Practical Cache Attacks

Mengjia Yan
Spring 2023
Leak Crypto Library #1: RSA

- Square-and-Multiply Exponentiation

**Input:**
- base $b$
- modulo $m$
- exponent $e = (e_{n-1} \ldots e_0)_2$

**Output:**
- $b^e \mod m$

```
r = 1
for i = n-1 to 0 do
    r = sqr(r)
    r = mod(r, m)
    if $e_i == 1$ then
        r = mul(r, b)
        r = mod(r, m)
    end
end
```
Leak Crypto Library #2: AES
Why Cache?

• Large attack surface. Shared across cores/sockets.

• Fast. Can be used to build high-bandwidth channels.

• Many states. Can encode secrets spatially to further improve bandwidth and precision.

• There exist many cache-like structures. The same attack concepts and tricks will apply.
The Goal: Monitor access patterns at cache line granularity
Attack Strategy #1: Flush+Reload

• The flush instructions allow explicit control of cache states
  • In X86, `clflush vaddr`
  • In ARM, `DC CIVAC vaddr`

• What are these flush instructions used for except for attacks?
  • For coherence, in the case when the data in the cache is inconsistent with the data in the DRAM.
    • 1) old time, incoherent DMA
    • 2) nowadays, Non-volatile memory for crash recovery
Flush+Reload

A shared cache line

Attacker: Flush
Victim: Access
Attacker: Reload -> low latency

Attacker: Flush
Victim: No Access
Attacker: Reload -> high latency

Cache

Victim
Attacker

DRAM

Time
Address Translation (4KB page)

Virtual Address (48bit):
- Virtual page number (VPN)
- Page offset (12 bits)

Physical Address (32bit):
- Physical page number (PPN)
- Page offset (12 bits)

Copy page offset

Page Table
### Shared Memory Between Untrusted Domains?

<table>
<thead>
<tr>
<th></th>
<th>Process-1</th>
<th>Process-2</th>
<th>Same?</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PPN of “printf”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PPN of “a stack variable”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PPN of “a heap variable”</td>
<td>After allocated</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>After read access</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>After write access</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Attack Code

In x86, 8 GPR:
- rax, rbx, rcx, rdx
- rsp, rbp
- rsi, rdi

“r” means 64-bit

replacing “r” with “e” means the lower 32 bits.

lfence:
- Load Fence
- Performs a serializing operation on all load instructions
A Demo
Attack Strategy #2: ?

• Cache state manipulation instructions
  • In X86, clflush vaddr
  • In ARM, DC CIVAC vaddr

• What if these instructions are not available in user space?
  • Apple devices
  • “Except ARMv8-A CPUs, ARM processors do not support a flush instruction”

_from ARMageddon: Cache Attacks on Mobile Devices (USENIX’16)_
Attack Strategy #2: **Evict**+Reload

A shared cache line

**Victim**

**Attacker**

**Cache**

**DRAM**

Attacker: Access a large buffer

Victim: Access

Victim: No Access

Attacker: Reload -> low latency

Attacker: Reload -> high latency

Attacker:

Access a large buffer

- low latency

- high latency
Lessons Learnt So Far:

The fundamental problem: shared memory between different security domains.

Source: https://kb.vmware.com/s/article/2080735
Attack Strategy #3: **Prime+Probe**

**Sender’s line**
- **Attacker:** Prime a cache set
- **Victim:** Access

**Receiver’s line**
- **Attacker:** Prime a cache set
- **Victim:** No Access
Attack Strategy #3: **Prime+Probe**

**Sender’s line**

**Receiver’s line**

**Attacker:** Prime a cache set

**Victim:** Access

**Attacker:** Prime a cache set

**Victim:** No Access
Attack Strategy #3: **Prime+Probe**

**Attacker:**
- Prime a cache set

**Victim:**
- Access
- No Access

**Sender’s line**
- Attacker: Prime a cache set
- Victim: Access

**Receiver’s line**
- Attacker: Probe -> **high** latency
- Victim: Probe -> **low** latency

**Cache**

**DRAM**
Analogy: Bucket/Ball

Each cache set is a bucket that can hold 8 balls.
N-way Set-Associative Cache

• Does cache use virtual address or physical address?
Using Caches with Virtual Memory

**Virtually-Addressed Cache**
- FAST: No virtual → physical translation on cache hits
- Problem: Must flush cache after context switch

**Physically-Addressed Cache**
- Avoids stale cache data after context switch
- SLOW: virtual → physical translation before every cache access
Best of Both Worlds (L1 Cache): Virtually-Indexed, Physically-Tagged Cache (VIPT)

Cache index comes entirely from address bits in page offset — don’t need to wait for TLB to start cache lookup!
## Cache and Address Translation

**Cache L1-D**  
- Size: 32KB ($2^{15}$ Bytes)  
- Line size: 64B ($2^8$ Bytes)  
- Associativity: 8

**Cache L3**  
- Size: 32MB ($2^{25}$ Bytes)  
- Line size: 64B ($2^8$ Bytes)  
- Associativity: 16

<table>
<thead>
<tr>
<th></th>
<th>Addr1</th>
<th>Addr2</th>
<th>Conflict?</th>
<th>Addr1</th>
<th>Addr3</th>
<th>Conflict?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Virtual</strong></td>
<td>Address</td>
<td></td>
<td>--</td>
<td></td>
<td></td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>Set Index</td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
<td>Conflict</td>
</tr>
<tr>
<td><strong>Physical</strong></td>
<td>Address</td>
<td></td>
<td>--</td>
<td></td>
<td></td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>Set Index</td>
<td></td>
<td>?</td>
<td></td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>
Using Huge Pages

- Huge page size: 2MB or 1GB

### Virtual Address: 4KB page

<table>
<thead>
<tr>
<th>Virtual page number</th>
<th>Page offset (12 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>12 11 0</td>
</tr>
</tbody>
</table>

### Cache mapping: (256 sets)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Set Index (8 bits)</th>
<th>Line offset (6 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Virtual Address: 2MB page

<table>
<thead>
<tr>
<th>Virtual page number</th>
<th>Page offset (21 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>21 20 0</td>
</tr>
</tbody>
</table>
Takeaways

• **Practical** challenges in implementing a reliable cache attack
  • Page sharing
  • Noise due to prefetchers
  • Uncertainty due to page mapping
  • Replacement policy
  • Etc.

• **Hardware and software optimizations** make attacks easier
  • Transparent page sharing
  • Copy-on-write
  • Huge pages
  • Virtually-indexed and physically-tagged caches
Next:
Transient Execution Attacks