Felix Holzknecht

Cache Template Attacks on Apple Silicon

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Daniel Gruss

Andreas Kogler

Institute of Applied Information Processing and Communications

Graz, June 2024

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. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Signed by:	Felix Maria Holzknecht
Date:	22.07.2024 12:09:04

This document is digitally signed!

This document bearing a qualified electronic signature has the same legal standing as a handwritten document in accordance with Article 25 (2) of Regulation (EU) No. 910/2014 of 23 July 2014 ("eIDAS-VO").

Validation: Information for checking the electronic signature can be found at: www.a-trust.at/pdf

Date, Signature

Acknowledgements

I want to extend my gratitude to all the people who have supported me throughout my journey in completing this master's thesis.

First, I want to thank my advisors, Daniel Gruss and Andreas Kogler, for their support while writing this thesis. I am very thankful for your feedback on the draft versions and input on several topics.

Second, I would like to thank my parents, Martha and Peter, for supporting my passion for computers since childhood. Without you, achieving this degree would not have been possible.

Finally, I want to thank my friends for their friendship and moral support since day one at university. I especially want to mention Mathias Oberhuber, Michael Steiner, Jonas Niedermair, Mathias Mattersberger, and Martin Nagele here. Together, we spent numerous days (and also nights) at university.

Thank you! Felix Holzknecht

Abstract

Cache template attacks are a significant threat to system security. A cache template attack allows adversaries to extract sensitive information by exploiting cache behavior. Recent research demonstrated the feasibility of cache attacks on Apple Silicon by implementing an Evict+Reload covert channel and a Spectre-PHT exploit on several Apple devices. This thesis examines the possibility and impact of cache template attacks on Apple Silicon, focusing on the Apple M1 and A15 Bionic CPUs.

First, we discuss the fundamental principles of cache attacks, like eviction set generation, timers, and covert channels. We establish an Evict+Reload-based covert channel on the Apple iPhone 13 mini equipped with the Apple A15 Bionic CPU. We optimize the covert channel on the iPhone 13 mini with application pinning and machine-learning-based post-processing. With these improvements, we achieve a transmission speed of 9.6 kbit/s and an error rate of 7.13 %. Next, we evaluate the optimized covert channel on the Apple Mac mini (M1), where we achieve a transmission speed of 122.46 kbit/s and an error rate of 3.75 %, making our covert channel the fastest Evict+Reload-based covert channel running on a CPU implementing the ARM instruction set. Furthermore, this thesis presents the first cache template attack on Apple Silicon. Based on the building blocks from the covert channel, we implement a cache template attack targeting an Apple Mac mini running macOS. This cache template attack detects specific user actions within applications, such as opening new tabs or playing back media content in Mozilla Firefox.

Our experiments confirm that recent Apple CPUs are vulnerable to classic cache template attacks, even without the availability of cache maintenance instructions. However, exploiting cache template attacks on Apple devices presents greater challenges than on classic x86-64 systems due to limited hardware documentation and restricted access to low-level interfaces in Apple's operating system.

Keywords: Covert Channel · Cache Template Attack · Side Channel · Apple Silicon

Kurzfassung

Cache-Template-Angriffe stellen eine erhebliche Bedrohung für die Systemsicherheit dar. Ein Cache-Template-Angriff ermöglicht es Angreifern sensible Informationen durch das Ausnutzen der Eigenschaften eines Cache zu extrahieren. Jüngste Forschungen haben gezeigt, dass Cache-Angriffe auch auf Apple CPUs möglich sind, indem sie basierend auf Evict+Reload einen verdeckten Kanal und einen Spectre-PHT-Angriff auf verschiedenen Apple-Geräten implementiert haben. Diese Arbeit untersucht die Möglichkeiten und Auswirkungen von Cache-Template-Angriffe auf Apple CPUs, wobei der Fokus auf den CPUs Apple M1 und Apple A15 Bionic liegt.

Zunächst diskutieren wir die grundlegenden Prinzipien von Cache-Angriffen, wie die Generierung von Eviction-Sets, Timer und verdeckte Kanäle. Wir implementieren einen Evict+Reload-basierten verdeckten Kanal auf dem Apple iPhone 13, welches eine Apple A15 Bionic CPU verwendet. Um die Performance des verdeckten Kanals zu optimieren, binden wir unsere Anwendungen an Cluster der CPU und nutzen maschinelles Lernen zur Datennachbearbeitung. Mit diesen Verbesserungen erreichen wir eine Übertragungsgeschwindigkeit von 9.6 kbit/s und eine Fehlerquote von 7.13 %. Anschließend evaluieren wir den optimierten verdeckten Kanal auf dem Apple Mac mini (M1), wo wir eine Übertragungsgeschwindigkeit von 122.46 kbit/s und eine Fehlerquote von 3.75 % erzielen, wodurch unser verdeckter Kanal der schnellste Evict+Reload-basierte verdeckte Kanal auf einer CPU mit ARM-Befehlssatz ist. Darüber hinaus präsentiert diese Arbeit den ersten Cache-Template-Angriff auf Apple Silicon. Basierend auf den Bausteinen des verdeckten Kanals implementieren wir einen Cache-Template-Angriff auf einem Apple Mac mini mit macOS. Dieser Cache-Template-Angriff erkennt spezifische Benutzeraktionen innerhalb von Anwendungen, wie das Öffnen neuer Tabs oder das Abspielen von Medieninhalten in Mozilla Firefox.

Unsere Experimente bestätigen, dass aktuelle Apple-CPUs anfällig für Cache-Template-Angriffe sind, selbst ohne die Verfügbarkeit von Cache-Wartungsanweisungen. Auf Apple-Geräten ist es im Vergleich zu klassischen x86-64-Systemen schwieriger Cache-Template-Angriffe durchzuführen, da die Hardware nicht vollständig öffentlich dokumentiert ist und der Zugriff auf niedrigstufige Schnittstellen in Betriebssystemen von Apple nur begrenzt möglich ist.

Schlagwörter: Verdeckter Kanal · Cache-Template-Angriff · Seitenkanal · Apple Silicon

Contents

Chapter 1

Introduction

Cache template attacks exploit the different durations of cache hits and cache misses and pose a significant security issue [10]. By performing a cache template attack, an attacker can deduce which cache lines are currently accessed by another application running on the same CPU. This information can potentially allow the attacker to infer sensitive data, such as cryptographic keys, or detect events like triggered keystrokes in this application [21].

In 1996, Kocher [31] demonstrates the first cache attack to leak cryptographic keys from other applications running on the same CPU using timing variations. In 2015, Gruss et al. [21] demonstrate the first cache template attack. They introduce a generic approach to leak information from applications running on CPUs that implement shared inclusive last-level caches. With this approach, they could leak cryptographic keys or events, like keyboard inputs, from applications running on the same CPU. In 2016, Lipp et al. [35] demonstrate that cache template attacks are feasible on ARM-powered devices. They leaked various events like keystrokes or touchscreen inputs on non-rooted ARM-powered Android devices.

Hetterich et al. [25] show that Apple's ARM-based CPUs used in the Mac mini (M1), iPhone 7, and iPhone 8 Plus are vulnerable to cache attacks based on Evict+Reload. In their work, they used a counting thread and a cache eviction approach to implement Evict+Reload. They successfully launched a covert channel and a Spectre-PHT [30] attack on these devices.

In this thesis, we demonstrate that the Apple A15, used in the iPhone 13 mini, is also vulnerable to cache attacks. Additionally, we show that Apple M1 CPUs are susceptible to cache template attacks. We discuss the challenges that need to be addressed to perform cache-based side-channel attacks on Apple silicon and present possible solutions for them. We show that it is possible to launch a covert channel based on Evict+Reload on the Apple Mac mini (M1) and the iPhone 13 mini. For this demonstration, we take the building blocks from Hetterich et al.'s [25] Spectre-PHT exploit and optimize the performance of these building blocks. Additionally, we improve the performance of the covert channel by using machine learning models to interpret the transmitted encoded data. To demonstrate our improvements, we evaluate the covert channel [53] on a Mac mini, powered by an Apple M1 processor, and an iPhone 13 mini, powered by an Apple

A15 processor. Our covert channel achieves data transmission speeds of up to 159 kbit/s with an error rate of 3.74 % on an Apple M1 CPU. These numbers result in a true capacity of 122.46 kbit/s. Our covert channel is currently the fastest Evict+Reload-based covert channel on a CPU implementing the ARM instruction set. Based on this covert channel, we implement a cache template attack on an Apple Mac mini (M1) running macOS Monterey (version 12.5). With our cache template attack, we can detect whether a website opened in Mozilla Firefox on the victim's device plays back multimedia content with an accuracy of 83.89 %.
Our contributions:

- We show that machine learning classifiers can significantly boost the performance of noisy Evict+Reload-based covert channels.
- We implement the currently fastest Evict+Reload-based covert channel on an ARM-based CPU.
- We present the first cache template attack on a device powered by an Apple M1 CPU and running stock macOS Monterey (version 12.5).

Chapter 2 outlines the foundational concepts. We start by giving a short overview of related work. Next, we present the different caches of ARMv8 CPUs and how they interact with the main memory. This knowledge is essential to understand the theory behind cache attacks. We briefly discuss virtual memory, which is required to understand the concept of shared memory. This chapter also describes how Hetterich et al. [25] circumvented the limitations on Apple CPUs to perform Evict+Reload-based attacks. Next, we explain how Evict+Reload can be used to set up a covert channel that allows data exchange between two processes via the CPU cache. Based on this covert channel, we describe how cache template attacks work. Machine learning classifiers can be used to interpret the measured data from cache attacks. Therefore, we conclude this chapter with a short overview of common machine learning classifiers.

Chapter 3 discusses our threat model. We briefly present the devices and the corresponding operating systems we target with our attack. We describe which preconditions are required to successfully launch a covert channel and mount a cache template attack in a real-world environment.

Chapter 4 gives a high-level overview of how we implement our covert channel and cache template attack. We describe the concept behind our covert channel, the task of the sender and receiver process, and the protocol the sender and receiver implement to exchange data. This chapter explains how to set up a cache template attack based on the building blocks we previously used in the covert channel. Cache template attacks are divided into two parts: a profiling phase and an exploitation phase. We explain the purpose of these phases and describe how we implement them.

Chapter 5 lists the different challenges we have to solve when implementing cache-based side-channel attacks on Apple Silicon. We describe how we solved these challenges. Apple CPUs do not provide a reliable high-resolution timer or an instruction to remove data from the cache from user-space applications. Moreover, Apple's operating systems offer limited control and debugging interfaces for the hardware, making it challenging to replace the missing flush instruction and timer reliably.

Chapter 6 evaluates the performance of our implemented attack. We start by identifying a cache hit and miss threshold and evaluating the reliability of cache eviction on our devices. Next, we evaluate the performance of the covert channel. For this task, we run the covert channel on an Apple iPhone 13 mini and an Apple Mac mini (M1) and apply different post-processing approaches. To demonstrate the effectiveness of the cache template attack, we show that we can successfully detect when a victim opens a new tab in Mozilla Firefox. Finally, we show that our cache template attack can reveal whether the victim is browsing a website with or without multimedia playback.

Chapter 7 provides a summary and conclusion for this master thesis.

Chapter 2

Background

In this chapter, we provide the required background knowledge for understanding cache attacks and the content of this thesis. In Section 2.1, we provide an overview of related work. Section 2.2 explains main memory, why modern CPUs require caches, and how caches usually work in detail. Furthermore, this section covers the basics of virtual memory, which isolates processes in modern operating systems and is a common attack vector for cache attacks. Section 2.3 discusses available timers on Apple devices, which are essential to perform cache attacks. Another requirement for most cache attacks is the ability to flush the CPU cache. Section 2.4 explains the basic concept behind cache attacks and discusses the widely known cache attack primitives Flush+Reload and Evict+Reload. Based on this explanation, Section 2.5 describes how these cache attack primitives allow an unprivileged attacker to set up a covert channel between two processes running on the same CPU. Finally, Section 2.6 explains the general concept of cache template attacks and how they can be used to leak information, such as triggered keystrokes from another process running on the same CPU.

2.1 Apple Silicon

In 2014, Apple started to design CPUs for its mobile devices based on the ARM instruction set. In 2020, Apple launched its first desktop CPU, the Apple M1, based on the ARMv8.5-A instruction set [16]. This CPU gained recognition for its high performance and energy efficiency [58].

Over the last few years, side-channel attacks have become a more prominent research topic. However, most of this research focuses on x86 CPUs. Despite their growing popularity, side-channel attacks on Apple CPUs are rare. There are two main reasons Apple devices are unattractive to security researchers: First, Apple does not publicly document its hardware. Second, Apple's macOS and iOS operating systems do not expose many low-level interfaces to the user-space. Users can interact with the underlying hardware only in a very restricted way. These limitations make security research on Apple devices a tedious job. Despite these hurdles, a few papers exist about physical and cache-based side-channel attacks, which we discuss in the following.

First, we discuss physical side channels. In 2016, Genkin et al. [13] leaked cryptographic keys via electromagnetic and power side channel attacks on an iPhone 4. Lisovets et al. [36] used a BootROM exploit and a power side channel to speed up passcode brute-forcing on an iPhone 4. Haas et al. [22] showed that the ARMv8 Cryptographic Extension (ARM CE) is vulnerable to electric radiation side-channel attacks. The ARM CE is a hardware extension that implements dedicated instructions to accelerate AES. By measuring electric radiation, they could leak AES keys from an iPhone 7.

Second, cache-based side channels. In 2021, Haas et al. [23] demonstrated the first cache-based side-channel attack on an iPhone 7 containing an Apple A10 CPU. They implemented a Prime+Probe covert channel and showed a practical secret-key extraction on AES T-tables. However, to perform these experiments, they run a custom BootROM toolkit on their device instead of stock iOS.

Finally, in 2022, Hetterich et al. [25] demonstrated that ARMv8-based Apple CPUs are vulnerable to Spectre attacks. Based on Evict+Reload, they successfully implemented a covert channel and a Spectre-PHT attack on devices running an Apple A10 Fusion, Apple A11 Bionic, and Apple M1 CPU.

2.2 Memory Hierarchy

Most modern computers contain a main memory. This memory stores the operating system's code, currently running applications, and the data required by these applications. Modern computers use Random Access Memory (RAM) as the main memory. Operating systems and CPUs implement virtual memory to manage the main memory. Virtual memory acts as an abstraction layer between hardware and operating system. It maps addresses used in software (virtual addresses) into addresses in hardware memory (physical addresses). This layer enables applications to interact with the main memory in a unified way.

2.2.1 CPU Caches and Designs

System RAM cannot keep up with the execution speed of CPUs because accessing data in RAM takes hundreds of CPU cycles. Therefore, modern CPUs improve memory access speed using caches. Caches are small, fast memory units that store data frequently accessed by the CPU. Before loading requested data from memory, the CPU checks the cache for this data. If the data is available in the cache (a cache hit), the CPU loads the data quickly from the cache instead of accessing the slower memory (a cache miss). This approach minimizes the need to fetch data from slower main memory.

Modern CPUs, including those based on the ARMv8 architecture, implement a cache hierarchy. The first cache level (L1) consists of two segments: the data cache and the

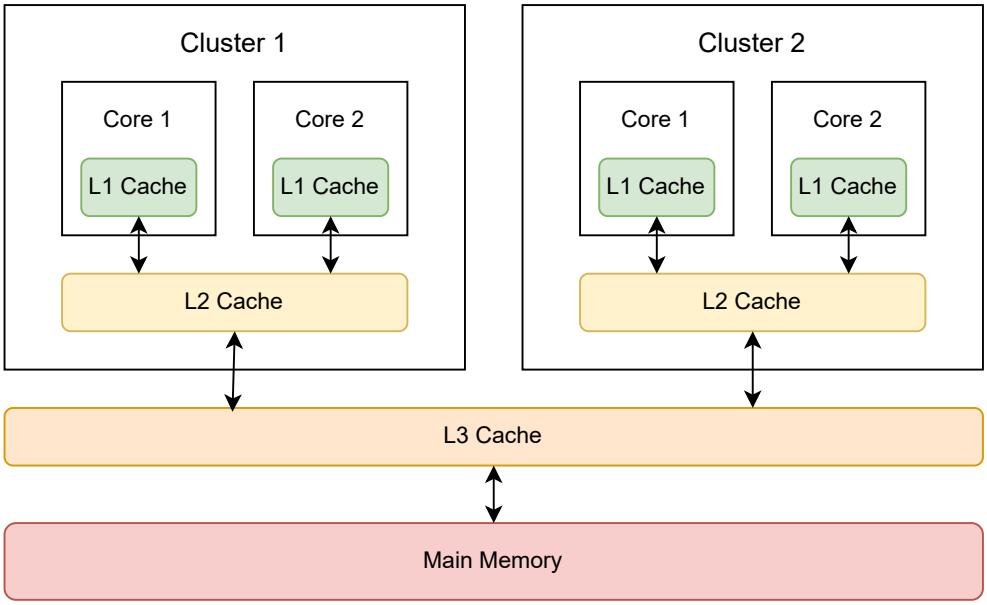


Figure 2.1: The memory hierarchy of a CPU based on the ARMv8 architecture [8].

instruction cache. The data cache stores frequently accessed data, while the instruction cache contains executable instructions. Each core contains its own L1 cache. The following level 2 cache (L2) provides a larger but slightly slower cache memory. The L2 cache gets shared between the cores within the same cluster. Additionally, CPUs based on the ARMv8 architecture can contain an L3 cache. This cache is shared between multiple clusters and is larger and slower than the L2 cache. Figure 2.1 sketches a possible cache hierarchy of an ARMv8 based quad-core processor configured into two clusters [8].

Furthermore, modern CPUs implement special-purpose caches like the Translation Lookaside Buffer (TLB) [24] alongside the described hierarchical data and instruction caches. The TLB increases memory access efficiency by caching recent virtual-to-physical address translations. This caching mechanism can significantly decrease the required amount of page table walks. Page table walks are necessary to resolve the corresponding physical address of a virtual address and are time-consuming. The TLB eliminates the need to repeatedly traverse the multi-level page table structure for frequently accessed memory addresses.

The smallest loadable unit of a cache is called a cache line. In an Apple M1 CPU, an L1 cache line has a size of 64B, and an L2 cache line has a size of 128B [56]. Each cache line has a valid and a dirty bit. The valid bit indicates if the content of a cache line and the corresponding tag are valid. When we modify cached data, the dirty bit gets set, indicating that these changes must be applied to the higher cache levels and the external memory. Additionally, each cache line stores a so-called tag. The tag identifies the way

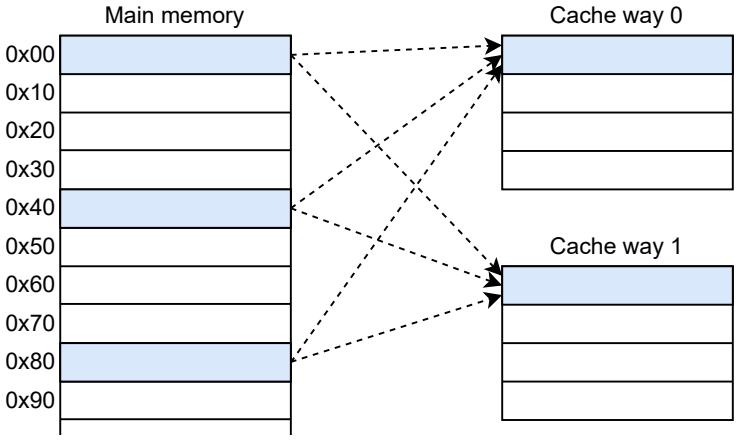


Figure 2.2: A two-way set associative cache with four cache lines per way [8]. Therefore, 0x00, 0x40, and 0x80 get assigned to cache lines of the same set.

containing the stored data. L1 caches are usually *Virtually Indexed, Physically Tagged* (VIPT), while L2 and L3 caches are typically *Physically Indexed, Physically Tagged* (PIPT) caches. In a PIPT cache, we derive the tag and the index from the physical address; in a VIPT cache, we derive the index from the virtual address and the tag from the physical address [24]. The index determines where in the cache a data block can be stored. Cache lines are organized within cache sets. If all lines of a cache set are occupied, the cache replacement policy decides which lines get removed.

Direct-Mapped Caches

Direct-mapped caches implement the most straightforward approach. Each cache line is uniquely mapped to a specific memory block determined by the index. The advantage of this design is that loading from the cache and writing to the cache is cheap. To store data in the cache, we use the index and the tag from the physical address of the data. We store the data and the tag in the corresponding cache line. If this cache line is already occupied, we overwrite it. To load data from a direct-mapped cache, we access the appropriate cache line via the index and check if the tag from the memory address corresponds with the tag from the cache line. However, in practice, direct-mapped caches are inefficient because loading new data into the cache frequently, results in evicting data that is still in use (cache thrashing) [3].

Fully Associative Caches

On the other hand, fully associative caches exist. Any memory block can be mapped to any cache line in a fully associative cache. Compared to direct-mapped caches, this design provides a better hit rate. However, to store data, we must iterate over all cache lines

until we find a cache line with an unset valid bit. In this cache line, we store the data and the tag. Also, to fetch data, we need to iterate over all cache lines until we find the cache line tagged with the desired tag. Therefore, interacting with such a cache is expensive [24].

Set-Associative Caches

A trade-off between the direct-mapped and fully associative cache design is the set-associative cache design. Many modern CPUs, including those based on the ARMv8 architecture [8], implement this approach. Here, the caches get divided into equally sized segments called ways, which consist of multiple cache lines. All cache lines to which a memory block can map form a cache set. A memory location is mapped to a specific cache set and can occupy one arbitrary cache line of this cache set instead of just one line in the whole cache. The index of the specific cache set to which a memory location gets mapped is determined by bits in its virtual or physical address. In Figure 2.2, we see a two-way associative cache. Here, the memory blocks 0x00, 0x40, and 0x80 get assigned to the same cache set.

We need to extract the index and tag from the address to load data from the cache. The index determines the cache set. The extracted tag gets compared with the tag of each valid cache line in the determined set. If a cache line tag matches with the tag extracted from the address, it is a cache hit, and the cached data is returned. If there is not a cache line that matches the extracted tag, it is a cache miss. In this case, the data is loaded from the next cache level or the main memory.

2.2.2 Inclusive, Exclusive, and Non-Inclusive Caches

We distinguish between inclusive, non-inclusive, and exclusive caches. Inclusive caches allow data to be present in different cache levels simultaneously. For example, if we perform a memory access, the CPU checks if the requested data is available in a cache before accessing the RAM. If the data is available in the LLC (Last Level Cache), it gets loaded to the L1 cache. However, this data is still present in the LLC cache. If there are no free cache lines in the L1, the content of an L1 cache line will be replaced with the new data. The replaced data can still be available in the LLC cache. When requested data is unavailable in the L1 and LLC cache, it gets inserted into both caches [8]. On the other hand, exclusive cache designs also exist. In such a design, data can always only be present in one cache level [8]. If a cache is neither inclusive nor exclusive, it is called non-inclusive. Non-inclusive caches do not strictly enforce that data in the L1 cache must be present in the LLC cache. However, data can be present in both the L1 and LLC cache without enforced redundancy [57].

2.2.3 Virtual Memory

When multiple applications run at the same time on a computer, they share hardware resources, like the CPU or main memory. However, this sharing of resources enables memory manipulations. For example, if an application has access to the complete main memory, it could leak or overwrite another application's memory. To prevent such memory manipulations, modern CPUs and operating systems implement the concept of virtual memory. Each process has its own virtual memory area. This virtual memory area is usually larger than the available physical memory. On systems based on the ARMv8 architecture, the virtual memory typically has a size of 2^{48} B, or 256 TB [7]. Processes are usually allowed to use every single byte of their memory area. The operating system and the CPU are responsible for mapping the currently required virtual memory on demand to the physical memory. For this task, the virtual and physical memory gets divided into so-called pages.

Paging

Pages divide virtual and physical memory into segments. Every physical page can be mapped to a page in the virtual memory of one or multiple processes. To manage access to pages, the CPU and operating system store additional information for each page, like whether a page is only accessible from the kernel space or can be read, written, or executed. Virtual memory also allows mapping pages backed by a file on a disk instead of RAM memory. The ARMv8 architecture supports page sizes of 4kB, 16kB, or 64kB [7]. Recent Apple CPUs, like the Apple M1, use 16kB pages [40]. The CPU and the operating system implement a tree-like data structure, the so-called page tables, to manage all pages.

On-Demand Paging

Modern CPUs and operating systems implement on-demand paging to accelerate memory requests and optimize memory usage. This mechanism is triggered when a process requests memory, for example, through `mmap`. Such a request does not immediately allocate additional physical memory or establish a mapping from virtual to physical pages. Instead, the operating system only registers the virtual address in the process's virtual address space. When the process attempts to write to this reserved address, the CPU raises a fault. A fault signals the operating system to allocate the necessary physical memory and create the mapping between the virtual and physical addresses. This approach reduces memory consumption and improves performance by mapping pages only when they get accessed. Furthermore, it allows processes to acquire more virtual memory than the amount of physically available memory.

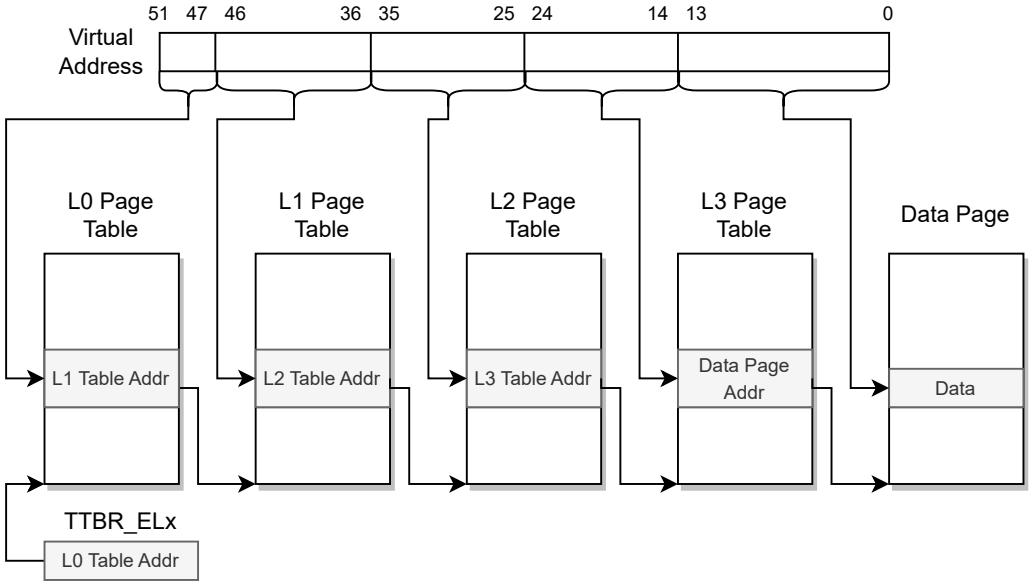


Figure 2.3: Exemplary sketch of a page walk on an ARMv8-A-based CPU with a page size of 16kB [7].

Shared Memory

Furthermore, virtual memory allows mapping a physical page multiple times to the same or different virtual memory spaces. Typically, shared libraries and binaries get mapped by the operating system as shared memory. Thus, when a binary gets executed multiple times, it is not loaded multiple times to the physical memory. This approach reduces physical memory consumption. Shared memory can also be used to exchange data efficiently between processes.

Page Table Structure

A tree-like structure called page tables is implemented to manage the mapping between virtual and physical memory addresses. The entries of the page tables consist of the physical address of the next page table in the hierarchy. Each entry contains additional bits that store permissions and access information (e.g., accessed, dirty) about the corresponding page. In the context of 64-bit ARMv8-based CPUs, these page tables can be organized into either a three-level or four-level structure [7].

Page Walk

When a virtual address requires translation to a physical address, the CPU needs to perform a page walk. The starting point of a page walk is the Translation Table Base Register (TTBR), which contains the base address of the root page table. To compute the required page table entry, the CPU extracts the table offset from the virtual address and

```
1  void* counter_thread(void* ctx){
2    __asm__ volatile(
3      "LDR x10, [%[counter]]\n"   // load counter once
4      "1:\n"                      // while(true) {
5        "ADD x10, x10, #1\n"      //     increment counter
6        "STR x10, [%[counter]]\n" //     store counter to memory
7        "B 1b\n"                  // }
8    :: [counter] "r" (ctx)        // read ctx (pointer to counter)
9    : "x10", "memory");           // writes x10 (holds counter value)
10   return NULL;
11 }
```

Listing 1: The source code of the counting thread used by Hetterich et al. [25] to perform Spectre attacks on Apple devices.

adds it to the table's base address. Each entry of a page table points to the base address of the following page table in the hierarchy or to a physical page in the case of the final page table. Figure 2.3 illustrates a four-level page walk on a CPU based on the ARMv8 architecture. Page walks are computationally expensive, and every memory operation requires a page walk. To optimize the translation process, modern CPUs implement a Translation Lookaside Buffer (TLB), which caches frequently accessed memory addresses and reduces the need for repeated page walks.

2.3 Measuring Time in Restricted Environments

To measure the execution time of single instructions, a high-resolution timer is needed. On x86, we can use the instruction `RDTSCP` to read the processor's timestamp counter [27]. Similarly, the ARMv8 platform exposes various timers to the user, like the `PMCCNTR_ELx` performance counters, which Lipp et al. [35] used to mount cache attacks on Android devices. Furthermore, ARM exposes the system counter registers `CNTPCT_ELx` and `CNTVCT_ELx` [8], which can be used to measure execution times. Besides the system and performance counter register, iOS and macOS provide library functions that return a timestamp. Both operating systems implement `mach_absolute_time` and `clock_gettime`.

Hetterich et al. [25] evaluated various timers on modern Apple CPUs and concluded that they are unsuitable for cache attacks. Reading the `PMCCNTR_ELx` performance counter as an unprivileged user raises an illegal-instruction exception. `CNTPCT_EL0` is readable, but the resolution is too low to measure a single instruction's duration. `CNTVCT_EL0` can be accessed by unprivileged users. However, it seems to be the same counter as `CNTPCT_EL0`. The library functions `mach_absolute_time` and `clock_gettime` use the system counter

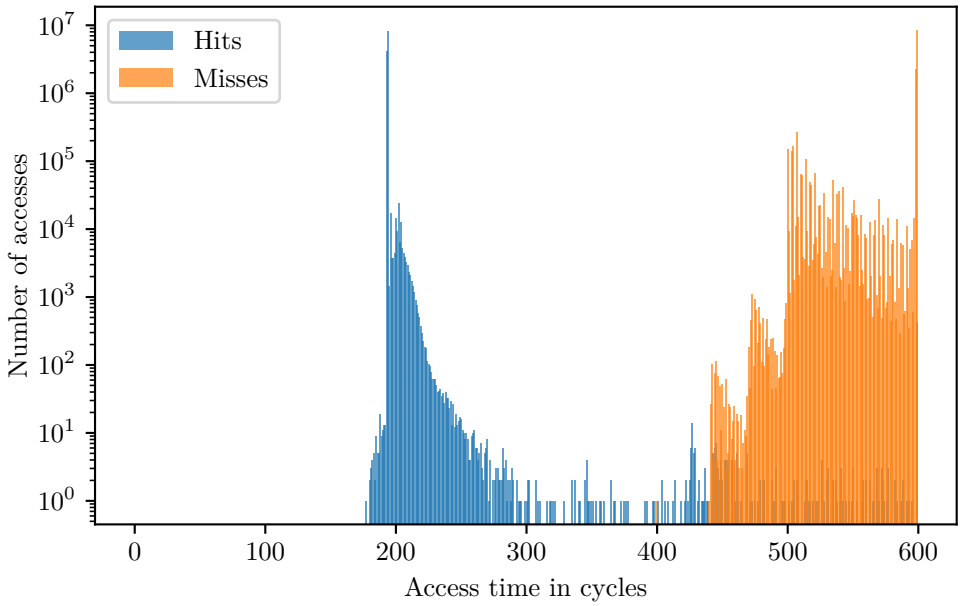


Figure 2.4: Cache hit (blue) and cache miss (orange) histogram on an AMD Ryzen 7 PRO 4750U.

registers internally. They provide the same values as when accessing the registers directly.

If no suitable timer is available, we can use a counting thread as an alternative to a traditional timer. This concept was already introduced in 1992 by Wray et al. [54]. A counting thread is a dedicated thread that runs an endless loop. In this loop, a globally accessible variable is incremented in each iteration. A timestamp is obtained by reading the current value of this variable. Hetterich et al. [25] use a counting thread to perform a Spectre attack on recent Apple CPUs. The source code of their counting thread is shown in Listing 1. Nevertheless, a counting thread does not always provide accurate timestamps compared to traditional timers. If other processes run on the same core as the counting thread, it happens that the counter does not get updated with a constant frequency because the counter thread is not scheduled, and does not get enough computation time. Additionally, the CPU frequency significantly impacts the accuracy of a counting thread. If the CPU frequency is unstable, the counter update frequency also will not be stable as we show in Chapter 5.

2.4 Cache Attacks

Cache attacks exploit the timing differences between accessing data from the cache and retrieving data not present in the cache [44]. Figure 2.4 illustrates the timing difference between a cache hit and a cache miss on an AMD Ryzen 7 PRO 4750U processor.

Through these timing variations, attackers can deduce the specific cache lines or cache sets an application accesses. This information enables them to extract sensitive data from other applications running on the same system or establish hidden communication channels (so-called covert channels) between two processes.

We divide cache attacks into two categories. The first category requires shared memory between victim and attacker. Prominent attacks from this category are Flush+Reload [55], Evict+Time [21], and Flush+Flush [20]. These attacks share the common requirement of being able to evict data from the cache. Many CPUs provide a flush instruction to unprivileged users. If this is not the case, we can remove data by eviction from the cache. Conversely, attacks like Prime+Probe [42] do not require shared memory. Prime+Probe can be mounted in more constrained environments but cannot be used to obtain set-based information [37].

2.4.1 Flush+Reload

The ARMv8 Programmer's Guide [8] lists several instructions for flushing lines from the data and instruction caches. These instructions use one of two approaches: Overwriting the cached data with zeros or modifying the *valid* and *dirty* flag of the cache line that contains the data to be flushed. By setting the *valid* flag of a cache line to 0, we mark the content of this line as invalid, and on the subsequent access, the data gets loaded from a lower cache or the memory. To propagate this flag modification to the lower cache levels, we need to set the *dirty* flag to 1. The *dirty* flag marks a cache line as modified, and on the subsequent access of this line, all the flag changes get applied to the lower cache level. So, the cache line also gets marked as invalid in the lower cache level. To perform cache attacks on ARMv8-based CPUs, the `DC CIVAC` instruction is usually used. This instruction takes a virtual address as an argument and removes the corresponding data from all cache levels [35].

Based on flush instructions, a well-known cache attack is Flush+Reload [55]. Loading data from the cache takes less time than loading it from the main memory. By observing memory access times, an attacker can deduce information about which data a victim process currently accesses. For a Flush+Reload attack, the victim and the attacker map a shared memory region into their virtual address space (e.g., by using `mmap`). This memory can be, e.g., a shared library or a binary in a real-world attack. As shown in Figure 2.5, when memory gets mapped as shared in the virtual address space of two different processes, both mappings resolve to the same physical memory. Thus, it maps to the same cache lines. To perform a Flush+Reload attack needs the ability to flush data from the cache (e.g., `DC CIVAC`) and a high-resolution timer [55] to measure cache hits and misses.

By repeatedly flushing the cache and reloading data from the shared memory, the attacker checks which regions of the shared memory the victim accesses. The attacker measures the time needed to reload the data. If reloading is fast, the data is within the cache,

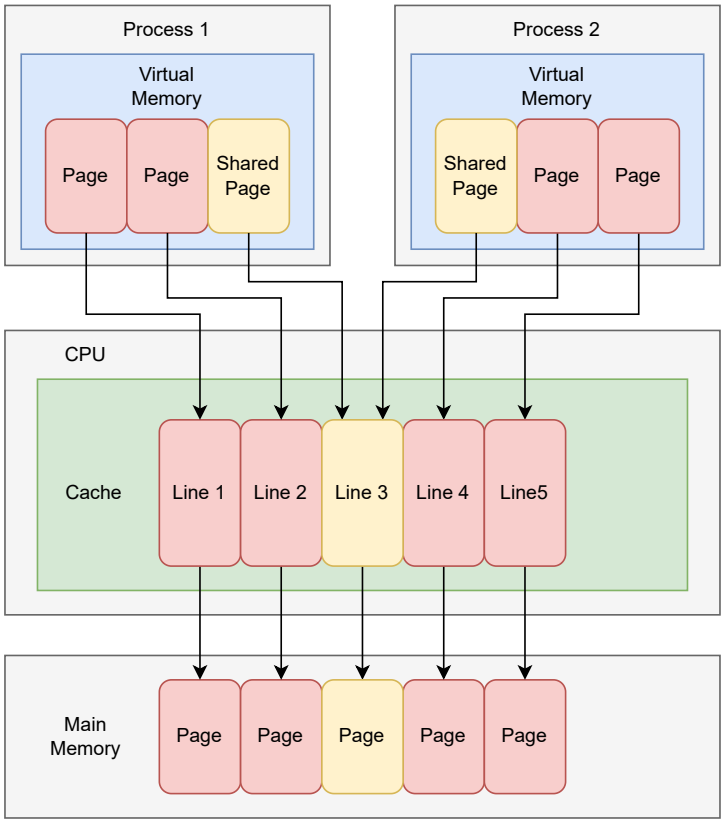


Figure 2.5: When two processes map the same memory page as shared in their virtual address space, both gain access to the same physical memory. Cache attacks like Flush+Reload exploit that this physical memory gets mapped to the same cache lines, regardless of which process accesses it.

meaning the victim accessed this data after the attacker flushed it previously from the cache. If the measured access time is high, the victim has not accessed this data since the attacker flushed it from the cache. This information is sufficient to spy on users or recover cryptographic keys. Flush+Reload attacks have been demonstrated to be effective in such scenarios, as shown by various researchers [21, 55, 19].

However, there are ARM-based CPUs where the `DC CIVAC` instruction does not work, e.g., on Apple CPUs. Although unprivileged users can execute this instruction, it does not remove the desired data from the cache. The ARMv8 Programmer's Guide [8] specifies also other flush instructions. Hetterich et al. [25] investigated these instructions and concluded that, although they are executable on Apple CPUs without exceptions, they fail silently and do not remove the desired data from the cache. Consequently, flush instructions cannot be used to implement cache template attacks on devices powered by

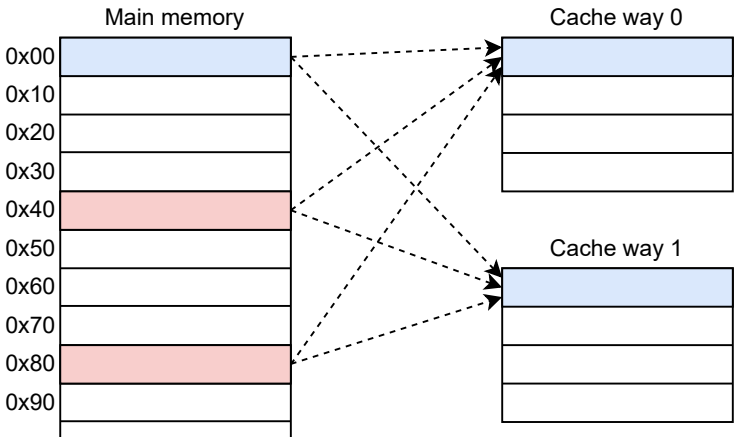


Figure 2.6: We need to find other physical addresses that map to the same cache lines to evict data from the cache. In this example, the physical addresses 0x40 and 0x80 form a potential eviction set to evict the data located at 0x00 in the physical memory from the cache.

Apple CPUs.

2.4.2 Evict+Reload

Instead of using flush instructions, we can evict a cache line by accessing selected virtual addresses that map to the same cache set. All required virtual addresses together form a so-called eviction set. In Figure 2.6, the virtual addresses mapping to the physical addresses 0x40 and 0x80 would form a potential eviction set to evict the data located at 0x00 in the physical memory from the cache.

In practice, generating eviction sets needs to be both fast and reliable. Generating eviction sets is straightforward when the mapping from virtual-to-physical addresses is accessible. In a last-level cache, parts of the physical address determine the set index. On Linux and older Android devices, this information is accessible through the file `/proc/self/pagemap`. Lipp et al. [35] used this approach to perform cache attacks on mobile devices. The virtual-to-physical mapping is not exposed to the user on macOS, iOS, and newer Android systems. However, we can use an approach based on group testing introduced by Vila et al. [52] to generate eviction sets efficiently. The original implementation of this approach is designed for the x86 architecture. Hetterich et al. [25] modified this implementation to work on the ARM platform.

The algorithm works as follows: First, the group testing approach takes a large set of virtual addresses, evicting the desired data from the cache. Second, we remove random addresses of this set. After each removal, we check if the eviction set still evicts

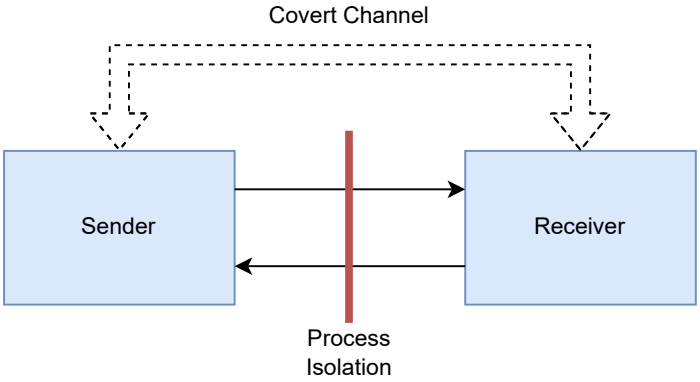


Figure 2.7: Covert channels allow to circumvent process isolation.

the desired data from the cache. Finally, we get an eviction set of minimal size. To generate eviction sets, we need the ability to distinguish cache hits from cache misses. When we access an address two times successively, the second access will likely be a cache hit. To measure cache misses, we need to allocate a new memory page. This page is typically not cached. However, if we now measure the access time, we would also measure the virtual-to-physical address translation and the time needed to fetch the data into RAM. To circumvent this issue, we need to access the first cache line of the allocated page first to load the address translation to the TLB. Now, accessing a different cache line of this page will result in a cache miss. Furthermore, we need a reliable high-resolution timer to perform the time measurements, as we discuss in Section 2.3.

In some environments, where flush instructions are not accessible by the user, an Evict+Reload attack can be executed [21]. Evict+Reload attacks work similarly to Flush+Reload attacks described in Section 2.4.1. However, we do not need a flush instruction. Instead of flushing the cache, we generate an eviction set for the targeted cache line as described above. By using the eviction set, we evict the data similar to a flush. When eviction sets are created using a reliable timer, this approach can achieve reliability comparable to the Flush+Reload technique [37].

2.5 Covert Channel

In computing, a communication channel transmits information from one or several senders to one ore several receivers [34]. According to Lampson [32], a covert channel is a communication channel that is not intended for information transfer. Such a covert channel can be established via a CPU cache using cache attacks. A CPU cache-based covert channel circumvents mechanisms like process isolation and allows an unprivileged user to exchange data between two processes running on the same system [53]. As shown in Figure 2.7, one process acts as the sender and the other as the receiver. Both processes need to be under the control of the same user. Common cache attack primitives like

Flush+Reload can be used to set up a covert channel [55].

To create a covert channel based on the Flush+Reload primitive, we need a flush instruction, a timer, and memory, which is readable and shared between the sender and receiver [55]. When these conditions are met, we can build a simple 1-bit covert channel. Before transmitting data, the sender and receiver must agree on how to encode the data bits to be sent. For example, the bit 1 can be transmitted by producing cache hits, and the bit 0 by cache misses. Additionally, we need to determine how long an average cache hit and cache miss takes on the current system. From this information, we can determine the threshold the receiver requires to interpret the measured access times. The sender and receiver can synchronize the transmission using a shared clock, such as the `rdtsc` register on x86 [27], which provides both actors with the same timestamp t. The sender application starts transmitting data at time t by accessing the shared memory at a predefined offset or idling for a static time interval i. In parallel, the receiver starts measuring at time t how long it takes to access the shared memory at the predefined offset. Subsequently, after each access, the receiver flushes the accessed data from the cache. The receiver performs these two steps for the time interval i. Finally, the receiver computes the average measured access time during the last interval and compares it to a previously determined threshold. If the average access time is lower than the threshold, the sender produced accesses during the last interval, and the receiver measured cache hits. If the average access time surpasses the threshold, the sender idled, and the receiver measured cache misses.

We usually use three parameters to describe the performance of a covert channel: raw capacity C, bit error rate p, and true capacity T [46]. To compute the raw capacity, we need the total number of bits sent b_s and the transmission duration t in seconds. The raw capacity describes how many bits are transmitted per second and is given by

$$C = \frac{b_s}{t}.$$

The bit error rate p indicates how reliably a covert channel works. To compute p, we need the total number of bits sent b_s, the total number of bits received b_r, and the number of incorrectly transmitted bits w. We can compute p using the following formula:

$$p = \frac{w + |b_r - b_s|}{max(b_s, b_r)}.$$

The true capacity T combines the raw capacity C and the bit error rate p. We can compute it by

$$T = C \cdot (1 + ((1-p) \cdot \log_2(1-p) + p \cdot \log_2(p))).$$

Today's state-of-the-art covert channels [37] implement packet protocols with error correction to prevent data corruption. These covert channels use multiple bits to transfer data

Table 2.1: The performance of the covert channel from Hetterich et al. [25] on different Apple devices.

Device	Raw Capacity	Error Rate	True Capacity
iPhone 8 Plus	2.4 kbit/s	7.84 %	1.448 kbit/s
M1 Mac mini	24 kbit/s	12.67 %	10.840 kbit/s

Algorithm 1 Cache template attack profiling phase

Input: victim binary b, duration d, threshold T
Output: hits per offset $p = [(\text{offset, hits})]$

```
1: address f ← map b as shared into virtual memory
2: for i ← 0 to size(b) do
3:     hits h ← 0
4:     time t ← current time
5:     while current time < (t + d) do
6:         if maccess(f + i[offset]) < T then
7:             h+ = 1
8:         end if
9:     end while
10:    store (i, h) in p
11: end for
12: return p
```

simultaneously to speed up the connection. With such improvements, Gruss et al. [20] created a covert channel based on the Flush+Reload attack with a raw capacity of 2384 kbit/s and an error rate of 0.005 %.

Hetterich et al. [25] created covert channels based on the Evict+Reload attack on Apple devices. As listed in Table 2.1, they successfully establish a covert channel on an M1 Mac mini with a raw capacity of 24 kbit/s and an error rate of 12.67 % and on an iPhone 8 Plus with a raw capacity of 2.4 kbit/s and an error rate of 7.84 %. Their covert channel on the M1 Mac mini is, as of this writing, the fastest-known Evict+Reload covert channel on an ARM-based CPU. However, they could not establish a covert channel on an iPhone 7 because only two cores can be active simultaneously on such a device, and no simultaneous multithreading is available [25].

2.6 Cache Template Attack: Side Channel

In 2006, Osvik et al. [42] demonstrated that the Prime+Probe cache vulnerability could be used to leak cryptographic keys from applications running on the same CPU. In 2015, Gruss et al. [21] introduced cache template attacks. Cache template attacks generalize

the approach from Osvik et al. [42]. By using the Flush+Reload primitive, they were able not only to leak cryptographic keys but also to demonstrate that cache vulnerabilities could be used to leak any event (e.g., keyboard inputs) from any application running on any CPU that implements shared, inclusive last-level caches. In 2016, Lipp et al. [35] demonstrated that cache template attacks are not limited to x86 CPUs. They successfully leaked various events, such as keystrokes and touchscreen taps, on non-rooted Android devices powered by ARM-based CPUs. A cache template attack consists of two phases: the profiling phase and the exploitation phase.

2.6.1 Profiling Phase

During the profiling phase, the attacker templates the cache activity. The attacker analyzes which cache lines are accessed when the victim application performs a specific action (e.g., handling a key press). The profiling phase can be performed directly on the victim's device or on a device with similar hardware and software specifications. Algorithm 1 summarizes the profiling phase. The attacker maps the victim binary in his virtual memory as shared, iterates over the mapped binary, and determines how many cache hits occur for each address in the binary using a cache attack primitive, such as Flush+Reload. In parallel, the attacker repeatedly performs the specific action he wants to template. Addresses where the attacker measures many hits point to code executed recently. We store the binary offset of each address along with the corresponding number of hits. By repeating this step multiple times and analyzing the recorded hits, we observe which addresses are specifically required to handle the action performed by the victim application. If we want to detect different events later during the exploitation phase (e.g., different keystrokes), we must profile each event with its own profiling phase. With this knowledge, the attacker switches to the exploitation phase.

2.6.2 Exploitation Phase

During the exploitation phase, the attacker monitors the number of hits on the binary offsets identified as required to perform the action profiled during the profiling phase. In contrast to the profiling phase, the attacker must perform the exploitation phase on the victim's device. The attacker process maps the binary of the victim application as shared memory to its virtual memory.

Then, the attacker iterates in a loop over the offsets where many accesses were registered during the profiling phase. For each of these offsets, the attacker performs the same cache attack primitive for the same time interval t as during the profiling phase. After each loop iteration, the attacker checks if a similar number of cache hits were registered on all required offsets, as observed during the profiling phase. If that is the case, the victim process most likely handled the profiled action. Algorithm 2 summarizes the exploitation phase and allows us to detect a single event. To detect multiple events, we can compare the number of hits per offset e in line 14 with different templates from the profiling phase.

Algorithm 2 Cache template attack exploitation phase

Input: victim binary b, hits per offset $p = [(\text{offset, hits})]$, duration d, threshold T

address $f \leftarrow$ map b as shared into virtual memory
while $True$ **do**
    hits per offset $e \leftarrow [(\text{offset, hits})]$
    for all i **in** p **do**
        hits $h \leftarrow 0$
        time $t \leftarrow$ current time
        while current time $< (t + d)$ **do**
            if maccess$(f + i[\text{offset}]) < T$ **then**
                $h+ = 1$
            end if
        end while
        store $(i[\text{offset}], h)$ in e
    end for
    if e similar to p **then**
        Victim likely performed templated action
    end if
end while

2.7 Machine Learning

Machine Learning problems usually try to find parameters W to create a model $g(W)$, which describes the distribution of a given dataset. We can create a model by optimizing its parameters W to minimize a cost or error function. There are mainly two approaches to tackle machine learning problems: supervised learning and unsupervised learning. Supervised learning tries to create a model based on N input-output pairs $\{(x_i, y_i)\}_{i=1}^{N}$ (so-called labeled data), which allows assigning unlabeled inputs x to their correct labels y. On the other hand, unsupervised learning does not require labeled data. It tries to find patterns and relations in unlabeled data $\{(x_i)\}_{i=1}^{N}$ [26]. An example of an unsupervised machine learning approach is the k-means clustering algorithm. In the following sections, we will focus on supervised learning.

2.7.1 Bayes Classifier

The Bayes classifier [39] is a classification algorithm that allows us to compute the posterior probability $P(y|x)$, which describes how likely an input x corresponds to class y [45]. The posterior probability can be expressed by the Bayes rule, which is denoted by

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)}.$$

We need labeled data to estimate the prior probability $P(y)$ and the parameters required to compute the likelihood term $P(x|y)$. The prior probability $P(y)$ describes the proba-

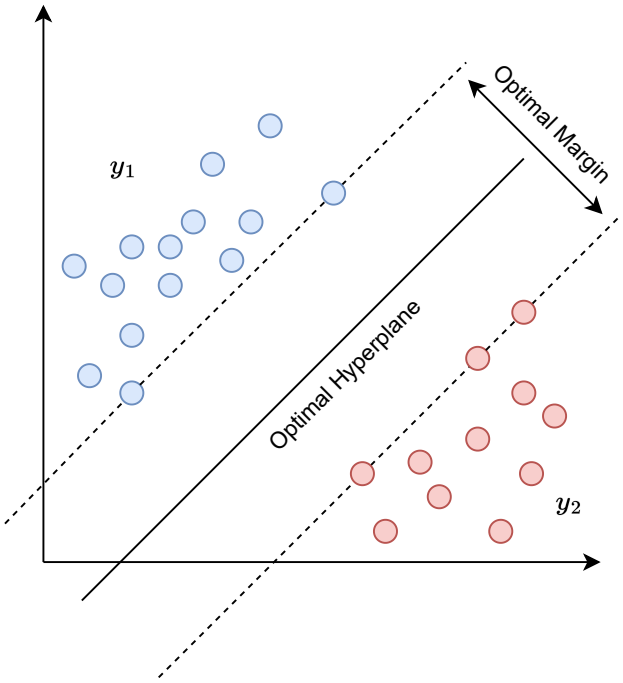


Figure 2.8: Support Vector Machines (SVM) can be used to classify linearly separable data [26].

bility of observing a particular label y before considering any evidence from the input data x. $P(y)$ can be estimated by dividing the number of data samples x, labeled with class y, by N. We must also determine the parameters that define our likelihood term $P(x|y)$. The likelihood term describes the probability of observing the input data x given a class y. We must choose a distribution that fits our data well to model the likelihood term. For example, if the data in each class is Gaussian distributed, we can model the likelihood term as $P(x_i|y) = N(x_i|\mu_y, \Sigma_y)$, where μ_y and Σ_y are the mean vector and covariance matrix for class y. To estimate the parameters μ_y and Σ_y, we can apply statistical techniques such as maximum likelihood estimation to our labeled data [45].

Using the Bayes rule for data classification, we can neglect the so-called evidence term $P(x)$ because it is independent of all labels y and, thus, has the same value for all labels. To classify data x, we assign the class y^* with the largest posterior probability [45]. So, we denote our classifier as

$$y^* = arg\max_y(P(y|x)) = arg\max_y(P(x|y)P(y)).$$

2.7.2 Support Vector Machine

Support Vector Machines (SVMs) also belong to the family of supervised learning algorithms. Figure 2.8 shows how SVMs create an optimal hyperplane between data

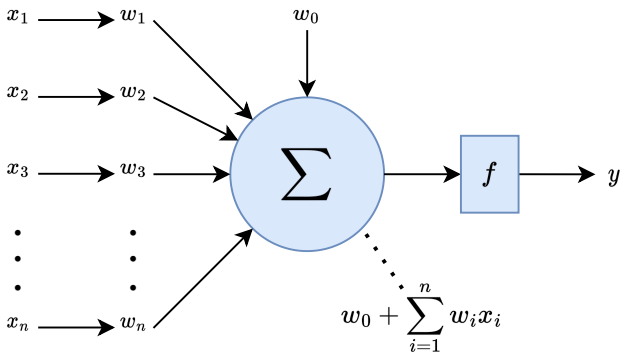


Figure 2.9: A single perceptron [26].

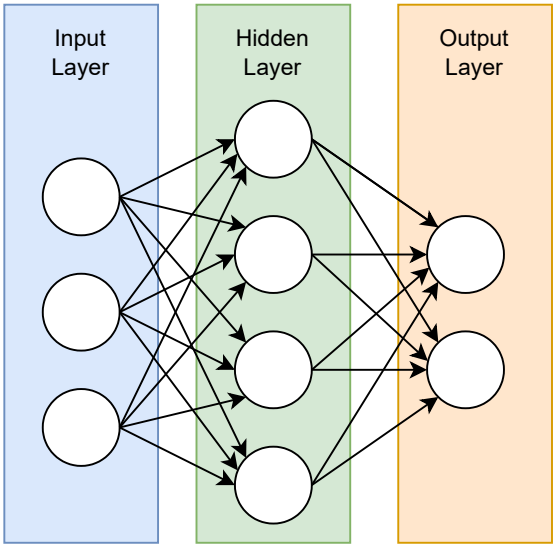


Figure 2.10: A multi-layer perceptron (MLP) with one hidden layer [26].

points belonging to two linearly separable classes, y_1 and y_2. Based on a weight vector w and a bias b, we denote the prediction function of the model by $w^T x + b$. This function predicts a positive value if the inputted data belongs to class y_1 and a negative value if it belongs to y_2 [26]. Furthermore, SVMs allow the classification of non-linear separable data by applying the kernel trick. The kernel trick projects the data in a higher-dimensional space, where the data becomes linearly separable again. For the kernel trick, it is essential to choose an appropriate kernel function [26].

2.7.3 Multi-Layer Perceptron Classifier

A Multi-layer Perceptron (MLP) is a neural network that can learn a non-linear classification function by training on labeled data. An MLP comprises many perceptrons stacked together to form a network [26]. A single perceptron, as shown in Figure 2.9, allows

the classification of linearly separable data. For this task, a perceptron computes the weighted sum $w_0 + \sum_{i=1}^{n} w_i x_i$. Here, w_0 denotes the bias weight of the perceptron, and $w = \{w_1, ..., w_n\}$ represents the weight of each input $x = \{x_1, ..., x_n\}$. The weighted sum is fed into an activation function f, such as the ReLU function $f(x) = max(0, x)$ [45]. The output of this activation function is the predicted label y [26].

Compared to a single perceptron, an MLP can classify linearly non-separable data. As shown in Figure 2.10, an MLP consists of different layers. The input layer represents the raw input data. Each neuron in the input layer is connected to each neuron in the first hidden layer. In Figure 2.10, there is one hidden layer, but real-world MLPs often have multiple hidden layers stacked together. The number of hidden layers required depends on the complexity of the function to learn. Using too many layers causes overfitting, while too few layers lead to underfitting. So, an appropriate amount of layers is crucial for good classification performance. The output layer directly maps to the given prediction labels [26].

We train a Multi-layer Perceptron with our labeled data $\{(x_i, y_i)\}_{i=1}^{N}$ by applying the gradient descent algorithm [38]. The gradient descent algorithm is a widely used iterative optimization technique in machine learning to minimize the error of a model. We can roughly divide the gradient descent algorithm into the following steps:

1. Initialization: Initialize all weights with random values.
2. Forward pass: Execute the MLP with our labeled training data as input and use an error function to compute the difference between the expected and obtained outputs.
3. Backpropagation: Compute the gradient of the error function for each weight in the network. Start at the output layer and propagate the error backward through the network.
4. Update weights: To minimize the error, adjust the weights in the opposite direction of the gradient.
5. Repeat until convergence: Repeat steps 2-4 until the desired accuracy is achieved or the maximum allowed number of iterations is reached.

Chapter 3

Attack Scenario

In this section, we present the devices on which we evaluate the covert channel and the cache template attack. We briefly introduce the devices we use to run our experiments in Section 3.1. Furthermore, we discuss the differences between the CPUs of these devices. Although we evaluate mobile and desktop CPUs, they work quite similarly internally. In Section 3.2, we discuss the operating systems running on our devices and their corresponding security features. Finally, in Section 3.3, we present our threat model for the covert channel and cache template attack.

3.1 Devices

We chose an Apple iPhone 13 mini and an Apple Mac mini (M1) as evaluation devices for the covert channel. We perform the cache template attack only on the Mac mini because, on the iPhone, the file system does not allow access to any shared libraries or application binaries. Thus, there are no rewarding attack targets for cache template attacks accessible on this platform. The iPhone 13 mini, released in 2021, is powered by an Apple A15 Bionic CPU based on the ARMv8 architecture. The A15 Bionic CPU consists of four energy-efficient cores, based on the so-called *Blizzard* microarchitecture, and two high-performance cores, based on the *Avalanche* microarchitecture. These cores are grouped into two clusters: the energy-efficient cores form the first cluster, and the high-performance cores form the second cluster. The Mac mini contains an Apple M1 CPU, released in 2020. Like the A15 Bionic, the M1 CPU is based on the ARMv8 architecture and groups its cores into two clusters. The first cluster consists of four energy-efficient cores based on the *Icestorm* microarchitecture. The second cluster contains four high-performance cores based on the so-called *Firestorm* microarchitecture. Since both Apple's desktop and mobile CPUs are based on the ARMv8 architecture, they share several characteristics. Both CPUs cannot adjust the clock frequency for each core separately. They can modify the clock frequency only for the entire cluster [8]. In both CPUs, each core has its own L1 cache. All cores within a cluster share an L2 cache, and the clusters are connected via a so-called system-level cache [8]. Recent mobile and desktop CPUs from Apple also seem to implement similar optimizations, such as data memory-dependent prefetchers [51]. Even speculative execution attacks, like Hetterich et al.'s [25] Spectre exploit, work on both desktop and mobile CPUs without major

modifications. Although our selected devices and CPUs are not the most recent, they are still widely used, making them valid attack targets.

3.2 Operating Systems

This Section briefly presents the operating systems we run on our devices. Each operating system provides different user interfaces and implements various security features designed to prevent attackers from mounting attacks. Our Apple iPhone 13 mini runs the mobile operating system iOS 15.5, released on May 16, 2022. Our Apple Mac mini (M1) runs macOS Monterey (version 12.5) out of the box. This macOS version was released on July 20, 2022. We also installed the Linux-based operating system Asahi Linux on the Apple Mac mini (M1). We used this operating system only during the development of the cache template attack. Compared to Apple's macOS, Asahi Linux offers more facilities and interfaces to debug the system and control the hardware.

3.2.1 Mobile Operating System iOS

Apple's iOS is a proprietary operating system for mobile devices, first launched in 2007. Apple initially developed this operating system for the first iPhone under the name iPhone OS. In 2010, they renamed it to iOS. In addition to the iPhone, Apple shipped all iPads to the iPad 7 with iOS [33]. Starting in 2019, Apple began shipping iPad tablets with iPadOS [4]. In 2022, iOS was the second most popular mobile operating system behind Google's Android, with a market share of 27.6 % [47].

OS Design and Security

Apple's iOS is based on the open-source operating system Darwin, which builds on the XNU kernel. The XNU kernel is a hybrid kernel, combining parts from the Mach kernel and FreeBSD [14]. The Darwin operating system is fully compatible with the POSIX standard. However, in iOS, some POSIX syscalls, like `fork`, are callable from user-space applications but are not functional for security reasons [28].

From the early days, security has been a major priority in the development process of iOS. Initially, it was impossible to install third-party applications on iOS devices. Since 2008, Apple has provided a software development kit (SDK) that allows third-party software developers to create their applications [33]. Apple must approve and sign these applications, and users can install them only via the official App Store. When developers want to test an app on a device, they must sign the application with their certificate within Apple's Xcode IDE. Apps signed with a developer certificate can only be installed on devices linked to the same developer account as the certificate.

Additionally, the iOS operating system implements various security features such as application sandboxing, which limits file system access, secure boot, and app signature verification. Apple allows only the installation of verified updates, and iOS version

downgrades are impossible [49]. In practice, these measures are very effective. Accordingly to CVE Details [12], 2854 vulnerabilities have been reported from 2009 to the beginning of 2023.

3.2.2 Desktop Operating System macOS

Apple's macOS (previously known as Mac OS X and OS X) has been the company's desktop operating system since 1998. Current versions support Apple's ARM-based CPUs and common x86-64 CPUs. Apple ships macOS only in combination with their devices. With few exceptions, installing macOS on an arbitrary x86-64 system is not possible. This limitation results in a small market share, but macOS is still the second most popular desktop operating system. According to Statista [48], in January 2023, 15.33 % of all personal computers ran macOS. On our Mac mini (M1) device, we ran macOS Monterey (version 12.5).

OS Design and Security

Like iOS, macOS is also based on the Darwin operating system and XNU kernel. However, macOS is fully UNIX-03 compatible [41], implying it also implements the full POSIX standard. Recent macOS versions implement security features similar to modern operating systems, such as process isolation and ASLR. Similar to iOS devices, macOS devices use a secure boot process to verify the integrity of the operating system and drivers [49]. Unlike iOS, macOS allows the execution of unsigned applications from third-party sources in addition to those from the official App Store. In recent macOS versions, applications from third-party sources get automatically scanned for malicious code during installation. This feature is called *Gatekeeper* [6]. The macOS operating system tries to reduce the attack surface by minimizing available interfaces. For example, access to system files is limited; shared system libraries are not accessible via the file system. In practice, these features do not significantly increase security: For 2022, CVE Details [12] lists 379 reported vulnerabilities. The same year, Linux was affected only by 307 security flaws.

3.2.3 Asahi Linux

Asahi Linux is a community-driven project started in 2020 that aims to bring Linux to Apple Silicon Macs and to reverse engineer and document Apple's self-designed hardware. In 2022, they released the first bootable alpha version based on Arch Linux ARM [9]. There are no official numbers about the market share of Asahi Linux, but it is likely to be a negligible amount. Therefore, it cannot be considered a worthwhile attack target. We used this operating system to test our attack concepts because it allows us to obtain more debug information and better control the hardware than macOS (e.g., binding threads to cores or setting a static CPU frequency). The Asahi Linux alpha version we used is built on Linux kernel version 6.1.0. Thus, Asahi Linux includes all the default

security features implemented in modern Linux kernels.

3.3 Threat Model

As target devices for the covert channel, we consider a Mac mini (M1) running macOS Monterey (version 12.5) and an iPhone 13 mini running iOS 15.5. Our attack scenario assumes that the user installs two applications on a device. These applications want to exchange data with each other. However, they run in a restricted environment and cannot create a shared resource (e.g., shared memory) with write permissions. The process isolation provided by virtual memory prevents them from exchanging data via distinct interfaces like shared memory or memory sockets. However, the applications can map a shared memory chunk with read permissions. This memory chunk allows the two applications to circumvent process isolation and exchange data by establishing a covert channel via the CPU cache. One application acts as the sender by encoding data with memory accesses, while the other application, the receiver, decodes this information by measuring memory access times.

We consider a Mac mini (M1) running macOS Monterey (version 12.5) as a threat model for the cache template attack. As described in Section 3.2.1, we do not consider the iPhone as a threat model because iOS executes applications in a sandboxed environment. Therefore, applications can only access files created by themselves or other applications from the same developer. This security measure prevents mapping other application binaries or shared libraries into the virtual memory of an application on an iPhone. A potential attack scenario on a Mac mini (M1) could look like this: The user runs an inconspicuous but malicious app downloaded from the internet on their Mac. The user executes this app in the background and browses the internet using Mozilla Firefox on their Mac. While the user browses the internet, the malicious app performs a cache template attack. This attack enables the malicious app to detect events, such as the user opening a new tab in Firefox, or to template websites, and consequently leak which websites the user visits.

Chapter 4

High-Level Overview

This chapter provides a high-level overview of our covert channel and cache template attack implementation. We discuss the cache attack primitive used to establish our covert channel, the transmission protocol, and how the sender and receiver applications synchronize the transmission in Section 4.1. In Section 4.2, we describe how we implemented a cache template attack on the same primitives as the covert channel and give an overview of our profiling and exploitation phase.

4.1 Covert-Channel

Hetterich et al. [25] established a covert channel via the CPU cache on an Apple Mac mini and an iPhone 8 Plus. This covert channel allows them to circumvent memory isolation between two processes by exploiting timing differences in memory read accesses. Figure 4.1 provides a high-level overview of such a covert channel. In our work, we improved the performance and ported the covert channel from Hetterich et al. [25] to an iPhone 13 mini. To interpret the measured timings, we use machine learning classifiers.

As described in Section 3.3, we must run two applications on the same device to set up a covert channel. The first application acts as the sender, and the second as the receiver. Both applications map the same file only with read permissions to their virtual memory. The sender application transmits data to the receiver process by producing memory accesses on the shared read-only memory. The receiver obtains this data by performing the Evict+Reload primitive, presented in Section 2.4.2. The sender and receiver must agree on a transmission protocol to exchange data successfully. For example, they need to specify which offsets in the shared memory should be used for transmission and how the apps should be synchronized.

For synchronization, the covert channel requires a timer accessible by both the sender and the receiver. With this timer, the sender and receiver can determine when a bit's transmission starts and ends. The system counter register `CNTVCT_EL0`, described in Section 2.3, provides timestamps with sufficient resolution for this task. The transmission of a bit itself takes a predefined duration t.

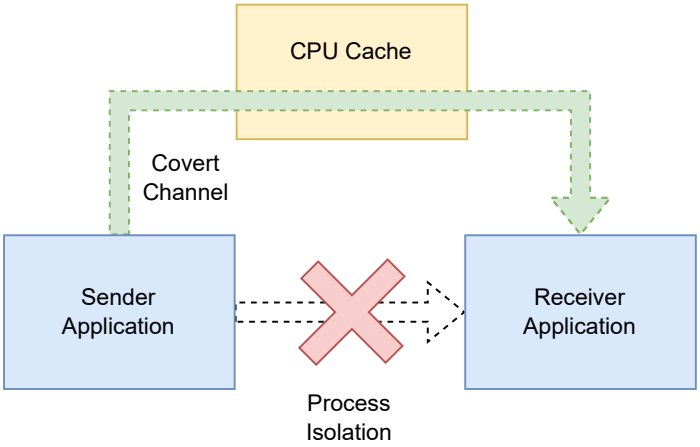


Figure 4.1: The process isolation of the operating system prevents the sender application from transmitting data via memory to the receiver application without write permission. The sender and receiver can establish a covert channel via the CPU cache to circumvent the process isolation.

Our covert channel uses two offsets in the shared memory to transmit one bit at a time. Using two offsets makes the covert channel more reliable and resistant to inaccurate measurements. The offsets are separated by 16kB (the size of a page). This separation ensures that the offsets point to two different pages, which map to different cache sets and, thus, different cache lines in the L2 and L3 cache when accessed. The sender and receiver applications must agree on how to encode and decode the data to be transmitted. For this task, we developed a simple transmission protocol. In Figure 4.2, we see the transmission of the bit sequence `01110001` encoded in our protocol. To transmit the bit 1, the sender produces read accesses only on the memory pointed to by the first offset for duration t. To send a 0, the sender application accesses the shared memory only at the second offset for time t.

While the sender application transmits data, the receiver tries to decode the sent data. For this task, it executes the Evict+Reload primitive for the period t on both offsets in the read-only shared memory. Afterward, the receiver computes the average access time for both offsets. The receiver then writes all the measured average access times to a file for later analysis.

As discussed in Section 2.3, we need a counting thread to perform cache-based side-channel attacks on Apple CPUs because these CPUs do not provide a high-resolution timer. In practice, memory access time measurements obtained with a counting thread can be noisy and inaccurate. To improve the reliability of our covert channel, we train different machine learning classifiers with labeled data. These classifiers require the measurements from the receiver application and the corresponding transmitted values for training. After training, we use the classifiers to classify newly obtained measurements

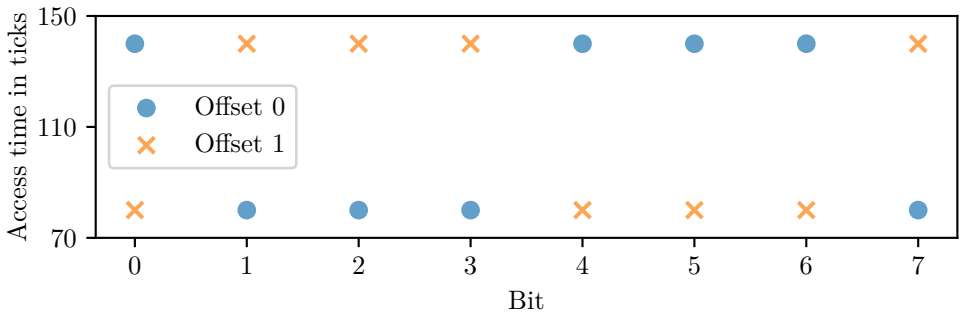


Figure 4.2: This plot visualizes the bit sequence `01110001` encoded in the transmission protocol used by our covert channel.

from the receiver. This approach reduces the error and increases the true capacity, for example, by 266.568 % on the iPhone 13 mini.

4.2 Cache Template Attack

We implement a cache template attack using the same primitives as the covert channel described in Section 4.1. With our attack, we want to detect when the victim opens a new tab in Mozilla Firefox and whether a website visited by the victim plays multimedia content. For this task, we map the shared libraries responsible for handling new tab openings and multimedia playback as shared into the virtual memory of our profiling and exploitation application. As discussed in Section 2.2.3, shared memory mapped to the virtual memory of multiple processes exists only once in physical memory and, therefore, also gets mapped to the same cache lines, regardless of which process accesses it. By performing the Evict+Reload primitive on offsets in this shared memory as described in Section 2.6, we can determine whether another process accesses parts of this shared memory. This concept is illustrated in Figure 4.3. During the profiling phase, we use this approach to analyze which parts of the shared library are necessary to handle a specific action (e.g., opening a new tab). In the exploitation phase, we use the same concept to monitor when the victim application accesses the previously as necessary identified parts. If the victim application accesses these parts, we can conclude that the victim performed the previously profiled action.

4.2.1 The Profiling Phase

Firstly, we need to determine how much time a cache hit and a cache miss consume on our system. We determine these values by generating a cache hit-miss histogram as shown in Figure 2.4. With such a histogram, we determine a threshold to distinguish cache hits from cache misses.

Next, the profiling application maps the target binary with read-only permissions to its virtual memory. To template an event, we trigger the event repeatedly, such as opening a new tab in Mozilla Firefox. While triggering the event, the profiling application iterates randomly over the victim binary. It is crucial to iterate randomly over the binary. Otherwise, the prefetcher distorts our measurements by predicting our subsequent memory accesses and loading the corresponding data into the cache.

At each offset in the binary, we perform the Evict+Reload primitive. When reloading consumes less time than the previously determined threshold, the access is counted as a cache hit. The offset and the number of hits are saved to an output file. After completing this process for each offset in the binary, we analyze the output file and sort the offsets by their corresponding number of hits. Offsets with many hits most likely point to code that the victim application uses to handle the triggered event. These offsets are essential for the exploitation phase. We do not need to perform the profiling phase on the victim system. A system with similar hardware and software usually produces the same results.

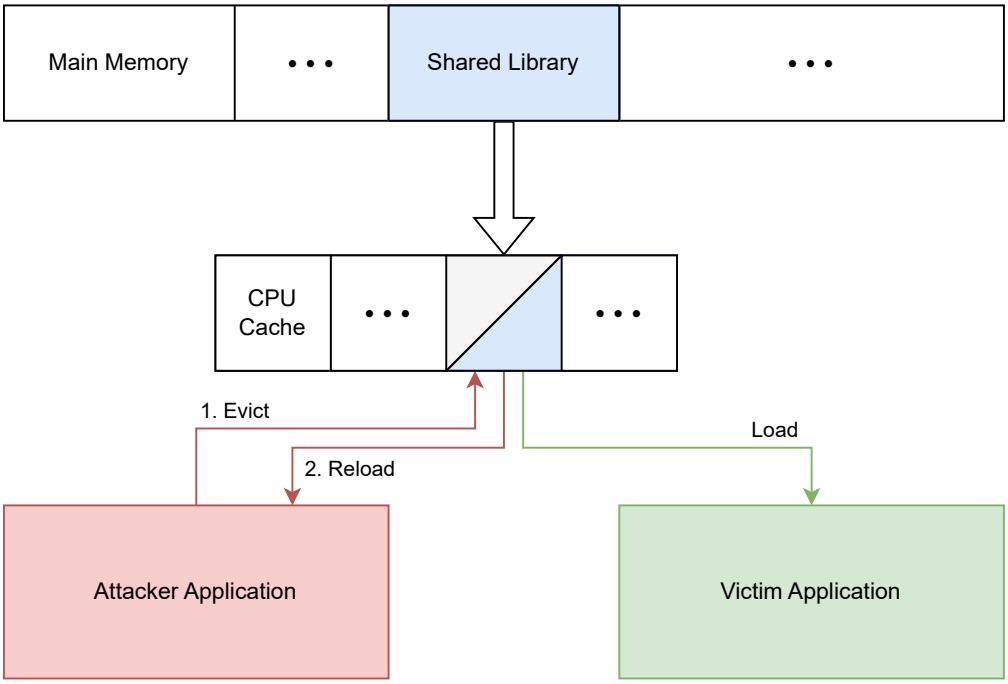


Figure 4.3: The attacker and victim applications map a shared library to their virtual address spaces. This shared library gets mapped to the same cache lines regardless of which application access. The attacker repeatedly evicts and reloads data from this shared library and monitors how long the reload of the evicted data takes. If the victim application accesses the same data from the shared library, the attacker measures a cache hit; otherwise, the attacker measures a cache miss. Using this information, the attacker can determine which locations in the shared library the victim application accesses.

4.2.2 The Exploitation Phase

During the exploitation phase, we detect the event that we templated during the profiling phase. For this task, we use the offsets where we measured many accesses during the profiling phase. As specified in the threat model, Section 3.3, the exploitation phase must run on the same system simultaneously as the victim process.

Firstly, we map the target binary with read-only permissions to the virtual address space of the exploitation application. Next, we perform the Evict+Reload primitive in an endless loop on the offsets identified during the profiling phase. For each reload, we measure the execution time. If the time consumed by the load instruction is lower than the threshold obtained from the cache hit-miss histogram, we record a cache hit. A cache hit indicates that the victim application accessed the code at the probed offset in the binary after we evicted it. If we observe a cache hit on all offsets, the victim application most likely handled our target event.

Chapter 5

Challenges

We address various hardware and software limitations to implement cache-based side-channel attacks or covert channels on our Apple devices. As we pointed out in Section 2.3, our devices do not expose a timer suitable for measuring the duration of single memory accesses. Additionally, the CPUs in these devices do not provide a flush instruction that reliably removes data from the cache. To circumvent the missing flush instruction, we use the eviction approach, described in Section 2.4.2. Typically, we compensate for a missing timer with a counting thread as described in Section 2.3. However, a counting thread does not provide reliable timestamps on our devices, particularly on the iPhone 13 mini. Reliable high-resolution timestamps are essential for an Evict+Reload-based side-channel attack. Therefore, the biggest challenge is implementing a reliable, high-resolution timer, which we discuss in the following sections.

5.1 Thread Scheduling

To increase the accuracy of a counting thread, it should be scheduled constantly, with minimal interference from other threads. This is particularly challenging on the iPhone 13 mini due to its limited number of CPU cores, which results in the counting thread being occasionally interrupted by other threads, leading to inaccurate timestamps. On a Linux device, we could mitigate this issue by isolating a core using the `isolcpus` kernel boot parameter [29] and binding our counting thread to this core. The `isolcpus` parameter allows us to specify one or more CPU cores to isolate from the system. Isolating a core means no user-space threads get scheduled on this core except threads explicitly bound to it. To bind a thread to a core, we need to set its CPU affinity. We set its CPU affinity using the `taskset` application on Linux [1]. On iOS and macOS, these features are not exposed to the user. Here, we can only specify whether a thread should be scheduled on performance or efficiency cores by assigning a quality of service class [5]. Thus, we cannot guarantee that the timing thread is the only thread on its core and that other threads will not interrupt it. We can only ensure that as few applications as possible are running on our devices to reduce the number of context switches. We conclude that we cannot fully solve this issue because we cannot bind threads to the isolated cores under macOS and iOS.

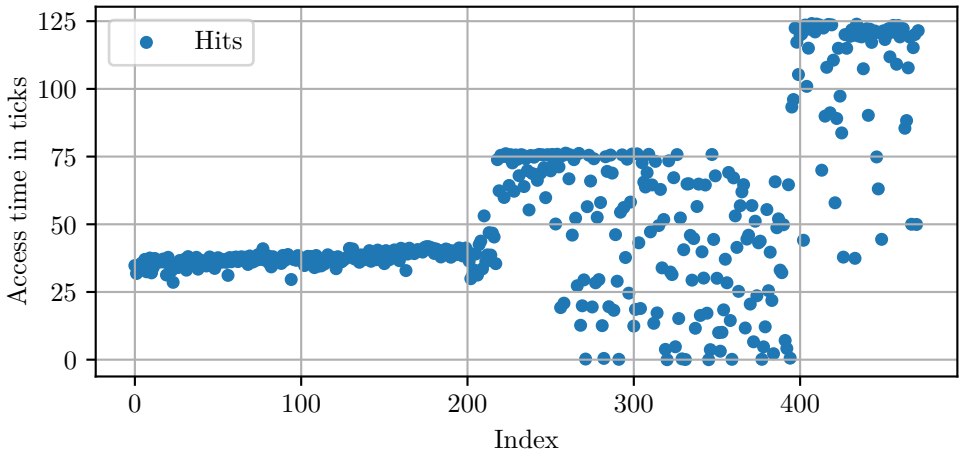


Figure 5.1: The impact of CPU throttling and scheduling on the counting thread on an iPhone 13 mini when measuring cache hits.

5.2 Unstable Frequency

Another issue concerning the counting thread is the fluctuating frequency. The operating system throttles the CPU frequency to prevent the device from overheating and to reduce energy consumption. When the CPU frequency is throttled, the update rate of our counting thread decreases. This behavior makes it difficult to determine a static threshold that distinguishes cache hits from misses. On a Linux device with root access, we could prevent this issue by pinning the CPU frequency to a static value using the user-space application `cpupower-frequency-set`, which relies on the `/sys/devices/system/cpu/cpu*/cpufreq/` system file [2]. Apple operating systems do not provide such an interface. However, we can run all relevant threads for our attack on cores from the same cluster (e.g., the performance cores). Cores from the same cluster always run on the same frequency [8], so frequency fluctuations influence all our threads similarly. This solution only works on a device that provides at least one cluster with more cores than threads that must be executed on the same frequency. As another approach, we can warm up the device before collecting measurements by performing expensive computations. Warming up the device minimizes frequency fluctuations during measurements because the CPU clocks at a low frequency from the beginning to prevent overheating.

5.3 Inaccurate Time Measurements

The issues described in Section 5.1 and Section 5.2 negatively influence our time measurements. In Figure 5.1, we see the impacts of these issues on our measurements. We

measured the duration of cache hits over an extended period on an iPhone 13 mini and computed a rolling mean (window size 30) over the measurements to reduce noise before plotting. The counting thread and the measurement thread were bound to different clusters. This plot shows how the CPU throttles the frequency of cores in different clusters differently. Shortly after index 200, the CPU throttled the core on which the counter or measurement thread ran. We observe the same behavior around index 400. Between index 250 and 400, we see some access times around approximately zero. These measurements are zero because the operating system did not schedule the counting thread while we measured the access time.

In Section 5.1 and Section 5.2, we describe how we could minimize the negative effects on our measurements caused by frequency fluctuations and thread scheduling. However, on Apple devices, many of these fixes are not applicable due to the minimal user interface provided by the operating systems. This circumstance makes it difficult to generate reliable timestamps with a counting thread and even harder to establish a reliable covert channel. To make our covert channel more resistant to inaccurate time measurements, we use two different cache lines to transmit data. Thus, the receiver probes two different cache lines for each transmitted bit. We trained different machine learning classifiers beforehand with labeled data to interpret the measured access times. This approach enhances the reliability of our covert channel and allows for faster data transmission.

5.4 AutoLock

AutoLock is a not publicly documented optimization feature, described by Green et al. [18], implemented on some ARM-based SoCs. The main purpose of AutoLock is to optimize cache performance by reducing unnecessary data movement between the L1 and the L2 caches. However, AutoLock poses a challenge for cache attacks because it prevents cross-core eviction. When we try to evict data from the L2 cache, this feature checks if the data is still present in an L1 cache of another CPU core. If so, the data gets locked and will not be removed from either cache. According to Yu et al. [56], the Apple M1 CPU implements AutoLock. During our experiments, we noticed that our iPhone's Apple A15 Bionic CPU most likely also implements this feature. Green et al. [18] noted that AutoLock requires the LLC to be inclusive, meaning the LLC contains the entire content of all core-private caches. Consequently, the L2 caches in the M1 and A15 CPUs are L1 inclusive. In our implementation, we use the approach of Hetterich et al. [25] to reduce the impact of AutoLock on the performance of the covert channel. After we transmit a bit by repeatedly accessing one of the cache lines, the sender application performs an eviction on this cache line. With this approach, we can evict that cache line from the L1 cache of the core running the sender application. However, this only works if the operating system does not schedule the sender application on different cores while we transmit a bit.

Chapter 6

Evaluation

In this chapter, we demonstrate that Apple Silicon is vulnerable to cache template attacks similar to traditional x86-64 CPUs. We overcome the challenges described in Chapter 5 to launch cache-based side-channel attacks on our Apple devices. Thus, we start by evaluating the functionality of the building blocks required to launch a cache template attack. In Section 6.1, we describe how we estimate a threshold on our devices that allows us to distinguish cache hits from cache misses. In Section 6.2, we evaluate the reliability of the eviction set generation from Hetterich et al. [25] on our devices. Next, we use the eviction approach to set up a covert channel on an Apple iPhone 13 mini and an Apple Mac mini (M1). We evaluate different post-processing methods to boost the performance of the covert channel. We conclude that our covert channel on the Apple Mac mini (M1) is the fastest Evict+Reload-based covert channel on an ARM-based CPU. Based on the covert channel, we implement a cache template attack. In Section 6.4, we demonstrate that we can detect when a victim opens a new tab in Mozilla Firefox on a Mac mini (M1) with our cache template attack. Additionally, we use our cache template attack to profile websites and recognize if a website visited by the victim in Mozilla Firefox is playing media.

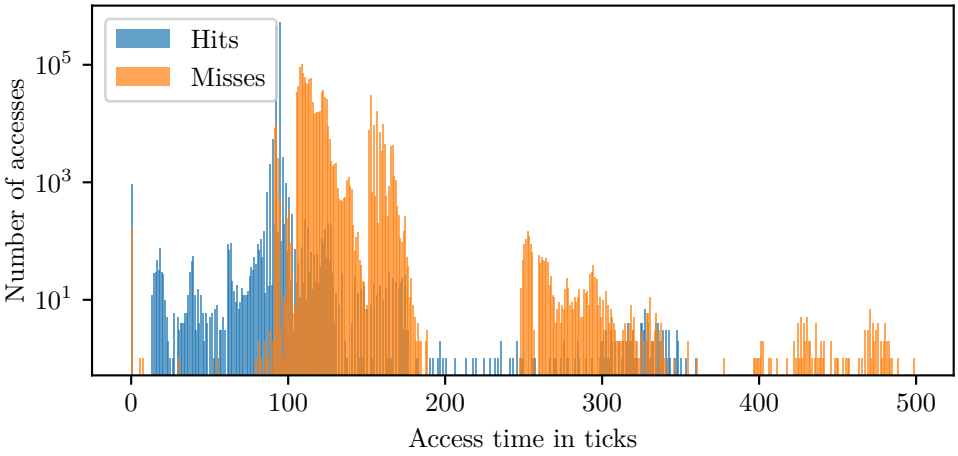


Figure 6.1: Calibration histogram on an Apple Mac mini (M1).

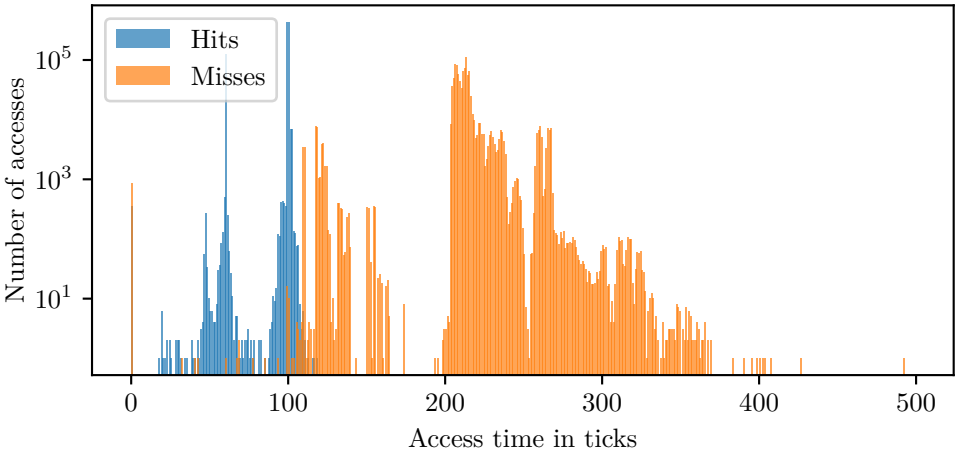


Figure 6.2: Calibration histogram on an iPhone 13 mini.

6.1 Cache Hit-Miss Threshold

To create eviction sets and mount a time-based cache attack, we need to be able to distinguish cache hits from cache misses by measuring the execution time of an `LDR` instruction. For this task, Hetterich et al. implemented a benchmark application that measures the time consumption of 10^6 cache hits and misses. We executed this application on our devices and created histograms to visualize the collected measurements.

To measure the cache hits, we first allocate one memory page using `malloc`. We access the address in the allocated memory twice to ensure that the data at this address is loaded into the cache so that the next access will result in a cache hit. Then, we perform another access to the same address and measure the duration of this access.

To generate a cache miss, we first allocate 100 pages. Next, we randomly pick one from these 100 pages. By randomly selecting a page, we make it impossible for the prefetcher to predict which memory address we will probe next. To cache the virtual-to-physical address translation in the TLB, we access a random address in the first half of the selected page. Without this step, our measurements would include the duration of the page translation process. We measure the duration of a cache miss by accessing a random address in the second half of the previously picked page.

In Figure 6.1, we see the hit-miss histogram from the Mac mini (M1), and in Figure 6.2 from the iPhone 13 mini. Because of the unstable counting thread, the histograms appear noisy. However, timing differences between cache hits and cache misses are visible. Most cache hits take less time than cache misses. On the Mac mini, memory accesses that take over 100 ticks are considered cache misses. On the iPhone, memory accesses taking longer than 109 ticks are most likely cache misses.

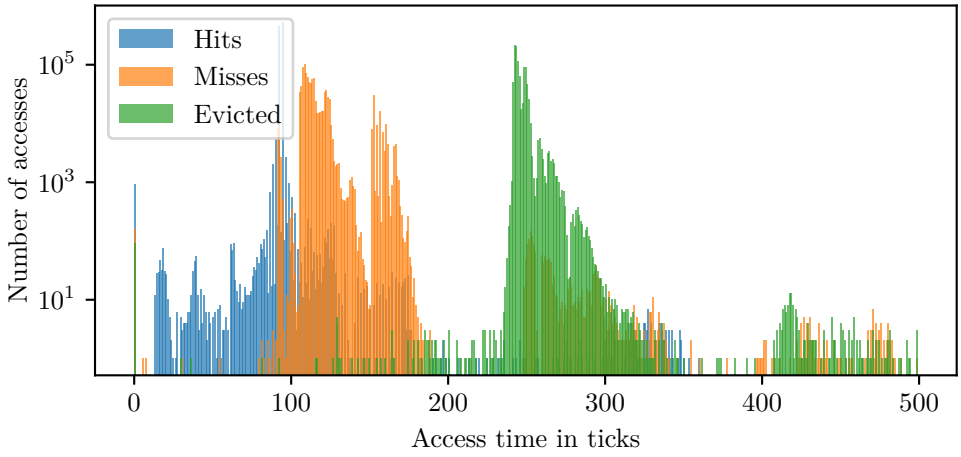


Figure 6.3: Comparison of the access duration of cached, uncached, and evicted memory on an Apple Mac mini (M1).

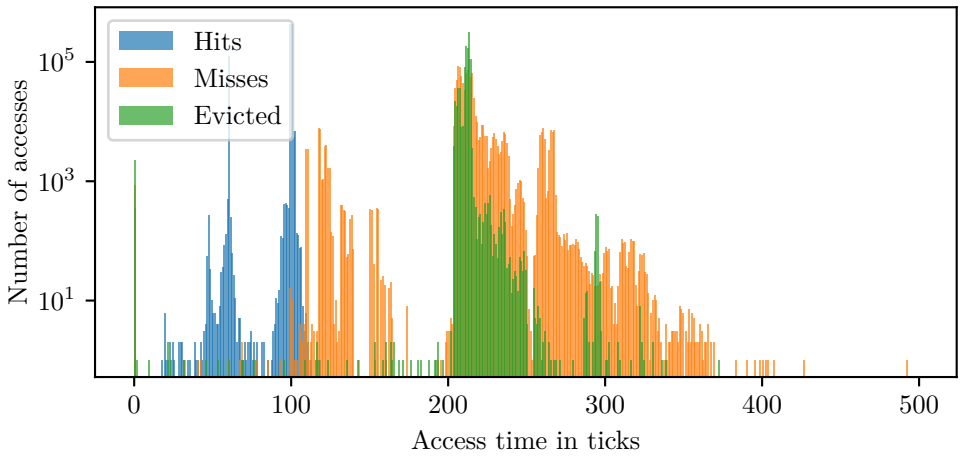


Figure 6.4: Comparison of the access duration of cached, uncached, and evicted memory on an Apple iPhone 13 mini.

6.2 Performance of the Cache Eviction

To generate eviction sets, we rely on the eviction set generation implementation from Hetterich et al. [25], which is based on the work of Vila et al. [52]. This implementation uses a counter thread to measure the duration of memory accesses and is fully compatible with the ARMv8 instruction set. First, we need to specify a so-called eviction threshold. The eviction threshold determines the minimal time consumed by a cache miss. We can estimate the eviction threshold using the histograms from Section 6.1. On the M1 Mac mini, we selected 220 ticks as the eviction threshold. This threshold is the same as that used by Hetterich et al. [25] for their Spectre exploit. On the iPhone 13 mini, we selected 185 ticks as the eviction threshold. We use these thresholds in all further experiments to evict data from the cache.

In Figure 6.3 and Figure 6.4, we can see how well the eviction performs on our devices. We generated these histograms by performing 10^6 cache hits, misses, and evictions on the iPhone 13 mini and the Mac mini (M1). We can see that accessing evicted addresses takes mostly longer than the selected threshold. On the Mac mini, 99.98 % of all accesses on evicted addresses take longer than our defined threshold of 220 ticks. On the iPhone, 99.77 % of all accesses on evicted memory take longer than 185 ticks. As far as we can validate, the group testing-based eviction set generation approach from Vila et al. [52] works reliably on our devices. However, we must rely on our counting thread to generate eviction sets and validate their performance. As mentioned in Section 5.3, our counting thread does not always provide reliable and accurate timestamps. Additionally, in this experiment, we did not perform any cross-core eviction. Therefore, AutoLock does not influence our eviction as described in Section 5.4. Further experiments have shown that the quality of the eviction sets varies when we generate different eviction sets over multiple hours. In such a scenario, we cannot be sure that we evict the desired data from all caches.

6.3 Covert Channel Performance

In this section, we discuss the performance of our covert channel on the iPhone 13 mini and the Mac mini (M1). For this task, we run our covert channel on both devices. We transmit random data for 60 seconds on both devices through our covert channel to evaluate the performance. Afterward, we apply different post-processing methods to interpret the raw measurements obtained by the receiver. We create a plot to visualize the performance of each post-processing approach. For better visualization, we plot only 2500 randomly selected bits. In such a plot, each data point represents a measured bit. Because we transmit uniformly distributed random data through our covert channel, our dataset consists of approximately equal numbers of bits with values of zero and one. As described in Section 4.1, our covert channel uses two cache lines for transmission. Thus, two values define a bit: the measured access time on memory at offset 0 and offset 1. We plotted the access time on memory at offset 0 on the x-axis and the access time on

memory at offset 1 on the y-axis. The color of the data point indicates the interpreted value of this bit. Bits with the assigned value 0 are marked as blue dots, and bits with 1 assigned are displayed as orange crosses.

First, we generate a reference plot for each dataset. This plot visualizes 2500 randomly selected bits of the transmitted data, labeled with their actual values. In theory, our data points in this plot should be divisible linearly by a diagonal line into zeros and ones. However, this is not the case in practice due to the challenges described in Chapter 5.

6.3.1 Post-Processing

We applied a naive post-processing approach to our measured memory access times. According to our self-defined transmission protocol (Section 4.1), if accessing offset 0 is fast and accessing offset 1 is slow, we transmitted the value 1. If accessing offset 0 is slow and accessing offset 1 is fast, the sender transmitted the value 0. Based on this knowledge, we checked which of the two memory offsets had a shorter access time and assigned the value 0 or 1 accordingly.

We apply a selection of machine learning classifiers to our data. Our selected classifiers are available in the python library `scikit-learn` [43]. We use the classifiers from this library to interpret our data. If not specified differently, we use the default parameter values to train the models and classify our data. First, we apply a Gaussian naive Bayes classifier, which assumes a Gaussian distribution of the data. Second, we evaluate the performance of a multilayer perceptron classifier on our data. Our multilayer perceptron classifier consists of two hidden layers. The first hidden layer contains ten perceptrons, and the second one has five perceptrons. We also apply a traditional support vector machine to classify our measurements. Support vector machines perform very well on linearly separable data. However, by applying the so-called kernel trick, they can also classify non-linear separable data. The kernel trick applies a kernel to the data and projects it to a higher-dimensional space, where the data is linearly separable. We use a radial basis function (RBF) kernel for our evaluation. Further details about each machine learning classifier used in this evaluation can be found in Section 2.7.

Classifier Configuration and Training

We need a dataset from each device containing the measured access times and the corresponding original bit values from the sender to train the classifier models. To collect this data, we execute our covert channel on both devices using the same parameters as we do for the dataset we want to classify later. Then, we label the measurements with the reference values. We randomly shuffle this data and use 80 % of the data as a training set and the remaining 20 % as a validation set. We use the training set to train the classifiers and the validation set to evaluate their performance.

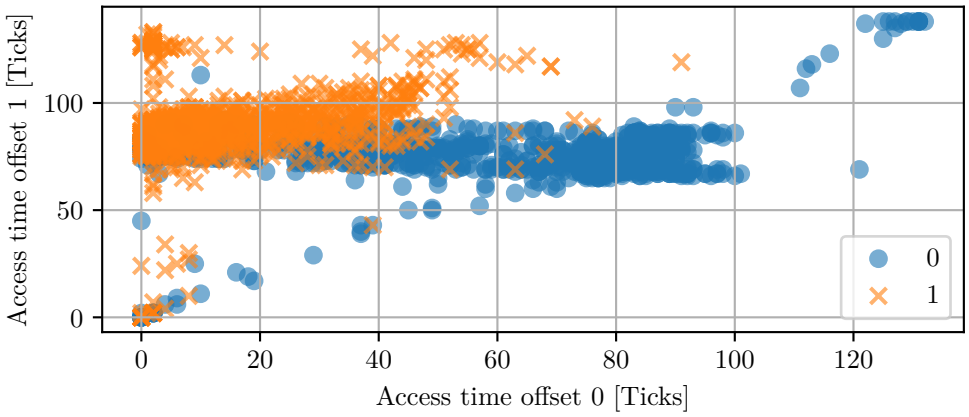


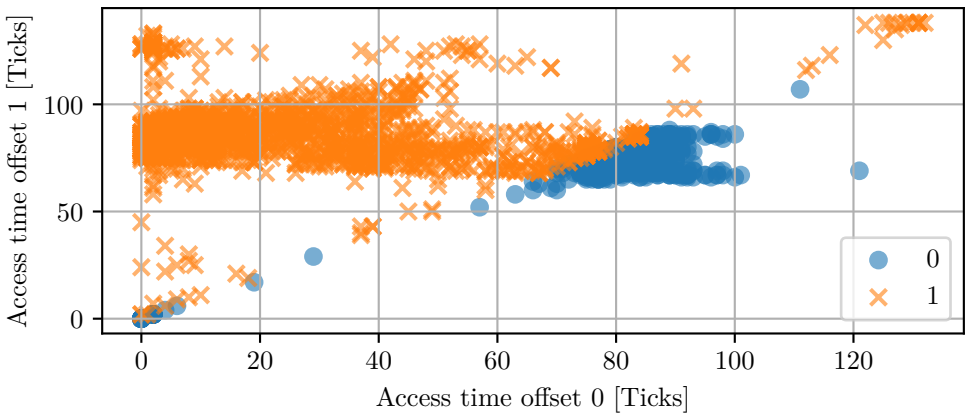
Figure 6.5: The reference plot. This plot displays the data collected on the iPhone 13 mini labeled with their correct bit values.

We train our naive Bayes classifier using a `GaussianNB` object from the `scikit-learn` library. We do not modify the default configuration of this classifier. For the multilayer perceptron classifier, we use the `MLPClassifier` from the `scikit-learn` library. We configure two hidden layers: the first hidden layer contains ten perceptrons, and the second five perceptrons. Additionally, we limit the number of iterations to 300. We use the `SVC` class of the `scikit-learn` library as our Support Vector Machine classifier. We initialize one SVM with a linear kernel and another with an RBF kernel. For both SVMs, we limit the cache size to 1 GB. We limit the number of iterations for the SVM with the linear kernel to 2000.

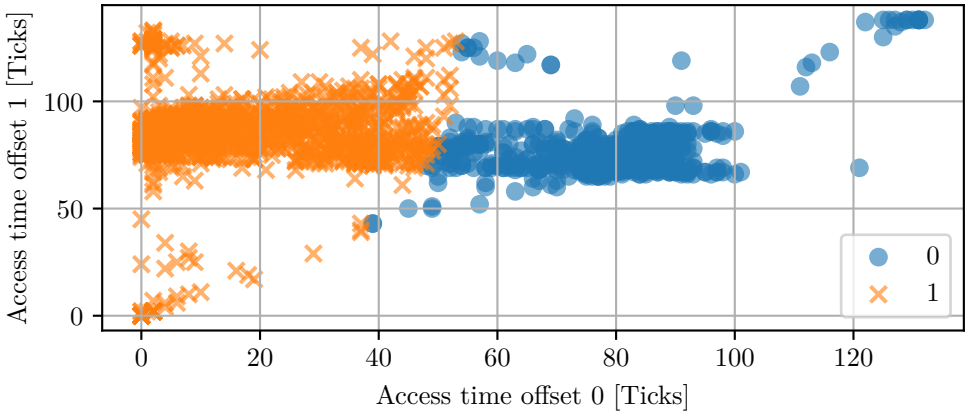
After initializing all classifiers, we invoke the training by calling the `fit` method on each classifier object. To this method, we pass the labeled training set as a parameter. After training the classifier, we evaluate its performance. To do this, we classify the validation set using the `predict` method from the classifier object. We generate the confusion matrix from the classified validation set and its original labels to compute the accuracy of the classifier. We compute the diagonal sum of the confusion matrix and divide it by the number of samples in our validation set. This gives us the percentage of correctly classified samples. To compute the confusion matrix and extract the accuracy, we use the `confusion_matrix` and `accuracy` functions from the `scikit-learn` library.

6.3.2 iPhone 13 mini Covert Channel Performance

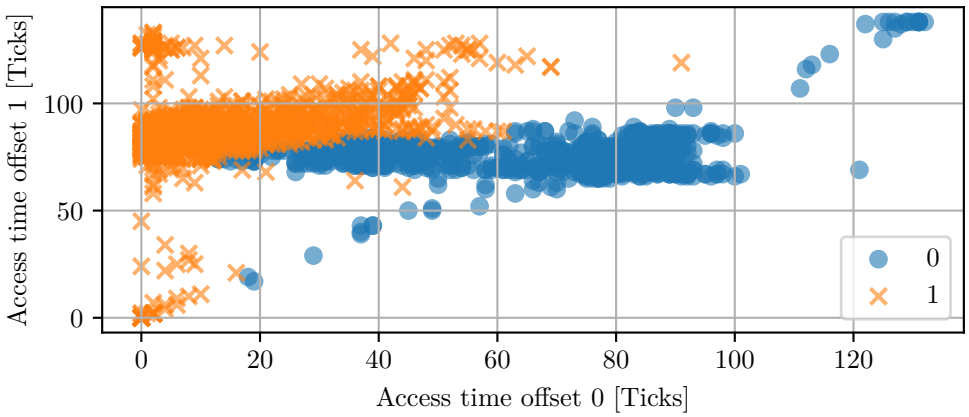
The Apple A15 Bionic CPU running in the iPhone 13 mini provides only two performance and four efficiency cores. These cores are assigned to two clusters. One cluster contains the performance cores, and the other contains the efficiency cores. Our covert channel implementation requires three cores: a core for the sender, the receiver, and the counting thread. Because we have only two performance cores, we cannot simply execute all threads



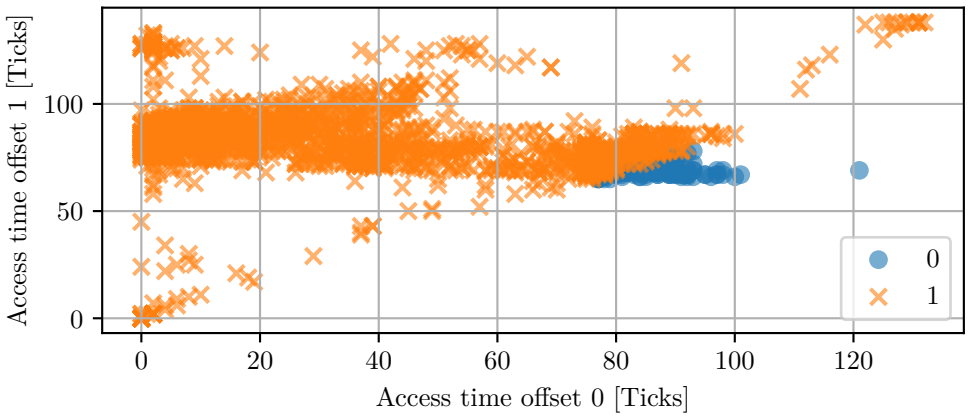
(a) Naive approach (Accuracy 73.895 %)



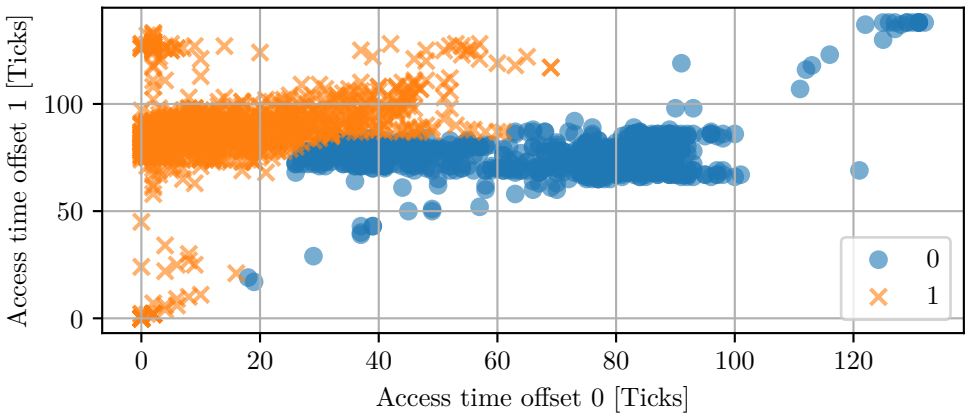
(b) Bayes classifier (Accuracy 88.949 %)



(c) Multi-layer perceptron classifier (Accuracy 92.807 %)



(d) Linear support vector machine (Accuracy 55.065 %)



(e) RBF support vector machine (Accuracy 92.872 %)

Figure 6.6: These plots show how different post-processing approaches interpret raw output data from our covert channel on an iPhone 13 mini.

on the performance cores. To evaluate our covert channel, we pin the counting thread to the performance core cluster and the sender and receiver threads to the efficiency core cluster. The higher clock frequency on the performance cores results in a higher update frequency of the counter thread, enhancing the resolution of the timestamps. By binding the sender and receiver threads to the efficiency core cluster, we ensure that both threads operate at the same clock frequency. While this setup does not completely avoid the issues introduced by frequency scaling, as described in Section 5.2, it minimizes their impact.

The dataset collected on the iPhone contains measurements of 6×10^5 transmitted bits. The transmission process took 62.5 seconds. Thus, we transmitted our data at a raw speed of 9.6 kbit/s. In Figure 6.5, we see the measured data points colored with their actual value. This plot contains only 2500 randomly selected data points from the whole dataset for better visualization. Connecting our measurements with the reference values allows us to estimate the distribution of our data roughly. As expected, nearly all data points labeled with 1 are located in the upper left half of the plot. For these values, we measured a short access time at offset 0 and a longer access time at offset 1 because the sender repeatedly accessed the shared memory only at offset 0. On the other hand, we expect data points labeled with 0 in the lower right side of the plot. However, this is not the case, as 25.561 % of the points labeled with 0 are located in the upper left half and overlap with points labeled as 1. Our generated eviction set for offset 0 does not evict data as reliably as the eviction set for offset 1. The challenges discussed in Chapter 5 are most likely causing these issues. Due to this data overlap in the upper left half, the unlabeled measurements are not linearly separable.

In Figure 6.6a, we apply the previously presented naive post-processing approach to our data. The plot shows that this approach uses a straight diagonal line as a decision boundary to separate the data. Due to our noisy measurements, this method works only partially well. We classified 98.91 % of the measurements we obtained while sending 1 correctly, but only 48.9 % of the transmitted zeros. Overall, this approach reconstructed 73.895 % of the transmitted bits correctly.

Next, we apply a Gaussian Naive Bayes classifier to our measured data. The results are shown in Figure 6.6b. With this approach, we correctly classify 79.48 % of all zeros and 98.46 % of all ones, resulting in an overall accuracy of 88.949 %. As shown in Figure 6.6c, the multilayer perceptron classifier separates our transmitted data very well. This classifier assigns the correct label to 92.80 % of all data points. This approach performs exceptionally well compared to the others on classifying the zeros. It classifies 89.45 % of all zeros correctly. Additionally, the classifier correctly assigns the label 1 to 96.12 % of all ones. Next, we apply a standard SVM to our measurements. As described in Section 2.7, the SVM classifies data using a linear decision boundary. This is clearly visible in Figure 6.6d. In this plot, the SVM divides our data into two classes with a straight diagonal line. This approach does not perform well because our data is not linearly separable. The SVM classifies only to 10.23 % of all zeros correctly. On the other

Table 6.1: Overview of the performance of our covert channel on the Apple iPhone 13 mini after applying different post-processing approaches.

Post Processing	Error Rate	True Capacity
Naive approach	26.105 %	1.648 kbit/s
Bayes classifier	11.050 %	4.786 kbit/s
MLP classifier	7.193 %	6.018 kbit/s
Linear SVM	44.935 %	0.712 kbit/s
RBF SVM	7.128 %	6.041 kbit/s

hand, it correctly classifies 99.96 % of all ones. The SVM classifies bits with the value 1 so well because it sets the decision boundary too low, assigning 94.86 % of all data points the label 1. We transmitted uniformly distributed random data via our channel. Thus, approximately 50 % of all transmitted bits should be labeled as 1s. Overall, we achieve an accuracy of 55.06 %. Finally, we apply a SVM that uses the kernel trick with a radial basis function (RBF) kernel. The kernel trick allows the SVM to classify data using a non-linear decision boundary. In Figure 6.6e, we see the performance of the SVM with an RBF kernel on our data. This approach classifies bits with the value 0 and bits with the value 1 very well. This classifier correctly assigns the label to 89.16 % of all transmitted bits with the value 0. Additionally, this approach correctly classifies 96.59 % of all transmitted 1s. This results in an overall accuracy of 92.87 %.

Table 6.1 provides an overview of the performance of different post-processing approaches. As listed in Table 2.1, Hetterich et al. [25] achieved an error rate of 7.84 % and a true capacity of 1.448 kbit/s. The results of Hetterich et al. are only to a limited extent comparable to ours because they used a different device, an iPhone 8 Plus. The CPU frequency, in particular, influences the results significantly. The Apple A11 Bionic CPU in the iPhone 8 Plus operates at up to 2390 MHz, while the A15 Bionic CPU in our iPhone 13 mini reaches a clock frequency of 3230 MHz. By applying our best-performing post-processing approach, the RBF support vector machine, to our measurements, we achieve an error rate of 7.13 %. We transmitted our data with a speed of 9.6 kbit/s. This results in a true capacity of 6.041 kbit/s, which is more than four times higher than the true capacity achieved by Hetterich et al. Even our naive approach outperforms Hetterich et al.'s covert channel. Our naive approach enables us to transmit data with a true capacity of 1.648 kbit/s.

6.3.3 Mac mini (M1) Covert Channel Performance

We execute our covert channel on an Apple Mac mini (M1). The M1 CPU, which powers the Mac mini, consists of two clusters. One cluster includes four performance cores, and the other includes four efficiency cores. Since the performance cluster contains four cores, we pin the sender, the receiver, and the counting thread to the performance core cluster.

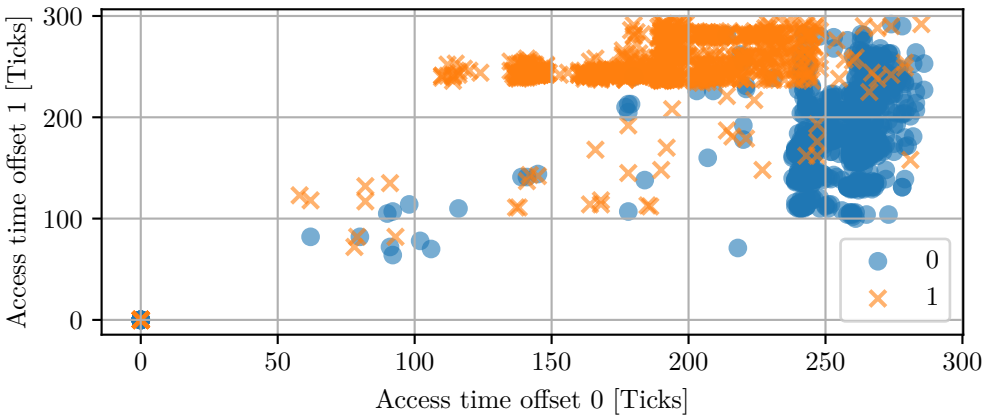
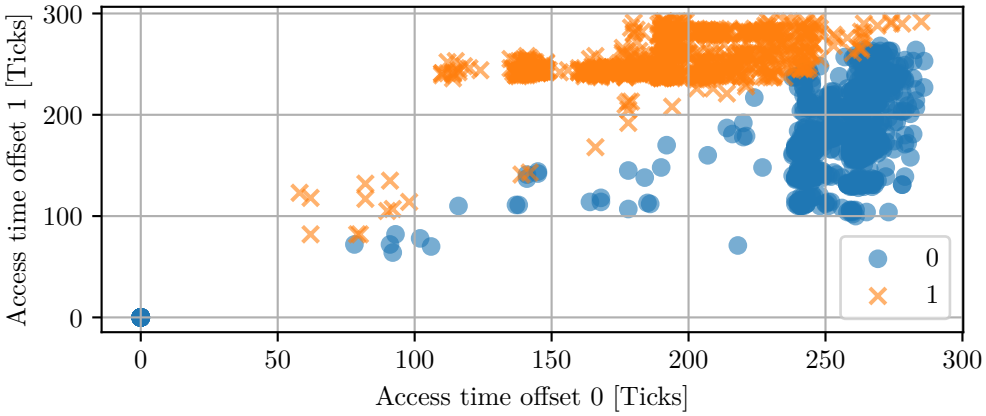


Figure 6.7: The reference plot. This plot displays the data collected on the Mac mini (M1) labeled with their correct bit values.

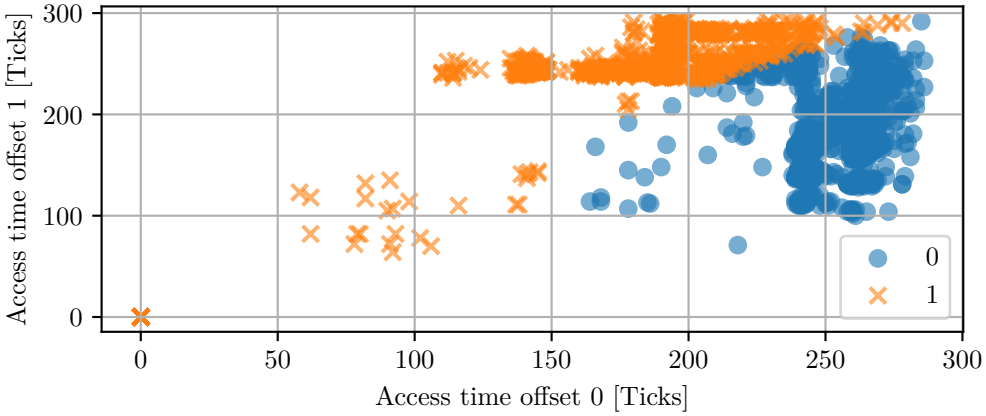
This setup effectively avoids the issues listed in Chapter 5. The high CPU frequency on the performance cores allows us to obtain timestamps with a high resolution. Each performance core is always clocked at the same speed. Thus, frequency fluctuations do not affect our measurements.

To collect the measurements for this evaluation, we transmitted 9.6×10^6 bit at a speed of 159.095 kbit/s. The entire transmission took 60.341 s. We applied the same post-processing approaches with the same parameters on the collected measurements as we previously applied to the measurements from the iPhone. In Figure 6.7, we see a sample of 2500 data points we transmitted for this evaluation over our covert channel. In this plot, we connected the measurements obtained from the receiver with the actual transmitted values. This plot allows us to estimate the distribution of our data points roughly. As expected, we measured on nearly all points labeled with 1 a longer average memory access time on offset 1 than on offset 0. These points are located in the upper left half of the plot. Similarly, most data points labeled with 0 are located in the lower right half of the plot. Here, we measured a shorter average memory access time on offset 1 compared to offset 0. Thus, our measured data is mostly linearly separable. We also have data points where we measured an average memory access time of 0 on both cache lines in this plot. During these measurements, the counting thread probably did not get scheduled, so its counter could not be updated.

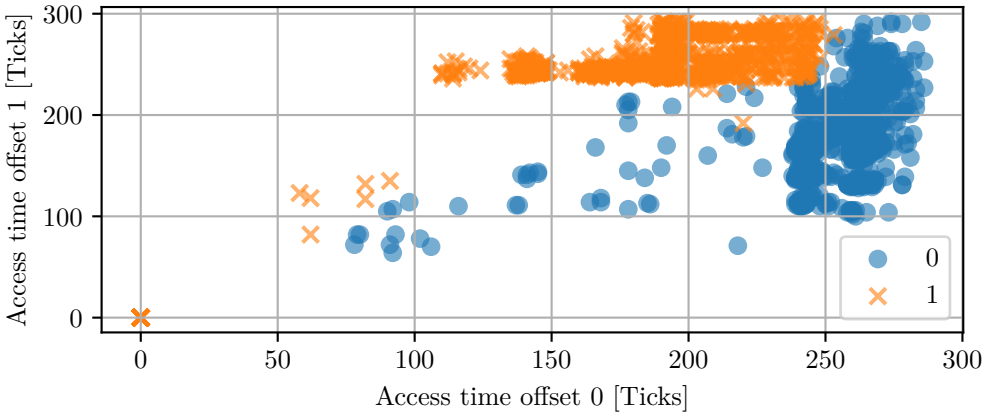
In Figure 6.8a, we apply the naive post-processing approach presented in Section 6.3.1. This plot shows that this post-processing method attempts to separate the data points with a straight line. This approach interprets the received data with an accuracy of 95.82 %. Specifically, it correctly classified 96.28 % of all zeros and 95.35 % of all ones. This approach achieves this high accuracy because our measured data contains very little



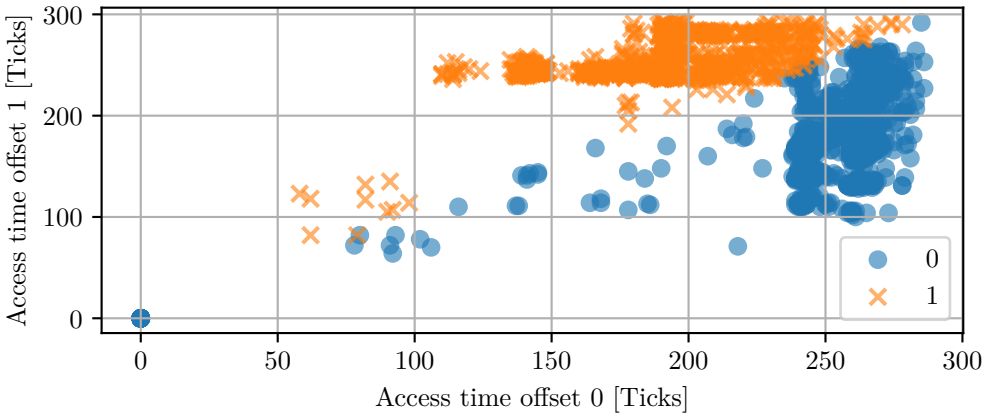
(a) Naive approach (Accuracy 95.821 %)



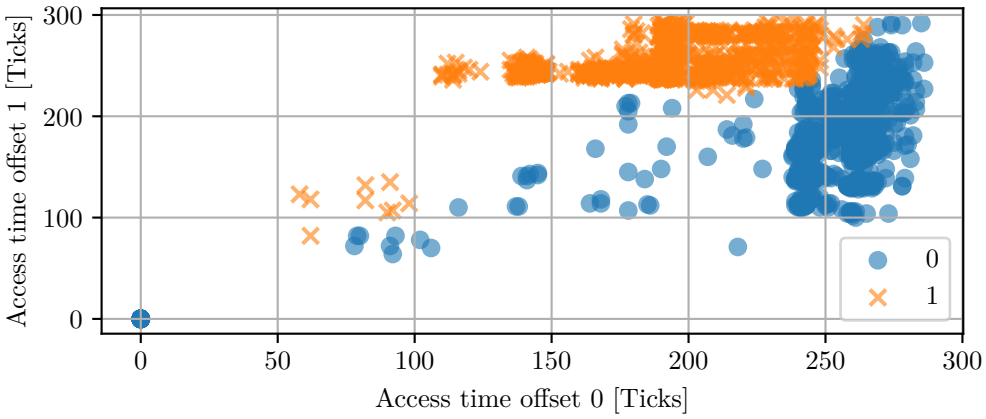
(b) Bayes classifier (Accuracy 93.803 %)



(c) Multi-layer perceptron classifier (Accuracy 96.169 %)



(d) Linear support vector machine (Accuracy 95.705 %)



(e) RBF support vector machine (Accuracy 96.260 %)

Figure 6.8: These plots show how different post-processing approaches interpret raw output data from our covert channel on an Mac mini (M1).

Table 6.2: Overview of the performance of our covert channel on the Apple Mac mini (M1) after applying different post-processing approaches.

Post Processing	Error Rate	True Capacity
Naive approach	4.179 %	119.253 kbit/s
Bayes classifier	6.197 %	105.765 kbit/s
MLP classifier	3.830 %	121.792 kbit/s
Linear SVM	4.295 %	118.423 kbit/s
RBF SVM	3.740 %	122.463 kbit/s

noise.

Next, we applied our machine learning-based post-processing approaches to the measured memory access times. We applied the Bayes classifier, as shown in Figure 6.8b. With this classifier, we achieve an accuracy of 93.80 %. This approach significantly underperforms in classifying ones. We correctly interpret only 91.33 % of the bits with value 1, while we correctly classify 96.97 % of all bits with value 0. The Gaussian distribution assumed by our classifier does not perfectly fit the data. We could likely achieve better performance with this approach if we use a different kernel. In Figure 6.8c, we applied the multilayer perceptron classifier to the data obtained during our test transmission run. With this approach, we correctly interpret 96.17 % of the transmitted data. By applying a linear SVM, we assign the correct value to 95.70 % of the measurements. This classifier performs much better on the Mac mini (M1) data compared to the iPhone 13 mini data because the data is nearly linearly separable. Finally, we apply an SVM that implements the kernel trick with a radial basis function. The kernel trick allows an SVM to separate data with a non-linear decision boundary. The classified data is visible in Figure 6.8e. This classifier provides the best results of the different post-processing approaches. We assign the correct value to 96.26 % of all measurements with this classifier.

In Table 6.2, we summarize the performance of the different post-processing approaches. We conclude that all approaches interpret the measurements from our covert channel with an acceptable error rate. The support vector machine with RBF kernel delivers the best performance, and the Bayes classifier performs the worst. However, the error rate achieved by these two classifiers differs only by 2.456 %. Additionally, the naive approach performs well with an error rate of 4.18 %. Compared to the iPhone covert channel, we cannot significantly increase the accuracy by using machine learning to interpret the raw data from the covert channel. Hetterich et al. [25] demonstrated the fastest Evict+Reload covert channel on ARM-based CPUs so far, with a true capacity of 10.840 kbit/s. We transmitted our test data with a raw speed of 159.095 kbit/s. By applying the true capacity formula from Shannon et al. [46] to the raw speed and the error rate from the RBF support vector machine, we obtain a true capacity of 122.463 kbit/s.

Thus, our covert channel is currently the fastest Evict+Reload covert channel on a CPU implementing the ARM instruction set.

6.4 Cache Template Attack

This section demonstrates that recent Apple CPUs are vulnerable to cache template attacks in the same ways as x86-64 systems. A cache template attack runs on the victim device in a different process and allows an attacker to profile and exploit cache-based information leakage from an attacker-controlled process [21]. With our cache template attack, we profile the handling of new tab openings in Mozilla Firefox (version 97.0.1) on a Mac mini (M1) computer running macOS Monterey (version 15.5). Additionally, our attack can recognize if a user is browsing a website that plays media in Mozilla Firefox. As explained in Section 2.6, a cache template attack consists of the profiling and the exploitation phase. During the profiling phase, we analyze which locations of a binary are accessed to handle the actions we want to detect later during the exploitation phase.

On x86-64 systems running Windows or Linux, system libraries mapped as shared are popular targets for cache template attacks. System libraries are inaccessible via the file system on the Mac mini (M1). On macOS, user-space applications must use an API provided by the operating system to access system libraries. Thus, we cannot attack system libraries because we cannot map the binary as shared to the virtual memory of our profiling application as described in Section 2.6. However, we can still attack libraries that are shipped with third-party applications. We profiled the new-tab-opening event in Mozilla Firefox to evaluate our implemented attack by analyzing the `firefox` binary. Additionally, we profiled static HTML websites and websites that are playing media. We mapped the `liblgpllkbs` library to our profiling application for this task. The `liblgpllibs` library comes with Mozilla Firefox and is responsible for playing media. We can distinguish which websites play media by analyzing the traces collected during the profiling process.

6.4.1 Detect Events in Mozilla Firefox

To demonstrate that we can leak events from a foreign process, we implement a cache template attack, which detects when a user opens a new tab in Mozilla Firefox. We map the binary `/Applications/Firefox.app/Contents/MacOS/firefox` to our profiling application for this task. We iterate over the binary and perform the Evict+Reload primitive on each offset multiple times for 200 ms. While iterating over the binary, we repeatedly open new tabs in Firefox by holding the shortcut Ctrl + t. We perform this profiling process three times and sum up the number of cache hits on each offset from each run to obtain more reliable measurements. Next, we remove the offsets that are not required to handle our target event but are in our data because they are accessed by the application to handle other tasks. To filter out those offsets, we execute the profile application thrice with the same parameters. During these measurements, we do

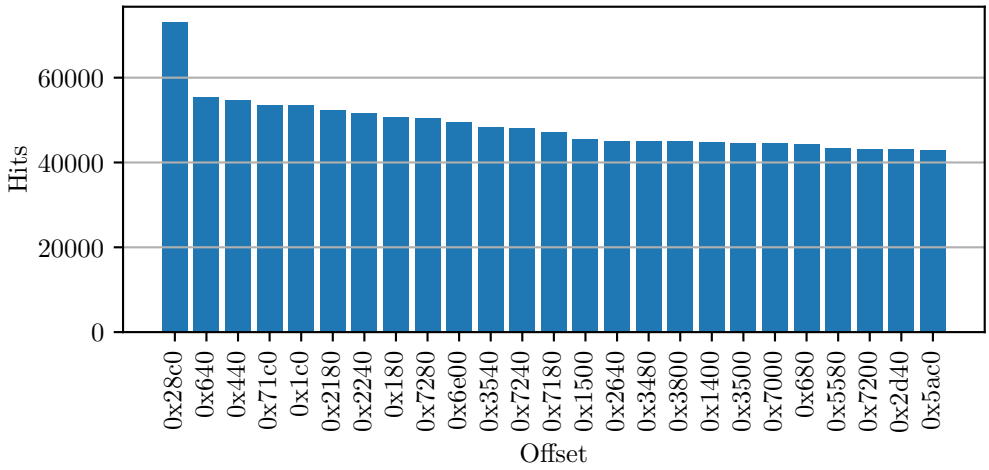


Figure 6.9: The 25 offsets with the most measured cache hits during the profiling phase.

not perform any action in Firefox. Additionally, we collect another three runs with the same parameters as our profiling application. In parallel to these runs, we close tabs in Firefox repeatedly by holding the shortcut Ctrl + w. We subtract the measured cache hits per offset from these six runs from the data obtained while performing the target action. Next, we sort the offsets by the corresponding amount of measured cache hits. In Figure 6.9, we plot the 25 offsets with the most hits. We see that we got the most hits (73 074) on offset 0x728c0. With these results, we proceed to the exploitation phase.

During the exploitation phase, we detect when the victim opens a new tab in Mozilla Firefox. For this task, we implement an exploitation application that works as described in Section 2.6.2. With such an application, we measure the number of hits on a specific offset in a binary. To our exploitation application, we need to pass the path to the binary we want to map, the duration for which the application should count cache hits, and the offset we want to probe. We pass `/Applications/Firefox.app/Contents/MacOS/firefox` as the binary path, 200 ms as the probing time, and as offset, we tried out the 25 offsets with the most hits, which we determined during the profiling phase. When we opened a new tab while executing the exploitation application, we observed a significant rise in measured cache hits on each offset. Thus, we conclude that all offsets identified during the profiling phase can reliably be used to detect when a user opens a new tab in Mozilla Firefox.

6.4.2 Detect Media Playback in Mozilla Firefox

By analyzing which addresses of the browser binary are accessed while the victim browses the internet, we can determine the type of websites the victim visits. Our cache template attack distinguishes whether the victim user visits a website with or without playback media in Mozilla Firefox. For this task, we use our profiling application to track which

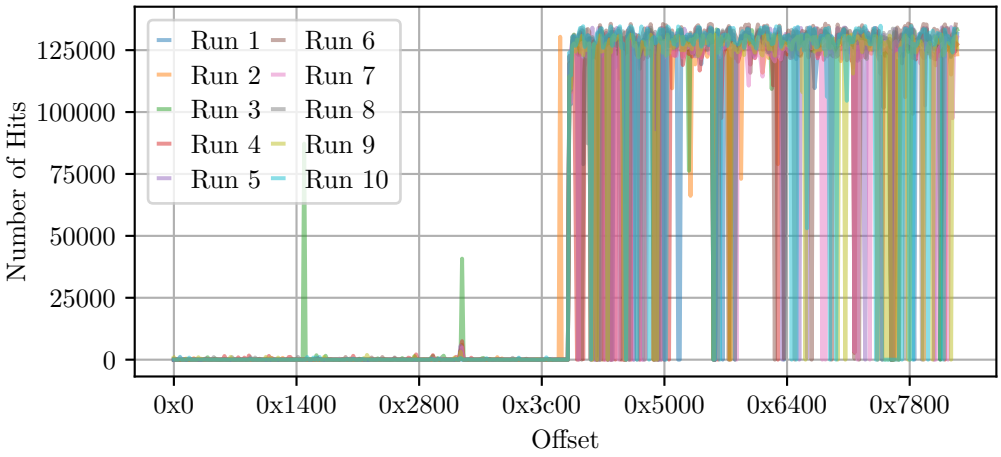


Figure 6.10: The 10 traces obtained from `kernel.org`.

offsets of the targeted binary are accessed when a website plays media. To mount such an attack, the attacker needs to be able to execute code on the victim's machine. In practice, an attacker could achieve code execution on a victim device, for example, by integrating the cache template attack into an inconspicuous application.

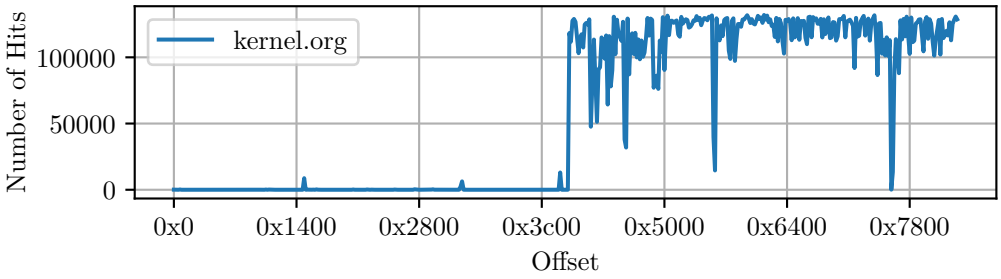
We demonstrate the functionality of our implemented attack on a Mac mini (M1) using six websites. We select `/Applications/Firefox.app/Contents/MacOS/liblgpllibs.dylib` as the victim binary. This dynamic library gets loaded by Gecko, Mozilla Firefox's HTML rendering engine [17]. According to several code comments in the Gecko repository on Github [17], the dynamic library `liblgpllibs` is a collection of various third-party libraries. One of the included libraries is `libav`. `libav` is a library that provides tools to process multimedia content like audio and video [15]. Thus, when we profile a website that plays media, we should see more activity on this library compared to a website that does not contain any media. We probe each offset of the `liblgpllibs` for 200 ms for this evaluation. We select `kernel.org`, `wikipedia.org`, and the default Mozilla Firefox start page as websites without any media playback. To demonstrate the detection of active multimedia on a website, we profile song playback on `soundcloud.com`, video playback on `youtube.com`, and the playback of a local MP4 video in Mozilla Firefox. We collect ten traces from each of these websites.

In Figure 6.10, we plotted the ten traces obtained while profiling `kernel.org`. The x-axis represents the binary offset, and the y-axis represents each offset's corresponding number of hits. We see that most of the code from the first half of the binary is not used. Furthermore, we observe many cache hits in the second half. However, in the second half, we see significant dips in the number of hits on some offsets. We computed the average number of cache hits from the ten collected traces to further analyze the measurements. In Figure 6.11, we plotted the average number of hits per offset measured while profiling

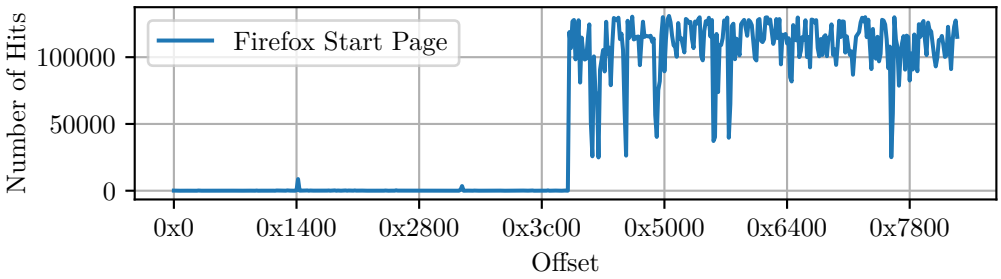
our six websites. Looking at these plots, we notice that the static websites' traces contain several significant dips compared to those obtained while profiling multimedia websites. Figure 6.12a shows us that the big dips in the averaged traces from the static websites are overlapping. For example, in all the plots of the static websites, we see dips at offset 0x74ff, 0x583f, and 0x49bf. These dips are not present in the plots of the multimedia websites. Therefore, we assume that the code of the shared library stored at the offsets where we measured significant dips is responsible for media playback.

We set up a convolutional neural network (CNN) to automatically interpret traces from websites. After training, we use this CNN to classify whether a website plays media. We use Keras [11] to implement our CNN. Our CNN consists of an input layer, a 1D convolutional layer, a batch normalization layer, a ReLU activation function layer, and an output layer. The 1D convolutional layer has 64 filters and a kernel size of 8. The output layer uses a softmax activation function. During training, we limit the number of iterations to 500.

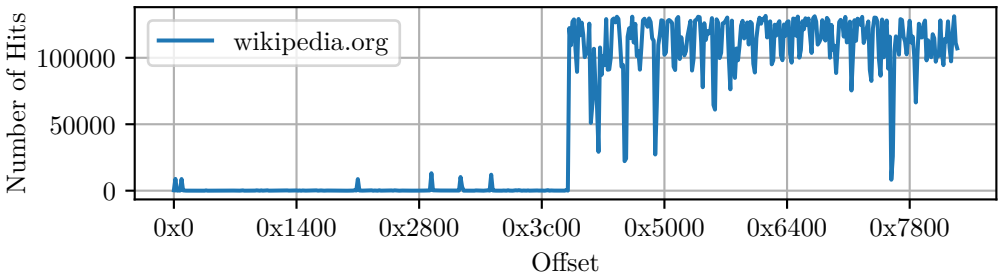
We could not collect additional traces to train and evaluate our CNN. Therefore, we reused the data used to create the plots in Figure 6.11. We profiled three static and three multimedia websites for Figure 6.11. From each website, we collected ten traces. Thus, we have only 60 traces available to train and evaluate our CNN. Due to the limited training data, the performance of the CNN fluctuates significantly depending on the data used for training and evaluation. To still estimate the performance of our CNN, we train the network through the traces of two static and two multimedia websites. Subsequently, we use the trained CNN to classify the traces from the remaining static and multimedia websites. We repeat these two steps and permute the data used for training and evaluation until we have covered all possible permutations. On average, we correctly detect whether a website plays media with an accuracy of 83.888 %. With more training data, we could likely achieve better accuracy.



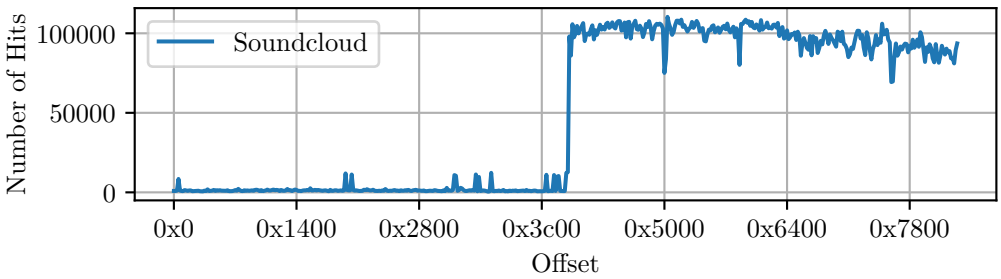
(a) Kernel.org



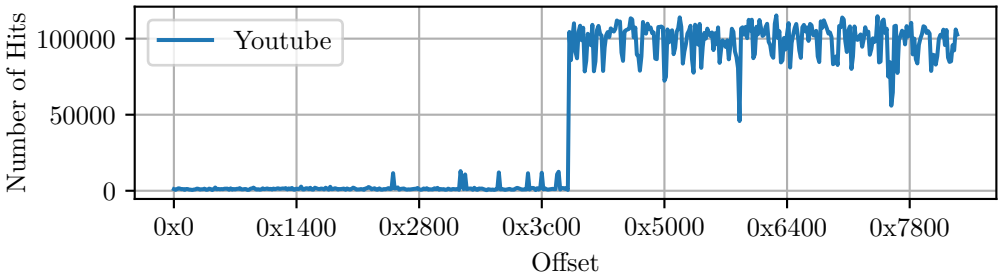
(b) Firefox default start page



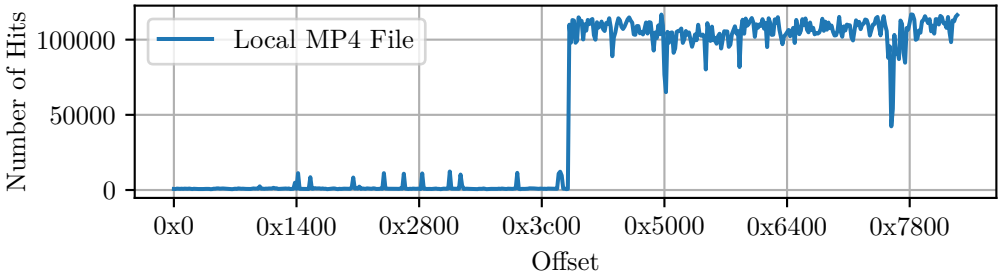
(c) wikipedia.org



(d) Soundcloud

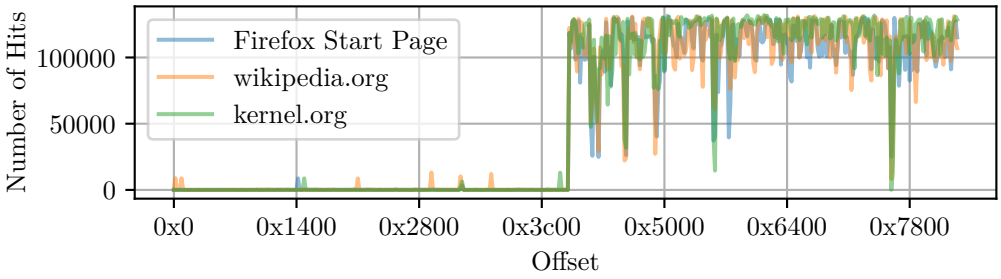


(e) YouTube

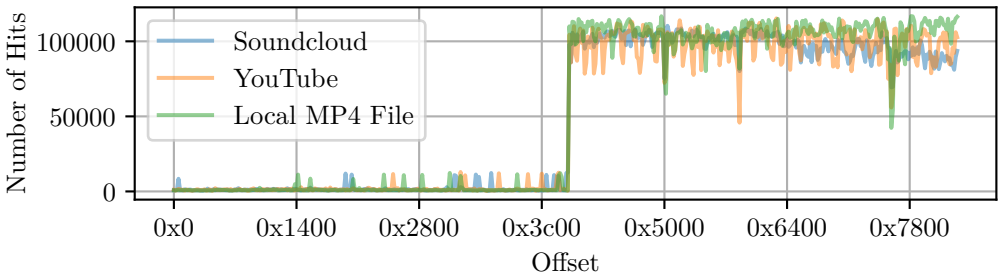


(f) Local MP4 file

Figure 6.11: In these plots, we see the averaged measurements of each profiled website. The traces of the static websites contain multiple significant dips, which are not present in the traces of the multimedia websites.



(a) Static web pages



(b) Media web pages

Figure 6.12: In Figure 6.12a, we see the averaged measurements from the three static websites in one plot. The significant dips in the plot's second half are noticeable and present at the same offsets in all traces. Similarly, Figure 6.12b displays the averaged measured data from the multimedia web pages. We can observe similarities between the traces of the different multimedia web pages. However, the traces in this plot are clearly distinguishable from those in Figure 6.12a because they have fewer significant dips.

Chapter 7

Conclusion

In this master thesis, we demonstrated that recent Apple CPUs are vulnerable to classic cache attacks, similar to traditional x86-64 CPUs. We developed a cache template attack that allows us to leak actions performed by a user in Mozilla Firefox running on an Apple Mac mini (M1). Furthermore, we demonstrated that an attacker can use this attack to obtain information about websites opened by a victim in Mozilla Firefox. Additionally, we analyzed the possibility of launching such an attack on an Apple mobile device, like an iPhone 13 mini. We concluded that it is difficult to launch cache template attacks on iPhones because iOS executes user-space applications in a sandboxed environment, preventing them from accessing other binaries.

We rely on the building blocks introduced by Hetterich et al. [25] to implement our covert channel [25]. They demonstrated that removing data from the CPU cache of an Apple Mac mini (M1) is possible using an eviction set-based approach. They also demonstrated that a counting thread could compensate for the missing CPU timers on Apple Silicon. Further, they showed that these building blocks could be used to set up an Evict+Reload-based covert channel, thus circumventing process isolation on an Apple Mac mini (M1). In our work, we demonstrated that we can also use these building blocks on an Apple iPhone 13 mini running an Apple A15 Bionic CPU. By strategically pinning our applications to CPU clusters, we significantly reduced noise in our timestamp introduced by frequency fluctuations and long scheduling queues. With these optimizations and machine-learning-based post-processing, we launched an Evict+Reload-based covert channel on an Apple iPhone 13 mini. This covert channel transmits data between two processes via the cache with an error rate of 7.13 % and a true capacity of 6.041 kbit/s. We also evaluated our covert channel on the Apple Mac mini (M1). On this device, we exchanged data with an error rate of 3.75 % and a true capacity of 122.46 kbit/s. This covert channel is currently the fastest Evict+Reload-based covert channel on ARM. Compared to state-of-the-art covert channels, our optimizations increased the true capacity by 1029 %. Finally, we implemented the first cache template attack on an Apple Silicon CPU based on the knowledge gained from the covert channel. We evaluated our attack on a recent Apple Mac mini (M1) running a recent macOS version. We demonstrated that our cache template attack can detect when a victim opens a new tab in Mozilla Firefox. Further, we demonstrated that this attack enables an attacker to detect if a website currently opened in Mozilla Firefox is playing media with an accuracy of 83.888 %.

Although launching a cache template attack on an Apple desktop CPU is more complex than on an x68-64 system, it is possible. We have only demonstrated that we could leak low-frequency events with our attack. However, Gruss et al. [21] have shown that cache template attacks can leak cryptographic keys from applications running on the same system. With further optimizations, our implementation could potentially recover cryptographic keys from applications running on Apple devices. In 2016, Van Der Veen et al. [50] demonstrated that Rowhammer attacks are possible on mobile ARM-based systems. Currently, Rowhammer attacks are not feasible on devices powered by Apple CPUs because there is no efficient and reliable method to evict data from the cache. If this limitation can be addressed, we may see such attacks on Apple devices in the future.

Bibliography

[1] Red Hat Inc. *Using the taskset Command to Set Processor Affinity*. 2023. URL: https://access.redhat.com/documentation/de-de/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/chap-affinity.

[2] Red Hat Inc. *Tuning CPUfreq Policy and Speed*. 2023. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/power_management_guide/tuning_cpufreq_policy_and_speed.

[3] Anant Agarwal and Stephen D Pudar. "Column-associative caches: A technique for reducing the miss rate of direct-mapped caches". In: *Annual international symposium on Computer architecture*. 1993.

[4] Apple Inc. *The new iPadOS powers unique experiences designed for iPad – apple.com*. 2019. URL: https://www.apple.com/newsroom/2019/06/the-new-ipados-powers-unique-experiences-designed-for-ipad.

[5] Apple Inc. *Prioritize Work with Quality of Service Classes – Apple Inc.* 2018. URL: https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/PrioritizeWorkWithQoS.html.

[6] Apple Inc. *Security. Built right in. – apple.com*. 2023. URL: https://www.apple.com/macos/security/.

[7] ARM. "ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile". In: 2013.

[8] ARM. "ARM Cortex-A Series Programmer's Guide for ARMv8-A". In: 2015.

[9] Asahi Linux Community. *Asahi Linux – asahilinux.org*. 2023. URL: https://asahilinux.org/.

[10] Billy Bob Brumley and Risto M Hakala. "Cache-timing template attacks". In: *ASIACRYPT*. 2009.

[11] François Chollet et al. *Keras*. https://keras.io. 2015.

[12] Serkan Özkan. *Top 50 Products By Total Number Of "Distinct" Vulnerabilities – cvedetails.com*. 2023. URL: https://www.cvedetails.com/top-50-products.php.

[13] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. "ECDSA key extraction from mobile devices via nonintrusive physical side channels". In: *ACM SIGSAC Conference on Computer and Communications Security*. 2016.

[14] Apple Inc. *darwin-xnu – github.com*. 2021. URL: https://github.com/apple/darwin-xnu.

[15] Libav. *Libav*. 2023. URL: https://github.com/libav/libav.

[16] LLVM.org. *llvm-project*. 2024. URL: https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/TargetParser/AArch64TargetParser.h.

[17] Mozilla. *gecko-dev*. 2023. URL: https://github.com/mozilla/gecko-dev.

[18] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. "AutoLock: Why Cache Attacks on ARM Are Harder Than You Think". In: *USENIX Security Symposium*. 2017.

[19] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. "Flush, gauss, and reload–a cache attack on the bliss lattice-based signature scheme". In: *International Conference on Cryptographic Hardware and Embedded Systems*. 2016.

[20] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+ Flush: a fast and stealthy cache attack". In: *DIMVA*. 2016.

[21] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache template attacks: Automating attacks on inclusive last-level caches". In: *USENIX Security Symposium*. 2015.

[22] Gregor Haas and Aydin Aysu. "Apple vs. EMA: electromagnetic side channel attacks on apple CoreCrypto". In: *ACM/IEEE Design Automation Conference*. 2022.

[23] Gregor Haas, Seetal Potluri, and Aydin Aysu. "itimed: Cache attacks on the apple a10 fusion soc". In: *IEEE International Symposium on Hardware Oriented Security and Trust*. 2021.

[24] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th. Morgan Kaufmann, 2017.

[25] Lorenz Hetterich and Michael Schwarz. "Branch Different-Spectre Attacks on Apple Silicon". In: *DIMVA*. 2022.

[26] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. "Applications of machine learning techniques in side-channel attacks: a survey". In: *Journal of Cryptographic Engineering* 10 (2020), pp. 135–162.

[27] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*. 2016.

[28] The iPhone Wiki. *Kernel Syscalls – theiphonewiki.com*. 2018. URL: https://www.theiphonewiki.com/wiki/Kernel_Syscalls.

[29] The kernel development community. *The Linux Kernel documentation – kernel.org*. 2023. URL: https://www.kernel.org/doc/html/latest/index.html.

[30] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *S&P*. 2019.

[31] Paul C Kocher. "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems". In: *Advances in Cryptology—CRYPTO*. 1996.

[32] Butler W Lampson. "A note on the confinement problem". In: *Communications of the ACM* 16.10 (1973), pp. 613–615.

[33] Sam Costello. *The History of iOS, from Version 1.0 to 16.0 – lifewire.com*. 2022. URL: `https://www.lifewire.com/ios-versions-4147730`.

[34] Moritz Lipp. "Exploiting Microarchitectural Optimizations from Software". In: *Diss., Graz University of Technology* (2021).

[35] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices". In: *USENIX Security Symposium*. 2016.

[36] Oleksiy Lisovets, David Knichel, Thorben Moos, and Amir Moradi. "Let's take it offline: Boosting brute-force attacks on iPhone's user authentication through SCA". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021).

[37] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." In: *NDSS*. 2017.

[38] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[39] Kevin P Murphy et al. "Naive bayes classifiers". In: *University of British Columbia* 18.60 (2006), pp. 1–8.

[40] messe. *Memory access on the Apple M1 processor – news.ycombinator.com*. 2021. URL: `https://news.ycombinator.com/item?id=25659615`.

[41] Red Hat Inc. *Apple Inc. Registered Products by Product Standard*. 2023. URL: `https://www.opengroup.org/openbrand/register/apple.htm`.

[42] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache attacks and countermeasures: the case of AES". In: *RSA Conference*. 2006.

[43] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[44] Colin Percival. "Cache missing for fun and profit". In: *BSDCan*. 2005.

[45] Franz Pernkopf and Christian Knoll. *Computational Intelligence*. 2021.

[46] Claude E Shannon. "A mathematical theory of communication". In: *The Bell system technical journal* 27.3 (1948), pp. 379–423.

[47] Petroc Taylor. *Mobile operating systems' market share worldwide from 1st quarter 2009 to 4th quarter 2022 – statista.com*. 2023. URL: `https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/`.

[48] Petroc Taylor. *Global market share held by operating systems for desktop PCs, from January 2013 to January 2023 – statista.com*. 2023. URL: `https://www.statista.com/statistics/218089/global-market-share-of-windows-7/`.

[49] Apple Inc. *Apple Platform Security – support.apple.com*. 2022. URL: `https://support.apple.com/guide/security/welcome/web`.

[50] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. "Drammer: Deterministic rowhammer attacks on mobile platforms". In: *ACM SIGSAC conference on computer and communications security*. 2016.

[51] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W Fletcher, and David Kohlbrenner. "Augury: Using data memory-dependent prefetchers to leak data at rest". In: *S&P*. 2022.

[52] Pepe Vila, Boris Köpf, and José F Morales. "Theory and practice of finding eviction sets". In: *S&P*. 2019.

[53] Zhenghong Wang and Ruby B Lee. "Covert and side channels due to processor architecture". In: *Annual Computer Security Applications Conference (ACSAC)*. 2006.

[54] John C Wray. "An analysis of covert timing channels". In: *Journal of Computer Security* 1.3-4 (1992), pp. 219–232.

[55] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *USENIX Security Symposium*. 2014.

[56] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W Fletcher. "Synchronization Storage Channels S^2C: Timer-less Cache Side-Channel Attacks on the Apple M1 via Hardware Synchronization Instructions". In: *USENIX Security Symposium*. 2023.

[57] Mohamed Zahran, Kursad Albayraktaroglu, and Manoj Franklin. "Non-inclusion property in multi-level caches revisited". In: *International Journal of Computers and Their Applications* 14.2 (2007), p. 99.

[58] Zixuan Zhang. "Analysis of the Advantages of the M1 CPU and Its Impact on the Future Development of Apple". In: *International Conference on Big Data & Artificial Intelligence & Software Engineering (ICBASE)*. 2021.