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.
$whoami

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

Weon Taek Na
1st Year PhD Student, MIT

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 analysis to uncover an unexpected vulnerability.

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 software attack, and vice versa. Our results reveal the importance of analyzing software and hardware vulnerabilities comprehensively.
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 A5X bits in the ID_AA64ISBAR1_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:
Not just M1...
Contributions

1. New way of thinking about compounding threat models.
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

Read/ Write Memory → Change Function Pointer → Arbitrary Code Execution
Memory Corruption

Read/Write Memory → Change Function Pointer → Arbitrary Code Execution

Pointer Authentication blocks changing pointers
Memory Corruption

- Read/Write Memory
- Change Function Pointer
- Arbitrary Code Execution

Pointer Authentication blocks changing pointers

Forge Signatures?
Just bruteforce it, right?
Key Insight: Avoid crashes using speculative execution!
BYOB

"Bring your own bug"
BYOBB "Bring your own bug"

- Read/Write Memory
- Change Function Pointer

Pointer Authentication blocks changing pointers

Arbitrary Code Execution
BYO'B "Bring your own bug"

- Read/Write Memory
- Change Function Pointer
- Arbitrary Code Execution

Attacker creates this

Pointer Authentication blocks changing pointers
BYOBB 
"Bring your own bug"

- Read/ Write Memory
- Change Function Pointer
- Arbitrary Code Execution

Attacker creates this

Pointer Authentication blocks changing pointers

PACMAN handles this.
4 New Tools + PoC
PacmanKit

IOKit driver for performing microarchitectural experiments.
Static analysis scripts for Ghidra to locate PACMAN Gadgets.
Patch your macOS kernel to enable high-resolution timers everywhere.
Run your own Rust experiments bare metal on M1.
Labs
There will be 5 laboratory exercises given throughout the semester related to lecture content.

<table>
<thead>
<tr>
<th>Lab</th>
<th>Released On</th>
<th>Due On</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Spectre</td>
<td>Wed, February 23</td>
<td>Wed, March 16</td>
</tr>
<tr>
<td>3. Website Fingerprinting</td>
<td>Wed, March 9</td>
<td>Wed, April 6</td>
</tr>
<tr>
<td>4. Rowhammer</td>
<td>Wed, March 30</td>
<td>Wed, April 20</td>
</tr>
<tr>
<td>5. ASLR Bypasses</td>
<td>Wed, April 13</td>
<td>Wed, May 4</td>
</tr>
</tbody>
</table>

Each lab is due at 11:59 PM.

Starter Code
All of the starter code is available at our GitHub.
There will be 5 laboratory exercises given throughout the semester related to lecture content.

<table>
<thead>
<tr>
<th>Lab</th>
<th>Released On</th>
<th>Due On</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Spectre</td>
<td>Wed, February 23</td>
<td>Wed, March 16</td>
</tr>
<tr>
<td>3. Meltdown</td>
<td>Wed, April 13</td>
<td>Wed, May 4</td>
</tr>
<tr>
<td>4. Rowhammer</td>
<td>Wed, March 30</td>
<td>Wed, April 20</td>
</tr>
<tr>
<td>5. ASLR Bypasses</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each lab is due at 11:59 PM.

**Starter Code**

All of the starter code is available at our GitHub.
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
Our design doesn't have 16 exabytes of RAM...

64-Bit Address
Pointer

Unused | 48-Bit Address
Let's put this to good use!
**Pointer Authentication**

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

Signed Pointer

- **PAC**: 16 Bits
- **Pointer**: 48 Bits

Verifies
Pointer Authentication

[PAC*] Insert a PAC

[AUT*] Check a PAC

Don't crash on AUT- crash on use
Pointer Authentication

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

Load signed pointer from memory

Address of object: 0x100000010

Memory

...
Pointer Authentication

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

Generate salt from pointer location & constant

```
salt = const | address
```

Address of object: 0x100000010
Pointer Authentication

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

Verify signature- store
invalid pointer in x16 if invalid

Address of object:
0x10000010

Memory

...
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: 0x100000010

Memory

... object ...

...
# Buffer Overflow

<table>
<thead>
<tr>
<th>Buffer[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer[1]</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Function Pointer</td>
</tr>
</tbody>
</table>
Buffer Overflow

Buffer[0]

Buffer[1]

...

Function Pointer

Buffer Overflow overwrites the function pointer!
Let's fix this bug with Pointer Authentication.
Buffer Overflow

<table>
<thead>
<tr>
<th>Buffer[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer[1]</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>PAC</td>
</tr>
</tbody>
</table>

PAC

- 16 Bits
- Verifies

Pointer

- 48 Bits
Buffer Overflow

Invalid PAC means we **crash**!

<table>
<thead>
<tr>
<th>Buffer[0]</th>
<th>Buffer[1]</th>
<th>...</th>
</tr>
</thead>
</table>

Buffer Overflow corrupts the PAC

<table>
<thead>
<tr>
<th>PAC</th>
<th>Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 Bits</td>
<td>48 Bits</td>
</tr>
</tbody>
</table>

Verifies
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

if?  A

Time
Speculative Execution

if (true)
  A
else
  B

In Order

if?
  A

Speculative

if?
Speculate A

Time
Speculative Execution

if (true)
  A
else
  B

In Order

if?
  A

Speculative

if?
  Speculate B

Time
Speculative Execution

In Order

if (true)
  A
else
  B

if?
  A

Speculative

if?
  Speculate B
  Undo B
  A

Microarchitectural side effects NOT undone

Time
How should we speculate on PAC values?
3 cases

- Ignore PACs (always load)
- Never load
- Treat them normally
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
Speculative PACs

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

In Order

if?  Return

Doesn't leak anything

Time
Speculative PACs

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

Speculative Correct Prediction

if?
  Return

Time
Speculative PACs

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

Speculative Correct Prediction

if?
    Return

Doesn't leak anything

Time
Speculative PACs

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

Speculative Misprediction

if?

Undo change to x

Return

Load signed ptr if correct, save in x

Time
Speculative PACs

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

Speculative Misprediction

if?
    Undo change to x
    Return

Load signed ptr if correct, save in x

Value still in the cache—this leaks info!

Time
Speculative PACs

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

Operating on PACs speculatively can leak PAC correctness without crashes!

Speculative Misprediction

if?
| Undo change to x | Return |

Load signed ptr if correct, save in x

Value still in the cache—this leaks info!

Time
Bird's Eye View

Write PAC guess into memory with existing software bug

Try guess speculatively ("PACMAN Gadget")

Speculative Load!

Correct

Incorrect

No Load
Memory is slow...

CPU

DRAM
100s of cycles
Memory is slow...

...so add some faster memory!

- **CPU**
- **Cache**
  - 10s of cycles
- **DRAM**
  - 100s of cycles
Cache
Cache

8 Sets

4 Ways
# Cache

<table>
<thead>
<tr>
<th>PAC</th>
<th>Address</th>
</tr>
</thead>
</table>

- **8 Sets**
- **4 Ways**
Cache

PAC | Address

Tag | Set | Offset

8 Sets

4 Ways
Cache

Which row (aka "set") do we map to?
Think like an attacker...
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!
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
PACMAN Gadget
Speculative check & use of a signed pointer.

```python
if (condition):
    check_pac(ptr)
    load(checked_ptr)
```
Data PACMAN Attack

if (condition):
    verified = AUT(pointer)
    load(verified)
We train the branch predictor to use a known signed pointer.

```python
if (condition):
    verified = AUT(pointer)
    load(verified)
```
Data PACMAN Attack

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

```python
if (condition):
    verified = AUT(good ptr)
    load(verified)
```
We train the branch predictor to use a known signed pointer.

if (condition):
    verified = AUT(good ptr)
    load(verified)
Data PACMAN Attack

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

\[
\text{if (condition):} \\
\quad \text{verified} = \text{AUT(good ptr)} \\
\quad \text{load(verified)}
\]
We train the branch predictor to use a known signed pointer.

```python
if (condition):
    verified = AUT(good ptr)
    load(verified)
```
Data PACMAN Attack

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

2  Reset the entire Cache
Data PACMAN Attack

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

Prime the Cache with an eviction set
Data PACMAN Attack

Prime the Cache with an eviction set

```
if (condition):
    verified = AUT(pointer)
    load(verified)
```
Data PACMAN Attack

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

Prime the Cache with an eviction set
Data PACMAN Attack

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

Call the gadget with the pointer and PAC to guess.
Data PACMAN Attack

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

Call the gadget with the pointer and PAC to guess.
Data PACMAN Attack

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

Call the gadget with the pointer and PAC to guess.
Data PACMAN Attack

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

Call the gadget with the pointer and PAC to guess.
Data PACMAN Attack

If (condition):
    verified = AUT(guess ptr)
    load(verified)

If the guess was correct...

Call the gadget with the pointer and PAC to guess.
Data PACMAN Attack

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

Call the gadget with the pointer and PAC to guess.
if (condition):
    verified = AUT(guess ptr)
    load(verified)

If the guess was incorrect...

Call the gadget with the pointer and PAC to guess.
Data PACMAN Attack

No Change

Cache

Branch Predictor

Taken

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

Speculative Data Abort!

Call the gadget with the pointer and PAC to guess.
Data PACMAN Attack

![Diagram of cache and branch predictor]

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

```python
if (condition):
    verified = AUT(guess_ptr)
    load(verified)
```
Ok, let's build it for real.
Given a correct and invalid PAC, can we tell which is which?

"Differentiation"

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

"Brute Force"

0x0000  ×
0x0001  ✓
0x0002  ×
...
Correct = Load, Incorrect = No load
Tell when kernel load occurs

Correct = Load, Incorrect = No load

Roadmap
Roadmap

Tell when kernel load occurs

Correct = Load, Incorrect = No load

Contend with kernel addresses in cache
Roadmap

1. Ability to time cache miss vs cache hit
2. Contend with kernel addresses in cache
3. Tell when kernel load occurs
4. Correct = Load, Incorrect = No load
Reverse Engineering M1
# Timer Sources on M1

<table>
<thead>
<tr>
<th>Source</th>
<th>Speed</th>
<th>Available to users?</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Counter</td>
<td>Slow (24 MHz)</td>
<td>Yes</td>
</tr>
<tr>
<td>ARM PMU</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Apple Custom Cycle Counter</td>
<td>Very Fast (Cycle Accurate)</td>
<td>No</td>
</tr>
<tr>
<td>Multi-threaded Counter</td>
<td>&quot;Good Enough&quot;</td>
<td>Yes</td>
</tr>
</tbody>
</table>
# Timer Sources on M1

<table>
<thead>
<tr>
<th>Timer Source</th>
<th>Speed</th>
<th>Available to users?</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Counter</td>
<td>Slow (24 MHz)</td>
<td>Yes</td>
</tr>
<tr>
<td>ARM PMU</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Apple Custom Cycle Counter</td>
<td>Very Fast (Cycle Accurate)</td>
<td>No</td>
</tr>
<tr>
<td>Multi-threaded Counter</td>
<td>&quot;Good Enough&quot;</td>
<td>Yes</td>
</tr>
</tbody>
</table>
## Timer Sources on M1

<table>
<thead>
<tr>
<th>Source</th>
<th>Speed</th>
<th>Available to users?</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Counter</td>
<td>Slow (24 MHz)</td>
<td>Yes</td>
</tr>
<tr>
<td>ARM PMU</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Apple Custom Cycle Counter</td>
<td>Very Fast (Cycle Accurate)</td>
<td>No</td>
</tr>
<tr>
<td>Multi-threaded Counter</td>
<td>&quot;Good Enough&quot;</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Reverse Engineer with this

Attack with this
Apple Performance Counters
osfmk/arm64/monotonic_arm64.c

/*
 * 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(0, A); X(1, A); X(2, A); X(3, A); X(4, A); X(5, A); \
 X(6, A); X(7, A)
Apple Performance Counters
osfmk/arm64/monotonic_arm64.c

/*
 * 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(0, A); X(1, A); X(2, A); X(3, A); X(4, A); X(5, A); \
     X(6, A); X(7, A)

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

osfmk/arm64/monotonic_arm64.c

/*
 * 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(0, A); X(1, A); X(2, A); X(3, A); X(4, A); X(5, A); X(6, A); X(7, A)

/*
 * 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)
Apple Performance Counters  
osfmk/arm64/monotonic_arm64.c

#include <stdint.h>

#define PMCR0_USEREN_EN (UINT64_C(1) << 30)

PMCR0_USEREN_EN (bit 30) is not set!

#define PMCR0_INIT (PMCR0_INTGEN_INIT | PMCR0_PMI_INIT | PMCR0_PCC_INIT)

/* user mode access to configuration registers */

/* 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(6, A); X(7, A)
Masked kpc sysctls

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

/* values */
SYSCALL_PROC(kpc, OID_AUTO, classes,
    CTLTYPE_INT | CTLFLAG_RW | CTLFLAG_ANYBODY | CTLFLAG_MASKED | CTLFLAG_LOCKED,
    (void*)REQ_CLASSES,
    sizeof(int), kpc_sysctl, "I", "Available classes");

SYSCALL_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");

SYSCALL_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");

SYSCALL_PROC(kpc, OID_AUTO, pmu_version,
    CTLTYPE_INT | CTLFLAG_RW | CTLFLAG_ANYBODY | CTLFLAG_MASKED | CTLFLAG_LOCKED,
    (void*)REQ_PMU_VERSION,
    sizeof(int), kpc_sysctl, "I", "PMU version for hardware");

/* faux values */
SYSCALL_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");

SYSCALL_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");

SYSCALL_PROC(kpc, OID_AUTO, sw_inc,
    CTLTYPE_INT | CTLFLAG_RW | CTLFLAG_ANYBODY | CTLFLAG_MASKED | CTLFLAG_LOCKED,
    (void*)REQ_SW_INC,
    sizeof(int), kpc_sysctl, "S", "Sw inc");
```
Masked kpc sysctls

Used by private kperf.framework

```c
int kpc_get_cpu_counters(int param_1, uint param_2, undefined*param_3, void*param_4)
{
    size_t iVar1;
    int iVar2;
    int iVar3;
    int iVar4;
    undefined* iVar5;
    ulong local_60;
    int local_58;
    int local_54;
    size_t local_50;
    uint local_44;

    local_50 = (ulong)param_2;
    local_58 = 1;
    if (param_1 == 0) {
        iVar3 = 0;
    } else {
        local_60 = local_50 | 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_stub::sysctlname("kpc.counter_count", &local_54, &local_50, &local_44, 4);
    iVar2 = 0;
    if (iVar4 < iVar2) {
        iVar2 = local_54;
    }
}
```
Masked kpc sysctls

```c
// 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_ACCESS_DENIED) {
      ktrace_unlock();
      return ret;
    }
    break;
}
```

- Require root
- Extra latency from trip to kernel
/dev/perfmon_uncore

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

bsd/dev/dev_perfmon.c
static int
permon_dev_ioctl(dev_t dev, unsigned long cmd, char *arg,
    int __unused *flag, proc_t __unused p)
{
    struct permon_device *device = permon_dev_get_device(dev);
    struct permon_source *source = permon_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 PERMON_CTL_GET_LAYOUT: {
        struct permon_layout *layout = (void *)arg;
        *layout = source->ps_layout;
        ret = 0;
        break;
    }
    case PERMON_CTL_LIST_REGS: {
        user_addr_t uptr = *(user_addr_t *)arg;
        size_t name_size = reg_count * sizeof(*source->ps_register_names[0]);
        ret = copyout(source->ps_register_names, uptr, names_size);
        break;
    }
    case PERMON_CTL_SAMPLE_REGS: {
        user_addr_t uptr = *(user_addr_t *)arg;
        uint64_t *sample_buf = device->pmdv_copyout_buf;
        size_t sample_size = reg_count * unit_count * sizeof(*sample_buf);
        permon_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
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?
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"

```rust
let t1 : u64;
let t2 : u64;
asm!
    "dsb sy",
    "isb",
    "ldr {t1}, [{cntr}]",
    "isb",
    "ldr {val_out}, [{addr}]",
    "isb",
    "ldr {t2}, [{cntr}]",
    "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;
```

Counting Thread

Timing a data access
Roadmap

1. Tell when kernel load occurs
   - Correct = Load, Incorrect = No load

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

PacmanKit
TLB + Cache Interactions

Stride 0
- Address x

Stride 1
- x + stride

Stride 2
- x + 2 * stride

addr[i] = x + (i*stride)
TLB + Cache Results

Latency from dTLB/dCache conflicts

Latency (Cycles)

Eviction Set Size (N)
TLB + Cache Results

Latency from dTLB/dCache conflicts

Eviction Set Size (N) vs. Latency (Cycles)

- ▲ 256 x 16KB
- ■ 2K x 16KB
- ○ 256 x 128B
- ✖ 32 x 16KB

12 Addr
TLB + Cache Results

Latency from dTLB/dCache conflicts

Eviction Set Size (N)

Latency (Cycles)

23 Addrs

12 Addrs

256 x 16KB

2K x 16KB

256 x 128B

32 x 16KB
Roadmap

1. Tell when kernel load occurs
   - Correct = Load, Incorrect = No load

2. Correct = Load, Incorrect = No load

3. Contend with kernel addresses in cache

4. Ability to time cache miss vs cache hit

Contend with kernel addresses in cache

Ability to time cache miss vs cache hit

Tell when kernel load occurs

Correct = Load, Incorrect = No load
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
A single cache set

Let the kernel load cache line K1
prime+probe

A single cache set

Let the kernel load cache line K1
A single cache set

Now we reload A1, A2, A3, A4 in the same order...
A single cache set

prime+probe
prime+probe

A single cache set

K1  A1  A2  A4
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. Ability to time cache miss vs cache hit
2. Contend with kernel addresses in cache
3. Tell when kernel load occurs
4. Correct = Load, Incorrect = No load
3 Target Programs

- **Basic**
  Very simple AUT -> LDR

- **Advanced**
  C++ vtable function call

- **Ultra**
  A real system call in XNU
`Basic Victim`

```python
if (lots of instructions that take a very long time):
    aut
    ldr
```
Basic Victim VS PACMAN I
20,000 Runs

(a) Incorrect PAC
(b) Correct PAC

AUT -> LDR

AUT -> BLR
Basic Victim VS PACMAN I

20,000 Runs

![Bar Chart]

- Correct PAC
- Incorrect PAC

Frequency vs. Number of Misses

(c)
Basic Victim VS PACMAN I
20,000 Runs

Almost exactly 50% incorrect PACs perform a load anyways!
Advanced Victim

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

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

```assembly
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

```c
return target->helper.extenalMethod();
```

```assembly
ldr x16, [helper]
mov x17, helper
movk x17, #0xd86, 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

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

```assembly
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
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

"helper" object

vtable ptr

member vars

...  

Original vtable

externalMethod

...  

Our forged vtable

Our Code :)

...
Advanced Victim

"helper" object

vtable ptr

member vars

Original vtable

externalMethod

Our forged vtable

Our Code :)

Don't know a good pointer for this location - can't train branch predictor!
Advanced Victim

"helper" object

vtable ptr
member vars

Original vtable
externalMethod
...

Train branch predictor on old vtable

Our forged vtable
Our Code :)
...

Train branch predictor on original vtable
Advanced Victim

"helper" object

vtable ptr

member vars

Original vtable

externalMethod

... 

Run trials on forged table

Our forged vtable

Our Code :)

... 

Our forged vtable
Lengthening the Window

```c
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

limit & target need to be in different sets!
if (user_argument < limit) {
    return target->helper.externalMethod(); // virtual method call
}
nop
nop
nop
nop
nop
nop
bl _IOLog
...

...
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 gadget in _proc_rlimit_control at [(‘d’, fffffe0007951ac4)]
I found 1 gadget in _memorystatus_available_memory at [(‘d’, fffffe000796dad8)]
I found 1 gadget 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 gadget in _quotactl at [(‘d’, fffffe000763a8c8)]
I found 1 gadget in _sendfile at [(‘d’, fffffe00079ce8a4)]
I found 1 gadget 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
_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]
```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);
    ...
}
```

```c
XNU_PTRAUTH_SIGNED_PTR("proc.task") task;
```
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... :(
if (p->p_memstat_memlimit > 0) {
    ledger_get_entries(p->task->ledger, ...);
}

Branch Condition

PACMAN Gadget

load p->p_memstat_memlimit

compare to 0

Speculation Ends!

load p->task

check signature

load p->task->ledger (or panic)
if (p->p_memstat_memlimit > 0) {
    ledger_get_entries(p->task->ledger, ...);
}

Lengthening window by evicting p->p_memstat_memlimit also lengthens p->task load!

Branch Condition

PACMAN Gadget

load p->p_memstat_memlimit

load p->task

compare to 0

check signature

load p->task->ledger (or panic)

Speculation Ends!
Poor choice of victim pointer-
Very commonly used field in a frequently accessed page
Kernel Panic

Differentiation Pattern trace1

Differentiation Pattern trace2

Differentiation Pattern trace3

Differentiation Pattern trace4

Differentiation Pattern trace5

Differentiation Pattern trace6
Beware of asynchronous accesses
All Code on our GitHub!

https://github.com/jprx/DEFCON30-PACMAN
Questions? Reach out:

@0xjprx

PACMANATTACK.COM