

---

# PACMAN!

**Breaking PAC on the Apple M1 with Hardware Attacks.**

---

**Joseph Ravichandran\*, Weon Taek Na\*, Jay Lang, Mengjia Yan**

\*Both authors contributed equally to this work.





**Joseph Ravichandran**  
1st Year PhD Student, MIT



**Weon Taek Na**  
1st Year PhD Student, MIT

# \$whoami



**Jay Lang**  
M. Eng. Student, MIT



**Mengjia Yan**  
Assistant Professor, MIT  
(Our fearless leader)





# PACMAN: Attacking ARM Pointer Authentication with Speculative Execution

Joseph Ravichandran\*  
MIT CSAIL  
Cambridge, MA, USA  
jravi@mit.edu

Jay Lang  
MIT CSAIL  
Cambridge, MA, USA  
jaytlang@mit.edu

Weon Taek Na\*  
MIT CSAIL  
Cambridge, MA, USA  
weontaek@mit.edu

Mengjia Yan  
MIT CSAIL  
Cambridge, MA, USA  
mengjiay@mit.edu

## ABSTRACT

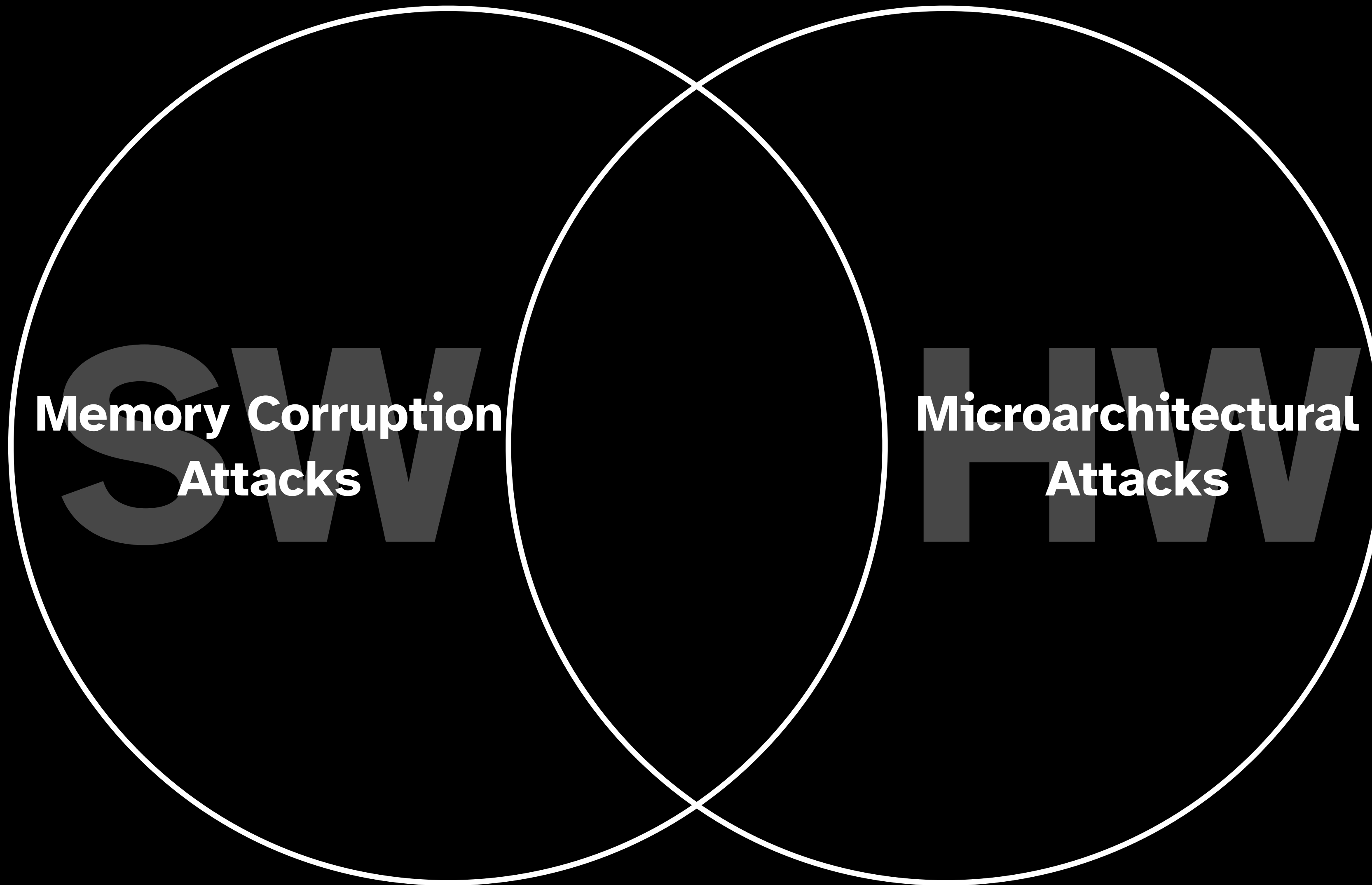
This paper studies the synergies between memory corruption vulnerabilities and speculative execution vulnerabilities. We leverage speculative execution attacks to bypass an important memory protection mechanism, ARM Pointer Authentication, a security feature that is used to enforce pointer integrity. We present PACMAN, a novel attack methodology that speculatively leaks PAC verification results via micro-architectural side channels without causing any crashes. Our attack removes the primary barrier to conducting control-flow hijacking attacks on a platform protected using Pointer Authentication.

We demonstrate multiple proof-of-concept attacks of PACMAN on the Apple M1 SoC, the first desktop processor that supports ARM Pointer Authentication. We reverse engineer the TLB hierarchy on the Apple M1 SoC and expand micro-architectural side-channel

## 1 INTRODUCTION

Modern systems are becoming increasingly complex, exposing a large attack surface with vulnerabilities in both software and hardware. In the software layer, memory corruption vulnerabilities [16, 56, 59, 61] (such as buffer overflows) can be exploited by attackers to alter the behavior or take full control of a victim program. In the hardware layer, micro-architectural side channel vulnerabilities [18, 25] (such as Spectre [37] and Meltdown [43]) can be exploited to leak arbitrary data within the victim program's address space. Today, it is common for security researchers to explore software and hardware vulnerabilities separately, considering the two vulnerabilities in two disjoint threat models.

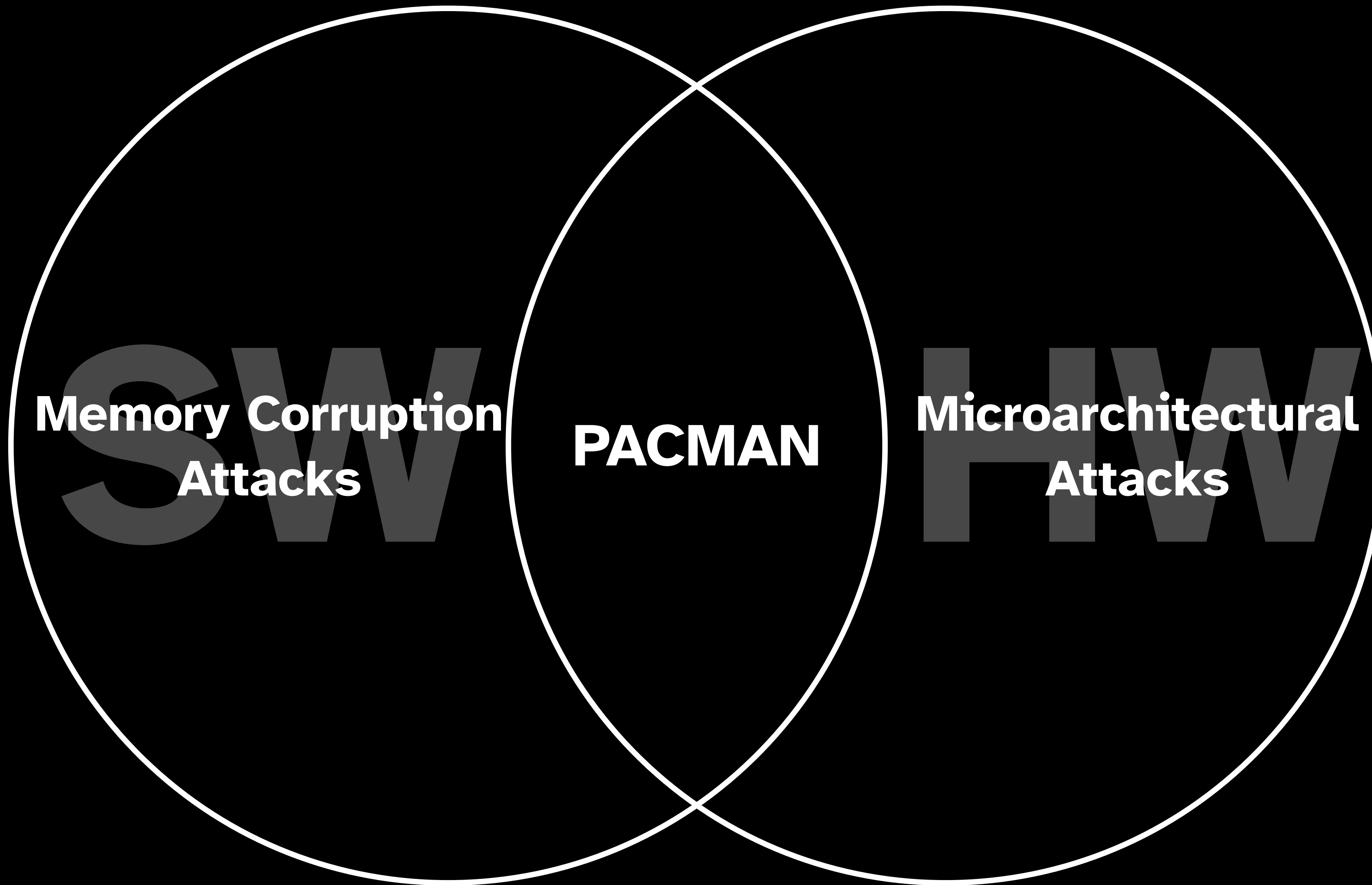
In this paper, we study the synergies between memory corruption vulnerabilities and micro-architectural side-channel vulnerabilities. We show how a hardware attack can be used to assist a



**Memory Corruption  
Attacks**

**Microarchitectural  
Attacks**





**Memory Corruption  
Attacks**

**PACMAN**

**Microarchitectural  
Attacks**



# Contributions

1

New way of thinking about  
compounding threat models.

3

Attack on  
Apple M1.

2

Hardware bypass for  
ARM Pointer Authentication.



**Think like  
an attacker.**

**Think like  
a CPU designer.**



**Is this  
a flaw?**



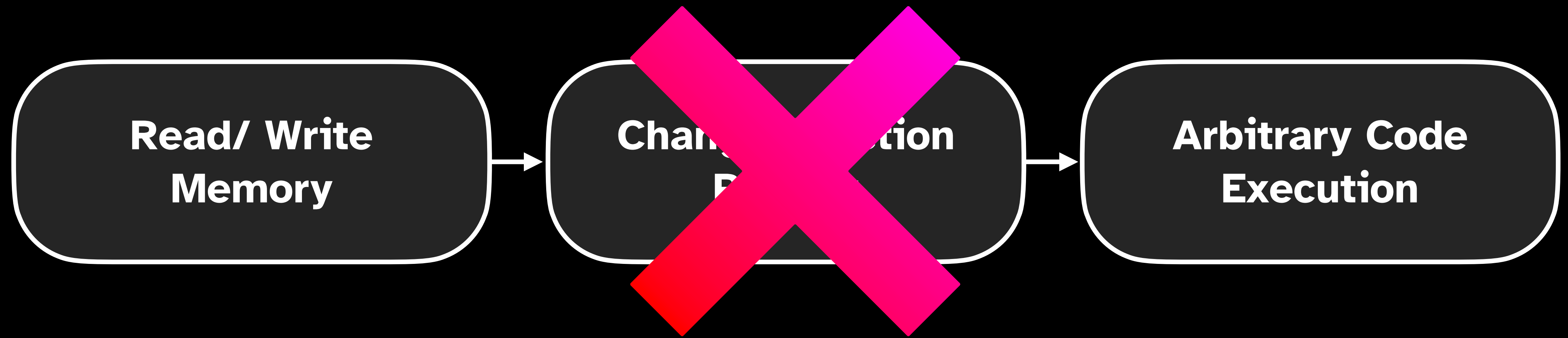
The **idea** in 60 seconds.

# Memory Corruption



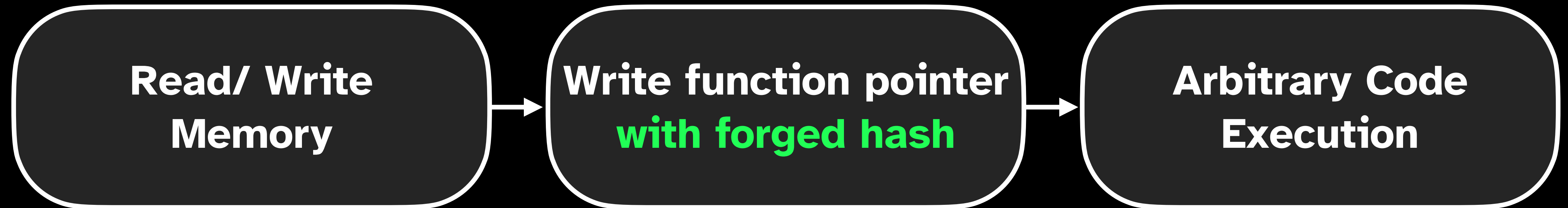


# Memory Corruption



**Pointer Authentication  
blocks changing pointers**

# Memory Corruption



**Just bruteforce it, right?**



**Key Insight:**  
**Avoid crashes using**  
**speculative execution!**

# Today

- 1 **Software**
- 2 **Hardware**
- 3 **PACMAN**
- 4 **Reverse Engineering M1**
- 5 **Putting it Together**

**4 New Tools + PoC**



# Announcing...

## PacmanGhidra

Static analysis scripts for Ghidra  
to locate PACMAN gadgets.

## PacmanKit

IOKit driver for performing  
PACMAN experiments.

## PacmanOS

Run your own Rust experiments  
bare metal on M1.

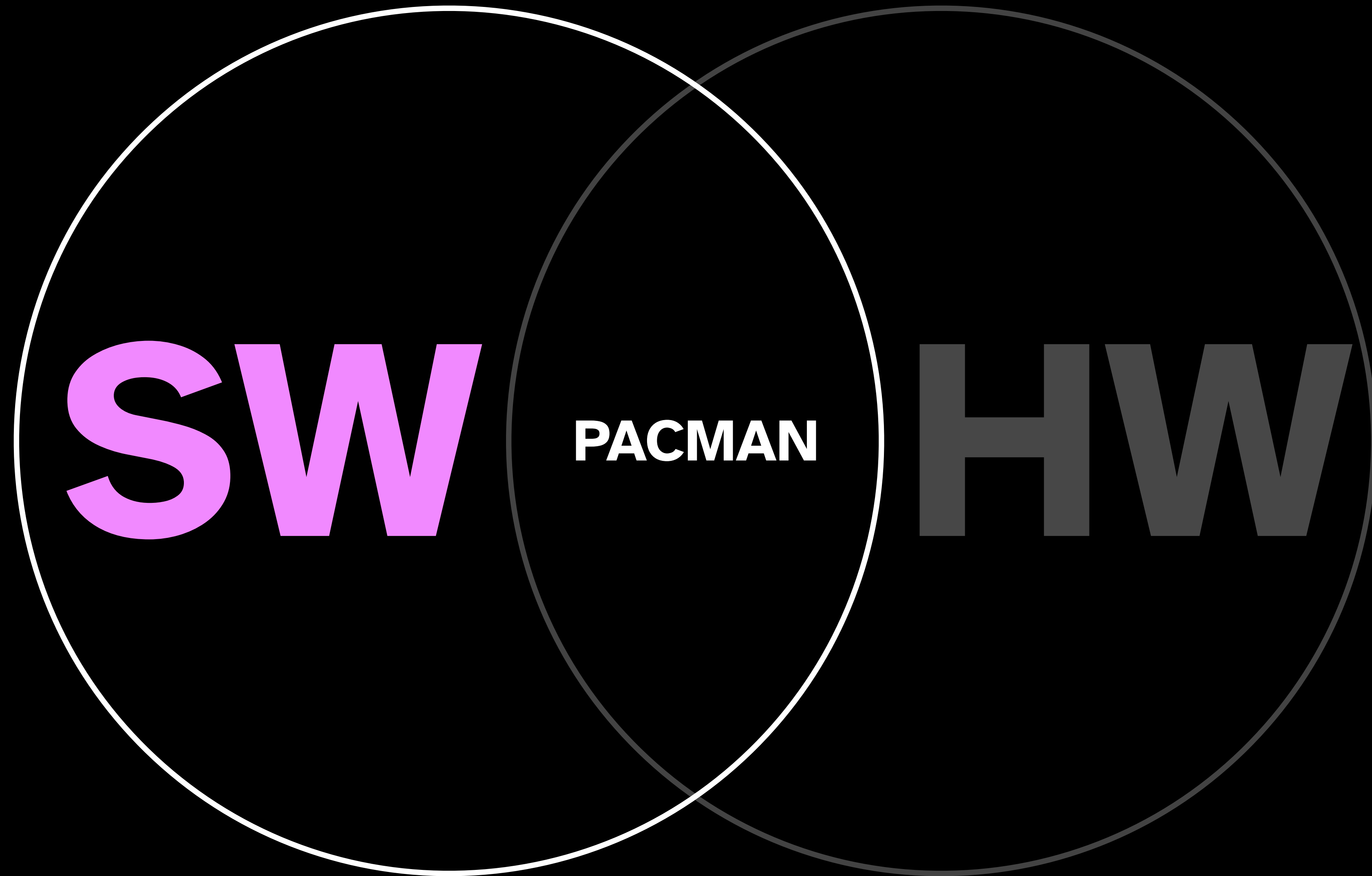
## PacmanPatcher

Patch your macOS kernel to  
enable high-resolution timers everywhere.

## PacmanPoC

Proof-of-Concept attack  
in both Rust and C.

**Software**



**Pointer**

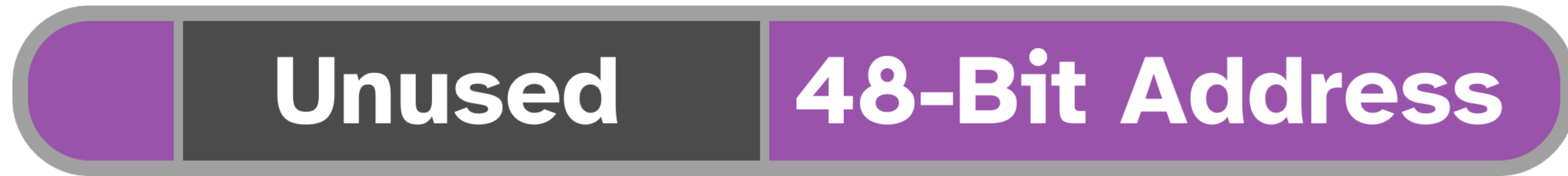
**64-Bit Address**

Our design doesn't  
have 16 exabytes of RAM...

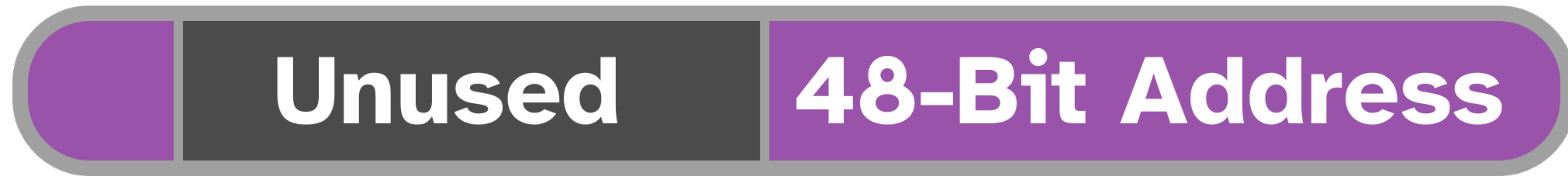
**Pointer**

**64-Bit Address**

**Pointer**



**Pointer**



**Let's put this to good use!**

# Pointer Authentication

$\text{PAC} = \text{hash}(\text{pointer}, \text{salt}, \text{key})$

Signed Pointer



**16 Bits**

**48 Bits**

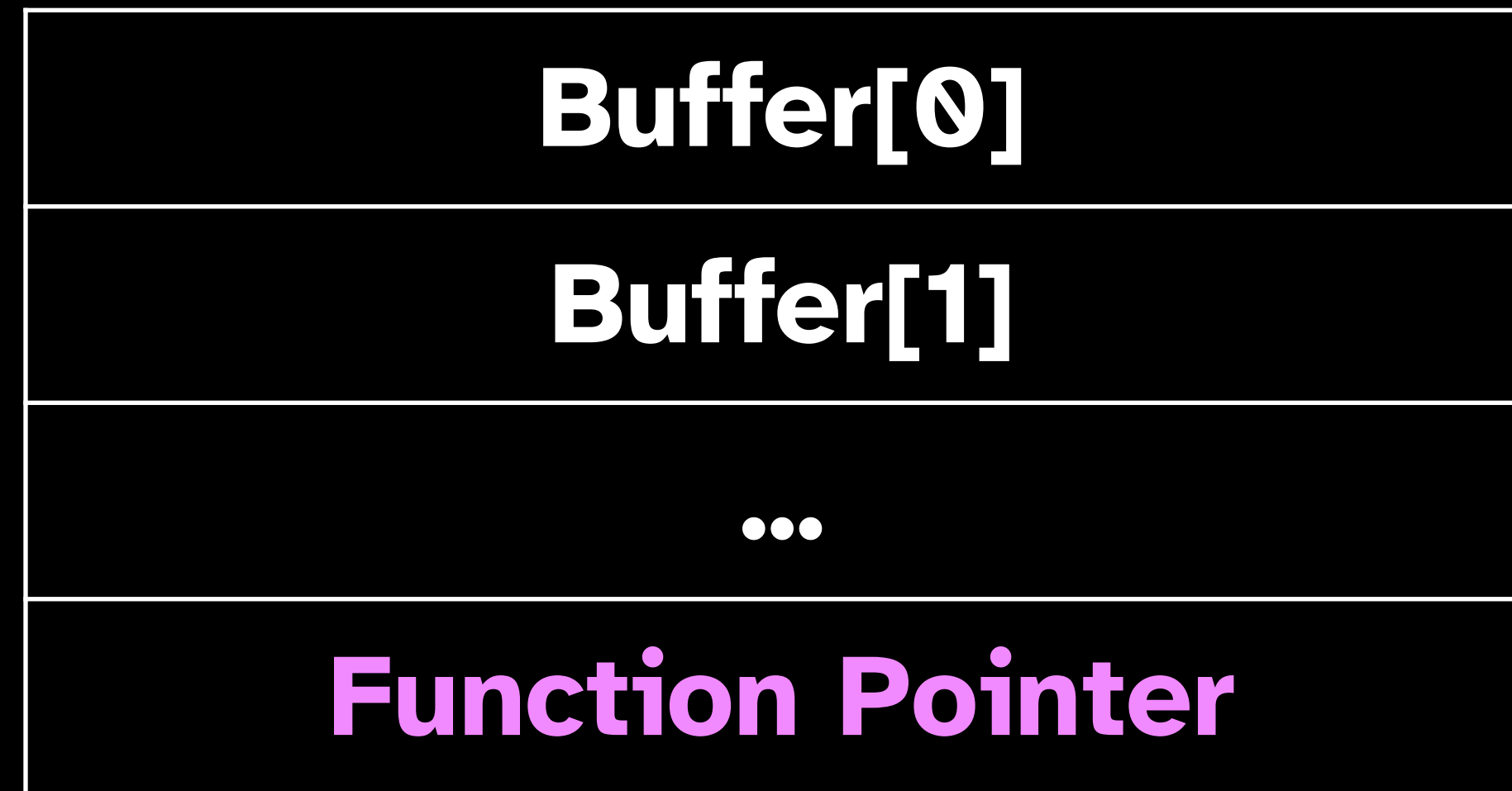
**Verifies**



# Buffer Overflow

|                         |
|-------------------------|
| <b>Buffer[0]</b>        |
| <b>Buffer[1]</b>        |
| <b>...</b>              |
| <b>Function Pointer</b> |

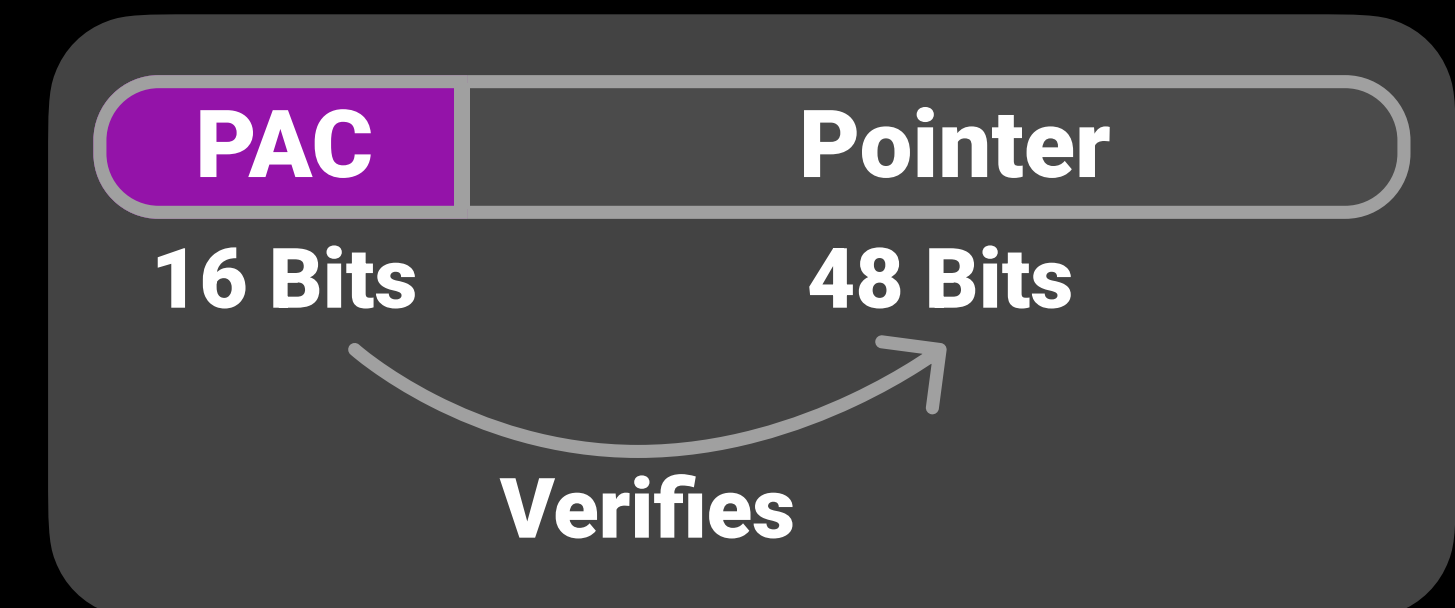
# Buffer Overflow



Buffer Overflow  
overwrites the  
function pointer!

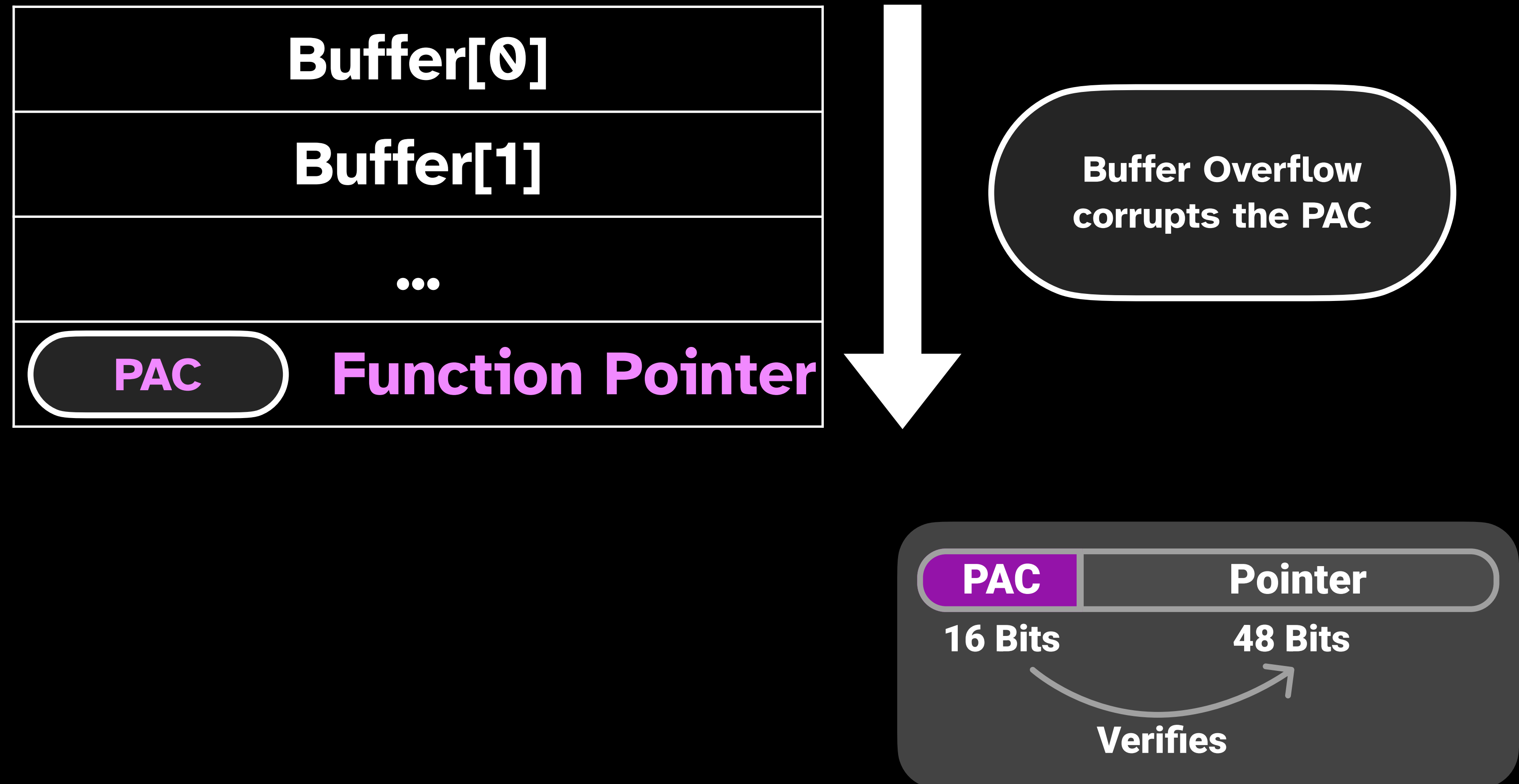
**Let's fix this bug with Pointer Authentication.**

# Buffer Overflow



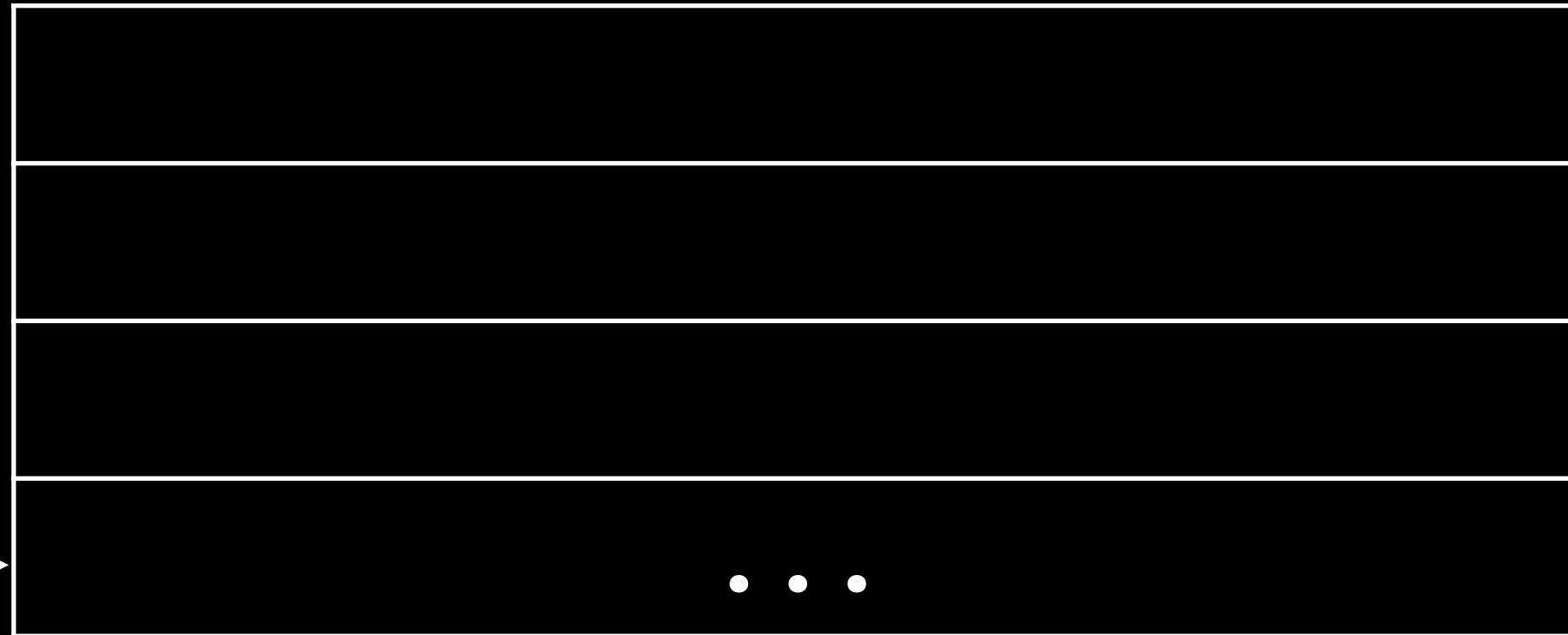
# Buffer Overflow

Invalid PAC means we **crash!**



# Buffer Overflow (No PAC)

sp

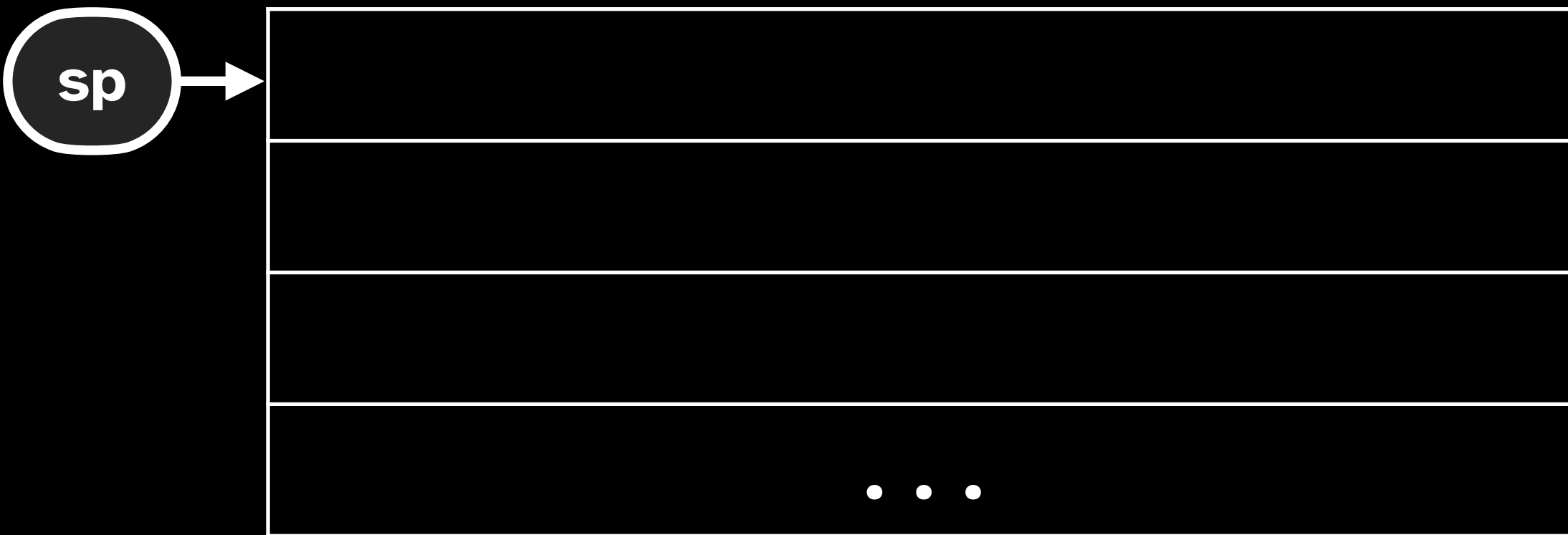


```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

\_vulnerable:

```
sub sp, sp, #48  
stp fp, lr, [sp, #32]  
...  
ldp fp, lr, [sp, #32]  
add sp, sp, #48  
ret
```

# Buffer Overflow (No PAC)



```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

\_vulnerable:

```
sub sp, sp, #48
```

```
stp fp, lr, [sp, #32]
```

...

```
ldp fp, lr, [sp, #32]
```

```
add sp, sp, #48
```

```
ret
```

**Create the stack frame**

# Buffer Overflow (No PAC)



```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

\_vulnerable:

```
sub sp, sp, #48
```

```
stp fp, lr, [sp, #32]
```

...

```
ldp fp, lr, [sp, #32]
```

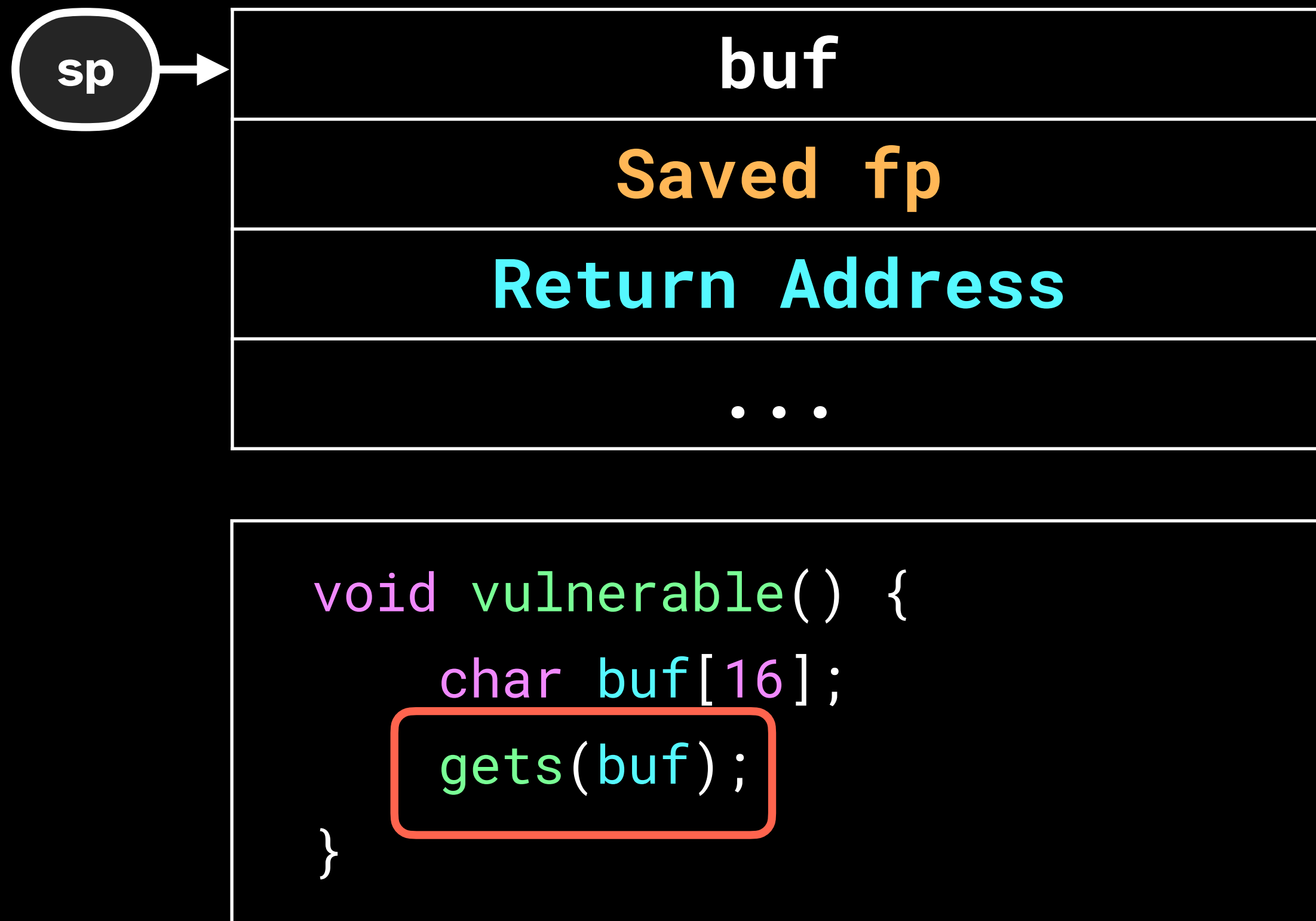
```
add sp, sp, #48
```

```
ret
```

**Save return address to the stack  
without any signature...**



# Buffer Overflow (No PAC)



\_vulnerable:

sub sp, sp, #48

stp fp, lr, [sp, #32]

...

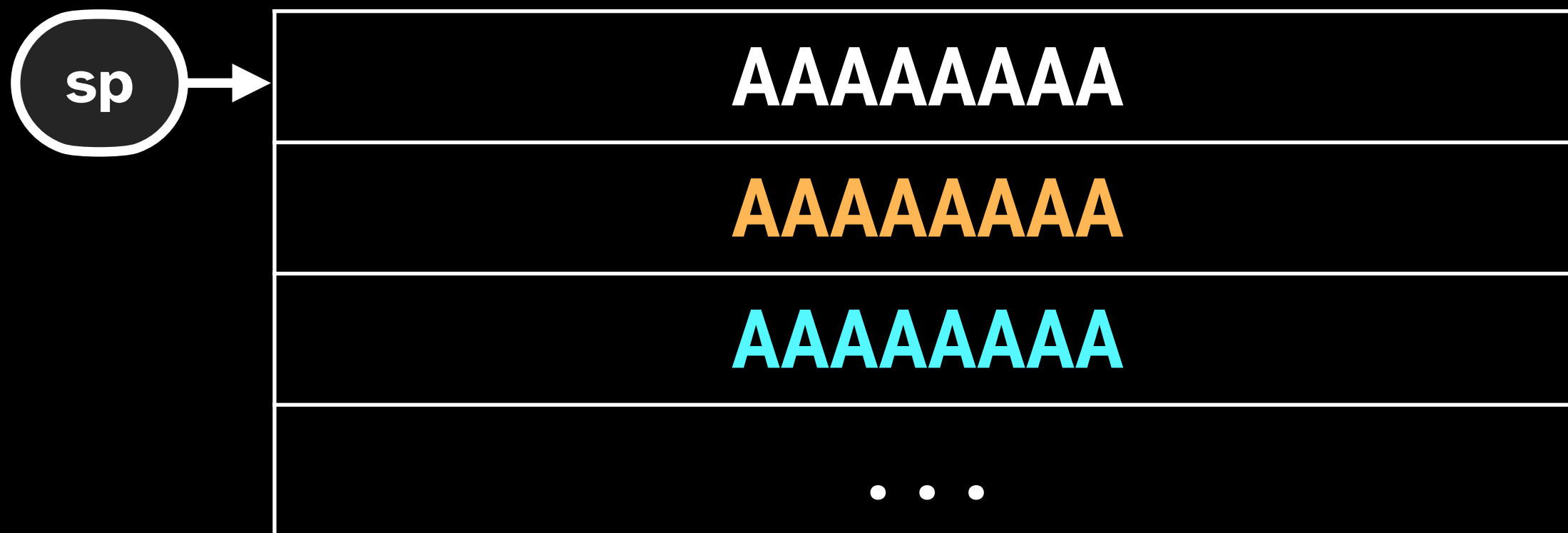
ldp fp, lr, [sp, #32]

add sp, sp, #48

ret

**This call allows us to overwrite the stack.**

# Buffer Overflow (No PAC)



```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

\_vulnerable:

```
sub sp, sp, #48
```

```
stp fp, lr, [sp, #32]
```

```
...
```

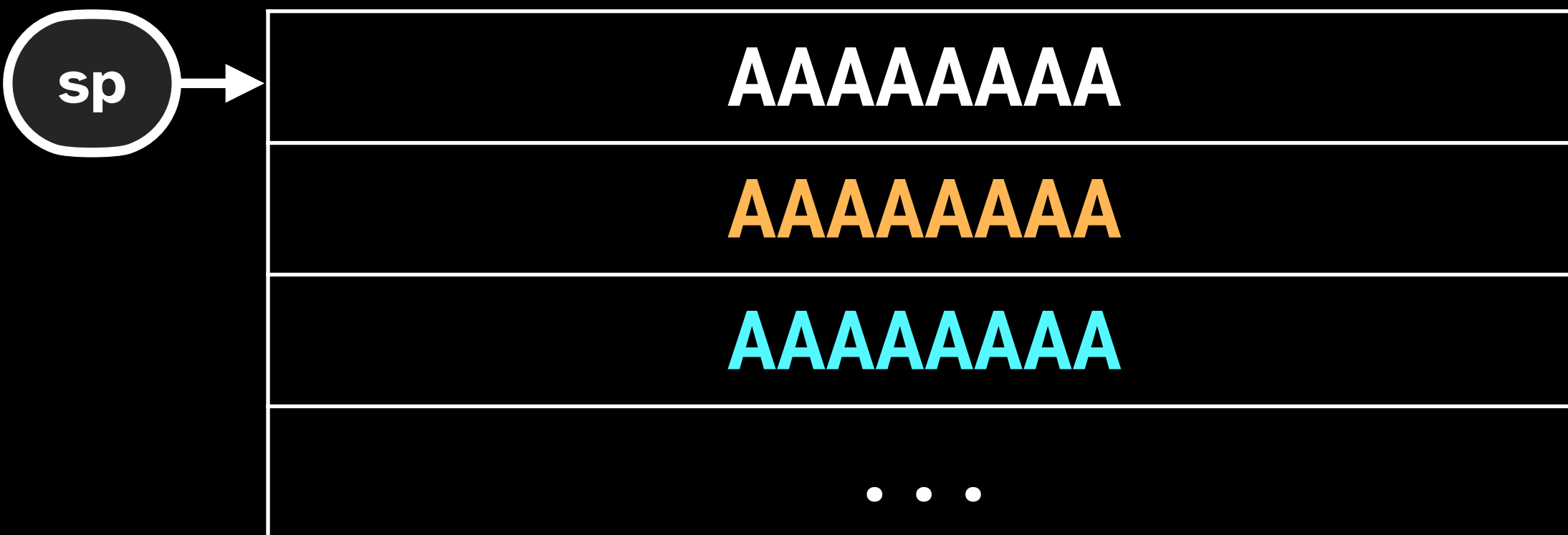
```
ldp fp, lr, [sp, #32]
```

```
add sp, sp, #48
```

```
ret
```

**This call allows us to write an unbounded number of chars into "buf".**

# Buffer Overflow (No PAC)



```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

`_vulnerable:`

```
sub sp, sp, #48
```

```
stp fp, lr, [sp, #32]
```

`...`

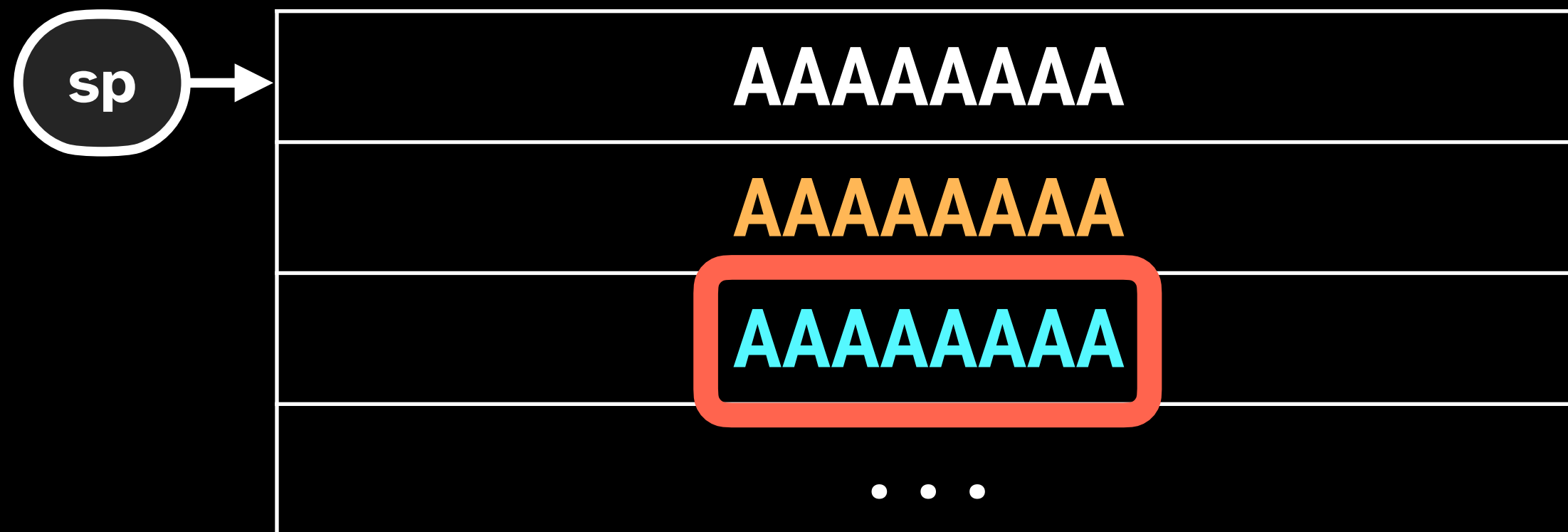
```
ldp fp, lr, [sp, #32]
```

```
add sp, sp, #48
```

```
ret
```

**Later, we trust the return address has not been changed!**

# Buffer Overflow (No PAC)



```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

`_vulnerable:`

```
sub sp, sp, #48
```

```
stp fp, lr, [sp, #32]
```

...

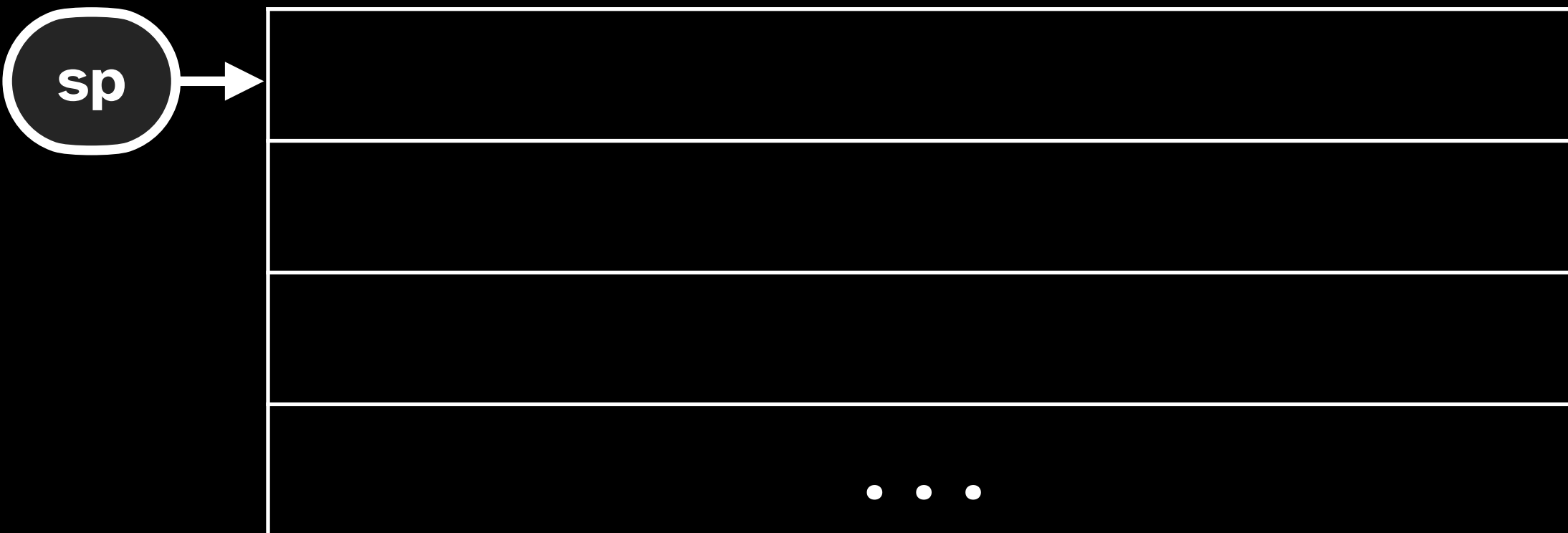
```
ldp fp, lr, [sp, #32]
```

```
add sp, sp, #48
```

```
ret
```

**And we return to the attacker-controlled address!**

# Buffer Overflow With PAC



```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

\_vulnerable:

paciza lr

sub sp, sp, #48

stp fp, lr, [sp, #32]

...

ldp fp, lr, [sp, #32]

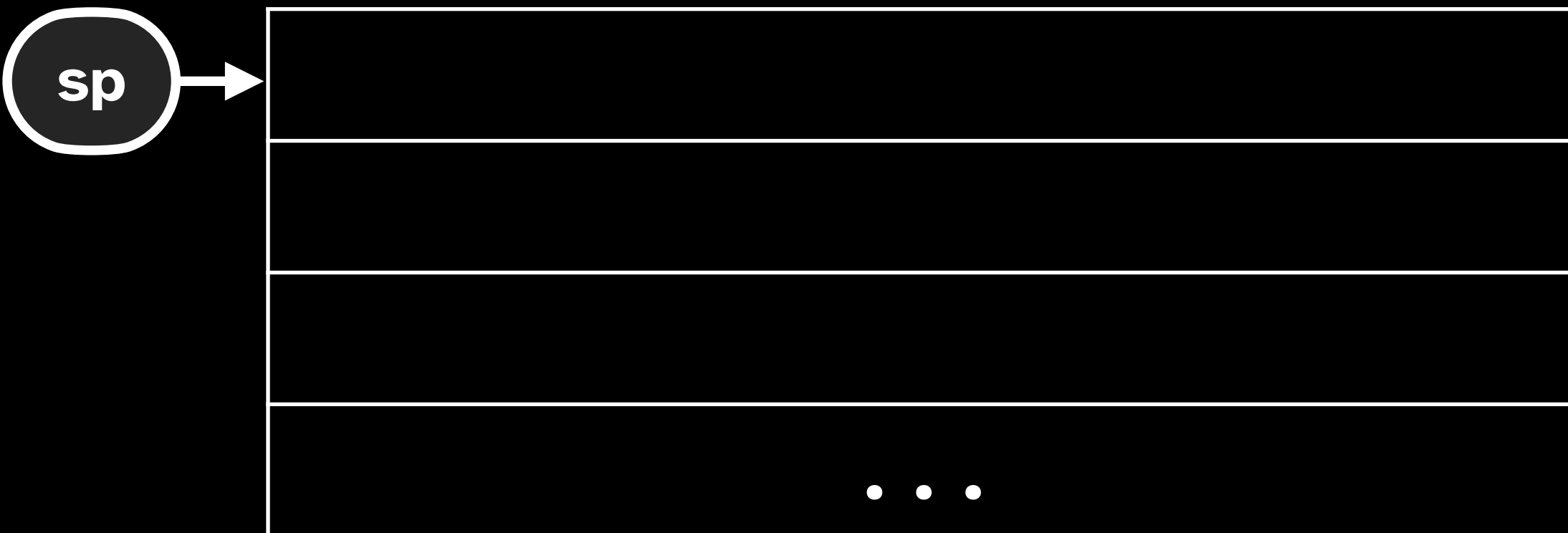
add sp, sp, #48

autiza lr

ret

**Sign and Verify instructions inserted to protect the return address.**

# Buffer Overflow With PAC



```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

\_vulnerable:

**paciza lr**

sub sp, sp, #48

stp fp, lr, [sp, #32]

...

ldp fp, lr, [sp, #32]

add sp, sp, #48

autiza lr

ret

**We begin by signing the return address.**

# Buffer Overflow With PAC



```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

`_vulnerable:`

`paciza lr`

`sub sp, sp, #48`

`stp fp, lr, [sp, #32]`

`...`

`ldp fp, lr, [sp, #32]`

`add sp, sp, #48`

`autiza lr`

`ret`

**Storing the *signed* pointer onto the stack.**

# Buffer Overflow With PAC



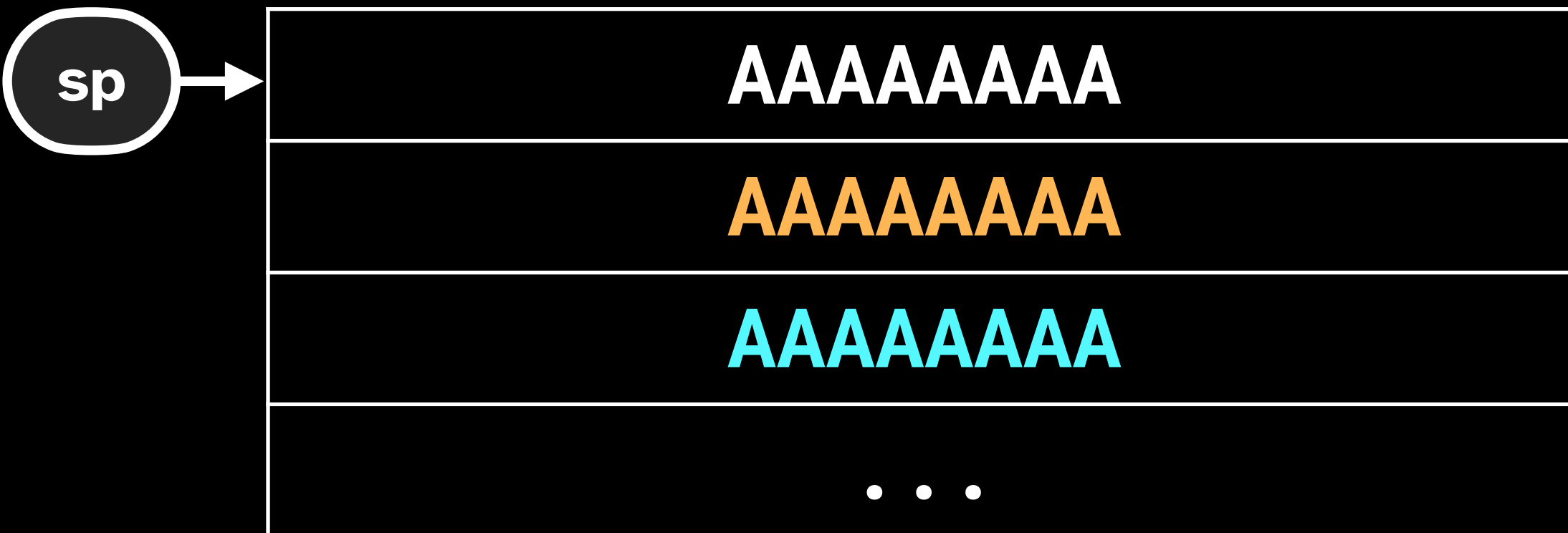
```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

```
_vulnerable:  
paciza lr  
sub  sp, sp, #48  
stp  fp, lr, [sp, #32]  
...  
ldp  fp, lr, [sp, #32]  
add  sp, sp, #48  
autiza lr  
ret
```

**Overflow occurs as normal.**



# Buffer Overflow With PAC

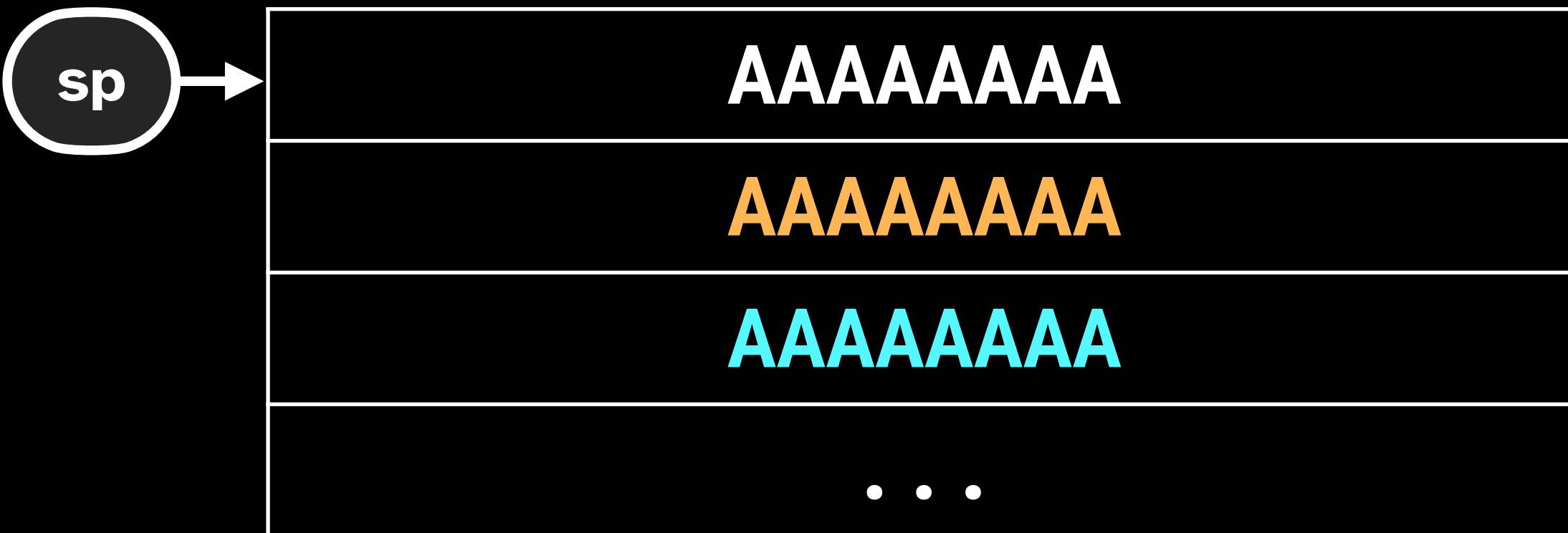


```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

```
_vulnerable:  
paciza lr  
sub  sp, sp, #48  
stp  fp, lr, [sp, #32]  
...  
ldp  fp, lr, [sp, #32]  
add  sp, sp, #48  
autiza lr  
ret
```

**Overflow occurs as normal.**

# Buffer Overflow With PAC

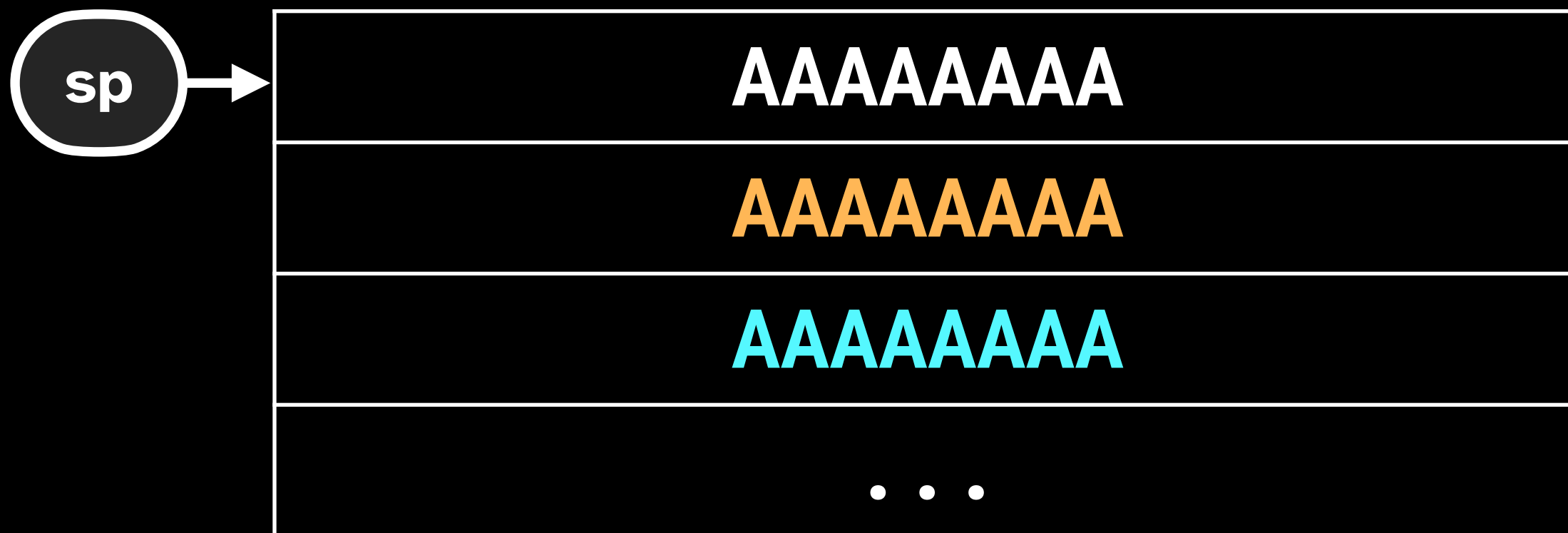


```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

```
_vulnerable:  
paciza lr  
sub  sp, sp, #48  
stp  fp, lr, [sp, #32]  
...  
ldp  fp, lr, [sp, #32]  
add  sp, sp, #48  
autiza lr  
ret
```

**Verify instruction will catch the changed value,  
setting lr to an "invalid pointer"**

# Buffer Overflow With PAC



```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

```
_vulnerable:  
paciza lr  
sub  sp, sp, #48  
stp  fp, lr, [sp, #32]  
...  
ldp  fp, lr, [sp, #32]  
add  sp, sp, #48  
autiza lr  
ret
```

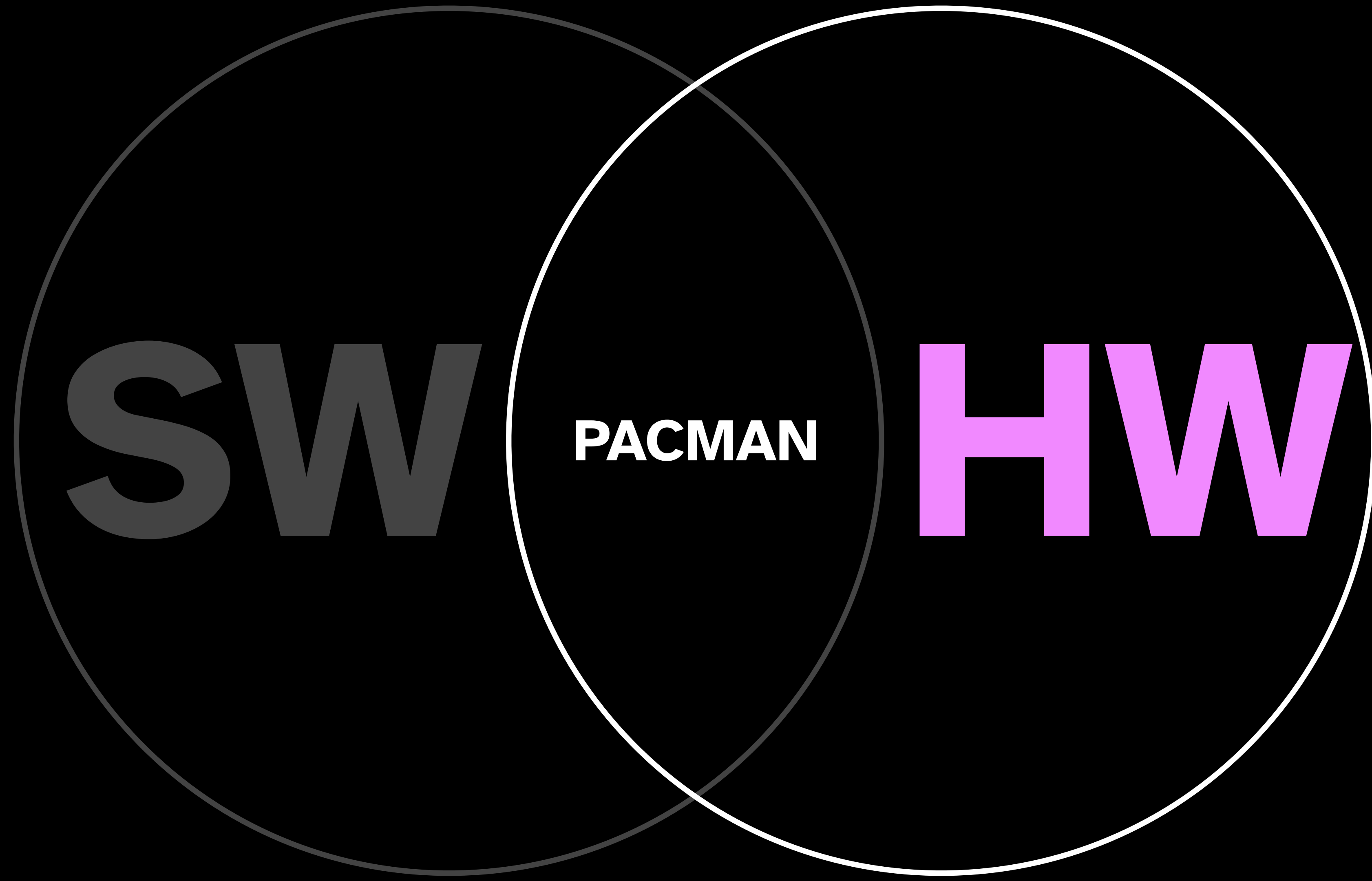
**We crash when we try to load the invalid pointer!**

# **THE GOAL**

**Reveal the PAC for an  
arbitrary pointer  
without crashing.**



**Hardware**



# Break PAC with Hardware Attacks

- Guess a PAC **speculatively** to prevent crashes
- Leak verification results via side channel

# Speculative Execution

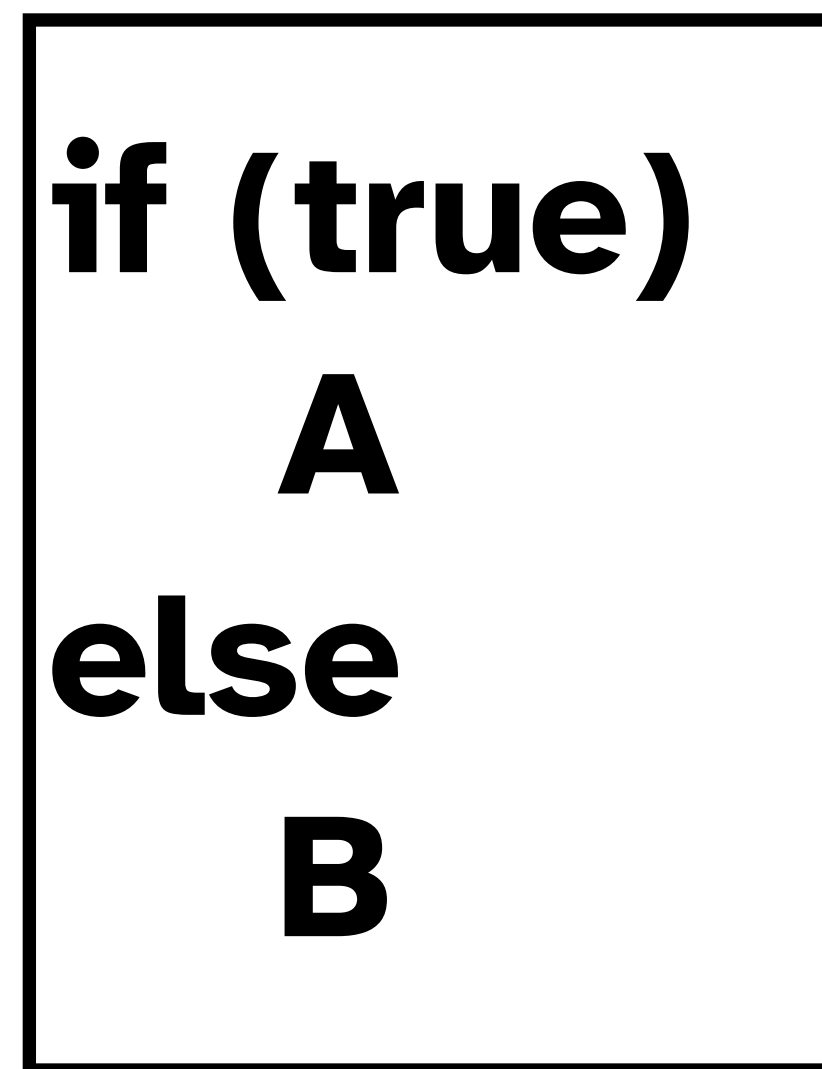
**if (true)**  
    **A**  
**else**  
    **B**

In Order



—————→ **Time**

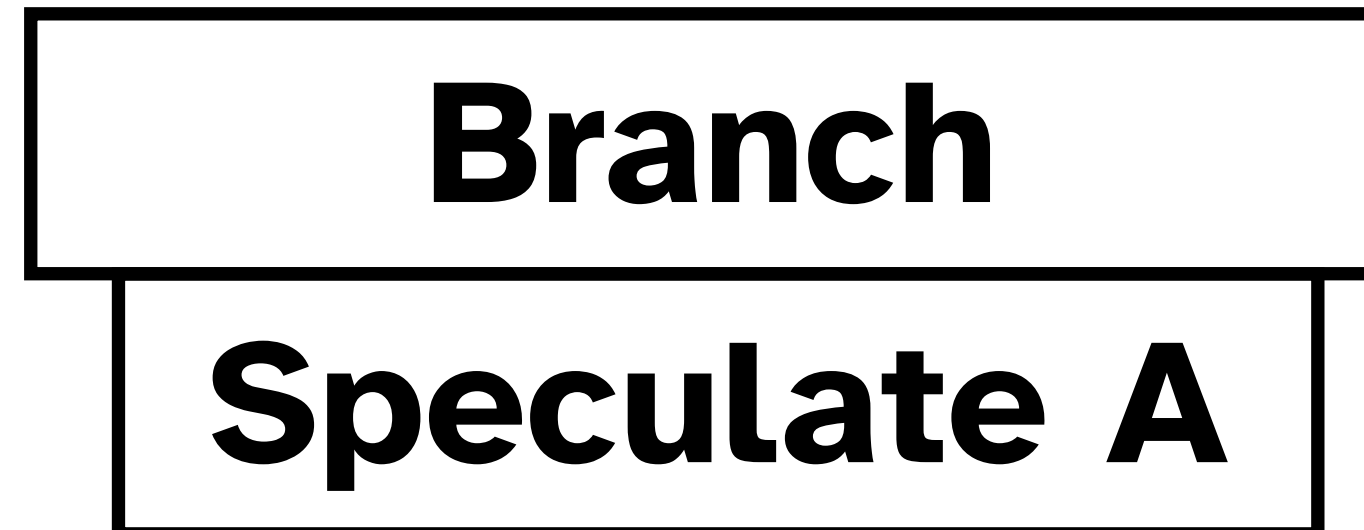
# Speculative Execution



In Order

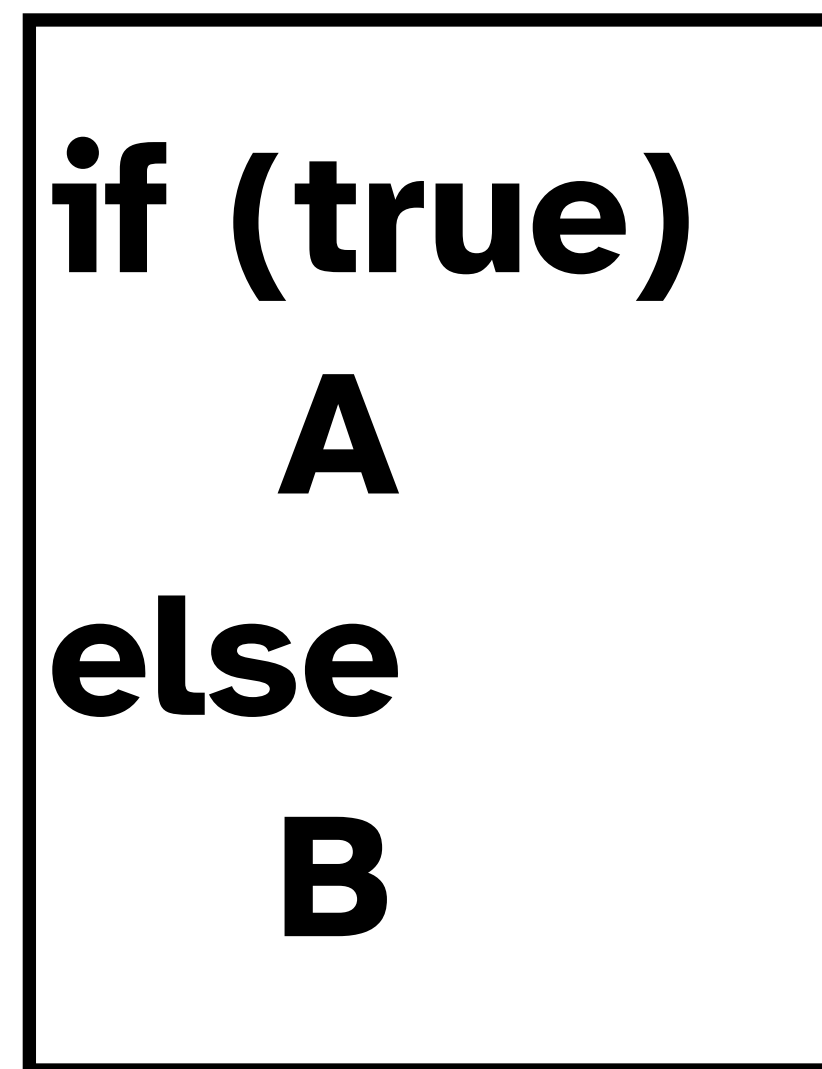


Speculative



→ Time

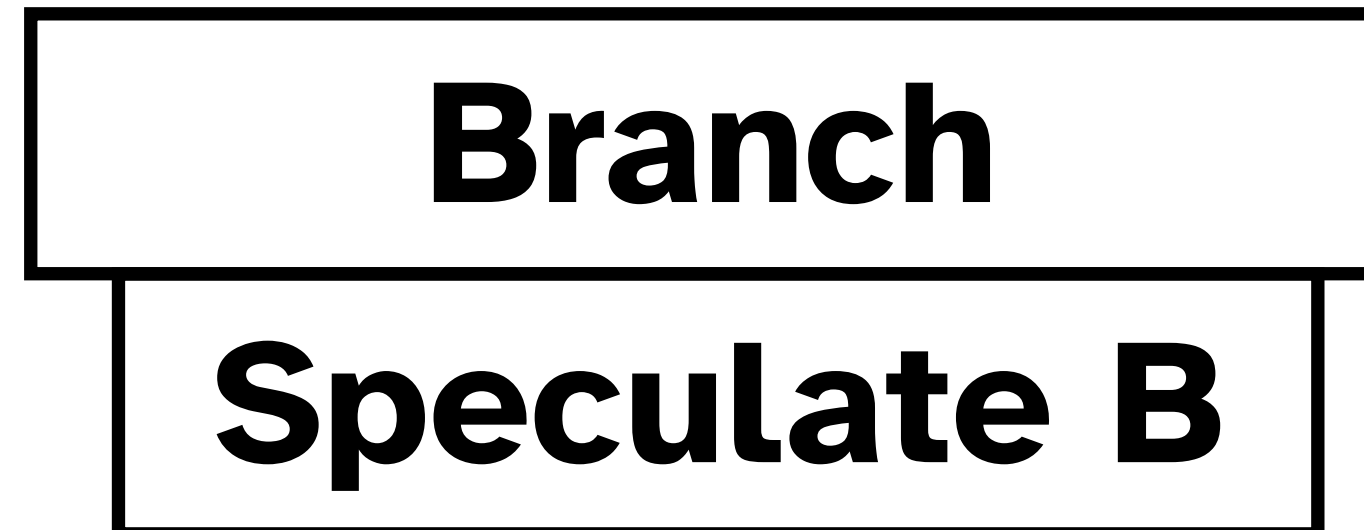
# Speculative Execution



In Order



Speculative



→ Time

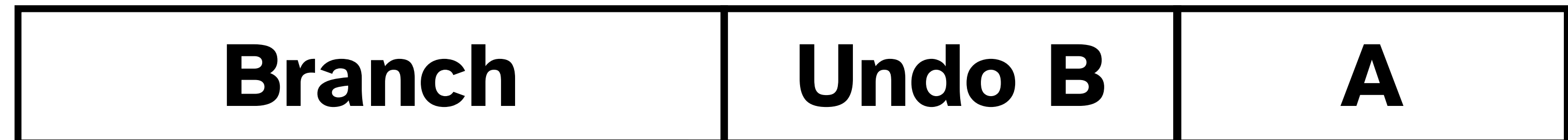
# Speculative Execution

**if (true)**  
**A**  
**else**  
**B**

In Order



Speculative



**Speculate B**

Microarchitectural  
side effects NOT undone

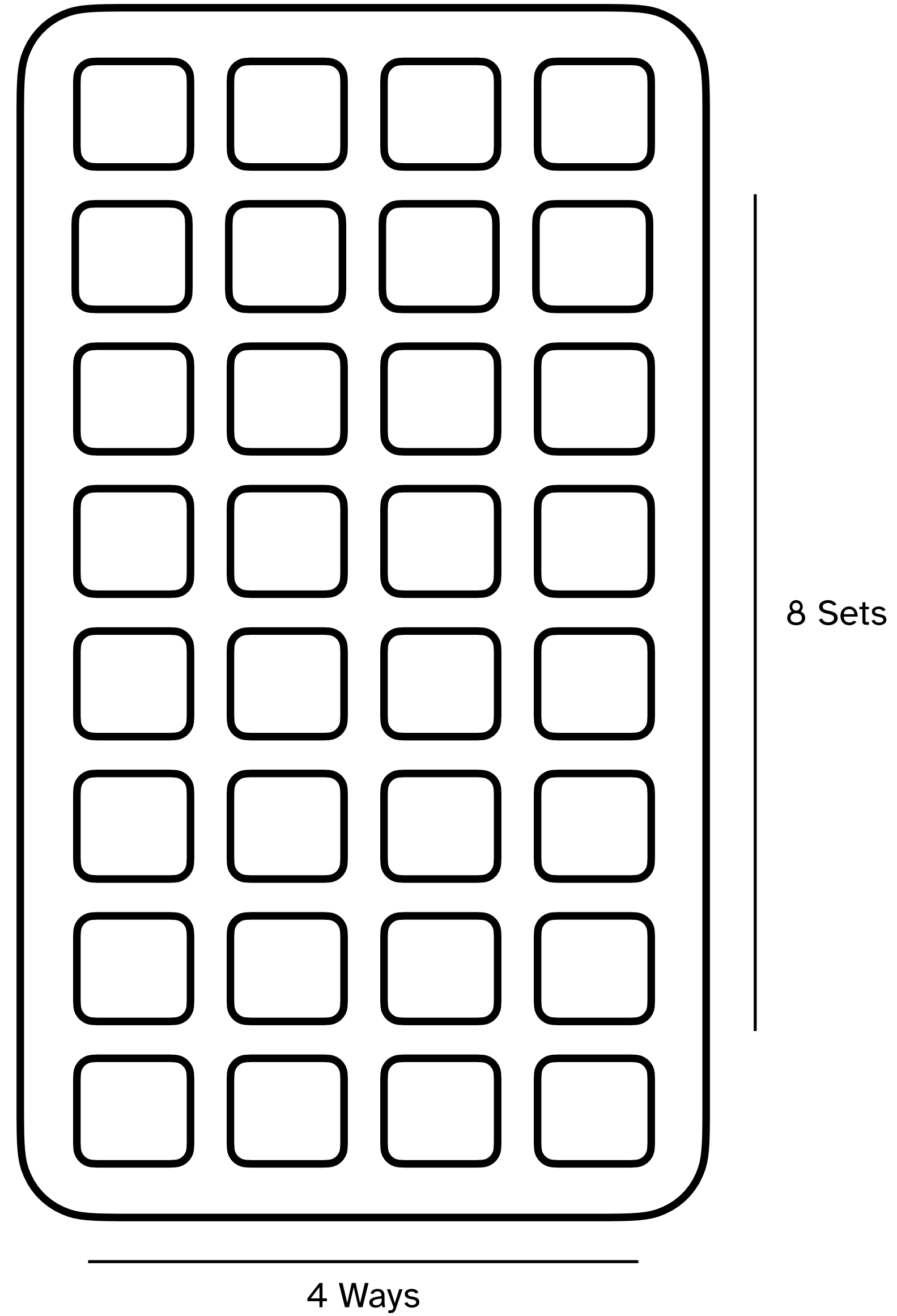
→ **Time**



# Cache

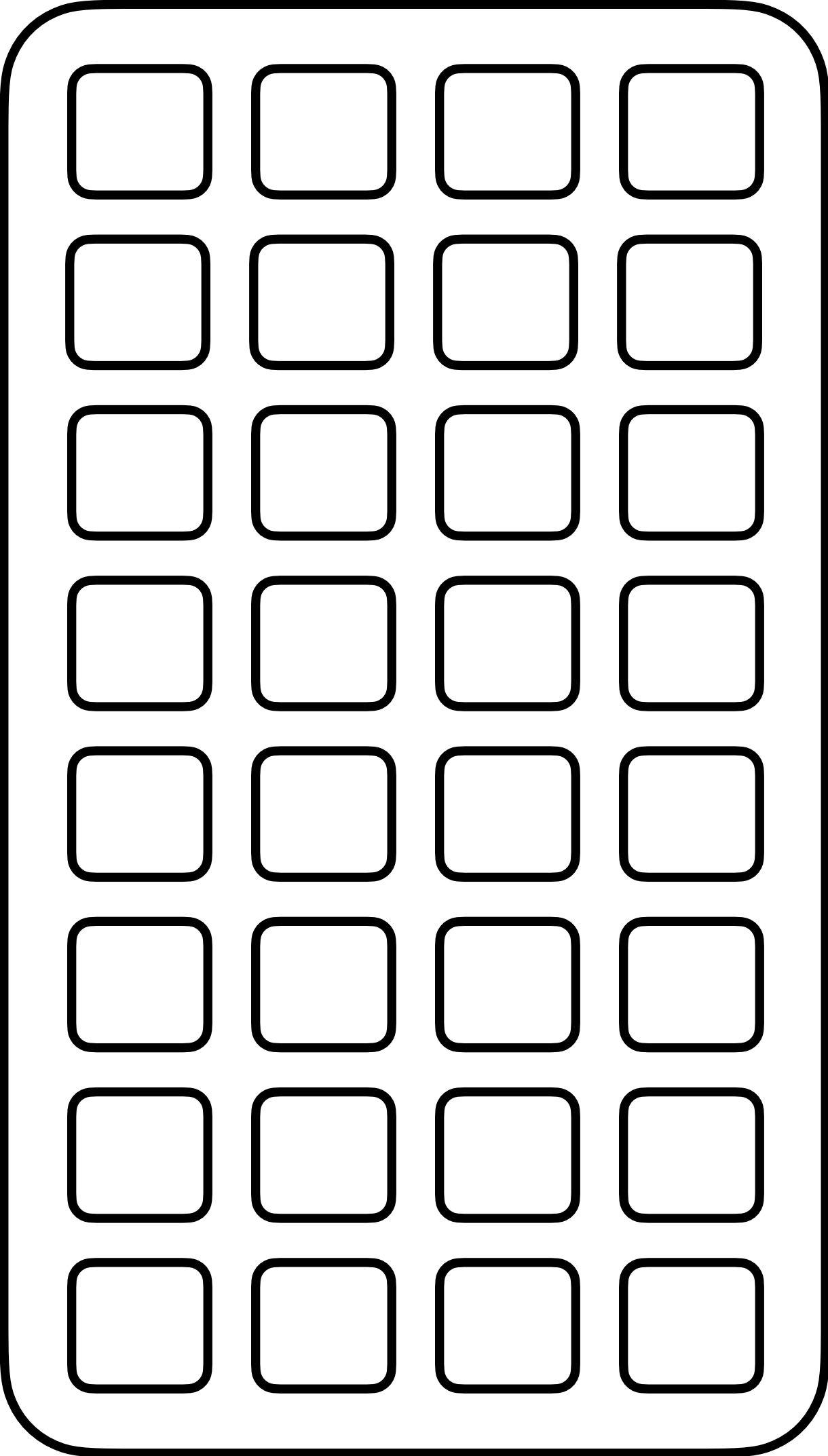
[illegible]

# Cache



# Cache

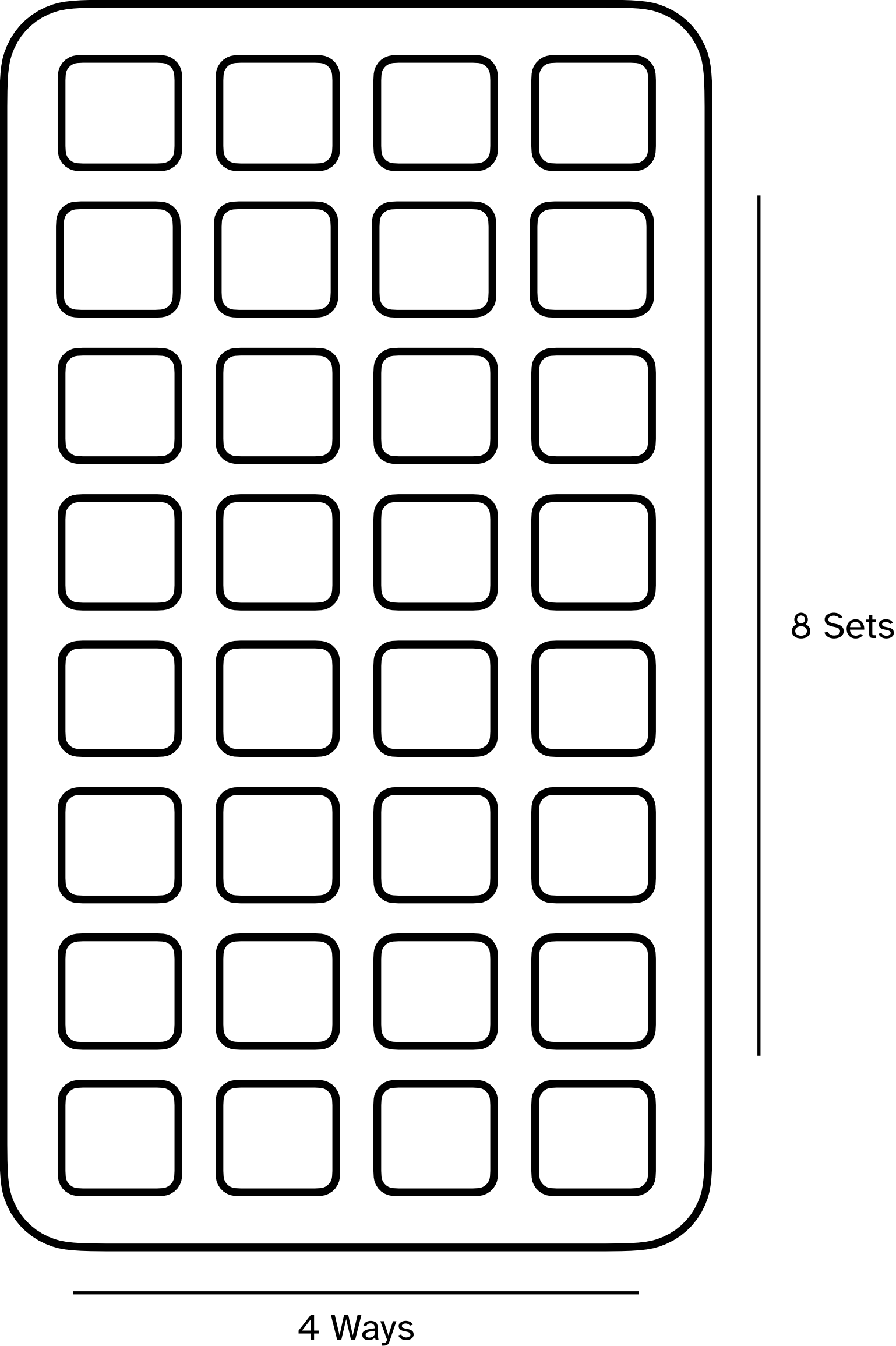
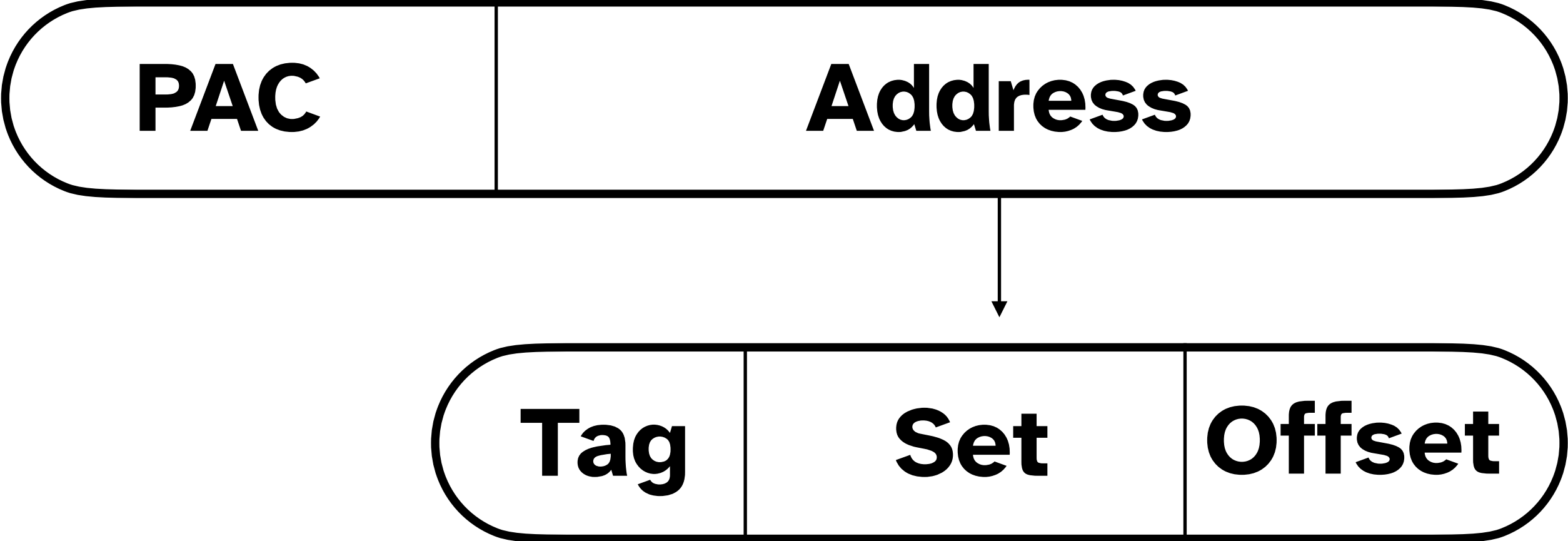
|     |         |
|-----|---------|
| PAC | Address |
|-----|---------|



4 Ways

8 Sets

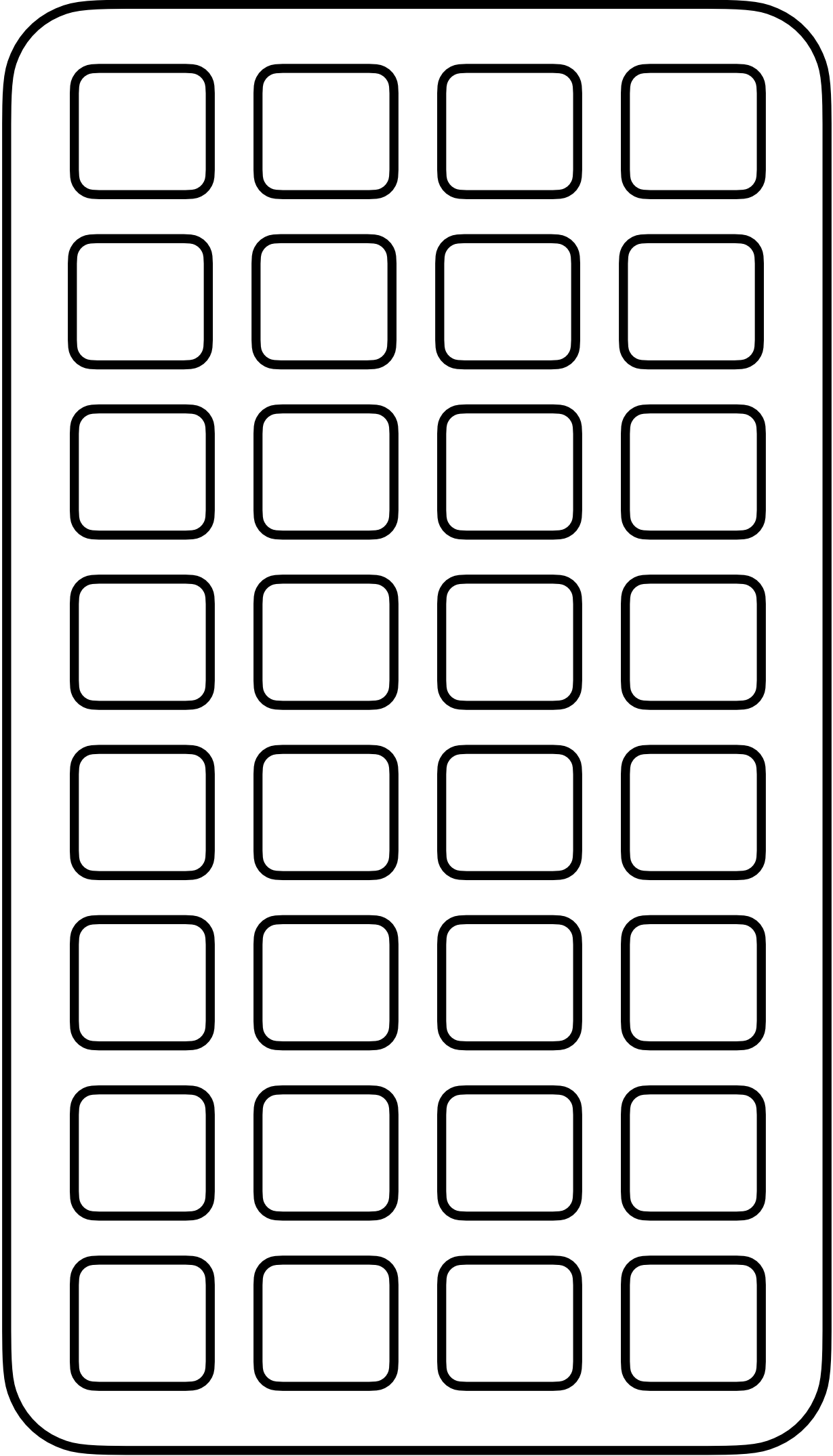
# Cache



# Cache



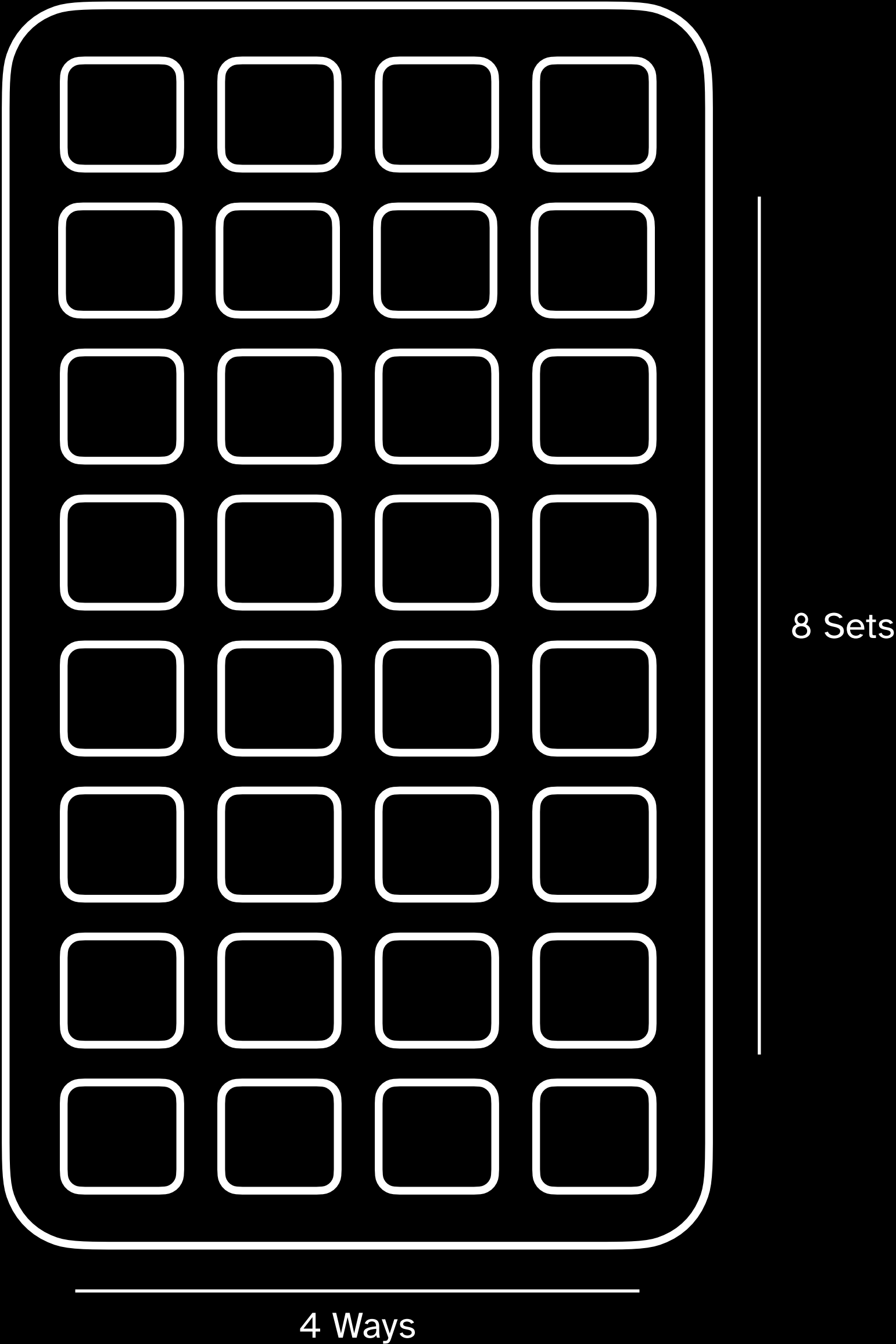
Which set do we map to?



8 Sets

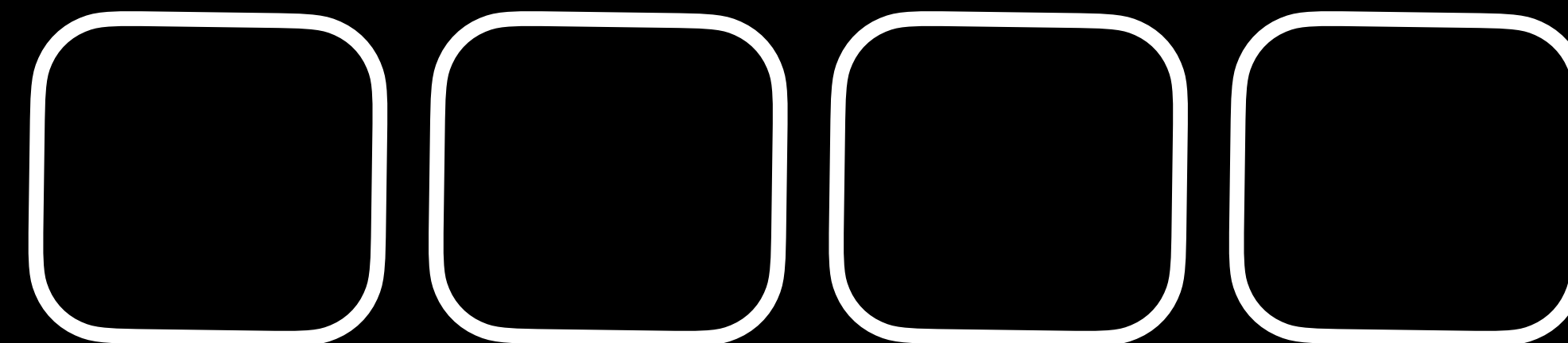
4 Ways

**Think like an attacker...**

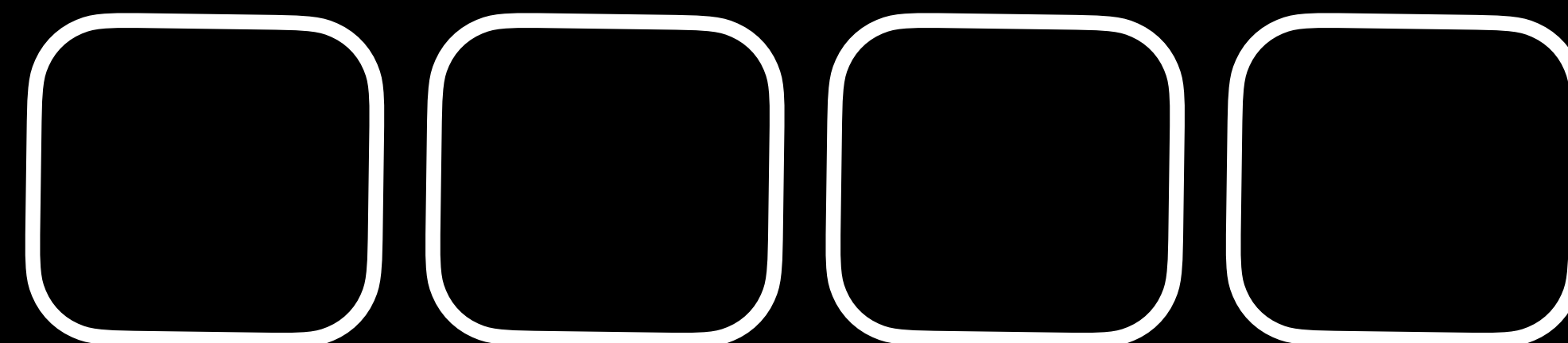




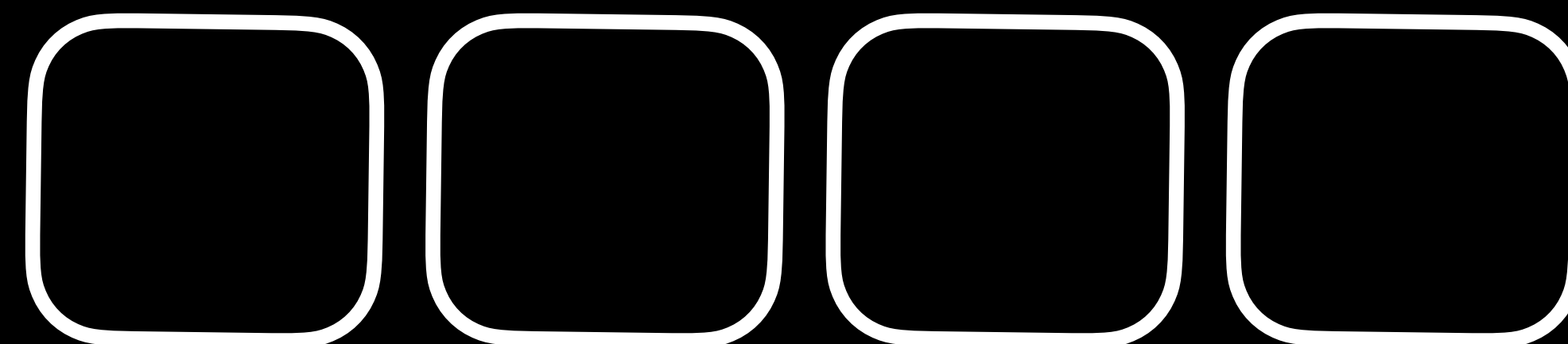
|     |      |        |
|-----|------|--------|
| Tag | 0000 | Offset |
|-----|------|--------|



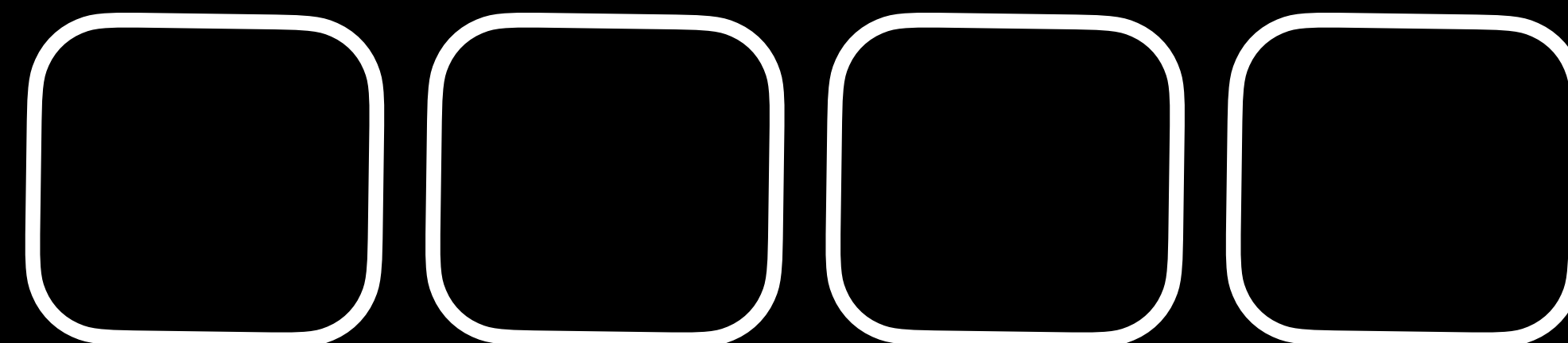
|     |      |        |
|-----|------|--------|
| Tag | 0001 | Offset |
|-----|------|--------|



|     |      |        |
|-----|------|--------|
| Tag | 0010 | Offset |
|-----|------|--------|

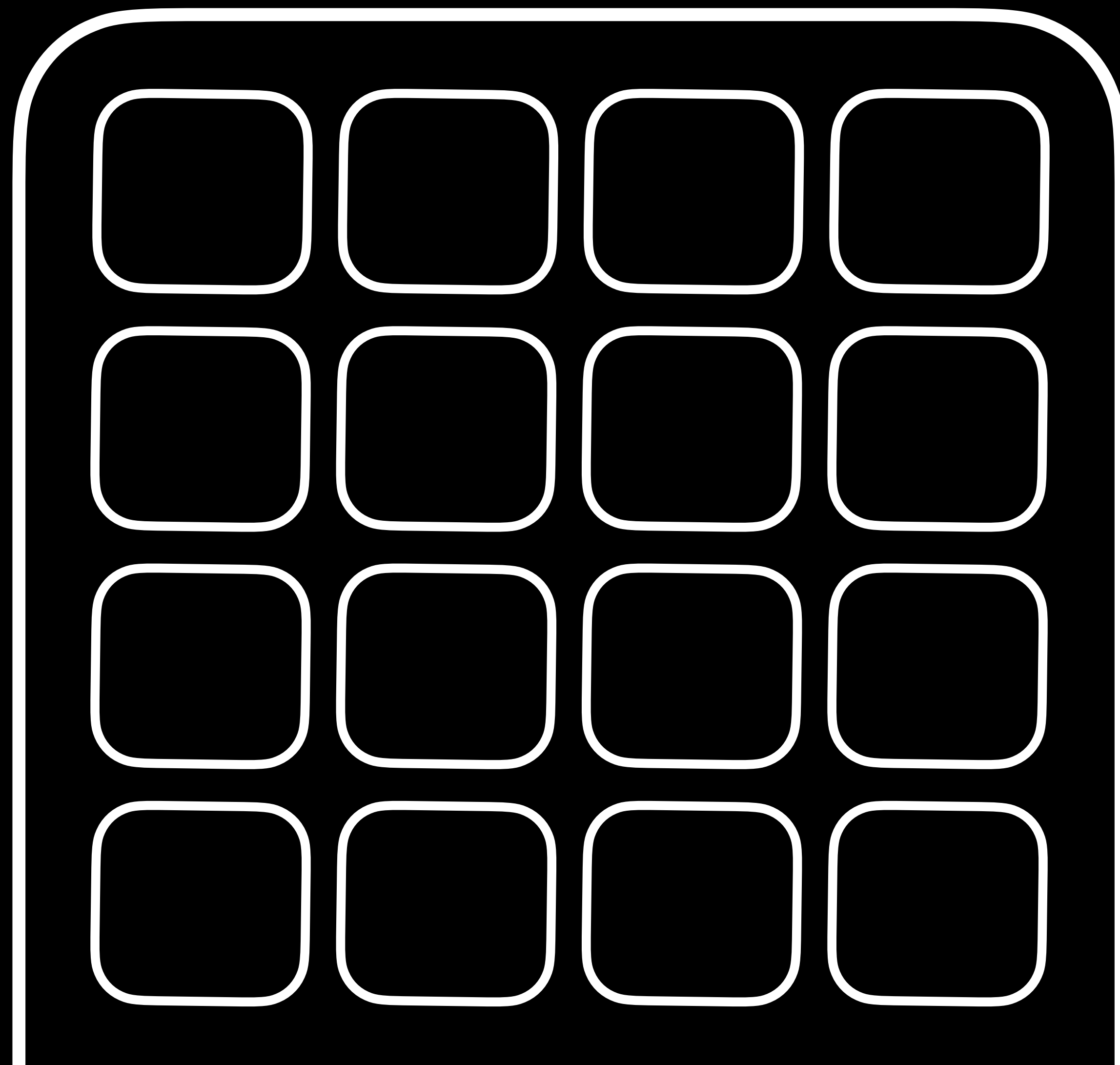


|     |      |        |
|-----|------|--------|
| Tag | 0011 | Offset |
|-----|------|--------|

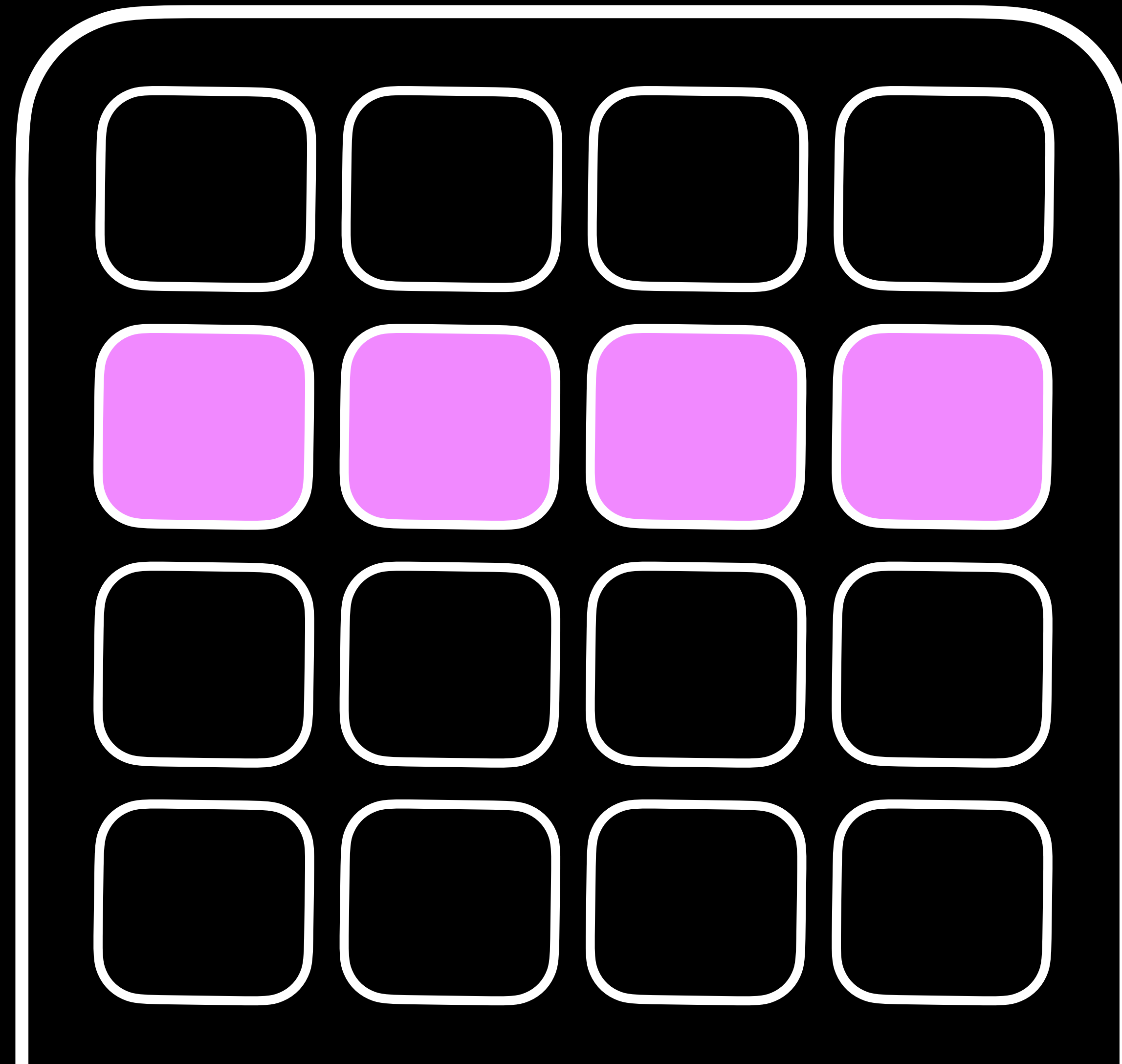
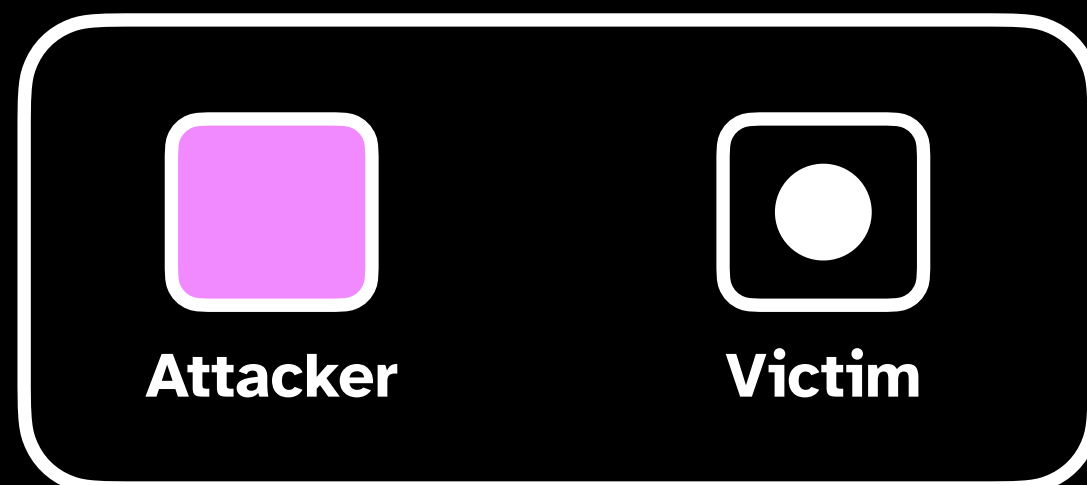


Attacker

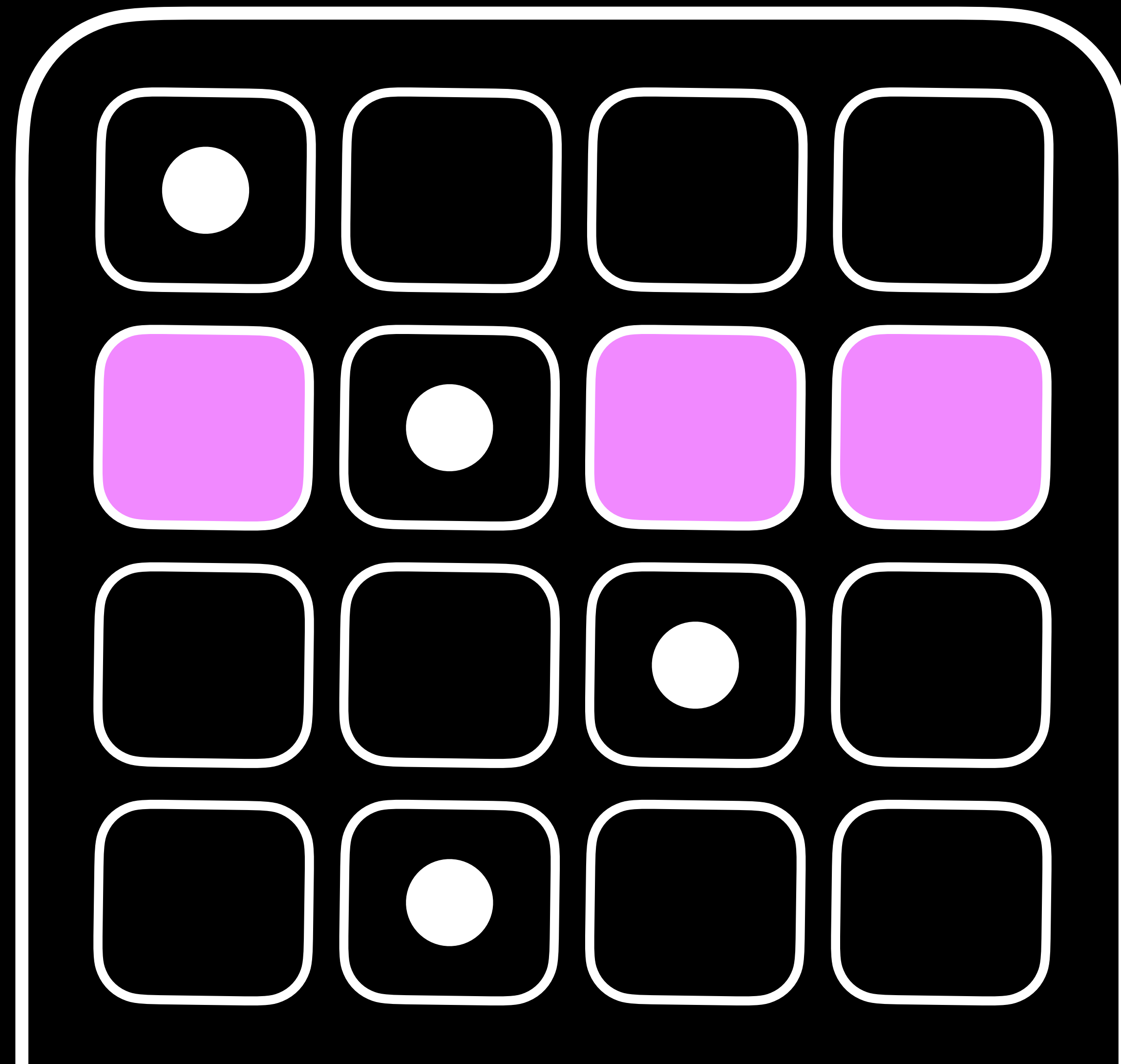
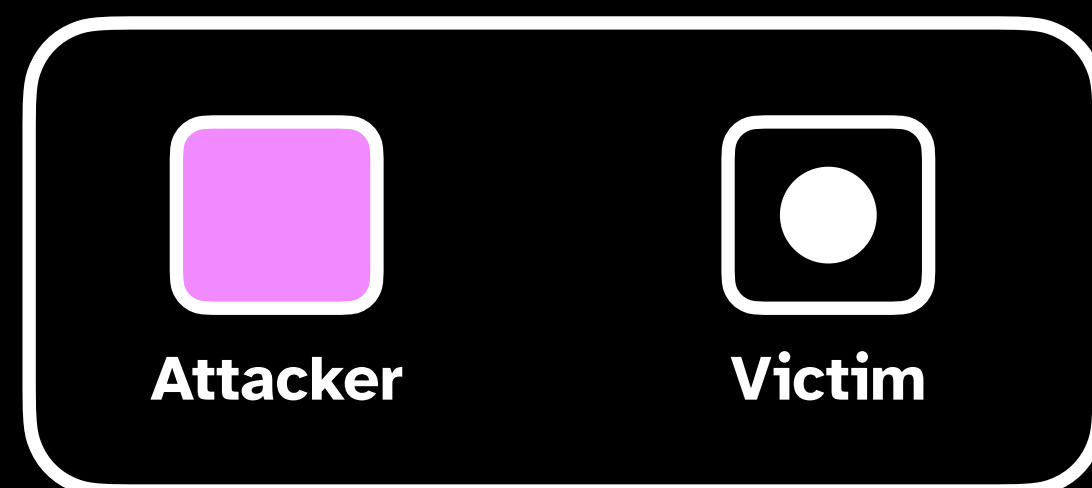
Victim



**1** Fill a set with our data.

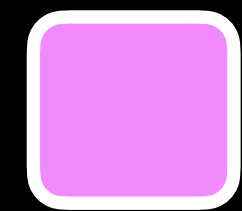


## 2 Let the victim run.

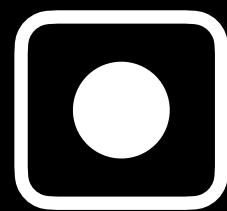


### 3 Re-access our data.

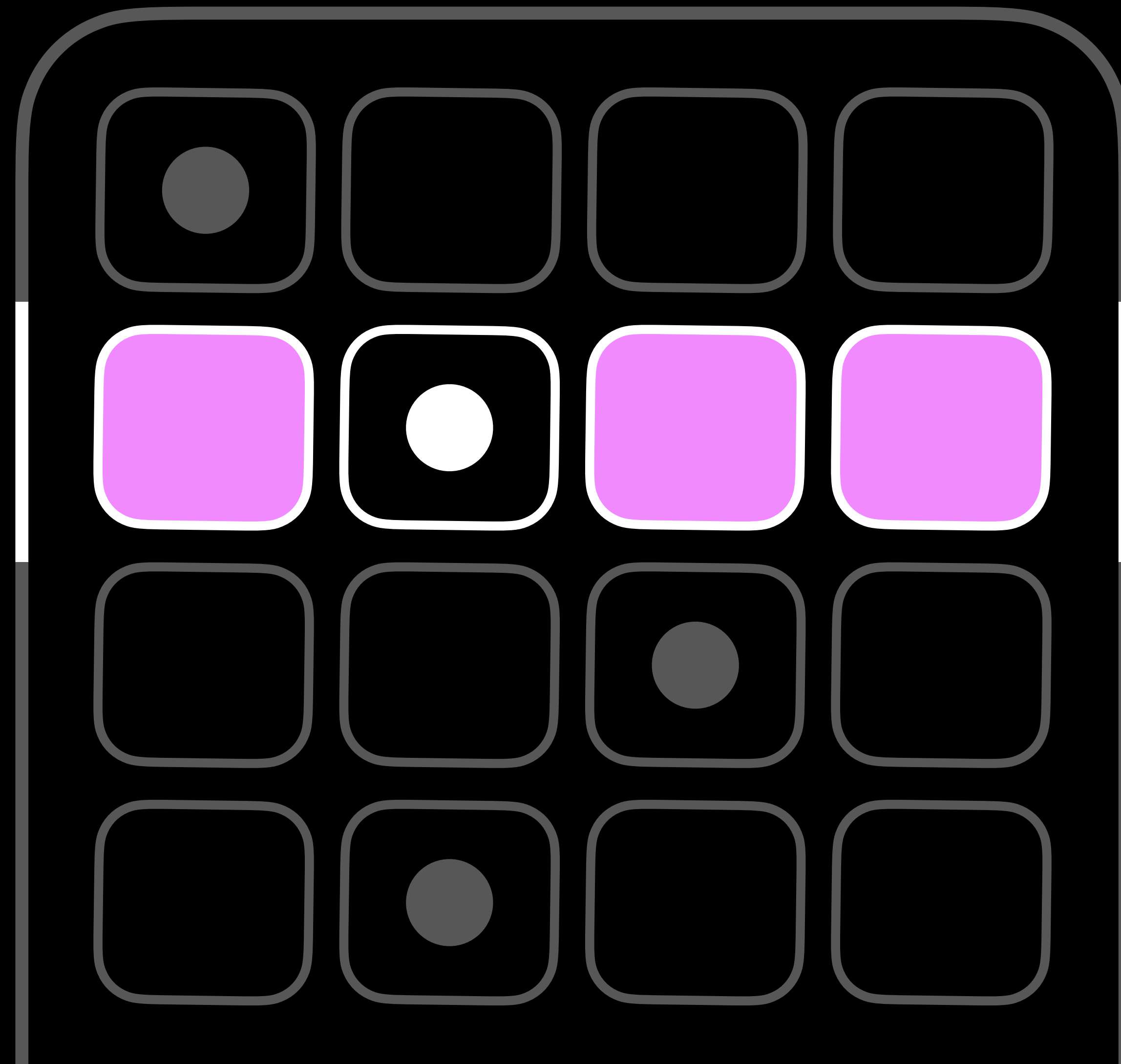
We can tell what the victim did by just watching the cache!



Attacker

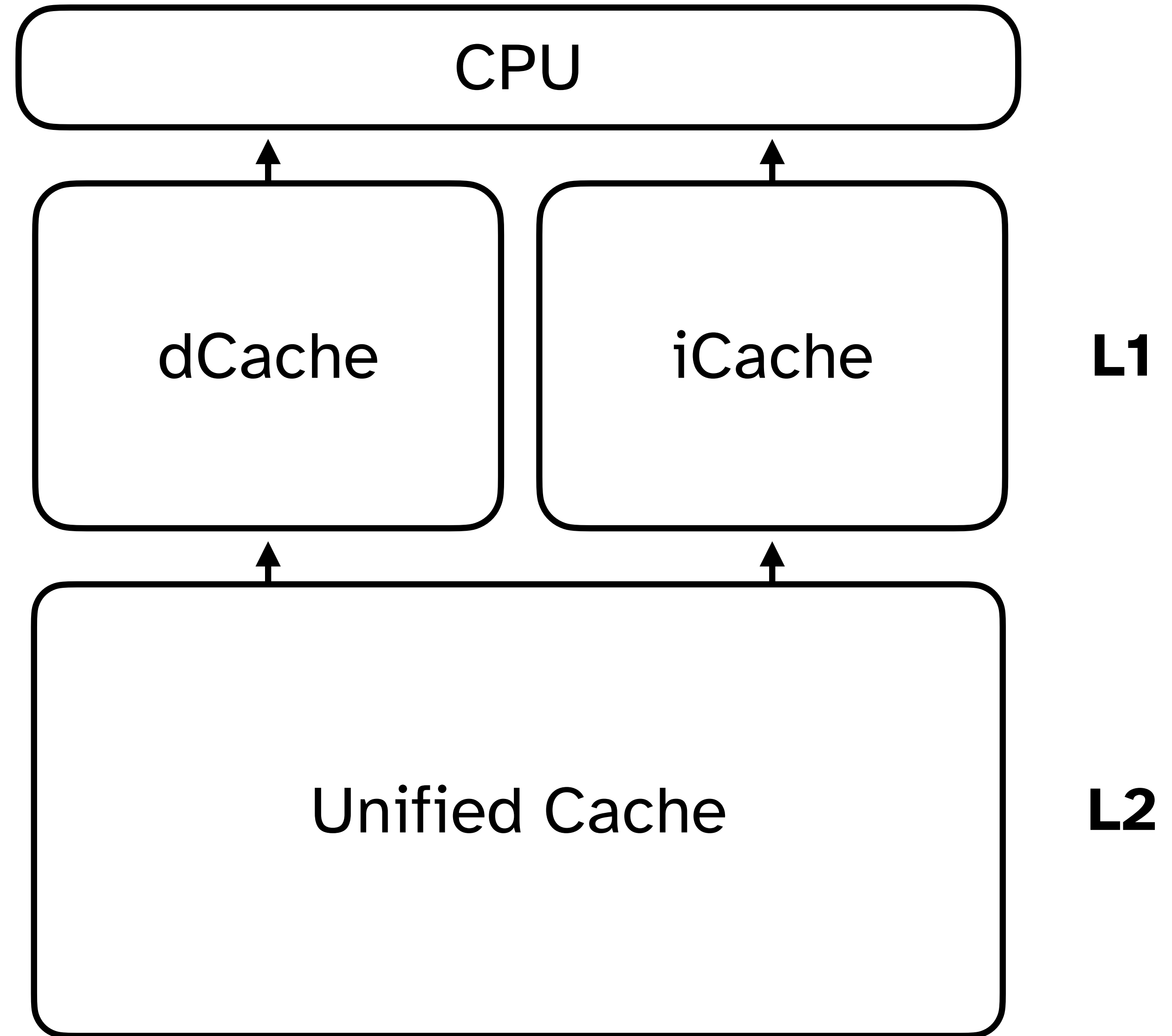


Victim



1 line is missing!

Our CPU needs to access  
instructions and data at  
the same time!



# M1 High Performance Cores

## **L1D**

8 ways, 256 sets  
64 byte lines

## **L1I**

6 ways, 512 sets  
64 byte lines

## **L2**

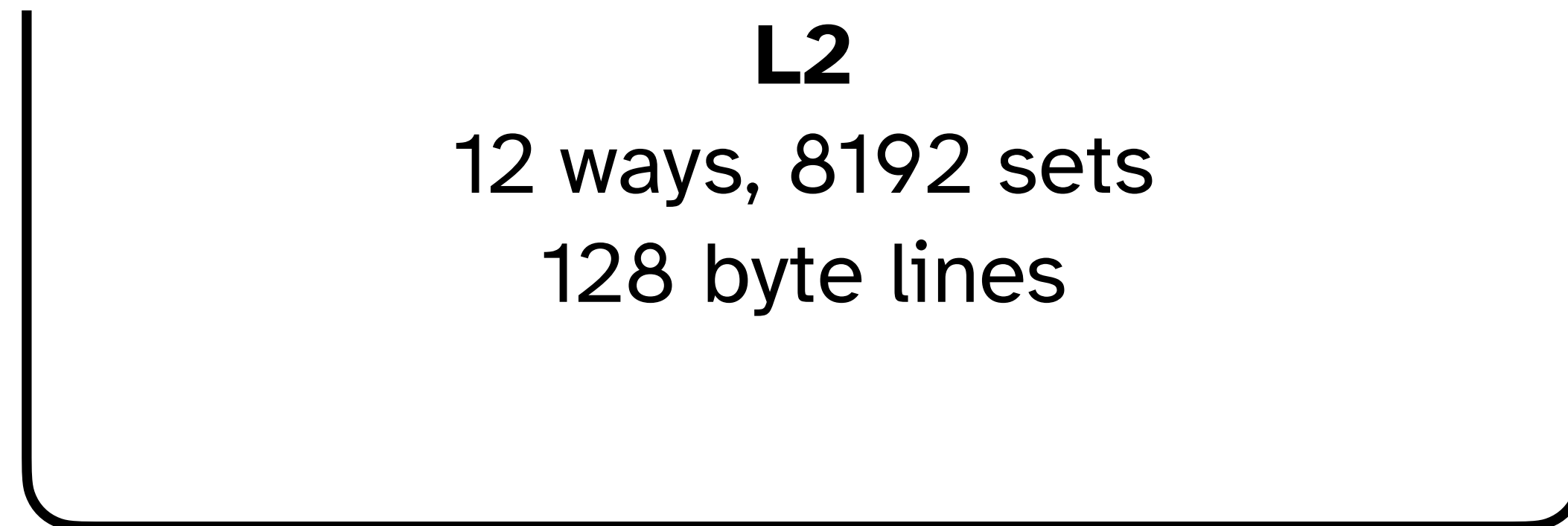
12 ways, 8192 sets  
128 byte lines

# M1 High Performance Cores



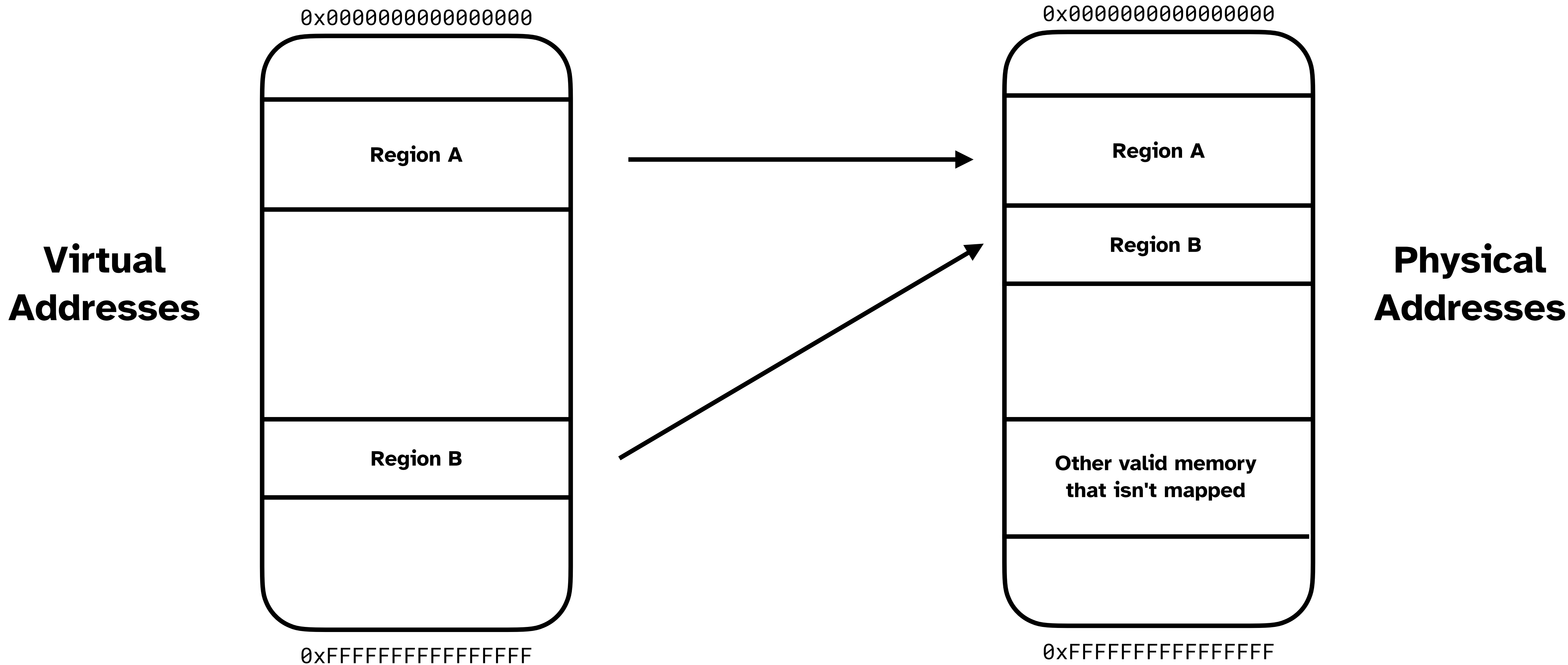
## Obtaining cache geometry parameters at runtime

Although not often, the kernel may still require deriving, one way or another, parameters like cache sizes and number of set/ways. XNU needs most of this information to perform set/way clean/invalidate operations. Prior to this work, these values were hardcoded for each supported target in `proc_reg.h`, and used in `caches_asm.s`. The ARM architecture provides the `CCSIDR_EL1` register, which can be used in conjunction with `CSSELR_EL1` to select the target cache and obtain geometry information.

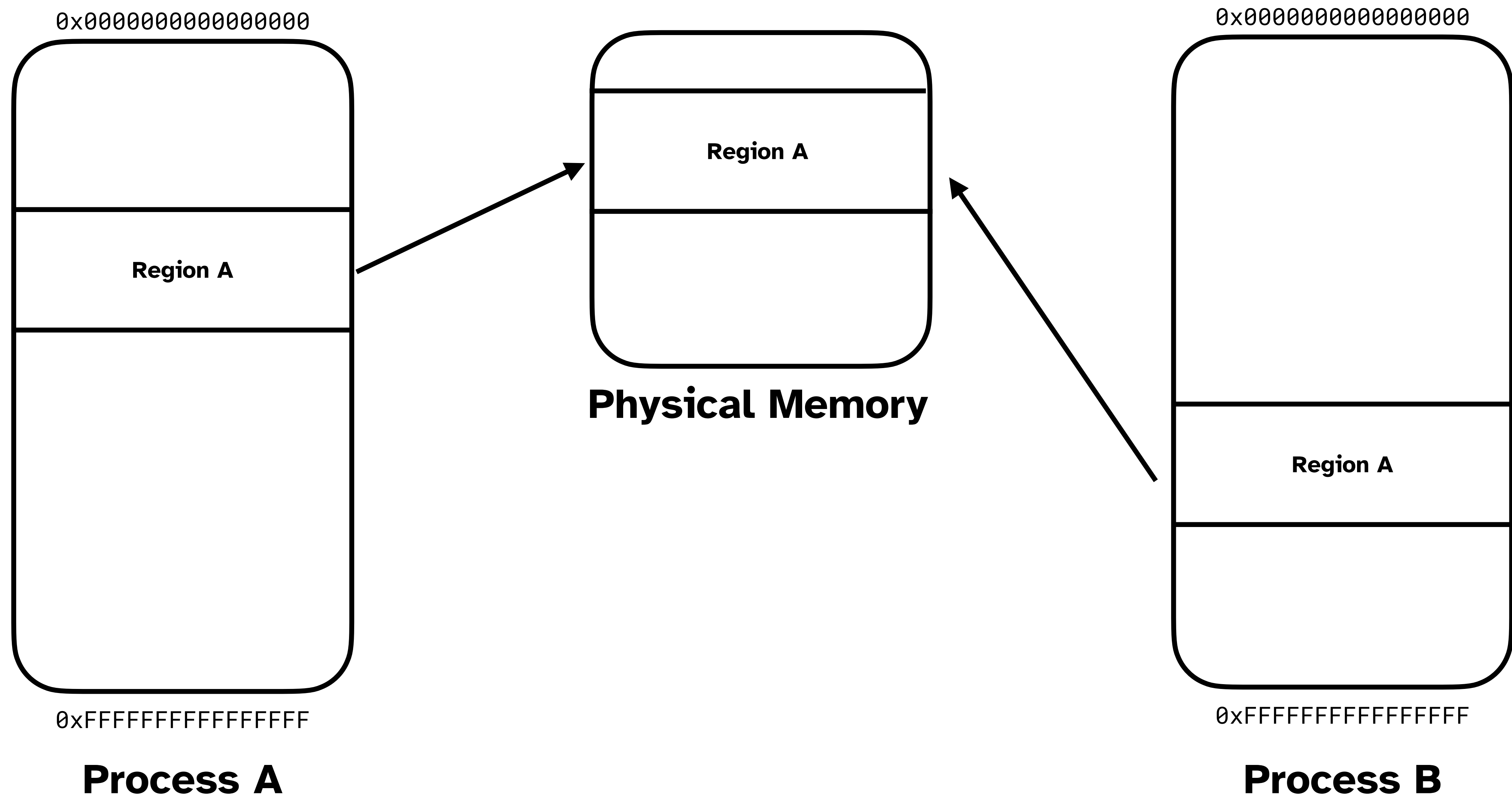




# Virtual Memory



# Multiple Processes can share memory



# Pages

A page is the smallest translation grouping.

All virtual addresses in a page go to the same physical page.



Pages on M1 are 16KB

# Translation Lookaside Buffer (TLB)

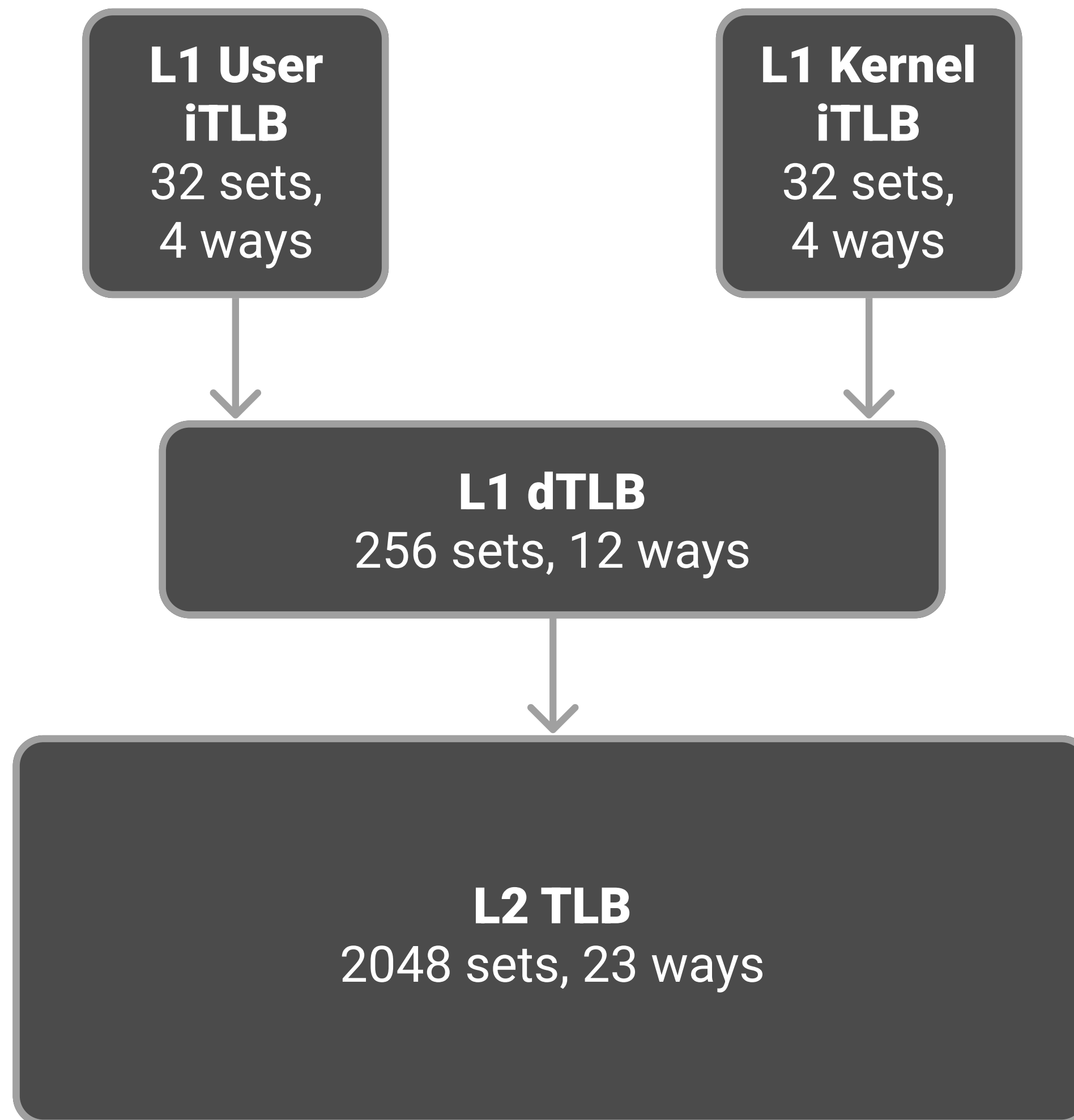
| Virtual Page | Physical Page |
|--------------|---------------|
| 0x1000000000 | 0x2000000000  |
| 0x100004000  | 0x3000000000  |
| 0x100008000  | 0x4000000000  |
| 0x10000C000  | 0x5000000000  |

**Virtual Page**



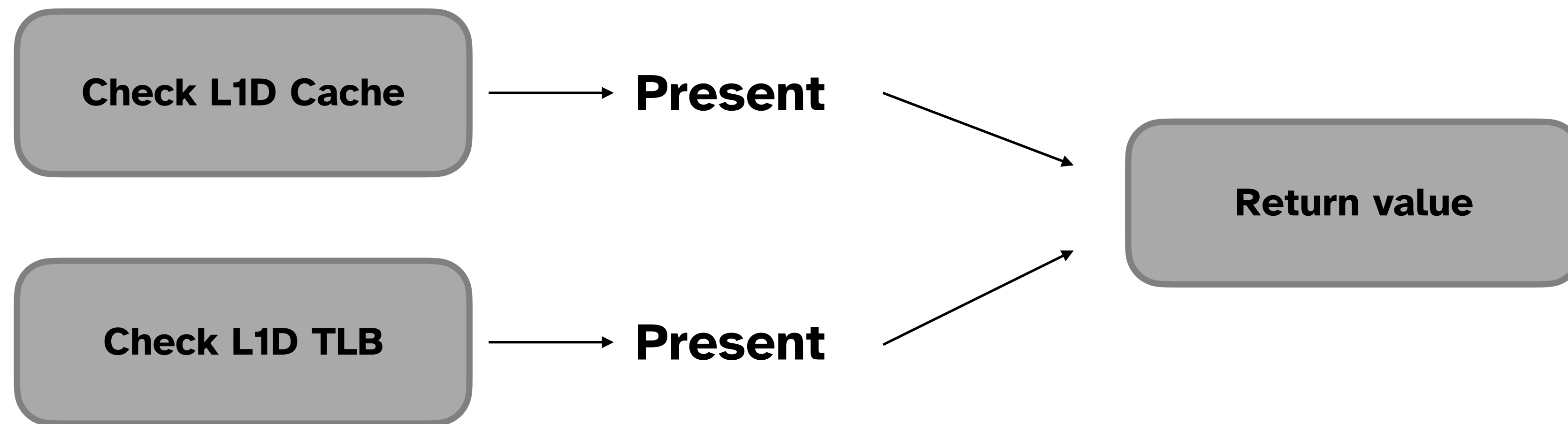
**Physical Page**

# Conjectured TLB Hierarchy



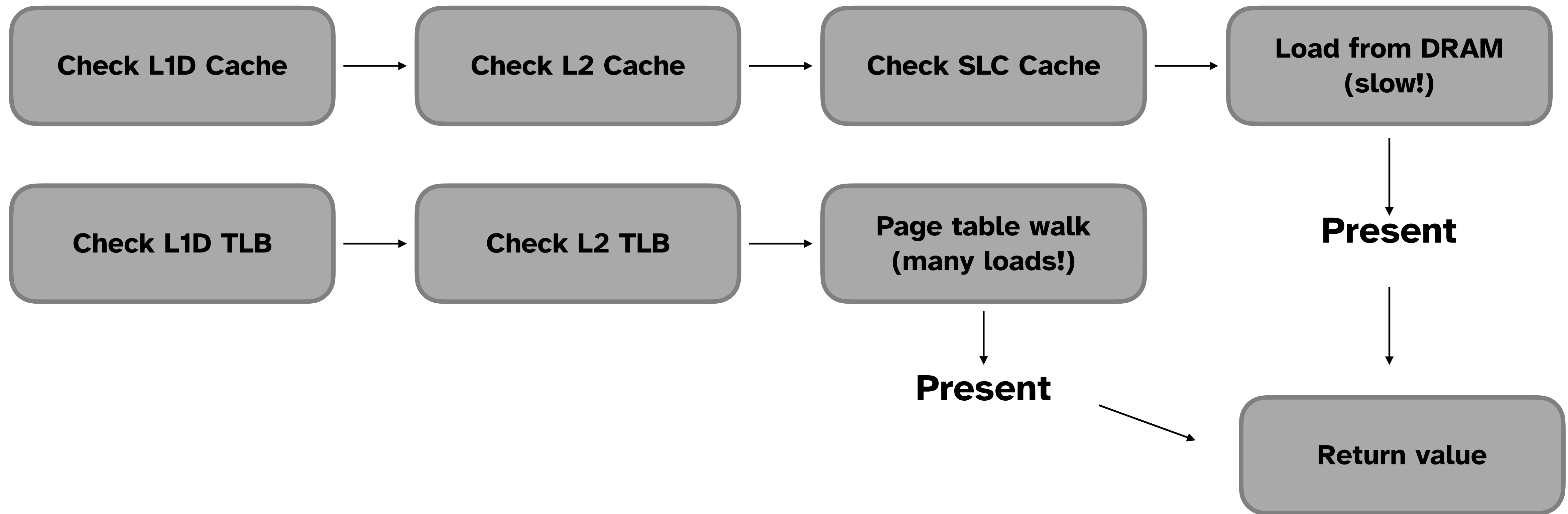
# What happens when we dereference a pointer?

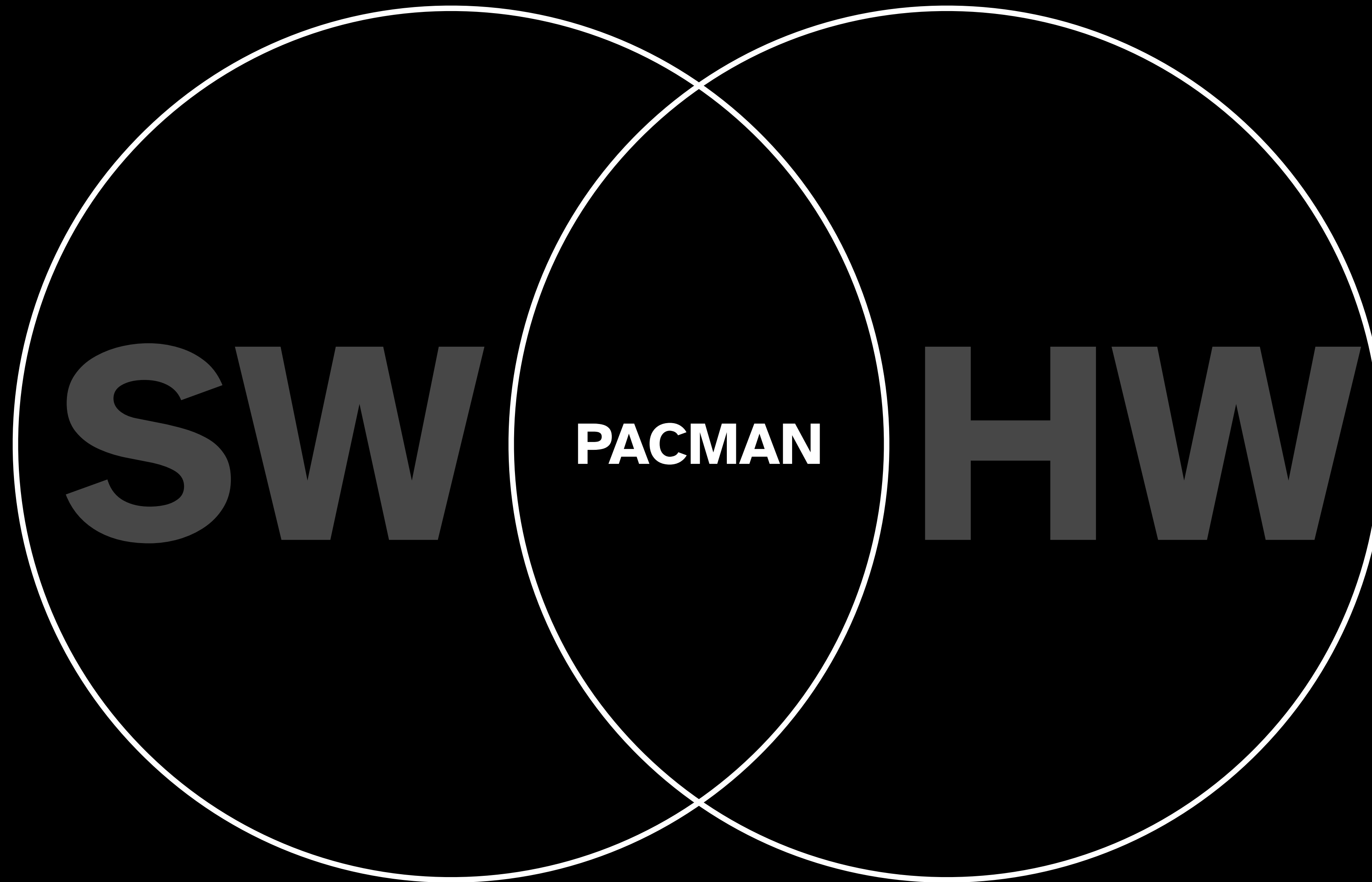
**Best case**



# What happens when we dereference a pointer?

**Worst case**







# BYOB

*"Bring your own bug"*

# BYOB

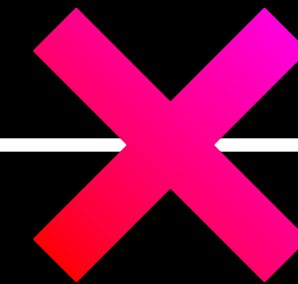
*"Bring your own bug"*

**Pointer Authentication  
blocks changing pointers**

**Read/ Write  
Memory**



**Change Function  
Pointer**



**Arbitrary Code  
Execution**

# BYOB

*"Bring your own bug"*

**Pointer Authentication**  
**blocks changing pointers**

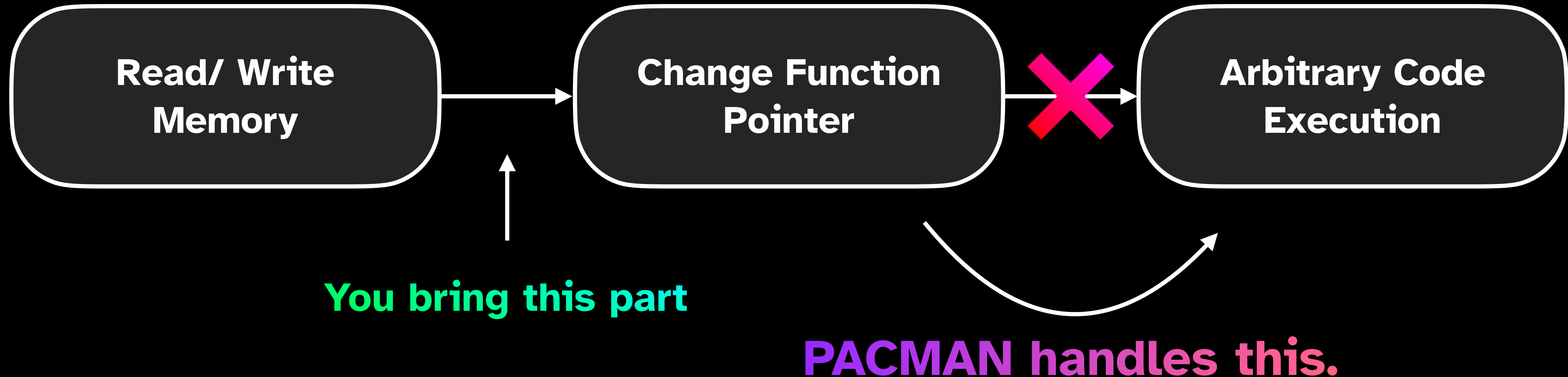


**You bring this part**

# BYOB

*"Bring your own bug"*

**Pointer Authentication  
blocks changing pointers**



**How should we speculate on  
PAC values?**

# **3 cases**

**Just ignore PACs  
(always load)**

**Never load**

**Treat them  
normally**

# 3 cases

**Just ignore PACs  
(always load)**



**Leads to other  
security issues!**

**Never load**



**Slow!**

**Treat them  
normally**



**This is how  
M1 does it.**

# PACMAN Gadget

Speculative check & use of a signed pointer.

```
if (condition):  
    check_pac(ptr)  
    load(checked_ptr)
```



# Data Gadget

```
if (condition):  
    verified_ptr = check_pac(guess_ptr)  
    load(verified_ptr)
```

# Data Attack

```
if (condition):  
    verified_ptr = check_pac(guess_ptr)  
    load(verified_ptr)
```

Correct PAC

**Mispredict  
Branch**



# Data Attack

```
if (condition):  
    verified_ptr = check_pac(guess_ptr)  
    load(verified_ptr)
```

Correct PAC

**Mispredict  
Branch**



**PAC Check  
Succeeds**

# Data Attack

```
if (condition):  
    verified_ptr = check_pac(guess_ptr)  
    load(verified_ptr)
```

Correct PAC

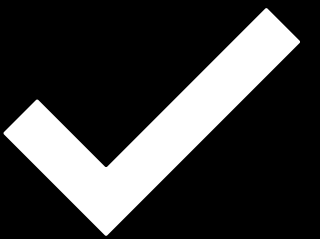
Mispredict  
Branch



PAC Check  
Succeeds



Speculative  
Load!



# Data Attack

```
if (condition):  
    verified_ptr = check_pac(guess_ptr)  
    load(verified_ptr)
```

Correct PAC

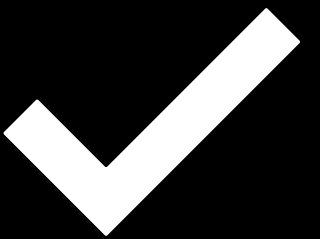
Mispredict  
Branch



PAC Check  
Succeeds



Speculative  
Load!



Incorrect PAC

Mispredict  
Branch



# Data Attack

```
if (condition):  
    verified_ptr = check_pac(guess_ptr)  
    load(verified_ptr)
```

Correct PAC

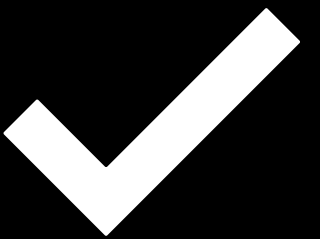
Mispredict  
Branch



PAC Check  
Succeeds



Speculative  
Load!



Incorrect PAC

Mispredict  
Branch



PAC Check  
Fails

# Data Attack

```
if (condition):  
    verified_ptr = check_pac(guess_ptr)  
    load(verified_ptr)
```

Correct PAC

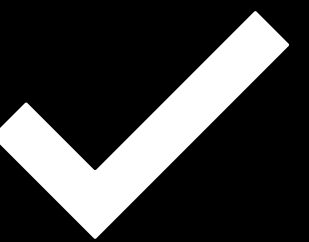
Mispredict  
Branch



PAC Check  
Succeeds



Speculative  
Load!



Incorrect PAC

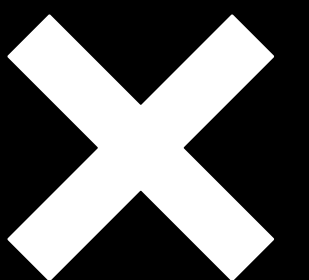
Mispredict  
Branch



PAC Check  
Fails



Speculative  
Exception



# Instruction Gadget

```
if (condition): #BR1  
    verified_ptr = check_pac(guess_ptr)  
    call(verified_ptr) #BR2
```



# Instruction Attack

```
if (condition): #BR1
    verified_ptr = check_pac(guess_ptr)
    call(verified_ptr) #BR2
```

Correct  
PAC

Mispredict  
BR1



PAC Check  
Succeeds



Mispredict  
BR2



Eager  
Squash



Speculative  
Load! ✓

Incorrect  
PAC

Mispredict  
BR1



PAC Check  
Fails



Mispredict  
BR2



Eager  
Squash



Speculative  
Exception ✗

# Bird's Eye View



# Requirements

- Kernel panic on **use** of incorrect pointer
- Need to train branch predictor on correctly signed pointer
- Need a destination pointer in a low-noise cache line/ page
- Must precisely construct an **eviction set**...

# PacmanGhidra

- Static analysis tool for finding PACMAN Gadgets
- Evaluates 32 instructions either direction of a branch
- Looks for any auth -> use patterns
- We are releasing this tool too!

# XNU-8019.80.24 PacmanGhidra Analysis

| Data<br>Gadgets | Instruction<br>Gadgets | Total  |
|-----------------|------------------------|--------|
| 13,867          | 41,292                 | 55,159 |

This list is not exhaustive, and no reachability analysis was performed.



# Reverse Engineering M1

# Timer Sources on M1

|                                   | Speed                      | Available to users? |
|-----------------------------------|----------------------------|---------------------|
| <b>System Counter</b>             | Slow (24 MHz)              | Yes                 |
| <b>ARM PMU</b>                    | NA                         | NA                  |
| <b>Apple Custom Cycle Counter</b> | Very Fast (Cycle Accurate) | No                  |
| <b>Multi-threaded Counter</b>     | "Good Enough"              | Yes                 |

# Timer Sources on M1

|                            | Speed                      | Available to users? |
|----------------------------|----------------------------|---------------------|
| System Counter             | Slow (24 MHz)              | Yes                 |
| ARM PMU                    | NA                         | NA                  |
| Apple Custom Cycle Counter | Very Fast (Cycle Accurate) | No                  |
| Multi-threaded Counter     | "Good Enough"              | Yes                 |

**PacmanPatcher**

Can enable the Apple cycle counter!  
We are releasing this tool as well.



# Timing with evict+reload

1

Load the address we  
want to measure

2

Load a bunch of  
addresses that might  
evict it

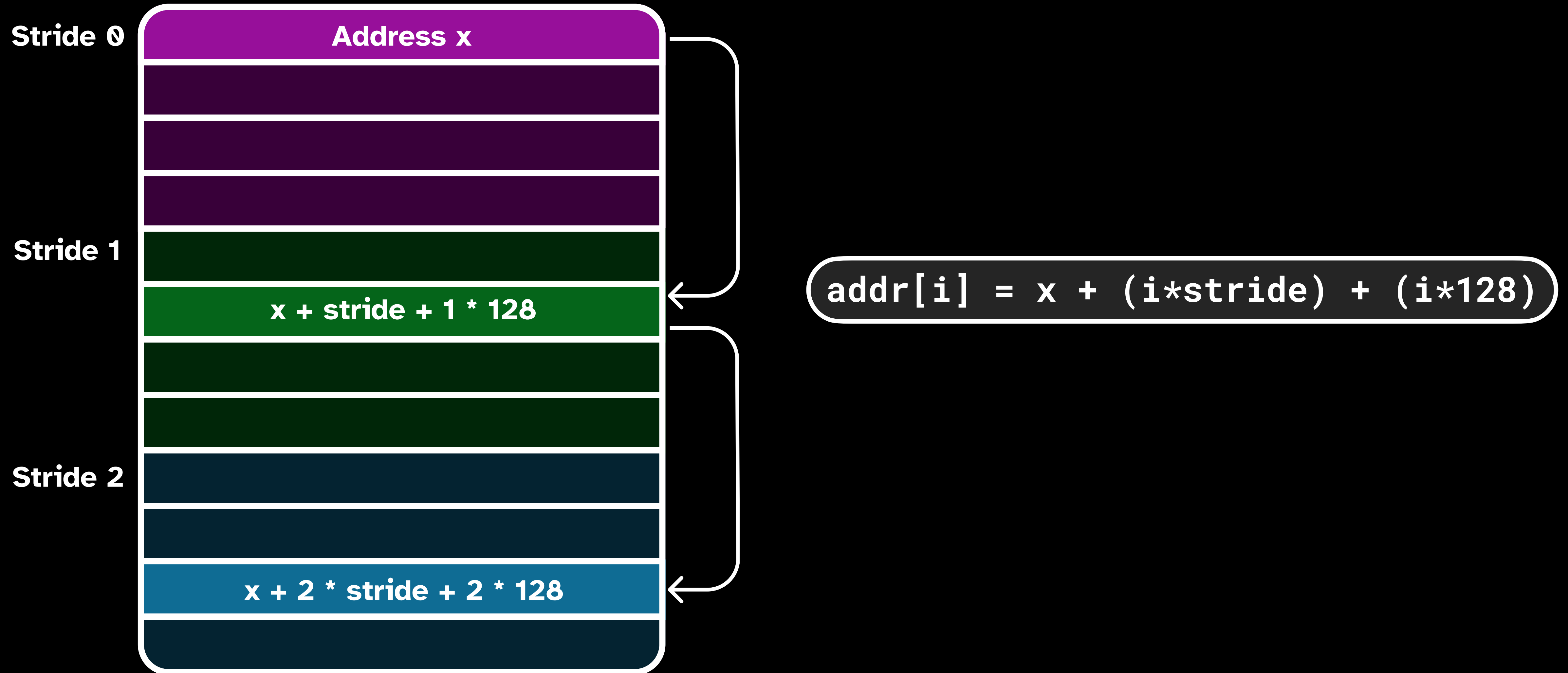
3

Check if we evicted  
it by **reloading**

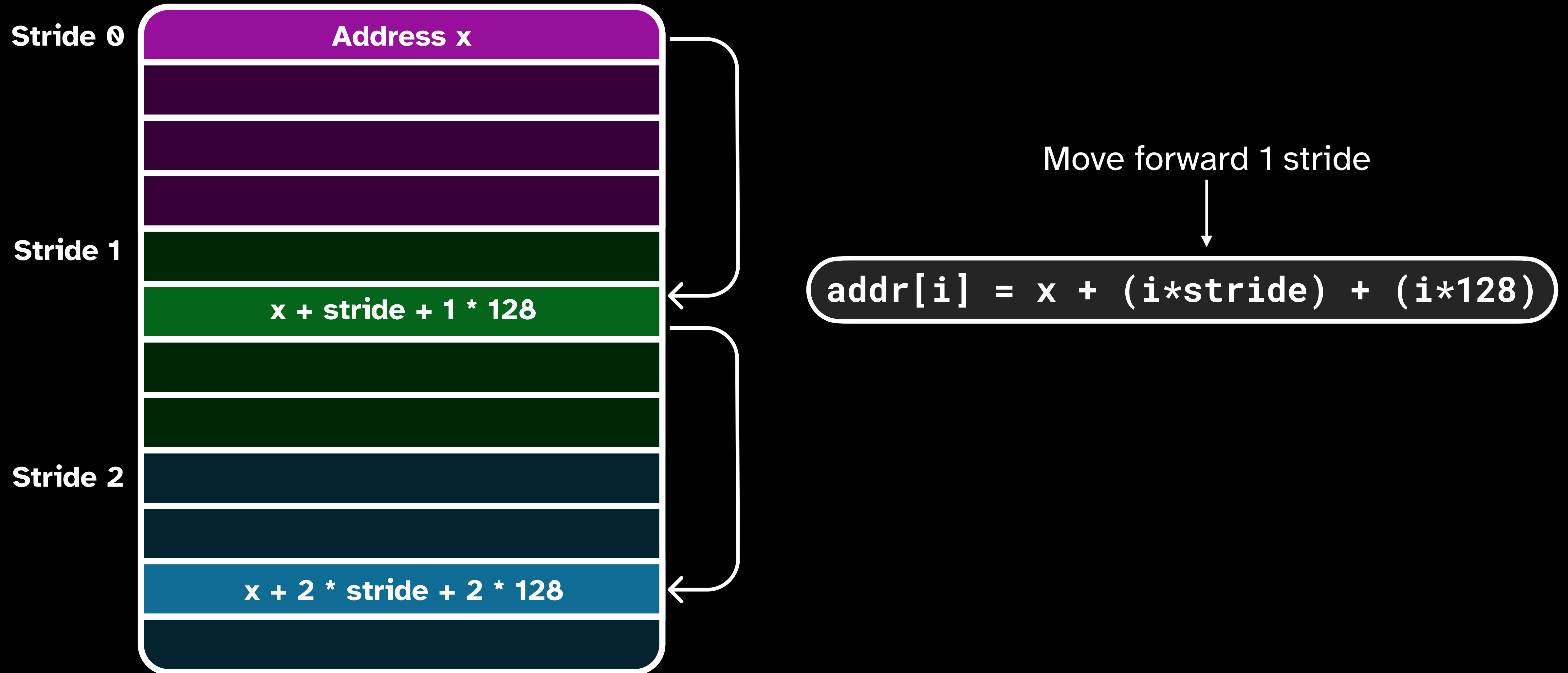
# **Experiment 1**

## **Data TLB Behavior**

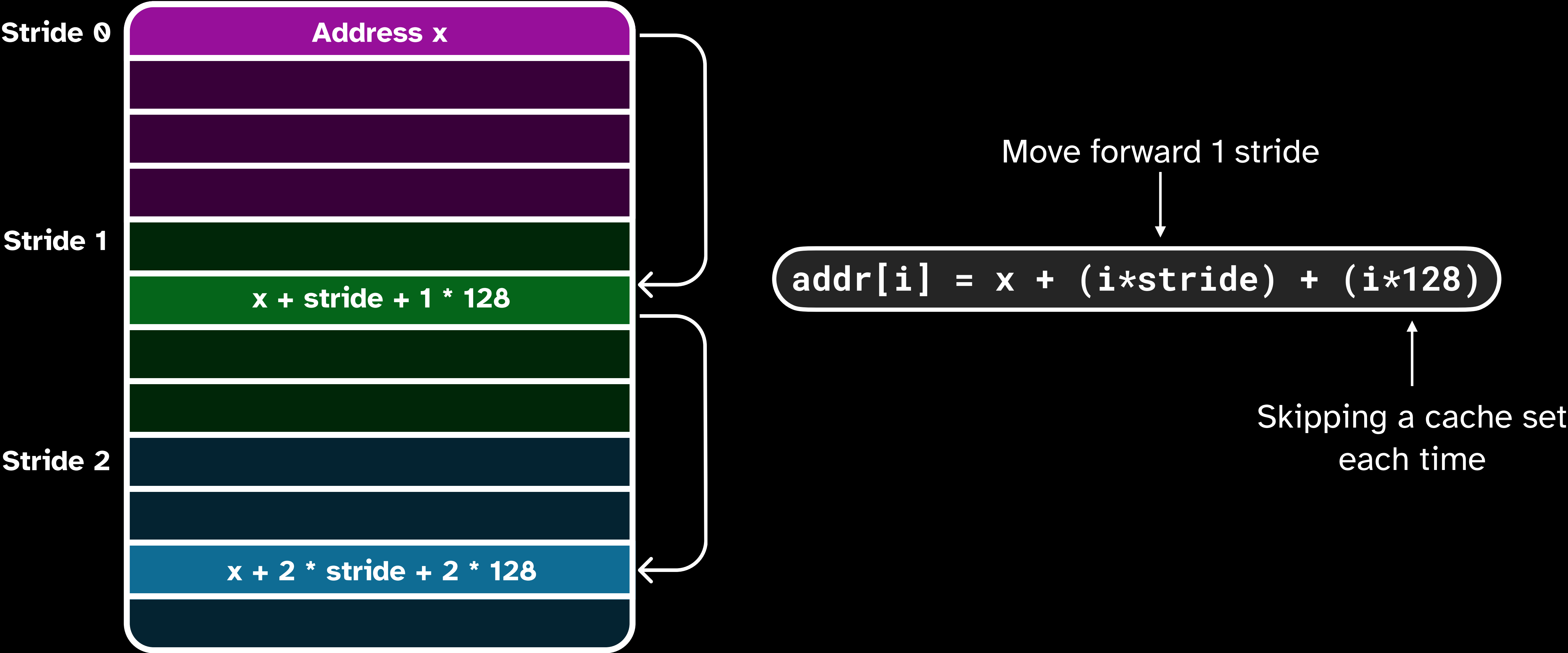
# Reversing the Data TLB



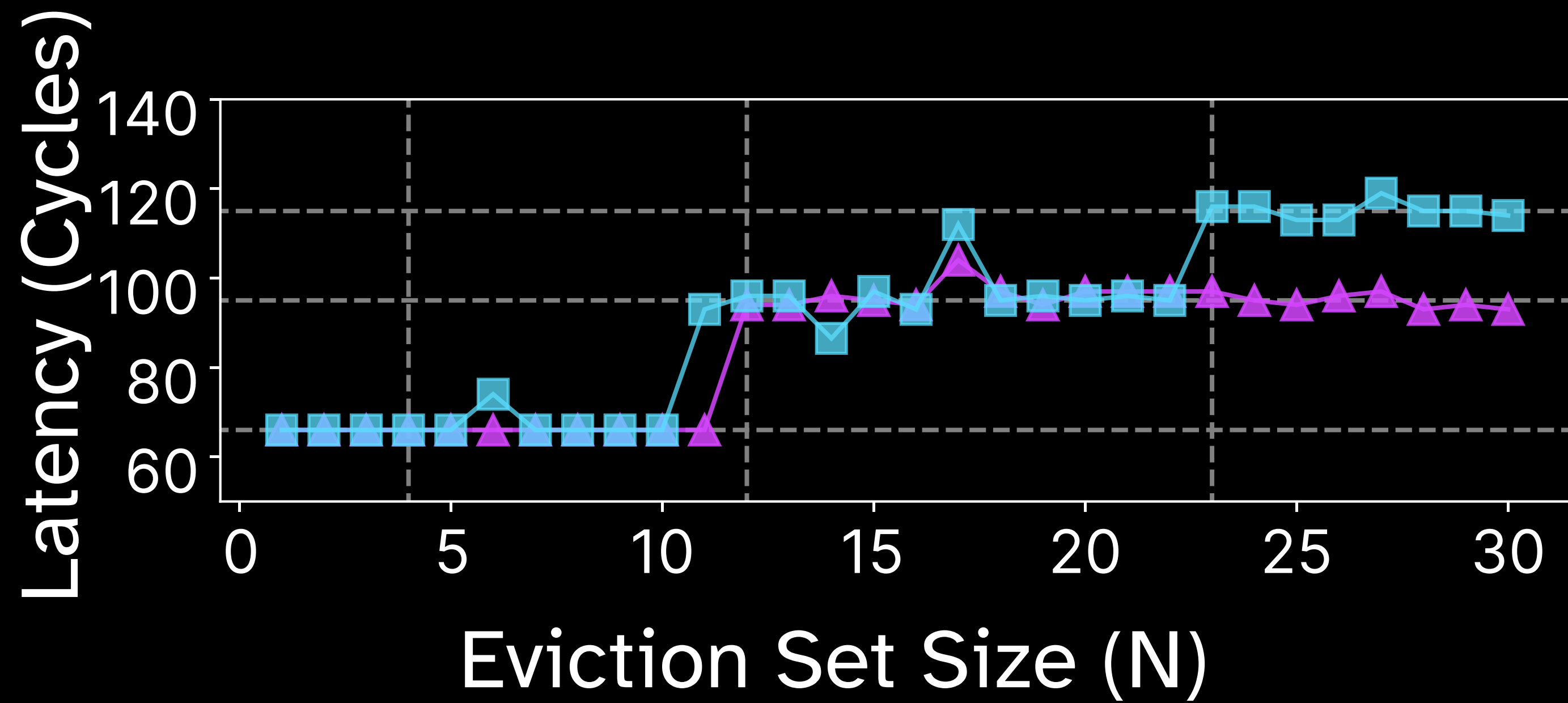
# Reversing the Data TLB



# Reversing the Data TLB



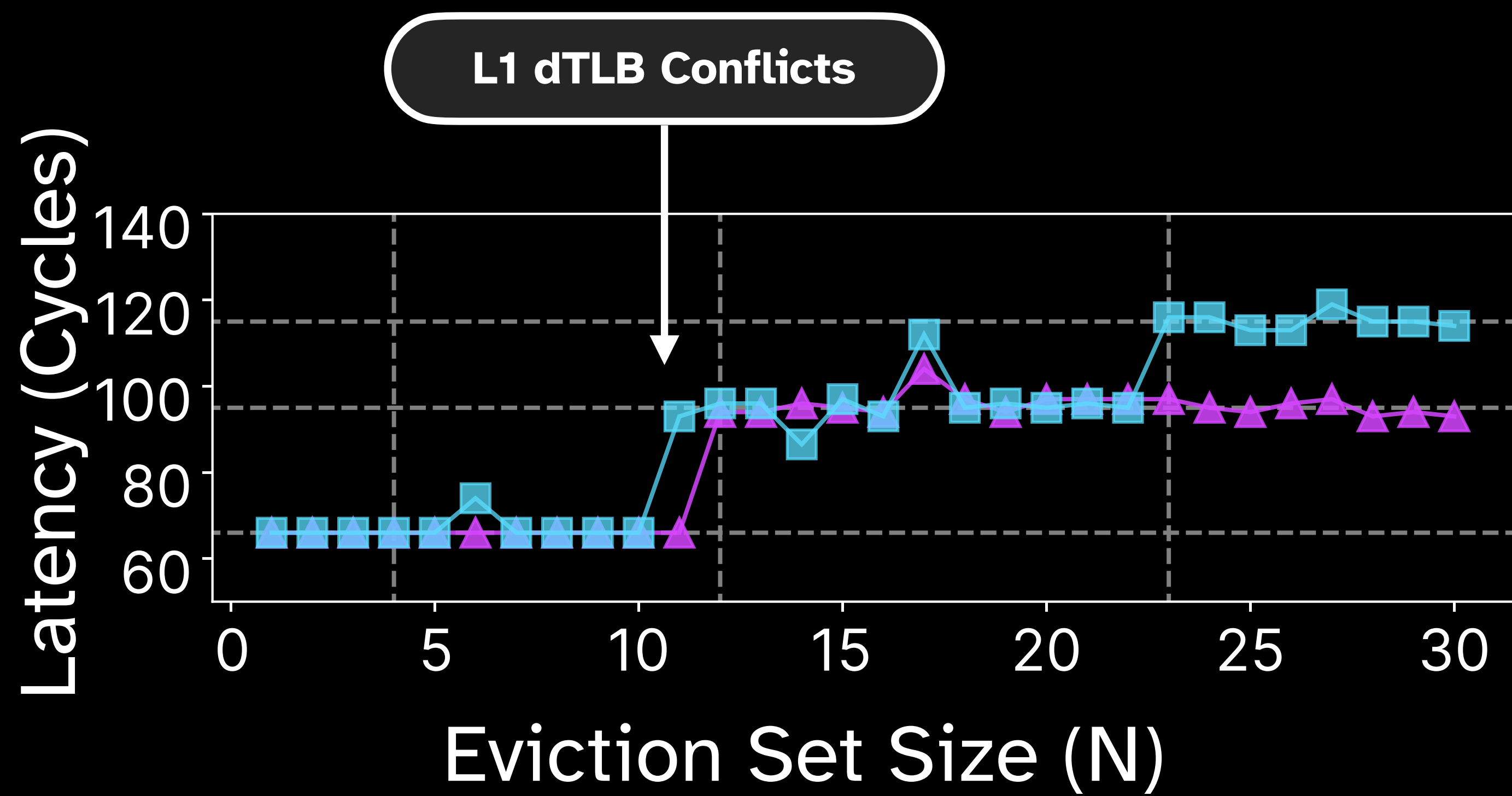
# Data TLB Results



Latency from dTLB conflicts



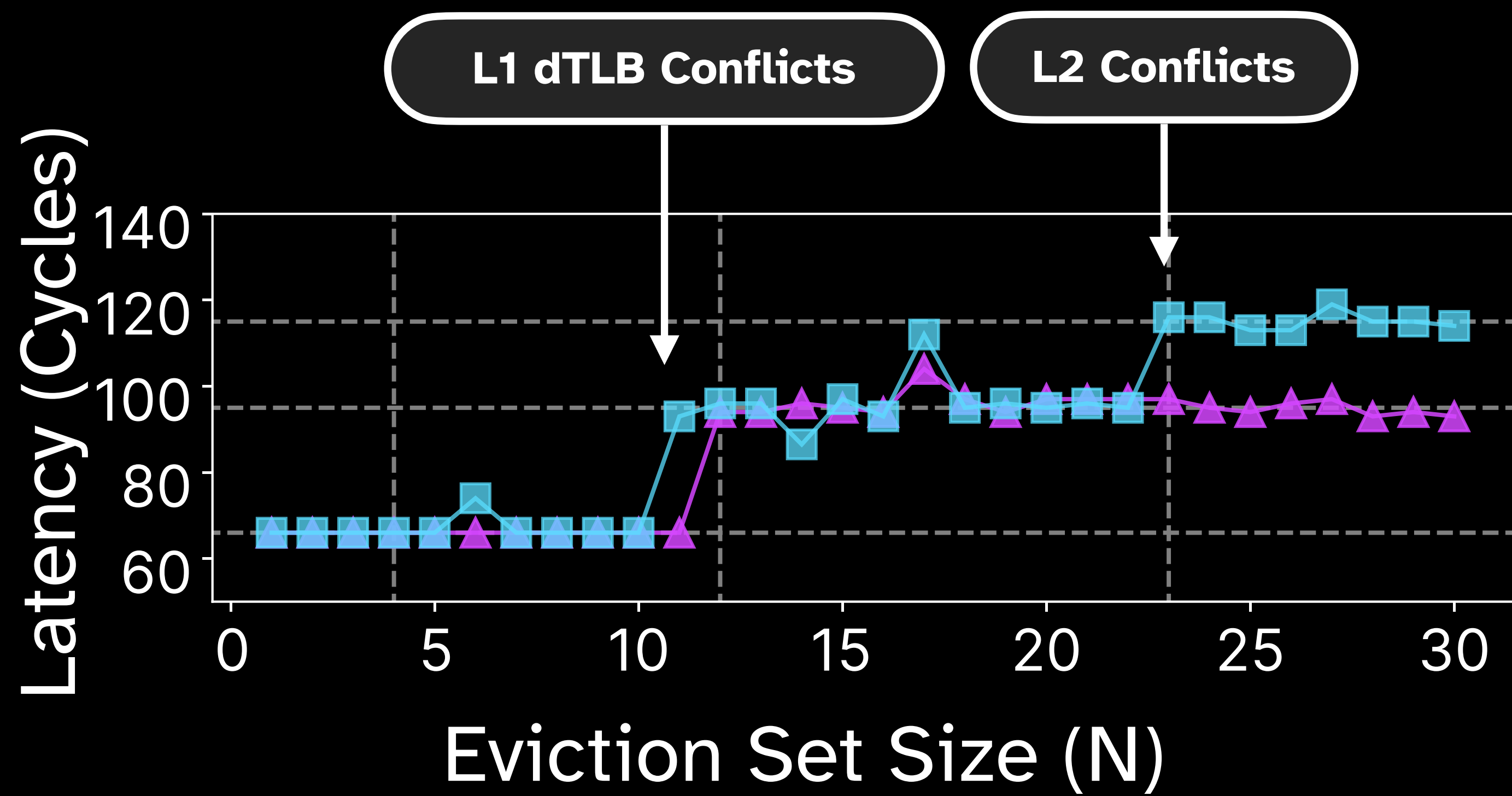
# Data TLB Results



Latency from dTLB conflicts



# Data TLB Results

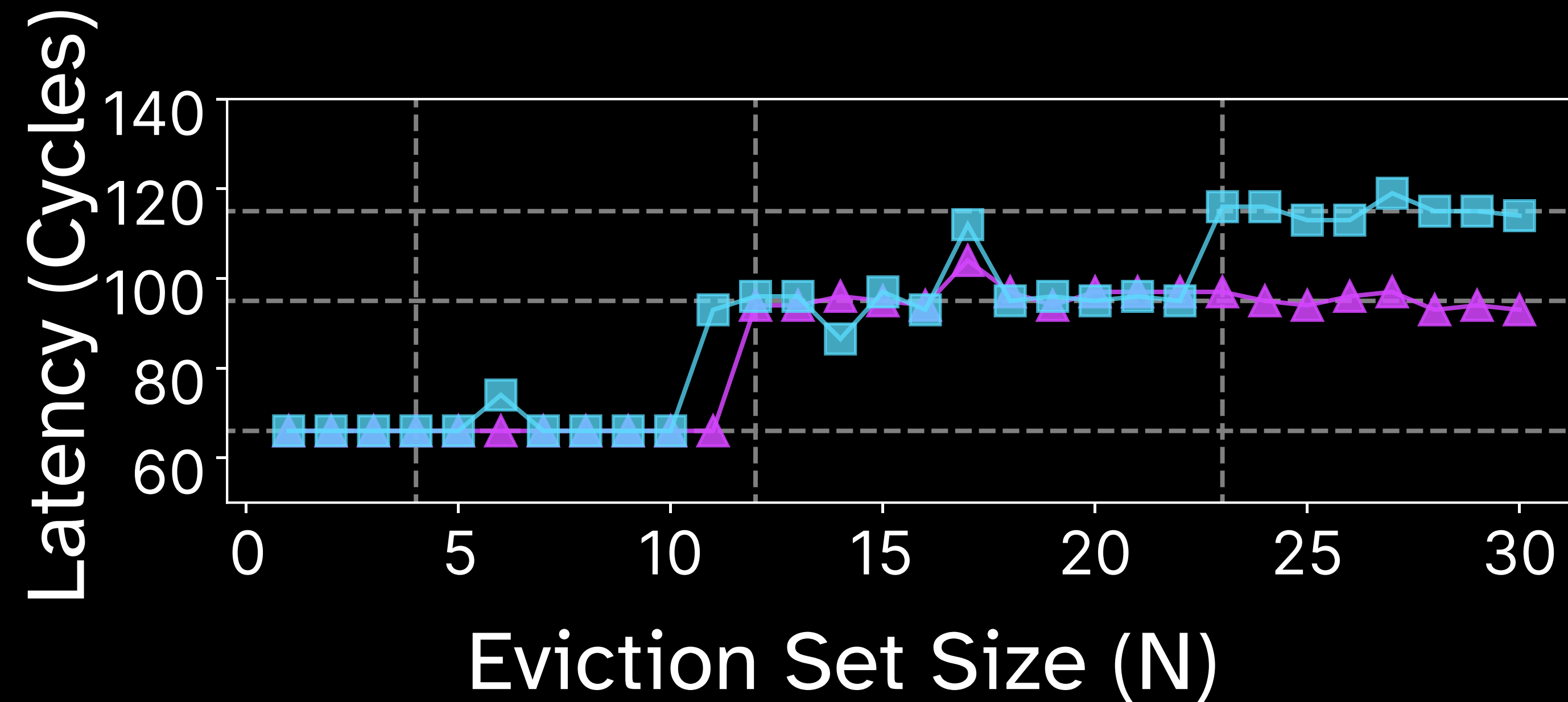


Latency from dTLB conflicts





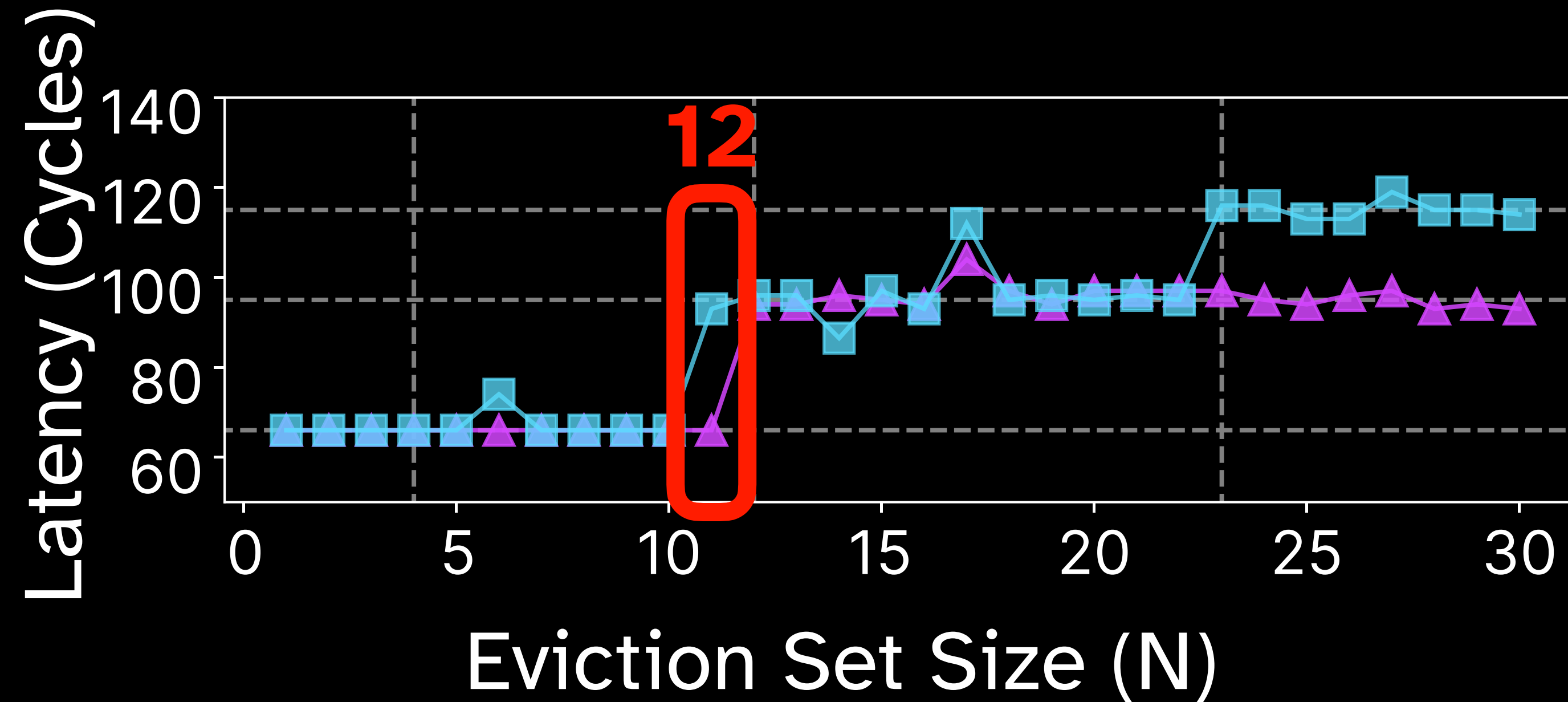
# Data TLB Eviction Sets



**Latency from dTLB conflicts**

—▲— 256 x 16KB    —■— 2K x 16KB    —●— 256 x 128B    —×— 32 x 16KB

# Data TLB Eviction Sets

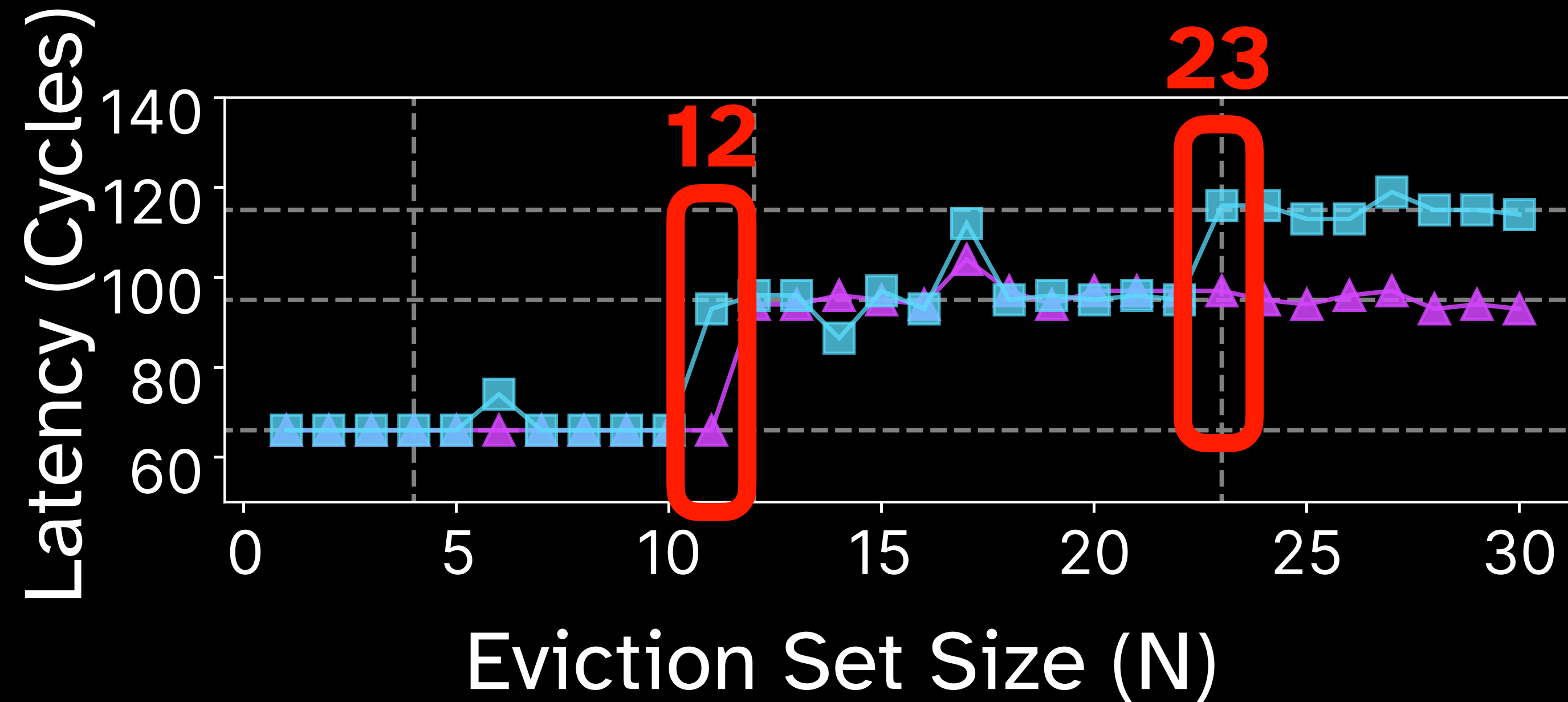


Latency from dTLB conflicts

**L1 dTLB**  
12 Addresses  
Stride of **256 Pages**

▲ 256 x 16KB    ■ 2K x 16KB    ● 256 x 128B    ✕ 32 x 16KB

# Data TLB Eviction Sets



Latency from dTLB conflicts

**L1 dTLB**  
12 Addresses  
Stride of **256 Pages**

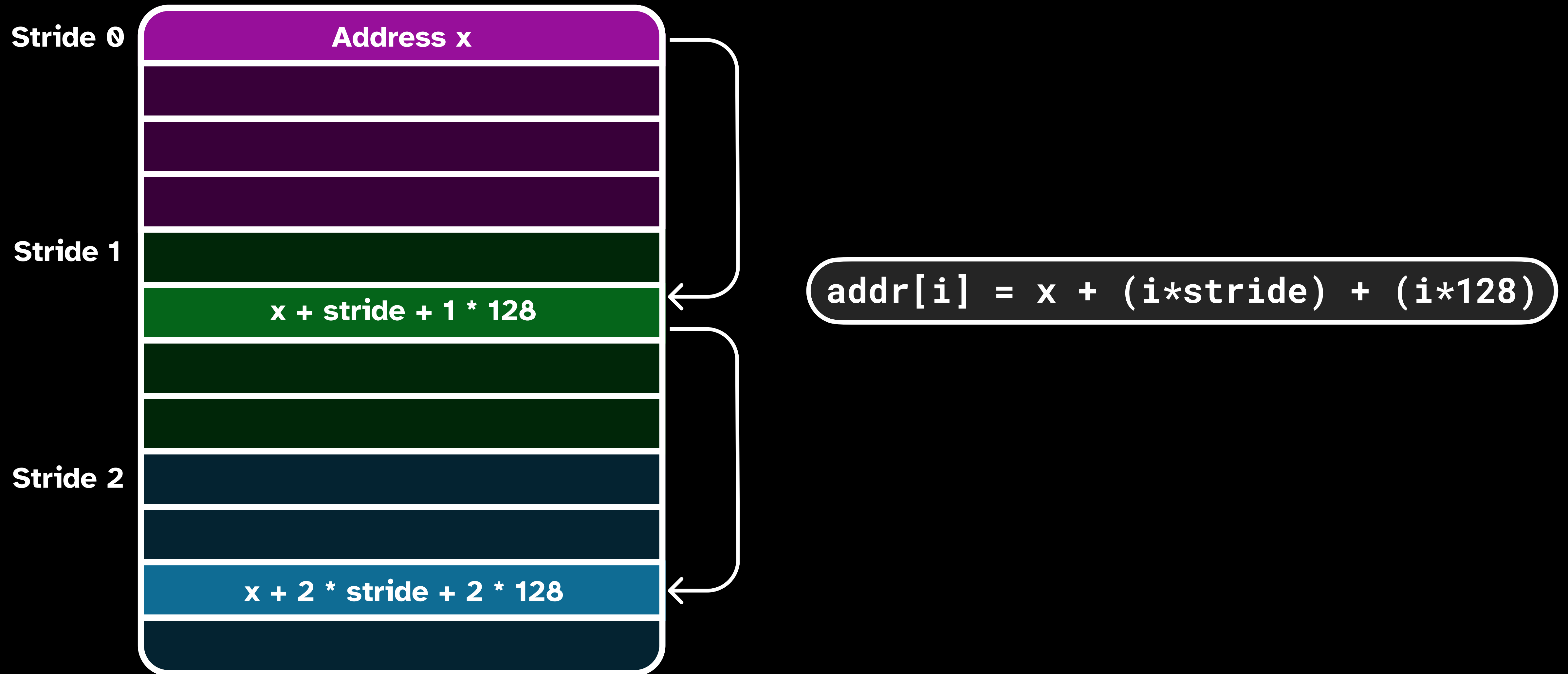
**L2 TLB**  
23 Addresses  
Stride of **2048 Pages**

▲ 256 x 16KB    ■ 2K x 16KB    ● 256 x 128B    ✕ 32 x 16KB

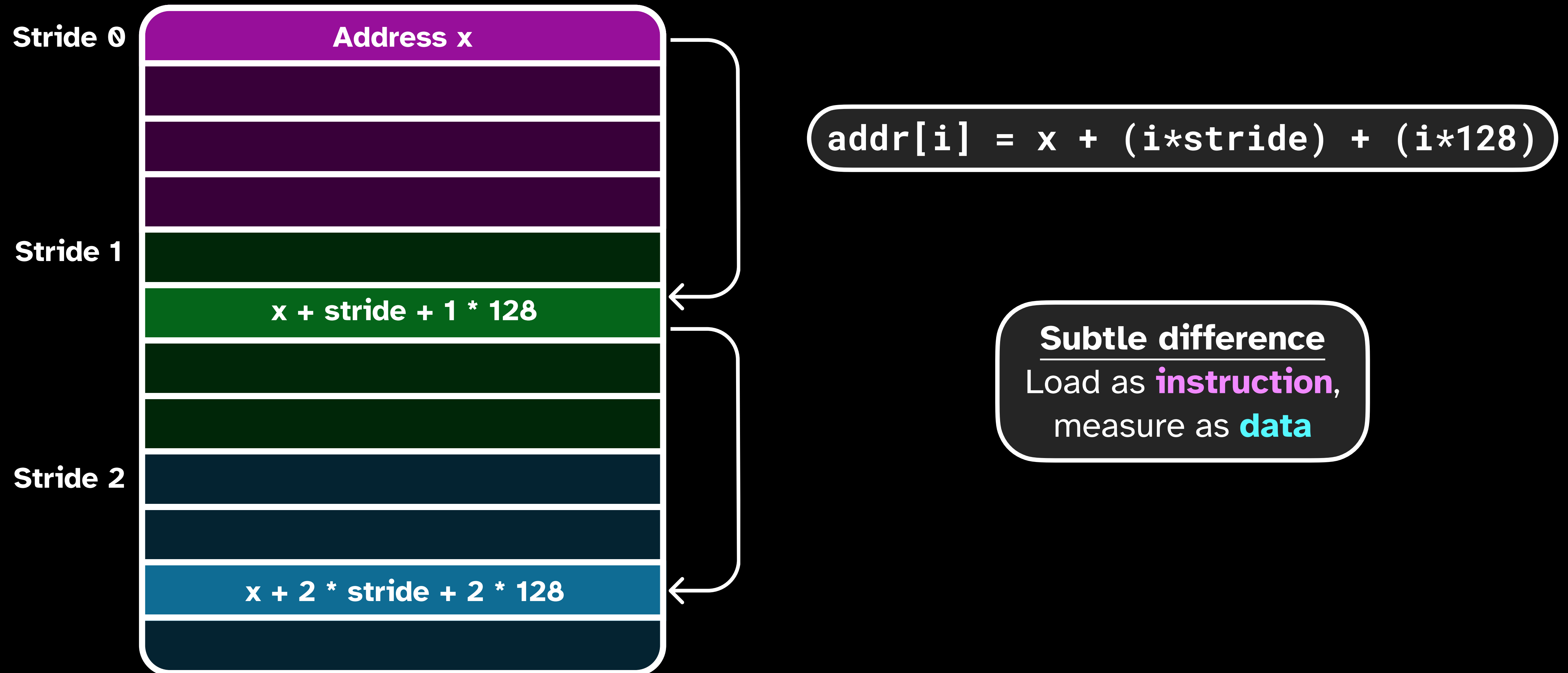
# **Experiment 2**

## **Instruction TLB Behavior**

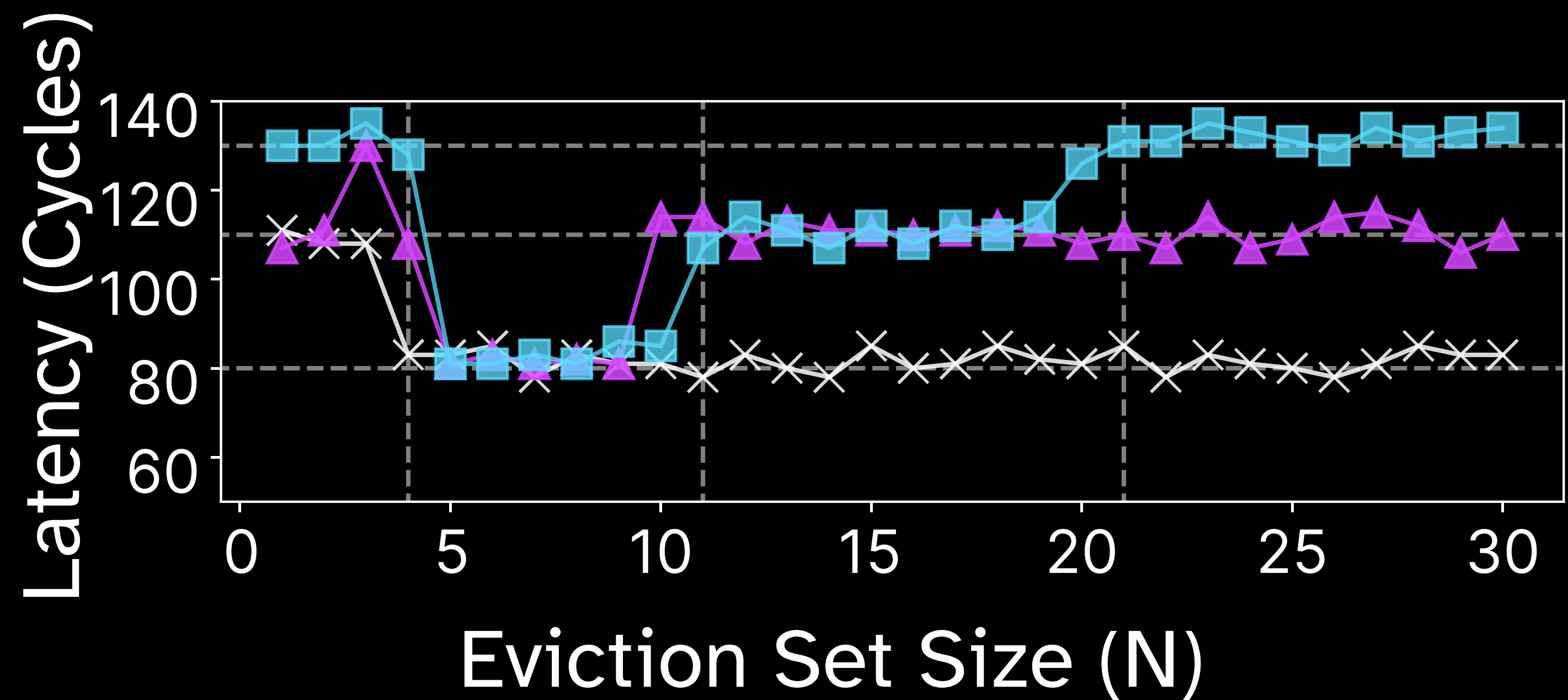
# Reversing the Instruction TLB



# Reversing the Instruction TLB



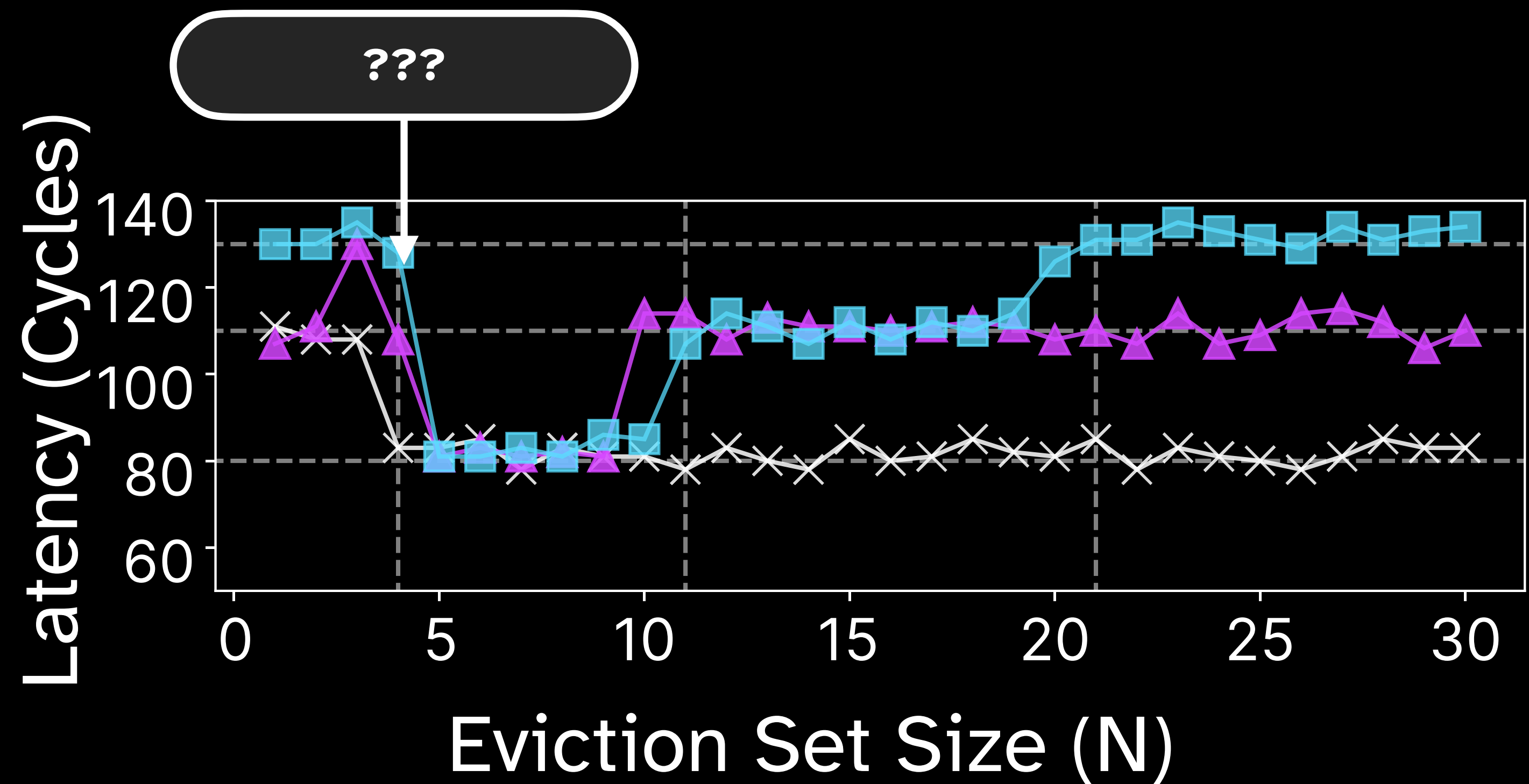
# Instruction TLB Results



Latency from dTLB/iTLB conflicts



# Instruction TLB Results



—▲— 256 x 16KB

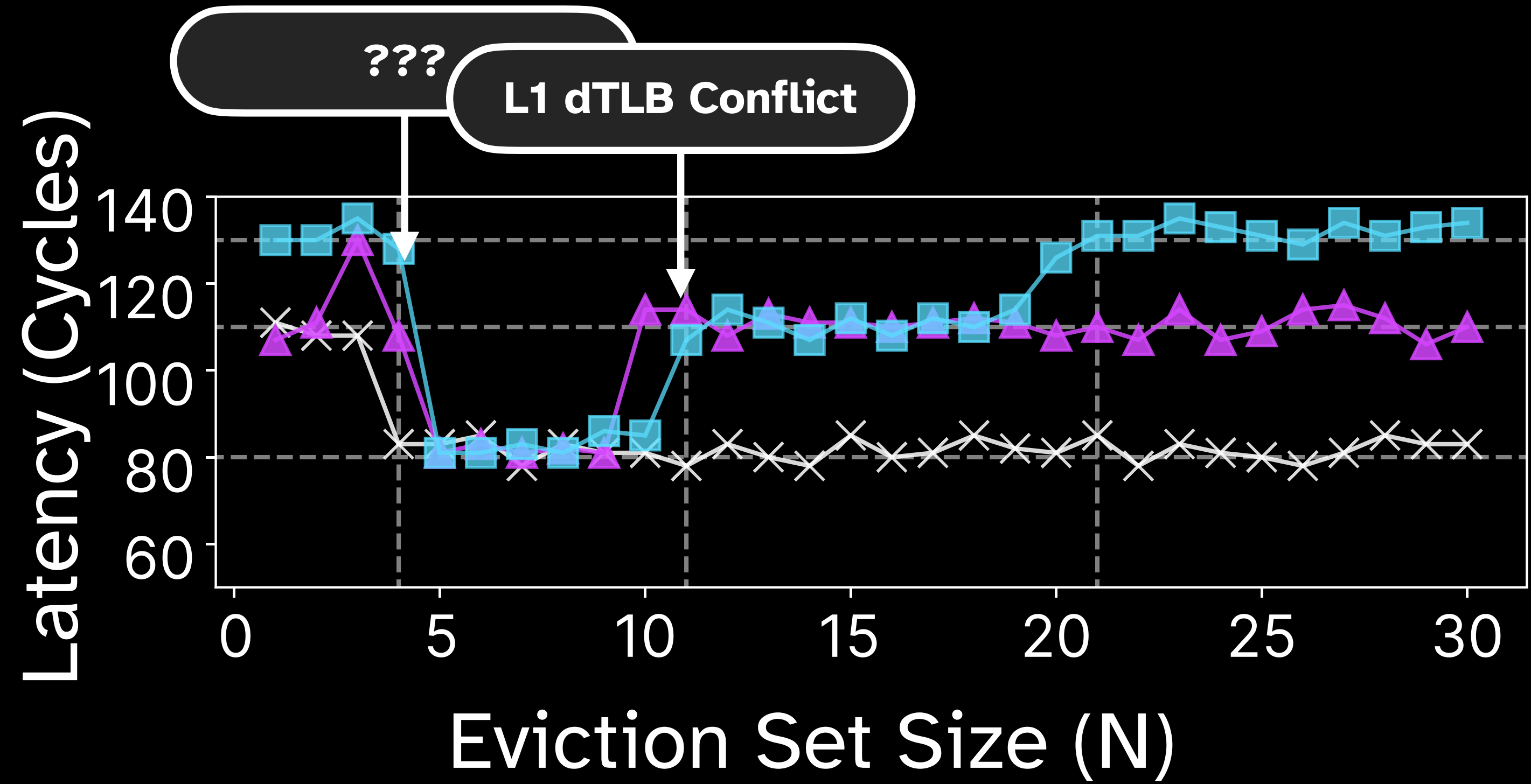
—■— 2K x 16KB

—●— 256 x 128B

—×— 32 x 16KB



# Instruction TLB Results



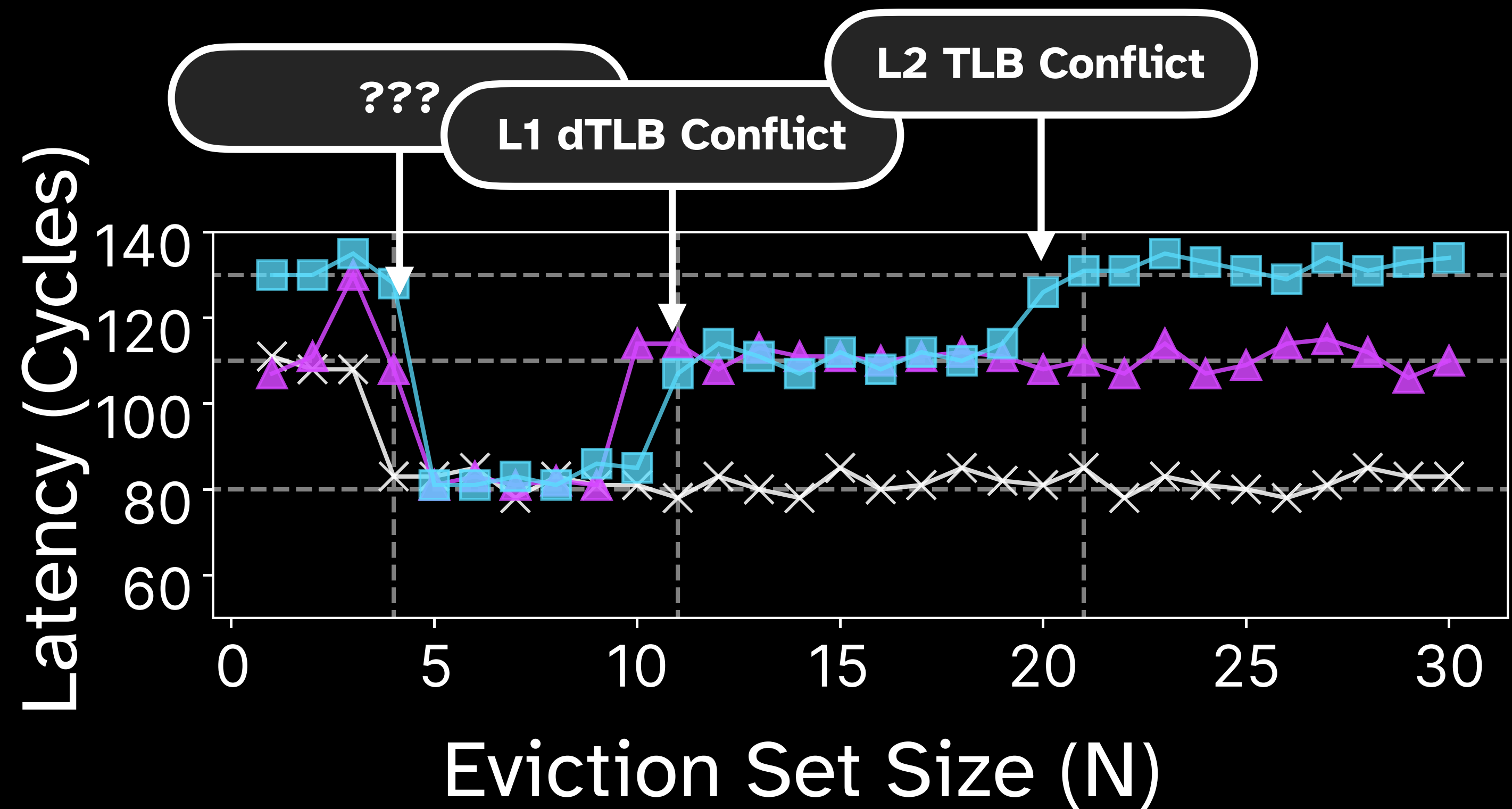
▲ 256 x 16KB

■ 2K x 16KB

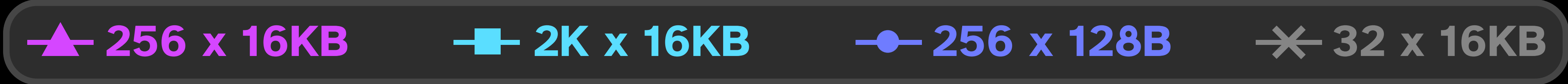
● 256 x 128B

✕ 32 x 16KB

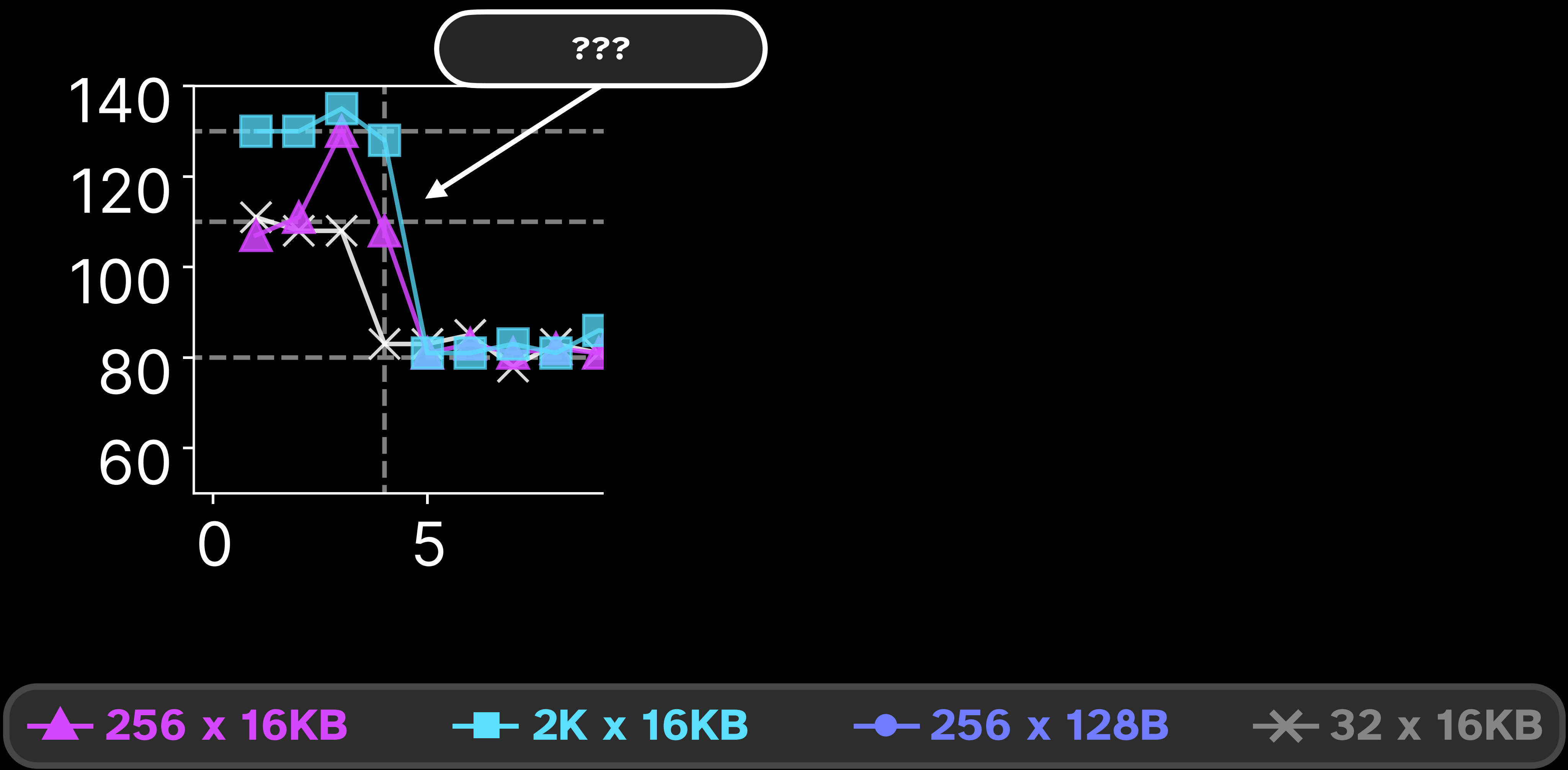
# Instruction TLB Results



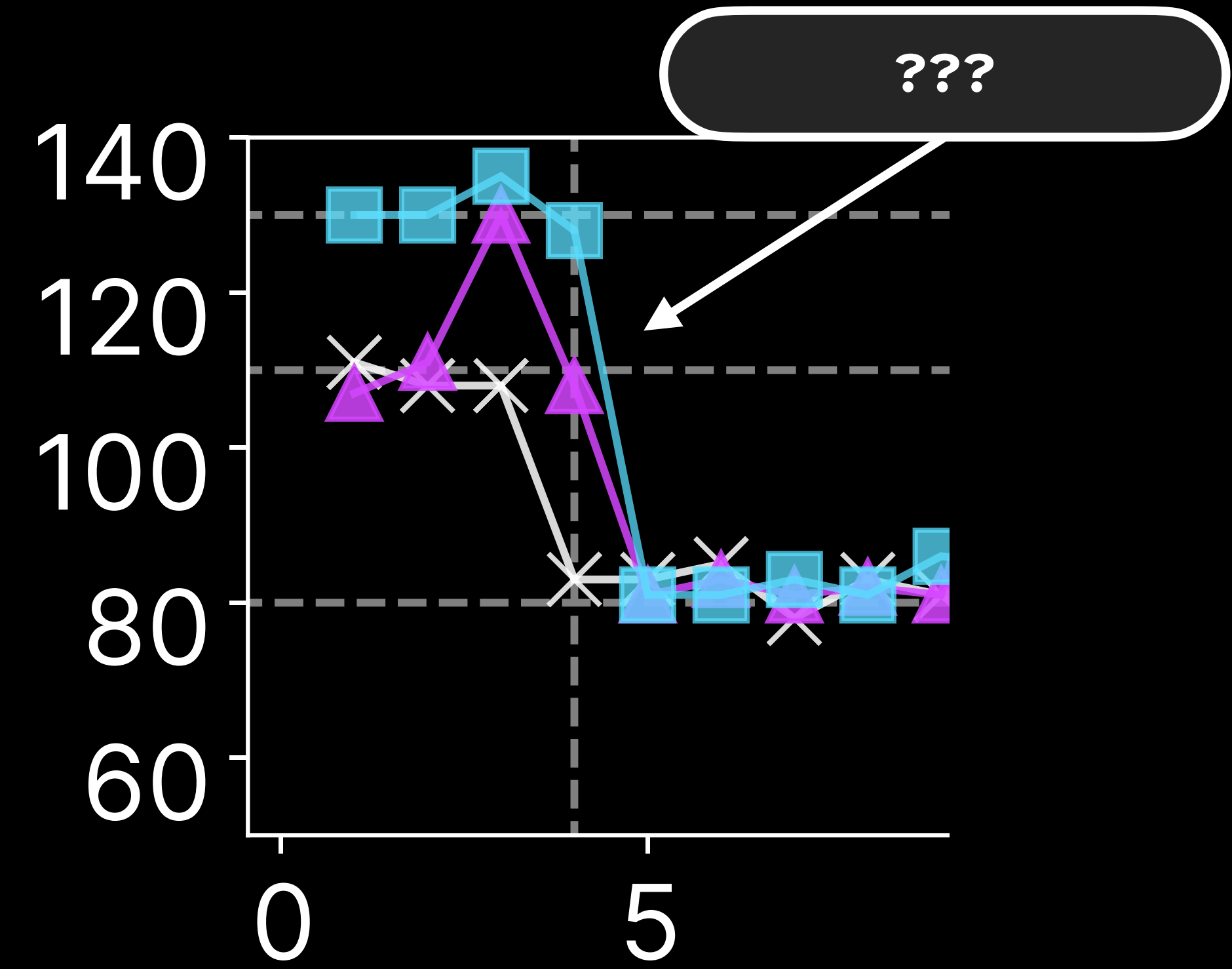
Latency from dTLB/iTLB conflicts



# Instruction TLB Results



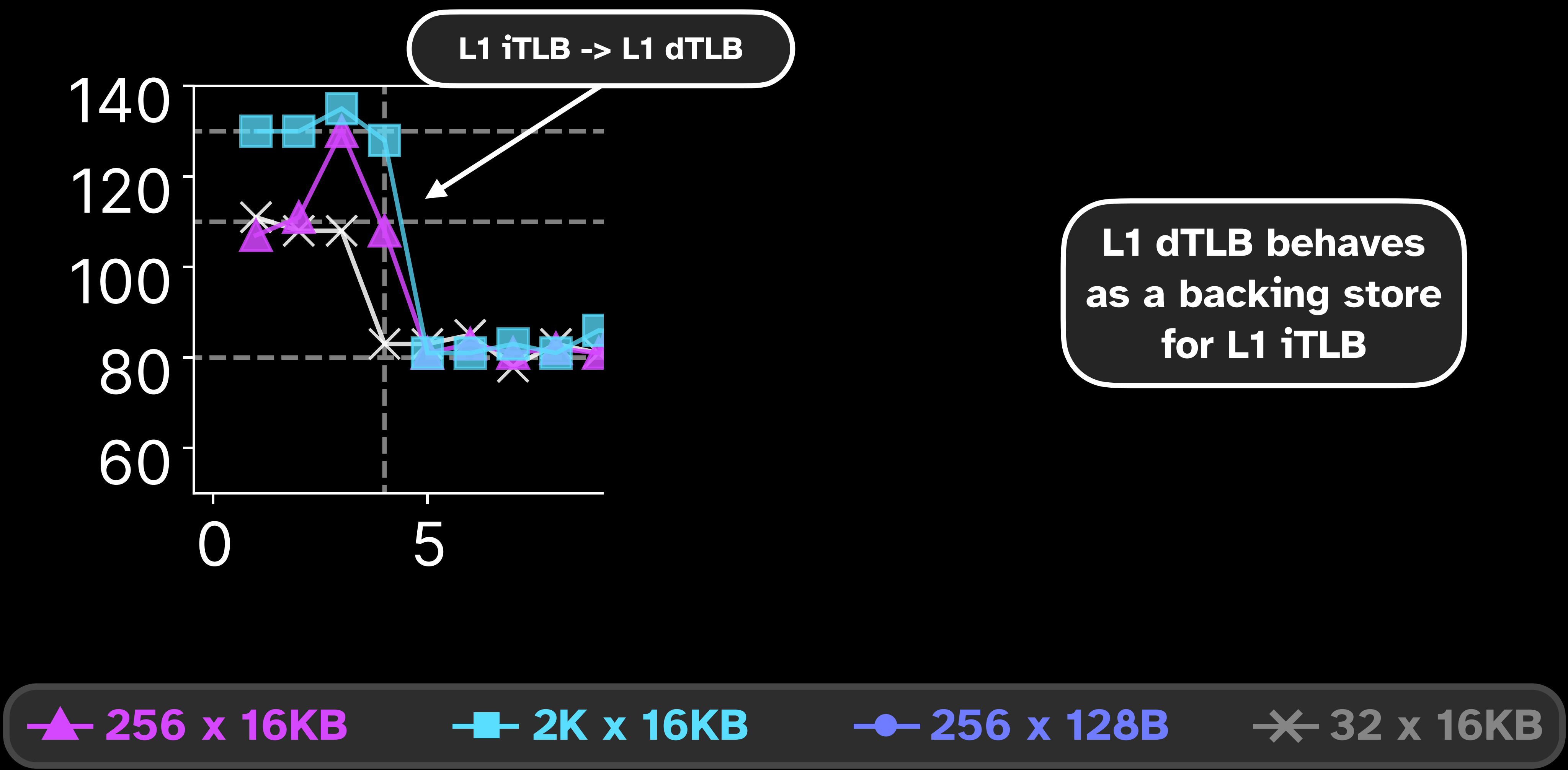
# Instruction TLB Results



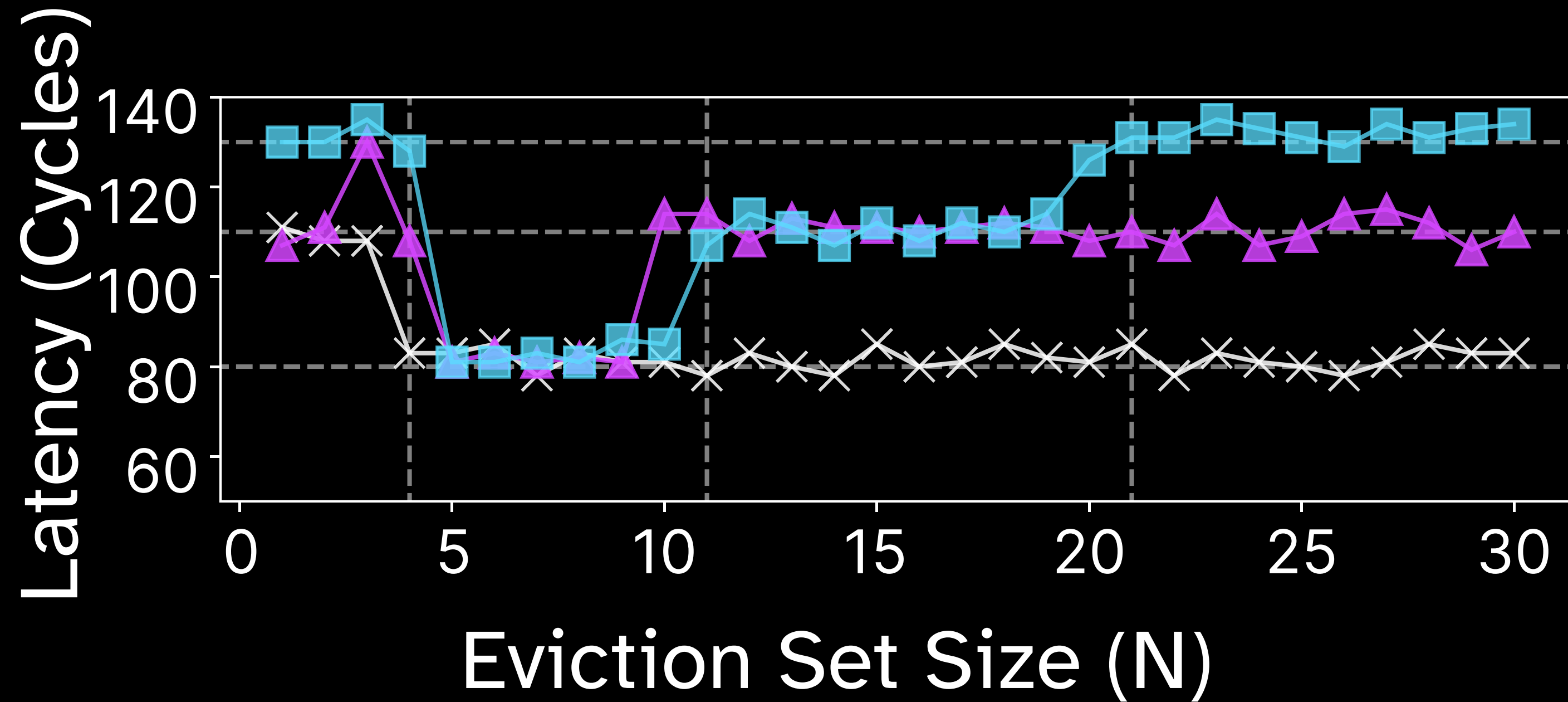
**Recall**  
Load as **instruction**,  
measure as **data**



# Instruction TLB Results



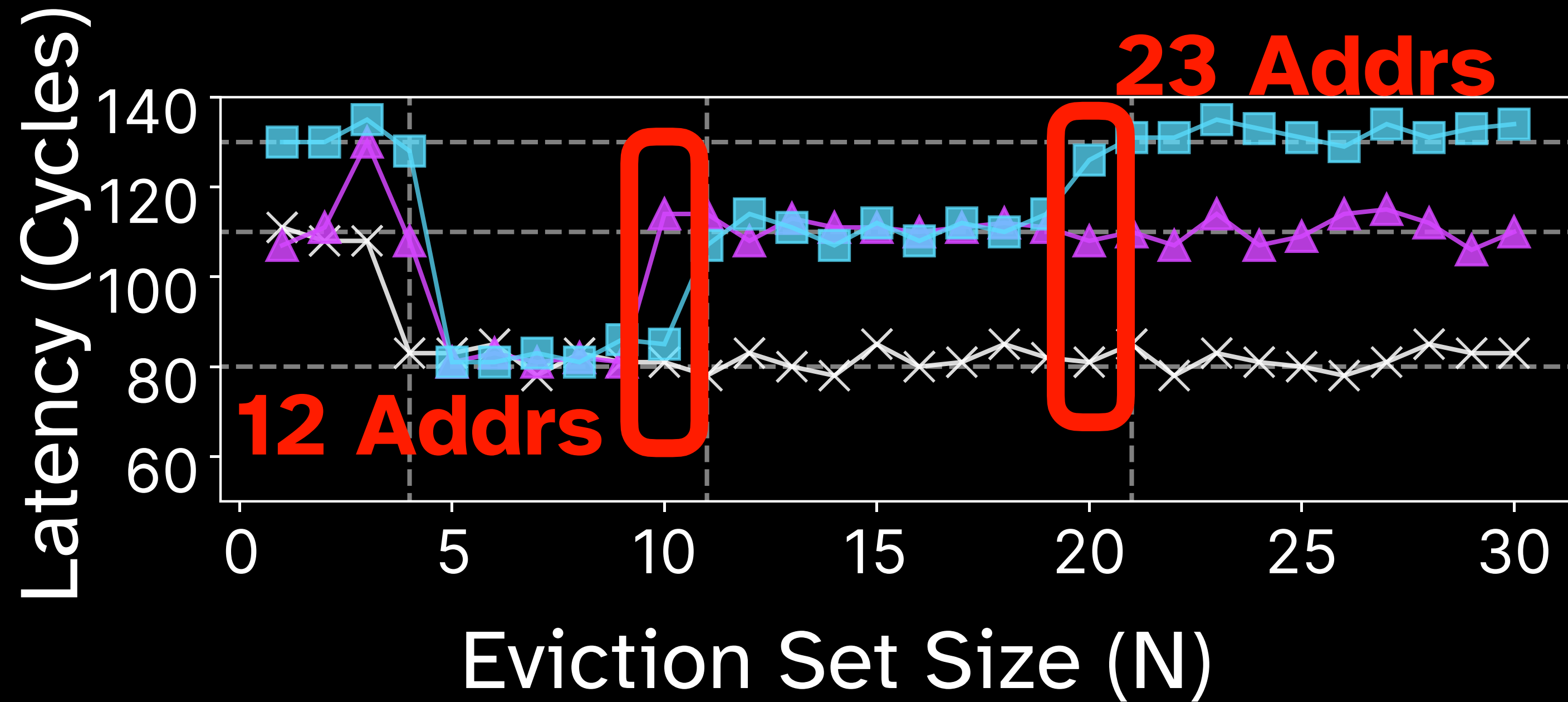
# Instruction TLB Eviction Sets



**Latency from dTLB/iTLB conflicts**

▲ 256 x 16KB    ■ 2K x 16KB    ● 256 x 128B    ✕ 32 x 16KB

# Instruction TLB Eviction Sets

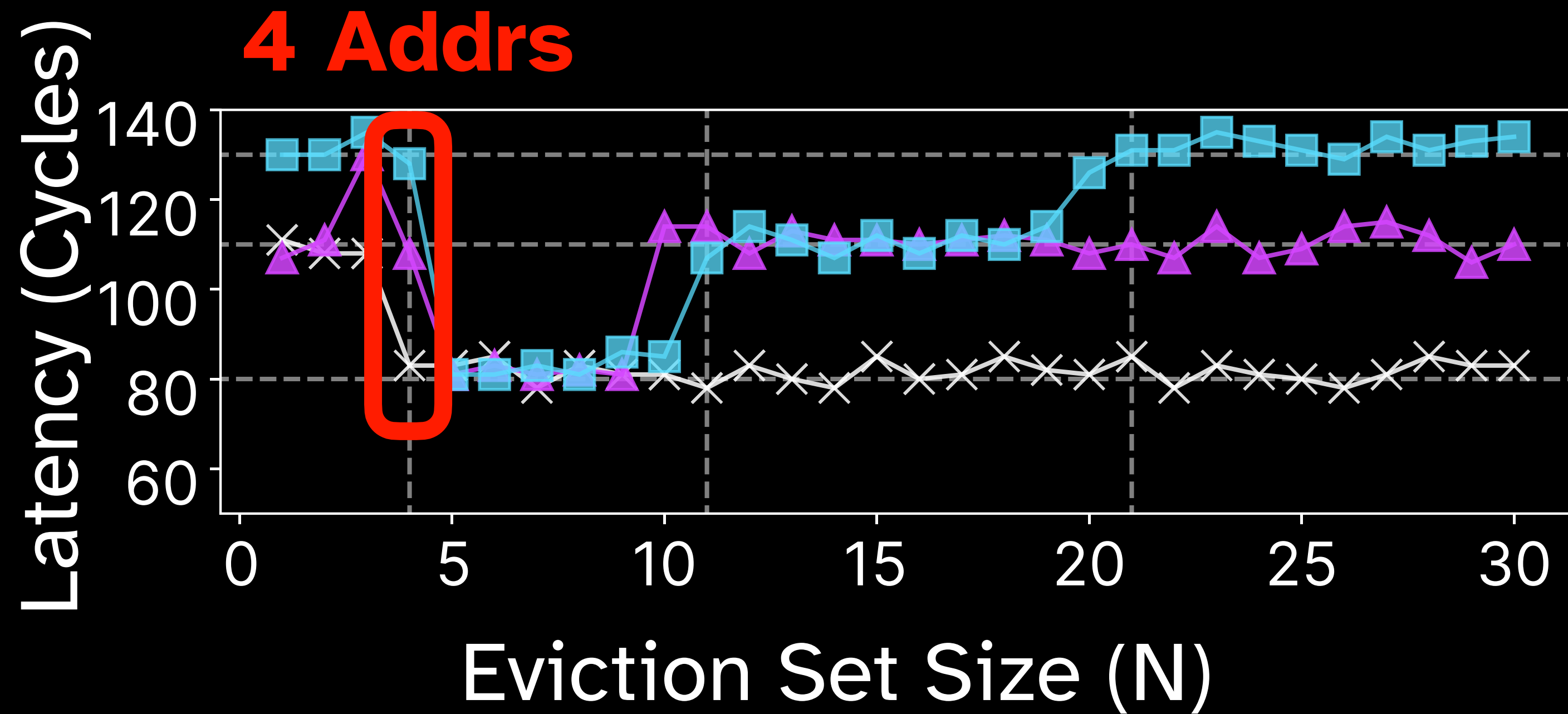


After moving to dTLB,  
same eviction sets  
from before still work.

Latency from dTLB/iTLB conflicts

▲ 256 x 16KB    ■ 2K x 16KB    ● 256 x 128B    ✕ 32 x 16KB

# Instruction TLB Eviction Sets

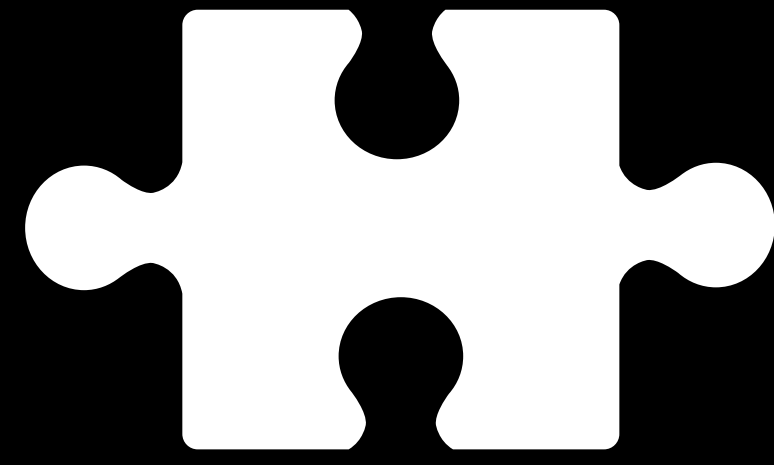


**L1 iTLB**  
4 Addresses  
Stride of 32 Pages.

**Latency from dTLB/iTLB conflicts**

▲ 256 x 16KB    ■ 2K x 16KB    ● 256 x 128B    ✕ 32 x 16KB





**Putting it All Together**

# Finding Vulnerable Object

- Target should be something rarely used
- Training address should be something we have a valid signature for (this can just be the correct contents of memory)
- Target and training address should be far apart (not in same cache set/ page)

# Victim

- We created an IOKit driver (PacmanKit) for experimentation
- PacmanKit has an IOUserClient object containing a C++ object we are attacking
  - Swap this with a PACMAN Gadget of your choice
- Not "real" (release) kernel code, instead a custom kernel extension
- We are demonstrating a vulnerability in the hardware, not the software
- PacmanKit includes kernel read/ write and kASLR leak primitives we will use
  - Swap these for your own!

# Our Victim

```
IOReturn CallServiceRoutine(OSObject *target, void *reference,  
IOExternalMethodArguments *args) __attribute__((aligned(0x80))) {  
  
    if (args->scalarInput[0] < LIMIT) {  
        return ((PacmanUser *)target)->helper.externalMethod();  
    }  
  
}
```

# Our Victim

```
return target->helper.externalMethod();
```

```
ldr x16, [helper]  
mov x17, helper  
movk x17, #0xd986, lsl #48  
autda x16, x17  
ldr x8, [x16]  
mov    x9, x16  
mov    x17, x9  
movk   x17, #0xa7d5, lsl #48  
autia x8, x17  
blr x8
```

# Our Victim

```
return target->helper.externalMethod();
```

```
ldr x16, [helper]
mov x17, helper
movk x17, #0xd986, lsl #48
autda x16, x17
ldr x8, [x16]
mov x9, x16
mov x17, x9
movk x17, #0xa7d5, lsl #48
autia x8, x17
blr x8
```

**Load vtable pointer  
from helper object**



# Our Victim

```
return target->helper.externalMethod();
```

```
ldr x16, [helper]
mov x17, helper
movk x17, #0xd986, lsl #48
autda x16, x17
ldr x8, [x16]
mov    x9, x16
mov    x17, x9
movk   x17, #0xa7d5, lsl #48
autia x8, x17
blr x8    ← Execute vtable entry
```

# Our Victim

```
return target->helper.externalMethod();
```

```
ldr x16, [helper]
mov x17, helper
movk x17, #0xd986, lsl #48
autda x16, x17
ldr x8, [x16]
mov x9, x16
mov x17, x9
movk x17, #0xa7d5, lsl #48
autia x8, x17
blr x8
```

**We can  
ignore salts**



# Gadget Restrictions

- We have found **blraa** and related "all in one" instructions are less vulnerable than `aut -> blr` pairs
- Potentially due to data race in pipeline between the implied authenticate and instruction access (they can happen in parallel)

# Creating Eviction Sets

- Start with virtual address matching victim TLB and cache lines
- Add virtual addresses in increments of 8192 bytes
- Access data eviction set via LDR
- Access inst eviction set via BLR
  - Inst eviction set should be filled with "ret"s and marked executable

# Creating Eviction Sets

|      |
|------|
| bits |
| bits |
| bits |
| bits |
| bits |
| bits |
| bits |
| ...  |

Data

|     |
|-----|
| ret |
| ret |
| ret |
| ret |
| ret |
| ret |
| ret |
| ... |

Inst

# Lengthening the Window

```
IOReturn CallServiceRoutine(OSObject *target, void *reference,  
IOExternalMethodArguments *args) __attribute__((aligned(0x80))) {  
  
    if (args->scalarInput[0] < LIMIT) {  
        return ((PacmanUser *)target)->helper.externalMethod();  
    }  
  
}
```

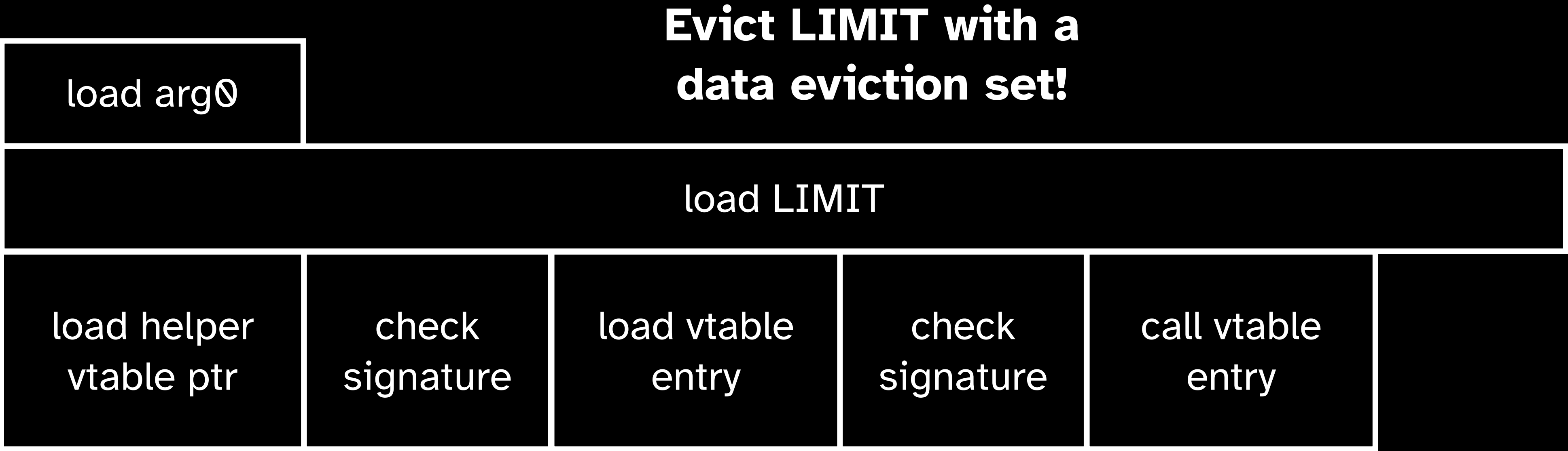
# Lengthening the Window

```
if (arg0 < LIMIT) {  
    return ((PacmanUser *)target)->helper.externalMethod();  
}
```

|                           |                    |                      |                    |                      |
|---------------------------|--------------------|----------------------|--------------------|----------------------|
| load arg0                 |                    |                      |                    |                      |
| load LIMIT                |                    |                      |                    |                      |
| load helper<br>vtable ptr | check<br>signature | load vtable<br>entry | check<br>signature | call vtable<br>entry |

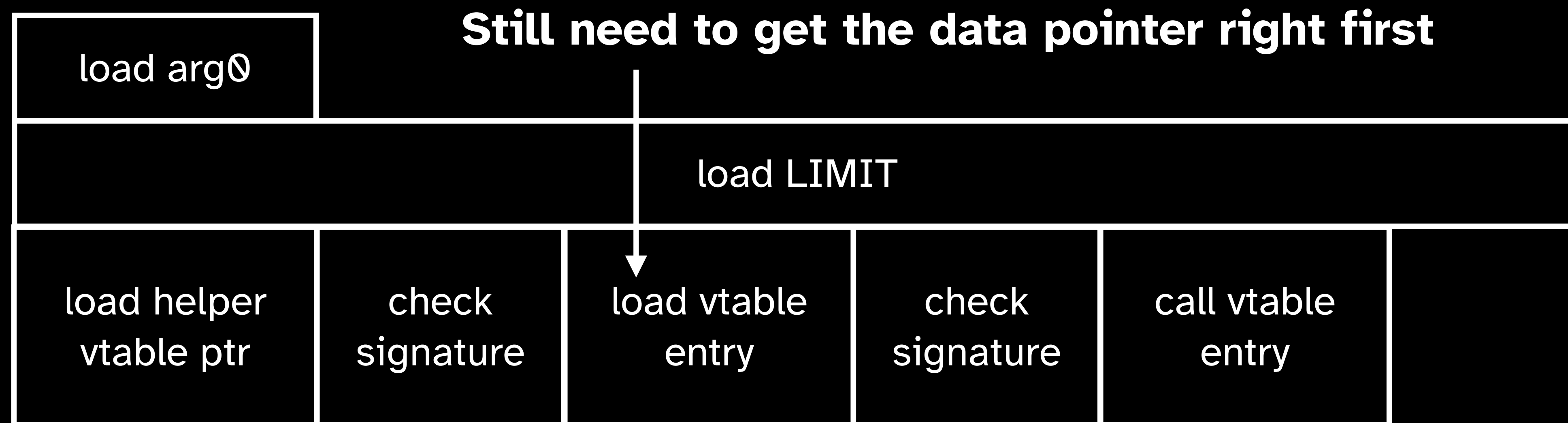
# Lengthening the Window

```
if (arg0 < LIMIT) {  
    return ((PacmanUser *)target)->helper.externalMethod();  
}
```



# Lengthening the Window

```
if (arg0 < LIMIT) {  
    return ((PacmanUser *)target)->helper.externalMethod();  
}
```



# Attack Overview

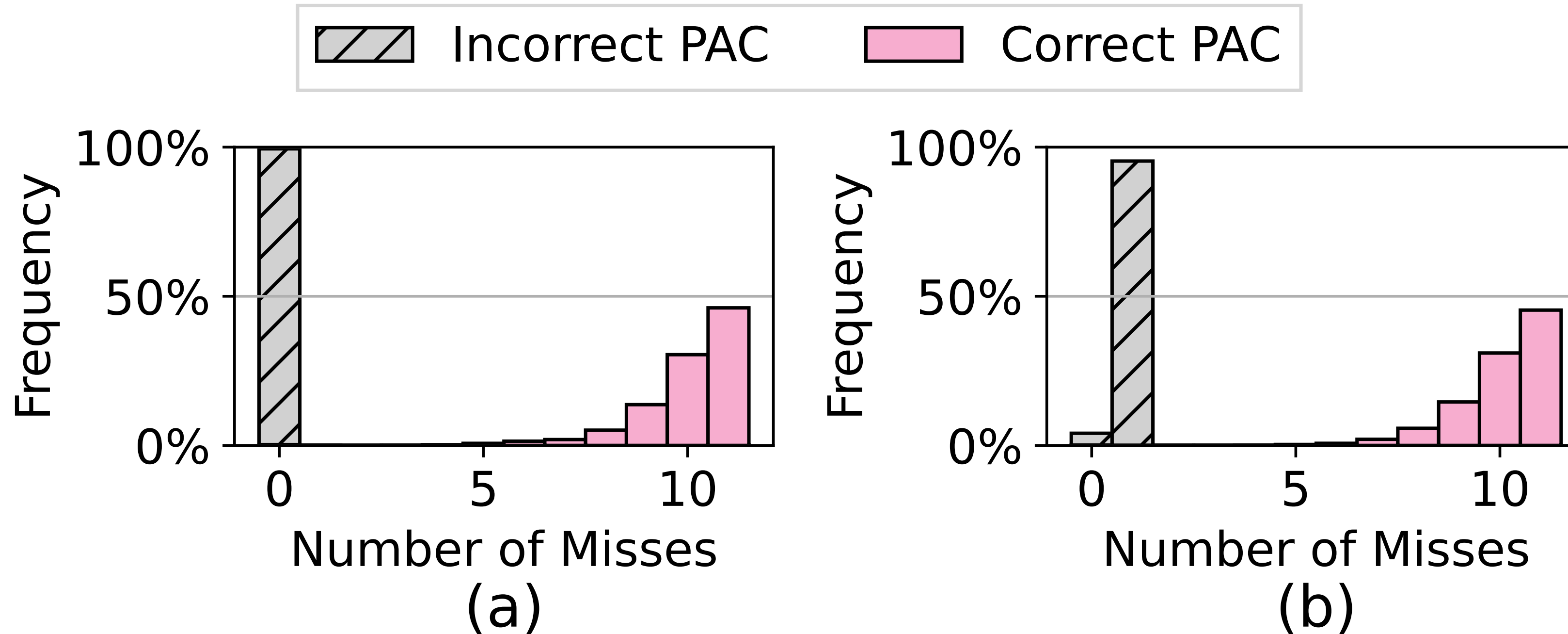
1. Train branch predictor with non-speculative calls using correctly signed pointer
2. Overwrite pointer with guess
3. Evict bounds check variable with separate eviction set
4. Prime the cache with attacker eviction set
5. Trigger speculative kernel execution by manipulating index parameter
6. Probe attacker cache set, measuring number of misses



# Two Implementations

- C version (seen in our ISCA paper)
  - Reliability
  - ~3 minutes/ pointer
  - Probe instruction port with data accesses (requires kernel iTLB flush)
- Rust version (DEF CON 30)
  - Performance
  - ~10 seconds/ data pointer
  - ~45 seconds/ inst pointer
  - Probe instruction port with instruction accesses (requires nothing special in the kernel)

# PAC Oracle Accuracy- C Version



Data

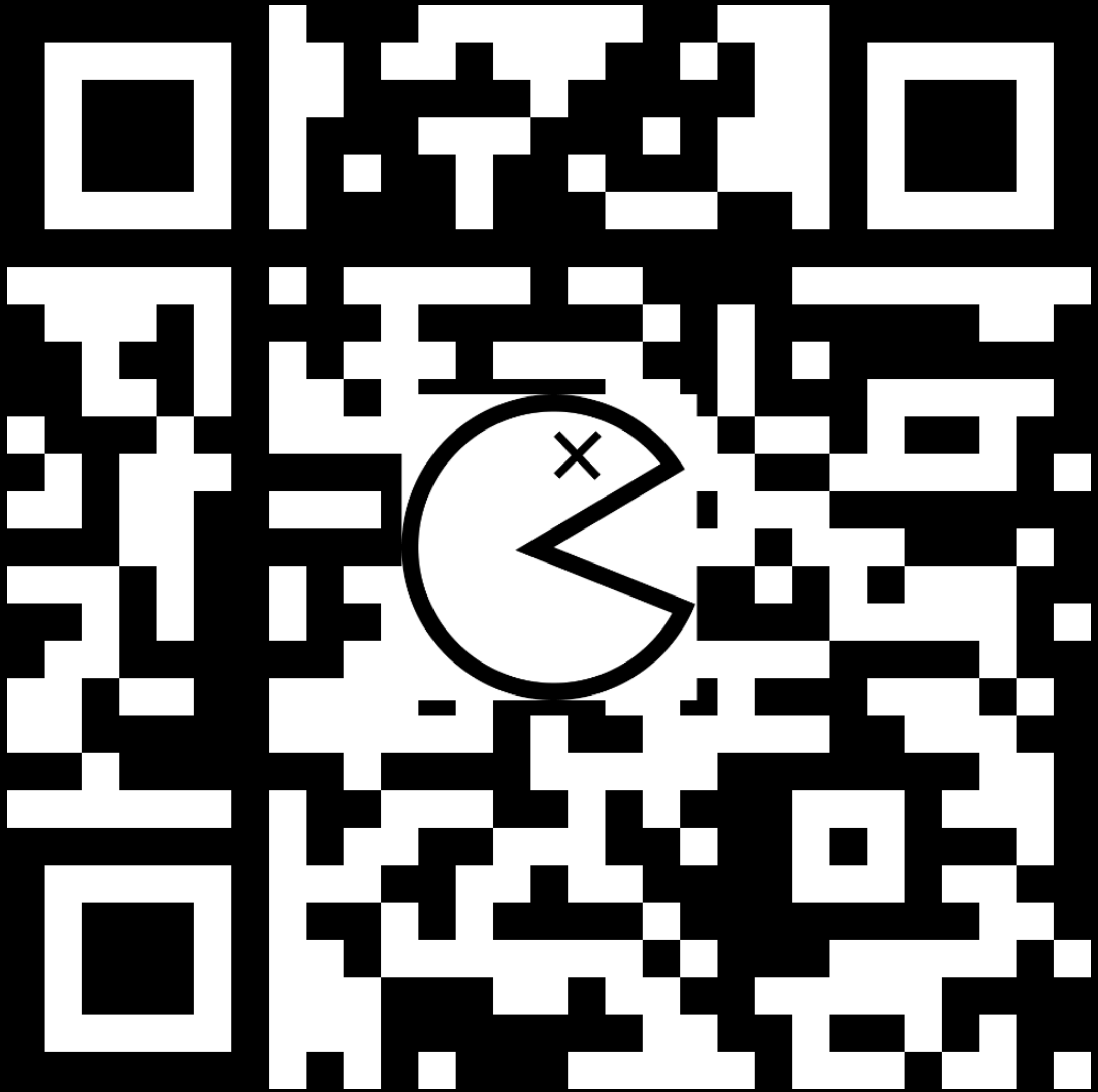
Instructions

**Demo**

**All Code on our GitHub!**

<https://github.com/CSAIL-Arch-Sec>

Follow us on Twitter!



compute. collaborate. create.

PACMANATTACK.COM

# Appendix

# Data PACMAN Attack



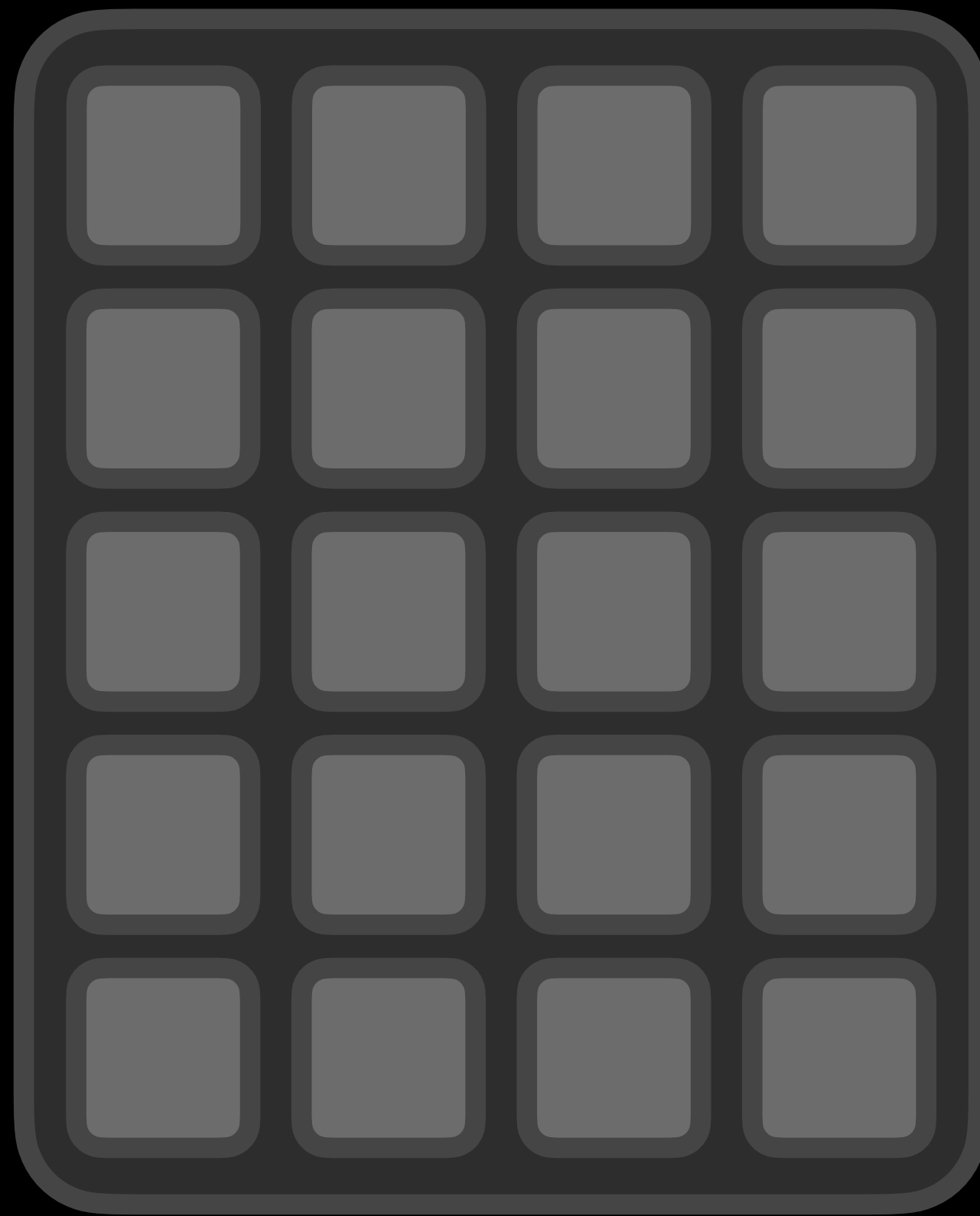
**L1 dTLB**



**Branch  
Predictor**

```
if (condition):  
    verified = AUT(pointer)  
    load(verified)
```

# Data PACMAN Attack



**L1 dTLB**



**Branch  
Predictor**

```
if (condition):  
    verified = AUT(pointer)  
    load(verified)
```

**1**

**We train the branch predictor  
to use a known signed pointer.**



# Data PACMAN Attack



L1 dTLB

Not  
Taken

Branch  
Predictor

```
if (condition):  
    verified = AUT(good ptr)  
    load(verified)
```

1

We train the branch predictor  
to use a known signed pointer.

# Data PACMAN Attack



L1 dTLB

Not  
Taken

Branch  
Predictor

```
if (condition):  
    verified = AUT(good ptr)  
    load(verified)
```

1

We train the branch predictor  
to use a known signed pointer.

# Data PACMAN Attack



L1 dTLB

Not  
Taken

Branch  
Predictor

```
if (condition):  
    verified = AUT(good ptr)  
    load(verified)
```

1

We train the branch predictor  
to use a known signed pointer.

# Data PACMAN Attack



L1 dTLB

Taken

Branch  
Predictor

```
if (condition):  
    verified = AUT(good ptr)  
    load(verified)
```

1

We train the branch predictor  
to use a known signed pointer.

# Data PACMAN Attack



**L1 dTLB**



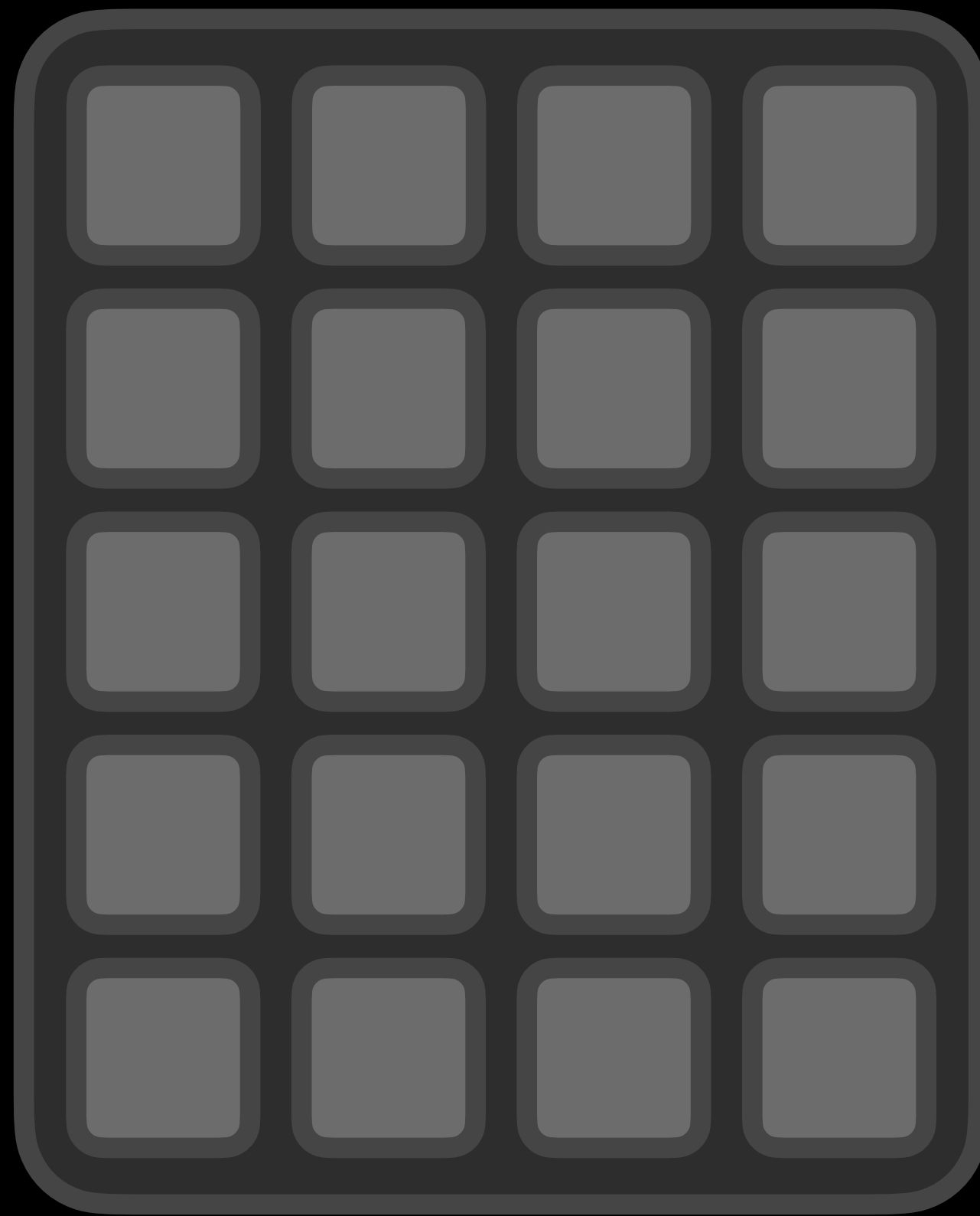
**Branch  
Predictor**

```
if (condition):  
    verified = AUT(pointer)  
    load(verified)
```

**2**

**Reset the entire TLB**

# Data PACMAN Attack



**L1 dTLB**



**Branch  
Predictor**

```
if (condition):  
    verified = AUT(pointer)  
    load(verified)
```

**3**

**Prime the L1 dTLB  
with an eviction set**

# Data PACMAN Attack



**L1 dTLB**



**Branch  
Predictor**

```
if (condition):  
    verified = AUT(pointer)  
    load(verified)
```

**3**

**Prime the L1 dTLB  
with an eviction set**

# Data PACMAN Attack



**L1 dTLB**

Eviction Set

Taken

Branch  
Predictor

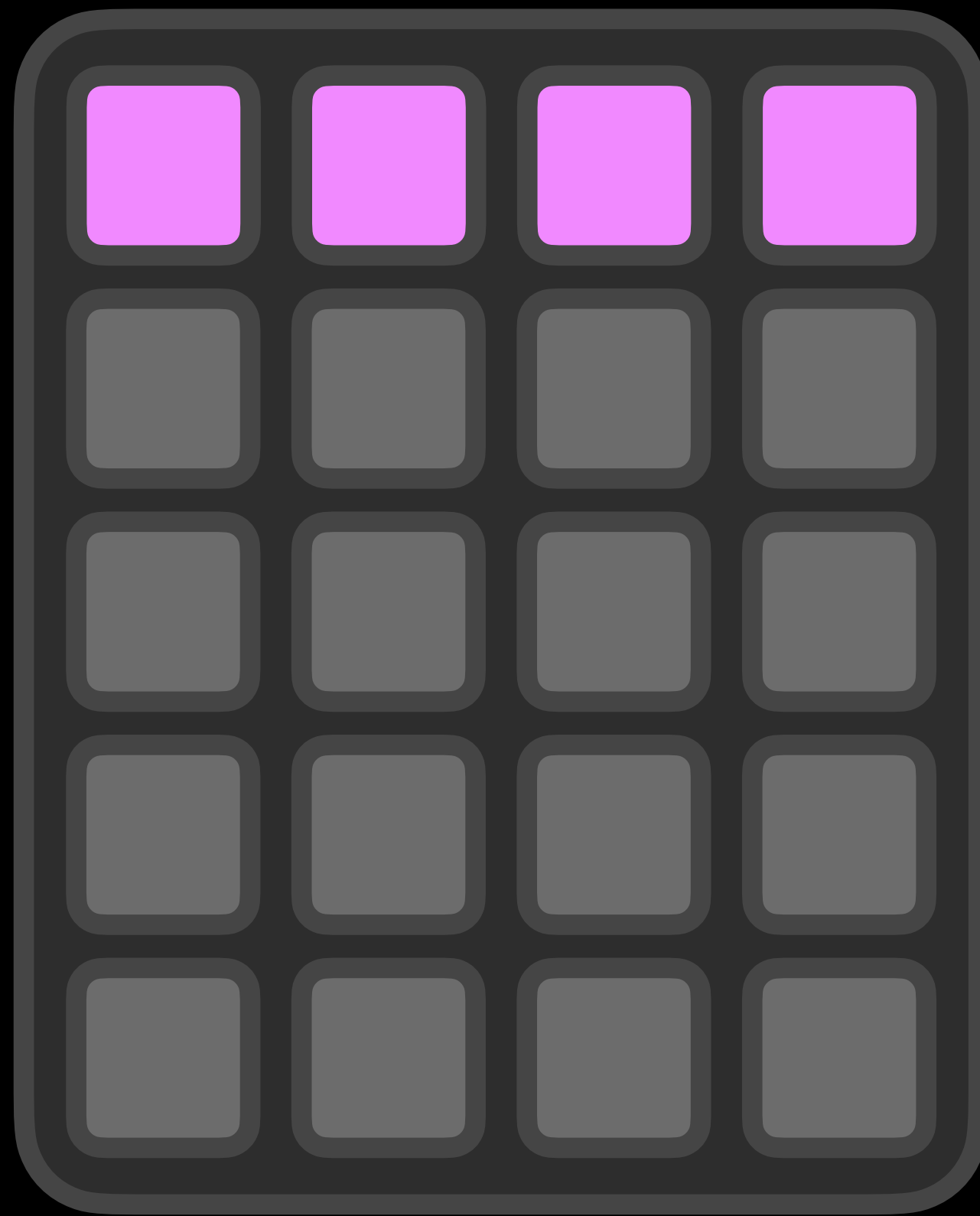
```
if (condition):  
    verified = AUT(pointer)  
    load(verified)
```

3

Prime the L1 dTLB  
with an eviction set



# Data PACMAN Attack



**L1 dTLB**



**Branch  
Predictor**

```
if (condition):  
    verified = AUT(pointer)  
    load(verified)
```

**4**

**Call the gadget with the  
pointer and PAC to guess.**

# Data PACMAN Attack



L1 dTLB

Taken

Branch  
Predictor

```
if (condition):  
    verified = AUT(guess ptr)  
    load(verified)
```

4

Call the gadget with the  
pointer and PAC to guess.

# Data PACMAN Attack



Speculative



L1 dTLB

Taken

Branch  
Predictor

```
if (condition):  
    verified = AUT(guess ptr)  
    load(verified)
```

4

Call the gadget with the  
pointer and PAC to guess.

# Data PACMAN Attack



Speculative



L1 dTLB

Taken

Branch  
Predictor

```
if (condition):  
    verified = AUT(guess ptr)  
    load(verified)
```

4

Call the gadget with the  
pointer and PAC to guess.

# Data PACMAN Attack



Speculative



L1 dTLB

Taken

Branch  
Predictor

```
if (condition):  
    verified = AUT(guess ptr)  
    load(verified)
```

If the guess  
was correct...

4

Call the gadget with the  
pointer and PAC to guess.

# Data PACMAN Attack



Speculative

Guess  
Pointer



L1 dTLB



Branch  
Predictor

```
if (condition):  
    verified = AUT(guess ptr)  
    load(verified)
```

4

Call the gadget with the  
pointer and PAC to guess.

# Data PACMAN Attack



Speculative



L1 dTLB

Taken

Branch  
Predictor

```
if (condition):  
    verified = AUT(guess ptr)  
    load(verified)
```

If the guess  
was incorrect...

4

Call the gadget with the  
pointer and PAC to guess.

# Data PACMAN Attack



Speculative

No Change



L1 dTLB



Branch  
Predictor

```
if (condition):  
    verified = AUT(guess ptr)  
    load(verified)
```

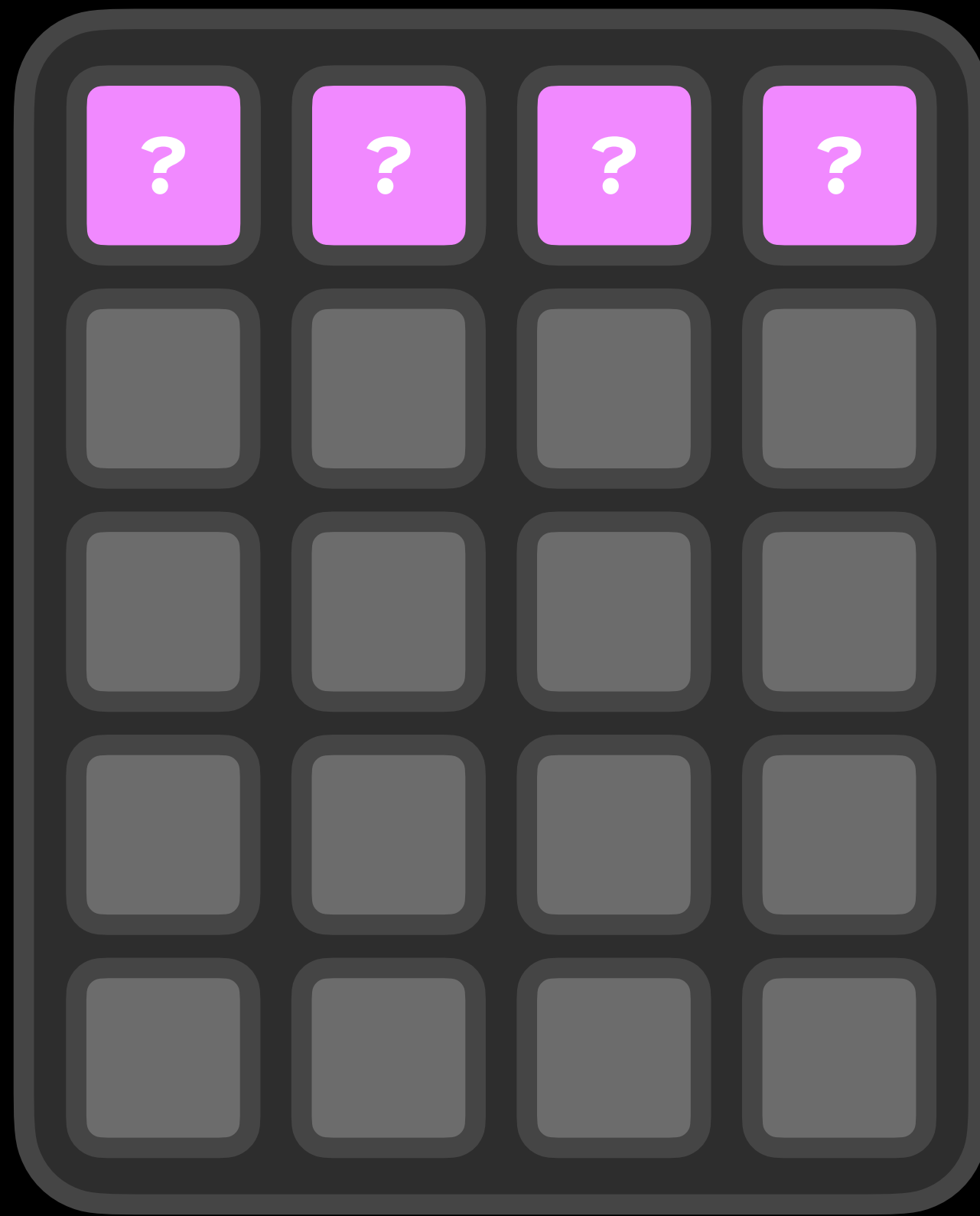
Speculative  
Exception!

4

Call the gadget with the  
pointer and PAC to guess.



# Data PACMAN Attack



**L1 dTLB**



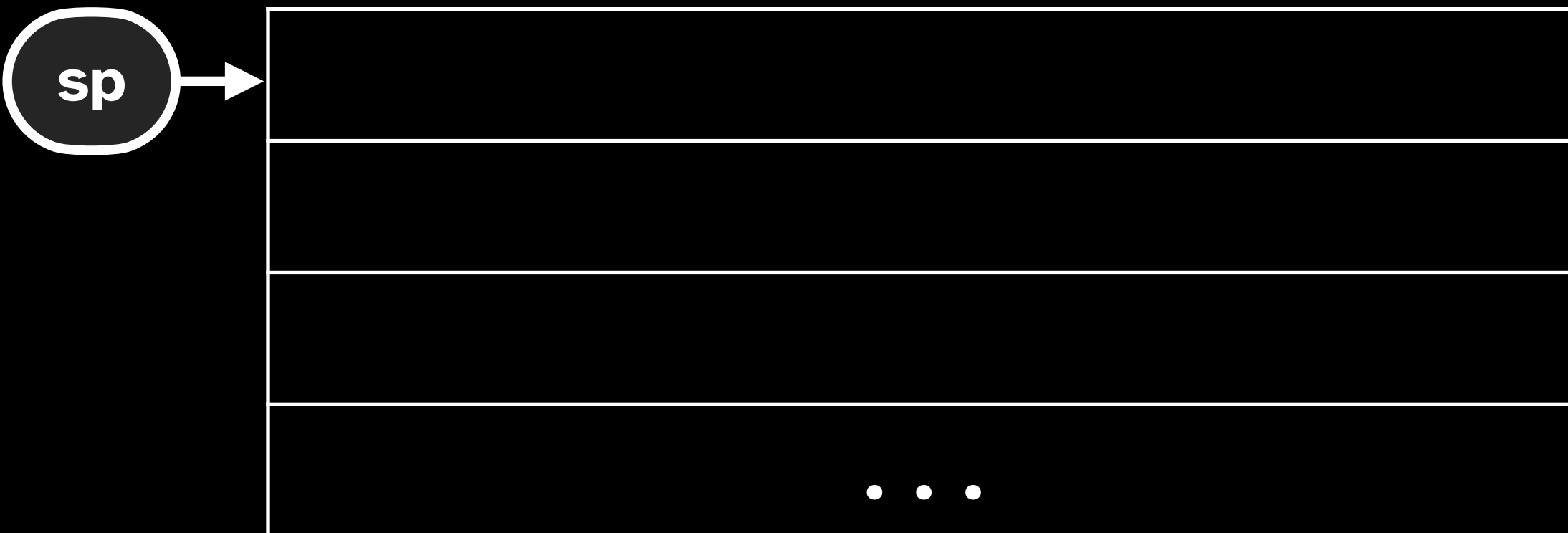
**Branch  
Predictor**

```
if (condition):  
    verified = AUT(guess ptr)  
    load(verified)
```

**5**

**Examine the eviction set and see  
if any lines were evicted.**

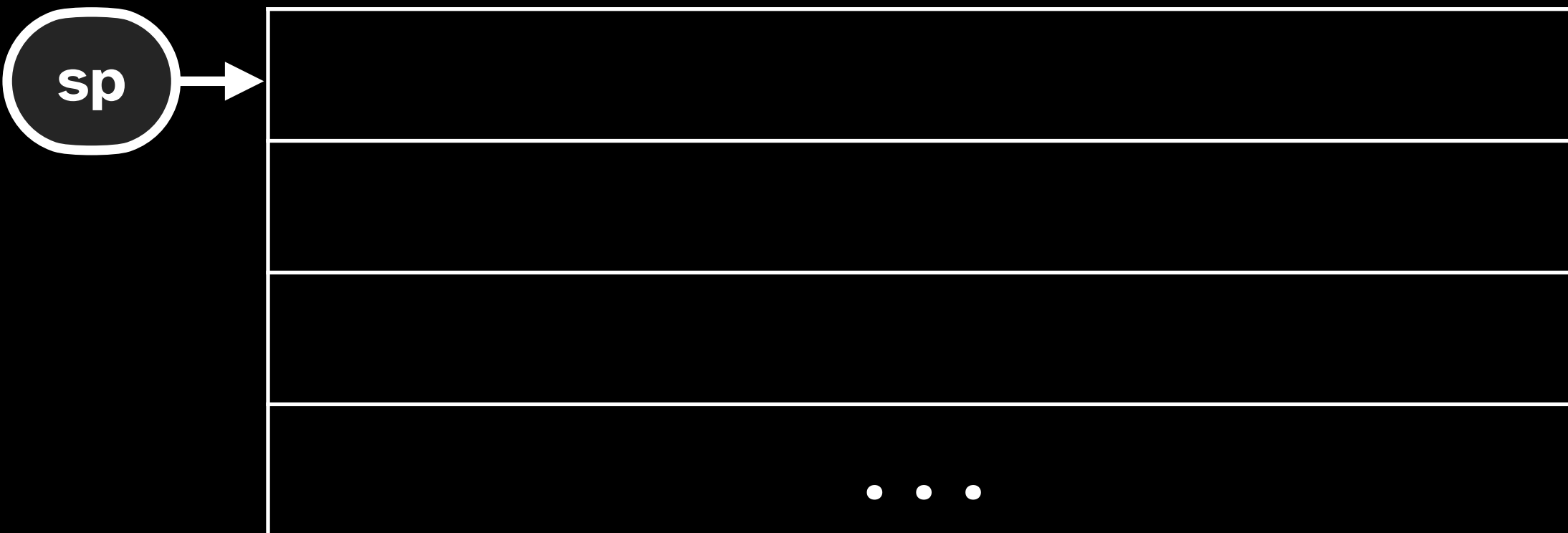
# Buffer Overflow **With PACMAN**



```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

```
_vulnerable:  
paciaz lr  
sub  sp, sp, #48  
stp  fp, lr, [sp, #32]  
...  
ldp  fp, lr, [sp, #32]  
add  sp, sp, #48  
autiaz lr  
ret
```

# Buffer Overflow With PACMAN



```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

\_vulnerable:

paciza lr

sub sp, sp, #48

stp fp, lr, [sp, #32]

...

ldp fp, lr, [sp, #32]

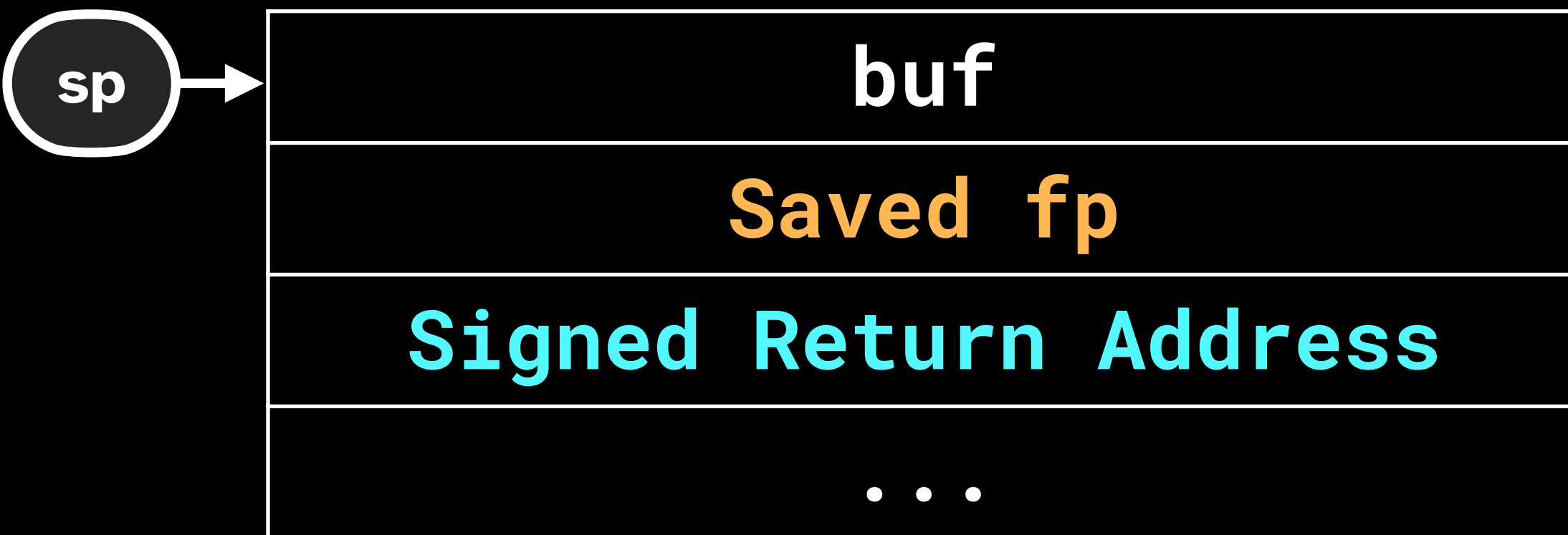
add sp, sp, #48

autiza lr

ret

**We begin by signing the return address.**

# Buffer Overflow **With PACMAN**



```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

\_vulnerable:

paciza lr

sub sp, sp, #48

stp fp, lr, [sp, #32]

...

ldp fp, lr, [sp, #32]

add sp, sp, #48

autiza lr

ret

**Storing the *signed* pointer onto the stack.**

# Buffer Overflow **With PACMAN**



```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

```
_vulnerable:  
paciza lr  
sub  sp, sp, #48  
stp  fp, lr, [sp, #32]  
...  
ldp  fp, lr, [sp, #32]  
add  sp, sp, #48  
autiza lr  
ret
```

**Overflow allows the attacker to overwrite the return address.**

# Buffer Overflow **With PACMAN**

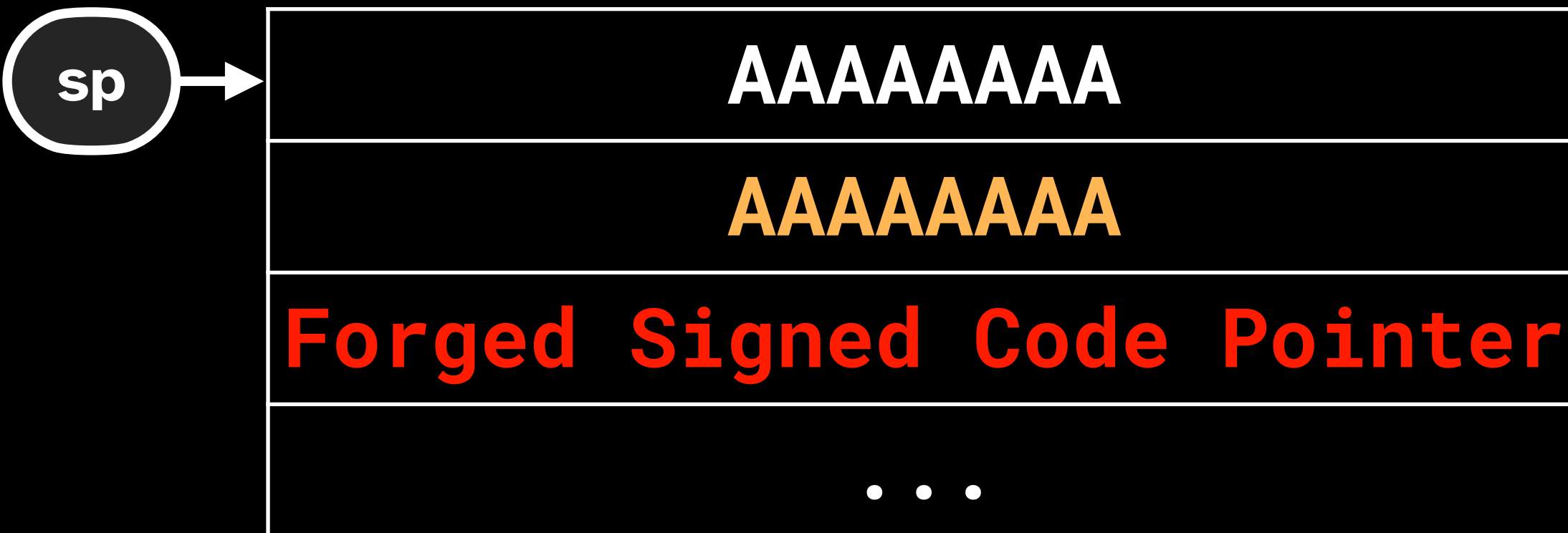


```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

```
_vulnerable:  
paciza lr  
sub    sp, sp, #48  
stp    fp, lr, [sp, #32]  
...  
ldp    fp, lr, [sp, #32]  
add    sp, sp, #48  
autiza lr  
ret
```

**The attacker, using read/ write primitives, triggers a PACMAN attack and brute forces all possible PAC values for the desired pointer.**

# Buffer Overflow **With PACMAN**

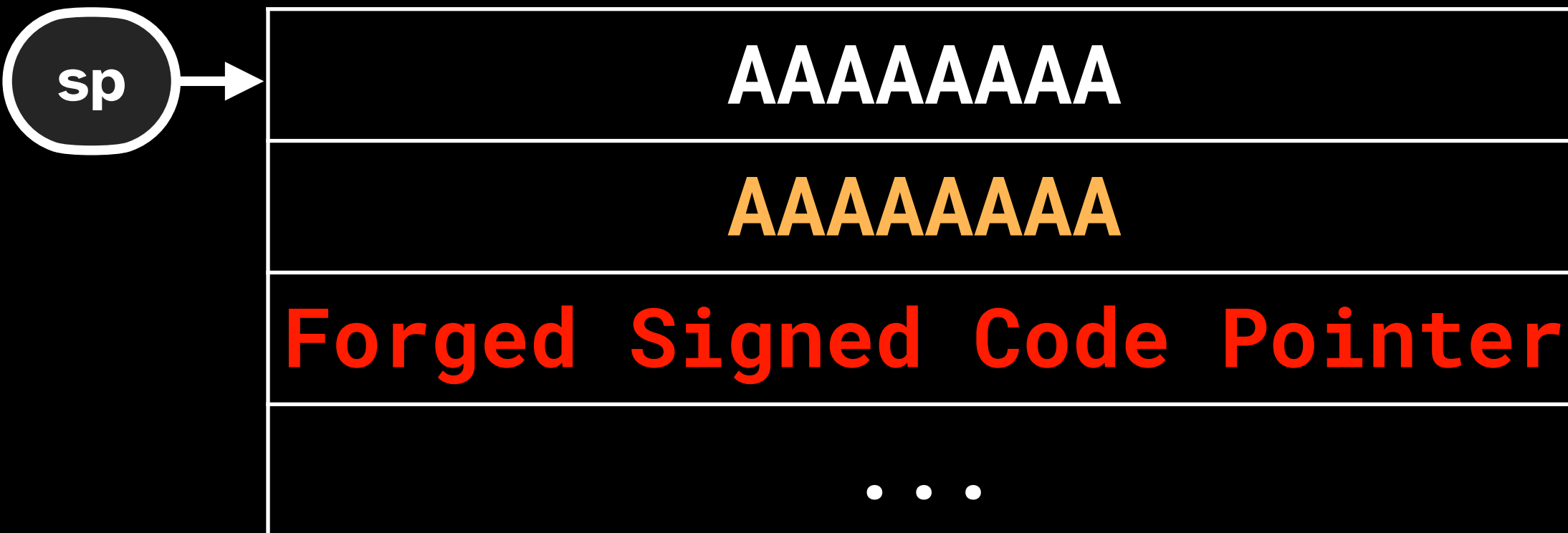


```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

```
_vulnerable:  
paciza lr  
sub    sp, sp, #48  
stp    fp, lr, [sp, #32]  
...  
ldp    fp, lr, [sp, #32]  
add    sp, sp, #48  
autiza lr  
ret
```

**Using the result of the PACMAN attack, we insert a forged signed pointer to the stack.**

# Buffer Overflow **With PACMAN**



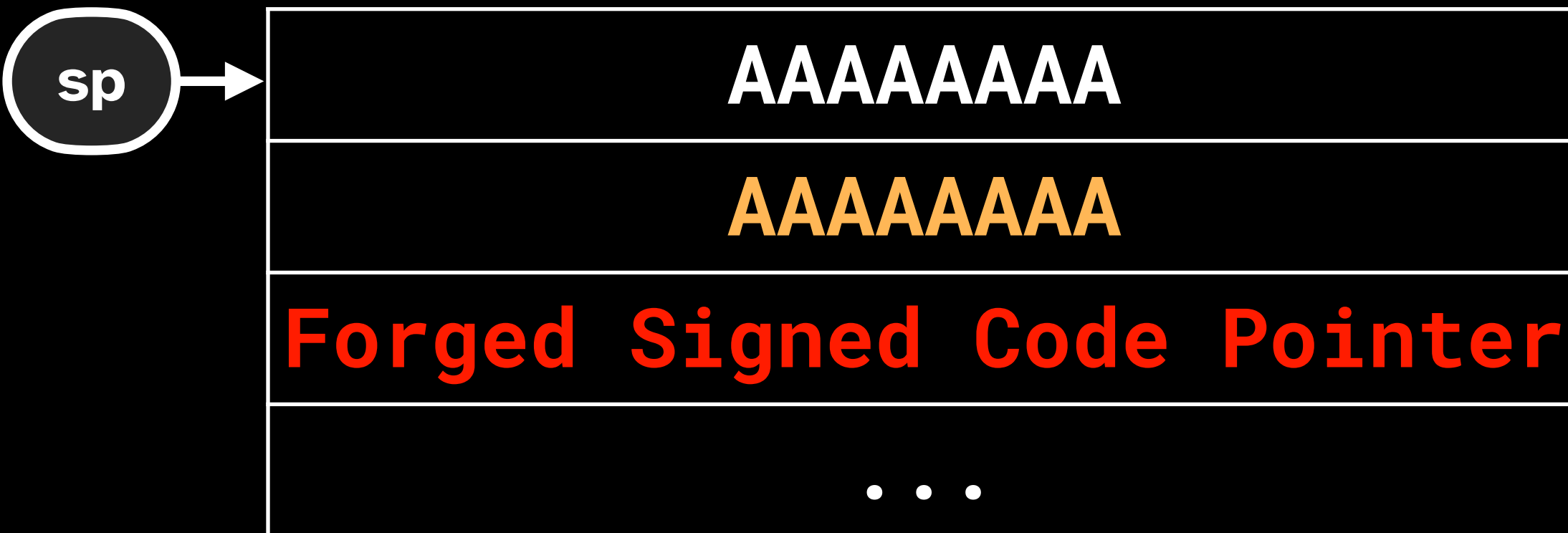
```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

```
_vulnerable:  
paciza lr  
sub  sp, sp, #48  
stp  fp, lr, [sp, #32]  
...  
ldp  fp, lr, [sp, #32]  
add  sp, sp, #48  
autiza lr  
ret
```

**Verify instruction will succeed, and the attacker forged return address becomes a valid pointer.**



# Buffer Overflow With PACMAN



```
void vulnerable() {  
    char buf[16];  
    gets(buf);  
}
```

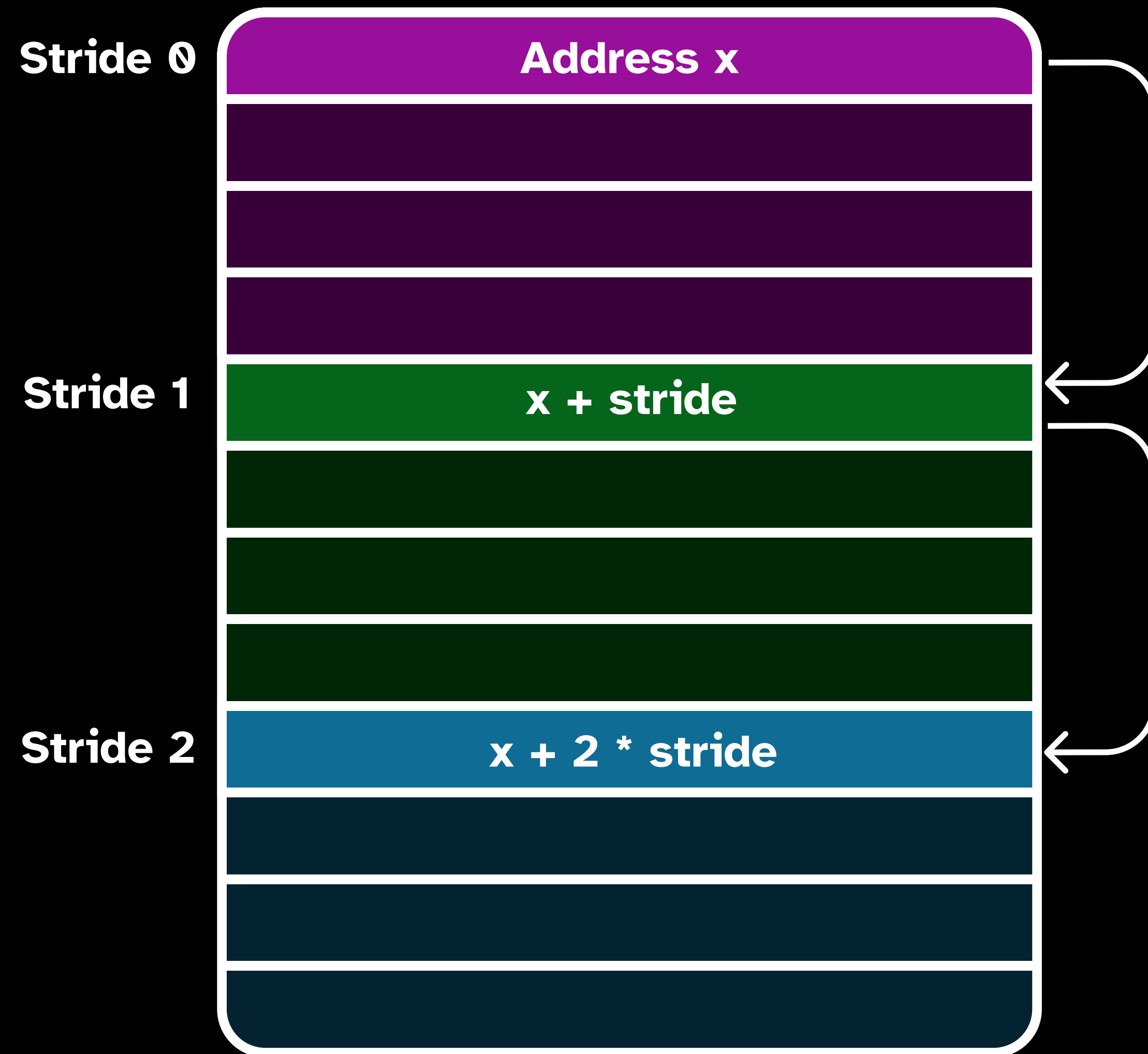
```
_vulnerable:  
paciza lr  
sub  sp, sp, #48  
stp  fp, lr, [sp, #32]  
...  
ldp  fp, lr, [sp, #32]  
add  sp, sp, #48  
autiza lr  
ret
```

**The attacker has gained arbitrary code execution.**

# **Experiment 3**

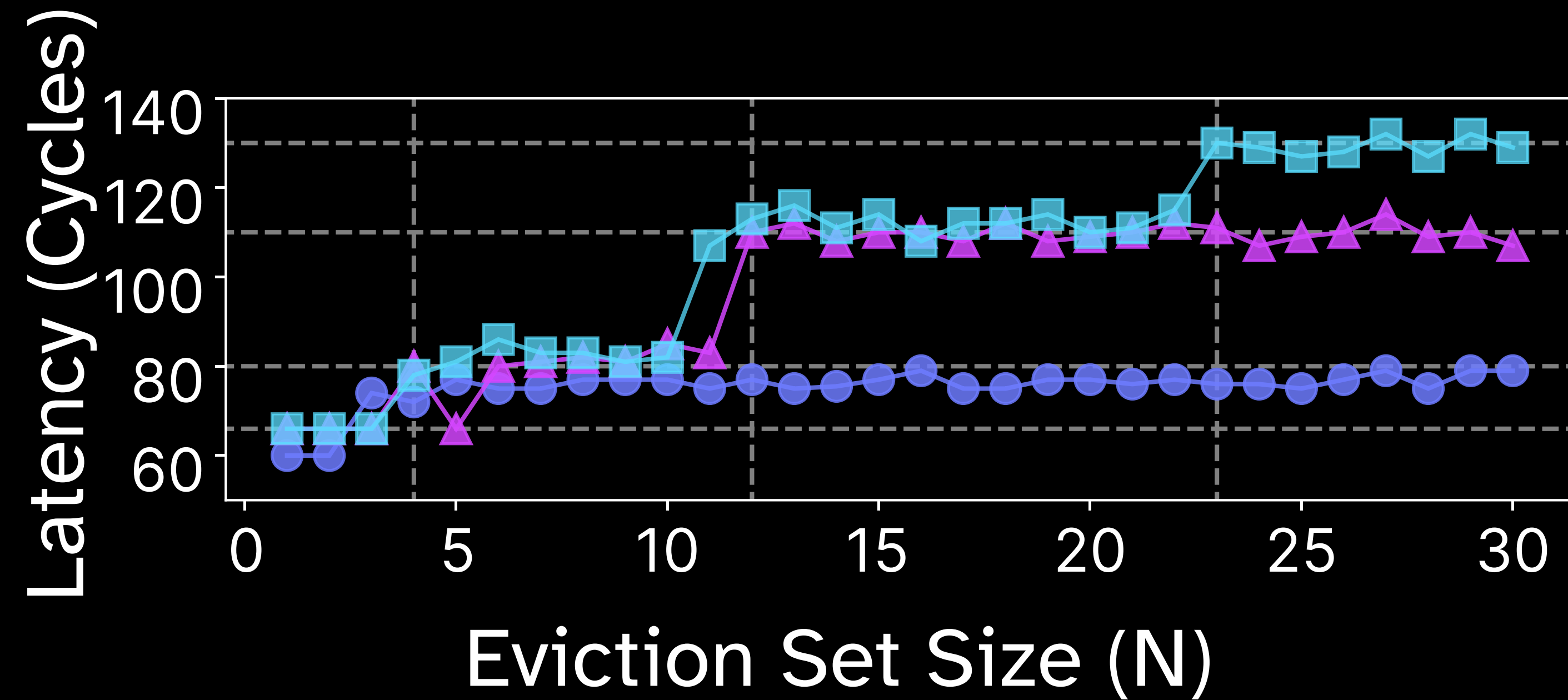
## **TLB and Cache Interactions**

# TLB + Cache Interactions



`addr[i] = x + (i*stride)`

# TLB + Cache Results



**Latency from dTLB/dCache conflicts**

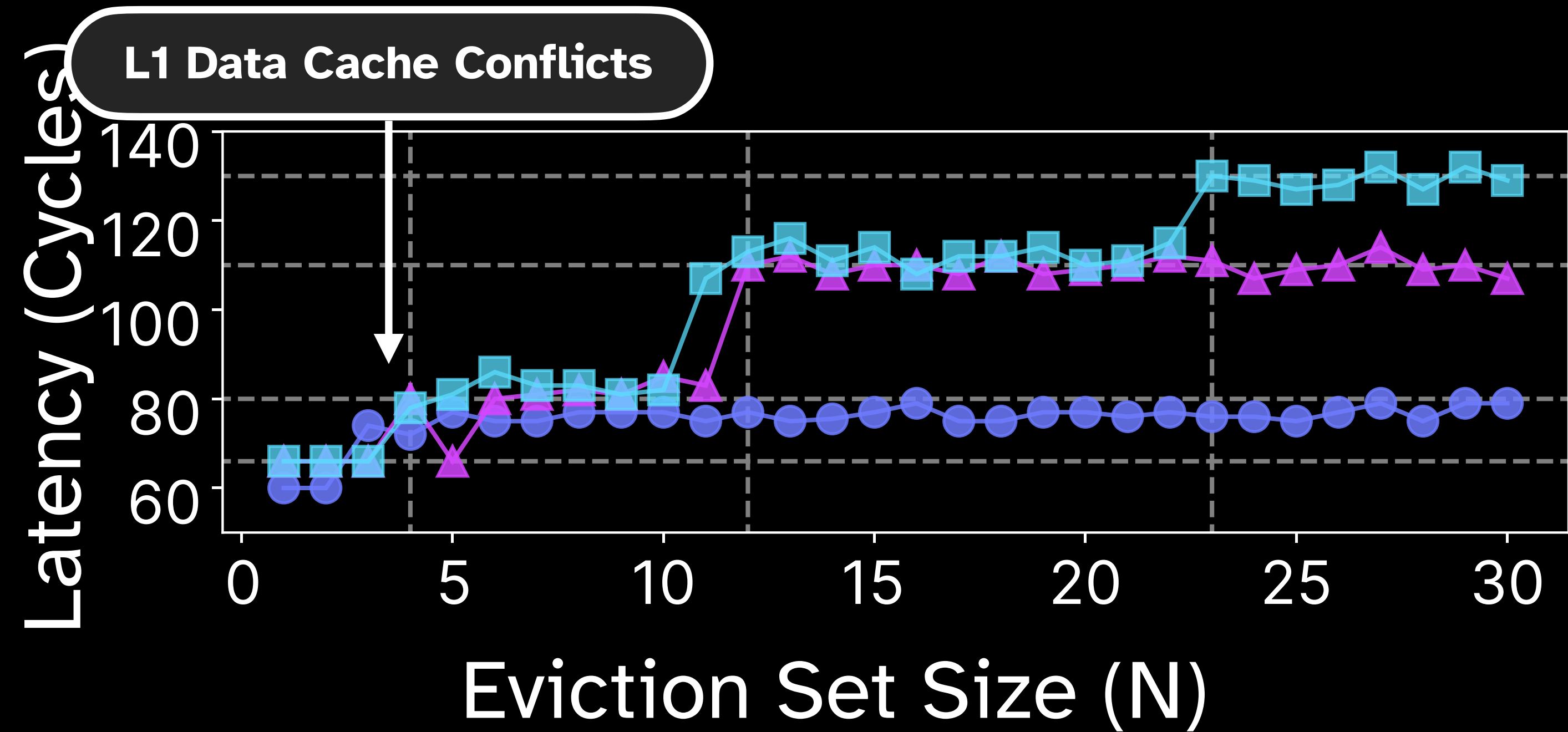
—▲— 256 x 16KB

—■— 2K x 16KB

—●— 256 x 128B

—×— 32 x 16KB

# TLB + Cache Results



**Latency from dTLB/dCache conflicts**

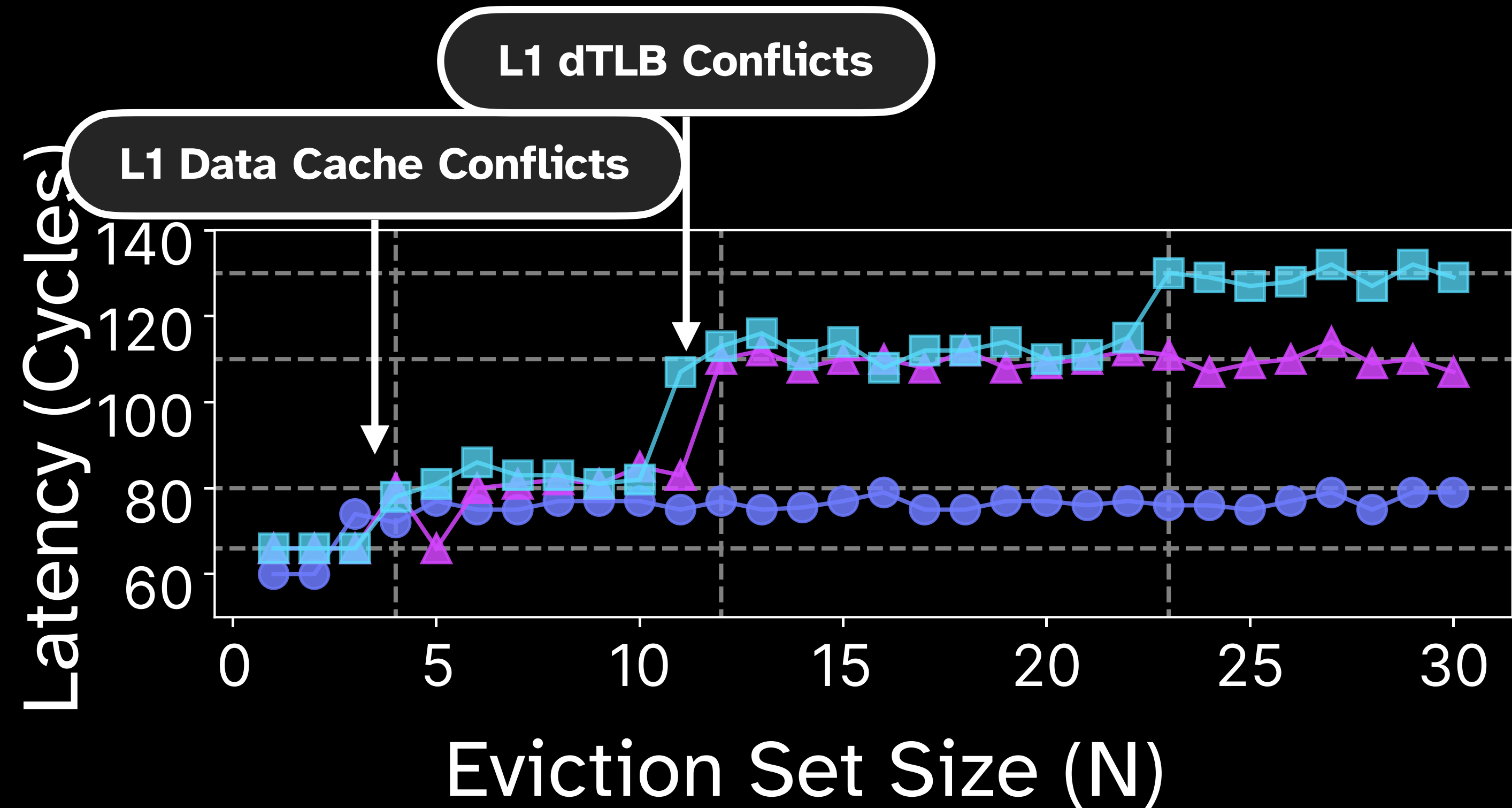
—▲— 256 x 16KB

—■— 2K x 16KB

—●— 256 x 128B

—×— 32 x 16KB

# TLB + Cache Results



Latency from dTLB/dCache conflicts

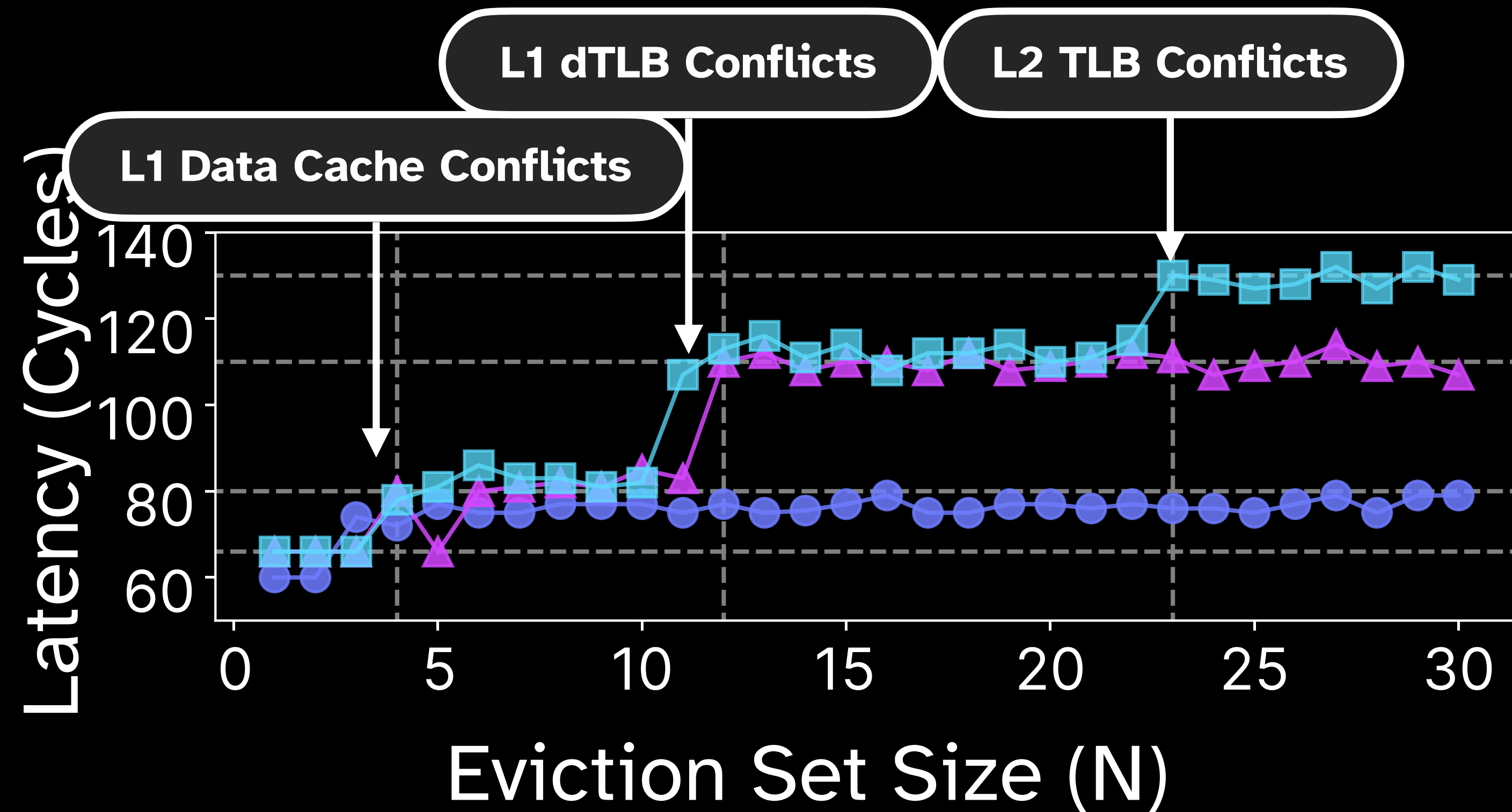
—▲— 256 x 16KB

—■— 2K x 16KB

—●— 256 x 128B

—×— 32 x 16KB

# TLB + Cache Results



Latency from dTLB/dCache conflicts

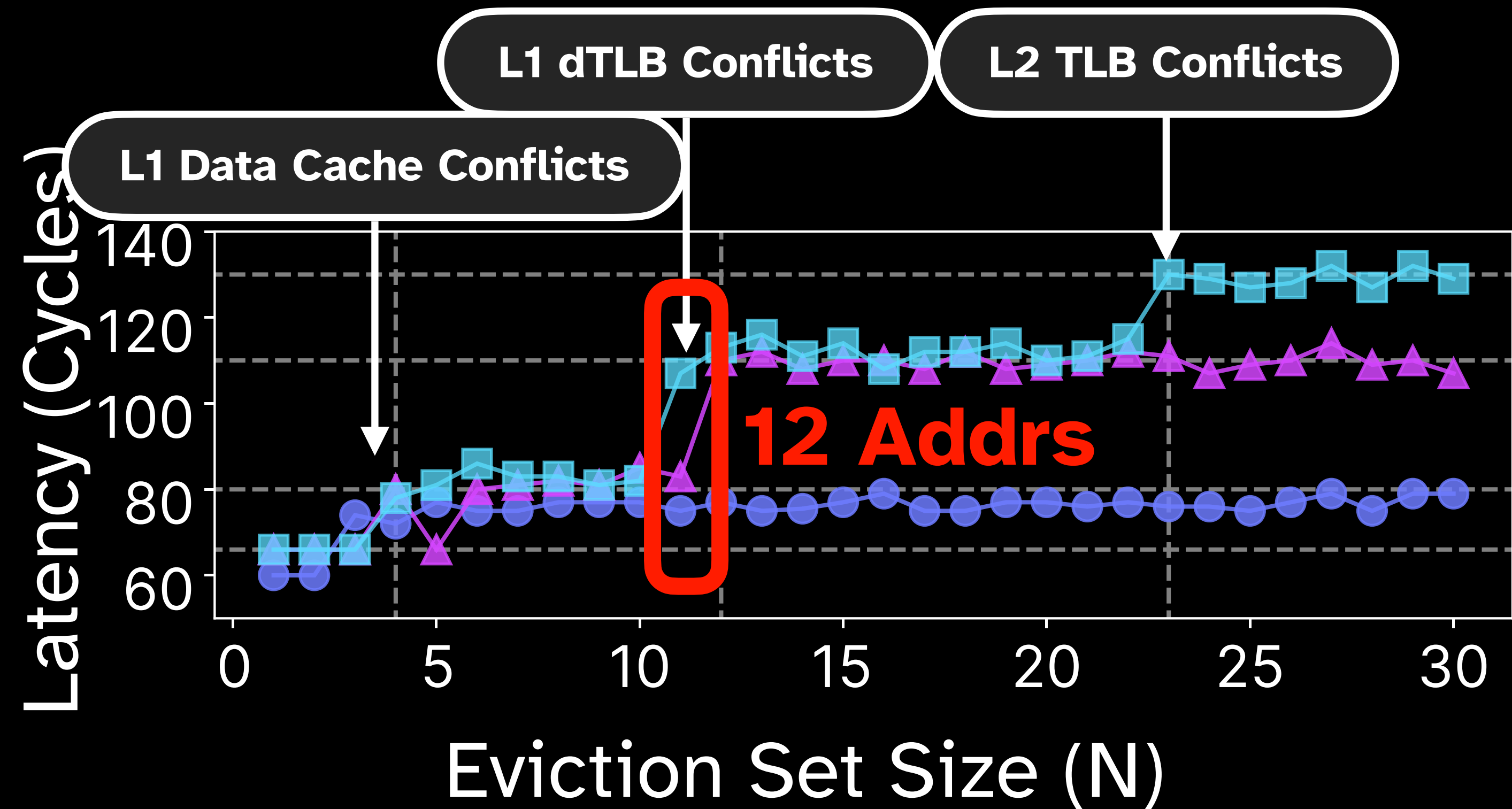
—▲— 256 x 16KB

—■— 2K x 16KB

—●— 256 x 128B

—×— 32 x 16KB

# TLB + Cache Results



Latency from dTLB/dCache conflicts

—▲— 256 x 16KB

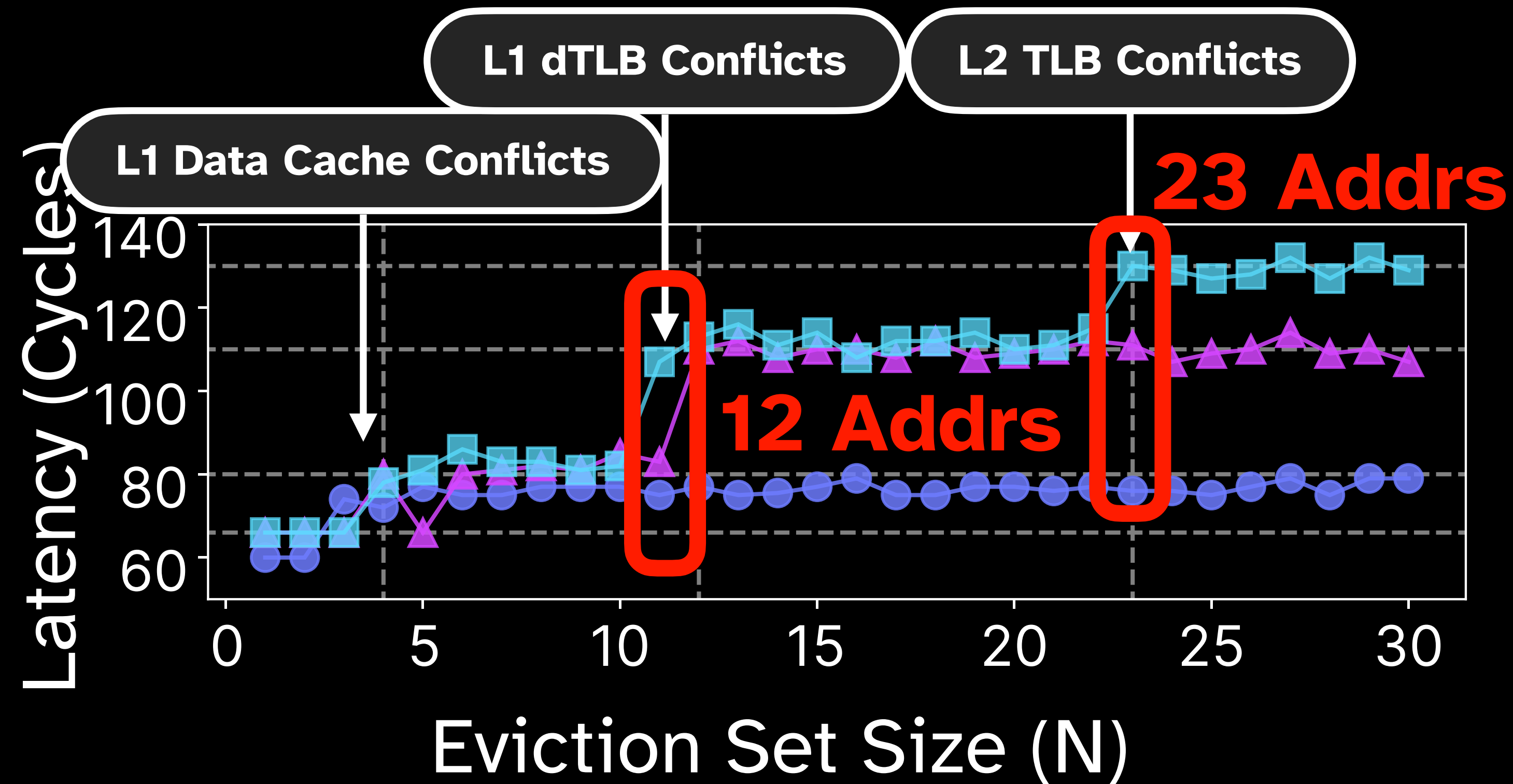
—■— 2K x 16KB

—●— 256 x 128B

—×— 32 x 16KB



# TLB + Cache Results



Latency from dTLB/dCache conflicts

▲ 256 x 16KB

■ 2K x 16KB

● 256 x 128B

✕ 32 x 16KB