Moritz Lipp, BSc

Cache Attacks and Rowhammer on ARM

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Assessor
Prof. Stefan Mangard

Supervisor
Daniel Gruss

Institute for Applied Information Processing and Communications

Graz, October 2016

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.

Date	Signature

ABSTRACT

In the last years, mobile devices have become the most important personal computing platforms and, thus, it is especially important to protect sensitive information that is stored and processed on these devices. In this thesis, we discuss the applicability of cache attacks and the rowhammer bug on mobile devices. As these attacks have been considered infeasible on ARM-based devices, we demonstrate how to solve key challenges to mount the most powerful cache attack techniques *Prime+Probe*, *Flush+Reload*, *Evict+Reload* as well as *Flush+Flush* and how to induce bit flips. We show the immense power of these attacks by implementing a high-performance covert-channel, spying on user input and attacking cryptographic algorithms. Finally, we discuss possible countermeasures.

Keywords: Side-channel attacks, Cache attacks, Rowhammer, Mobile platforms, ARM, *Prime+Probe*, *Flush+Reload*, *Evict+Reload*, *Flush+Flush*, Cross-CPU attack

ACKNOWLEDGEMENTS

First and foremost, I want to thank my supervisor Daniel Gruss for sharing his passion with cache attacks and the rowhammer bug with me and his exceptional support throughout this thesis. Furthermore, I want to thank my professor Stefan Mangard for giving me this opportunity. I want to express my gratitude to Raphael Spreitzer and Clémentine Maurice without whom the success of the resulting research paper would not have been possible. In addition, I want to thank Michael Schwarz for his time working on the rowhammer bug with me.

I want to especially thank Johannes Winter and Sebastian Ramacher for the exchange of knowledge and meaningful discussions in extended coffee breaks over the last years. I also want to thank my friends for their loyal friendship and support.

Finally, I want to thank my parents, Peter and Michaela, as well as my siblings, Ilona, Lukas and Nikolaus, for all their love and support throughout my entire life. I want to be grateful to my better half, Natascha, for her supporting love and patience during my time working on this thesis.

Thank you.

Contents

Chapter 1

Introduction

In the last years, mobile devices like smartphones and tablets have become the most important personal computing platform. The default tasks like making phone calls and managing contacts in an address book have been heavily extended such that these devices assist us nowadays with our everyday tasks. Besides responding to emails and taking photos as well as videos, it is possible to approve bank transfers, create digital signatures of official documents or to use it as a navigation device for the car. These tasks require to gather, process and store more and more sensible personal data on the device which exposure to the public would not only be a fatal infringement of privacy but could also have disastrous consequences regarding for financial security, identity theft, and social acceptance. Thus, hardware and software-based protection mechanisms need to be in place to prevent unauthorised access and exfiltration of personal data to make such devices more secure.

Side channel attacks exploit an unintentional information leakage regarding the variation of power consumption, execution time or electromagnetic radiation to compromise the security of the executing platform or the underlying algorithm. In 1996 Kocher [45] showed that it is possible to find fixed Diffie-Hellman exponents, factor RSA keys and to break other cryptographic systems by measuring the execution time of private key operations. In 1999 Kocher et al. [46] introduced Simple Power Analysis (SPA) and Differential Power Analysis (DPA) where an attacker can extract cryptographic keys by studying the power consumption of a de-

vice. Based on the publication of Van Eck [19] in 1985 that showed the possibility of eavesdropping on video display units by measuring the electromagnetic interference, Callan et al. [15] demonstrated a key logger for a personal computer using a smartphone with a radio antenna in 2014. In 2016 Genkin et al. [21] present the first physical side-channel attack on elliptic curve cryptography by extracting the secret decryption key from a target located in another room.

A different side channel on modern computing architectures is introduced by the memory hierarchy that stores subsets of the computer's memory in smaller but faster memory units, so-called caches. Cache side-channel attacks exploit the different access times of memory addresses that are either held in the cache or the main memory. In 2014 Bernstein [11] demonstrated complete AES key recovery from known-plaintext timings of a network server. While the *Evict+Time* and *Prime+Probe* techniques by Osvik et al. [63] also explicitly targeted cryptographic algorithms, Yarom and Falkner [87] introduced the so-called *Flush+Reload* attack in 2014, which laid the foundation for new attack scenarios. It allows an attacker to determine which specific parts of a shared library or a binary executable have been accessed by the victim with an unprecedented high accuracy. Based on this work Gruss et al. [28] demonstrated the possibility to exploit cache-based side channels via cache template attacks in an automated way and showed that besides efficiently attacking cryptographic implementations, it can be used to infer keystroke information and even log specific keys.

A complete different type of attack that exploits computational errors to extract cryptographic key has been presented by Boneh et al. [14] in 1997. In the same year Biham and Shamir [12] described Differential Fault Analysis (DFA) attacks that use various fault models as well as various cryptanalysis techniques to recover secret values of presumably tamper-resistant devices like smart cards. If physical access is given, the power supply voltage, the frequency of the clock or the environment in terms of temperature or radiation can be changed, to force the smart card to malfunction. However, it is also possible to induce hardware faults by software, and thus from a remote location, if the device could be brought outside of the specified working conditions.

In 2014 Kim et al. [43] demonstrated that accessing specific memory locations in a high repetition rate can cause random bit flips in Dynamic Random-Access Memory (DRAM) chips. Since DRAM technology scales down to smaller dimensions, it is much more difficult to prevent single cells from electrically interacting with each other. They observe that activating the same row in the memory corrupts data in nearby rows. In 2015 Seaborn [72] demonstrated that this side effect could be exploited for privilege escalation and in the same year, Gruss et al. [26] showed that such bit flips can also be triggered by JavaScript code loaded on a website. However, this attack has so far only be demonstrated on Intel and AMD systems using DDR3 and modern DDR4 [25, 49] modules. So far, it is unknown if the rowhammer attack on ARM platforms is possible [49, 73].

With this thesis we will explore the applicability of cross-core cache-attacks and the rowhammer bug on mobile devices. The results of this work have been published as a part of a research paper, "ARMageddon: Cache Attacks on Mobile Devices", at the USENIX Security 2016 conference [50] and will be presented at the Black Hat Europe 2016 conference [51].

1.1 Motivation

Although mobile devices have become the most important personal computing platform and cache attacks represent a powerful way of exploiting the memory hierarchy of modern system architectures, only a few publications on cache attacks on smartphones exist. Also, they exclusively focus on attacks on AES-table implementations [13, 75–77, 84] and do not discuss the more recent and efficient cross-core attack techniques *Prime+Probe* [87], *Flush+Reload* [87], *Evict+Reload* [26], and *Flush+Flush* [27], nor the Rowhammer attack [43]. In fact, Yarom and Falkner [87] doubted that cross-core attacks can be mounted on ARM-based devices at all.

Since smartphones are storing more and more sensitive data as well as collecting personal information, it is especially important to furthe investigate the ARM platform with respect to cache attacks and the rowhammer bug.

1.2 Key Challenges and Results

Existing cross-core cache attacks [27, 28, 30, 35, 52, 54, 55, 62, 87] rely on the property that last-level caches are inclusive. While the x86 architecture always fulfils this requirement, only the recent ARMv8 architecture uses inclusive last-level caches. Thus, it was believed that these attacks are not applicable on the ARM architecture [87]. However, to make sure that multiple copies of data that reside in different caches are up-to-date, so-called cache coherence protocols are used. These protocols can be exploited to mount cross-core cache attacks on mobile devices with non-inclusive shared last-level caches. In addition, these protocols also can be exploited on modern smartphones that have multiple CPUs and do not share a cache, because these protocols allow CPUs to load cache lines from remote caches faster than from the main memory.

Attack techniques like *Flush+Reload* and *Flush+Flush* utilize the unprivileged x86 flush instruction `clflush` to evict a cache line. But with the exception of ARMv8-A CPUs, ARM processors do not have an unprivileged flush instruction, and therefore cache eviction must be used. In order to acquire measurements in a high-frequency that are required for recent cache attacks, the eviction has to be fast enough. Previously proposed eviction strategies are too slow [76]. In order to find fast eviction strategies, we utilized Rowhammer attack techniques [26] to evaluate several thousand different eviction strategies automatically.

In order to obtain cycle accurate timings, previous attacks [13, 75–77, 84] relied on the performance counter register that is only accessible with root privileges [4]. We evaluated possible alternative timing sources that are accessible without any privileges or permissions and show that they all can be used to mount cache attacks.

In addition, a pseudo-random replacement policy decides on ARM CPUs which cache line will be evicted from a cache set. This policy introduces additional noise [75] for robust time-driven cache attacks [77] and is the reason why *Prime+Probe* could not be mounted on ARM so far [76]. However, we find access patterns to congruent addresses that allow us to successfully launch *Prime+Probe* attacks despite the pseudo-random replacement policy.

After solving these key challenges systematically, we used the resulting building blocks to demonstrate the wide impact of cache attacks on ARM. In contrast to previous cache attacks on smartphones [13, 75–77, 84] we do not only attack cryptographic implementations but also utilize these attacks to infer various sensitive information, e.g., to differentiate between entered letters and special keys on the keyboard or the measure the length of swipe and touch gestures. In addition, we show that cache attacks can also be applied to monitor the activity of the cache caused by the ARM TrustZone.

Furthermore, we show that ARM-based devices are also vulnerable to the rowhammer bug [43] by inducing bit flips in various ways. In addition, we have reverse engineered the DRAM addressing functions of several smartphones with techniques by Pessl et al. [67] and, thus, increased the likelihood and the number of bit flips.

With this work, we aim to show that despite reasonable doubt well-studied attack techniques are also applicable on smartphones and to demonstrate the immense threat of the presented attacks. Since those attacks can be mounted on millions of stock Android devices without the requirement of any privileges or permissions, it is crucial to deploy effective countermeasures.

1.3 Contributions

The main contributions of this master thesis can be summarized as follows.

- We show that all of the highly efficient cache attacks like *Prime+Probe*, *Flush+Reload*, *Evict+Reload*, and *Flush+Flush* can be mounted on ARM-based devices.
- Thousands of different eviction strategies for various devices have been evaluated in order to determine the best strategies that can be used on devices where no unprivileged flush instruction is available.
- Three possible timing sources have been identified as an alternative to the privileged performance registers. They are accessible without

any privileges or permissions and each of them can be used to mount cache attacks on ARM successfully.

- The demonstrated attacks can be applied independently of the actual cache organization and, thus, also on non-inclusive last-level caches. In particular, the attacks can be mounted against last-level caches that are instruction-inclusive and data-non-inclusive as well as against caches that are instruction-non-inclusive and data-inclusive.
- We are the first to show last-level cache attacks on ARM-based devices that can be applied cross-core and also cross-CPU.
- Our developed cache-based covert channel is more than 250 times faster than all existing covert channels on Android.
- We use these attacks to attack cryptographic implementations in Java, to monitor the TrustZone's cache activity and to spy on user input.
- All techniques have been implemented in the form of a library called `libflush` that allows to easily develop platform-independent cache attacks for the x86 as well as the ARM platform and can be extended for additional platforms quite easily. All of the described attacks have been built on top of this library.
- ARM-based devices are also vulnerable to the rowhammer bug and bit flips can be triggered in a reliable way.

1.4 Test devices

In this section, we want to describe the test devices as listed in Table 1.1, that we have used to demonstrate the attacks presented in Chapter 4.

The OnePlus One uses a Snapdragon 801 SoC with a Krait 400 CPU that is an ARMv7-A CPU with a 4-way 2×16 KB L1 cache and a non-inclusive shared 8-way 2 048 KB L2 cache.

The Alcatel One Touch Pop 2 uses a Snapdragon 410 SoC with a Cortex-A53 ARMv8-A CPU. However, the stock Android ROM is compiled for the ARMv7-A instruction set, and thus ARMv8-A instructions are not used.

Table 1.1: Test devices used in this thesis.

Device	System on Chip	CPU (cores)	L1 caches	L2 cache	Inclusiveness
OnePlus One	Qualcomm Snapdragon 801	Krait 400 (2) 2.5 GHz	2× 16 KB, 4-way, 64 sets	2 048 KB, 8-way, 2 048 sets	non-inclusive
Alcatel One Touch Pop 2	Qualcomm Snapdragon 410	Cortex-A53 (4) 1.2 GHz	4× 32 KB, 4-way, 128 sets	512 KB, 16-way, 512 sets	instruction-inclusive, data-non-inclusive
Samsung Galaxy S6	Samsung Exynos 7 Octa 7420	Cortex-A53 (4) 1.5 GHz	4× 32 KB, 4-way, 128 sets	256 KB, 16-way, 256 sets	instruction-inclusive, data-non-inclusive
		Cortex-A57 (4) 2.1 GHz	4× 32 KB, 2-way, 256 sets	2 048 KB, 16-way, 2 048 sets	instruction-non-inclusive, data-inclusive

It has a 4-way 4×32 KB L1 cache and a 16-way 512 KB L2 cache that is inclusive on instruction side and non-inclusive on the data side.

The Samsung Galaxy S6 uses a Samsung Exynos 7 Octa 7420 SoC that has two ARMv8-A CPU clusters and the big.LITTLE [7] technology. The first CPU is a Cortex-A53 with a 4-way 4×32 KB L1 cache and a 16-way 256 KB instruction-inclusive and data-non-inclusive L2 cache. The second CPU is a Cortex-A57 with a 2-way 4× 32 KB L1 cache and a 16-way 2 048 KB L2 cache that is non-inclusive on instruction side and inclusive on the data side.

1.5 Outline

The remainder of this thesis is structured as follows. Chapter 2 provides background information on CPU caches, DRAM, and the rowhammer bug. In addition, it discusses cache coherency and presents different cache attacks.

In Chapter 3, the techniques that are the building blocks of our attacks are shown: The model of eviction and the identification of optimal eviction

strategies is described in Section 3.2. Different timing sources that can be used to successfully mount cache attacks on devices where no dedicated performance counter can be accessed are identified and evaluated in Section 3.1.

Chapter 4 explains how the cache organization is defeated and demonstrates powerful attack scenarios that can be performed using our findings. We present and evaluate fast cross-core and cross-CPU covert channels on Android in Section 4.1. Section 4.2 demonstrates cache template attacks on user input events. Attacks on cryptographic implementations used in practice as well the possibility to observe cache activity of cryptographic computations within the TrustZone are shown in Section 4.3. In Section 4.4 we discuss different methods to induce the rowhammer bug on ARM-based devices successfully.

We discuss countermeasures in Chapter 5 and conclude and summarize this thesis in Chapter 6.

Chapter 2

Background

In this chapter, we want to discuss the background information necessary to understand cache attacks. In Section 2.1 we will give a deeper insight into CPU caches, define their properties and show the difference between Intel and ARM CPU caches. Section 2.2 will explain cache coherence protocols that guarantee that data is coherent in all caches on the system. The concept of shared memory will be described in Section 2.3, and at last in Section 2.4 various cache attack techniques will be discussed. In Section 2.5.2 we will see that that it is possible to reverse-engineer the undocumented DRAM mapping functions that can be used to find addresses that are more suited for performing the rowhammer attack.

2.1 CPU caches

The performance of the CPU today does not only depend on its clock frequency, but it is also influenced by its latency of instructions and interactions with other devices. Since computer memory should be served as fast as possible to the CPU, a memory hierarchy as shown in Figure 2.1 is employed as a solution to overcome the latency of system memory accesses. Frequently used data is buffered in multiple layers of fast and small memories, so-called caches.

Since high-speed memory is expensive, the memory hierarchy is organized into multiple levels where each level that is closer to the CPU is

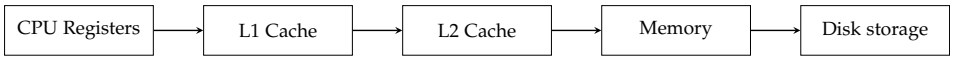
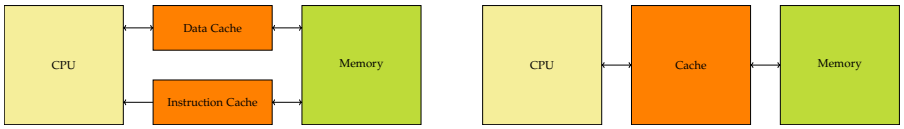


Figure 2.1: Memory hierarchy

smaller, faster and more expensive. Accesses to the cache memory are significantly quicker than those to the main memory.



(a) Harvard architecture (b) Von Neumann architecture

Figure 2.2: The Harvard architecture and the Von Neumann architecture.

For performance reasons the first level cache (L1) is typically connected directly to the core logic which fetches instructions and handles load and store instructions. While a von-Neumann architecture, illustrated in Figure 2.2b, uses a single cache for instruction and data (*unified cache*), the Harvard architecture, illustrated in Figure 2.2a, has two separate buses for instruction and data. Therefore, it consists of two caches, an instruction cache (I-Cache) and a data cache (D-Cache) which allows transfers to be performed simultaneously on both buses and therefore increases the overall performance and allows larger L1 caches.

A program tends to reuse the same addresses over time repeatedly (temporal locality) and addresses that are near to each other (spatial locality) are likely used as well. E.g., if a program uses a loop, the same code gets executed over and over again.

Thus, the goal is to buffer code and data, that is often used, in faster memory making subsequent accesses quicker and therefore the program execution significantly faster. However, this is also a disadvantage that does not exist in a core without caches: The execution time can vary widely, depending on which pieces of codes are cached. This could lead to a problem in real-time systems that expect strongly deterministic behaviour [60].

The cache only holds a subset of the contents of the main memory. Thus, it must store both the address of the item in the main memory as well as the associated data. When the core wants to read or write a particular address, it will look into the cache. If a word is not found in the cache, the

word must be fetched from a lower level in the memory hierarchy which could be another cache or the main memory and be loaded into the cache before continuing. For efficiency reasons multiple words, further called cache line or block, are moved at the same time, because they are likely to be needed due to spatial locality such that access times for subsequent loads and stores are reduced. Each cache block contains a tag to indicate which memory address it corresponds to.

2.1.1 Direct-mapped cache

There are different ways to implement a cache where the simplest one is a *direct-mapped* cache. In a direct-mapped cache, each entry in the main memory can only go to one location in the cache which yields to the result that many addresses will map to the same cache location.

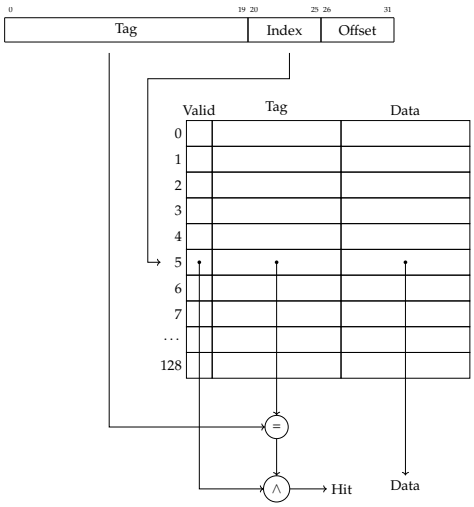


Figure 2.3: Direct-Mapped Cache

Figure 2.3 illustrates a 4KB direct-mapped cache with 128 locations and 64-byte line size. The cache controller will use 6 bits ($2^6 = 64$) of the address as the offset to select a word within the cache line and 7 bits to determine the location index to which the address is mapped to. The remaining 19 bits ($32 - 7$ bits (index) - 6 bits (block offset)) will be stored as the tag value. An additional bit is used to determine if the content of the entry is valid.

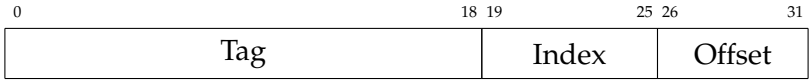


Figure 2.4: Direct-Mapped Cache Address

To look up a particular address in the cache the index bits of the address are extracted, and the tag value that is associated with that line in the cache is compared to the tag value of the address. If they are equal, and the valid bit in said cache line is set, it is a cache hit. Using the offset portion of the address, the relevant word of the cache line data can be extracted and used. However, if the valid bit is not set or the tag in the cache line differs from the one of the address, a cache miss happens, and the cache line has to be replaced by data from the requested address.

2.1.2 Set-associative caches

If the replacement policy can choose any entry in the cache to hold the data, the cache is called *fully associative*. Modern caches are organized in sets of cache lines, which is also known as *set-associative* caches. If one copy can go to any of N places in the cache, it is called *N-way associative*. Thus, each memory address maps to one of these cache sets and memory addresses that map to the same cache set are called congruent. Congruent addresses compete for cache lines within the same set where the previously described replacement policy needs to decide which line will be replaced.

For illustration purposes, we take a look at the L2 cache of the Alcatel One Touch Pop 2, a phone used in experiments for this thesis. The L2 cache has a capacity of 512KB and is a 16-way cache with a block size of 64 bytes. Thus, the cache index requires

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Line size} \times \text{Set associativity}} = \frac{512\text{KB}}{64 \times 16} = 512 = 2^9$$

or 9 bits to map each of the 512 sets.

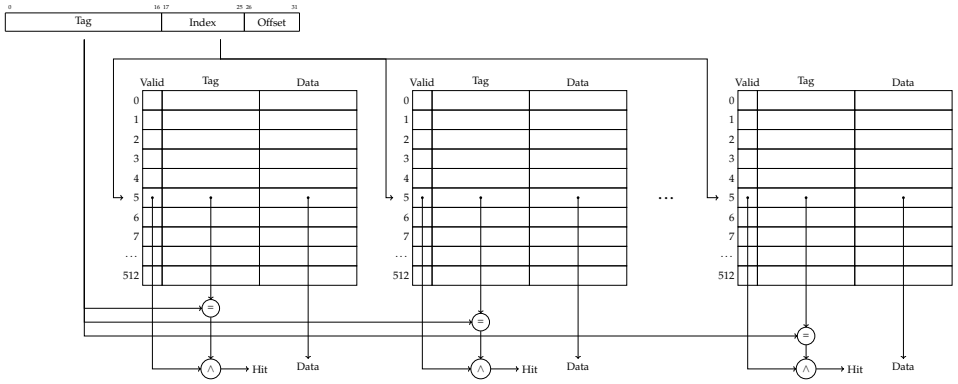


Figure 2.5: N-way associative cache

2.1.3 Translation Lookaside Buffer (TLB)

If a system uses virtual memory, the memory space is often split into fixed sized segments called pages. Those are mapped from the virtual address space to the physical address space via a page table. Protection flags are included in each entry of said page table. With paged virtual memory every memory access would take much longer since one memory access is required to obtain the physical address and a second one is used to get the data. To improve on this issue, the principle of locality is used. As described previously, memory accesses have the property of locality. Thus, also the address translations have locality. If the address translations are kept in a particular cache, then a memory access will rarely require a second access for the translation. This cache is called Translation Lookaside Buffer (TLB) and holds the virtual and physical address. If the requested address is present in the TLB, it is referred to as a TLB-Hit. If the requested address is not present in the TLB, it is called a TLB-Miss, and a page table structure is walked in the main memory to perform the translation.

2.1.4 Cache-replacement policy

When the cache is full and a cache miss occurs, buffered code and data must be evicted from the cache to make room for new cache entries. The heuristic that is used to decide which entry to evict is called replacement policy. The replacement policy has to decide which existing cache entry is least likely to be used in the near future. Least-Recently Used (LRU)

is a replacement policy that replaces the least recently used cache entry. ARM processors use a pseudo-LRU replacement policy for the L1 cache. Two different cache replacement policies, namely round-robin and pseudo-random replacement policy, can be used for the last-level cache. However, due to performance reasons only the pseudo-random replacement policy is used in practice. We will refer to the following cache-replacement policies throughout this work:

Least-recently-used replacement policy
The least recently used cache entry will be replaced.

Round-Robin replacement policy
A very simple replacement policy that will replace the first entry that has been loaded into the cache. The next replacement will evict the second entry that has been loaded etc.

Pseudo-random replacement policy
A random cache entry will be selected and evicted based on a pseudo-random number generator.

2.1.5 Virtual and physical tags and indexes

CPU caches can be *virtually indexed* and *physically indexed* and, thus, derive the index from the virtual or physical address respectively. Virtually indexed caches are faster in general because they do not require virtual to physical address translation before the cache lookup. Using the virtual address can lead to the situation that the same physical address is cached in different cache lines. Again, this reduces the performance. To uniquely identify the actual address that is cached within a specific line, the tag is used. This tag can also be based on the virtual or physical address. The advantages and disadvantages of the various possibilities are as follows:

VIVT - virtually indexed, virtually tagged
The virtual address is used for both, the index and the tag, which improves performance since no address translation is needed. However, the virtual tag is not unique and shared memory may be held more than once in the cache.

PIPT - physically indexed, physically tagged
The physical address is used for both, the index and the tag. This method is slower since the virtual address has to be looked up in the TLB. However, shared memory is only held once in the cache.

PIVT - physically indexed, virtually tagged
The physical address is used for the index, and the virtual address is used for the tag. This combination has no benefit since the address needs to be translated, the virtual tag is not unique and shared memory still can be held more than once in the cache.

VIPT - virtually indexed, physically tagged
The virtual address is used for the index, and the physical address is used for the tag. The advantage of this combination compared to PIPT is the lower latency since the index can be looked up in parallel to the TLB translation. However, the tag can not be compared until the physical address is available.

2.1.6 Inclusiveness

CPUs use multiple levels of caches where levels closer to the CPU are usually faster and smaller than the higher levels. As illustrated in Figure 2.6 the Alcatel One Touch Pop 2 employs a 4-way 32KB L1 cache with 128 sets and a 16-way 512KB L2 cache with 512 sets. A modified Harvard architecture is used such that there are separate L1 caches for instructions and data using the same address space. While each of the four cores has their private L1 cache, the last level (L2) cache is shared amongst all cores.

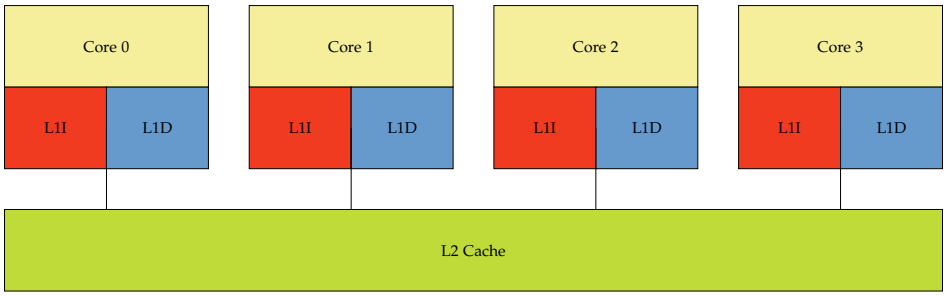


Figure 2.6: Cache hierarchy on the Alcatel One Touch Pop 2

If a word is read from the cache, the data in the cache will be identical to the one in the main memory. However, when the core executes a store instruction, a cache lookup to the address that is written to is performed. If a cache hit occurs, there are two possible policies:

Write-back policy

With the *write-back* policy, writes are performed only on the cache and not to the main memory which means that cache lines and the main memory can contain different data. To mark the cache lines with the newer data, an associated *dirty bit* is used that is set when a write happens that updates the cache but not the main memory. If the replacement policy evicts a cache line where the dirty bit is set, it is written out to the main memory.

Write-through policy

With the *write-through* policy, writes are performed to both the cache and the main memory which means that they are kept coherent. Since there are more writes to the main memory, this policy is slower than the write-back policy.

If the memory hierarchy consists of multiple levels of caches, some design decisions have to be defined with respect to which cache levels hold copies of the data.

Inclusive cache

A higher-level cache is called *inclusive* with regard to the lower-level cache if all cache lines from the lower-level cache are also stored in the higher-level cache.

Exclusive cache

Caches are called *exclusive* if a cache line can only be kept in one of two cache levels.

Non-inclusive cache

If a cache is neither *inclusive* nor *exclusive*, it is called *non-inclusive*.

While modern Intel CPUs have inclusive last-level caches, and AMD CPUs have exclusive last-level caches, most ARM CPUs have non-inclusive last level caches. However, with the release of ARM Cortex-A53/Cortex-A57

CPUs an inclusive last-level cache is used. E.g., the ARM Cortex-A53 is inclusive on the instruction side and exclusive on the data side.

2.1.7 Invalidating and cleaning the cache

It can be required to clean and invalidate the memory in the cache when the content of external memory has been changed, and stale data should be removed from the cache.

2.1.7.1 Intel

The Intel x86 architecture provides the `clflush` instruction that invalidates the cache line containing the passed address from all levels of the cache hierarchy (data and instruction). If a line at any level of the hierarchy is dirty, it is written back to memory before invalidation [33]. With the 6th generation of Intel CPUs, the `clflushopt` instruction has been introduced which has a higher throughput than the `clflush` instruction. Both instructions are not privileged and thus available to the userspace [33].

2.1.7.2 ARM

ARM employs cache invalidate and cache clean operations that can be performed by cache set, or way, or virtual address. Furthermore, they are only available to the privileged modes and cannot be executed in userspace. Invalidate and clean are defined as follows [6]:

Invalidate
: If a cache line is invalidated, it is cleared of its data by clearing the valid bit of the cache line. However, if the data in the cache-line has been modified, one should not only invalidate it, because the modified data would be lost as it is not written back.

Clean
: If a cache line is cleaned, the contents of the dirty cache line are written out to main memory, and the dirty bit is cleared, which makes

the contents of the cache and the main memory coherent. This is only applicable if the write-back policy is used.

If the cache invalidates and cache clean operations are performed by the cache set or way, they are executed on a specified level of the cache. In contrast, operations that use a virtual address are defined by two conceptual points [5]:

Point of Coherency (PoC)

The point at which all blocks, e.g., cores or DMA engines, that can access the memory is guaranteed to see the same copy of the address is called Point of Coherency (PoC). Commonly, this will be the main external system memory.

Point of Unification (PoU)

The point at which the instruction cache and the data cache of the core are guaranteed to see the same copy of the address is called Point of Unification (PoU). For example, a unified level 2 cache would be the PoU in a system with a modified Harvard level 1 cache and a Translation Lookaside Buffer (TLB). The main memory will be the PoU if there is no external cache present.

In contrast to the ARMv7 architecture, the ARMv8 architecture defines the `SCTLR_EL1` register that contains a bit described as `UCI`. If this bit is set, userspace access is enabled for the following four instructions [5]:

Instruction	Description
`DC CVAU`	Data or Unified Cache line Clean by VA to PoU
`DC CIVAC`	Data or Unified Cache line Clean and Invalidate by VA to PoC
`DC CVAC`	Data or Unified Cache line Clean by VA to PoC
`IC IVAU`	Instruction Cache line Invalidate by VA to PoU

Table 2.1: The cache maintenance instructions that can be accessed from userspace if the `SCTLR_EL1.UCI` bit is set.

The reset value of the `SCTLR_EL1.UCI` is architecturally unknown. However, experiments show that on the Samsung Galaxy S6 it is set by default. Thus, the possibility to flush the cache from userspace is given without the need of a kernel module that sets this bit in privileged mode.

2.2 Cache coherence

Modern systems implement shared memory in hardware allowing each core of the processor to read and write in a single shared address space. This enables part of applications to run simultaneously on different cores working on the same memory. To make sure that multiple cached copies of data that reside in different cores are up-to-date, so-called cache coherence protocols are defined [74].

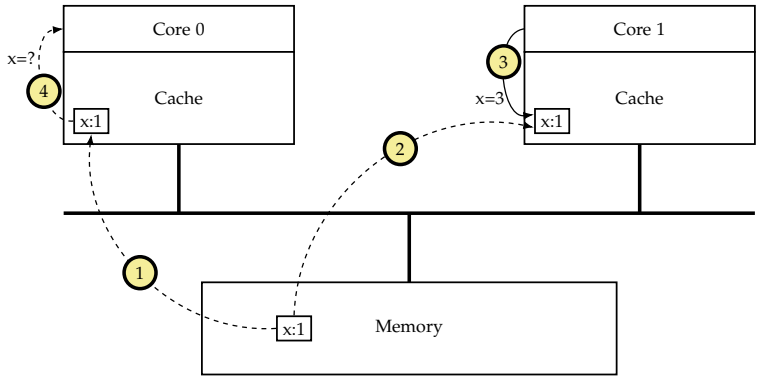


Figure 2.7: Example cache coherence problem. x is a location in the main memory. In step 1 the first processor reads the value of x and in step 2 the second processor does the same. In step 3 the second processor sets x to a new value, namely 3. If, as illustrated in step 4, the first processor re-accesses the value of x again, it would read a stale value out of its cache.

Figure 2.7 illustrates the necessity of cache coherence and shows two processors with caches connected by a bus to the main memory: x represents an arbitrary location in the memory with the value 1 that is read and written to by both processors. In the first step, the first processor reads the value out of the main memory into its cache. In the second step, the second processor does the same, before it updates its copy of x to the new value 3 in step 3. In step 4 the first processor accesses its copy of x again, which is now stale and neither it nor the main memory contains the up-to-date value of x, namely 3.

Such coherence problems can even occur in single processor systems when I/O operations are performed by Direct Memory Access (DMA) devices that move data between the main memory and peripherals [17]. E.g., if such a device would sit on the bus of Figure 2.7 and would access the memory location x of the main memory at step 4, it would read a stale

value, because the most up-to-date value is still in the cache of the second processor. Similarly, if such a device would write to location x, both processors would read stale values from their caches.

There are three mechanisms that maintain coherency between the caches [6]:

Disabling caching
: The most simple mechanism is to disable the caching mechanism. However, this brings also the highest performance loss, because every data access from every core has to be fetched from main memory impacting both power and performance.

Software-managed coherency
: The historical solution to manage coherency between the caches and the main memory is software-managed coherency. Software must clean dirty data and invalidate old data to enable sharing with other processes.

Hardware-managed coherency
: The most efficient solution is hardware managed coherency. It will guarantee that any data that is shared between cores will always be up-to-date in every core. While on the one hand, it adds complexity, on the other hand, it will greatly simplify the software and enable coherency to software applications that do not implement it themselves.

Since reads and writes of shared data occur in a high frequency in multiprocessor systems, disabling the cache or invoking the operating system on all references to shared memory is no option. Therefore, cache coherence should be solved in hardware.

One approach are *Directory-based* cache coherence protocols in which the sharing status of a particular cache line is kept in one location called *directory* [74] and, thus, everything that is shared is kept at one central place. However, solutions based on bus snooping will be discussed in the following section in more detail as they are employed on ARM-based devices.

2.2.1 Bus snooping

Cache coherency problems can be tackled in a simple and elegant way using the very nature of the bus. A bus is a set of wires that connects several devices. Each device can observe every action on the bus, e.g., every read and write to the bus. If a processor issues a request to its cache, the cache controller examines the current state of the cache. Depending on the state it takes a suitable action, e.g., a bus transaction to access memory. As illustrated in Figure 2.8 the coherence property is preserved as every cache controller *snoops* on the bus, thereby monitoring occurring transactions and taking action if such transaction is pertained to it, e.g., a write to a memory block of which it has its own copy in the cache [22]. The coherence is managed at the granularity of a cache line and, thus, either an entire cache line is valid or invalid.

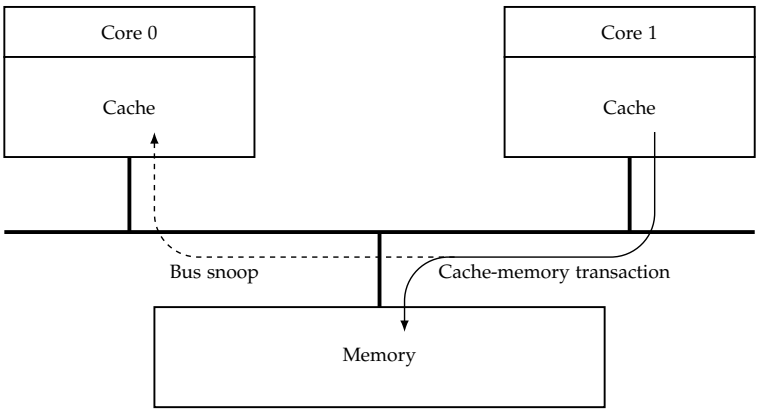


Figure 2.8: Bus snoop - Core 1 performs a memory transaction while Core 0 snoops on on the bus to react if the performed transaction is relevant to it.

If a cache that snoops on the bus owns a copy of the cache line, it has two options: The cache can either invalidate its copy or update it with the new data directly. Protocols that invalidate other cache copies are called *invalidation-based protocols* and protocols that update their cache copies are called *update-based protocols* [17]. In both cases the processor will get the most recent value either through a cache miss or because the updated value will be in the cache already.

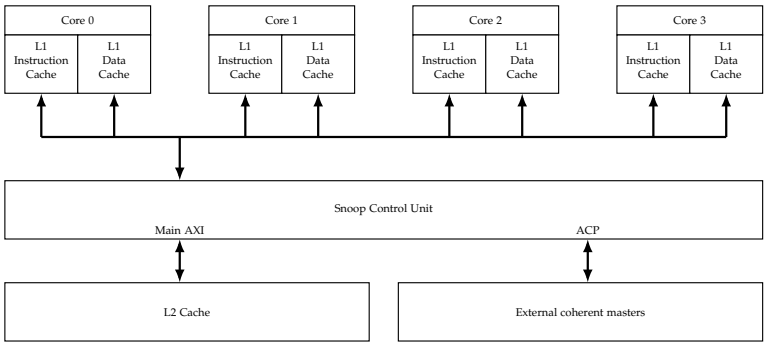


Figure 2.9: Snoop Control Unit.

2.2.1.1 Bus snooping on ARM

On ARM Cortex-A systems a Snoop Control Unit (SCU) connects up to four processors to the memory system through the Advanced eXtensible Interface Bus (AXI) interface [6]. It maintains data cache coherency between the cores and arbitrates L2 requests from the CPU cores and the Accelerator Coherency Port (ACP). The ACP port can be used to connect a DMA engine or a non-cached coherent master. Figure 2.9 illustrates the SCU of an ARM Cortex-A system.

Modern smartphones like the Samsung Galaxy S6 use multiple CPUs and employ ARMs big.LITTLE power management technology where high-performance CPU cores are combined with the low-power CPUs [7]. Figure 2.10 shows a simplified illustration of one Cortex A-57 CPU cluster and one Cortex-A53 CPU cluster that are connected via a ARM CoreLink CCI-400 Cache Coherent Interconnect [10]. This interconnect enables full cache coherence between two CPU clusters as well as I/O coherency for devices such as the GPU.

2.2.2 Coherence protocols

In this subsection, we will discuss snoopy cache coherence protocols in general before taking a more detailed look at the MESI protocol in Section 2.2.2.2 and the MOESI protocol in Section 2.2.2.3 as they are used on the ARM platform. The MOESI protocol is used in the ARM CoreLink CCI-400 [10] to provide cache coherence between two multi-core clusters,

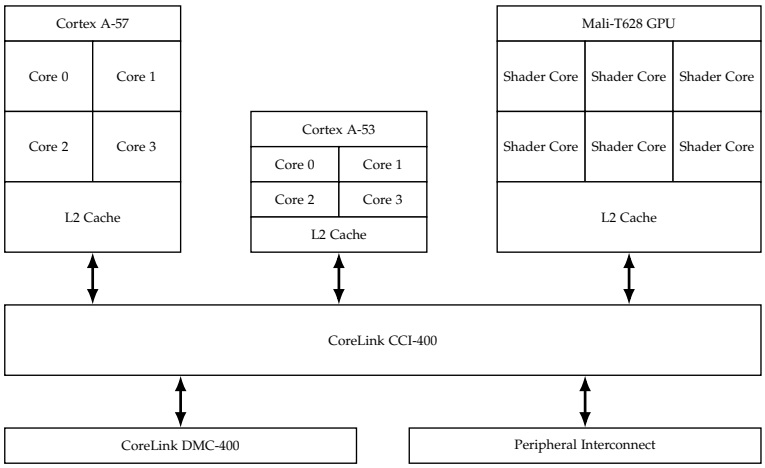


Figure 2.10: Simplified illustration of ARMs big.LITTLE technology that connects high-performance CPU cores with the low-power CPU cores and uses an interconnect to establish system wide hardware coherency and virtual memory management.

namely the ARM Cortex-A53 and the ARM Cortex-A57. While the Cortex A53 uses the MOESI protocol itself to maintain data coherency between multiple cores [9], the Cortex A-57 uses a hybrid form of the MESI and the MOESI protocol [8].

In general, a snoopy cache coherency protocol consists out of bus transactions and a cache policy. Bus transactions are needed for the communication of the devices on the bus and consist of three phases: In the first phase, called the arbitration phase, devices that want to send data over the bus assert their bus request. Then the bus arbiter selects one of the requesting devices by asserting its grant signal. The selected device places the command, e.g., read or write, on the command lines and the associated address on the address lines. All other devices on the bus will read the address and the one responsible for it will respond.

The cache policy is defined as a finite state machine that can be viewed as a *cache line state transition diagram*. For a single processor system with a write-through cache, the cache policy consists only of two states, namely *valid* and *invalid* [31]. The initial state of each cache line is *invalid*. If a processor generates a cache miss, a bus transaction to load the line from memory is generated, and the cache line is then marked as *valid*. Because the cache is write-through, a bus transaction to update the memory is generated if a write occurs. Additionally, if the cache line is marked as

valid in the cache, it is updated as well. Otherwise, writes do not change the state of the cache line. However, if a write-back cache is in place an additional state is required to indicate a *dirty* cache line [17].

If the system has multiple processors, each cache line has a state in each cache of the system that also changes accordingly to the state transition diagram. Hence, we describe a cache line's state as a vector of n states where n is the number of caches [17]. The state is managed by n finite state machines implemented by each of the cache's controller. The state transition diagram is the same for all cache lines and all caches. However, the current states of a cache line may differ from cache to cache.

2.2.2.1 Snoopy cache coherence protocols

In a snoopy cache coherence protocol, each cache controller has not only to consider the memory requests issued by the processor itself, but also all transactions from other caches that have been snooped from the bus. For each input the controller received, it has to update the state of each corresponding cache line according to its current state and the input. A snoopy cache coherence protocol is defined by [17]:

- A set of states associated with cache lines in the cache.
- A state transition diagram that takes the current state as well as the processor requests or snooped bus transactions as the input and generates a new state as the output.
- Actual actions associated with each state transition.

For example Figure 2.11 illustrates the transition state diagram for a write-through cache [31]. Each cache line has only two possible states, *Invalid* (I) and *valid* (V), and each transition is labelled with the input that has caused the transition as well as the output that is generated by the transition. The processor can issue two types of requests: A processor read (`PrRd`) and a processor write (`PrWr`) to a memory block that can be in the cache or not. When the processor tries to read a memory block that is not in the cache, a Bus Read (`BusRd`) transaction is generated. If the transaction is complete, the new state is the *valid* state. When the processor issues a write to a location, a bus transaction is generated as well, but the state

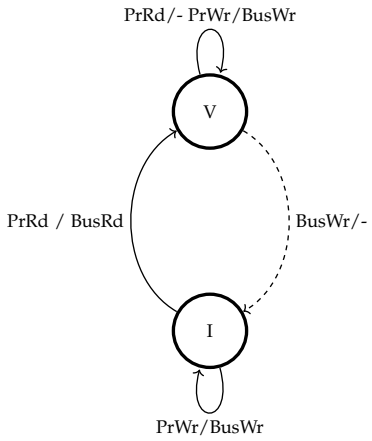


Figure 2.11: A snoopy coherence protocol for a multiprocessor with write-through no-write-allocate caches [31].

does not change. However, the key difference of this protocol compared to a single processor state diagram is that whenever a write transaction (`BusWr`) is read from the bus, the cache controller marks its copy as *invalid*. Therefore, if any cache controller generates a write for a cache line on the bus, all other cache controllers will invalidate their copies making them coherent.

2.2.2.2 MESI protocol

The Modified Exclusive Shared Invalid (MESI) protocol is the most common protocol that is used for write-back caches and has been introduced by Papamarcos and Patel [65] in 1984. The protocol uses four different states for each cache line which are described as following:

M (Modified)

The *modified* state marks the cache line as the most up-to-date version and forbids that other copies exist within other caches. Thus, the content of the cache line is no longer coherent with the main memory.

E (Exclusive)

The *exclusive* state is set if the cache line is present in this cache and when it is coherent with the main memory. No other copies are allowed to exist in other caches. Thus, it is exclusive to this cache.

S (Shared)

The *shared* state is almost the same as the *exclusive* attribute. However, copies of the cache line can also exist in other caches.

I (Invalid)

The *invalid* state marks the cache line as invalid.

Figure 2.12 illustrates the MESI protocol: In addition to the previous example the bus allows an additional Bus Read Exclusive (`BusRdX`) transaction where a cache controller can ask for an exclusive copy that it wants to modify. It is important to know that the data is supplied by the memory system and this may not be only the main memory but could also be another cache. However, after the data has been transmitted, all other caches need to invalidate their corresponding cache line. This transaction is generated by a write of a processor to a cache line that is either not in the cache at all or in the *modified* state.

If a memory block is read the first time by a processor and a valid copy of this memory block exists in any other cache, the state of the corresponding cache line will be the `S` (shared) state. Otherwise, if there no valid copy exists, it will have the `E` (exclusive) state.

If a memory block is in the `E` (exclusive) state and it is written by the processor, its state can directly change to the `M` (modified) state since it is the only copy in the system. If a shared memory block is written, the state changes to the `M` as well. However, a `BusRdX` transaction is created and, thus, all copies of the cache line are invalidated in the other caches.

If a cache would obtain a copy of an `E` (exclusive) block of another cache, the state of the exclusive block would change to the `S` (shared) state since two unmodified copies of the same data exist in multiple caches. To make this possible, the bus provides an additional signal, the shared signal, so that controllers can determine on a `BusRd` if any other cache holds the data already [17]. In the address phase of the transaction, all caches will check if they have a copy of the requested data and if so, assert the signal S. Accordingly `BusRd(`S`)` means that if a bus read transaction has occurred, the signal S has been asserted and $\neg S$ means that the signal has not been asserted. If a block is in the `M` (modified) state and a `BusRd` has been

observed; the block needs to be flushed on the bus so that other controllers are notified of the changes that have been made.

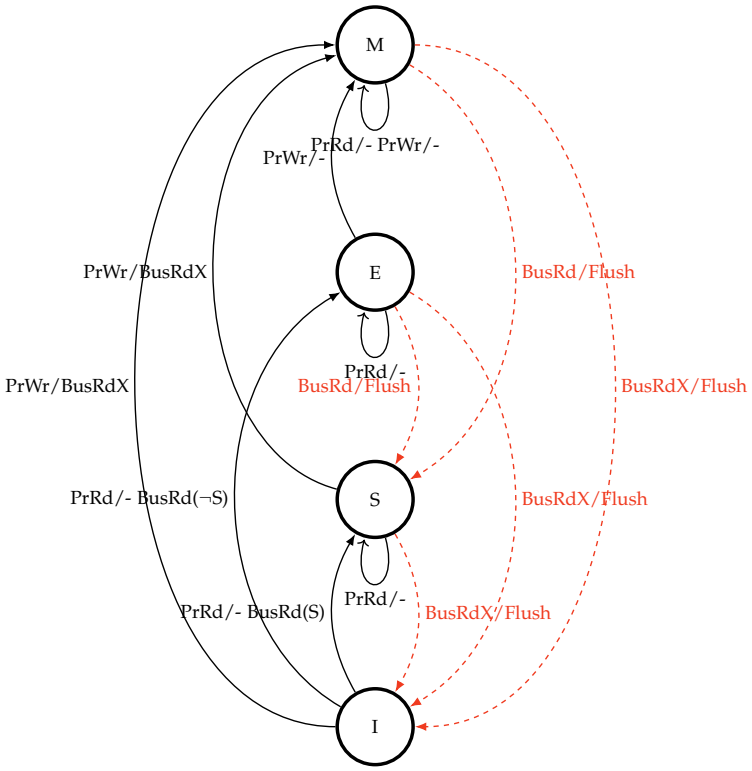


Figure 2.12: Modified Exclusive Shared Invalid (MESI) protocol state diagram. The transitions are labelled as action observed/action performed: `PrRd` (Read request from the processor), `PrWr` (Write request from the processor), `BusRd` (Read request from the bus, S denotes shared), `BusRdX` (Exclusive read request from the bus with intent to modify).

2.2.2.3 MOESI protocol

Many coherence protocols use the five state Modified Owned Exclusive Shared Invalid (MOESI) model that has been introduced by Sweazey and Smith in 1986 [78]. The states that refer to attributes that are assigned by the Snoop Control Unit (SCU) to each line in the cache [6] are based on the MESI protocol but have been extended by one additional state:

O (Owned)

The *owned* attribute is used if the cache line has been modified and might exist in other caches as well. While only one core can hold the owned state and, thus, the most recent and therefore correct copy of

the data. Other cores can only hold the data in the *shared* state. The cache having the *owned* state has the exclusive rights to change the data.

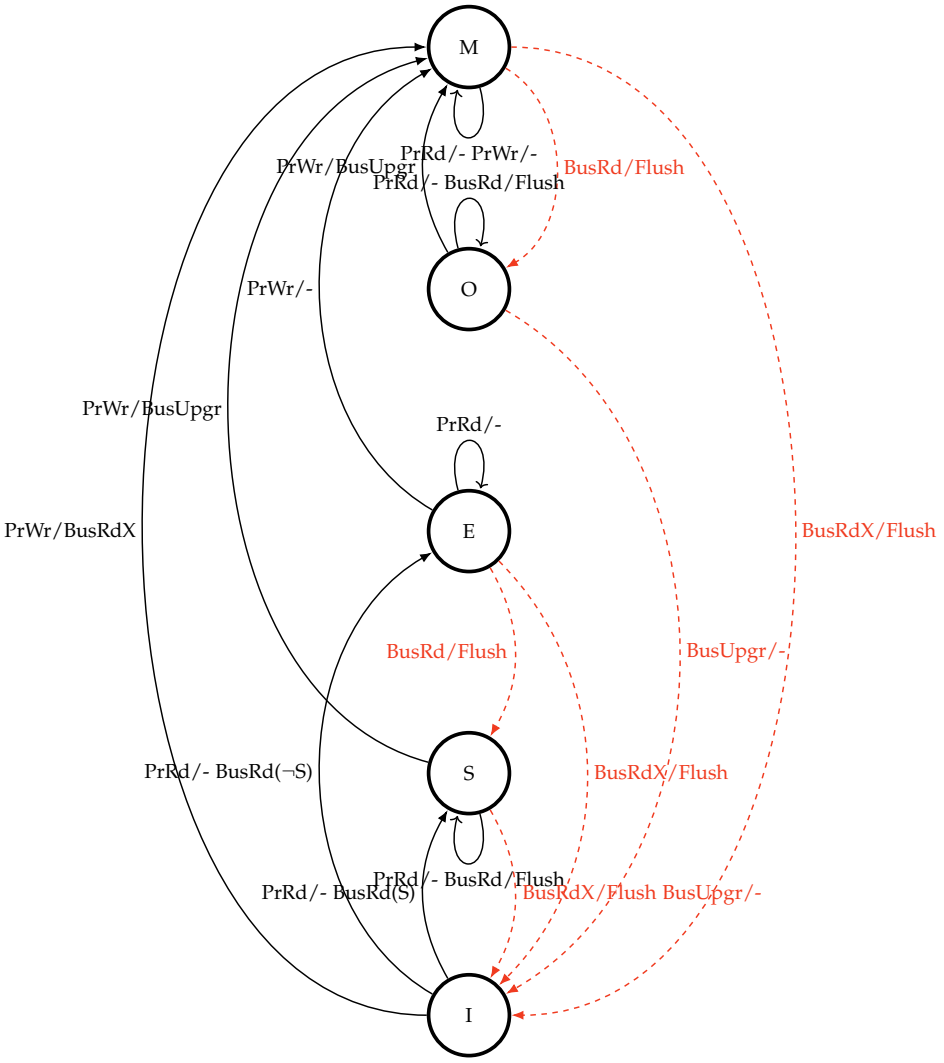


Figure 2.13: Modified Owned Exclusive Shared Invalid (MOESI) protocol

The `owned` state allows a cache controller to provide a modified cache line directly to another processor. Thus, the cache line has not be written back to the main memory first. This property plays an important role in our attacks since the communication latency between CPUs is lower than to the main memory.

Figure 2.13 illustrates the MOESI protocol that extends the MESI protocol with the *owned* state: If a `BusRdX` occurs, the memory controller can not distinguish if the requesting cache already has a copy of the memory block and just needs to update its state or if it has no copy of the block and needs to request it from the main memory. In the first case, the memory controller would unnecessarily fetch a block from the main memory that is in fact not needed. To solve this issue, a new bus request called bus upgrade (`BusUpgr`) is introduced. If a cache has already a valid copy of the block, it will issue a `BusUpgr` instead of a `BusRdX` and the memory controller will not act on this request [79]. If a cache block is in the `O` (owned) state, the `BusUpgr` invalidates all other cached copies if the processor issues a write. If the processor issues a read, no action must be taken. However, if a read occurs on the bus, the cache controller must notify the other caches of the changes.

2.3 Shared memory

Shared memory is a memory that can be accessed by multiple programs to either provide communication between them or to avoid redundant copies of the memory. While it provides an efficient way to pass data between programs that might run on a single processor or across multiple processors, we focus in this section on conserving memory space using shared libraries.

A shared library or shared object is a file that is intended to be shared by other shared objects or executable files. This gives the opportunity to provide the same functionality to multiple programs, e.g., parsing of a file or rendering a website, while there only exists a single copy of the code. In addition, this provides the functionality to programs for loading executable code into memory during run time and providing a plugin like systems instead of being linked into a single executable. Shared libraries reduce the memory footprint and enhance the speed by lowering cache contention as shared code is kept only once in the main memory, the CPU caches as well as in address translation units.

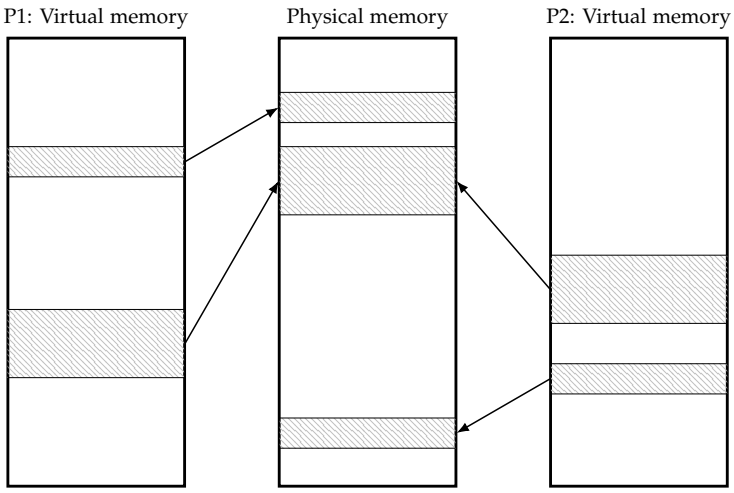


Figure 2.14: The operating systems maps the same physical memory the virtual address space of both, process 1 (P1) and process 2 (P2).

The operating system implements shared memory by mapping the same physical memory into the virtual address space of each application that uses it as illustrated in Figure 2.14. When self-modifying code or just-in-time (JIT) compiled code is used, this advantage can not be used in general. Usually, Android applications are written in Java and, thus, would incur just-in-time compilation. Typically, this code is not shared. However, there have been several approaches to improve the performance. First with optimized virtual machine binaries and more recently with native code binaries. The Android Runtime Engine (ART) [2] compiles those native binaries from Java byte code, allowing them to be shared.

Since it is irrelevant for this memory sharing mechanism how a certain file has been opened or accessed, a binary can be mapped as a shared read-only memory using system calls like `mmap`. Therefore, it is possible for another application to map code or data of a shared library or any accessible binary into its address space even if the application is statically linked.

Content-based page deduplication is another form of shared memory where the operating system scans the entire system memory for identical physical pages and merges them to a single physical page which is then marked as copy-on-write. This mechanism can enhance the system performance

where system memory is limited such as servers with many virtual machines or smartphones.

On both, Linux and Android, processes can retrieve information on virtual and physical address mappings using operating-system services like `/proc/<pid>/maps` or `/proc/<pid>/pagemap`. While Linux has gradually restricted unprivileged access to these resources, these patches have not yet been applied to Android stock kernels. Thus, a process can retrieve a list of all loaded shared-object files and the program binary of any process and even perform virtual to physical address translation without any privileges.

2.4 Cache Attacks

Cache side channel attacks exploit information leakage caused by micro-architectural time differences when data is loaded from the cache rather than the main memory. Since data that resists in the cache can be accessed much faster than data that has to be loaded from the main memory, one can whether decide if a specific portion of data resides in the cache and thus, implying that it has been accessed recently. The resulting information leakage has a potential risk, especially for cryptographic algorithms, which leads to the compromise of secret keys.

In 1996 Kocher [45] described that by carefully measuring the amount of time required to perform private key operations, one might be able to break various cryptographic systems. He has been the first to mention that the CPU cache can be a possible information leak in ciphers that do not use data identically in every encryption. Four years later Kelsey et al. [42] discussed the notion of side-channel cryptanalysis and concluded that attacks based on cache hit ratio in ciphers using large S-boxes like Blowfish [71] and CAST [16] are possible.

Based on these theoretical observations more practical attacks against Data Encryption Standard (DES) have been proposed by Page [64] and also by Tsunoo et al. [82]. With the standardization of Advanced Encryption Standard (AES) [18], cache attacks against this block cipher have been investigated as well. Bernstein [11] presented the well-known cache-timing attack

against AES that has further been analyzed by Neve [57] and Neve et al. [59].

In this section, we will discuss and describe various attack types that have been shown on the x86 architecture in their chronological order: *Evict+Time*, *Prime+Probe*, *Flush+Reload*, *Evict+Reload* and *Flush+Flush*.

2.4.1 *Evict+Time*

In 2005 Percival [66] and Osvik et al. [63] proposed more fine-grained exploitations of memory accesses to the CPU cache. In particular, Osvik et al. formalized two concepts, namely *Evict+Time* and *Prime+Probe* that we will discuss in this and the following section. The basic idea is to determine which specific cache sets have been accessed by a victim program.

Algorithm 1 *Evict+Time*

1: Measure execution time of victim program.
2: Evict a specific cache set.
3: Measure execution time of victim program again.

The basic approach, outlined in Algorithm 1, is to determine which cache set is used during the victim's computations. At first, the execution time of the victim program is measured. In the second step, a specific cache set is evicted before the program is measured a second time in the third step. By means of the timing difference between the two measurements, one can deduce how much the specific cache set is used while the victim's program is running.

Osvik et al. [63] and Tromer et al. [81] demonstrated with *Evict+Time* a powerful type of attack against AES on OpenSSL implementations that requires neither knowledge of the plaintext nor the ciphertext.

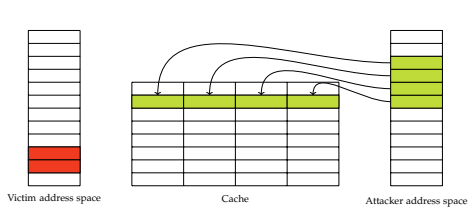
2.4.2 *Prime+Probe*

The second technique that Osvik et al. [63] described is called *Prime+Probe*. Similar as *Evict+Time* described in Section 2.4.1 it allows an adversary to determine which cache set is used during the victim's computations.

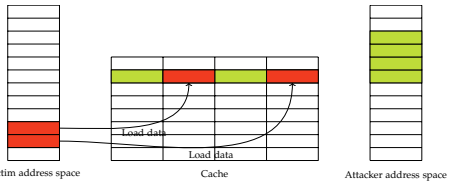
Algorithm 2 *Prime+Probe*

1: Occupy specific cache sets.
2: Schedule victim program.
3: Determine which cache sets are still occupied.

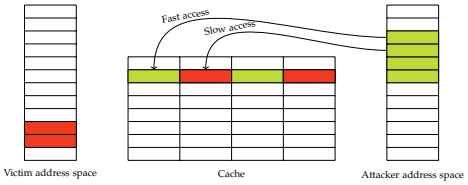
Algorithm 2 outlines the idea behind this approach: In the first step, one or multiple specific cache sets are occupied with memory owned by the adversary. In the second step the victim's program is scheduled, in the third step, the adversary determines which and in what amount he is still occupying the cache set. This can be done by measuring the execution time for accessing the addresses the adversary used to fill the cache set in the first step.



(a) `Prime`: Occupy the cache set



(b) Schedule the victim's program that accesses data in the same cache set.



(c) `Probe`: Determine cache set occupation.

Figure 2.15: Illustration of the Prime+Probe *attack by means of a 4-way (columns) cache with 8 sets (rows). In step a, the adversary occupies the target cache set represented by the green color. In step b, the victim's program is scheduled which also loads memory in the cache represented by the red color. In step c, the adversary determines how much he still occupies the cache set by loading the previously used addresses into the cache again.*

Figure 2.15 illustrates the attack in a more detailed manner: The grid represents a 8-way (columns) cache with 6 sets (rows), and the attacker performs the attack on the fourth cache set. In step a, the attacker accesses congruent addresses that all map to the target cache set, thus, it is filled with memory that the attacker owns. This is represented by the green color.

In step b, the victim's program is scheduled and consequently filling the cache sets with addresses that it used, pictured by the blue color. If the program uses addresses that map to the same cache set as the attacker's addresses, it will, therefore, evict addresses that the attacker used to occupy the cache set.

If now, in step c, the attacker reaccesses the used addresses, he can determine how much he still occupies the cache set by measuring the access times of those addresses. If an address in the cache set has been replaced by the victim's program, it has to be loaded from the main memory. Thus the access time is much higher than if the address has still been in the cache. Due to the pseudo-random replacement policy, it might happen that the access to one congruent address evicts a previously accessed address from the attacker and thus it is possible that during the probing phase false positives occur. Figure 2.16 shows measurements of the execution time it took to reaccess the addresses of two separate runs. It is easily distinguishable if the victim has accessed a congruent address.

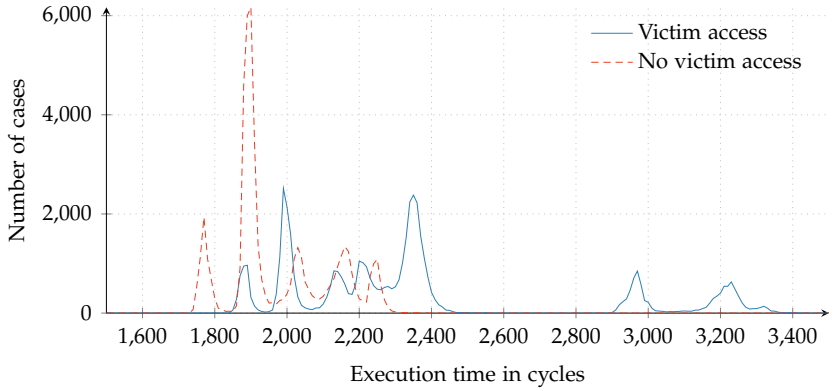
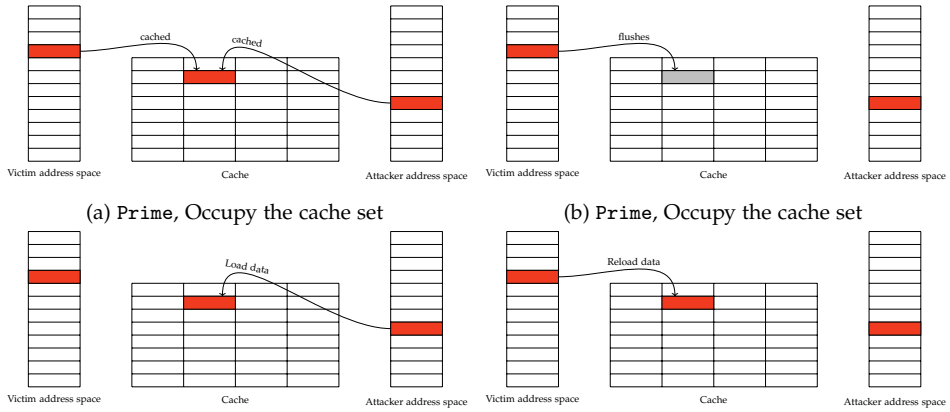


Figure 2.16: Histogram of Prime+Probe *timings depending on whether the victim accesses congruent memory on the ARM Cortex-A53.*

In 2006 Osvik et al. [63] and Tromer et al. [81] demonstrated a key recovery attack on OpenSSL AES and Linux's `dm-crypt` with this attack technique. In 2015 Liu et al. [52] used *Prime+Probe* to mount a cross-core, cross-VM covert channel and to attack ElGamal decryption in GnuPG. In the same year, Irazoqui et al. [35] attack the OpenSSL AES implementation in the cloud environment.

2.4.3 *Flush+Reload*

In 2011 Gullasch et al. [29] proposed the basic idea of using `clflush` to run an efficient access-driven cache attack on AES. They utilize the `clflush` instruction to evict the monitored memory locations from the cache and then check if those locations have been loaded back into the cache after the victim's program executed a small number of instructions. Yarom and Falkner [87] extended this idea after they observed that the `clflush` instruction evicts the cache line from all cache levels including the shared Last-Level-Cache (LLC). They present the *Flush+Reload* attack which allows the spy and the victim process to run on different cores of the CPU.



(a) `Prime`, Occupy the cache set

(b) `Prime`, Occupy the cache set

(c) `Prime`, Occupy the cache set

(d) `Prime`, Occupy the cache set

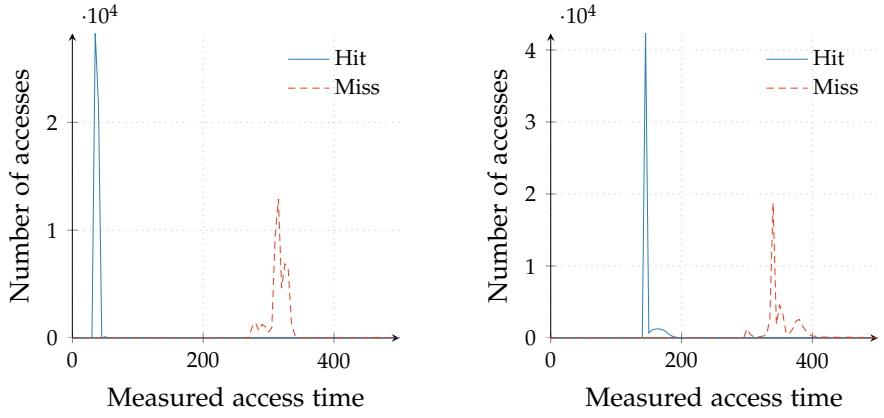
Figure 2.17: Illustration of the Flush+Reload *attack by means of an 4-way (columns) cache with 8 sets (rows). In step a, the target address is cached for both, the adversary and the attacker. In step b, the attacker flushes the address out of the cache and in c, the victim's program is scheduled which loads the address back in the cache. In step d, the adversary determines how much it takes to access the target address to decide if the victim has accessed the address in the meantime.*

Algorithm 3 *Flush+Reload*

1: Map binary (e.g., shared object) into address space.
2: Flush a cache line (code or data) from the cache.
3: Schedule the victim's program.
4: Check if the corresponding cache line from step 2 has been loaded by the victim's program.

Algorithm 3 and Figure 2.17 summarize the *Flush+Reload* attack principle. In the first step, the attacker needs to map a binary, which could be a

shared object or an executable, into his address space using a system call like `mmap`. In the second step, the attacker flushes, utilizing the `clflush` instruction, a cache line from the cache including the shared LLC. In the third step the victim's program is scheduled and in the third step the cache line that has been flushed in the second step is accessed by the attacker. The attacker measures the execution time that it takes to access the address and decides upon that if the access has been loaded from the cache or the main memory. Figure 2.18 illustrates the timing difference between a cache hit and a cache miss. If the attacker measures a cache hit, the victim's program has accessed the cache line in the mean time.



(a) *Flush+Reload* on Samsung Galaxy S6.

(b) *Flush+Reload* on Intel x86.

Figure 2.18: Flush+Reload*: Histogram of access times of addresses that are in the cache and addresses that have been evicted with a flush instruction.*

In contrast to *Prime+Probe*, *Flush+Reload* enables more fine-grained attacks that have already been demonstrated against cryptographic algorithms. Irazoqui et al. [37] demonstrate full-key recovery of AES implementations in VMWare virtual machines. Yarom et al. [86] recover OpenSSL Elliptic Curve Digital Signature Algorithm (ECDSA) nonces and thus the secret key. In 2015 Gülmezoglu et al. [30] exploit a shared resource optimization technique called memory deduplication to mount a powerful known-ciphertext cross-VM attack on an OpenSSL implementation of AES. In the same year, Irazoqui [38] use a *Flush+Reload* side channel to detect and distinguish different cryptographic libraries running on different virtual machines on the same computer. In addition, Irazoqui et al. [39] show a new

significantly more accurate covert channel to perform Lucky Thirteen [1] on co-located virtual machines in the cloud.

In 2015 Gruss et al. [28] used the *Flush+Reload* technique to automatically exploit cache-based side-channel information. Furthermore, they showed that besides attacking cryptographic implementations, the attack could be used to infer keystroke information as well. Thus, by exploiting the cache side-channel, they were able to build a keylogger.

2.4.4 *Evict+Reload*

In 2015 Gruss et al. [28] introduced the *Evict+Reload* technique that uses eviction in *Flush+Reload* instead of the flush instruction. While this attack has no practical use on x86 CPUs since the `clflush` instruction is unprivileged, it can be used on ARM CPUs that do not provide a unprivileged flush instruction.

To evict an address from the cache one has to fill the cache set with as many congruent addresses such that the replacement policy decides to replace the target address. Depending on the replacement policy in place, the number of addresses required and the access pattern, the way those addresses are accessed can vary. If too few addresses are used or if they are not accessed often enough, the eviction rate, the probability in which an address will be evicted, can be too low. However, this matter is exhaustively studied in Section 3.2.

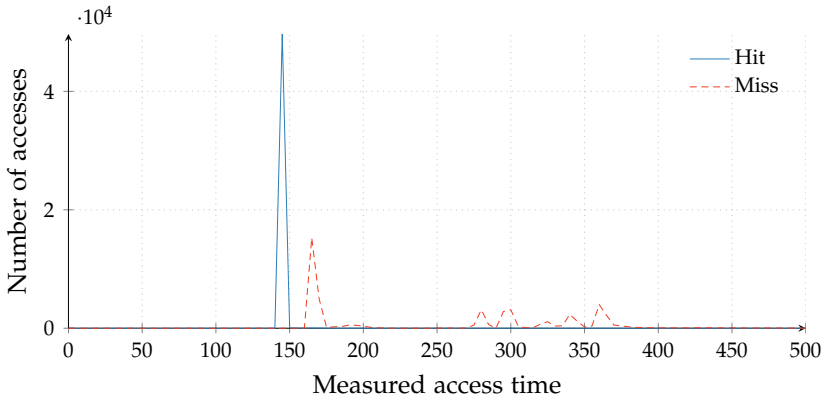


Figure 2.19: Evict+Reload*: Histogram of access times of addresses that are in the cache and addresses that have been evicted with an eviction strategy.*

Figure 2.19 shows a histogram of memory accesses to addresses that have been kept in the cache (Hit) and addresses that have been evicted by the eviction strategy. The small peak of the Miss-measurements next in the measured area where only cache hits occur points to an ineffective eviction strategy. Thus, the used eviction strategy does not guarantee to evict the target address all the time.

Gruss et al. [26] showed that with an effective and fast eviction strategy it is possible to trigger bit flips in adjacent rows of DRAM modules by repeatedly accessing a row of memory from JavaScript. Thus, the attack shown by Seaborn [72] does not rely on the privileged x86 `clflush` instruction anymore.

2.4.5 *Flush+Flush*

Since the *Flush+Reload* and the *Prime+Probe* attack cause numerous cache references and cache misses that can be monitored using hardware performance counters and thus can subsequently be detected. Based on those observations Gruss et al. [27] presented *Flush+Flush* that only depends on the execution time of the flush instruction that varies depending on whether the data is cached or not.

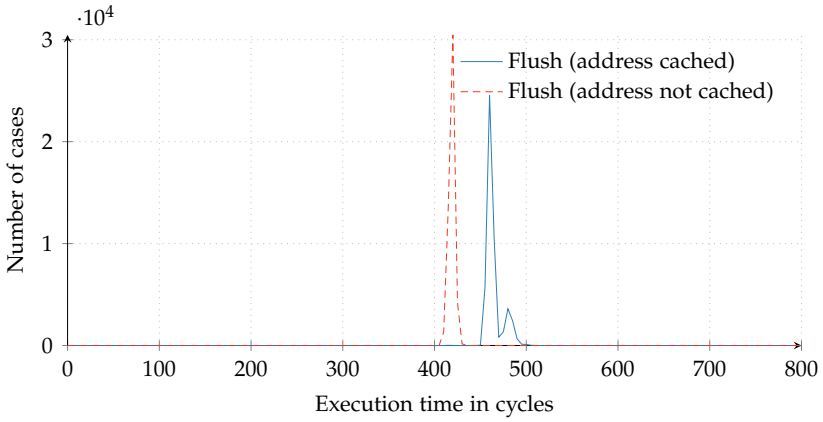


Figure 2.20: Histogram of the execution time of the flush operation on cached and not cached addresses measured on the Samsung Galaxy S6.

The attack is basically the same as *Flush+Reload*. A binary or a shared object file is mapped into the address space of the attacker. An address is

flushed from the cache, and the victim's program is scheduled. However, instead of the reloading step where the monitored address is accessed, it is flushed again causing no cache misses compared to *Flush+Reload* or *Prime+Probe*. Figure 2.20 shows that it is easily distinguishable if the address has been cached or not.

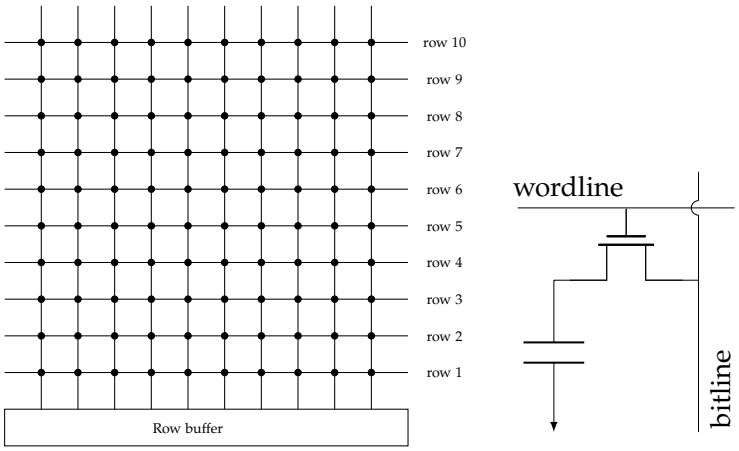
Gruss et al. [27] used this attack technique to implement a powerful covert channel that achieved a transmission rate almost seven times faster than any previously published covert channels.

2.5 DRAM

DRAM chips are manufactured in different configurations, varying in their capacity as well as in their bus width. An individual DRAM chip has only a small capacity and therefore multiple chips are coupled together to form a so-called DRAM rank. One DRAM module then consists out of one or multiple ranks.

A single DRAM chip consists out of a two-dimensional array of DRAM cells as illustrated in Figure 2.21a. Each single DRAM cell is made out of a capacitor and an access transistor is shown in Figure 2.21b. The charged state of the capacitor, either fully charged or fully discharged, represents a binary data value. Each cell in the grid is connected to the neighbored cell with a wire forming a horizontal wordline and a vertical bitline. If a wordline of a row is raised to a high voltage, all access transistors in that row are activated, thus, connecting all capacitors to their respective bitline. By doing that the charge representing the data of the row is transferred to the so-called row buffer.

To access data in a memory bank the desired row needs to be opened at first by raising the corresponding wordline. By that the row is connected to all bitlines and the data is transferred into the row buffer. Then the data in the row buffer is accessed and modified by reading or writing in the row buffer. If data from a different row but in the same bank needs to be accessed, the current row needs to be closed by lowering the corresponding wordline and the row buffer is cleared.



(a) Rows of cells (b) Single cell

Figure 2.21: Simplified view of a DRAM chip and a single DRAM cell

However, the charge stored in the capacitor of a DRAM cell is not persistent because its charge can disperse. This means that after some time data is lost. To avoid this, the cell's charge must be refreshed by fully charging or discharging it. DRAM specifications require that all cells in a rank are refreshed within a certain amount of time, the so-called refresh rate.

Modern mobile phones are equipped with low power versions of DDR3 or DDR4 memory, namely Low Power Double Data Rate (LPDDR) DRAM. While older generations of the standard SDRAM required a supply voltage of 2.5V, it has been reduced on LPDDR to 1.8V or lower. In addition, the power consumption has been reduced by temperature-compensated refresh, where a lower refresh rate is required at low temperatures. However, with each generation, the transfer speed and internal fetch size have been increased.

2.5.1 Rowhammer Bug

In 2014 Kim et al. [43] demonstrated that accessing specific memory locations in DRAM in a high repetition rate random bit flips can occur. Since cells are getting smaller and smaller and are built closer together, disturbance errors that are caused by activating the row very often can change the charge of the capacitor in a cell and therefore the data.

It has been shown that this behaviour can be exploited for privilege escalation [72] by simple accessing two addresses that are in the same bank but on different rows. In order to evict the address from the cache and to access it again from main memory, the `clflush` instruction has been used on x86 architecture. However, such bit flips can also be triggered using eviction in JavaScript code that is loaded from a website [26] and therefore removing the `clflush` instruction can not prevent attacks.

2.5.2 Reverse Engineering DRAM addressing

In order to hammer a certain memory location we need to find two addresses in the same bank but in different rows as the target address. Certain bits of the address are used to select the rank, channel and bank of the memory location. However, these mapping functions are not documented.

In 2015 Pessl et al. [67] presented a way to fully automate the reverse engineering of said functions by exploiting the fact that row conflicts lead to higher memory access times. Their approach is to find addresses that map to the same bank but a different row by repeatedly measuring the access time of two random addresses. For some address pairs, the access time is higher than for others meaning that they belong to different rows but to the same bank. Subsequently, these addresses are then grouped into sets having the same channel, DIMM, rank and bank. The identified addresses are then used to reconstruct the addressing functions by generating all linear functions and applying them to all addresses of a randomly selected set. Since the search space is small enough, a brute-force search is sufficient.

Since no `clflush` function is available on the ARM platform and eviction yielded unsatisfying results, we have developed a kernel module that has been used to flush addresses using the privileged flush instruction from userspace. Using this module, we have successfully reverse-engineered the DRAM mapping functions for the Samsung Galaxy S6, the OnePlus One, the LG Nexus 4, the LG Nexus 5 and the Samsung Nexus 10 as shown in Table 2.2.

Table 2.2: DRAM mapping functions showing the bits responsible for the selected bank, rank and channel.

Device	Banks	Ranks	Channel
Samsung Galaxy S6	14, 15, 16	$8 \oplus 13$	$7 \oplus 12$
OnePlus One	13, 14, 15		10
LG Nexus 4	13, 14, 15		10
LG Nexus 5	13, 14, 15		10
Samsung Nexus 10	13, 14, 15		7

Chapter 3

Attack primitives

In this chapter, we discuss the challenge to obtain a high-resolution timing source or a dedicated performance counter to distinguish between cache hits and cache misses. Then we will take up on the challenge of finding performant eviction strategies that can be used to evict individual addresses from the cache when an unprivileged flush instruction is not available.

3.1 Timing Measurements

In order to obtain high resolution timestamps for cache attacks on x86, the unprivileged `rdtsc` instruction can be used. However, a similar instruction is not provided by the ARMv7-A or ARMv8-A architecture. Rather, a Performance Monitoring Unit (PMU) is used to monitor the activity of the CPU. Section 3.1.1 shows how one of those performance counters can be used to distinguish between cache hits and misses. However, the performance counters of the PMU can not be accessed from userspace by default and, thus, root privileges are required to make use of them.

For this reason, we search for alternative timing sources that do not require any permissions or privileges. With each option we identified, we lower the requirements of the running system. In Section 3.1.2 we make use of an unprivileged system call, in Section 3.1.3 we use a POSIX function and

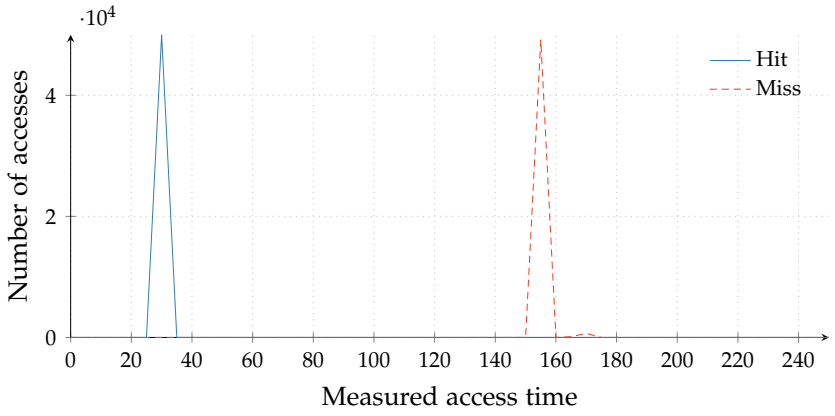


Figure 3.1: Histogram of cache hits and misses on the Alcatel One Touch Pop 2 using the Performance Monitor Cycle Count Register (`PMCCNTR`).

in Section 3.1.4 we implement a dedicated thread timer. Additionally, we show that all of those methods can be used to perform cache attacks.

3.1.1 Performance Interface

While x86 CPUs have the unprivileged `rdtsc` [33] function to obtain a sub-nanosecond timestamp, a similar function does not exist on neither the ARMv7-A nor the ARMv8-A architecture. However, a Performance Monitoring Unit (PMU) allows gathering statistics on the operation of the processor and memory system.

It offers one performance monitor register denoted as Performance Monitor Cycle Count Register (`PMCCNTR`) [4, 5] which counts processor cycles. Figure 3.1 illustrates the measured access time with the `PMCCNTR` register of an address residing in the cache or the main memory. Cache hits and cache misses are easily distinguishable.

While those measurements are fast and precise, the access to those performance counters is restricted to the kernelspace by default. However, the User Enable Register (`PMUSERENR`), that is writable only in privileged modes, can be configured to allow userspace access to the `PMCCNTR` [4]. Therefore a kernel module and hence root privileges are required making this timing source hardly accessible in serious attacks.

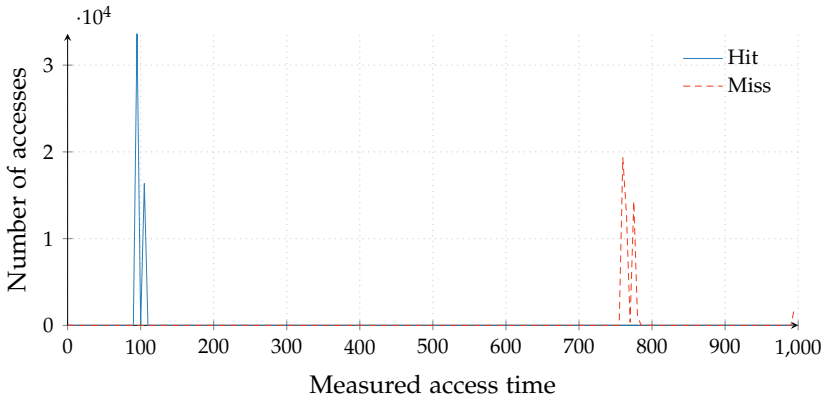


Figure 3.2: Histogram of cache hits and misses on the Alcatel One Touch Pop 2 using the perf interface.

The ARMv8-A architecture provides similar registers, yet their corresponding register names have `_EL0` as a suffix e.g., `PMCCNTR_EL0` or `PMUSERENR_EL0` [5].

3.1.2 Unprivileged system call

In Linux 2.6.31 [20] `perf` has been introduced to the Linux kernel. It provides a powerful interface to instrument CPU performance counters and tracepoints, independently of the used hardware. The system call `perf_event_open` is used to access such information from userspace, e.g., the `PERF_COUNT_HW_CPU_CYCLES` returns an accurate cycle count just as the privileged instructions described in Section 3.1.1. However, due to the fact that this approach relies on a system call to acquire the cycle counter value, a latency overhead can be observed. This can also be seen in Figure 3.2 where a hit is measured at around 100 cycles and a miss at around 780 cycles.

Listing 1 shows an implementation of accessing the `perf` interface in C. A struct describing the performance counter we want to access is defined and passed as an argument to the *perf_event_open* function. The current value of the performance counter can then be obtained by reading from the file descriptor. Using `ioctl` commands, it is possible to enable and disable the counter as well as to reset it.

Listing 1 Accessing the cycle count from userspace

```
static struct perf_event_attr attr;
attr.type = PERF_TYPE_HARDWARE;
attr.config = PERF_COUNT_HW_CPU_CYCLES;
attr.size = sizeof(attr);
attr.exclude_kernel = 1;
attr.exclude_hv = 1;

int fd = syscall(__NR_perf_event_open, &attr, 0, -1, -1, 0);
assert(fd >= 0);

long long result = 0;
if (read(_fddev, &result, sizeof(result)) < (ssize_t) sizeof(result)) {
  return 0;
}

printf("Cycle count: %llu\n", result);

close(fd);
```

3.1.3 POSIX function

The `perf` interface described in Section 3.1.2 is enabled by default on most devices. However, we have observed that this is not the case for our Samsung Galaxy S6 and support for `perf` events would require a customized kernel. Thus, we need a different timing source.

The POSIX function `clock_gettime()` retrieves the time of a clock that is passed as a parameter. Depending on the used clock, it allows obtaining timing information with a resolution in the range of microseconds to nanoseconds. In Figure 3.3 we see a histogram using the `CLOCK_MONOTONIC` clock as the source for the timer. We observe that although small timing differences are not distinguishable anymore, cache hits and misses can still be clearly distinguished.

3.1.4 Dedicated thread timer

In the worst case that neither access to the performance counters nor the `clock_gettime()` POSIX function is granted, an attacker can implement a thread running on a different core that increments a variable in a loop. Our experiments show that this approach works reliable on smartphones

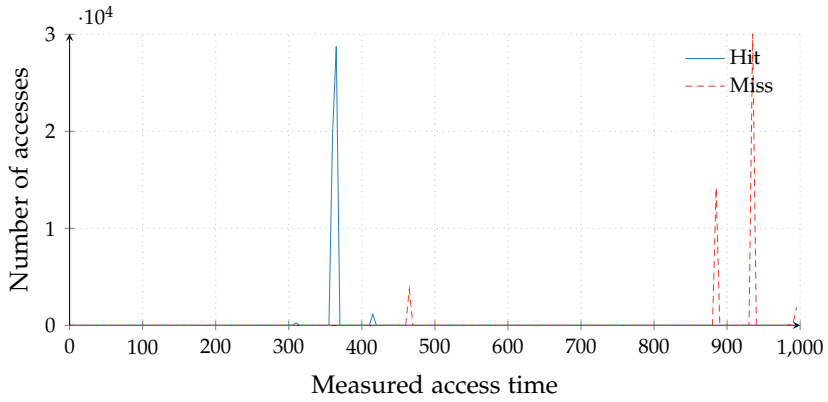
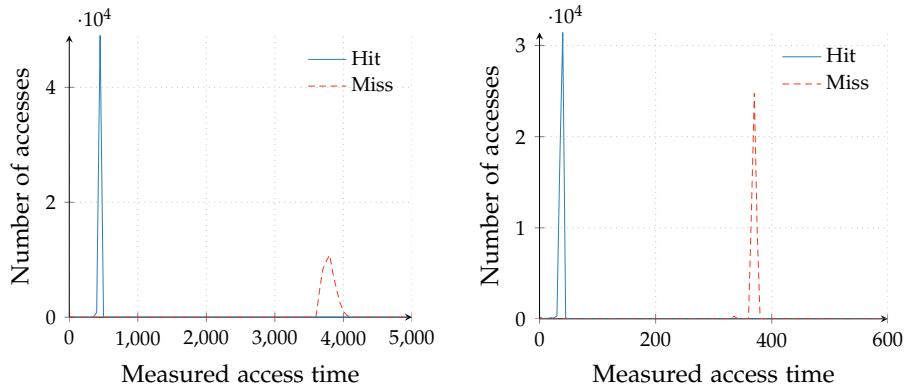


Figure 3.3: Histogram of cache hits and misses on the Alcatel One Touch Pop 2 using the monotonic counter.

as well as on x86 CPUs in Figure 3.4. The resolution of this threaded timing information is by far high enough to differentiate between cache hits and cache misses.



(a) Alcatel One Touch Pop 2

(b) x86

Figure 3.4: Histogram of cache hits and misses on the Alcatel One Touch Pop 2 and on x86 using the dedicated thread timer.

3.2 Cache Eviction

In order to evict an address from the cache to the main memory, one can make use of a flush instruction like the unprivileged `clflush` instruction on x86. While ARM does provide cache maintenance functions they are not enabled in the unprivileged mode and can only be unlocked for user

access on recent CPUs. The second method to evict data is to access congruent addresses, *i.e.*, that map to the same cache set, so that the cache replacement policy decides to evict the target address from the cache. The strategy that describes how to access congruent addresses is called *eviction strategy*.

Since only on one of our test devices the flush instruction is available (Samsung Galaxy S6), we need to rely on eviction strategies for all other devices. Thus, we need to defeat the cache replacement policy (see Section 2.1.4) that directly influences the number and patterns of accesses needed for the eviction strategy. If an Least-Recently Used (LRU)-replacement policy is used, accessing as many congruent locations as the number of ways of the last-level cache evicts the targeted address with high probability. For adaptive cache replacement policies, e.g., pseudo-random replacement policy, different eviction strategies need to be crafted. While the L1 cache in Cortex-A53 and Cortex-A57 CPUs has a small number of ways and a LRU replacement policy is in use, we also need to evict the cache lines from the L2 cache, which employs a pseudo-random policy, to perform a full cache eviction.

In order to evict data from the cache several approaches have been presented in the past. While some of them can only be applied to Least-Recently Used (LRU) replacement policies [52, 54, 62] and, thus, are not suited for ARM CPUs, others introduce too much overhead [32]. However, Spreitzer and Plos [76] proposed an eviction strategy for ARMv7-A devices that tries to overcome the pseudo-random replacement policy by accessing more addresses than there are ways per cache set. In addition, Gruss et al. [26] demonstrated an automated way to find fast eviction strategies on Intel x86.

In this chapter we will describe how to find eviction strategies in a fully automated way based on the concept of Gruss et al. [26] which allows us to replace of a flush instruction in any cache attack in order to enable such attacks on devices where a flush instruction is not available.

In the first section of this chapter, we will describe the model that is used to describe eviction strategies, before we will discuss eviction strategies we found and their evaluation in the second section.

3.2.1 Model

In this section, we will describe the parameters that define an eviction strategy based on the algorithm presented by Gruss et al. [26]. The success of each cache eviction strategy is rated by testing whether the targeted address is still in the cache or has successfully been evicted to the main memory. This is tested in many experiments and finally results in an average success rate.

In their paper Gruss et al. [26] made three observations that all have an influence on the average success rate of an eviction strategy, thus, forming adjustable parameters for the strategy model:

1. Besides cache maintenance functions, only cache hits and cache misses to addresses in the same cache set have a non-negligible influence on the cache. This was verified by taking an eviction algorithm and randomly adding addresses that do not map to the same cache set. One can observe that those random non-congruent addresses do not influence the average success rate and that the effectiveness of the strategy depends on the *eviction set size*.

2. In addition, they have noted that addresses are indistinguishable to the cache. Therefore access patterns are presented as sequences of address labels a_i, e.g., $a_1a_2a_3$, where each address label corresponds to a different address and thus defines which address to access at a given time frame. A pattern $a_1a_2a_3$ is equivalent to any pattern $a_ia_ja_k$ where $i \neq j \neq k$ and if the pattern is run in a loop, the *number of accesses to different addresses per loop* has an influence on the effectiveness of the eviction strategy.

3. The cache replacement policy can prefer to evict recently added cache lines over older ones. Thus, it may be necessary to repeatedly access the same address to keep it in the cache. As an example, they state that changing the eviction sequence from $a_1a_2 \ldots a_{17}$ to $a_1a_1a_2a_2 \ldots a_{17}a_{17}$ reduces the execution time by more than 33% and increases the eviction rate on Intel Haswell CPUs. However, they have observed that after a certain number of accesses, additional

accesses do not increase the eviction rate; instead, it can get even worse.

Algorithm 4 Eviction loop for pattern testing.

```
Require: Eviction set with N congruent addresses
for i = 0; i < N-D; i++ do
    for j = 0; j < A; j++ do
        for k = 0; k < D; k++ do
            Access (i+k)th address of eviction set
        end for
    end for
end for
```

Based on these observations Gruss et al. [26] have defined three parameters that are adjustable and depend on the cache and the cache replacement policy that is in place to achieve a good eviction strategy:

- N: Number of different addresses in the eviction set.
- D: Number of different addresses accessed in each loop round.
- A: Number of accesses to each address in each loop round.

Algorithm 4 describes how those parameters are used to generate the described access patterns.

3.2.2 Strategies

In order to find an eviction strategy that is optimal in the sense of having a relatively high average success rate as well as having a decent low execution time, one needs to have knowledge about the system in terms of cache organization and cache replacement policies. While finding optimal eviction strategies for caches with a simple cache replacement policy (e.g., round-robin) is rather easy, a pseudo-random replacement policy makes this task rather hard to solve without any detailed knowledge about the implementation on the device.

In order to find optimal eviction strategies for our test devices, we have tested and evaluated thousands of different eviction strategies. We have written a tool that automatically generates different eviction strategies,

Number of addresses	Number of accesses per address	Number of different addresses in each loop round	Cycles	Eviction rate
11	2	2	1 578	100.00%
12	1	3	2 094	100.00%
13	1	5	2 213	100.00%
16	1	1	3 026	100.00%
24	1	1	4 371	100.00%
13	1	2	2 372	99.58%
11	1	3	1 608	80.94%
11	4	1	1 948	58.93%
10	2	2	1 275	51.12%
Privileged flush instruction			549	100.00%

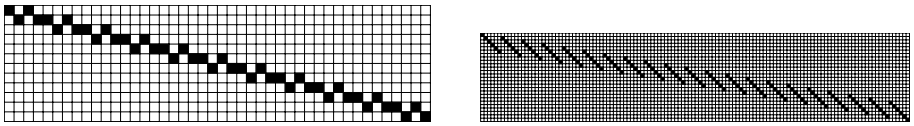
Table 3.1: Different eviction strategies on the Krait 400.

executes and evaluates them. The tool is platform independent and allows communication with the test device over `adb` resulting in a convenient way of finding optimal eviction strategies for any device.

For the Krait 400, we have generated and evaluated 1 863 different strategies that are summarized in Table 3.1. We have identified ($N = 11$, $A = 2$, $D = 2$), illustrated in Figure 3.5a, as our optimal eviction strategy for devices using this platform as it achieves an eviction rate of 100% and takes 1 578 CPU cycles. In contrast, the very similar ($N = 11$, $A = 4$, $D = 1$) strategy takes slightly longer and only achieves an eviction rate of $58,93\%$. If we use a so-called LRU eviction where every address in the eviction set is only accessed one, we need at least 16 addresses in our eviction set. While fewer memory addresses are accessed, it requires more CPU cycles (3 026 cycles). If we compare this result with the eviction strategy used by Spreitzer et al. [76] taking 4 371 cycles, it clearly shows the benefit of an optimized eviction strategy as our strategy is almost three times as fast.

In addition, we have also measured the privileged flush instruction that gives the best result in terms of execution time. It only requires 549 cycles and, thus, it is almost three times faster than our best eviction strategy.

Although the Alcatel One Touch Pop 2 uses an ARM Cortex-A53 CPU that supports the ARM-v8 instruction set, the ROM running on the phone has been build against the ARM-v7 instruction set. Therefore we can not use the unprivileged flush instructions and must rely on eviction as well.



(a) Krait 400 (b) Cortex A53

Figure 3.5: Access pattern on the Krait 400 (OnePlus One) and the Cortex A53 (Alcatel One Touch Pop 2)

Number of addresses	Number of accesses per address	Number of different addresses in each loop round	Cycles	Eviction rate
23	2	5	6 209	100.00%
23	4	6	16 912	100.00%
22	1	6	5 101	99.99%
21	1	6	4 275	99.93%
20	4	6	13 265	99.44%
800	1	1	142 876	99.10%
200	1	1	33 110	96.04%
100	1	1	15 493	89.77%
48	1	1	6 517	70.78%
Privileged flush instruction			767	100.00%

Table 3.2: Different eviction strategies on the Cortex-A53.

We have evaluated 2 295 different strategies and summarized them in Table 3.2. We observed that the fastest eviction strategy with an eviction rate of 100%, ($N = 23$,$A = 2$,$D = 5$), has an average execution time of 4 275 cycles. However, the ($N = 21$,$A = 1$,$D = 6$) strategy has only an eviction rate of 99.93% but is faster by almost 2 000 cycles. It is one of the best eviction rates we have found for the Cortex-A53, but it is still 5 times slower than the privileged flush instruction. We have illustrated its access pattern in Figure 3.5b.

In addition, we observed that LRU eviction would require 800 different addresses to achieve an eviction rate of only 99.10%. Since data needs to be evicted from the L1 cache to be allocated to the last-level cache, it is a better choice to access data that is already in the last-level cache instead of accessing different additional addresses. Thus, LRU eviction as used in previous work [76] is not suitable for attacks on the last-level cache on the Cortex-A53.

3.3 Defeating the Cache-Organization

Yarom and Falkner [87] considered the *Flush+Reload* attack the ARM architecture as not applicable to smartphones due to their differences in the cache organization. In comparison to Intel CPUs, ARM CPUs are very heterogeneous when it comes to caches, whether or not a CPU has a second-level cache can be decided by the manufacturer. Nevertheless, the last-level cache on ARM CPUs is usually shared amongst all cores. However, it can have different inclusiveness properties for instructions and data. Since, only modern CPUs like the Cortex A53 and Cortex-A57 have an inclusive last-level cache, attacks on the last-level cache have been considered impracticable before.

The cache coherence protocols discussed in detail in Section 2.2, guarantees that shared memory is kept in a coherent state in all cores and all CPUs. These protocols also enable us to distinguished between cache hits and cache misses, because accesses to remote caches are faster than accesses to the main memory [9, 10]. If a cache is non-coherent, a cross core-attack is not possible anymore. However, the attacker could run its spy processor simultaneously on all cores of all CPUs and, thus, launch a parallel same-core attack. Simultaneously to our work, Irazoqui et al. [36] exploited cache coherence protocols on AMD x86 CPUs.

In order to perform a cross-core attack, we need to evict the target address from the other cache to the main memory. Depending on the cache architecture we can fill the cache directly or indirectly: On the Alcatel One Touch Pop 2, the last-level cache is instruction-inclusive, thus, we can evict instructions from local caches of other cores by filling the last-level cache as illustrated in Figure 3.6. In the first step, an instruction used by the first core is allocated in its instruction cache as well as in the last-level cache. In the second step, the second core fills its data cache and thereby evicts cache lines into the last level cache. As the second core is filling up the last-level cache using only data accesses in step 3, it will at some point evict the instruction of the first core from the last-level cache. Since the cache is instruction-inclusive the instruction will also be evicted from the instruction cache of the first core.

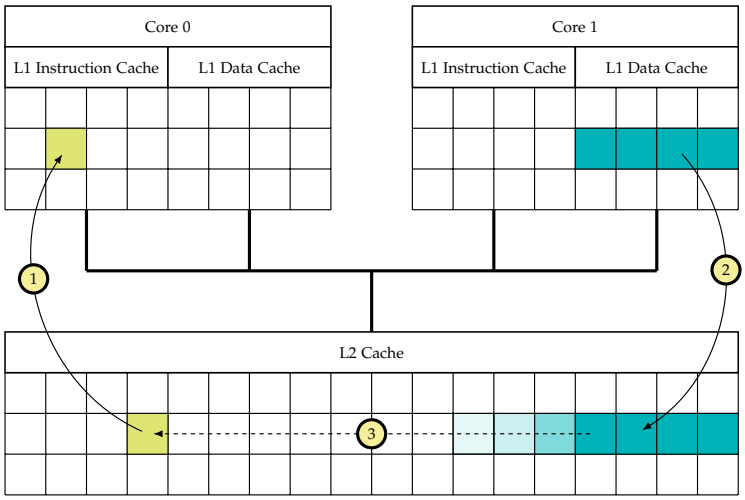


Figure 3.6: Cross-core instruction cache eviction through data accesses on a instruction-inclusive, data-non-inclusive cache

In the case of a cache that is non-inclusive on both, data and instruction side, we still can be successful in evicting data from another cache. Figure 3.7 illustrates this approach: The L2 cache is kept filled with data in order to make the other core evict the address to the main memory and not into the L2. As the first step illustrates more and more addresses that are used for the eviction are stored in the L1 cache or the last-level cache. On ARM the L1 caches typically have a very low associativity, thus, the probability that an address is evicted from L1 due system activity is very high. As the last-level cache is filled with the addresses used by the eviction strategy, it is very likely that the instruction in the first core will be evicted to the main memory (step 2).

For the *Evict+Reload* and the *Flush+Reload* attack, we need to test if the victim process running on a different core has accessed the target address. As an example, we want to check if the instruction address as in Figure 3.7 has been loaded into any cache. If the attack process is running on the second core, one would think that if it accesses the target address, it would need to be loaded from the main memory as the address is neither in the local cache of this core nor in the last-level cache. However, the snoopy cache coherence protocols [9, 10, 48] described in Section 2.2.1 allow addresses to be fetched from remote cores because such accesses are faster than loading the address from the main memory.

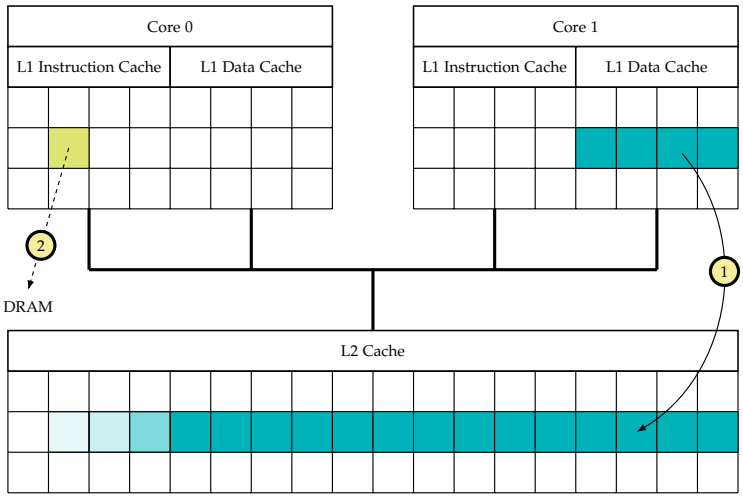


Figure 3.7: Cross-core instruction cache eviction through data accesses on an entirely non-inclusive cache

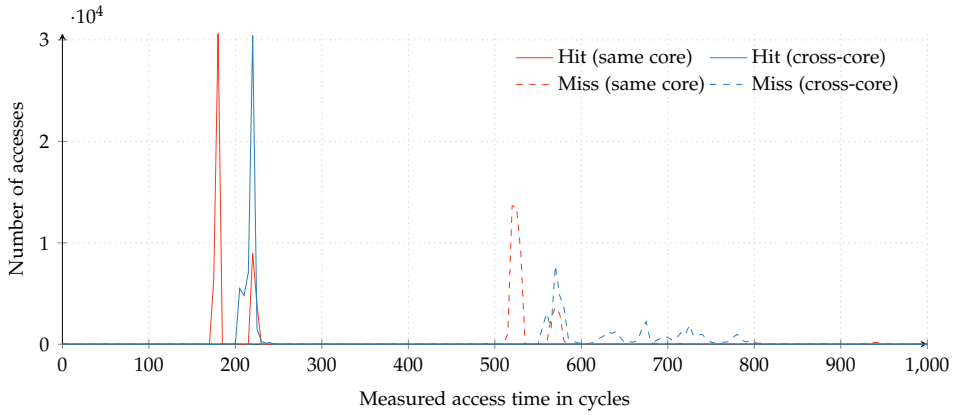


Figure 3.8: Histograms of cache hits and cache misses measured same-core and cross-core on the OnePlus One

Figure 3.8 shows the cache hit and cache miss histogram across cores on the OnePlus One. Loading an address from a remote core only takes around 40 cycles longer than if the address has been in the local cache. As loading an address from the main memory takes more than 500 cycles, cache hits can be distinguished from cache misses easily.

Figure 3.9 illustrates a similar histogram but cross-CPU on the Samsung Galaxy S6. While loading an address from a remote core of another CPU takes around 200 cycles, a cache miss still takes at least 50 cycles longer.

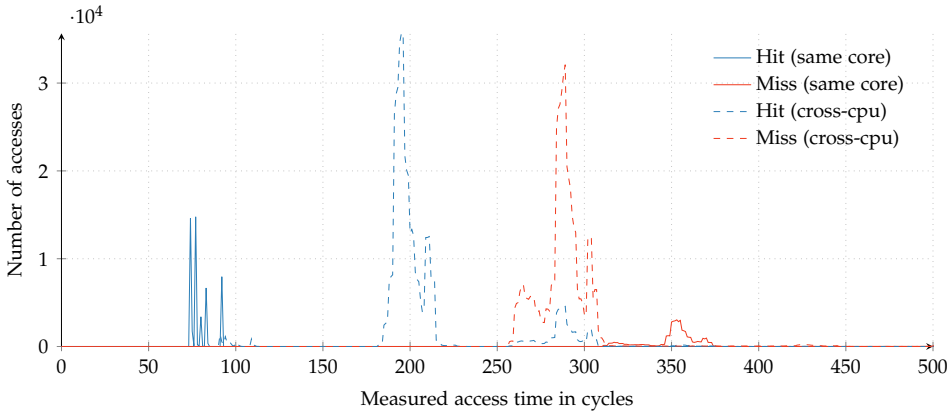


Figure 3.9: Histogram of cache hits and cache misses measured cross-CPU on the Samsung Galaxy S6.

Thus, we can also identify if a process on a remote-CPU has accessed an address.

Chapter 4

Attack Case Studies

In this chapter we want to present our attack techniques and how we can take advantage of our findings from Section 3.2 and Section 3.1. While the attacks techniques discussed in Section 2.4 have been proposed and investigated for Intel processors, the same attacks were considered not applicable to modern smartphones due to differences in the instruction set, the cache organization [87], and in the multi-core and multi-CPU architecture. Thus only same-core cache attacks have been demonstrated so far [11, 75, 77, 84, 85]. These attacks also require the access to the privileged cycle count register to acquire accurate timings and thus, root access.

Most of the presented attacks in this chapter can be established in a real world scenario where a user installs one or multiple malicious applications on its device. In contrast to previously demonstrated attacks, the presented attacks do not rely on any permissions and can be executed as an unprivileged userspace application. Thus, they would not look suspicious during the installation processes. The attacks can be performed on stock Android ROMs as they do not exploit any vulnerability in the Android system.

In Section 3.3 we overcome the challenge of non-inclusive last-level caches on ARM. A property that cross-core attacks relied on. We also show how to overcome the difficulty that modern smartphones can have multiple CPUs that do not share a cache. In Section 4.1 we demonstrate covert channels that outperform state-of-the-art covert channels on Android by several

orders of magnitude, and in Section 4.2 we present attacks to monitor tap and swipe events as well as keystrokes. In addition, we show in Section 4.3 attacks to cryptographic primitives in Java and monitor cache activity in the ARM TrustZone. Finally, we show in Section 4.4 how the rowhammer bug can be triggered on mobile devices.

4.1 High Performance Covert-Channels

In this section, we describe a high-performance cross-core and cross-CPU cache covert channel on modern smartphones that uses either *Flush+Reload*, *Evict+Reload* or *Flush+Flush*. A covert channel enables two unprivileged applications on a system to communicate with each other without using any data transfer mechanisms provided by the operating system. This communication evades sandboxing and the permission system. Particularly on Android, this is a problem, as this covert channel can be used to exfiltrate private data from the device that the Android permission system would normally restrict. An attacker could use one application that has access to the personal contacts of the owner of the device to send data via the covert channel to another application that has Internet access (cf. collusion attacks [53]). In such a scenario an adversary can steal personal information.

4.1.1 Design

The basic idea of our covert channel is that both, the sender and the receiver, agree on a set of addresses of a shared library. These are used to transmit information by either loading the address into the cache or evicting them from the cache, e.g., if the address is in the cache it represents a 1. Since our goal is to establish a covert channel on devices where no flush instruction is available, every picked address must be mapped to a different cache set. Otherwise, the problem might occur that addresses that both agreed on might be evicted by accident and, thus, transmitting wrong data.

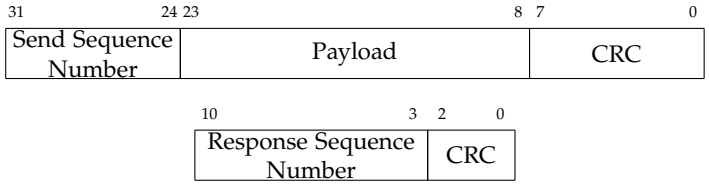


Figure 4.1: Format of send data frames (above) and response data frames (below).

We implement the covert channel using a simple protocol. To transmit an arbitrary amount of data, it is split-up and transmitted in multiple packets that are illustrated in Figure 4.1. A single packet consists out of an n-bit payload, an s-bit sequence number and a c-bit checksum that is computed over the payload and the sequence number. The sequence number is used to send consecutively numbered packets and the checksum is used to check the integrity of the package. If a received packet is valid, the receiver sends back a response packet containing the s-bit sequence number and an additional x-bit checksum calculated over the response sequence number. By sending back the sequence number of the last successfully received packet, the sender can resend packets.

For each bit in the send packet and the response packet, the sender and receiver must agree on one address in the shared library. This is possible as both processes use the same code to choose addresses of the shared library. In order to transmit a bit value of 1, the sender loads the corresponding address of the shared library into the cache by accessing it. If the receiver measures the access time to this address, it will interpret the measured cache hit as a 1. On the other hand, if the sender wants to transmit a 0, the sender will not load the corresponding address into the cache and, thus, the receiver will measure a cache miss. For the *Flush+Flush* version of the covert channel, the receiving unit decides based on the execution time of the flush instruction if the address has been loaded into the cache or not.

In addition to the bits that are required for both packets, two extra bits are used. To start a transmission, the sender sets a *sending* bit as well as all the bits for the sending packet and waits until the receiver has set a *acknowledge* bit. The *acknowledge* bit indicates that the receiver has successfully received the sending packet. The sender will then measure the response sequence number and check if the response checksum is valid. If this is the case, it

will continue sending the next packet. If the response has been invalid, it will continue sending the current packet.

However, depending on how noisy the system is and how the parameters have been set, it might happen that the sender measures an acknowledgement of the receiver even if the receiver has never received the packet successfully. In that case, the sender would continue sending the next packet while the receiver still tries to receive a previous one and the whole transmission would get stuck. To overcome this issue, the sender will read the response of the receiver multiple times to detect this situation and will roll-back to the previous packet. Algorithm 5 and Algorithm 6 provide a formal description of the sender and the receiver, respectively.

Algorithm 5 Sending data

Require: Mapped shared library m.
Require: Data to send d.
1: $sn \leftarrow$ Initial sequence number.
2: **for** $fn \leftarrow 1$ **to** Number of frames **do**
3: $p \leftarrow$ Current package (sn, d_x, $CS(fn, d_x)$)
4: $received \leftarrow false$;
5: **do**
6: Access sending bit address
7: Access or evict packet bit addresses
8: $ack \leftarrow$ Access Acknowledge bit address
9: **if** $ack \equiv true$ **then**
10: Measure response data addresses
11: $sn_m, cs_m \leftarrow$ Response sequence number, CRC
12: **if** $CS(sn, d_x) \equiv cs_m$ and $sn \equiv sn_m$ **then**
13: $received \leftarrow true$
14: **end if**
15: **end if**
16: **while** $received \equiv false$
17: **end for**

4.1.2 Results

We have implemented this covert channel using three different cache attack techniques and evaluated them in different scenarios on our test devices. We show that the covert channel can be realized using *Evict+Reload*, *Flush+Reload* and *Flush+Flush* and it can be executed cross-core as well as

Algorithm 6 Receiving data

Require: Mapped shared library m.

```
1: while true do
2:     received ← false
3:     do
4:         sn ← Initial sequence number
5:         sending ← false
6:         do
7:             sending ← Measure sending bit address
8:         while sending ≡ false
9:         Measure packet data addresses
10:        sn_m, d_m, cs_m ← Sequence number, data, CRC
11:        if CS(sn_m, d_m) ≡ cs_m then
12:            if sn ≡ sn_m then
13:                received ← true
14:                Report d_m
15:                sn ← sn + 1
16:            end if
17:            Access acknowledge bit address
18:            Access or evict response data bit addresses
19:        else
20:            Evict acknowledge bit address
21:        end if
22:    while received ≡ false
23: end while
```

Table 4.1: Comparison of covert channels on Android.

Work	Type	Bandwidth [bps]	Error rate
Ours (Samsung Galaxy S6)	*Flush+Reload*, cross-core	**1 140 650**	1.10%
Ours (Samsung Galaxy S6)	*Flush+Reload*, cross-CPU	**257 509**	1.83%
Ours (Samsung Galaxy S6)	*Flush+Flush*, cross-core	**178 292**	0.48%
Ours (Alcatel One Touch Pop 2)	*Evict+Reload*, cross-core	**13 618**	3.79%
Ours (OnePlus One)	*Evict+Reload*, cross-core	**12 537**	5.00%
Marforio et al. [53]	Type of Intents	4 300	–
Marforio et al. [53]	UNIX socket discovery	2 600	–
Schlegel et al. [70]	File locks	685	–
Schlegel et al. [70]	Volume settings	150	–
Schlegel et al. [70]	Vibration settings	87	–

cross-CPU. We transmitted several megabytes of randomly generated data and compared the received file to the original one to measure an error rate.

Table 4.1 summarizes the results and compares them to existing covert channels on Android. We achieve the highest transmission rate in a cross-core scenario using *Flush+Reload* on the Samsung Galaxy S6 with 1 140 650 bps and an error rate of 1.10%. In comparison to existing covert channels [53, 70] this is 265 times faster. Using the Cortex-A53 and the Cortex A-57 on the Samsung Galaxy S6 we can establish a *Flush+Reload* covert channel between both CPU's that achieves a transmission rate of 257 509 bps at an error rate of 1.83%. Using *Flush+Flush* we achieve 178 292 bps at an error rate of 0.48% across two cores.

On the OnePlus One and the Alcatel One Touch Pop 2 we have no unprivileged flush instruction and have to use eviction. On the OnePlus One we achieve a transmission rate of 12 537 bps at an error rate of $5,00\%$ and on the Alcatel One Touch Pop 2 we achieve a transmission rate of 13 618 bps with an error rate of 3.79%. This cross-core transmission is still 3 times faster than previously published covert channels.

4.2 Spying on User input

In this section, we will demonstrate access-driven cache side-channel attacks on mobile Android devices. We demonstrate in Section 4.2.1 cache template attacks as described by Gruss et al. [28] to accurately profile the

cache usage and automatically exploit it using the *Flush+Reload* attack. In Section 4.2.2 we will present what libraries we have used to be capable of spying on user input and in Section 4.2.3 we show how we attack ahead-of-time compiled Android Runtime Engine (ART) [2] executables.

4.2.1 Template attacks

In 2015 Gruss et al. [28] presented cache template attacks to exploit cache-based side-channel information based on the *Flush+Reload* attack automatically. Cache template attacks consist of a profiling and an exploitation phase:

Profiling phase

In the *profiling phase*, every address of the shared memory is tested if a cache hit occurs on this address when a specific event has been triggered. Every address is measured multiple times to reduce noise and the result is stored in a so-called template matrix. Such a matrix can be visualized by a heat map where the lightness of the color represents the likelihood that this address is accessed if the event is triggered.

Exploitation phase

In the *exploitation phase*, the computed matrix from the profiling phase is used to detect the occurrence of events by measuring cache hits on corresponding addresses.

In order to scan shared libraries or binaries for leaking addresses, an attacker must be able to map them as read-only shared memory into its own address space. As this memory is shared between processes, countermeasures of the operating systems like Address Space Layout Randomization (ASLR) do not apply to this memory.

Another strength of cache template attacks is that both phases can be performed on the attacked device online as long as the attacker is capable of triggering the events that he wants to spy on remotely on the device. In addition, target addresses can be searched offline on a different device beforehand and then be used in the exploitation phase on the target device.

4.2.2 Spying on Shared libraries

The Android operating system uses like Linux a large number of shared libraries. Those libraries are shared across different applications to fulfill certain purposes, e.g., camera or GPS access. We inspected the names of the libraries on our test devices in order to determine what their individual purpose is. For instance, the `libinput.so` might be responsible for handling user input.

As we wanted to find addresses that correspond to user input triggered on the touch screen of smartphones, we automatically scanned all these libraries as described in Section 4.2.1. Since manually executing user input events like pressing a button on the device or swiping over the display in order to detect corresponding addresses is impractical, we used different methods to simulate this events:

- `input` command-line tool
  We executed the `input` command line tool over the Android Debug Bridge (`adb`) that allows simulating user input events.
- `/dev/input/event*`
  An alternative approach is to sent event messages directly to the `/dev/input/event*` interfaces. While this is more device specific, it is much faster than the first approach, as it only requires a single `write()` call.

When we tried to profile differences between different letter keys on the keyboard, we began to use the latter approach as it is much faster and reduces the overall time of scanning the libraries significantly.

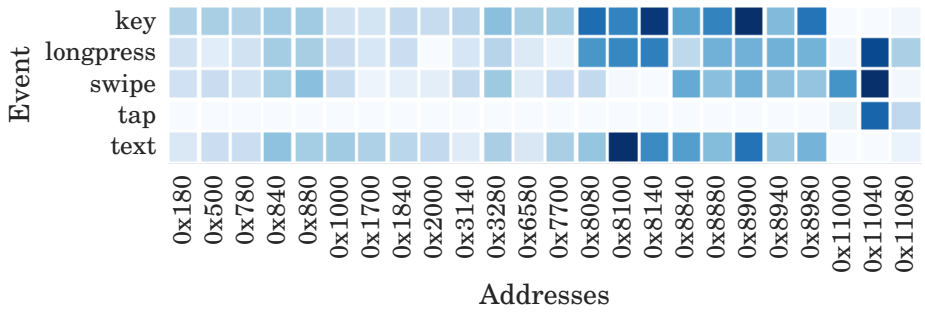


Figure 4.2: Cache template matrix for `libinput.so` on the Alcatel One Touch Pop 2.

We trigger different events while probing addresses within the `libinput.so` library at the same time. We simulate *key* events like the power button, *long press* and *swipe* events as well as *tap* and *text* events. For each address we flush the address into the main memory, trigger the event before we measure whether accessing the address results in a cache hit or cache miss. We repeat this process several times for each address and visualize parts of our results in Figure 4.2. We can clearly see that address `0x11040` can be used to detect *tap*, *swipe* as well as *long press* events.

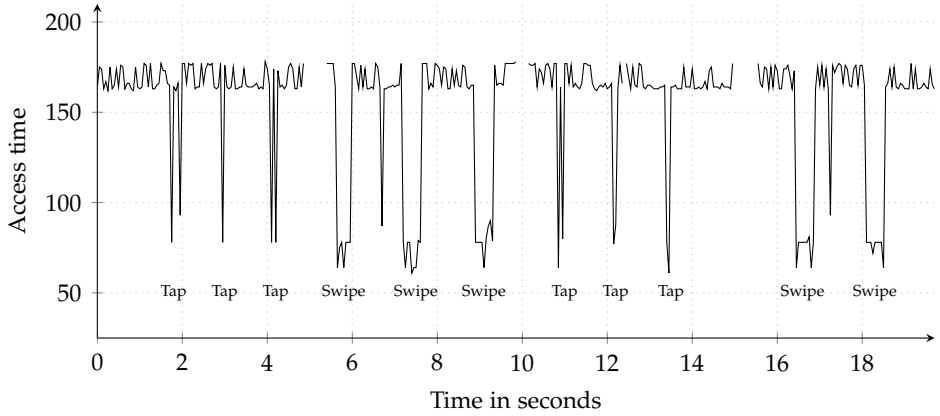


Figure 4.3: Distinguishing tap and swipe events by monitoring the access time of address `0x11040` of `libinput.so` on the Alcatel One Touch Pop 2.

Figure 4.3 shows a sequence of 3 tap and 3 swipe events followed by two additional swipes. A swipe action causes cache hits as long as the screen is touched and, thus, a single address can be used to distinguish taps and swipes. The gaps in the measurements are periods of time where our spy application was not scheduled on the CPU. Events that occur in said periods might be missed.

Cache template attacks provide a convenient way to automatically search for addresses that can be used to infer events. However, if an attacker has access to the shared libraries, he can also use reverse-engineering techniques to find addresses he wants to spy on. To reproduce the plot for the OnePlus One and the Samsung Galaxy S6, we manually identified a corresponding address in the `libinput.so` library. The resulting measurements are visualized in Figure 4.4 and Figure 4.5.

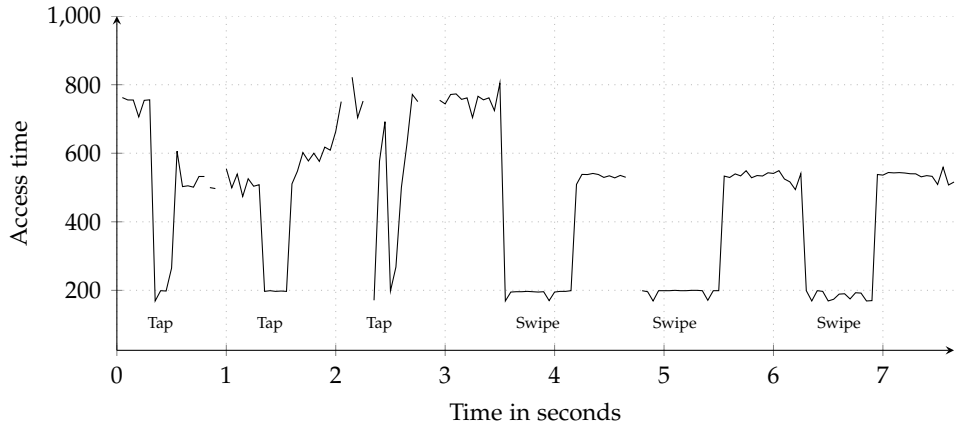


Figure 4.4: Distinguishing tap and swipe events by monitoring the access time of address `0xBFF4` of `libinput.so` on the OnePlus One.

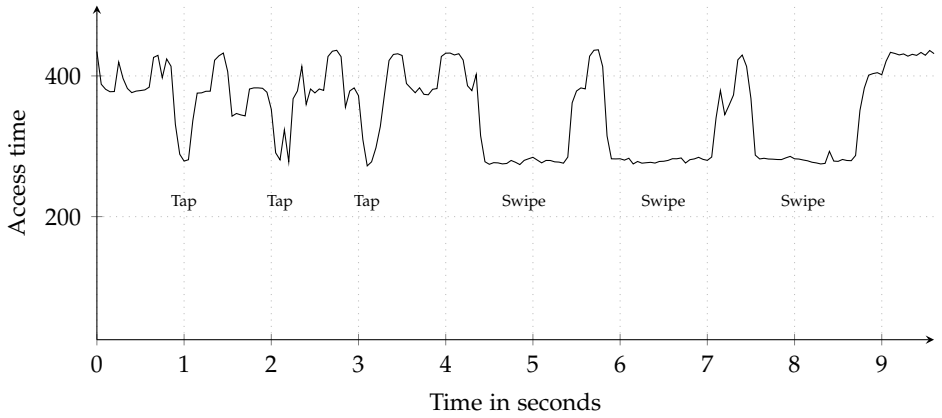


Figure 4.5: Distinguishing tap and swipe events by monitoring the access time of address `0xDC5C` of `libinput.so` on the Samsung Galaxy S6.

We have found various libraries listed in Table 4.2 that handle different hardware modules as well as software events on the device. For instance, the attack can be used to reveal when the user uses the GPS sensor, when the Bluetooth is active or when the camera is used.

4.2.3 Spying on ART

The more recent Android Runtime Engine (ART) creates optimized virtual machine binaries ahead of time. These binaries can be exploited by our attack as well.

Library	Responsible for
gps.msm8974.so	GPS
bluetooth.default.so	Bluetooth
camera.vendor.bacon.so	Camera
libnfc-nci.so	NFC
vibrator.default.so	Vibrator
libavcodec.so	Audio/Video
libstagefright.so	Audio/Video
libwebviewchromium.so	Websites
libpdfium.so	PDF library

Table 4.2: Libraries responsible for different hardware and software

We scanned the default Android Open Source Project (AOSP) keyboard on the Alcatel One Touch Pop 2 in order to find addresses that allow us to distinguish which letter a user has entered on the soft-keyboard. In order to simulate a key press, a touch event is triggered to the coordinates of the letter on the screen. Figure 4.6 shows the resulted cache template matrix. We observe that all addresses that we have identified for single letters of the alphabet show a quite similar result, and, thus we cannot differ between them.

However, we identified addresses that allow us to distinguish between a press to a letter and to one to the space bar or to the enter key. This information is especially useful as it allows to determine the length of every word entered. In addition, a combination of addresses can be used to detect whether the backspace button has been pressed and, thus, if a previous letter has been removed.

Figure 4.7 shows the timing measurements for an entered sentence. The blue line represents the access time on an address that is triggered when any key is pressed, while the measurements displayed as red dots of a different address can be used to distinguish between a key and the space bar. Thus, the length of each word in the entered sentence can be deduced. Zhang et al. [89] have shown that inter-keystroke timings can be exploited to infer the entered characters.

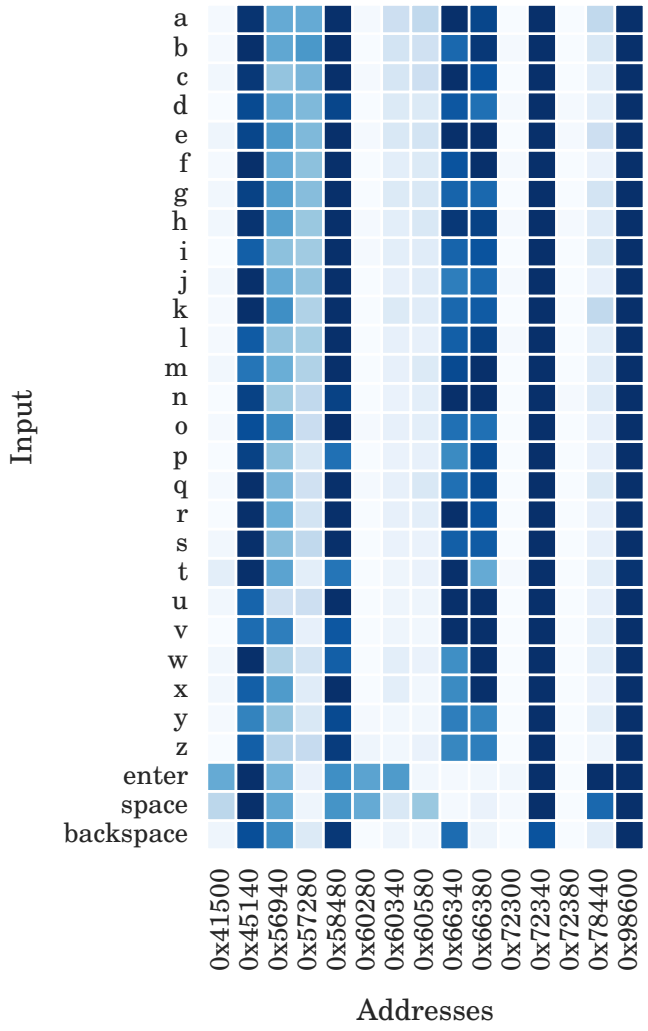


Figure 4.6: Cache template matrix for the AOSP keyboard

4.3 Attacks on Cryptographic Algorithms

In this section, we first attack cryptographic algorithms implemented in Java that are still in use on today's Android devices despite the well-known fact that they can be exploited by cache attacks. Furthermore, we will show that it is possible to monitor the cache activity of the ARM TrustZone from within the normal world.

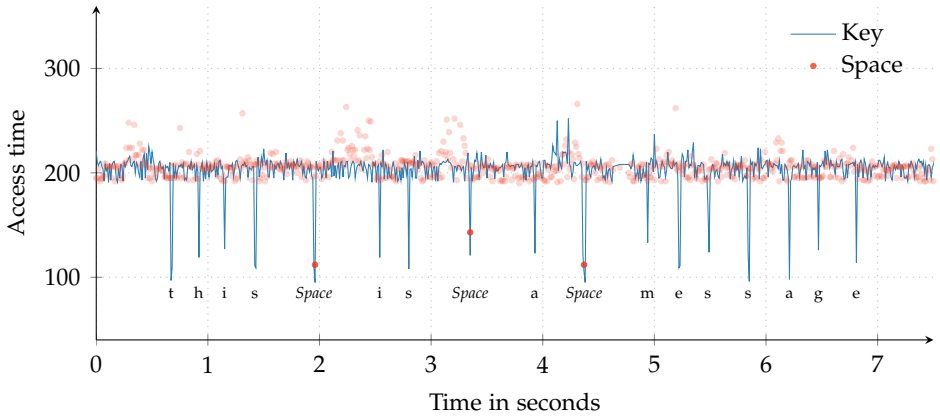


Figure 4.7: Evict+Reload *on 2 addresses in* `custpack@app@without libs@LatinIME.apk@classes.dex` *on the Alcatel One Touch Pop 2 while entering the sentence "this is a message".*

4.3.1 AES T-Tables

Advanced Encryption Standard (AES) is a specification for the encryption and decryption of data [18] and is used amongst other applications for full-disk encryption on Android [24]. An efficient version of AES is an optimized T-Table implementation.

The three different steps (*SubBytes*, *ShiftRows*, and *MixColumns*) of the AES round transformation can be combined into a single step of look-ups to four so-called T-tables $T_0, \ldots, T_3$ on 32-bit or 64-bit CPUs [18]. Since there is no *MixColumns* operation used in the last round, an additional T-table T_4 is required. However, to save space, it is possible to extract T_4 from the other tables by masking [18]. Each of the tables $T_0, \ldots, T_3$ contain 256 4-byte words and, thus, require 4KByte of memory.

Given a 16-byte plaintext $p = (p_0, \ldots, p_{15})$ and a 16-byte secret key $k = (k_0, \ldots, k_{15})$ that is expanded in ten round keys in the key schedule phase. A round key k^r for round r is a quadruplet of words of 4 bytes each: $k^r = (k_0^r, k_1^r, k_2^r, k_3^r)$. To encrypt the plain text p it is combined with a bitwise xor to the secret key k to result in the initial state x_0:

$$(x_0, \ldots, x_{15}) = (p_0 \oplus k_0, \ldots, p_{15} \oplus k_{15})$$

Then the 9 rounds consisting out of the *SubBytes*, *ShiftRows* and *MixColumns* are applied using the pre-computed T-tables $T_0, \ldots, T_3$. Thus, the intermediate state x is updated for $r = 0, \ldots, 8$ [81]:

$$
\begin{aligned}
(x_0^{r+1}, x_1^{r+1}, x_2^{r+1}, x_3^{r+1}) &\leftarrow T_0[x_0^r] \oplus T_1[x_5^r] \oplus T_2[x_{10}^r] \oplus T_3[x_{15}^r] \oplus k_0^{r+1} \\
(x_4^{r+1}, x_5^{r+1}, x_6^{r+1}, x_7^{r+1}) &\leftarrow T_0[x_4^r] \oplus T_1[x_9^r] \oplus T_2[x_{14}^r] \oplus T_3[x_3^r] \oplus k_1^{r+1} \\
(x_8^{r+1}, x_9^{r+1}, x_{10}^{r+1}, x_{11}^{r+1}) &\leftarrow T_0[x_8^r] \oplus T_1[x_{13}^r] \oplus T_2[x_2^r] \oplus T_3[x_7^r] \oplus k_2^{r+1} \\
(x_{12}^{r+1}, x_{13}^{r+1}, x_{14}^{r+1}, x_{15}^{r+1}) &\leftarrow T_0[x_{12}^r] \oplus T_1[x_1^r] \oplus T_2[x_6^r] \oplus T_3[x_{11}^r] \oplus k_3^{r+1}
\end{aligned}
$$

In the last round the *MixColumns* operation is omitted and the T_4 round table is used.

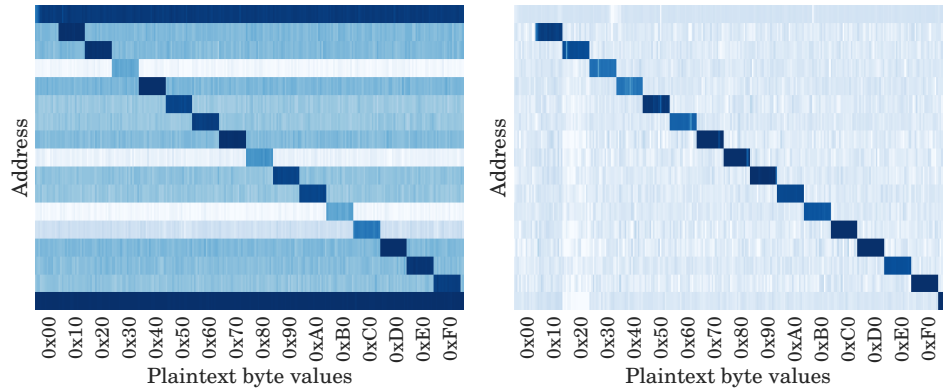
The first-round attack by Osvik et al. [63] exploit the fact that the initial state is defined by $x_i = p_i \oplus k_i$ for $i = 0, \ldots, 15$ and, thus, does only depend on the plaintext p and the key k. If an attacker knows the plaintext byte p_i and can monitor which entries of the T-table are accessed, he can directly deduce the key byte k_i. In addition to this attack, attacks that consider the first two rounds and attacks the consider the last round have been presented [58, 81].

Many cache attacks against implementations using T-tables have been demonstrated in the past [11, 30, 56, 58, 63, 76] and appropriate countermeasures and alternative implementations have been presented. For instance, bit-sliced implementations allow the algorithm to be executed in parallel as logical bit operations [41, 47, 68]. Intel and ARM even introduced dedicated instructions [5, 34] for AES.

Nonetheless, Bouncy Castle [3], a cryptographic library that is widely used in Android applications, deploys three different implementations of AES where the default option is a T-table implementation. The default crypto provider on Android, OpenSSL [61], as well as Google's BoringSSL [23] use bit-sliced implementations if ARM NEON instructions or dedicated AES instruction (ARMv8-A) are available. Otherwise they use a T-Table implementation as well. Although recent versions of OpenSSL all include bit-sliced implementations, the T-table implementation is still officially supported and still used on Android devices like the Alcatel One Touch Pop 2.

4.3.1.1 Attack on Bouncy Castle

For our first experiment, we assume that shared memory is available and demonstrate that both, a *Flush+Reload* attack as well as an *Evict+Reload* attack would be feasible. We trigger the encryption 256 times for all different values for a random plaintext where only the plaintext byte p_0 is fixed for every 256 values. Figure 4.8a shows a template matrix of the first T-table during the triggered encryptions revealing the upper 4 key bits of k_0 [63, 76]. Thus, the key space is reduced to 64 bits. However, the attack can be extended to a full key-recovery attack by targeting more than the first round [30, 37, 69, 81].



(a) *Evict+Reload* (b) *Flush+Reload*

Figure 4.8: Attack on Bouncy Castle's AES implementation using Evict+Reload *on the Alcatel One Touch Pop 2 and* Flush+Reload *on the Samsung Galaxy S6.*

Spreitzer et al. [76] showed that if the T-tables are placed on a different boundary such that we can obtain arbitrary disalignments of the T-tables, the key space can be reduced by 20 bits on average and further be brute forced. Depending on the disalignment also a full-key recovery is possible [76, 80]

4.3.1.2 Real-world cross-core attack on Bouncy Castle

In our first experiment we assumed that memory region containing the T-tables can be shared between the attacker and the victim. However, during the Java class initialization a copy of the T-tables is created. Thus, no

shared memory containing the T-tables exists. In such scenarios, it is possible to apply the *Prime+Probe* attack. Even though *Prime+Probe* is inherently noisier, it can be used for a real-world cross-core attack on Bouncy Castle.

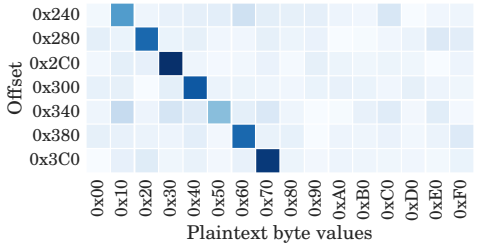


Figure 4.9: Excerpt of the attack on Bouncy Castle's AES using Prime+Probe. *The plaintext is fixed to* `0x00`. *Offset* `0x2C0` *is best to perform the attack.*

In order to run *Prime+Probe*, the attacker must identify the cache sets into which a T-table maps. Those cache sets can be detected by measuring the activity of each cache set when random encryptions have been triggered. After we have detected the corresponding cache sets, we successfully launched a *Prime+Probe* attack against the AES implementation of BouncyCastle. As *Prime+Probe* is noisier, we triggered the encryption 100 000 times. While it is only necessary to monitor a single address, we show our measurements in Figure 4.9 for each combination of plaintext byte and offset.

4.3.2 TrustZone Cache Activity

The ARM TrustZone [5] technology is a hardware based security technology built into ARM SoCs to provide a secure execution environment and roots of trust. The basic idea is to separate the system in a secure and a non-secure world that are hardware isolated from each other to prevent information from leaking from the trusted world to the other and to generally reduce the attack surface. The switch between these worlds is accomplished by a so-called secure monitor.

On Android the TrustZone technology is used among other things for a hardware-backed credential store, a secure element for payments, Digital Rights Management (DRM), verified boot as well as kernel integrity

measurements. Such services are implemented by so-called trustlets, applications that run in the secure world.

Since the secure monitor can only be called from the supervisor context the kernel must provide an interface for the userspace to communicate with the TrustZone. On the Alcatel One Touch Pop 2 a device driver called QSEECOM (Qualcomm Secure Execution Environment Communication) and a library `libQSEEComAPI.so` is used to communicate with the TrustZone. The key master trustlet on the Alcatel One Touch Pop 2 provides an interface to generate hardware backed RSA keys. In addition, it can be used for the signature creation and verification of data inside the TrustZone.

We used this service to mount a *Prime+Probe* attack on the signature creation process within the TrustZone. At first, we prime the cache set before we trigger a signature creation on random data with a fixed key. Then, we probe the same cache set to evaluate how many ways have been replaced by the TrustZone and store the measurement. We show the mean squared error over multiple measurements for every tested key in Figure 4.10. There is a difference in the set activity if a valid or an invalid key has been used. We have visualized the cache activity of sets 250-350 in Figure 4.11 to highlight cache sets than can be used to distinguish between valid and invalid keys.

This shows that the secure world of the TrustZone leaks information to the non-secure world that could be exploited by an attacker.

4.4 Rowhammer on ARM

In this section, we want to present the results of our experiments and show that even if we were not able to induce any bit flips using eviction, mobile devices are still prone to bit flips.

Simultaneously to our research, Van der Veen et al. [83] investigated the rowhammer bug on ARM-based devices as well. They successfully triggered the bug on multiple mobile devices and built a root exploit for Android.

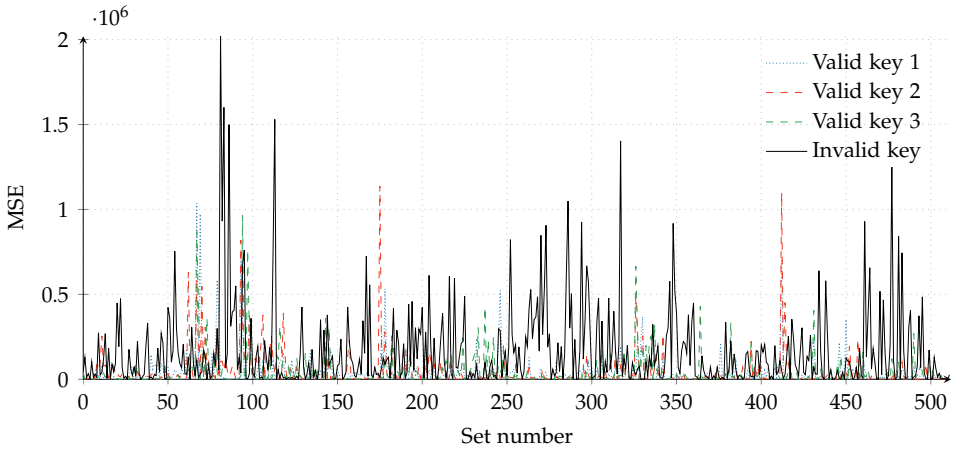


Figure 4.10: Mean squared error for Prime+Probe *cache profiles of 3 valid keys and one corrupted key to the average of valid keys. The keys were processed in the Alcatel One Touch Pop 2 TrustZone and the cache timings measured cross-core.*

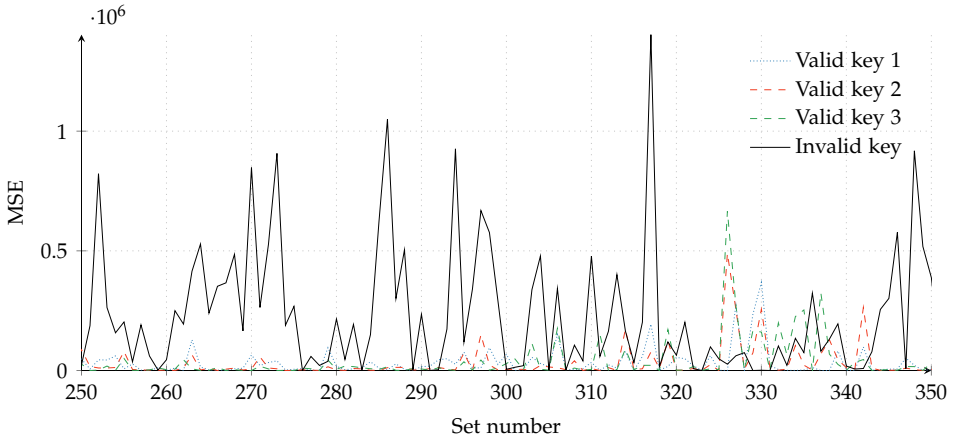


Figure 4.11: Mean squared error for Prime+Probe *cache profiles on a subset of cache sets to illustrate the difference between valid keys in contrast to an invalid key.*

4.4.1 Hammering from Userspace

Since a flush instruction is only available to us on the Samsung Galaxy S6, we need to use eviction on all other test devices. In Section 3.2 we have shown how fast and efficient eviction strategies can be found and thus used in our experiments.

In order to induce bit flips, we have written a tool that allocates memory and utilises the reverse-engineered DRAM mapping function to detect addresses in three rows next to each other in the same bank. Next, the data of the row in the middle is set to a defined value and the attack is started by repeatedly accessing the row on the sides. After that, the content of the row in the middle is checked if any bits differ from the pre-defined and previously set value. This is repeatedly done for all found triples of rows in order to detect vulnerable rows.

On the Samsung Galaxy S6, we were able to use the `DC CIVAC` instruction and thus flush an address to the main memory from userspace without the need of applying eviction. However, we were not able to produce any bit flips on our device which could be due mitigation techniques implemented in the used LPDDR4 DRAM. The LPDDR4 memory standard [40] defines optional hardware support for the Target Row Refresh (TRR) that identifies possible victim rows by counting the number of row activations. The value is then compared to a pre-defined chip specific Maximum Activation Count (MAC) and the row is refreshed if necessary.

Using eviction to produce bit flips on our ARMv7-based test devices was also not successful. The probable reason is that despite the fact that we have found a fast eviction strategy with a high eviction rate, we are still not able to access addresses repeatedly enough. In addition, the refresh rate of the memory decreases if the temperature increases. Thus, the window of time in that we are required to access the different addresses shrinks which in fact is the case as the repeated access increases the CPU load and thus the temperature. In our first experiments we even mounted a CPU cooler of a desktop unit on top of the CPU of the OnePlus One to lower the refresh rate. We even put the whole setup in a fridge in order to rule the possibility of a too high temperature out. However, as we can see now in the results of Van der Veen et al. [83], not every device, even if it is the

same model, is vulnerable to the rowhammer bug and, thus, more devices should have been tested.

4.4.2 Hammering using a Kernel Module

In order to verify if our devices are just not vulnerable to bit flips and if our eviction strategies are too slow to allow high repetition accesses, we used the privileged flush instruction. We have implemented a kernel module that registers itself as a device such that it is possible for a userspace program to communicate with it. Therefore, it allows the userspace program to define two addresses that should be hammered alternately and the number of repetitions. Thus, our previous tool could be easily modified to use the kernel module and with that the privileged flush instruction.

While we were not able to induce bit flips neither on the OnePlus One nor the Samsung Galaxy S6, we had success on the LG Nexus 4, LG Nexus 5 and Samsung Nexus 10.

4.4.3 Hammering using ION

With the release of Android 4.0 Google introduced the ION memory manager to replace the fragmented memory management interfaces across different Android devices with a unified one [88]. ION manages multiple memory pools called ION heaps that can be set to serve special hardware requirements, e.g., for the display controllers or the GPU. In addition, ION allows buffer allocations with uncached memory access from userspace without any privileges or permissions.

Using ION we can circumvent the cache and access the data in the main memory directly. Thus, there is no need for any flush instruction or eviction strategy anymore. However, ION only allows allocating memory up to 4MB large chunks and as much memory as the memory pool is configured to. We have modified our attack tool to exhaust the memory pool and then search for physically contiguous memory in DRAM rows next to each other and to perform the double-sided row hammering. With this we can reproduce bitflips reliable from userspace reliable on the LG Nexus 4, LG Nexus 5 and Samsung Nexus 10.

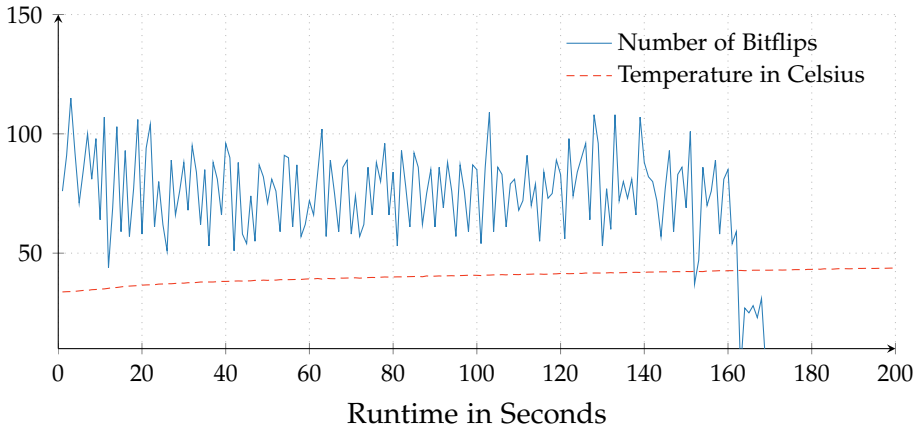


Figure 4.12: Number of bitflips depending on the temperature on the LG Nexus 5

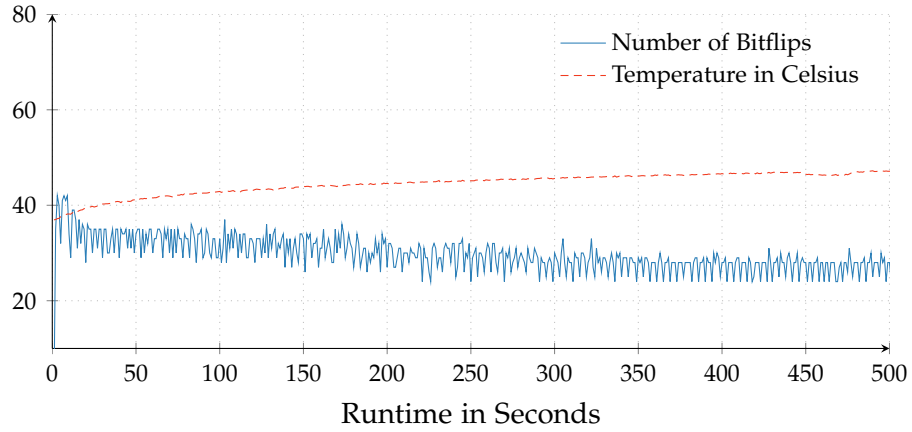


Figure 4.13: Number of bitflips depending on the temperature on the LG Nexus 4.

4.4.4 Observations

Figure 4.12 shows the number of bitflips per second as well as the increasing temperature of the device. We can observe a sudden drop of the number of occurring bitflips as soon as the temperature reaches a certain point. This behaviour is caused by the increase of the refresh rate of the DRAM that depends on certain configured temperature levels [40]. On the LG Nexus 4 in Figure 4.13 we can also observe a clear decline in the number of bitflips as the temperature increases.

In Figure 4.14 we measured the number of bitflips that occur for each row index. We can see that not every row is as prone to bitflips as others.

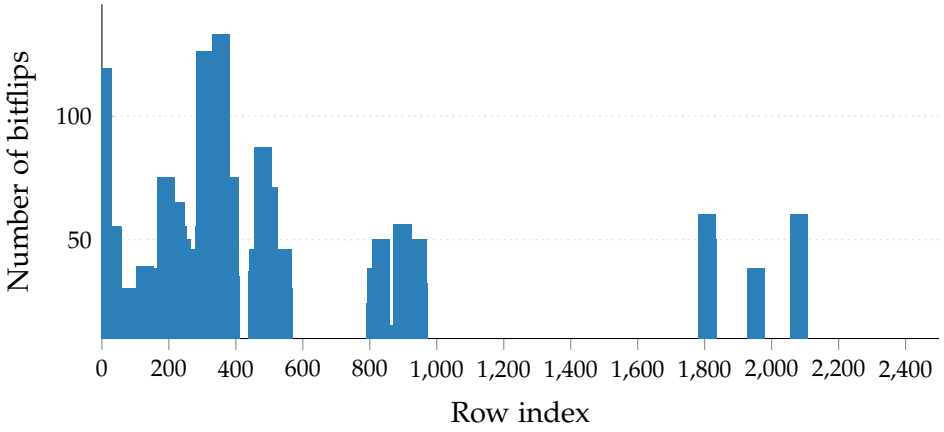


Figure 4.14: Number of bitflips per row on the LG Nexus 4

Chapter 5

Countermeasures

In this chapter, we will present and evaluate possible countermeasures as discussed in our USENIX paper [50]. While the presented attacks exploit weaknesses in hardware, countermeasures implemented in software could make such attacks much more difficult to execute.

The first software weakness we exploit is the unprivileged access to the `perf_event_open` system call interface. This interface allows access to an accurate cycle counter among a variety of other accurate hardware performance counters. In our attack, we use this interface as a replacement to the x86 `rdtsc` instruction. If an attacker were not able to retrieve an accurate cycle count, it would be significantly harder to determine whether a memory access has been a cache hit or a cache miss. Therefore, we suggest making the system call interface only available to privileged processes and especially prevent Android apps from accessing it.

However, we showed in Section 3.1.4 that performing cache attacks without accurate timing is possible by running a second thread that increments a global variable all the time. By reading the value of the variable an attacker can retrieve a timestamp. Thus, only restricting access to the `perf_event_open` system call interface is not sufficient to make the attack impossible.

One big issue is that the operating system supplies information that facilitates our attacks. The `procfs` presents information about running processes on the system in a file-like structure. For every process there will

be a subdirectory named after its process id in `/proc`, containing further files representing information like `status` which provides details about the run state and memory usage of the process. Furthermore, the `pagemap` entry can be used to resolve virtual addresses to physical addresses which we can exploit to build eviction sets. The issue is that this information can be retrieved for any other process on the device. While access to `/proc/pid/pagemap` and `/proc/self/pagemap` has been restricted in Linux in early 2015 [44], the Android kernels on our test devices have not applied this patch yet and, thus, these resources can still be accessed without any permission.

Additionally, we exploit the fact that shared memory in the form of shared libraries is used on the system. While disabling shared libraries would not yield a satisfying solution, at least access to `dex` and `art` optimised program executables should be restricted entirely. At the moment, one cannot retrieve a directory listing of `/data/dalvik-cache/`. However, every file is readable for any process and, thus, *Evict+Reload*, *Flush+Reload* and *Flush+Flush* attacks on Android applications possible.

We have shown cache attacks against AES T-table implementation in Section 4.3. While it has been well-known for quite some time that these attacks are possible, a vulnerable T-table implementation is still employed as the default implementation in BouncyCastle [3]. It is advised to implement AES using dedicated instructions as they are available in the ARM NEON instruction set. Since OpenSSL 1.0.2 a bit-sliced implementation has been introduced on ARMv8-A devices [61].

Since we were able to induce bit flips on mobile devices using the ION memory manager, the userspace interface to ION could be restricted. In addition, memory isolation can be used such that allocated memory from the userspace does not reside next to kernel memory and, thus, the rowhammer bug can't be triggered on security critical memory. As we have observed that the number of bit flips decrease depending on the refresh rate, a higher refresh rate might also be used by default.

Chapter 6

Conclusion

Despite the fact that powerful cross-core cache attacks have been considered not applicable [87] on smartphones, we showed the possibility of highly accurate attacks on ARM. In this thesis, we demonstrated that the cross-core cache attacks *Prime+Probe*, *Flush+Reload*, *Evict+Reload* and *Flush+Flush* can be performed on ARM-based devices.

In order to run these attacks without the requirement of any permission or privileges, we identified different timing sources to retrieve high-resolution timings. Furthermore, we show that coherence protocols can be used to attack caches that are not inclusive and thus enable cache attacks on various ARM platforms. We demonstrate that eviction can be successfully used to compensate a missing unprivileged flush instruction and identify optimal eviction strategies for our test devices.

To emphasize the potential of these attacks, we implemented a covert-channel that is more than 250 times faster than any other state-of-the-art covert channel on Android. In addition, we showed that this covert channel can be established not only cross-core but also cross-CPU. We used these attacks to spy on user input to determine the length of entered words on the soft-keyboard and attack cryptographic implementations in Java. Moreover, we showed that it is possible to monitor the cache activity of the ARM TrustZone from within the normal world.

We showed that ARM-based devices are also vulnerable to the rowhammer bug and are capable of triggering bit flips in a reliable way.

Finally, we have implemented all presented techniques in the form of an open-source library called `libflush` that allows the development of platform-independent cache attacks for the x86 as well as the ARM platform.

Since mobile devices like smartphones and tablets have become the primary personal computing platform, it is of particular importance to deploy hardware and software-based protection mechanisms to prevent unauthorized access and theft of personal data. While our presented attacks by no means cover all possible exploitable information leaks, they stress that it is necessary to deploy effective countermeasures against cache attacks as well as the rowhammer bug.

List of Tables

List of Figures

Bibliography

[1] AlFardan, Nadhem J. ; Paterson, Kenneth G.: Lucky thirteen: Breaking the TLS and DTLS record protocols. In: *Proceedings - IEEE Symposium on Security and Privacy*, 2013. – ISBN 9780769549774, S. 526–540

[2] Android Open Source Project: *Configuring ART*. `https://source.android.com/devices/tech/dalvik/configure.html`. Version: 2015

[3] Archive, Crypto D.: *Bouncy Castle*. `https://www.bouncycastle.org`. Version: 2015

[4] ARM Limited: *ARM Architecture Reference Manual*. ARM Limited, 2007. – 1–1138 S. `http://dx.doi.org/ARMDDI0406C.c`. `http://dx.doi.org/ARMDDI0406C.c`. – ISBN 0201737191

[5] ARM Limited: *ARM Architecture Reference Manual*. ARM Limited, 2007. – 1–1138 S. `http://dx.doi.org/ARMDDI0406C.c`. `http://dx.doi.org/ARMDDI0406C.c`. – ISBN 0201737191

[6] ARM Limited: *ARM Cortex-A Series - Programmers Guide*. 4.0. ARM Limited, 2013

[7] ARM Limited: *big.LITTLE Technology: The Future of Mobile*. 2013

[8] ARM Limited: *Cortex-A15 MPCore Processor Technical Reference Manual*. ARM Limited, 2013

[9] ARM Limited: *ARM® Cortex-A53 MPCore Processor Technical Reference Manual*. ARM Limited, 2014 `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500g/DDI0500G_cortex_a53_trm.pdf`

[10] ARM Limited: *ARM CoreLink CCI-400 Cache Coherent Interconnect Technical Reference Manual*. r1p5. ARM Limited, 2015

[11] Bernstein, Daniel J.: *Cache-timing attacks on AES*. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`. Version: 2005

[12] Biham, Eli ; Shamir, Adi: Differential Fault Analysis of Secret Key Cryptosystem. In: *Advances in Cryptology – CRYPTO '97* Bd. 1294. London, UK, UK : Springer-Verlag, 1997 (CRYPTO '97). – ISBN 3–540–63384–7, 513–525

[13] Bogdanov, Andrey ; Eisenbarth, Thomas ; Paar, Christof ; Wienecke, Malte: Differential cache-collision timing attacks on AES with applications to embedded CPUs. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 5985 LNCS, Springer, 2010 (LNCS). – ISBN 3642119247, S. 235–251

[14] Boneh, Dan ; DeMillo, Richard A. ; Lipton, Richard J.: On the Importance of Checking Cryptographic Protocols for Faults. In: *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques*. Berlin, Heidelberg : Springer-Verlag, 1997 (EUROCRYPT'97). – ISBN 978–3–540–62975–7, 37–51

[15] Callan, Robert ; Zajic, Alenka ; Prvulovic, Milos: A Practical Methodology for Measuring the Side-Channel Signal Available to the Attacker for Instruction-Level Events. In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA : IEEE Computer Society, 2014 (MICRO-47). – ISBN 978–1–4799–6998–2, 242–254

[16] Carlisle, Adams: Constructing of Symmetric ciphers using the CAST design Procedure. In: *Designs, Codes, and Cryptography* 12 (1997), Nov, Nr. 3, S. 283–316. `http://dx.doi.org/10.1023/A:1008229029587`. – DOI 10.1023/A:1008229029587. – ISSN 09251022

[17] Culler, David E. ; Gupta, Anoop ; Singh, Jaswinder P.: *Parallel Computer Architecture: A Hardware/Software Approach*. 1st. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1997. – ISBN 1558603433

[18] Daemen, Joan ; Rijmen, Vincent: *The Design of Rijndael*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2002. – 255 S. http://dx.doi.org/10.1007/978-3-662-04722-4. http://dx.doi.org/10.1007/978-3-662-04722-4. – ISBN 3540425802

[19] Eck, Wim van: Electromagnetic radiation from video display units: An eavesdropping risk? In: *Computers and Security* 4 (1985), Dec, Nr. 4, S. 269–286. http://dx.doi.org/10.1016/0167-4048(85)90046-X. – DOI 10.1016/0167–4048(85)90046–X. – ISSN 01674048

[20] Edge, Jake: *Perfcounters added to the mainline*. http://lwn.net/Articles/339361/. Version: Jul 2009

[21] Genkin, Daniel ; Pachmanov, Lev ; Pipman, Itamar ; Tromer, Eran: *ECDH key-extraction via low-bandwidth electromagnetic attacks on PCs*. Cryptology ePrint Archive, Report 2016/129

[22] Goodman, James R.: Retrospective: Using cache memory to reduce processor-memory traffic. In: *ACM SIGARCH Computer Architecture News* Bd. 11. New York, NY, USA : ACM, 1983 (ISCA '83 3). – ISBN 1581130589, 124–131

[23] Google Inc.: *BoringSSL*. https://boringssl.googlesource.com/boringssl. Version: 2016

[24] Google Inc.: *Full-Disk Encryption*. https://source.android.com/security/encryption/full-disk.html. Version: 2016

[25] Gruss, Daniel: *Rowhammer bitflips on Skylake with DDR4*. https://twitter.com/lavados/status/685618703413698562. Version: Jan 2016

[26] Gruss, Daniel ; Maurice, Clémentine ; Mangard, Stefan: Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: *arXiv:1507.06955v1* 2016 (2015). http://arxiv.org/abs/1507.06955

[27] Gruss, Daniel ; Maurice, Clémentine ; Wagner, Klaus: Flush + Flush : A Stealthier Last-Level Cache Attack. In: *arXiv:1511.04594* abs/1511.0 (2015), 1–14. http://arxiv.org/abs/1511.04594

[28] Gruss, Daniel ; Spreitzer, Raphael ; Mangard, Stefan: Cache template attacks: Automating attacks on inclusive last-level caches. In:

Proceedings of the 24th USENIX Security Symposium, USENIX Association, 2015. – ISBN 9781931971232, 897–912

[29] Gullasch, David ; Bangerter, Endre ; Krenn, Stephan: Cache games - Bringing access-based cache attacks on AES to practice. In: *Proceedings - IEEE Symposium on Security and Privacy*, IEEE Computer Society, 2011. – ISBN 9780769544021, S. 490–505

[30] Gülmezoglu, Berk ; Inci, Mehmet S. ; Irazoqui, Gorka ; Eisenbarth, Thomas ; Sunar, Berk: A Faster and More Realistic Flush + Reload Attack on AES. In: *Proceedings of the 6th international workshop on Constructive Side-Channel Analysis and Secure Design (COSADE'15)* Bd. 9064, Springer, 2015 (LNCS). – ISBN 978–3–319–21476–4; 978–3–319–21475–7, S. 111–126

[31] Hennessy, J.L. ; Patterson, D.a.: *Computer architecture: a quantitative approach - Appendix D*. 3. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2003. – ISBN 1558607242

[32] Hund, Ralf ; Willems, Carsten ; Holz, Thorsten ; Bochum, Ruhr-university: Practical Timing Side Channel Attacks Against Kernel Space ASLR. In: *IEEE Symposium on Security and Privacy – S&P*, IEEE, 2013, S. 191–205

[33] Intel: *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2005. – 1–660 S. http://dx.doi.org/10.1535/itj.0903.05. http://dx.doi.org/10.1535/itj.0903.05. ISSN 15222594

[34] Intel: *Intel' s Advanced Encryption Standard (AES) Instructions Set*. 2010

[35] Irazoqui, Gorka ; Eisenbarth, Thomas ; Sunar, Berk: S$A : A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing — and its Application to AES. In: *IEEE Symposium on Security and Privacy – S&P*, IEEE Computer Society, 2015

[36] Irazoqui, Gorka ; Eisenbarth, Thomas ; Sunar, Berk: Cross Processor Cache Attacks. In: *Proceedings of the 2016 ACM Asia Conference on Computer and Communications Security (AsiaCCS'16)* (2016), 353–364. http://dx.doi.org/10.1145/2897845.2897867. – DOI 10.1145/2897845.2897867. ISBN 9781450342339

[37] Irazoqui, Gorka ; Inci, Mehmet S. ; Eisenbarth, Thomas ; Sunar, Berk: Wait a minute! A fast, cross-VM attack on AES. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 8688 LNCS, Springer, 2014 (LNCS). – ISBN 9783319113784, S. 299–319

[38] Irazoqui, Gorka ; IncI, Mehmet S. ; Eisenbarth, Thomas ; Sunar, Berk: Know Thy Neighbor: Crypto Library Detection in Cloud. In: *Proceedings on Privacy Enhancing Technologies* 1 (2015), Nr. 1, 25–40. http://dx.doi.org/10.1515/popets-2015-0003. – DOI 10.1515/popets–2015–0003. – ISSN 2299–0984

[39] Irazoqui, Gorka ; IncI, Mehmet S. ; Eisenbarth, Thomas ; Sunar, Berk: Lucky 13 strikes back. In: *ASIACCS 2015 - Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ACM, 2015. – ISBN 9781450332453, 85–96

[40] Jedec Solid State Technology Association: *Low Power Double Data Rate 3*. http://www.jedec.org/standards-documents/docs/jesd209-4a. Version: 2013

[41] Käsper, Emilia ; Schwabe, Peter: Faster and timing-attack resistant AES-GCM. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 5747 LNCS, Springer, 2009 (LNCS). – ISBN 364204137X, S. 1–17

[42] Kelsey, John ; Schneier, Bruce ; Wagner, David ; Hall, Chris: Side channel cryptanalysis of product ciphers. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 1485 (1998), Nr. 2/3, S. 97–110. http://dx.doi.org/10.1007/BFb0055858. – DOI 10.1007/BFb0055858. – ISBN 3540650040

[43] Kim, Yoongu ; Daly, Ross ; Kim, Jeremie ; Fallin, Chris ; Lee, Ji H. ; Lee, Donghyuk ; Wilkerson, Chris ; Lai, Konrad ; Mutlu, Onur ; Labs, Intel: Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. Piscat-

away, NJ, USA : IEEE Press, 2012 (ISCA '14). – ISBN 9781479943944, 1–12

[44] Kirill A. Shutemov: *Pagemap: Do Not Leak Physical Addresses To Non-Privileged Userspace*. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce`. Version: 2015

[45] Kocher, Paul C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: *Proc. of Advances in Cryptology (CRYPTO 1996), Lecture Notes in Computer Scicence 1109* Bd. 1109, Springer, 1996 (LNCS). – ISBN 978–3–540–61512–5, 104–113

[46] Kocher, Paul C. ; Jaffe, Joshua ; Jun, Benjamin: Differential Power Analysis. In: *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*. London, UK, UK : Springer-Verlag, 1999 (CRYPTO '99). – ISBN 3–540–66347–9, 388–397

[47] Könighofer, Robert: A fast and cache-timing resistant implementation of the AES. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 4964 LNCS, Springer, 2008 (LNCS). – ISBN 3540792627, S. 187–202

[48] Lal Shimpi, AnandTech: *Answered by the Experts: ARM's Cortex A53 Lead Architect, Peter Greenhalgh*. `http://www.anandtech.com/show/7591/answered-by-the-experts-arms-cortex-a53-lead-architect-peter-greenhalgh`. Version: 2013

[49] Lanteigne, Mark: *How Rowhammer Could Be Used to Exploit Weakness Weaknesses in Computer Hardware*. `http://www.thirdio.com/rowhammer.pdf`. Version: Mar 2016

[50] Lipp, Moritz ; Gruss, Daniel ; Spreitzer, Raphael ; Maurice, Clémentine ; Mangard, Stefan: ARMageddon: Cache Attacks on Mobile Devices. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX : USENIX Association, August 2016. – ISBN 978–1–931971–32–4, 549–564

[51] Lipp, Moritz ; Maurice, Clémentine: ARMageddon: How Your Smartphone CPU Breaks Software-Level Security and Privacy. In:

Black Hat Europe (2016), Nov. https://www.blackhat.com/eu-16/briefings/schedule/index.html#armageddon-how-your-smartphone-cpu-breaks-software-level-security-and-privacy-4887

[52] Liu, Fangfei ; Yarom, Yuval ; Ge, Qian ; Heiser, Gernot ; Lee, Ruby B.: Last-level cache side-channel attacks are practical. In: *Proceedings - IEEE Symposium on Security and Privacy* Bd. 2015-July, IEEE Computer Society, 2015. – ISBN 9781467369497, S. 605–622

[53] Marforio, Claudio ; Ritzdorf, Hubert ; Francillon, Aurélien ; Capkun, Srdjan: Analysis of the communication between colluding applications on modern smartphones. In: *Proceedings of the 28th Annual Computer Security Applications Conference*, ACM, 2012. – ISBN 9781450313124, 51–60

[54] Maurice, Clémentine ; Neumann, Christoph ; Heen, Olivier ; Francillon, Aurélien: C5: Cross-cores cache covert channel. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 9148, Springer, 2015 (LNCS). – ISBN 9783319205496, S. 46–64

[55] Maurice, Clémentine ; Scouarnec, Nicolas le ; Neumann, Christoph ; Heen, Olivier ; Francillon, Aurélien: Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 9404, Springer, 2015 (LNCS). – ISBN 9783319263618, S. 48–65

[56] Mowery, Keaton ; Keelveedhi, Sriram ; Shacham, Hovav: Are AES x86 cache timing attacks still feasible? In: *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop - CCSW '12*, ACM, 2012. – ISBN 9781450316651, 19

[57] Neve, Michael: *Cache-based vulnerabilities and SPAM analysis*, UCL, thesis, 2006. http://dial.academielouvain.be/vital/access/services/Download/boreal:5035/PDF_01

[58] Neve, Michael ; Seifert, Jean-Pierre: Advances on Access-Driven Cache Attacks on AES. In: *Sac* Bd. 4356, Springer, 2006 (LNCS). –

ISBN 978–3–540–74461–0, S. 147–162

[59] Neve, Michael ; Seifert, Jean-Pierre ; Wang, Zhenghong: A refined look at Bernstein's AES side-channel analysis. In: *Proceedings of the 2006 ACM Symposium on Information, computer and communications security - ASIACCS '06*, ACM, 2006. – ISBN 1595932720, 369

[60] Nilsen, Kelvin: *Cache Issues in Real-Time Systems*. 1994

[61] OpenSSL Software Foundation: *OpenSSL Project*. `http://www.openssl.org`. Version: 2014

[62] Oren, Yossef ; Kemerlis, Vasileios P. ; Sethumadhavan, Simha ; Keromytis, Angelos D.: The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, ACM, 2015. – ISBN 9781450338325, 1406–1418

[63] Osvik, Dag A. ; Shamir, Adi ; Tromer, Eran: Cache Attacks and Countermeasures: the Case of AES. In: *Topics in Cryptology – CT-RSA* Bd. 3860, Springer, 2005 (LNCS), S. 1–20

[64] Page, Dan: Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. In: *IACR Cryptology ePrint Archive* 2002 (2002), Nr. August, S. 169

[65] Papamarcos, Mark S. ; Patel, Janak H.: A low-overhead coherence solution for multiprocessors with private cache memories. In: *ACM SIGARCH Computer Architecture News* Bd. 12. New York, NY, USA : ACM, 1984 (ISCA '84). – ISBN 0–8186–0538–3, 348–354

[66] Percival, Colin: *Cache missing for fun and profit*. `http://pdos.csail.mit.edu/6.858/2011/readings/ht-cache.pdf`. Version: 2005

[67] Pessl, Peter ; Gruss, Daniel ; Maurice, Clémentine ; Schwarz, Michael ; Mangard, Stefan: Reverse Engineering Intel DRAM Addressing and Exploitation. In: *ArXiv e-prints* abs/1511.0 (2015). `http://arxiv.org/abs/1511.08756`

[68] Rebeiro, Chester ; Selvakumar, David ; Devi, A. S. L.: Bitslice Implementation of AES. In: *Cryptology and Network Security – CANS* Bd. 4301, Springer, 2006 (LNCS), 203–212

[69] Savaş, Erkay ; Yilmaz, Cemal: A Generic Method for the Analysis of a Class of Cache Attacks: A Case Study for AES. In: *The Computer Journal* 58 (2015), Nr. 10, 2716–2737. http://dx.doi.org/10.1093/comjnl/bxv027. – DOI 10.1093/comjnl/bxv027. – ISSN 0010–4620

[70] Schlegel, Roman ; Zhang, Kehuan ; Zhou, Xiaoyong: Soundcomber: A stealthy and context-aware sound trojan for smartphones. In: *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, The Internet Society, 2011, 17–33

[71] Schneier, Bruce: Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In: *Fast Software Encryption. Lecture Notes in Computer Science. Cambridge Security Workshop Proceedings (December 1993)* Bd. 809. London, UK, UK : Springer-Verlag, 1994. – ISBN 978–3–540–58108–6, 191–204

[72] Seaborn, Mark: *Exploiting the DRAM rowhammer bug to gain kernel privileges*. http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html. Version: Mar 2015

[73] Seaborn, Mark: *Exploiting the DRAM rowhammer bug to gain kernel privileges*. https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf. Version: Aug 2015

[74] Sorin, Daniel J. ; Hill, Mark D. ; Wood, David A.: *A Primer on Memory Consistency and Cache Coherence*. 1st. Morgan & Claypool Publishers, 2011. – 1–212 S. http://dx.doi.org/10.2200/S00346ED1V01Y201104CAC016. http://dx.doi.org/10.2200/S00346ED1V01Y201104CAC016. – ISBN 9781608455645

[75] Spreitzer, Raphael ; Gérard, Benoît: Towards more practical time-driven cache attacks. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 8501 LNCS, Springer, 2014 (LNCS). – ISBN 9783662438251, S. 24–39

[76] Spreitzer, Raphael ; Plos, Thomas: Cache-Access Pattern Attack on Disaligned AES T-Tables. In: *Constructive Side-Channel Analysis and*

Secure Design – COSADE Bd. 7864, Springer, 2013 (LNCS), 200–214

[77] Spreitzer, Raphael ; Plos, Thomas: On the applicability of time-driven cache attacks on mobile devices. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 7873 LNCS, Springer, 2013 (LNCS). – ISBN 9783642386305, S. 656–662

[78] Sweazey, P. ; Smith, a. J.: A class of compatible cache consistency protocols and their support by the IEEE futurebus. In: *ACM SIGARCH Computer Architecture News* Bd. 14. Los Alamitos, CA, USA : IEEE Computer Society Press, 1986 (ISCA '86 2). – ISBN 0–8186–0719–X, 414–423

[79] Systems, Multicore: *Fundamentals of Parallel Computer Architecture*. 1st. Chapman & Hall/CRC, 2015. – ISBN 9780984163007

[80] Takahashi, Junko ; Fukunaga, Toshinori ; Aoki, Kazumaro ; Fuji, Hitoshi: Highly accurate key extraction method for access-driven cache attacks using correlation coefficient. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 7959 LNCS, Springer, 2013 (LNCS). – ISBN 9783642390586, S. 286–301

[81] Tromer, Eran ; Osvik, Dag A. ; Shamir, Adi: Efficient cache attacks on AES, and countermeasures. In: *Journal of Cryptology* 23 (2010), Nr. 1, S. 37–71. http://dx.doi.org/10.1007/s00145-009-9049-y. – DOI 10.1007/s00145–009–9049–y. – ISBN 978–3–540–31033–4

[82] Tsunoo, Yukiyasu ; Tsujihara, Etsuko ; Minematsu, Kazuhiko ; Miyauchi, Hiroshi: Cryptanalysis of Block Ciphers Implemented on Computers with Cache. In: *International Symposium on Information Theory and Its Applications* Bd. 2779, Springer, 2002 (LNCS), S. 803–806

[83] Veen, Victor van d. ; Fratantonio, Yanick ; Lindorfer, Martina ; Gruss, Daniel ; Maurice, Clémentine ; Vigna, Giovanni ; Bos, Herbert ; Razavi, Kaveh ; Giuffrida, Christiano: Drammer : Deterministic Rowhammer Attacks on Commodity Mobile Platforms. In: *ACM Conference on Computer and Communications Security – CCS*, 2016

[84] Weiss, Michael ; Heinz, Benedikt ; Stumpf, Frederic: A cache timing attack on AES in virtualization environments. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 7397 LNCS, Springer, 2012 (LNCS). – ISBN 9783642329456, S. 314–328

[85] Weiss, Michael ; Weggenmann, Benjamin ; August, Moritz ; Sigl, Georg: On cache timing attacks considering multi-core aspects in virtualized embedded systems. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 9473, Springer, 2015 (LNCS). – ISBN 9783319279978, S. 151–167

[86] Yarom, Yuval ; Benger, Naomi: Recovering OpenSSL ECDSA Nonces Using the Flush+Reload Cache Side-channel Attack. In: *Cryptology ePrint Archive, Report 2014/140* (2014)

[87] Yarom, Yuval ; Falkner, Katrina: FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: *USENIX Security Symposium*, USENIX Association, 2014, S. 719–732

[88] Zeng, Thomas M.: *The Android ION memory allocator*. `http://lwn.net/Articles/480055/`. Version: 2012

[89] Zhang, Kehuan ; Wang, Xiaofeng: Peeping Tom in the Neighborhood : Keystroke Eavesdropping on Multi-User Systems. In: *USENIX Security Symposium* Bd. 20, USENIX Association, 2009, 23