



**blackhat**<sup>®</sup>  
EUROPE 2024  
DECEMBER 11-12, 2024  
BRIEFINGS

# SysBumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon

Speaker : Hyerean Jang

# \$Whoami



## Hyerean Jang

Ph. D Student @Korea University

✉ Email : hr\_jang@korea.ac.kr

🔍 Research interests :  
Microarchitectural side-channel  
vulnerability, System security

# Contributors



**Taehun Kim**

Ph. D Student  
@Korea University

✉ Email : taehunk@korea.ac.kr



**Youngjoo Shin**

Professor  
@Korea University

✉ Email : syoungjoo@korea.ac.kr

# Apple Silicon

Apple's proprietary arm-based processor



The screenshot shows a web browser window for support.apple.com. The title bar includes standard OS X/Mac controls (red, yellow, green buttons, minimize/maximize, close) and the URL 'support.apple.com'. Below the title bar is a navigation menu with links for Store, Mac, iPad, iPhone, Watch, Vision, AirPods, TV & Home, Entertainment, Accessories, Support, a search icon, and a shopping bag icon. The main content area has a dark background and features the heading 'Mac computers with Apple silicon' in large, bold, white font. A paragraph below the heading states: 'Starting with certain models introduced in late 2020, Apple began the transition from Intel processors to Apple silicon in Mac computers.' A bulleted list follows, identifying specific models:

- [MacBook Pro](#) introduced in 2021 or later, plus MacBook Pro (13-inch, M1, 2020)
- [MacBook Air](#) introduced in 2022 or later, plus MacBook Air (M1, 2020)
- [iMac](#) introduced in 2021 or later

# Apple Silicon

Apple's proprietary arm-based processor



A screenshot of a web browser window showing the support.apple.com website. The page title is "Mac computers with Apple silicon". The text below states: "Starting with certain models introduced in late 2020, Apple began the transition from Intel processors to Apple silicon in Mac computers." It then lists three models: "MacBook Pro" (introduced in 2021 or later, plus MacBook Pro (13-inch, M1, 2020)), "MacBook Air" (introduced in 2022 or later, plus MacBook Air (M1, 2020)), and "iMac" (introduced in 2021 or later).

support.apple.com

Store Mac iPad iPhone Watch Vision AirPods TV & Home Entertainment Accessories Support

## Mac computers with Apple silicon

Starting with certain models introduced in late 2020, Apple began the transition from Intel processors to Apple silicon in Mac computers.

[MacBook Pro](#) introduced in 2021 or later, plus MacBook Pro (13-inch, M1, 2020)

[MacBook Air](#) introduced in 2022 or later, plus MacBook Air (M1, 2020)

[iMac](#) introduced in 2021 or later



# Exploring Microarchitectural Side-Channel Vulnerabilities on macOS for Apple Silicon

# What is SysBumps Attack?

KASLR breaking attack on macOS for Apple silicon

SYSBUMPS

# What is SysBumps Attack?

KASLR breaking attack on macOS for Apple silicon

In system call



Speculative  
Execution

SYSBUMPS

# What is SysBumps Attack?

KASLR breaking attack on macOS for Apple silicon

In system call



**SYSBUMPS**

Speculative  
Execution

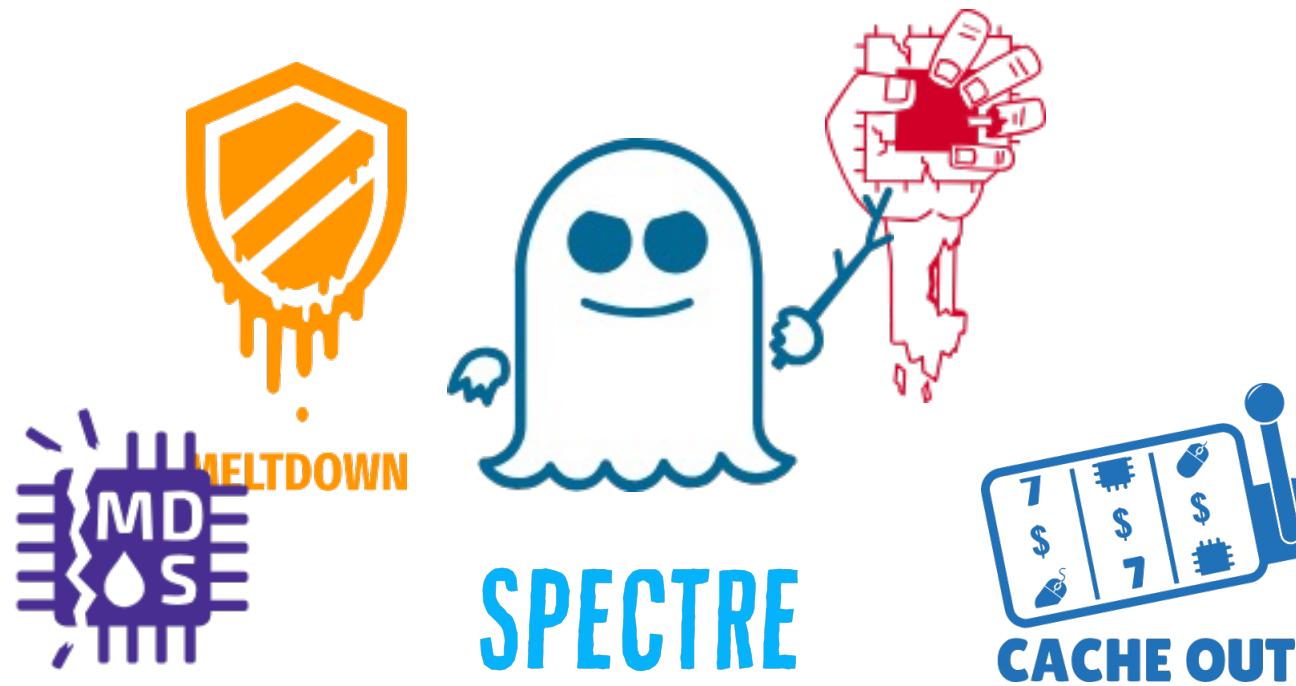
TLB-based  
Side-channel

# Outline

- Background
- Existing Microarchitectural Attack on KASLR
- Challenges
- Our Approaches
- SysBumps Attack
- Mitigations
- Takeaway

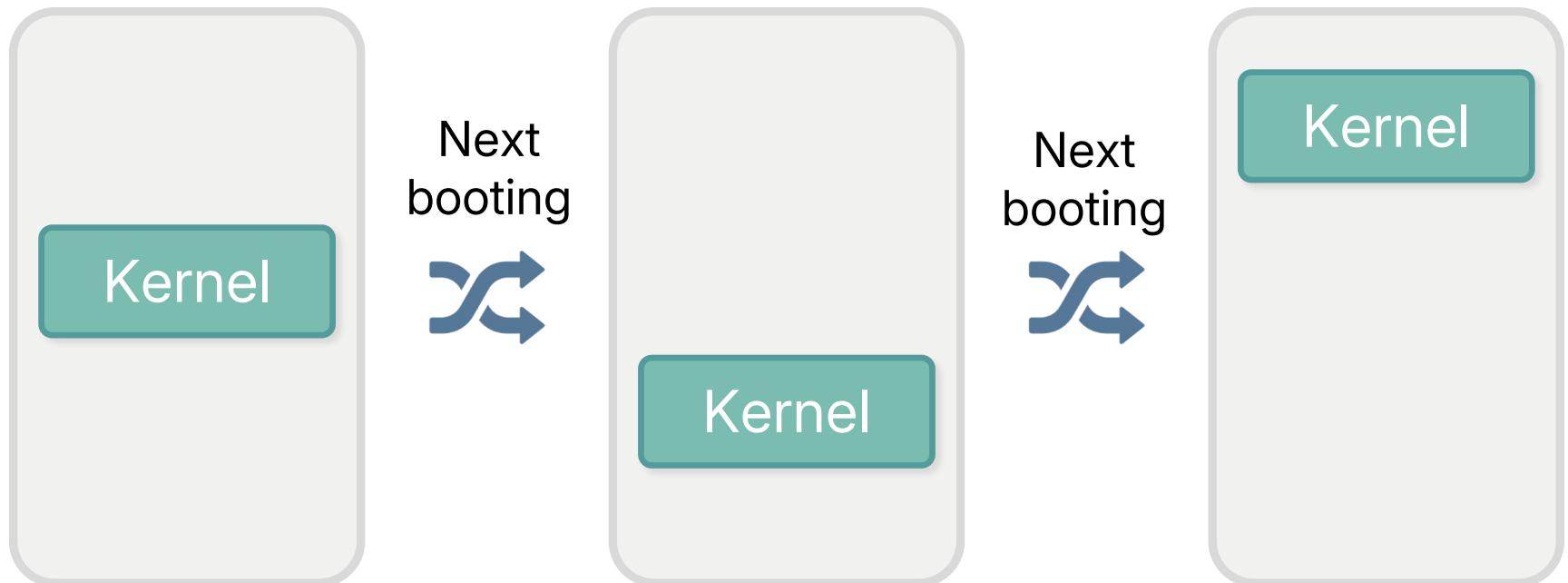
# Microarchitectural Side-Channel Attack

- Exploit CPU design flaws to extract information through indirect leakages
  - Cache, TLB, branch predictors, ...



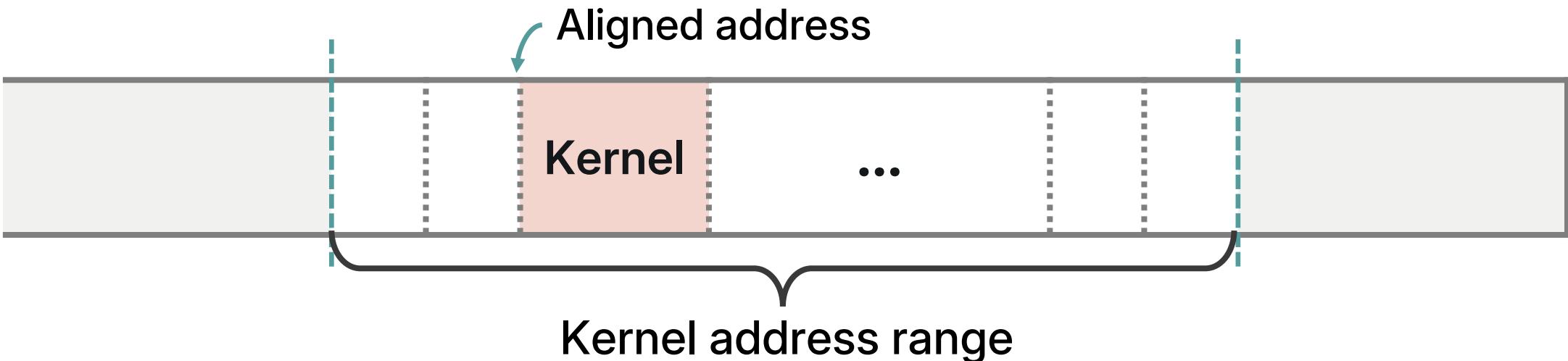
# Kernel Address Space Layout Randomization

- Load kernel into random location
  - Prevent attackers from predicting target kernel addresses for exploits



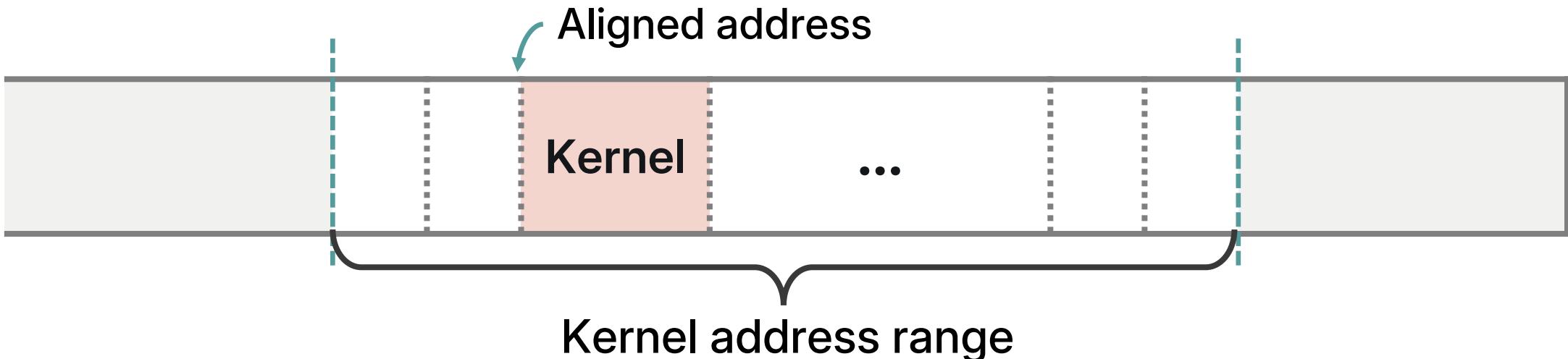
# Kernel Address Space Layout Randomization

- Kernel is loaded within a reserved range of kernel addresses
  - Aligned address to a specific size



# Kernel Address Space Layout Randomization

- Kernel is loaded within a reserved range of kernel addresses
  - Aligned address to a specific size
    - Linux : 0xFFFF FFFF 8000 0000 ~ 0xFFFF FFFF C000 0000 (16MB aligned)
    - Windows : 0xFFFF F800 0000 0000 ~ 0xFFFF F804 0000 0000 (2MB aligned)



# **Translation Lookaside Buffer (TLB)**

- Store translated virtual-to-physical address mappings

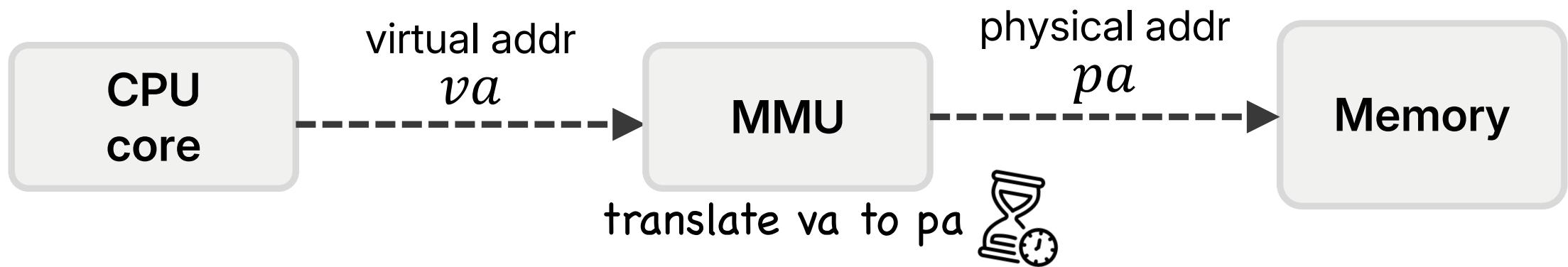
# Translation Lookaside Buffer (TLB)

- Store translated virtual-to-physical address mappings



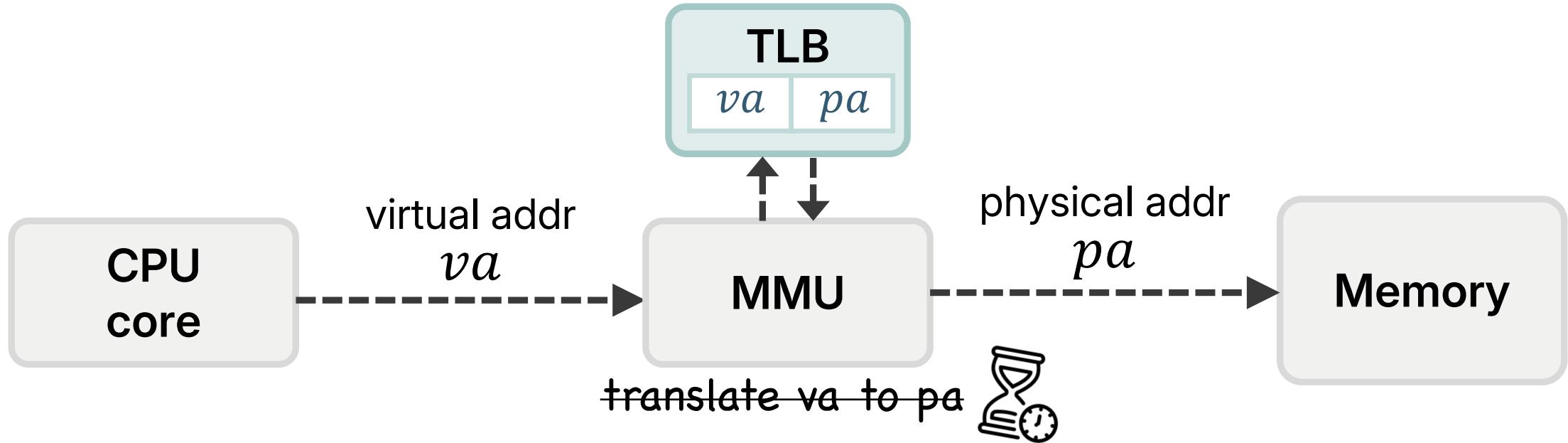
# Translation Lookaside Buffer (TLB)

- Store translated virtual-to-physical address mappings



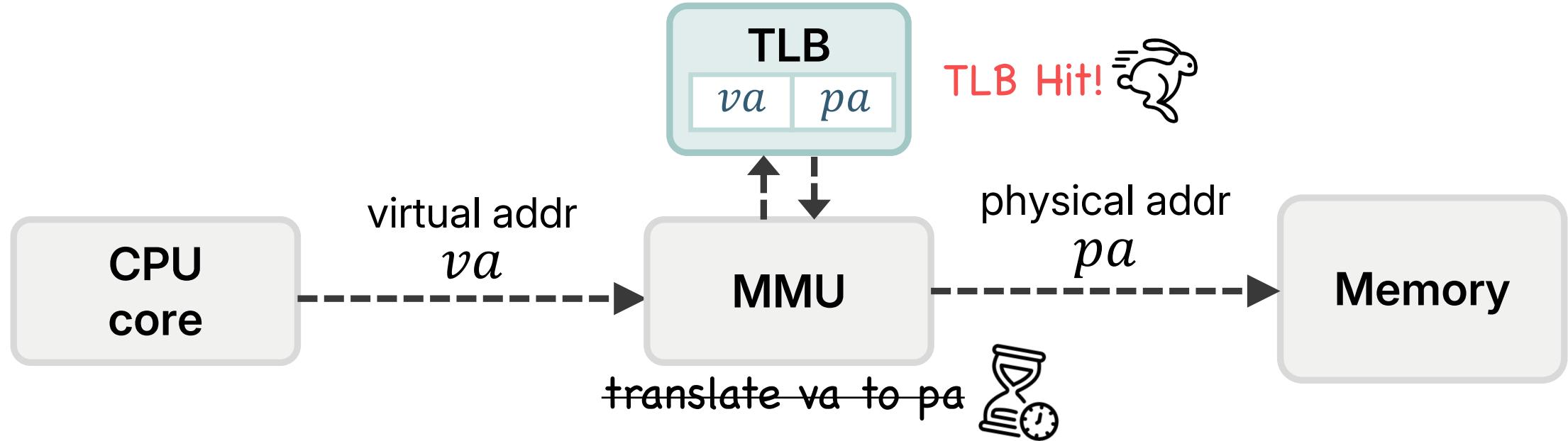
# Translation Lookaside Buffer (TLB)

- Store translated virtual-to-physical address mappings



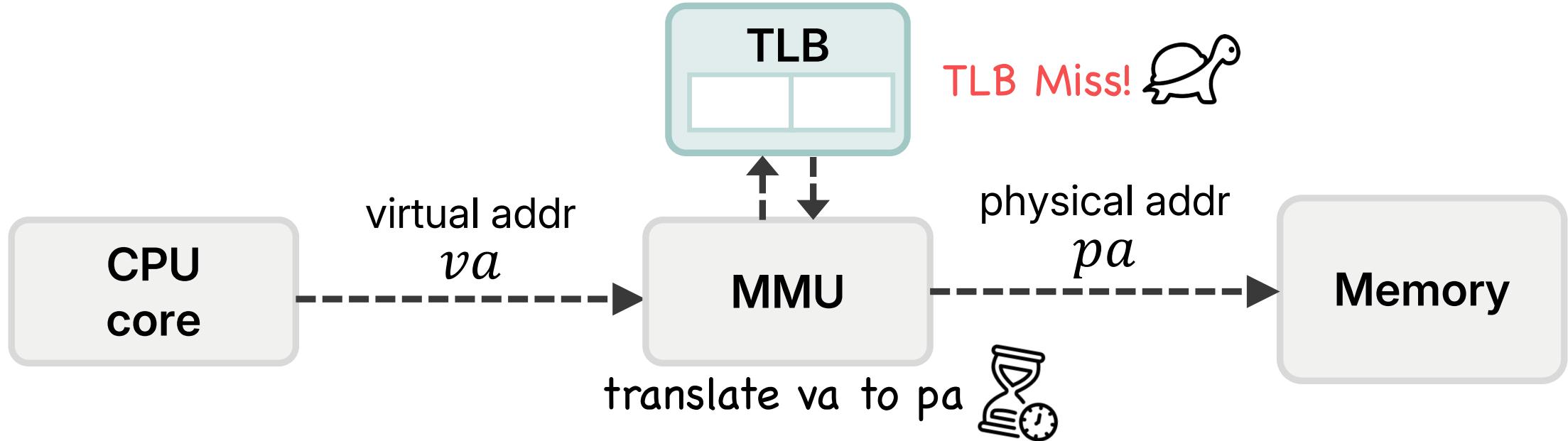
# Translation Lookaside Buffer (TLB)

- Store translated virtual-to-physical address mappings



# Translation Lookaside Buffer (TLB)

- Store translated virtual-to-physical address mappings



# Translation Lookaside Buffer (TLB)

- Store translated virtual-to-physical address mappings

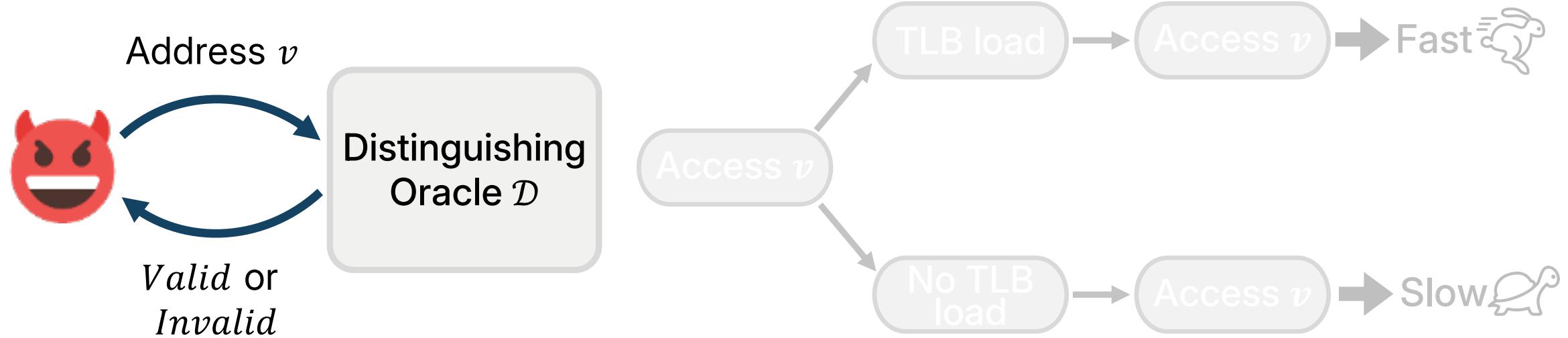
1. TLB stores translations for valid virtual addresses only

# Translation Lookaside Buffer (TLB)

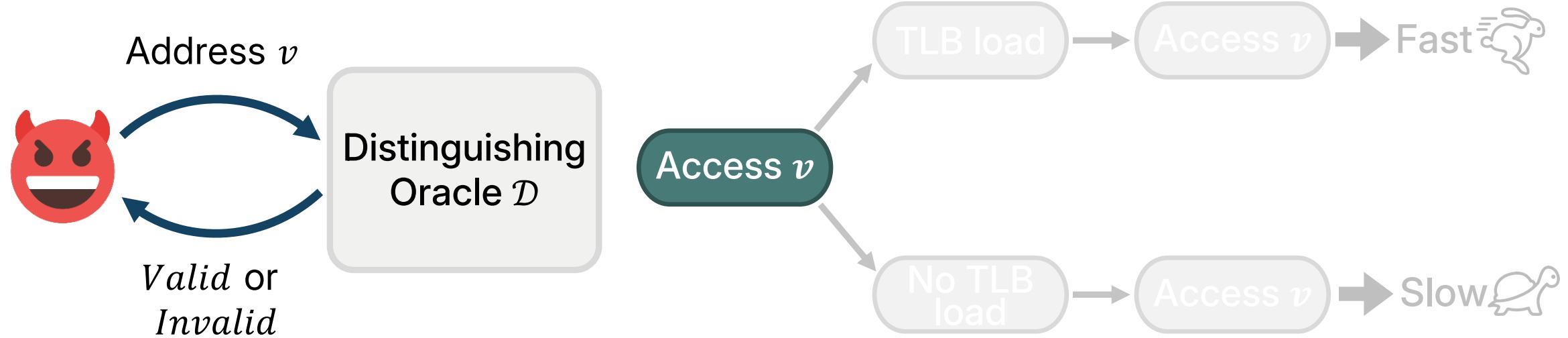
- Store translated virtual-to-physical address mappings

1. TLB stores translations for valid virtual addresses only
2. Access timing differences can expose sensitive information

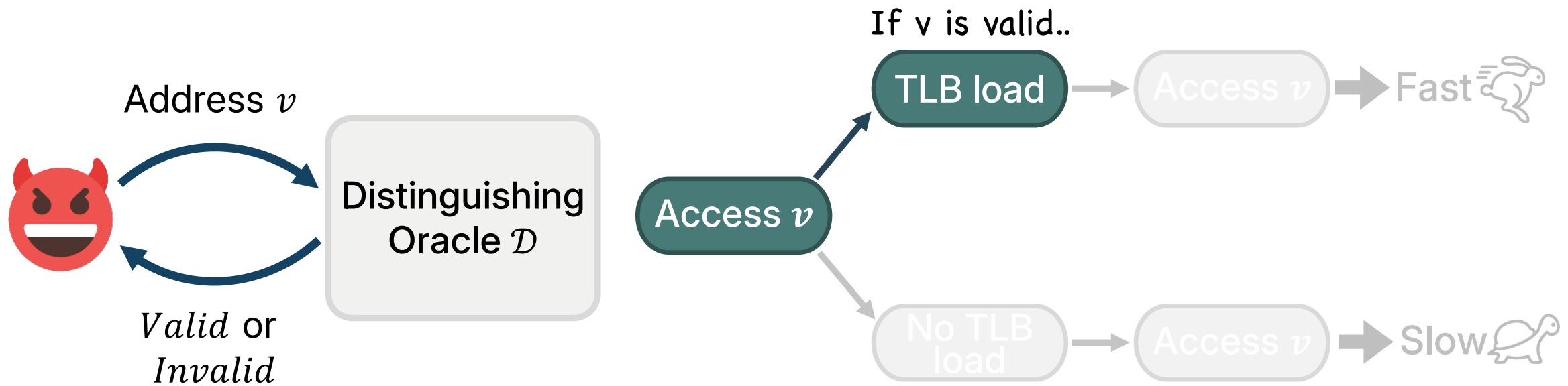
# Existing Microarchitectural Attack on KASLR



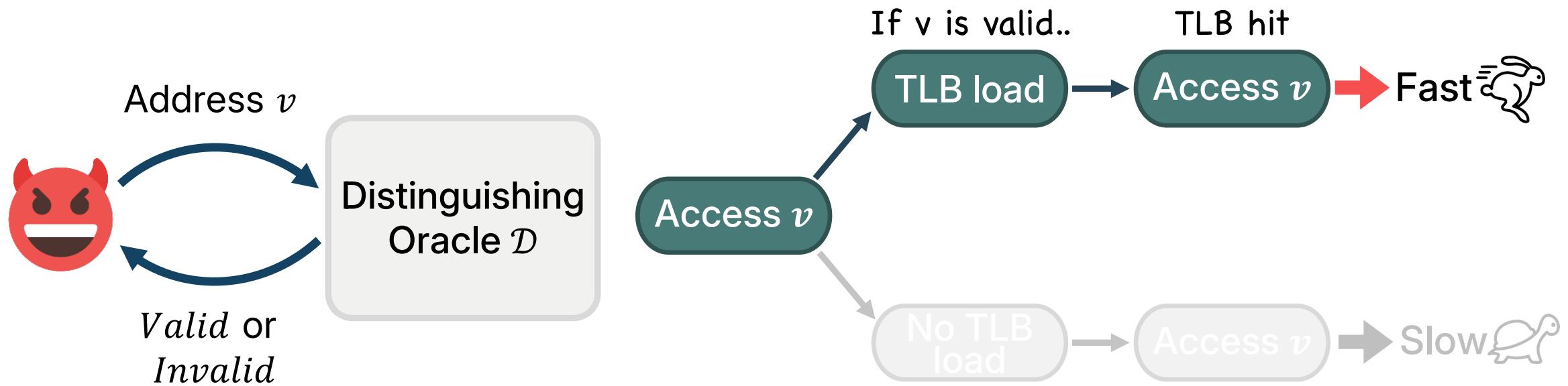
# Existing Microarchitectural Attack on KASLR



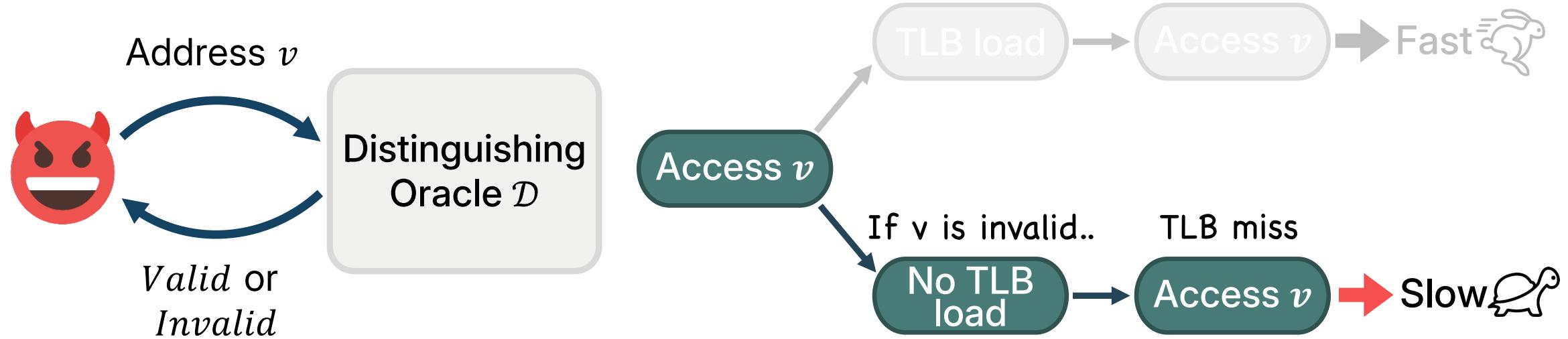
# Existing Microarchitectural Attack on KASLR



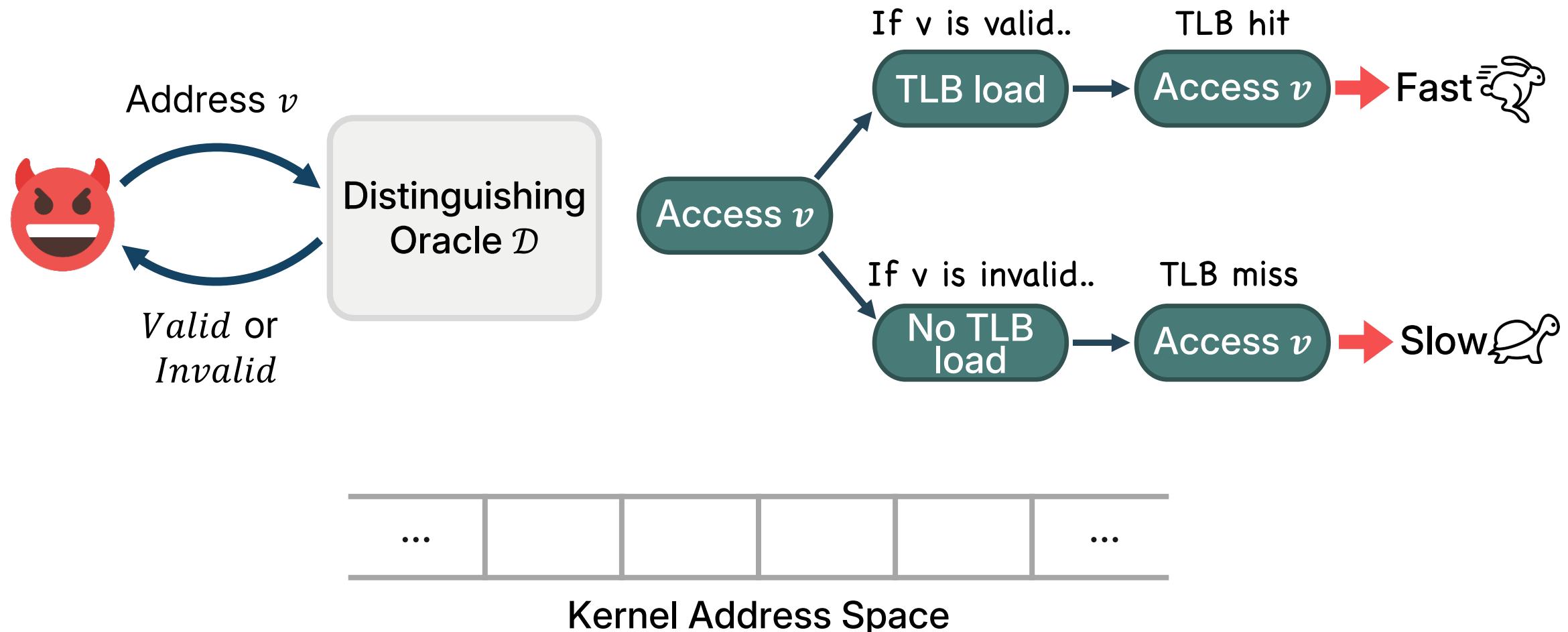
# Existing Microarchitectural Attack on KASLR



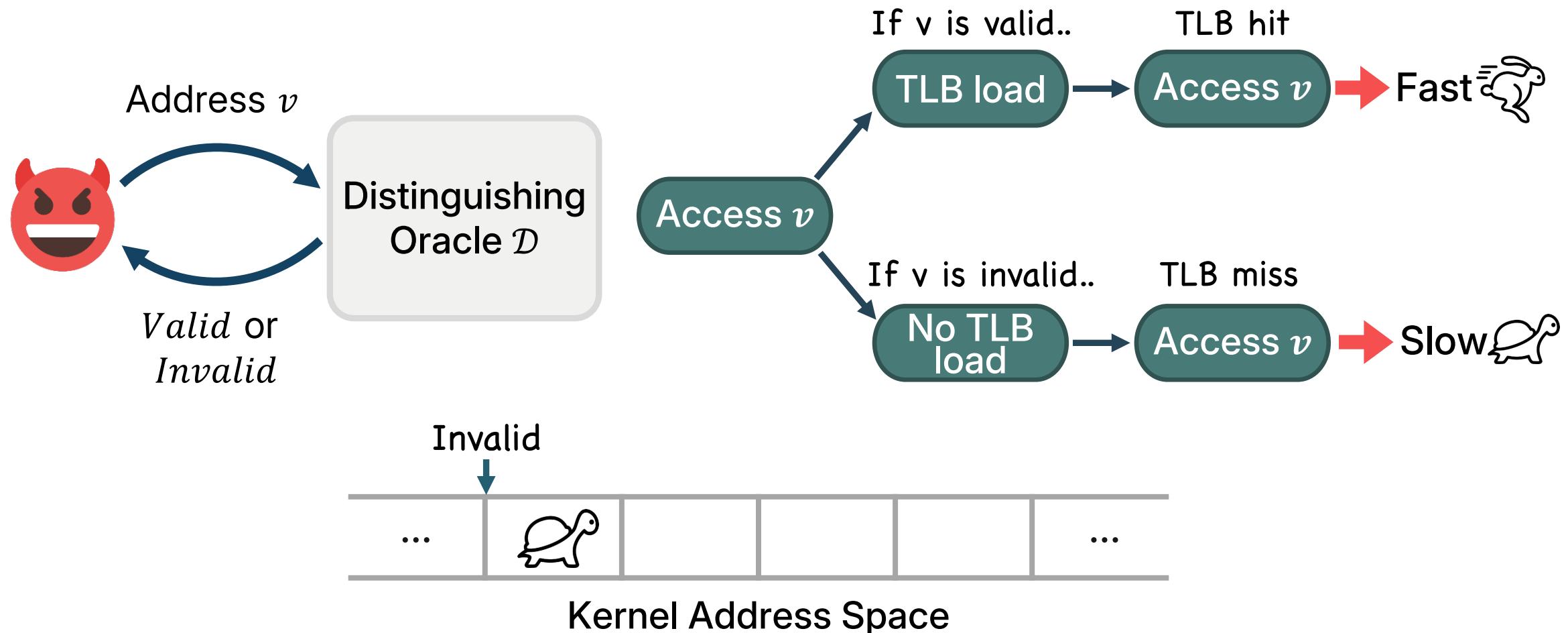
# Existing Microarchitectural Attack on KASLR



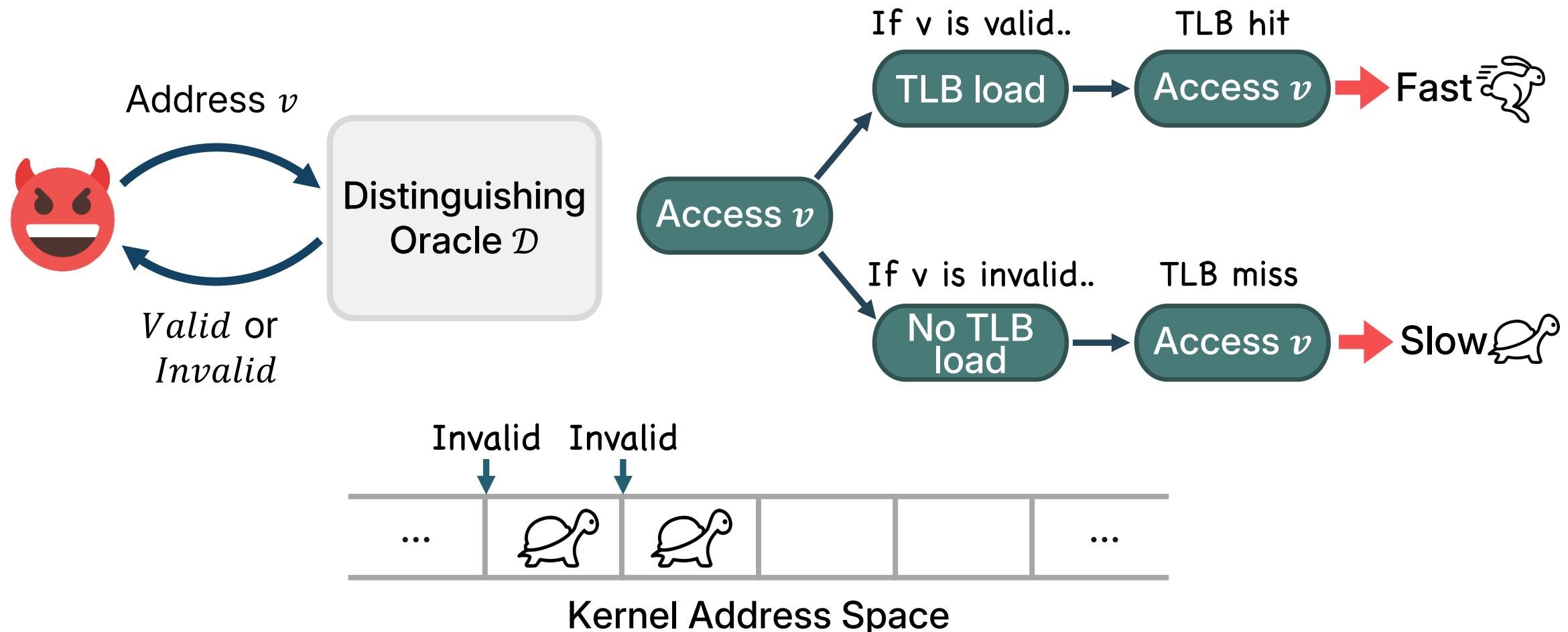
# Existing Microarchitectural Attack on KASLR



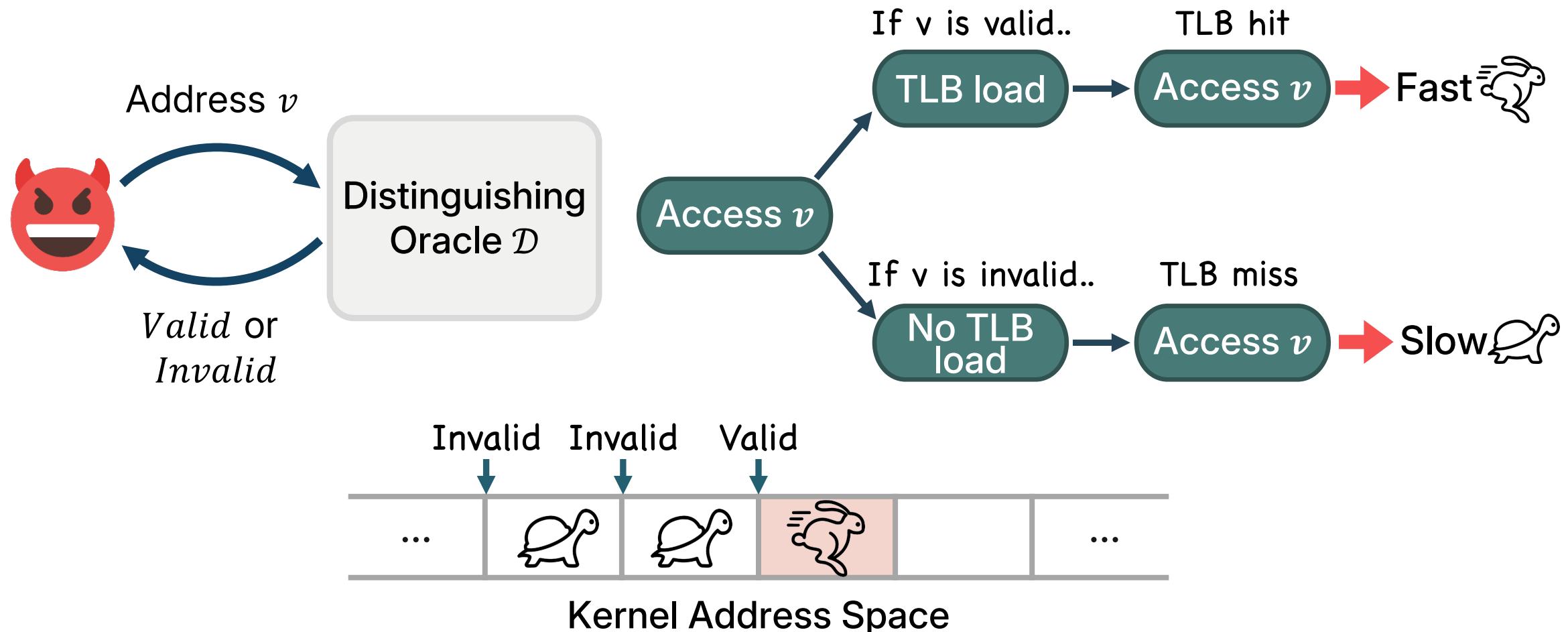
# Existing Microarchitectural Attack on KASLR



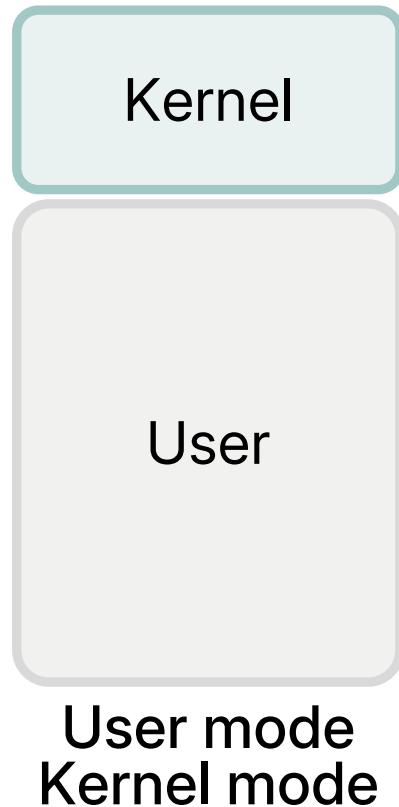
# Existing Microarchitectural Attack on KASLR



# Existing Microarchitectural Attack on KASLR

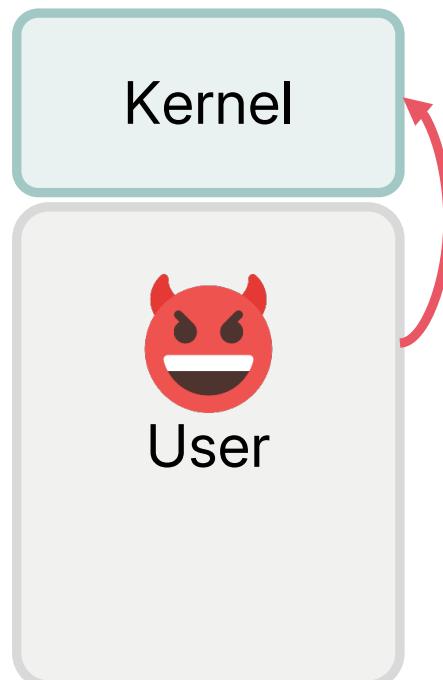


# How Can Unprivileged Attacker Access Kernel Addresses?



# How Can Unprivileged Attacker Access Kernel Addresses?

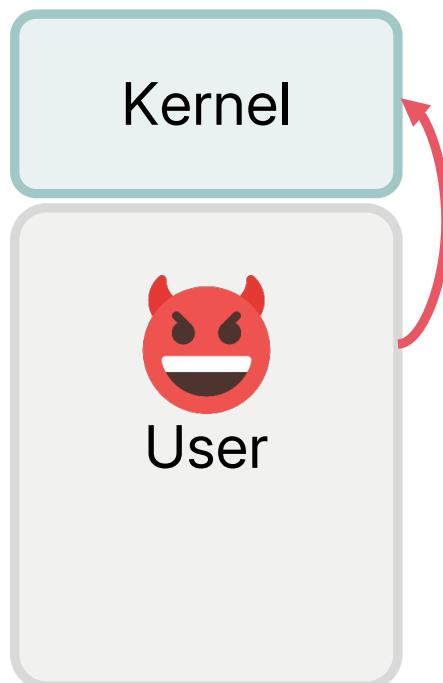
Fault suppressing with  
prefetch, TSX, AVX, ...



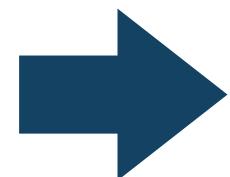
User mode  
Kernel mode

# How Can Unprivileged Attacker Access Kernel Addresses?

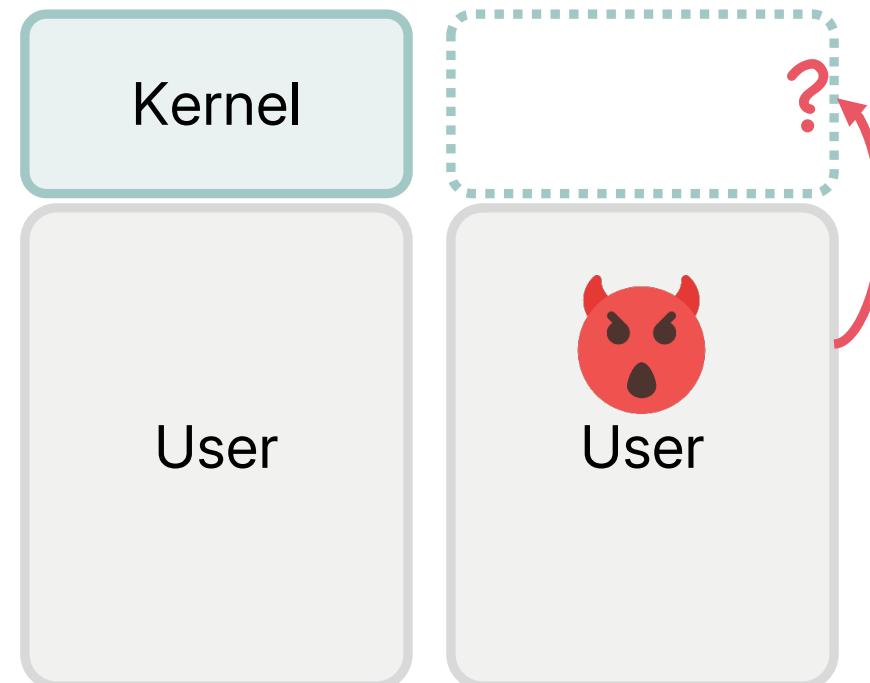
Fault suppressing with  
prefetch, TSX, AVX, ...



To mitigate



Kernel Page-Table Isolation (KPTI)

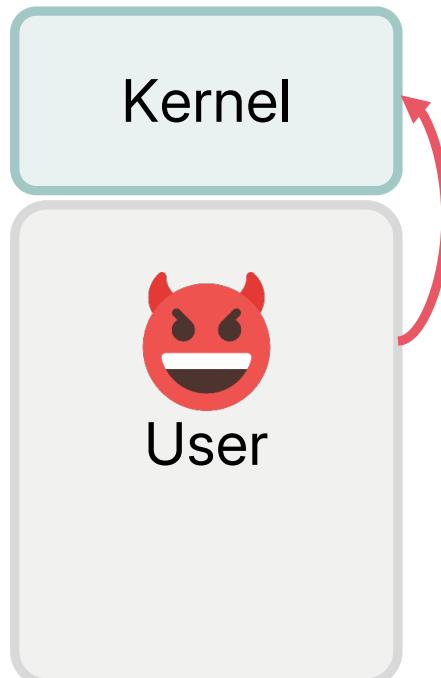


User mode  
Kernel mode

Kernel mode      User mode

# How Can Unprivileged Attacker Access Kernel Addresses?

Fault suppressing with  
prefetch, TSX, AVX, ...



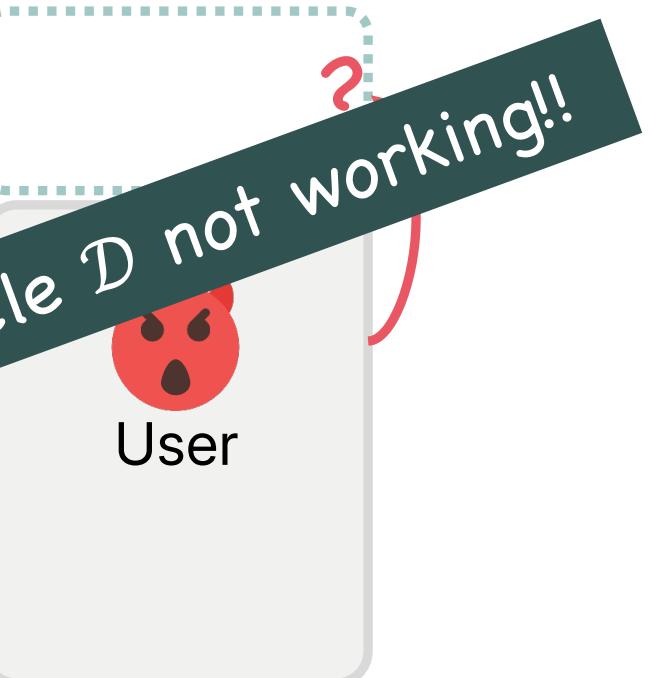
User mode  
Kernel mode

To mitigate

Kernel Page-Table Isolation (KPTI)



Kernel mode



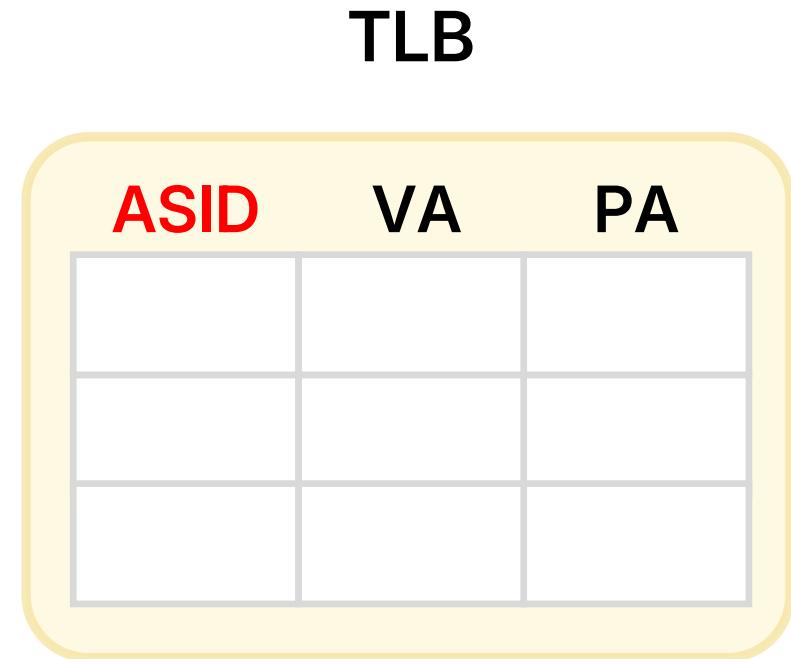
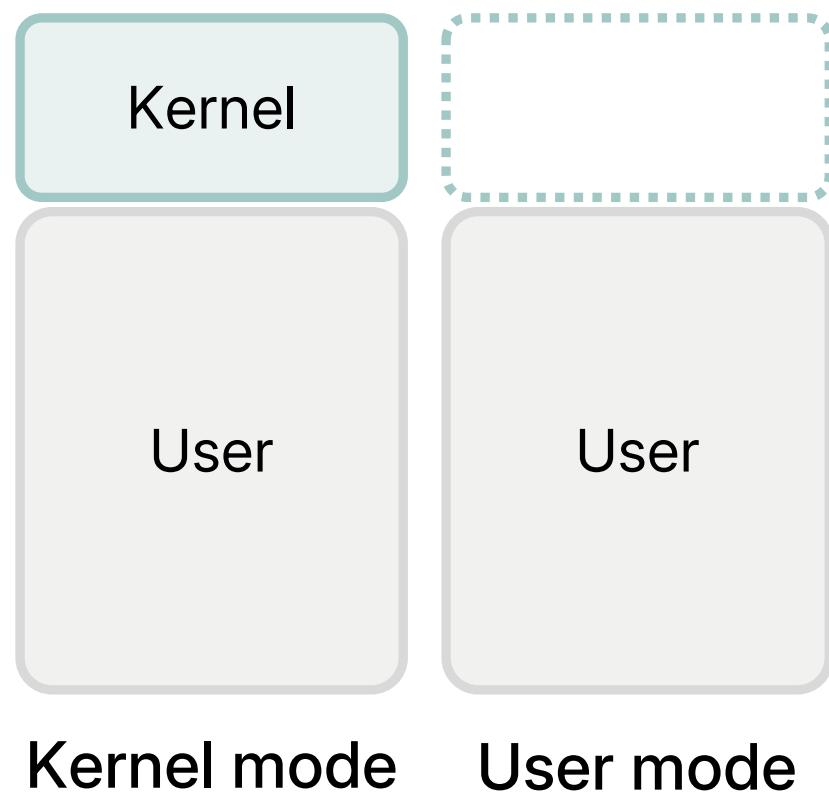
User mode

# **Challenges to Breaking KASLR**

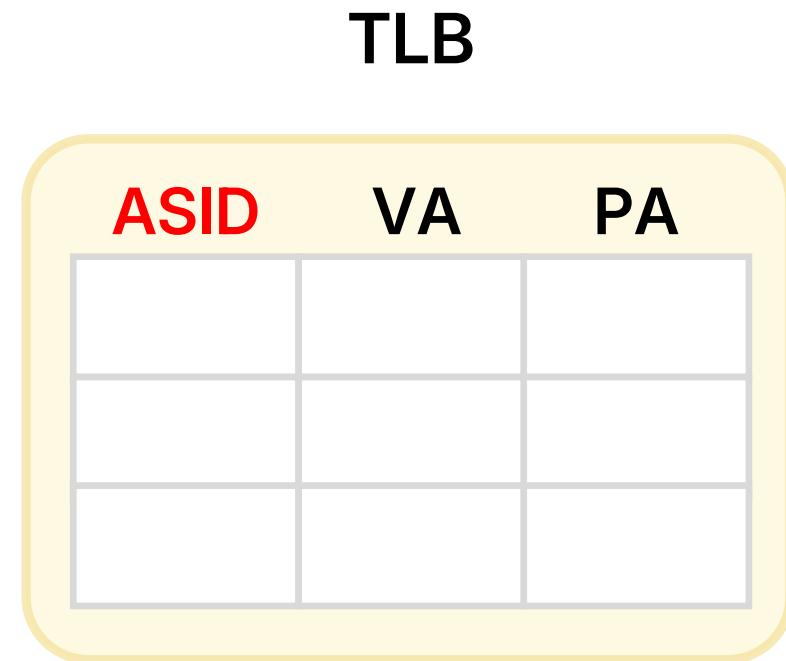
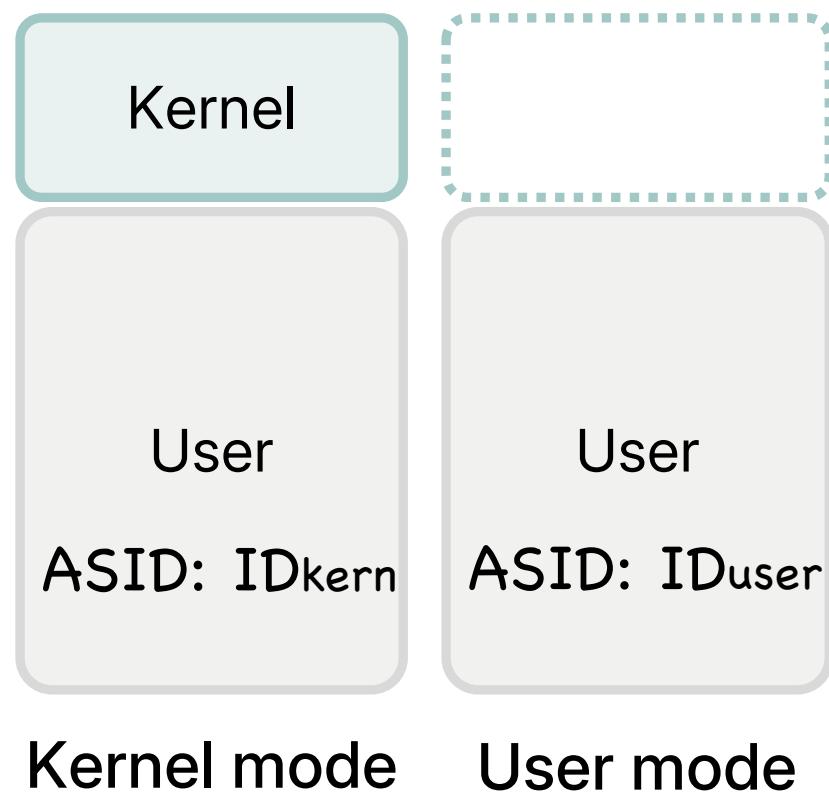
# Challenges to Breaking KASLR

- C1. How to cache a kernel address in the TLB?
  - Kernel page table isolation (KPTI)

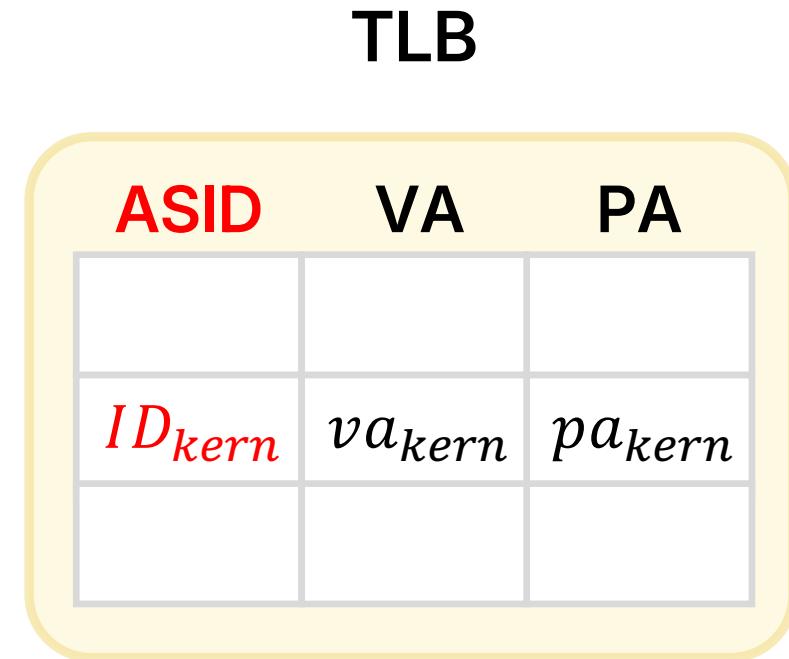
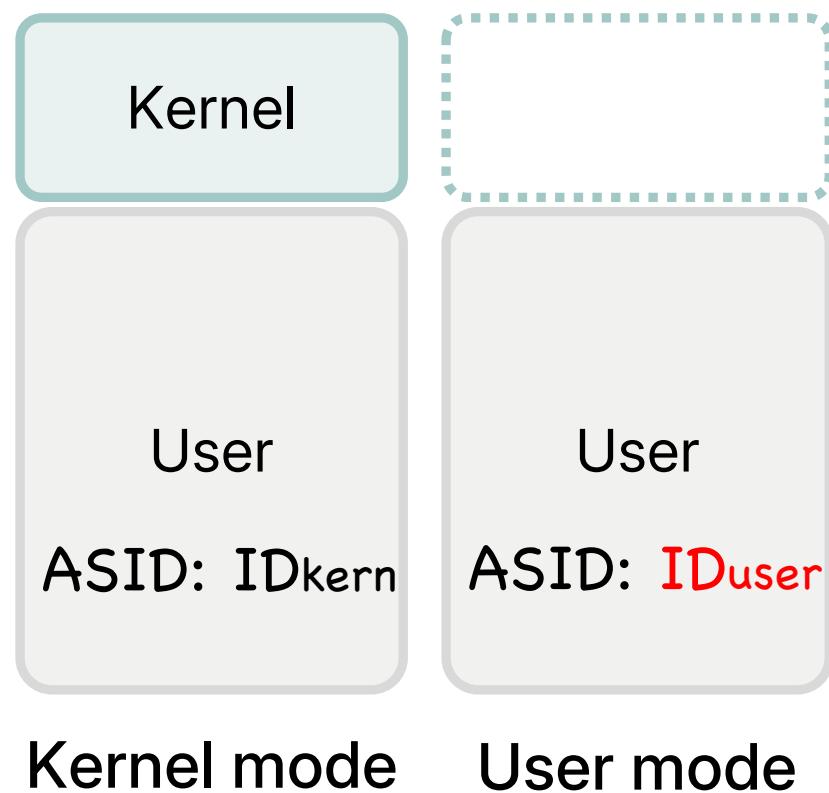
# Address Space Identifier



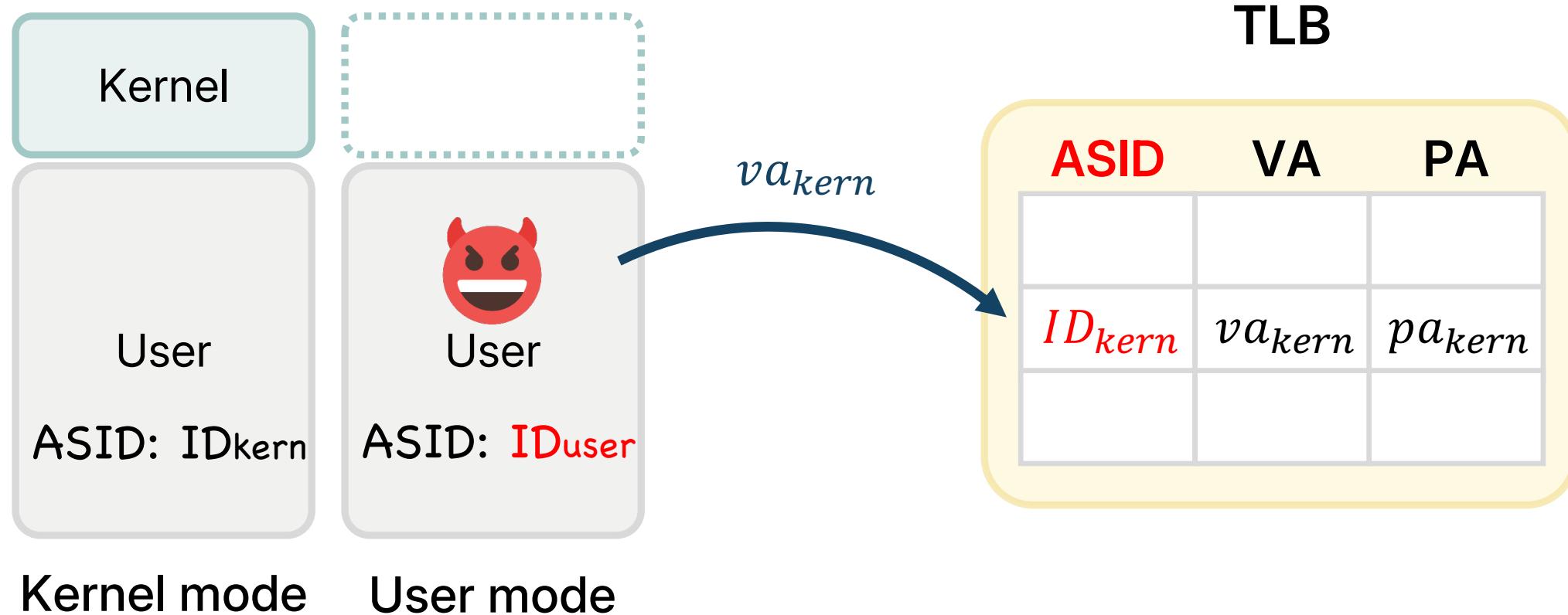
# Address Space Identifier



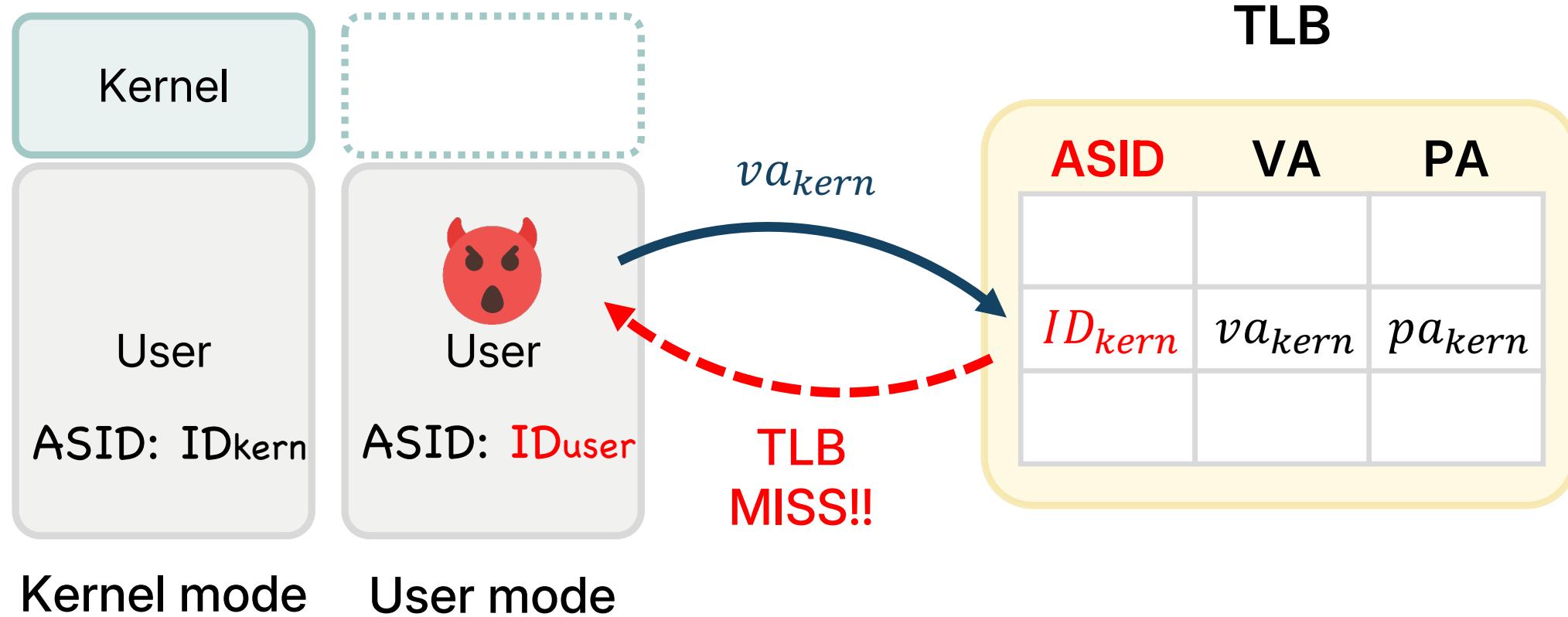
# Address Space Identifier



# Address Space Identifier



# Address Space Identifier

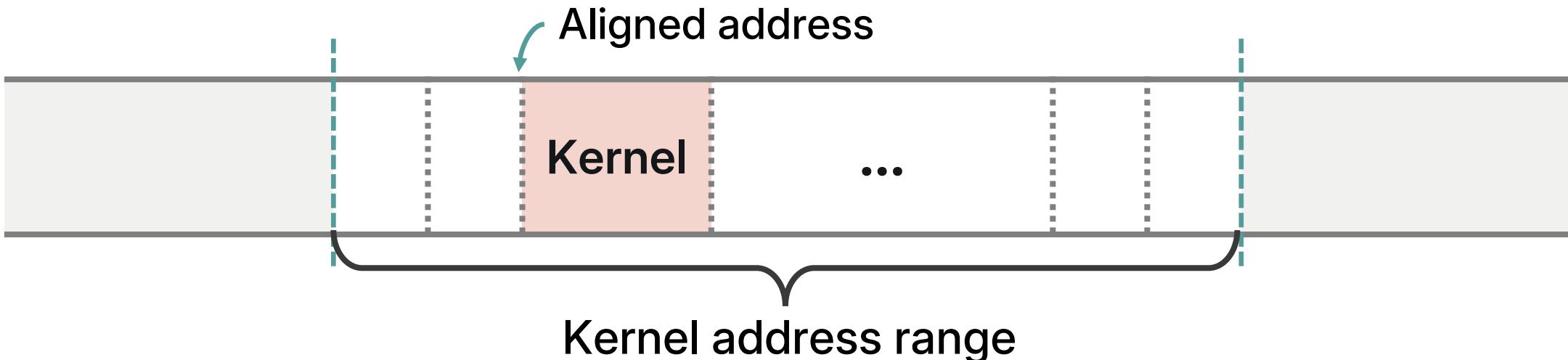


# Challenges to Breaking KASLR

- C1. How to **cache a kernel address** in the TLB?
  - Kernel page table isolation (KPTI)
- C2. How to **check if a kernel address is cached** in the TLB?
  - Address space identifier (ASID)

# Kernel Address Space Layout Randomization

- Kernel is loaded within a reserved range of kernel addresses
  - Aligned address to a specific size
    - Linux : 0xFFFF FFFF 8000 0000 ~ 0xFFFF FFFF C000 0000 (16MB aligned)
    - Windows : 0xFFFF F800 0000 0000 ~ 0xFFFF F804 0000 0000 (2MB aligned)
    - macOS for Apple silicon : ???



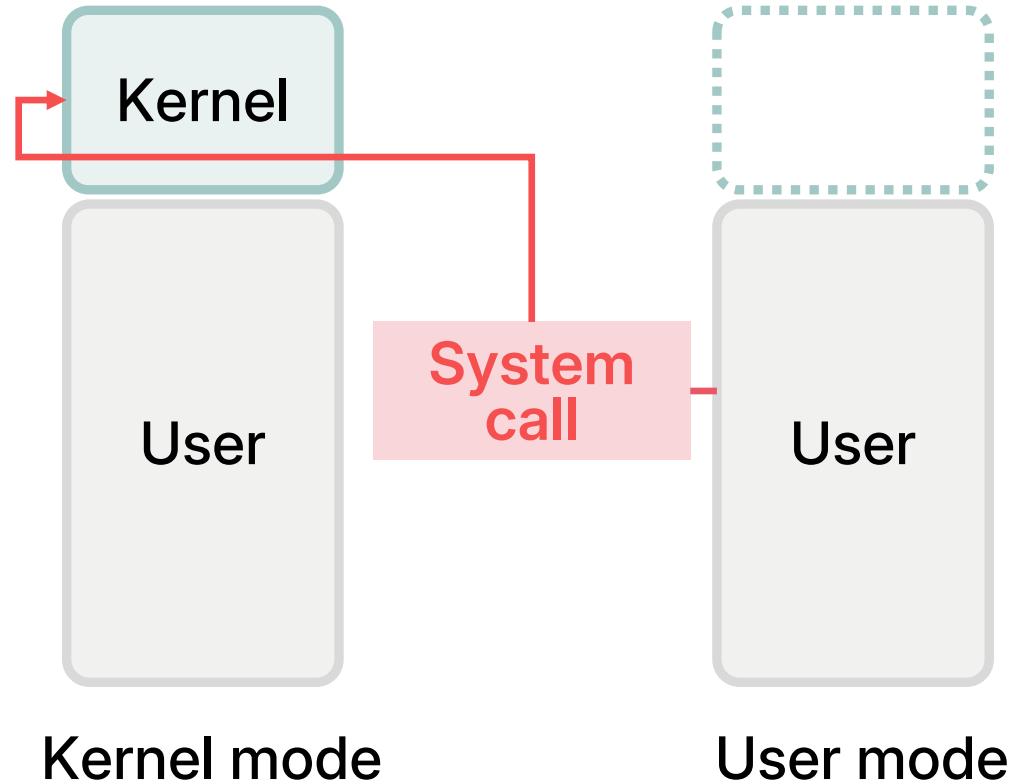
# Challenges to Breaking KASLR

- C1. How to **cache a kernel address** in the TLB?
  - Kernel page table isolation (KPTI)
- C2. How to **check if a kernel address is cached** in the TLB?
  - Address space identifier (ASID)
- C3. **KASLR implementation details** on macOS for Apple silicon

# Challenges to Breaking KASLR

- C1. How to cache a kernel address in the TLB?
  - Kernel page table isolation (KPTI)
- C2. How to check if a kernel address is cached in the TLB?
  - Address space identifier (ASID)
- C3. KASLR implementation details on macOS for Apple silicon

# Accessing the Inaccessible : Kernel Space under KPTI



**System call**

Operating in kernel mode  
&  
Passing user input to the kernel

```
int chdir(user_addr_t path)  
int pathconf(char *path, int name)
```

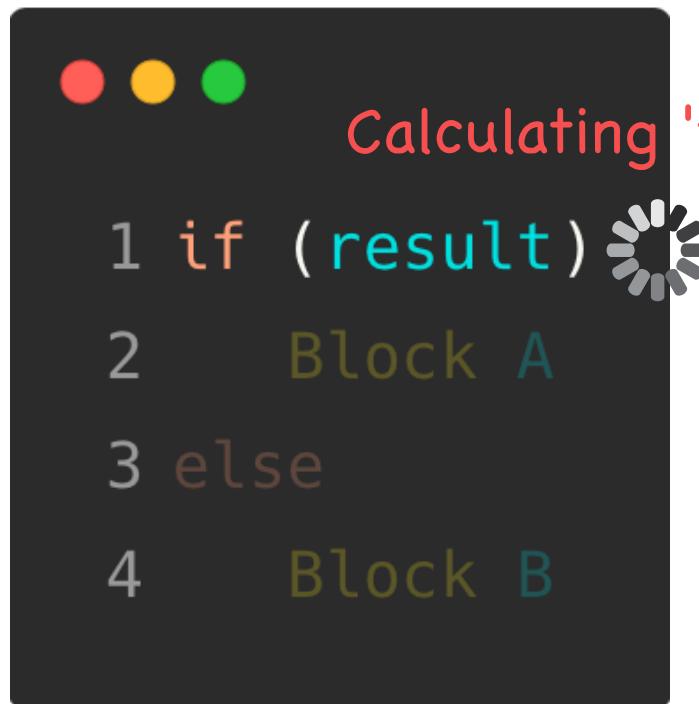
(+) Most system calls do not cause  
argument-induced faults

# Accessing the Inaccessible : Kernel Space under KPTI

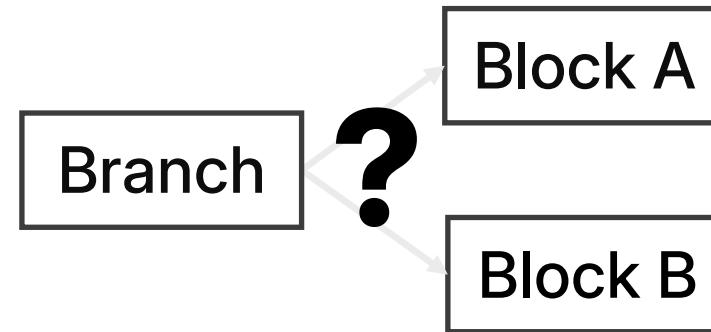


# Speculative Execution

Predict branch direction based on its history and execute in advance



```
● ● ●
Calculating 'result'...
1 if (result) 
2     Block A
3 else
4     Block B
```

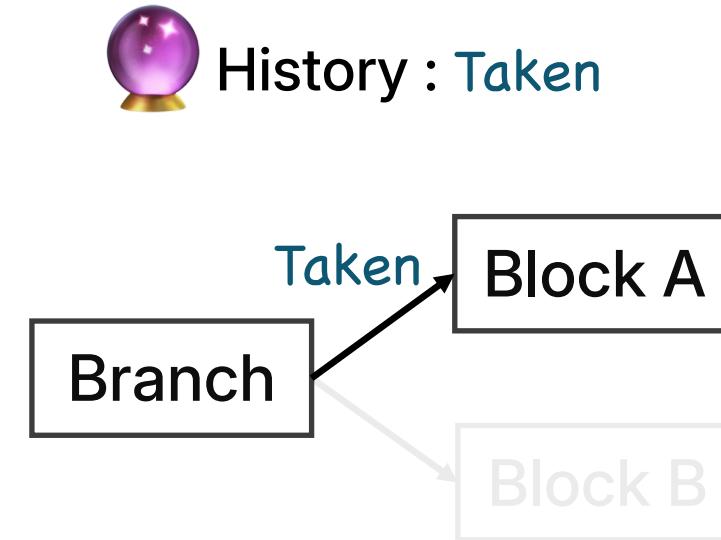


# Speculative Execution

Predict branch direction based on its history and execute in advance

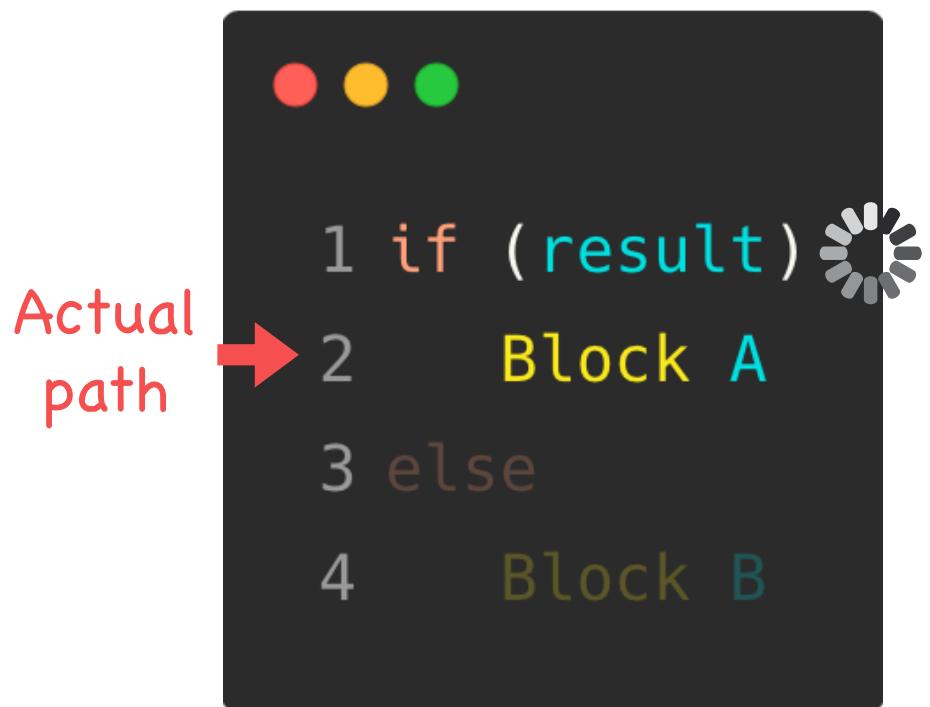
```
● ● ● Calculating 'result'...
1 if ( result )
2     Block A ←
3 else
4     Block B
```

Guess and go



# Speculative Execution

Predict branch direction based on its history and execute in advance



History : Taken

Correct Prediction : 

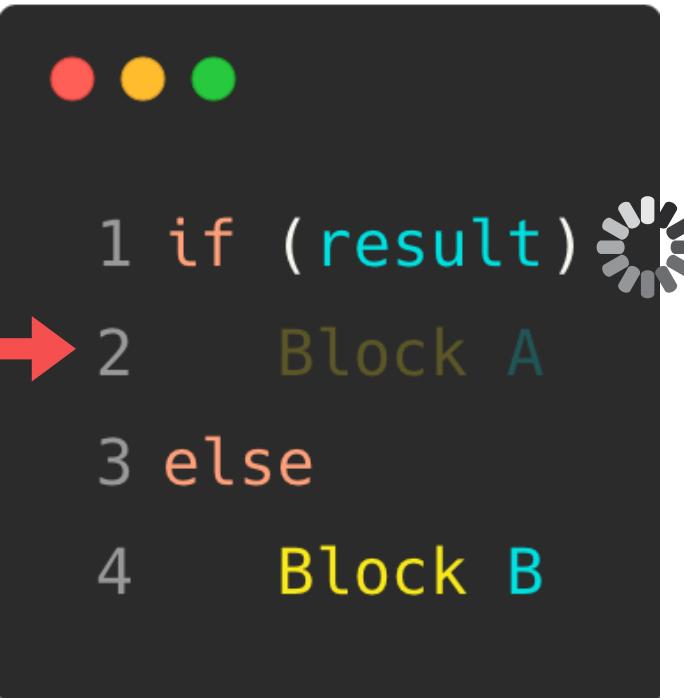
Branch	Block A
--------	---------

Wrong Prediction : 

Branch	Block B	Undo	Block A
--------	---------	------	---------

# Speculative Execution

Predict branch direction based on its history and execute in advance



A screenshot of a debugger interface showing assembly code. The code is:

```
1 if (result)    ; Branch prediction history icon
2     Block A
3 else
4     Block B
```

A red arrow points to the number 2, labeled "Actual path" in red text, indicating the path that was actually taken.



History : Not taken

Correct  
Prediction :

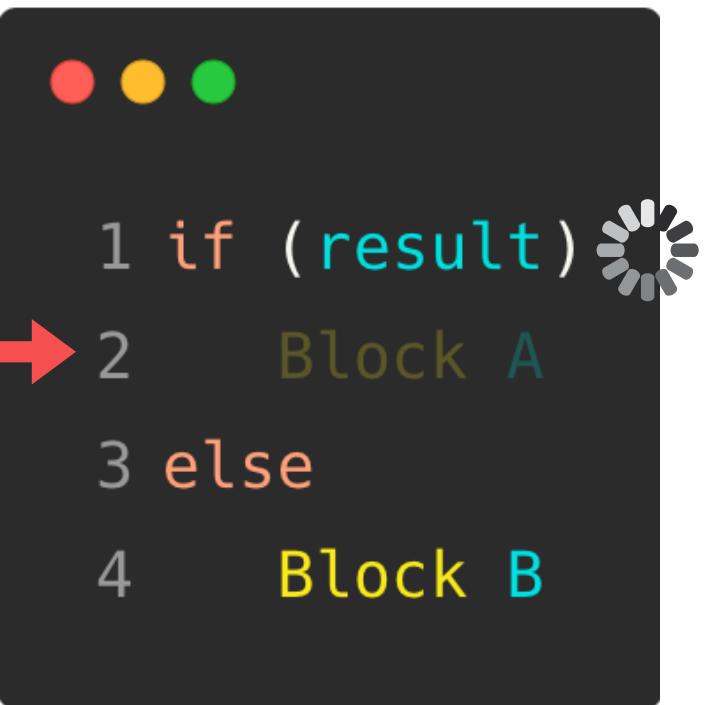


Wrong  
Prediction :



# Speculative Execution

Predict branch direction based on its history and execute in advance



A screenshot of a debugger interface showing assembly code. The code is:

```
1 if (result)    ; Branch predictor icon
2     Block A
3 else
4     Block B
```

An orange arrow points to the first instruction, labeled "Actual path". Above the code, there are three colored circles (red, yellow, green) followed by a small loading icon.

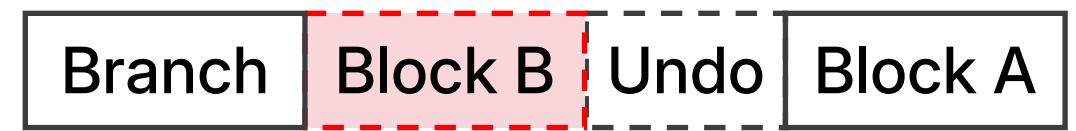


History : Not taken

Correct  
Prediction :



Wrong  
Prediction :



Remains in cache !

# SysBumps Gadget in macOS for Apple silicon

```
int chdir(char *path)
```



```
1 ...
2 result = copy_validate(path, ...);
3
4 if(result){
5     return result;
6 }
7
8 result = _bcopyinstr(path, ...);
9 ...
```

# SysBumps Gadget in macOS for Apple silicon

```
int chdir(char *path)
```



```
1 ...
2 result = copy_validate(path, ...);(1)
3
4 if(result){
5     return result;
6 }
7
8 result = _bcopyinstr(path, ...);
9 ...
```

1. Validate 'path'

# SysBumps Gadget in macOS for Apple silicon

```
int chdir(char *path)
```

```
1 ...
2 result = copy_validate(path, ...);(1)
3
4 if(result){(2)
5     return result;
6 }
7
8 result = _bcopyinstr(path, ...);
9 ...
```

1. Validate 'path'



2. Check the result

# SysBumps Gadget in macOS for Apple silicon

```
int chdir(char *path)
```

```
1 ...
2 result = copy_validate(path, ...);(1)
3
4 if(result){(2)
5     return result;
6 }
7
8 result = _bcopyinstr(path, ...); (3)
9 ...
```

1. Validate 'path'



2. Check the result



3. Access 'path'

# SysBumps Gadget in macOS for Apple silicon

```
int chdir(char *path)
```

```
1 ...
2 result = copy_validate(path, ...);(1)
3
4 if(result){(2)
5     return result;
6 }
7
8 result = _bcopyinstr(path, ...);
9 ...
```

If 'path' is kernel address ...

1. Validate 'path'



2. Check the result



3. Access 'path'

# SysBumps Gadget in macOS for Apple silicon

```
int chdir(char *path)
```

```
1 ...
2 result = copy_validate(path, ...);(1)
3
4 if(result){(2) ⏪
5     return result;
6 }
7
8 result = _bcopyinstr(path, ...);
9 ...
```

If 'path' is kernel address ...

1. Validate 'path'



2. Check the result



3. Access 'path'

# SysBumps Gadget in macOS for Apple silicon

```
int chdir(char *path)
```

```
● ● ●  
1 ...  
2 result = copy_validate(path, ...);(1)  
3  
4 if(result){ (2) → Training  
5   return result; } for wrong speculation  
6 }  
7  
8 result = _bcopyinstr(path, ...); Goal  
9 ...
```

If 'path' is kernel address ...

1. Validate 'path'



2. Check the result



3. Access 'path'

# SysBumps Gadget in macOS for Apple silicon

```
int chdir(char *path)
```

```
1 ...
2 result = copy_validate(path, ...);(1)
3
4 if(result){ (2) Not taken
5     return result;
6 }
7
8 result = _bcopyinstr(path, ...);
9 ...
```

If 'path' is kernel address ...

1. Validate 'path'



2. Check the result



3. Access 'path'

Guess and go

# Challenges to Breaking KASLR

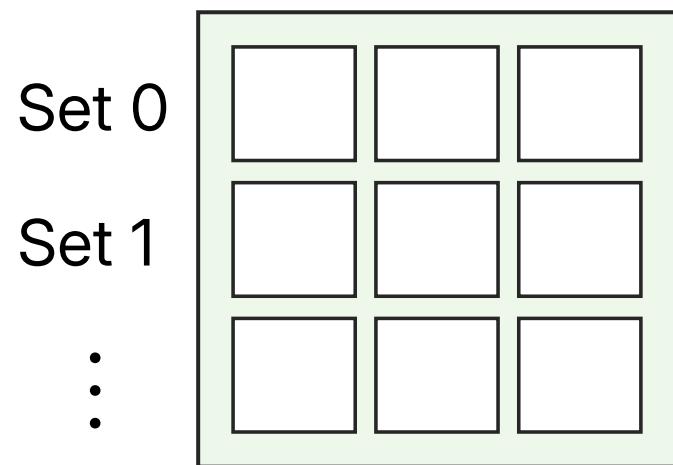
-  C1. How to cache a kernel address in the TLB?
  - Kernel page table isolation (KPTI)
- C2. How to check if a kernel address is cached in the TLB?
  - Address space identifier (ASID)
- C3. KASLR implementation details on macOS for Apple silicon

# Challenges to Breaking KASLR

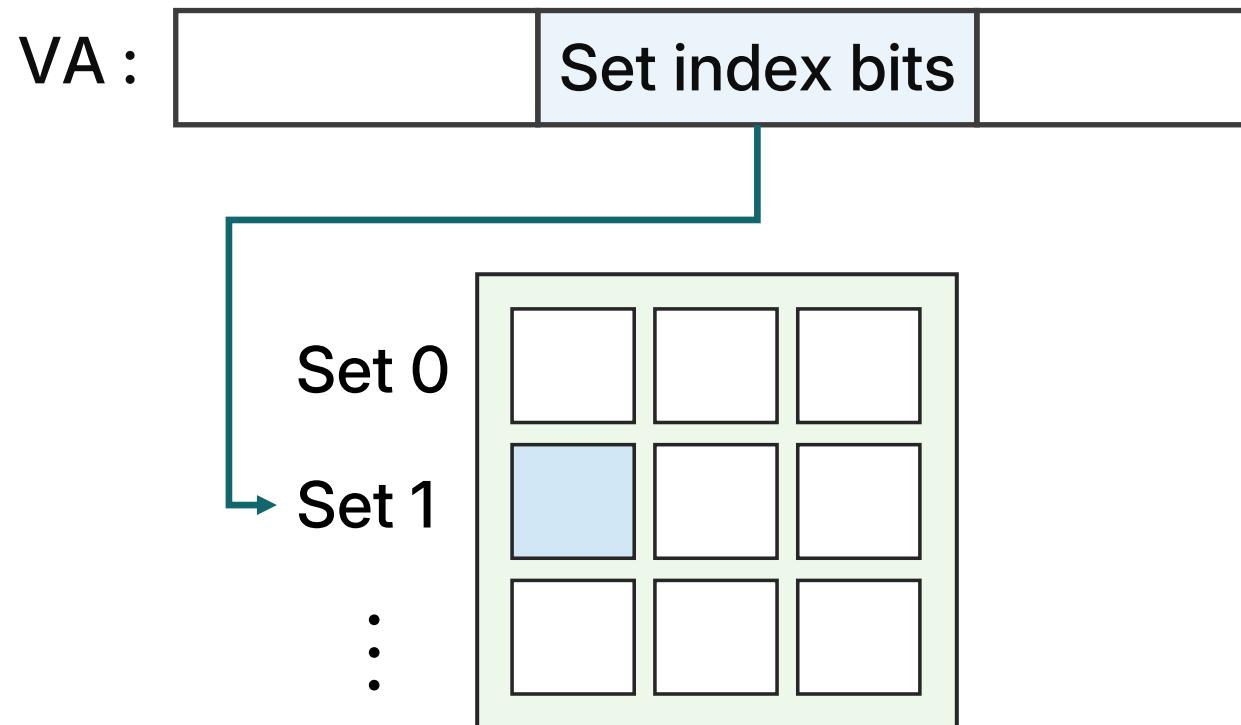
-  C1. How to **cache a kernel address** in the TLB?
  - Kernel page table isolation (KPTI)
- C2. How to **check if a kernel address is cached** in the TLB?
  - Address space identifier (ASID)
- C3. KASLR implementation details on macOS for Apple silicon

# **Prime + Probe**

# Prime + Probe



# Prime + Probe



# Prime + Probe

(Goal) Observe  
address  $v$  is loaded



1. Calculate eviction set  
for address  $v$

$$\text{EvictionSet}(v) = \{A, B, C\}$$

# Prime + Probe

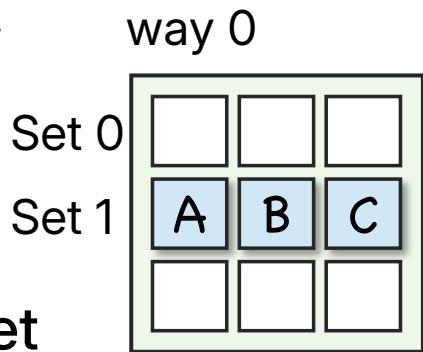
## 2.Primes

(Goal) Observe  
address  $v$  is loaded

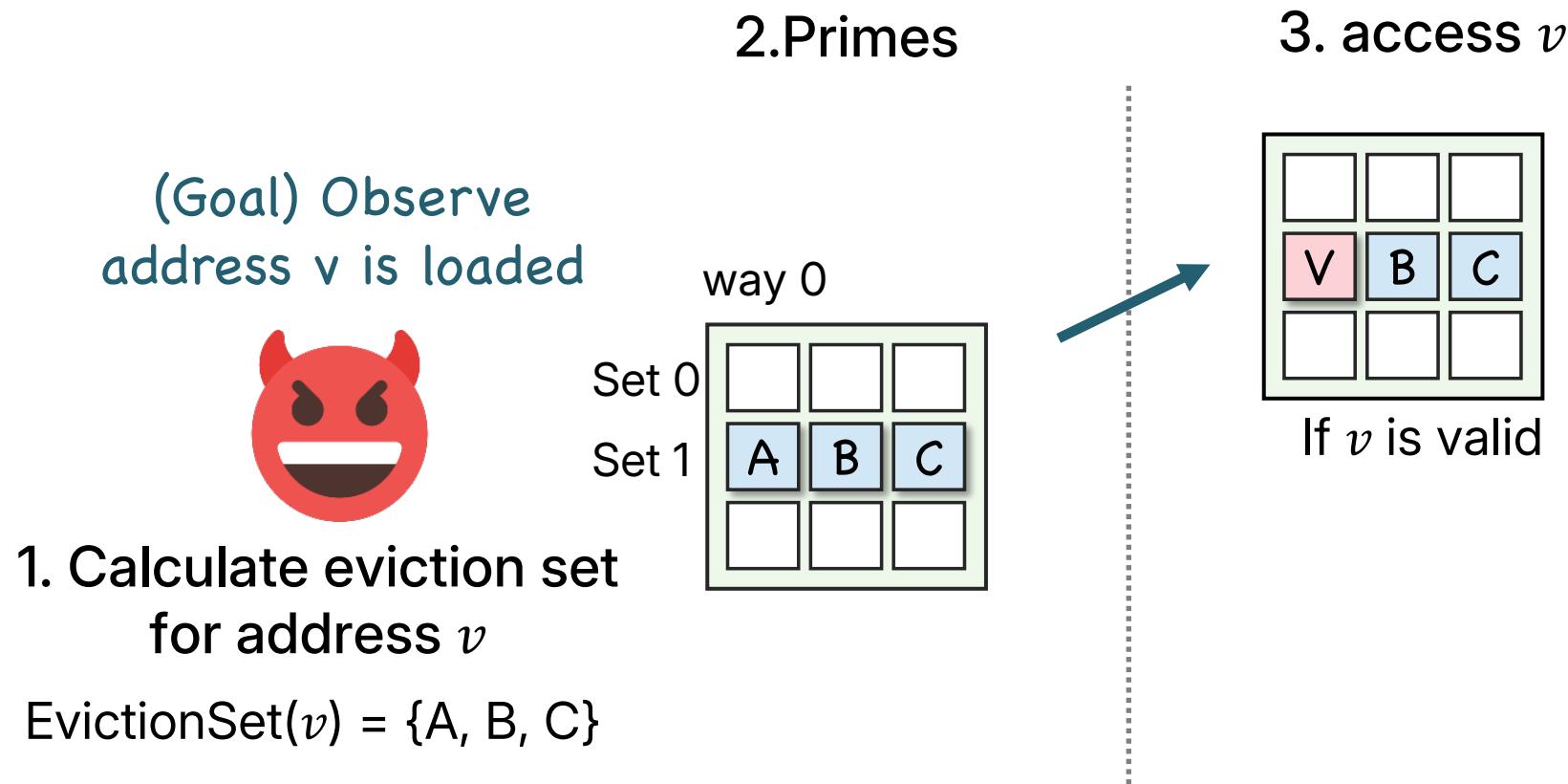


1. Calculate eviction set  
for address  $v$

$$\text{EvictionSet}(v) = \{A, B, C\}$$



# Prime + Probe



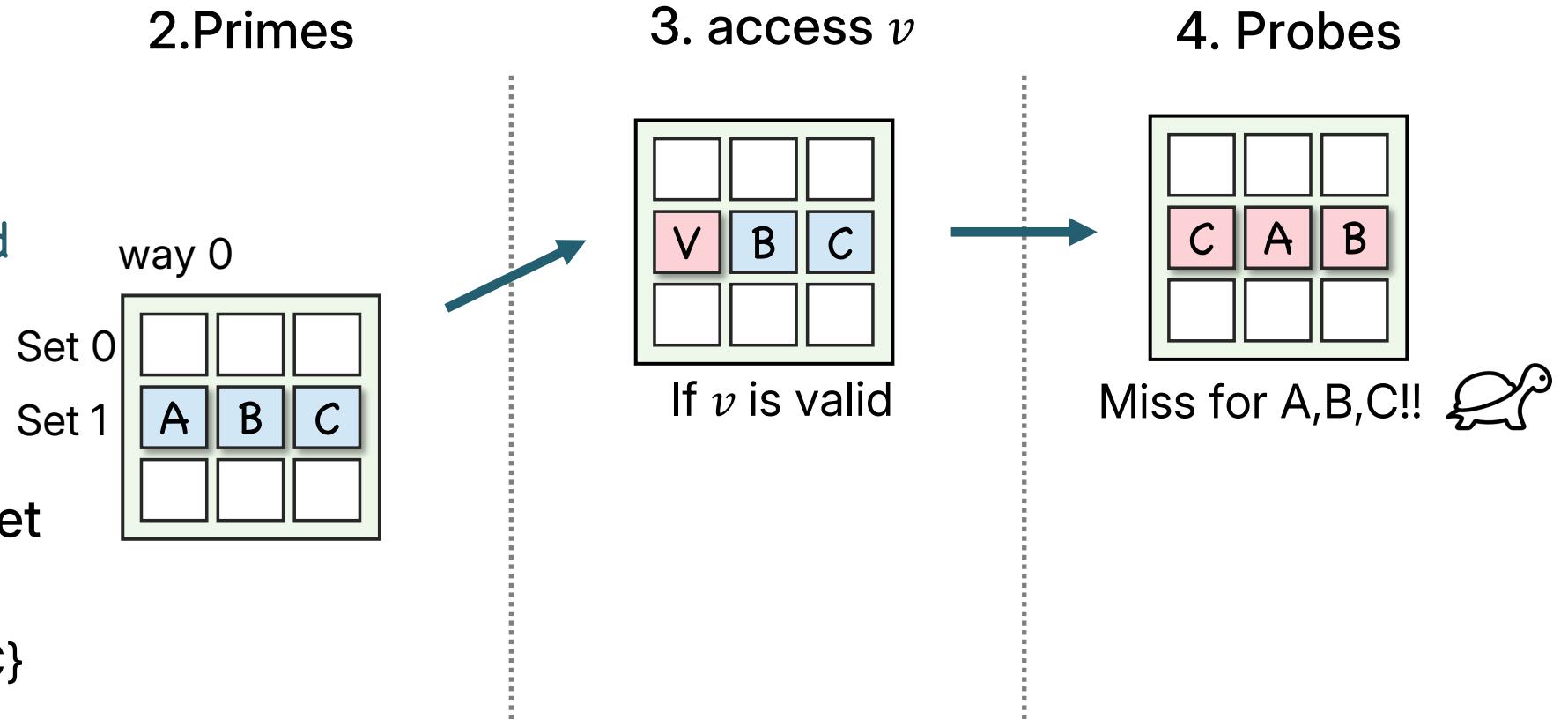
# Prime + Probe

(Goal) Observe  
address  $v$  is loaded

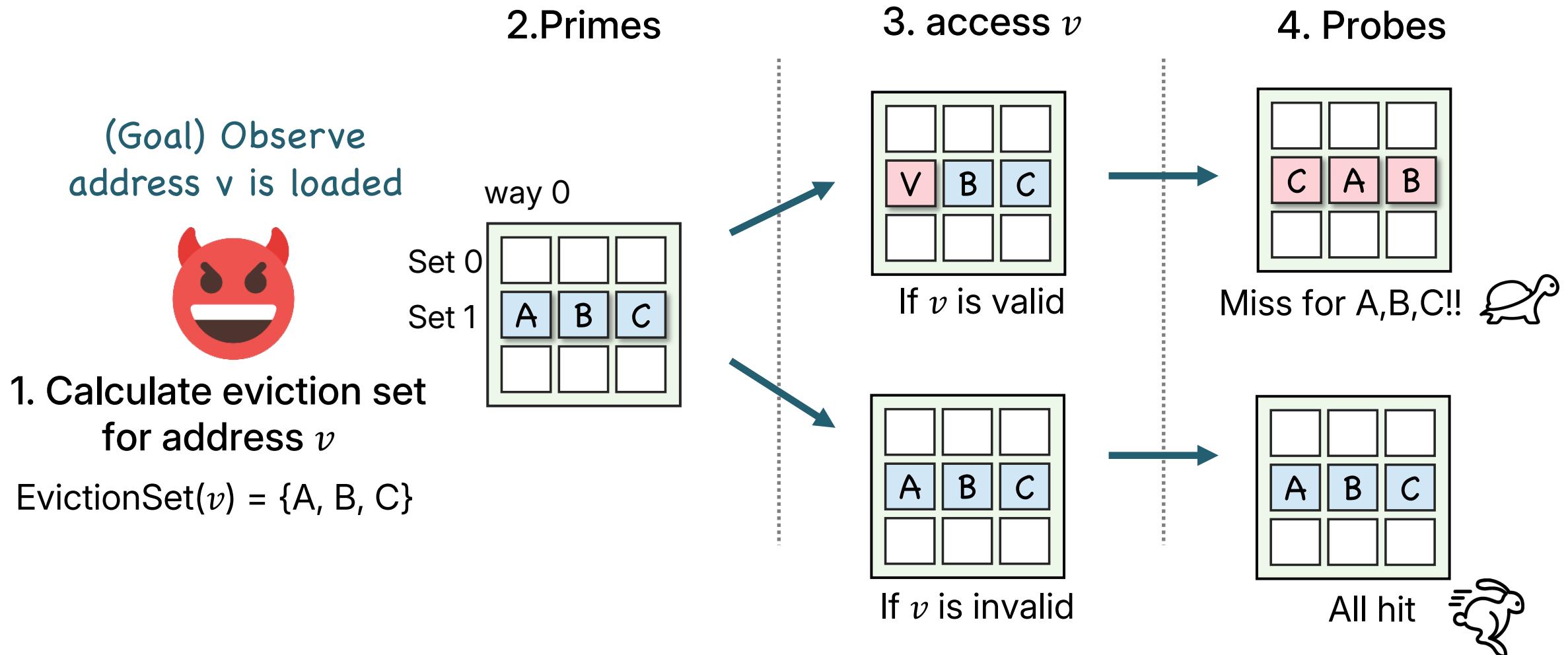


1. Calculate eviction set  
for address  $v$

$$\text{EvictionSet}(v) = \{A, B, C\}$$



# Prime + Probe



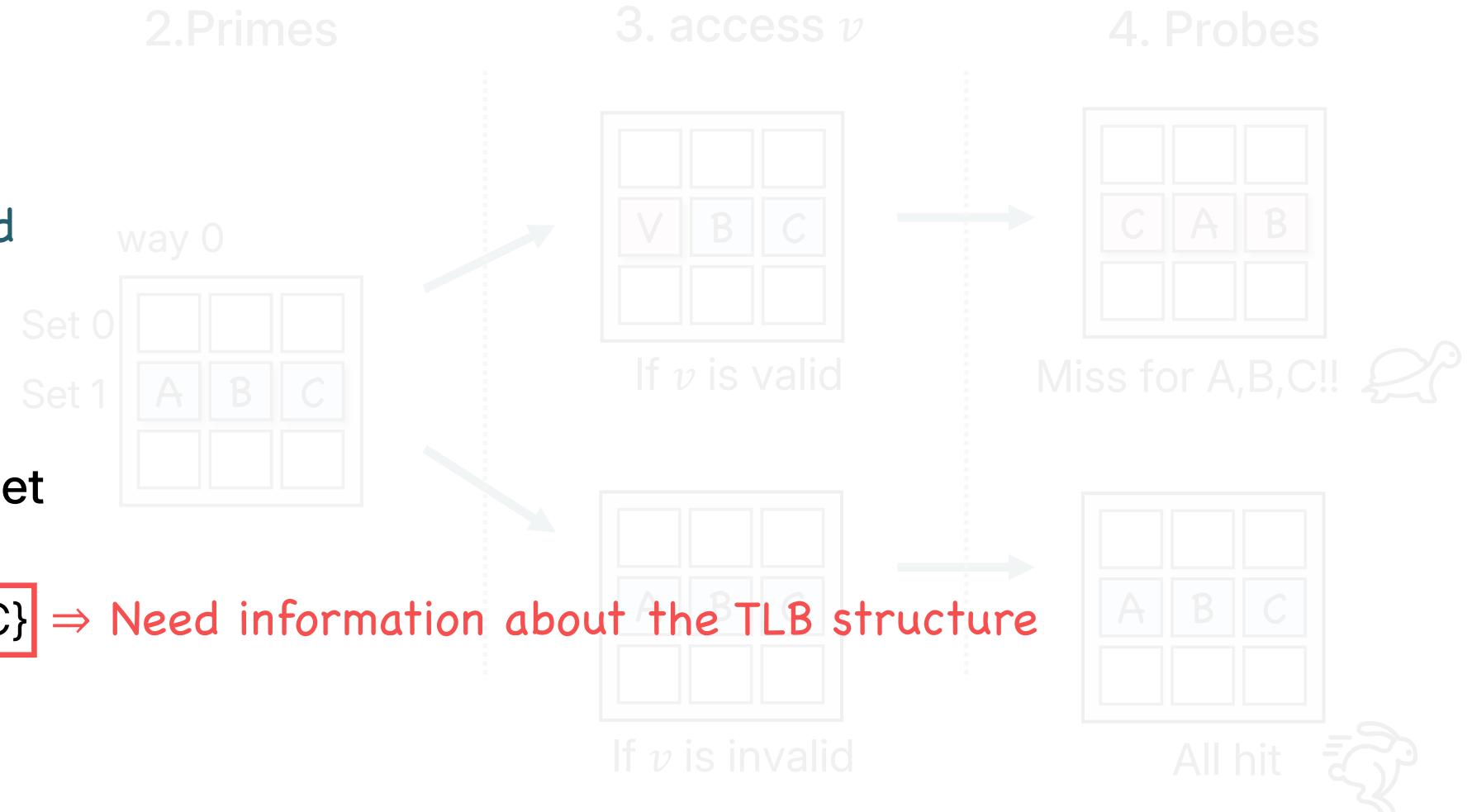
# Prime + Probe

(Goal) Observe  
address  $v$  is loaded



1. Calculate eviction set  
for address  $v$

$\text{EvictionSet}(v) = \{A, B, C\}$   $\Rightarrow$  Need information about the TLB structure



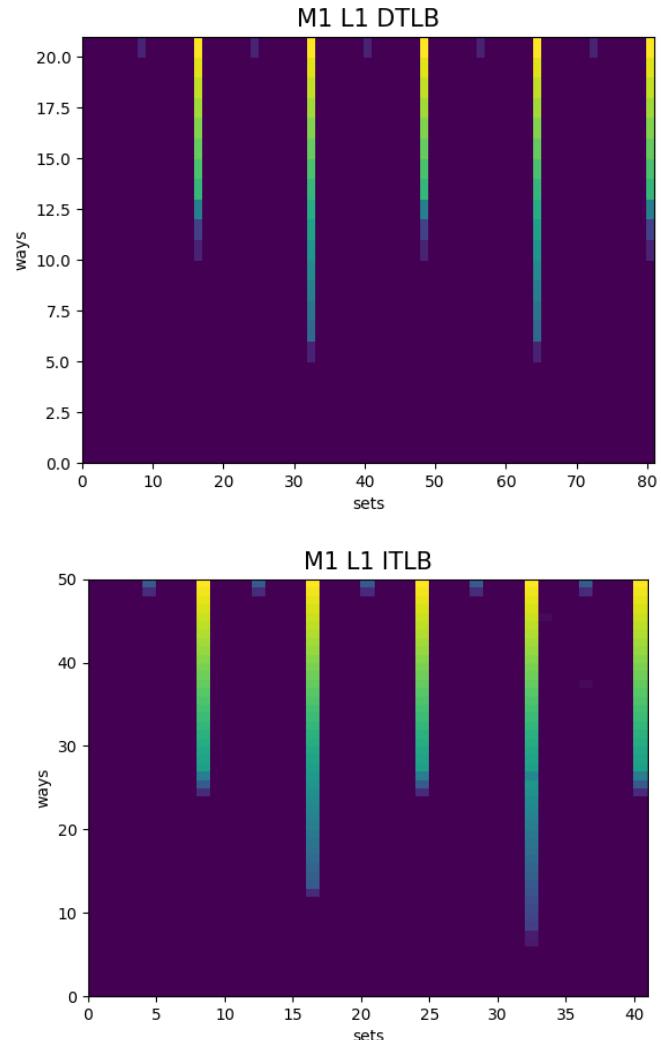
# TLB Reverse Engineering

- Experimental environment

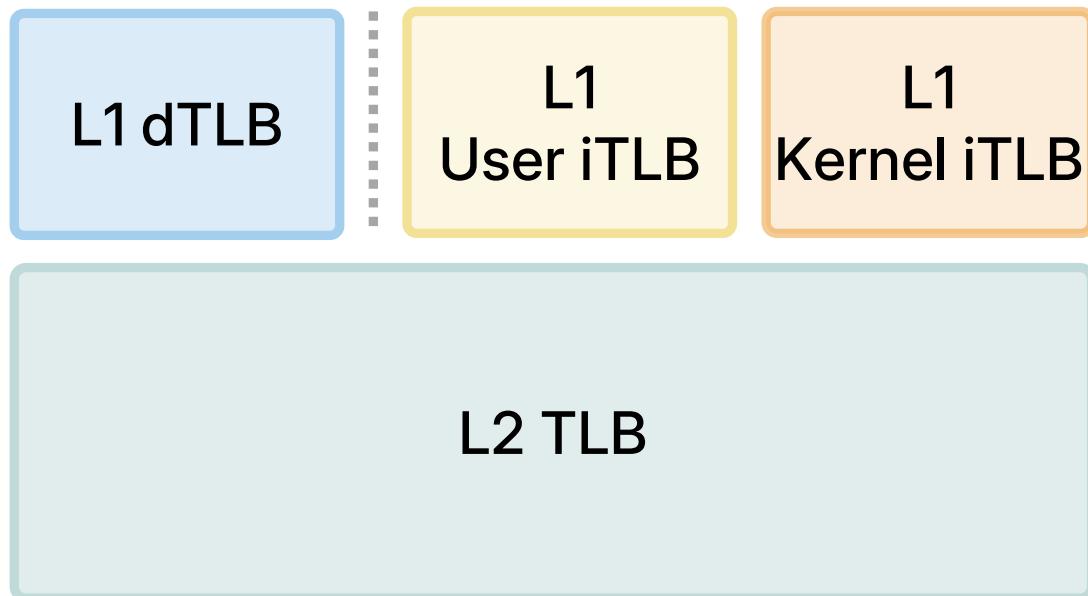
CPU	Device	OS
M1	Mac Mini (2021)	macOS Sonoma 14.3
M1 Pro	MacBook Pro 16	macOS Ventura 13.5
M2	Mac Mini (2023)	macOS Ventura 13.2
M2 Pro	Mac Mini (2023)	macOS Ventura 13.2
M2 Max	MacBook Pro 14	macOS Ventura 13.4

# TLB Reverse Engineering

- Using CPU performance counters
  - Ibireme/kpc\_demo.c [1]
- Counters
  - L1D\_TLB\_MISS\_NONSPEC
  - L1I\_TLB\_MISS\_DEMAND
  - L2\_TLB\_MISS\_DATA
  - L2\_TLB\_MISS\_INSTRUCTION

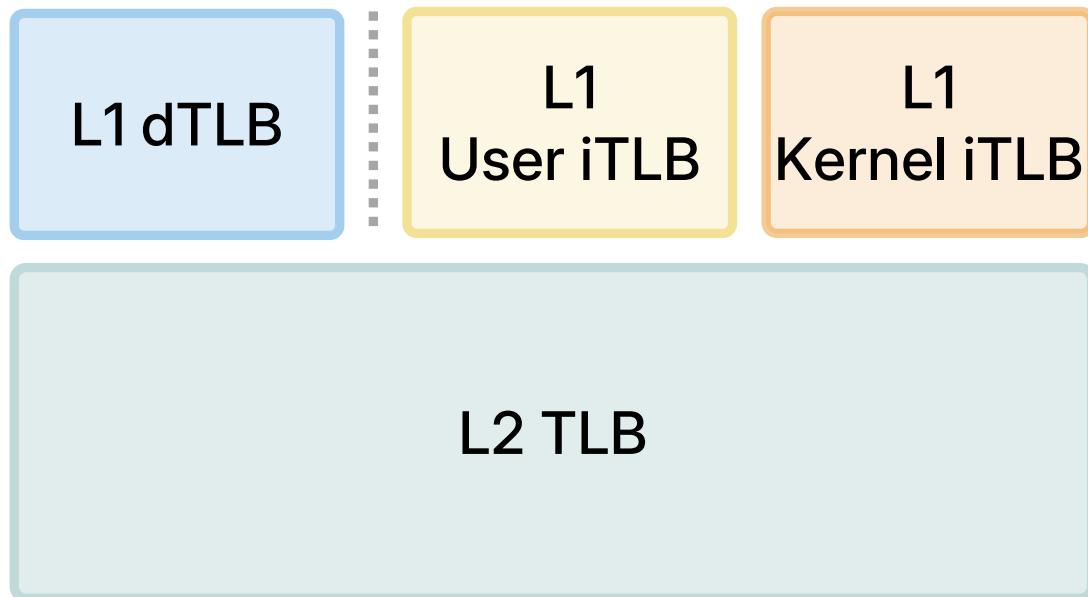


# Empirical TLB hierarchy



	M1-series	M2-series
L1 dTLB	32-set, 5-way	64-set, 4-way
L1 iTLB	32-set, 6-way	32-set, 6-way
L2 TLB	256-set, 12-way	256-set, 12-way

# Empirical TLB hierarchy



	M1-series	M2-series
✓ L1 dTLB	32-set, 5-way	64-set, 4-way
L1 iTLB	32-set, 6-way	32-set, 6-way
✓ L2 TLB	256-set, 12-way	256-set, 12-way

# Challenges to Breaking KASLR

-  C1. How to cache a kernel address in the TLB?
  - Kernel page table isolation (KPTI)
-  C2. How to check if a kernel address is cached in the TLB?
  - Address space identifier (ASID)
- C3. KASLR implementation details on macOS for Apple silicon

# Challenges to Breaking KASLR

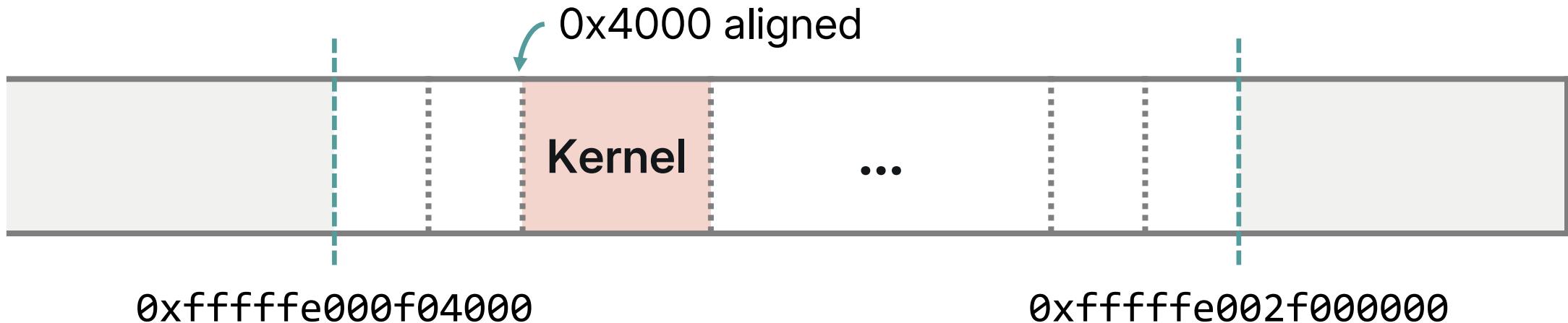
-  C1. How to **cache a kernel address** in the TLB?
  - Kernel page table isolation (KPTI)
-  C2. How to **check if a kernel address is cached** in the TLB?
  - Address space identifier (ASID)
- C3. **KASLR implementation details** on macOS for Apple silicon

# KASLR entropy on macOS for Apple silicon

- Empirical analysis : Reboot 50,000 times to collect kernel base

# KASLR entropy on macOS for Apple silicon

- Empirical analysis : Reboot 50,000 times to collect kernel base
- 32,515 possible kernel base addresses, (approximately 15bits entropy)



# Challenges to Breaking KASLR

-  1. How to **cache a kernel address** in the TLB?
  - Kernel page table isolation (KPTI)
-  2. How to **check if a kernel address is cached** in the TLB?
  - Address space identifier (ASID)
-  3. **KASLR implementation details** on macOS for Apple silicon

# Distinguishing Oracle with SysBumps Gadget

## 1. Training



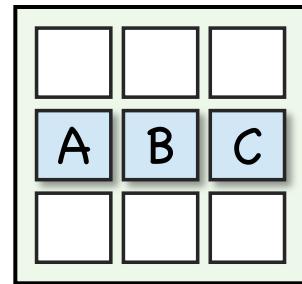
```
syscall(user_addr);
syscall(user_addr);
syscall(user_addr);
```

# Distinguishing Oracle with SysBumps Gadget

## 1. Training

```
● ● ●  
syscall(user_addr);  
syscall(user_addr);  
syscall(user_addr);
```

## 2. Primes



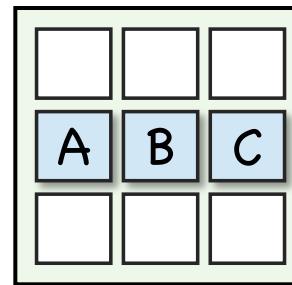
$$\begin{aligned}\text{EvictionSet}(\nu) \\ = \{A, B, C\}\end{aligned}$$

# Distinguishing Oracle with SysBumps Gadget

## 1. Training

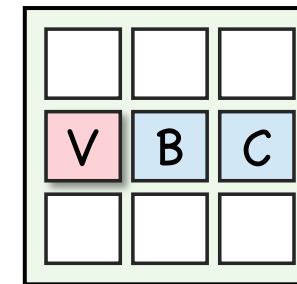
```
● ● ●  
syscall(user_addr);  
syscall(user_addr);  
syscall(user_addr);
```

## 2. Primes

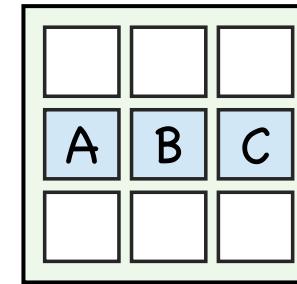


$\text{EvictionSet}(\nu) = \{A, B, C\}$

## 3. Syscall with $\nu$



If  $\nu$  is valid



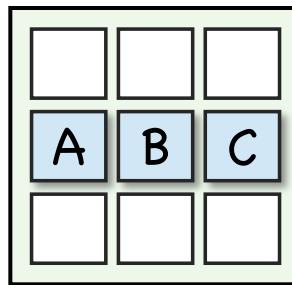
If  $\nu$  is invalid

# Distinguishing Oracle with SysBumps Gadget

## 1. Training

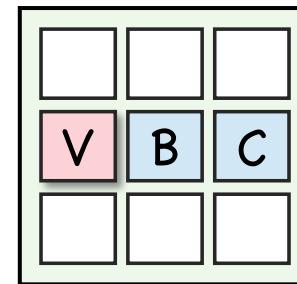
```
● ● ●  
syscall(user_addr);  
syscall(user_addr);  
syscall(user_addr);
```

## 2. Primes

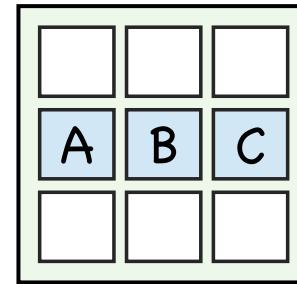


EvictionSet( $v$ )  
= {A, B, C}

## 3. Syscall with $v$

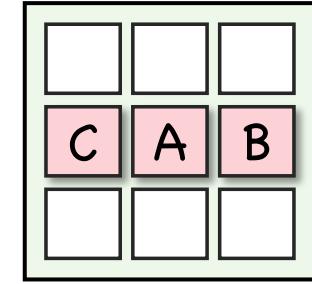


If  $v$  is valid

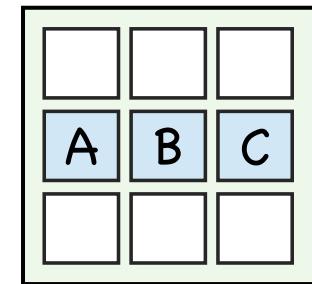


If  $v$  is invalid

## 4. Probes



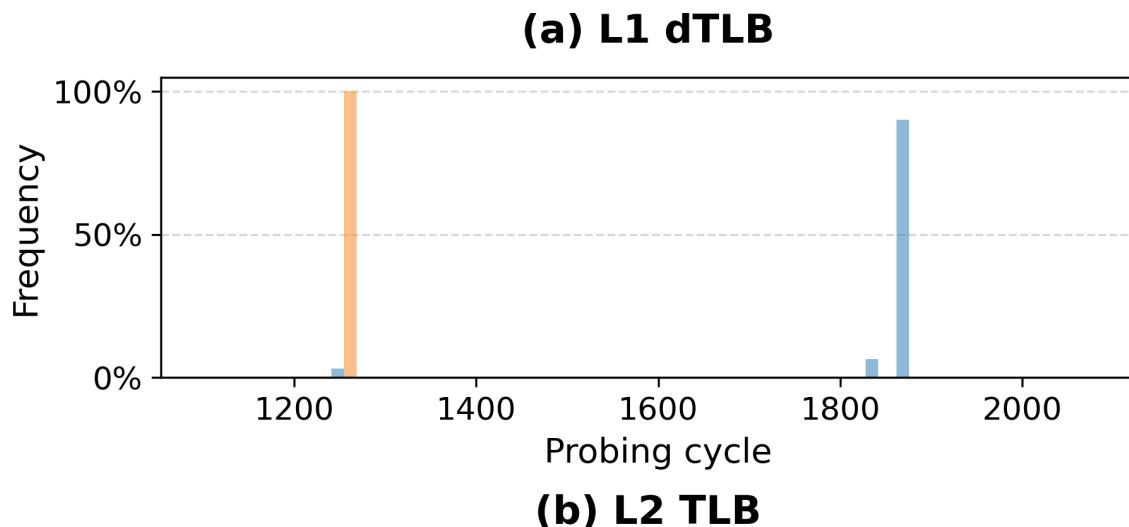
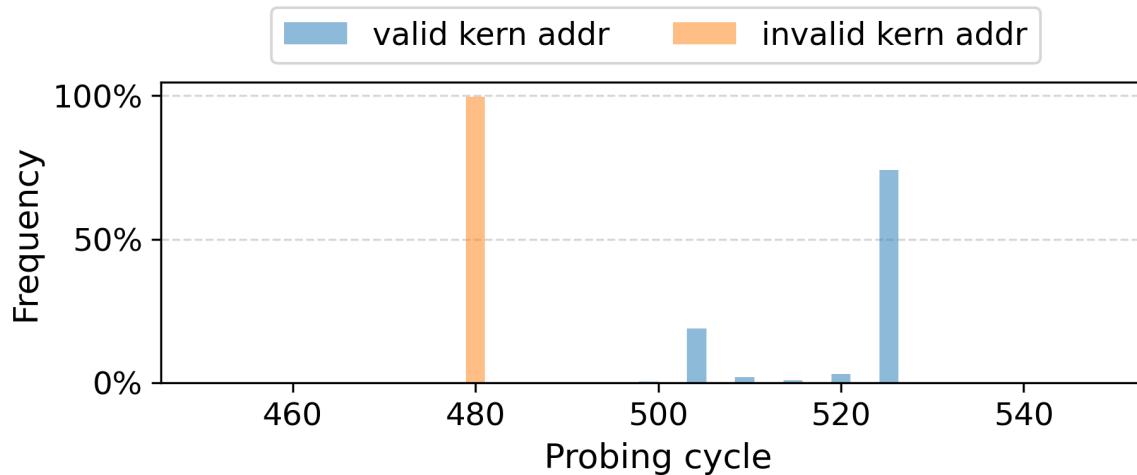
Miss for A,B,C!!



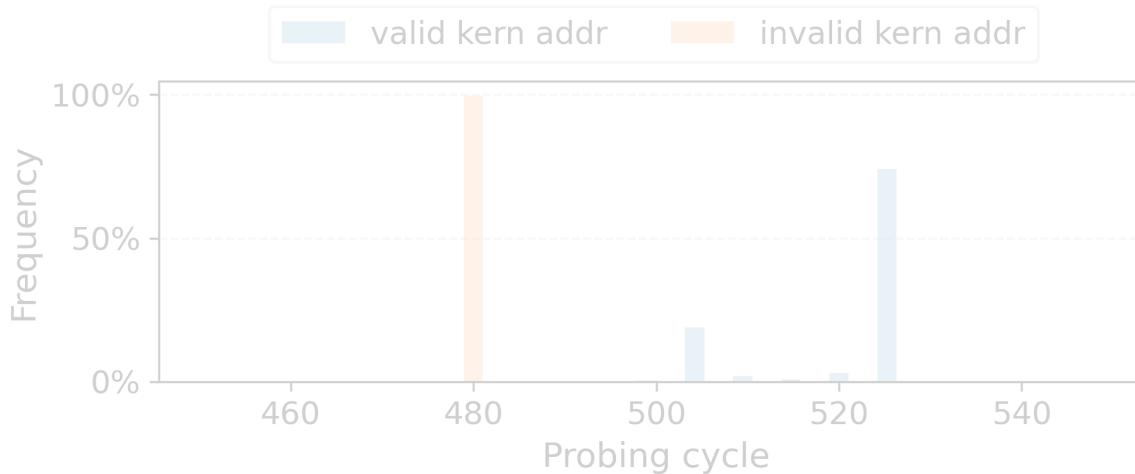
All hit



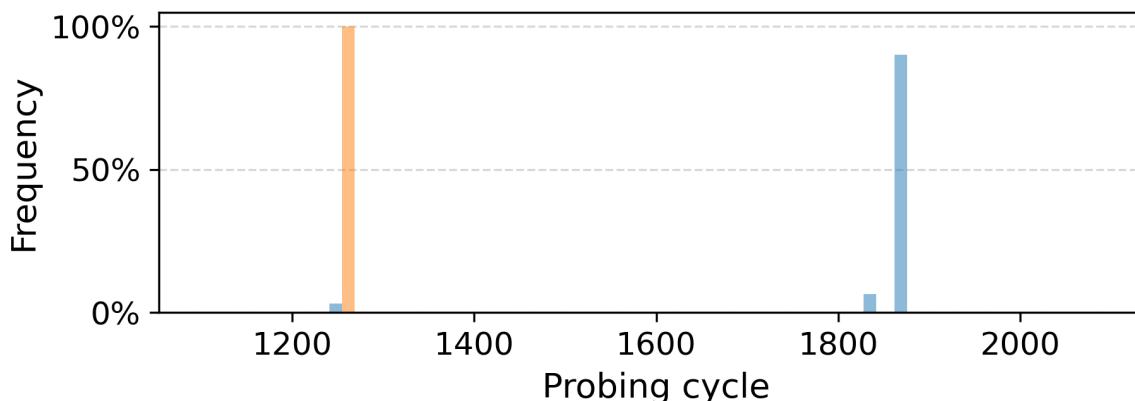
# Accuracy of Distinguishing Oracle



# Accuracy of Distinguishing Oracle



(a) L1 dTLB



(b) L2 TLB



# Exploitable System calls

Syscall no.	Prototype	Description
5	int access(const char *path, int amode)	Check access permissions of a file or pathname
9	int link(user_addr_t path, user_addr_t link)	Make a hard file link
10	int unlink(user_addr_t path)	Remove directory entry
12	int chdir(user_addr_t path)	Change current working directory
15	int chmod(user_addr_t path, int mode)	Change mode of file
34	int chflags(char *path, int flags)	Set file flags
56	int revoke(char *path)	Revoke file access
57	int symlink(char *path, char *link)	Make symbolic link to a file
58	int readlink(char *path, char *buf, int count)	Read value of a symbolic link
136	int mkdir(user_addr_t path, int mode)	Make a directory file
165	int quotactl(user_addr_t path, int mode)	Manipulate filesystem quotas
188	int stat(const char *path, int cmd, ... )	Get file status
190	int lstat(user_addr_t path, user_addr_t ub)	Get file status
191	int pathconf(char *path, int name)	Get configurable pathname variables
200	int truncate(char *path, off_t length)	Truncate or extend a file to a specified length
220	int getatrlst(const char *path, struct attrlist *alist, ... )	Get file system attributes
221	int setatrlst(const char *path, struct attrlist *alist, ... )	Set file system attributes
223	int exchangedata(const char *path1, const char *path2, ... )	Atomically exchange data between two files
234	user_size_t getxattr(user_addr_t path, user_addr_t attrname, ... )	Get an extended attribute value
235	user_size_t fgetxattr(int fd, user_addr_t attrname, ... )	Get an extended attribute value

# Exploitable System calls

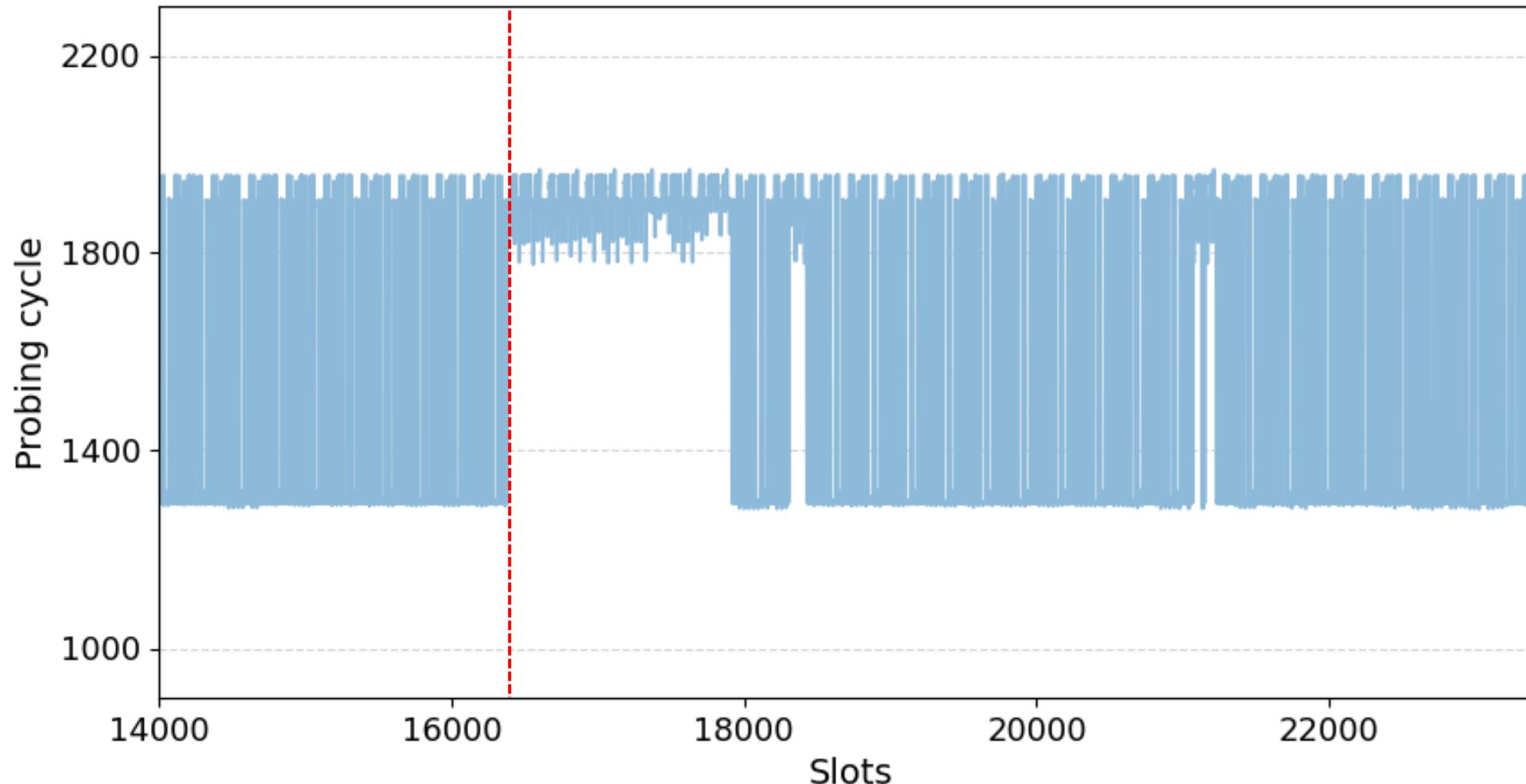
Syscall no.	Prototype	Description
5	int access(const char *path, int amode)	Check access permissions of a file or pathname
9	int link(user_addr_t path, user_addr_t link)	Make a hard file link
10	int unlink(user_addr_t path)	Remove directory entry
12	int chdir(user_addr_t path)	Change current working directory
15	int chmod(user_addr_t path, int mode)	Change mode of a file
34	int chflags(char *path, int flags)	Change file flags
56	int revoke(char *path)	Revoke a symbolic link to a file
57	int symlink(char *path, char *link)	Read value of a symbolic link
58	int readlink(char *path)	Make a directory file
136	int mknod(char *path, dev_t dev, mode_t mode)	Manipulate filesystem quotas
140	int stat(user_addr_t path, user_addr_t ub)	Get file status
141	int fstat(user_addr_t fd, user_addr_t ub)	Get file status
191	int pathconf(char *path, int name)	Get configurable pathname variables
200	int truncate(char *path, off_t length)	Truncate or extend a file to a specified length
220	int setattrlist(const char *path, struct attrlist *alist, ... )	Get file system attributes
221	int setattrlist(const char *path, struct attrlist *alist, ... )	Set file system attributes
223	int exchangedata(const char *path1, const char *path2, ... )	Atomically exchange data between two files
234	user_size_t getxattr(user_addr_t path, user_addr_t attrname, ... )	Get an extended attribute value
235	user_size_t fgetxattr(int fd, user_addr_t attrname, ... )	Get an extended attribute value

25 system calls can be used for SysBumps attack!!

# Exploring Kernel Space with SysBumps

- Demo

# Exploring Kernel Space with SysBumps



# Evaluation

- Boot 100 times and perform 10 attacks on different kernel bases
  - Total 1000 attacks

# Evaluation

- Boot 100 times and perform 10 attacks on different kernel bases
  - Total 1000 attacks

	chdir()		getxattr()		pathconf()	
	Accuracy	Time(s)	Accuracy	Time(s)	Accuracy	Time(s)
M1	98.8%	2.34	96.6%	2.35	98.0%	2.33
M1 Pro	95.7%	2.54	92.1%	2.52	94.5%	2.50
M2	97.6%	2.11	97.9%	2.12	97.0%	2.09
M2 Pro	96.3%	2.21	94.9%	2.21	95.9%	2.15
M2 Max	97.0%	2.15	95.9%	2.14	96.0%	2.09

# Evaluation

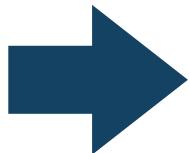
- Boot 100 times and perform 10 attacks on different kernel bases
  - Total 1000 attacks

**“We can find the kernel base”  
with an average accuracy of 96.58%  
in under 3 seconds using SysBumps!**

# Mitigations

- Preventing speculative execution using **fence instructions**
  - Inserts serializing instruction, such as DSB, ISB for ARM64, before the conditional branch

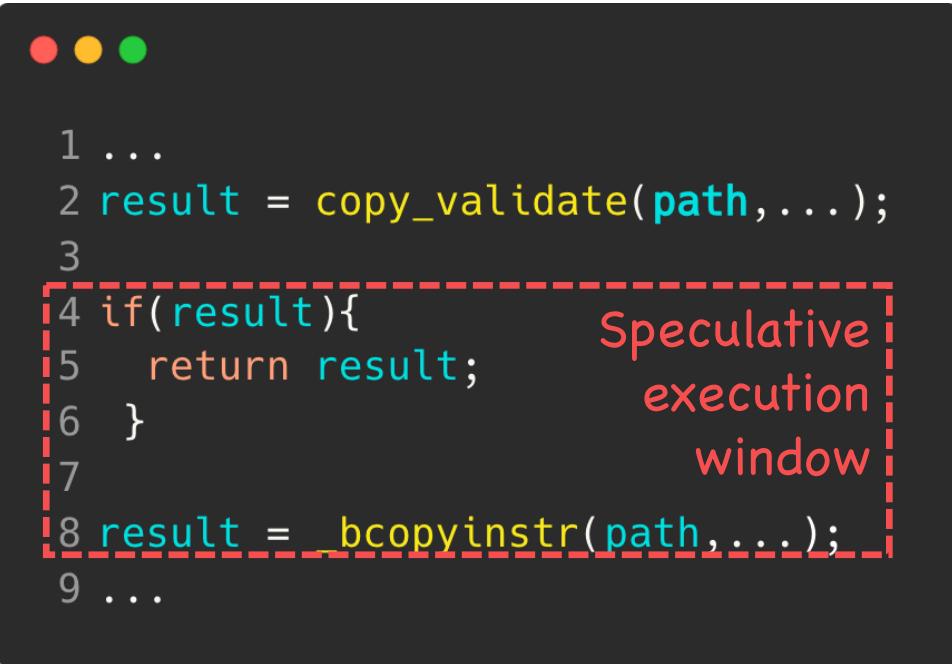
```
● ● ●  
1 ...  
2 result = copy_validate(path,...);  
3  
4 if(result){  
5     return result;  
6 }  
7  
8 result = _bcopyinstr(path,...);  
9 ...
```



```
● ● ●  
1 ...  
2 result = copy_validate(path,...);  
3  
4 __asm__("dsb sy;"  
5          "isb sy;"  
6          ::: "memory")  
7  
8 if(result){  
9     return result;  
10 }  
11  
12 result = _bcopyinstr(path,...);  
13 ...
```

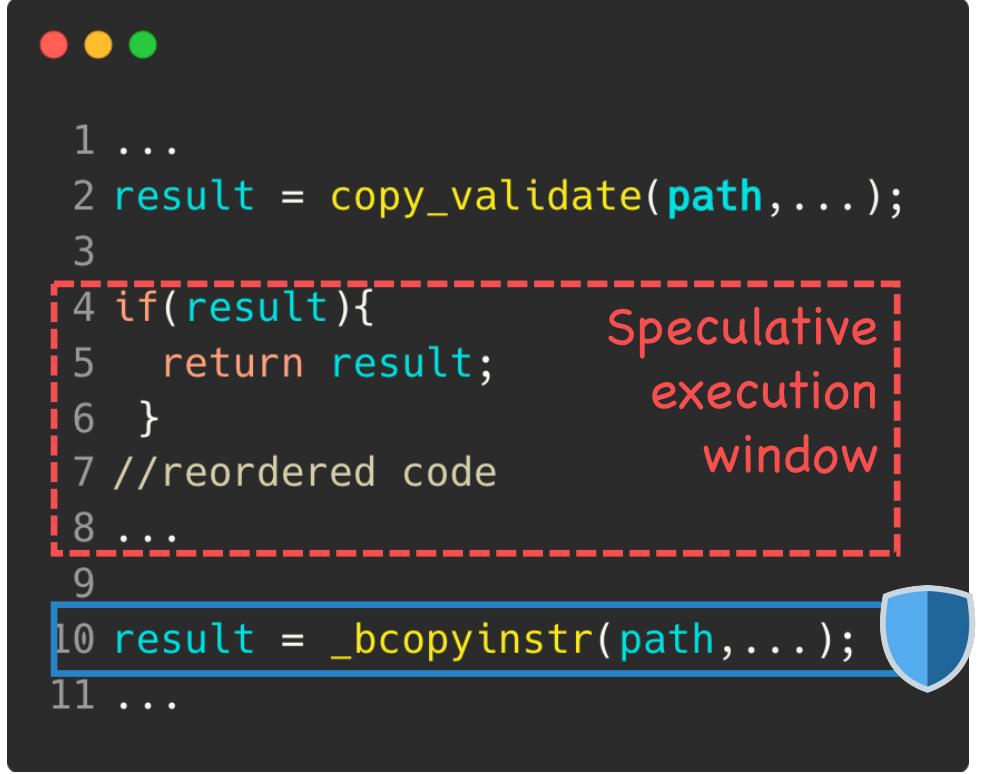
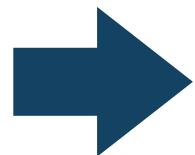
# Mitigations

- Restricting **speculative execution window** to reach target instruction
  - Reorders the code sequence to place the target instruction farther from the conditional branch



```
1 ...
2 result = copy_validate(path, ...);
3
4 if(result){
5     return result;
6 }
7
8 result = _bcopyinstr(path, ...);
9 ...
```

Speculative execution window



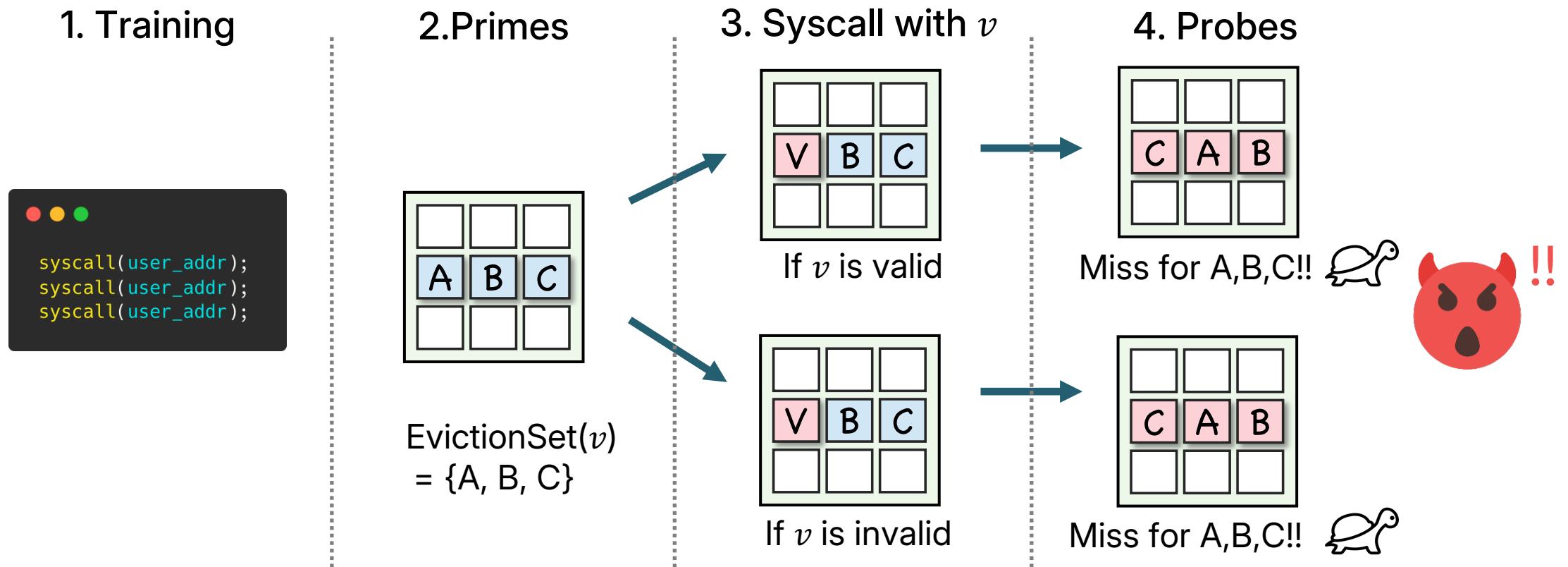
```
1 ...
2 result = copy_validate(path, ...);
3
4 if(result){
5     return result;
6 }
7 //reordered code
8 ...
9
10 result = _bcopyinstr(path, ...);
11 ...
```

Speculative execution window



# Mitigations

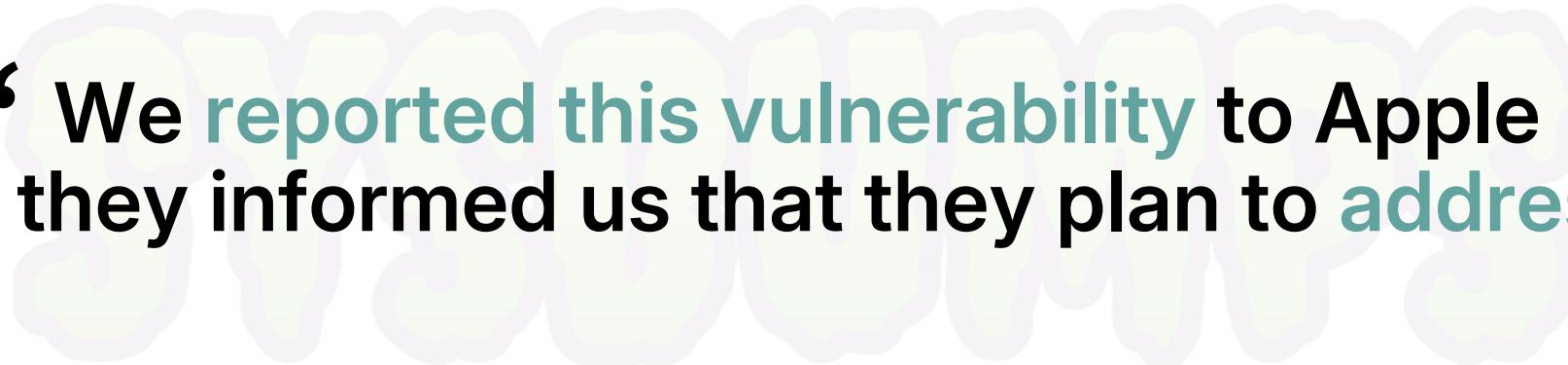
- Allocating TLB entry for **invalid address**
  - No longer distinguish the validity of kernel addresses through TLB



# Mitigations

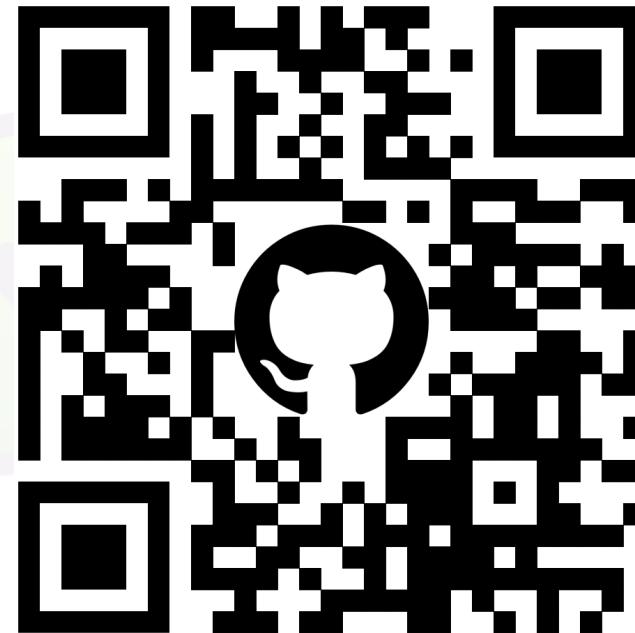
- Partitioning dTLB between user and kernel space
  - Eliminates contention on the dTLB





**“ We reported this vulnerability to Apple ”**  
**and they informed us that they plan to address it.**

# Proof of Concept



[github.com/koreacsI/SysBumps.git](https://github.com/koreacsI/SysBumps.git)

# Takeaway

- KASLR on macOS for Apple Silicon is **vulnerable** to microarchitectural side-channel attack
- Combined microarchitectural side-channel techniques can bypass even advanced mitigations
- This vulnerability can be **mitigated** through hardware or software improvements

# Q&A

Thank you :-)



Contact : [hr\\_jang@korea.ac.kr](mailto:hr_jang@korea.ac.kr)