Advanced Computer Architecture

—

Part III: Hardware Security
Cache Attack Lab

Shanqing Lin & Paolo Ienne

<shanqing.lin@epfl.ch> & <paolo.ienne@epfl.ch>

Inspired by an assignment by Marten van Dijk, CWI, The Netherlands

Victim Function: Look-Up Table

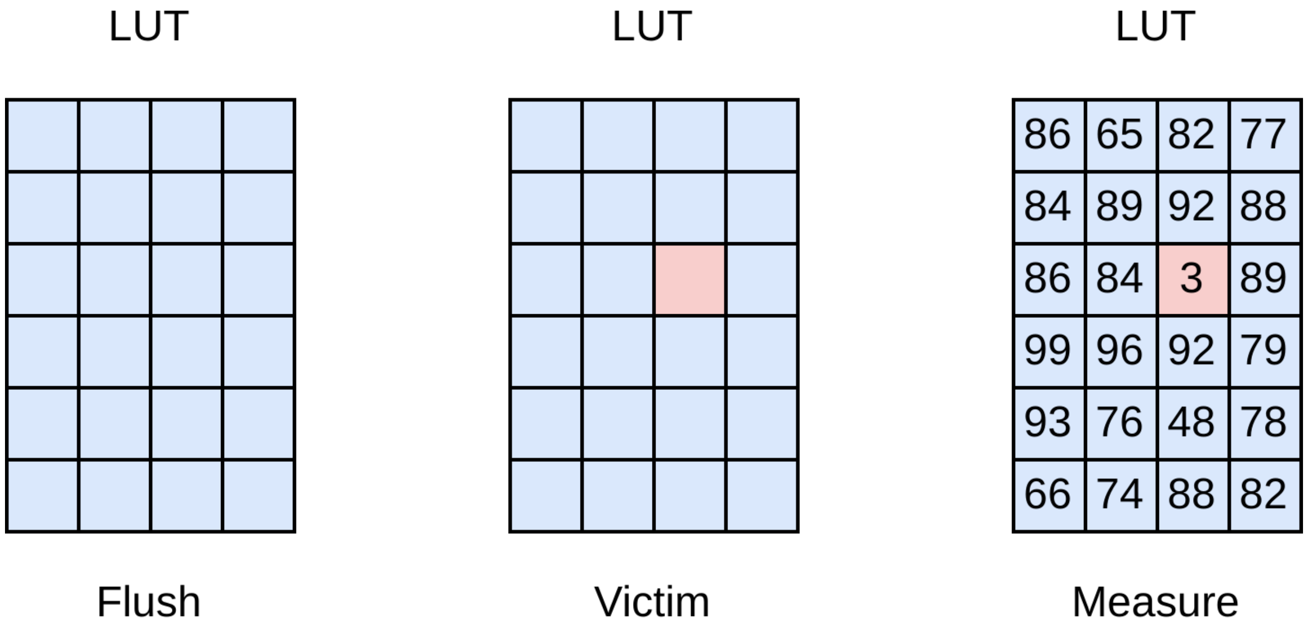
- We want to guess a number used by a victim to look up a table (`index`)

```
// See attack.c
int LUT[256 * 512];
int victim(int input){
  // some processing over input
  int index = (input * 163) & 0xFF;
  // Use it to access the LUT.
  volatile int internal_value = LUT[index * 512];
  // Other processing...
  return (internal_value * 233) & 0xFFFF;
}
```

- For simplicity:
  - Accesses are distributed in different cache blocks (`index * 512`)
  - The attacker and the victim share the same process (and therefore the same virtual addresses)

Goal: Flush+Reload Attack

- Flush+Reload attack: since we share the address space with the victim, we can reload the flushed victim data and directly see if we get a hit (= victim accessed data):
  - Flush LUT from cache
  - Call the victim (it will access an unknown entry which will reveal the secret)
  - Access each element of LUT and measure time (hits vs. misses)



Environment Setup

- You need a C compiler **on an x86 machine** for this lab
- Linux users:
  - Install gcc or clang according to your distribution
- Windows users:
  - You can use any C/C++ IDE (Visual Studio, CLion)
  - To use gcc, check this link: https://nuwen.net/mingw.html
- MacOS X users:
  - **This lab cannot be run without modifications on an M1 CPU**
  - You can use any C/C++ IDE (XCode, CLion)
  - Install XCode command line tools: `xcode-select -install`
  - Or use brew to install gcc: `brew install gcc`

How to Flush a Variable from the Cache?

- Use x86 intrinsic _mm_clflush and _mm_mfence
- To use them, include <x86intrin.h> (GCC or Clang) or <intrin.h> (MSVC)

```
// here is a variable
int variable_to_flush = 100;
// you want to flush it
_mm_clflush(&variable_to_flush);
// wait several cycles for clflush to commit
for (volatile int i = 0; i < 100; i++);
// memory fence
_mm_mfence();
// this will trigger a cache miss
variable_to_flush = 200;
```

- However, we don't know if a cache miss occurs without measuring time...

Why the Volatile For Loop and the Memory Fence?

- Why the volatile for loop?
  - Your x86 is an out-of-order processor and other instructions may be executed before `cflush` get executed and committed (including those to measure time)
  - A `for` loop waits for some cycles so that the following instructions will not enter the pipeline before `cflush` commits.
  - Since the loop body is empty, adding `volatile` to the variable ensures that the compiler will not remove the loop altogether
- Why the memory fence?
  - The processor and compiler may advance later memory instructions to improve performance
  - A memory fence prevents later memory instruction to get executed until the `mfence` instruction is committed
  - Probably the volatile for loop and the memory fence are partly mutually redundant, but there is no harm in using both for safety

How to Measure Time?

- Use x86 intrinsic __rdtscp, which requires to include <x86intrin.h>
- Also include <stdint.h> for 64-bit integers

```
volatile unsigned int junk = 0; // A junk number as parameter
uint64_t t0 = __rdtscp(&junk); // get current time stamp
// execute the operation to time
someOperation();
uint64_t delta = __rdtscp(&junk) - t0; // delta is the duration
```

- The `delta` is measured in CPU cycles
- This method will make it possible to differentiate cache hits and misses

Step 1: Time Difference between Cache Hits and Misses

- Write a simple program `diff.c` to measure the difference
  - Define a variable
  - Flush its content from the cache
  - Access it and measure time (you are measuring a miss)
  - Access it again and measure time (you are measuring a hit, now)
- Hint:
  - When an OS assigns a virtual page for a variable, many OSes map it to a physical page full of zeros, shared by all unitialized virtual pages
  - Only on the first write to an address in the virtual page, it detects a write on a shared physical page, copies the content to a new physical page, assigns the new physical page to the virtual page, and finally performs the write (copy-on-write policy)
  - Since you do not want to measure all the above, remember to write something in the variable before flushing the cache and thus avoid any issues

Time Difference for Cache Hit and Miss

- Try to compile the program and run it
  - Here the example uses `clang` as the C compiler and `Slide` is the current working directory:

```
➜  Slide clang diff.c -o diff
➜  Slide ./diff
Miss: 841, Hit: 145
➜  Slide
```

- The output could vary depending on your own machine
- Run it many times to see if the output is reasonably stable
- Based on the output, choose a threshold to distinguish hits from misses

Step 2: Attack the Victim

- Use the provided attack.c file
- Write the code for a Flush+Reload attack on the victim:
    1. Flush `LUT` from cache
    2. Call the victim (it will access an unknown entry which will reveal the secret)
    3. Access each element of `LUT` and measure time (hits vs. misses)
    4. Repeat 1-3 several times (tens or hundreds?) and record which accesses were hits
    5. Find the most frequently detected location (i.e., the most likely correct `index`)
    6. Compare with the correct answer
- Use the provided file as a template and follow the guidance there

Possible (Good) Result

```
➜  attack1 clang attack.c -o attack && ./attack
Attack index: 94, Correct index: 94
Attack index: 73, Correct index: 73
Attack index: 172, Correct index: 172
```

If It Does Not Work for You...

- Attacks may not always work on the first attempt…

```
➜  attack1 clang attack.c -o attack && ./attack
Attack index: 5, Correct index: 94
Attack index: 5, Correct index: 73
Attack index: 2, Correct index: 172
```

- Try to print the hit count for each possible `index`

```
0: 0    1: 0    2: 98   3: 100  4: 99   5: 100  6: 99   7: 100
8: 100  9: 100  10: 100 11: 100 12: 100 13: 100 14: 100 15: 100
16: 100 17: 100 18: 100 19: 100 20: 100 21: 100 22: 100 23: 99
24: 99  25: 99  26: 100 27: 100 28: 100 29: 100 30: 100 31: 100
```

- This may give some ideas of what is not working…

A Typical Issue You May Observe

- The program always gives a small guess for `index`

```
➜  attack1 clang attack.c -o attack && ./attack
Attack index: 5, Correct index: 94
Attack index: 5, Correct index: 73
Attack index: 2, Correct index: 172
```

- And after a few positions, all `index` values are counted as hits

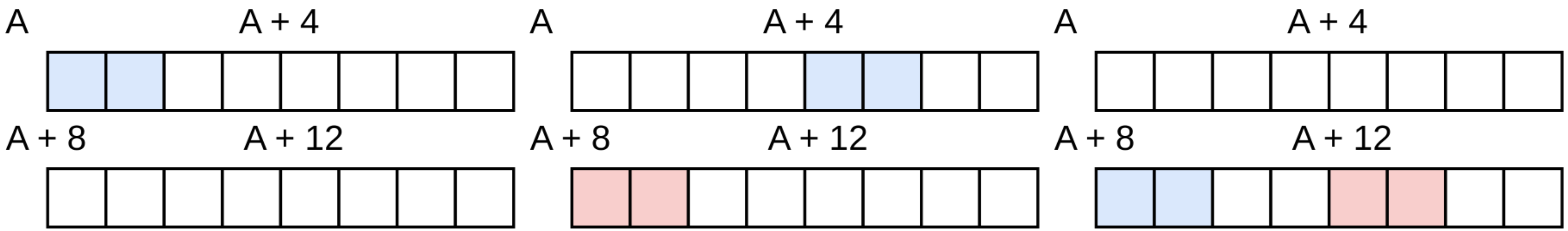
```
0: 0     1: 0     2: 98    3: 100   4: 99    5: 100   6: 99    7: 100
8: 100   9: 100   10: 100 11: 100 12: 100 13: 100 14: 100 15: 100
16: 100 17: 100 18: 100 19: 100 20: 100 21: 100 22: 100 23: 99
24: 99   25: 99   26: 100 27: 100 28: 100 29: 100 30: 100 31: 100
```

- What is happening?!

What Is Wrong? Data Prefetching

- Modern CPUs employ data prefetching to improve performance

Time	0	1	2
Pipeline	Access A	Access A + 4	Access A + 8
Prefetcher	Nothing	Bring A+8 into Cache	Bring A+12 into Cache



- Prefetchers try to learn simple access patterns, such as a sequential scan of an array
- Do we access LUT sequentially?

Where Do We Trigger Prefetching?

- The LUT is accessed only in two places
  - In victim function: we only access the LUT once, so no prefetching
  - In the measurement?
    - How do we measure the access time of all the blocks in LUT?
      - Enumerate all the blocks and try to access each one by one
    - What order do you use to access each block? Sequential? The prefetcher might kick in…
- How to avoid the prefetcher to screw up our accesses?
  - Shuffle the access order for time measurement to confuse the prefetcher

Step 3 (if needed): Shuffle the Access Order

- We need a nonlinear and exact 1 to 1 mapping
- Possible solution: generate a table from 0 to 255, shuffle it, and use it to translate sequential addresses into randomized ones
- python3 code to generate a header file:

```
import random

map_table = list(range(0, 256))
random.shuffle(map_table)

with open("shuffle_map.h","w") as f:
  f.write("#pragma once \n")
  f.write("const int forward[256] = {")
  f.write(",".join(map(str, map_table)))
  f.write("};\n")
```

- The provided file `shuffle_map.h` is the output of the above script