Session 7: Caches and Microarchitectural Timing Attacks

1 Introduction

The goal of this session is to provide an in-depth look at one very important microarchitectural optimization technique, namely caching. We will introduce the technicalities of cache organization in modern processors by mainly studying them from an attacker's perspective.

As processor speed has been growing much more rapidly than memory speed over the past decades, caches have proven to be indispensable to maintain overall performance in modern processors. However, the increased use of caching has also given rise to an entirely new class of microarchitectural timing side-channel attacks. These attacks assume that a spy and a victim program executing on the same machine may be conceptually isolated (e.g., as enforced through address space separation by the operating system, or classes in the Java Virtual Machine), but still share the same underlying CPU cache. From a high level, memory accesses performed by a victim program may unintendedly replace cache blocks brought into the cache earlier by a spy program. When this is the case, the spy program will be slowed down accordingly. As such, the measurable timing differences for memory accesses that hit or miss the cache, can be abused by a carefully crafted spy program to infer secret-dependent memory accesses in a victim program.

In the first section, we develop some hands-on experience with microarchitectural timing measurements on a modern Intel x86 processor, by attacking a naive password comparison program through a timing side-channel. Next, we develop an elementary Flush+Reload cache time attack to investigate side-channel information leakage from the CPU cache. Finally, we move on to more advanced and realistic Prime+Probe cache attacks, which require a detailed understanding of the processor's underlying cache organization.

Note. Since caching (or any notion of microarchitectural optimizations for that matter) are not implemented by the RARS simulator, the exercises below require pen and paper, plus a recent Intel x86 processor for some hands-on timing experiments.

If you are using `gcc` on a Windows machine with MinGW, make sure to use the 64-bit version available at `https://mingw-w64.org/`. The exercises will not work as expected on regular MinGW. If you install MinGW-64, make sure to modify your PATH variable so that the bin folder of MinGW64 is found before the old MinGW installation (or simply uninstall the 32-bit MinGW).

2 A Hands-On Guide to Execution Timing Attacks

As a warm-up exercise, we will start by explaining the concept of a timing side-channel by attacking a rudimentary example program. The program compares a user-provided input against an unknown secret PIN code to decide access. Your task is to infer the secret PIN code, without modifying any of the code. For this, you will have to cleverly provide inputs to the program, and carefully observe the associated execution timings being printed.

Exercise 2.1. *Download the* `passwd.c` *plus corresponding* `secret.h` *and* `cacheutils.h` *helper files from*

```c
int check_pwd(char *user, int user_len, char *secret, int secret_len)
{
    int i;

    /* reject if incorrect length */
    if (user_len != secret_len)
        return 0;

    /* reject on first byte mismatch */
    for (i=0; i < user_len; i++)
    {
        if (user[i] != secret[i])
            return 0;
    }

    /* user password passed all the tests */
    return 1;
}
```

Figure 1: Password comparison function vulnerable to a timing side-channel.

Toledo, place them all in the same directory, and compile the application using "`gcc passwd.c -o passwd`".[1] *Try to understand what the program is doing by examining its source code (`passwd.c`). However, make sure to not yet open the `secret.h` file at this point, or you'll miss out on all the fun! ;-)*

After running and/or examining the program, you will have noticed that the only way to get access is to provide the unknown PIN code. However, besides printing an "access denied" message, the program also prints a timing measurement. More specifically, it prints the amount of CPU cycles needed to execute the `check_pwd` function in Fig. 1 (expressed as the median over 100,000 repeated runs).[2]

Exercise 2.2. *Explain why the execution timings being printed are not not always exactly the same when repeatedly providing the exact same input. Why is it a good idea to print the median instead of the average?*

Solution: Execution time non-determinism in modern processors is caused by a wide range of microarchitectural optimizations including (instruction+data) caching, pipelining, branch prediction, dynamic frequency scaling, etc. Ultimately, a single execution timing measurement may be unreliable, and it's better to aggregate over multiple measurements. We compute the median, since a single outlier (e.g., due to an operating system context switch or interrupt) may strongly affect the average.

The `check_pwd` function performs the actual password comparison, and only returns 1 if the password string pointed to by the `user` argument exactly matches a `secret` string. Otherwise a return value of zero is returned. While this is clearly functionally correct behavior, `check_pwd` does not always execute the same instructions for every input.

Exercise 2.3. *Can you identify a timing side-channel in Fig. 1? Keep in mind that you only control the `user` and `user_len` arguments (by providing inputs to the program), while `secret` and `secret_len` remain fixed unknown values. Hint: Try to come up with a way to iteratively provide inputs and learn something useful from the associated timings. First infer `secret_len`, before finally inferring all the `secret` bytes. You can assume the secret PIN code uses only numeric digits (0-9).*

Solution: The program early-outs when providing an incorrect password length. Hence, a password of the correct

[1] This program can only be executed on a (recent) Intel/AMD x86 processor, which is most likely what's in your laptop (if you don't get any error messages).

[2] The code in Fig. 1 is somewhat simplified. The real C program includes additional dummy `delay` function calls to artificially delay the program, with the purpose of amplifying the timing channel for educational purposes.

length will take slightly longer (even if the individual are still not correct). Once the correct password length has been established, every correct byte increases the execution time (extra for loop iteration). Hence, timing can be used to brute-force *individual* PIN digits one byte at a time.

See for example the following input sequence (as measured on an Intel i7-6500U CPU @ 2.50GHz):

```
$ gcc passwd.c -o passwd && ./passwd
Enter super secret password ('q' to exit): 1
    time (med clock cycles): 110
Enter super secret password ('q' to exit): 11
    time (med clock cycles): 108
Enter super secret password ('q' to exit): 111
    time (med clock cycles): 586
Enter super secret password ('q' to exit): 211
    time (med clock cycles): 594
Enter super secret password ('q' to exit): 311
    time (med clock cycles): 590
Enter super secret password ('q' to exit): 411
    time (med clock cycles): 584
Enter super secret password ('q' to exit): 511
    time (med clock cycles): 1072
Enter super secret password ('q' to exit): 521
    time (med clock cycles): 1582
Enter super secret password ('q' to exit): 531
    time (med clock cycles): 1068
Enter super secret password ('q' to exit): 521
    time (med clock cycles): 1572
Enter super secret password ('q' to exit): 522
    time (med clock cycles): 1608
Enter super secret password ('q' to exit): 523
    time (med clock cycles): 1578
Enter super secret password ('q' to exit): 524
    time (med clock cycles): 2028
Enter super secret password ('q' to exit): 524
    time (med clock cycles): 2030
Enter super secret password ('q' to exit): q
```

---

3 Cache Timing Attacks on Shared Memory (FLUSH+RELOAD)

The above execution timing attack example measured an obvious timing difference caused by early-outing a function (i.e., executing more vs. less instructions before returning). Most real-world programs do not exhibit such explicit input-dependent timing dependencies, however, and we need another way to extract information. We will therefore abuse subtle timing differences caused by the underlying CPU cache to infer memory accesses by the victim.

If the attacker and the victim share some memory locations (e.g., as is often the case for shared library code), we can rely on the seminal FLUSH+RELOAD attack technique. This technique is conceptually very simple, but has been proven to be extremely powerful (e.g., FLUSH+RELOAD has recently been applied for instance as an important building block for the high-impact Meltdown, Spectre, and Foreshadow attacks). FLUSH+RELOAD proceeds in 3 distinct phases to determine whether a victim program accessed some shared data A:

1. **Flush:** The attacker first initializes the shared CPU cache in a known state by explicitly flushing the address A from the cache. This ensures that any subsequent access to A will suffer a cache miss, and will hence be brought back into the cache from memory.

```
void secret_vote(char candidate)
{
    if (candidate == 'a')
        votes_a++;
    else
        votes_b++;
}
```

Figure 2: Voting function with secret-dependent data accesses.

2. **Victim execution:** Now the attacker simply waits for the victim to execute. If a certain secret is true, the victim will load some data A into the cache. If the secret is false, however, the victim loads some other data B into the cache.
3. **Reload:** After completing the victim execution, the attacker measures the amount of time (CPU clock cycles) it takes to reload the shared data A. A fast access reveals that the victim execution in step 2 above brought A into the cache (i.e., cache hit, secret = true), whereas a slow access indicates that A was not accessed during the victim's execition (i.e., cache miss, secret = false).

Exercise 3.1. *Download the* `flush-and-reload.c` *plus corresponding* `cacheutils.h` *helper files from Toledo, place them all in the same directory, and compile the application using "*`gcc flush-and-reload.c -o fnr`*".*

Exercise 3.2. *Consider the example victim program in Fig. 2. Describe a way to mount a successful* Flush+Reload *attack to reconstruct the secret vote, solely through timing differences induced by the CPU cache.*

Solution:

1. Flush `votes_a` from the CPU cache.
2. Execute `secret_vote` with unknown input.
3. Reload `votes_a`: low access time → cache hit → victim voted for candidate A.

Exercise 3.3. *Edit the* `main` *function in* `flush-and-reload.c` *to implement the missing "flush" and "reload" attacker phases. You can use respectively the provided* `void flush(void *adrs)` *and* `int reload(void *adrs)` *functions. The latter returns the CPU cycle timing difference needed for the reload. If necessary, compensate for timing noise from modern processor optimizations by repeating the* Flush+Reload *experiment (steps 1-3 above) a sufficient amount of times and taking the median or average.*

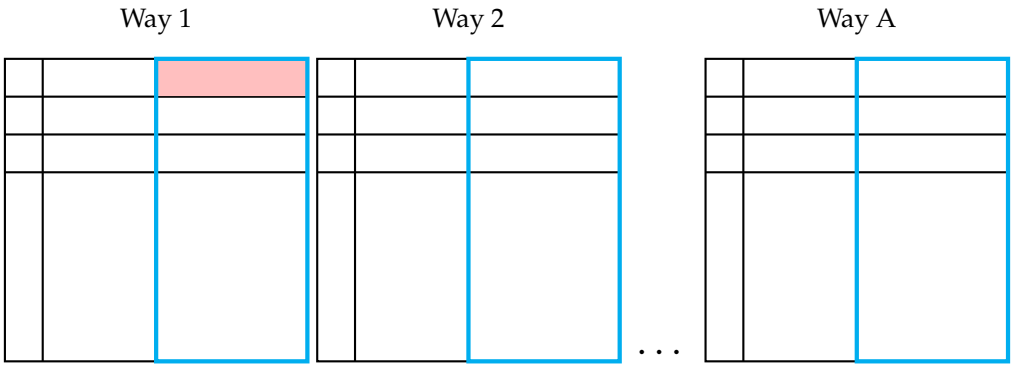
Solution:

```
int main()
{
    int i, tsc;

    for (i=0; i < NUM_SAMPLES; i++)
    {
        /* 1. Flush */
        flush(&votes_a);
        flush(&votes_b);

        /* 2. Victim exec */
        secret_vote('b');

        /* 3. Reload */
        tsc = reload(&votes_a);
        printf("Time votes_a (CPU cycles): %d\n", tsc);
        tsc = reload(&votes_b);
```



The pink box represents one block (2^b = B bytes). The sum of the blue boxes is the cache data size (S bytes).

Figure 3: The cache

```
18          printf("Time votes_b (CPU cycles): %d\n", tsc);
19          printf("-------------------------\n");
20      }
21
22      return 0;
23  }
```

Example output:

```
Time votes_a (CPU cycles): 324
Time votes_b (CPU cycles): 96
-------------------------
Time votes_a (CPU cycles): 312
Time votes_b (CPU cycles): 52
-------------------------
Time votes_a (CPU cycles): 316
Time votes_b (CPU cycles): 52
-------------------------
Time votes_a (CPU cycles): 316
Time votes_b (CPU cycles): 52
-------------------------
Time votes_a (CPU cycles): 316
Time votes_b (CPU cycles): 48
-------------------------
```

4 Cache Organization Concepts

The above Flush+Reload attack is relatively easy to understand. Since it only relies on a measurable timing difference for memory accesses that miss the cache, without requiring any knowledge of internal cache organization. A more powerful class of advanced cache attacks, on the other hand, requires a detailed understanding of cache mapping schemes, address bits, and replacement policies. The exercises in this section therefore aim to build up such understanding, before moving to the Prime+Probe attacks in the next section.

Exercise 4.1. *Suppose a computer's address size is k bits (using byte addressing), the cache data size is S bytes, the block size is B bytes, and the cache is A-way set-associative. Assume that B is a power of two, so B=2^b. Figure out what the following quantities are in terms of S, B, A, b and k:*

- *the number of sets in the cache:*
- *the number of index bits in the address;*
- *the number of bits to implement the cache.*

Solution:

- the number of sets in the cache = S/(B*A) because each set contains B*A bytes;
- the number of index bits in the address = $\log_2(S/(B*A))$;
- the number of bits to implement the cache = S/B blocks * (1 valid bit + (k - $\log_2(S/(B*A))$ - b) tag bits + 8B data bits per block). Note: this includes the cache data!

Exercise 4.2. *Consider a processor with a 32 bits addressing mode and a direct mapped cache, with 8-word block size and a total size of 1024 blocks. Calculate how many bits of the address are used to tag a block, to select the block, a word in the block, a byte in the word. The sum of these numbers must be 32! Calculate the total size needed to implement this cache.*

Solution: We have 1024 blocks so we need 10 index bits. We also need 3 bits to select the correct word and 2 bits to select the byte. We then need the remainder of the bits as tag, this is 32-10-3-2=17. The total size of the cache is then 1024 * (1 valid bit + 17 tag bits + 8 32-bit words) = 280 576 bits.

Solution: The strength of the 1-word block cache is that it has 16 separate blocks. Although the set-associative has just as many blocks, each block has twice as many potential memory addresses mapped on it. If we use a FIFO replacement, a sequence of 8,16,0 would insert and remove 8 out of a set-associative cache. The direct mapped cache still has the 8.

Exercise 4.3. *Here is a series of address references given as word addresses: 2, 3, 11, 16, 21, 13, 64, 48, 19, 11, 3, 22, 4, 27, 11. Label each reference in the list as a hit or a miss and show the final content of the cache, assuming:*

- *direct-mapped cache with 16 1-word blocks that is initially empty;*
- *direct-mapped cache with 4-word block size and total size of 16 words;*
- *2-way set-associative cache, 2-word block size. Total size of 16 words.*

Note: Table 4.3 shows an example format of a possible solution.

Exercise 4.4. *Associativity usually improves the hit ratio, but not always. Consider a direct mapped cache with 16 1-word blocks and a 2-way set-associative cache with 1-word block size and a total size of 16 words. Find a sequence of memory references for which the associative cache experiences more misses than the direct mapped cache.*

Solution:

First solution: one hit at the second reference to 11.

Second solution: 2, 3 hit, 11, 16, 21, 13, 64, 48, 19, 11 hit, 3, 22 hit, 4, 27, 11.

Third solution: 2, 3 hit, 11, 16, 21, 13, 64, 48, 19, 11 hit, 3, 22, 4, 27, 11

Block	Word 1	Word 2
0	/	/
1	2	19
2	/	/
3	/	/
4	/	/
5	/	/
6	/	/
7	/	/
8	/	/

Table 1: Example solution for exercise 4.3 (direct mapped cache with 8 2-word blocks)

Block	Word Address
0	48
1	/
2	2
3	3
4	4
5	21
6	22
7	/
8	/
9	/
10	/
11	11
12	/
13	13
14	/
15	/

Table 2: Solution for 1-word direct mapped cache

Block	Word 0	Word 1	Word 2	Word 3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Table 3: Solution for 4-word direct mapped cache

Set	Block	Word 0	Word 1
0	0	48	49
	1	64	65
1	2	26	27
	3	10	11
2	4	4	5
	5	12	13
3	6	22	23
	7	/	/

Table 4: Solution for 2-way set-associative cache

5 (Optional) Cache Timing Attacks Across Protection Domains (PRIME+PROBE)

A crucial limitation of the aforementioned FLUSH+RELOAD technique, is that the attacker and the victim should share memory. In a more realistic setting, the operating system ensures that the attacker and the victim can each only access their own non-overlapping part of the address space. However, even if the attacker cannot directly access a victim's memory locations, she can still abuse timing differences caused by the underlying shard CPU cache. After obtaining a detailed understanding of the CPU cache's internal mapping scheme, the attacker again proceeds in 3 distinct phases to determine whether a victim program accessed some private data A:

1. **Prime:** The attacker first initializes the cache in a known state, this time by loading one or more carefully chosen attacker-controlled memory locations C into the shared CPU cache. More specifically, the addresses C loaded during the "priming" phase should be such that they map to the same cache set as the private victim address A.
2. **Victim execution:** Now the attacker simply waits for the victim to execute. If a certain secret is true, the victim will load some data A into the cache. If the secret is false, however, the victim loads some other data B into the cache.
3. **Probe:** After completing the victim execution, the attacker measures the amount of time (CPU clock cycles) it takes to reload the addresses C that were brought into the cache during the priming phase. A slow access now reveals that the victim execution in step 2 above replaced the attacker data C when loading A into the cache (i.e., cache miss, secret = true). A fast access on the other hand, indicates that A was not accessed during the victim's execution (i.e., cache hit, secret = false).

Exercise 5.1. *Is the above* PRIME+PROBE *technique applicable for each of the cache mapping schemes you know: (1) direct mapping, (2) N-way set associative, (3) fully associative with LRU replacement policy? Explain why (not). If applicable, describe a high-level strategy to replace cached victim memory locations by constructing an attacker-controlled eviction set during the "prime" phase. What is the granularity at which an attacker can see victim accesses during the "probe" phase?*

Solution:

1. **Direct mapping:** PRIME+PROBE is applicable, since multiple addresses in the main memory map to the exact same cache location. It suffices for the attacker to access a *single address with the same cache index address bits* in order to replace a cached victim memory address during the "prime" phase. The granularity at which the attacker can later see victim accesses during the "probe" phase, is equal to the cache block size. That is, PRIME+PROBE cannot distinguish multiple victim accessses *within* the same cache block.
2. **N-way set associative:** PRIME+PROBE is applicable, since multiple addresses in the main memory map to the same cache set. However, in order to make sure that a cached victim memory address is replaced during the "prime" phase, the attacker now has to access *N addresses with the same cache index address bits* (assuming a least-recently used cache replacement policy). The best-case granularity at which the attacker can later see victim accesses during the "probe" phase, is equal to the cache block size. That is, PRIME+PROBE cannot distinguish multiple victim accesses *within* the same cache set.
3. **Fully associative:** PRIME+PROBE is not applicable, since all addresses map to a *single* set. As such, there is no secret-dependent replacement that can be measured during the "probe" phase. Even if the attacker fully fills the cache during the "prime" phase, she can only learn afterwards that the least-recently used attacker entries were replaced by the victim. In other words, the attacker can only learn how many, but not *which* victim entries exactly where loaded into the cache.

Exercise 5.2. *Consider a processor with a 32 bits addressing mode and a direct mapped cache, with a 1-word block size and a total size of 16 blocks. First calculate index and tag address bits, as in Exercise 4.2 above. Say that all memory addresses in the range* $[0, 100[$ *belong to the victim, whereas the attacker owns addresses in the range* $\left[100, 2^{32}\right[$*. The victim performs a series of address references given as word addresses: either (2, 3, 11, 16, 21, 13, 64, 48, 19, 11, 3, 22, 4, 27, 11) if secret=1, or (2, 3, 11, 16, 21, 13, 64, 48, 19, 11, 70, 36, 7, 91, 75) if secret=0.*

Construct a memory access sequence to be performed by an attacker during the "prime" phase, in order to learn the victim's secret in the "probe" phase later on.

Solution: 32-bit address divided as follows: (26 tag bits), (4 index bits), (2 data bits within word)

Victim cache usage is as follows (highlighted secret dependent access sequence for *secret=1* vs. *secret=0*):

Block	Word 0 (most recent right)
0	~~16, 64,~~ 48
1	/
2	2
3	~~19,~~ *3*
4	*4* or *36*
5	21
6	*22* or *70*
7	*7*
8	/
9	/
10	/
11	~~11,~~ (*27, 11*) or (*91, 75*)
12	/
13	13
14	/
15	/

Now observe that cache block 7 is *only* accessed by the victim if secret=0. It thus suffices to load an attacker-controlled memory location that maps to cache block 7 during the "prime" phase (e.g., word address 135), and reload that location in the "probe" phase after the vicim's execution. A long access time (i.e., cache miss) reveals that the attacker's memory location in cache block 7 was replaced by the victim, and hence secret=0.

Exercise 5.3. *Repeat the previous exercise for a processor with a 32-bit addressing mode and a 2-way set-associative cache, with 2-word block size and a total size of 16 words. You can assume a least-recently used cache replacement policy.*

Solution: 32-bit address divided as follows: (27 tag bits), (2 index bits to select set), (1 bit to select word within set), (2 data bits within word)

Victim cache usage is as follows (highlighted secret dependent access sequence for *secret=1* vs. *secret=0*):

Set	Block	Word 0 | Word 1 (most recent right)
0	0	~~16 | 17~~, 48 | 49
	1	64 | 65
1	2	~~2 | 3~~, ~~18 | 19~~, (~~2 | 3~~, *10 | 11*) or *90 | 91*
	3	~~10 | 11~~, *26 | 27* or *74 | 75*
2	4	~~20 | 21~~, *4 | 5* or *36 | 37*
	5	12 | 13
3	6	*22 | 23* or *70 | 71*
	7	*6 | 7*

Now observe that cache set 3 has a higher pressure when secret=0 (two replacements when accessing words 70 and 7) vs. when secret=1 (only a single replacement when accessing word 22). It thus sufficices to completely fill cache set 3 with attacker-controlled words (e.g., word addresses 126 and 134) during the "prime" phase. After executing the victim, the attacker reloads both word 126 and 134 in the "probe" phase. Only when *both* accesses are slow (i.e., cache miss), the victim replaced two cache blocks in cache set 3, and hence secret=0.
