
# 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



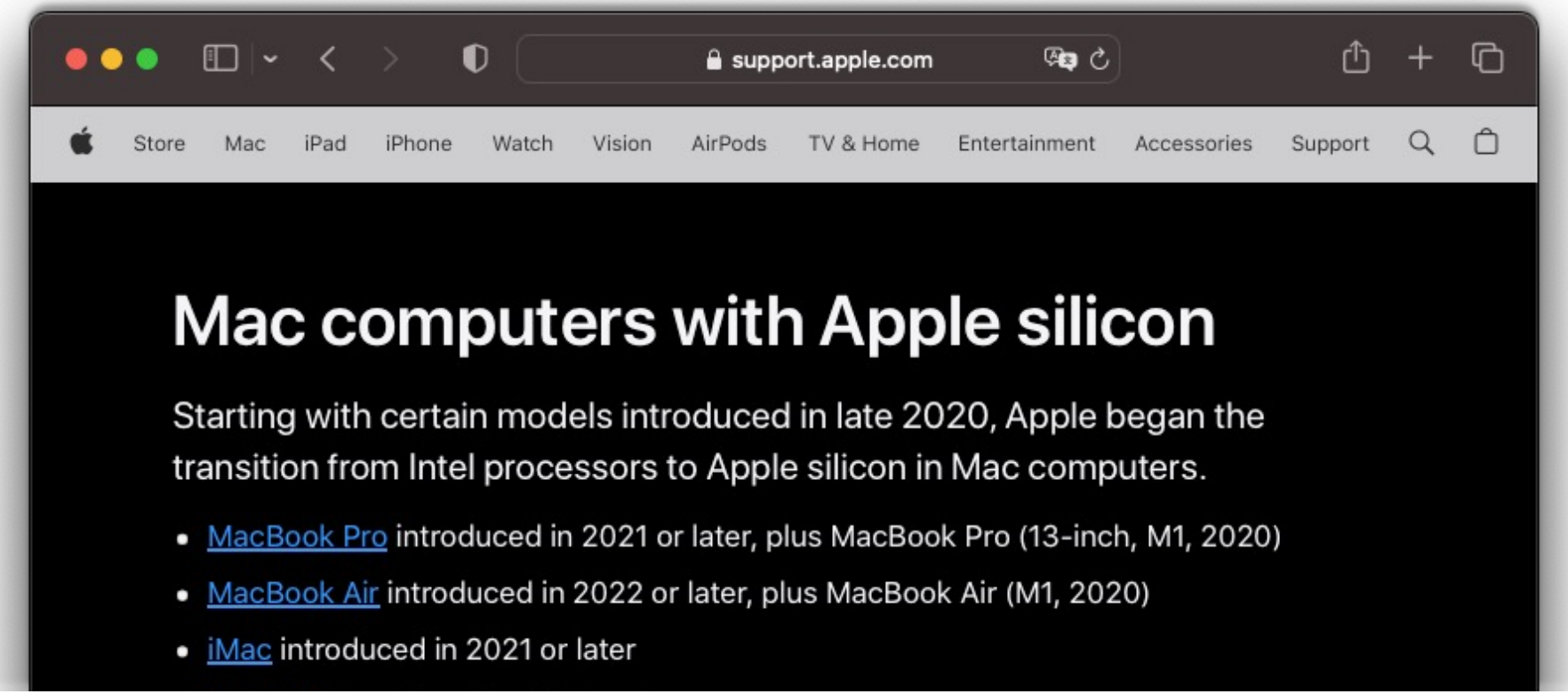
## Youngjoo Shin

Professor
@Korea University

✉️ Email : syoungjoo@korea.ac.kr

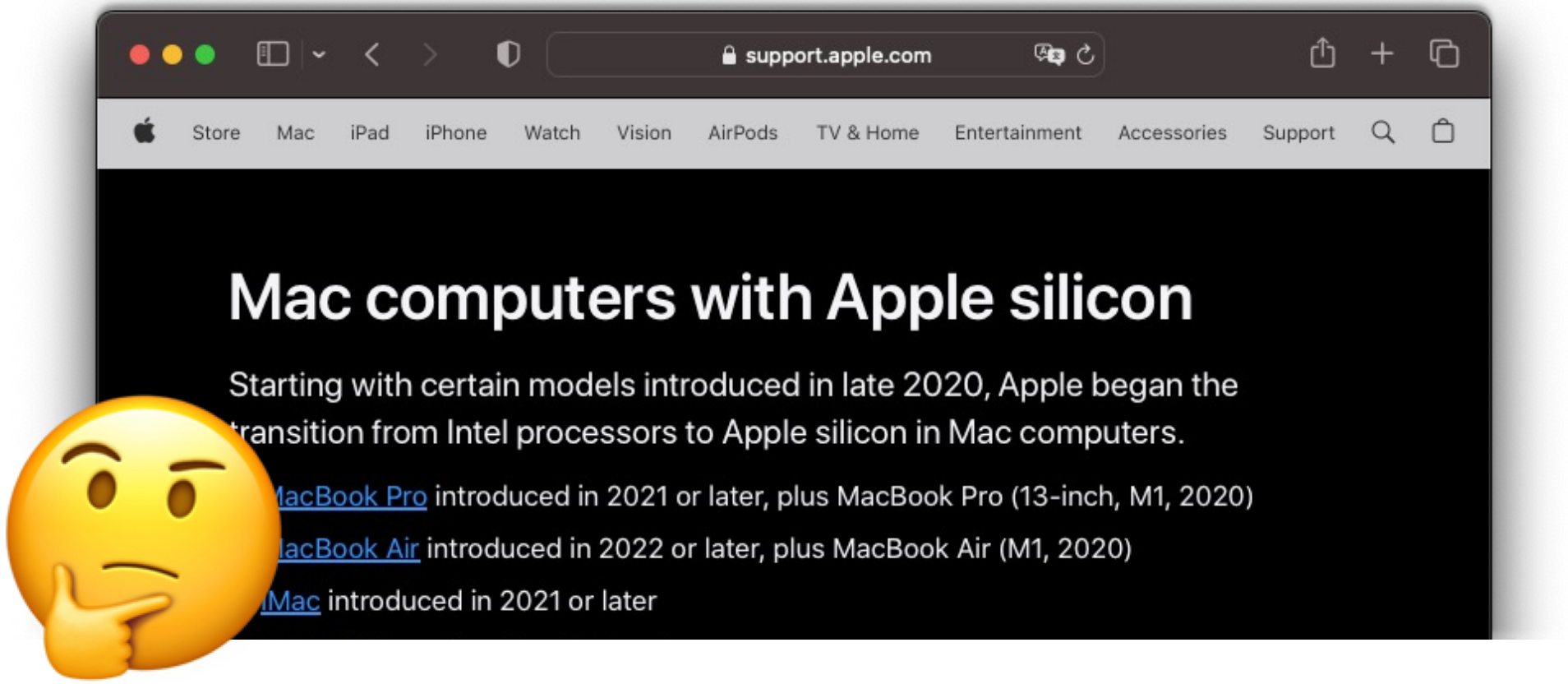
# Apple Silicon

Apple’s proprietary arm-based processor



# Apple Silicon

Apple’s proprietary arm-based processor




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

# What is SysBumps Attack?

KASLR breaking attack on macOS for Apple silicon

# 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

Speculative Execution

+

TLB-based Side-channel

=

SYSBUMPS

# Outline

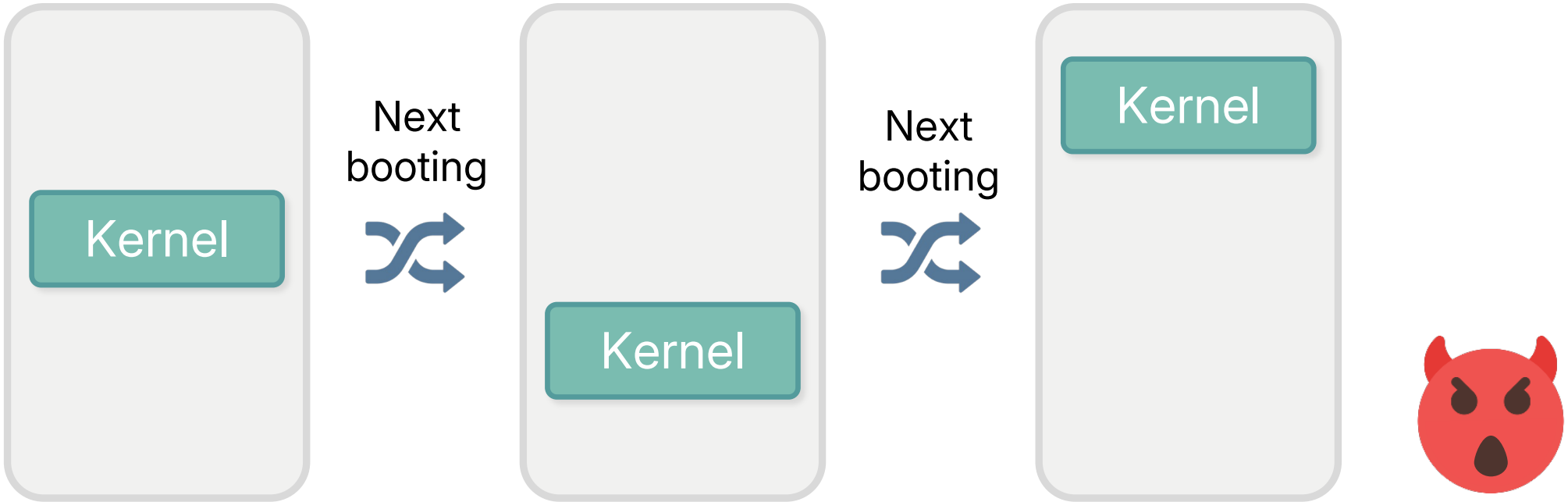
- Background
- Existing Microarchitectural Attack on KASLR
- Challenges
- Our Approaches
- SysBumps Attack
- Mitigations
- Takeaway

# Microarchitectural Side-Channel Attack

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

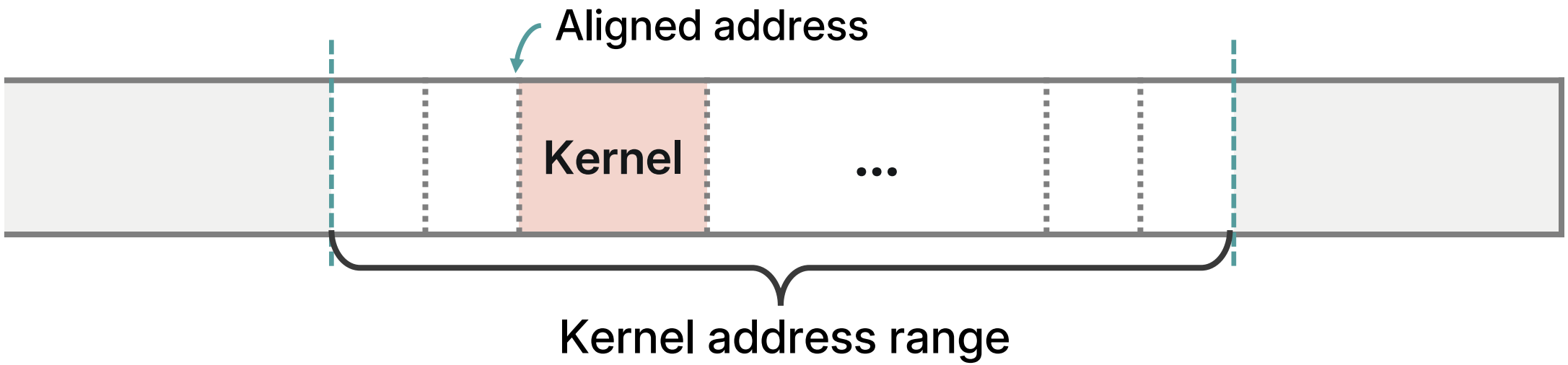
# Kernel Address Space Layout Randomization

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



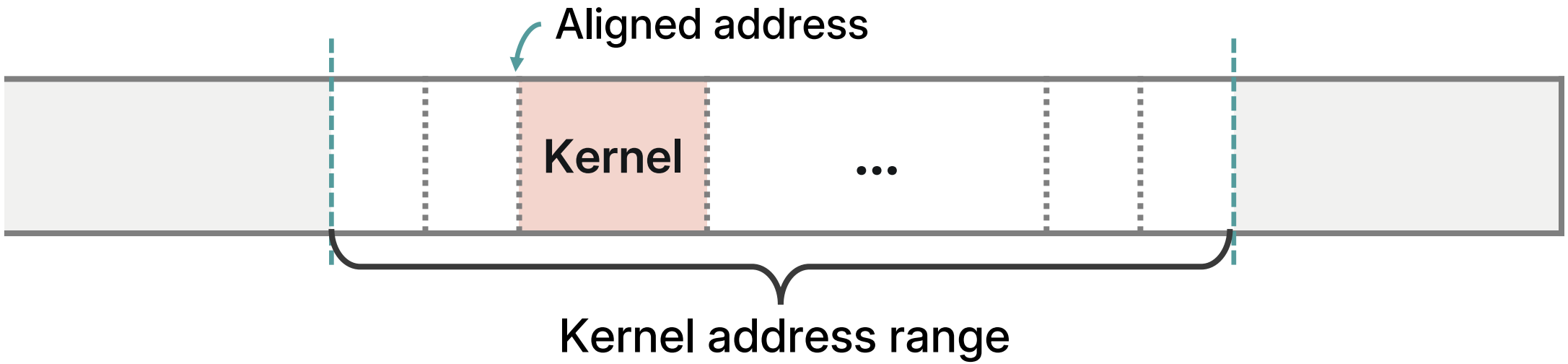
# Kernel Address Space Layout Randomization

- 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)

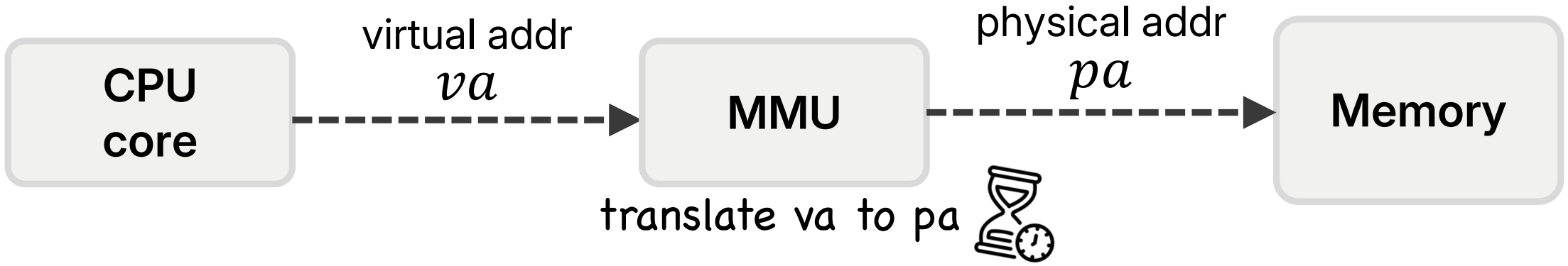
- Store translated virtual-to-physical address mappings

# Translation Lookaside Buffer (TLB)

- Store translated virtual-to-physical address mappings

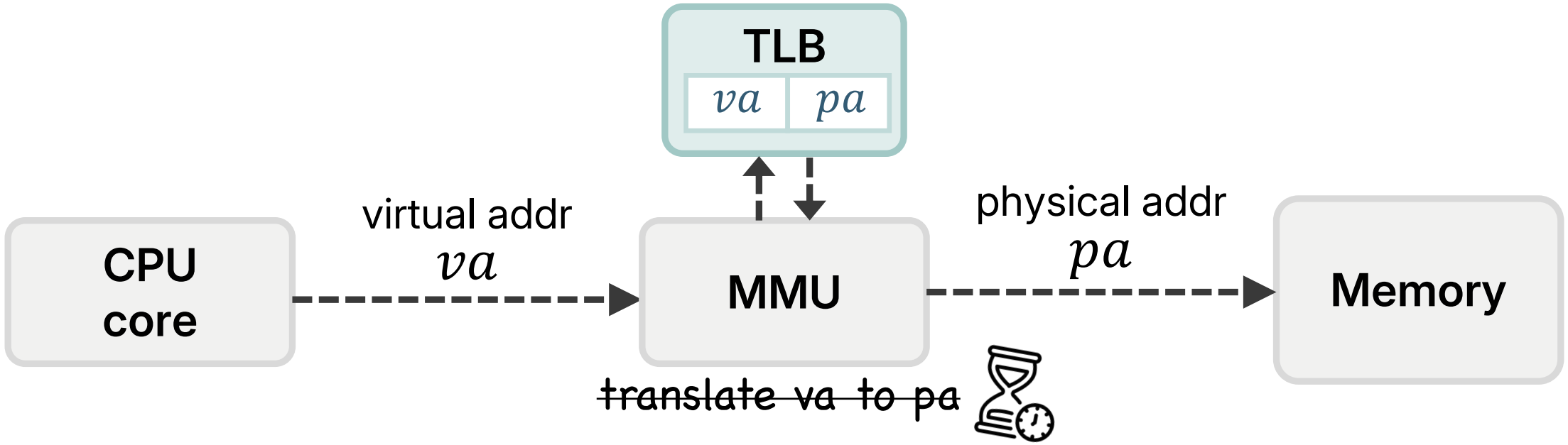
# Translation Lookaside Buffer (TLB)

- Store translated virtual-to-physical address mappings



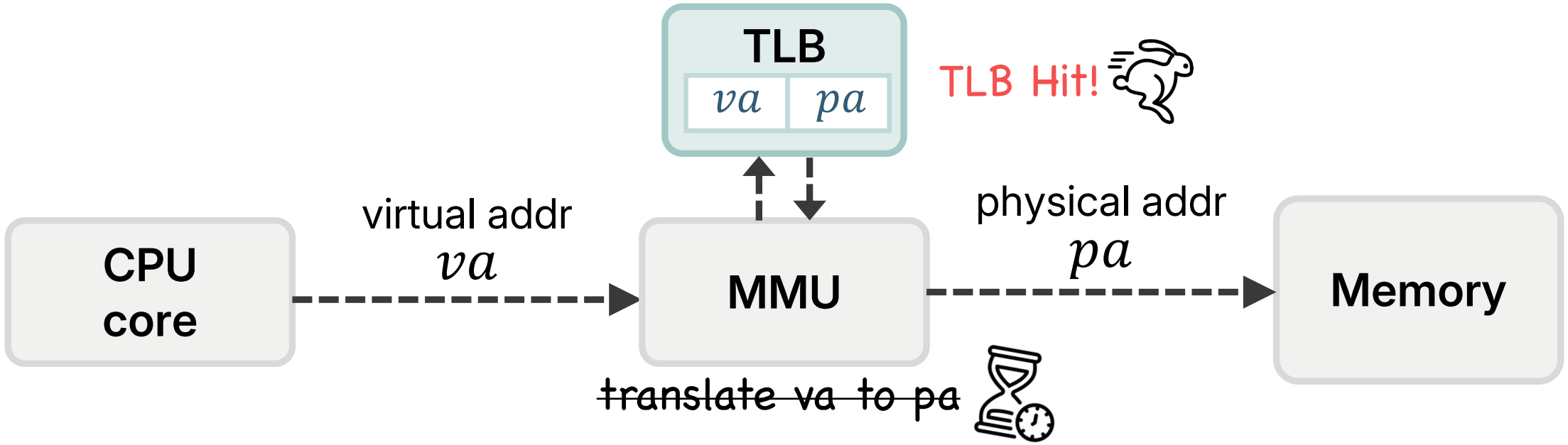
# Translation Lookaside Buffer (TLB)

- Store translated virtual-to-physical address mappings



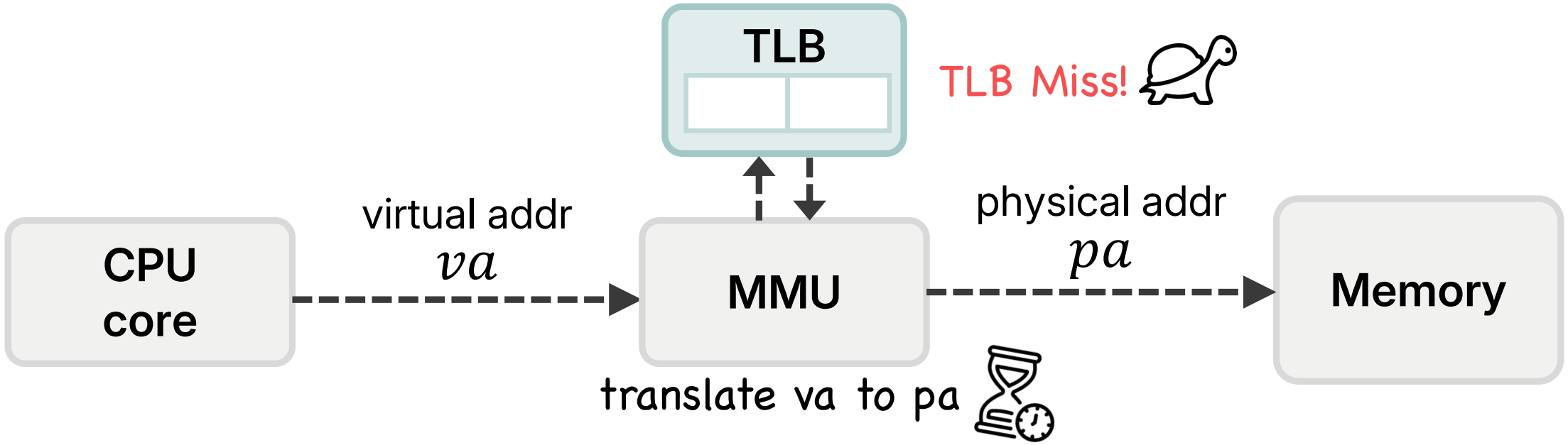
# Translation Lookaside Buffer (TLB)

- Store translated virtual-to-physical address mappings



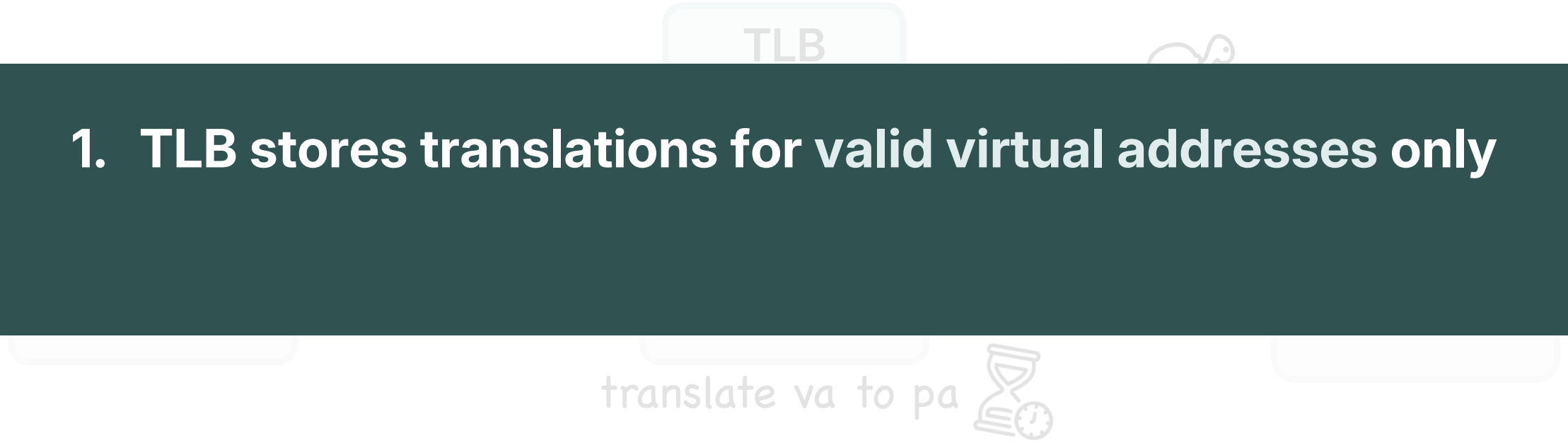
# Translation Lookaside Buffer (TLB)

- Store translated virtual-to-physical address mappings



# Translation Lookaside Buffer (TLB)

- Store translated virtual-to-physical address mappings



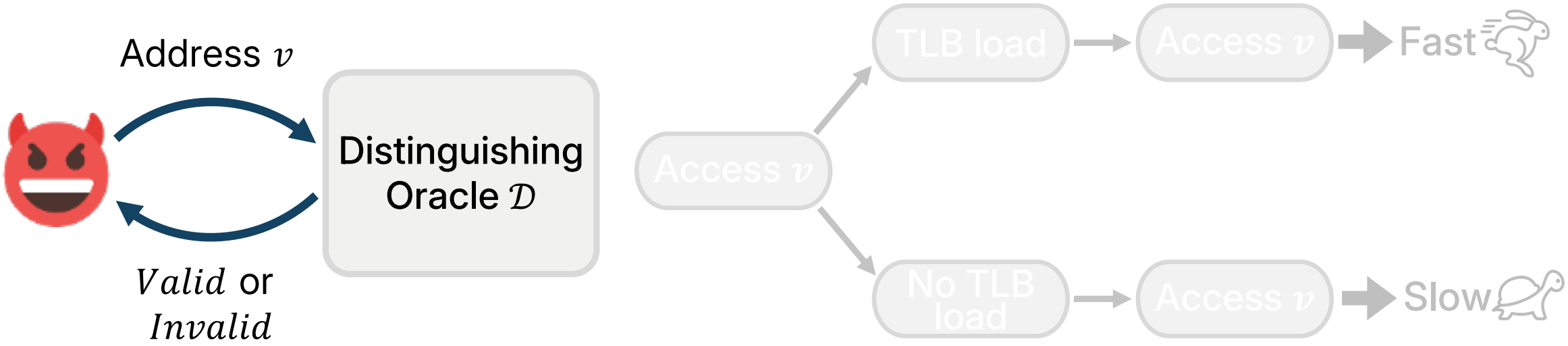

1. **TLB stores translations for valid virtual addresses only**

# Translation Lookaside Buffer (TLB)

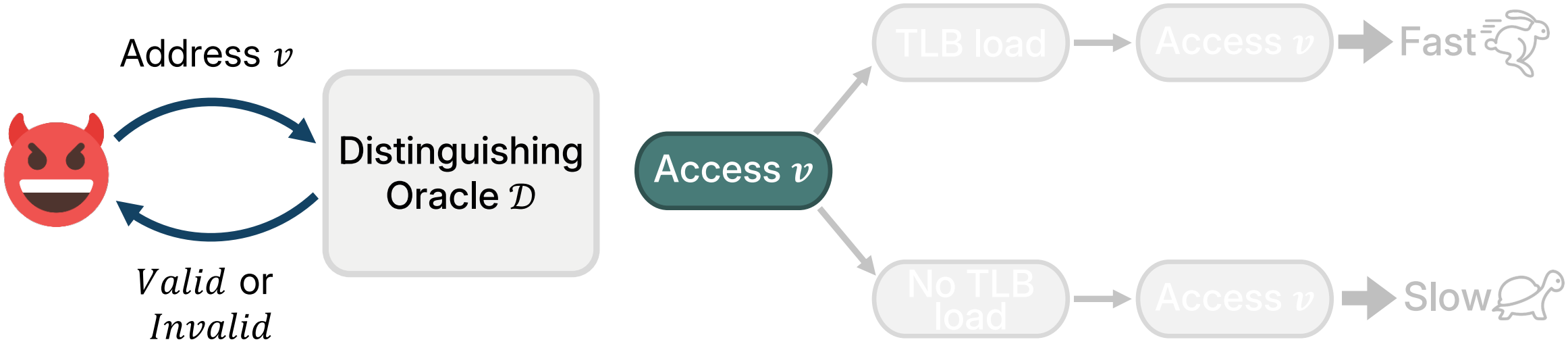
- Store translated virtual-to-physical address mappings


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

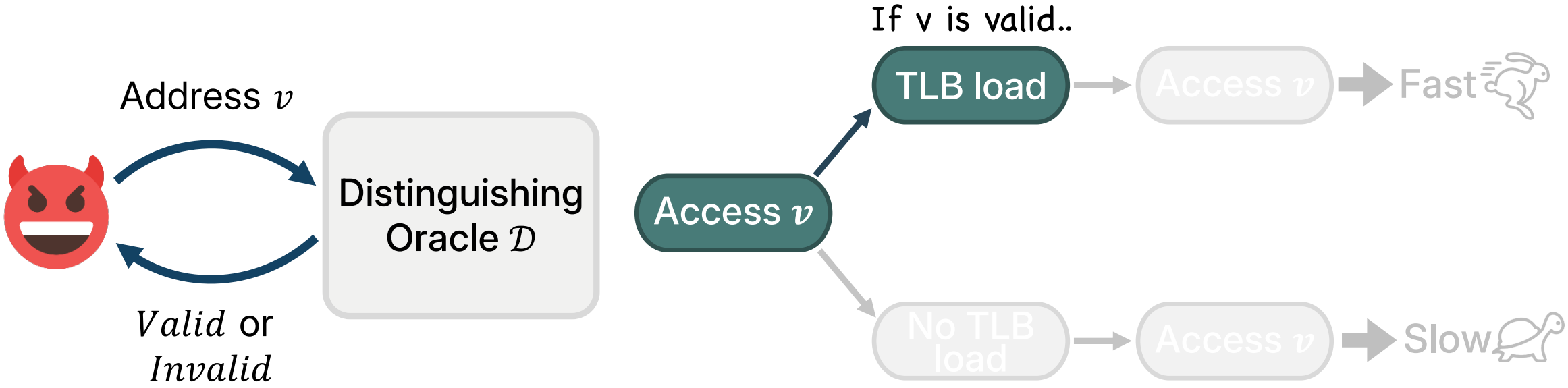
# Existing Microarchitectural Attack on KASLR



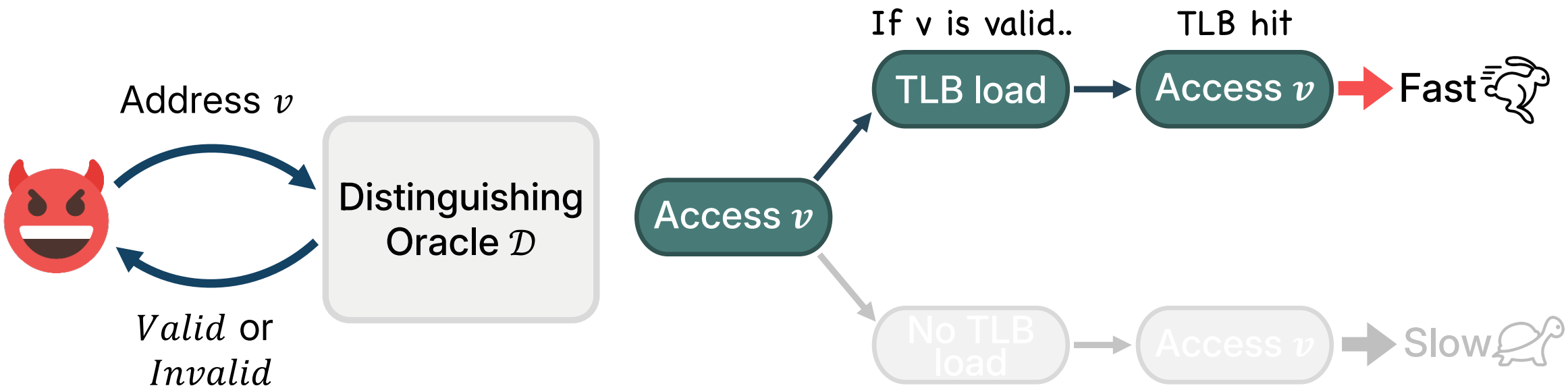
# Existing Microarchitectural Attack on KASLR



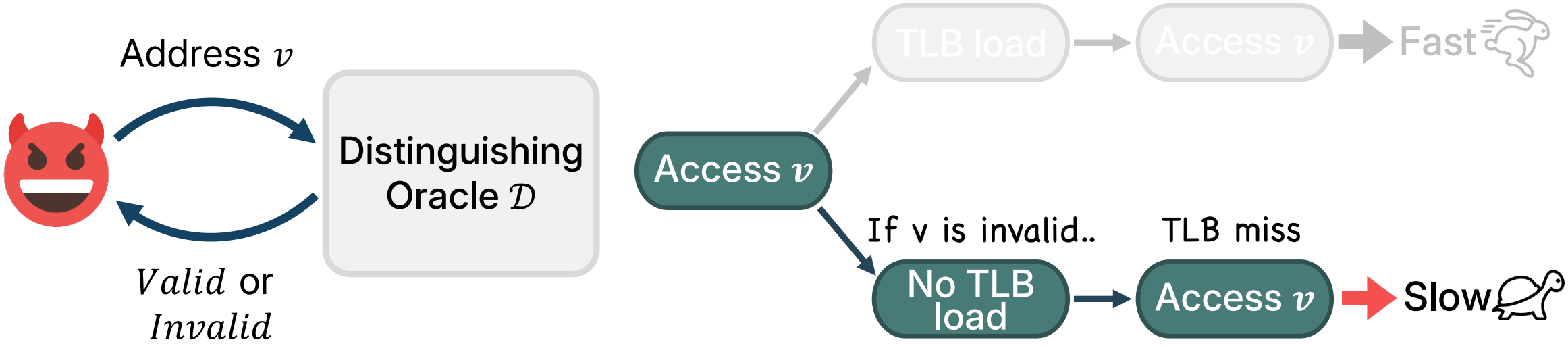
# Existing Microarchitectural Attack on KASLR



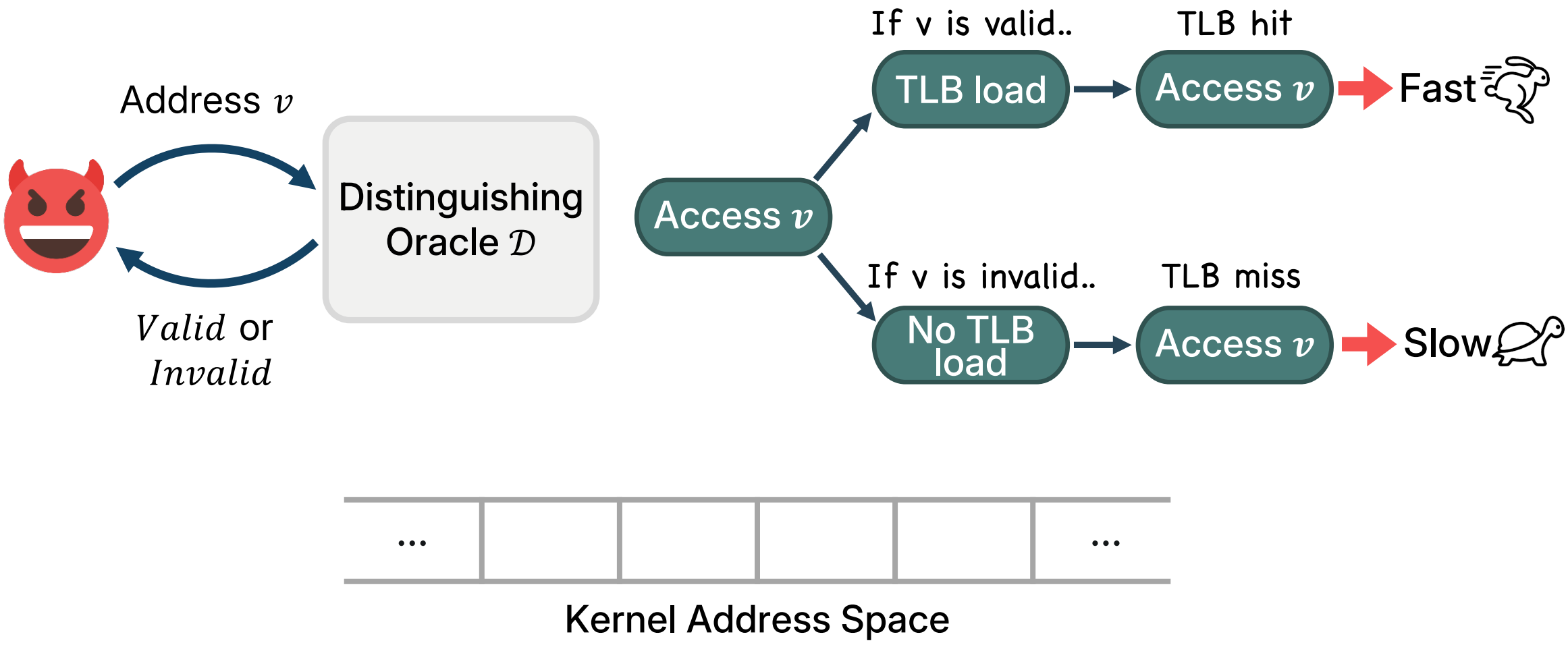
# Existing Microarchitectural Attack on KASLR



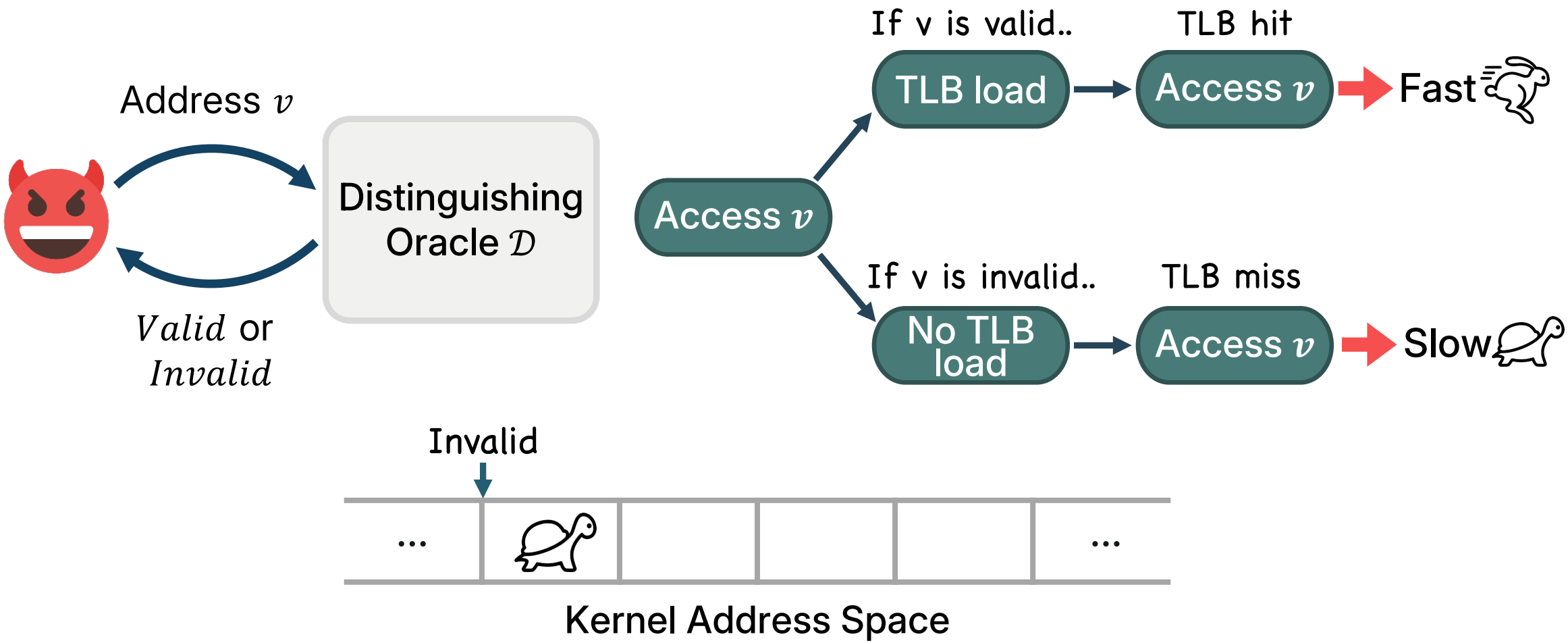
# Existing Microarchitectural Attack on KASLR



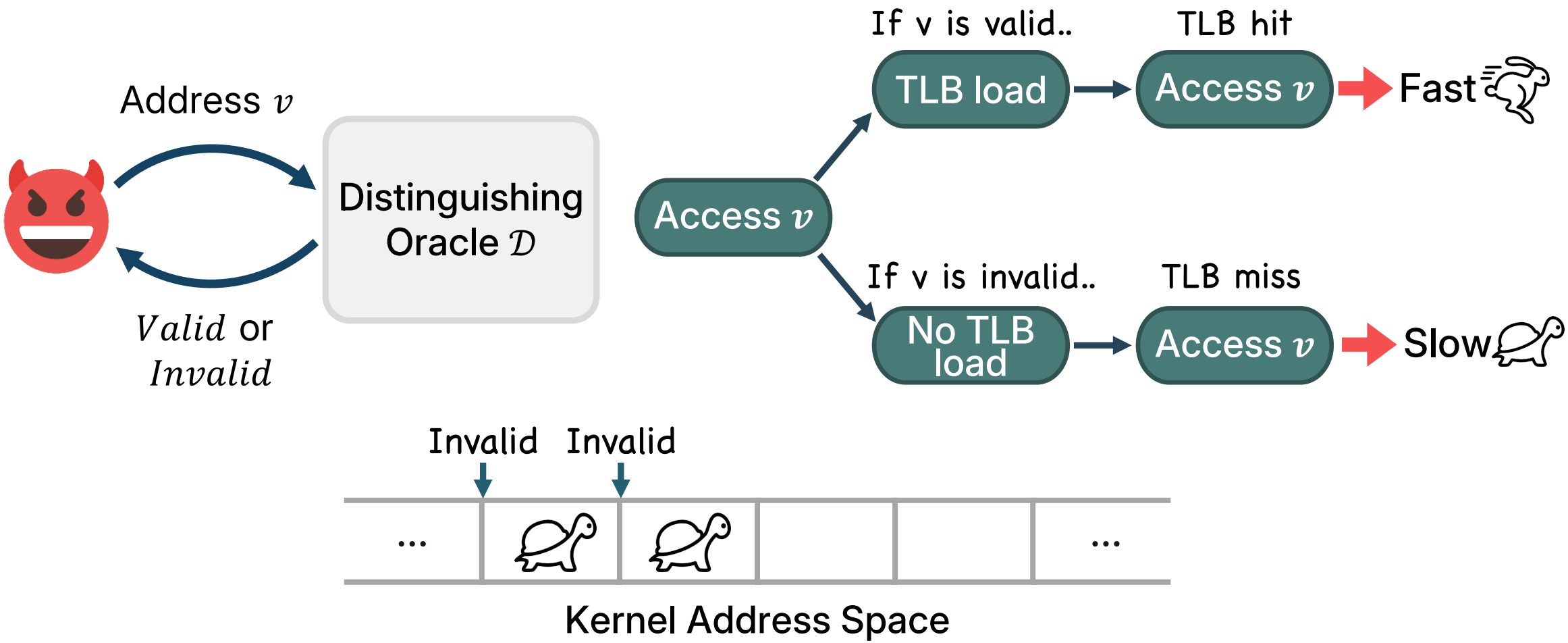
# Existing Microarchitectural Attack on KASLR



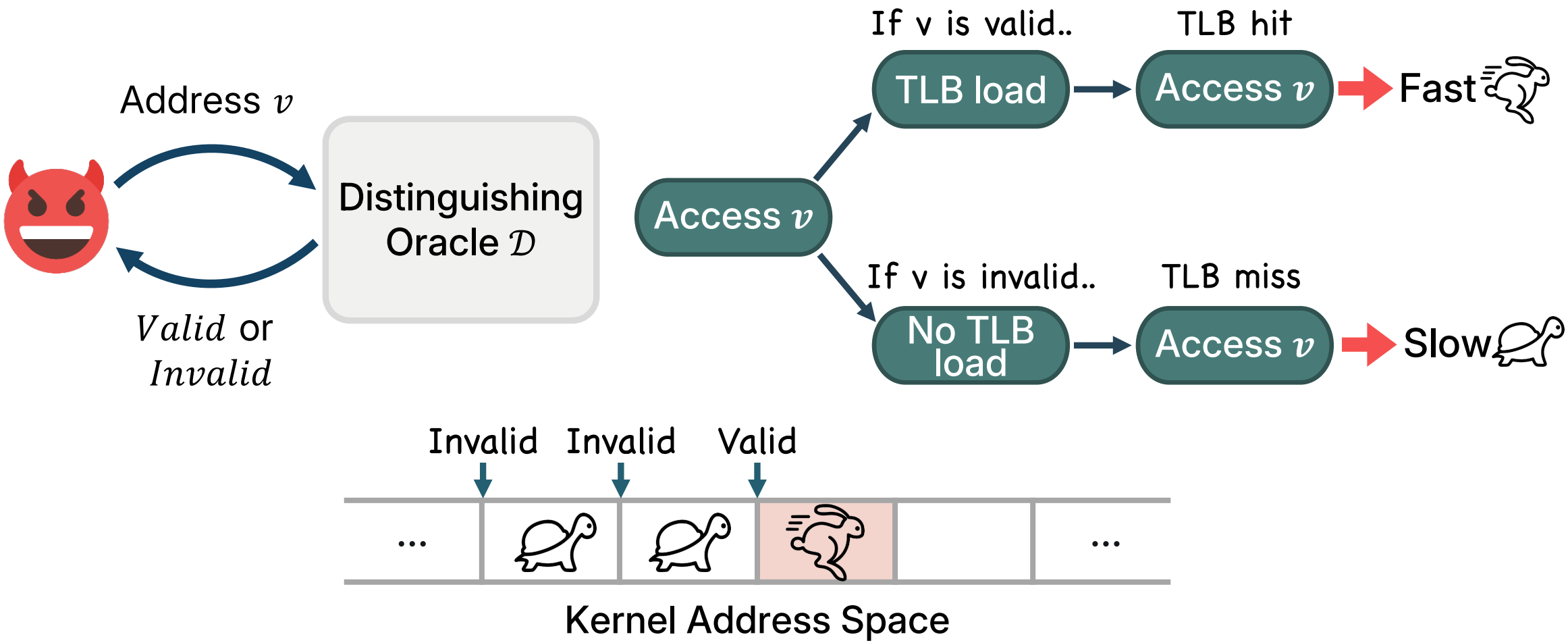
# Existing Microarchitectural Attack on KASLR



# Existing Microarchitectural Attack on KASLR



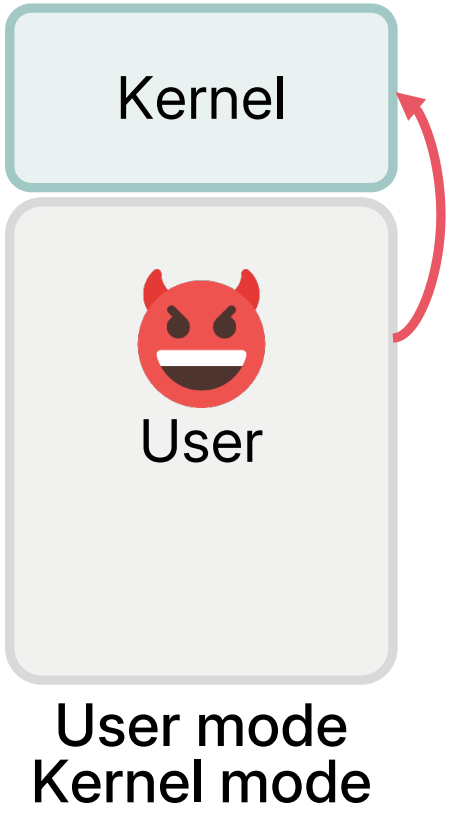
# Existing Microarchitectural Attack on KASLR



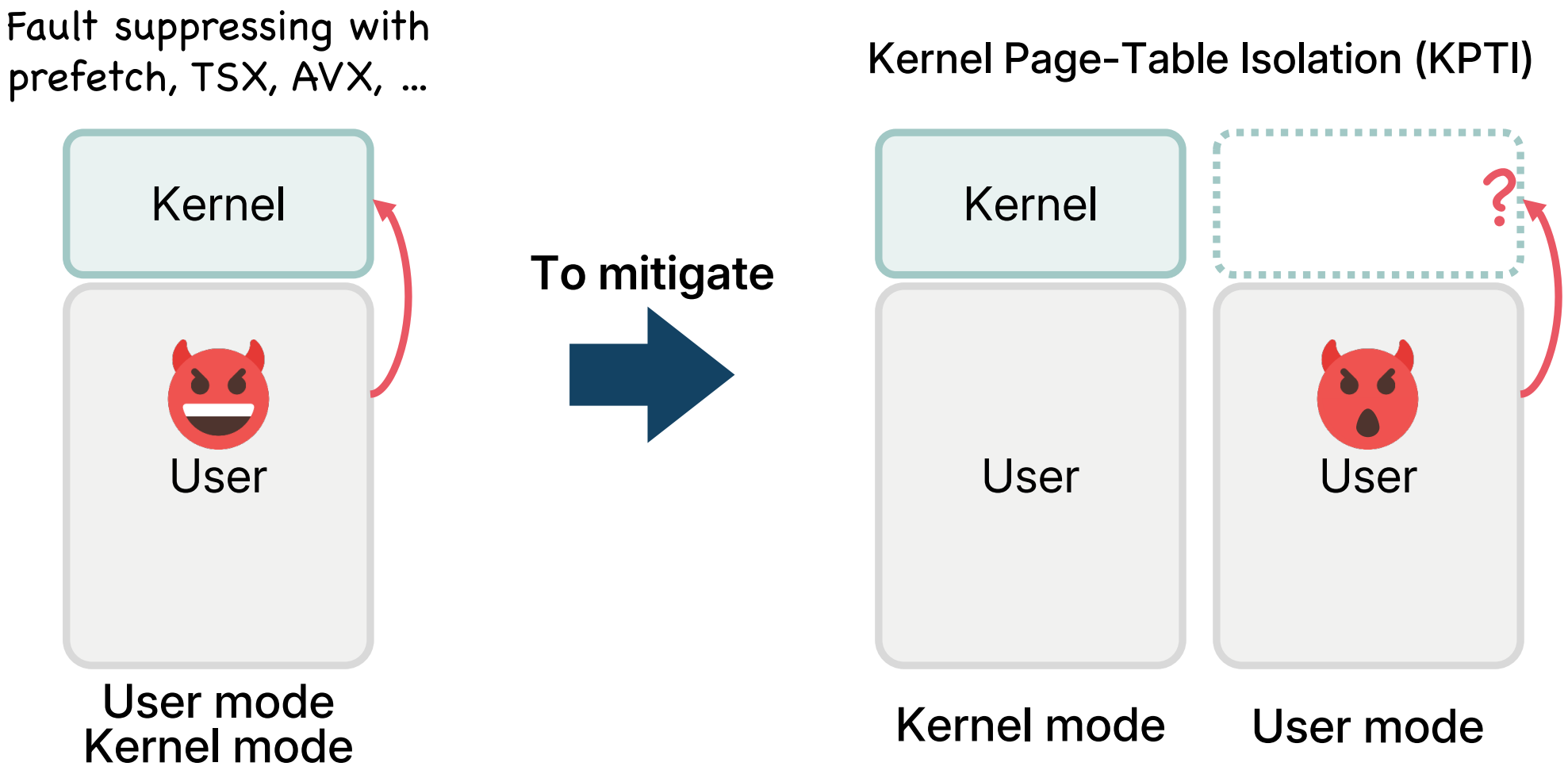
# How Can Unprivileged Attacker Access Kernel Addresses?

# How Can Unprivileged Attacker Access Kernel Addresses?

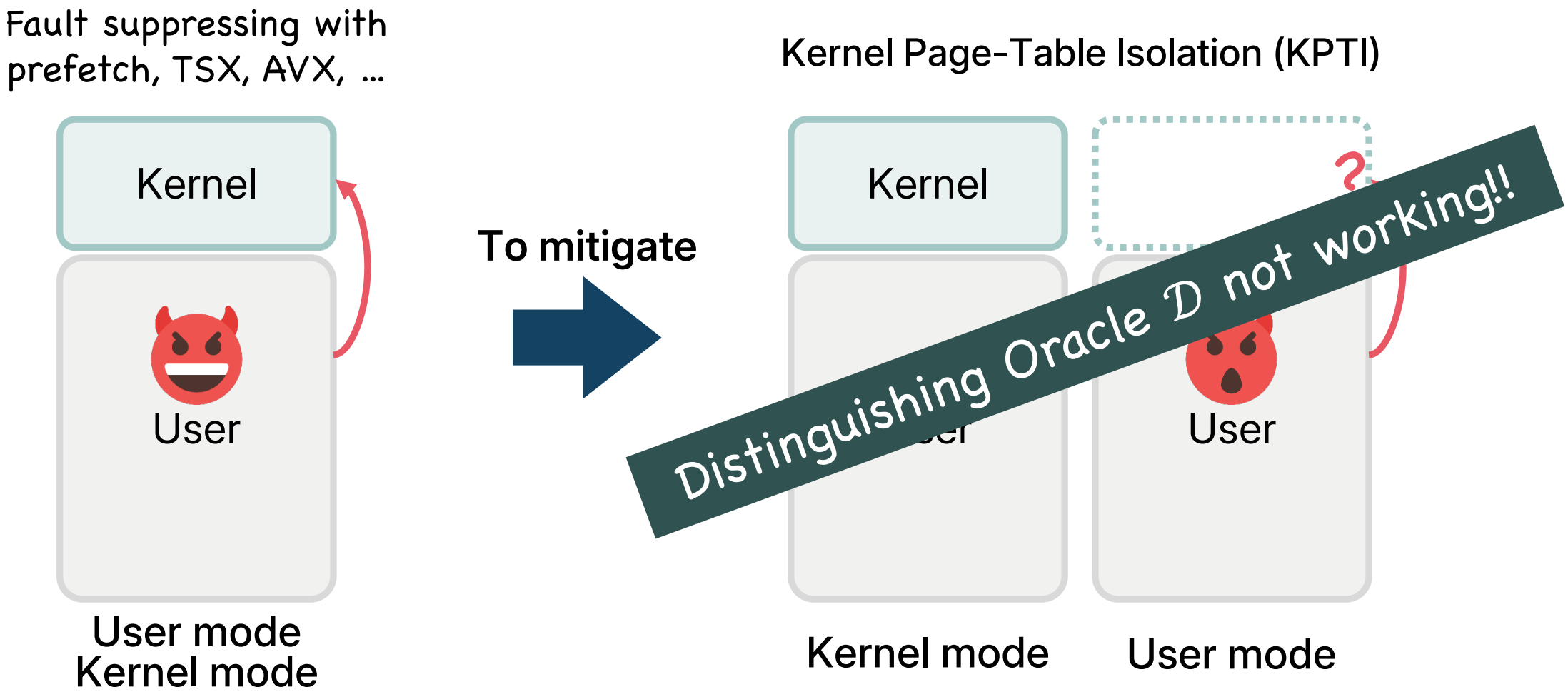
Fault suppressing with prefetch, TSX, AVX, ...



# How Can Unprivileged Attacker Access Kernel Addresses?



# How Can Unprivileged Attacker Access Kernel Addresses?

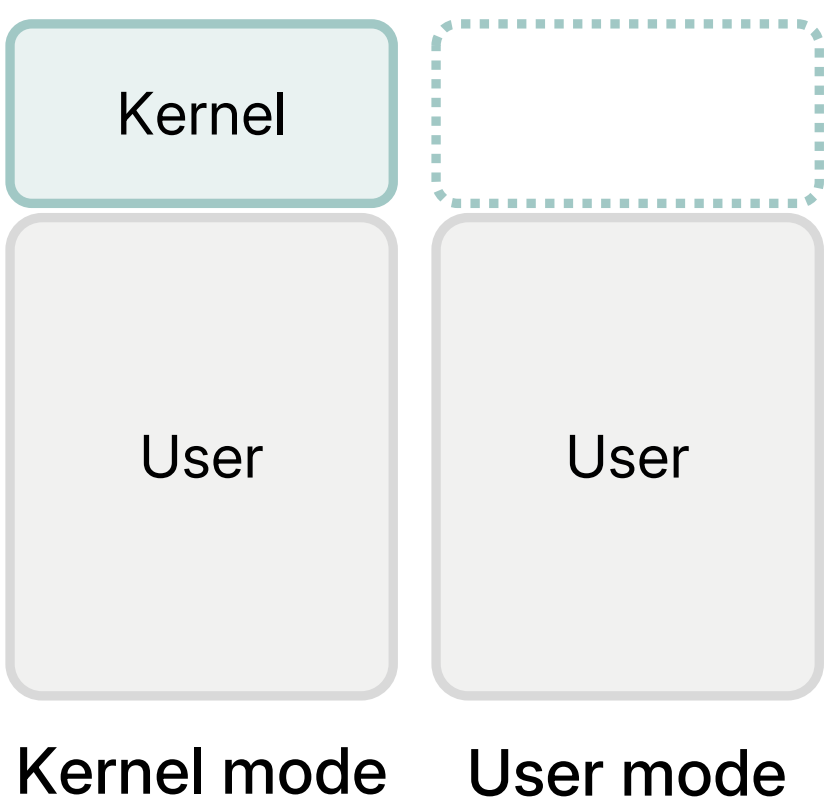


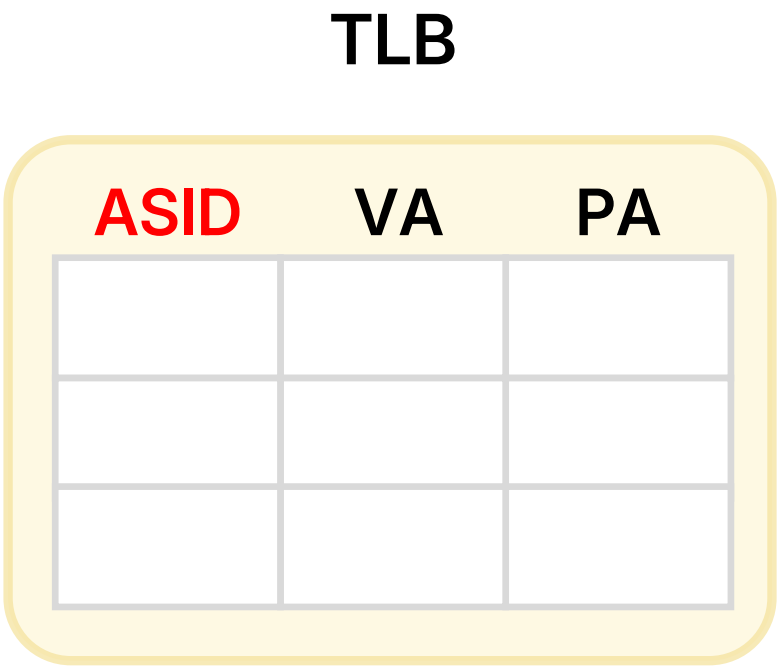
# Challenges to Breaking KASLR

# Challenges to Breaking KASLR

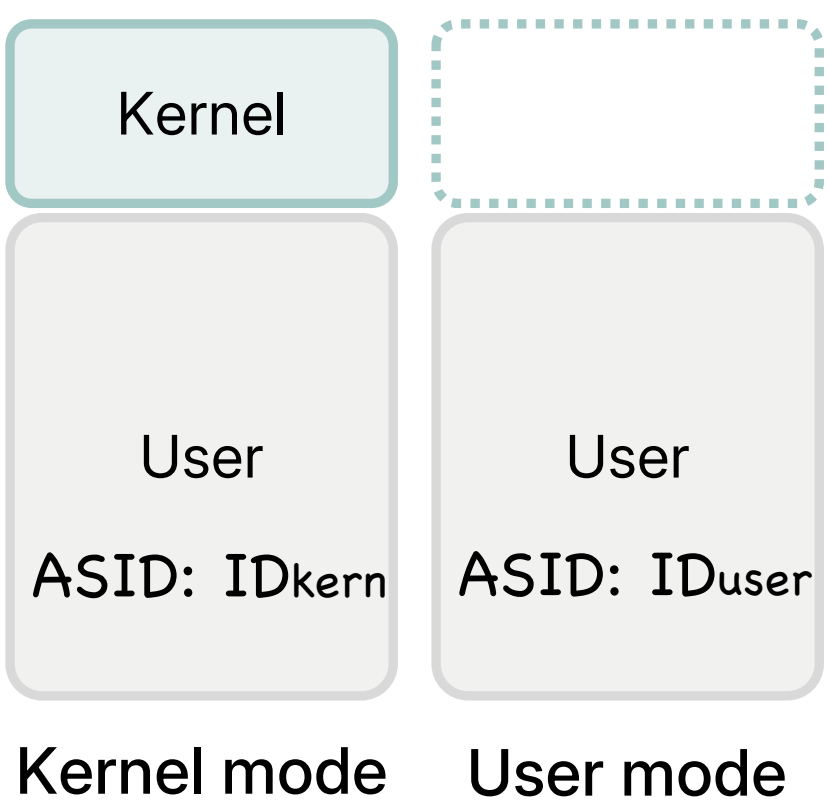
- C1. How to cache a kernel address in the TLB?
  - Kernel page table isolation (KPTI)

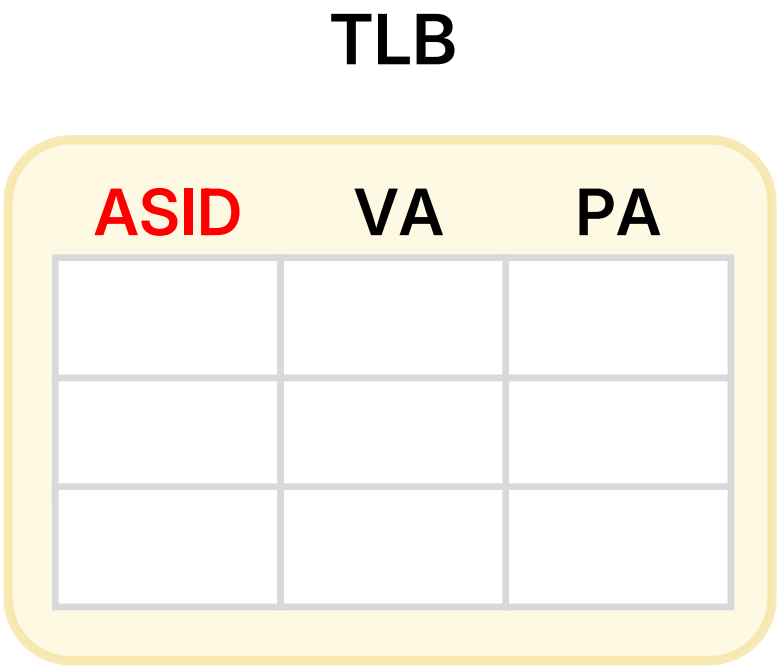
# Address Space Identifier



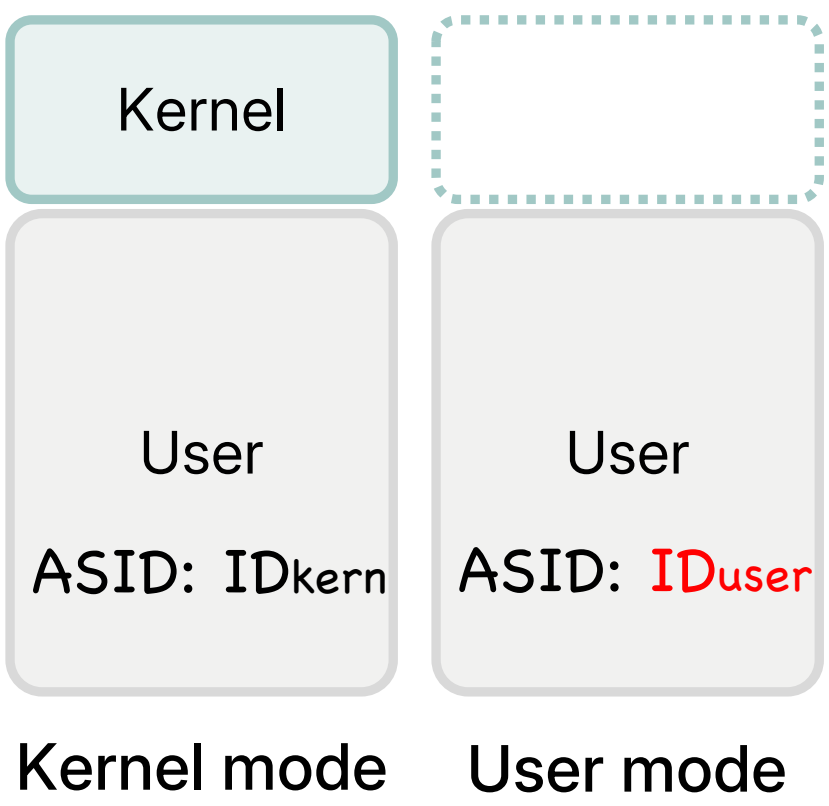



# Address Space Identifier



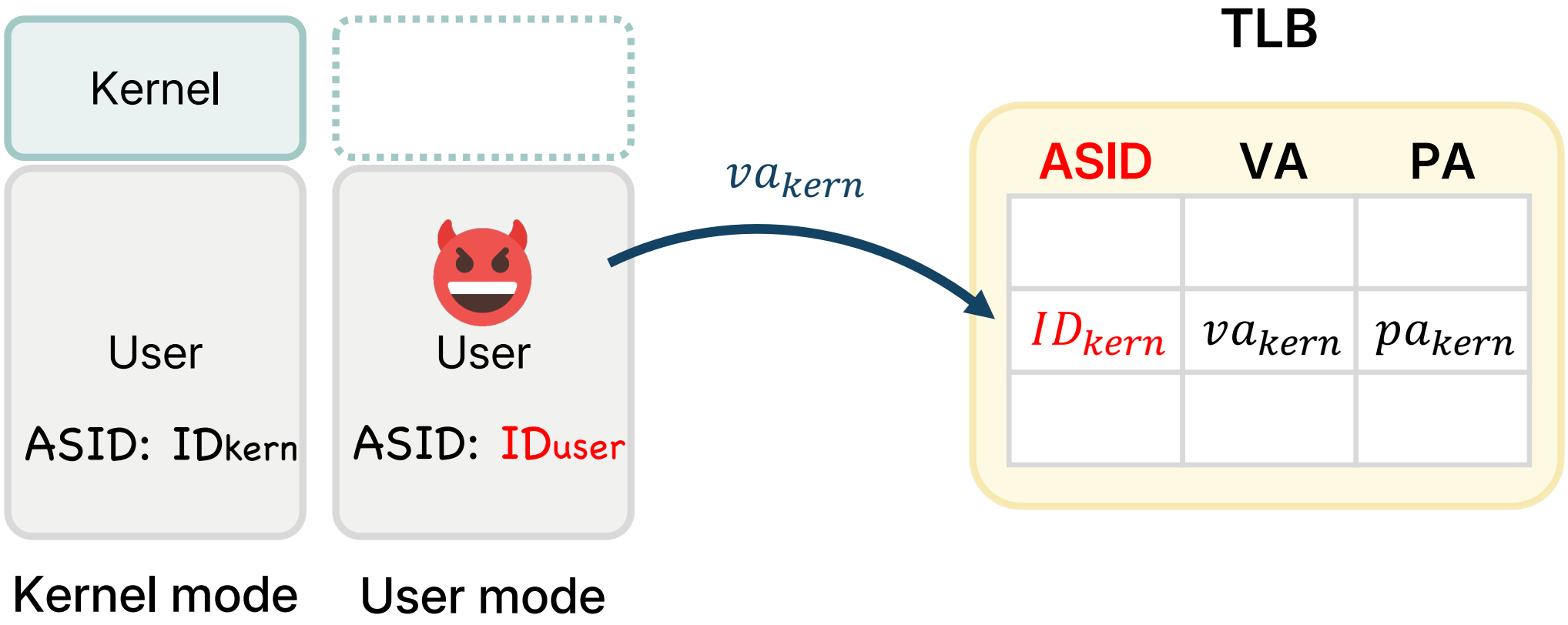
# Address Space Identifier



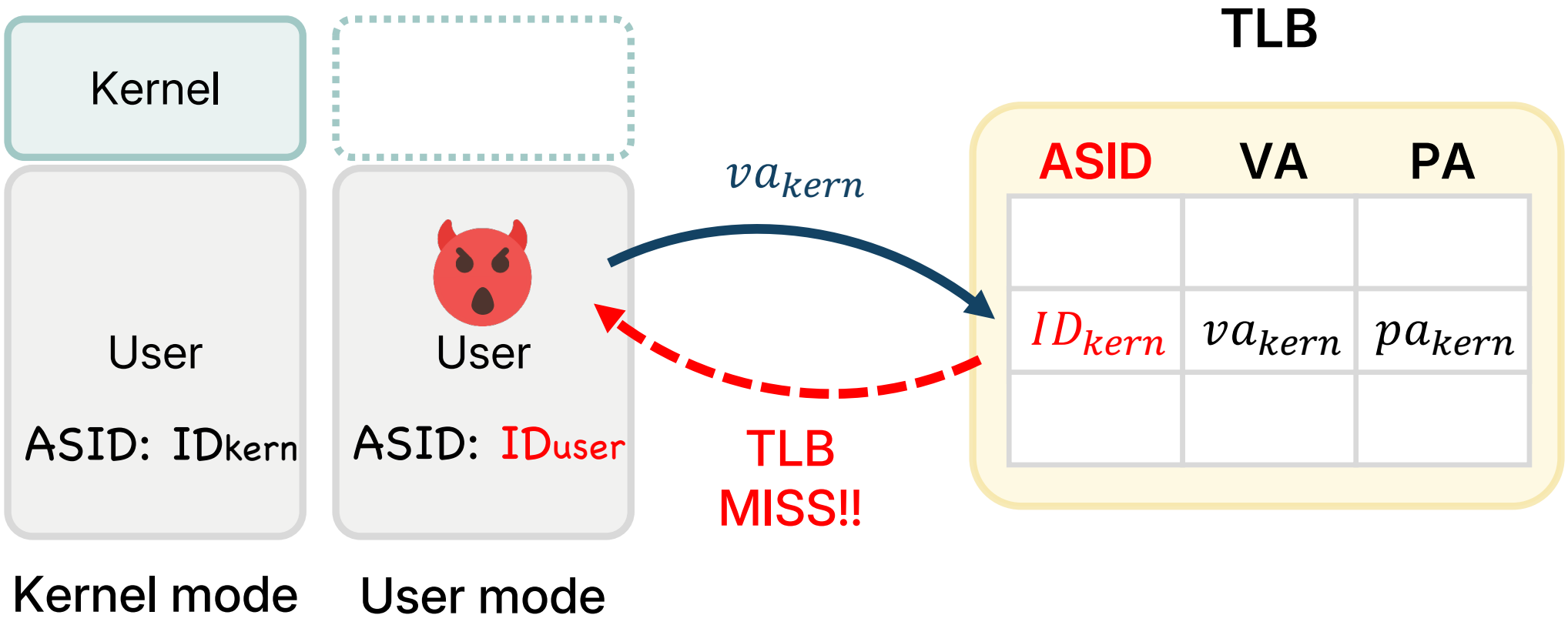

TLB

| ASID | VA | PA |
|---|---|---|
| | | |
| $ID_{kern}$ | $va_{kern}$ | $pa_{kern}$ |
| | | |

# Address Space Identifier



# Address Space Identifier



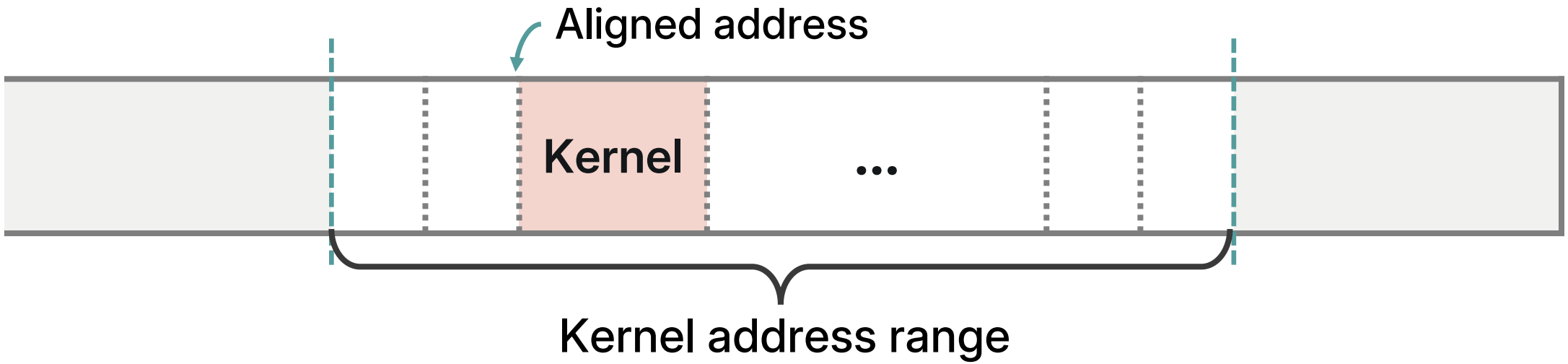
# Challenges to Breaking KASLR

- 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

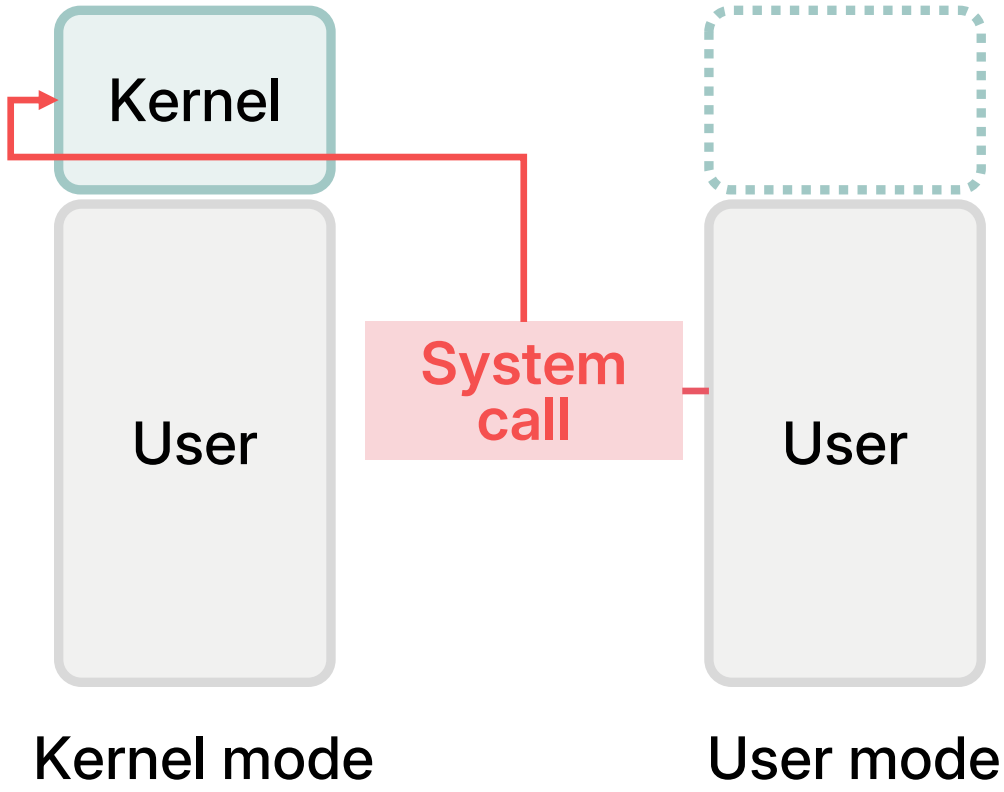
# Challenges to Breaking KASLR

- 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



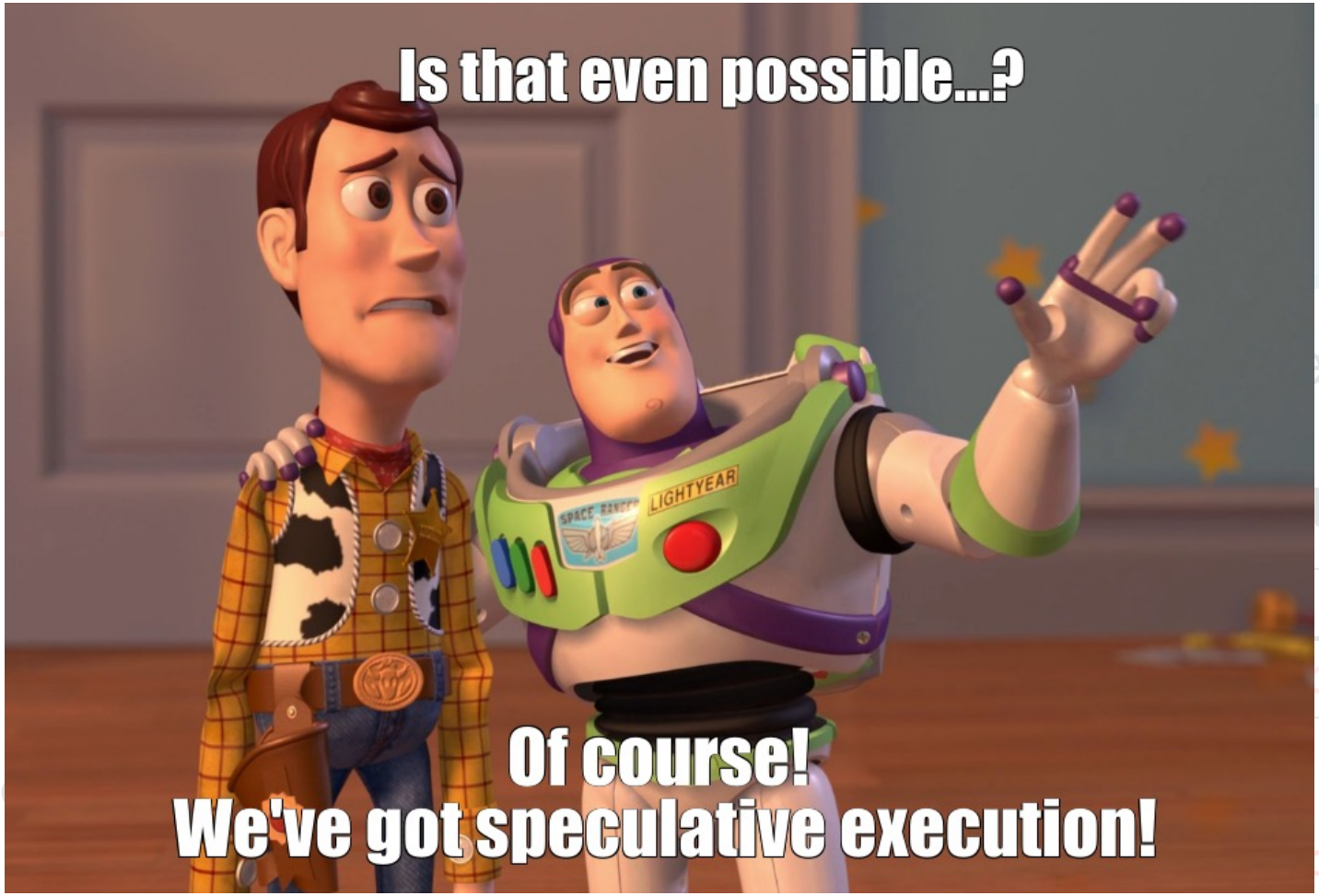

System call

Operating in kernel mode
&
Passing user input to the kernel

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

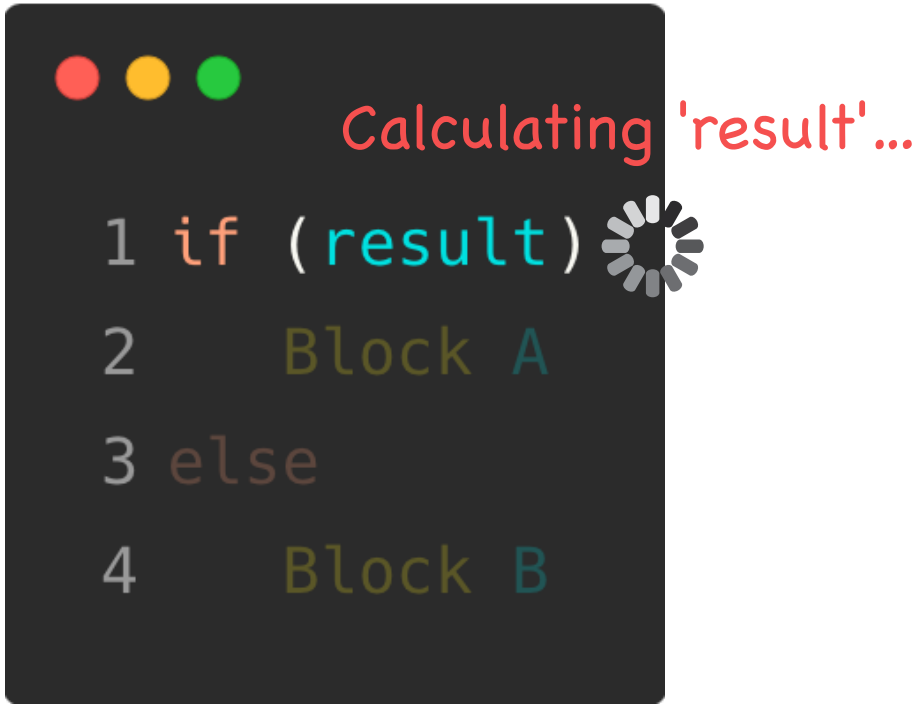
(+) Most system calls do not cause argument-induced faults

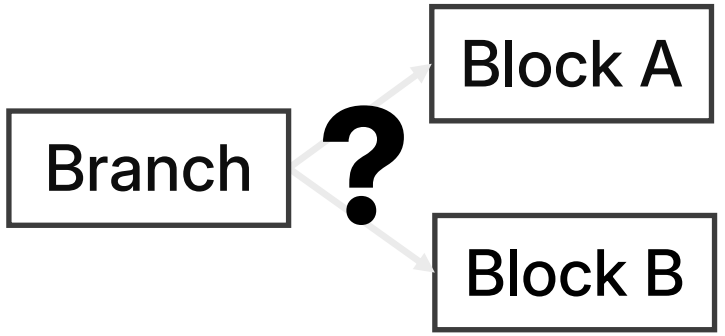
# Accessing the Inaccessible : Kernel Space under KPTI



# Speculative Execution

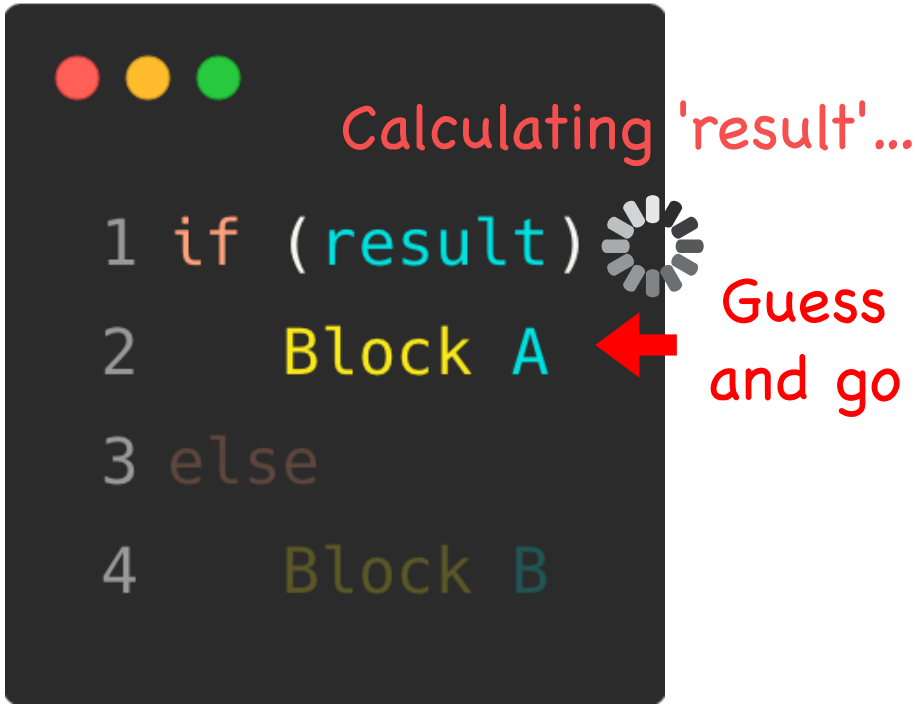
Predict branch direction based on its history and execute in advance

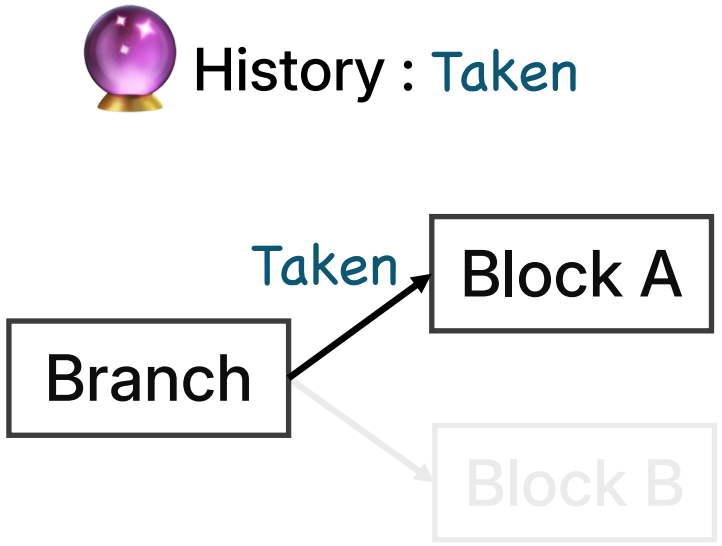


# Speculative Execution

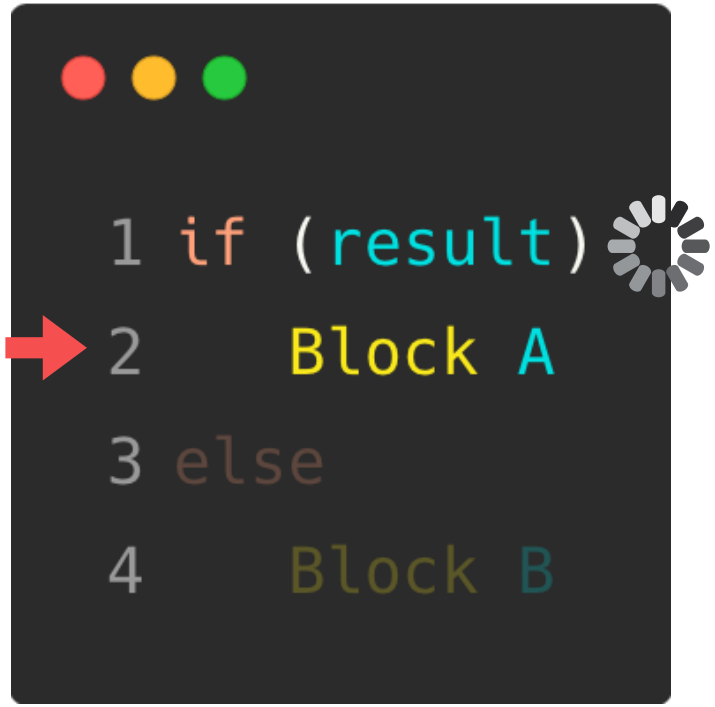
Predict branch direction based on its history and execute in advance



# Speculative Execution

Predict branch direction based on its history and execute in advance




Actual path

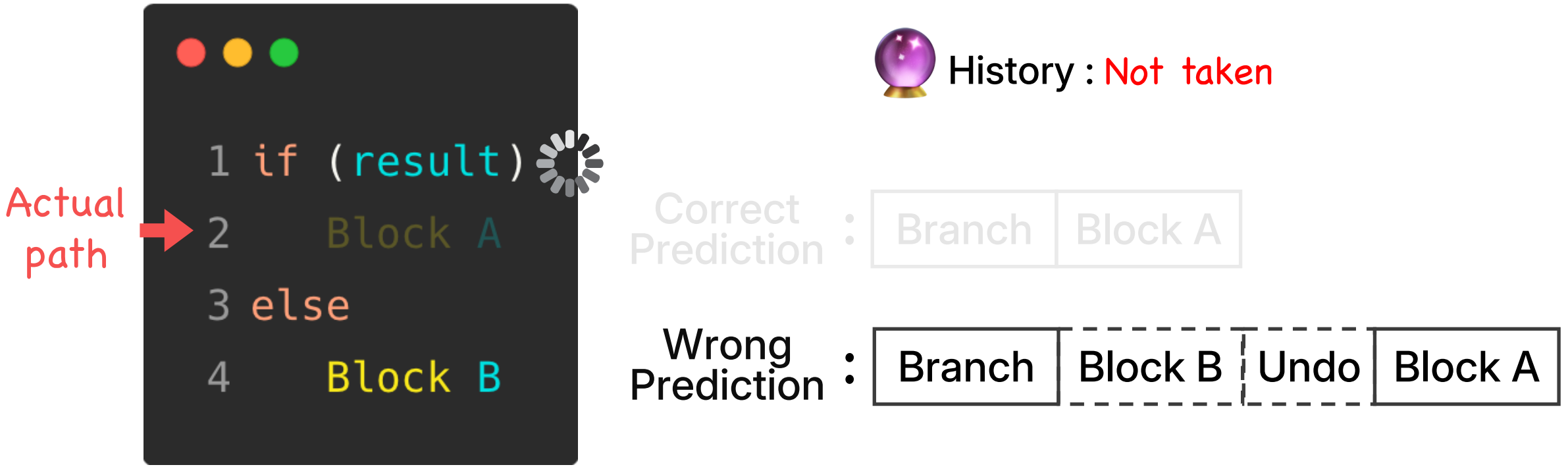
History : Taken

Correct Prediction : | Branch | Block A |

Wrong Prediction : | Branch | Block B | Undo | Block A |

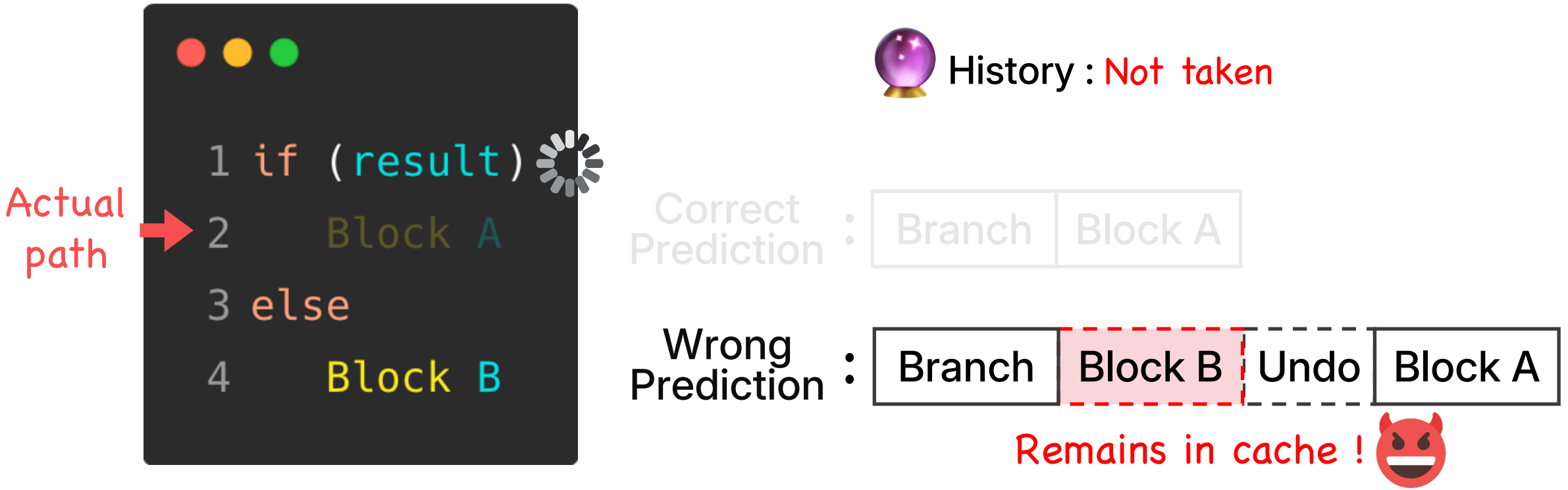
# Speculative Execution

Predict branch direction based on its history and execute in advance



# Speculative Execution

Predict branch direction based on its history and execute in advance



# 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)
5   return result;
6 }
7
8 result = _bcopyinstr(path,...);
9 ...
```

Not taken

Guess and go

If 'path' is kernel address ...

1. Validate 'path'
2. Check the result
3. Access 'path'

# 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

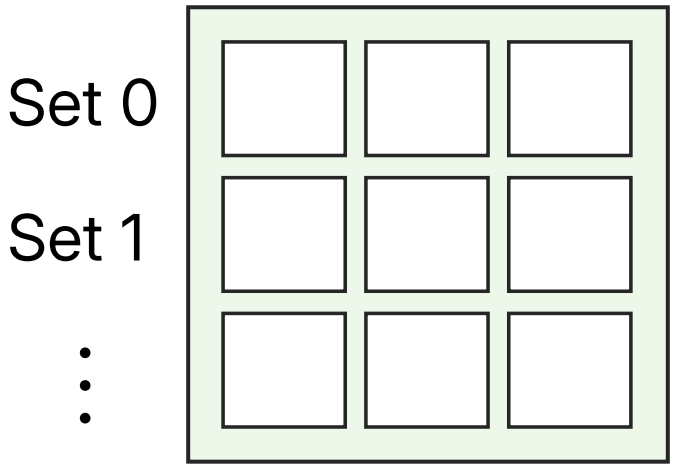
- C1. How to cache a kernel address in the TLB?
  - Kernel page table isolation (KPTI)

- C2. How to check if a kernel address is cached in the TLB?
  - Address space identifier (ASID)

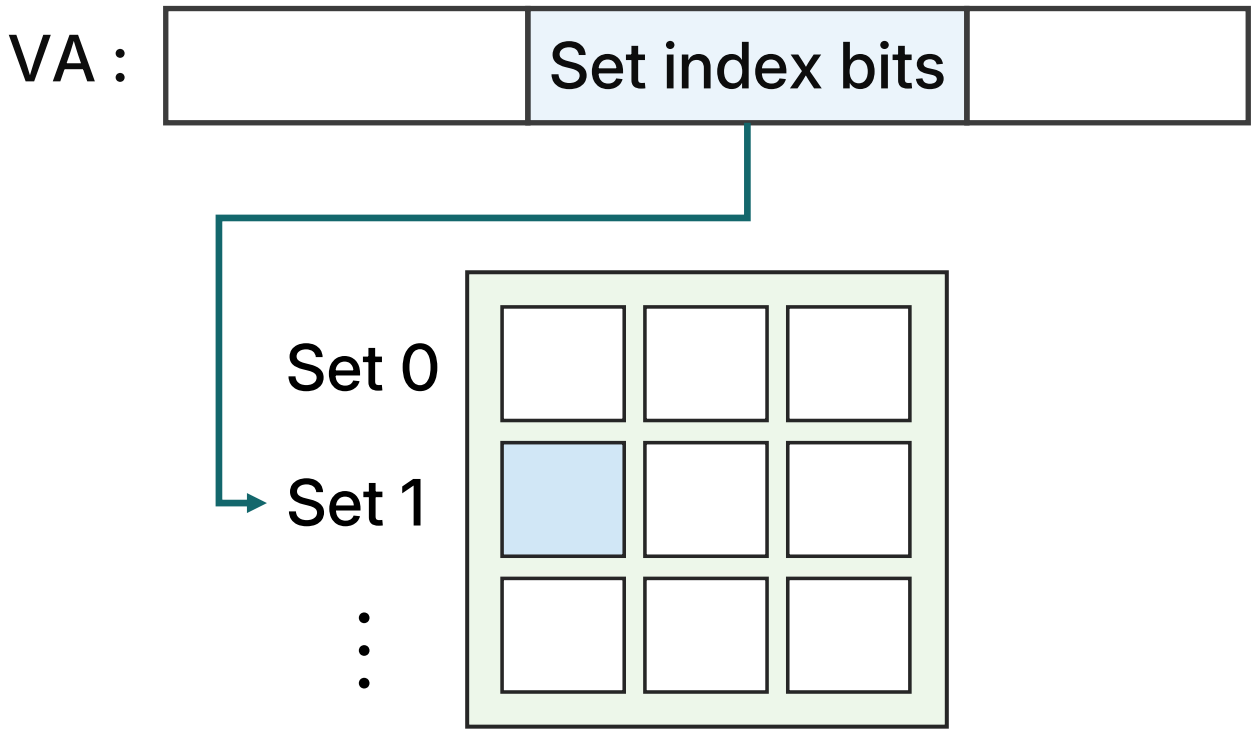
- C3. KASLR implementation details on macOS for Apple silicon

# Prime + Probe

# Prime + Probe



# Prime + Probe



# Prime + Probe

(Goal) Observe
address v is loaded

1. Calculate eviction set
for address $v$

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

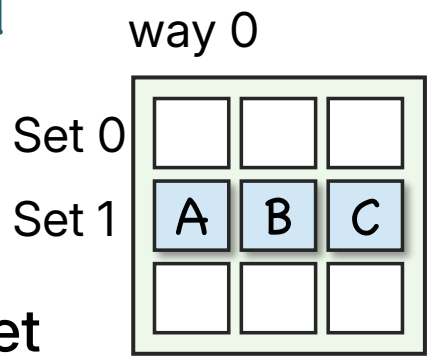
# Prime + Probe

2.Primes

(Goal) Observe
address v is loaded

1. Calculate eviction set
for address $v$

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



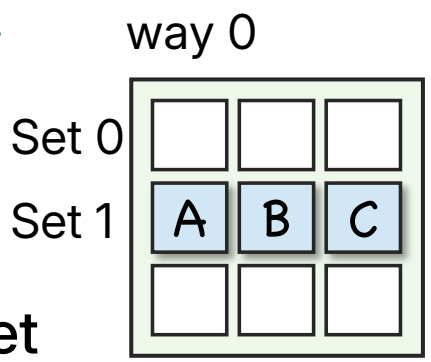
# Prime + Probe

(Goal) Observe address v is loaded

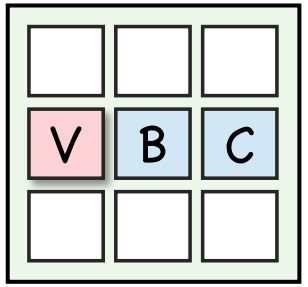
1. Calculate eviction set for address $v$

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

2.Primes




3. access $v$




If $v$ is valid

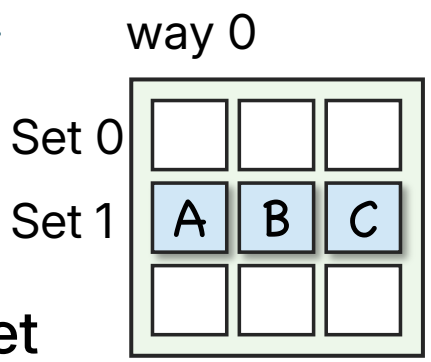
# Prime + Probe

(Goal) Observe address v is loaded

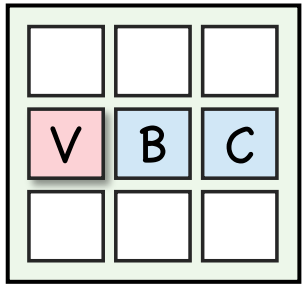
1. Calculate eviction set for address $v$

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

2.Primes

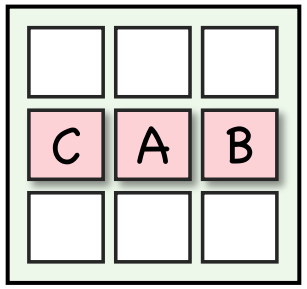



3. access $v$




If $v$ is valid

4. Probes




Miss for A,B,C!!

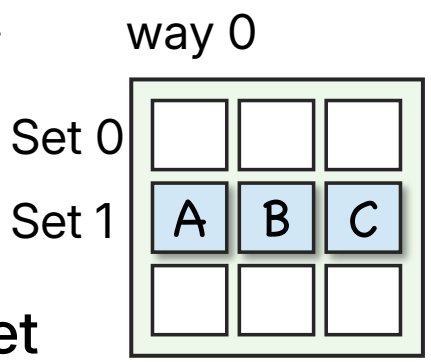
# Prime + Probe

(Goal) Observe address v is loaded

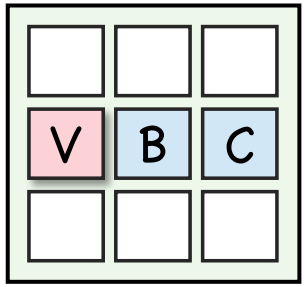
1. Calculate eviction set for address $v$

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

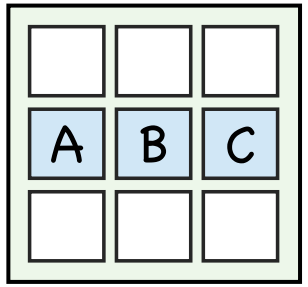
2.Primes




3. access $v$

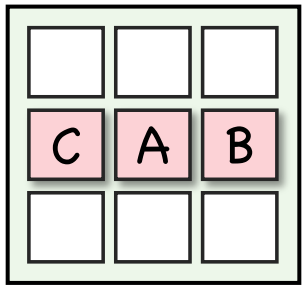



If $v$ is valid

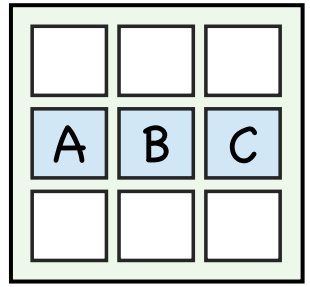



If $v$ is invalid

4. Probes




Miss for A,B,C!!



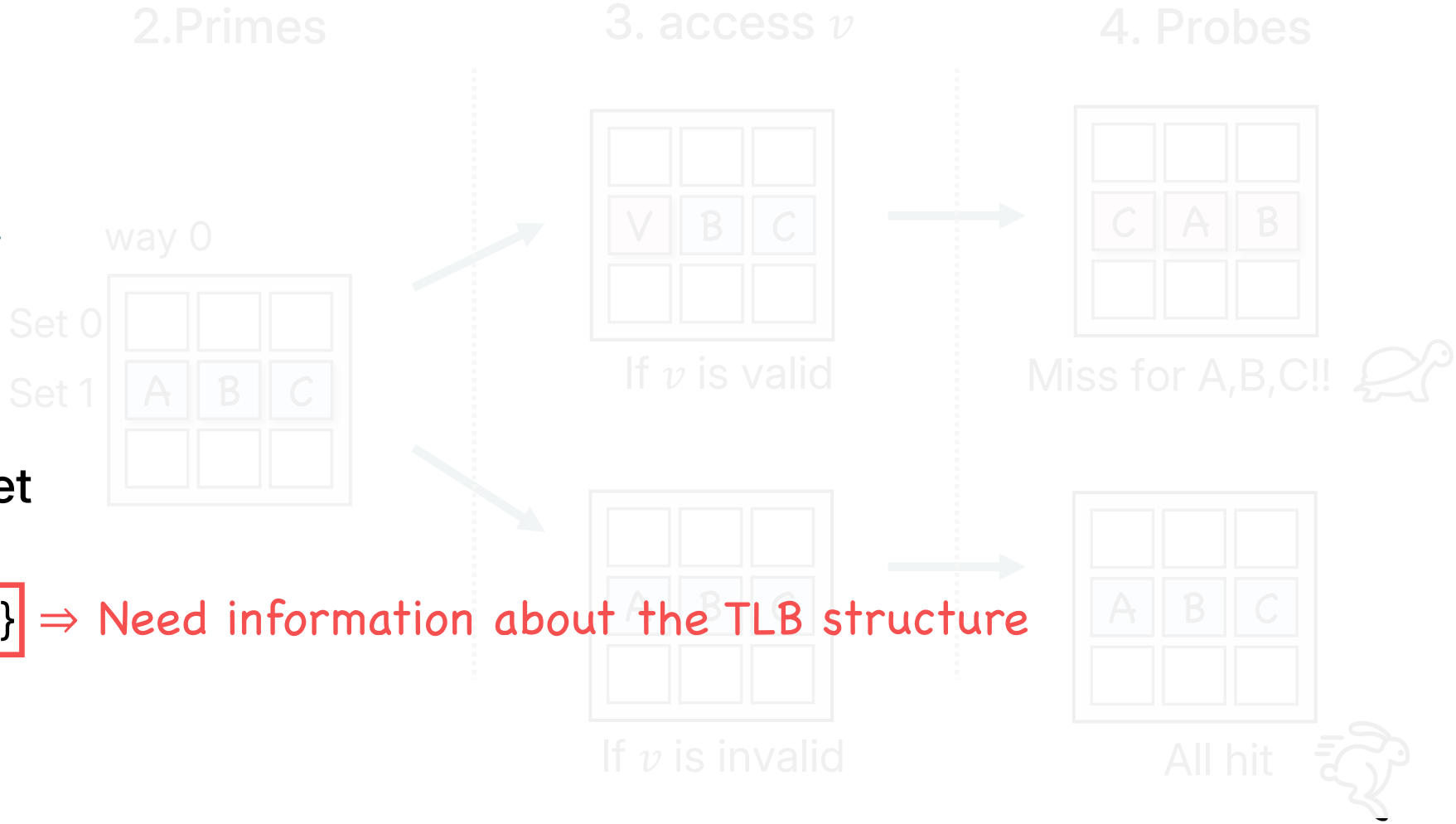

All hit

# Prime + Probe

(Goal) Observe address v is loaded

1. Calculate eviction set for address $v$

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



# TLB Reverse Engineering

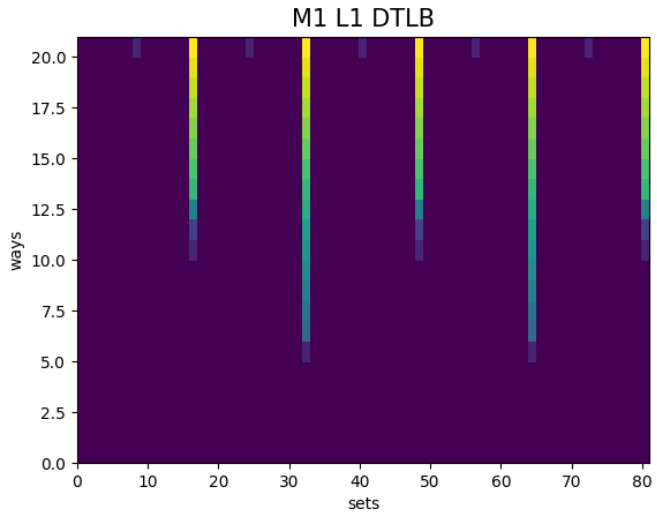
- Experimental environment

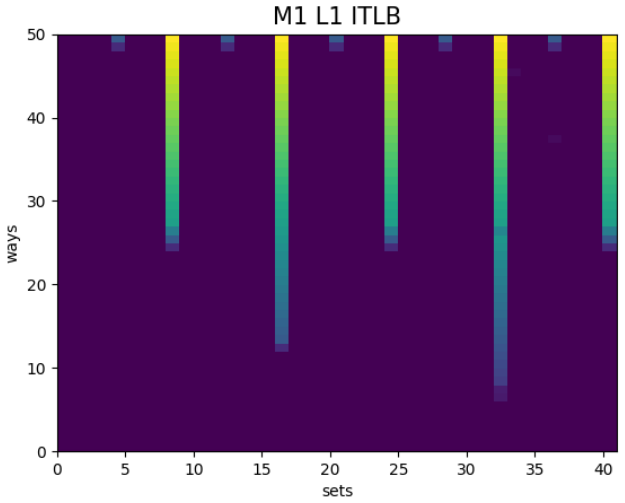
| CPU | Device | OS |
|---|---|---|
| M1 | Mac Mini (2021) | macOS Sonoma 14.3 |
| M1 Pro | MacBook Pro 16 | macOS Ventura 13.5 |
| M2 | Mac Mini (2023) | macOS Ventura 13.2 |
| M2 Pro | Mac Mini (2023) | macOS Ventura 13.2 |
| M2 Max | MacBook Pro 14 | macOS Ventura 13.4 |

# TLB Reverse Engineering

- Using CPU performance counters
  - Ibireme/kpc_demo.c [1]

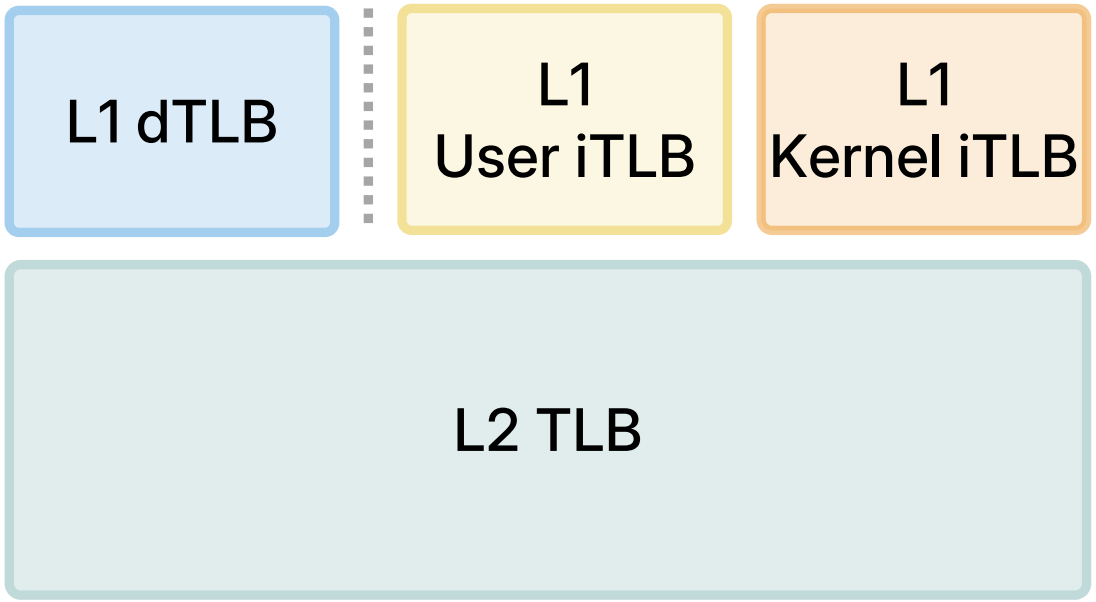
- Counters
  - `L1D_TLB_MISS_NONSPEC`
  - `L1I_TLB_MISS_DEMAND`
  - `L2_TLB_MISS_DATA`
  - `L2_TLB_MISS_INSTRUCTION`



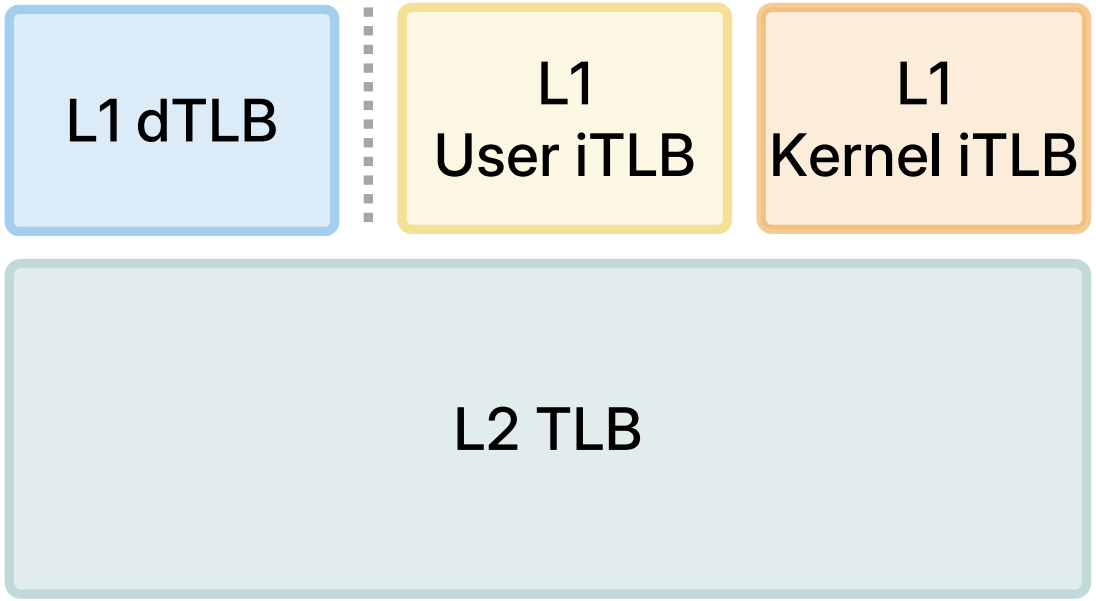

[1] https://gist.github.com/ibireme/173517c208c7dc333ba962c1f0d67d12

# Empirical TLB hierarchy




| | M1-series | M2-series |
|---|---|---|
| L1 dTLB | 32-set,<br>5-way | 64-set,<br>4-way |
| L1 iTLB | 32-set,<br>6-way | 32-set,<br>6-way |
| L2 TLB | 256-set,<br>12-way | 256-set,<br>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)

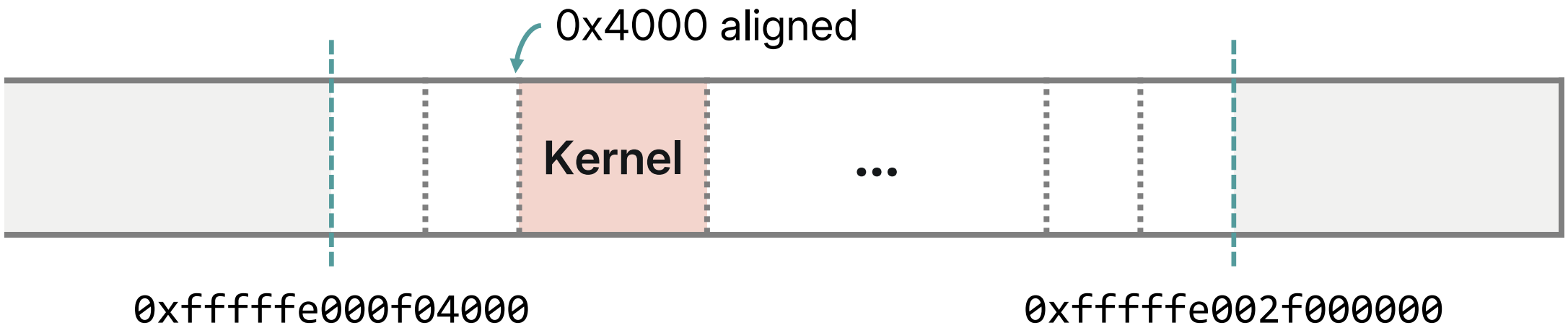
- C3. KASLR implementation details on macOS for Apple silicon

# 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

- ✅ 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

# Distinguishing Oracle with SysBumps Gadget

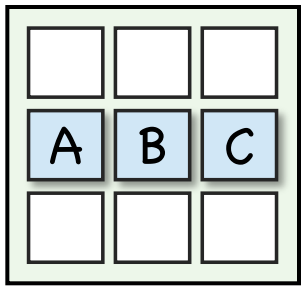
1. Training

# Distinguishing Oracle with SysBumps Gadget

1. Training

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

2.Primes



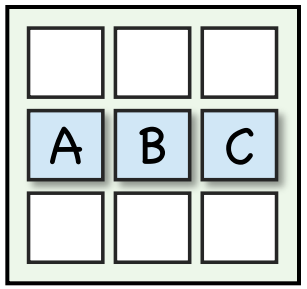

EvictionSet($v$)
= {A, B, C}

# Distinguishing Oracle with SysBumps Gadget

1. Training

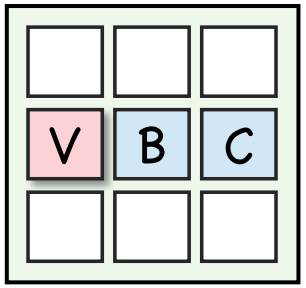
```
syscall(user_addr);
syscall(user_addr);
syscall(user_addr);
```

2.Primes

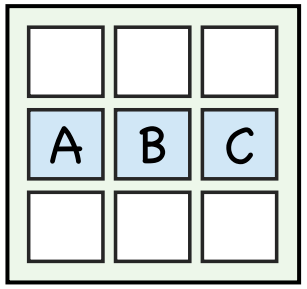



EvictionSet($v$)
= {A, B, C}

3. Syscall with $v$




If $v$ is valid



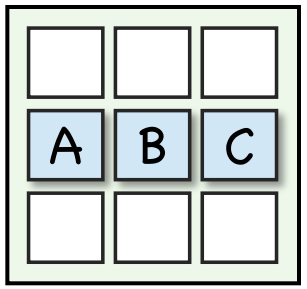

If $v$ is invalid

# Distinguishing Oracle with SysBumps Gadget

1. Training

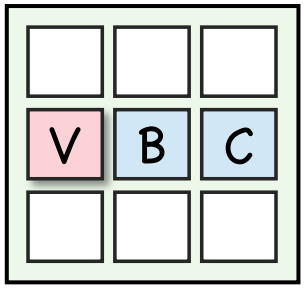
```
syscall(user_addr);
syscall(user_addr);
syscall(user_addr);
```

2.Primes

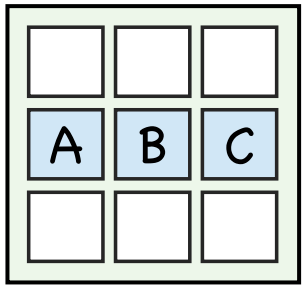



EvictionSet($v$)
= {A, B, C}

3. Syscall with $v$

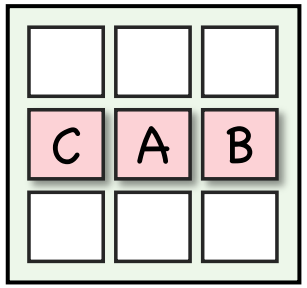



If $v$ is valid

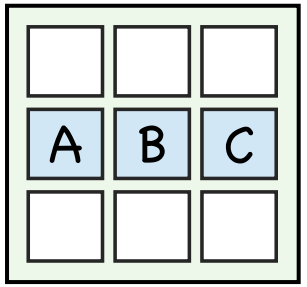



If $v$ is invalid

4. Probes

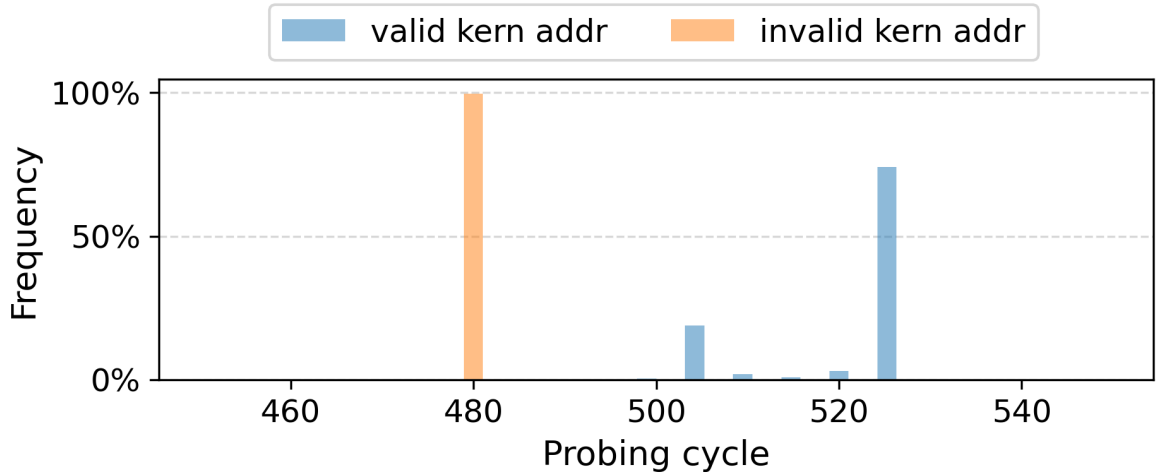



Miss for A,B,C!!

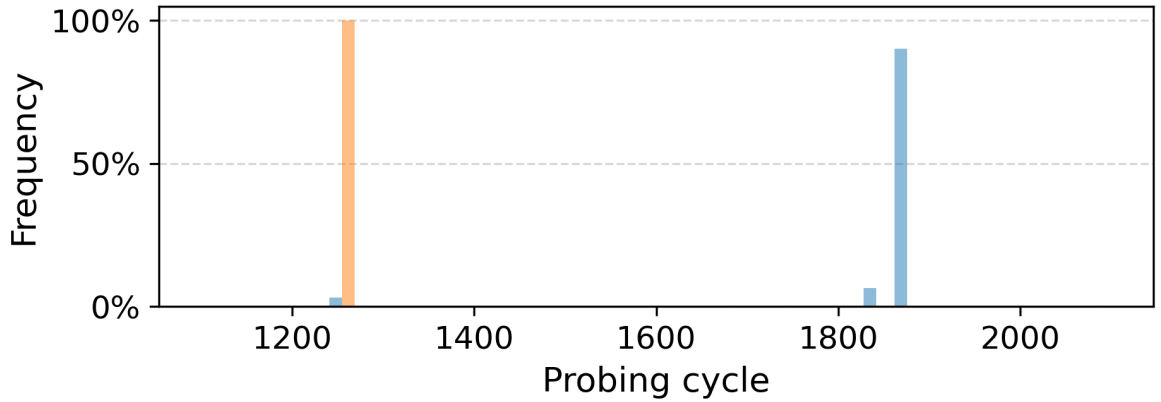



All hit

# Accuracy of Distinguishing Oracle




**(a) L1 dTLB**

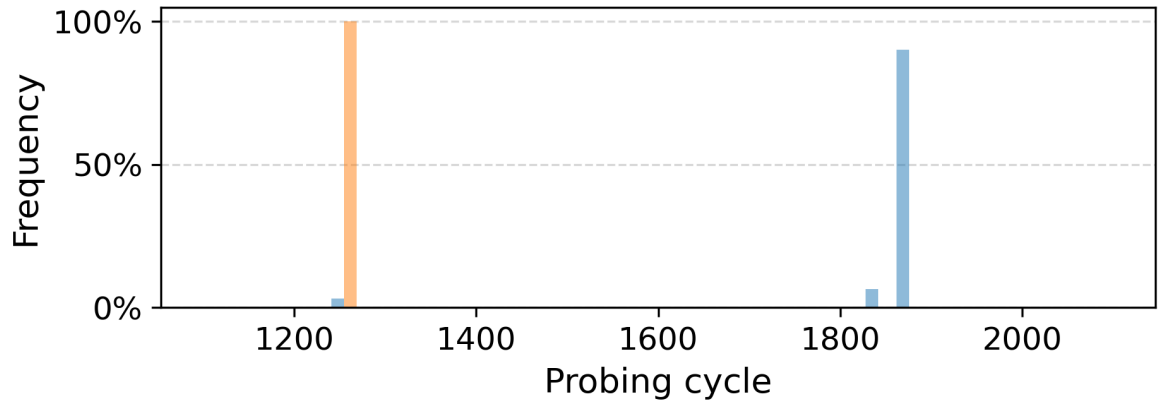



**(b) L2 TLB**

# 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 getattrlist(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_ssize_t getxattr(user_addr_t path, user_addr_t attrname, ... )` | Get an extended attribute value |
| 235 | `user_ssize_t fgetxattr(int fd, user_addr_t attrname, ... )` | Get an extended attribute value |

# Exploitable System calls

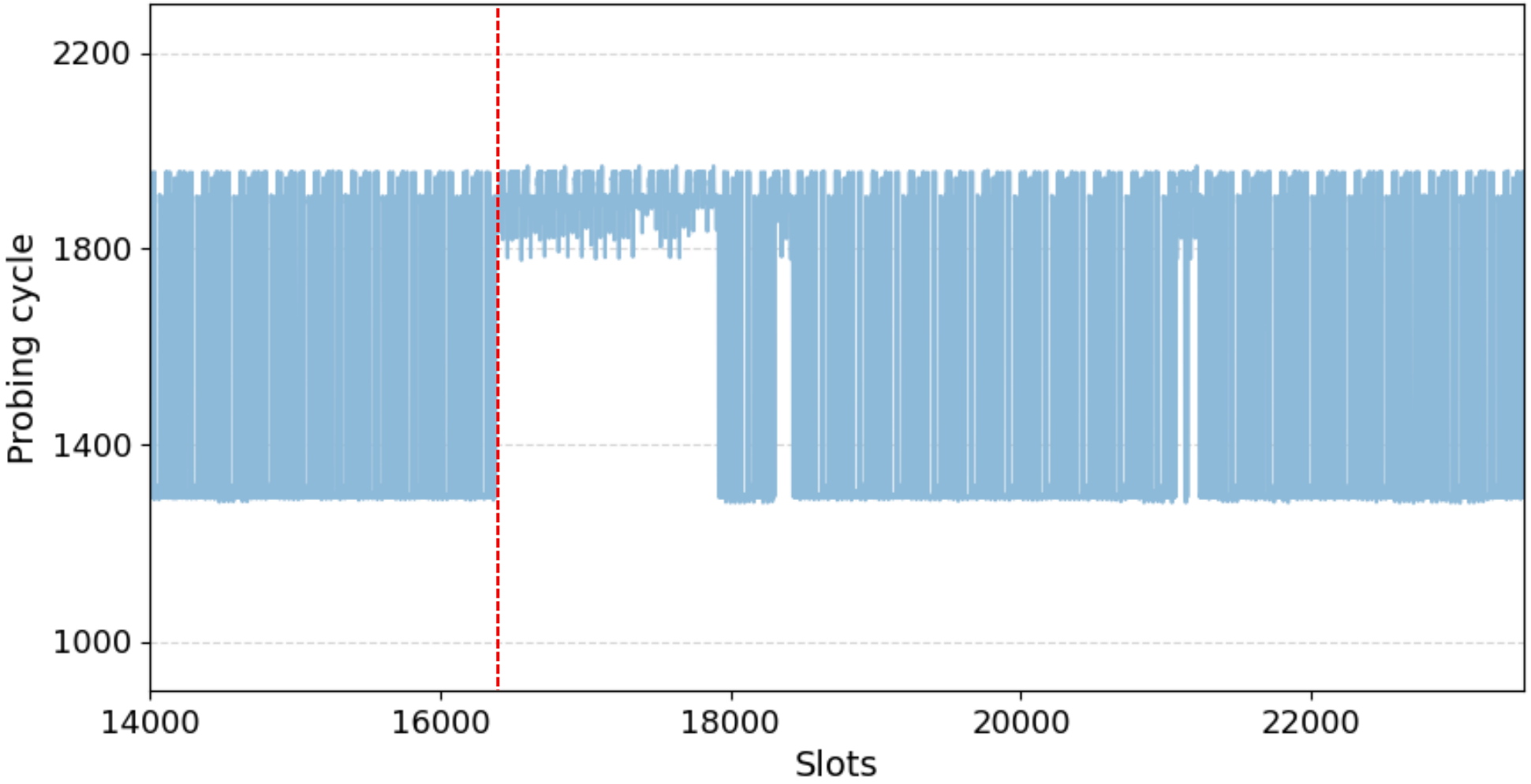
| Syscall no. | Prototype | Description |
|---|---|---|
| 5 | int access(const char *path, int amode) | Check access permissions of a file or pathname |
| 9 | int link(user_addr_t path, user_addr_t link) | Make a hard file link |
| 10 | int unlink(user_addr_t path) | Remove directory entry |
| 12 | int chdir(user_addr_t path) | Change current working direct |
| 15 | int chmod(user_addr_t path, int mode) | Change mod |
| 34 | int chflags(char *path, int flags) | |
| 56 | int revoke(char *path) | |
| 57 | int symlink(char *path, char *link) | bolic link to a file |
| 58 | int readlink(char *pat | Read value of a symbolic link |
| 136 | int mkd | Make a directory file |
| | mode) | Manipulate filesystem quotas |
| | path, int cmd, ... ) | Get file status |
| | stat(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 getattrlist(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_ssize_t getxattr(user_addr_t path, user_addr_t attrname, ... ) | Get an extended attribute value |
| 235 | user_ssize_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

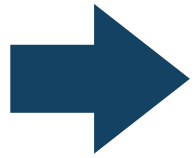
- Boot 100 times and perform 10 attacks on different kernel bases
  - Total 1000 attacks

**“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

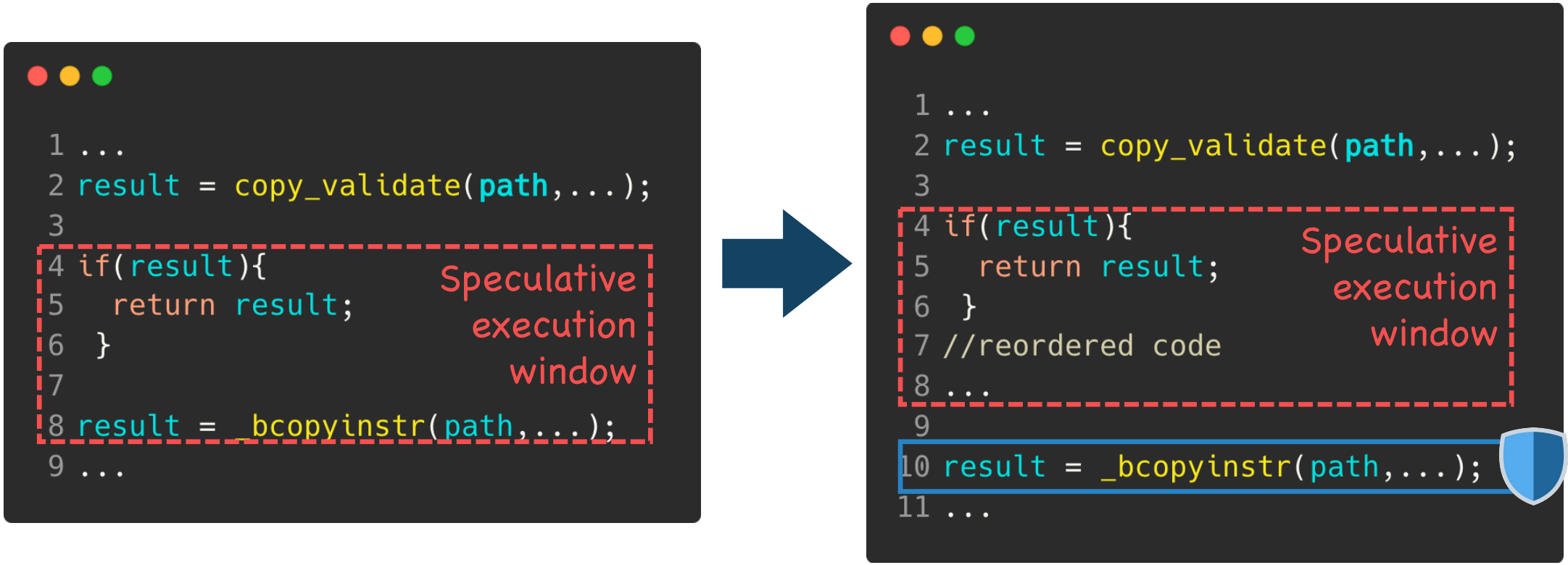
```
1 ...
2 result = copy_validate(path,...);
3
4 if(result){
5   return result;
6  }
7
8 result = _bcopyinstr(path,...);
9 ...
```



```
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 ...
```

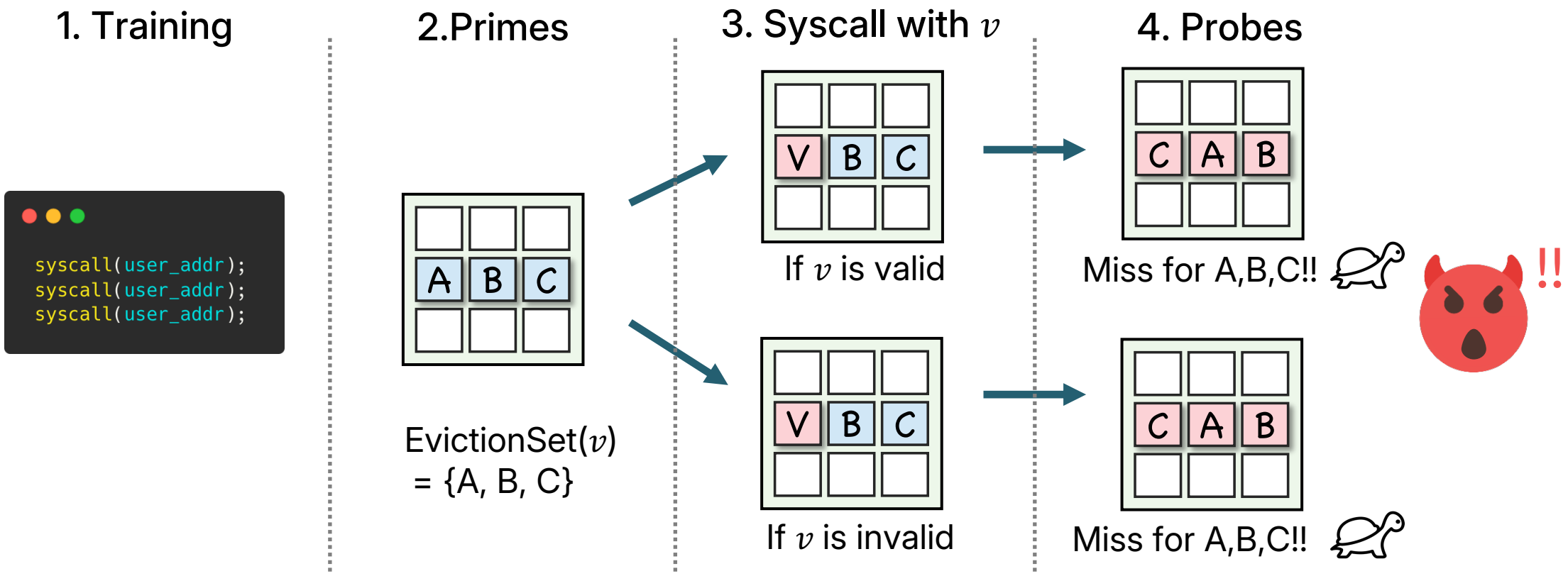
# Mitigations

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



# 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/koreacsl/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