Non-transient Side Channels

Mengjia Yan
Fall 2020
Lab Assignment

• Handout on course website
• Each (regular) student will receive an email
  • Solo or 2-person group
  • Individual GitHub repo
  • Info about accessing a server machine
• Listeners can send us an email if you want to try the lab

• Advice:
  • Start early. The first step is not to implement the attack, but to reverse engineer the machine.
Recap: Prime+Probe

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

<table>
<thead>
<tr>
<th>Cache Set</th>
<th># ways</th>
<th>Shared Cache</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6.888</td>
<td>L5</td>
<td></td>
</tr>
</tbody>
</table>

Non-transient Side Channels: 3
Recap: Prime+Probe

Prime + Probe

Shared Cache

Sender

Receiver

Sender line

Receiver line

# ways

Cache Set

Shared Cache

Time

Access

Prime

Wait

6.888 L5-Non-transient Side Channels
Recap: Prime+Probe

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

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

How many cache lines in total in the system? How to find the bucket used by the sender?
Practical Cache Side Channels
Cache Mapping – Directly Mapped Cache

- Can think cache mapping as a hash table with limited size
- Linear cache set mapping using modular arithmetic

### Physical Address:

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data (64 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set Index = (Addr / Block Size) % Number of Sets

32bit
Cache Mapping – Directly Mapped Cache

- Can think cache mapping as a hash table with limited size
- Linear cache set mapping using modular arithmetic

**Physical Address:**

<table>
<thead>
<tr>
<th>31</th>
<th>9</th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag (high order bits)</td>
<td>Set Index (3 bits)</td>
<td>Line offset (6 bits)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To distinguish addresses in the same set

Number of bits for set index = $\log_2(\text{Number of sets})$

**Question:** Given an 1MB L2 with 1024 sets, how many bits are used for set index?

Assuming byte-addressable
Cache Mapping – Set Associative Cache

- Can think cache mapping as a hash table with limited size
- Linear cache set mapping using modular arithmetic

**Physical Address:**

- Tag (high order bits)
- Set Index (3 bits)
- Line offset (6 bits)

**2-way cache**

**Find eviction set**

- Find addresses with the same set index bits

**Question:** How to decide which way to use?

**Answer:** Cache replacement policy.
Address Translation (4KB page)

Programmer’s view
Virtual Address (48bit):

Virtual page number
Page offset (12 bits)

system’s view
Physical Address (32bit):

physical page number
Page offset (12 bits)

6.888 L5-Non-transient Side Channels
### Find Eviction Set Using Virtual Addresses

<table>
<thead>
<tr>
<th>Virtual Address (48bit):</th>
<th>Physical Address (32bit): 4KB page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual page number</td>
<td>physical page number</td>
</tr>
<tr>
<td>Page offset</td>
<td>Page offset (12 bits)</td>
</tr>
</tbody>
</table>

**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.

6.888 L5-Non-transient Side Channels
Huge Pages

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

Virtual Address:
4KB page

- Virtual page number
- Page offset (12 bits)

Virtual Address:
2MB page

- Virtual page number
- Page offset (21 bits)

Cache mapping:
(256 sets)

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

6.888 L5-Non-transient Side Channels
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>Size</td>
<td>32KB</td>
<td>256KB</td>
<td>1MB/core</td>
<td>16GB</td>
</tr>
<tr>
<td>Associativity</td>
<td>4 or 8</td>
<td>8</td>
<td>16</td>
<td>N/A</td>
</tr>
<tr>
<td>( narrow ways)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latency (cycles)</td>
<td>1-5</td>
<td>12</td>
<td>~40</td>
<td>~150</td>
</tr>
</tbody>
</table>

6.888 L5-Non-transient Side Channels
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 L5-Non-transient Side Channels
Eviction Set Construction Algorithm

Sender

Receiver

Sender line

Receiver line

Time

Access Candidate Addresses

Eviction Set Construction Algorithm


6.888 L5-Non-transient Side Channels
Eviction Set Construction Algorithm

Sender

Receiver

Access Candidate Addresses

Wait

Measure Latency of Each Candidate Address

Sender line

Receiver line

Time

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

Initial: [Blank]
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?
Measure Latency of Multiple Accesses

- HW Prefetcher + Out-of-order execution

\[ T_1 = \text{rdtsc()} \]
\[ \text{Dummy1}=\text{Ld(Addr1)} \]
……
\[ \text{Dummy8}=\text{Ld(Addr8)} \]
\[ T_2 = \text{rdtsc()} \]
\[ \text{Latency} = T_2 - T_1 \]

What we expect:

What actually will happen:
Out-of-Order Processor

Check whether the register to read is ready.

Question: How to serialize data accesses?
Serialize Data Accesses

- A special instruction “mfence” 
- Add data dependency by creating a linked list

6.888 L5-Non-transient Side Channels
Handle Noise

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

What you generally see in papers:

```
for i = n-1 to 0 do
    r = sqr(r) mod n
    if e_i == 1 then
        r = mul(r, b) mod n
    end
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 {
475     mpi_ptr_t prod_endp = prodp + usize + vsize - 1;
476     mpi_limb_t cy;
477     struct karatsuba_ctx ctx;
478     if( vsize < KARATSUBA_THRESHOLD ) {
479         mpi_size_t t;
480         mpi_limb_t v_limb;
481         if( !vsize )
482             return 0;
483         /* Multiply by the first limb in V separately, as the result can be
484          * stored (not added) to PROD. We also avoid a loop for zeroing. */
485         v_limb = vp[0];
486         if( v_limb <= 1 ) {
487             if( v_limb == 1 )
488                 MPN_COPY( prodp, up, usize );
489             else
490                 MPN_ZERO( prodp, usize );
491             cy = 0;
492         }
493         else
494             cy = mpihelp_mul_1( prodp, up, usize, v_limb );
495     }
496     prodp[usize] = cy;
497     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++;
518         return cy;
519     }
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;
```
Access latencies measured in the probe operation in Prime+Probe.
A sequence of “01010111011001” can be deduced as part of the exponent.
There may exist other problems

• Tips for lab assignment
  • Build the attack step-by-step
  • Recommend to read “Last-Level Cache Side-Channel Attacks are Practical”
  • Ask questions via Piazza
Defenses
Micro-architecture Side Channels

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

Defenses:

- Block creation of signals: Oblivious execution, speculative execution defenses, 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

6.888 L5-Non-transient Side Channels
Defense Design Considerations

- Security
- Performance
- Portability

6.888 L5-Non-transient Side Channels
The Problem: The ISA Abstraction

• Interface between HW and SW: ISA
  • Advantage: HW optimizations without affecting usability/portability
DEC — Decrement by 1

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE /1</td>
<td>DEC r/m8</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Decrement r/m8 by 1.</td>
</tr>
<tr>
<td>REX + FE /1</td>
<td>DEC r/m8*</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Decrement r/m8 by 1.</td>
</tr>
<tr>
<td>FF /1</td>
<td>DEC r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Decrement r/m16 by 1.</td>
</tr>
<tr>
<td>FF /1</td>
<td>DEC r/m32</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Decrement r/m32 by 1.</td>
</tr>
<tr>
<td>REX.W + FF /1</td>
<td>DEC r/m64</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Decrement r/m64 by 1.</td>
</tr>
<tr>
<td>48+rw</td>
<td>DEC r16</td>
<td>O</td>
<td>N.E.</td>
<td>Valid</td>
<td>Decrement r16 by 1.</td>
</tr>
<tr>
<td>48+rd</td>
<td>DEC r32</td>
<td>O</td>
<td>N.E.</td>
<td>Valid</td>
<td>Decrement r32 by 1.</td>
</tr>
</tbody>
</table>

* In 64-bit mode, r/m8 cannot be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM: r/m (r, w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>O</td>
<td>opcode + rd (r, w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Subtracts 1 from the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (To perform a decrement operation that updates the CF flag, use a SUB instruction with an immediate operand of 1.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, DEC r16 and DEC r32 are not encodable (because opcodes 48H through 4FH are REX prefixes). Otherwise, the instruction’s 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST ← DEST − 1;

From https://www.felixcloutier.com/x86/index.html
The Problem: The ISA Abstraction

• Interface between HW and SW: ISA

• ISA specifies functionality, not performance/timing
  • Compare Intel Ivy Bridge and Cascade Processor

Example:

DEC [addr]
Data Oblivious/“Constant time” Programming

Write program w/o data-dependent behavior

Original:

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

Data Oblivious:

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

secret

6.888 L5-Non-transient Side Channels
Programming in Circuit Abstraction

- Program = DAG ("circuit")
- Operations = nodes ("gates")
- Data transfers = edges ("wires")

- Topology must be confidential data-**independent**
- Each gate’s execution must hide its inputs
- Each wire must hide the value it carries
What assumptions underpin the model?

- **Rule 1**: instruction/gate execution = confidential data-independent
- **Rule 2**: data transfer/wire = confidential data-independent
- **Rule 3**: circuit/program topology = fixed

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

- `secret` = confidential
- `addr1` = public
- `addr2` = public

```
a ← load addr1
b ← load addr2
```

```
cmov secret, b, a
```

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

```c
secret = confidential
addr1 = public
addr2 = public
```
Today’s machines can violate these assumptions

Violations due to:

Data-dependent instruction optimizations

(e.g., zero-skip, early exit, microcode, silent stores, ...)

- **Rule 1:** instruction/gate execution = confidential data-independent
- **Rule 2:** data transfer/wire = confidential data-independent
- **Rule 3:** circuit/program topology = fixed
Today’s machines can violate these assumptions

Violations due to:

Data at rest optimizations

(e.g., compression in register file/uop fusion, cache, page tables, …)

• Rule 1: instruction/gate execution = confidential data-independent
• Rule 2: data transfer/wire = confidential data-independent
• Rule 3: circuit/program topology = fixed
Today’s machines can violate these assumptions

Violations due to:
Speculative/OoO execution

- **Rule 1:** instruction/gate execution = confidential data-independent
- **Rule 2:** data transfer/wire = confidential data-independent
- **Rule 3:** circuit/program topology = fixed
HW Resource Partition

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

- 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?
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

• Randomize cache mapping functions
  • Pros and cons?
    + Generally low performance overhead (still allow cache to be shared)
    - Difficult to reason about security
    +/- Can reduce attack bandwidth, but unlikely to eliminate attacks
Next Lecture:
Transient Side Channels