

Harald Deutschmann

Cache covert channels

Bachelor's Thesis

to achieve the university degree of

Bachelor of Science

Bachelor's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisors

Daniel Gruss, Clémentine Maurice

Institute of Applied Information Processing and Communications
Head: Prof. Reinhard Posch

Graz, June 2018

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.

Date	Signature

Abstract

In recent years, it has been shown that modern processors are vulnerable to cache attacks and may expose a person's private data. Covert channels have been proven to be able to smuggle private information from a trojan process to a spy process. Private information gets transferred from a privileged to an unprivileged environment with a high transmission rate and can then be published by the attacker. Modern browsers can be exploited to supply such covert channels on the internet to reveal secret data.

Therefore, researchers have started to focus on cache attacks and especially on covert channels to gain further information and find countermeasures against such attacks. Detecting such hidden communication is very challenging.

In this project, basics of cache attacks and fundamentals of a covert channel in native code and JavaScript are explained.

Contents

Abstract iii

1 Introduction 1

2 Background 3
- 2.1 CPU Caches . . . 3
- 2.2 Cache Attacks . . . 7
- 2.3 Covert Channel . . . 10

3 Challenges for Cache Attacks in JavaScript 12
- 3.1 Memory Addresses in JavaScript . . . 12
- 3.2 Accurate Timing Resource . . . 12

4 Implementation and Work 14
- 4.1 Prerequisites for the Implementation . . . 14
- 4.2 Prime+Probe Implementation . . . 16

5 Remaining Challenges and Future Work 18
- 5.1 Next Level - Covert Channel . . . 18
- 5.2 From Native to Sandboxed Scripting Language . . . 19

6 Conclusion 20

Bibliography 21

List of Figures

2.1 CPU Cache Hierachy. 5
2.2 Cache Addressing Scheme. 6
2.3 Set-associative cache mapping 8
2.4 Covert Channel Strategy. 11

4.1 Prime+Probe Hit and Miss Histogram. 17

1 Introduction

In recent years, it has been shown that modern processors are vulnerable to cache attacks and may expose a person's private data [2]. Covert channels have proved to be able to smuggle private information from a trojan process to a spy process [18]. Private information gets transferred from a privileged to an unprivileged environment with a high transmission rate and can then be used by the attacker. Modern browsers can be exploited with such covert channels to reveal secret data [12].

The basis of a cache covert channel consists of three parts: A cache attack based on the measurement of memory accesses, depending on whether data is cached or not. A synchronous communication pair consisting of a sender and receiver process and a simple communication protocol. In this context, caches play a major role as a hidden medium for communication. Because a cache is a very fast and small type of memory and is shared across all processes, a covert channel can be built up with a high transmission rate.

JavaScript is now a very dominant language on the internet and up to the 94.7 % of 10 million most popular websites use JavaScript [17]. Therefore, a cache covert channel built up in JavaScript can influence many web pages. This cache covert channel in JavaScript contains additional requirements like having an accurate timing resource or bypassing the restriction to physical memory addresses.

Caches are also shared between tenants in virtualized environments. This, in fact, means that cache covert channels can be disturbed by noise from other system activities and interrupts. For instance, the data in the cache gets evicted by other processes running in parallel, the operating system, and the hypervisor. This makes it difficult to distinguish between a sender or simply noise. Noise can introduce transmission errors and, therefore, influence the communication. But previous work [15, 11] has shown that even with

noise sender and receiver can agree on communication channels. Maurice et al. [11] show how to cope with communication disturbances like noise and scheduled sender and receiver processes to achieve a high-bandwidth robust covert channel.

In this thesis, we mainly discuss the basis for a cache covert channel in native code and talk about challenges for a cache covert channel in JavaScript. We briefly discuss problems coming with cache attacks. Moreover, we are going to deal with one specific cache attack called Prime+Probe. On the one hand, the Prime+Probe technique does not require any privileges. On the other hand, additional eviction strategies and error handling methods must be introduced. We then discuss covert channels in general and how to build a covert channel with cache attacks. After that, we cover the main challenges for a cache covert channel implementation in JavaScript. Then we talk about a Prime+Probe implementation, because it is a very good cache attack for cache covert channels. Afterward, some remaining challenges such as a cache covert channel coping with operating system interrupts and scheduling and a simple communication protocol to make the covert channel more robust, are mentioned.

The remainder of the paper is organized as follows. Section 2 provides basic information about CPU caches, cache architectures, cache attacks and covert channels. Section 3 defines two major challenges for cache attacks in JavaScript. The main differences between a native and JavaScript implementation are discussed. Section 4 describes the Prime+Probe implementation which makes the hidden communication possible. Section 5 details the missing challenges for establishing a robust and fast covert channel in JavaScript. In Section 6, we conclude the project.

2 Background

CPU caches are very fast and small memories, which operate between CPU and main memory (RAM). With CPU caches, the number of main memory accesses can be reduced. This chapter describes the cache architecture, Prime+Probe cache attacks and the principles of a covert channel.

2.1 CPU Caches

2.1.1 Important Cache Notions

A CPU cache is a small and fast memory, usually static random access memory (SRAM), inside the CPU. The task of a cache is to enhance the performance and memory access speed by keeping copies of frequently used main memory locations.

Every time the processor performs read or write instructions, data will be first fetched from the cache if available. A cache hit means that data was loaded from the cache. If data is not available the processor must fetch data from the main memory (DRAM). This is called a cache miss.

In case of a read, the cache will be searched after available data. If data is available, it will be loaded from the cache without accessing the main memory. If data is not available the read or write is passed onto the system bus and data will be loaded from the main memory. In case of a write, the cache's content is updated.

After data is written to the cache it must also be written to the main memory to maintain data coherency between cache and main memory. In the write-through approach, every time data is written to the cache, data is written

to the main memory. This induces a latency increase. Alternatively, in a write-back cache, data is not immediately written to the main memory. The cache instead marks the data which has to be updated in the main memory as dirty. The dirty flag means that data is written back to the main memory as soon as the data gets evicted from the cache.

2.1.2 Cache Components

Basically, the cache consists of at least two components, static random-access memory (SRAM) and the cache controller. The SRAM is a memory block which holds the data. The size of the SRAM determines the size of the cache. The cache controller is responsible for the functional part. Write and replacement policies are implemented in the cache controller. The cache controller also determines cache hits and cache misses.

To accelerate the retrieval of data, a translation lookaside buffer (TLB) is part of the processor and stores recent translations of virtual addresses to physical addresses. If a virtual address is referenced by a program, the CPU first starts to search the data in the fastest cache level. Ideally, the data will be returned at a high rate. The TLB will always be checked for a quick reference to the location of the physical memory. If the address is not found in the TLB, the physical memory must be searched with a memory page walk operation.

The cache is typically organized into data blocks with a fixed size. These data blocks are called cache sets. Moreover, cache sets are partitioned into small cache ways or also called cache lines. The size of a cache set depends on the cache size itself. Additionally, the last-level cache can be divided into cache slices where each processor core has its own cache slice partition in the last-level cache. Each cache slice consists of the same number of cache sets.

Caches can either be inclusive or exclusive. An exclusive cache means a particular cache line is stored once in only one cache level. Whereas L3 inclusive implies that L1 and L2 are in L3. For instance, at least parts of a cache set in the last-level cache can also be in the L2-cache and L1-cache.

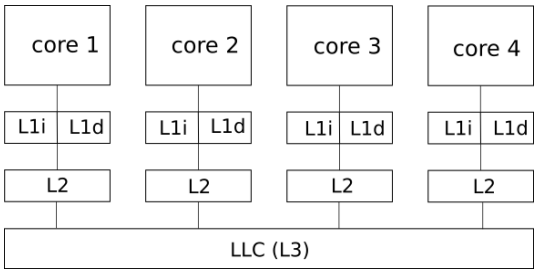

Figure 2.1: Cache hierachy of an intel cpu with three cache layers.

In order to write new data to the cache, the cache may have to evict existing data. The decision of which cache line to evict is defined by the replacement policy. The main issue of any replacement policy is that it must predict which existing cache data is least likely to be used in the future. For instance, the replacement policy Least Recently Used (LRU) replaces the least recently accessed cache line from the cache.

2.1.3 Cache Architecture

Nowadays, processors have multiple levels of caches which are often divided into an instruction cache and a data cache. In this thesis, we focus more on Intel processors. The most recent processors have a split L1 cache and a L2 cache. Additionally, processors have a L3 cache also called last-level cache (LLC) [1]. As depicted in Figure 2.1, the hierarchy of all cache layers typically starts at L1 and goes down to the last-level cache. The L1 cache is the smallest and fastest cache, near the processor. In contrast, the last-level cache is the biggest and slowest cache. An important property of the last-level cache is that it is shared among all cores and inclusive, thus if data gets evicted from the last-level cache data will also be evicted from the core's local L1 and L2 cache. In further consideration, the cache can have three different organizations:

- fully-associative
- directly-mapped
- set-associative

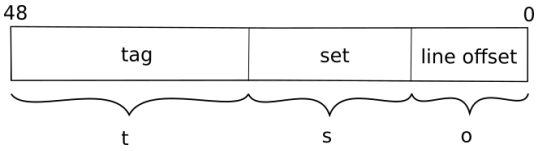

Figure 2.2: Cache addressing scheme of virtual or physical addresses.

Firstly, the fully-associative approach merely uses cache lines. A fully-associative cache allows storing data from the main memory into any cache line. Therefore, a fully-associative cache implementation can be very complex. In order to check if a data is available in the cache, the processor has to compare all cache lines in parallel. For larger caches, the fully-associative approach is power consuming.

Secondly, a directly-mapped cache consists of cache lines or bigger cache locations. The main memory is partitioned into memory locations with the size of a cache location. With this in mind, our memory address must be split into an indexing and a tag part. With a simple mapping algorithm using the indexing part of the memory address, memory locations are mapped to specific cache locations. Because the cache is much smaller than the main memory, some memory locations map to the same cache location. Thus, we need a tag to distinguish between addresses which map to the same cache line. As a result, the directly-mapped method is the most straightforward approach of all three. Therefore, it is also the cheapest one to implement.

Thirdly, in contrast to the other organizations, set-associative caches consist of cache sets. Set-associative caches are often used in modern processors. As already mentioned before a cache set can be divided into many cache ways, therefore we can have a so-called m-way set-associative cache architecture. An m-way set-associative cache typically has cache sets divided into m-ways. Hence the cache is divided into x cache sets $x = m \cdot$way size (line size).

2.1.4 Cache Indexing and Mapping

In this section, the m-way set-associative architecture is explained in more detail, because the processor under test is an Intel Core i7 and Intel's processors have set-associative caches. The memory address is used to

compute the cache location where the data is copied to. There are techniques where either the virtual or physical address is used.

The mapping of any line in a set-associative cache can be reconstructed as depicted in Figure 2.2. The memory address either virtually or physically is separated by the cache into three parts, to uniquely identify the corresponding memory location in the cache: the tag, the set, and the line offset. The lowest o bits determine the offset of the line. It can be computed with $o = log_2(\text{line size})$. The line offset is then followed by s bits ($s = log_2(\text{number of sets})$). The rest of the address can be used as a unique tag inside the cache set. Sometimes lower address bits are used for the tag as well. Virtually-indexed virtually-tagged caches use virtual addresses to locate the data in the cache. But modern processors more often use a physically-indexed physically-tagged (PIPT) cache. There, physical addresses are used to locate the data in the cache. Important to mention are virtually-indexed physically-tagged (VIPT) and PIPT caches, thus Intel's L1 caches are typically virtually-indexed physically-tagged and L2, L3 caches are physically-indexed physically-tagged.

Illustrated in Figure 2.3 a n-way set-associative L3 cache can be seen. At first, a 4-way set-associative cache is shown. After that a m-way set-associative L3 is depicted, illustrating cache slices.

2.2 Cache Attacks

Cache attacks are side-channel attacks and exploit the latency of data loaded from main memory or from the cache. Cache attacks can gain information of cached or not cached data in a program [16]. Since reading from the cache enhances the performance of the CPU, the attacker can track the victim's activities. Even so, writing data into the cache can cause the same issue as explained in Section 2.1.1. This way, secret keys of cryptographic algorithms can be computed, web pages and various programs can be attacked. To give an illustration a very common approach to use cache attacks in web pages is to build a covert channel. Covert channels can easily send secret information from a restricted to an unrestricted environment and publish information. A simple example to show how libraries in the operating system can easily

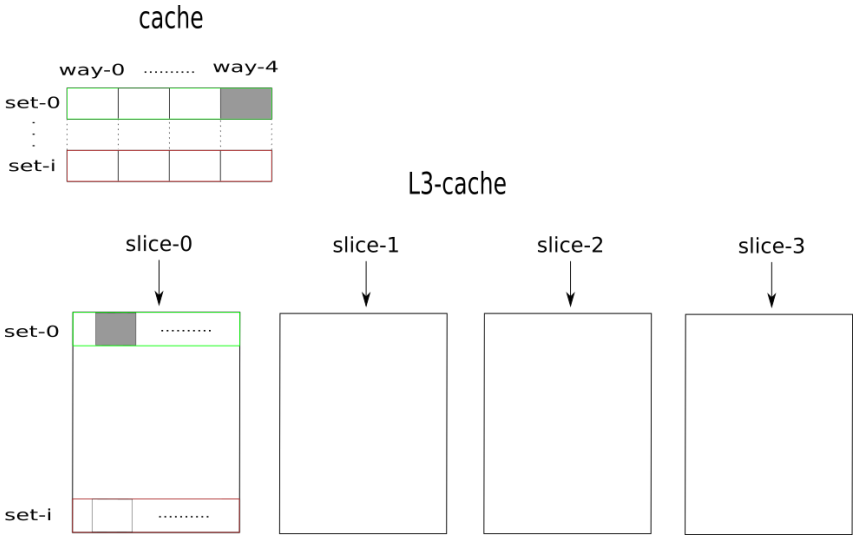

Figure 2.3: Set-associative cache mapping

be used to gain secret information, to track mobile device events, is shown by Lipp et al. [6].

2.2.1 Types of Cache Attacks

Cache attacks can be classified into three main categories:

- time-driven attacks
- trace-driven attacks
- access-driven attacks

When a load instruction triggers a cache miss, data must be loaded from the physical memory into a cache line. As a result, the memory access requires more time as compared to a cache hit. Consequently, an attacker can distinguish cache hits from cache misses by, for example, comparing access time differences. The concept behind a time-driven cache attack is that the attacker observes the overall execution time of a program. The attacker then must gain information by analyzing cache hits and misses on many executions. The advantage of time-driven attacks is that they have the

least restrictions for the attacker. For example, the attacker does not need to have the ability to watch the outcome of memory accesses [8].

An attacker can gain a lot of information if he is able to perform measurements during an AES encryption. In a trace-driven cache attack, the attacker obtains the traces of cache hits and misses for a sample of encryptions. He then recovers the secret key of a cryptosystem using this data. A trace is defined as a sequence of cache hits and misses. Through these traces the attacker then can determine whether a particular access during an encryption is a hit or miss. This attack reveals more information to the attacker but requires a large number of traces.

Access-driven attacks can further be categorized into two types. If the attacker shares memory with its victim, like a shared library, the attacker can use Flush+Reload and other variants such as Evict+Reload [19]. Otherwise, the attacker can use the Prime+Probe method. This technique does not require shared libraries. The attacker has to find congruent addresses. With congruent addresses, the attacker can access the same cache sets in a short time interval to determine which cache set and therefore memory addresses the victim recently has accessed. In general, noise caused by other processes can influence the analysis of cache hits and misses and mislead the attacker. In addition to the noise, the replacement policy makes it difficult to completely evict the cache set at once. We focus on the Prime+Probe technique.

The Prime+Probe technique consists of a Prime phase and a Probe phase. In the Prime phase, the attacker evicts some cache sets. After that, the attacker waits some time to get the victim process scheduled. In the Probe phase, the attacker process accesses the same sets again and determines which sets were evicted. The access to those sets which are slower was most likely accessed by the victim or other processes. This way the attacker can gain information about the user's accesses or use the cache as a covert channel.

2.3 Covert Channel

2.3.1 Covert Channel Overview

A covert channel typically uses methods that are not intended for communications. For instance, checking a file for being locked to convey a 1 or 0. In a covert channel, one process leaks information to another process, which typically has no access to that information. A covert channel is always a communication channel that is hidden. There is a purpose to hide its existence from someone who tries to protect the system. For example, by measuring the CPU performance.

As mentioned before, covert channels always need two attacker processes geared to each other, to actively exchange information. They are used to communicate sensitive information to a spy process that shares some physical memory. Outlined by Maurice et al. [9], functions of address translation and also functions that map a physical address to a cache set are not shown to the user. Consequently, there is a uncertainty of the physical memory mapping to a specific location in the cache. Therefore, it is difficult to establish a common communication channel between sender and receiver. Liu et al. [7] use a 4KB memory page allocation technique to congruent addresses. Yan et al. [18] introduce a 2MB array allocation technique typically allocated by the browser when loading a web page [12].

2.3.2 Cache Covert Channel

For a cache-based covert channel to work some requirements must be fulfilled. First of all to convey a 1 or 0 a basic cache attack scheme must be used. Therefore, an accurate timing source with CPU cycle accuracy is highly recommended. This, for example, gives the possibility to make a sender process communicate with a receiver process by exploiting the different access delays between cache and main memory. A receiver process can interpret a cache hit, i.e., a fast access, as a 1. Conversely, the cache miss, i.e., a slow access, as a 0.

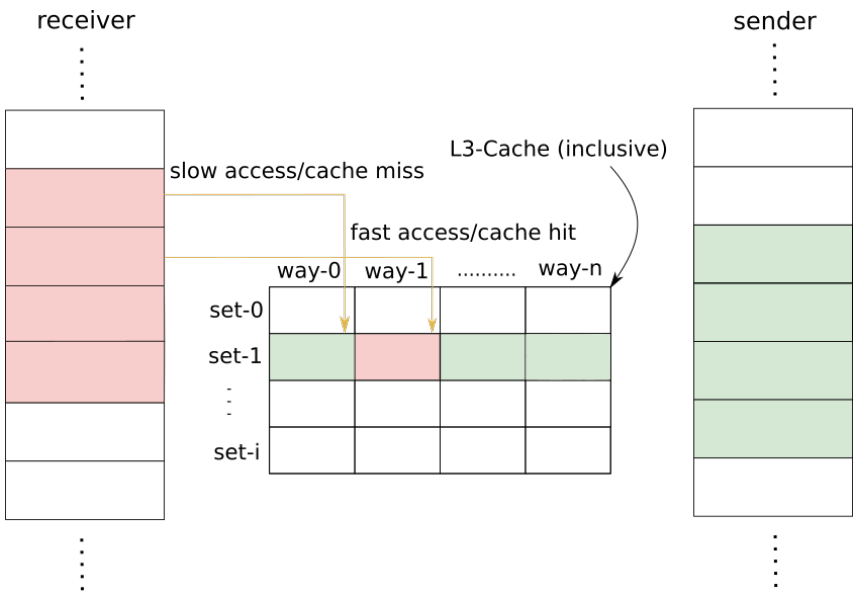

Figure 2.4: Covert channel strategy with Prime+Probe eviction.

First and foremost, the sender and the receiver have to agree on the same cache sets. As shown by Maurice et al. [11], a so-called jamming agreement handshake, a robust method initially used in wireless sensor networks was introduced. This method grants robust communication channels over cache sets. Even at a high noise rate, it is possible that the sender and the receiver coordinate on the same cache sets. Figure 2.4 illustrates a hidden communication example over one specific cache set. In this example, the sender has decided to send 0 by producing a cache miss on the receiver's side. The sender first fills up the cache set with his own data, marked as the red boxes in the L3-cache. This step can be seen as a Prime phase from the sender's side. After some time the receiver is able to probe the cache set and sum all cache hits or cache misses. In case of more cache hits than cache misses the receiver interprets a 1, otherwise 0. For cache set eviction Gruss et al. [4] introduced an eviction strategy.

3 Challenges for Cache Attacks in JavaScript

As discussed by Schwarz et al. [13, 14] and Oren et al. [12] in contrast to native code, JavaScript code is sandboxed and inherently single-threaded, posing a challenge for the attacker. Despite this, the following chapter discusses two main challenges for a JavaScript implementation.

3.1 Memory Addresses in JavaScript

JavaScript is a sandboxed scripting language. The programmer is neither aware of pointers nor aware of virtual addresses. Instead, the closest to virtual addresses are ArrayBuffers. ArrayBuffers are used to allocate memory in JavaScript. ArrayBuffers are very fast, efficient and contiguous blocks of virtual memory. If the programmer is able to identify one virtual address within the ArrayBuffer the remaining addresses are known as well, because the memory addresses and the array indices are linear [13]. If we allocate an ArrayBuffer with a size of more than 2MB we then further know 21 LSB, because browsers allocate 2MB large pages.

3.2 Accurate Timing Resource

For side-channel attacks, an accurate timing measurement is the most important part. Modern browsers use the Performance object to access a timing source with the Performance.now() function. But the resolution of timing sources in JavaScript is restricted to 5 microseconds of accuracy. However,

cache timing differences, and interrupt timings require a timing resolution in a range of 200 nanoseconds to determine a cache hit using Prime+Probe. It is possible to get a high-resolution clock by using SharedArrayBuffers. One thread is continuously incrementing the value of the buffer without checking for any events on the events queue. The other thread reads the current value in the buffer [5, 14, 3].

4 Implementation and Work

This chapter contains a description of the fundamentals of a cache-based covert channel in native code. First, we consider the Prime+Probe technique and the creation of an accurate threshold between cache misses and cache hits. Secondly, a cache set agreement implementation is made on both sides, sender, and receiver. To begin with, the main requirements that facilitate the implementation of Prime+Probe in the virtual memory space are discussed.

4.1 Prerequisites for the Implementation

4.1.1 Virtual Addresses and Large Page Allocation

In Chapter 2, we heard about the unknown virtual memory to physical memory location mapping to the processes. Therefore, we need a solution to bypass the address uncertainty to processes. Today's browsers typically allocate 2MB memory pages. The 2MB physical memory page requested from the browser is partitioned into many cache sets. Each cache slice is partitioned into many cache sets. In a set-associative cache, this means that within the 2MB memory space, more than one virtual address is mapped to one cache set. Furthermore, by virtually allocating 2MB pages at once, referred to the binary system 2MB$= 2^{21}$B , the 21 least significant bits (LSB) of the virtual memory and physical memory are the same. Thus, we know the 21 least significant bits of a physical address. Because the cache is also smaller than one cache slice, much physical memory addresses map to the same cache set, so-called congruent addresses. If we iterate over the virtual addresses which map to more than one 2MB physical memory page we

then get many congruent addresses for each cache set. After collecting these congruent addresses for one particular cache set we can force data to be evicted from this cache set. However, processes can start on different CPU cores and addresses are assigned to different L3-cache locations, respectively cache slices. Therefore a method that uses the information given by the reverse-engineered function from Maurice et al. [10] without privileges must be described. Knowing only virtual addresses, the slice virtual index is computed by cutting off the 21 least significant bits of the address and applying the reverse-engineered addressing function. Inside a 2MB page, addresses that have the same slice virtual index, belong together. To find addresses from the same slice the slice virtual index for each page must correspond to the same physical index for all pages [10, 11].

Referring to Section 2.1.4, we can now extract all relevant bits from the physical address to map the virtual addresses to cache sets. The following example solves a given address uncertainty problem. Given an Intel dual-core CPU with a 4MB 16-ways L3-cache and a 64-byte line size, we can now determine the cache indices. Firstly, in a dual-core processor, our L3-cache is split into two slices. The cache size can be divided by two, thus the amount of cache sets is divided by the slice's count. The mapping from physical addresses to slices is not documented by Intel. A function has been reversed-engineered by Maurice et al. [10] to determine the slice of the memory address. Starting with 4MB $= 2^{22}$ bytes we now have 2^{21} bytes for each core. Then our lowest bits refer to the line offset o, $o = \log_2(64) = 6$ bits. Each cache set consists of 16-ways, hence each cache set has $16 \cdot 2^6 = 2^{10}$ bytes. If we divide the cache size of each core by the size of one cache set we get the number of cache sets and the cache set index, $\frac{2^{21}}{2^{10}} = 2^{11}$ and $s = \log_2(2^{11}) = 11$ bits. To fully determine the physical memory location in the cache we merely need to know $11 + 6$ least significant bits of the whole physical address. With the 2MB large pages prerequisite, 21 least significant bits are known.

4.1.2 Accurate Time Resources

For a Prime+Probe based covert channel to exist, it is very important to have access to a high timing resolution. Nowadays memory accesses happen

within 200 nanoseconds, cache accesses can be in faster than 1 nanoseconds. With the *x*86 instruction set down to 1 nanosecond can be measured with the *rdtsc* instruction. This makes it easy to measure the differences in time of accesses.

4.1.3 Eviction Strategy and Cache Set Agreement

To increase the probability of evicting a cache set, the access patterns of Gruss et al. [4] are used. By increasing the count of congruent addresses the eviction probability can increase to almost 100 % with the drawback of time consumption. The main goal is to find sufficient congruent addresses to evict the whole cache set. In my specific up to date example, Intel Core i7-4600U with a 4 MB, 16-ways L3-cache and a 64-byte line size, 18 congruent addresses for the Prime phase are used.

4.2 Prime+Probe Implementation

For a Prime+Probe implementation to work we first need to find congruent addresses. This is done by iterating through each cache line in the 2MB allocation or more. By observing the address of each cache line we get the cache slice index and cache set index. Each 2MB allocation can have up to $\frac{2^{21}}{\text{slices} \cdot 2^{s+o}} = \frac{2^{21}}{2 \cdot 2^{11+6}} = 8$ congruent addresses. It then is highly recommended to observe many evictions with varying numbers of addresses used for the Prime phase and for the Probe phase to determine the amount of needed prime addresses and probe addresses. Figure 4.1 points out one cache hit and miss evaluation with 18 Prime addresses and 5 Probe addresses using an Intel Core i7-4600U with a 4MB, 16-ways L3-cache, and a 64-byte line size. A threshold between cache hits and misses can be well defined.

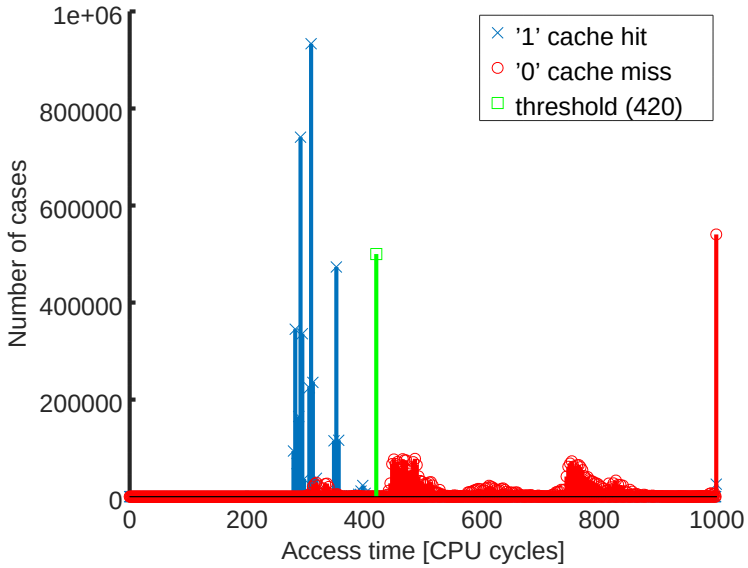

Figure 4.1: Prime+Probe hit/miss histogram

5 Remaining Challenges and Future Work

This section summarizes remaining challenges for a fluently running covert channel based on JavaScript. We discuss what is missing from a basic covert channel with a simple communication protocol. Besides that, the missing steps from native code to JavaScript are briefly discussed.

5.1 Next Level - Covert Channel

5.1.1 Covert Channel Fundamentals

In principle, the sender process sends bits with the usage of the Prime method. Either the sender totally evicts the receiver's data or he does nothing for a given period of time. Conversely, for the receiver, a probing phase is introduced for a given amount of time. If the receiver probes and gets an access time over the threshold, his data got evicted and results in a cache miss. Otherwise, the receiver interprets it as a cache hit.

The main problems in covert channel implementations remain in the sender and receiver synchronization. As explained by Maurice et al. [11] hardware and scheduling interrupt disturb synchronous communications. Scheduling induces either insertion or deletion errors in a covert channel. It is important to handle scheduling from the operating system, which induces errors during the communication, to establish a covert channel.

5.1.2 Simple Communication Protocol

The main task of a simple communication protocol is to manage noise errors and make the covert channel more robust. During communication, each data frame should be consistent all the way from the sender to the receiver. At best, transmission errors should be recognized quickly and retransmitted accordingly. Maurice et al. [11] introduce a robust covert channel with a data-link layer and a physical-layer protocol.

5.2 From Native to Sandboxed Scripting Language

As already mentioned in Section 3, transferring the native code into a sandboxed scripting language involves additional tasks to solve. Before implementing a covert channel in JavaScript it is necessary to think of a stable and fast concept in native code. After that, the work lies in basics of different programming language behaviors and not in the concept of the attack. Finally, the programmer can then stick to the implementation of an accurate timing resource with a resolution of nanoseconds and circumvent the virtual memory constraints in JavaScript.

6 Conclusion

In this thesis, the Prime+Probe technique was fully implemented in native code and important steps to realize an implementation in JavaScript have been discussed. A lot of background, providing us information about caches, cache architectures, cache attacks and covert channels, was discussed. There we talked about covert channels and about advantages using cache based covert channels. We learned that caches are an ideal resource, shared across all processes across cores. That is why it is possible to evict the data from different processes in virtualized environments. The Prime+Probe technique permits to evict any data from the shared cache and does not rely on shared libraries and, therefore, is the most practical method. Additionally, remaining challenges were introduced indicating the reader what remains to be solved for a robust cache covert channel.

Bibliography

[1] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer Manuals*. Available at: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf. 2016 (cit. on p. 5).

[2] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. *A survey of microarchitectural timing attacks and countermeasures on contemporary hardware*. Google Scholar. 2018 (cit. on p. 1).

[3] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. *ASLR on the Line: Practical Cache Attacks on the MMU*. NDSS'17. 2017 (cit. on p. 13).

[4] D. Gruss, C. Maurice, and S. Mangard. *Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript*. DIMVA'16. 2016 (cit. on pp. 11, 16).

[5] L. T. Hansen. *Shared memory: Side-channel information leaks*. Available at: https://github.com/tc39/ecmascript_sharedmem/blob/master/issues/TimingAttack.md. 2016 (cit. on p. 13).

[6] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. *ARMageddon: Cache Attacks on Mobile Devices*. USENIX Security'16. 2016 (cit. on p. 8).

[7] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. *Last-Level Cache Side-Channel Attacks are Practical*. IEEE Symposium on Security and Privacy. 2015 (cit. on p. 10).

[8] C. Maurice. "Information leakage on shared hardware: Evolutions in recent hardware and applications to virtualization." PhD thesis. Thesis, Oct. 2015 (cit. on p. 9).

[9] C. Maurice, C. Neumann, O. Heen, and A. Francillon. *C5: Cross-Cores Cache Covert Channel*. DIMVA'15. 2015 (cit. on p. 10).

[10] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon. *Reverse Engineering Intel Complex Addressing Using Performance Counters*. RAID'15. 2015 (cit. on p. 15).

[11] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, K. Römer, and S. Mangard. *Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud*. NDSS'17. Available at: https://gruss.cc/files/hello.pdf. 2017 (cit. on pp. 1, 2, 11, 15, 18, 19).

[12] Y. Oren, V.P. Kemerlis, S. Sethumadhavan, and A.D. Keromytis. *The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications*. CCS'15. 2015 (cit. on pp. 1, 10, 12).

[13] M. Schwarz, M. Lipp, and D. Gruss. *JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks*. NDSS'18. 2018 (cit. on p. 12).

[14] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard. *Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript*. FC'17. 2017 (cit. on pp. 12, 13).

[15] M. Schwarz and M. Weber. *CJAG: Cache-based Jamming Agreement*. Black Hat Asia. 2017 (cit. on p. 1).

[16] R. Spreitzer and T. Plos. *On the Applicability of Time-Driven Cache Attacks on Mobile Devices*. Springer Link. 2013 (cit. on p. 7).

[17] *Usage of JavaScript for websites*. W3Techs'17. Available at: https://w3techs.com/technologies/details/cp-javascript/all/all. 2017 (cit. on p. 1).

[18] M. Yan, Y. Shalabi, and J. Torrellas. *ReplayConfusion: Detecting Cache-based Covert Channel Attacks Using Record and Replay**. NDSS'16. 2016 (cit. on pp. 1, 10).

[19] Y. Yarom and K. Falkner. *FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack*. USENIX Security'14. 2014 (cit. on p. 9).