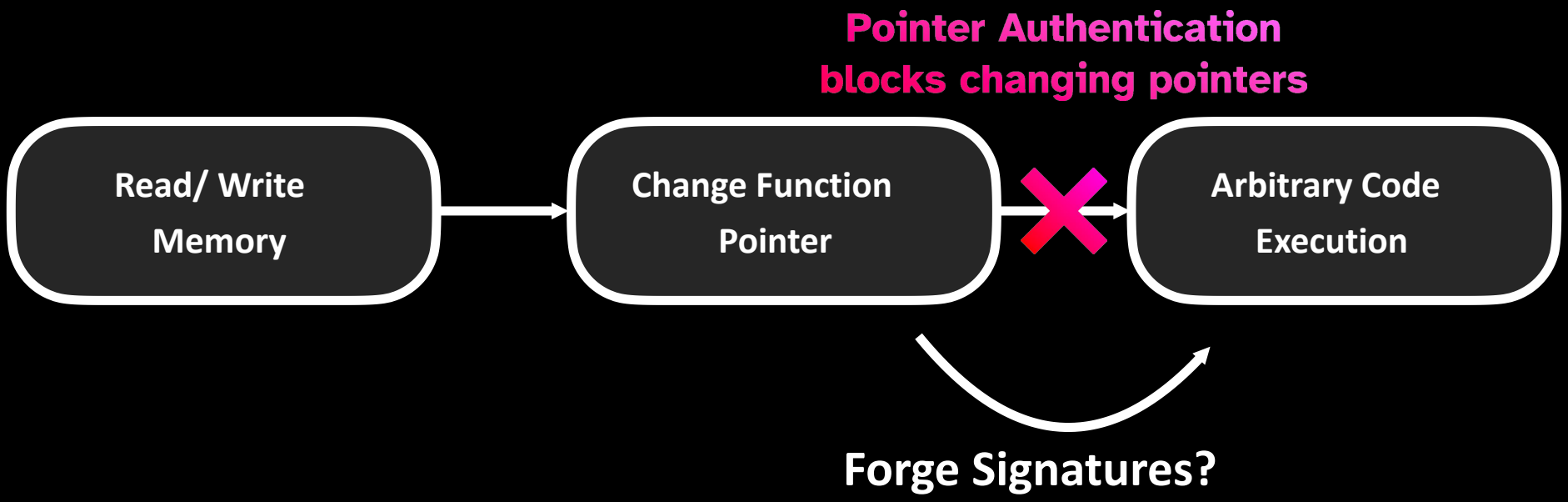
PACMAN!

Breaking PAC on the Apple M1 with Hardware Attacks.

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

Memory Corruption

Memory Corruption



Just bruteforce it, right?

Key Insight:

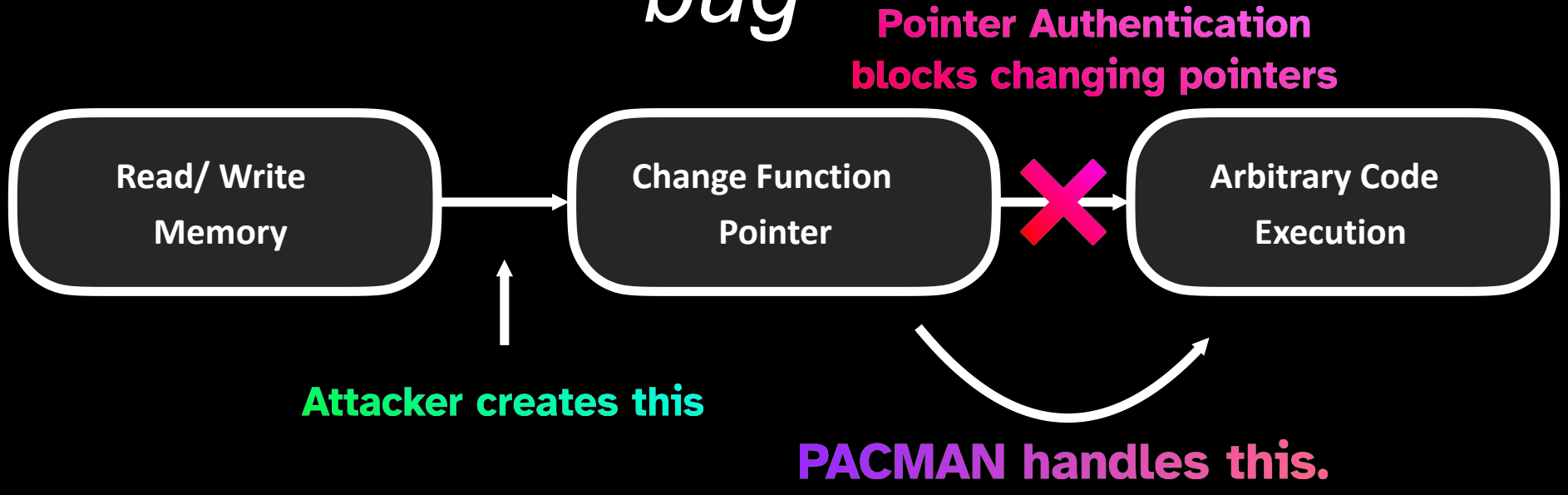
Avoid crashes using speculative execution!

BYOB

"Bring your own bug"

BYOB

"Bring your own bug"

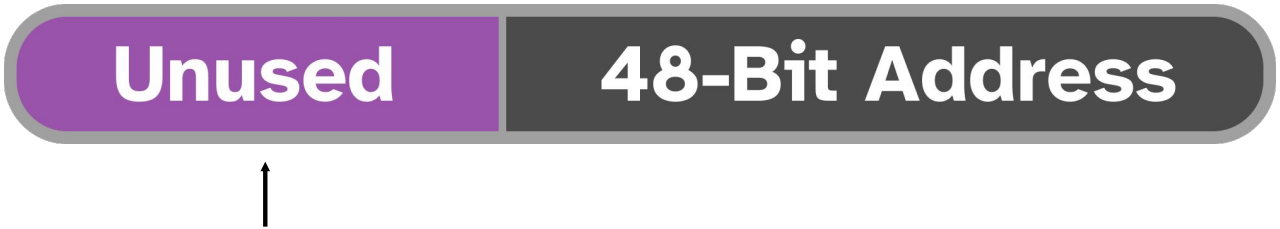


Pointer

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

Pointer

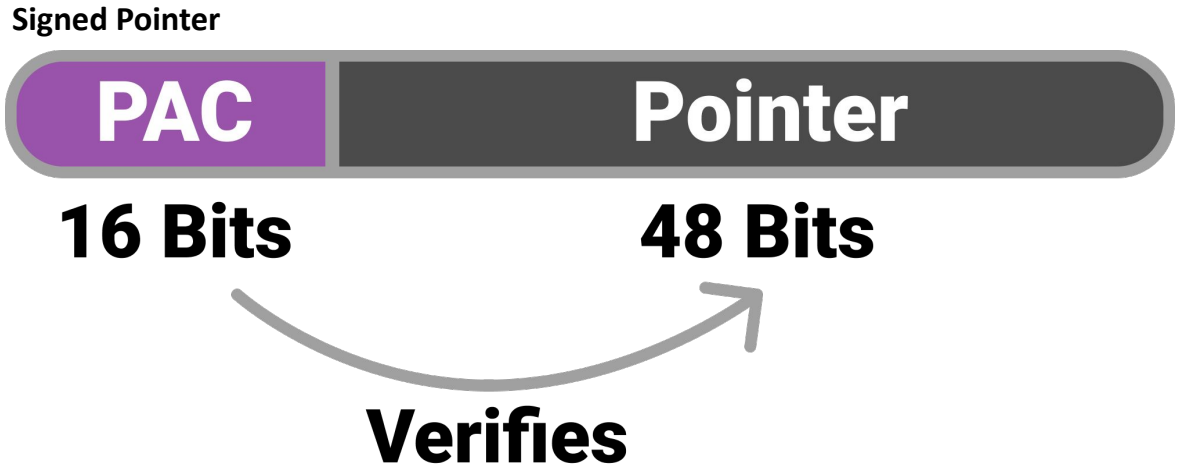
Pointer



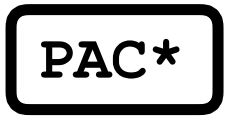
Let's put this to good use!

Pointer Authentication

```
PAC = hash(pointer, salt, key)
```



Pointer Authentication



Insert a PAC

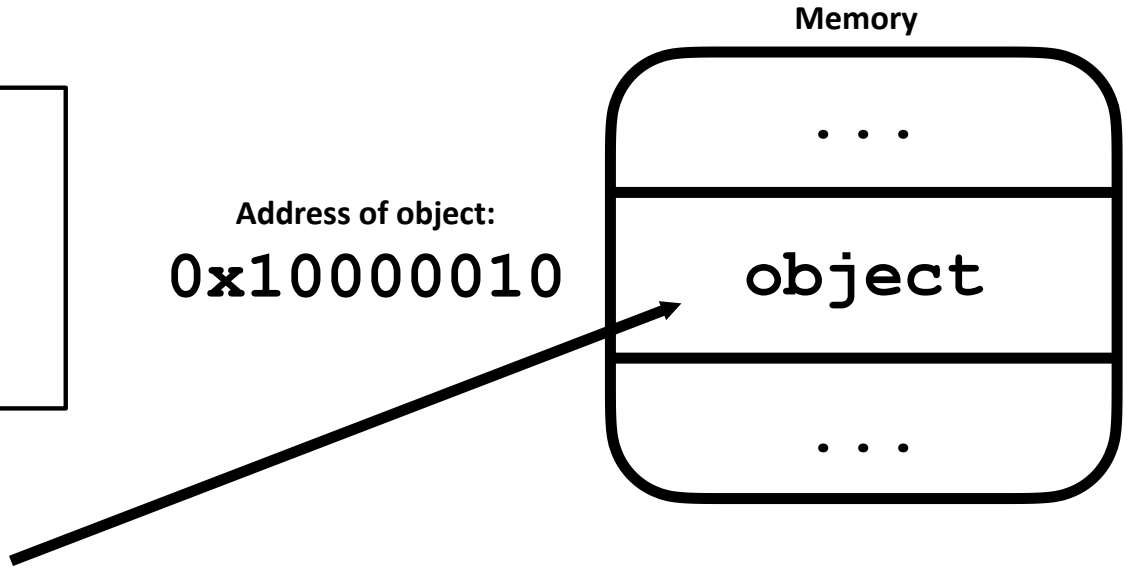
Check a PAC

Don't crash on AUT- crash on use

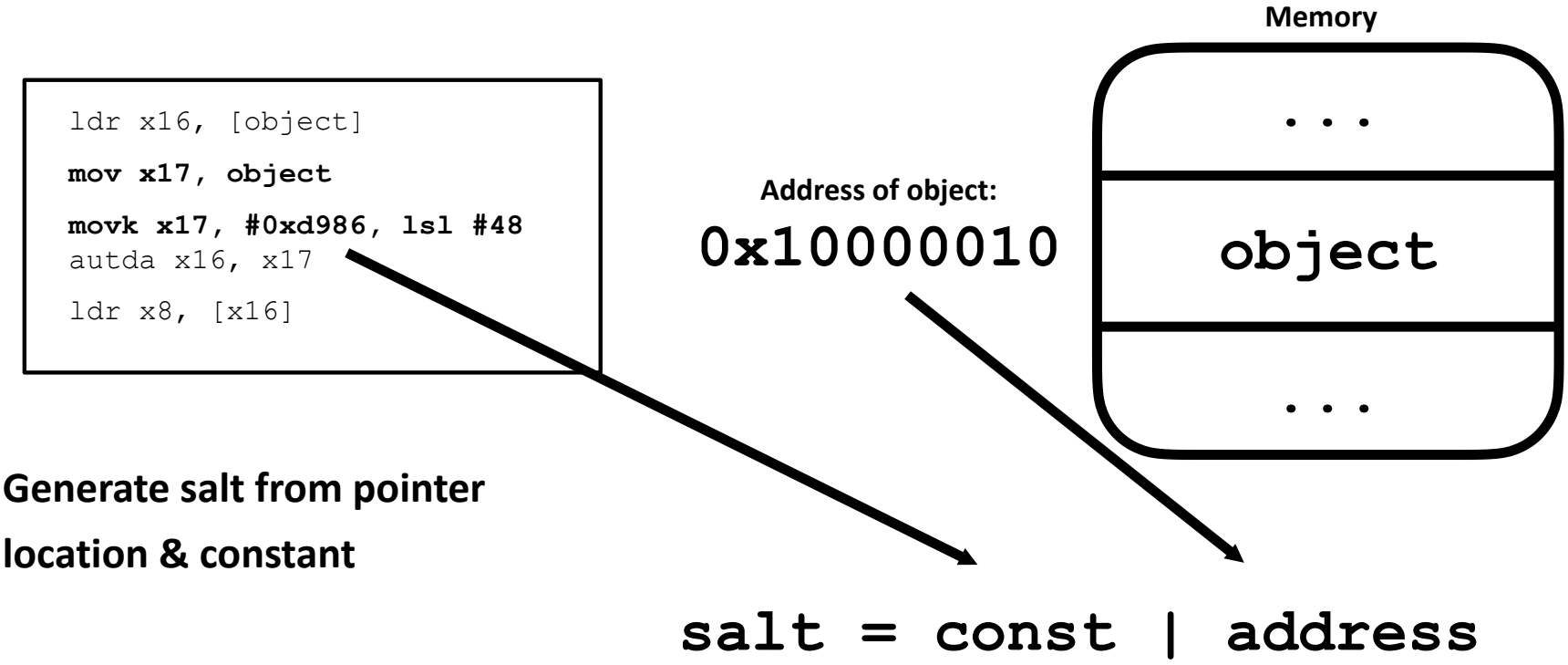
Pointer Authentication

```
ldr x16, [object]

mov x17, object
movk x17, #0xd986, lsl #48

autda x16, x17

ldr x8, [x16]
```

Load signed pointer from memory



Pointer Authentication

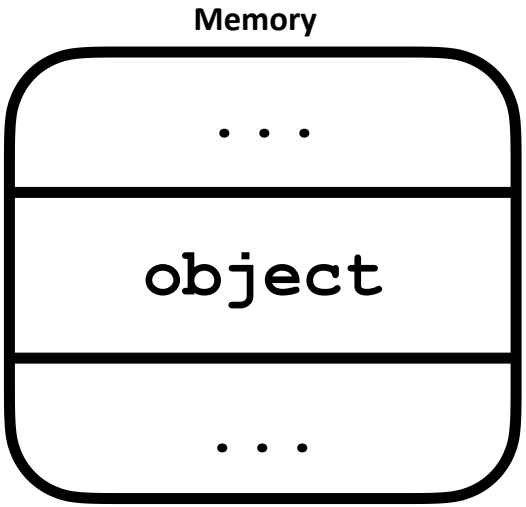


Pointer Authentication

```
ldr x16, [object]
mov x17, object
movk x17, #0xd986, lsl #48
autda x16, x17
ldr x8, [x16]
```

Address of object:

0x10000010



Verify signature- store invalid pointer in x16 if invalid

Pointer Authentication

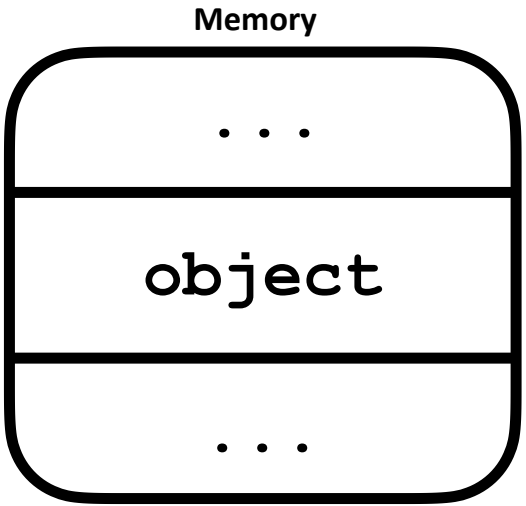
```
ldr x16, [object]
mov x17, object
movk x17, #0xd986, lsl #48
autda x16, x17
ldr x8, [x16]
```

Attempt to load

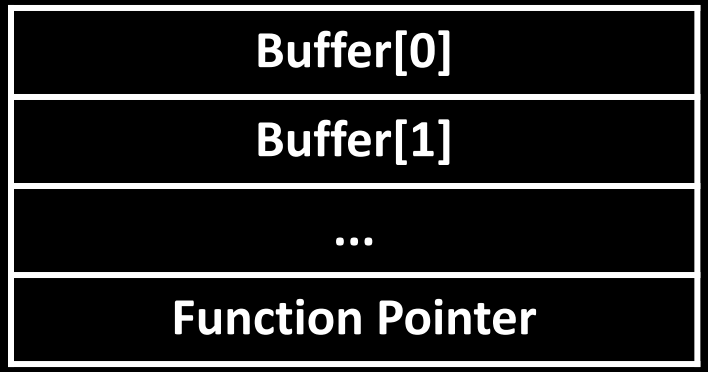
This is where the crash will happen!

Address of object:

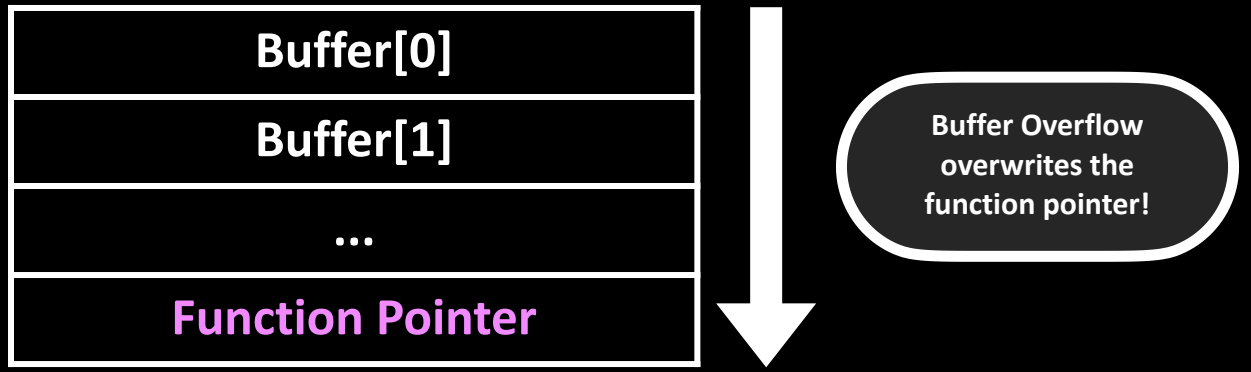
0x10000010



Buffer Overflow

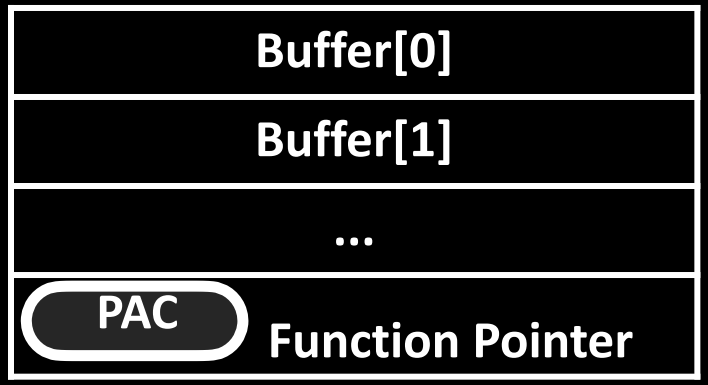


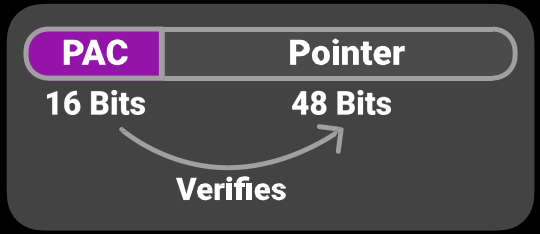
Buffer Overflow



Let's fix this bug with Pointer Authentication.

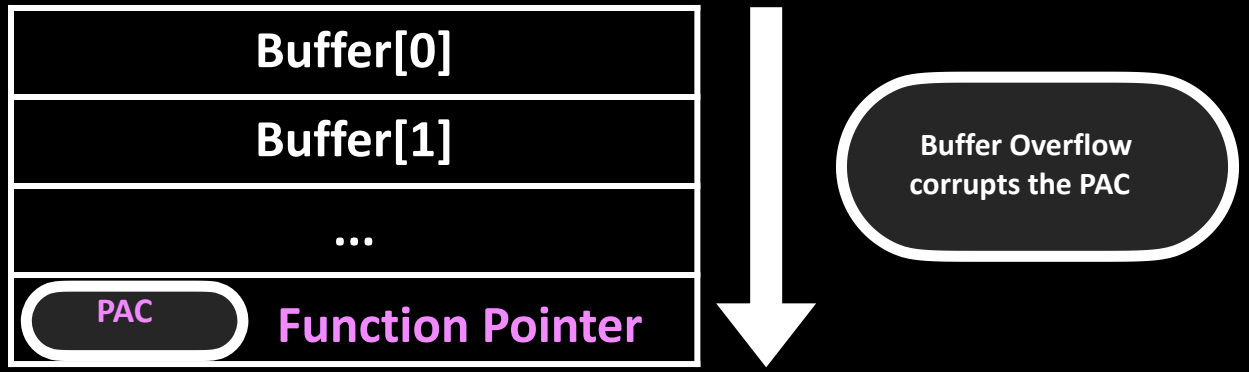
Buffer Overflow

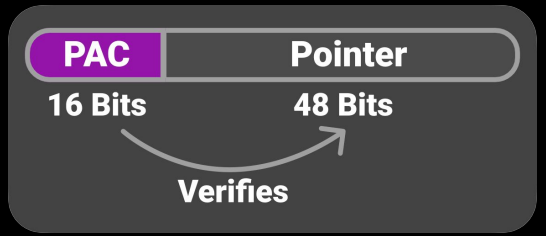




Buffer Overflow

Invalid PAC means we crash!





THE

GOAL

Reveal the PAC for an arbitrary pointer without crashing.

Break PAC with Hardware Attacks

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

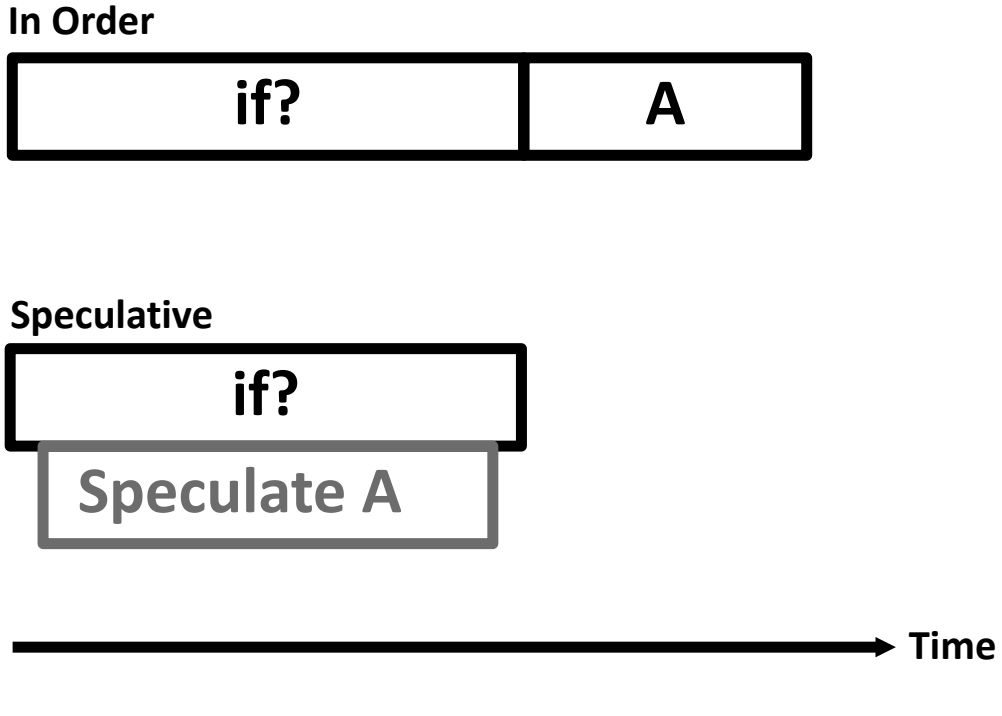
Speculative Execution

```
if (true)
    A
else
    B
```

In Order

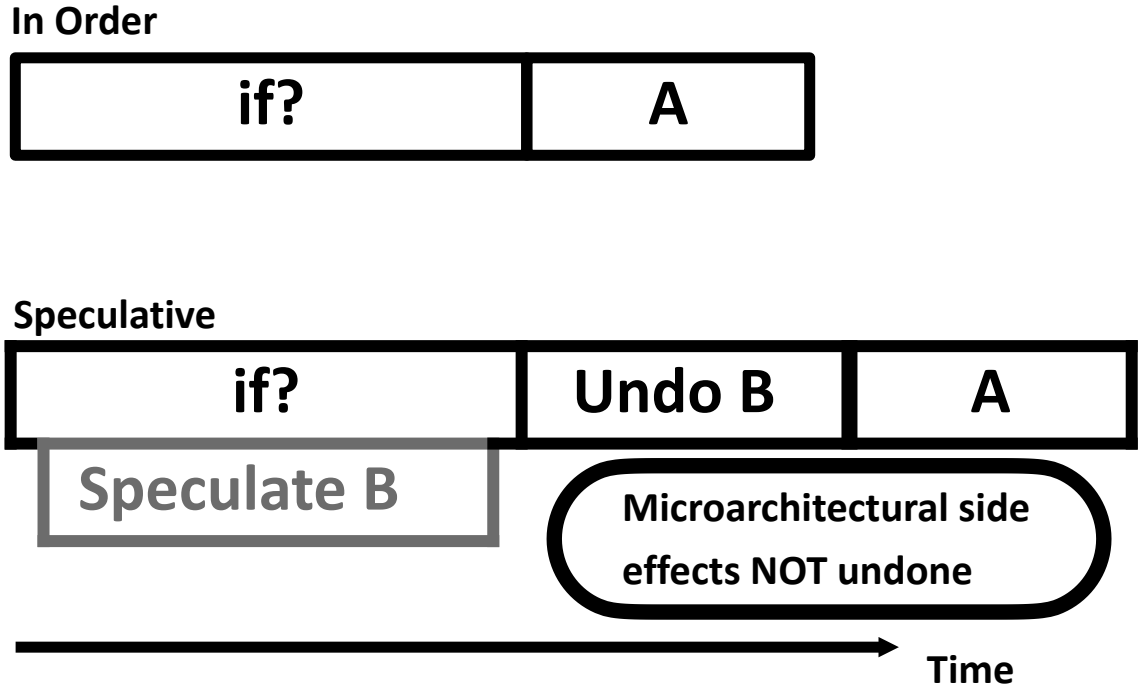
Speculative Execution

```
if (true)
    A
else
    B
```



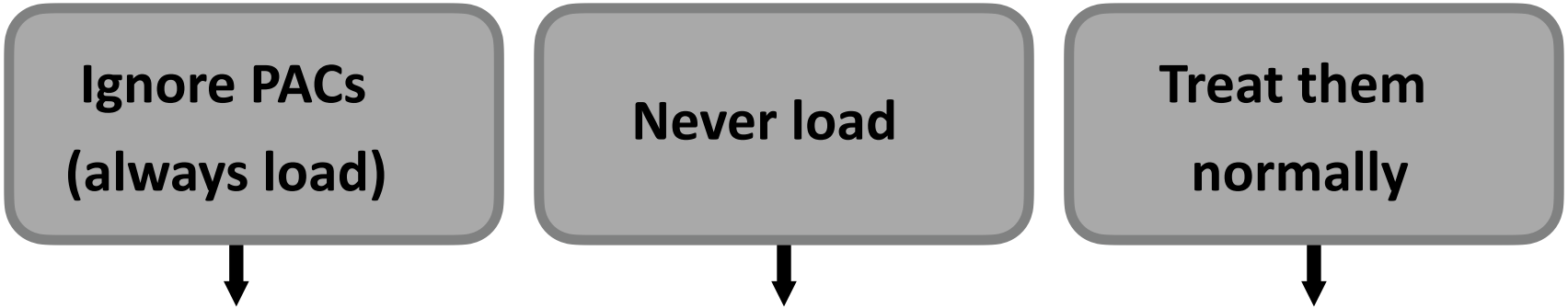
Speculative Execution

```
if (true)
    A
else
    B
```



How should we speculate on PAC values?

3 cases


Ignore PACs (always load)
Never load
Treat them normally
Leads to other security issues!
Slow!
This is how M1 does it.

Speculative PACs

```
if (true)
    return
else
    check signed ptr
    x = load signed ptr
```

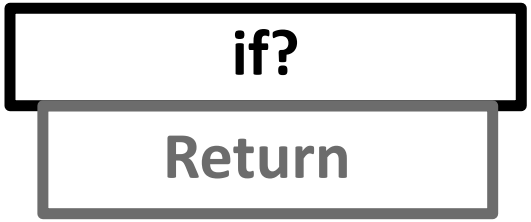
In Order

Doesn't leak anything

Speculative PACs

```
if (true)
    return
else
    check signed ptr
    x = load signed ptr
```

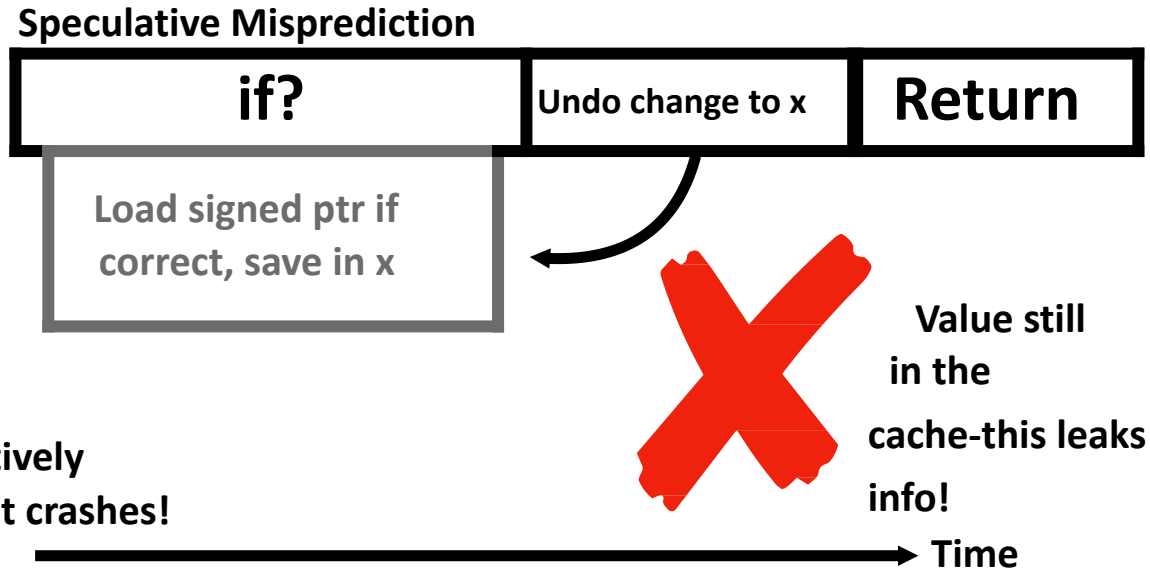
Speculative Correct Prediction



Doesn't leak anything

Speculative PACs

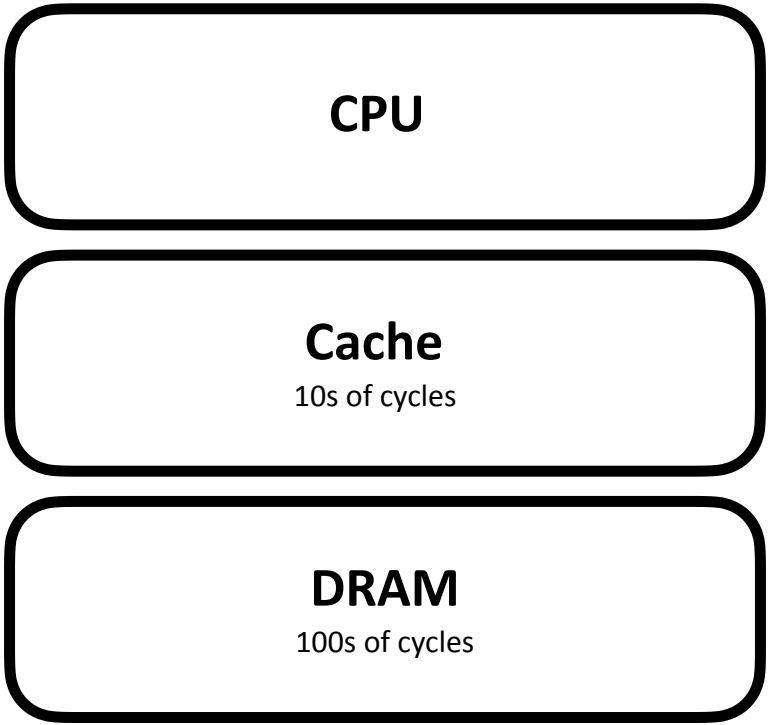
```
if (true)
    return
else
    check signed ptr
    x = load signed ptr
```



Operating on PACs speculatively can leak PAC correctness without crashes!

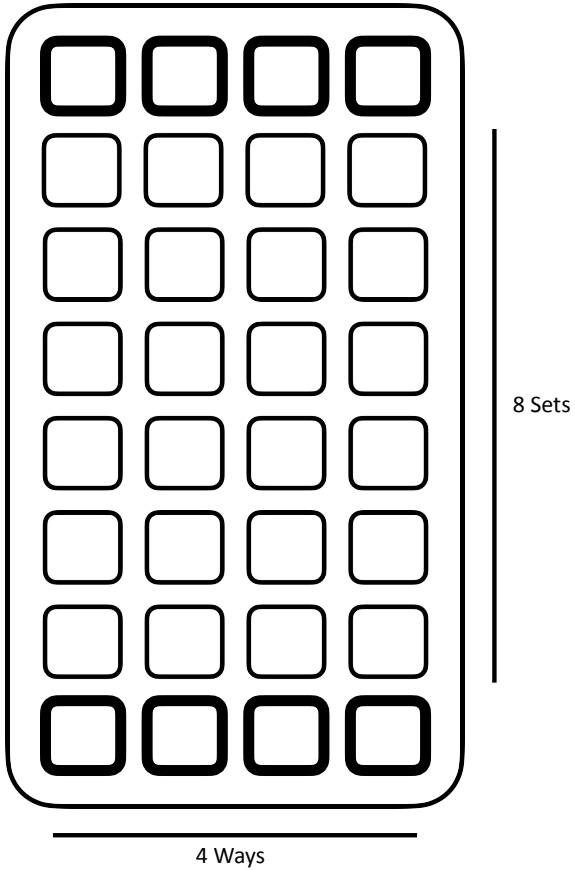
Bird's Eye View

Memory is slow...

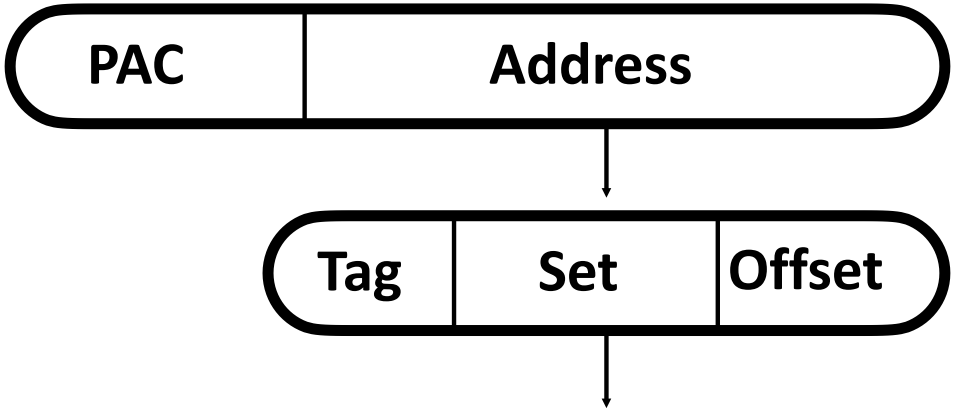


...so add some faster memory!

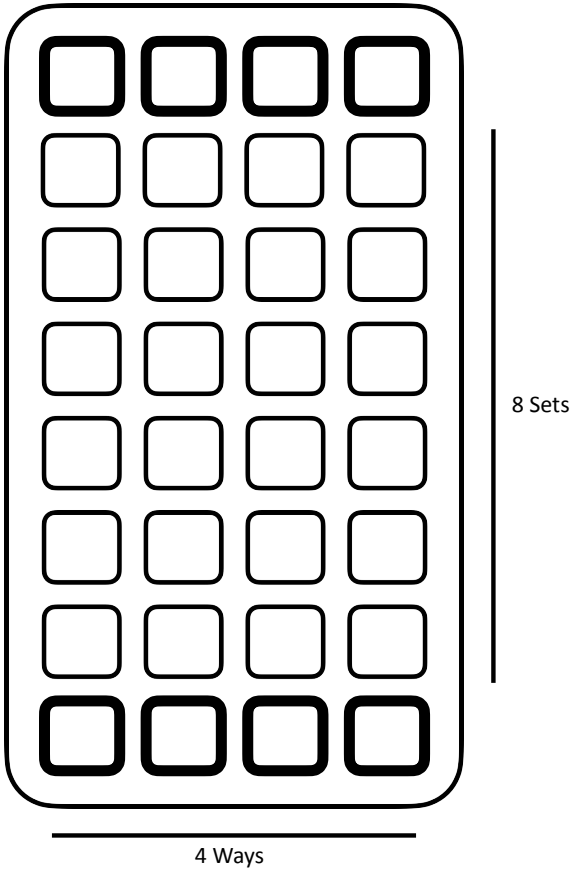
Cache

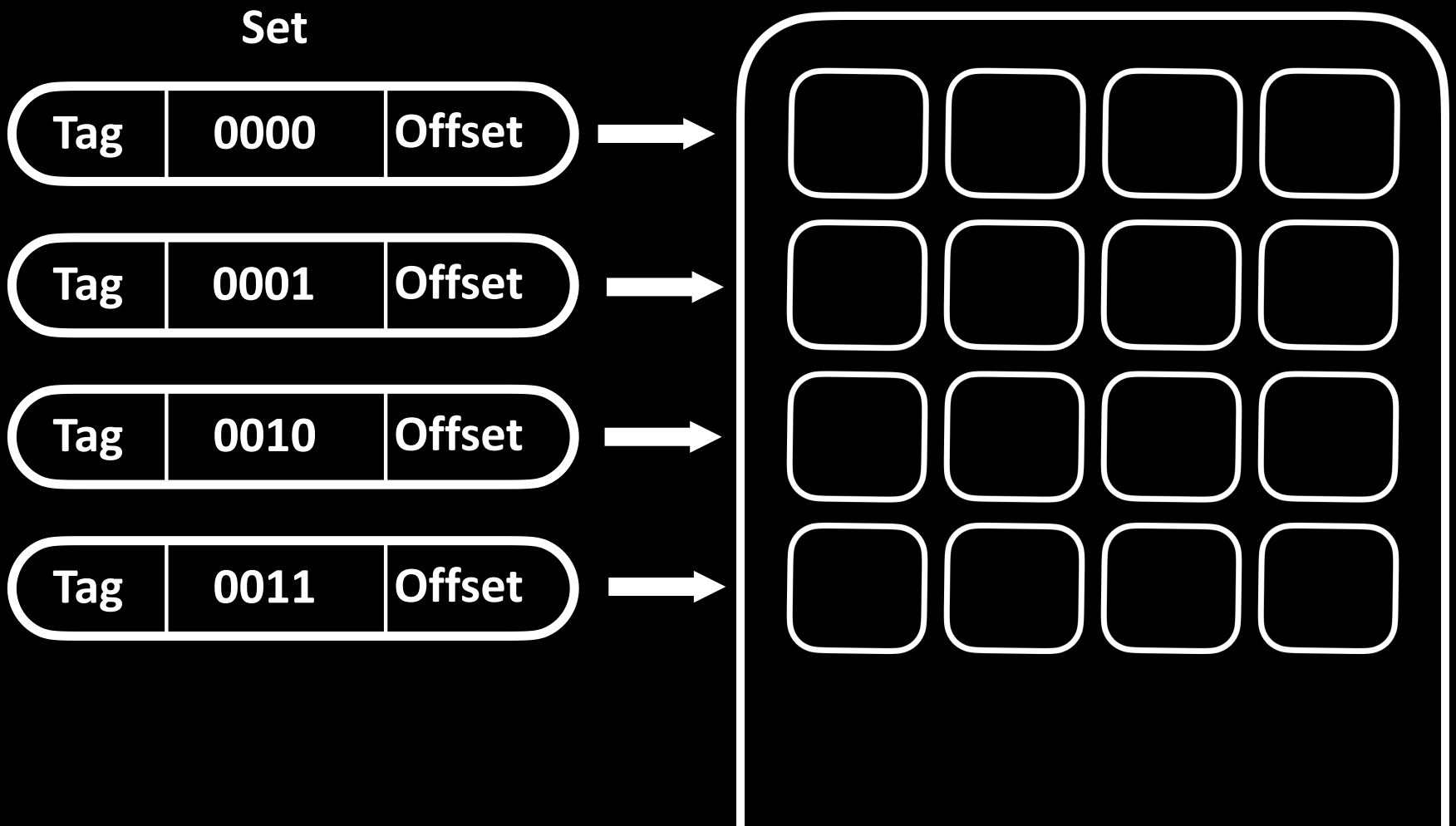


Cache

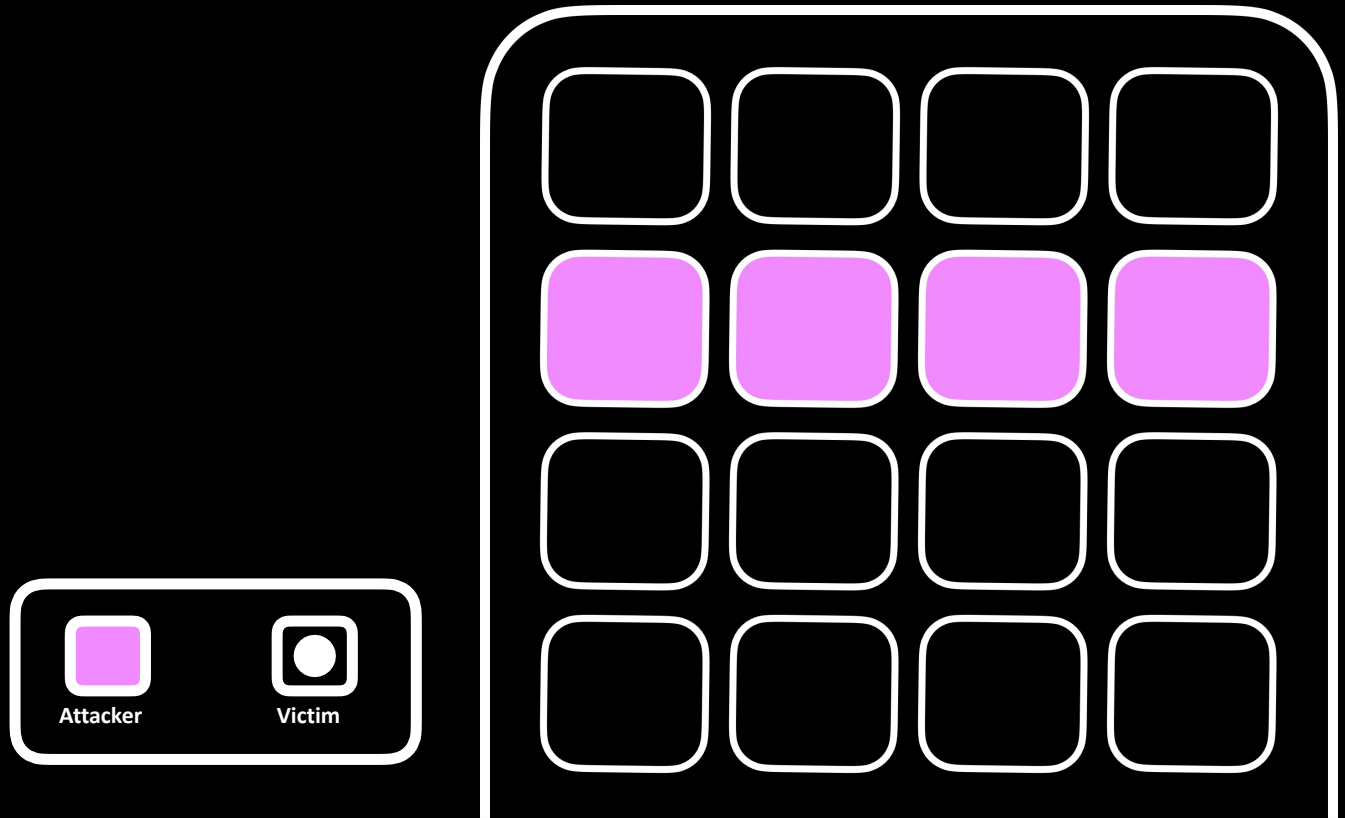


Which row (aka "set") do we map to?

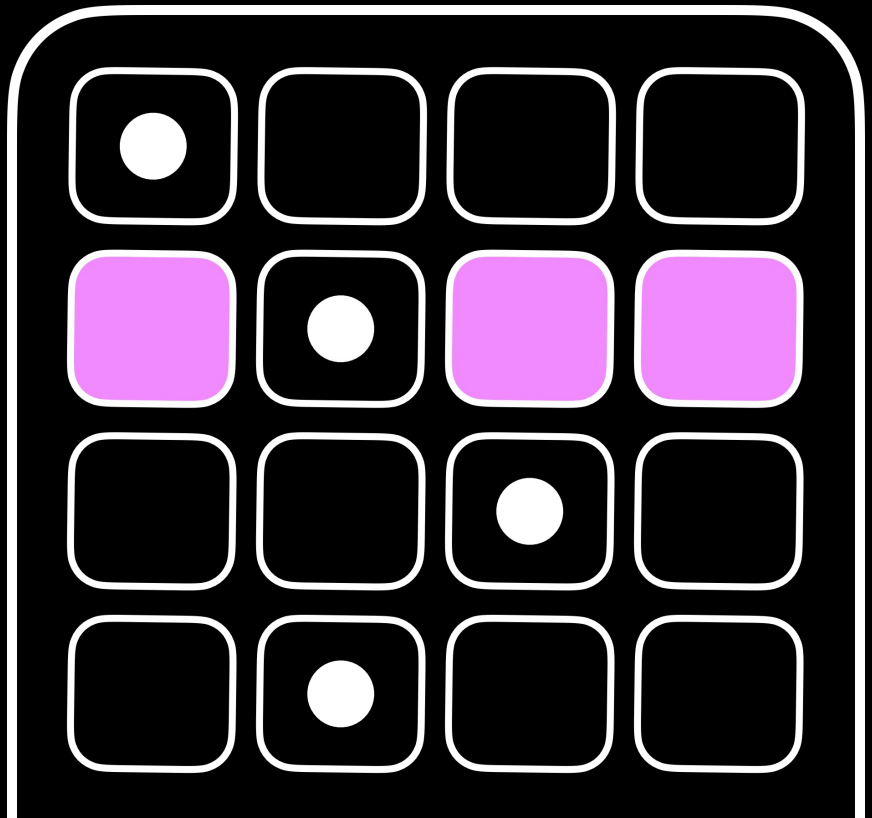


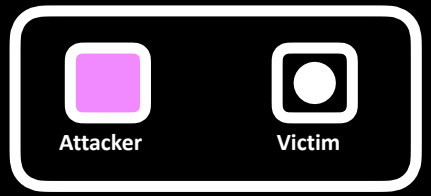

Set
Tag
0000
Offset
Tag
0001
Offset
Tag
0010
Offset
Tag
0011
Offset

1 Fill a set with our data.



2 Let the victim run.





3 Re-access our data.

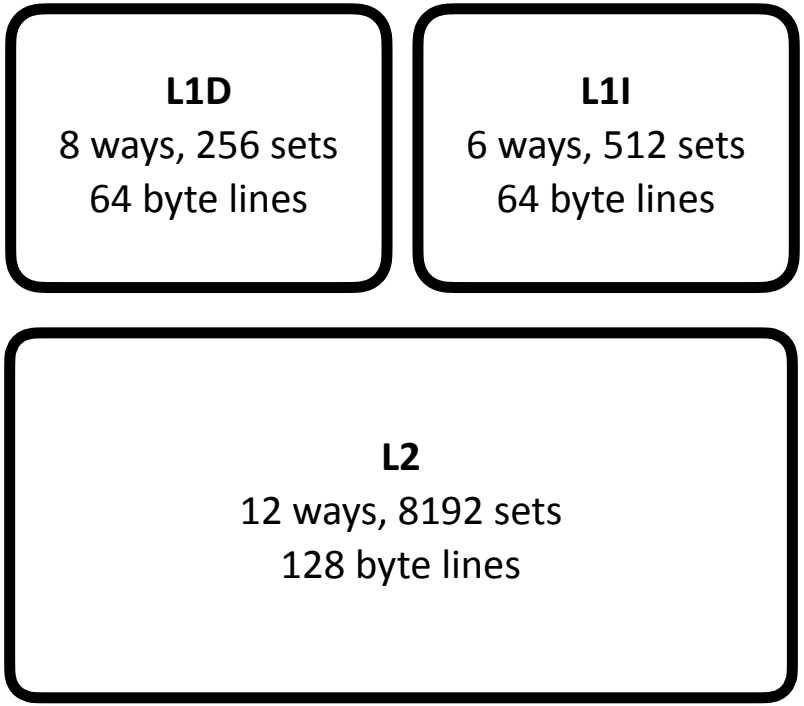
We can tell what the victim did by just watching the cache!

1 line is missing!

Attacker

Victim

M1 High Performance Cores

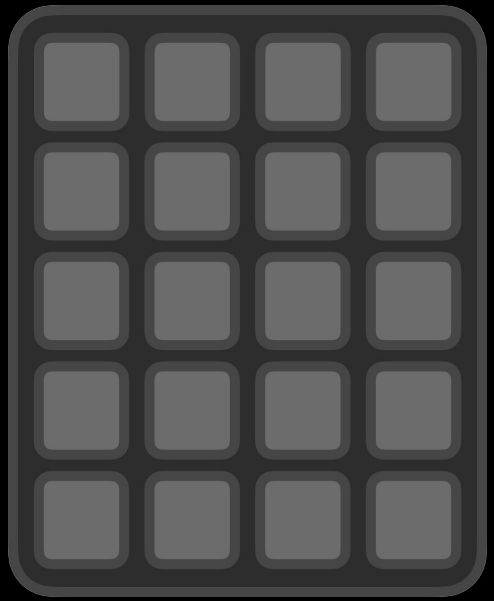


PACMAN Gadget

Speculative check & use of a signed pointer.

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

Data PACMAN Attack

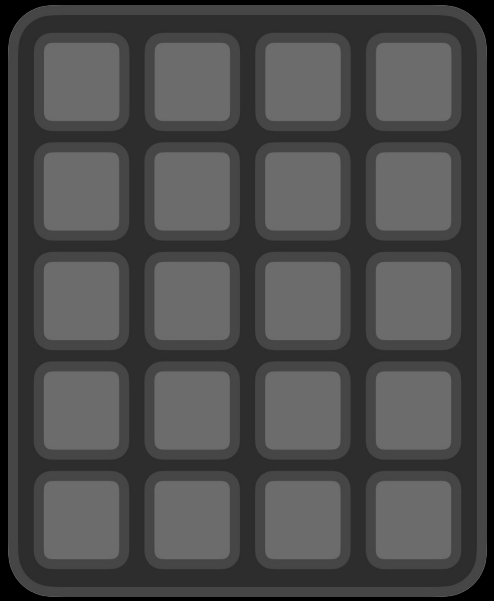


Cache

Branch
Predictor

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

Data PACMAN Attack



Cache

Branch
Predictor

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

1

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

Data PACMAN Attack

Cache

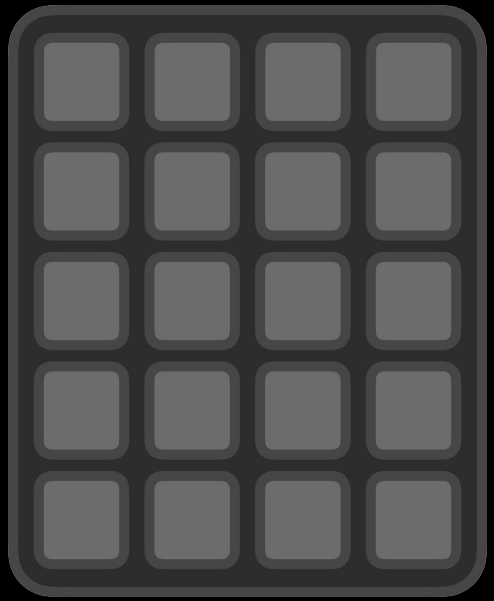
Branch Predictor

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

1

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

Data PACMAN Attack



Cache

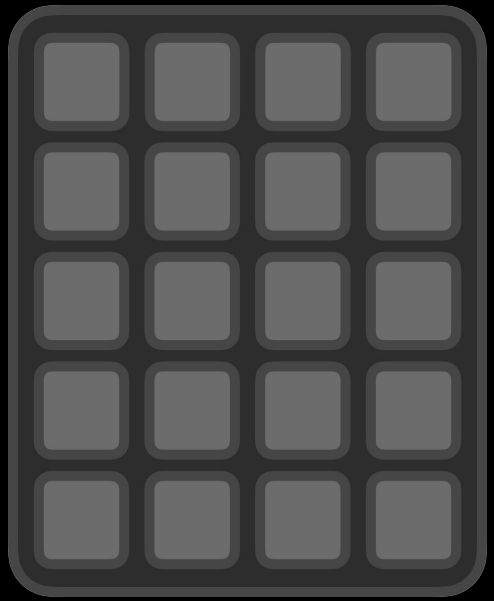
Branch Predictor

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

2

Reset the entire Cache

Data PACMAN Attack



Cache

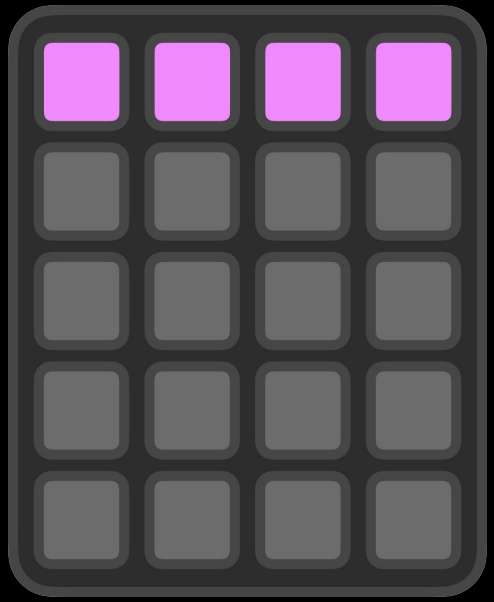
Branch Predictor

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

3

Prime the Cache with an eviction set

Data PACMAN Attack



Cache

Eviction Set

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

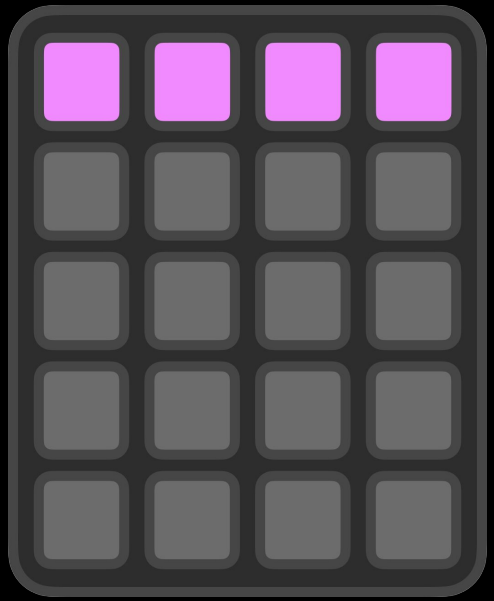
Taken

Branch Predictor

with an eviction set

Prime the Cache

Data PACMAN Attack



Cache

Branch Predictor

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

4

Call the gadget with the pointer and PAC to guess.

Data PACMAN Attack

Cache

Branch Predictor

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

4

Call the gadget with the pointer and PAC to guess.

Data PACMAN Attack

Cache

Branch Predictor

Speculative

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

4

Call the gadget with the pointer and PAC to guess.

Data PACMAN Attack

Cache

Branch Predictor

Speculative

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

4

Call the gadget with the pointer and PAC to guess.

Data PACMAN Attack

Cache

Branch Predictor

Speculative

```
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

Guess Pointer

Cache

Branch Predictor

Speculative

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

4

Call the gadget with the pointer and PAC to guess.

Data PACMAN Attack

Cache

Branch Predictor

Speculative

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

If the guess was incorrect...

4

Call the gadget with the pointer and PAC to guess.

Data PACMAN Attack

No Change

Cache

Branch Predictor

Speculative

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

Speculative Data Abort!

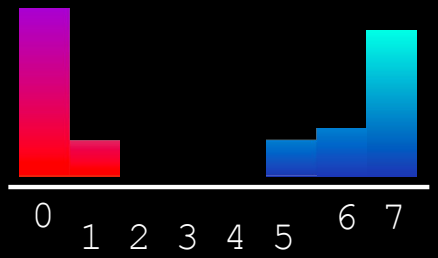
4

Call the gadget with the pointer and PAC to guess.

Ok, let's build it for real.

"Differentiation"

Given a correct and invalid PAC, can we tell which is which?



"Brute Force"

Try all possible PACs and tell which is correct for a given pointer.

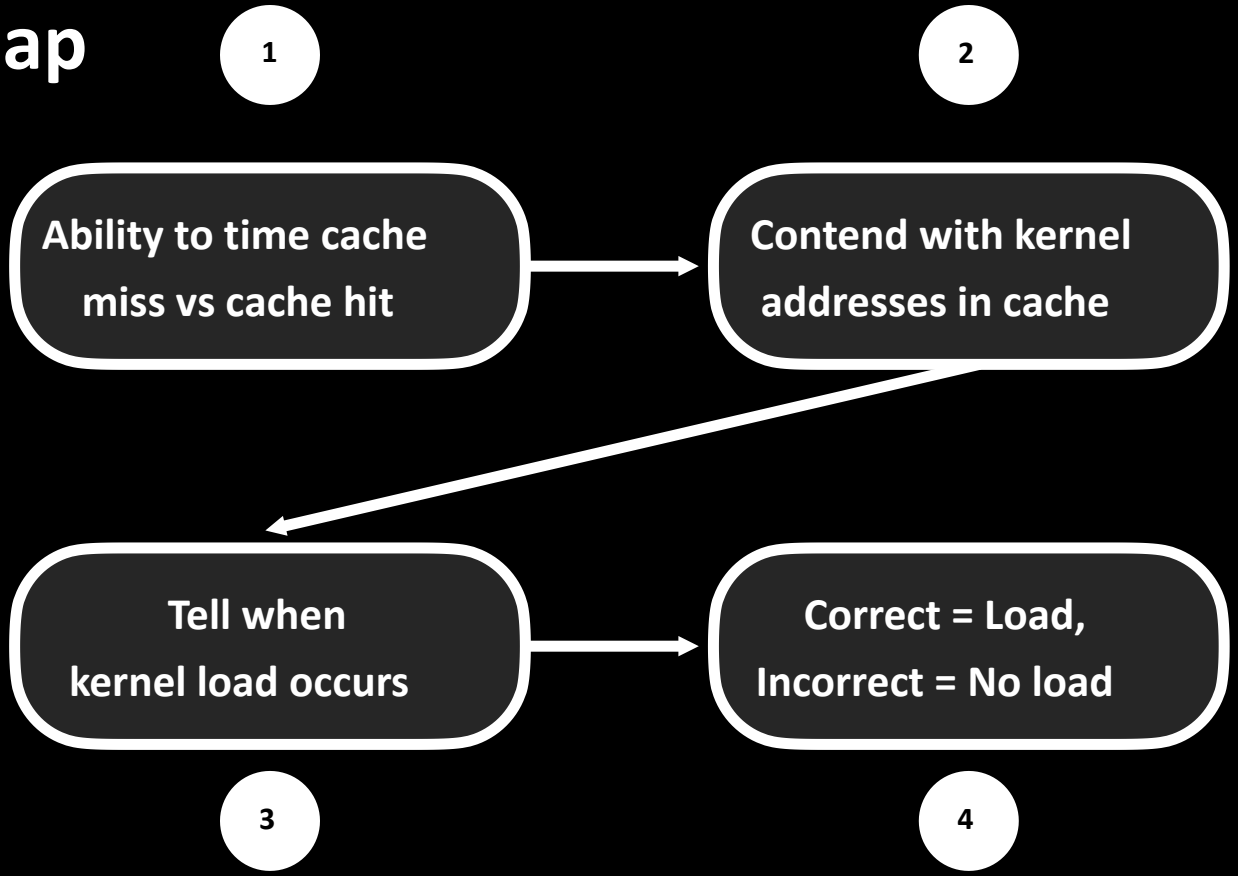
0x0000 ✗

0x0001 ✓

0x0002 ✗

. . .

Roadmap


1
2
Ability to time cache miss vs cache hit
Contend with kernel addresses in cache
Tell when kernel load occurs
Correct = Load, Incorrect = No load
3
4

Reverse Engineering M1

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

Timer Sources on M1

	Speed	Available to users?
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
Apple Custom Cycle Counter	Very Fast (Cycle Accurate)	No
Multi-threaded Counter	"Good Enough"	Yes

Attack with this

Reverse Engineer with this

Multi-Threaded Counter

"Actually works pretty well"

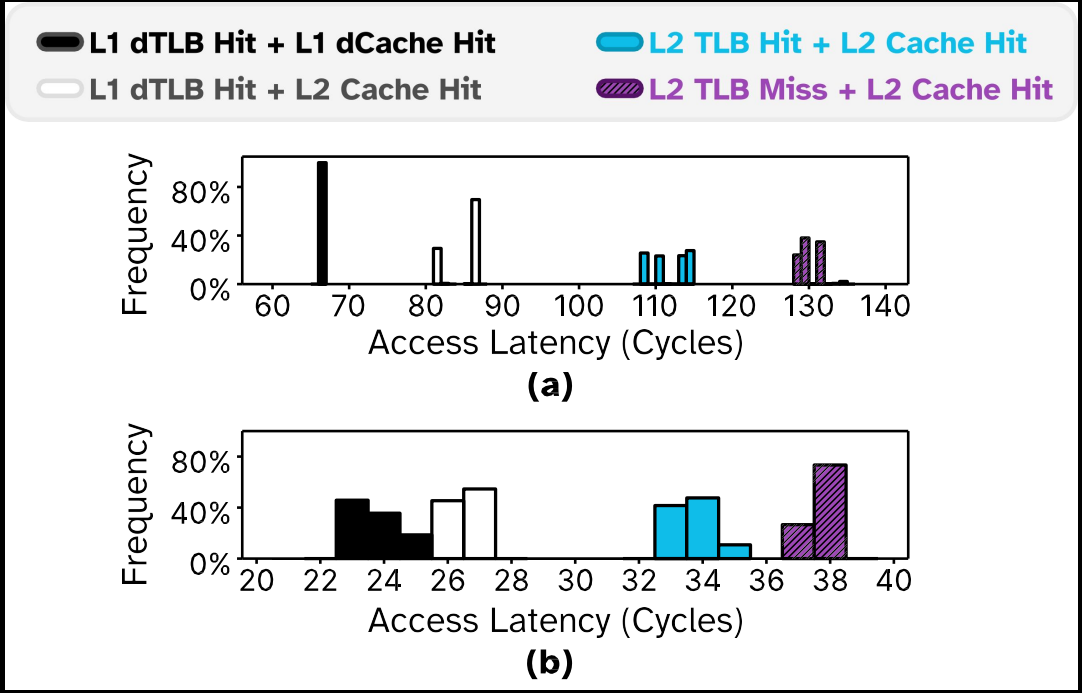
```
while(true):
    counter++;
```

Apple

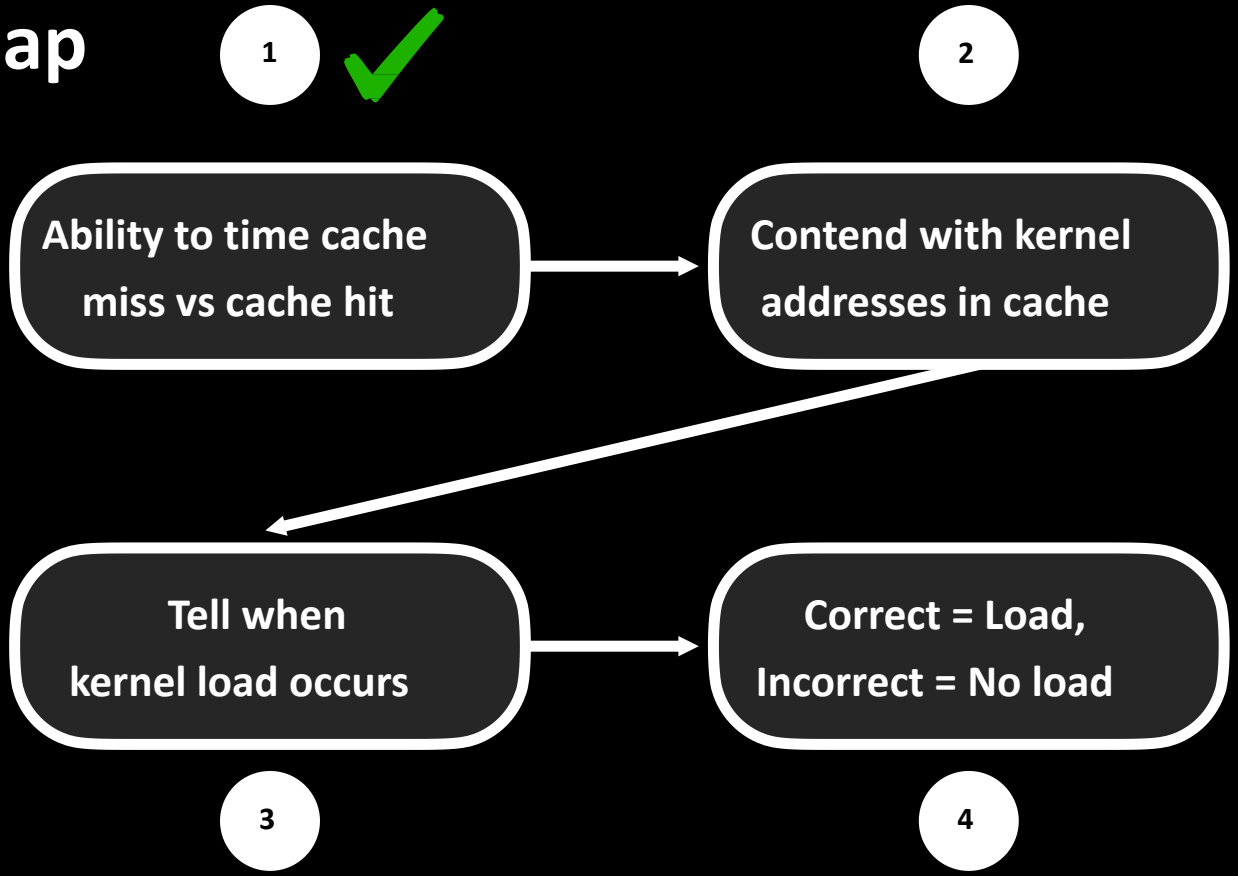
Performance Counters

Multithreaded

Timer



Roadmap


1
2
Ability to time cache miss vs cache hit
Contend with kernel addresses in cache
Tell when kernel load occurs
Correct = Load, Incorrect = No load
3
4

Contending with `evict+reload`

1

Load the address we want to measure

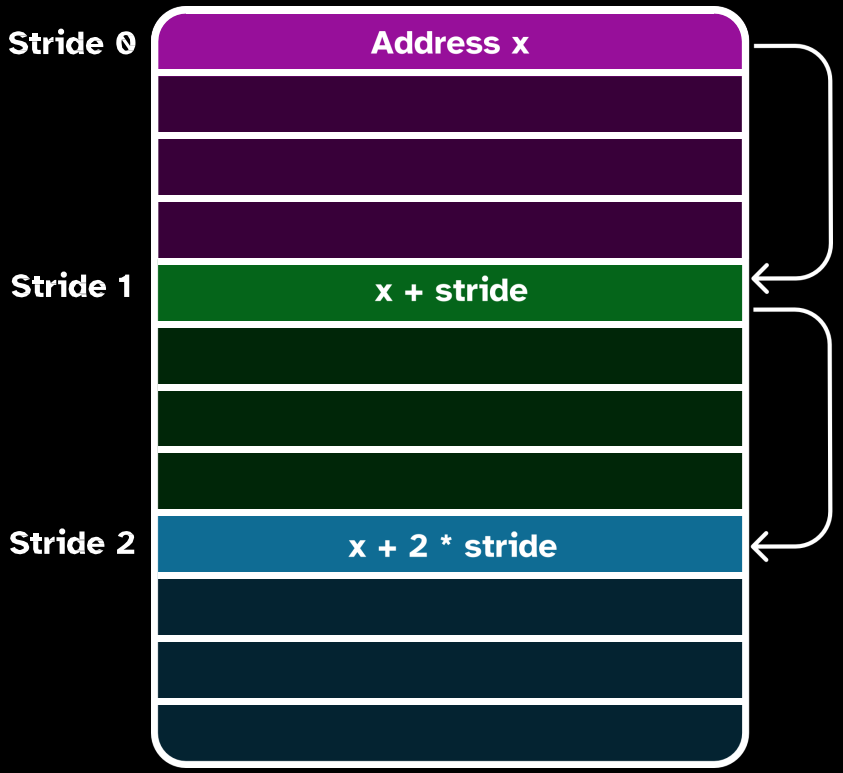
2

Load a bunch of addresses that might evict it

3

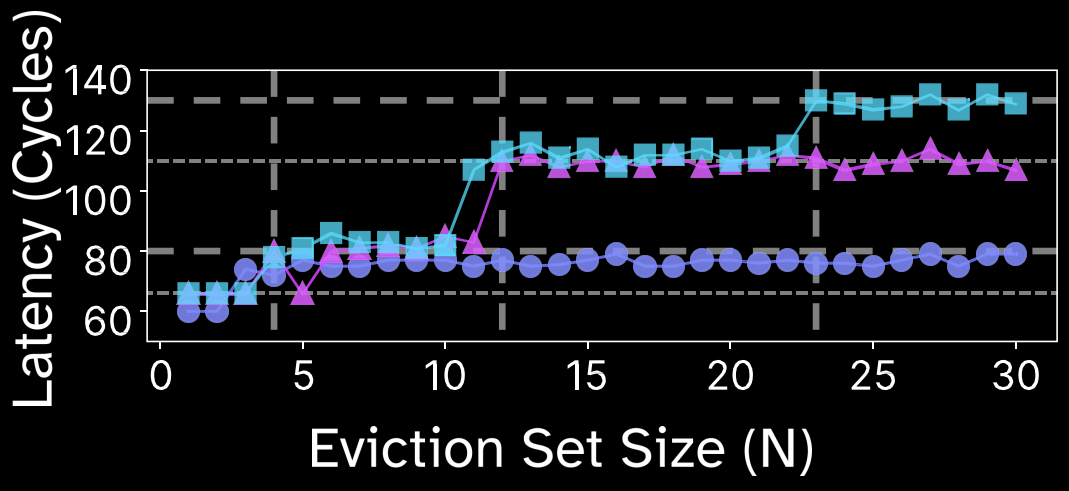
Check if we evicted it by reloading

TLB + Cache Interactions




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

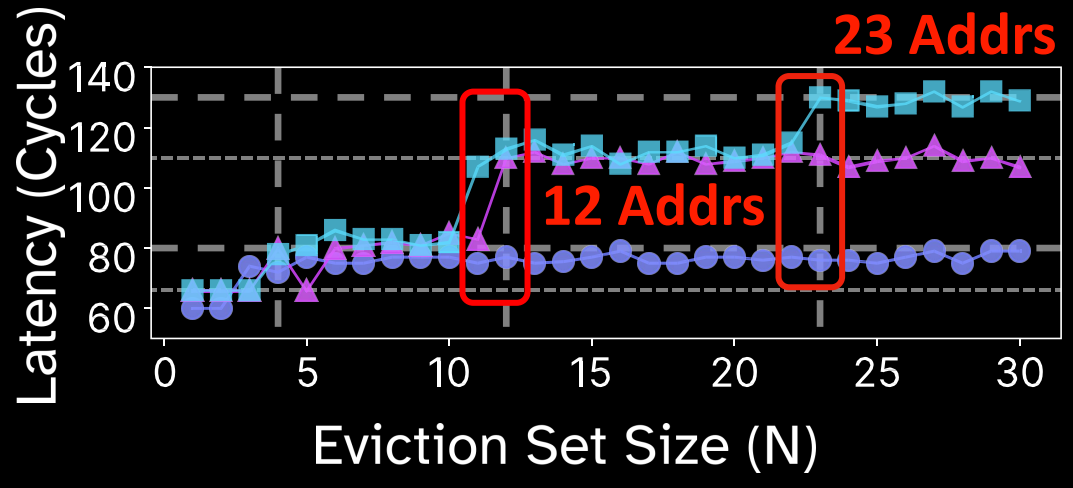
TLB + Cache Results



Latency from dTLB/dCache conflicts

256 x 16KB 2K x 16KB 256 x 128B 32 x 16KB

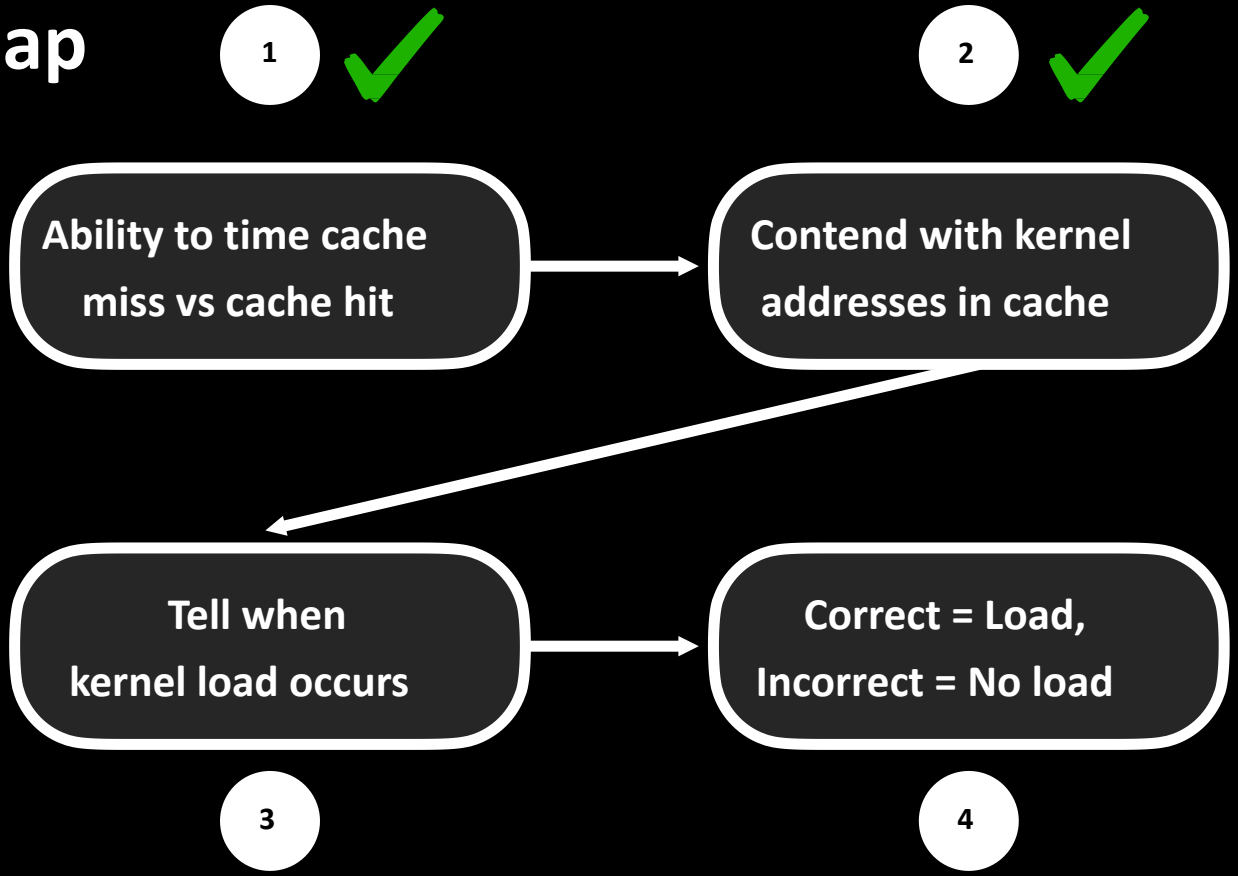
TLB + Cache Results



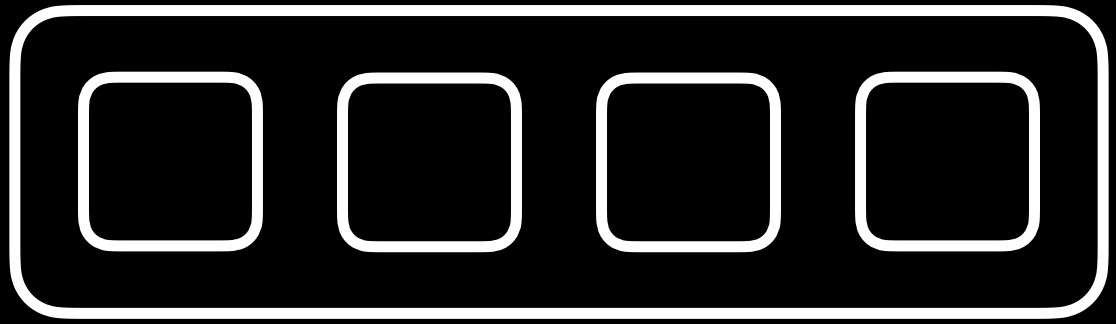
Latency from dTLB/dCache conflicts

256 x 16KB 2K x 16KB 256 x 128B 32 x 16KB

Roadmap

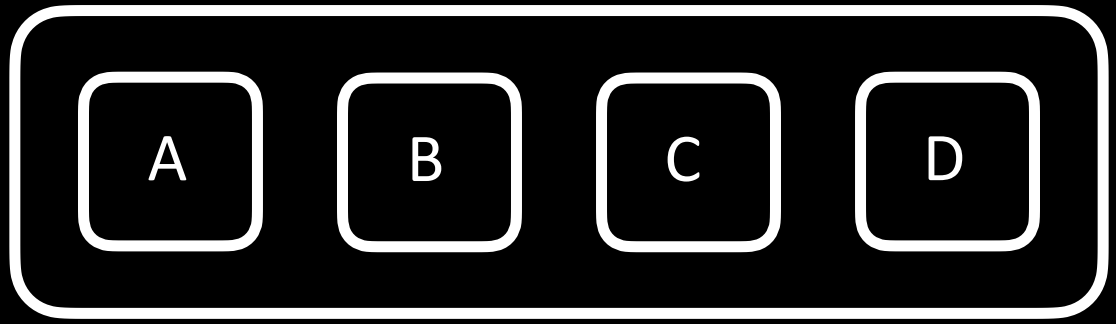

1
2
Ability to time cache miss vs cache hit
Contend with kernel addresses in cache
Tell when kernel load occurs
Correct = Load, Incorrect = No load
3
4

LRU Replacement Policy



A single
cache set

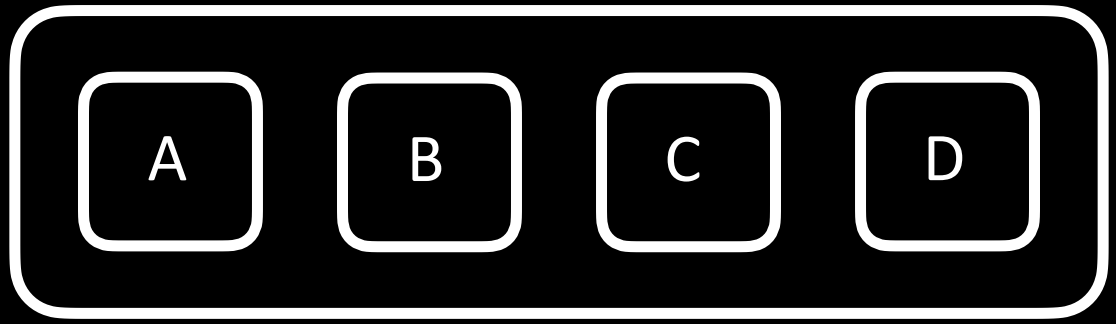
LRU Replacement Policy



A single
cache set

Load A, B, C, D in order

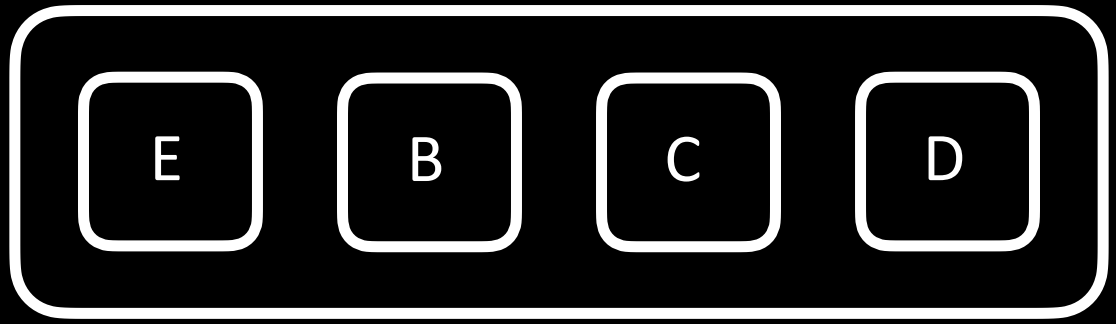
LRU Replacement Policy



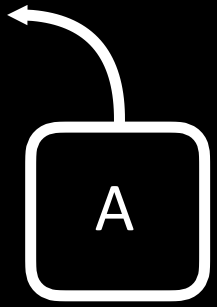
A single cache set

Now we want to load E

LRU Replacement Policy

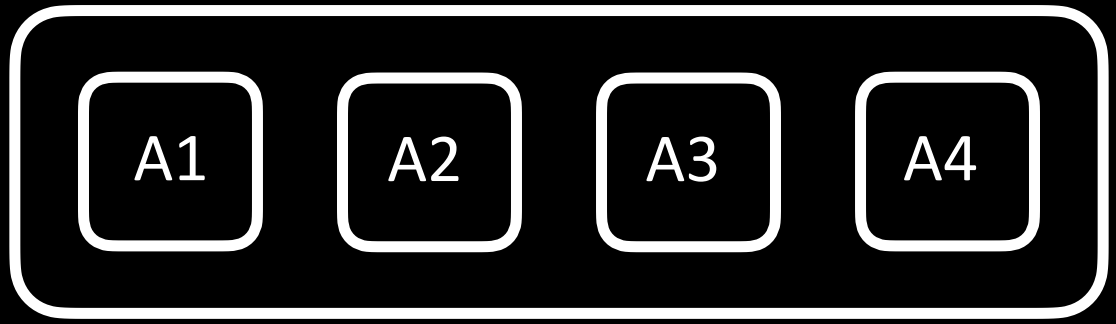


A single cache set



E takes A's spot since A is the least recently loaded

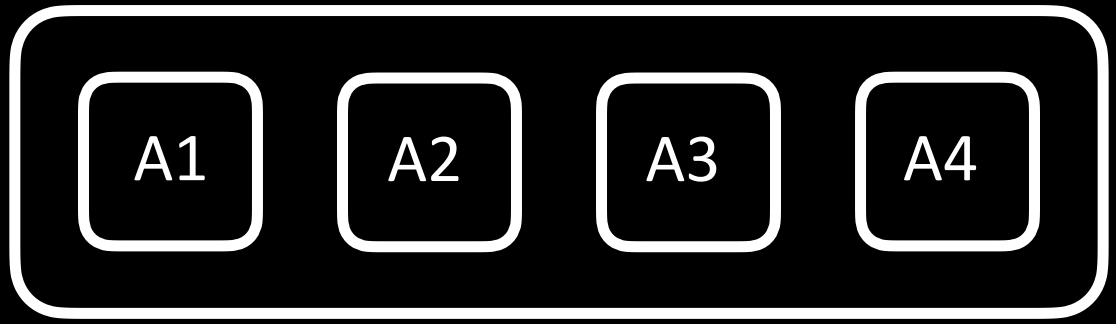
prime+probe



A single
cache set

Load attacker-controlled
A1, A2, A3, A4

prime+probe

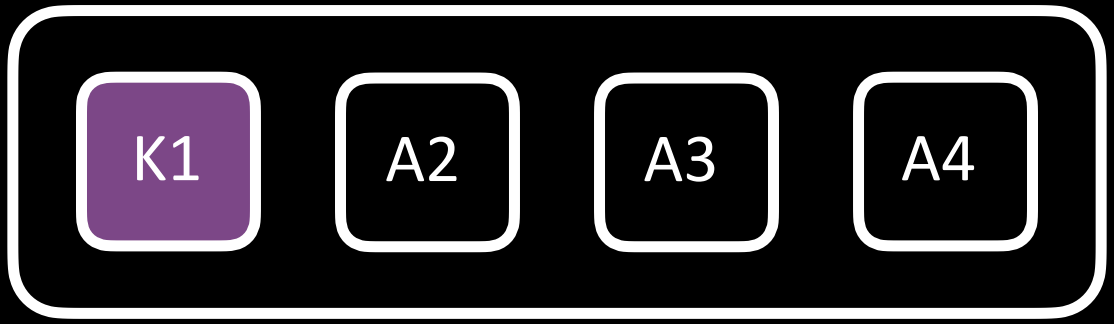


A single cache set

K1

Let the kernel load cache line K1

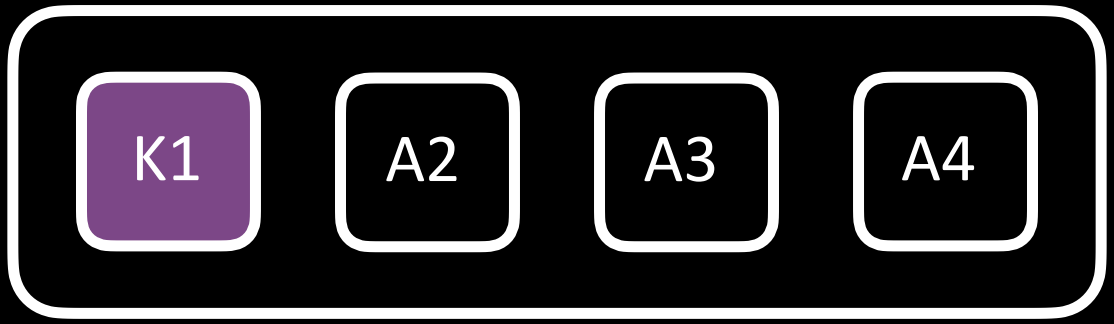
prime+probe



A single
cache set

Let the kernel load
cache line K1

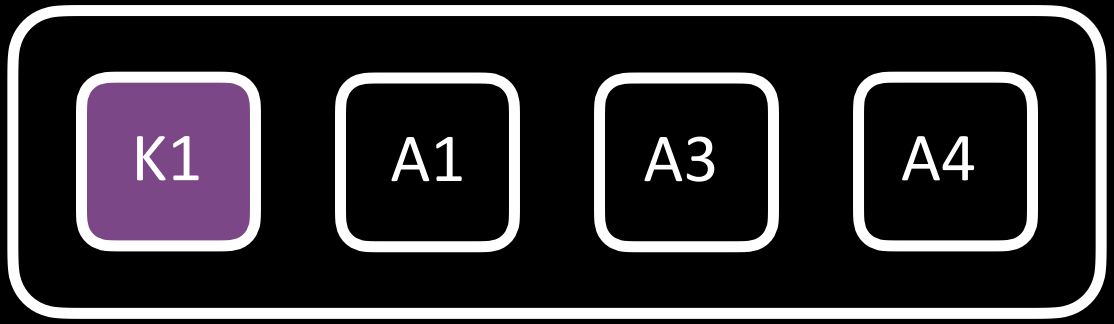
prime+probe



A single
cache set

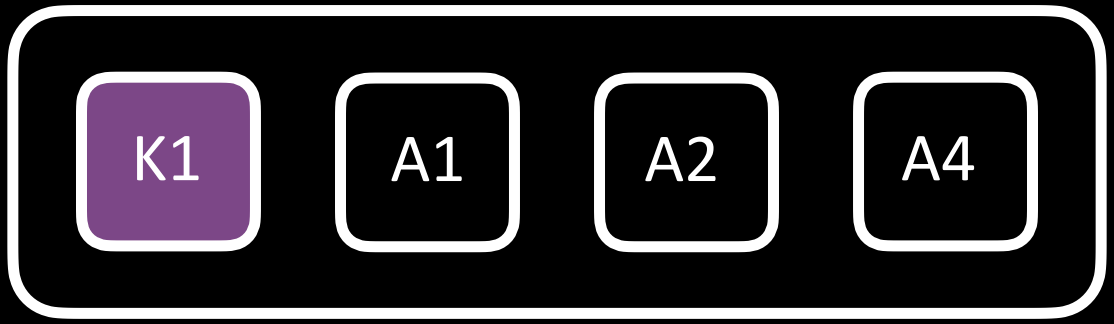
Now we reload
A1, A2, A3, A4
in the same order...

prime+probe



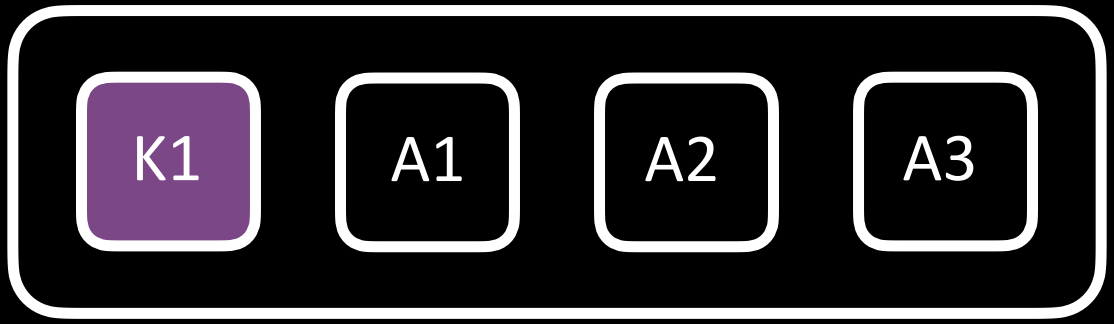
A single
cache set

prime+probe



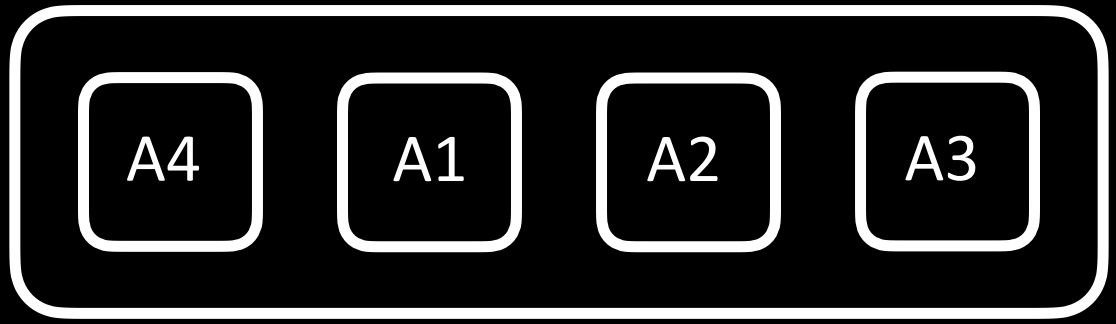
A single
cache set

prime+probe



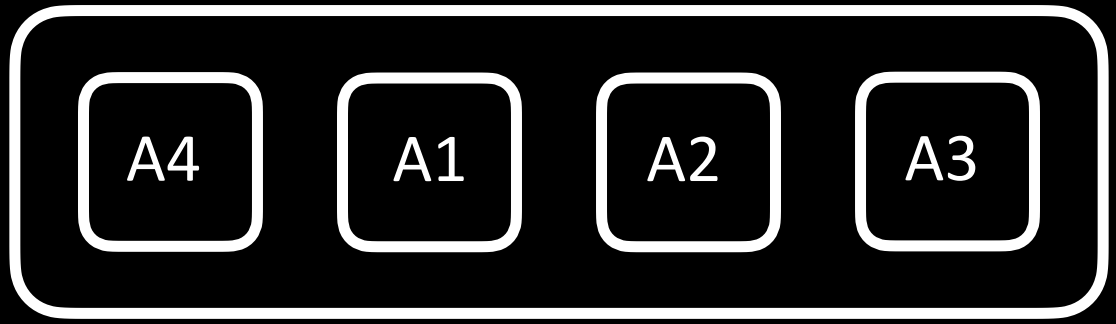
A single
cache set

prime+probe



A single
cache set

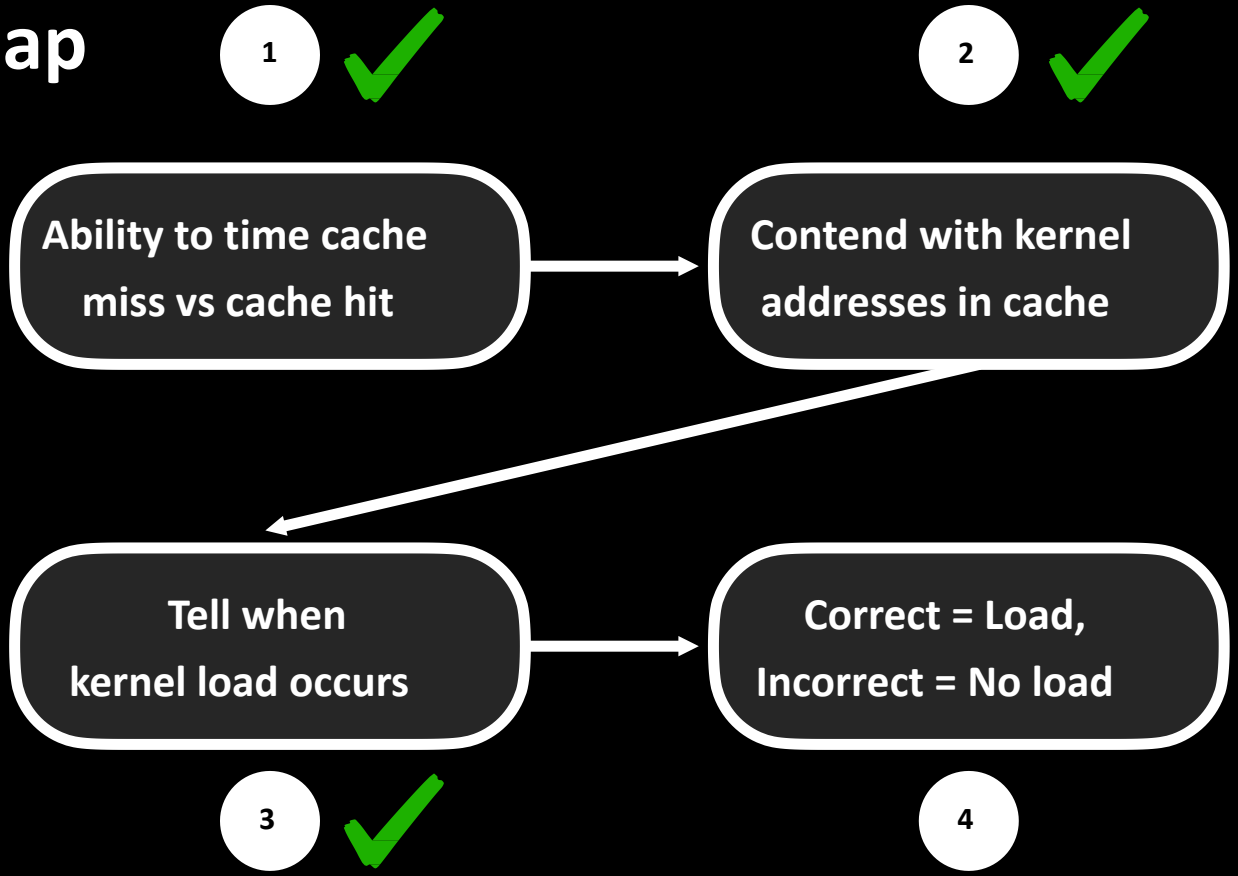
prime+probe



A single
cache set

One evicted address
turned into 4 misses!

Roadmap


1
2
Ability to time cache miss vs cache hit
Contend with kernel addresses in cache
Tell when kernel load occurs
Correct = Load, Incorrect = No load
3
4

Cool!

2 Target Programs

Basic
Very simple AUT -> LDR

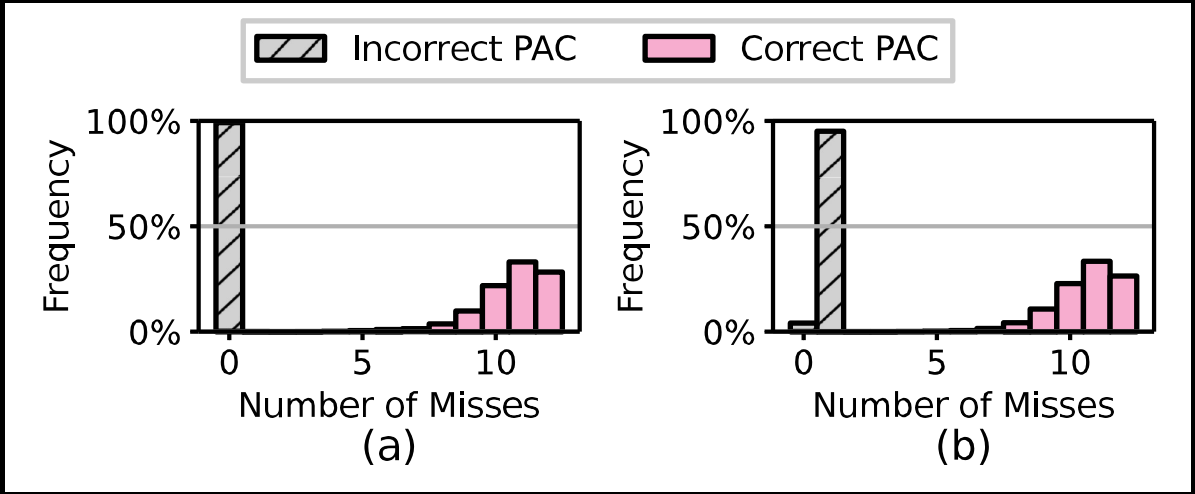
Advanced
C++ vtable function call

Basic Victim

```
if (lots of instructions that take a very long time):
    aut

    ldr
```

Basic Victim VS PACMAN

20,000 Runs



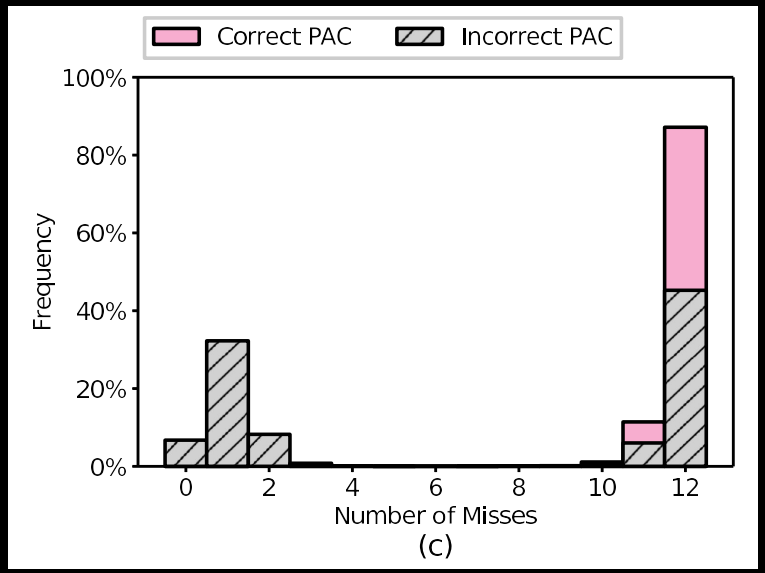
AUT -> LDR

AUT -> BLR

Basic Victim VS PACMAN

20,000 Runs

Almost exactly 50% incorrect PACs perform a load anyways!



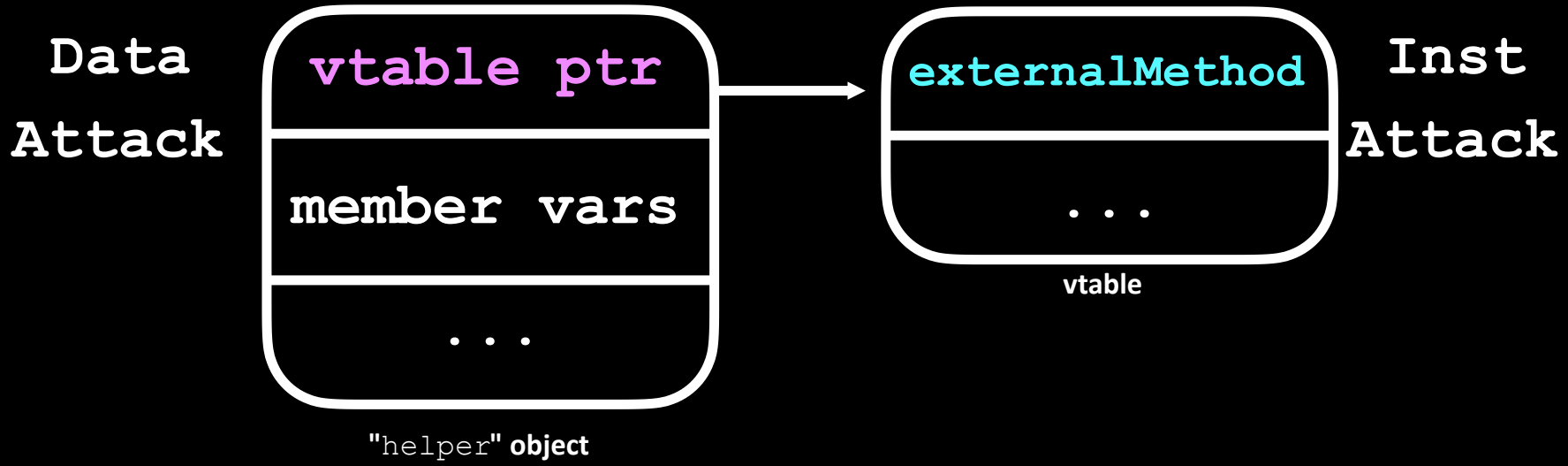
BLRAA

Advanced Victim

```
if (user_argument < limit) {
    return target->helper.externalMethod(); // virtual method call
}
```

Advanced Victim

```
if (user_argument < limit) {
    return target->helper.externalMethod(); // virtual method call
}
```



Advanced Victim

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

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

Advanced 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
```

Verify signed vtable data pointer

Advanced Victim

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

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

Load `externalMethod` **ptr from vtable**

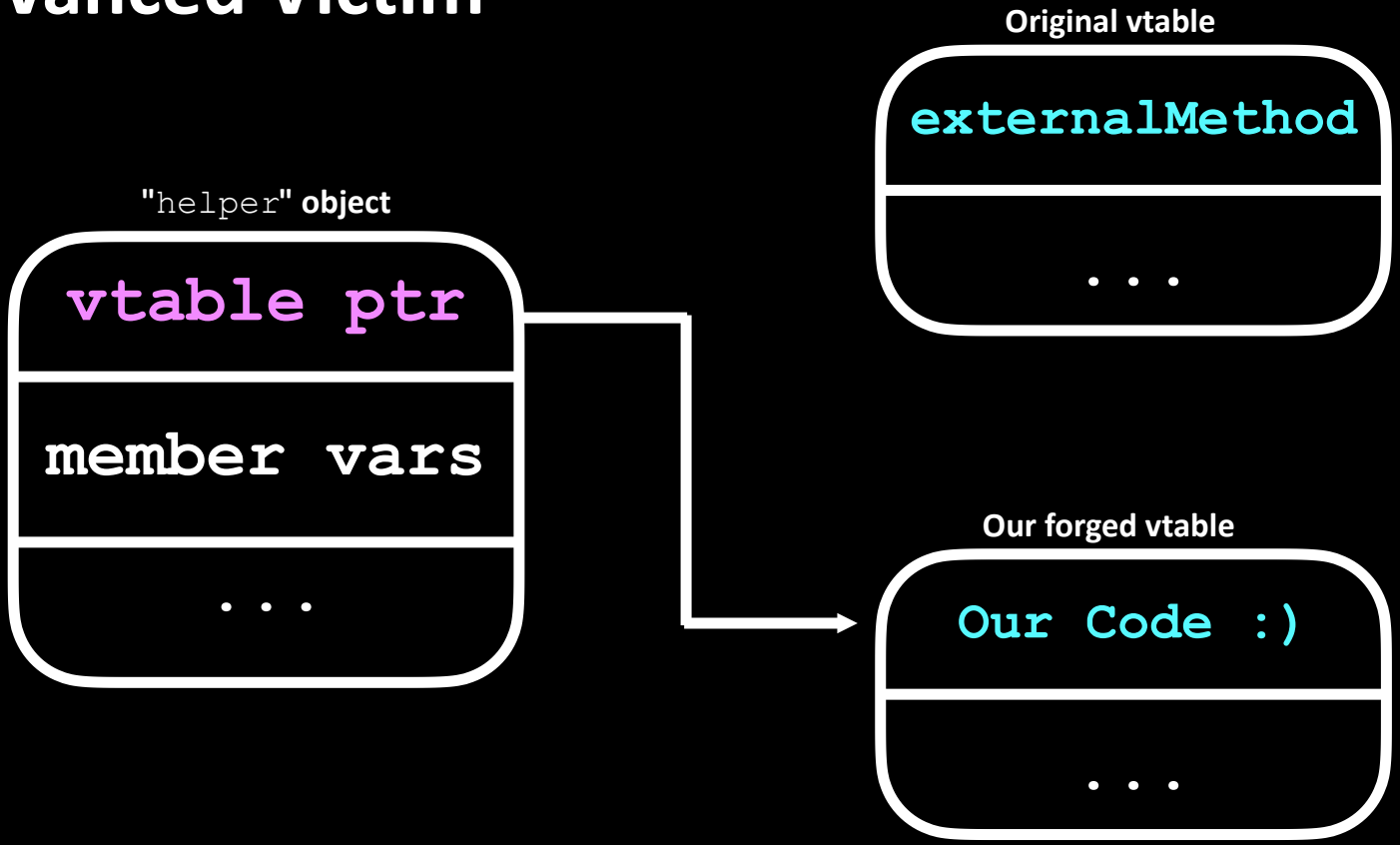
Advanced Victim

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

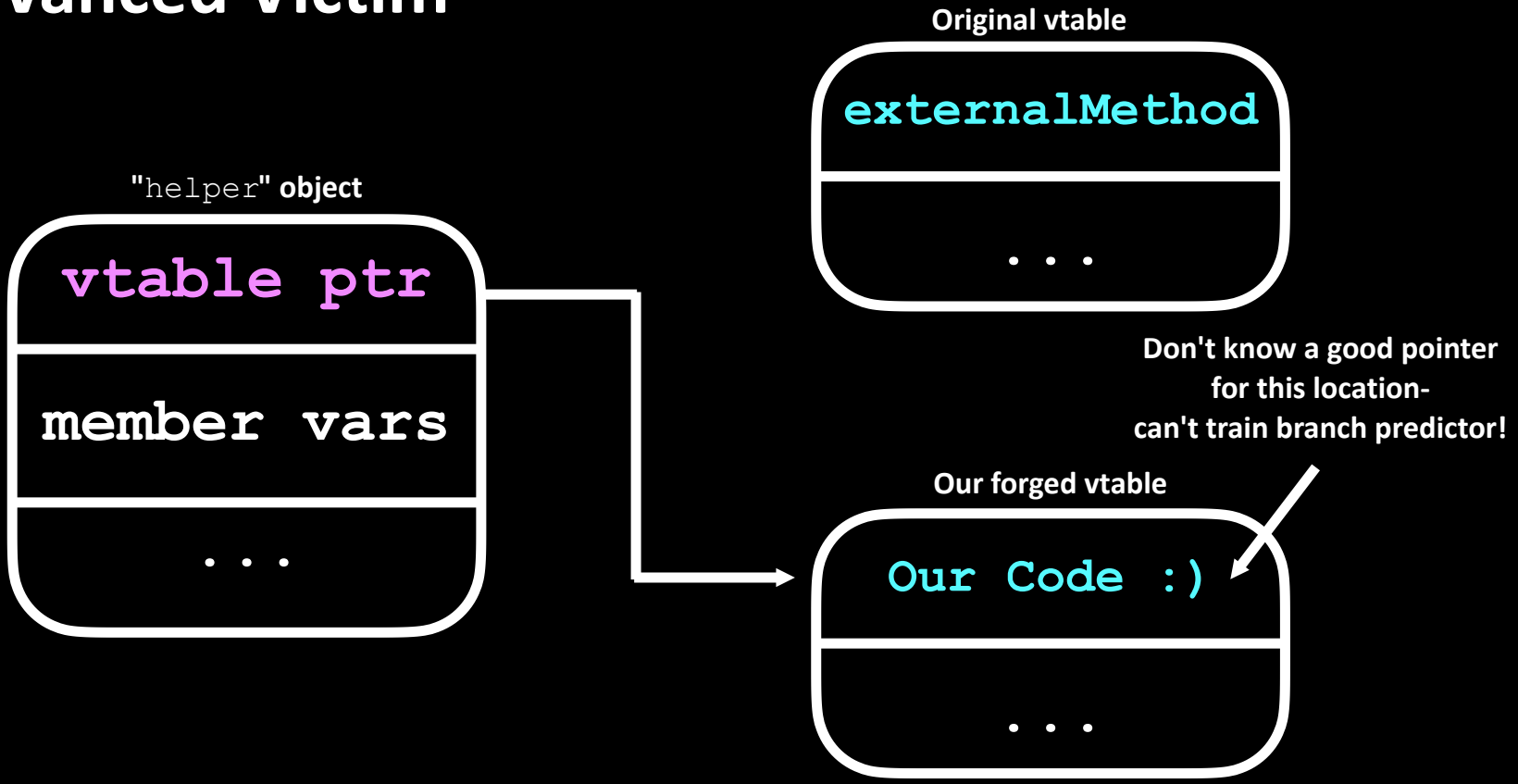
```
ldr x16, [helper]
mov x17, helper
movk x17, #0xd986, lsl #48
autda x16, x17
ldr x8, [x16]
mov x9, x16
mov x17, x9
movk x17, #0xa7d5, lsl #48
autia x8, x17
blr x8
```

Verify externalMethod inst pointer

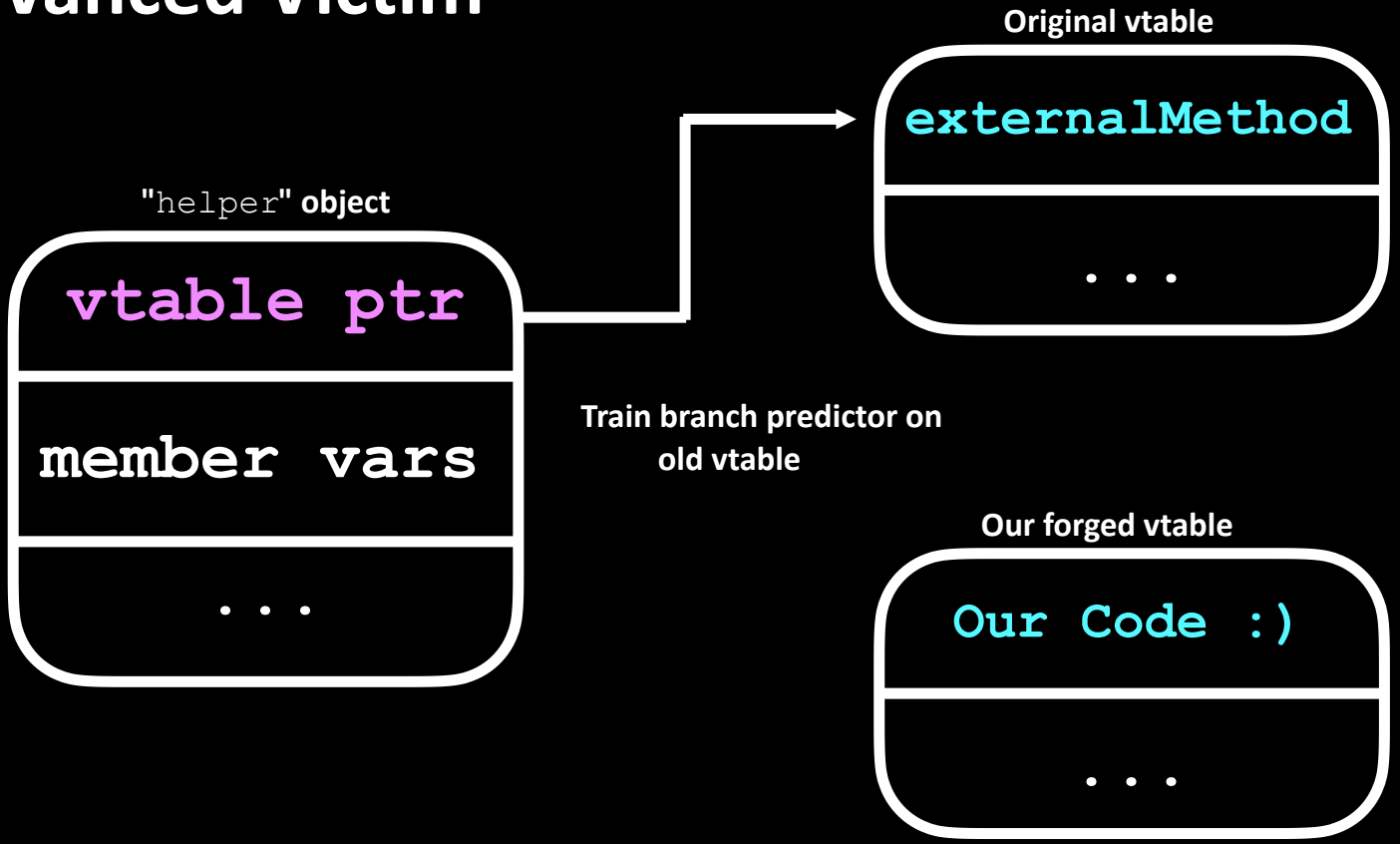
Advanced Victim



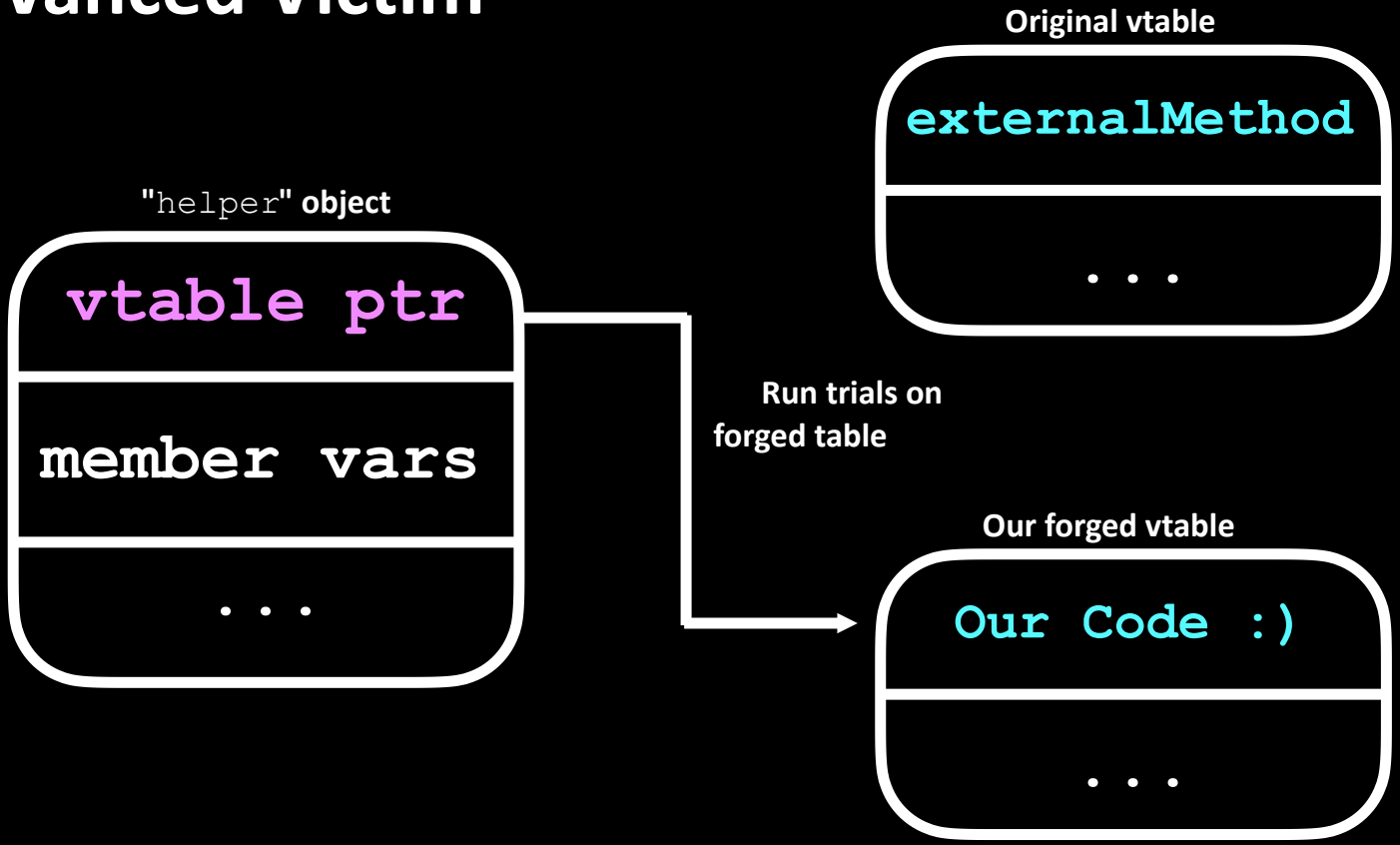
Advanced Victim



Advanced Victim

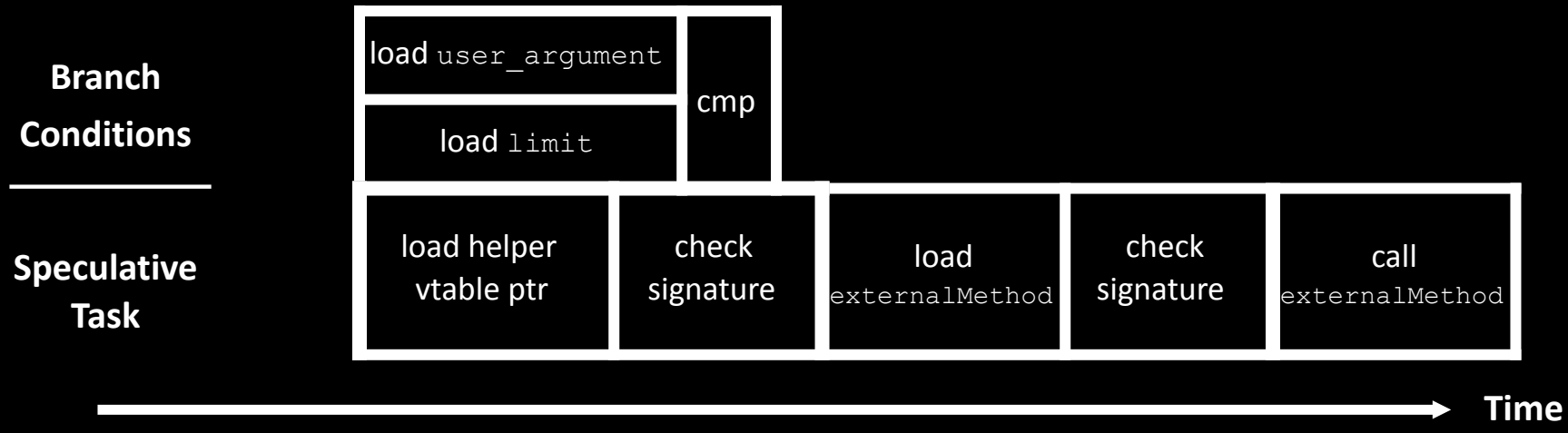


Advanced Victim



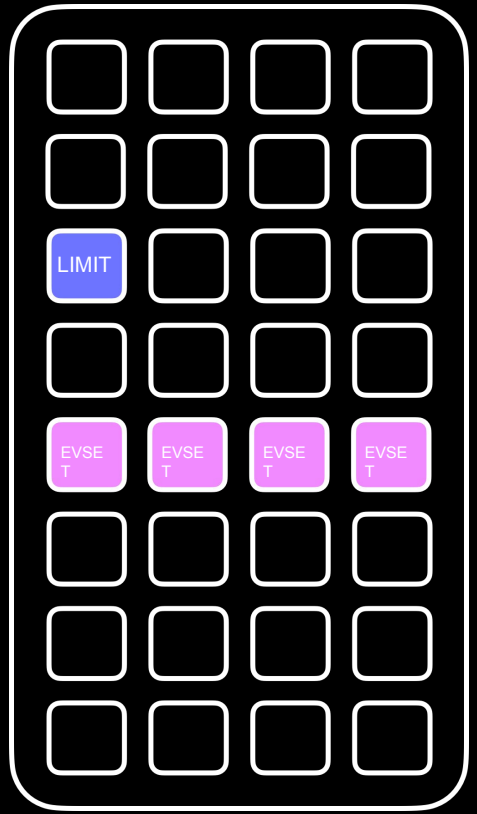
Lengthening the Window

```
if (user_argument < limit) {
    return target->helper.externalMethod(); // virtual method call
}
```

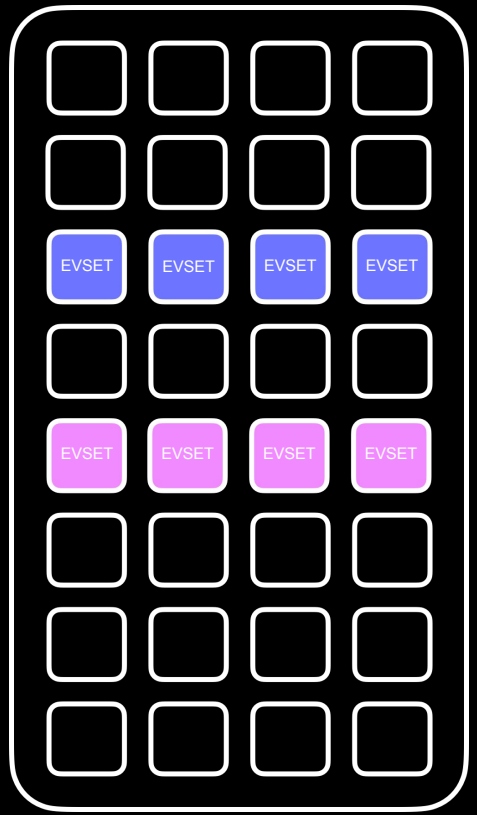


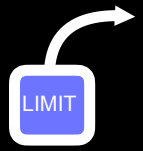
Lengthening the Window

Problem:
`limit` **variable loads too quickly!**



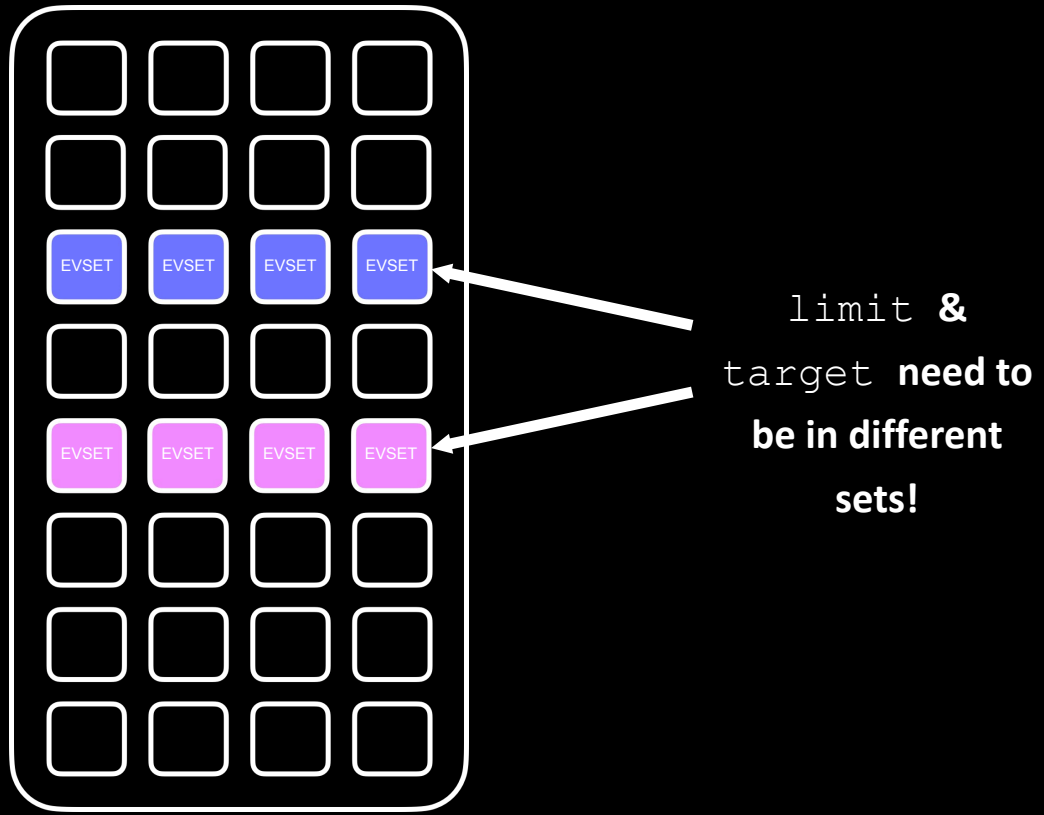
Lengthening the Window





Kick limit out with a second eviction set!

Lengthening the Window



Lengthening the Window

```
if (user_argument < limit) {
    return target->helper.externalMethod(); // virtual method call
}
```