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.

MIT CSAIL

compute. collaborate. create.

$whoami

Joseph Ravichandran
1st Year PhD Student, MIT
@0xjprx

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

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

Jay Lang
MIT CSAIL
Cambridge, MA, USA
jaytlang@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

Q1. What is the PACMAN paper about?

Researchers at the Massachusetts Institute of Technology have published a paper describing the PACMAN technique. This method uses malicious software to brute-force pointer authentication codes (PAC). This technique uses speculation to create an "oracle" for determining those codes. That would allow the attacker to use those codes to reduce the protection that Pointer Authentication provides against Return Oriented Programming (ROP) and Jump Oriented Programming (JOP) attacks. ROP and JOP chain together "gadgets" made up of pieces of existing routines; these gadgets can then be used to form a larger sequence of functions that can be used for further attacks.

The published paper is available at https://pacmanattack.com/.

Q2. Are Arm CPUs vulnerable?

Yes. Some Cortex-A, Neoverse CPUs, and other Arm CPU implementations that support the Pointer Authentication feature (FEAT_PAuth or FEAT_PAuth2) are vulnerable to PACMAN attacks.

The Pointer Authentication feature was implemented in Arm Architecture version Armv8.3-A. In later versions of the Arm Architecture, the Pointer Authentication feature has been enhanced with a faulting feature (FEAT_FPAC) which allows faulting on failed pointer authentication attempts. This faulting feature is necessary for the mitigation explained in *Q10* at the end of this article.

Note

For Cortex-A and Neoverse CPUs, the `API` bits in the `ID_AA64ISAR1_EL1` register indicate whether pointer authentication or the faulting feature (FEAT_FPAC) is available on that CPU.

The following table provides information about whether released CPUs that support pointer authentication are affected:

Neoverse V1

Cortex-A78C

Cortex-A78AE

Cortex-X3

Not just M1...

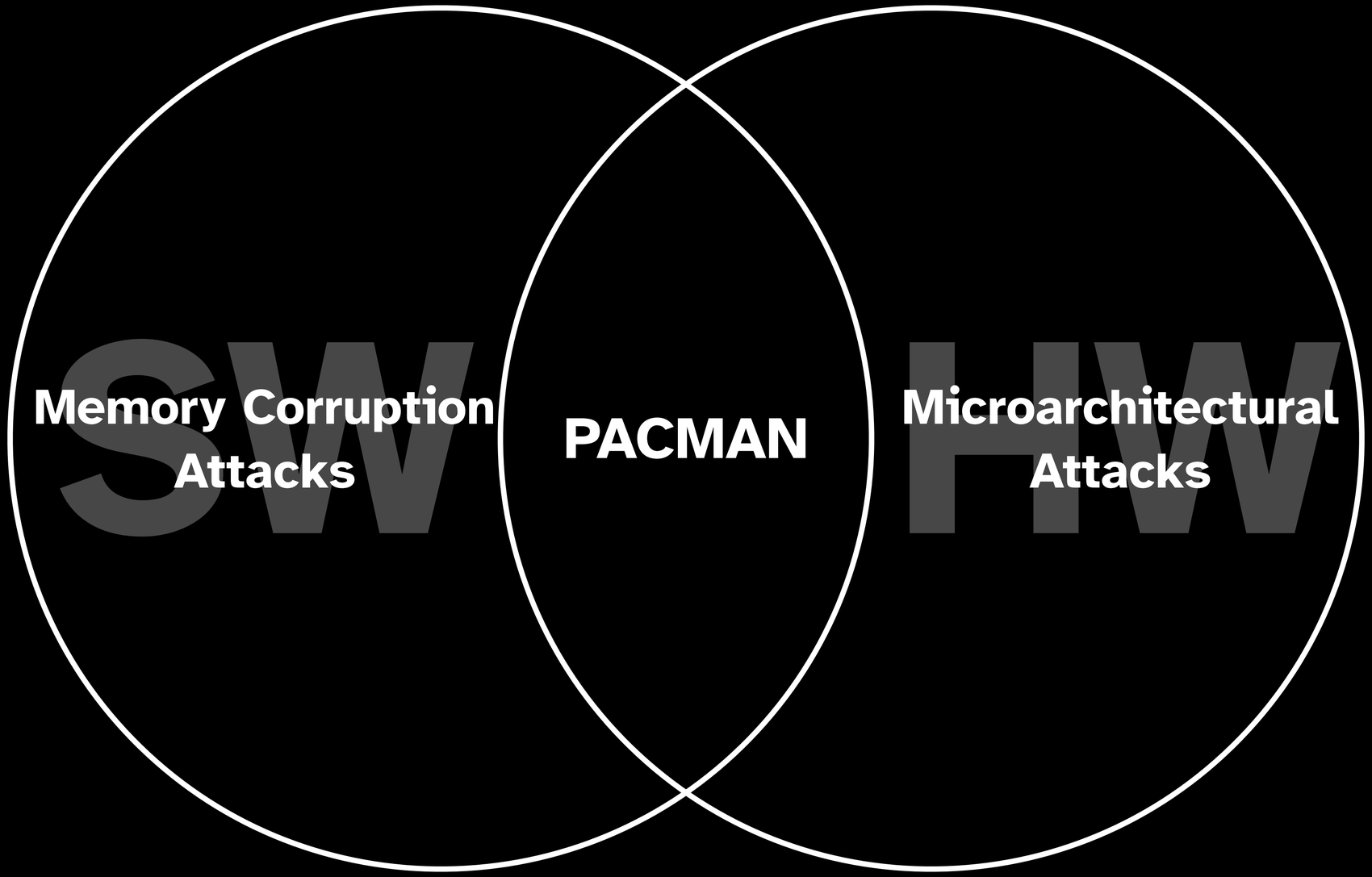
Neoverse N2

Cortex-A710

Cortex-A715

Cortex-X1C

Cortex-X2


SW
Memory Corruption Attacks
PACMAN
HW
Microarchitectural Attacks

Contributions

1

New way of thinking about compounding threat models.

2

Hardware bypass for ARM Pointer Authentication.

3

Attack on Apple M1.

Think like
an attacker.

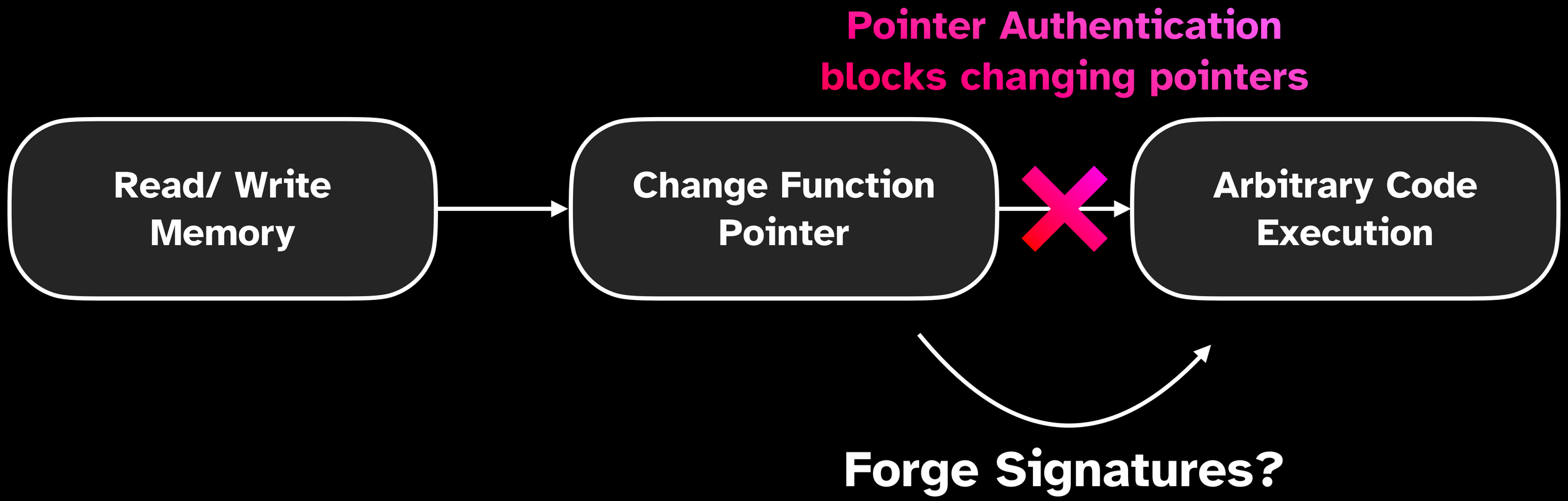
Think like
a CPU designer.

Is this a flaw?

The idea in 60 seconds.

Memory Corruption

Memory Corruption



Just bruteforce it, right?

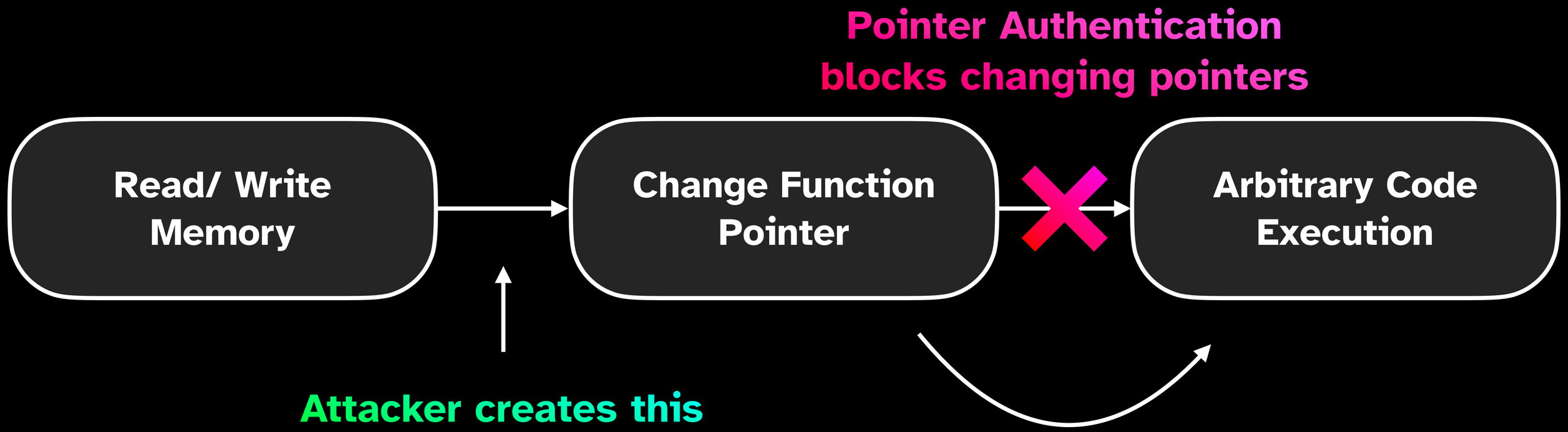
Key Insight:
Avoid crashes using
speculative execution!

BYOB

"*Bring your own bug*"

BYOB

"Bring your own bug"



PACMAN Finder

4 New Tools + PoC

PACMAN Patcher

IOKit driver for performing
microarchitectural experiments.

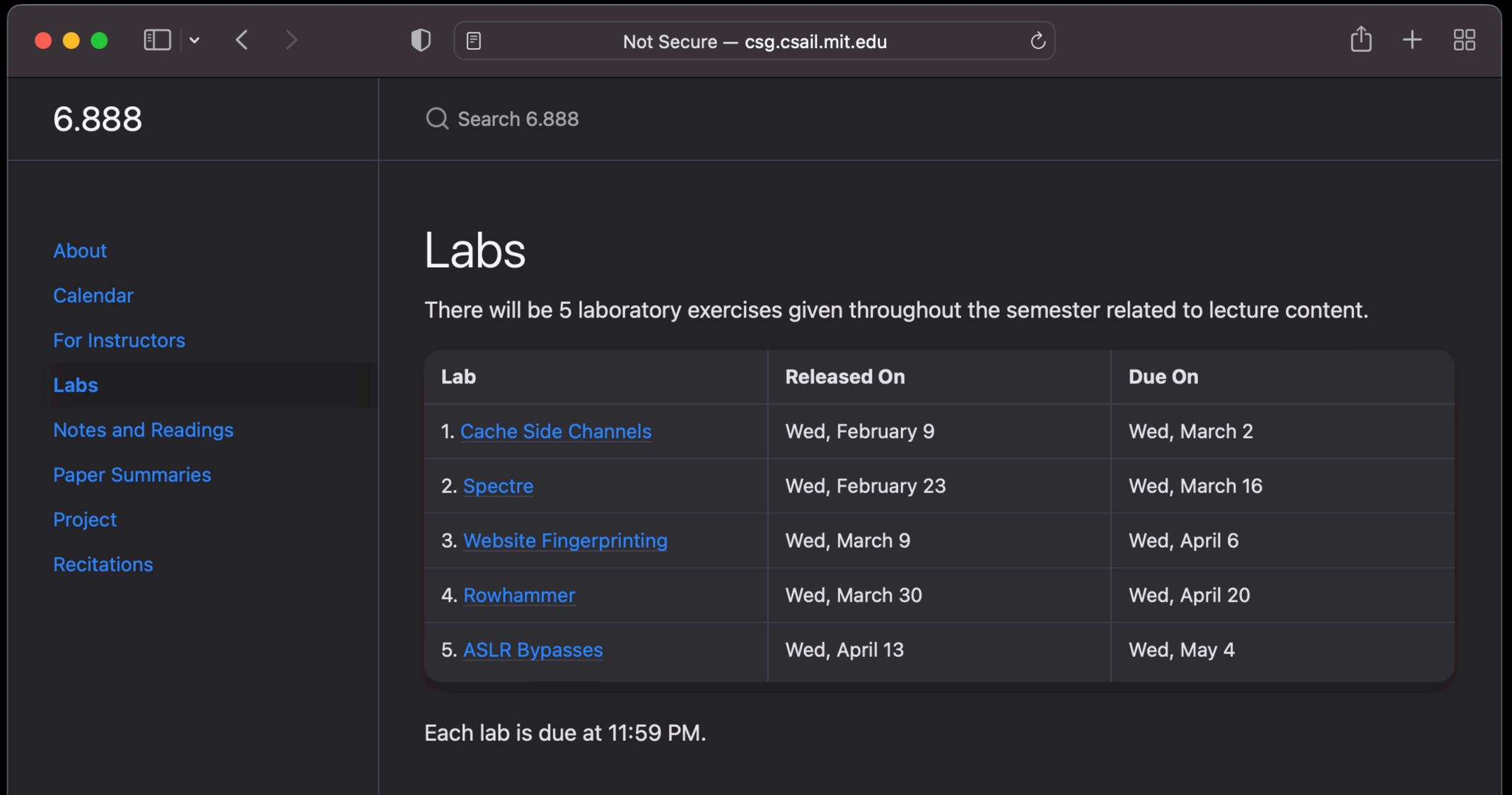
Static analysis scripts for Ghidra
to locate PACMAN Gadgets.

PACMAN

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

Run your own Rust experiments
bare metal on M1.

"MIT Secure Hardware Design"



PACMAN I

Our ISCA 2022 Paper

PACMAN II

Announcing today at DEF CON 30

PACMAN I

Show that it works

PACMAN II

Show it in action

PACMAN I

Simple victim

PACMAN II

Realistic victim

PACMAN I

3 minutes/ ptr

PACMAN II

11 seconds/ ptr

PACMAN I

"Bare Bones" C Attack

PACMAN II

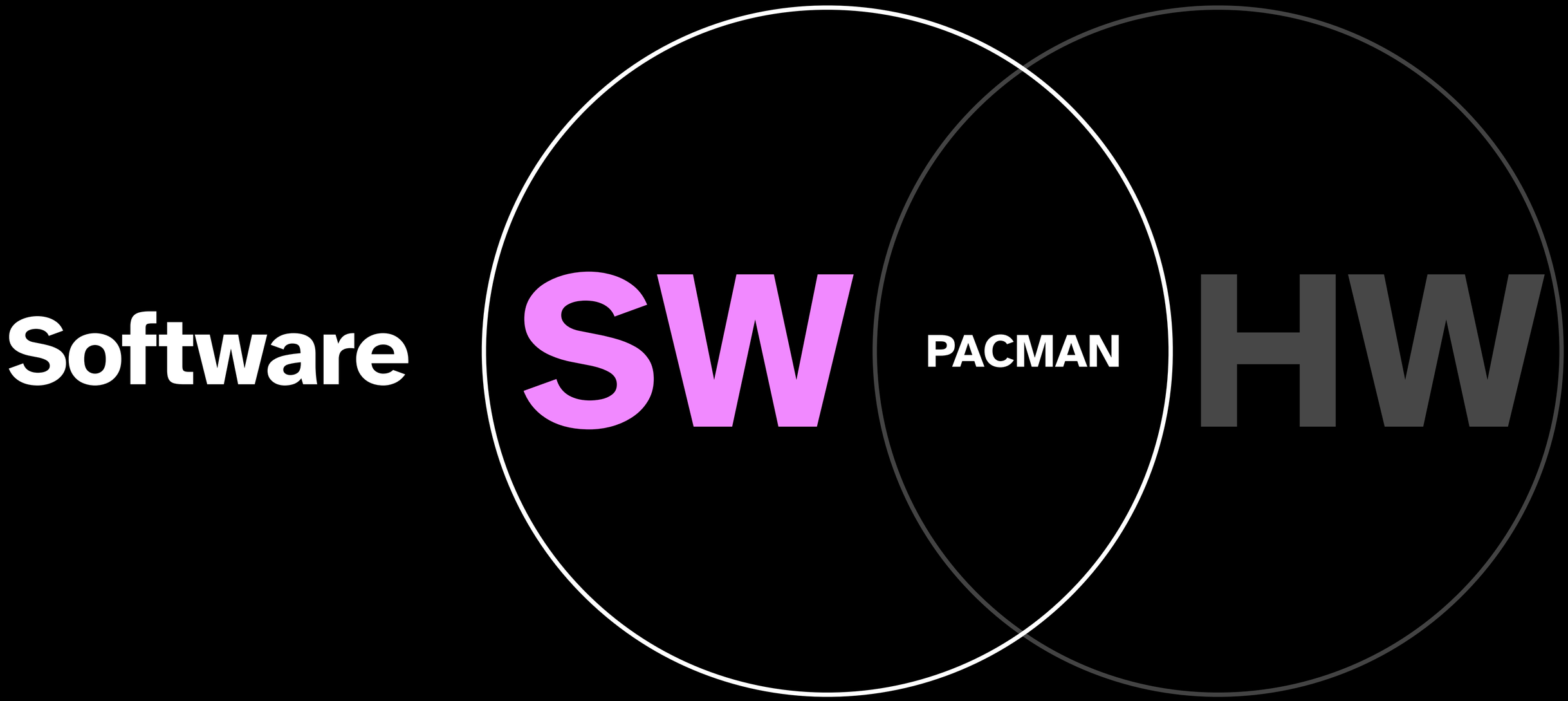
Extensible Rust Framework

PACMAN I

Assumes kernel iTLB eviction via syscalls

PACMAN II

Time instruction latencies instead


Software
SW
PACMAN
HW

Pointer

64-Bit Address

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

Pointer

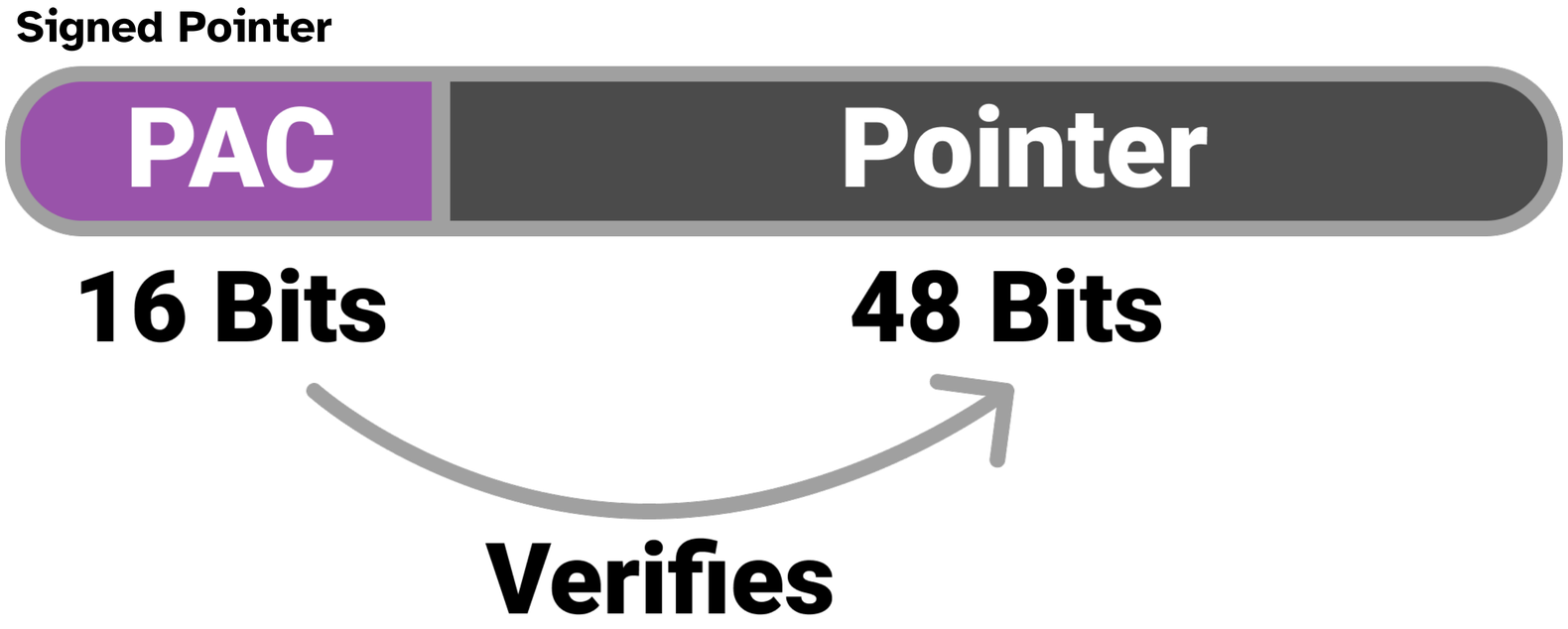
Pointer

Pointer

Let's put this to good use!

Pointer Authentication

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



Pointer Authentication

`PAC*`

Insert a PAC

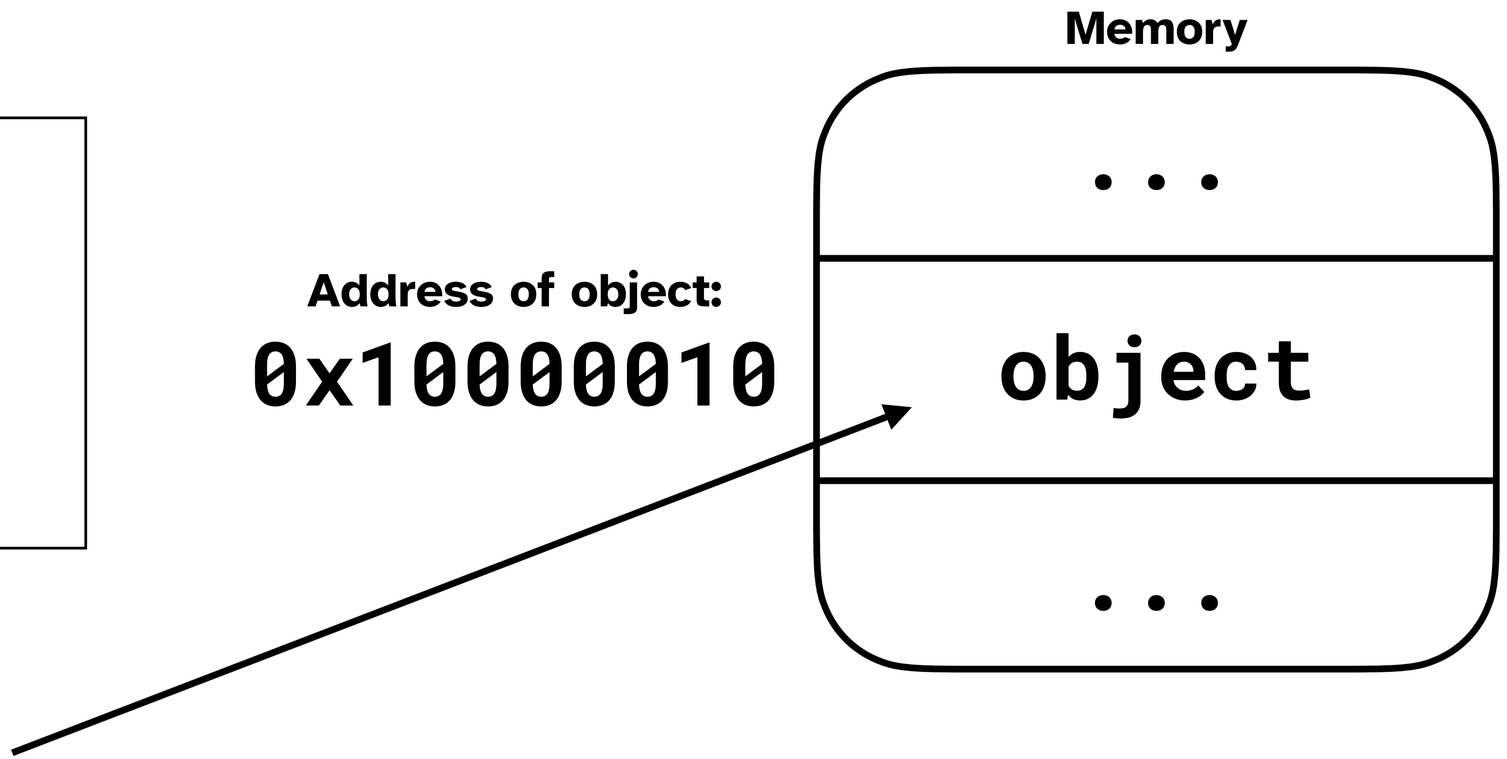
`AUT*`

Check a PAC

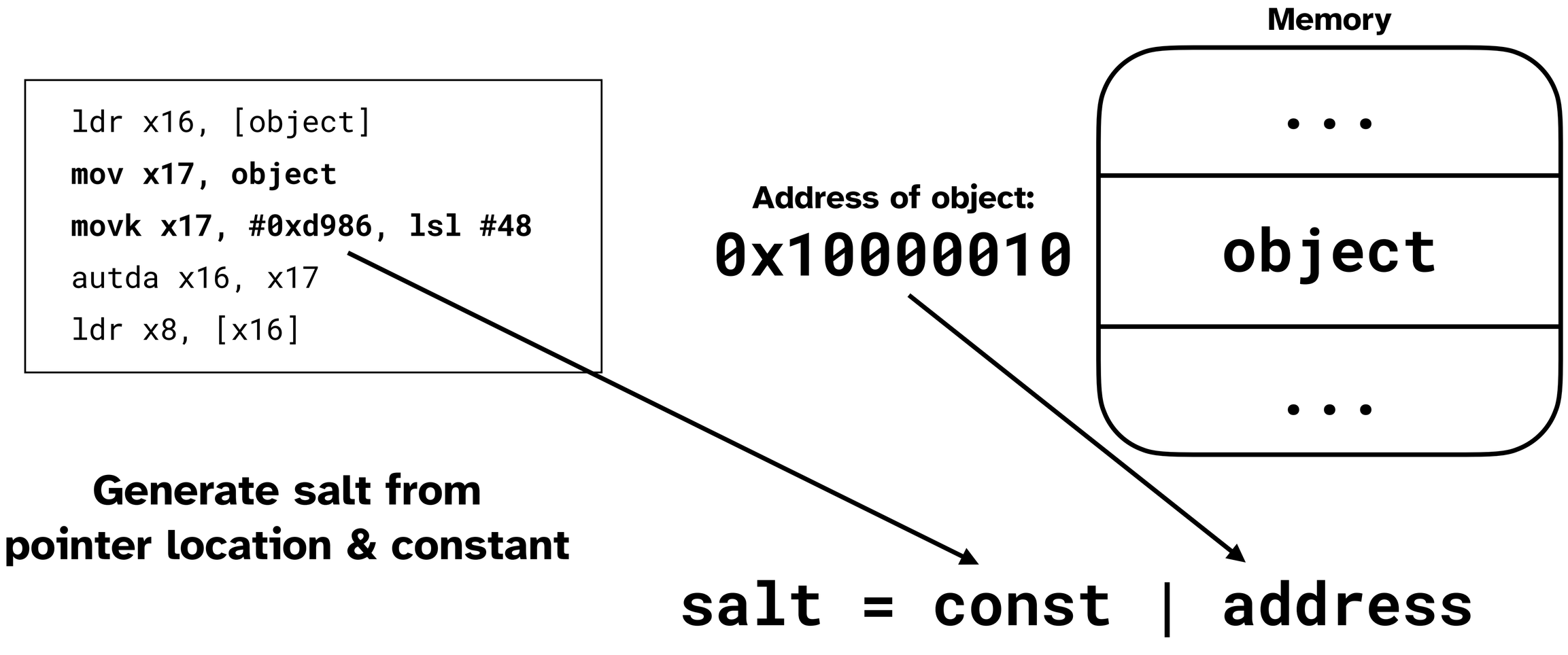
Don't crash on AUT- crash on use

Pointer Authentication

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



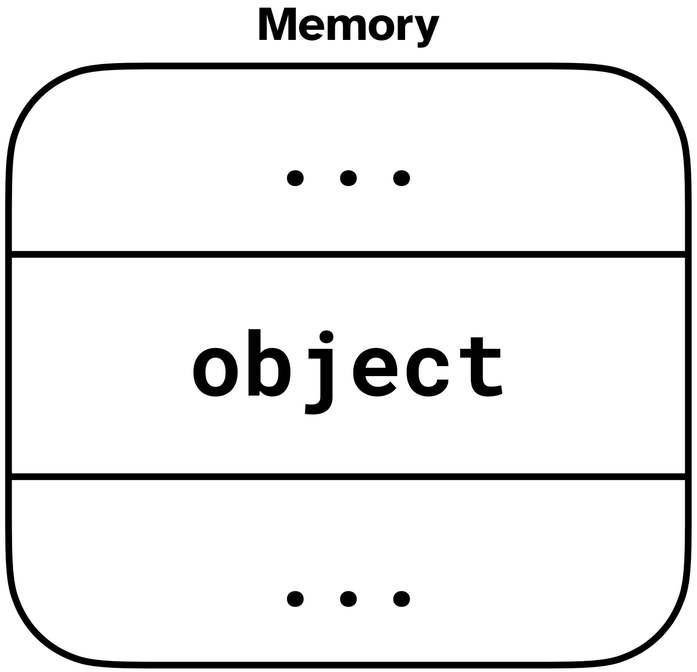
Pointer Authentication



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

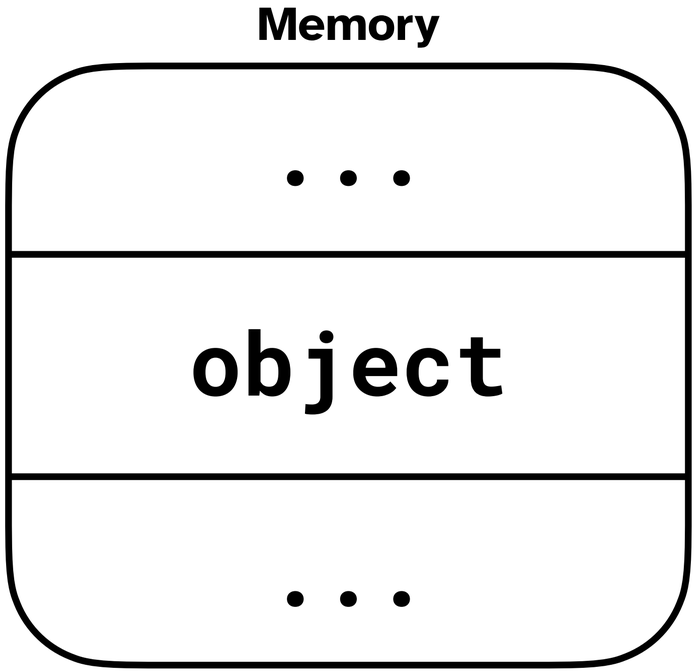
```
ldr x16, [object]
mov x17, object
movk x17, #0xd986, lsl #48
autda x16, x17
ldr x8, [x16]
```

Attempt to load

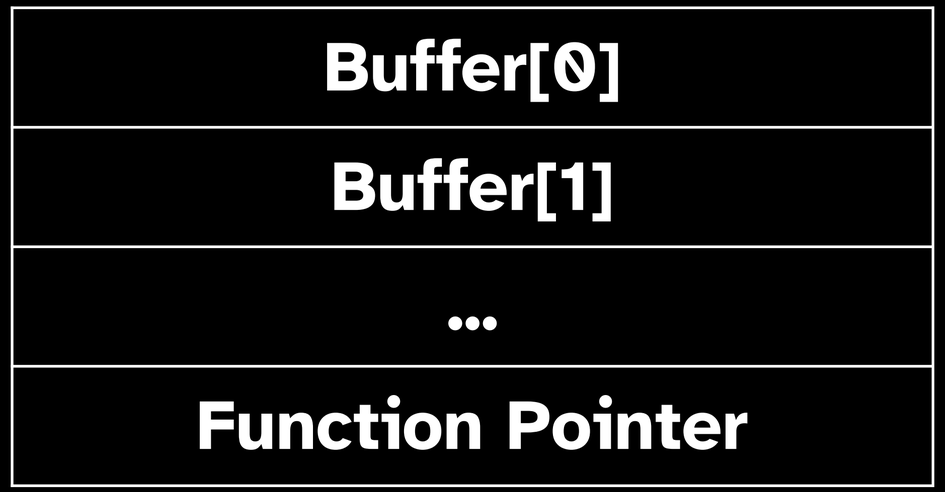
This is where the crash will happen!

Address of object:

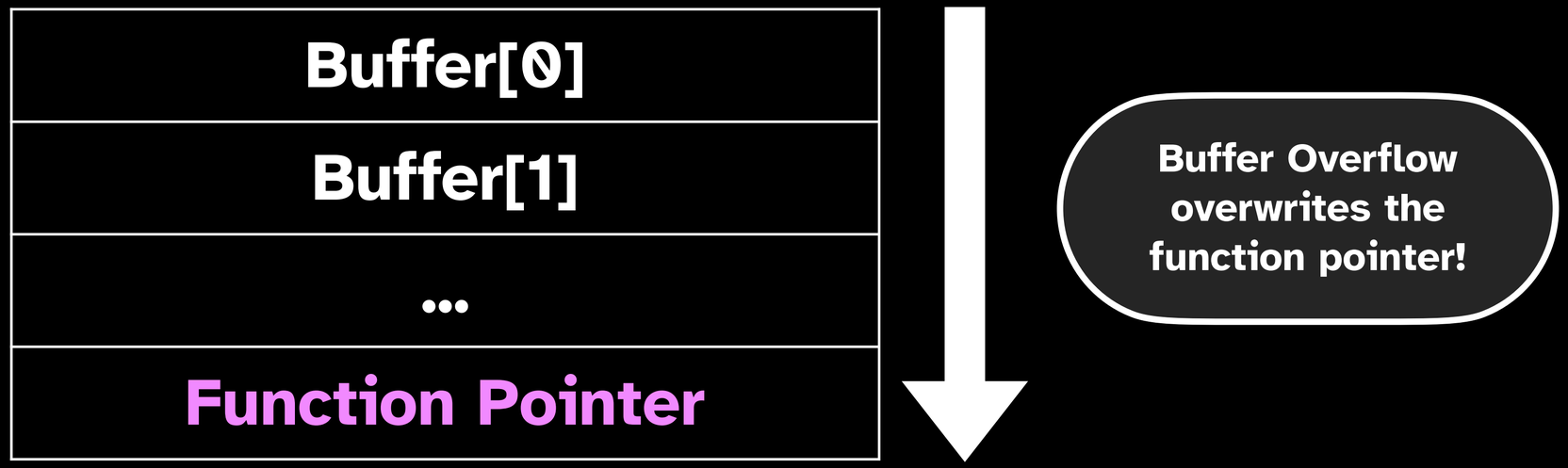
0x10000010



Buffer Overflow

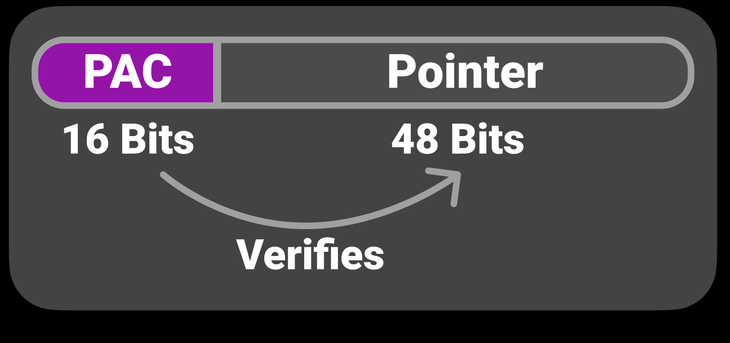


Buffer Overflow



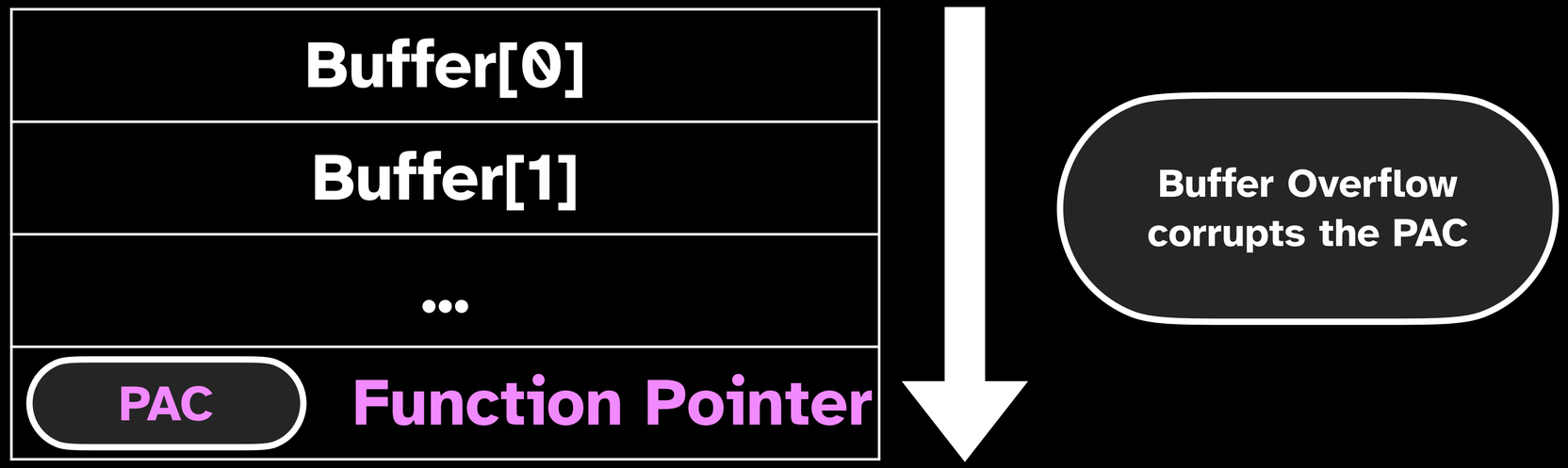
Let's fix this bug with Pointer Authentication.

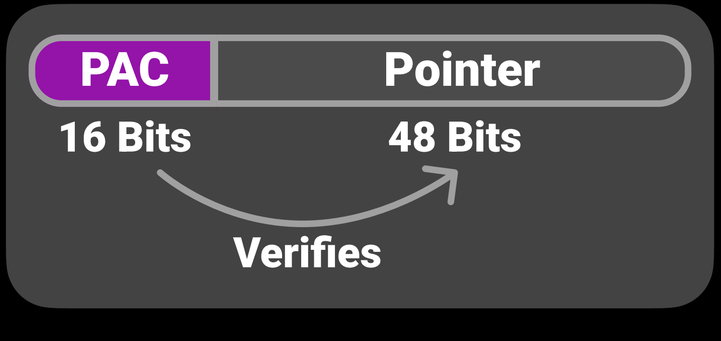
Buffer Overflow



Buffer Overflow

Invalid PAC means we crash!

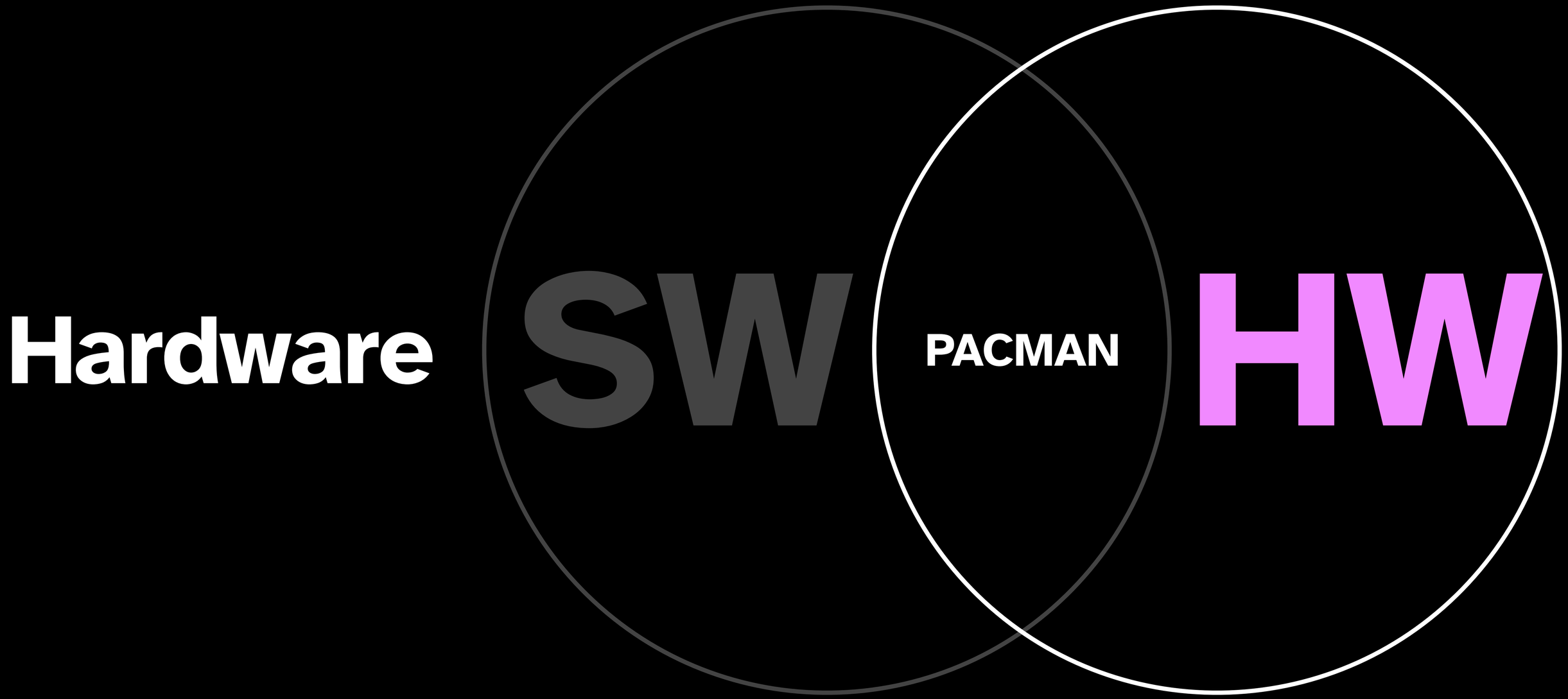




THE

GOAL

Reveal the PAC for an arbitrary pointer without crashing.


Hardware
SW
PACMAN
HW

Break PAC with Hardware Attacks

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

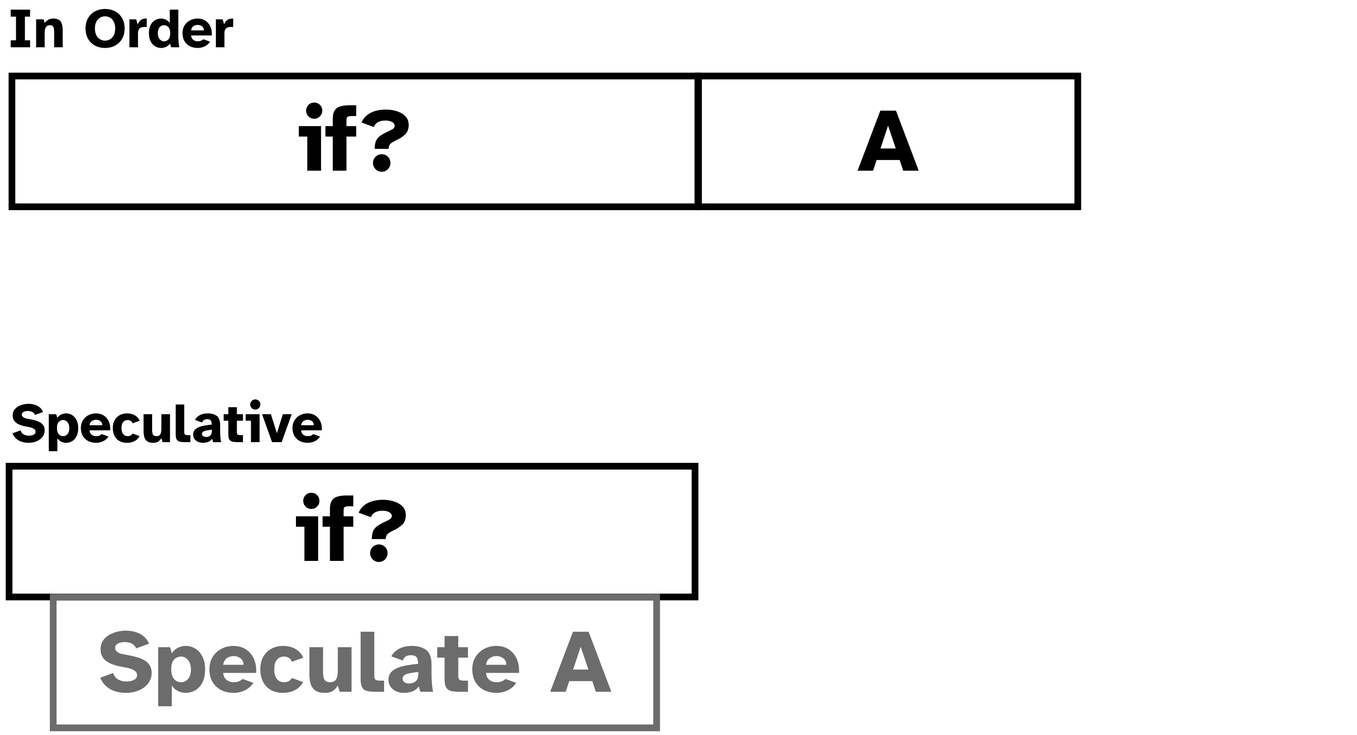
Speculative Execution

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

In Order

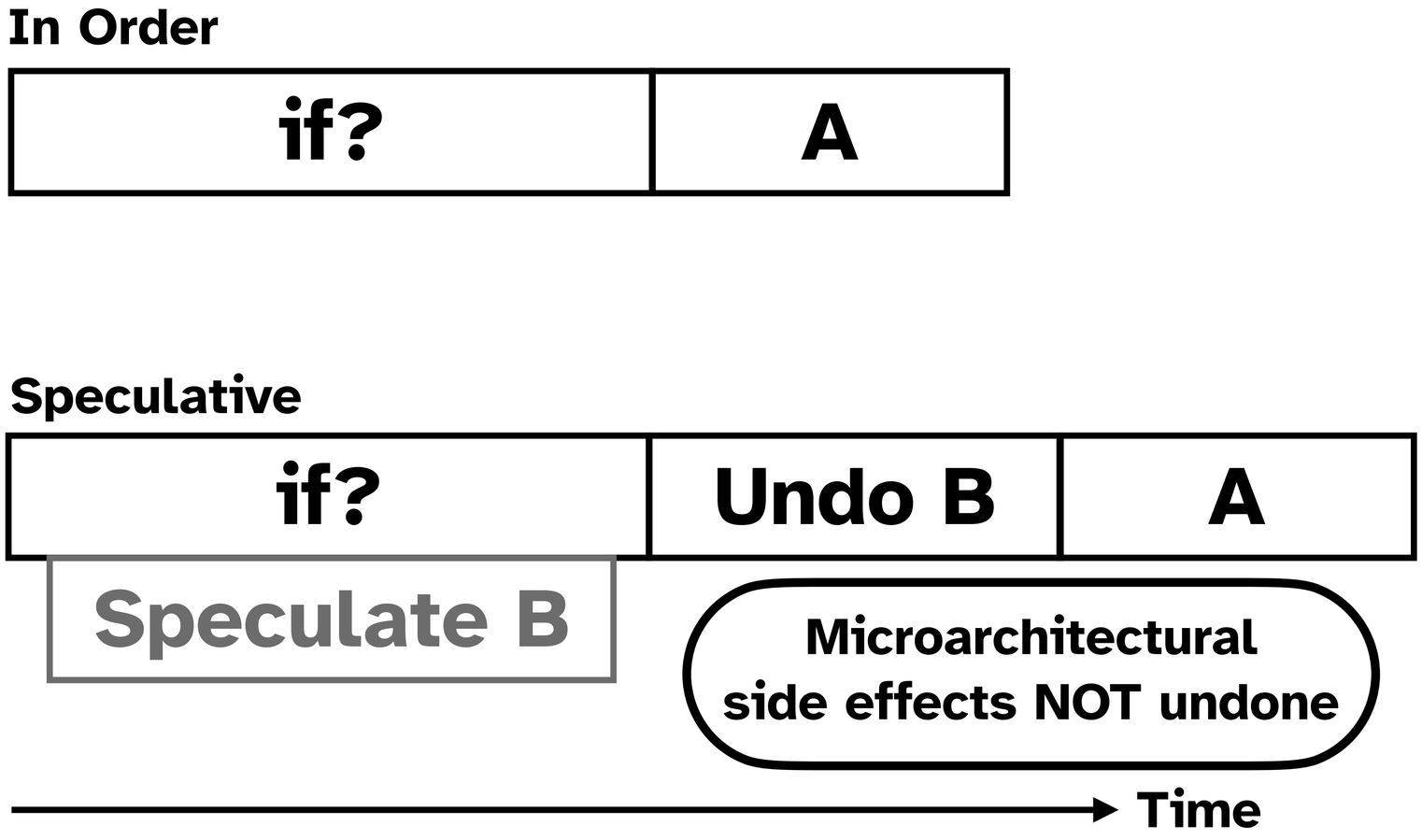
Speculative Execution

```
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)

↓

Leads to other security issues!

Never load

↓

Slow!

Treat them normally

↓

This is how M1 does it.

Speculative PACs

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

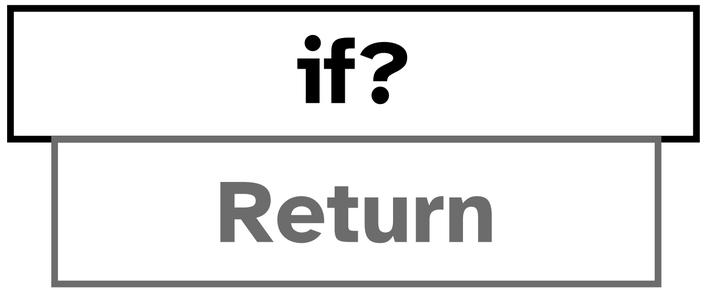
In Order

Doesn't leak anything

Speculative PACs

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

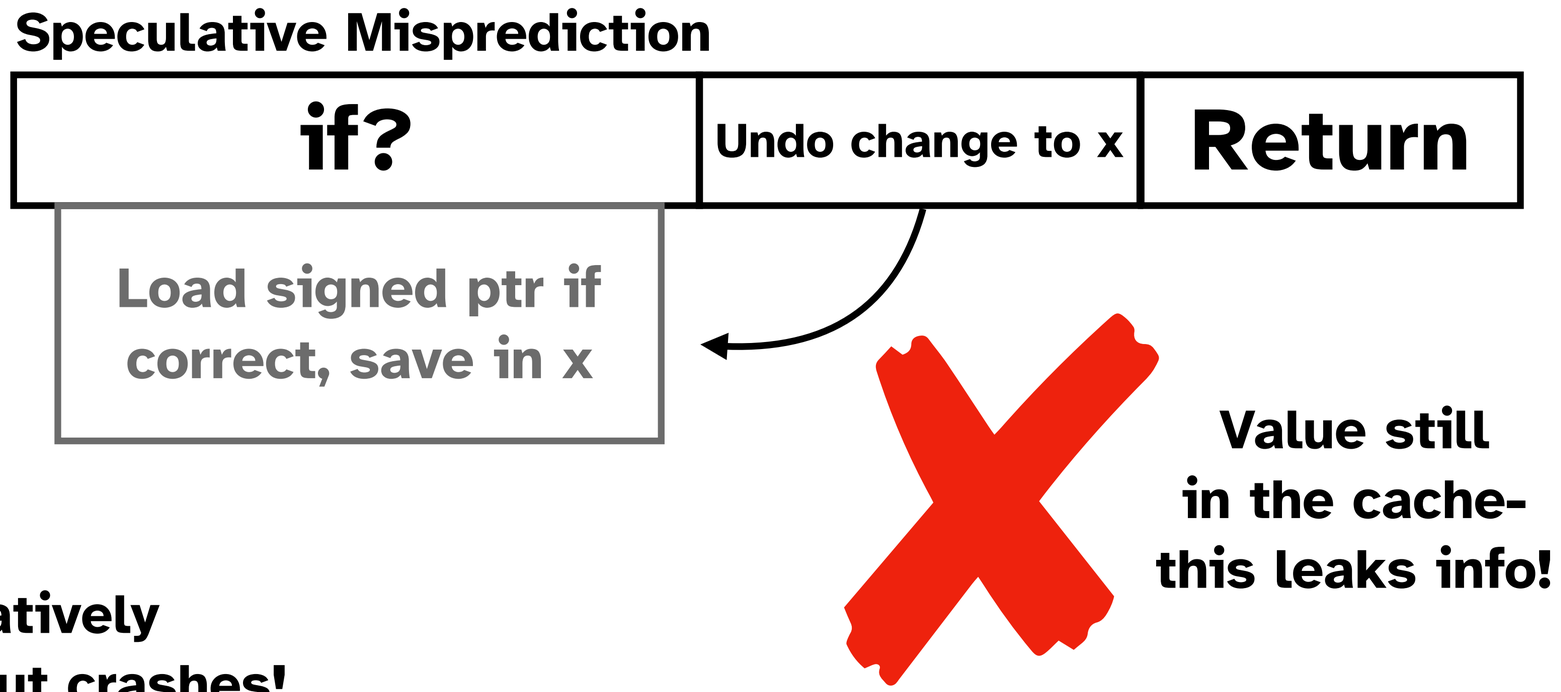
Speculative Correct Prediction



Doesn't leak anything

Speculative PACs

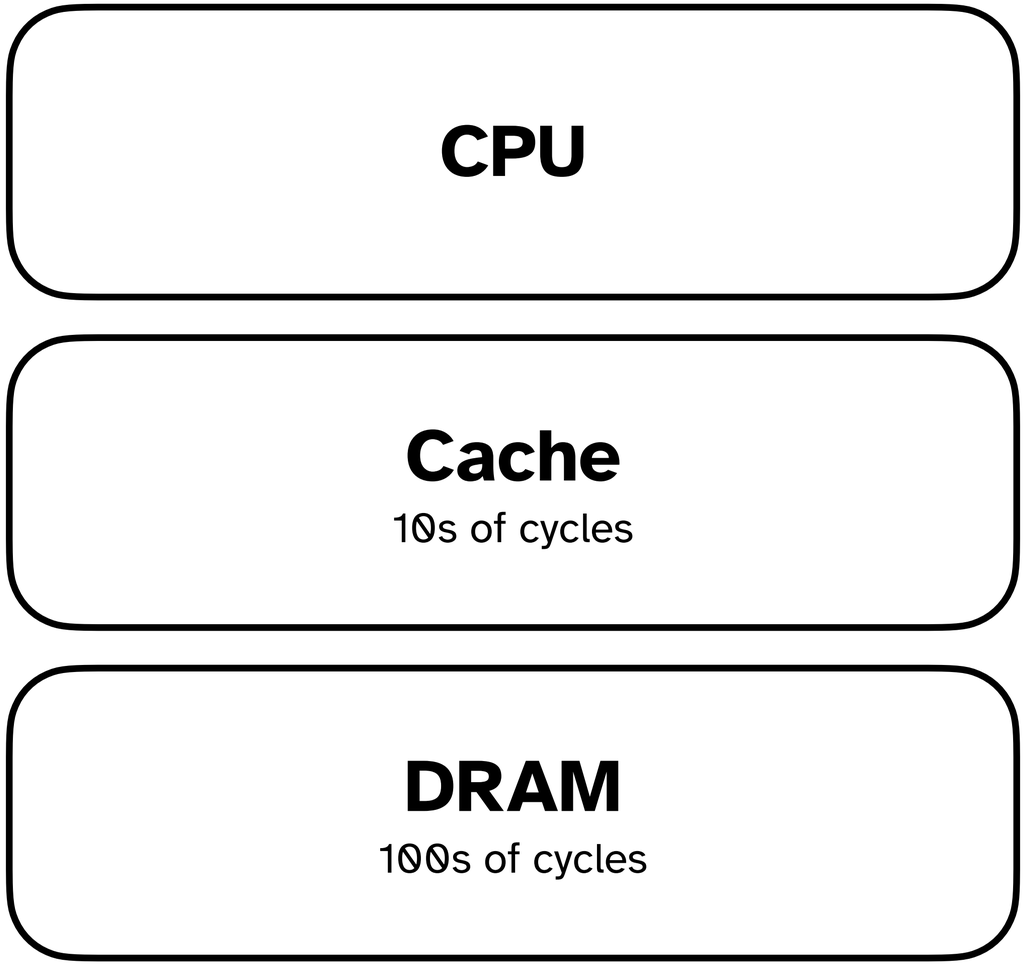
```
if (true)
    return
else
    check signed ptr
    x = load signed ptr
```



Operating on PACs speculatively can leak PAC correctness without crashes!

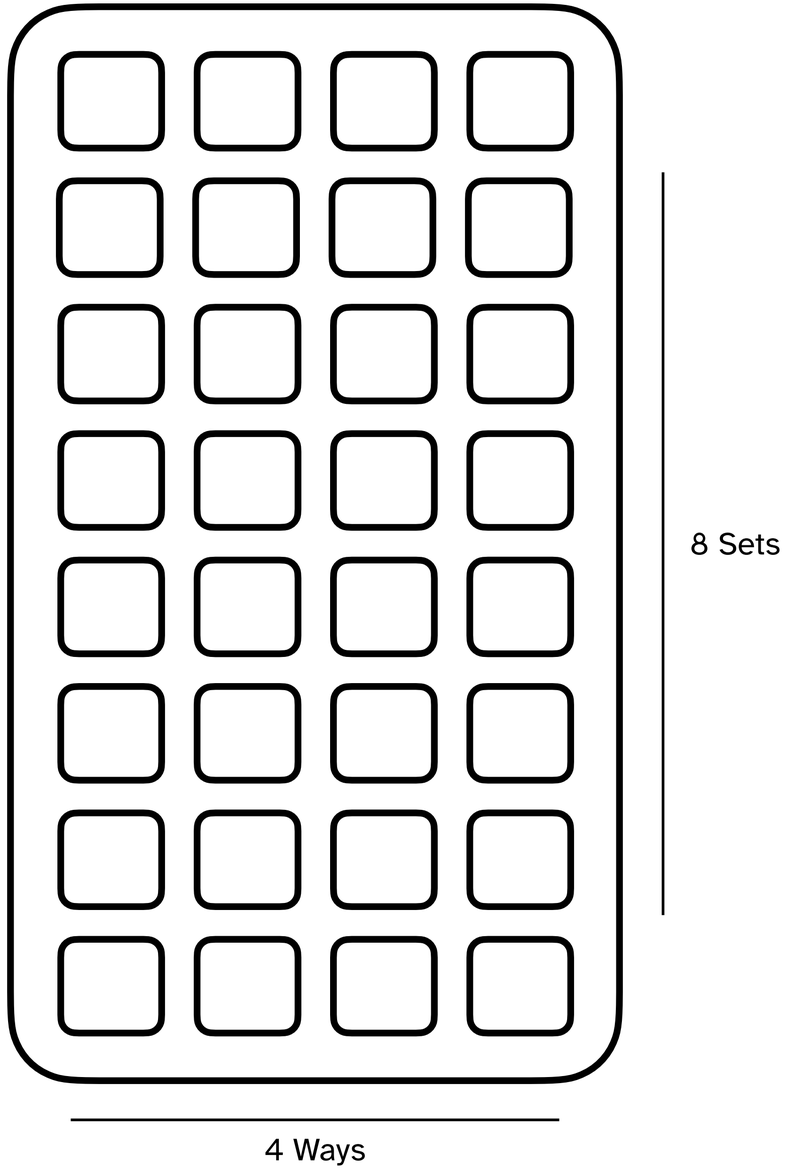
Bird's Eye View

Memory is slow...

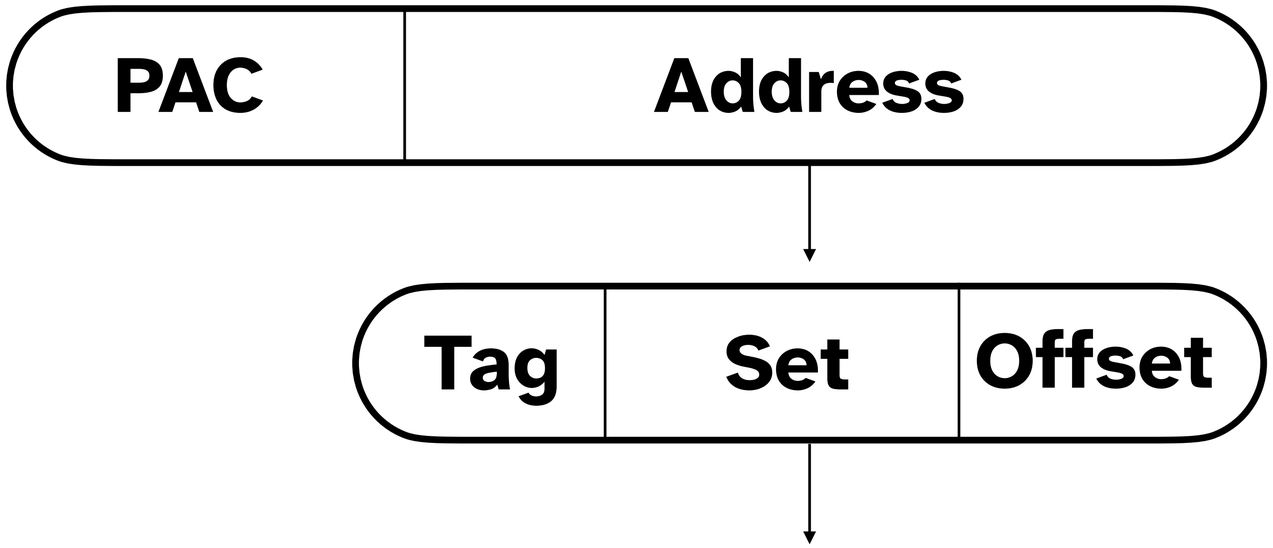


...so add some faster memory!

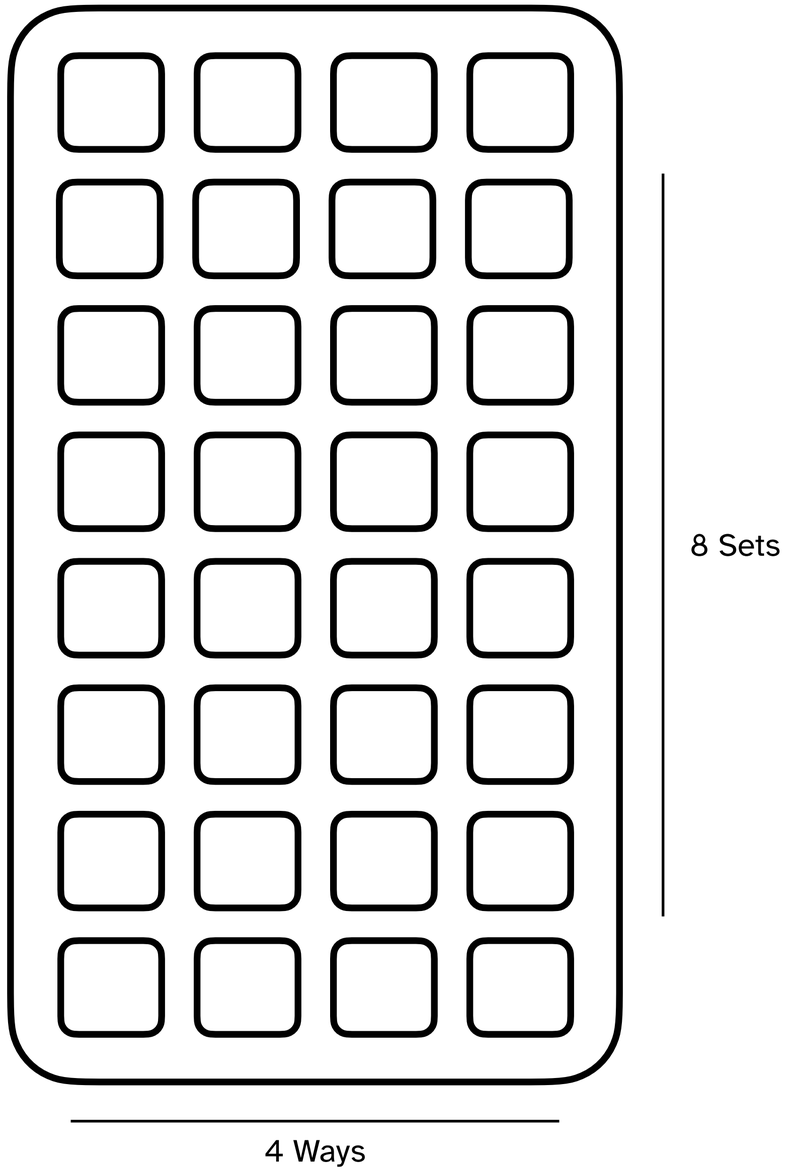
Cache



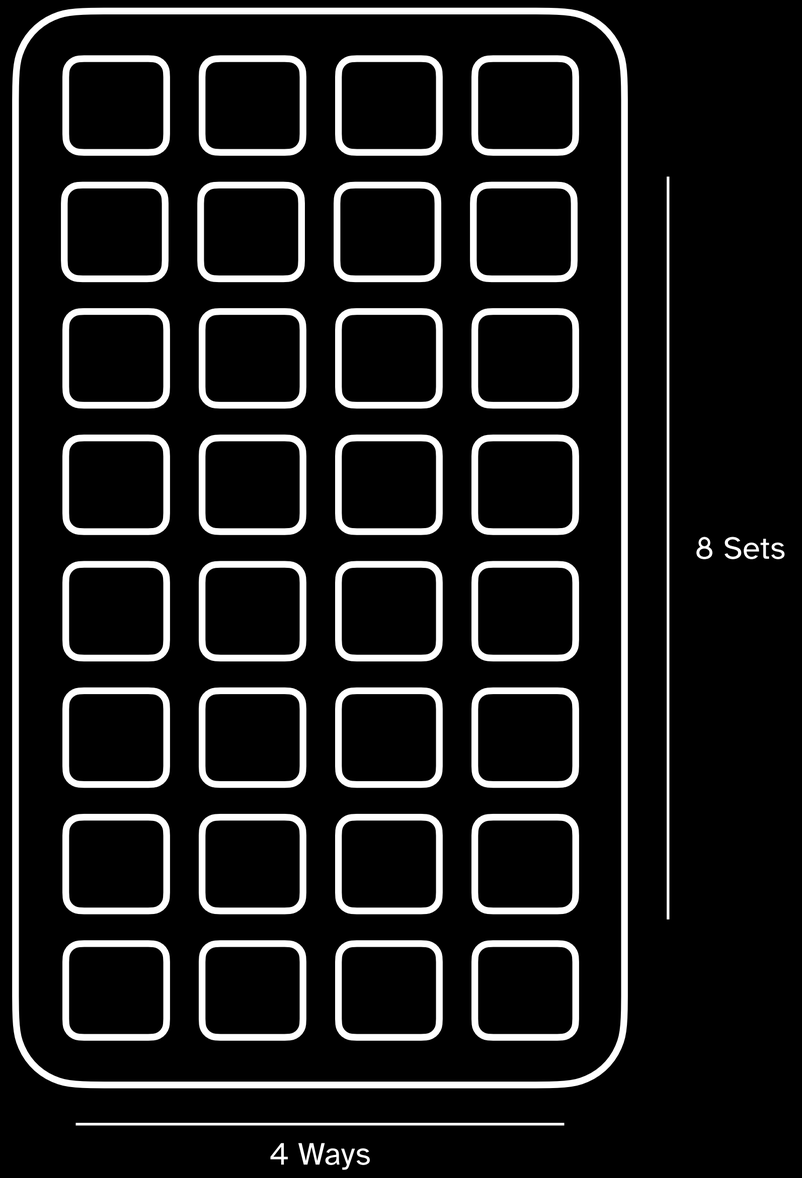
Cache

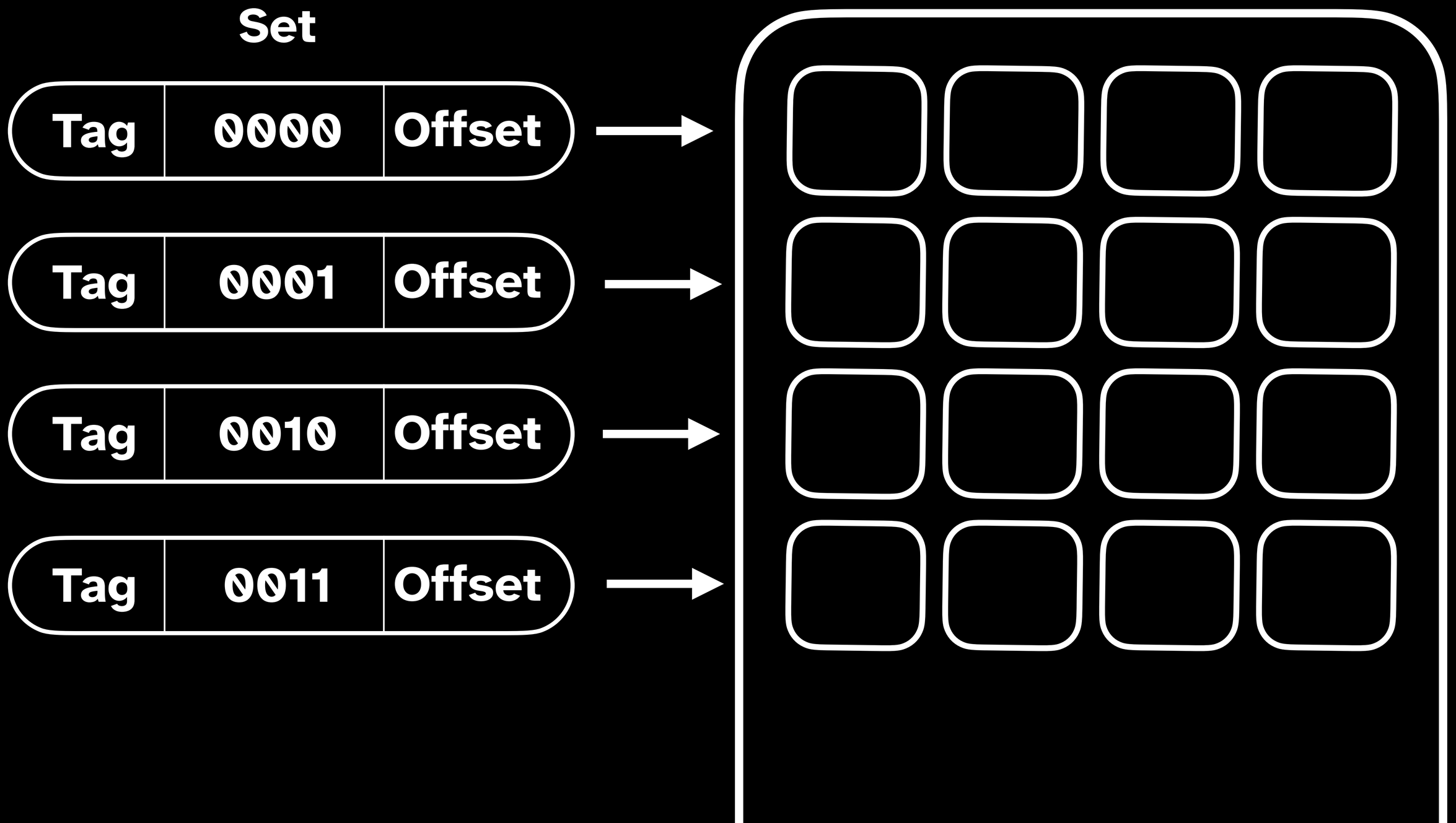


Which row (aka "set") do we map to?

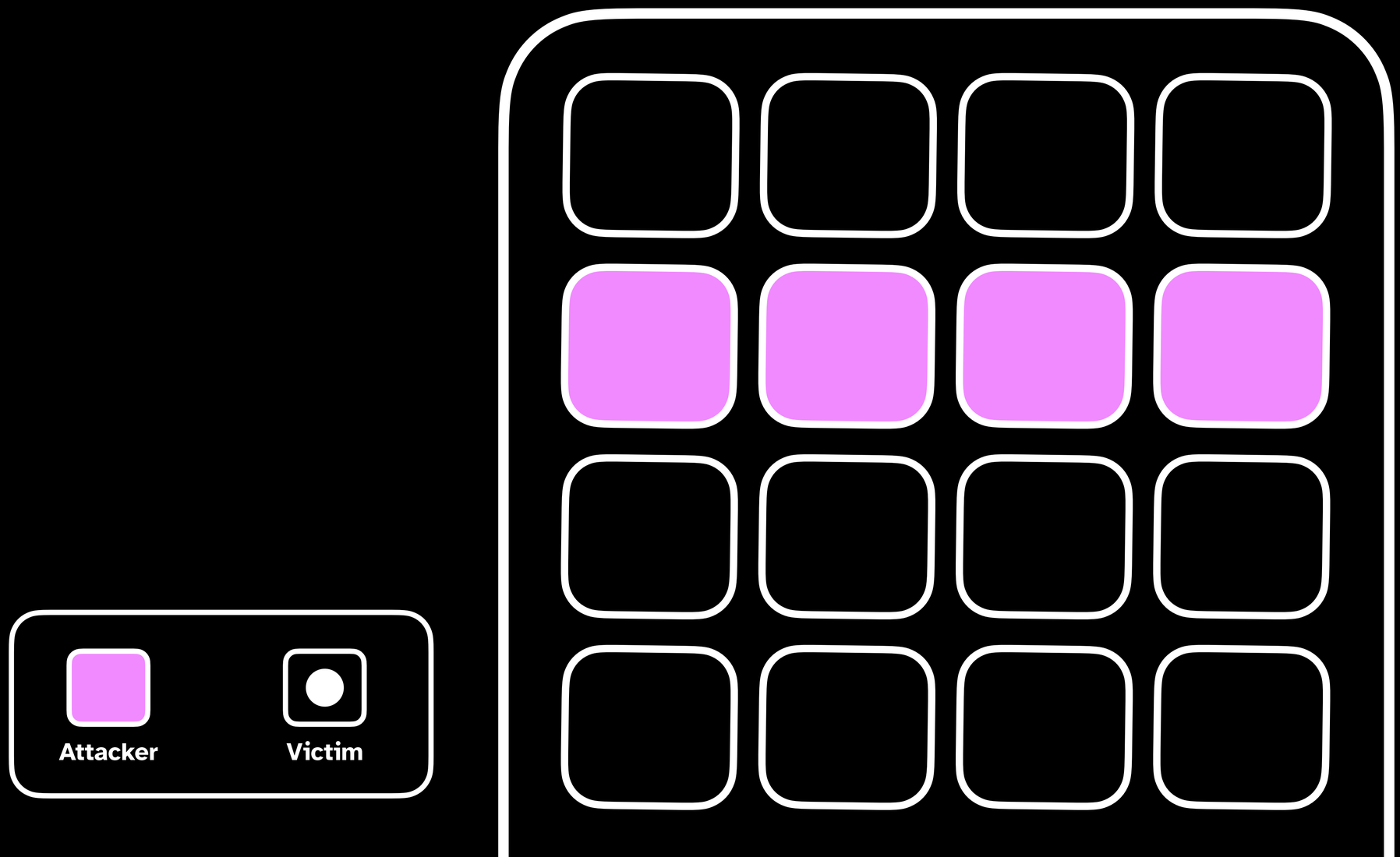


Think like an attacker...

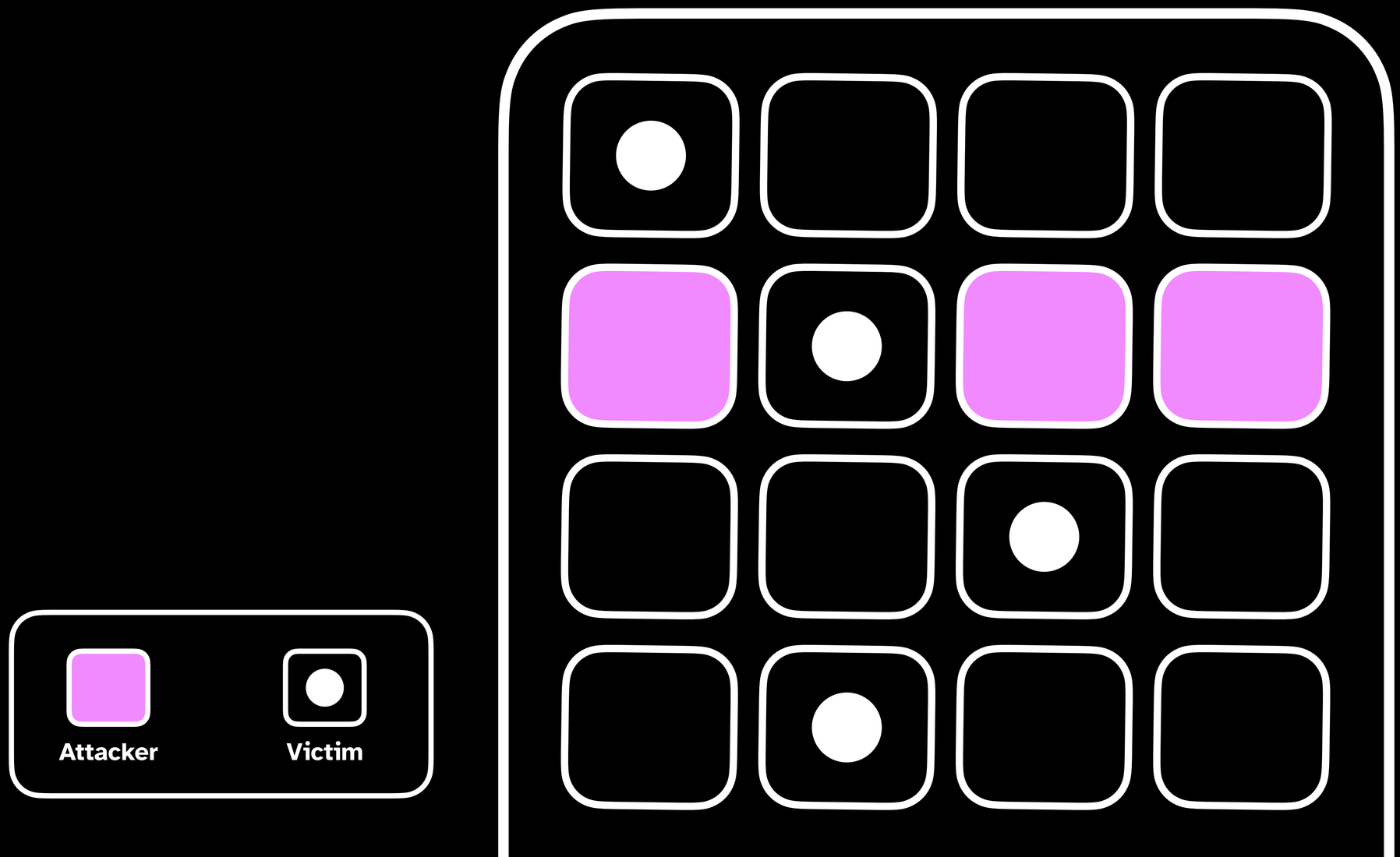



Set
Tag
0000
Offset
Tag
0001
Offset
Tag
0010
Offset
Tag
0011
Offset

1 Fill a set with our data.



2 Let the victim run.



3 Re-access our data.

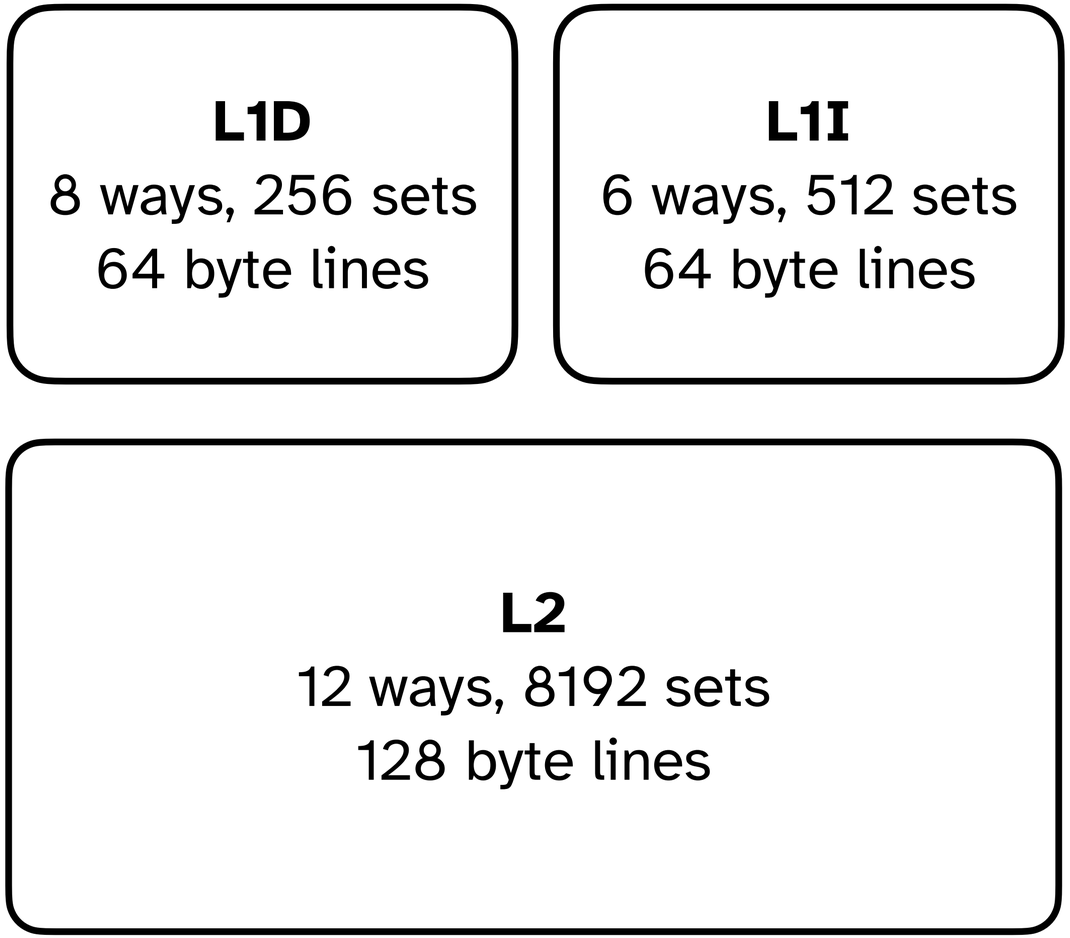
We can tell what the victim did by just watching the cache!

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(good ptr)
  load(verified)
```

1

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

Data PACMAN Attack

Cache

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

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

Taken

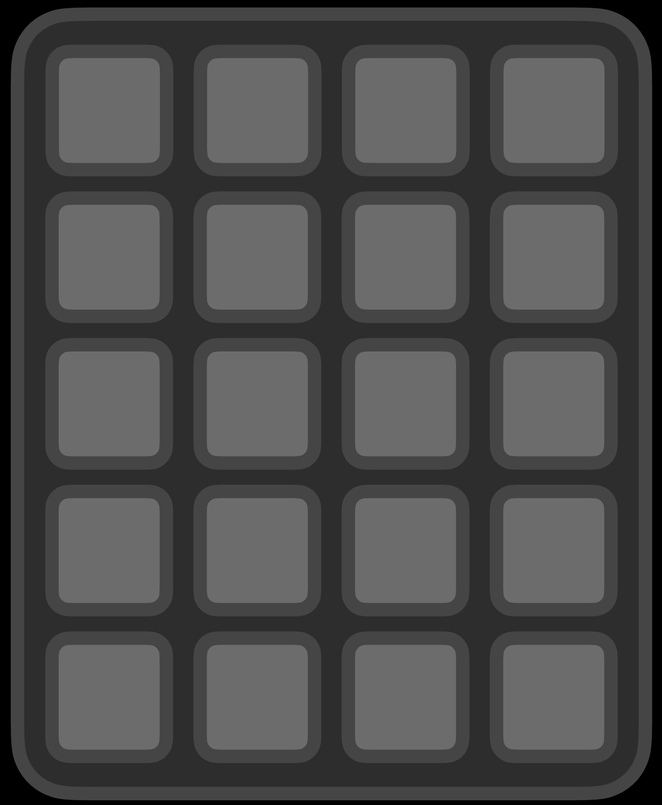
Branch Predictor

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

3

Prime the Cache with an eviction set

Data PACMAN Attack



Cache

Taken

Branch Predictor

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

3

Prime the Cache
with an eviction set

Data PACMAN Attack

Cache

Eviction Set

Taken

Branch Predictor

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

3

Prime the Cache
with an eviction set

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

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

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

Taken

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

Taken

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 incorrect...

4

Call the gadget with the pointer and PAC to guess.

Data PACMAN Attack

No Change

Cache

Taken

Branch Predictor

Speculative

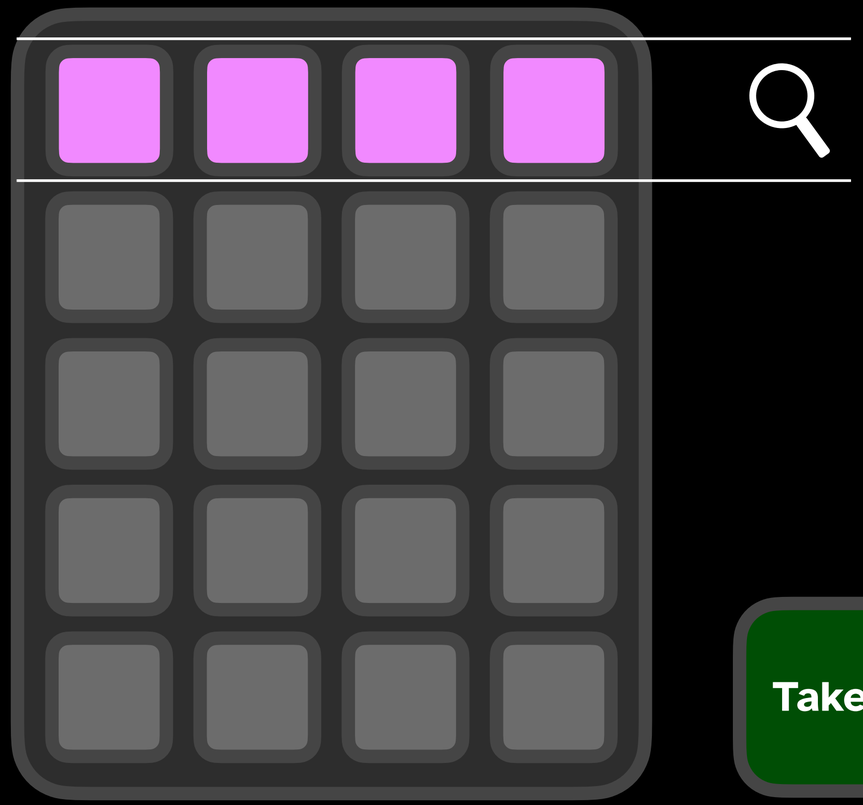
```
if (condition):
  verified = AUT(guess ptr)
  load(verified)
```

Speculative Data Abort!

4

Call the gadget with the pointer and PAC to guess.

Data PACMAN Attack



Cache

Taken

Branch Predictor

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

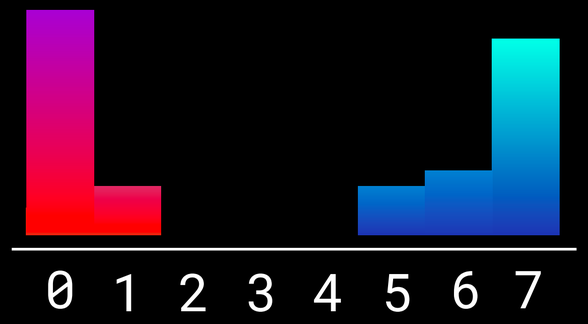
5

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

Ok, let's build it for real.

"Differentiation"

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



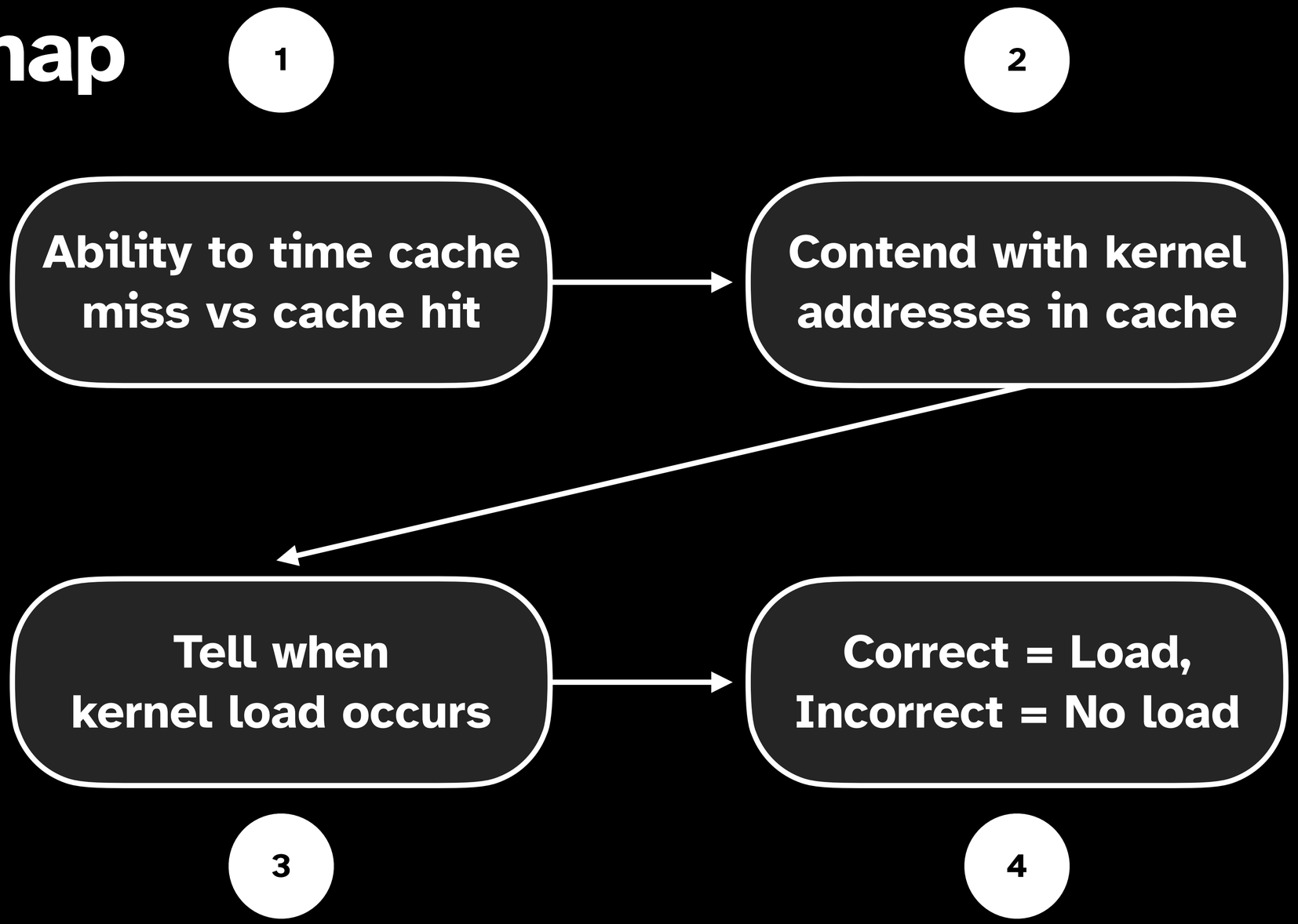
Number of Cache Misses

"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

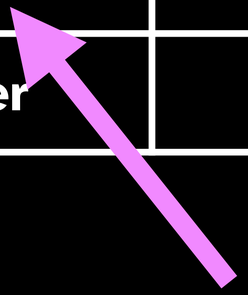
Timer Sources on M1

	Speed	Available to users?
~~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?
~~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

Attack with this



Reverse Engineer with this

Apple Performance Counters

osfmk/arm64/monotonic_arm64.c

```
/* user mode access to configuration registers */
#define PMCR0_USEREN_EN (UINT64_C(1) << 30)
```

```
/*
 * PMC[0-1] are the 48/64-bit fixed counters -- PMC0 is cycles and PMC1 is
 * instructions (see arm64/monotonic.h).
 *
 * PMC2+ are currently handled by kpc.
 */
#define PMC_0_7(X, A) X
    X(6, A); X(7, A)
```

PMCR0_USEREN_EN (bit 30) is not set!

```
/*
 * PMCR0 is the main control register for the performance monitor.  It
 * controls whether the counters are enabled, how they deliver interrupts, and
 * other features.
 */
```

```
#define PMCR0_INIT (PMCR0_INTGEN_INIT | PMCR0_PMI_INIT | PMCR0_PCC_INIT)
```

Masked kpc sysctls

```
/* root kperf node */
SYSCTL_NODE(, OID_AUTO, kpc, CTLFLAG_RW | CTLFLAG_LOCKED, 0,
    "kpc");

/* values */
SYSCTL_PROC(_kpc, OID_AUTO, classes,
    CTLTYPE_INT | CTLFLAG_RD | CTLFLAG_ANYBODY | CTLFLAG_MASKED | CTLFLAG_LOCKED,
    (void*)REQ_CLASSES,
    sizeof(int), kpc_sysctl, "I", "Available classes");

SYSCTL_PROC(_kpc, OID_AUTO, counting,
    CTLTYPE_INT | CTLFLAG_RW | CTLFLAG_ANYBODY | CTLFLAG_MASKED | CTLFLAG_LOCKED,
    (void*)REQ_COUNTING,
    sizeof(int), kpc_sysctl, "I", "PMCs counting");

SYSCTL_PROC(_kpc, OID_AUTO, thread_counting,
    CTLTYPE_INT | CTLFLAG_RW | CTLFLAG_ANYBODY | CTLFLAG_MASKED | CTLFLAG_LOCKED,
    (void*)REQ_THREAD_COUNTING,
    sizeof(int), kpc_sysctl, "I", "Thread accumulation");

SYSCTL_PROC(_kpc, OID_AUTO, pmu_version,
    CTLTYPE_INT | CTLFLAG_RD | CTLFLAG_ANYBODY | CTLFLAG_MASKED | CTLFLAG_LOCKED,
    (void *)REQ_PMU_VERSION,
    sizeof(int), kpc_sysctl, "I", "PMU version for hardware");

/* faux values */
SYSCTL_PROC(_kpc, OID_AUTO, config_count,
    CTLTYPE_INT | CTLFLAG_RW | CTLFLAG_ANYBODY | CTLFLAG_MASKED | CTLFLAG_LOCKED,
    (void*)REQ_CONFIG_COUNT,
    sizeof(int), kpc_sysctl, "S", "Config count");

SYSCTL_PROC(_kpc, OID_AUTO, counter_count,
    CTLTYPE_INT | CTLFLAG_RW | CTLFLAG_ANYBODY | CTLFLAG_MASKED | CTLFLAG_LOCKED,
    (void*)REQ_COUNTER_COUNT,
    sizeof(int), kpc_sysctl, "S", "Counter count");

SYSCTL_PROC(_kpc, OID_AUTO, sw_inc,
    CTLTYPE_INT | CTLFLAG_RW | CTLFLAG_ANYBODY | CTLFLAG_MASKED | CTLFLAG_LOCKED,
    (void*)REQ_SW_INC,
```

Used by private kperf.framework

```
int _kpc_get_cpu_counters(int param_1,uint param_2,undefined4 *param_3,void *param_4)

{
  size_t sVar1;
  int iVar2;
  int iVar3;
  int iVar4;
  undefined8 *puVar5;
  ulong local_60;
  int local_58;
  int local_54;
  size_t local_50;
  uint local_44;

  local_60 = (ulong)param_2;
  local_58 = 1;
  if (param_1 == 0) {
    iVar3 = 8;
  }
  else {
    local_60 = local_60 | 0x80000000;
    iVar3 = _kpc_cpu_count(&local_58);
    if (iVar3 != 0) {
      return -1;
    }
    iVar3 = local_58 << 3;
  }
  local_54 = 0;
  local_50 = 4;
  local_44 = param_2;
  iVar4 = __auth_stubs::_sysctlbyname("kpc.counter_count",&local_54,&local_50,&local_44,4);
  iVar2 = 0;
  if (-1 < iVar4) {
    iVar2 = local_54;
  }
```

Masked kpc sysctls

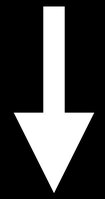
```
// Most sysctls require an access check, but a few are public.
switch ((uintptr_t) arg1) {
case REQ_CLASSES:
case REQ_CONFIG_COUNT:
case REQ_COUNTER_COUNT:
    // These read-only sysctls are public.
    break;

default:
    // Require kperf access to read or write anything else.
    // This is either root or the blessed pid.
    if ((ret = ktrace_read_check())) {
        ktrace_unlock();
        return ret;
    }
    break;
}
```

Require root

Extra latency from trip to kernel

/dev/perfmon_uncore **and** /dev/perfmon_core

```
bash-3.2$ ls -lah /dev/perfmon_uncore
crw-rw-rw-  1 root  wheel  0x1e000000 Jul 23 21:28 /dev/perfmon_uncore
```



bsd/dev/dev_perfmon.c

/dev/perfmon_uncore and /dev/perfmon_core

```
static int
perfmon_dev_ioctl(dev_t dev, unsigned long cmd, char *arg,
    int __unused fflag, proc_t __unused p)
{
    struct perfmon_device *device = perfmon_dev_get_device(dev);
    struct perfmon_source *source = perfmon_dev_get_source(dev);
    int ret = 0;

    lck_mtx_lock(&device->pmdv_mutex);

    unsigned short reg_count = source->ps_layout.pl_reg_count;
    unsigned short unit_count = source->ps_layout.pl_unit_count;

    switch (cmd) {
    case PERFMON_CTL_GET_LAYOUT:;
        struct perfmon_layout *layout = (void *)arg;
        *layout = source->ps_layout;
        ret = 0;
        break;

    case PERFMON_CTL_LIST_REGS: {
        user_addr_t uptr = *(user_addr_t *)(void *)arg;
        size_t names_size = reg_count * sizeof(source->ps_register_names[0]);
        ret = copyout(source->ps_register_names, uptr, names_size);
        break;
    }

    case PERFMON_CTL_SAMPLE_REGS: {
        user_addr_t uptr = *(user_addr_t *)(void *)arg;
        uint64_t *sample_buf = device->pmdv_copyout_buf;
        size_t sample_size = reg_count * unit_count * sizeof(sample_buf[0]);
        perfmon_source_sample_regs(source, sample_buf, reg_count);
        ret = copyout(sample_buf, uptr, sample_size);
        break;
    }
```

No root required!

Still takes a trip to the kernel

/dev/perfmon_uncore and /dev/perfmon_core

UPMCR0, UPMESR0, UPMESR1, UPMCR1, UPMECM0, UPMECM1, UPMECM2, UPMECM3, AFLATCTL1, AFLATCTL2, AFLATCTL3, AFLATCTL4, AFLATCTL5, AFLATVALBIN0, AFLATVALBIN1, AFLATVALBIN2, AFLATVALBIN3, AFLATVALBIN4, AFLATVALBIN5, AFLATVALBIN6, AFLATVALBIN7, AFLATINFLO, AFLATINFHI, UPMC0, UPMC1, UPMC2, UPMC3, UPMC4, UPMC5, UPMC6, UPMC7, UPMC8, UPMC9, UPMC10, UPMC11, UPMC12, UPMC13, UPMC14, UPMC15, PMCR0, PMCR1, PMCR2, PMCR3, PMCR4, PMESR0, PMESR1, PMSR, OPMAT0, OPMAT1, PMCR_BVRNG4, PMCR_BVRNG5, PM_MEMFLT_CTL23, PM_MEMFLT_CTL45, PMMMAP, PMC0, PMC1, PMC2, PMC3, PMC4, PMC5, PMC6, PMC7, PMC8, PMC9

Can read all of these regs via ioctl without root!

tests/perfmon_tests.c has sample code

If we just need timers, why not use a kext to enable them?

PACMAN

Download KDK Kernel → ./patch kernel.development → Rebuild kernelcache & restart macOS

Multi-Threaded Counter

"Actually works pretty well"

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

Multi-Threaded Counter

"Actually works pretty well"

```
asm!{
  "eor x0, x0, x0",
  "1:",
  "str x0, [{cntr}]",
  "add x0, x0, 1",
  "b 1b",
  cntr = in(reg) &mut counter::CTR as
                 *mut u64 as u64,
}
```

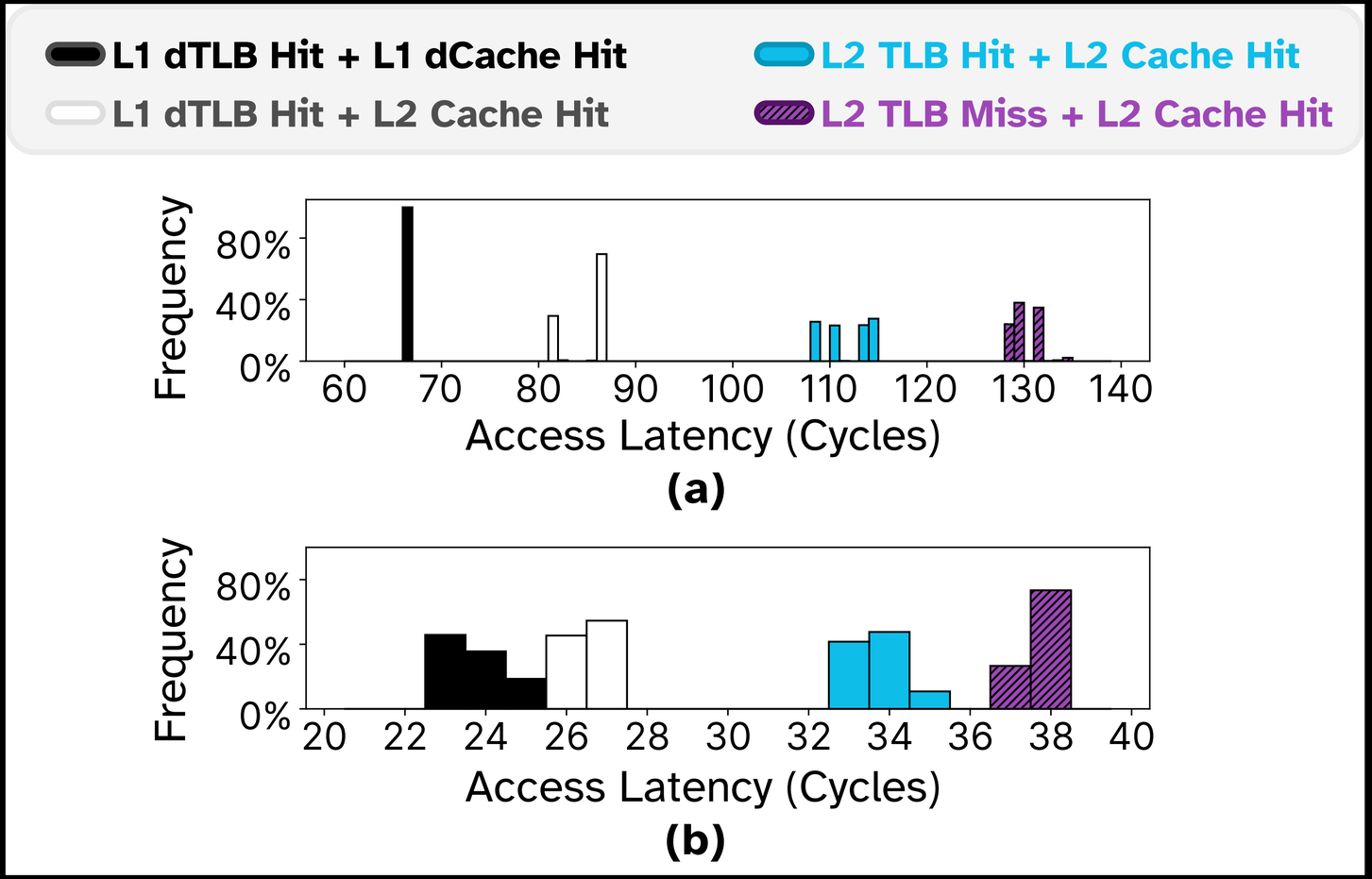
Counting Thread

```
let t1 : u64;
let t2 : u64;
asm!{
  "dsb sy",
  "isb",
  "ldr {t1}, [{cntr}]",
  "isb",
  "ldr {val_out}, [{addr}]",
  "isb",
  "ldr {t2}, [{cntr}]",
  "isb",
  "dsb sy",
  val_out = out(reg) _,
  addr = in(reg) addr,
  cntr = in(reg) &mut counter::CTR as *mut u64 as u64,
  t1 = out(reg) t1,
  t2 = out(reg) t2,
}
return t2 - t1;
```

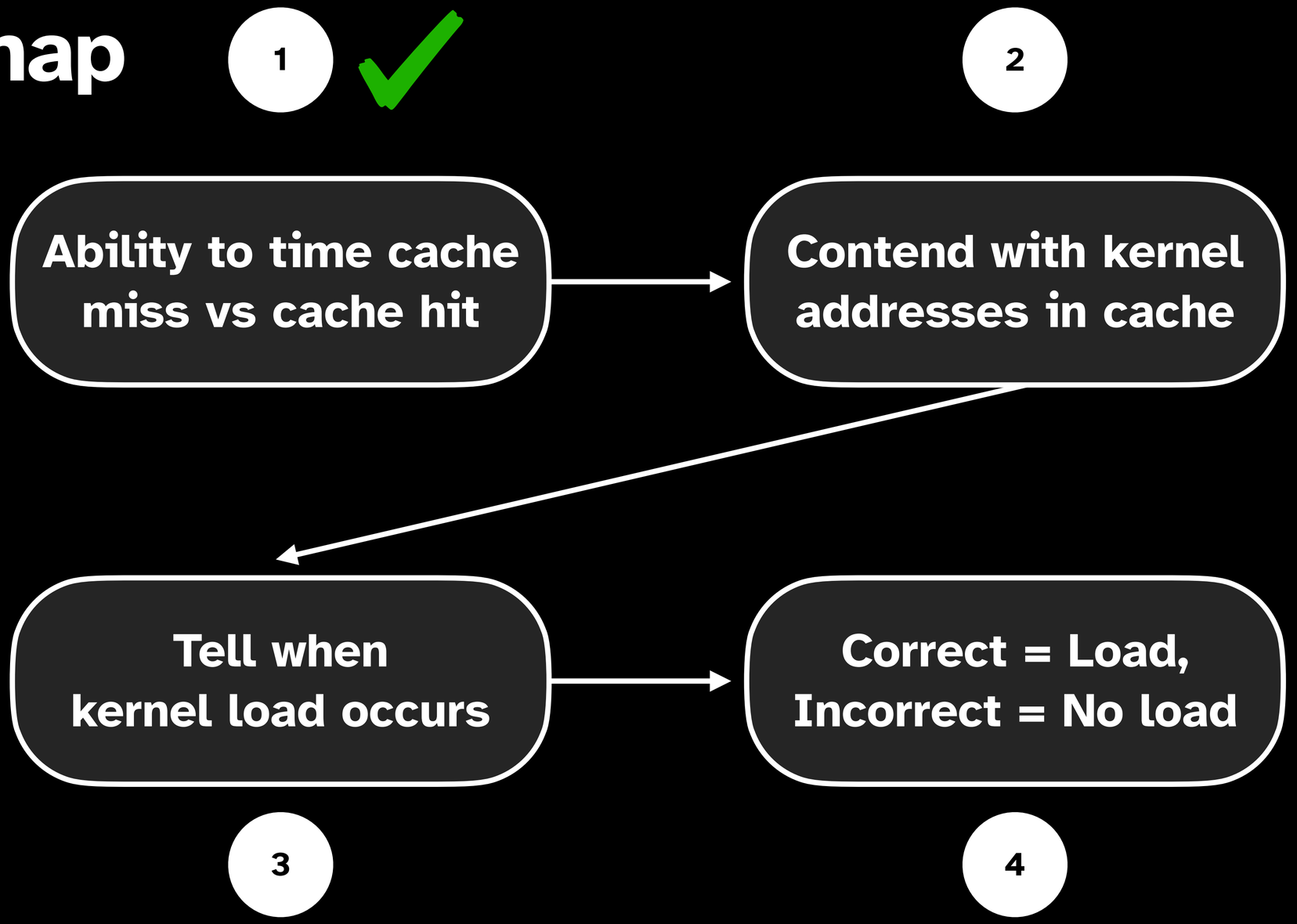
Timing a data access

Apple Performance Counters

Multithreaded Timer



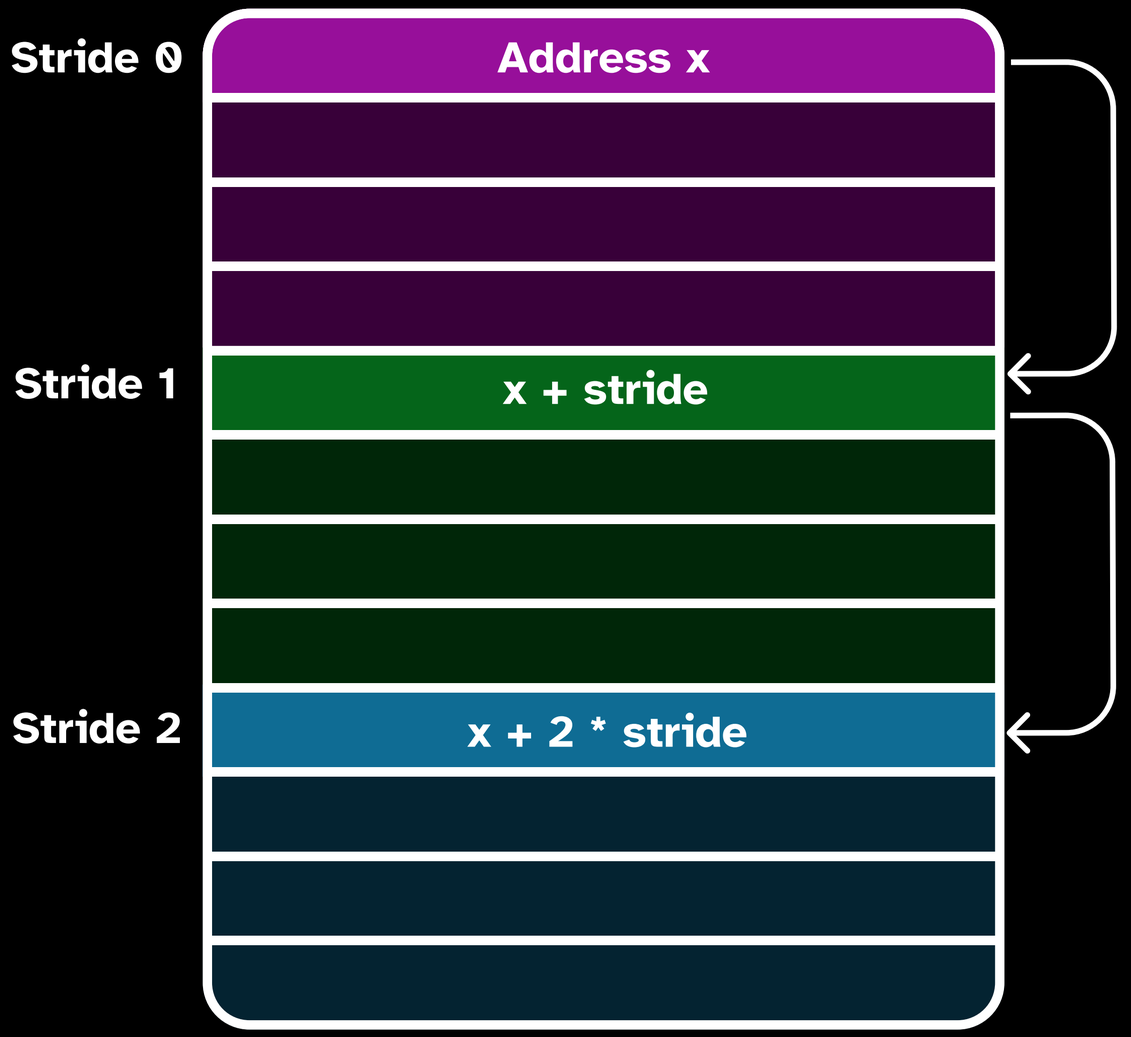
Roadmap


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`

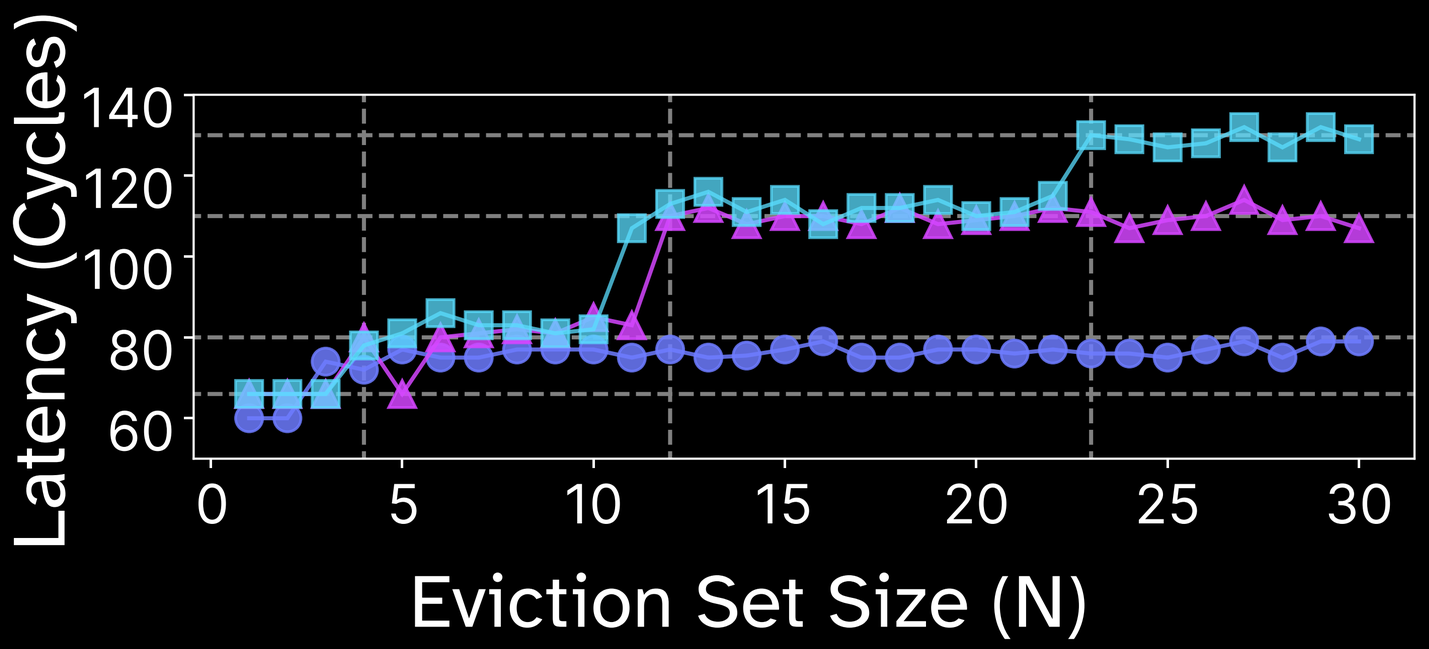
1. Load the address we want to measure
2. Load a bunch of addresses that might **evict** it
3. Check if we evicted it by **reloading**

TLB + Cache Interactions




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

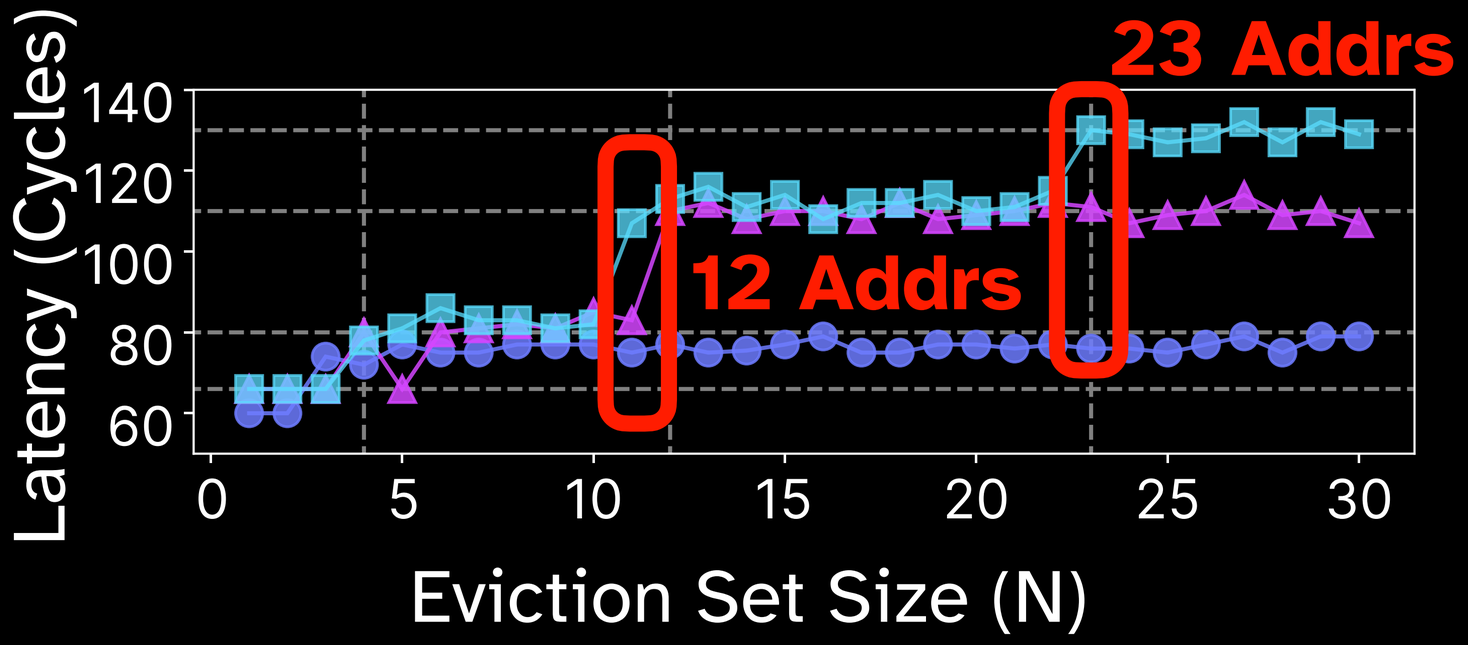
TLB + Cache Results



Latency from dTLB/dCache conflicts

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

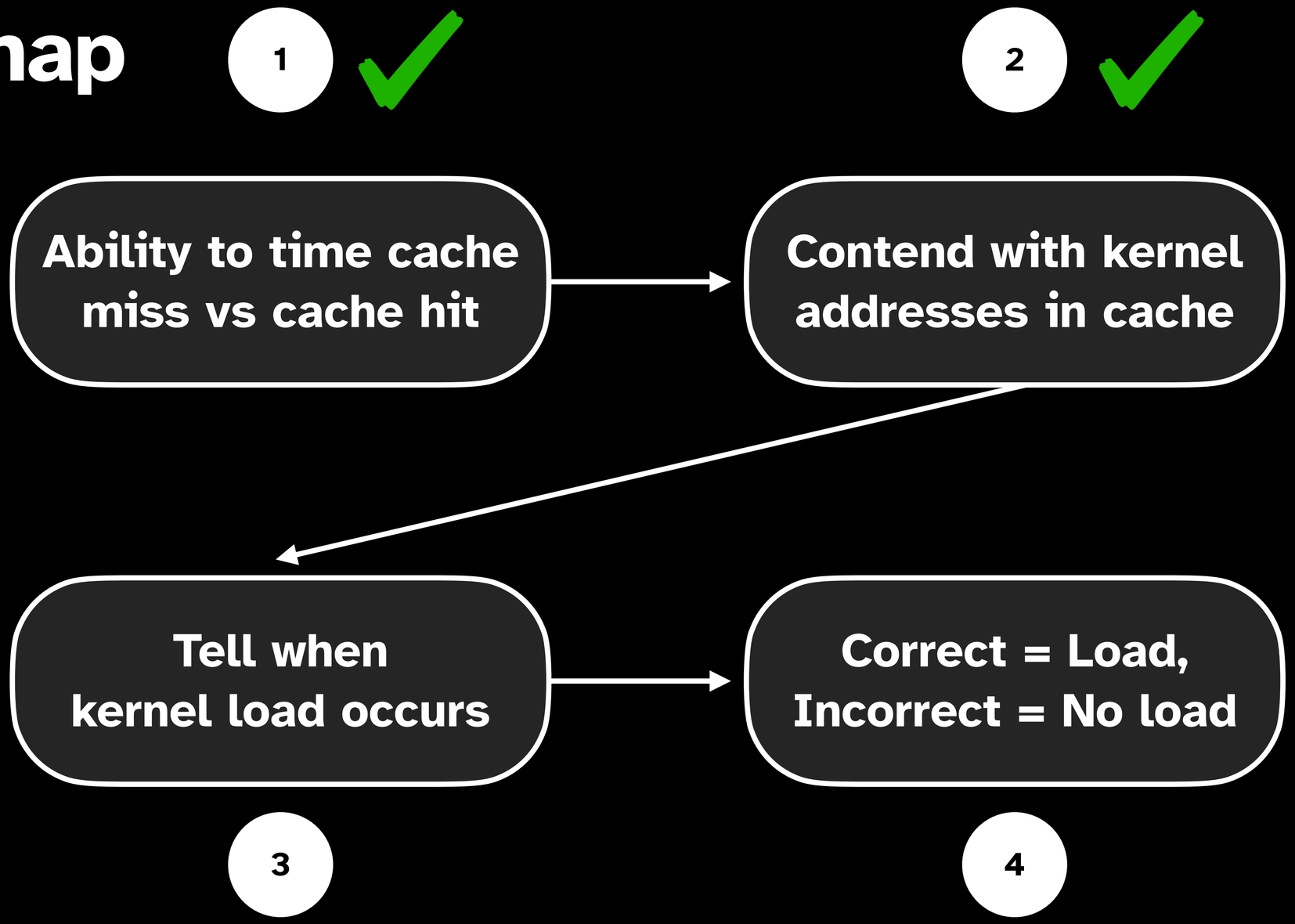
TLB + Cache Results



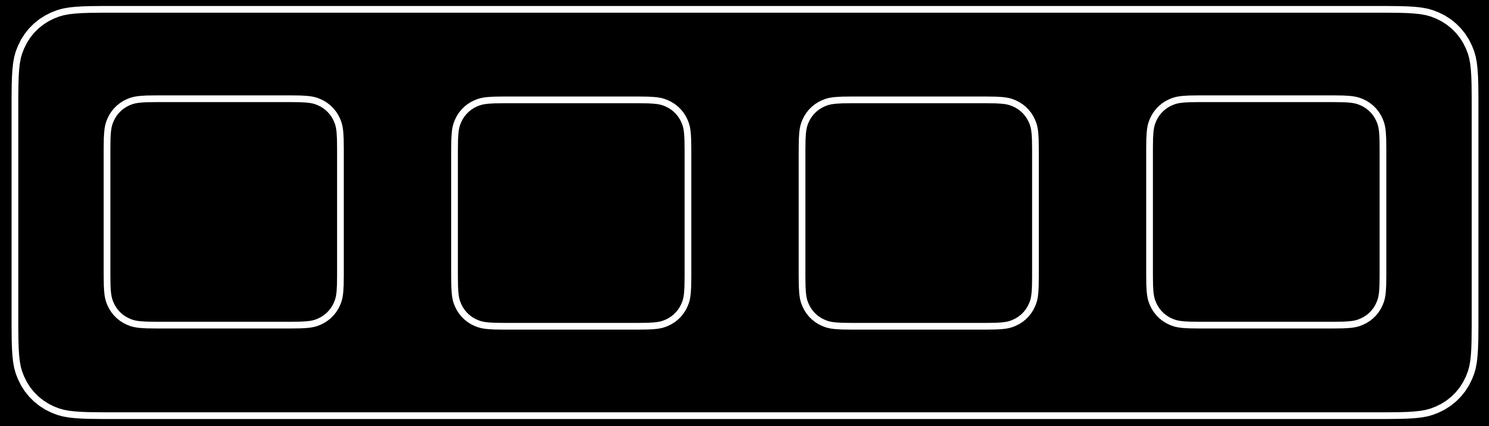
Latency from dTLB/dCache conflicts

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

Roadmap

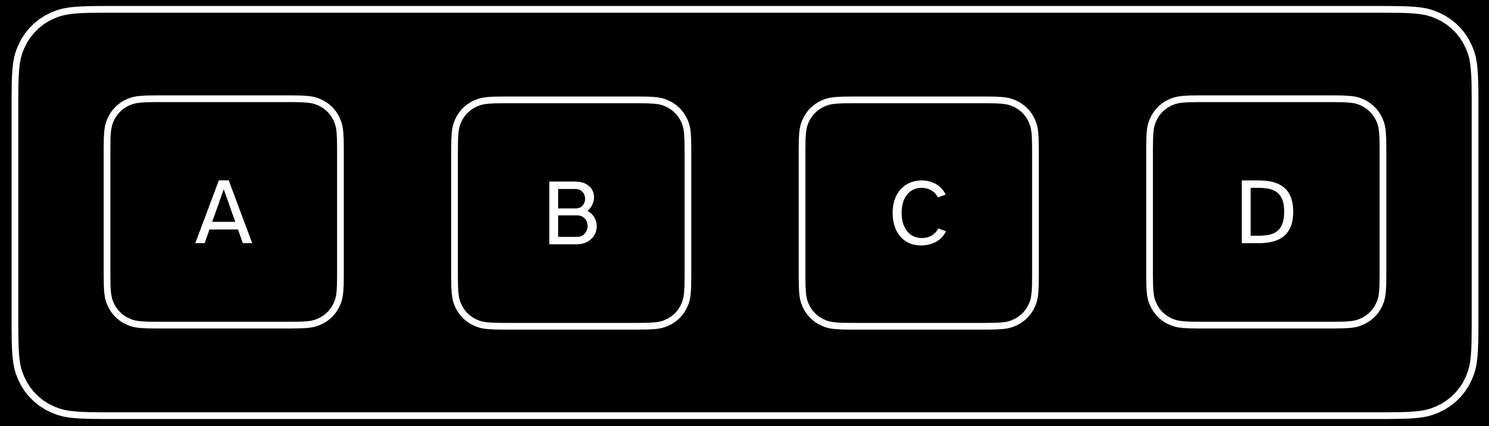

1
2
Ability to time cache miss vs cache hit
Contend with kernel addresses in cache
Tell when kernel load occurs
Correct = Load, Incorrect = No load
3
4

LRU Replacement Policy



A single
cache set

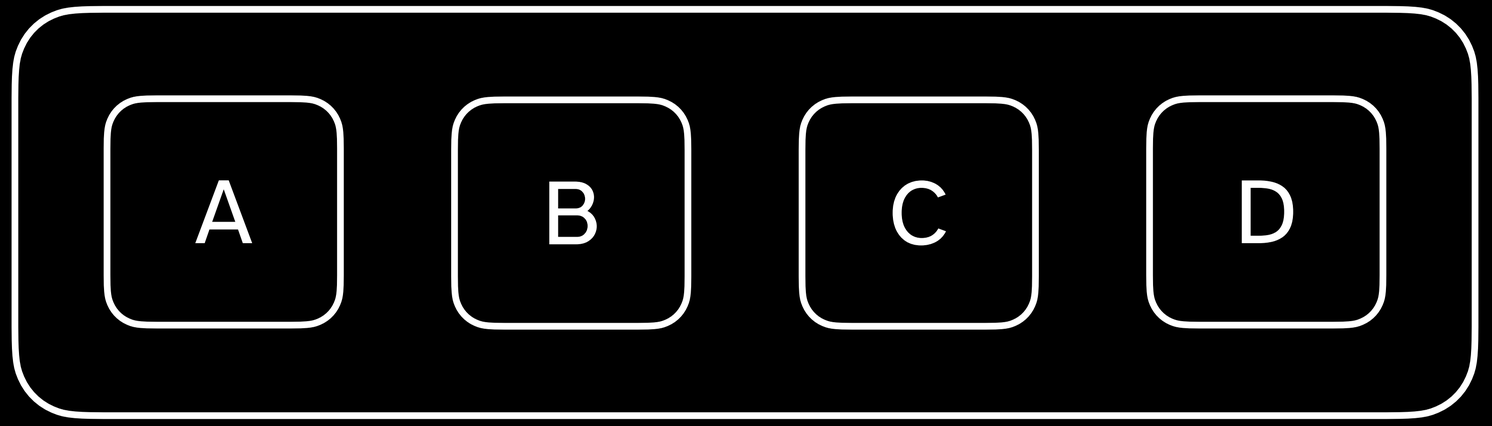
LRU Replacement Policy



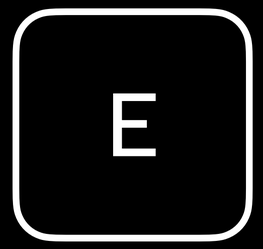
A single cache set

Load A, B, C, D in order

LRU Replacement Policy

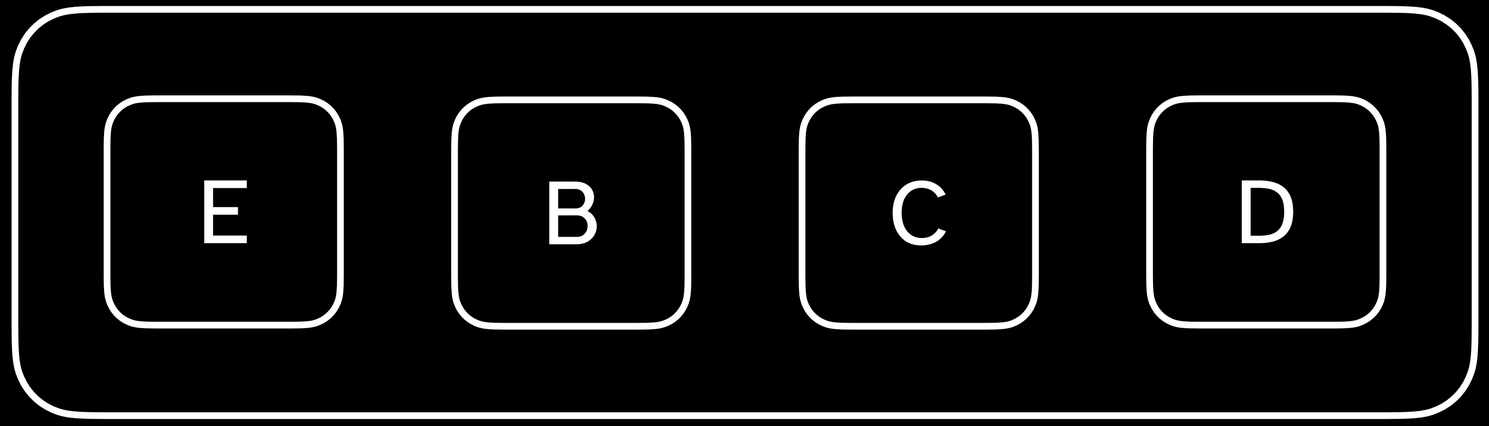


A single cache set

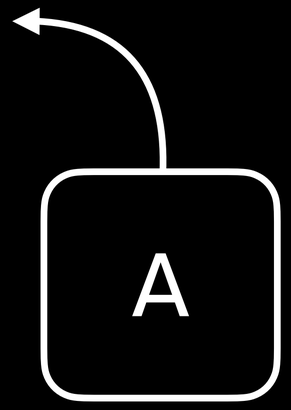


Now we want to load E

LRU Replacement Policy

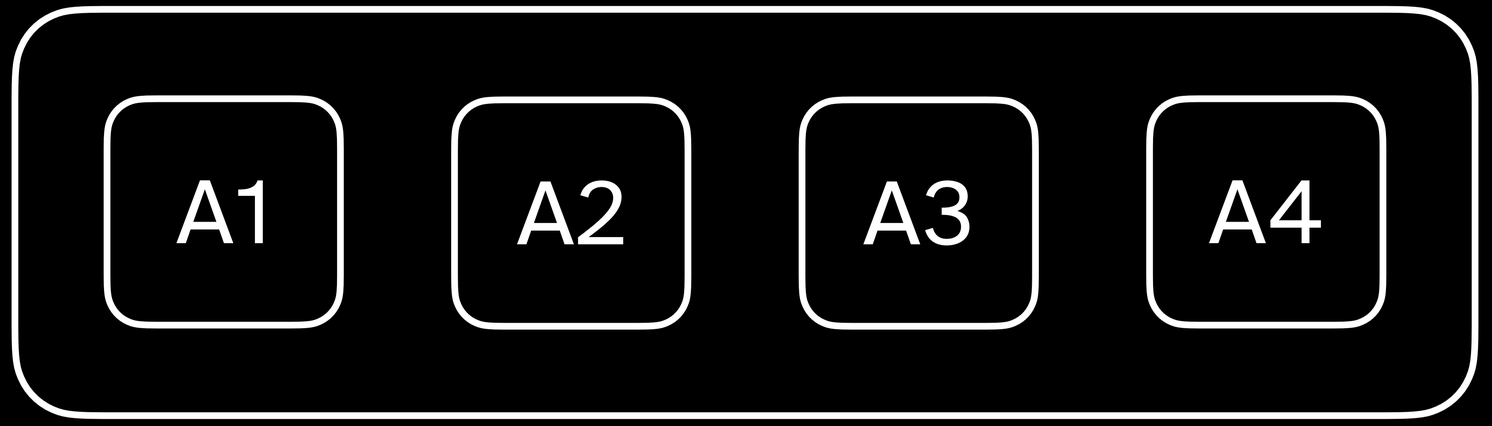


A single cache set



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

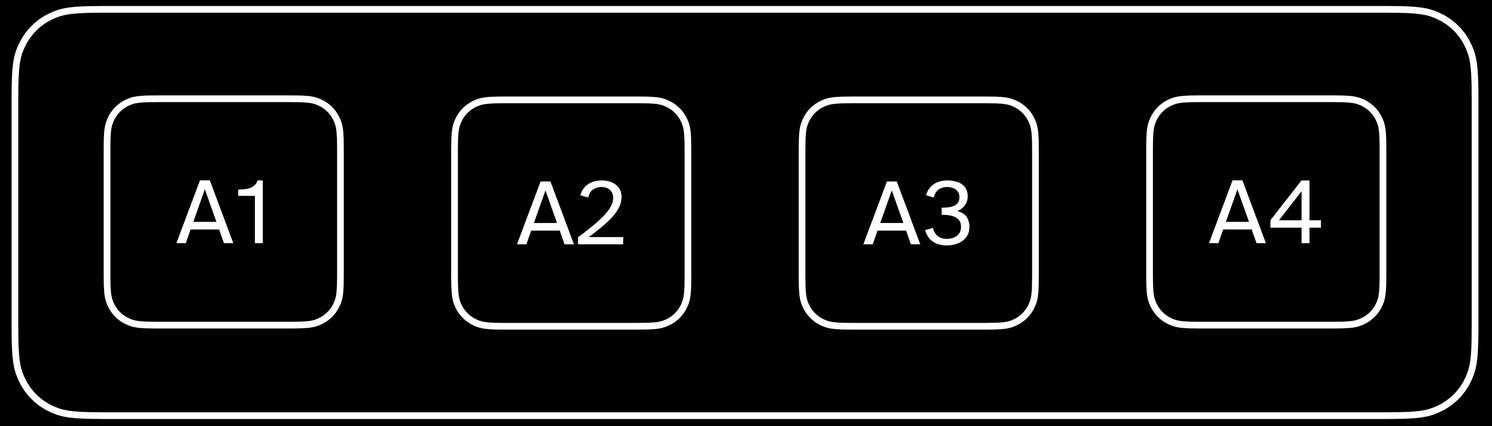
prime+probe



A single cache set

Load attacker-controlled A1, A2, A3, A4

prime+probe

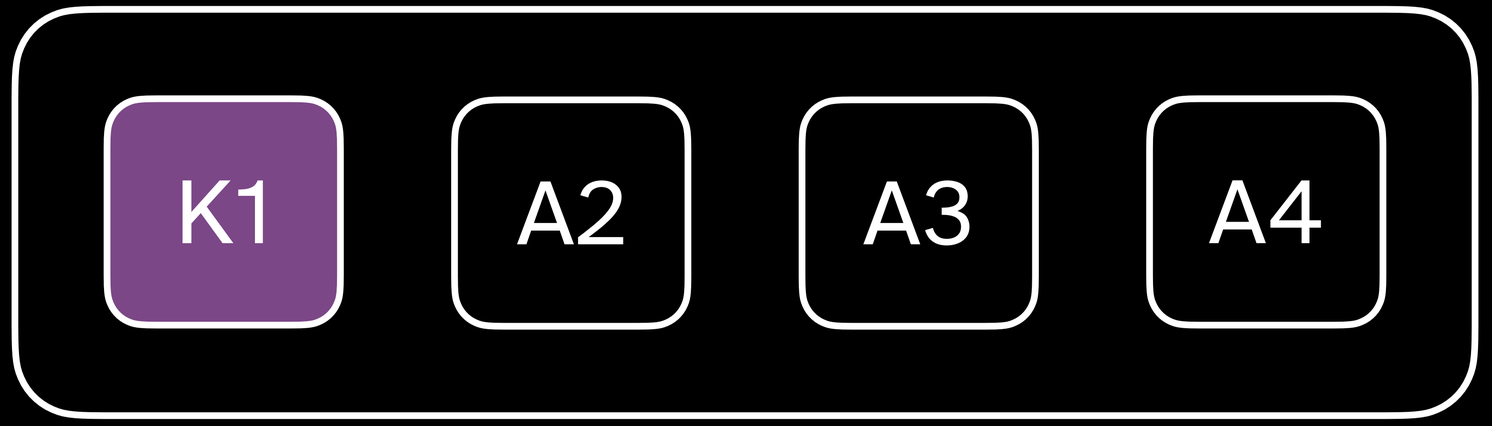


A single cache set

K1

Let the kernel load cache line K1

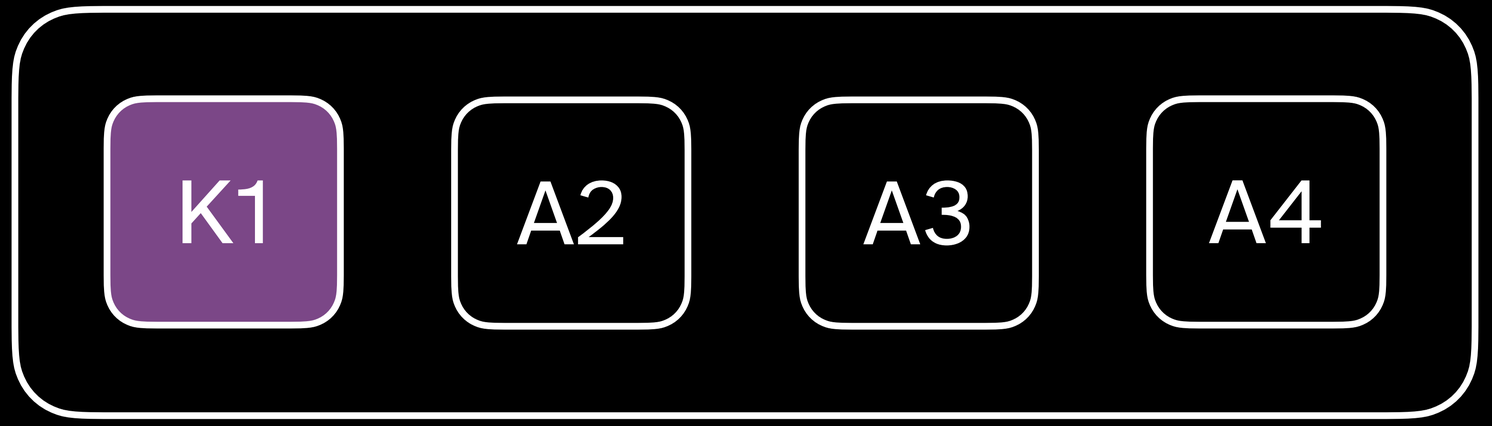
prime+probe



A single cache set

Let the kernel load cache line K1

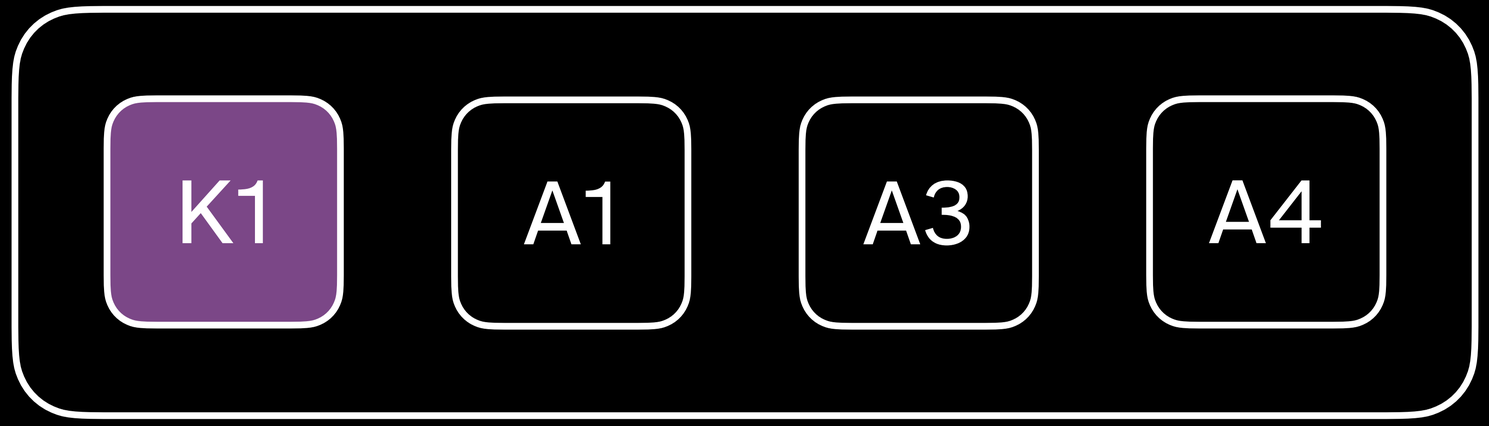
prime+probe



A single cache set

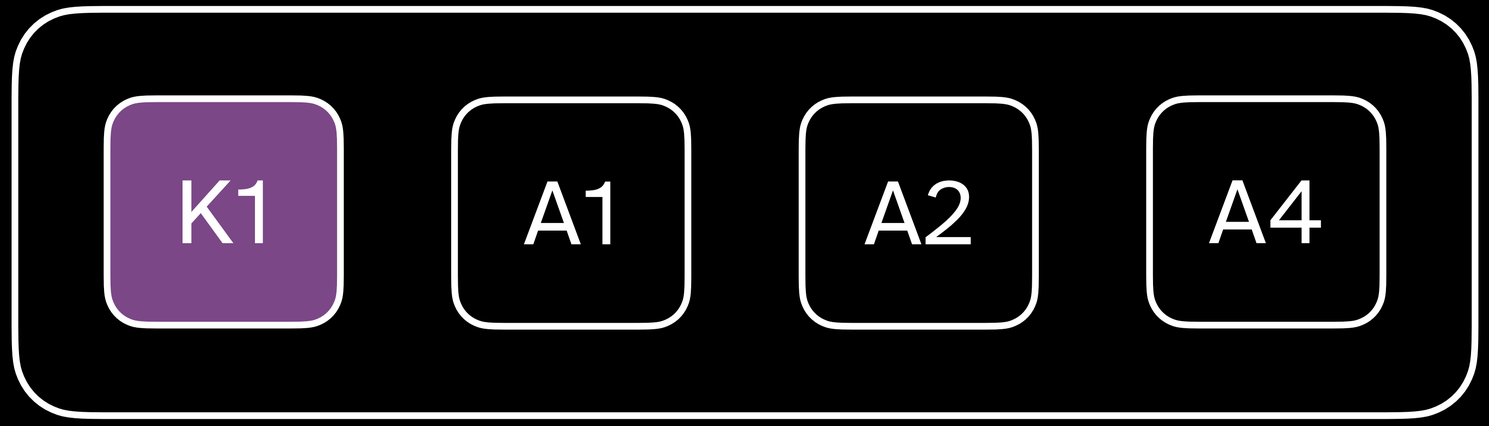
Now we reload
A1, A2, A3, A4
in the same order...

prime+probe



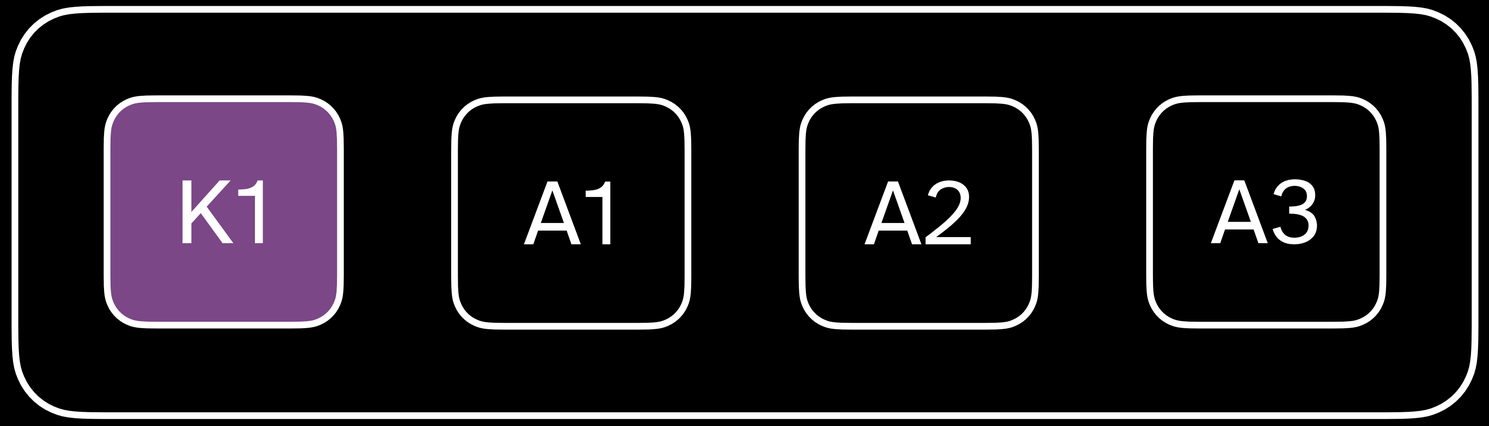
A single cache set

prime+probe



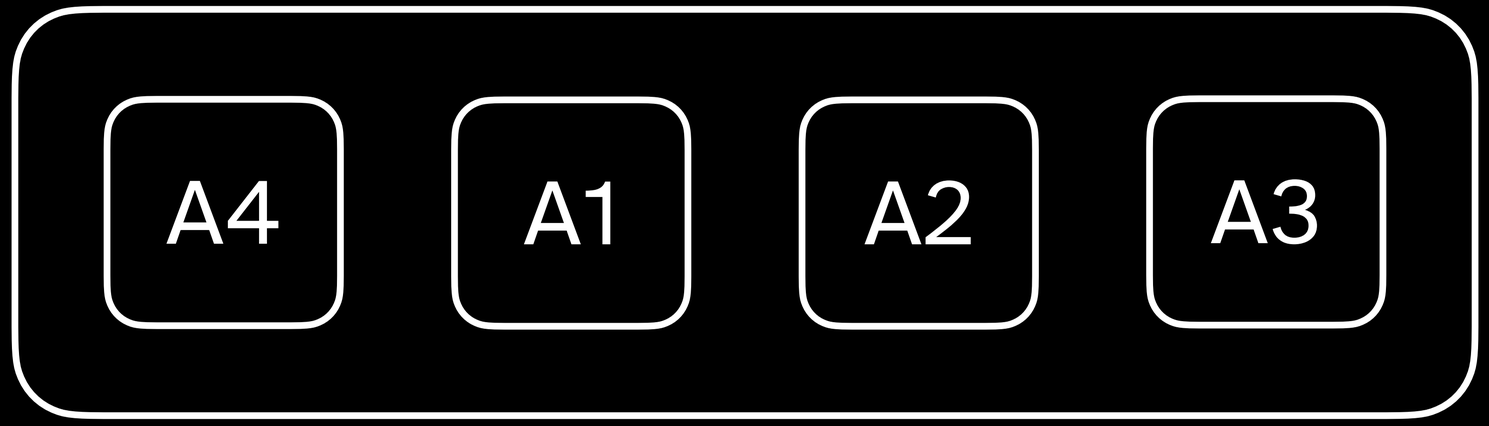
A single cache set

prime+probe



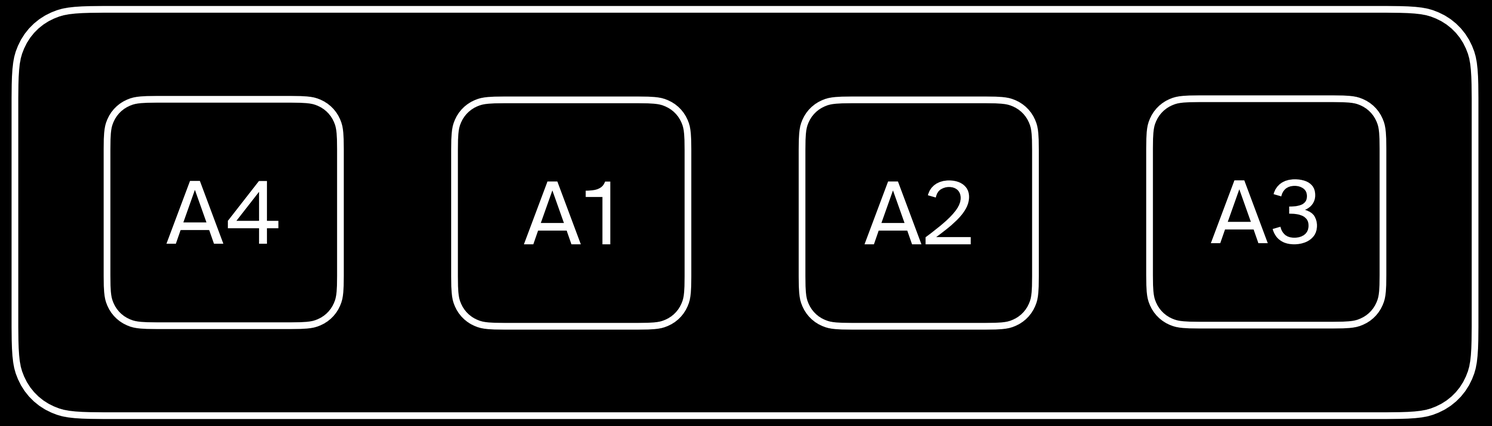
A single cache set

prime+probe



A single cache set

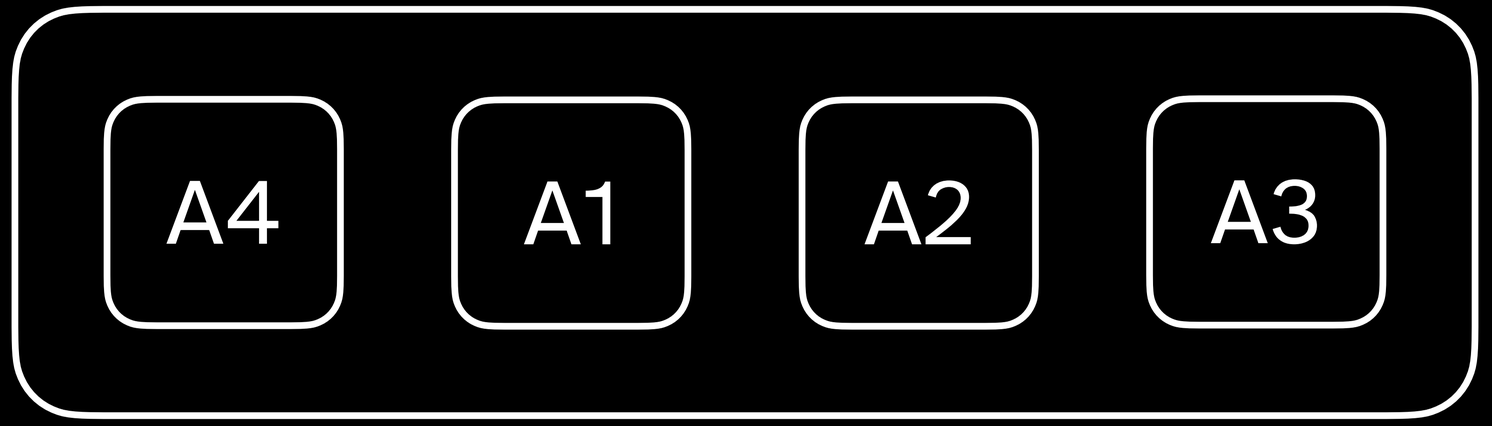
prime+probe



A single cache set

One evicted address turned into 4 misses!

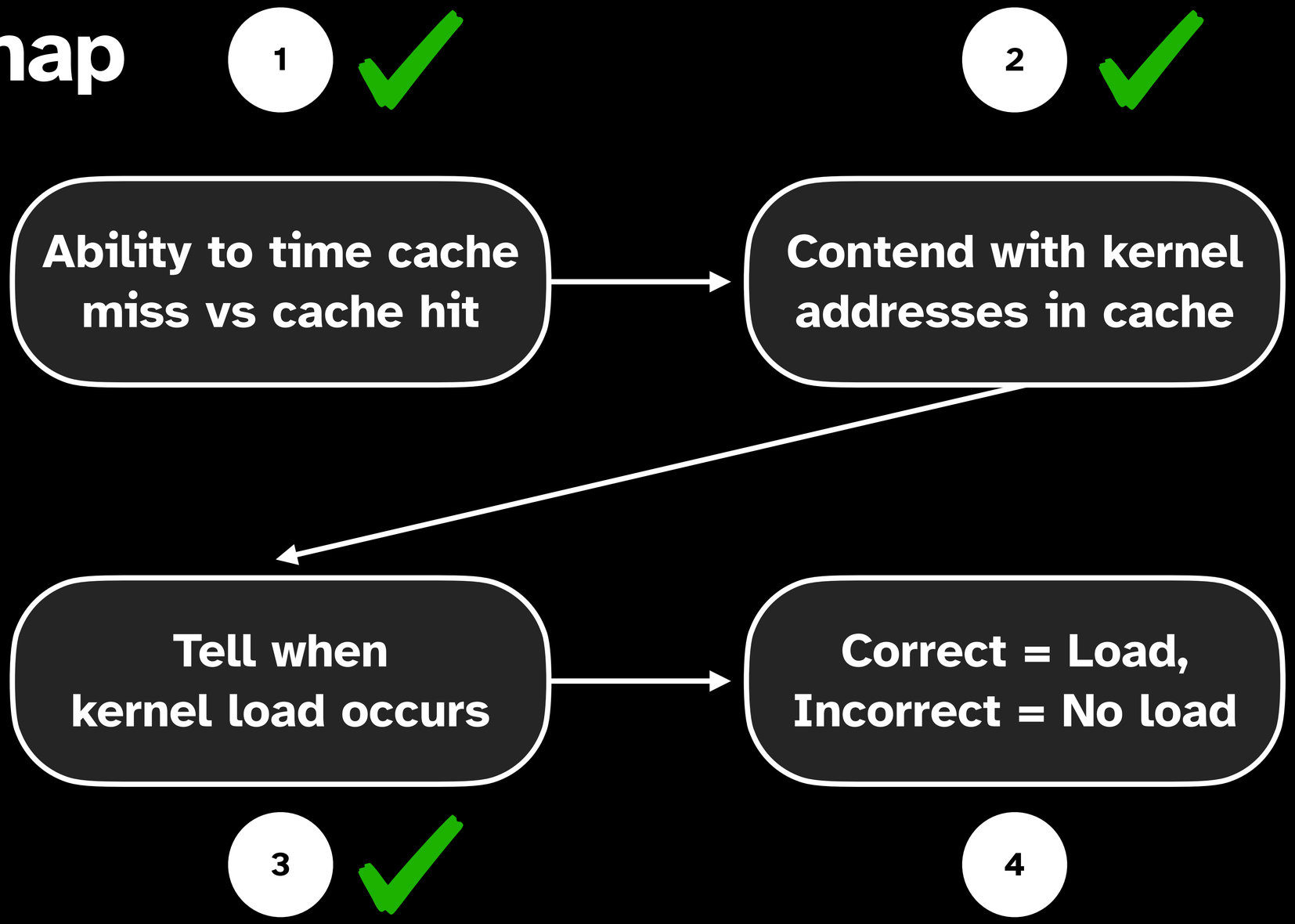
prime+probe



A single cache set

To avoid this-
reload A4, A3, A2, A1

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

3 Target Programs

Basic
Very simple AUT -> LDR

Advanced
C++ vtable function call

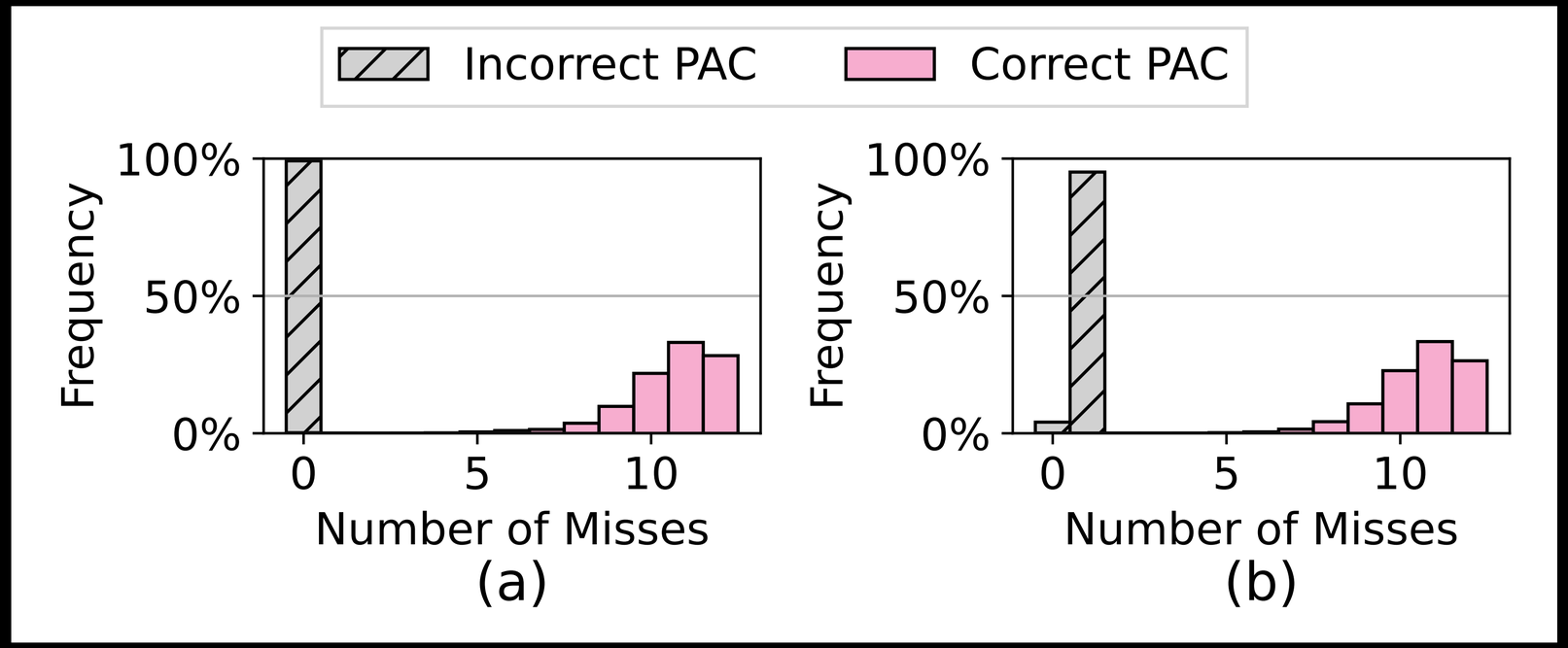
Ultra
A real system call in XNU

Basic Victim

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

Basic Victim VS PACMAN I

20,000 Runs



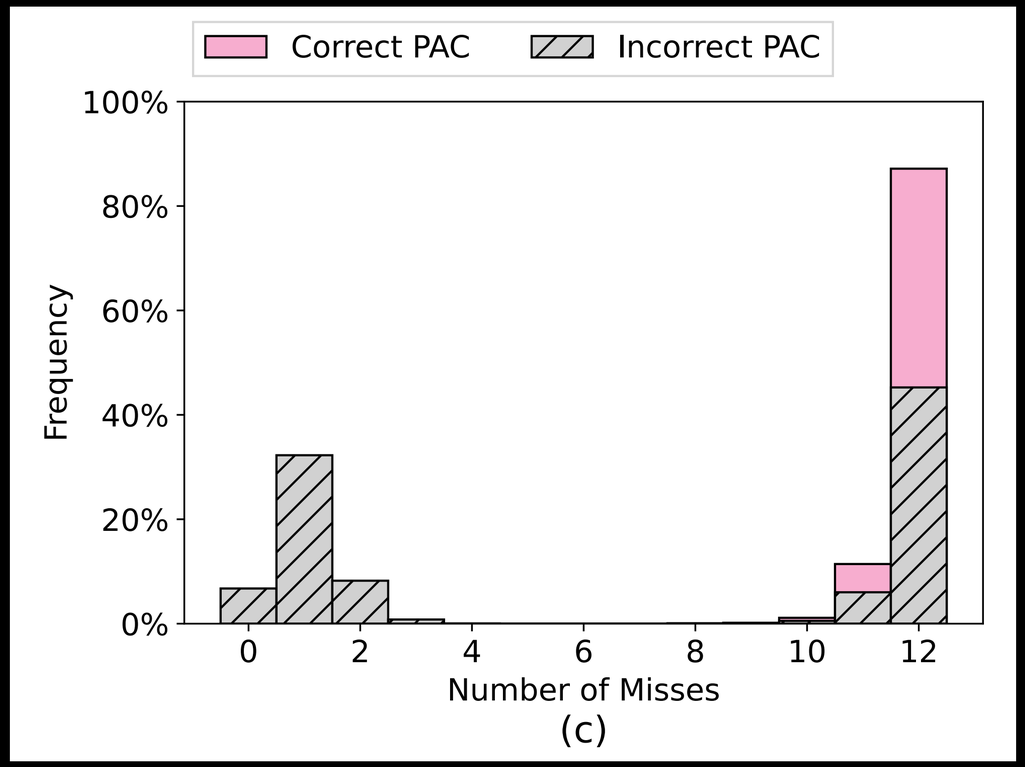
AUT -> LDR

AUT -> BLR

Basic Victim VS PACMAN I

20,000 Runs

Almost exactly 50% incorrect PACs perform a load anyways!



BLRAA

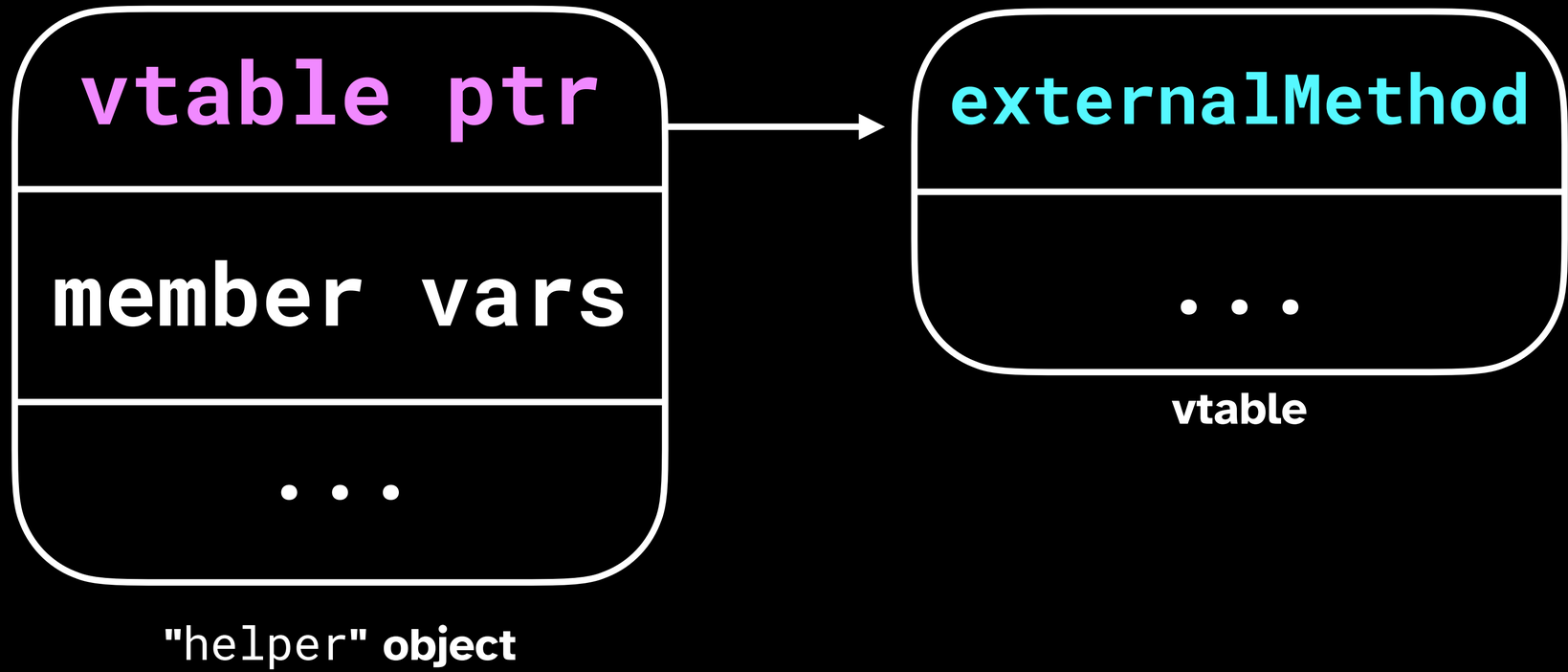
Advanced Victim

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

Advanced Victim

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

Data Attack



Inst Attack

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
```

Advanced Victim

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

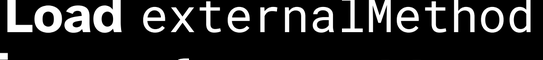
```
ldr x16, [helper]
mov x17, helper
movk x17, #0xd986, lsl #48
autda x16, x17
ldr x8, [x16]
mov x9, x16
mov x17, x9
movk x17, #0xa7d5, lsl #48
autia x8, x17
blr x8
```

Verify signed vtable data pointer

Advanced Victim

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

```
ldr x16, [helper]
mov x17, helper
movk x17, #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 `externalMethod` **ptr from vtable**

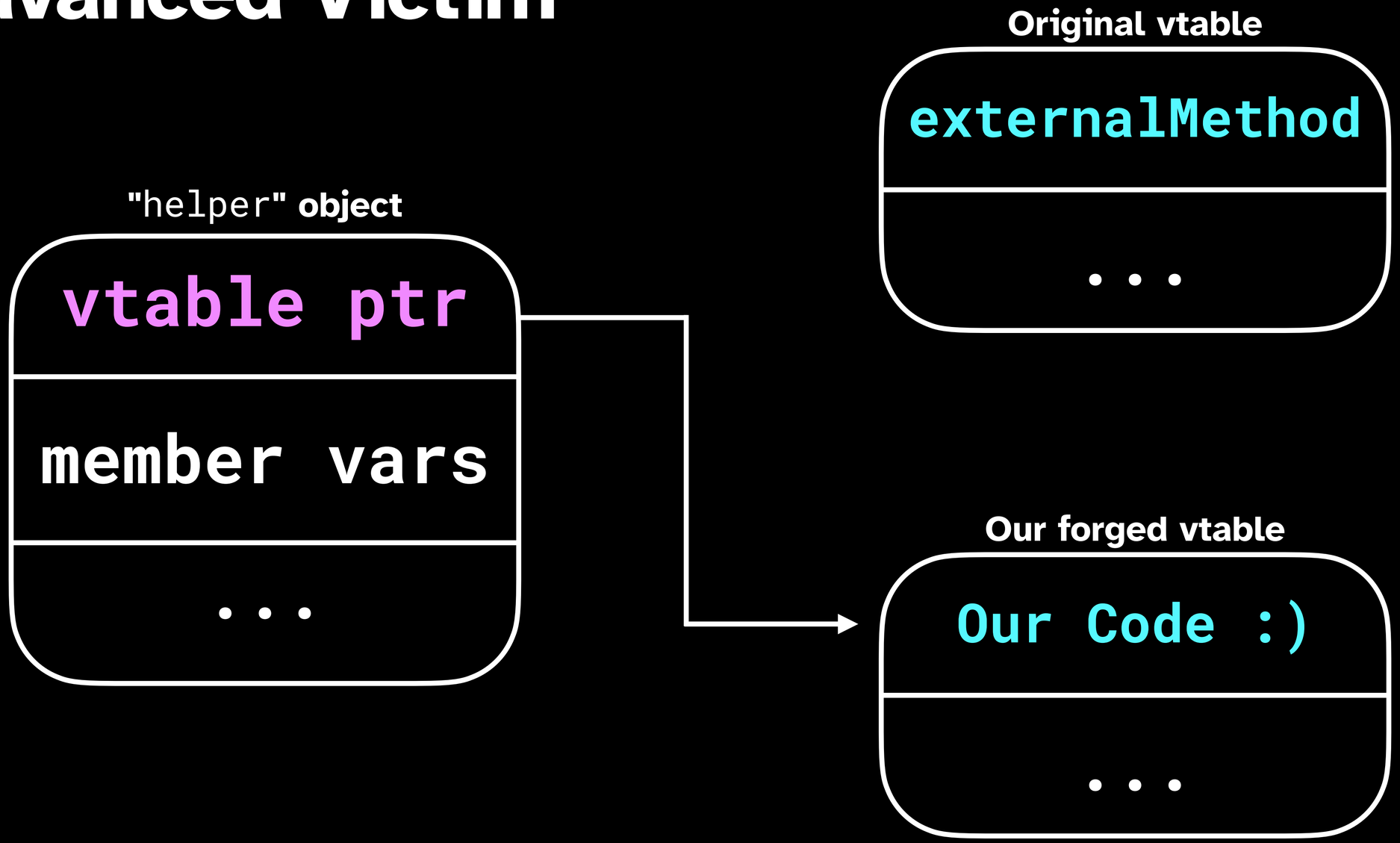
Advanced Victim

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

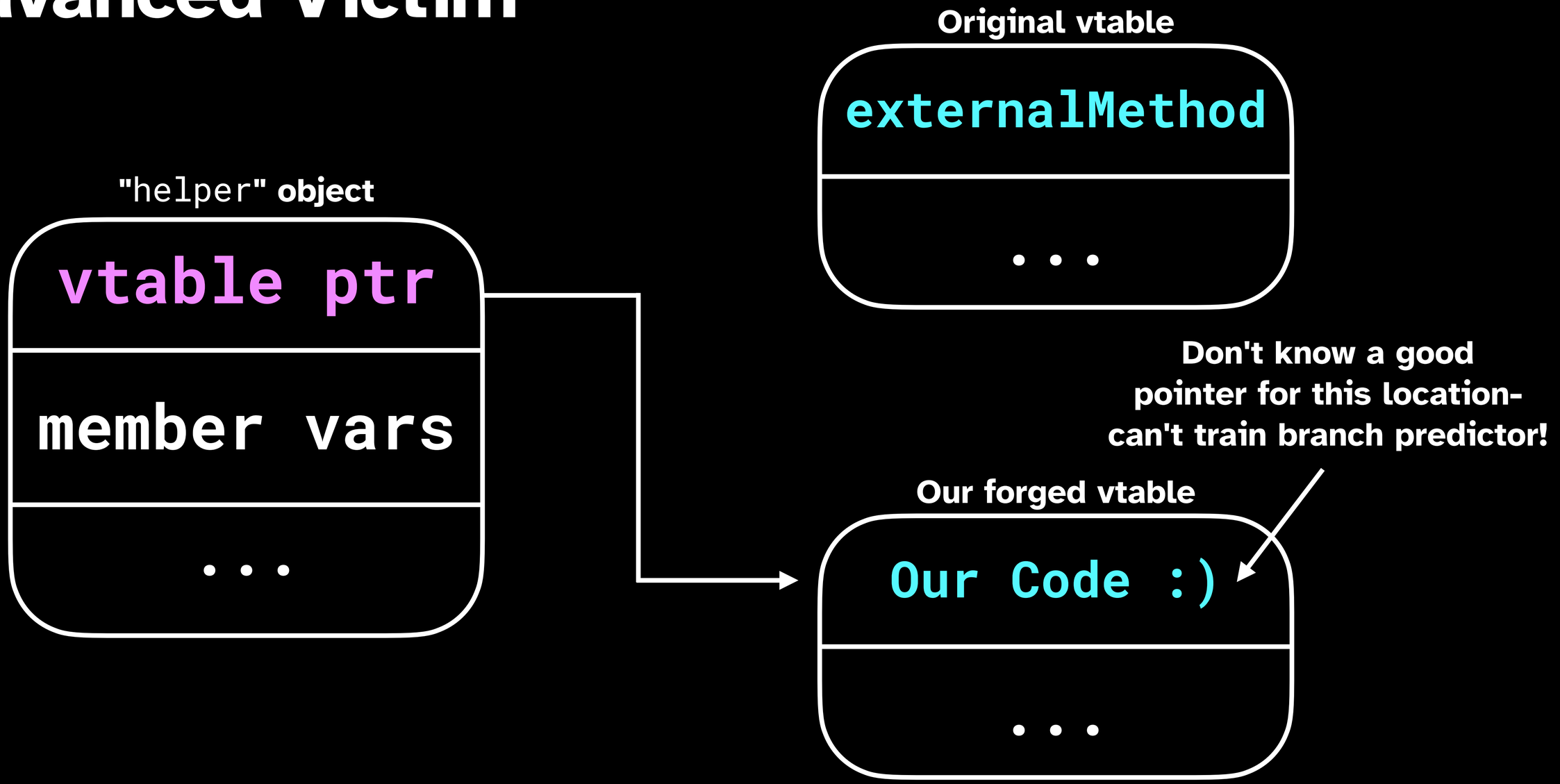
```
ldr x16, [helper]
mov x17, helper
movk x17, #0xd986, lsl #48
autda x16, x17
ldr x8, [x16]
mov x9, x16
mov x17, x9
movk x17, #0xa7d5, lsl #48
autia x8, x17
blr x8
```

Verify `externalMethod` **inst pointer**

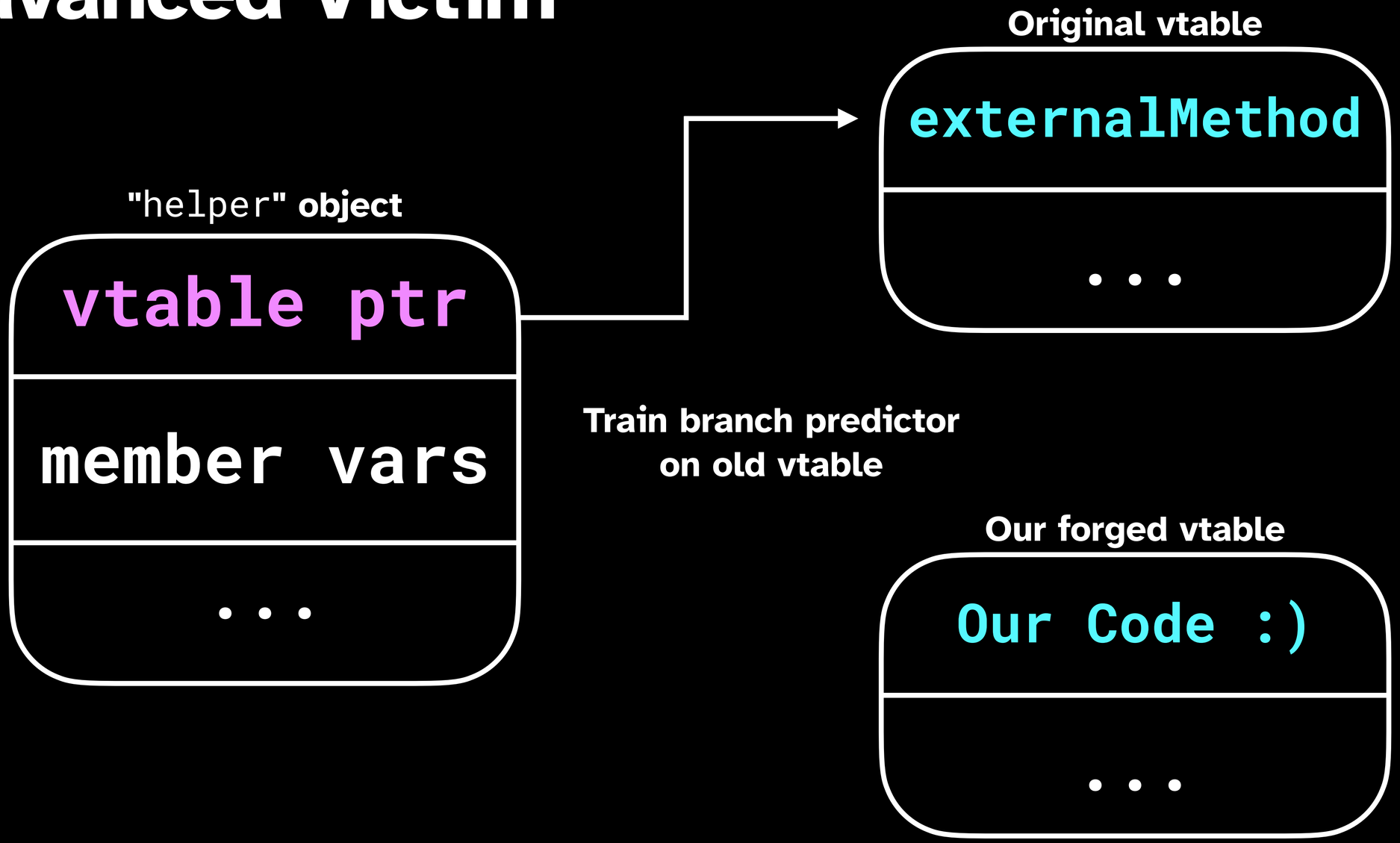
Advanced Victim



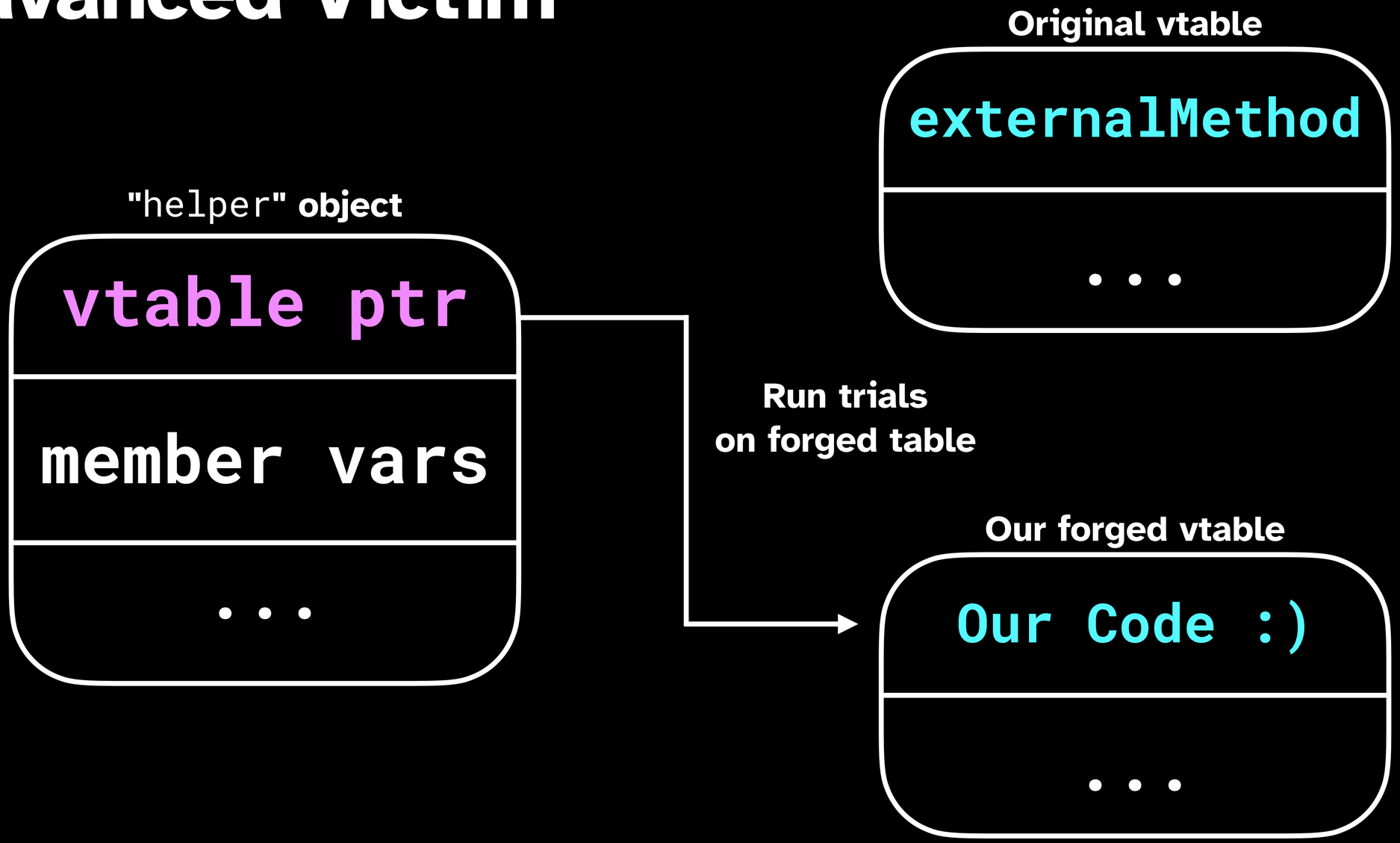
Advanced Victim



Advanced Victim

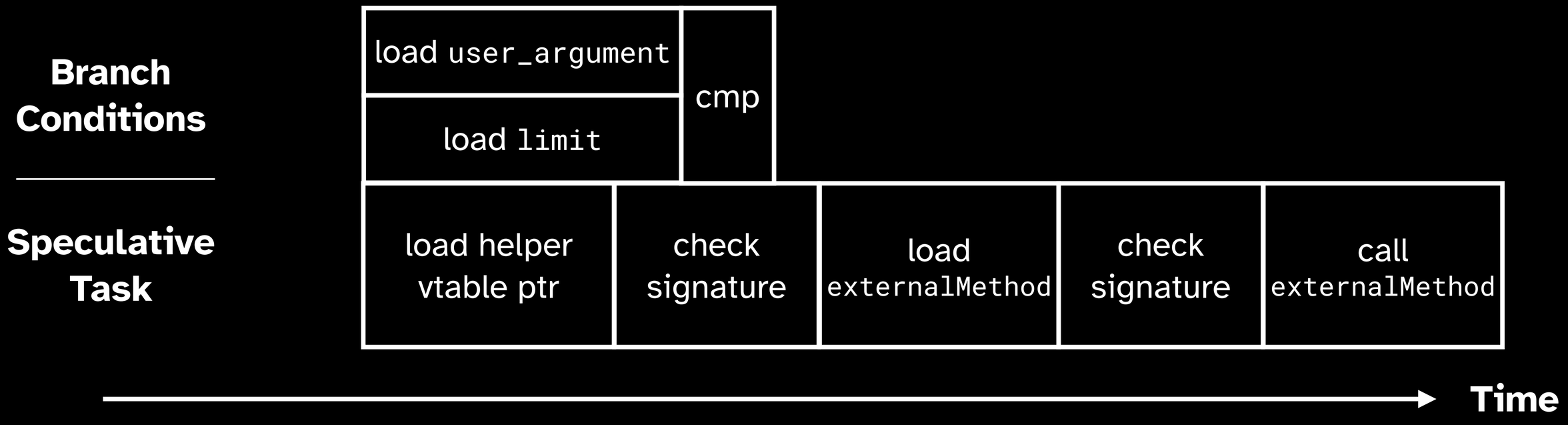


Advanced Victim



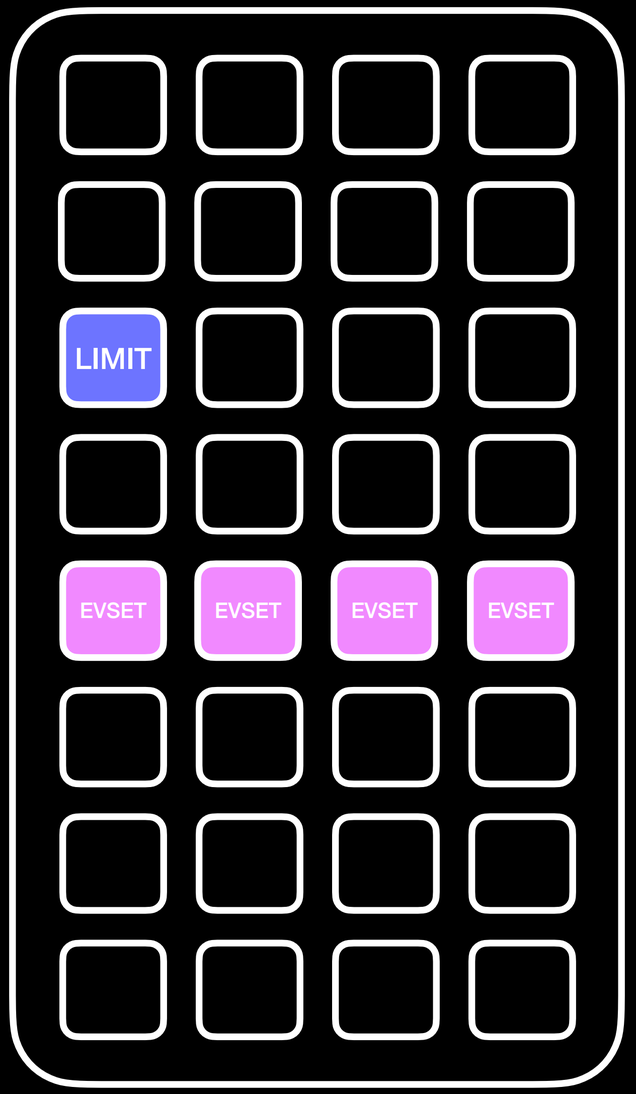
Lengthening the Window

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

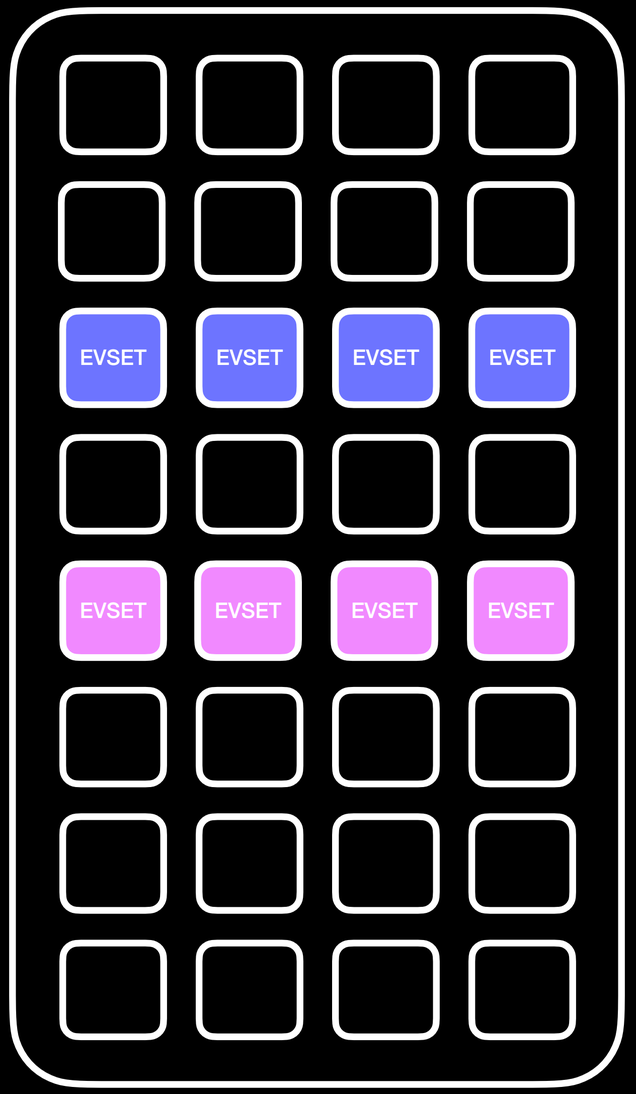


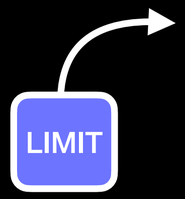
Lengthening the Window

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



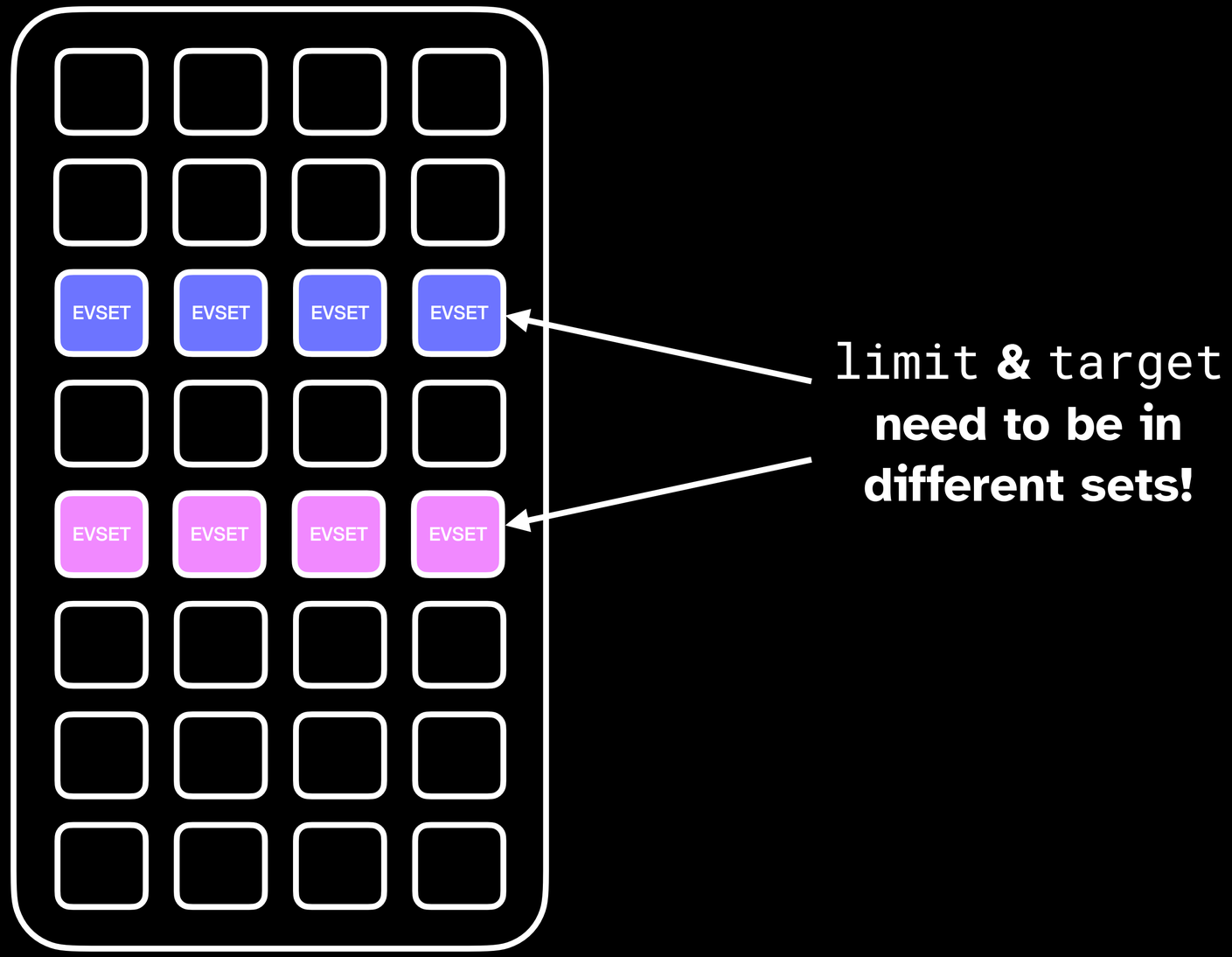
Lengthening the Window





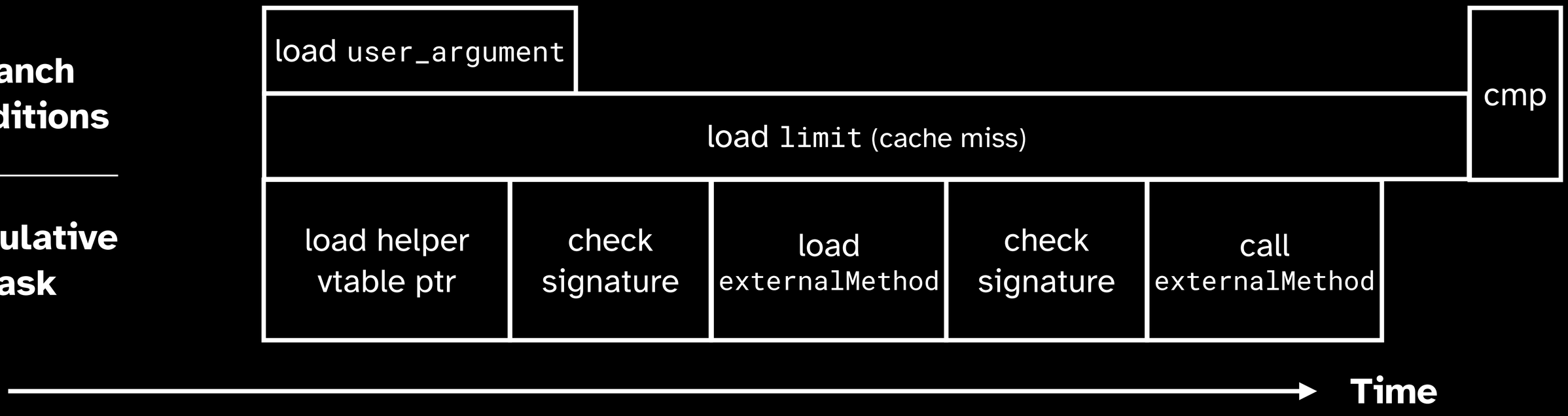
Kick limit out with a second eviction set!

Lengthening the Window



Lengthening the Window

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



Demo

Case Study Case Study
Case Study Case Study
Case Study Case Study
Case Study Case Study
Case Study Case Study
Case Study Case Study

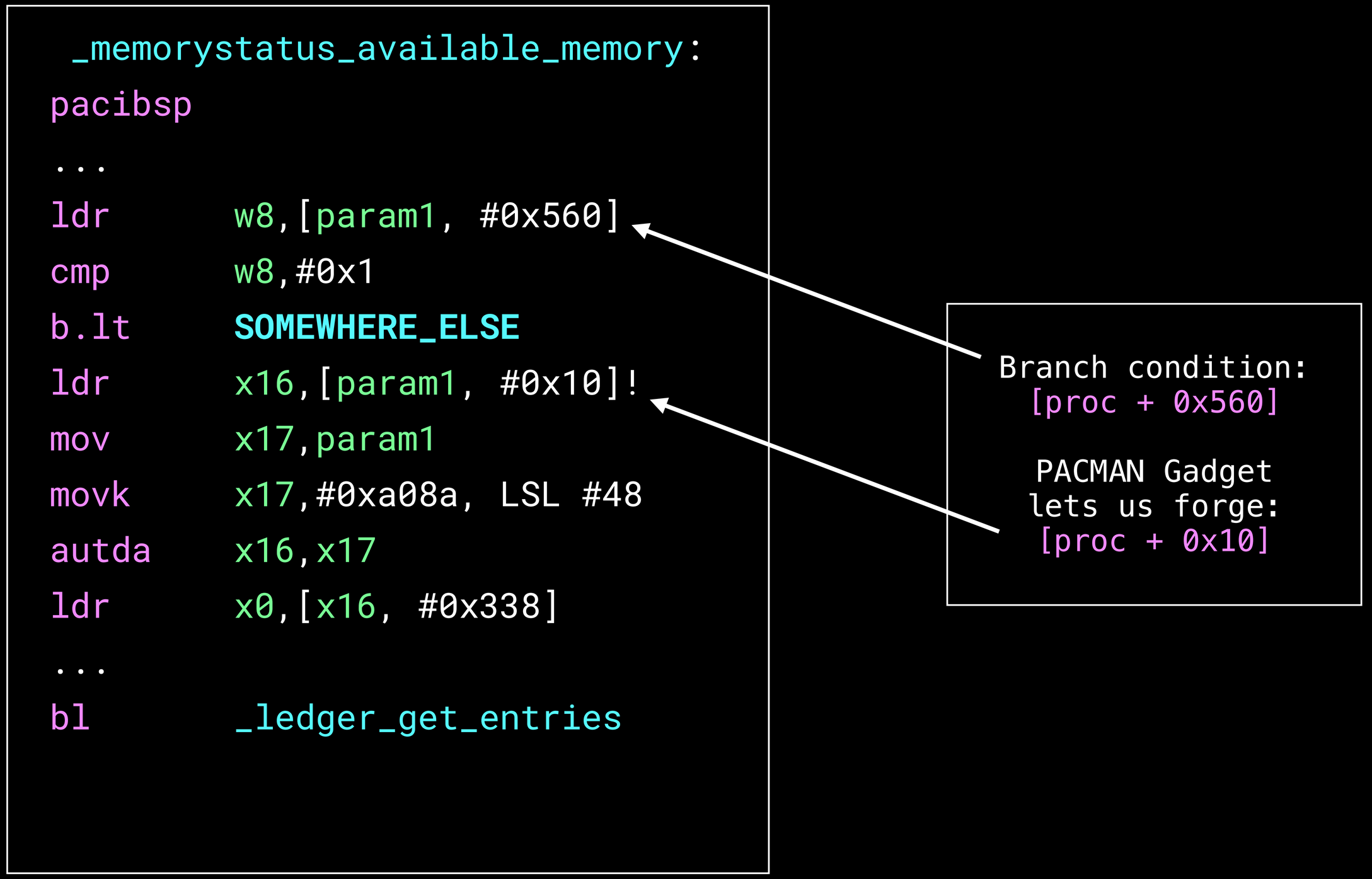
PACMAN Finder


```
I found 2 gadgets in _getattrlistbulk at [('d', fffffe00076080c0), ('d', fffffe00076080c4)]
I found 1 gadgets in _proc_rlimit_control at [('d', fffffe0007951ac4)]
I found 1 gadgets in _memorystatus_available_memory at [('d', fffffe000796dad8)]
I found 1 gadgets in _getdirentriesattr at [('d', fffffe000764925c)]
I found 2 gadgets in _fs_snapshot at [('d', fffffe000764e194), ('d', fffffe000764e198)]
I found 3 gadgets in _lseek at [('d', fffffe0007640fc8), ('d', fffffe0007641134), ('d', fffffe0007641138)]
I found 1 gadgets in _quotactl at [('d', fffffe000763a8c8)]
I found 1 gadgets in _sendfile at [('d', fffffe00079ce8a4)]
I found 1 gadgets in _process_policy at [('d', fffffe00079f25cc)]
```


```
I found 2 gadgets in _getattrlistbulk at [('d', fffffe00076080c0), ('d', fffffe00076080c4)]
I found 1 ga
I found 1 ga    _memorystatus_available_memory at [('d', fffffe000796dad8)]
I found 1 ga
I found 2 gadgets in _fs_snapshot at [('d', fffffe000764e194), ('d', fffffe000764e198)]
I found 3 gadgets in _lseek at [('d', fffffe0007640fc8), ('d', fffffe0007641134), ('d', fffffe0007641138)]
I found 1 gadgets in _quotactl at [('d', fffffe000763a8c8)]
I found 1 gadgets in _sendfile at [('d', fffffe00079ce8a4)]
I found 1 gadgets in _process_policy at [('d', fffffe00079f25cc)]
```

```
 _memorystatus_available_memory:
pacibsp
...
ldr       w8,[x0, #0x560]
cmp       w8,#0x1
b.lt      SOMEWHERE_ELSE
mov       x21,x0
ldr       x16,[x21, #0x10]!
mov       x17,x21
movk      x17,#0xa08a, LSL #48
autda     x16,x17
ldr       x0,[x16, #0x338]
...
bl        _ledger_get_entries
```

Ghidra


_memorystatus_available_memory:
pacibsp
...
ldr w8,[param1, #0x560]
cmp w8,#0x1
b.lt SOMEWHERE_ELSE
ldr x16,[param1, #0x10]!
mov x17,param1
movk x17,#0xa08a, LSL #48
autda x16,x17
ldr x0,[x16, #0x338]
...
bl _ledger_get_entries
Branch condition:
[proc + 0x560]
PACMAN Gadget
lets us forge:
[proc + 0x10]

Ghidra

bsd/kern/kern_memorystatus.c

```
uint64_t memorystatus_available_memory_internal(struct proc *p) {
    if (p->p_memstat_memlimit <= 0) {
        return 0;
    }
    const uint64_t footprint_in_bytes = get_task_phys_footprint(p->task);
    ...
}
```

bsd/sys/proc_internal.h

```
XNU_PTRAUTH_SIGNED_PTR("proc.task") task;
```

bsd/kern/kern_memorystatus.c

```
uint64_t memorystatus_available_memory_internal(struct proc *p) {
    if (p->p_memstat_memlimit <= 0) {
        return 0;
    }
    const uint64_t footprint_in_bytes = get_task_phys_footprint(p->task);
    ...
}
```

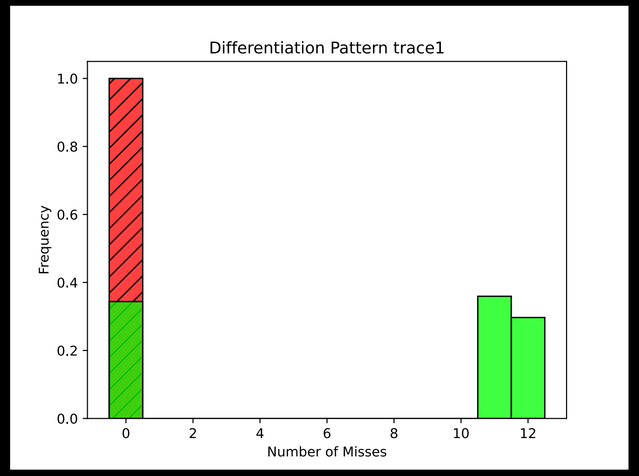
Branch condition:
p->p_memstat_memlimit

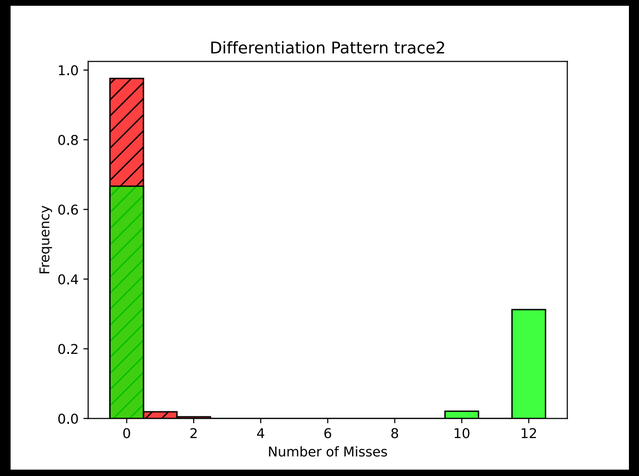
PACMAN Gadget
lets us forge:
p->task

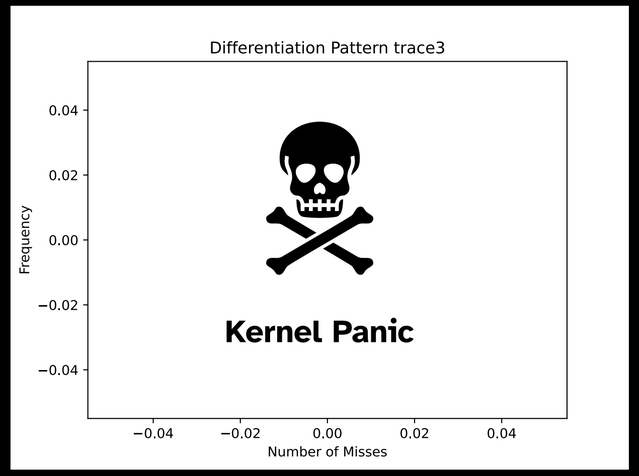
same page... :(

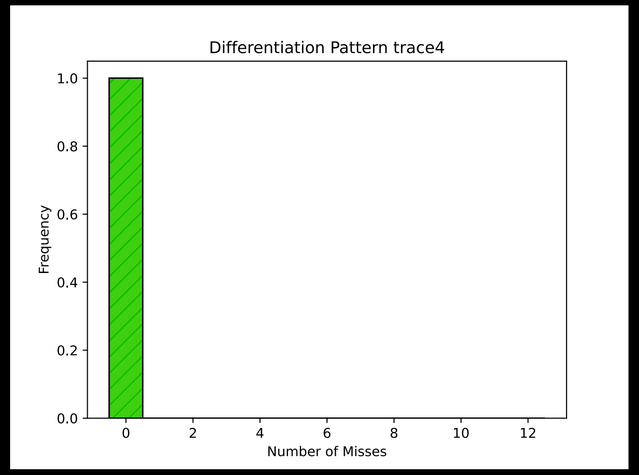
```
struct proc {
    LIST_ENTRY(proc) p_list;                /* List of all processes. */

    void *XNU_PTRAUTH_SIGNED_PTR("proc.task") task;       /* corresponding task (static)*/
    struct  proc *XNU_PTRAUTH_SIGNED_PTR("proc.p_pptr") p_pptr;   /* Pointer to parent process.(LL) */
    ...
}
```

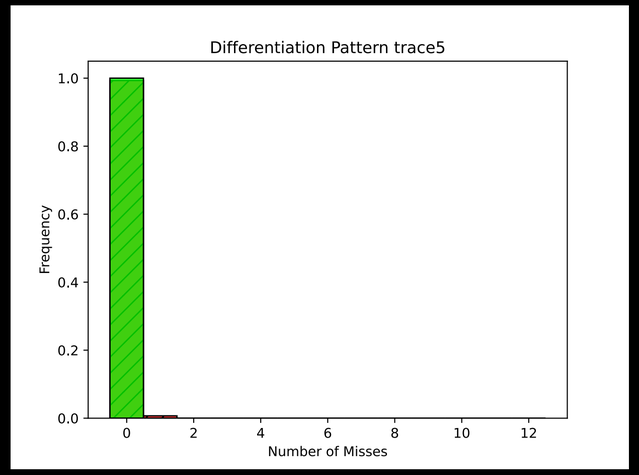
Poor choice of victim pointer-
Very commonly used field
in a frequently accessed page

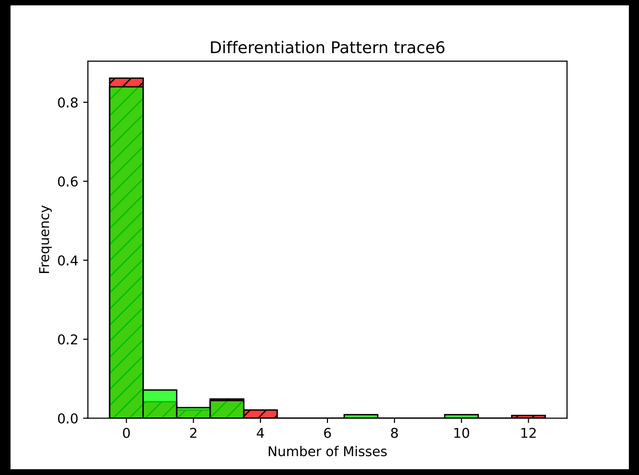

Differentiation Pattern trace1
Frequency
Number of Misses
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
12

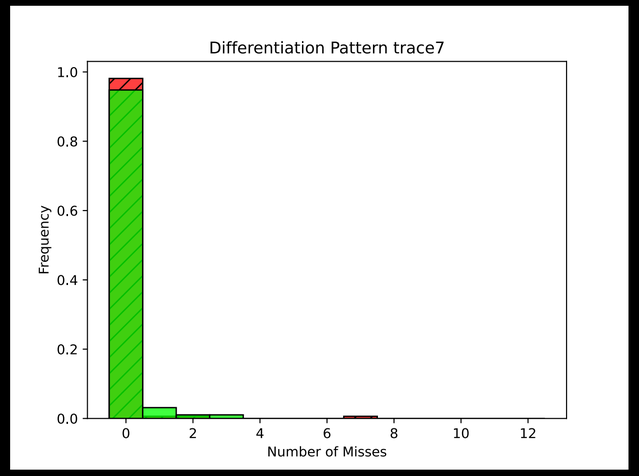

Differentiation Pattern trace2
Frequency
Number of Misses
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
12

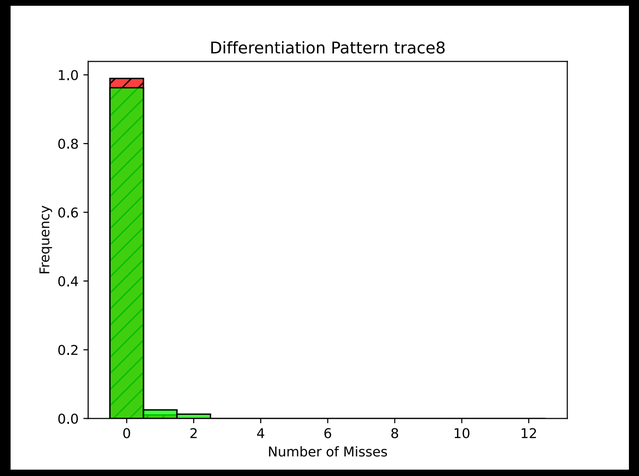

Differentiation Pattern trace3
Frequency
Number of Misses
0.04
0.02
0.00
−0.02
−0.04
Kernel Panic

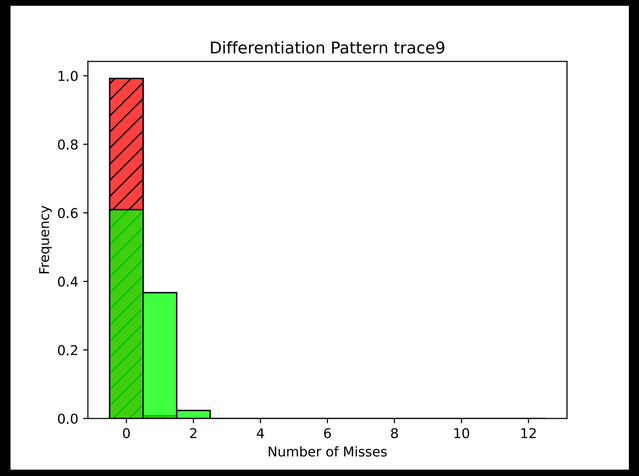

Differentiation Pattern trace4
Frequency
Number of Misses
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
12

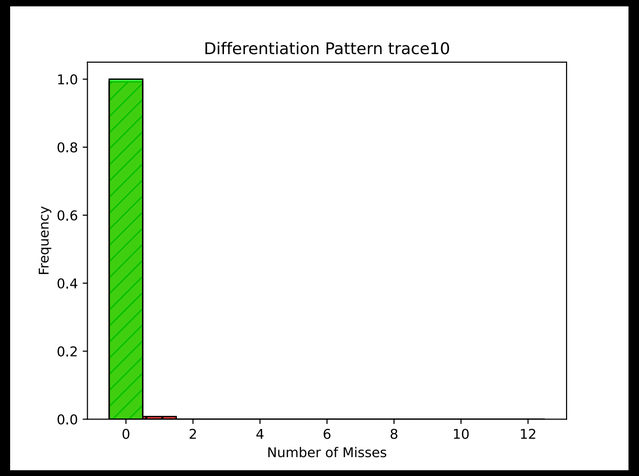

Differentiation Pattern trace5
Frequency
Number of Misses
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
12

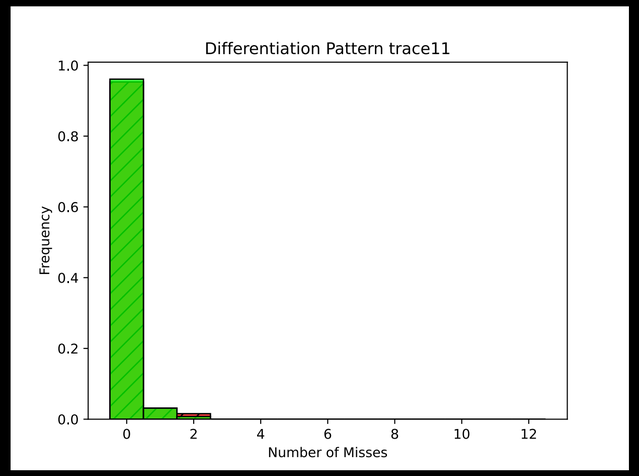

Differentiation Pattern trace6
Frequency
Number of Misses
0.0
0.2
0.4
0.6
0.8
0
2
4
6
8
10
12

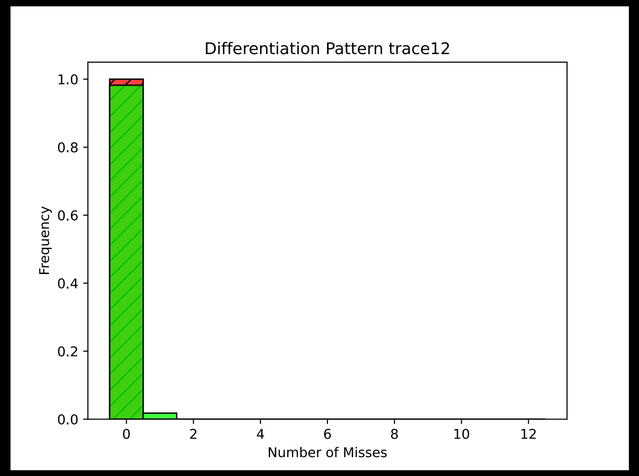

Differentiation Pattern trace7
Frequency
Number of Misses
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
12

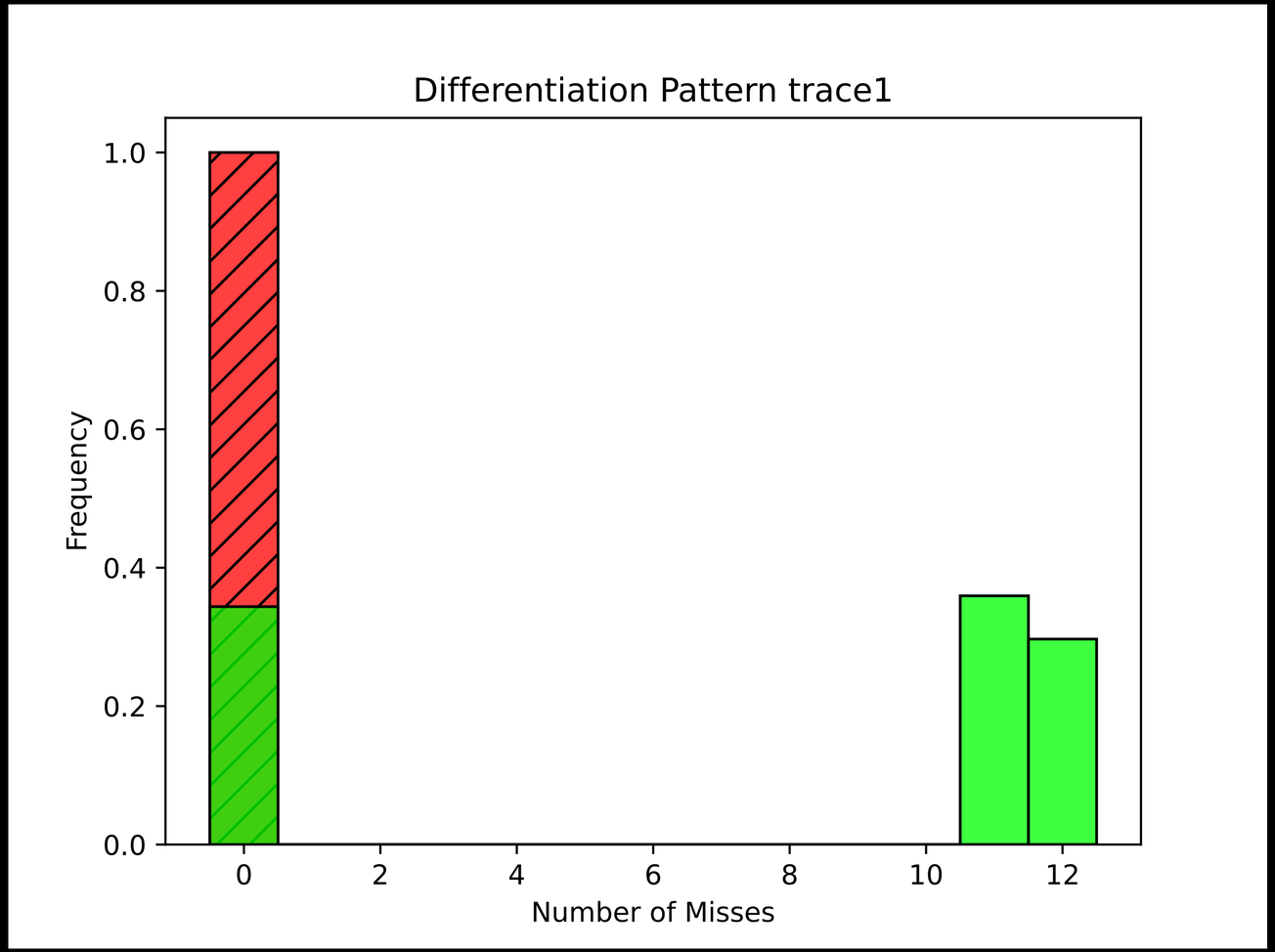

Differentiation Pattern trace8
Frequency
Number of Misses
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
12

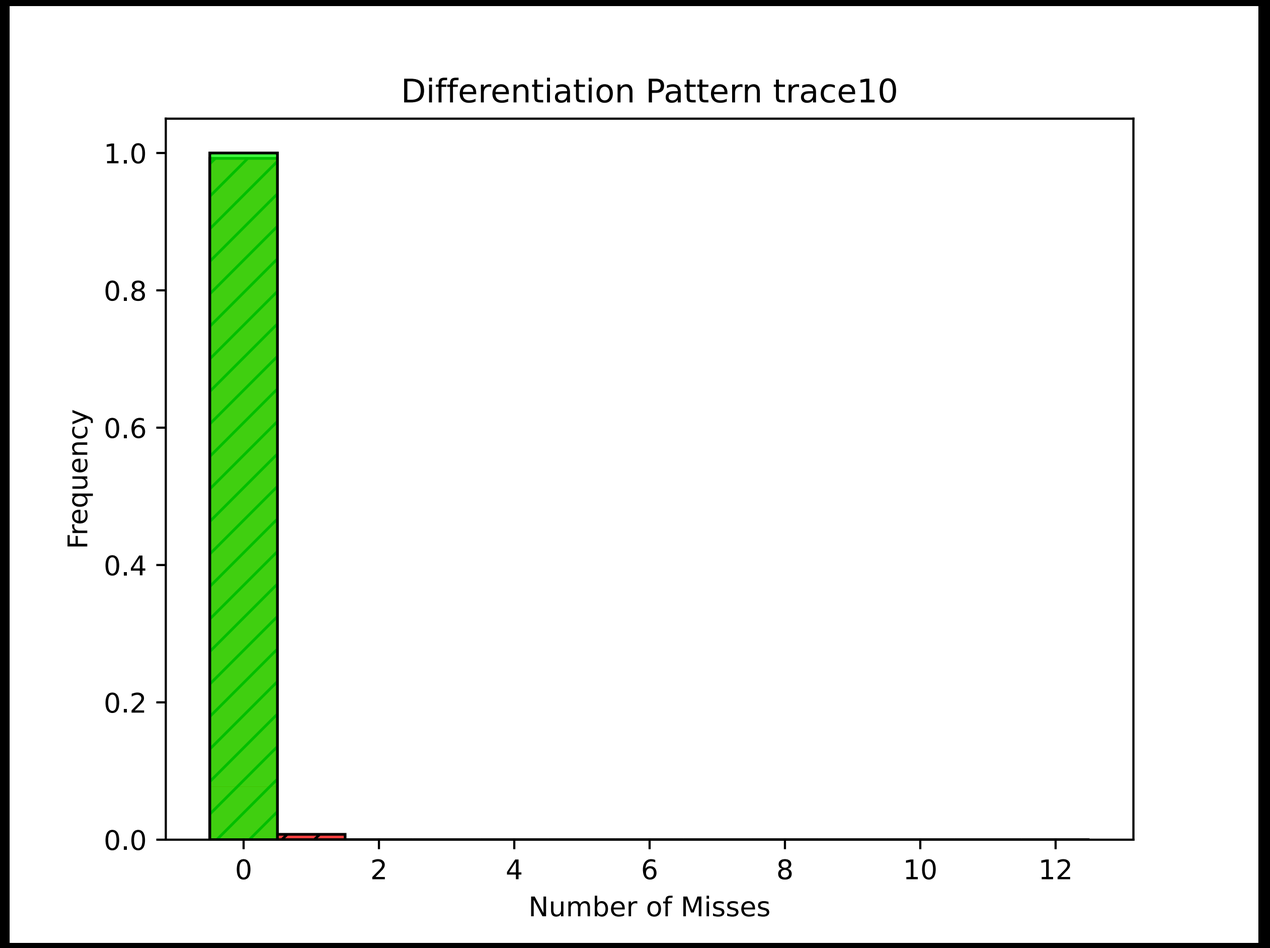

Differentiation Pattern trace9
Frequency
Number of Misses
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
12


Differentiation Pattern trace10
Frequency
Number of Misses
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
12

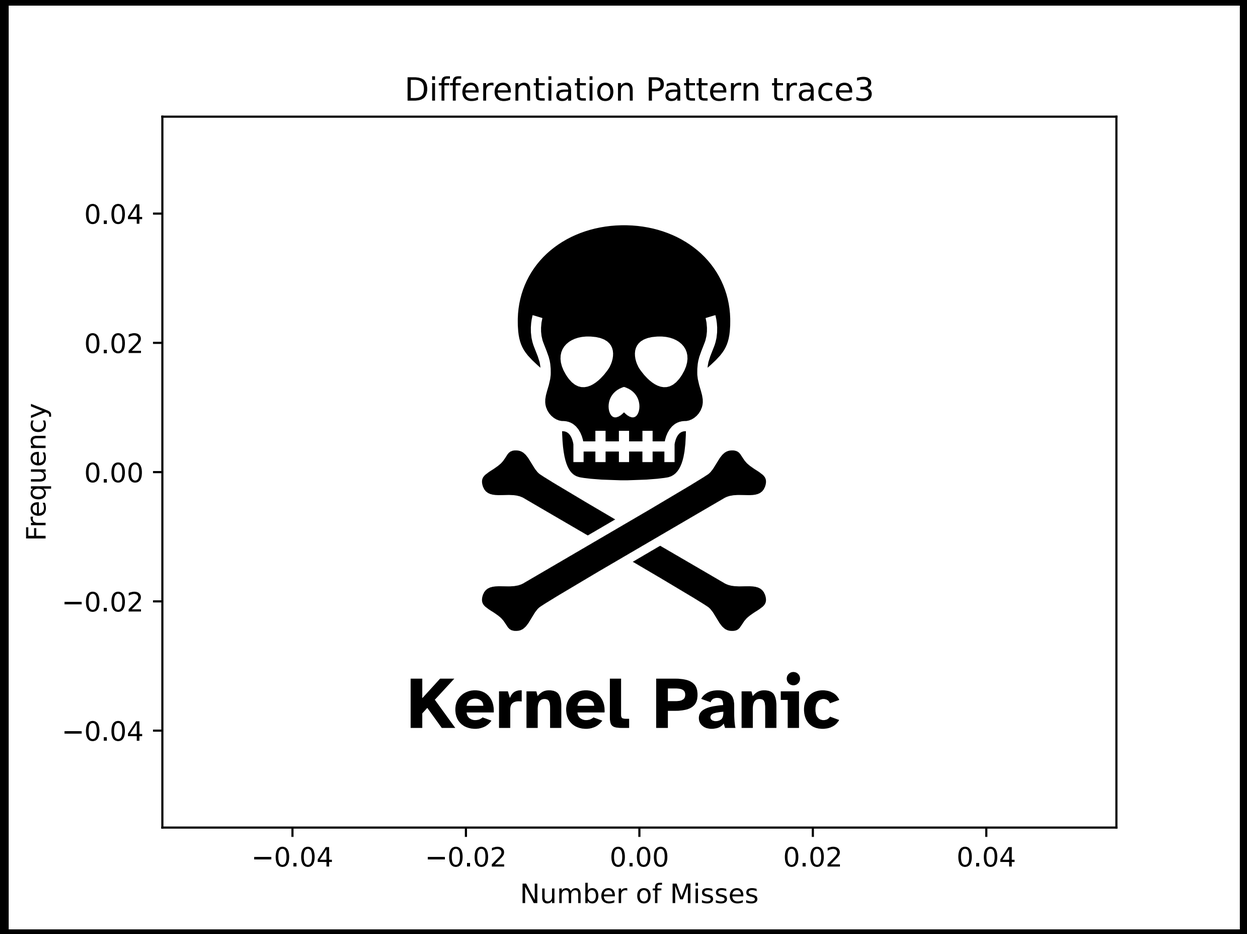

Differentiation Pattern trace11
Frequency
Number of Misses
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
12


Differentiation Pattern trace12
Frequency
Number of Misses
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
12


Differentiation Pattern trace1
Frequency
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
12
Number of Misses


Differentiation Pattern trace10
Frequency
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
12
Number of Misses

Beware of asynchronous accesses



Is this a flaw?

All Code on our GitHub!

https://github.com/jprx/DEFCON30-PACMAN

Questions? Reach out:

MIT CSAIL

compute. collaborate. create.

PACMANATTACK.COM