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 MARS 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 **??** (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?*

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* **??***? 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).*

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

[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 **??** 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.

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

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

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.
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* **??**. *Describe a way to mount a successful* FLUSH+RELOAD *attack to reconstruct the secret vote, solely through timing differences induced by the CPU cache?*

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.*

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.*

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.*

Exercise 4.3. *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.*

5 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?*

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* **??** *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.

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.*