Practical Cache Attacks

Mengjia Yan
Spring 2022
Reminder

• Lab 1 will be out TODAY

• Fill in paper preference on HotCRP by Friday
Recap: Prime+Probe

- **Prime**
- **Probe**
- **Shared Cache**
- **Sender**
- **Receiver**

**Cache Set**

**# ways**

**Shared Cache**

**Time**

**Sender line**

**Receiver line**

6.888 L4 - Practical Cache Attacks
Recap: Prime+Probe

Prime+Probe

Shared Cache

Sender

Receiver

Sender line

Receiver line

Cache Set

# ways

Shared Cache

Access

Prime

Wait

Time

6.888 L4 - Practical Cache Attacks
Recap: Prime+Probe

Receive “1” = 8 accesses → 1 miss
Analogy: Bucket/Ball

Each cache set is a bucket that can hold 8 balls.

How to find addresses that map to the same set?

Sender's address

Receiver's address

Sender

Receiver

# ways

Cache Set

Shared Cache
Review Cache and Address Translation

On blackboard

• Cache
  • Data + tag, directly mapped, N-way associative

• Page translation
  • Page table
  • The case when the same virtual address in different processes map to different physical addresses
  • The case when multiple virtual addresses map to the same physical address

• Hierarchical page tables and TLB

• Cache and virtual address
Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with $2^W$ lines):
  - Index into cache with $W$ address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit $== 1$ and tag matches upper address bits, HIT
- Example 8-line direct-mapped cache:

```
32-bit BYTE address
```

```
0000000000000000000000000000000011101000
```

```
Valid bit
```

```
Tag (27 bits)
```

```
Data (32 bits)
```

```
Index bits
```

```
Byte offset bits
```

```
=?
```

= HIT

April 15, 2021

6.888 L4 - Practical Cache Attacks
MIT 6.004 Spring 2021

L15-8
N-way Set-Associative Cache

- Use multiple direct-mapped caches in parallel to reduce conflict misses
- Nomenclature:
  - # Rows = # Sets
  - # Columns = # Ways
  - Set size = #ways = “set associativity” (e.g., 4-way → 4 lines/set)
- Each address maps to only one set, but can be in any way within the set
- Tags from all ways are checked in parallel
- Fully-associative cache: Extreme case with a single set and as many ways as cache lines
Address Translation (4KB page)

Programmer’s view
Virtual Address (48bit):

System’s view
Physical Address (32bit):
Hierarchical Page Table

Virtual Address

Root of the Current Page Table

(Processor Register)

Level 1 Page Table

Level 2 Page Tables

Data Pages

page in primary memory
page in secondary memory
PTE of a nonexistent page
Translation Lookaside Buffer (TLB)

Problem: Address translation is very expensive! Each reference requires accessing page table

Solution: *Cache translations in TLB*

- TLB hit \(\Rightarrow\) *Single-cycle translation*
- TLB miss \(\Rightarrow\) *Access page table to refill TLB*

Virtual address

<table>
<thead>
<tr>
<th>V</th>
<th>R</th>
<th>W</th>
<th>D</th>
<th>tag</th>
<th>PPN</th>
</tr>
</thead>
</table>

(\(VPN =\) virtual page number)

Physical address

<table>
<thead>
<tr>
<th>PPN</th>
<th>offset</th>
</tr>
</thead>
</table>

(\(PPN =\) physical page number)

fault? hit?
Using Caches with Virtual Memory

Virtually-Addressed Cache

Physically-Addressed Cache

Which one is better?

Can we do even better?

Hint: some part of the virtual address and physical address are the same
Best of Both Worlds: Virtually-Indexed, Physically-Tagged Cache (VIPT)

OBSERVATION: If cache index bits are a subset of page offset bits, tag access in a physical cache can be done in parallel with TLB access. Tag from cache is compared with physical page address from TLB to determine hit/miss.

Problem: Limits # of bits of cache index → can only increase cache capacity by increasing associativity!
Find Conflicting Addressing From Virtual Address

Virtual Address (48bit):

- Virtual page number
- Page offset

Physical Address (32bit):

- 4KB page
- Physical page number
- Page offset (12 bits)

Cache mapping:

- (8 sets)
  - Tag
  - Index (3 bits)
  - Line offset (6 bits)

- (256 sets)
  - Tag
  - Set Index (8 bits)
  - Line offset (6 bits)

Not controllable via virtual address.
w/ Huge Pages

- Huge page size: 2MB or 1GB
- Number of bits for page offset?

### Virtual Address: 4KB page

- Virtual page number: 48
- Page offset: 12, 11, 0

### Virtual Address: 2MB page

- Virtual page number: 48, 21, 20
- Page offset: 21 bits

### Cache mapping: (256 sets)

- Tag: 6 bits
- Set Index: 8 bits
- Line offset: 6 bits

6.888 L4 - Practical Cache Attacks
Micro-arch Side Channel Generalization

Sender

if (send ‘1’)
    Use resource
else
    idle

Hardware resource

Receiver

t1 = rdtsc()
Use resource
t2 = rdtsc()

if (t2 - t1 > THRESH)
    read ‘1’
else
    read ‘0’
From Covert \( \rightarrow \) Side Channels

**Covert channel:**

\[
\text{if (send ‘1’) Use resource} \\
\text{else idle}
\]

**Side channel:**

\[
\text{if (secret) Use resource} \\
\text{else idle}
\]

**Software resource:**

\[
t1 = \text{rdtsc() Use resource} \\
t2 = \text{rdtsc()}
\]

**Side channel:**

\[
\text{if (t2 – t1 > THRESH) read ‘1’} \\
\text{else read ‘0’}
\]
Breaking RSA

• A real-world example: Square-and-Multiply Exponentiation

What you generally see in papers:

\[
\text{for } i = n-1 \text{ to } 0 \text{ do}
\]

\[
\begin{align*}
    r &= \text{sqr}(r) \\
    r &= r \mod n \\
    \text{if } e_i &= 1 \text{ then} \\
    r &= \text{mul}(r, b) \\
    r &= r \mod n \\
    \text{end} \\
\end{align*}
\]

\text{end}
The Multiply Function

```c
471 mpi_limb_t
472 mpihelp_mul( mpi_ptr_t prodp, mpi_ptr_t up, mpi_size_t usize,
473           mpi_ptr_t vp, mpi_size_t vsize)
474 { mpi_ptr_t prod_endp = prodp + usize + vsize - 1;
475   mpi_limb_t cy;
476   struct karatsuba_ctx ctx;
477   if( vsize < KARATSUBA_THRESHOLD ) {
478     mpi_size_t t;
479     mpi_limb_t v_limb;
480     if( !vsize )
481       return 0;
482   } /* Multiply by the first limb in V separately, as the result can be */
483   /* stored (not added) to PROD. We also avoid a loop for zeroing. */
484   v_limb = vp[0];
485   if( v_limb <= 1 ) {
486     if( v_limb == 1 )
487       MPN_COPY( prodp, up, usize );
488     else
489       MPN_ZERO( prodp, usize );
490     cy = 0;
491   } else
492     cy = mpihelp_mul_1( prodp, up, usize, v_limb );
493   prodp[usize] = cy;
494   prodp++;
501   /* For each iteration in the outer loop, multiply one limb from */
502   /* U with one limb from V, and add it to PROD. */
503   for( i = 1; i < vsize; i++ ) {
504     v_limb = vp[i];
505     if( v_limb <= 1 )
506       cy = 0;
507     else
508       cy = mpihelp_add_n(prodp, prodp, up, usize);
509     prodp[usize] = cy;
510     prodp++;
511   }
512   return cy;
513 }
521   memset( &ctx, 0, sizeof ctx );
522   mpihelp_mul_karatsuba_case( prodp, up, usize, vp, vsize, &ctx );
523   mpihelp_release_karatsuba_ctx( &ctx );
524   return *prod_endp;
525 }
```
Access latencies measured in the probe operation in Prime+Probe.
A sequence of “01010111011001” can be deduced as part of the exponent.
More Advanced Cache Attacks

You may not need the following tips for Lab 1
Multi-level Caches

• Motivation:
  • A memory cannot be large and fast. Add level of cache to reduce miss penalty

A typical configuration of Intel Ivy Bridge. Configurations are different with processor types.

<table>
<thead>
<tr>
<th></th>
<th>L1-I/D cache</th>
<th>L2 cache</th>
<th>L3 cache (LLC)</th>
<th>DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size</strong></td>
<td>32KB</td>
<td>256KB</td>
<td>1MB/core</td>
<td>16GB</td>
</tr>
<tr>
<td><strong>Associativity (# ways)</strong></td>
<td>4 or 8</td>
<td>8</td>
<td>16</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Latency (cycles)</strong></td>
<td>1-5</td>
<td>12</td>
<td>~40</td>
<td>~150</td>
</tr>
</tbody>
</table>
Multi-level Caches

• Motivation:
  • A memory cannot be large and fast. Add level of cache to reduce miss penalty

• LLC is generally divided into multiple slices
  • Conflict happens if addresses map to the same slice and the same set

Slice ID = Hash(bits)

An undocumented secret hash function

6.888 L4 - Practical Cache Attacks
Eviction Set Construction Algorithm

Sender line

Receiver line

Time

Access Candidate Addresses

Eviction Set Construction Algorithm

Eviction Set Construction Algorithm

Access Candidate Addresses
Wait
Measure Latency of Each Candidate Address

**Problems Due to Replacement Policy**

- Self-eviction due to replacement policy
  - An LRU (least recently used) example

- A small trick:
  - Access addresses in reverse order

![Diagram showing cache access and eviction process]

- Initial:
- Prime: 1 2 3 4 5 6 7 8
- Victim access: 9 2 3 4 5 6 7 8
- Probe: 9 2 3 4 5 6 7 8
- Which to evict?
Defenses
Micro-architecture Side Channels A Communication Model

Kiriansky et al. DAWG: a defense against cache timing attacks in speculative execution processors. MICRO’18
Micro-architecture Side Channels

Victim

secret-dependent execution

(a micro-architecture structure)

Attacker

Defenses:

Block creation of signals:
Oblivious execution, etc.

Close the channel:
Isolation, etc.

Block detection of signals:
Randomization, etc.

Kiriansky et al. DAWG: a defense against cache timing attacks in speculative execution processors. MICRO’18
Defense Design Considerations

- Security
- Performance
- Portability
Breaking RSA

• A real-world example: Square-and-Multiply Exponentiation

What you generally see in papers:

\[
\text{for } i = n-1 \text{ to } 0 \text{ do}
\]

\[
\begin{align*}
  & r = \text{sqr}(r) \\
  & r = r \mod n \\
  & \text{if } e_i == 1 \text{ then} \\
  & \quad r = \text{mul}(r, b) \\
  & \quad r = r \mod n \\
  & \text{end}
\end{align*}
\]

\text{end}
Data Oblivious/“Constant time” Programming

Write program w/o data-dependent behavior

Original:

```c
if (secret)
    a = *(addr1);
else
    a = *(addr2);
```

Data Oblivious:

```c
a ← load (addr1);
b ← load (addr2);
cmov a = (secret) ? a : b;
```

**secret** = confidential
addr1 = public
addr2 = public

How about nested branches?
Data Oblivious/“Constant time” Programming

Original:

\[
\begin{align*}
a &= \text{buffer}[\text{secret}] \\
\text{secret} &= \text{confidential} \\
\text{buffer} &= \text{public}
\end{align*}
\]

Data Oblivious:

\[
\begin{align*}
&\text{for (int } i=0; i<\text{size}; i++) \\
&\quad \text{tmp} = \text{buffer}[i]; \\
&\quad \text{cmov } a = (i==\text{secret}) \ ? \ \text{tmp} : a;
\end{align*}
\]

\[
\begin{align*}
\text{secret} &= \text{confidential} \\
\text{buffer} &= \text{public}
\end{align*}
\]

Other data-dependent instruction optimizations:
\begin{itemize}
  \item e.g., zero-skip, early exit, microcode, silent stores, ...
\end{itemize}

<table>
<thead>
<tr>
<th></th>
<th>Zero</th>
<th>Normal</th>
<th>Subnormal</th>
<th>Infinity</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency (in cycles) of the SQRTSS instruction for various operands.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>11</td>
<td>153</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Rane et al. Secure, Precise, and Fast Floating-Point Operations on x86 Processors. USENIX’16

6.888 L4 - Practical Cache Attacks 35
Micro-architecture Side Channels

Defenses:

Block creation of signals:
Oblivious execution, etc.

Close the channel:
Isolation, etc.

Block detection of signals:
Randomization, etc.

Kiriansky et al. DAWG: a defense against cache timing attacks in speculative execution processors. MICRO’18
HW Resource Partition

- Cache way-partition v.s. Set partition
- Temporal Partition v.s. Spatial Partition
- Security v.s. Quality of Service (QoS)
  - Intel Cache Allocation Technology (CAT)

Challenges nowadays:
- Security domain determination is tricky nowadays
- Scalability: what is #domains > #partitions
- How to partition inside cores?
- Why not execute applications on a single node?

From
Randomization/Fuzzing

• Introduce noise to time measurement/Make time measurement coarse-grained
  • Pros and cons?

  + Simple and no performance overhead
  + Effective towards a group of popular attacks
  ......
  - Not effective to attacks that do not measure time
  - Not effective to victims that cause big timing difference
  - Affect usability if benign application needs to use a fine-grained timer

*Shusterman et al. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. USENIX’21*
Next: Paper Discussion

Opening Pandora’s Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data
Review
Address Translation and Cache

Slides from 6.004
Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with $2^W$ lines):
  - Index into cache with $W$ address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, HIT
- Example 8-line direct-mapped cache:

![Diagram of Direct-Mapped Cache](image)
N-way Set-Associative Cache

- Use multiple direct-mapped caches in parallel to reduce conflict misses
- **Nomenclature:**
  - # Rows = # Sets
  - # Columns = # Ways
  - Set size = #ways = “set associativity” (e.g., 4-way → 4 lines/set)
- Each address maps to only one set, but can be in any way within the set
- Tags from all ways are checked in parallel

- Fully-associative cache: Extreme case with a single set and as many ways as cache lines
Paged Memory Systems

- Divide physical memory in fixed-size blocks called **pages**
  - Typical page size: 4KB

- Interpret each virtual address as a pair `<virtual page number, offset>`

- Use a **page table** to translate from virtual to physical page numbers
  - Page table contains the physical page number (i.e., starting physical address) for each virtual page number
Size of Linear Page Table

With 32-bit addresses, 4 KB pages & 4-byte PTEs:

⇒ $2^{20}$ PTEs, i.e., 4 MB page table per user
⇒ 4 GB of swap space needed to back up the full virtual address space

Larger pages?

- Internal fragmentation (Not all memory in a page is used)
- Larger page fault penalty (more time to read from disk)

What about 64-bit virtual address space???

- Even 1MB pages would require $2^{44}$ 8-byte PTEs (35 TB!)

What is the “saving grace”??
Hierarchical Page Table

Virtual Address

\[
\begin{array}{c}
31 & 22 & 21 & 12 & 11 & 0 \\
p1 & p2 & \text{offset} & 10\text{-bit} & 10\text{-bit} & \text{L1 index} \,<,\, \text{L2 index} \\
\end{array}
\]

Root of the Current Page Table

(Processor Register)

Level 1 Page Table

Level 2 Page Tables

Page in primary memory

Page in secondary memory

PTE of a nonexistent page

Data Pages

February 9, 2022
Translation Lookaside Buffer (TLB)

Problem: Address translation is very expensive! Each reference requires accessing page table

Solution: *Cache translations in TLB*

- **TLB hit** ⇒ *Single-cycle translation*
- **TLB miss** ⇒ *Access page table to refill TLB*

![Diagram showing virtual and physical address translation]

(VPN = virtual page number)

(PPN = physical page number)

fault?  hit?

virtual address

6.888.14 - Practical Cache Attacks

MIT 6.004 Spring 2021
Address Translation

*Putting it all together*

Virtual Address

- TLB Lookup
  - hit
  - miss
    - Page Table Lookup
      - the page is in memory
        - Page Fault (OS loads page)
          - Resume process at faulting instruction
        - not in memory
          - Protection Fault
            - SEGFAULT
  - Protection Check
    - denied
    - permitted
    - Physical Address (to mem)

- Update TLB
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: Virtually-Indexied, Physically-Tagged Cache (VIPT)

OBSERVATION: If cache index bits are a subset of page offset bits, tag access in a physical cache can be done in parallel with TLB access. Tag from cache is compared with physical page address from TLB to determine hit/miss.

Problem: Limits # of bits of cache index → can only increase cache capacity by increasing associativity!