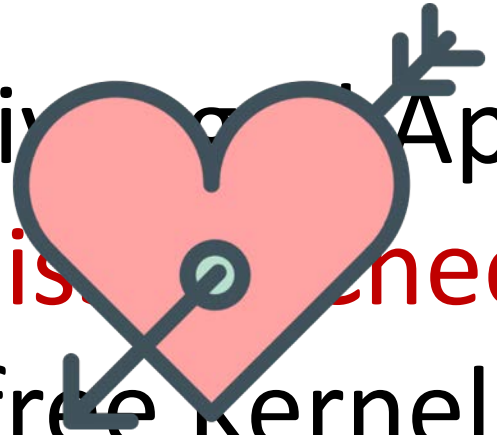
```
if (x < array1_size)
{
        temp &= array2[array1[x] * 512];
}
```

Get Unauthorized Data

1.Unpri App +

2.Permis hecking +

3.Bug-free Kernel



However...

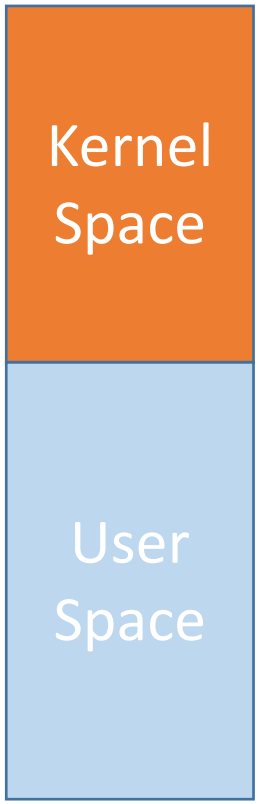
KPTI

~~Meltdown~~ FAILED

SMAP

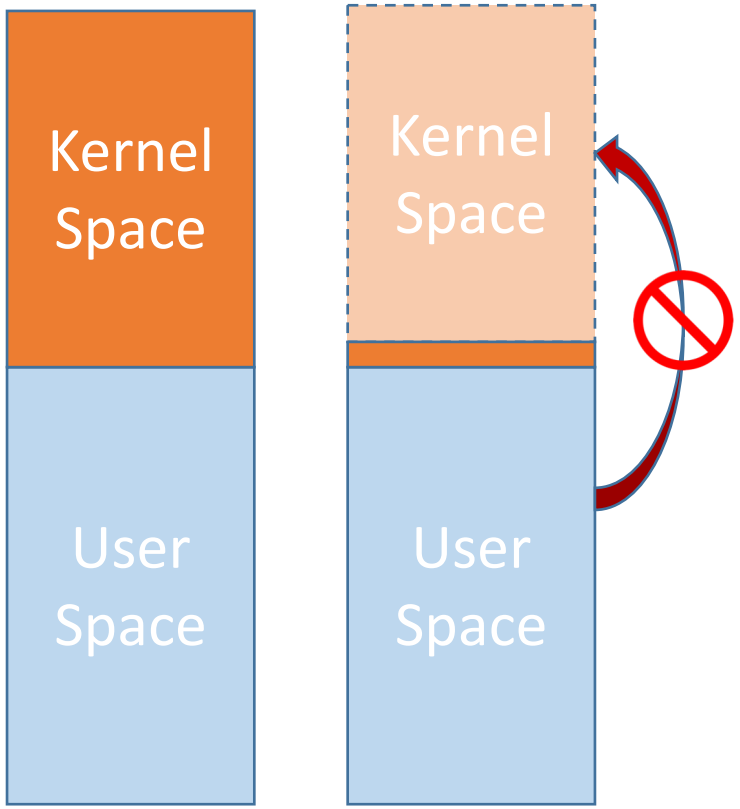
~~Spectre~~ FAILED

KPTI



User/kernel mode

Before KPTI



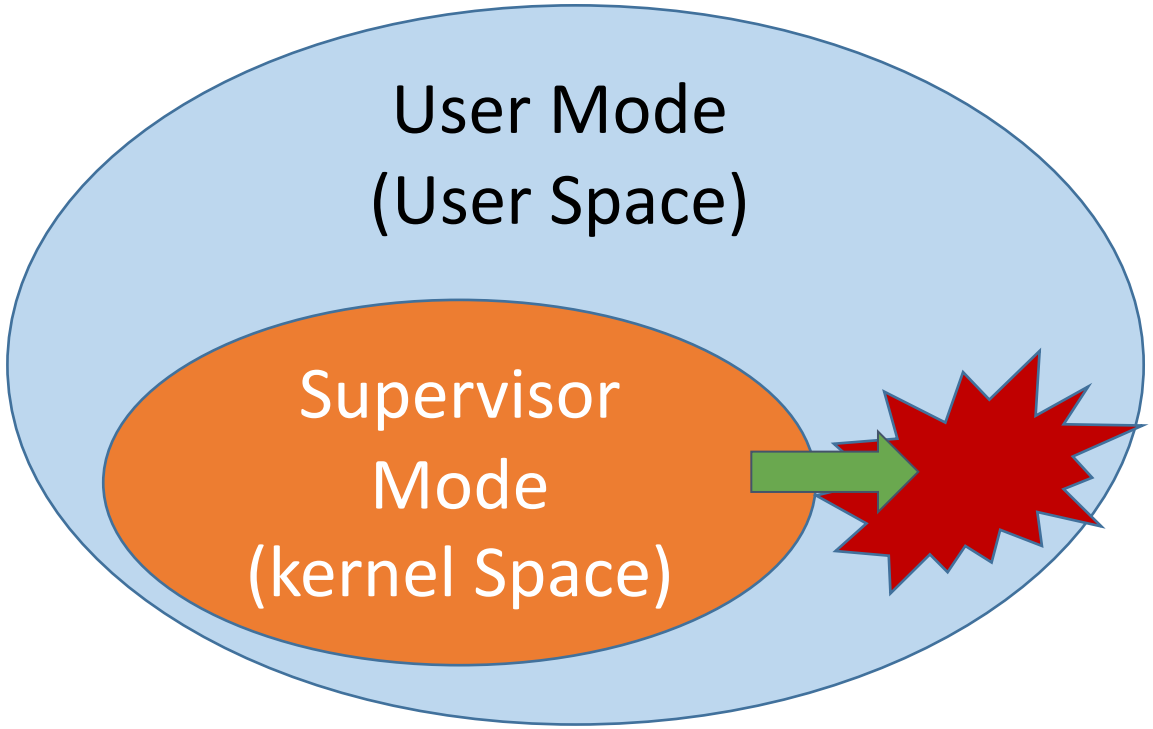
kernel mode

User mode

After KPTI

PCID helps performance

SMAP



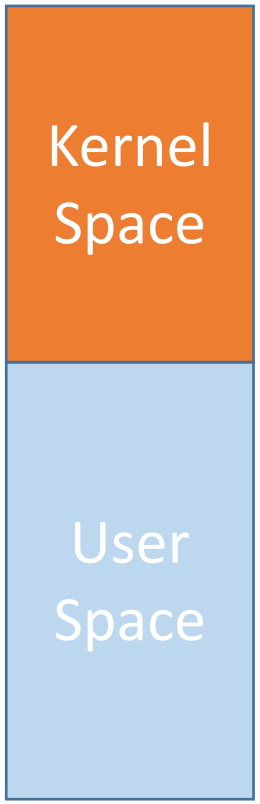
- ✓ SMAP is enabled when the SMAP bit in the CR4 is set
- ✓ SMAP can be temporarily disabled by setting the EFLAGS.AC flag
- ✓ **SMAP checking is done long before retirement or even execution**

Even we put the Spectre gadget into the kernel space, SMAP will stop it

Despair...

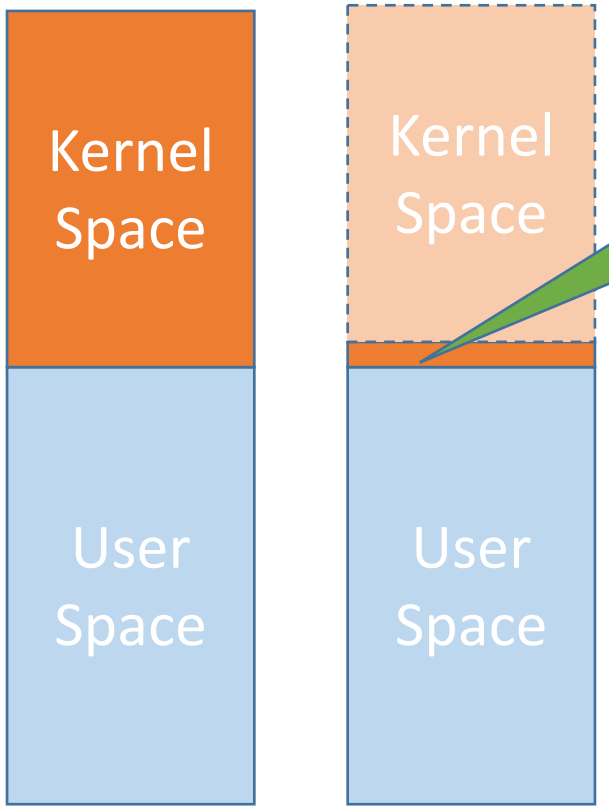
Image from http://nohopefor.us/credits

Hope in Despair



User/kernel mode

Before KPTI



kernel mode

User mode

After KPTI

This part cannot be eliminated

Shared range as a **bridge** to leak kernel data

New Variant Meltdown v3z

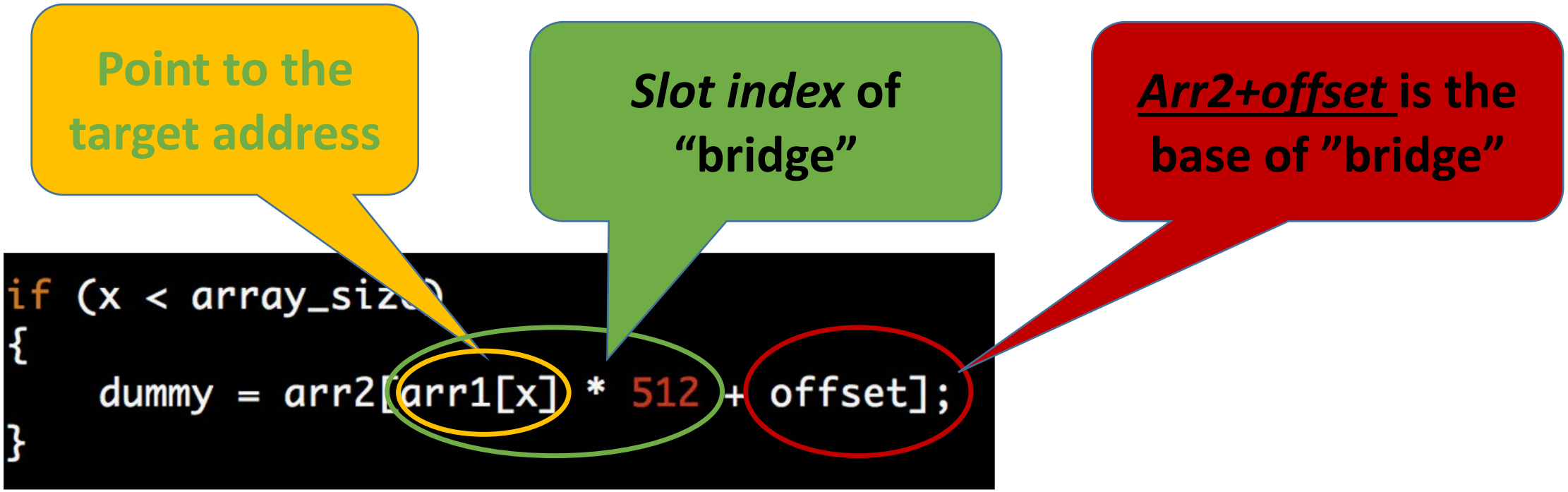
Breaking SMAP + KPTI + user-kernel Isolation

1: Use **new gadget** to build data-dependence between target kernel data and the bridge (bypass SMAP)

2: Use **Reliable Meltdown** to probe bridge to leak kernel data (bypass KPTI and user-kernel isolation)

1st Step: Trigger New Gadget

Similar to Spectre gadget, but not exact the same



x **and** ***offset*** **should be controlled by the adversary!!**

How to Trigger the New Gadget

There are many sources to trigger the new gadget

1: Syscalls
2: /proc and /sys etc. interfaces
3: Interrupt and exception handlers
4: eBPF
5: …

How to Find the New Gadget

Source Code Scanning

We use smatch for Linux Kernel 4.17.3,

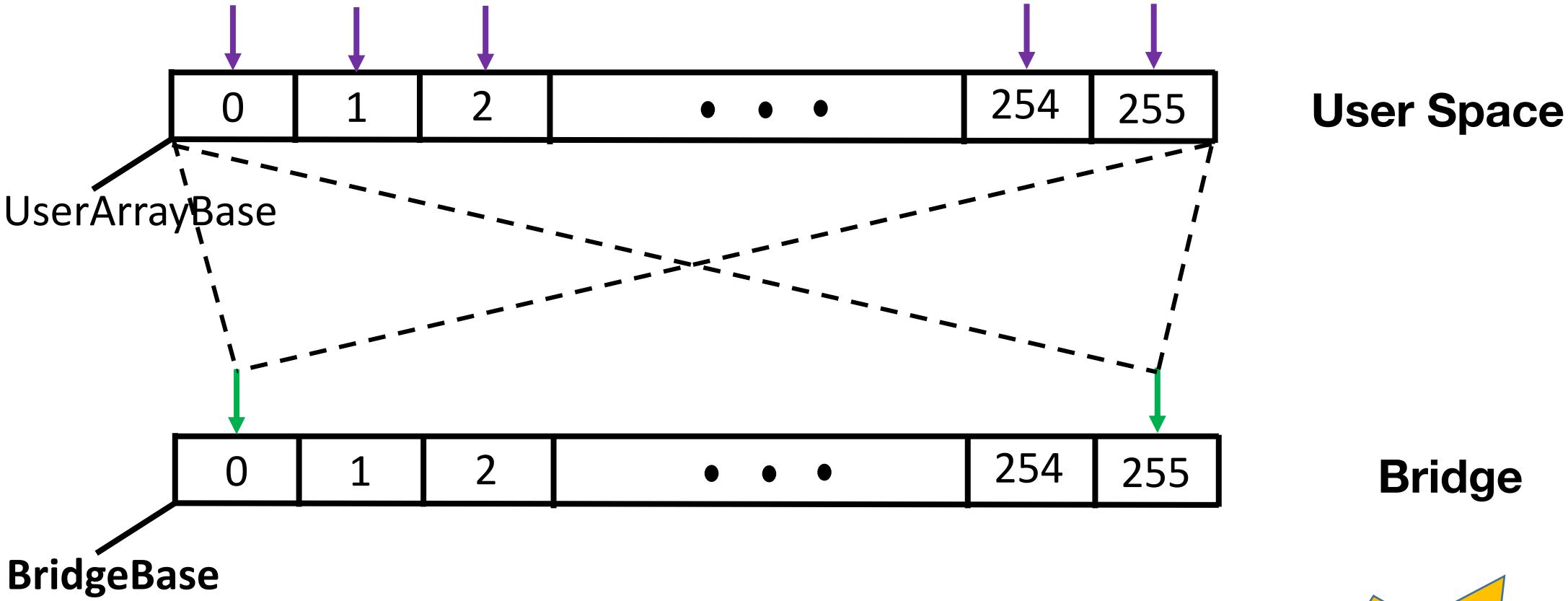
- ➢ Default config: 36 gadget candidates
- ➢ Allyes config: 166 gadget candidates

However, there are many restrictions to the gadget in real exploits

- ✓ Offset range
- ✓ Controllable invocation
- ✓ Cache noise
- ✓ …

Binary Code Scanning??

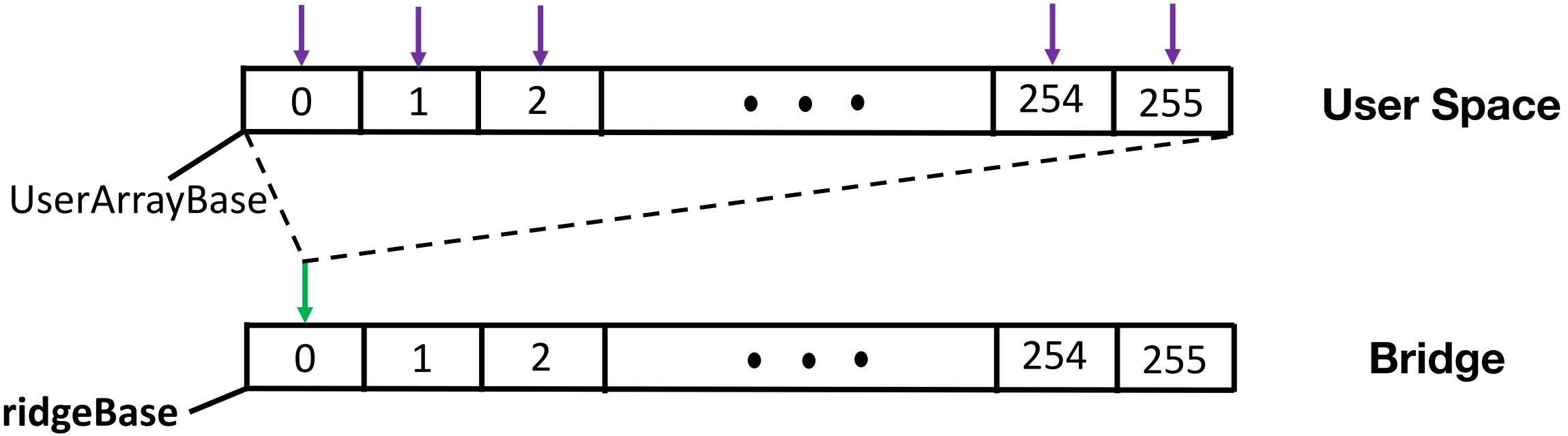
2nd Step: Probe Bridge



Obviously, in each round there are **(256*256) probes**
To make the result reliable, usually we need **multiple rounds**

Inefficient

Make it Practical/Efficient



Why do we need to probe 256 times in Meltdown?

If we know the value of the slot 0 of the BridgeBase, we probe it only **once**.

Can we know the values in advance?

No for Meltdown (v3)

Meltdown is able to read kernel data.
But, it requires that the target **data** is in the **CPU L1d cache.**

If the target data is **NOT** in L1d cache, **0x00** returns.

We need reliably reading kernel data!

Reliable Meltdown (V3r)

V3r has two steps:

1st step: bring data into L1d cache

```
if (x < size) {
        data = array[x];
}
```

Point to the target address

Everywhere in kernel

2nd step: use v3 getting data

We test it on Linux 4.4.0 with Intel CPU E3-1280 v6, and MacOS 10.12.6 (16G1036) with Intel CPU i7-4870HQ

Put Everything Together

Offline phase:

- Use v3r dumping bridge data, and save them into a table

Online phase:

- 1st step: Build data dependence between target data and bridge slot
- 2nd step: Probe each slot of the bridge

Efficiency:

- from **several minutes** (even around 1 hour in certain cases) to only **several seconds** to leak **one byte**.

Demo Settings

Kernel: Linux 4.4.0 with SMAP + KPTI

CPU: Intel CPU E3-1280 v6

In kernel space, we have a
secret msg, e.g., **xlabsecretxlabsecret**,
location is at, e.g., **0xffffffffc0e7e0a0**

Countermeasure Discussions

Software Mitigations

- ✓ Patch kernel to eliminate all expected gadgets
- ✓ Minimize the shared “bridge” region
- ✓ Randomize the shared “bridge” region
- ✓ Monitor cache-based side channel activities

Countermeasure Discussions

Hardware Mitigations

- ✓ Do permission checking during or even execution stage
- ✓ Revise speculative execution and out-of-order execution
- ✓ Use side channel resistant cache, e.g., exclusive/random cache
- ✓ Add hardware-level side channel detection mechanism

Take Away

- Trinational Spectre and Meltdown have been defeated by KPTI + SMAP + user-kernel Isolation combination.
- Our new Meltdown variants is able to break the strongest protection (KPTI + SMAP + user-kernel Isolation).
- All existing kernels need to be patched to mitigate our new attack

Oh No! KPTI Defeated Unauthorized Data Leakage is Still Possible

Baidu X-Lab Medium: https://medium.com/baiduxlab

Q&A image is from https://i.redd.it/wbiwgnokgig11.jpg