Cache attacks

Research Project

Sarpant - Hubert - Poupi - Maëlle - SanSa

May 2, 2023

Contents

Abstract

Our research project focuses on two side-attacks of the cache memory. They are Flush & Reload and Prefetch & Reload. The aim of our project is to implement and compare those attacks in order to determine which is the most efficient.

1 Introduction

Today, we know the existence of multiple ways to perform side-channel attacks. Trying to exploit the cache memory is one of them. Indeed, there are many ways to inspect the cache memory (located in the CPU) and then, extract information from it. Multiple attacks today are already well known like Flush & Reload, Prime & Probe or more recently Prefetch & Reload. Those techniques are used to retrieve sensible information like encryption keys, passwords, by monitoring the accesses to the cache memory.

Cache side-channel attacks on cache memory became a major concern in term of computer security, as they allow attackers to retrieve sensible information by monitoring memory accesses. Also those same techniques are used to see right through the functioning of malwares, like ransomwares, in order to counter them. Those kinds of attacks use vulnerabilities in the conception and the implementation of modern computer systems, notably computers and smartphones. In that context it is important to compare and understand the differences between those attacks to determine their efficiency and the potential impact they can have on softwares.

Thus our work focused on two attacks in order to compare their efficiency : Flush & Reload[2] and Prefetch & Reload[3]. This research aim to provide a comparative analysis of both of those attacks on the cache memory, by highlighting their advantages and drawbacks, in order to better understand the security risks. We will first review the different research articles that we had to analyze so that we could achieve this research. We will review the way the cache memory works, while putting in evidence the vulnerabilities exploited by each attack. For each we will explain their principle then in what manner we implemented them. Finally, each implementation will go through a series of performance tests thanks to our own benchmark to precisely assess their efficiency.

In order to better manipulate memory accesses, cache memory and different threads, all attacks and benchmarks have been implemented in C.

2 Literature review

2.1 Timing attacks[1]

This article helped us to introduce the subject of timing attacks. Due to performance optimizations, the computation of encryption operations does not run in constant time. This timing delta can be used to determine values such as the encryption keys.

The attack consists of measuring the timing variation of each multiplication and squaring operation and their error rate. The results shown in the paper were made on a RSA toolkit named RSAREF. With a statistical resolution, it is possible to recover a private key.

In order to prevent the attack, you can use constant time operations but it is not perfect. If you use a timer to add delay, the power consumption and CPU usage can still leak information about the state of computation. There are more efficient ways to do it, like adapting the blinding of signatures. This technique is not perfect. It can still leak data with the timing.

2.2 Flush & Reload[2]

The paper introduced a technique for a cache side-channel attack that targets the last level cache L3. They demonstrate the effectiveness of this technique against cryptographic implementations such as the RSA GnuPG implementation. This attack exploits a weakness in the Intel X86 processors, and works by monitoring the timing from retrieving data from memory or cache levels and exploiting the differences in them. By using this attack during the encryption process they could extract the secret key used for encryption and decrypt the encrypted message. This paper's finding represent a significant threat to computer security and shows there is a need to develop countermeasures.

When multiple processes run on the same processor, they can share pages and cache memory, which reduces memory usage but can result in information leakage. Cache-based side-channel attacks, specifically timing-based cache attacks, exploit this shared cache to extract sensitive information from a victim's system. The paper discusses the attack technique called Flush+Reload and demonstrate how it can recover RSA private keys from the GnuPG implementation. The attack works in three phases, involving flushing a monitored memory line from the cache hierarchy, waiting for the victim to access it, and reloading it to measure the time taken. The authors explain that their implementation relies on the cflush command and can have false positives due to

processor optimizations. Overall, the technique is more powerful than prior micro-architectural side-channel attacks because it focuses on the Last Level Cache (LLC) and identifies access to specific memory lines. The authors tested the attack on the RSA implementation of GnuPG in both same-OS and cross-VM scenarios.
They describe their methodology to perform the attack, involving the Flush+Reload technique to monitor the cache activity during the encryption process. They witnessed speculative execution on both the HP and Dell machines on which they tested their attack. They demonstrated that thanks to the cache side-channel information they can recover the secret key used for encryption which allows them to decrypt the encrypted message.

In terms of implications in computer security the paper's findings represent a real immediate threat. There is a need to prevent this kind of attack by elaborating countermeasures. The lack of permission checks considering the cflush instruction is a weakness, a solution would be to limit its power. The authors suggest restricting its use to memory pages to which the process has write access and to which the system allows its access. The Flush+Reload technique is not applicable on the ARM architecture as the instructions to evict cache lines are restricted to when the processor is in an elevated privilege mode. It does not work also on contemporary AMD processors, maybe because their caches are not inclusive, meaning that evicting data from the LLC does not evict it from other cores. Those countermeasures are hardware-based and even if effective they do not provide an immediate solution as developing them will take time and will not protect existing hardware. They suggest to develop software-based solutions to provide an immediate solution to the problem, like preventing page sharing between processes. But this solution is unachievable as it would mean an increase in the memory requirements of modern operating systems. Another suggested solution would be introducing noise to clock measurements, but its limitation is that the attacker can still use other methods to generate high resolution clocks. The GnuPG teams has since released new versions of their product and their implementation actually successfully mitigate the Flush+Reload attack. To conclude this paper present an effective Flush+Reload attack that represents a clear threat in computer security. They demonstrated how to use this attack to retrieve private keys on GnuPG cryptographic implementation. Their technique exploits the page sharing principle and the lack of restrictions on the use of the cflush instruction, thus exposing a high security weakness of the Intel implementation of the X86 architecture.

2.3 Prefetch & Reload[3]

We have learned from this paper several things about the side channel attack prefetch+reload, the reason of the flaw and how the attack works. To begin with, it is explained that most modern x86 processors have prefetch instructions which allow performance improvement but can lead to security problems which are here due to the PREFETCHW instruction.

The authors provided evidence about this and then managed to implement 3 cache attacks, Prefetch+Load, Prefetch+Reload and Prefetch+Prefetch that exploit these flaws. In this review, we will focus on the Prefetch + Reload attack.

In order to understand what the flaws of the PREFETCHW instruction are based on, the authors remind us how the MESI coherency protocol works (it's explained again in this report in the "Reminders" section). With the MESI protocol we will observe two things that can happen during a memory request coming from the CPU, a change of coherence attribute for the requested cache line and a time difference in the data recovery depending on the coherence attribute which is one of the causes of the PREFETCHW flaws.

It is indicated that this generates an important flaw in the PREFETCHW instruction due to the characteristic of the MESI protocol, because the time difference mentioned before is the core of the prefetch+reload attack. This is simply explained by the fact that a change of attribute can happen during a write or read request depending on the attributes that the concerned cache line has.

They take the example of a CPU core (core 1) which would like to read a cache line which is both present in the LLC and in the private cache of another core (core 0) whose attribute is Modified (M), since the private cache of core 0 has been modified, we must first update the LLC which has become "stale" before being able to read its content, to do this the LLC will retrieve the cache line contained in the private cache of core 1, change the coherence attribute of this private cache line from Modified to Shared (S) and replace its "stale" content by the new line it has just retrieved. It will then be able to send its new content back to core 0 which will also have the Shared attribute. Another case would be if a CPU core (core 0) wanted to write a new cache line in its private cache but this line has the Shared attribute, then this private cache must first send a request to the LLC to have a write permission, this will have the effect that the LLC will send a signal to all the private caches having the same cache line in order to make them Invalid (I), then the attribute of the cache line of the core 0 will change from Shared to Modified so that it can be updated, it will not be automatically copied in the other private caches.

As we can imagine, a request to read a cache line with a Modified attribute will take longer than if it were Shared, because the time to fetch data from another private cache as well as to modify the coherence attributes of the concerned private caches makes the whole thing slower than if the line of our private cache was Shared from the beginning.

There is another problem with the PREFETCHW instruction, as it has been stated before, when we want to modify a cache line with the Shared attribute, it invalidates all the other cache lines sharing the old data, except that it is also possible to achieve the same result by simply reading the content of the Shared data, the authors have demonstrated this through an experiment with 2 threads that execute in turn a read on a target Shared data, this means that the write verifications are not done (for Intel processors) which leads to security flaws and which allowed attack by prefetch.

Prefetch+Reload is an attack that allows to reveal the access pattern of the shared cache of a victim and to deduce some secrets. To do this the attacker will first have to prefetch the Shared cache line in order to pre-set the coherence argument, which means that he will be the owner of this cache line and will at the same time put the other cache lines sharing the same data in the Invalid (I) state, we then wait for the victim to access the data and then in another thread we will reload the data in order to see if it was indeed accessed by the victim. The details about the working principle are explained later in our report in the part treating the prefetch+reload.

3 Reminders

Here are some essential reminders to understand later parts.

3.1 Cache memory

Cache memory is located inside a CPU (central processing unit) and is faster than the RAM. It stores useful data at a time in order to access it faster during operations.

Cache memory is split in different levels. Levels **L1** and **L2** are "privates". They are only available for their CPU core. The level **L3** or **LLC** (last-level cache) is shared between the different cores. This level allows to store, for example, a shared data between many processes only once.

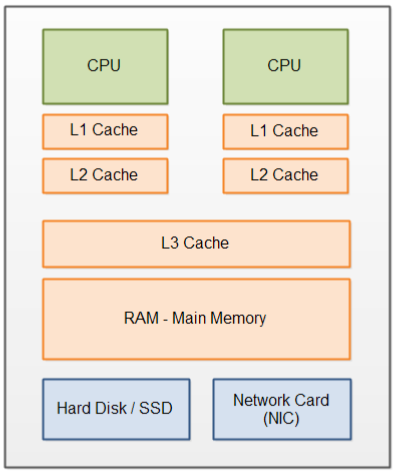

Figure 1: Modern Cache Architecture [4]

Since a memory line can be used by several processes and therefore several cores, it is essential to check the consistency of the data in the various caches. For this matter, modern processors use a protocol named `MESI (Modified, Exclusive, Shared, Invalid)`. The latter attributes one of the following 4 states to a memory block present in the cache.

- `Modified` : the block is present in the cache and has been modified. Other caches must be notified of this change to maintain data consistency.
- `Exclusive` : the block is present in a single cache, but has not been modified. Other caches cannot access this block until the owner releases it.
- `Shared` : the block is present in several caches and has not been modified. Data can be shared between processors.

- `Invalid` : the block is not present in the cache.

When a processor wishes to access a block of memory, the MESI protocol determines the state of the block in the other caches. If the block is marked as modified, the processor must first update the other caches to maintain consistency. If the block is marked as exclusive or shared, the processor can access it directly. If the block is marked as invalid, the processor must retrieve the data from main memory.

The MESI protocol thus makes it possible to guarantee that the data stored in the caches is always coherent, which makes it possible to improve the performance of the system by reducing access times to the main memory and avoiding data errors caused by copies not synchronized.

3.2 Side-channel & Timing Attack

Side-channel attacks aim to exploit the environment of a system instead of the system itself. For the case of a encryption algorithm like RSA, it can be its implementation, the CPU on which it is running, etc.

4 Flush & Reload[2]

4.1 Attack's principle

The Flush & Reload attack is an attack that allows an attacker or analyst to deduce the behaviour of a process, by tracing data into the cache. Like previously announced, the cache functions in a way that new data used by the processor are stored in the cache's memory so that they can be used later on. This principle shows the importance to ensure the coherence of the cache, meaning that the data stored in the cache is useful. When it is not the case, then the processor can flush this data from the cache. The space can then be used to receive the next useful data. It is possible as an user to flush (ourselves) a data from the cache memory with the instruction `clflush` (x86). Yet, we reminded that the cache memory was a fast access memory accessible way faster than RAM memory (10 to 100 times faster).

Its from those 2 principles that the attack comes out: Flush & Reload. The principle is the following. We look for a particular data to trace, one that is shared between multiple processes (shared library, file, etc). We then flush it, meaning that we delete this data from the cache memory. At this moment the data is not present in the cache and it will be needed to charge it from the RAM for the next access(es).

The next step consists of waiting for an arbitrary length of time, the goal being to find out if a data was used during this elapsed time. Next we are going to try to access this data while measuring our time access. Thus, thanks to the time needed to access it we can conclude something:

- Either the time was short, which means that the data was replaced inside the cache in the meantime (reused).
- Either the time was long, meaning that no process used the data `flush`.

Algorithm 1 Flush & Reload

```
Require: addr, threshold, rounds
  $i \leftarrow 0$
  $access[rounds]$
  for $i$ in $rounds$ do
    $cflush(addr)$
    $sleep()$
    $delta \leftarrow memaccesstime(addr)$
    $access[i] \leftarrow delta < threshold$
  end for
```

4.2 Implementation

As explained in the reminders, the data we can trace must be shared. For our tests, we chose to trace the accesses to a file.

We used several functions from the `Mastik` library [5]. To retrieve the target file, we use the `map_offset(const char *, uint64_t)` function. This will map the file from an offset and return the address.

To measure memory access time, we use the `memaccesstime(void *)` function. The `rdtscp64()` function returns the number of clock ticks passed. The `delayloop(uint32_t cycles)` function allows you to wait for a certain number of processor cycles.

```
inline uint32_t memaccesstime(void *v) {
  uint32_t rv;z
  asm volatile (
      "mfence\n"
      "lfence\n"
      "rdtscp\n"
      "mov %%eax, %%esi\n"
      "mov (%1), %%eax\n"
      "rdtscp\n"
      "sub %%esi, %%eax\n"
      : "=&a" (rv): "r" (v): "ecx", "edx",
          "esi");
  return rv;
}
```

```
inline uint64_t rdtscp64() {
  uint32_t low, high;
  asm volatile ("rdtscp": "=a" (low), "=d"
      (high) :: "ecx");
  return (((uint64_t)high) << 32) | low;
}

void delayloop(uint32_t cycles) {
  uint64_t start = rdtscp64();
  while ((rdtscp64()-start) < cycles)
    ;
}
```

Figure 2: Code Mastik [5]

Moreover, in order to flush the cache, we defined the following function `cflush(void *)`. The `flush_and_reload` function uses `memaccesstime` to measure the access time to the data. After emptying the data from the cache, the access time is returned.

```
inline void clflush(void *v) {
  asm volatile ("clflush 0(%0)": : "r" (v):);
}
```

Figure 3: Code cflush

```
int64_t flush_and_reload(void *addr) {
  int64_t delta = memaccesstime(addr);
  clflush(addr);
  return delta > UINT16_MAX ? UINT16_MAX :
      delta;
}
```

Figure 4: Code flush_and_reload

The whole is used in a loop, where each loop represents a round. We have chosen to iterate to infinity, with a stop by `SIGINT` signal. The result of the access prediction is not stored but is simply displayed to the attacker. Indeed, keeping these results in memory will not be useful for our benchmark.

Between each measurement, the program waits for 10000 cycles to pass. With the different tests, this value seems to be the most optimal. Thus, each time the `delta` is lower than the threshold, we indicate to the attacker that an access has been measured as well as the delta concerned. It is important to note that the threshold is a value that can vary from one machine to another. Indeed, depending on the processor and several other factors, the access time to the cache and to the RAM can change. For these reasons, our implementation so far does not make this attack portable and requires upstream testing to determine the threshold.

To overcome this problem, we have implemented a calibration function to perform these tests and determine the threshold.

To do this, our function performs several training rounds during which it will `flush` and instantly perform a `memaccesstime()`, to average the access time outside the cache. In the same way, for the same number of rounds, it will simply `memaccesstime()` to obtain the average access time in the cache. With both averages, the function will return the corresponding median.

```
int main (void) {

  ...
  while (1) {

    int64_t delta = flush_and_reload(addr);
    if (delta < threshold)
      printf("%li, access ! \n", delta);

    delaylooop(10000);
  }
  return 0;
}
```

Figure 5: Main function of F&R

```
int64_t fr_probethreshold (void) {
  static char dummy;
  int64_t res_a = 0;
  int64_t res_n = 0;
  for (int i = 0; i < CALIB_TRY; i++) {
    _mm_clflush(&dummy);
    res_a += memaccesstime(&dummy);
  }

  for (int i = 0; i < CALIB_TRY; i++)
    res_n += memaccesstime(&dummy);

  res_a /= CALIB_TRY;
  res_n /= CALIB_TRY;
  return (res_a + res_n) / 2;
}
```

Figure 6: Calibration of the threshold for F&R

We can display the 2 averages in order to have an idea of the difference between an access in the cache or not.

```
avg no access time: 244, avg access time: 32
probe time: 138 delta < probe = access
```

During the calibration, the average access time in the cache is up to 7.6 times faster. The deduced threshold is therefore 138. It is possible to perform simple tests to verify the correct operation of our attack. After launching the attack, opening our target file with `nano` will be enough to trigger our program. If we display all the measured deltas, at the time of a `nano`, we find much lower deltas.

```
delta: 95, access !
delta: 214, no access
delta: 217, no access
delta: 95, access !
delta: 913, no access
delta: 92, access !
```

5 Prefetch & Reload[3]

Contrary to Flush & Reload, the following attack will not determine if a data is in the cache. Indeed, the principle is to determine if the tracked data is in the LLC cache or in a private cache.

5.1 Prefetch

Prefetching is the principle of preloading data into the cache before it is used in order to improve processor performance. The x86 processors offer many software prefetch instructions: `PREFETCHT0`, `PREFETCHT1`, `PREFETCHT2` and `PREFETCHNTA`... They are used to indicate to the processor that a memory location is most likely to be accessed or modified in the near future, the processor then preloads the corresponding data, which speeds up future accesses to this data.

5.2 Attack's principle

The Prefetch & Reload attack is an attack that reveals the access pattern to a victim's shared cache and deduces certain secrets. The attacker controls two threads named `trojan` and `spy`, these threads and the `victim` must all be located on different CPU cores from each other. The attack takes place in 3 steps as follows:

1. The Trojan executes `PREFETCHW` on the targeted cache line, brings it into its private cache and this changes the coherence state of the cache line to `Modified` at the same time as invalidating (`Invalid`) the copies of this cache line in the private caches of the `spy` and the `victim`.
2. The attacker waits for the `victim` to perform an action. If the `victim` accesses the cache line then the state will change from `Modified` to `Shared` and this cache line will be copied to the `victim`'s private cache. The change of coherence state caused by the `victim`'s access cannot be observed by the `trojan`: if the `trojan` later accesses this cache line (`Reload`), it will see a private cache hit regardless of whether the `victim` has performed an access or not because this access does not invalidate the copy of the cache line present in the `trojan`'s private cache.
3. The `spy` already has an invalidated copy thanks to step 1, so when he accesses the cache line he can measure the access time to determine if the state is `Modified` or `Shared`. If the state is `Modified` then the `victim` has not accessed the cache line, if it is `Shared` then it has accessed it.

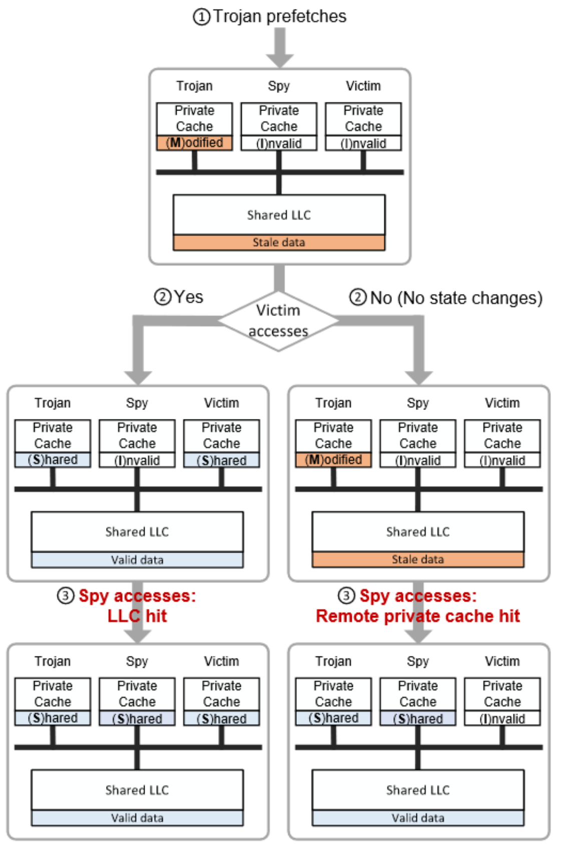

Figure 7: The details of steps for Prefetch and Reload [3]

5.3 Implementation

Here is the pseudo-code proposed by the authors of the article for the Prefetch & Reload attack:

```
void* thread0 (void* addr_d0, int expt_idx){
    for(int i = 0; i < 1000000; i++){
        /* check the experiment index */
        if(expt_idx == 0){
            /* execute prefetchw on d0*/
            prefetchw(addr_d0);}
        /*let thread1 execute 1 iteration*/
        wait_for_thread1();
    }}

void* thread1 (void* addr_d0){
    for(int i = 0; i < 1000000; i++){
        /*let thread0 execute 1 iteration*/
        wait_for_thread0();
        int result = read_and_time(addr_d0);
    }}


int main() {
    /* open and map a file as read-only*/
    int fd = open(FILE_NAME, O_RDONLY);
    int* addr_d0 = mmap(fd, PROT_READ, ...);

    /*pin thread0 on core0 and start thread0*/
    /*pin thread1 on core1 and start thread1*/
    ...
```

Figure 8: Code of prefetch reload by the authors [3]

In order to better compare their attack, we used their pseudo-code to implement the attack. The first version of it worked with heavy processes. The current version uses threads. As explained in the principle of the attack, the attacker launches 2 threads running on 2 different cores. The function `wait_for_threadX()` presented in the pseudo-code was made thanks to semaphores.

```
/* trojan thread. Prefetch line in cache, and
    wait for spy */
static void * trojan (void * addr) {
  memaccesstime(addr);

  for (size_t i = 0; i < TRY; i++) {
    sem_wait(&sem_spy);
    asm volatile("lfence");
    prefetchw(addr);
    sem_post(&sem_trojan);
  }
}
```

Figure 9: Trojan function

```
/* Spy trhead. Wait for trojan, and measure
    timeacces to data */
static void * spy (void * addr) {
  size_t delta;
  for (size_t i = 0; i < TRY; i++) {
    sem_wait(&sem_trojan);
    asm volatile("lfence");
    delta = memaccesstime(addr);
    delta = delta > UINT16_MAX ? UINT16_MAX :
        delta;
    if (delta < threshold)
      printf("Delta: %ld - Access !\n",
          delta);

    sem_post(&sem_spy);
  }
}
```

Figure 10: Spy function

Thanks to semaphores the trojan and spy functions successively perform once their loop instructions were executed. Thus, trojan will prefetch the address to obtain the exclusivity inside the private cache of the core on which it runs, then it hands over to spy. This one will then measure the time access to the data and determine if the data went through the LLC, or it it was still inside the private cache of the core of trojan. Finally, spy steps down in favour of trojan and so on. In order that the attack succeed, it is imperative to execute the trojan, the spy, and the victim on different cores. For that, in the main function, we execute both our functions thanks to threads whose CPU's mask we modify.

```
int main () {

  /* init semaphore */
  /* init threads */

  cpu_set_t cpuset_trojan;
  CPU_ZERO(&cpuset_trojan);
  CPU_SET(0, &cpuset_trojan);
  if (pthread_attr_setaffinity_np(&attr_trojan, sizeof(cpu_set_t),
                                  &cpuset_trojan) != 0) {
    perror("pthread_attr_setaffinity_np");
    exit(EXIT_FAILURE);
  }

  if (pthread_create(&trojan_thread, &attr_trojan, (void *) trojan, NULL) != 0) {
    perror("pthread_create");
    exit(EXIT_FAILURE);
  }

  /* same thing for the spy function */

  pthread_join(trojan_thread, NULL);
  pthread_join(spy_thread, NULL);

  /* sempahore destroy */
```

Figure 11: Execution of spy and trojan function for prefetch reload

To run the victim on a different core, we reused `sched_setaffinity()` directly in the victim program. In real case, this can be controlled at the runtime thanks to `taskset`, or directly during the execution of the program thanks to its PID and `sched_setaffinity()`. In the same way of Flush & Reload, we implemented more or less the same algorithm of calibration to determine the threshold.

6 Benchmark

The simulation of a victim's process is made through a simple program in an infinite loop that will constantly access the traced data with a defined latency. We can vary the latency between each access by passing the number of cycles the program should wait as an argument when we execute the program.

```
/* memory acces to addr */
int func_to_spy(void * addr) {
  while(1) {
      delayloop(cycles_to_wait);
      memaccess(addr);
  }
}
```

Figure 12: Victim program

In order to evaluate our attacks performances we cannot only compare the raw results to the number of accesses the victim made. Each result of an attack is a two-way event. Either the victim used the data, or it has not. Each result can either be true or false. For two events, actually four results are possible for the performance's evaluation: true positive, false positive, true negative, or false negative. Thus, in order to precisely measure the performance, we have to measure those four aspects of precision. Each result predicted by the attack needs to be compared directly with what the victim actually did. For that we use a parent program that knows exactly when the victim makes an access and retrieves also each- result of the evaluated attack. Each access made by the victim will increment a counter and each result verifies the counter's state to determine the validity of the result.

First, in order to control the victim and the attacking program, both are executed using `execve()` funciton call. To retrieve the attacker's results, we simply pipe its standard output. The attacker can then simply write its result to the standard output.

```
/* launch the victim */
pid_t launch_victim(char *envp[], char *
    time_to_wait) {

  pid_t pid = fork();

  if (pid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
  }
  if (pid)
    return pid;

  char *args[] = {"./victim",
      time_to_wait, NULL};
  execve(args[0], args, NULL);

  perror("execve");
  exit(EXIT_FAILURE);
}
```

Figure 13: Launch victim program

```
/* launch the attack */
pid_t launch_attack(int pipe_fd[]) {
  pid_t pid = fork();
  if (pid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
  }
  if (pid)
    return pid;

  close(pipe_fd[READ_END]);
  close(1);
  dup(pipe_fd[WRITE_END]);

  char *args[] = {"./flushreload", NULL};
  execve(args[0], args, NULL);

  perror("execve");
  exit(EXIT_FAILURE);
}
```

Figure 14: Launch attack program

To know when the victim makes an access, the latter will send a signal to its parent after each access. In the performance program, we therefore use an interrupt routine to increment our counter on each signal received from the victim. To retrieve the attacker's result, we perform a `read()` on the pipe.

```
void handle_victime_sig (int sig) {

  v_access++;
  if (v_access > 1) {
    faux_n++;
    v_access = 1;
  }
  total_access++;
}
```

Figure 15: Victim signal handler

```
while (total_access < access_train) {
  /* recv message from attacker */
  read(pipe_fd[0], &p_access,
      sizeof(size_t));

  /* if access predicted by attacker */
  if (p_access == 1) {

    total_predict++;
    if (v_access == 1) { /* good prediction
        */
      total_g_predict++;
      v_access = 0;
    }

    else if (v_access == 0) { /* false
        positive */
      faux_p++;
    }
    p_access = 0; /* reset prediction*/
  }
}
```

Figure 16: Attacker's result reception

At the reception of the attacker's result, we check the coincidence with the counter.

- If the attacker measured an access and the counter of the victim's access `v_access` is equal to 1, then the attacker correctly predicted the last access. We reset the value of the counter of the victim accesses to 0.
- If the attacker predicted an access but the counter is still equal to 0, we have a false positive.

Depending on the validity of the prediction we increase by one `total_g_predict` that represents the number of correct predictions. In the other case we increase by one `faux_p` which represents the number of false positives.

During the reception of the signal we increase the counter, but we also verify its state.

If the counter of the victim's access `v_access` is greater than 1, then the previous access was not detected by the attacker. We then increment the counter of false negatives, and we reset to 1 the counter of the victim accesses. We also increment the counter of the total accesses made by the victim `total_access` so we can establish a statistic afterwards.

The benchmark ends when the victim has made the number of accesses we decided. In the `main`, the benchmark is successively called while increasing the waiting time between each victim's access in an exponential way. Once the benchmark is over, the results are written in a csv file.

```
  res_fp = fopen("result.csv", "w");
  if (res_fp == NULL) {
    perror("fopen");
    exit(EXIT_FAILURE);
  }
  fprintf(res_fp, "time_to_wait,faux_p,faux_n,total_g_predict,total_predict,total_access\n");
  char time_to_wait[10];
  /* test benchmark with different time to wait */
  for (size_t i = 1; i < 10000000; i = i * 2) {
    reset();
    printf("\n\n\n----- Benchmark with time to wait: %ld ----- \n", i);
    sprintf(time_to_wait, "%ld", i);
    benchmark(access_train, time_to_wait, envp);
    resume(i);
  }
```

Figure 17: Main function of the benchmark program

The changes to the attacks and the victim program are slim. For the victim, it is enough to send a signal to its parent at each access. For the attacker, we write the result to the `stdout` output.

```
...
memaccess(addr);
kill(getppid(), SIGUSR1);
...
```

Figure 18: Victim modified for the benchmark

```
access = 0;
int64_t delta = flush_and_reload(addr);
if (delta < threshold) {
  printf("%li, access ! \n", delta);
  access = 1;
}
write(STDOUT, &access, sizeof(int));
delayloop(10000);
```

Figure 19: Flush & Reload modified for the benchmark

Note that here we directly send our boolean result of the attack to the output `stdout`. A signal like the one from the victim, could therefore have been sufficient. But the pipe could allow more information to pass through for future testing.

7 Results

Using our benchmark, we can accurately compare the performance of our attacks, measuring false negatives/positives according to the number of cycles between each access made by the victim. Each attack will be tested over 10,000 hits by the victim, with a cycle count of up to 10,000,000.

7.1 Flush & Reload

The following tests were made on an Intel I5 processor:

```
vendor_id       : GenuineIntel
cpu family      : 6
model           : 60
model name      : Intel(R) Core(TM) i5-4460S CPU @ 2.90GHz
microcode       : 0x28
cpu MHz         : 800.000
cache size      : 6144 KB
cpu cores       : 4
```

Overall, the results show that the Flush & Reload is able to predict the victim's accesses between `90%` and `95%` from the moment the victim waits for `1,000,000` cycles before the next access. Below, the Flush & Reload timeout and noise prevent the attack from properly measuring hits. Moreover, it is only after `2048` cycles that Flush & Reload can perform these first true positives. Regarding false positives, these are strongly present when the expectation cycles are close to 0 (≈25%), but they drop drastically below 1%.

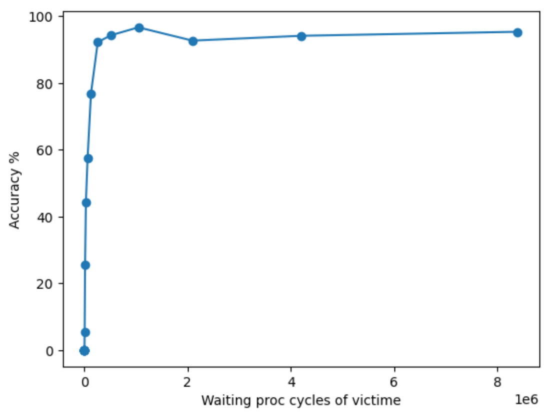

Figure 20: FR: Accuracy according to cycles of wait

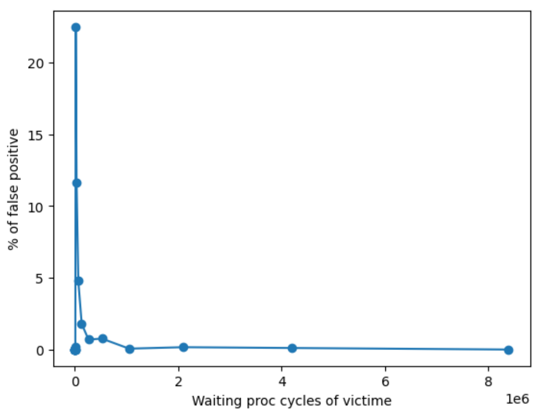

Figure 21: FR: False positive according to cycles of wait

The curves show a pike of precision in the 1e6 wait cycles. They also show a constant precision after that pike.

Here is an example of results of a benchmark with `4 000 000` cycles between each access.

```
    ----- Benchmark with time to wait: 4194304 -----
current time: 4194304
rd: 1
---> Résumé:
Total du nombre d'access: 10000
Total des accès prédits: 9552
Total des accès correctement prédits: 9549
Faux positif: 3
Faux negatif: 451
```

Within 10 000 accesses made by the victim, 9549 were correctly predicted by the attacker. Out of 10 000 accesses, 451 were not predicted by the attacker. Out of 9552 access predictions, 9549 were correct, giving a rate of positives close to 0.03%.

7.2 Prefetch & Reload

A lot of tests were made with our implementation, but none were conclusive. To overcome this issue we recovered the covert channel implementation proposed by the authors on their git [6]. We re-adapted slightly their implementation of prefetch reload for the covert channel, so that it could be used as an attack.
That way, the authors display on their git a script allowing to visualize the time access whether a prefetch was made or not, in order to determine the threshold. The result obtained from this script showed that there were no differences between prefetch or not, on the processor used until now.

The tests were made on a processor Intel I7 of 11th generation. On that one the script picked up a difference of 30 cycles on average, in the case of a prefetch or not.

```
vendor_id       : GenuineIntel
cpu family      : 6
model           : 140
model name      : 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
cpu MHz         : 1524.529
cache size      : 12288 KB
cpu cores       : 4
```

It is important to highlight that the tests were once again made on this processor and that the results were roughly the same. Globally with a similar victim as for flush and reload, our results show that prefetch and reload capped at a precision of only 30/35% out of 10 000 accesses. Thus, those results can significantly vary between an attack to another, depending on the machine's state. Some times, it happened that results fall at a rate of 5% of correct predictions. However the false negatives stay at 1%. So, we display here the optimum results that we were able to measure.

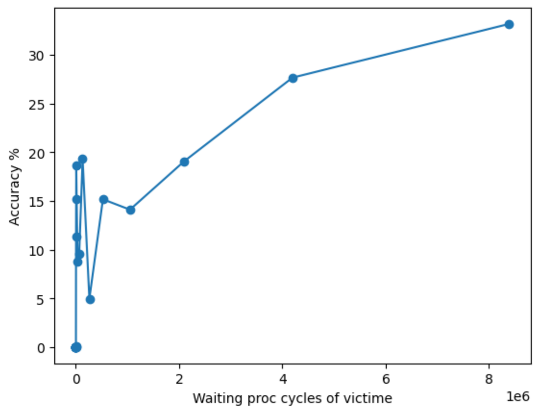

Figure 22: PR: Accuracy according to cycles of wait

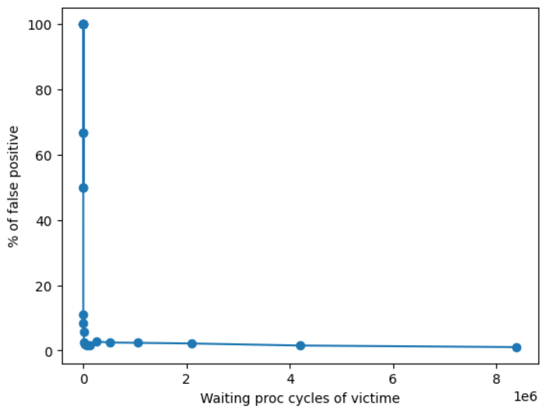

Figure 23: PR: False positive according to cycles of wait

The performances were really doubtful compared to what was indicated in the research paper [3], we tried to increase at best the results. We found one conclusive solution. By modifying only the victim program and especially the way it waits between each access, we noticed an obvious change. In fact, until now the victim waited thanks to the function `delayloop()` that was only passing a certain amount of processor's cycles. But if we replace this call with a microseconds (µs) wait with `usleep(time)`, the attacker increases his accuracy to 45%, even 60% in some tests.

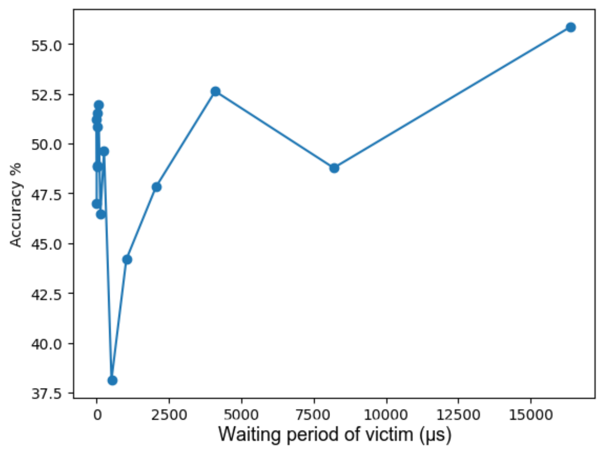

Figure 24: PR: Accuracy according to µs of wait

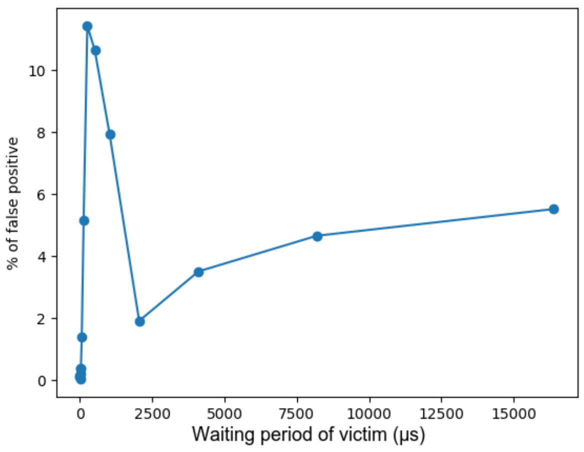

Figure 25: PR: False positive according to µs of wait

The curves show here that the global accuracy of Prefetch & Reload can tend to 50%. However, the false negatives can go up to 5/6%. Here again, our results have a tendency to vary a lot from an execution to another. It is once again the optimum results we could measure.

7.3 Discussion

The side channel attacks Flush & Reload [2] and Prefetch & Reload [3] are attacks that have a goal to trace the utilisation of specific data. Whether it is to discover an encryption key or how a secret is used, the precision of the predictions is at stake. Each false negative or false positive can induce in error an attack on a larger scale using those attacks.

Our researches so far have shown the Flush & Reload attack tends to be portable and effective, with overall accuracy reaching between 90-95%, regardless of how often the victim's traced data is used. Conversely, our tests carried out on our implementation of Prefetch & Reload, as well as on those proposed by the authors, have shown that it is sorely lacking in precision and does not allow us to consider its use in a real case. Indeed, we noted that its performance depended directly on the processor and the frequency of data use by the victim. Thus, in a realistic situation, it would be impossible to use this attack on a larger scale.

Nevertheless it is important to note that our tests were made on everyday machines. The results could not allow to establish a generality on the attack's performance. Thus, we are comparing attacks here only on similar machines, with a similar victim. It is possible that an attack is more efficient on a type of data or in a specific context.

8 Conclusion

The research paper focused on two side-channel attacks on cache memory : Flush & Reload and Prefetch & Reload. The aim was to implement and compare these attacks to determine which was more efficient. Our paper also provides insights into the vulnerabilities of modern computer systems, cache memory and side-channel attacks.

From our researches results we can conclude that the most efficient attack is Flush & Reload. Indeed, we have a great accuracy on the guesses instead of Prefetch & Reload where we get unstable results with a pretty terrible accuracy. While the Flush & Reload attack could accurately predict the victim's accesses at a rate between 90 and 95%, Prefetch's accuracy was capped at 30 and 35%, showing a terrible ability to track the victim's memory accesses on the tested processor. In addition, during the tests, this attack showed a great instability in its results,and finds out to be less portable than Flush & Reload. Indeed, contrary to Flush & Reload, on the first CPU tested, no timing difference couldn't be noticed between a prefetched data or not.

This are the reasons why in a similar context of our tests, we would recommand to use an attack like Flush & Reload in order to perform an attack or a wide range analysis. Moreover, it would be possible to improve predictions precision with speculative execution, like the `Spectre` attack.

In order to tend this kind of attack's precision towards 100% it would be necessary to make tests on other attacks, or improve the ones we already know to be more or less precise with techniques like mentioned before.

However it is important to keep in mind that those results were acquired on similar machines and targets, and as results might vary depending on several factors, we cannot draw any general conclusions about those attacks from them.

[1] [2] [3] [7]

References

[1] P. C. Kocher, “Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems,” in *Advances in Cryptology - CRYPTO ’96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, vol. 1109 of *Lecture Notes in Computer Science*, pp. 104–113, Springer, 1996.

[2] Y. Yarom and K. Falkner, “FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack,” in *23rd USENIX Security Symposium (USENIX Security 14)*, (San Diego, CA), pp. 719–732, USENIX Association, Aug. 2014.

[3] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang, “Adversarial prefetch: New cross-core cache side channel attacks,” *CoRR*, vol. abs/2110.12340, 2021.

[4] J. Jenkov, “Modern hardware,” *jenkov*, 2015/09/09.

[5] Y. Yarom, “Mastik: A micro-architectural side-channel toolkit.”

[6] Y. Guo and A. Zigerelli, “Adversarial prefetch: New cross-core cache side-channel attacks.”

[7] Y. Guo, X. Xin, Y. Zhang, and J. Yang, “Leaky way: A conflict-based cache covert channel bypassing set associativity,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 646–661, 2022.