Pacman
ATTACKING ARM POINTER AUTHENTICATION WITH SPECULATIVE EXECUTION

Joseph Ravichandran*, Weon Taek Na*, Jay Lang, Mengjia Yan
*Both authors contributed equally to this work.
$whoami$

Joseph Ravichandran
1st Year PhD Student, MIT

Weon Taek Na
1st Year PhD Student, MIT
Memory Corruption Attacks

Microarchitectural Attacks
Memory Corruption Attacks

Microarchitectural Attacks

PACMAN
Contributions

1. New way of thinking about compounding threat models.
2. Hardware bypass for ARM Pointer Authentication.
Memory Corruption

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

Read/Write Memory → Change Execution → Arbitrary Code Execution

Pointer Authentication blocks changing pointers
Memory Corruption

- Read/Write Memory
- Write function pointer with forged hash
- Arbitrary Code Execution
Just bruteforce it, right?
Key Insight: Avoid crashes using speculative execution!
Agenda

1. Background
2. High Level View
3. Data Attack
4. Instruction Attack
5. Analysis
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

<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 overwrites the function pointer!
Let's fix this bug with **Pointer Authentication**.
ARM Pointer Authentication

\[ \text{PAC} = \text{crypto\_fn}(\text{pointer}, \text{salt}, \text{key}) \]
Buffer Overflow

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

Invalid PAC means we **crash**!
THE GOAL

Reveal the PAC for an arbitrary pointer without crashing.
Break PAC with **Hardware Attacks**

- Try a PAC **speculatively** to prevent crashes
- Leak verification results via side channel
Speculative Execution

if (true)
  A
else
  B

In Order

| Branch | A |

Speculative

<table>
<thead>
<tr>
<th>Branch</th>
<th>Undo B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speculate B</td>
<td>Microarchitectural side effects NOT undone</td>
<td></td>
</tr>
</tbody>
</table>

Time
We use side channels to transmit the verification results of a pointer without causing a crash.
Bird's Eye View

Write PAC guess into memory with existing software bug

Run PACMAN Gadget with guess

Correct
Observe Load!

Incorrect
No Load
if (condition):
    verified_ptr = check_pac(guess_ptr)
    load(verified_ptr)
if (condition):
    verified_ptr = check_pac(guess_ptr)
load(verified_ptr)
if (condition):
    verified_ptr = check_pac(guess_ptr)
load(verified_ptr)
if (condition):
    verified_ptr = check_pac(guess_ptr)
load(verified_ptr)
if (condition):
    verified_ptr = check_pac(guess_ptr)
load(verified_ptr)
Data Attack

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

Correct PAC

Mispredict Branch

PAC Check Succeeds → Speculative Load!

Incorrect PAC

Mispredict Branch

PAC Check Fails
if (condition):
    verified_ptr = check_pac(guess_ptr)
    load(verified_ptr)
Instruction Gadget

```python
if (condition): #BR1
    verified_ptr = check_pac(guess_ptr)
    call(verified_ptr) #BR2
```
The world's first desktop CPU that supports Pointer Authentication.
Challenges of Real World HW

- No documentation of microarchitectural details.
- No high resolution timer.
- macOS is a difficult system to integrate attacks on.

Essentially, we had to reinvent the wheel.
Conjectured TLB Hierarchy

- L1 User iTLB
  - 32 sets, 4 ways

- L1 Kernel iTLB
  - 32 sets, 4 ways

- L1 dTLB
  - 256 sets, 12 ways

- L2 TLB
  - 2048 sets, 23 ways
Conjectured TLB Hierarchy

L1 User iTLB
32 sets, 4 ways

L1 Kernel iTLB
32 sets, 4 ways

L1 dTLB
256 sets, 12 ways

L2 TLB
2048 sets, 23 ways
Conjectured TLB Hierarchy

- **L1 User iTLB**: 32 sets, 4 ways
- **L1 Kernel iTLB**: 32 sets, 4 ways
- **L1 dTLB**: 256 sets, 12 ways
- **L2 TLB**: 2048 sets, 23 ways
Experiment Testbed:
We insert a vulnerable kernel extension to target for our experiments.
PAC Oracle Accuracy

(a) Incorrect PAC
(b) Correct PAC

Frequency of Number of Misses

Data
Instructions
Under the PACMAN kext, we find each run takes 2.69ms.

This will likely be longer for real kernel code.

We can bruteforce an entire 16-bit PAC (from 0x0000 to 0xFFFF) in under 3 minutes.
### XNU-8019.80.24

<table>
<thead>
<tr>
<th>Data Gadgets</th>
<th>Instruction Gadgets</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>13,867</td>
<td>41,292</td>
<td>55,159</td>
</tr>
</tbody>
</table>

PACMAN Gadgets are readily available in large codebases.

This list is not exhaustive, and no exploitability analysis was performed.
More in the Paper!

- Reverse Engineering Experiments
- Example jump2win C++ Attack
- TLB Details
- CPU Cache Details
- Countermeasures
- Timers on M1
- And more!
PacmanOS
A Rust-based bare metal environment for performing experiments.
Top news

The M1 has a big security loophole, and Apple can't patch it
4 hours ago

MIT researchers uncover 'unpatchable' flaw in Apple M1 chips
51 minutes ago

MIT Finds New Arm Vulnerability Present in Apple M1, Demos PACMAN Attack
4 hours ago

PACMAN M1 chip attack defeats 'the last line of security' – but requires physical access
2 hours ago

All coverage

Apple M1 Affected By "PACMAN" Hardware Vulnerability In Arm Pointer Authentication
4 hours ago

MIT researchers discover Apple M1 chip vulnerability
3 hours ago

Apple M1 chip contains hardware vulnerability that bypasses memory defense
4 hours ago

Design Weakness Discovered in Apple M1 Kernel Protections
3 hours ago

Experts warn of 'PACMAN' flaw in M1 chip that can't be patched
1 hour ago
PACMAN: Attacking ARM Pointer Authentication with Speculative Execution
Appendix
Data PACMAN Attack

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

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

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

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

Reset the entire TLB

L1 dTLB

Branch Predictor

Taken
Data PACMAN Attack

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

Prime the L1 dTLB with an eviction set
Data PACMAN Attack

Prime the L1 dTLB with an eviction set

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

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

Prime the L1 dTLB with an eviction set
Data PACMAN Attack

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

Call the gadget with the pointer and PAC to guess.

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

Call the gadget with the pointer and PAC to guess.

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

If the guess was incorrect...

Call the gadget with the pointer and PAC to guess.

L1 dTLB

Branch Predictor

Taken

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

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

Speculative Exception!

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

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

Examine the eviction set and see if any lines were evicted.
Experiment 1
Data TLB Behavior
Reversing the Data TLB

Stride 0
- Address x

Stride 1
- \( x + \text{stride} + 1 \times 128 \)

Stride 2
- \( x + 2 \times \text{stride} + 2 \times 128 \)

\[ \text{addr}[i] = x + (i \times \text{stride}) + (i \times 128) \]
Reversing the Data TLB

addr[i] = x + (i*stride) + (i*128)

Move forward 1 stride

Stride 0
- Address x

Stride 1
- x + stride + 1 * 128

Stride 2
- x + 2 * stride + 2 * 128
Reversing the Data TLB

Address x

Stride 0

Stride 1

Stride 2

move forward 1 stride

\( \text{addr}[i] = x + (i \times \text{stride}) + (i \times 128) \)

Skipping a cache set each time
Data TLB Results

Latency from dTLB conflicts

Eviction Set Size (N)

Latency (Cycles)

- ▲ 256 x 16KB
- □ 2K x 16KB
- ● 256 x 128B
- ✗ 32 x 16KB
Data TLB Results

Latency from dTLB conflicts

Eviction Set Size (N)

Latency (Cycles)

L1 dTLB Conflicts

- ▲ 256 x 16KB
- □ 2K x 16KB
- ● 256 x 128B
- ✗ 32 x 16KB
Data TLB Results

Latency from dTLB conflicts

Latency (Cycles)

Eviction Set Size (N)

L1 dTLB Conflicts

L2 Conflicts

- ▲ 256 x 16KB
- ■ 2K x 16KB
- ● 256 x 128B
- ⚫ 32 x 16KB
Data TLB Eviction Sets

Latency from dTLB conflicts

Latency (Cycles)

Eviction Set Size (N)

- 256 x 16KB
- 2K x 16KB
- 256 x 128B
- 32 x 16KB
Data TLB Eviction Sets

Latency from dTLB conflicts

L1 dTLB
12 Addresses
Stride of 256 Pages
Data TLB Eviction Sets

Latency (Cycles)

Eviction Set Size (N)

Latency from dTLB conflicts

L1 dTLB
12 Addresses
Stride of 256 Pages

L2 TLB
23 Addresses
Stride of 2048 Pages
Experiment 2
Instruction TLB Behavior
Reversing the Instruction TLB

Stride 0

Address x

Stride 1

x + stride + 1 * 128

Stride 2

x + 2 * stride + 2 * 128

addr[i] = x + (i*stride) + (i*128)
Reversing the Instruction TLB

addr[i] = x + (i*stride) + (i*128)

Subtle difference
Load as instruction, measure as data
Instruction TLB Results

Latency from dTLB/iTLB conflicts

- ▲ 256 x 16KB
- ■ 2K x 16KB
- ● 256 x 128B
- ◯ 32 x 16KB
Instruction TLB Results

Latency (Cycles)

Eviction Set Size (N)

Latency from dTLB/iTLB conflicts

▲ 256 x 16KB   ■ 2K x 16KB   ● 256 x 128B   ◯ 32 x 16KB
Instruction TLB Results

Latency (Cycles)

Eviction Set Size (N)

Latency from dTLB/iTLB conflicts

- 256 x 16KB
- 2K x 16KB
- 256 x 128B
- 32 x 16KB
Instruction TLB Results

Latency (Cycles)

Eviction Set Size (N)

Latency from dTLB/iTLB conflicts

- 256 x 16KB
- 2K x 16KB
- 256 x 128B
- 32 x 16KB
Instruction TLB Results

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

Diagram showing various data points and trends.
Instruction TLB Results

Recall
Load as instruction, measure as data

256 x 16KB  2K x 16KB  256 x 128B  32 x 16KB
L1 dTLB behaves as a backing store for L1 iTLB.

Instruction TLB Results

- ▲ 256 x 16KB
- ■ 2K x 16KB
- ● 256 x 128B
- ✗ 32 x 16KB
Instruction TLB Eviction Sets

Latency (Cycles)

Eviction Set Size (N)

Latency from dTLB/iTLB conflicts

- 256 x 16KB
- 2K x 16KB
- 256 x 128B
- 32 x 16KB
After moving to dTLB, same eviction sets from before still work.
Instruction TLB Eviction Sets

4 Addresses

L1 iTLB
4 Addresses
Stride of 32 Pages.

Latency (Cycles)

Eviction Set Size (N)

Latency from dTLB/iTLB conflicts

- 256 x 16KB
- 2K x 16KB
- 256 x 128B
- 32 x 16KB
Experiment 3
TLB and Cache Interactions
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

- ▲ 256 x 16KB
- ▼ 2K x 16KB
- ● 256 x 128B
- × 32 x 16KB
TLB + Cache Results

Latency from dTLB/dCache conflicts

Latency (Cycles)

Eviction Set Size (N)

L1 Data Cache Conflicts

256 x 16KB  2K x 16KB  256 x 128B  32 x 16KB
TLB + Cache Results

Latency from dTLB/dCache conflicts

Eviction Set Size (N)

Latency (Cycles)

L1 Data Cache Conflicts

L1 dTLB Conflicts

256 x 16KB
2K x 16KB
256 x 128B
32 x 16KB
TLB + Cache Results

L1 dTLB Conflicts
L2 TLB Conflicts
L1 Data Cache Conflicts

Latency (Cycles)

Eviction Set Size (N)

Latency from dTLB/dCache conflicts

256 x 16KB
2K x 16KB
256 x 128B
32 x 16KB
TLB + Cache Results

L1 dTLB Conflicts
L2 TLB Conflicts
L1 Data Cache Conflicts

Latency (Cycles)

Eviction Set Size (N)

Latency from dTLB/dCache conflicts

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

12 Addrs
TLB + Cache Results

- L1 Data Cache Conflicts
- L1 dTLB Conflicts
- L2 TLB Conflicts
- 23 Addrs
- 12 Addrs

Latency from dTLB/dCache conflicts

Latency (Cycles)
Eviction Set Size (N)