On the Applicability of Cache Attacks on Mobile Devices

Raphael Christian Spreitzer
raphael.spreitzer@gmail.com

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Master's Thesis

Supervisor: Dipl.-Ing. Dr. techn. Thomas Plos
Assessor: Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Karl-Christian Posch

September, 2012

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ……. ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

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 marked all material which has been quoted either literally or by content from the used sources.

……. ……..
date (signature)

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, Thomas Plos, who piqued my curiosity in this study and finally proposed the interesting topic of implementation attacks on mobile devices. He always had a sympathetic ear for my concerns and pointed me towards the right direction. I would also like to thank the whole team of the IAIK for their valuable suggestions, especially Johannes Winter for the discussions regarding Linux systems and memory management. Furthermore, I would like to thank the Faculty of Computer Science at the TU Graz for supporting this study with a scholarship.

Most importantly, I would like to express my sincere gratitude towards my parents, Christian and Jutta Spreitzer, for their dedicated support in every imaginable situation throughout my life. Besides, I would like to thank my girlfriend, Michaela, for her sympathetic support and her endless patience with me. Last but certainly not least, I would like to thank my two brothers and my friends for making my spare time more pleasant.

Abstract

The ubiquitous usage of mobile devices like smartphones and tablet computers in our everyday life imposes new challenges in terms of security. In order to cope with these challenges cryptographic primitives can be applied. Cryptographic primitives are used to protect sensitive data on the device itself and to facilitate the secure communication of applications over the Internet. One of these important cryptographic primitives is denoted by symmetric encryption algorithms, with the Advanced Encryption Standard (AES) being the most prevalent algorithm of this type today.

Though the AES is considered to be secure, i.e., no mathematical flaws have been announced yet, the implementation of the algorithm itself might not always be secure. Cache attacks are a special form of implementation attacks and focus on exploiting weaknesses in the implementation of a specific algorithm. In particular, cache attacks exploit different memory-access times within the memory hierarchy. In this thesis we investigate three different cache attacks on current Android-based mobile devices. Therefore, we analyze two time-driven attacks and demonstrate that execution times leak information about memory accesses on mobile devices. This timing information allows us to deduce parts of the secret key used by an AES implementation and with a subsequent brute-force key search it might be even possible to reveal the whole secret key. In addition, we demonstrate a new approach based on the analysis of memory-access patterns, which in some cases allows us to recover the secret key even without a subsequent brute-force key search. Based on the fact that only a few encryptions are necessary in order to recover the secret key this attack might succeed in just a few minutes or even seconds. Thus, this attack poses a serious threat for cryptographic implementations on today's mobile devices. The results presented in this thesis clearly emphasize the importance of considering side-channel attacks at the implementation level and the need for countermeasures in order to prevent such attacks, even on mobile devices.

Keywords: AES, Android, cache attacks, implementation attacks, memory-access patterns

Kurzfassung

Die allgegenwärtige Verwendung von mobilen Endgeräten, z.B. Smartphones und Tablet-Computern, im alltäglichen Leben stellt eine Herausforderung in Bezug auf die Sicherheit dar. Um diese Herausforderungen zu bewältigen werden kryptografische Verfahren eingesetzt. Diese Verfahren werden verwendet um sensible Daten am Endgerät zu schützen und eine sichere Kommunikation von Applikationen über das Internet zu ermöglichen. Eines dieser kryptografischen Verfahren stellen symmetrische Verschlüsselungsalgorithmen dar, wobei der Advanced Encryption Standard (AES) heute der am häufigsten verwendete dieser Art ist.

Obwohl der AES als sicher gilt und somit keine mathematischen Schwachstellen bekannt sind, muss eine spezifische Implementierung dieses Algorithmus nicht notwendigerweise auch sicher sein. Cache-Attacken sind eine spezielle Form von Seitenkanalattacken, die solche Schwachstellen in Implementierungen bestimmter Algorithmen ausnutzen. Insbesondere profitieren Cache-Attacken von den unterschiedlichen Zugriffszeiten die sich auf Grund der Speicherhierarchie ergeben. In dieser Arbeit untersuchen wir drei unterschiedliche Cache-Attacken auf aktuellen Android-basierten Geräten. Im Speziellen untersuchen wir zwei zeitbasierte Attacken und zeigen, dass Ausführungszeiten Informationen über Speicherzugriffe preisgeben. Die Ausführungszeiten ermöglichen es uns Teile des geheimen Schlüssels zu ermitteln und mit einem darauffolgenden Brute-Force-Angriff möglicherweise sogar den vollständigen Schlüssel zu gewinnen. Weiters stellen wir einen neuen Ansatz vor, der auf der Analyse von Speicherzugriffsmustern basiert. Dieser ermöglicht es uns unter gewissen Umständen den geheimen Schlüssel sogar ganz ohne Brute-Force-Angriff zu ermitteln. Auf Grund der Tatsache, dass dieser Ansatz nur wenige Verschlüsselungen benötigt, ist es möglich den geheimen Schlüssel innerhalb weniger Minuten oder sogar Sekunden zu gewinnen. Somit stellt dieser Angriff eine ernstzunehmende Bedrohung für kryptografische Implementierungen auf mobilen Endgeräten dar. Die Ergebnisse dieser Arbeit betonen die Gefahr von Seitenkanalattacken und unterstreichen die darausfolgende Notwendigkeit entsprechende Gegenmaßnahmen bereits auf Implementierungsebene zu treffen.

Stichwörter: AES, Android, Cache-Attacken, Implementierungsattacken, Speicherzugriffsmuster

Contents

Chapter 1

Introduction

Motivated by the fact that more and more smartphones are activated every day the aim of this master's thesis is to analyze whether cache attacks are applicable on today's mobile devices. Besides the standard functionality, e.g., texting, calling, and browsing the web, these devices are also used for business applications, managing banking transactions, and even for payment applications. Thus, these devices inevitably hold sensitive data and private information about the user, which must be protected against adversaries. Standardized cryptographic algorithms like the Advanced Encryption Standard (AES) might be employed in order to secure the ubiquitous usage of these devices. Therefore, such algorithms might be implemented in software to provide the required functionality. However, the implementation of an algorithm which is considered to be secure does not necessarily lead to a secure implementation. This is where implementation attacks come into play. Implementation attacks exploit such weak implementations and aim at recovering the used secret key.

With the availability of powerful processors the usage of central-processing unit (CPU) caches has become inevitable in order to overcome the gap between the high clock frequencies of the CPU and the slow main-memory access times. Therefore, caches are used to hold frequently used data as close as possible to the CPU. Today, not only desktop computers but also mobile devices employ such powerful processors and hence also CPU caches. Though today's processors are quite powerful, software implementations of any kind are optimized for performance, and so are encryption algorithms. Encryption algorithms are usually made up of complex mathematical operations and in case of software implementations of the Advanced Encryption Standard (AES) these operations are precomputed and looked-up later on. The crucial point is that data located within CPU caches might be fetched an order of magnitude faster than data from main memory. Cache attacks aim at exploiting these different access times within the memory hierarchy. Since the cache is a shared resource between all processes on the same CPU an attacker might even cause manipulations of the cache in order to induce such timing differences within other processes. Thus leading to even more sophisticated cache attacks.

Cache attacks have been launched successfully on a variety of desktop computers [15, 46] and also on embedded devices like the ARM9 board [17]. Some of them haven been shown to impose serious threats for existing desktop computers and their corresponding applications. The rising popularity of mobile devices in our everyday life clearly states the need for an analysis of such implementation attacks on mobile devices. In this master's thesis we investigate the applicability of cache attacks on mobile devices in real-world environments. According to our knowledge, we are the first to analyze whether mobile

devices, which employ powerful ARM Cortex-A8 and ARM Cortex-A9 processors running a fully-functioning operating system, are vulnerable to these types of attacks.

In the following we briefly outline the basic structure of this thesis.

Chapter 2 states the need for CPU caches in today's computer systems and introduces a common understanding of cache-mapping schemes as well as different replacement strategies.

Chapter 3 introduces implementation attacks in general and outlines the classification of these attacks. This chapter focuses on cache attacks and illustrates their main working principles.

Chapter 4 sketches the required preliminaries, including an overview of the Android operating system, a short introduction of the AES with a focus on its implementation in software, and the basic principles of the ARM architecture. Finally, we also introduce Intel's Time-Stamp Counter (TSC).

Chapter 5 outlines an approach in order to evict specific parts of the CPU cache. The main part of this chapter deals with the applicability of three different cache attacks on mobile devices. This chapter also outlines our suggested attack approach, which is based on the analysis of cache-access patterns.

Chapter 6 concludes this work and gives an overview of possible future research areas in this field.

Chapter 2

CPU Caches

In this chapter we introduce the basic principles of CPU caches in order to deal with cache attacks in the remainder of this thesis. First of all, we outline the memory hierarchy and emphasize the necessity of CPU caches in today's computer systems at all. Secondly, we cover details of cache-mapping schemes and finally we state the most commonly implemented replacement strategies.

2.1 Memory Hierarchy

Ideally, memory would be indefinitely large, extremely fast — meaning the central processing unit (CPU) could access it without any delay — and cheap. Unfortunately, due to financial reasons, computer systems include only a small portion of fast memory. Furthermore, CPU frequencies tend to increase much faster than memory access times [24]. For instance, Williams [47] states that a CPU with a clock frequency of 2 GHz would require main memory access times of 0.5 ns in order to ensure the CPU operating without any stalls or wait states. Such access times are simply not possible by now, at least in case of the main memory, and due to the fact that fast memory is expensive such an approach (a large and fast main memory) would not be a cost-efficient solution at all.

In order to overcome the gap between CPU clock frequencies and memory-access times the memory hierarchy comes into play. This is a cost-efficient solution to hide the memory latency from the CPU. Fast and expensive memory is only available in small quantities and therefore located at the top of the hierarchy. Going downwards in the hierarchy the memory size as well as the access times increase, whereas the price decreases. Figure 2.1 visualizes the memory hierarchy. Regarding the relatively small size of fast memory available, the question is, how does the memory hierarchy overcome the performance gap between the CPU frequency and the memory latency. This question was answered by researchers at IBM in the 1960s, who observed that code as well as data are exceptionally repetitive [24]. This is quite convenient since we can build a memory hierarchy with fast and small memories at the top, and cheap and large memories at the bottom. Data accessed recently and probably accessed in the near future is loaded into the fast memory regions, whereas data currently not needed is left in slower memory further down in the hierarchy. As a result CPU wait states can be reduced.

Literature typically refers to two principles formulating the repetitive portions of code and data: the *principle of spatial locality* and the *principle of temporal locality* (cf. [24], [25], [47]). These principles are described shortly in the following two paragraphs.

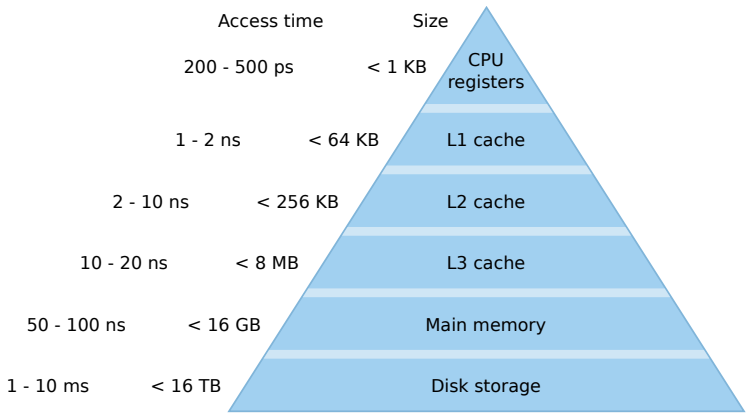


Figure 2.1: Memory hierarchy indicating the typical access time and the memory size for every layer (cf. Hennessy and Patterson [26, p 72]).

Principle of Spatial Locality. This principle, also known as locality in space, states that programs execute the same instructions and access the same data over and over again, and therefore are executed out of a small area. Hennessy and Patterson [25, p 38] state that "*A widely held rule of thumb is that a program spends 90 % of its execution time in only 10 % of the code.*"

Principle of Temporal Locality. This principle, also known as locality in time, states that data accessed recently is much more likely to be accessed again than data which was accessed a long time ago. The temporal comparison refers to a few cycles ago and many thousands of cycles ago.

The above mentioned principles clarify why caches are supposed to close the performance gap between the CPU and the main memory. Data and code currently used, or probably used in the near future, are moved from slower memory (random-access memory (RAM)) to faster memory (CPU cache). Due to the principle of spatial locality it is convenient to load multiple bytes at once, known as *cache line*.

2.2 Cache-Mapping Schemes

Since main memory is typically larger than the cache a mapping has to be established. In order to ensure a common understanding of necessary terms, required for the definition of these cache-mapping schemes, we outline the terms *block address* as well as *virtual* and *physical caches* first. For the sake of clarity, we use byte-addressable memory, which means that each memory location (address) identifies one byte within main memory.

According to Hennessy and Patterson [25], we point out that in the context of caches we usually do not talk about memory addresses, but *block addresses*. Figure 2.2 illustrates which part of the address is used as block address. Such a block address is used to address blocks or lines of data, containing multiple bytes. The number of bytes making up a block, usually 32 or 64 bytes, is determined by the number of *offset* bits, i.e., n bits for 2^n bytes. For mapping a block of data to the cache the *index* bits of the address are used. As many blocks from main memory map to the same location within the cache, a unique identification of the currently mapped block has to be established. This unique identification is accomplished with the *tag* bits.

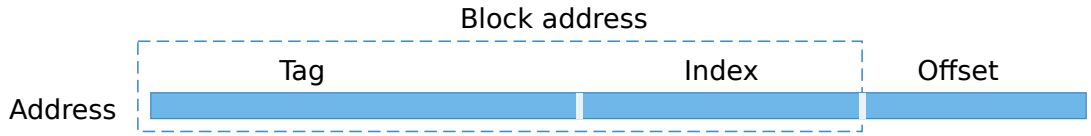


Figure 2.2: Address composition indicating which part of the memory address is used as block address.

In the context of memory addresses we also have to differentiate between physical and virtual addresses and therefore caches can be classified as either *physical* or *virtual caches*. Caches connected to the CPU directly are named virtual or logical caches, whereas caches connected to the CPU with the memory-management unit (MMU) in between are named physical caches [24]. It might be possible to use either the physical address for indexing and the virtual address for tagging or vice versa and, of course, the most obvious disposal to use either the physical or the virtual address for both indexing and tagging [25]. Therefore, caches are further classified into four categories: (1) *virtually indexed, virtually tagged*, (2) *physically indexed, physically tagged*, (3) *virtually indexed, physically tagged*, and (4) *physically indexed, virtually tagged*.

Virtually Indexed, Virtually Tagged (VIVT). As the name already suggests, the virtual address is used for both indexing and tagging. Since the CPU already operates on virtual addresses these caches are considered to be much faster because no address translation is necessary, at least for cache hits. If a cache miss occurs, the virtual address must be translated in order to load the data from main memory. However, one problem of this approach is that multiple virtual addresses might map to the same physical address, which in turn means that the same data is cached separately. Problems arise if data is written to one of these locations. Though the specific cache line (and according to Handy [24] also the physical memory (RAM)) is updated, there might be other cache lines which are not updated. This problem is referred to as *aliasing*. Solutions to this problem, e.g., additional hardware which ensures that each cache line maps to a unique physical address, are stated in [24, 25].

Physically Indexed, Physically Tagged (PIPT). These types of caches use the physical address to determine the index as well as to uniquely identify the location. Basically, this means that the memory-management unit has to translate the virtual address before checking the cache and it circumvents the *aliasing* problem mentioned in the context of VIVT caches. However, PIPT caches are considered slower than VIVT caches because of the inevitable address translation [24, 25].

Virtually Indexed, Physically Tagged (VIPT). In this case the virtual address is used for indexing. While accessing the cache by the index the virtual address is simultaneously translated into the physical address. Later on the physical address is used for the tag comparison. This technique also avoids the *aliasing* problem [24, 25].

Physically Indexed, Virtually Tagged (PIVT). Legitimately, Jacob [31] entitles this combination as peculiar and we mention this technique only for the sake of completeness. These types of caches would use the physical address to access the cache by the index, which means that the virtual address must have been translated already.

Later on the virtual address is used for the tag comparison. For further information about this technique we refer to Hennessy and Patterson [25] as well as Jacob [31].

Property	**Size**
Main-memory size	2^m bytes
Cache-line size	2^l bytes
Number of cache lines	2^n
Cache size	$2^c = 2^l \cdot 2^n$ bytes
Number of cache sets	2^s

Table 2.1: Definition of basic memory attributes (cf. Neve [37]).

Now that we have discussed the basics of memory addresses and the distinction between physical and virtual caches we continue by defining the different cache-mapping schemes. Table 2.1 defines some basic memory attributes which are used throughout the following paragraphs. Referring to Hennessy and Patterson [25], and Williams [47] we define three cache-mapping schemes: *fully associative*, *direct mapped*, and *set associative*.

Fully Associative Caches

In fully associative caches any location within main memory can go to any location within the cache. In order to determine which location is currently mapped to a specific line within the cache the so called *tag* bits are used. As can be seen in Figure 2.3 the *offset* bits (composed of l bits) are used to refer to bytes within a cache line. Note that we defined the cache-line size to be 2^l bytes. The remaining upper bits, the tag bits, are copied to the tag field in the cache. This ensures a unique identification of the address which is currently mapped to this cache line.

When the CPU requests data from memory the cache controller checks whether the requested address is already cached. Therefore, the tag bits of the requested address are compared against all tag fields within the cache. Since checking all cache lines sequentially would be too slow a considerable amount of additional hardware is necessary in order to allow all cache lines to be checked simultaneously. The *offset* bits are then used to return the requested byte within the cache line.

When writing data to the cache all bytes from a block are copied to any line within the cache memory. The number of bytes is determined by the number of bits used as *offset*, e.g., if 5 bits are used as offset then $2^5 = 32$ bytes are copied. Furthermore, the tag bits of the address are copied to the corresponding *tag* field.

Direct-Mapped Caches

For direct-mapped caches there exists exactly one location within the cache where a specific block from main memory can be placed. In contrast to fully associative caches this approach circumvents the huge amount of necessary hardware to check multiple cache lines at once. As can be seen in Figure 2.4 the lower l bits of the memory address are used to access a byte within a cache line. Again, we defined the cache-line size to be 2^l bytes. The next n bits are used to identify the correct location within the cache, which is also referred to as *line* or *index*. These bits establish the fixed mapping between the main memory and the cache. The remaining bits, the so called tag bits, are used to uniquely

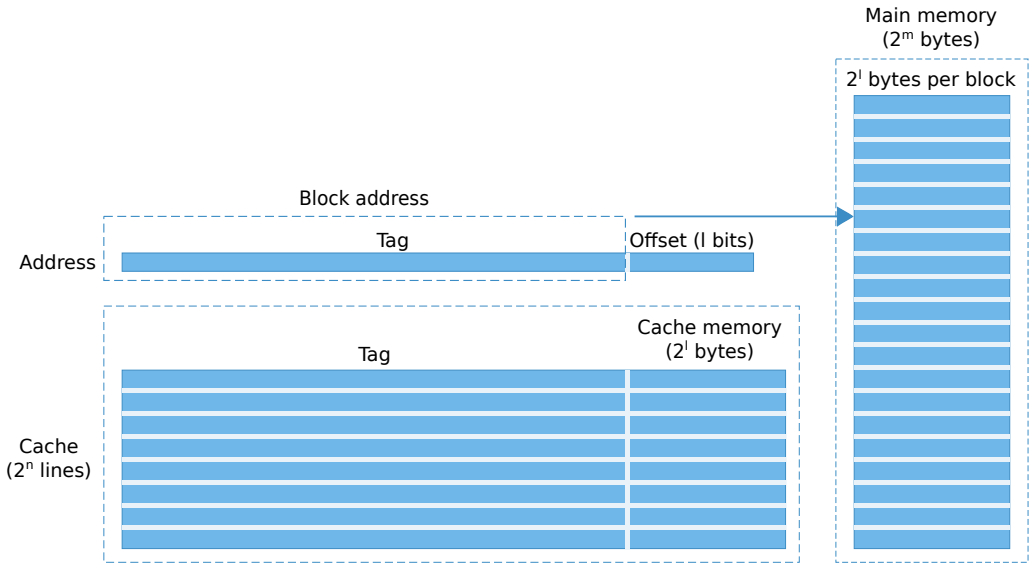


Figure 2.3: Fully associative cache.

identify the main-memory location which is currently mapped to this cache line. This is necessary since multiple blocks from main memory map to the same location within the cache. Hence, the tag bits are simply copied to the tag field within the cache.

When retrieving data from the cache, the cache controller simply determines the location via the index bits and checks whether the tag field matches the tag bits within the address. If the tag bits match, the cache signals a cache hit and the data can be provided directly from the cache[1]. Therefore, the offset is used to slide along the line and to return the requested byte. If the tag bits do not match, a cache miss occurs and the data has to be fetched from main memory.

When data is written to the cache the *line* bits uniquely determine the line where the data should go to. After determining the line, the bytes from the block are copied to the corresponding *cache memory*. As already described above, the *tag* bits are simply copied to the *tag* field within the cache.

One problem of the direct-mapped approach is that two lines mapping to the same location within main memory cannot reside in cache memory at the same time. Since the cache slot is replaced on every access this might result in a poor performance.

Set-Associative Caches

Today, caches are most commonly organized as set-associative caches. Set-associative caches combine the direct-mapped approach and the fully associative approach mentioned above and therefore combine the advantages of these two techniques. A set-associative cache is divided into equally sized parts, each containing k cache lines. Such a cache is said to be *k-way associative*. Blocks from main memory are first mapped to a unique set and then the block can be placed in any line within this set. Of course, additional hardware is necessary in order to check all cache lines within a set simultaneously. However, the

[1]Besides the tag bits there might be other bits to be checked, e.g., the valid bit which determines whether the data currently held at this location is valid. For the sake of simplicity, we omit these details here.

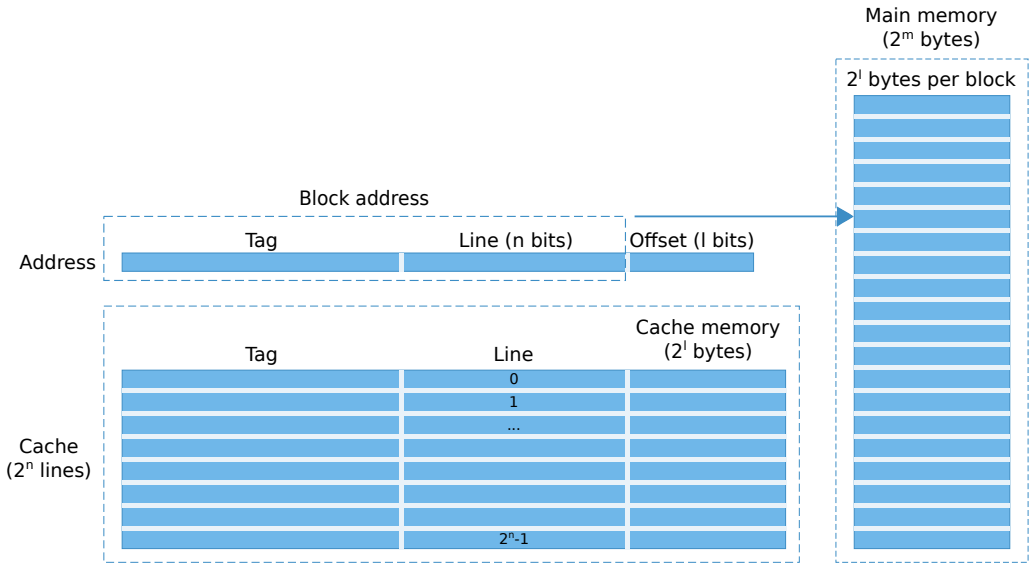


Figure 2.4: Direct-mapped cache

hardware costs are much lower compared to the fully associative approach. Note that direct-mapped caches and fully associative caches are special cases of set-associative caches. A direct-mapped cache is a *1-way* associative cache, whereas a fully associative cache is just a 2^l-way associative cache.

Figure 2.5 illustrates the mapping scheme. When writing data to the cache, the *set* bits of the address are used to determine the unique set within the cache. Then all the bytes from the block are copied to the *cache memory* of any line within the set determined in the previous step. Finally, the *tag* bits of the address are copied to the tag field of the corresponding line within the set.

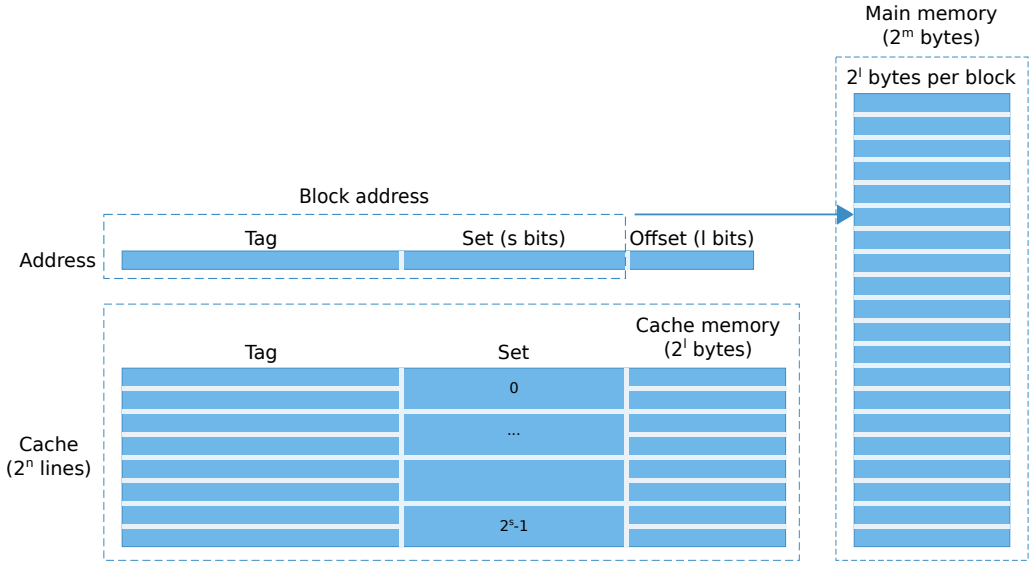


Figure 2.5: Set-associative cache (2-way associative).

Summary of Cache-Mapping Schemes

Figure 2.6 summarizes the three different cache-mapping techniques according to Hennessy and Patterson [25]. Supposing we have a main memory of 32 blocks, each block consisting of 2 bytes, and a cache with 8 lines. Furthermore, the set-associative cache is organized as a 2-way associative cache, consisting of 4 sets. Now we intend to cache block 16. In fully associative caches this block can go anywhere within the cache. In direct-mapped caches there exists exactly one location within the cache where this block can go to ($16 \bmod 8 \equiv 0$). In set-associative caches the block is mapped to a unique set ($16 \bmod 4 \equiv 0$) and can be stored in any line within set 0.

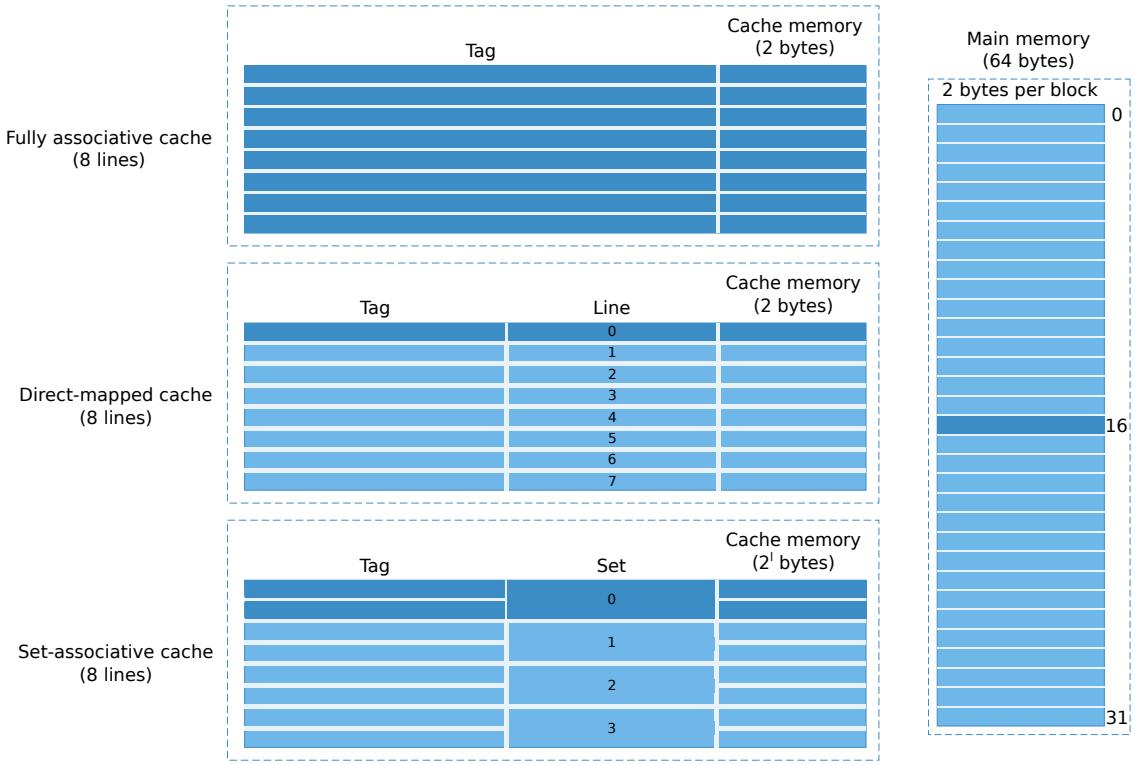


Figure 2.6: Overview of the different cache-mapping schemes when caching block 16 (cf. Hennessy and Patterson [25, p C-7]).

2.3 Replacement Strategies

Since there is only one location where a block can go to in a direct-mapped cache, a new block simply overrides the one which formerly mapped to this location. However, for fully associative caches and set-associative caches there are multiple locations where a block can go to. Now, the question is, which block should be evicted from the cache in order to free up a line for new data. Replacement strategies address this question. According to Handy [24] there are two common replacement strategies: *random* and *least-recently used (LRU)*. Since ARM processors might also support the round-robin replacement policy we also describe this replacement strategy in the following.

Random. Random or pseudo-random replacement means that the line to be evicted is

selected randomly. Since this replacement policy does not need information related to the access frequency, caches with this strategy are kept simple. Furthermore, in case of cache trashing — a commonly used location is replaced all the time — the random-replacement strategy performs better than direct-mapped caches. For example, in an 8-way associative cache the random-replacement policy replaces a recently used cache line with a probability of 12.5 %, whereas the direct-mapped cache always replaces the same line because there is only one location where a specific block can go to.

Least-Recently Used (LRU). According to the principle of temporal locality evicting recently used cache lines should be avoided. The LRU approach keeps track of accesses to every cache line and therefore eliminates the risk of evicting cache lines accessed recently. The downside of this approach is the complexity of keeping LRU statistics up to date and the required memory. If a block can go to N possible locations, then $\log_2(N!)$ bits are necessary to keep track of this information. While for a 2-way associative cache one additional bit is acceptable, an 8-way associative cache would require $\log_2(8!) = 16$ additional bits per set. Furthermore, this information must be updated on every memory access (read and write operation), resulting in significant additional overhead.

Round-Robin. The round-robin replacement policy can be implemented with a simple counter. This counter circularly points to the cache line that should be evicted next [12]. Assuming an initially empty cache, the counter increments until it reaches the last line. Then the counter is reset to zero and the line which was cached first is the first one to be evicted. Hence, the round-robin replacement policy is also known as first in, first out (FIFO) scheme.

Chapter 3

Cache Attacks

This chapter provides an overview of implementation attacks in general. We briefly outline the classification of implementation attacks and state examples for each class of attacks. Finally we also state the main concepts of cache attacks.

3.1 Implementation Attacks

Rather than focusing on the mathematical details of a cryptographic algorithm, implementation attacks exploit weaknesses in the implementation itself. These include hardware weaknesses as well as software weaknesses. Hence, implementation attacks are also known as physical attacks. Koeune and Standaert [34] classify physical attacks along two axes. The first axis determines whether the attack is invasive or non-invasive and the second axis determines whether it is active or passive. While non-invasive attacks exploit only information which can be observed/measured without manipulating the device itself, e.g., run time, the purpose of invasive attacks is to manipulate the device in order to observe details of the computation itself. However, invasive attacks do not influence the computation itself. Influencing the computation itself in order to reveal secret information is specific to active attacks, e.g., fault attacks. Figure 3.1 illustrates the classification of implementation attacks according to these two axes and it also illustrates the positioning of more specialized attacks. Examples for attacks according to this classification are given in the following sections.

3.1.1 Probing Attacks

The first class are probing attacks, which are invasive attacks since the attacker manipulates the device in order to observe details of the cryptographic computation. However, invasive attacks do not involve the manipulation of the computation itself. According to Koeune and Standaert [34], cryptographic devices, especially smart cards, might be attacked by dismantling the device and connecting wires to the data buses in order to monitor the data transfers or using a microscope in order to observe the content of a memory cell.

3.1.2 Fault Attacks

Fault attacks are another class of implementation attacks. The purpose of fault attacks is to introduce faults into the computation and thus leaking secret information. Since smart

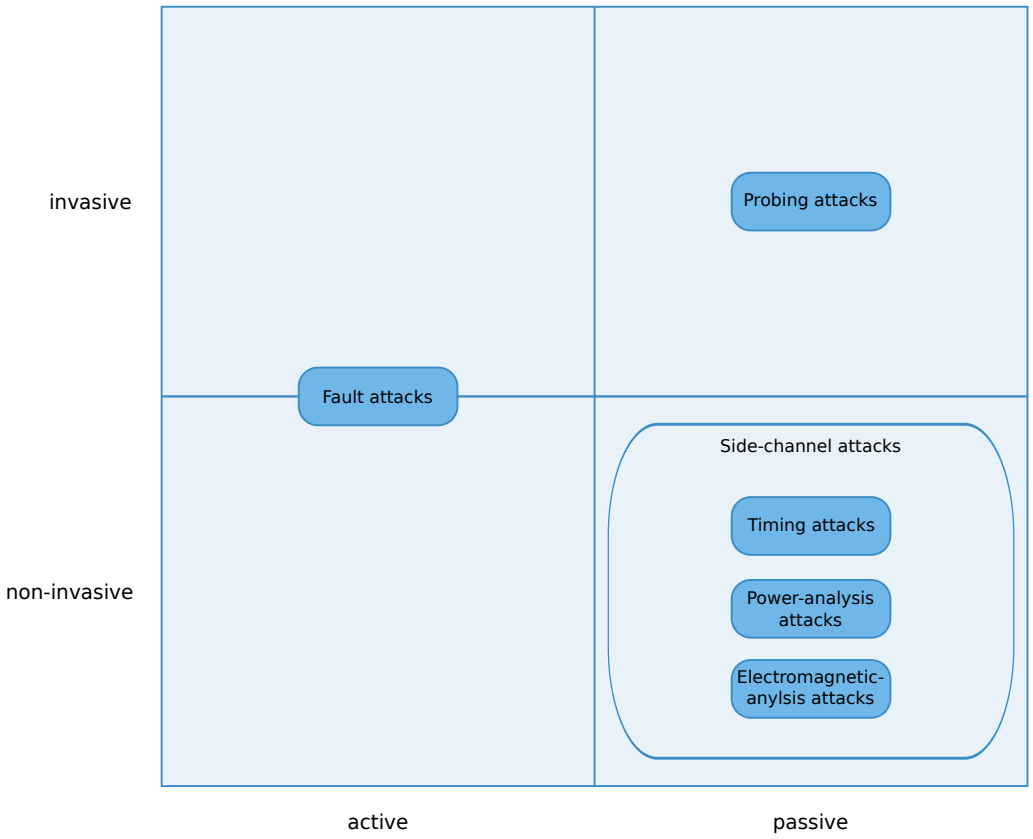


Figure 3.1: Classification of implementation attacks.

cards are powered by their reader, faults might be introduced easily, e.g., voltage peaks. Fault attacks can be either invasive or non-invasive. For instance, introducing faults by increasing the supply voltage of a smart card, also known as *spike* attacks, does not require the manipulation of the smart card. Instead the reader supplies the voltage peaks. The same holds true for changing the clock frequency, also known as *glitching* attacks. In contrast, Skorobogatov and Anderson [44] mention another fault-induction attack with a laser beam. After depackaging the chip they were able to change the state of memory cells by simply illuminating certain areas of the chip. Actually, they claim their attack to be semi-invasive but we do not cover the details here.

3.1.3 Side-Channel Attacks

Besides the intended output of a cryptographic device, i.e., the computed ciphertext of a given plaintext or vice versa, these devices generally also leak unintended information. Different inputs to the cryptographic device might lead to different timing behaviors, different heat, different sounds, or different power consumption. An attacker who is able to measure these differences might also be able to exploit them and to recover secret information. Lawson [36] claims that each component, e.g., CPU, memory, etc., involved in the computation process has its own characteristic, depending on the input provided. These different characteristics might be exploited by attackers.

One of the most famous, and probably one of the earliest, side-channel attacks dates back to 1956. Peter Wright [48], a former officer at the British military intelligence (MI5),

reports an attack on a Hagelin encryption machine. The Hagelin machine was a famous and wide-spread encryption machine in the 1950s, operating on rotating wheels. The initial position of these rotating wheels corresponded to the used secret key. The positioning of these wheels produced a click sound and with a high-sensitive microphone, installed near the Hagelin machine, the MI5 was able to determine the initial position of the rotating wheels. With this information they were able to read the ciphertexts.

The following paragraphs outline the most common types of side-channel attacks, namely *power-consumption attacks*, *electromagnetic-analysis attacks*, and *timing attacks*.

Power-Consumption Attacks. As already mentioned above, different instructions operating on different data might leak secret information through variations in their power consumption. Power-consumption attacks can be further classified into Simple Power Analysis (SPA) and Differential Power Analysis (DPA).

According to Kocher et al. [32], SPA attacks exploit the different power consumptions based on the execution of different instructions. Since power-consumption variations according to instructions are easier detectable, SPA attacks usually require only one measurement with an adequate resolution. The analysis, following the measurement phase, is usually done visually. Power-consumption variations based on the manipulated data are usually smaller than variations based on instructions. Hence, DPA attacks exploit these smaller variations by applying statistical methods to many data samples. For further information, including an illustration of a Differential Power Analysis of DES, we refer to Kocher et al. [32].

Electromagnetic-Analysis Attacks. The term *Tempest* occurs rather often in the literature [6, 19, 34]. It is a codeword, used by the US government, referring to studies and standards regarding the leakage of information through electromagnetic emanations. This clearly shows that military and governmental organizations have been aware of electromagnetic emanations and its security impact at least since the 1960s [19, 35]. With the reconstruction of a distant display unit, Van Eck [22] brought the issue of electromagnetic emanations to the general public in 1985. Later, Quisquater and Samyde [43] extended the idea of Kocher et al. [32, 33], regarding power and timing attacks, to electromagnetic emanations. Electric current, flowing in any conductor, generates an electromagnetic field which can be measured with coils and can be exploited later on. According to Agrawal et al. [6], different components within a device influence the emanations of other components and any component might provide a different source of information leakage. Hence, electromagnetic-analysis attacks are considered far more powerful than power-analysis attacks, where only the overall power consumption of all components is available.

Timing Attacks. Timing attacks are another important class of side-channel attacks. Today, software implementations are optimized for performance and therefore unnecessary operations are simply skipped in order to speedup computations. In cryptography timing variations due to conditional statements and data-dependent execution branches are highly inappropriate, since such timing variations render cryptographic implementations insecure. This results from the fact that an attacker might deduce the executed branches and even the processed data by carefully measuring the execution time.

In 1996 Paul Kocher [33] suggested the exploitation of cryptographic implementations whose execution times depend on the provided input. He states that the modular exponentiation — used, for instance, in RSA-based cryptosystems — compromises the cryp-

Algorithm 1 Square and multiply exponentiation.

Input: m, n, and $d = (d_{l-1}, \ldots, d_0)$, where $d_i \in \{0, 1\}$
Output: $m^d \mod n$

```
x ← m
for i = l − 2 downto 0 do
    x ← x^2 mod n
    if d_i == 1 then
        x ← x · m mod n
    end if
end for
return x
```

tographic implementation if it does not run in a fixed time. For instance, the commonly used *square-and-multiply* algorithm, as stated in Algorithm 1, computes the modular exponentiation. Therefore, it iterates over the binary representation of the exponent **d** and in every iteration the intermediate result **x** is squared. In addition, depending on the current bit of the exponent **d** the intermediate result is either multiplied with the base **m** or not. Now the problem is that due to this key-dependent multiplication one might reveal the secret key (exponent **d**) by carefully measuring the time taken to encrypt different messages and afterwards performing a statistical analysis.

While the timing attacks mentioned in this section are mainly based on data-dependent algorithms, cache attacks exploit different timing behaviors due to cache hits and cache misses. Such timing attacks represent another serious threat and we cover them in the next section.

3.2 Types of Cache Attacks

The following subsections outline the different types of cache attacks and their basic concepts. These attacks are considered to be a special form of side-channel attacks, which exploit different execution times due to different memory-access times within the memory hierarchy. As the name already suggests, these attacks are applicable for cryptographic implementations depending heavily on the cache memory, e.g., implementations employing S-Boxes and look-up tables. At present cache attacks are separated into three categories: *time-driven attacks*, *access-driven attacks*, and *trace-driven attacks*. While *time-driven attacks* require the most measurement samples, these attacks require less knowledge of the implementation and the underlying hardware architecture under attack. In contrast, *access-driven attacks* as well as *trace-driven attacks* require far less measurement samples, but more sophisticated knowledge of the implementation and the cache architecture is necessary.

Generally, the number of recoverable bits per key byte is limited by the number of table elements (S-Box elements or T-table elements) per cache line. Equation 3.1 outlines the number of non-recoverable bits per key byte, which simply results from the fact that one cannot distinguish between accessed elements in one cache line[1]. According to Osvik et al. [46] this is the number of non-recoverable bits per key byte, at least for attacks considering only the first round. Later on we discuss the advantage of attacking disaligned

[1] As already mentioned in Chapter 2 in case of a cache miss multiple bytes are loaded from main memory into the cache.

tables, which permit recovering more bits per key byte within the first round. S-Box tables or T-tables are considered to be disaligned if the start address of these tables is not properly aligned according to memory addresses which are mapped to the beginning of a cache line.

$$\textit{non-recoverable bits per key byte} = \log_2 \frac{\textit{cache-line size in bytes}}{\textit{table-element size in bytes}} \tag{3.1}$$

3.2.1 Time-Driven Attacks

As Bernstein [15] claims, the use of secret data being used as look-up indices into precomputed data structures, e.g., S-Boxes and other look-up tables, might leak exploitable timing information and hence lead to insecure cryptographic implementations. The basic idea of time-driven attacks, as described in [15], is to gather timing information of encryptions under a known secret key as well as timing information under an unknown secret key and to correlate this timing information. Based on the assumption that similar look-up indices yield similar timing information, an attacker might reveal the secret key.

Since cache memory is smaller than main memory many locations within the main memory are mapped to the same location within the cache. This leads to cache evictions of already cached data items, if other data mapping to the same location is accessed. Hence, the interference of different memory accesses, resulting from different processes and even the process performing the cryptographic computation itself, leads to exploitable timing information. Furthermore, Neve [37] claims that memory accesses due to other processes are not completely random and, hence, by averaging over multiple samples of timing information and evaluating these samples with statistical methods an attacker might reveal different timing information depending on the provided input.

3.2.2 Access-Driven Attacks

Though access-driven attacks also exploit timing information leaked through cache hits and cache misses, respectively, the purpose of this type of attack is to determine which cache lines were actually accessed during the cryptographic computation. According to Osvik et al. [46] there are two approaches in order to determine which cache lines are accessed during an encryption. The first approach requires the attacker to populate the whole cache with temporary data. After the encryption the attacker simply checks which cache lines are still present by measuring the time taken to reaccess memory blocks of the afore mentioned temporary data array. Within the second approach the attacker triggers the encryption of a plaintext and afterwards accesses arbitrary data in order to cause the eviction of already loaded table elements (either S-Box or T-table elements). By measuring the encryption time of the same plaintext again the attacker is able to determine whether the evicted data is required for the encryption of the plaintext or not. Obviously, this attack requires knowledge of the location of the precomputed S-Boxes or T-tables in memory as well as information about the cache architecture, e.g., cache size, cache-line size, and associativity.

3.2.3 Trace-Driven Attacks

This type of attack is based on the assumption that for every memory access of the encryption function, i.e., for every look-up into a T-table or an S-Box, an adversary is able to determine whether it resulted in a cache hit or a cache miss. More formally, a

detailed cache profile based on the information of every single memory access is necessary for this attack to be successful. As suggested by Bertoni et al. [16] the use of power traces might leak such a detailed cache-access profile for every memory access. Additionally, Aciiçmez and Koç [5] suggest using performance counters in order to establish such a memory-access profile.

The approach suggested by Bertoni et al. [16] is as follows. The attacker triggers an encryption under a known plaintext which simply initializes the cache, i.e., loads the required S-Box or T-table elements into the cache memory. Afterwards the attacker accesses arbitrary data in order to provoke selective cache evictions of already loaded S-Box elements. Within the last step the attacker captures the power trace of the encryption under the same plaintext again. Supposing a cache miss within the first round plus information about the cryptographic implementation itself the attacker is able to deduce the secret key, or at least parts of it. The second approach, as suggested by Aciiçmez and Koç [5], assumes a clean cache, i.e., filled with arbitrary data. Again the attacker triggers the encryption of a known plaintext and captures the cache trace. Since the attack starts with an empty cache the first access always results in a cache miss. However, the information whether the second access to one and the same table results in a cache hit or a cache miss leaks partial information about the key bytes. Considering the implementation of the AES in software, the first two memory accesses into the T-table $\mathbf{T}_0$ are $s_0 = p_0 \oplus k_0$ and $s_4 = p_4 \oplus k_4$, respectively. Hence, there are two possible cases. First, a cache hit means that the same look-up index was used, thus revealing information about the key-byte difference: $p_0 \oplus p_4 = k_0 \oplus k_4$. Second, a cache miss means that the attacker can restrict the key-byte differences according to $p_0 \oplus p_4 \neq k_0 \oplus k_4$. After infering possible bytes for the first two key bytes, i.e., k_0 and k_4, the attacker continues with the third access to $\mathbf{T}_0$ and so on until all memory accesses to $\mathbf{T}_0$ have been considered. Then the attack continues with the rest of the $\mathbf{T}$-tables analogously.

Chapter 4

Requirements

In this chapter we focus on the required preliminaries in order to investigate cache attacks in the remainder of this thesis. At first, we introduce the Android software stack and the basic architecture of the Android operating system. Second, we outline the required basics of the Advanced Encryption Standard (AES). We especially focus on the software implementation employing T-tables. In addition, we introduce the ARM architecture with a focus on the coprocessor registers which are used for precise timing measurement. Finally, we also cover Intel's Time-Stamp Counter (TSC) which is used on desktop computers for precise timing measurements.

4.1 Android

Developed by the Open Handset Alliance (OHA) [40], with Google probably being the main driving force, Android is an open-source software stack including an operating system and applications providing basic functionalities. Currently, Android is one of the most popular operating systems for mobile devices. Figure 4.1 shows the system architecture of Android, which will be described in the following paragraphs, from bottom to top. Based on Becker and Pant [14], the Android Developer's Guide [8], and Brady [18] we outline the basic characteristics of the Android architecture which consists of the following basic layers: *Linux kernel*, *libraries*, *runtime*, *application framework*, and *applications*.

Linux Kernel. A Linux kernel (version 2.6)[1], which forms the basis of the Android operating system, provides the core system functionalities. Besides process management and power-management functionalities this also includes basic hardware drivers.

Libraries. The Android software stack also includes a bunch of core libraries, written in C/C++, residing above the kernel. One of these libraries is the standard-C system library, also known as *libc*, which provides wrappers for basic system calls. These include file I/O as well as memory management. However, Android uses a custom *libc* implementation called *Bionic*, which is a BSD-derived implementation of the standard-C system library. Issues regarding the implementation of a custom *libc* include (1) keeping the GPL license out of the user space, (2) optimizations and enhancements for use in embedded systems, and (3) built-in support for Android-specific services [18]. Furthermore, these core libraries

[1] Since Android 4.x a Linux kernel 3.x forms the basis.

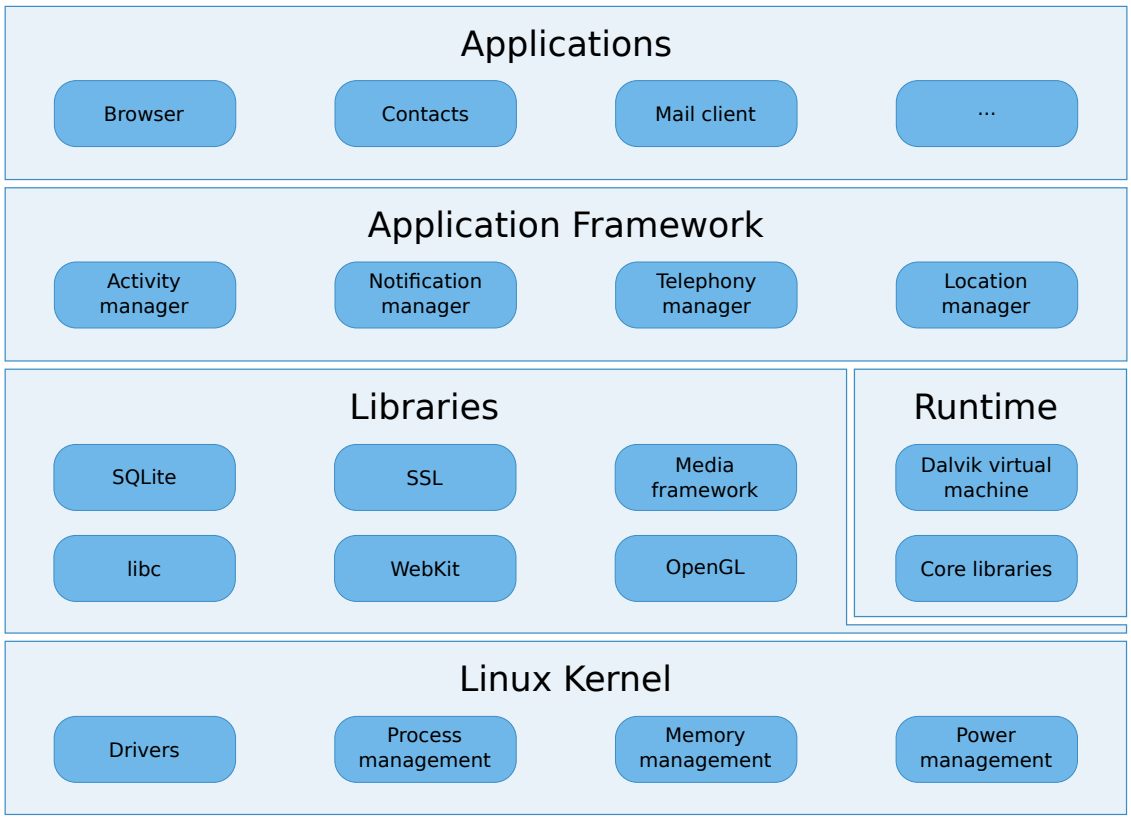


Figure 4.1: Android system architecture (according to [8]).

include basic support for SQLite databases, standard audio and video formats, and many more.

Runtime. Every Android application is intended to run within a separate *Dalvik Virtual Machine* (DVM), which forms the heart of the Android runtime. The DVM is not a standard Java Virtual Machine, but was developed especially for ARM-based devices, which is the most common architecture used in today's mobile devices. The concept of a separate DVM for every application should increase the security of the overall system. However, a separate DVM for every application requires sophisticated optimizations in order to run on resource-constrained devices.

The DVM is capable of executing *.dex* (Dalvik Executable) files, which are generated by a dedicated tool after compiling the Java sources into Java byte code. The *dx*-tool, shipped with the Android SDK, is in charge of this step. These *.dex* files are compressed within an *.apk*, ready to be installed on Android devices. For further information about the Dalvik Virtual Machine we refer to [9].

Application Framework. The application framework provides the ability to access essential low-level services, which are typically not accessed directly by the application, in a more comfortable way. This includes accessing the device hardware, location information, notification services, and many more. Accessing these services takes place through different `xxxManager` classes. For instance, the `android.app.NotificationManager` can be used for notification purposes by utilizing the flash light, LEDs, and the status bar.

The `android.telephony.TelephonyManager` might be used to access telephony services and states, as well as subscriber information. Furthermore, Android significantly stresses the importance of reusable components. Therefore, applications may provide dedicated capabilities to other applications.

Applications. On top of the software stack reside the applications, which provide the interface between the user and the mobile device. Android applications are usually written in Java and compiled with the Android SDK [8]. Android also allows developers to integrate native code in their applications and therefore Google provides the Android Native Development Kit (NDK) [8]. The integration of native libraries into Java applications is done through the Java Native Interface (JNI).

4.2 Advanced Encryption Standard (AES)

In 2000 the National Institute of Standards and Technology announced Rijndael, designed by J. Daemen and V. Rijmen, as the Advanced Encryption Standard (AES) [38]. The AES is an iterated block cipher, processing data blocks of 128 bits, supporting key lengths of 128, 192, or even 256 bits. Rijndael is made up of 10, 12, or 14 rounds, depending on the key length, each consisting of four byte-oriented round transformations. Our investigations are limited to 128-bit keys.

For illustration purposes we denote the plaintext, the key, and the state as consecutive arrays of bytes, i.e., $\mathbf{p} = \{p_0, \ldots, p_{15}\}$, $\mathbf{k} = \{k_0, ...k_{n-1}\}$ (where n is either 16, 24 or 32, depending on the key length), and $\mathbf{s} = \{s_0, \ldots, s_{15}\}$, respectively. As can be seen in Figure 4.2 these consecutive arrays of bytes are represented as two-dimensional arrays of bytes, consisting of four rows and four columns. Furthermore, this figure also illustrates the initial round transformation, defined as the bitwise XOR of the plaintext and the key, i.e., $\mathbf{s} = \mathbf{p} \oplus \mathbf{k}$. The resulting state represents the input for the subsequent round transformations.

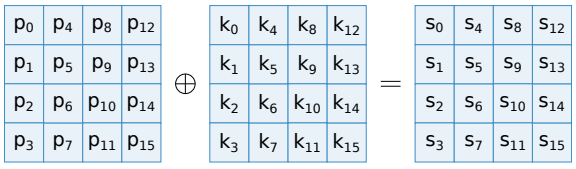


Figure 4.2: AES initial transformation computing plaintext XOR key.

The following paragraphs shortly describe the four round transformations: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. Within the attacks described later on we do not need to know the details of these round transformations. Therefore, we just refer to Daemen and Rijmen [20] as well as the Federal Information Processing Standard [38] for further details about the mathematical background, the round transformations, as well as the key-scheduling algorithm.

SubBytes. Substitutes each byte within the state with a byte retrieved from a precomputed S-Box.

ShiftRows. The last three rows within the state are cyclically shifted to the left over different offsets.

MixColumns. Operates on the state column-by-column, multiplying each column with a fixed polynomial. This operation might be implemented as a matrix multiplication.

AddRoundKey. The last transformation of each round is the addition of the round key to the state, i.e., state XOR key.

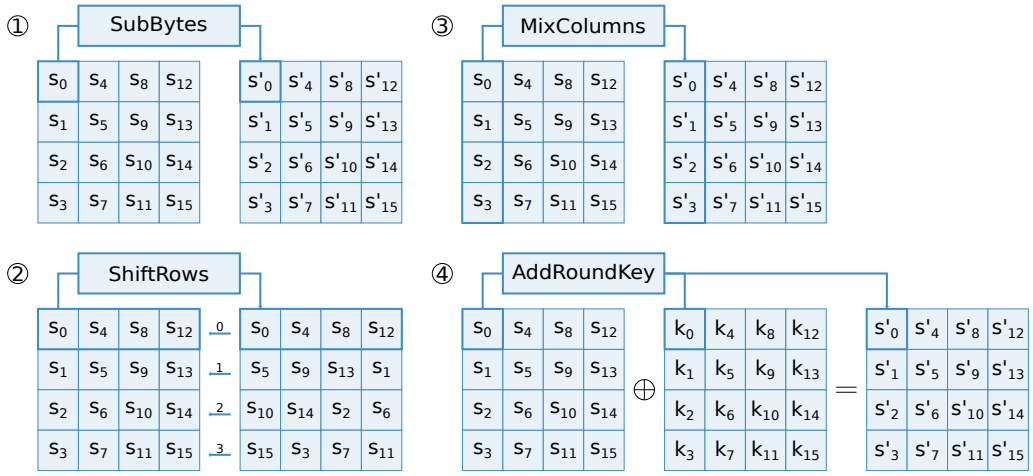


Figure 4.3: AES round transformation.

Figure 4.3 illustrates the general round transformations of the AES. At this point we have to mention that prior to the first round a precomputation step is necessary and that the last round is slightly different than the preceding nine rounds. The initial round transformation simply consists of adding the plaintext to the initial key, as already described above. In all following rounds a generated round key, derived from the initial round key, is added to the state. Furthermore, in the final round the MixColumn operation is omitted. Due to performance reasons software implementations of the AES usually combine the transformations SubBytes, ShiftRows, and MixColumns into precomputed look-up tables [20, pp 53–49].

In the following we outline a possible AES software implementation based on look-up tables. Encryption and decryption each use five precomputed look-up tables ($\mathbf{T}_0, \ldots, \mathbf{T}_4$), each having 1 KB. Four of these look-up tables ($\mathbf{T}_0, \ldots, \mathbf{T}_3$) are used in the rounds 1–9. The fifth table ($\mathbf{T}_4$) results from the omission of the *MixColumns* transformation in the last round. Hence, this table is used solely in the last round. Now, suppose the AES encryption takes the following input parameters: the plaintext $\mathbf{p}$, and the key $\mathbf{k}^0$ which denotes the initial round key. The computed round keys are denoted $\mathbf{k}^r$. After the initial key addition we denote the state $\mathbf{s}$ as $\mathbf{s}^0 = \mathbf{p} \oplus \mathbf{k}^0$. While Equation 4.1 outlines the round computations for rounds 1–9, Equation 4.2 outlines the last round computation and the last state represents the resulting ciphertext, i.e., $\mathbf{c} = \{\mathbf{s}_0^{10}, \ldots, \mathbf{s}_{15}^{10}\}$. Note, that in this case the AES can be implemented by using only look-up operations and bitwise XOR operations. For a fine grasp of the syntax used in the previously mentioned equations, we define the following notations.

Notation 1. *We define the concatenation of bytes as* $\{b_0, b_1, b_2, b_3\} = b_0 b_1 b_2 b_3$

Notation 2. *We consider a look-up table* ($\boldsymbol{T}$) *as a function, mapping a one-byte value to a four-byte value, e.g.,* $\boldsymbol{T}[a] = b_0 b_1 b_2 b_3$. *Consequently, the expression* $(\boldsymbol{T}[a])_i$ *refers to the byte at position* i *of the resulting look-up within table* $\boldsymbol{T}$ *using index* a.

$$
\begin{aligned}
(s_0^{r+1}, s_1^{r+1}, s_2^{r+1}, s_3^{r+1}) &= T_0[s_0^r] \oplus T_1[s_5^r] \oplus T_2[s_{10}^r] \oplus T_3[s_{15}^r] \oplus \{k_0^{r+1}, k_1^{r+1}, k_2^{r+1}, k_3^{r+1}\} \\
(s_4^{r+1}, s_5^{r+1}, s_6^{r+1}, s_7^{r+1}) &= T_0[s_4^r] \oplus T_1[s_9^r] \oplus T_2[s_{14}^r] \oplus T_3[s_3^r] \oplus \{k_4^{r+1}, k_5^{r+1}, k_6^{r+1}, k_7^{r+1}\} \\
(s_8^{r+1}, s_9^{r+1}, s_{10}^{r+1}, s_{11}^{r+1}) &= T_0[s_8^r] \oplus T_1[s_{13}^r] \oplus T_2[s_2^r] \oplus T_3[s_7^r] \oplus \{k_8^{r+1}, k_9^{r+1}, k_{10}^{r+1}, k_{11}^{r+1}\} \\
(s_{12}^{r+1}, s_{13}^{r+1}, s_{14}^{r+1}, s_{15}^{r+1}) &= T_0[s_{12}^r] \oplus T_1[s_1^r] \oplus T_2[s_6^r] \oplus T_3[s_{11}^r] \oplus \{k_{12}^{r+1}, k_{13}^{r+1}, k_{14}^{r+1}, k_{15}^{r+1}\}
\end{aligned} \tag{4.1}
$$

$$
\begin{aligned}
(s_0^{10}, s_1^{10}, s_2^{10}, s_3^{10}) &= \{\left(T_4[s_0^9]\right)_0, \left(T_4[s_5^9]\right)_1, \left(T_4[s_{10}^9]\right)_2, \left(T_4[s_{15}^{10}]\right)_3\} \oplus \{k_0^9, k_1^9, k_2^9, k_3^9\} \\
(s_4^{10}, s_5^{10}, s_6^{10}, s_7^{10}) &= \{\left(T_4[s_4^9]\right)_0, \left(T_4[s_9^9]\right)_1, \left(T_4[s_{14}^9]\right)_2, \left(T_4[s_3^{10}]\right)_3\} \oplus \{k_4^9, k_5^9, k_6^9, k_7^9\} \\
(s_8^{10}, s_9^{10}, s_{10}^{10}, s_{11}^{10}) &= \{\left(T_4[s_8^9]\right)_0, \left(T_4[s_{13}^9]\right)_1, \left(T_4[s_2^9]\right)_2, \left(T_4[s_7^{10}]\right)_3\} \oplus \{k_8^9, k_9^9, k_{10}^9, k_{11}^9\} \\
(s_{12}^{10}, s_{13}^{10}, s_{14}^{10}, s_{15}^{10}) &= \{\left(T_4[s_{12}^9]\right)_0, \left(T_4[s_1^9]\right)_1, \left(T_4[s_6^9]\right)_2, \left(T_4[s_{11}^{10}]\right)_3\} \oplus \{k_{12}^9, k_{13}^9, k_{14}^9, k_{15}^9\}
\end{aligned} \tag{4.2}
$$

Now the crucial point is that, according to Bernstein [15] and Osvik et al. [46], in the first round the bytes of the state $\mathbf{s}^0$ are used as indices into the look-up tables. In short this means that the look-up indices are key dependent. If an attacker knows the plaintext to be encrypted and if one is further able to determine whether the access to the table resulted in a cache hit or a cache miss it might be possible to reveal the secret key. As we will see in the next chapter, implementations of the AES, and block ciphers in general, using precomputed look-up tables are vulnerable to side-channel attacks, more formally such implementations are vulnerable to cache attacks.

4.3 ARM Architecture

Advanced RISC Machines Limited [1] (ARM Ltd.) was found in England in 1990, and is the leading company in providing 32-bit architecture processors for mobile devices today [1, 12]. ARM Ltd. does not manufacture and sell chips directly, but simply licenses the Intellectual Property (IP). This means that semiconductor companies buy licenses and implement the design and specifications of ARM processors in their own products.

One has to distinguish between (1) *architecture* and (2) *processor* [12]. While the architecture specifies, for instance, the instruction set and therefore how the processor must behave, the processor is an implementation of the architecture and might be manufactured by many different companies. Currently, the most wide-spread architecture in use is version ARMv7. Though this architecture is divided into three different profiles, ranging from ARMv7-A for application processors to be used in smartphones and tablet computers, over ARMv7-R for real-time systems, to ARMv7-M for micro controllers, we solely focus on the ARMv7-A profile. The ARMv7-A profile is implemented, for instance, in the Cortex-A8 and Cortex-A9 processors as well as in processors from Qualcomm [4], e.g., Qualcomm Scorpion which is part of the Qualcomm Snapdragon platform [12].

The following three subsections are organized as follows. First, we introduce the ARM Cortex-A series processors, a common processor architecture used in many mobile devices today. Second, we outline the concept of coprocessors and finally we illustrate the performance-monitor registers which can be used for timing and performance measurements.

4.3.1 Cortex-A Series Processors

Since our cache attacks are based on mobile devices operated by a Cortex-A processor we focus on this processor series, which implements the ARMv7-A architecture. The Cortex-

Bit	Value	Description
14	0	Default replacement policy, usually pseudo-random replacement.
	1	Round-robin replacement strategy, if supported.

Table 4.1: *System Control Register* (cf. [10, pp B3-94–B3-100]).

A series is designed for mobile devices with limited power constraints and therefore is a wide-spread mobile processor for smartphones, tablet computers, and netbooks. According to the *Cortex-A Series Programmer's Guide* [12], the Cortex-A series processors use physically indexed, physically-tagged data caches (PIPT). Furthermore, ARM processors usually support two types of cache-replacement policies. These are the round robin and the pseudo-random replacement strategy. Though ARM processors might support both strategies, the pseudo-random replacement policy is preferred due to its simplicity. Obviously, if there are multiple implemented strategies there must be a way to choose between them. Indeed, a register for changing the replacement policy exists. The *ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition)* [10] lists the *System Control Register (SCTLR)* as being intended for managing some memory system settings. The corresponding bit for changing the replacement policy, bit 14, is also referred to as *RR* bit. Table 4.1 lists the possible values and the corresponding settings. However, writing the *RR* bit in any of our tested environments, i.e., *Google Nexus S*, *Acer Iconia A510*, and *Samsung Galaxy SIII*, did not have any effects. According to our observation it was not even possible to flip the *RR* bit, though executed out of privileged code. This might be due to the fact that this bit is listed as *Implementation Defined* and the *Cortex-A8 Technical Reference Manual* [11] does not even document the bits 14–24 of the *Control Register*[2] in more detail, but simply documents that a static value will be returned when reading this range of the register. The *Cortex-A9 Technical Reference Manual* [13] lists this bit as configuration bit for the instruction cache but not for the data cache. Thus we conclude that we are not able to change the replacement policy for any of the three devices.

The differences between the ARM Cortex-A8 and the ARM Cortex-A9 processor are rather subtle. While the ARM Cortex-A8 is a single-core processor with clock frequencies ranging from 600 MHz to 1 GHz [2], the Cortex-A9 is available either as single or multi-core processor with clock frequencies from 800 MHz to 2 GHz [3]. Furthermore, the Cortex-A8 employs a 4-way set associative L1 cache and might support a cache size of 16 KB or 32 KB with a cache-line size of 64 bytes. In contrast, the Cortex-A9 employs a 4-way set associative L1 cache with a configurable size of either 16 KB, 32 KB, or 64 KB and a cache-line size of 32 bytes. Both processors employ a separated L1 cache for data and instructions. The most important difference in terms of cache attacks is the cache-line size.

4.3.2 Coprocessors

Coprocessors extend the functionality of a processor for specific tasks and applications. They are named coprocessors as they are usually integrated into the processor [12]. Currently, up to 16 coprocessors might be implemented for ARM processors. According to [10], only four of these coprocessors are in use by now. While the coprocessors 0–7 are

[2] While [11] refers to this register simply as control register, [10] refers to it as system-control register.

reserved for vendor-specific extensions, coprocessors 8, 9, 12, and 13 are reserved for future implementations by ARM Ltd. The remaining four coprocessors are implemented as follows. Coprocessors 10 and 11 (*CP10* and *CP11*) extend the functionality of the processor for floating-point operations and NEON[3] support. Coprocessor 14 (*CP14*) is used for hardware debugging and coprocessor 15 (*CP15*) provides access to system-control features and performance-monitor registers. Since our cache attacks utilize features of the CP15 exclusively, we strongly focus on this one. CP15 is used to manage cache and MMU configurations as well as for performance-monitoring purposes. Basically, the CP15 is a set of registers, with some registers being accessible in privileged mode only.

The *Cortex-A Series Programmer's Guide* [12] lists five classes of coprocessor instructions: `LDC` and `STC` for moving data between coprocessor registers and memory, `MRC` and `MCR` for moving data between ARM core registers and coprocessor registers, as well as `CDP` for coprocessor-specific instructions. However, we only make use of the `MRC` and `MCR` instruction. While `MCR` is used to pass values from an ARM core register to a coprocessor register, `MRC` is used to pass values from a coprocessor register to an ARM core register. Besides the instruction name, both instructions have the same syntax and are outlined in Listing 4.1. The corresponding parameters are described in Table 4.2. For further details, especially for conditional usage and other encodings, see the *ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition)* [10].

Listing 4.1: MCR and MRC instruction.

```
[MCR|MRC] <coproc>, <opc1>, <Rt>, <CRn>, <CRm>{, <opc2>}
```

Parameter	Description
`coproc`	Specifies the name of the coprocessor [p0, ..., p15].
`opc1`	Coprocessor-specific opcode [0, ..., 7].
`Rt`	Specifies the ARM register.
`CRn`	Specifies the coprocessor register.
`CRm`	Specifies an additional coprocessor register.
`opc2`	Another coprocessor-specific opcode [0, ..., 7]. Default 0.

Table 4.2: MCR and MRC instruction parameters (cf. [10, p A8-187]).

Performance-Monitor Registers

This set of registers is capable of counting clock cycles used for precise and accurate timing measurements. Furthermore, up to 31 different events might be counted and also managing and controlling these counters is possible with this set of registers [10]. Usually, the performance-monitor registers are only accessible from privileged mode, but setting the appropriate bit within the *User Enable Register* allows code executed from user mode to access these registers.

The following paragraphs introduce the performance-monitor registers and the most important bits of these registers required for our attack in the next chapter. For a complete and detailed description of all available performance-monitor registers we refer to [10, pp C10-100] and [11, pp 3-2].

[3]The NEON technology refers to the Advanced *Single Instruction Multiple Data* (SIMD) support.

User Enable Register (PMUSERENR). Though PMUSERENR is a 32-bit register only the first bit is of concern. Bit 0 of this register controls whether code executed from the user mode is allowed to access the performance-monitor registers or not. Therefore, if this bit is set, the registers described in the remainder of this section are accessible from user mode. Of course, this register is writeable in privileged mode only and therefore we wrote a kernel module[4], intended to set this bit. Listing 4.2 shows the corresponding instruction and Table 4.3 lists the one and only bit of concern within this register.

Listing 4.2: Enable PMUSERENR.

```
asm volatile("mcr p15, 0, %0, c9, c14, 0" : /* no output */ : "r"(1));
```

Bit	Value	Description
0	0/1	Disable/enable access to performance monitors from user mode.

Table 4.3: *User Enable Register* (cf. [10, pp C10-110–C10-111]).

Performance Monitor Control Register (PMCR). This register is used to control and manage the performance monitors. Listing 4.3 shows a sample configuration for counting all cycles. Table 4.4 lists the bits of concern.

Listing 4.3: Sample configuration of PMCR.

```
// PMCR bit[0] = 1 Enable all counters.
unsigned int value = 1;

// PMCR bit[1] = 1 Reset all event counters, except the cycle counter.
value |= 2;

// PMCR bit[2] = 1 Reset cycle counter (PMCCNTR).
value |= 4;

// PMCR bit[3] = 0 Count every clock cycle

// Write Performance Monitor Control Register (PMCR)
asm volatile("mcr p15, 0, %0, c9, c12, 0\n" :/* no output */: "r"(value));
```

Count Enable Set Register (PMCNTENSET). This register is used to enable event counters as well as the cycle counter. Note that writing a 0 to any bit of this register does not have any effect. Therefore, disabling counters is done via the *Count Enable Clear Register (PMCNTENCLR)*. Listing 4.4 shows an example of how to enable the *Cycle Count Register*. Table 4.5 describes the most important bits of this register.

Listing 4.4: Enable *Cycle Counter Register*.

```
// PMCNTENSET bit[31] = 1 Enable Cycle Count Register.
unsigned int value = 0x80000000;
asm volatile("mcr p15, 0, %0, c9, c12, 1\n" :/* no output */: "r"(value));
```

[4]Further information about this kernel module can be found in Appendix C.

Bit	Value	Description
0	0/1	Disable/enable all counters (including the cycle counter).
1	1	Reset all counters, except the cycle counter.
2	1	Reset cycle counter.
3	0/1	Count every clock cycle / every 64^{th} cycle.
11–15	-	Holds the number of implemented event counters. 0 indicates the presence of the cycle-count register only. Read-only field.

Table 4.4: *Performance Monitor Control Register* (cf. [10, pp C10-100–C10-102]).

Bit	Value	Description
0–$(N-1)$	1	Enable the corresponding event counter. N refers to the number of available event counters (see PMCR).
31	1	Enable the *Cycle Count Register*.

Table 4.5: *Count Enable Set Register* (cf. [10, pp C10-103–C10-104]).

Count Enable Clear Register (PMCNTENCLR). This register is used to disable the event counters as well as the cycle counter. Again, writing a 0 to any bit of this register does not have any effect. In order to enable counters the *Count Enable Set Register* (PMCNTENSET) is used. Listing 4.5 shows an example of how to disable the *Cycle Count Register*. Table 4.6 describes the most important bits of this register.

Listing 4.5: Disable *Cycle Count Register*.

```
// PMCNTENSET bit[31] = 1 Disable Cycle Count Register.
unsigned int value = 0x80000000;
asm volatile("mcr p15, 0, %0, c9, c12, 2\n" :/* no output */: "r"(value));
```

Bit	Value	Description
0–$(N-1)$	1	Disable the corresponding event counter. N refers to the number of available event counters (see PMCR).
31	1	Disable the *Cycle Count Register*.

Table 4.6: *Count Enable Clear Register* (cf. [10, pp C10-104–C10-105]).

Cycle Count Register (PMCCNTR). A 32-bit register that, if enabled, counts core clock cycles. This register can be configured to count either all core clock cycles or to count every 64^{th} clock cycle. For configuration purposes see register PMCR above. Listing 4.6 shows the instruction to read the *Cycle Count Register*. For further information about this register we refer to [10, p C10-108].

Analyzing the behavior of this register revealed that this timer allows measuring times with a resolution of 1 ns (on devices with a clock frequency of 1 GHz). Measuring the

resolution of the *Cycle Count Register* was done as suggested by Davison [21]. Listing 4.7 shows the general approach for measuring the resolution of a timing method.

On a device with a clock frequency of 1 GHz the counter increments 10^9 times a second. Since the counter is only 32-bits wide it overflows every $\frac{2^{32}}{10^9} \approx 4.3$ seconds.

Listing 4.6: Read *Cycle Count Register*.

```
unsigned int value;
asm volatile("mrc p15, 0, %0, c9, c13, 0\n" : "=r"(value) :/* no input */);
```

Listing 4.7: Code example for measuring the resolution of timing methods.

```
unsigned int start = 0;
unsigned int end = 0;
unsigned long long total = 0;
unsigned int i = 0;
double resolution = 0;

for ( i = 0; i < num_runs; ++i ) {
  start = get_time();           // Replace with whatever method is used
  end = get_time();
  while ( start == end) {
    end = get_time();
  }
  total += (end - start);
}
resolution = total / (double) num_runs;
```

4.4 Intel's Time-Stamp Counter

Related work in the field of cache attacks usually employs Intel's Time-Stamp Counter (TSC) for precise timing measurements. The TSC is available on Intel platforms since the introduction of the Intel Pentium processors [30]. The purpose of this 64-bit register is to count clock cycles and hence this register is incremented on every clock tick. Accessing the time-stamp counter register is done through the `RDTSC` instruction [29]. This instruction loads the lower 32 bits of the time-stamp counter into the EAX register and the upper 32 bits of the time-stamp counter into the EDX register. On 64-bit systems the RAX and RDX registers are used, respectively, whereat the upper 32 bits of the RAX and RDX registers are cleared. Listing 4.8 illustrates the usage of the `RDTSC` instruction on a 64-bit system. The *Intel Software Developer's Manual - Volume 2B* [29] also points out that due to the out-of-order execution on Intel processors it cannot be assured that this instruction is executed in the exact same order as it appears in the source code. Hence, the manual lists the `CPUID` instruction [29, pp 3-198–3-235] as a possible serialization instruction and on newer systems, e.g., Intel Core i7, the `RDTSCP` instruction [29, pp 4-462–4-463] might be used in order to wait until all previous instructions have been executed. However, the `RDTSCP` instruction does not prevent subsequent instructions from being executed but only waits until all previous instructions have been executed. This means that subsequent instructions might distort the timing measurement. For further information on this topic

	CPU frequencies		
# of NOPs	**800 MHz**	**1600 MHz**	**2670 MHz**
16 384	120	60	36
32 768	120	60	36
65 536	120	60	36

Table 4.7: Average execution time of a NOP instruction in clock cycles. The values clearly indicate the impact of the *Intel SpeedStep Technology* on the execution time.

we refer to Paoloni [42], who provides a detailed insight into how to measure cycle times correctly.

Listing 4.8: Usage of the RDTSC instruction on 64-bit systems.

```
unsigned int upper, lower;
asm volatile("RDTSC\n mov %%edx, %0\n mov %%eax, %1\n"
             : "=r"(top), "=r"(bottom)
             : /* no input */
             : "%rax", "%rdx");
```

Another important thing to keep in mind is that modern Intel processors support the *Enhanced Intel SpeedStep Technology* [28], which is used to reduce the power consumption of the CPU. Therefore, the *SpeedStep Technology* simply adjusts the core frequency of the processor in order to meet the current requirements and has an impact on the time-stamp counter, i.e., depending on the actual clock frequency of the CPU the time-stamp counter increases faster or slower. The following experiment clearly demonstrates this behavior. In our test environment, i.e., Lenovo Thinkpad W500, we deactivated the second CPU, i.e., passed the boot paramater `maxcpus=1` to the Linux system at boot time. Afterwards we started a root shell and pinned the frequency of the remaining CPU to one of the supported clock frequencies as shown in Listing 4.9. The second line prevents the operating system from changing the CPU's frequency in the future and the third line scales the CPU to the minimum supported frequency, which in our case is 800 MHz[5]. Next we timed multiple *NOP* operations and computed the mean execution time. Table 4.7 shows the result of this experiment for different CPU frequencies. This experiment clearly indicates that the faster the CPU clock frequency is, the less cycles per instruction are measured.

Listing 4.9: Pin CPU frequency to the minimum frequency supported.

```
# cd /sys/devices/system/cpu/cpu0/cpufreq/
# echo userspace > scaling_governor
# cat cpuinfo_min_freq > scaling_setspeed
```

[5]Though we talk about MHz in this context, the values in the filesystem */sys/devices/system/cpu/cpu*/* are given in kHz.

Chapter 5

Conducted Attacks

Before we cover the details of the conducted attacks in this chapter we have to clarify some basic issues and decisions. First, we briefly mention the basic attack environment and the general project structure. Especially the eviction of cache sets needs some clarification and we also briefly outline the advantage of disaligned T-tables for cache attacks. Finally, we outline the conducted attacks. For each of the conducted attacks we explain the basic concept and the attack scenario. Furthermore, we analyze the applicability of these attacks on the ARM Cortex-A series processor platform.

5.1 Attack Environment

In this thesis we strongly focus on Android smartphones and Android tablet computers equipped with ARM processors. Though, the outlined attacks might also be applicable on other mobile devices as well. Since the implementation of our attacks on the Android platform requires the Android-SDK as well as the Android-NDK, we refer to [8] for the required toolkits themselves and detailed information regarding the installation and configuration of these toolkits. For further information regarding the specification of the devices under attack see Appendix B. Before covering the details of the conducted attacks we outline the basic project structure used to launch the attacks.

Kernel Module. As already mentioned in Section 4.3, access to the cycle-count register is granted solely to privileged applications, except for the case that access is explicitly granted to unprivileged applications. The purpose of this kernel module is to set a specific bit within an ARM control register and hence allows unprivileged applications to access this cycle-count register. Since loading a kernel module requires root access, a rooted mobile device is necessary in order to launch these attacks. For further information about this kernel module see Appendix C.

Attack Module. This component implements the conducted attacks outlined in the following sections and is written in Java and C respectively. The purpose of this module is to trigger the application which performs the encryption and to gather measurement samples. For reasons of simplicity, in our case the attack module triggers the AES encryption function directly. Future work in this field might consider triggering the encryption within an external application in order to simulate a more realistic scenario.

OpenSSL Module. As the name already suggests, this component holds the implementation of the OpenSSL project [41] including the AES implementation. Since the Assembler implementations of the AES are usually optimized for performance and might also implement countermeasures against side-channel attacks, we attack the standard C implementation using T-tables. Currently we launch the attacks against *OpenSSL 0.9.7a*, though newer versions like, for instance, *OpenSSL 1.0.1c* neither resist these attacks since the T-table implementation is always the same.

Android Application. This is the main entry point and might represent a fancy application the user is willing to grant root access. In order to prevent the attack from being stopped after the user ends the application, the attack itself is implemented as a *Service*, i.e., the execution is not bound to the execution of an *Android Activity*. This is of utmost importance for time-driven attacks since these attacks usually take hours in order to gather the required number of measurement samples. Of course, the application itself takes care of loading the kernel module before launching the attack.

Though Android devices are shipped with a *libcrypto.so* library, which can be found under `/system/lib/`, it is not listed as stable API within the NDK and hence it should not be used since it might change in future versions or it might even be removed. This means that *libcrypto.so* is not available directly, e.g., by building and linking the application via `LOCAL_LDLIBS := lcrypto` within the Android build file. Though, since the *Dynamic Linker Library* is available one might use *dlopen()*, *dlsym()*, and *dlclose()* in order to access the implementation of *libcrypto.so* dynamically, thus circumventing the fact that it is not listed as stable API. However, we believe that the more common approach to make use of AES encryptions in native code is to include an AES implementation within the application itself. For the sake of completeness, we have to mention that *SpongyCastle* is the default provider for Java-based AES encryptions in Android. Since the performance of the AES encryption of *SpongyCastle* in comparison to a native T-table implementation is rather worse, software developers might consider using a native T-table implementation like the one of the *OpenSSL* project. Based on these considerations we assume our attack scenario to be fairly realistic.

5.2 Eviction of Cache Sets

The eviction of specific cache sets represents an integral part of cache attacks. Osvik et al. [46] give a brief overview of cache-set evictions. In general this might be accomplished as follows. Suppose we have an L1 data cache with a number of cache sets denoted as $L1\_S$, a cache-line size of $L1\_B$ bytes, and an associativity of $L1\_W$. Furthermore, we denote the overall cache size in bytes as $L1\_Cache\_Size = L1\_S \cdot L1\_W \cdot L1\_B$. For virtual caches, i.e., caches that use the virtual address in order to map data from main memory to the cache, the memory regions are mapped contiguously to the cache. For instance, the first $L1\_B$ bytes map to a specific cache set $\mathbf{s}$, the next $L1\_B$ bytes map to the next cache set $\mathbf{s} + 1$ and so on. Thus, in order to evict a specific cache set from the cache, i.e., replace the data currently stored within the cache set with other data, we allocate an array of bytes which is exactly as large as the L1 cache, i.e., $L1\_Cache\_Size$ bytes. The eviction of a specific cache set is done by accessing at least $L1\_W$ elements, which are exactly $L1\_S \cdot L1\_B$ bytes apart. Since accessing only one byte from a memory block that

corresponds to a single cache line is enough in order to cache the whole memory block we do not need to access every byte within the memory block but only one, e.g., the first one.

Assuming a virtual cache and a deterministic replacement policy the above outlined approach might be used to evict specific cache sets. In contrast, according to [10] ARM processors employ a physically indexed, physically-tagged data cache (PIPT) with a random-replacement policy. These subtle differences cause slight adaptions to the above outlined approach. We address these two problems separately within the next paragraphs. First, we cover the scenario for a physically indexed, physically-tagged cache (PIPT). Finally, we state an approach in order to evict specific cache sets from caches implementing a random-replacement policy.

PIPT Caches. As already outlined in Chapter 2, physical caches use the physical address in order to establish a mapping between memory locations within the main memory and cache sets. In general we are supposed to gain knowledge of the virtual address of data structures easily. However, in case of physical caches knowledge of the physical address might be necessary in order to evict specific parts of the cache. Therefore, the Linux kernel provides the interface `/proc/pid/pagemap`, which might be used for this purpose. For each virtual page it holds the corresponding physical page frame. Given the information that the *Samsung Galaxy SIII* as well as the *Acer Iconia A510* have a page size of 4 KB, a cache-line size of 32 bytes, and 256 cache sets we extract the virtual page number of a 32-bit address as follows. First, we remove the lower $\log_2 4\,096 = 12$ bits, i.e., the page offset bits. We query `/proc/pid/pagemap` at the position denoted by the remaining virtual page number and retrieve the physical page-frame number. Then we append the initial offset bits to the page-frame number again and retrieve the physical address. In order to obtain the corresponding cache set we remove the lower $\log_2 32 = 5$ bits, i.e., the bits used to index elements within the cache line. The lower $\log_2 256 = 8$ bits of the remaining address yield the corresponding cache set for a physical cache.

Another problem of attacking physical caches is the fact that, in general, allocated memory is not contiguous within the physical memory. At least in case the allocated memory spreads across a physical page frame. Thus, allocating a data structure of 32 KB on a system with a page size of 4 KB usually results in 8 different blocks of memory scattered across the physical memory. Hence, accessing elements within this data structure, which are exactly a multiple of $L1\_S \cdot L1\_B$ bytes apart, might not map to the same cache set in a PIPT cache. However, given the knowledge of how to extract the corresponding cache set we are able to access precisely the memory blocks which map to a specific cache set. In other words, given this information we should be able to establish a mapping, such that we know which memory blocks in our data structure must be accessed in order to evict a specific cache set.

Random-Replacement Policy. Another problem regarding the eviction of cache sets might be imposed by the non-deterministic replacement policy of ARM caches. For instance, Figure 5.2 visualizes a cache set consisting of four cache lines. The intention is to ensure the eviction of the cache line which holds parts of a precomputed AES T-table. For the purpose of the following explanation we assume a temporary data structure with a size of $L1\_Cache\_Size$ bytes, such that exactly 4 memory blocks of this data structure map to each of the $L1\_S$ cache sets. For virtual caches such a mapping can be established easily by allocating a contiguous memory region, whereas for physical caches the mapping must be ensured manually.

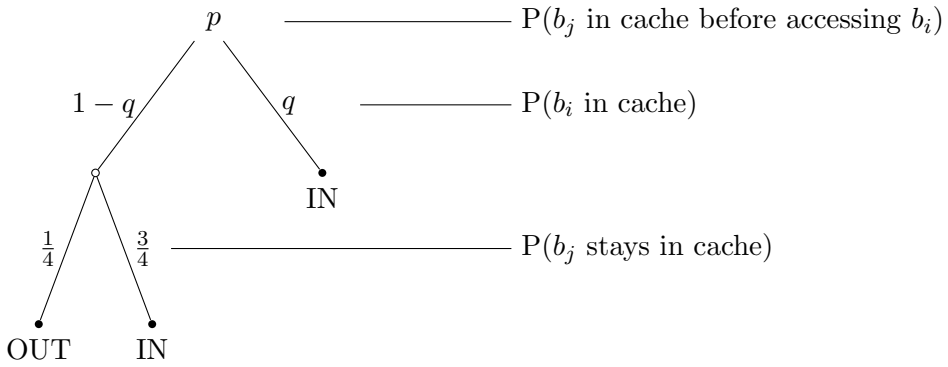


Figure 5.1: Probability tree for a memory block b_j still being cached after re-accessing a memory block b_i of the temporary byte array.

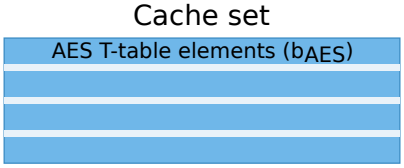


Figure 5.2: Cache set with a memory block (b_{AES}) of a precomputed AES T-table in one line.

On accessing an element within the temporary data structure the random-replacement policy selects an element (cache line) from the set of all possible cache lines $\Omega = \{1, 2, 3, 4\}$ randomly. Equation 5.1 shows the probability that a specific cache line, e.g., the one which holds the memory block of the AES T-table, is still present after accessing the first memory block within the temporary byte array. Thus, the probability of the AES T-table elements still being cached after the whole temporary byte array has been accessed once, i.e., four accesses per cache set, is $\approx 31.64\,\%$. For the purpose of cache attacks this probability is too low. The corresponding probabilities for a cache line still being present after accessing a specific memory block from the temporary byte array once are listed in Table 5.1.

Hence, due to the non-deterministic replacement policy of ARM caches we have two possibilities in order to evict a specific cache set with a certain probability: (1) *access the L1_W memory blocks more than once*, or (2) *allocate a larger block of memory*, e.g., three times the cache size. In terms of probability theory the second approach is considered to be more efficient. This results from the fact that the second access to a specific memory block b_i might not manipulate the cache at all, i.e., the accessed memory block is still located within the cache. In order to clarify this observation Figure 5.1 visualizes the computation of the probability that a specific memory block b_j stays in cache after accessing a memory block b_i the second time. Given this probability tree, Equation 5.2 formalizes the corresponding probability for a specific block of memory b_j still being in cache after accessing another block of memory b_i of the temporary byte array.

Iteration	**Memory block (b_i) accessed**	**Probability**
1	b_1	$\mathrm{P}(b_{AES} \text{ in cache}) = \frac{3}{4}$ $\mathrm{P}(b_1 \text{ in cache}) = 1$
1	b_2	$\mathrm{P}(b_{AES} \text{ in cache}) = \left(\frac{3}{4}\right)^2$ $\mathrm{P}(b_1 \text{ in cache}) = \left(\frac{3}{4}\right)$ $\mathrm{P}(b_2 \text{ in cache}) = 1$
1	b_3	$\mathrm{P}(b_{AES} \text{ in cache}) = \left(\frac{3}{4}\right)^3$ $\mathrm{P}(b_1 \text{ in cache}) = \left(\frac{3}{4}\right)^2$ $\mathrm{P}(b_2 \text{ in cache}) = \left(\frac{3}{4}\right)$ $\mathrm{P}(b_3 \text{ in cache}) = 1$
1	b_4	$\mathrm{P}(b_{AES} \text{ in cache}) = \left(\frac{3}{4}\right)^4$ $\mathrm{P}(b_1 \text{ in cache}) = \left(\frac{3}{4}\right)^3$ $\mathrm{P}(b_2 \text{ in cache}) = \left(\frac{3}{4}\right)^2$ $\mathrm{P}(b_3 \text{ in cache}) = \left(\frac{3}{4}\right)$ $\mathrm{P}(b_4 \text{ in cache}) = 1$

Table 5.1: Probability for any memory block b_j still being present after accessing a specific memory block b_i.

$$
\begin{aligned}
&\mathrm{A} := \textit{Line 1 still in cache after the first memory access} \\
&\mathrm{B} := \textit{Line 1 still in cache after the fourth memory access} \\
&P(\mathrm{A}) = \frac{|\{2,3,4\}|}{|\Omega|} = \frac{3}{4} \\
&P(\mathrm{B}) = P(\mathrm{A})^4 = \left(\frac{3}{4}\right)^4 \approx 0.3164
\end{aligned}
\tag{5.1}
$$

$$
\begin{aligned}
&A := \textit{Memory block } b_j \textit{ in cache before accessing memory block } b_i \\
&B := \textit{Memory block } b_i \textit{ already in cache} \\
&C := \textit{Memory block } b_j \textit{ still in cache after accessing memory block } b_i \\
&P(A) = p \\
&P(B) = q \\
&P(C) = p\left[q + (1-q)\,\frac{3}{4}\right]
\end{aligned}
\tag{5.2}
$$

Equation 5.3 states, for example, the corresponding probability for the AES block b_{AES} still being in cache after accessing the memory block b_1 of the temporary byte array within the second iteration.

$$
\begin{aligned}
&A := \textit{Memory block } b_{AES} \textit{ in cache before accessing element } b_1 \textit{ the second time} \\
&B := \textit{Memory block } b_1 \textit{ already in cache} \\
&C := \textit{Memory block } b_{AES} \textit{ still in cache after accessing memory block } b_1 \\
P(A) &= p = \left(\frac{3}{4}\right)^4 \\
P(B) &= q = \left(\frac{3}{4}\right)^3 \\
P(C) &= p\left[q + (1-q)\frac{3}{4}\right] \\
&= \left(\frac{3}{4}\right)^4 \left[\left(\frac{3}{4}\right)^3 + \left(1 - \left(\frac{3}{4}\right)^3\right)\frac{3}{4}\right] \\
&\approx 0.2707
\end{aligned}
\tag{5.3}
$$

Table 5.2 lists the corresponding probabilities for a specific memory block b_j still being in cache after accessing a memory block b_i for up to three iterations. The probability for the memory block of the AES T-table still being cached after accessing all four memory blocks of the temporary byte array mapping to this cache set, is about $\approx 15.17\,\%$, for three iterations.

Algorithm 2 Eviction of a specific cache set in virtual caches.

Input: *set* - the set to be evicted, *data* - array three times the size of the L1 cache

```
for i = 0 to (3 · L1_W) − 1 do
    addr ← (L1_B · set) + (L1_S · L1_B · i)
    read_and_write_operation(data[addr])
end for
```

If the contiguous allocation of main memory can be ensured, i.e., for virtual caches, one might use a temporary byte array with three times the cache size. In this case elements of this array do not have to be accessed multiple times and therefore the probability for a specific line still being present in cache after 12 different memory accesses can be reduced to $\left(\frac{3}{4}\right)^{12} \approx 0.0316$. Thus, for the same number of memory accesses this approach yields a better probability, i.e., is far more efficient. Algorithm 2 states the appropriate algorithm for evicting a specific cache set with a probability of $(100 - 3.16) = 96.84\,\%$. In order to prevent the compiler from optimizing away the memory access in case the accessed element, e.g., $data[addr]$, is not used we also perform a write operation at the given address.

Intricacies on the ARM Cortex-A Series Processors

Since many official ARM documents [10, 11, 12] state that the ARM Cortex-A8 employs a PIPT cache we implemented the eviction according to the considerations mentioned above. Unfortunately, experiments did not yield appropriate results. For instance, the localization of AES T-tables as it will be described in Section 5.4 did not yield the correct position. Thus, we tried the approach considering a virtually-indexed data cache. Indeed, with this approach we were able to establish a mapping between memory blocks within a

Iteration	**Memory block b_i accessed**	**Probability**
2	b_1	$P(b_{AES}$ in cache$) = 0.2707$ $P(b_1$ in cache$) = 1$ $P(b_2$ in cache$) = 0.4812$ $P(b_3$ in cache$) = 0.6416$ $P(b_4$ in cache$) = 0.8555$
2	b_2	$P(b_{AES}$ in cache$) = 0.2356$ $P(b_1$ in cache$) = 0.8703$ $P(b_2$ in cache$) = 1$ $P(b_3$ in cache$) = 0.5584$ $P(b_4$ in cache$) = 0.7445$
2	b_3	$P(b_{AES}$ in cache$) = 0.2096$ $P(b_1$ in cache$) = 0.7742$ $P(b_2$ in cache$) = 0.8896$ $P(b_3$ in cache$) = 1$ $P(b_4$ in cache$) = 0.6623$
2	b_4	$P(b_{AES}$ in cache$) = 0.1919$ $P(b_1$ in cache$) = 0.7089$ $P(b_2$ in cache$) = 0.8145$ $P(b_3$ in cache$) = 0.9156$ $P(b_4$ in cache$) = 1$
3	b_1	$P(b_{AES}$ in cache$) = 0.1779$ $P(b_1$ in cache$) = 1$ $P(b_2$ in cache$) = 0.7552$ $P(b_3$ in cache$) = 0.8489$ $P(b_4$ in cache$) = 0.9272$
3	b_2	$P(b_{AES}$ in cache$) = 0.1670$ $P(b_1$ in cache$) = 0.9388$ $P(b_2$ in cache$) = 1$ $P(b_3$ in cache$) = 0.7970$ $P(b_4$ in cache$) = 0.8705$
3	b_3	$P(b_{AES}$ in cache$) = 0.1585$ $P(b_1$ in cache$) = 0.8912$ $P(b_2$ in cache$) = 0.9492$ $P(b_3$ in cache$) = 1$ $P(b_4$ in cache$) = 0.8263$
3	b_4	$P(b_{AES}$ in cache$) = 0.1517$

Table 5.2: Probabilities for a specific memory block still being cached after accessing a specific memory block b_i.

temporary data structure and the data cache, such that we know which memory blocks must be accessed in order to evict a specific cache set. The following observations harden our assumption that the Cortex-A8 in fact uses a virtually-indexed data cache.

Kernel-Message Buffer. An investigation of the `dmesg` output on the *Google Nexus S* indicated that the Linux kernel on this device considers the data cache as being a VIPT cache. The corresponding output states the following: "*CPU: VIPT nonaliasing data cache, VIPT nonaliasing instruction cache*".

Literature Research. The *ARM Architecture Reference Manual* [10, p B3-22] states that implementations of the ARMv7 architecture might use a VIPT data cache if hardware alias avoidance is supported. Furthermore, Grisenthwaite [23] (an employee at ARM Ltd.) also states that the Cortex-A8 uses a VIPT data cache with alias detection.

Concluding the investigation of cache-set evictions on the ARM architecture we point out that further analysis might be necessary in order to determine whether the ARM Cortex-A8 uses a PIPT or a VIPT data cache. However, for the purpose of cache attacks we are able to establish a correct mapping between our temporary data structure and cache sets, which allows us to evict specific cache sets.

5.3 Aligned and Disaligned Tables

In order to state the difference between aligned and disaligned AES T-tables we use the following definitions.

Definition 1. *We denote the maximum number of T-table elements per cache line as δ. Since we consider systems with 4-byte integers, we define: $\delta = \frac{L1\_B}{4}$.*

Definition 2. *We denote the number of cache sets a T-table is supposed to take as γ. Since a T-table has 256 elements we define: $\gamma = \frac{256}{\delta}$ for aligned T-tables and $\gamma = \frac{256}{\delta} + 1$ in case of disaligned T-tables.*

Due to reasons of security, each AES T-table should start at a memory address which is mapped to the beginning of a cache line. Equation 5.4 outlines the optimal placement of an AES T-table $\mathbf{T}$ in memory. In this case the first cache set related to a specific AES T-table holds exactly the first δ table elements, which is the optimum. In general this means that an attacker cannot distinguish between any accessed element of this cache line. Thus, for first-round attacks the theoretical number of recoverable key bits per key byte is limited to the upper $8 - \lceil \log_2 \delta \rceil$ bits. In case of $\delta = 16$ only half of the key bits might be recovered, i.e., 64 bits of the total 128 bits. However, in practice we observe disaligned tables. This means that the address of the precomputed look-up table does not correspond to the start of a cache line, i.e., Equation 5.4 does not hold. Hence, the first cache set related to a specific T-table holds less than δ table elements. This in turn means that the number of indistinguishable key bits per key byte gets smaller and hence more key bits can be recovered. The investigated mobile devices revealed that even a disalignment of $\delta - 1$ is possible, i.e., the first cache set holds only one element, which reveals the whole secret key without a subsequent brute-force attack. Especially the *cache-access pattern attack* mentioned below exploits this fact ruthlessly.

$$address(\mathbf{T}[0]) \equiv 0 \mod L1\_B \tag{5.4}$$

5.4 Cache-Access Pattern Attack

As the name already suggests, this attack belongs to the class of access-driven attacks. Based on the work of Osvik et al. [46] we developed an attack which employs a pattern-matching approach. The intention of this attack is to extract the memory-access patterns of the actual AES implementation and match them against precomputed access patterns. In addition, we state an enhancement which might further reduce the remaining key space in case the T-tables are not properly aligned and thus leak further information about the key bytes.

In case of the AES the precomputed look-up tables (T-tables) consist of 256 integer values, each four bytes wide. Hence, each of these T-tables takes up 1 KB of space in memory. The exact number of cache sets required to cache such a T-table depends on the cache-line size and whether the T-table is properly aligned or not. For instance, on a system with a cache-line size of 64 bytes a T-table takes up 16 consecutive cache sets, whereas on systems with a cache-line size of 32 bytes it takes up 32 consecutive cache sets. We denote the number of cache sets a T-table is supposed to consume as γ. Hence, on systems with a cache-line size of 64 bytes $\gamma = 16$, and on systems with a cache-line size of 32 bytes $\gamma = 32$. In addition, if the T-table is disaligned then γ increases by one.

Observe that the look-up indices within the first round of the AES are computed as $\mathbf{s}_i = \mathbf{p}_i \oplus \mathbf{k}_i$. Furthermore, on a system with a cache-line size of 64 bytes the first cache set contains the corresponding table elements of the look-up indices $\texttt{0x00} \leq \mathbf{s}_i \leq \texttt{0x0F}$, i.e., the first 16 elements of the T-table. Thus, given information about the accessed cache set and hence information about the accessed look-up index $\mathbf{s}_i$ and the corresponding plaintext byte $\mathbf{p}_i$ yields the upper four bits of the secret key $\mathbf{k}_i$.

The scenario for the following cache-access pattern attack is as follows. We are able to trigger AES encryptions with an unknown but fixed key and a chosen plaintext. Though the attack might also work for equally-distributed random plaintexts. Furthermore, we assume to be able to measure the encryption time with a sufficient-high resolution and to find an appropriate way that allows us to determine the memory-access patterns. Osvik et al. [46] suggest two different approaches in order to gather the memory-access patterns: (1) *Prime and Probe*, and (2) *Evict and Time*. We briefly outline these two approaches in the following paragraphs.

Prime and Probe

The basic idea of this approach is to observe cache evictions through a data structure within the attacker's control. Therefore, the attacker allocates a data structure with the same size of the L1 data cache and loads this data structure into the cache, i.e., by accessing elements within this data structure. After the whole L1 data cache has been initialized the attacker triggers the encryption of a plaintext. Obviously, the encryption evicts parts of the initially loaded data structure. Hence, by precisely measuring the memory-access times of the data structure loaded before triggering the encryption the attacker determines which memory blocks have been evicted by the encryption. Assuming the attacker also knows the start address of the first T-table and that T-tables are mapped contiguously into the L1 data cache the attacker is able to deduce information about the accessed cache sets.

Though Osvik et al. [46] state that this approach is extremely efficient, since the attacker only measures a simple memory access within the memory space of the attack

process, this approach is not applicable on processors with a random replacement policy. This non-deterministic approach evicts cache lines within a cache set randomly. Thus, we are not able to populate the L1 data cache with a data array of the same size. For instance, suppose we have a 4-way set-associative cache with 128 cache sets and a cache-line size of 64 bytes. The probability to initialize all four cache lines in one cache set with four memory accesses is given in Equation 5.5. As Equation 5.6 indicates, taking all cache sets into consideration it is impossible to load the whole data cache with the required data structure.

$$\begin{aligned} P(\text{Set initialized}) &= \frac{4}{4} \cdot \frac{3}{4} \cdot \frac{2}{4} \cdot \frac{1}{4} \\ &= 0.09375 \end{aligned} \tag{5.5}$$

$$\begin{aligned} P(\text{Cache initialized}) &= P(Set)^{128} \\ &= 0.09375^{128} \\ &\approx 2.584 \cdot 10^{-132} \\ &\approx 0 \end{aligned} \tag{5.6}$$

Evict and Time

The idea of this approach is to observe cache evictions based on the encryption time itself. Again, the attacker allocates a data structure which is as large as the L1 data cache. However, this time the attack proceeds as follows. After triggering the encryption of a plaintext **p**, the attacker evicts a specific cache set. Finally, by measuring the encryption time of the same plaintext **p** again the attacker might determine whether a cache set required for the encryption of plaintext **p** has been evicted or not. Since this approach seems to be feasible on processors that implement a random-replacement policy we focus on the *Evict and Time* approach.

5.4.1 Attack

The attack is composed of four steps: (1) *locate the AES T-tables*, (2) *gather cache-access patterns*, (3) *compute possible cache-access patterns*, and (4) *extract the secret key by pattern matching*. These phases are outlined in the following paragraphs.

Locate the AES T-tables. The purpose of this phase is to determine the location of the precomputed AES look-up tables. More formally, the intention is to establish a mapping which allows us to determine which specific memory block of a temporary data array must be accessed in order to evict a specific cache set. Within the next phase such a mapping is necessary in order to evict specific parts of the precomputed AES T-tables.

The localization of the AES T-tables is based on the above mentioned *Evict and Time* approach and works as follows. A random plaintext is encrypted three times. The first encryption ensures that the required instructions and the required data are loaded into the instruction cache and the data cache, respectively. Subsequent to this *warm-up phase* the time of the second encryption of the same plaintext will be measured. Afterwards, the data of a specific cache set will be evicted. This

means that we replace the data of a cache set, which might contain data from a precomputed look-up table, with other data and measure the encryption time of the same plaintext again. Based on these two timing measurements we formulate the following hypothesis: *If the encryption time after the eviction of a specific cache set is slower than before, we assume that this cache set contained data of a precomputed AES look-up table.* In order to overcome the random-replacement policy as well as noise of the attack application itself and other applications running in parallel on the same system, we perform this time comparison N times. The comparison might be based on the average encryption time, however, our experiments showed that the comparison of the absolute timing measurements in each run yields the best results, i.e., we compare the encryption time before the eviction of a specific cache set with the encryption time afterwards and treat this as our measurement score. This measurement score allows us to determine which cache sets potentially hold parts of the precomputed AES T-tables. Since we assume the T-tables to be located contiguously within the cache we simply search for the longest sequence of cache sets where the encryption time increased.

Gather Cache-Access Patterns. In this phase the attacker gathers the memory-access patterns. Therefore, each key byte $\mathbf{k}_i$ is attacked separately by considering the following steps. We set $\mathbf{p}_i = \texttt{0x00}$, choose the rest of the plaintext randomly, and perform the following steps in order to gather the measurement samples. First, we encrypt the chosen plaintext $\mathbf{p}$ in order to load the necessary T-table elements into the data cache and the corresponding instructions into the instruction cache. We refer to this first step as *warm-up step*. Second, we encrypt the plaintext $\mathbf{p}$ again and this time we measure the encryption time. Afterwards, we evict a specific cache set s where the corresponding T-table $\mathbf{T}_j$ of the attacked key byte $\mathbf{k}_i$ resides. Recall, that $i \equiv j \mod 4$. Subsequent to this eviction we measure the encryption time of the same plaintext $\mathbf{p}$ again. Hence, the second measurement provides some kind of measurement score, of which we keep track of in a data structure $\mathbf{t}_i[b][s]$, with $b \in \{\texttt{0x00}, ..., \texttt{0xFF}\}$ representing all possible values $\mathbf{p}_i$ might take, and $s \in \{0, ..., \gamma\}$[1] representing the evicted cache set of Table $\mathbf{T}_j$. In order to eliminate noise and to retrieve stable measurement results the encryption of random plaintexts with $\mathbf{p}_i = \texttt{0x00}$ and the eviction of a specific cache set s is performed R times. Afterwards, we advance to the next possible byte value $\mathbf{p}_i = \texttt{0x01}$ and perform the same steps again, until we finally reach $\mathbf{p}_i = \texttt{0xFF}$. Algorithm 3 outlines the approach more formally.

In other words, for each possible plaintext byte $\mathbf{p}_{i|i\in\{0,...,15\}} \in \{\texttt{0x00}, ..., \texttt{0xFF}\}$ of the plaintext $\mathbf{p}$ we establish a data structure $\mathbf{t}_i[b][s]$. The purpose of this data structure is to illustrate for which specific plaintext bytes $\mathbf{p}_i = b$ the performance decreases after evicting a specific cache set s. There might be multiple different values to be used as a measurement score. In our case we retrieved stable measurement results, and hence distinctive access patterns, by simply comparing the encryption time of the second encryption with the encryption time of the third encryption. Thus, $\mathbf{t}_i[b][s]$ simply counts the number of encryptions where the performance decreases.

Figure 5.3 illustrates an example of such a data structure after performing the above steps for a specific key byte, e.g., $\mathbf{k}_5$ in this case. We gathered this information

[1] Note that in case the T-tables are aligned s is limited by $\gamma - 1$.

Algorithm 3 Gather cache-access patterns for an aligned T-table.

Input: The byte i to be attacked.
Output: $\mathbf{t}_i[b][s]$

```
t_i[b][s] ← 0
for b = 0 to 255 do
  for s = 0 to γ − 1 do
    for r = 0 to R − 1 do
      p ←random()
      p_i ← b
      t_1 ←measure(AES_k(p))
      t_2 ←measure(AES_k(p))
      evict_set(s)
      t_3 ←measure(AES_k(p))
      if t_2 < t_3 then
        t_i[b][s] ← t_i[b][s] + 1
      end if
    end for
  end for
end for
```

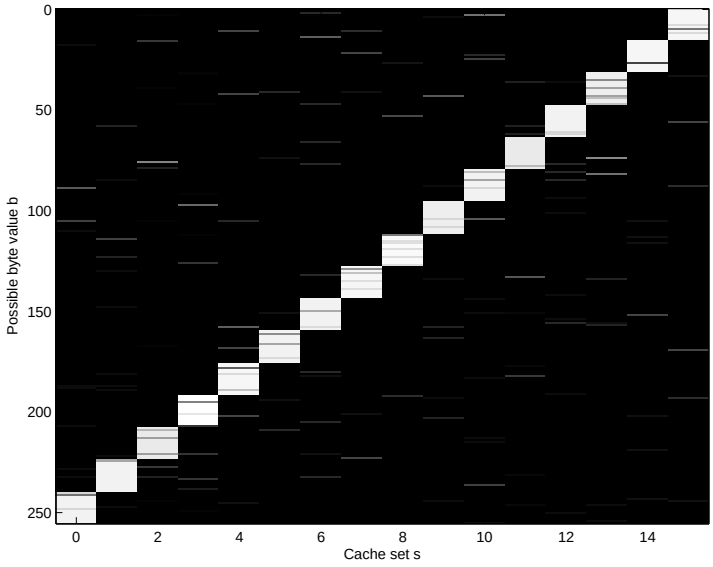


Figure 5.3: Plot of all possible plaintext bytes, encrypted with $\mathbf{k}_5 = \texttt{0xF3}$ for T-table $\mathbf{T}_1$. Brighter areas represent slower encryptions after evicting the corresponding set.

on the *Google Nexus S*. The vertical axis shows all possible plaintext bytes for $\mathbf{p}_5$, and the horizontal axis shows the evicted cache set s. One clearly observes the visible pattern which will be exploited in the following two phases. The pattern can be interpreted as follows. After the eviction of cache set 0 (relating to $\mathbf{T}_1$) the encryption performed slower for plaintext bytes $\mathbf{p}_5 \in \{\texttt{0xF0}, \ldots, \texttt{0xFF}\}$.

We explain the occurrence of this pattern as follows. Recall, that the index used to access a precomputed T-table element $\mathbf{s}_i = \mathbf{p}_i \oplus \mathbf{k}_i$ is composed of 8 bits. In this case we assume a cache-line size of 64 bytes and thus the lower $\log_2 \delta = \log_2 \frac{64}{4} = 4$ bits determine the T-table element within a cache line. The remaining upper $8 - \log_2 \delta = 4$ bits determine the cache set of the resulting index. Table 5.3 outlines the occurrence of this pattern for a key byte $\mathbf{k}_i = \texttt{0xF3}$. The column *Set* represents the upper 4 bits of the resulting index. Given a plaintext byte $\mathbf{p}_i \in \{\texttt{0xF0}, \ldots, \texttt{0xFF}\}$, and the key $\mathbf{k}_i = \texttt{0xF3}$ the resulting look-up indices map into cache set 0. For aligned tables there exist 16 unique patterns, which can be used to reduce the initial key space from 128 bits to 64 bits. In this case the plot in Figure 5.3 only reveals the upper 4 bits of the key byte, e.g., $\mathbf{k}_i \in \{\texttt{0xF0}, \ldots, \texttt{0xFF}\}$.

$\mathbf{p}_i$	$\mathbf{p}_i \oplus \mathbf{k}_i$	**Set**	**Index**
0x00	0xF3	1111	0011
⋮	⋮	⋮	⋮
0xF0	0x03	0000	0011
0xF1	0x02	0000	0010
0xF2	0x01	0000	0001
0xF3	0x00	0000	0000
0xF4	0x07	0000	0111
0xF5	0x06	0000	0110
0xF6	0x05	0000	0101
0xF7	0x04	0000	0100
0xF8	0x0B	0000	1011
0xF9	0x0A	0000	1010
0xFA	0x09	0000	1001
0xFB	0x08	0000	1000
0xFC	0x0F	0000	1111
0xFD	0x0E	0000	1110
0xFE	0x0D	0000	1101
0xFF	0x0C	0000	1100

Table 5.3: Computed look-up indices for all possible bytes and a fixed key byte $\mathbf{k}_i = \texttt{0xF3}$.

Compute Possible Cache-Access Patterns. As we have seen in the previous phase the output of the online phase clearly reveals a visible memory-access pattern. In order to exploit the information leaked through these access patterns we use the following approach. For a hypothetical key byte $\mathbf{h} \in \{\texttt{0x00}, \ldots, \texttt{0xFF}\}$, a number of T-table elements per cache line δ, and a disalignment $\mathbf{d} \in \{0, \ldots, \delta - 1\}$ we are able to compute the memory-access patterns within a specific set. Recall that the lower $\log_2 \delta$ bits are used as index bits, i.e., these bits determine the element within the cache line. The remaining upper bits are the set bits, and hence determine the

cache set which holds the corresponding T-table element. For instance, if the cache line holds $\delta = 16$ table elements, the lower $\log_2 16 = 4$ bits are used as index bits and the 4 remaining upper bits determine the cache set.

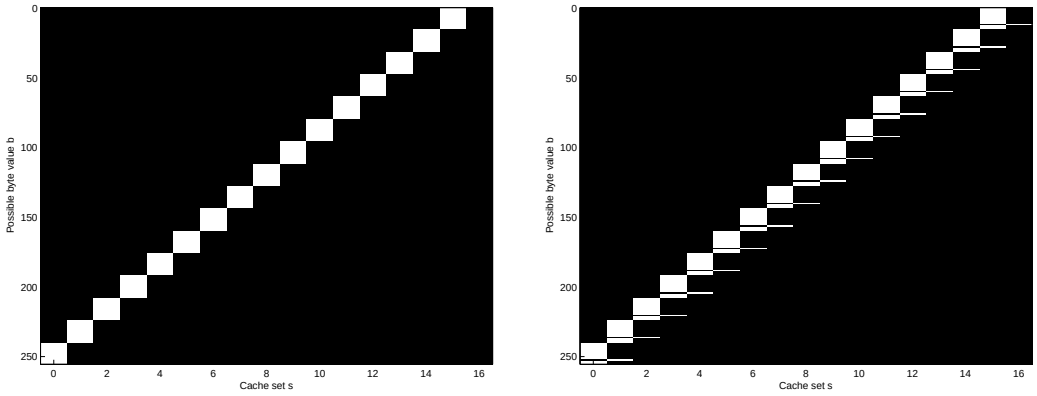
Given this information, Equation 5.7 formalizes the computation of the memory-access patterns for a specific key hypothesis $\mathbf{h} \in \{\texttt{0x00}, \ldots, \texttt{0xFF}\}$, a specific disalignment $\mathbf{d} \in \{0, \ldots, \delta - 1\}$, all possible plaintext byte values $b \in \{\texttt{0x00}, \ldots, \texttt{0xFF}\}$, and all possible cache sets $s \in \{0, \ldots, \gamma\}$. Note that the set s refers to the relative set number of the T-table $\mathbf{T}_j$ and does not represent the absolute number of a set within the cache.

$$\begin{aligned} & b \in \{\texttt{0x00}, ..., \texttt{0xFF}\} \\ & s \in \{0, ..., \gamma\} \\ \text{pattern}_{\mathbf{h},\mathbf{d}}\,[b]\,[s] = & \begin{cases} 1, & \text{iif shift\_right}(\,(b \oplus \mathbf{h}) + \mathbf{d},\ \log_2 \delta) = s \\ 0, & \text{otherwise} \end{cases} \end{aligned} \tag{5.7}$$

Considering all possible disalignments and all possible byte values we deduce that there are $\delta \cdot 256$ patterns per cache set. For $\delta = 16$ this results in 4 096 possible patterns per cache set. However, further analysis revealed that there exist pairs of hypothetical key bytes and disalignments $(\mathbf{h}, \mathbf{d})$ that produce the same cache-access pattern within a specific set. Thus, we found out that there are only 1 376 unique patterns for a cache set. Hence, the patterns are stored in a way, such that a pattern maps to a list of pairs $(\mathbf{h}, \mathbf{d})$ that generate this specific pattern.

Figure 5.4 illustrates the plots of the computed patterns corresponding to the hypothetical key $\mathbf{h} = \texttt{0xF3}$ and two different disalignments $\mathbf{d}$ as outlined in Equation 5.7. For visualization purposes we computed the pattern for all cache sets a specific T-table can take. For the actual attack one might even recover the secret key by computing the patterns for only one cache set. Figure 5.4(a) visualizes the generated pattern for an aligned T-table. In this case the T-table consumes exactly 16 cache sets and each cache set holds 16 table elements. In contrast, Figure 5.4(b) visualizes a generated pattern for a disaligned T-table. The first cache set, i.e., cache set 0, holds only 15 table elements, indicated by a small gap at byte `0xFC`. Consequently, 16 cache sets are not enough in order to hold all table elements and hence 17 cache sets are required. Though cache set 16 only holds the last table element with the index $\mathbf{s}_i = \texttt{0xFF}$, indicated at byte `0x0C`. An interesting property of the single element within cache set 17 is, that $\texttt{0xFF} \oplus \texttt{0x0C} = \texttt{0xF3}$ and thus yields the correct key byte immediately. We will exploit this specific property later.

Note that in case of a smaller cache-line size, e.g., 32 bytes, the T-table elements spread across $\gamma = 32$ cache sets in case of an aligned T-table and across $\gamma = 33$ cache sets in case of a disaligned T-table. This results from the fact that less T-table elements can be located within one cache line. Furthermore, in case of smaller cache-line sizes the number of recoverable bits per key byte increases. This means that for a cache-line size of 32 bytes the number of recoverable bits per key byte is $8 - \log_2 \delta = 5$. Figure 5.5 illustrates the computed pattern for a hypothetical key byte $\mathbf{h} = \texttt{0xF3}$ and two different disalignments $\mathbf{d}$ assuming a cache-line size of 32 bytes.



(a) Generated pattern for $\mathbf{h} = \texttt{0xF3}$ and disalignment $\mathbf{d} = 0$.

(b) Generated pattern for $\mathbf{h} = \texttt{0xF3}$ and disalignment $\mathbf{d} = 1$.

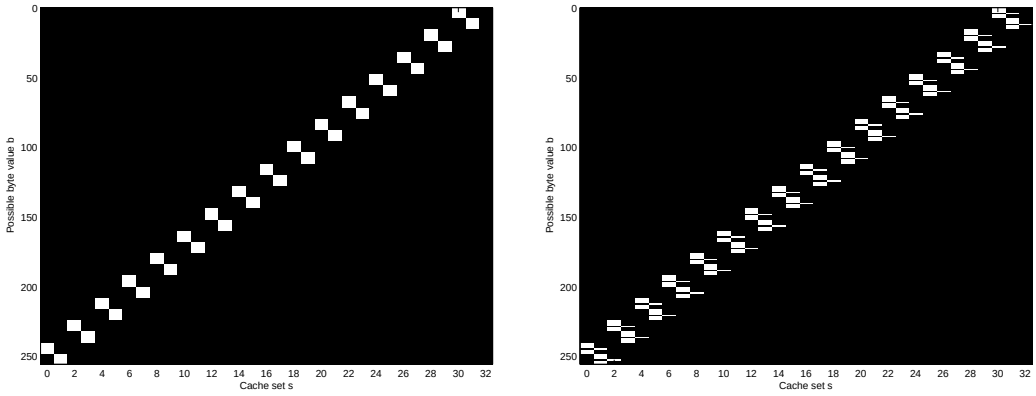
Figure 5.4: Generated pattern for $\mathbf{h} = \texttt{0xF3}$ and two different disalignments for a cache-line size of 64 bytes.

Extract the Secret Key by Pattern Matching. From the measurement scores $\mathbf{t}_i[b][s]$ gathered in the online phase we extract a pattern vector. Therefore, we compute the $mean_s$ and the standard deviation std_s of each cache set s, i.e., of the columns in $\mathbf{t}_i[b][s]$. We apply the following empirically detected threshold: If the measurement score of a specific byte value b within a specific cache set s is greater than the $mean_s$ plus the standard deviation std_s, we assume that this byte value has been accessed during the encryption. Consequently, if the measurement score is below this threshold, we assume that this byte value has not been accessed during the encryption. Equation 5.8 outlines the approach to extract a pattern vector for a specific cache set s, and an attacked byte i.

$$\text{pattern\_vector}_i[b][s] = \begin{cases} 1, & \text{iif } \mathbf{t}_i[b][s] > mean_s + std_s \\ 0, & \text{otherwise.} \end{cases} \tag{5.8}$$

With this pattern vector we are able to query the data structure generated in the previous phase. Actually, we compute the element-wise product of this pattern vector for a specific cache set s with all possible pattern vectors computed for the same cache set s. If the result matches the computed pattern again, we retrieve a list of possible key candidates and disalignments $(\mathbf{h}, \mathbf{d})$ that yield this specific pattern. Furthermore, we save the key candidates and disalignments $(\mathbf{h}, \mathbf{d})$ with the greatest compliance according to the compared pattern vectors. In the rare case that no exact match was found, we simply treat the result with the greatest compliance as possible key candidates.

A possibly noisy pattern might hinder the extraction of the pattern within a single cache set. Thus, we extract the pattern of multiple sets and match them against the precomputed patterns. We count the number of cache sets that consider $\mathbf{h}$ as a possible key candidate. The more cache sets report a possible key candidate $\mathbf{h}$ the more likely it might be the real key byte. If the access patterns are clearly visible among the first set, an attacker might also consider exploiting only the access



(a) Generated pattern for $\mathbf{h} = \texttt{0xF3}$ and disalignment $\mathbf{d} = 0$.

(b) Generated pattern for $\mathbf{h} = \texttt{0xF3}$ and disalignment $\mathbf{d} = 1$.

Figure 5.5: Generated pattern for $\mathbf{h} = \texttt{0xF3}$ and two different disalignments for a cache-line size of 32 bytes.

pattern of the first cache set. The above mentioned approach has the advantage that it works even in case that some of the cache sets do not provide a specific pattern.

Another interesting property is that the generated patterns for a specific key-byte hypothesis $\mathbf{h}$ and a specific disalignment $\mathbf{d}$ are equal in all odd sets s, besides some offset. This might be of value if one considers the exploitation of all odd sets. In this case the pattern for the first set is computed as outlined above and the remaining odd patterns might be generated by shifting the computed pattern.

Enhancement

Further analysis of the extracted memory-access patterns revealed that the generated patterns leak even more information, at least in case of disaligned T-tables. Recall that the lookup indices into the T-tables are computed as $\mathbf{s}_i = \mathbf{p}_i \oplus \mathbf{k}_i$ in case of the first round. Hence, if the upper $8 - \log_2 \delta$ bits of the encrypted plaintext byte $\mathbf{p}_i$ equal the upper $8 - \log_2 \delta$ bits of the secret key $\mathbf{k}_i$, the resulting look-up index goes straight into the first cache set related to T-table $\mathbf{T}_j$, with $i \equiv j \mod 4$. Thus, the resulting index will be visualized within cache set 0, at least in case where noise does not pollute these cache accesses. Unfortunately, we cannot determine which plaintext byte $\mathbf{p}_i$ equals the unknown secret key byte $\mathbf{k}_i$, unless there is only one table element within cache set 0.

The crucial observation, that allows further reduction of the remaining key space is that the correct key byte is always within the largest block of the first set as well as the largest block of the last set. We exploit this fact and extract the possible key candidates from the smaller one of these two, i.e., the block that holds fewer possible key candidates. In case we extract the key candidates from the last set we have to compute the XOR with `0xFF` from each of the candidates in order to invert all bits. In order to clarify the observation that the larger block always contains the correct key byte we denote α as the number of table elements within the first cache set. Thus, in case of disaligned T-tables we always have $\alpha < \delta$. Starting by $\alpha = 1$ and increasing it continuously leads to a change within the lower $\lceil \log_2 \alpha \rceil$ bits, with the remaining upper bits staying constant. Thus, considering only the upper $8 - \lceil \log_2 \alpha \rceil$ bits these lookup indices $s_i \in \{0, \ldots, \alpha - 1\}$ form

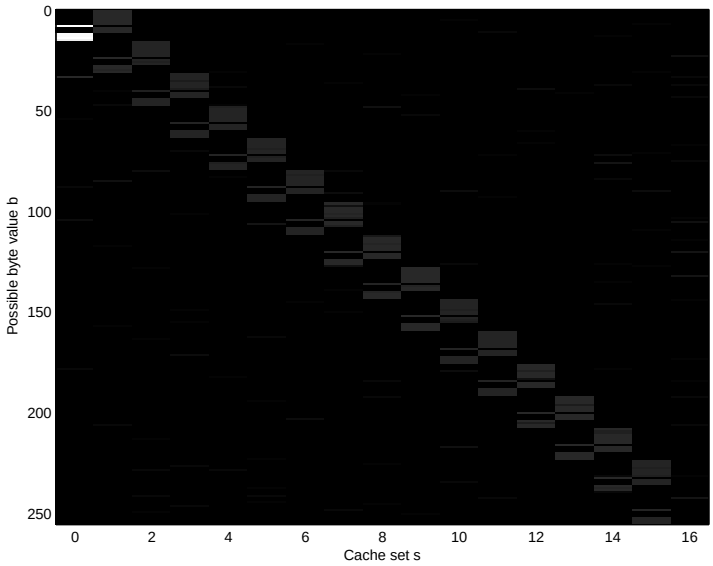
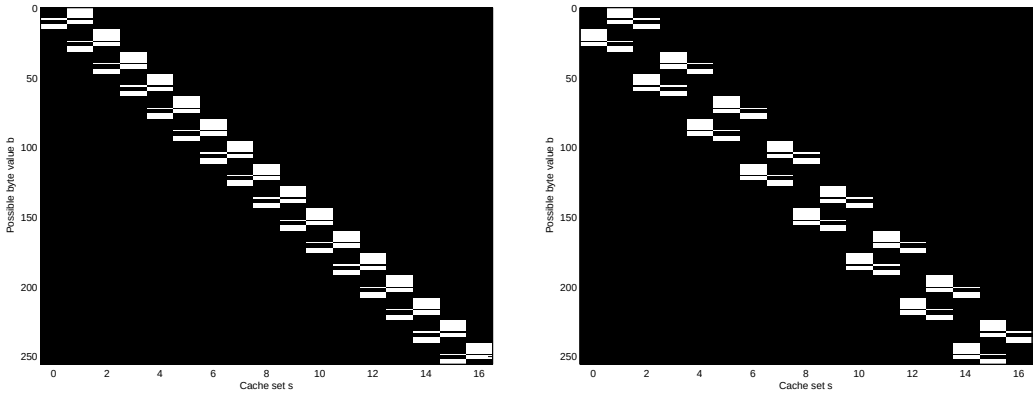


Figure 5.6: Gathered cache-access pattern on the Google Nexus S with $R = 150$ iterations for key byte $\mathbf{k}_0 = \texttt{0x0C}$.

a group. Now, consider the inverse operation $\mathbf{p}_i = \mathbf{s}_i \oplus \mathbf{k}_i$ with a key which's value is irrelevant for this explanation. Within such a group, which is composed of the lookup indices $\mathbf{s}_i \in \{0, \ldots, \alpha - 1\}$ the XOR operation might flip some bits. Nevertheless, the upper $8 - \lceil \log_2 \alpha \rceil$ bits flip to the same state and the lower $\lfloor \log_2 \alpha \rfloor$ bits form the largest group of $2^{\lfloor \log_2 \alpha \rfloor}$ indices, with 0 always being part of this group. Since the lookup index where the plaintext byte $\mathbf{p}_i$ equals the correct key byte $\mathbf{k}_i$ is always the one with index $\mathbf{p}_i \oplus \mathbf{k}_i = 0$ we can be assured that the correct key byte is always within the largest block.

Figure 5.6 illustrates such a run where this observation yields the key byte $\mathbf{k}_i = \texttt{0x0C}$ without a brute-force attempt. After analyzing the odd sets, e.g., 1, 3, 5, 7, 9, 11, 13, and 15, the attack yields two key candidates, namely `0x0C` and `0x13`. Figure 5.7 outlines the possible patterns for `0x0C` and `0x13` with a disalignment of 11 and 5, respectively. Indeed, the patterns in all odd sets are equal to the one extracted from the measurement samples. What is even more interesting is the fact that we can further exploit the information within the first and the last set. Therefore, we search for the largest block within the first set and also for the largest block within the last set. Hence, our attack procedure checks whether the largest block in the first set (from `0x0C` to `0x0F`) or the largest block within the last set (from `0xF0` to `0xF7`) is smaller. Of course, the first one is smaller and hence the attack procedure computes the intersection of the set of key candidates gathered within the pattern-matching approach and the set of keys from `0x0C` to `0x0F`. Thus, with this approach we might be able to recover the correct key byte without a single brute-force key search.

Due to reasons of noise the detection of the largest block might not always work with the desired accuracy. Thus, we do not compute the intersection but simply extract the upper 4 bits and compare them against the upper 4 bits of the extracted keys from the pattern-matching approach. Nevertheless, the result is exactly the same, yielding the correct key byte without a single brute-force test.



(a) Generated pattern for $\mathbf{h}$ = `0x0C` and disalignment 11.

(b) Generated pattern for $\mathbf{h}$ = `0x13` and disalignment 5.

Figure 5.7: The two patterns which match the extracted pattern in Figure 5.6.

5.4.2 Analysis

Especially on the *Google Nexus S* this attack yields stable and reliable results. The localization of the AES T-tables seems to be possible with a minimum of $N = 4$ runs per cache set. Thus, considering the three AES encryptions per run, this yields a total of $4 \cdot 3 \cdot L1\_S$ AES encryptions in order to localize the AES T-tables. Since the *Google Nexus S* has a total of 128 cache sets this leads to 1 536 AES encryptions. However, if the T-tables are located within cache sets which are heavily used by the localization procedure, the number of measurements might be increased.

Gathering cache-access patterns has been observed to be successful for $R = 10$ runs per attacked key byte $\mathbf{k}_i$ and every cache set of the related T-table $\mathbf{T}_j$. Figure 5.8 represents the gathered cache-access patterns after $R = 10$ iterations. Though the pattern might not be visible at first glance, our attack procedure outlined above suggests four possible key candidates, i.e., `0x94`, `0x95`, `0x96`, and `0x97`. Indeed, in this case the correct key byte was `0x95`. The visual inspection clearly shows a bright area within the first set from `0x94` to `0x97`, which reveals these four key candidates.

In practice, however, one might consider exploiting the cache-access patterns of only one specific cache set per key byte $\mathbf{k}_i$. The exploitation of only one specific cache set, e.g., cache set 1 of every T-table $\mathbf{T}_j$, also implies that the possible cache-access patterns must be computed for only one set, which results in a total of $256 \cdot 256 \cdot \delta$ XOR, addition, and bitshift operations, with δ representing the number of possible disalignments.

Table 5.4 summarizes the main results of this attack on the *Google Nexus S*. We clearly observe that in case of disaligned T-tables the implemented enhancement, i.e., attacking the largest block within the first or last set, improves the result in terms of remaining bits. According to this table some disalignments, e.g., only 5 or 9 elements within the first set, are highly insecure. In this case our enhanced attack is able to recover the whole secret key without a single brute-force encryption. The low success probability of 80 % and 50 %, respectively, might be due to noisy memory-access patterns. For instance, it might be possible that the patterns could not be extracted ambiguously since we gathered too few measurement samples, e.g., $R = 20$ in this case. However, for most of the runs we observe a success probability of 100 % and less than 32 remaining bits to be searched

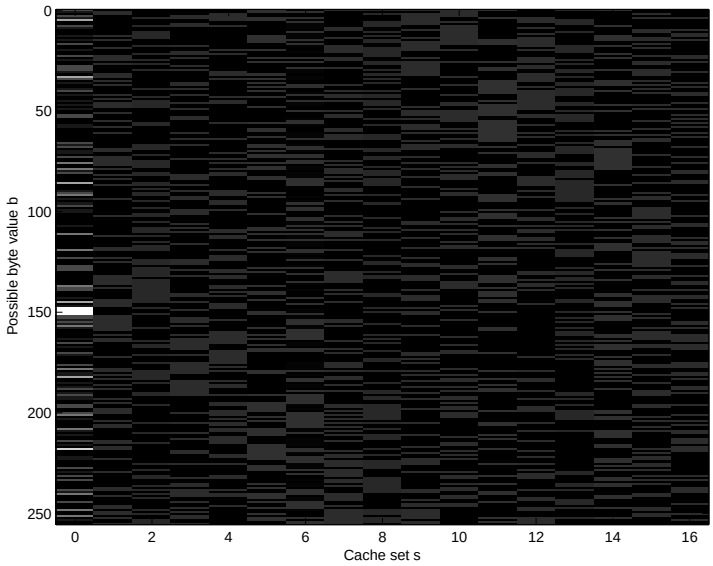


Figure 5.8: Gathered cache-access pattern on the Google Nexus S with $R = 10$ iterations.

exhaustively.

As already outlined above, if we assume that the Cortex-A8 of the *Google Nexus S* uses a virtual cache, we retrieve stable measurement results and are able to reliably determine the secret key. Our investigations revealed that devices employing an ARM Cortex-A9, e.g., *Samsung Galaxy SIII* and *Acer Iconia A510*, also leak memory-access patterns. However, for some key bytes we are not able to determine the correct key byte. Further investigations and adaptions to the attack application might be necessary in order to recover the whole secret key on the ARM Cortex-A9. For instance, the threshold values for the extraction of the pattern vector might need some refinement. As already mentioned before, access-driven attacks require sophisticated knowledge of the hardware and the device under attack. Thus, we claim that given enough information it should be possible to launch this attack successfully on the Cortex-A9 processor.

Regarding the proposed enhancement of the attack we suggest trying different approaches. For instance, finding the largest block might be achieved by either searching for the block with the greatest sum, or by searching for the block with the longest sequence according to the extracted pattern vector, etc.

Another possible extension of this attack might be that the *pattern-matching phase* only considers key candidates with the same disalignment. Since the T-tables are usually located contiguously within the memory the disalignment is the same for all T-tables. Thus, for some runs this might even further reduce the remaining key space.

According to our investigations a simple countermeasure would be the proper alignment of T-tables. This might be achieved easily by declaring the T-tables as `__attribute__((aligned(64))) static const uint32_t Te0[256]`.
The attribute *aligned* instructs the compiler to align this data structure according to a specific byte boundary. In order to align the T-tables independently of the actual cache-line size they might be aligned to a 4096-byte boundary, i.e., a page boundary. In this case our memory-access pattern approach only recovers half of the key bits on a system with a cache-line size of 64 bytes.

Attack	R	α	$\mathbf{P}_{success}$	Remaining Bits
Pattern	20	10	90 %	32
Pattern + Largest Block	20	10	90 %	18
Pattern	20	14	80 %	32
Pattern + Largest Block	20	14	50 %	25
Pattern	150	2	100 %	32
Pattern + Largest Block	150	2	100 %	16
Pattern	150	16	100 %	64
Pattern + Largest Block	150	16	100 %	64
Pattern	150	5	100 %	16
Pattern + Largest Block	150	5	100 %	0
Pattern	150	9	100 %	16
Pattern + Largest Block	150	9	100 %	0

Table 5.4: Results of the access-driven attack on the Google Nexus S.

5.4.3 Complexity

The overall complexity of the cache-access pattern attack is based on four different phases. The localization of the AES T-tables requires $N \cdot L1\_S \cdot 3$ AES encryptions. In case of the *Google Nexus S*, which employs 128 cache sets, we observed that it is possible to locate the T-tables with $N = 10$ iterations. Thus, leading to a total of $10 \cdot 128 \cdot 3 = 3\,840 \approx 2^{12}$ AES encryptions. The second phase, i.e., the gathering of the cache-access patterns, has a complexity of $16 \cdot 256 \cdot 3 \cdot \gamma \cdot R$ AES encryptions. For all 16 plaintext bytes $\mathbf{p}_i$ and all possible byte values this plaintext byte might take we trigger the AES implementation 3 times. The first encryption represents the *warm-up step* and the two subsequent encryptions represent the measurement score. Between the second and the third encryption we evict a cache set of the corresponding T-table $\mathbf{T}_j$. Our experiments are based on the eviction of all cache sets related to a specific T-table $\mathbf{T}_j$. Thus, leading to $\gamma = 17$ cache-set evictions per encrypted plaintext. The parameter $R = 150$ represents the number of iterations in order to achieve stable measurement results. Overall, this leads to a complexity of $16 \cdot 256 \cdot 3 \cdot 17 \cdot 150 \approx 2^{25}$ AES encryptions in order to gather all cache-access patterns.

We also state the complexity of the pattern computation and the extraction of the secret key in AES encryptions. Therefore, we measured the execution times of these two phases and determined how many AES encryptions our device is able to perform during this period. This yields a complexity of 2^{21} AES encryptions for the computation of the possible cache-access patterns and 2^{25} AES encryptions for the pattern-matching phase. Table 5.5 summarizes the attack complexity of this cache-access pattern attack in terms of AES encryptions.

On the *Google Nexus S* this attack takes about 80 seconds, excluding the brute-force step. This might be reduced even further, if an attacker only considers one cache set instead of all γ cache sets. Furthermore, the parameter R was chosen in order to visualize the cache-access patterns, i.e., $R = 150$. Reducing the number of iterations to $R = 20$ reduces the execution time to about 40 seconds.

Phase	
Locate the AES T-tables	2^{12}
Gather cache-access patterns	$2^{21} - 2^{25}$
Compute possible cache-access patterns	2^{21}
Extract the secret key by pattern matching	2^{25}
Brute-force key search	$0 - 2^{64}$

Table 5.5: Complexity of the cache-access pattern attack on the *Google Nexus S* given in AES encryptions.

5.4.4 Summary

We developed and analyzed an attack based on the approach by Osvik et al. [46] that ruthlessly exploits timing information leaked through the AES T-table implementation. Based on the generated memory-access patterns it might be possible to reveal the secret key without a single brute-force computation. Since we assume the T-tables to be located contiguously in memory the disalignment is the same for all T-tables. Thus, the attack might be further enhanced, for instance, by considering only key candidates which have the same disalignment. As a concluding remark of this analysis, we significantly stress the importance of aligned T-tables. Though this does not prevent timing information from being leaked it is the lesser of the two evils since only half of the key bits can be recovered on the *Google Nexus S* through simple first-round attacks.

Though the attack seems to work reliable on the *Google Nexus S* we experienced problems when launching the attack on the *Samsung Galaxy SIII* and the *Acer Iconia A510*. The problems range from architecture differences, to different noise due to possibly different operating-system versions and different services running in the background. Thus it might be difficult to develop a generic approach which gathers the cache-access patterns on all platforms. For instance, on the *Google Nexus S* comparing the absolute timing differences works reliable whereas this approach does not always work on the Samsung Galaxy SIII and the Acer Iconia A510. Attacks might be tailored at hand and launched against the platform under attack.

5.5 Time-Driven Attack

As already shown in Section 4.2, AES implementations in software usually use look-up tables in order to speed up the encryption, e.g., $\mathbf{T}_0[\mathbf{p}_0 \oplus \mathbf{k}_0]$ for the first plaintext byte in the first round. Obviously, the time for such a look-up operation depends on the used index and the cache state, e.g., whether the corresponding part of the look-up table is already located within the cache or not. Bernstein [15] claims that the time for the overall encryption of a plaintext correlates with these single lookup operations. He assumes that an attacker is able to send a plaintext $\mathbf{p}$ to a remote server, which in turn encrypts the plaintext under a known secret key $\mathbf{k}$ and returns the time needed for this encryption with a sufficient-high resolution. Due to reasons of simplicity, we assume the known key $\mathbf{k}$ to be zero. An attacker simply collects this timing information for many plaintexts and determines the encryption time for all possible values a specific plaintext byte can take, i.e., the encryption time of $\mathbf{p}_{i|i\in\{0,\ldots,15\}} \in \{\texttt{0x00}, \ldots, \texttt{0xFF}\}$. After collecting enough measurement samples the attacker determines for which value of $\mathbf{p}_i$ the overall encryption

time is maximized. We refer to this process as *study phase*. Furthermore, Bernstein [15] assumes that the attacker is also able to gather the same timing information for many samples of plaintexts $\tilde{\mathbf{p}}$ with an unknown key $\tilde{\mathbf{k}}$. Given this information he determines for which value of $\tilde{\mathbf{p}}_i$ the overall encryption time is maximized and correlates this information with the information gained before. Hence, he claims that an attacker is able to recover the secret key byte $\tilde{\mathbf{k}}_i$ by simply solving the equation $\mathbf{p}_i \oplus \mathbf{k}_i = \tilde{\mathbf{p}}_i \oplus \tilde{\mathbf{k}}_i \Rightarrow \tilde{\mathbf{k}}_i = \tilde{\mathbf{p}}_i \oplus \mathbf{k}_i \oplus \mathbf{p}_i$.

Though the real attack also considers the remaining timing information, not only the maximum values, the above mentioned scenario clearly outlines the basic idea of Bernstein's attack. He also claims that eliminating the possibility of measuring the encryption time directly on the server does not stop the attack, since the attacker might measure the encryption time on his local machine and simply average over more samples in order to eliminate noise. In contrast to Bernstein, who outlines the attack considering a remote server, Neve [37] suggests using a single computer which does all the work, i.e., the encryption, timing measurement, and even the analysis of the timing information, for simplicity reasons. We follow the approach of Neve and investigate the applicability of this time-driven attack on mobile devices.

5.5.1 Attack

The time-driven attack suggested by Bernstein [15] is composed of four phases which are: *study phase*, *attack phase*, *correlation phase*, and *key-search phase*. The following paragraphs outline the attack phases in more detail. Neve [37] also provides a detailed analysis of this time-driven attack, which we also consider in the following paragraphs. Especially in terms of the used notations, e.g., for data structures and computations, we stick to the ones introduced by Neve.

Study Phase. In this phase the attacker is supposed to know the secret key $\mathbf{k}$ under which the plaintexts are encrypted. The attacker sends multiple random plaintexts $\mathbf{p}$ to the attacked server/function/application and precisely measures the encryption time for the given plaintext. Furthermore, the attacker keeps track of this timing information in a data structure $\mathbf{t}[j][b]$, where j represents the corresponding plaintext byte position $\mathbf{p}_j$ and b the actual value of this plaintext byte. More formally, $\mathbf{t}[j][b]$ holds the sum of all encryption times of plaintexts with plaintext byte $\mathbf{p}_j = b$. In addition, a data structure $\mathbf{n}[j][b]$ keeps track of the number of encrypted plaintexts with plaintext byte $\mathbf{p}_j = b$.

As outlined in Equation 5.9, with this information it is possible to compute the average encryption time for each possible plaintext byte, compared to the overall average encryption time for all possible plaintext bytes. Neve [37] refers to this information as the plaintext-byte *signature*.

$$\mathbf{v}[j][b] = \frac{\mathbf{t}[j][b]}{\mathbf{n}[j][b]} - \frac{\sum_j \sum_b \mathbf{t}[j][b]}{\sum_j \sum_b \mathbf{n}[j][b]} \tag{5.9}$$

The attacker further computes the standard deviation of the average encryption time of each possible byte, which is used for the computation of a threshold value within the correlation phase.

Attack Phase. In this phase the attacker does not know the secret key $\tilde{\mathbf{k}}$, under which the plaintexts in this phase are encrypted. Nevertheless, the procedure in this phase

is exactly the same as in the *study phase* mentioned above. The attacker sends random plaintexts $\tilde{\mathbf{p}}$ to the encryption function and stores the gathered information in the data structures $\tilde{\mathbf{t}}[j][b]$, $\tilde{\mathbf{n}}[j][b]$, and $\tilde{\mathbf{v}}[j][b]$ respectively. Again $\tilde{\mathbf{t}}[j][b]$ stores the sum of all encryption times of plaintexts with plaintext byte $\tilde{\mathbf{p}}_j = b$, and $\tilde{\mathbf{n}}[j][b]$ counts the number of plaintexts with plaintext byte $\tilde{\mathbf{p}}_j = b$. The corresponding plaintext-byte *signature* is computed exactly as outlined in the learn phase above, except that $\mathbf{t}$, $\mathbf{n}$, and $\mathbf{v}$ are replaced with $\tilde{\mathbf{t}}$, $\tilde{\mathbf{n}}$, and $\tilde{\mathbf{v}}$, respectively.

Correlation Phase. Within the correlation phase the attacker combines the timing profiles gained in the study phase with the timing profiles gained in the attack phase, i.e., $\mathbf{v}$ and $\tilde{\mathbf{v}}$. According to Neve [37] the following heuristic is introduced: If pairs of plaintext bytes and key bytes in the study phase ($\mathbf{p}_i$, $\mathbf{k}_i$) and in the attack phase ($\tilde{\mathbf{p}}_i$, $\tilde{\mathbf{k}}_i$) have the same difference, i.e., $\mathbf{p}_i \oplus \mathbf{k}_i = \tilde{\mathbf{p}}_i \oplus \tilde{\mathbf{k}}_i$, then these encryptions might have a similar timing profile. Hence, by simply rearranging this equation for pairs with a similar timing profile, one retrieves possible key candiates $\tilde{\mathbf{k}}_i$ for known plaintext bytes $\mathbf{p}_i$ and $\tilde{\mathbf{p}}_i$, as well as the known key $\mathbf{k}_i$.

$$\mathbf{c}[j][\mathbf{b}] = \sum_{i=0}^{255} \mathbf{v}[j][i] \cdot \tilde{\mathbf{v}}[j][i \oplus \mathbf{b}] \tag{5.10}$$

Equation 5.10 states the computation of this correlation. Note that we assume the known key $\mathbf{k}$ to be zero and hence we do not have to consider $\mathbf{k}$ here. The resulting correlation values $\mathbf{c}[j]$ are sorted in descending order and the value $\mathbf{b} \in \{\texttt{0x00}, ..., \texttt{0xFF}\}$ represents possible key candidates with respect to the correlation. The higher the correlation $\mathbf{c}[j][\mathbf{b}]$ of the timing profiles, the more likely the corresponding value of $\mathbf{b}$ represents a possible key candidate. The published source code of Bernstein [15] (*correlate.c*) states the usage of a threshold based on the standard deviation computed in the study phase and the attack phase, respectively. Simply spoken, the correlation phase searches for values of $\mathbf{b}$ for which the timing profiles correlate most.

Search Phase. Due to the fact that the correlation phase usually outputs multiple possible key candidates for each key byte, a brute-force key search is necessary in order to reveal the correct secret key.

5.5.2 Analysis

Since O'Hanlon and Tonge [39] reported problems in launching the attack on a Pentium III and a Pentium IV processor, we cover their main findings first. Later on we state our main findings of the time-driven attack on a desktop PC and finally we outline the results of this attack on the three mobile devices.

Within their first approach O'Hanlon and Tonge [39] tried to reproduce Bernstein's timing attack on a *Pentium IV* processor, running *GCC 4.0.0* and *OpenSSL 0.9.7f*. After gathering 2^{30} measurement samples in the learn phase and 2^{25} measurement samples in the attack phase, they were not able to narrow down the key space at all. They also launched the attack on a *Pentium III* processor, also running *GCC 4.0.0* and *OpenSSL 0.9.7f*, and found out that though the key space was not reduced significantly, two key bytes leaked timing information and were reduced to 8 and 16 possible key candidates, respectively. Within another approach they compiled *OpenSSL 0.9.7a* with *GCC 2.95.3* and launched

Processor	AES implementation	GCC	Remaining key space
Pentium IV	OpenSSL 0.9.7f	4.0.0	128 bits
Pentium III	OpenSSL 0.9.7f	4.0.0	$\sim$ 116 bits
Pentium III	OpenSSL 0.9.7f	2.95.3	$\sim$ 116 bits
Pentium III	OpenSSL 0.9.7a	4.0.0	$\sim$ 116 bits
Pentium III	OpenSSL 0.9.7a	2.95.3	$\sim$ 103 bits
Pentium III	MIRACL	2.95.3	$\sim$ 34 bits

Table 5.6: Summary of the results by O'Hanlon and Tonge (cf. [39]).

the attack again, which led to a slight improvement. This time four key bytes were reduced to 8 and 16 possibilities. However, most of the remaining key bytes were not reduced at all, which means that they still showed up 256 possible candidates per key byte. Actually, O'Hanlon and Tonge [39] blame the larger cache[2] of their *Pentium III* for the failure of this attack and claim that Bernstein investigates a smaller cache size, which leads to more cache misses. Hence, instead of attacking the *OpenSSL* implementation they started attacking *MIRACL's* AES implementation, since they claim that this implementation takes up more space in cache memory. They immediately declared success. The remaining key space, with a complexity of approximately 2^{34} key-byte combinations, could be searched easily. Table 5.6 summarizes their main findings with the corresponding configuration and the remaining key-space complexity.

Due to the fact that ARM processors implement a random replacement policy, which might exacerbate Bernstein's timing attack, we also start by launching the attack on a desktop computer before moving on to the mobile devices.

Desktop Machines

In order to investigate the applicability of Bernstein's timing attack on modern desktop machines we use the following two devices: (1) a *Lenovo Thinkpad W500* and (2) a *Lenovo Thinkpad W520*. The *Thinkpad W500* employs an *Intel Core2 Duo* dual-core CPU, clocked at 2.66 GHz, and runs *GCC 4.5.2*. Furthermore, it employs an 8-way associative L1 data cache, with a total size of 32 KB, and a cache-line size of 64 bytes. The *Thinkpad W520* employs an *Intel Core i7-2670QM* quad-core CPU, clocked at 2.2 GHz, and runs *GCC 4.6.1*. The properties of the L1 data cache are exactly the same as for the *Thinkpad W500*. The OpenSSL implementations used for the following experiments were taken from the OpenSSL website [41]. In order to use the T-table implementation, rather than the optimized and (probably) timing-attack resistant assembler implementation, we configured and compiled the OpenSSL implementations with the flag `no-asm`.

Table 5.7 summarizes the most promising results of this attack for different configurations and different parameters on both machines. The columns *Study* and *Attack* denote the number of measurement samples to be gathered in the corresponding phase. The column *Cores* state information about the number of active cores and the CPU frequency, i.e., whether we pinned the CPU frequency or allowed the machine to adjust the frequency according to the *Intel SpeedStep Technology* [28] dynamically. For further information regarding the deactivation of CPU cores and how to pin the CPU frequency we refer to

[2]Bernstein does not explicitly state the cache size of the attacked *Pentium III*. Nevertheless, he points out that a typical *Pentium III* has 16 KB of L1 cache, with a cache-line size of 32 bytes.

Machine	Cores	Study	Attack	OpenSSL	Success	Remaining
W500	1 (800 MHz)	2^{25}	2^{25}	0.9.8o	2^{25}, 2^{25}	65 bits
W520	4 (dynamic)	2^{34}	2^{34}	0.9.7a	2^{33}, 2^{31}	52 bits
	4 (dynamic)	2^{30}	2^{30}	0.9.7a	2^{30}, 2^{30}	83 bits
	4 (dynamic)	2^{30}	2^{30}	1.0.1c	2^{30}, 2^{30}	83 bits

Table 5.7: Results of the time-driven attack on the two desktop machines.

Section 4.4. The column *Success* denotes the number of samples in the study phase and in the attack phase, respectively, for which the correlation yields all correct key bytes with respect to the lowest brute-force complexity. Observe, that this column does not necessarily yield the same number of samples as denoted by the columns *Study* and *Attack*. We state the reason for this as follows. As suggested by Bernstein [15], we output the statistical values in the study phase as well as in the attack phase after gathering a number of samples which is exactly a power of two. For instance, suppose we want to gather 2^{30} samples in the study phase as well as in the attack phase. After processing 2^{15} measurement samples in each phase we output the statistical values for the first time. After processing 2^{16} measurement samples we output the statistical values again and so on, until we finally reach the supposed 2^{30} measurement samples. For analysis purposes we correlate all possible combinations of measurement samples from the study phase with the measurement samples from the attack phase, i.e., the statistical values after 2^i measurement samples within the study phase with the statistical values after 2^j measurement samples within the attack phase, for $15 \leq j \leq 30$ and $15 \leq i \leq 30$. The last column denotes the number of remaining bits x to be searched within the brute-force key-search phase, i.e., 2^x AES encryptions.

On the Thinkpad W500 we observed the best results when simulating a standard situation while performing the attack, e.g., browsing the web (Google Chromium) and a mail client (Mozilla Thunderbird) continuously checking for new mails. Furthermore, we did not retrieve remarkable results with both cores enabled. Hence, in order to prevent the attack process and the noise-generation applications from being executed on different cores we disabled the second CPU core and pinned the remaining CPU core to a frequency of 800 MHz. However, even with only one CPU enabled the attack did not yield stable results, i.e., two successive runs did not yield approximately the same amount of key bits. This inconsistency might be due to the multiple processes running on the system in the background. In contrast, on the Lenovo Thinkpad W520 we did not even had to disable any core in order to retrieve stable results. We simply fired up a web browser (Google Chromium) and generated continuous memory accesses by playing videos. Indeed, on this machine the attack constantly yielded a remaining key space of approximately 85 bits. Note that we did not further analyze the impact of multi-core processors, hyper-threading, nor the L2 and L3 data cache on the attack.

As already mentioned in Section 4.4, the usage of Intel's Time-Stamp Counter (TSC) might cause inaccurate timing measurements on processors which support out-of-order execution. Hence, a complex serialization process is necessary in order to ensure accurate timing measurements. Neve [37] also mentioned the problem of the RDTSC instruction, but since he snipped the corresponding timing function in the provided source code we do not know whether he adopted to this fact or not. Nevertheless, in our case the complex serialization process did not yield better results.

# candidates	Key byte	Possible values								
8	0	b3	b4	**b5**	b1	b7	b2	b6	b0	
200	1	34	36	**32**	30	33	37	35	31	...
8	2	5d	58	**5a**	59	5b	5e	5f	5c	
8	3	a2	a3	a4	a7	**a6**	a0	a5	a1	
32	4	a5	a7	**a1**	a3	a6	a4	a2	a0	...
32	5	d4	d6	**d7**	d5	d2	d3	d1	d0	...
32	6	f3	f6	f1	f2	**f5**	f4	f7	f0	...
8	7	f4	f6	**f1**	f2	f0	f3	f7	f5	
32	8	19	1e	1f	1d	1a	**1b**	18	1c	...
8	9	21	23	27	**25**	20	24	26	22	
8	10	f5	f3	f2	f0	f1	f7	f6	**f4**	
199	11	0f	0e	**0c**	0a	0d	09	08	0b	...
8	12	7f	7e	**7b**	7c	7d	79	78	7a	
8	13	b0	b4	b7	b2	b1	b3	b6	**b5**	
8	14	2b	2a	2d	2e	2c	2f	**28**	29	
8	15	94	95	90	**97**	93	96	91	92	

Table 5.8: Sample output of the correlation phase on the Thinkpad W500.

Table 5.8 visualizes the output of the correlation phase of an attack on the Lenovo Thinkpad W500. For this specific example we gathered 2^{25} measurement samples in both phases, the study phase as well as the attack phase. The first column states the number of recovered key candidates for the corresponding key byte, the second column denotes the position i of the key byte ($\mathbf{k}_i$), and the third column denotes the recovered key-byte candidates. The correct key bytes are underlined and marked in bold. For key bytes where the number of possible key candidates is larger than eight we only state the first eight candidates. Computing the product of the values within the first column yields the number of possible key combinations, i.e., the complexity of the remaining brute-force search. In this case the corresponding brute-force complexity is $\sim 2^{65}$ AES encryptions, which is still out of reach. However, we observe a significant leakage of timing information. Interestingly, it can be observed that all correct key bytes are within the first eight possible key-byte candidates, which would reduce the complexity to 2^{48} AES encryptions. Unfortunately, multiple experiments showed that this is not always the case and hence we cannot rely on this observation.

According to our experiments we observed that sometimes the key space is reduced too much. This means that for some key bytes the correct key is not present anymore in the candidate list. Table 5.9 illustrates this problem. In this example the brute-force complexity would be 2^{30} AES encryptions. However, nearly half of the correct key bytes are not present anymore, which renders this run completely useless. Table 5.10 illustrates the same problem for a run with 2^{34} samples in both online phases. Except for key byte 7, where the correct byte ($8d$) is missing, all key bytes were recovered successfully. This output would lead to a remaining complexity of approximately 2^{55} AES encryptions. But again, for one key byte the correct byte value is missing which also renders this run completely useless. Interestingly, for most of the key bytes the correct key is listed at the first position.

Since further analysis of Bernstein's attack on desktop machines would exceed the scope

# candidates	Key byte	Possible values								
1	0	95								[missing **b5**]
1	1	1c								[missing **32**]
8	2	5b	59	**5a**	5f	5d	58	5e	5c	
8	3	a2	a3	a0	a1	a5	a7	**a6**	a4	
2	4	81	83							[missing **a1**]
2	5	f7	f6							[missing **d7**]
8	6	f6	f0	f2	f4	f1	f7	**f5**	f3	
8	7	f4	f7	f2	**f1**	f0	f3	f6	f5	
1	8	39								[missing **1b**]
1	9	05								[missing **25**]
2	10	d7	d6							[missing **f4**]
8	11	08	0a	0d	0b	**0c**	0f	09	0e	
8	12	7c	79	78	7d	7a	**7b**	7e	7f	
8	13	b6	b7	b4	b0	**b5**	b1	b2	b3	
8	14	2f	2e	2d	29	2c	2b	**28**	2a	
8	15	93	92	91	95	90	94	**97**	96	

Table 5.9: Sample output of the correlation phase on the Thinkpad W500 where the key space was reduced too much.

of this work and we already observed that timing information leaks through multiple AES encryptions we continue with the analysis on mobile devices. For a more detailed analysis regarding the applicability of Bernstein's attack we refer to Bernstein [15] and Neve [37].

Mobile Devices

The best results, according to our investigations, can be achieved by simply generating the required memory accesses (noise) within the attack application itself. This is done by changing the size of data arrays within the implementation and hence perform additional memory accesses. Actually, this is the same approach as suggested by Bernstein [15]. Within multiple consecutive runs, Bernstein simply sends data of different length to the server. Though the server only encrypts 16 bytes of the transmitted data, the rest of the data is copied to a temporary data array and hence possibly generates cache evictions of already loaded T-table elements.

We also tried to generate a more realistic scenario. In this scenario we mounted the attack while watching videos, or while watching an image slideshow on the mobile devices. However, according to our observations the generation of noise due to external applications, e.g., applications launched simultaneously to the application that performs the attack, does not yield better results of this attack. Only minor timing information leaked and finally the remaining key space was not reduced significantly. Based on our observations we conclude that either the cache evictions affect the wrong cache sets, i.e., cache sets which do not contain T-table elements, or the generated noise is not constant and hence corrupts the timing profiles. Furthermore, on multi-core devices, as for instance the *Acer Iconia A510* and the *Samsung Galaxy SIII*, the two applications might be executed on different cores. We further conclude that a fairly realistic approach would be to wrap the attack in a fine-grained application, i.e., a fancy game or another long running application, and to

# candidates	Key byte	Possible values								
1	0	**b5**								
32	1	90	91	**93**	92	82	81	83	9c	...
22	2	**11**	c1	39	01	f9	29	e9	c9	...
43	3	**66**	7a	64	73	72	78	61	70	...
4	4	ae	**af**	ad	ac					
32	5	cc	d4	c4	dc	2c	34	...	**bc**	...
15	6	**cd**	fd	05	f5	35	c5	15	dd	...
224	7	8c	8e	f2	e2	f1	ea	f3	02	... [missing 8d]
1	8	**12**								
32	9	**b9**	b1	a9	a1	59	51	49	41	...
36	10	**55**	15	05	1d	4d	0d	f5	e5	...
4	11	**12**	10	11	13					
1	12	**92**								
4	13	**4a**	4b	48	49					
32	14	**6b**	8b	b3	23	73	bb	83	9b	...
4	15	9f	9e	9c	**9d**					

Table 5.10: Sample output of the correlation phase on the Lenovo Thinkpad W520.

Device	Study	Attack	Success	Remaining key space
Google Nexus S	2^{30}	2^{30}	2^{30}, 2^{29}	65 bits
	2^{29}	2^{29}	2^{29}, 2^{28}	69 bits
Samsung Galaxy SIII	2^{30}	2^{30}	2^{30}, 2^{29}	58 bits
	2^{30}	2^{30}	2^{30}, 2^{30}	61 bits
Acer Iconia A510	2^{30}	2^{30}	2^{30}, 2^{27}	73 bits
	2^{30}	2^{30}	2^{30}, 2^{29}	78 bits

Table 5.11: Results of the time-driven attack on the mobile devices.

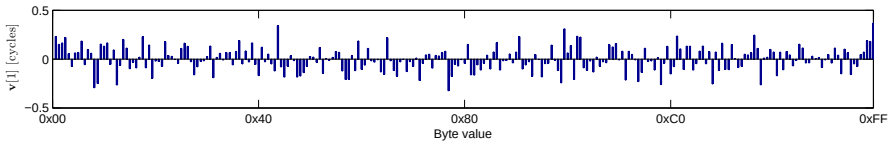
control the generated memory accesses within this application.

Table 5.11 summarizes the best attack results on the tested mobile devices. For analysis purposes we state the output of a sample run in Table 5.12. This specific run yields a remaining brute-force complexity of 2^{61} AES encryptions. Considering the output of the *correlation phase*, e.g., for key byte 1, indicates that a high number of key candidates is proposed. The corresponding byte-signature plot in Figure 5.9 clarifies the problem. The x axis shows the possible values the corresponding byte $\mathbf{p}_i$ might take, i.e., `0x00` $\leq \mathbf{p}_i \leq$ `0xFF`, and the y axis the average encryption time of this specific byte value subtracted by the overall average encryption time. In case of key byte 1 the byte signatures of both phases do not reveal any obvious patterns and look noisy. Hence the correlation fails, which means that many key candidates are proposed. In contrast, for byte 7 the key space was reduced to 4 possible key candidates. The corresponding signature plot in Figure 5.10 reveals a clear pattern in both phases.

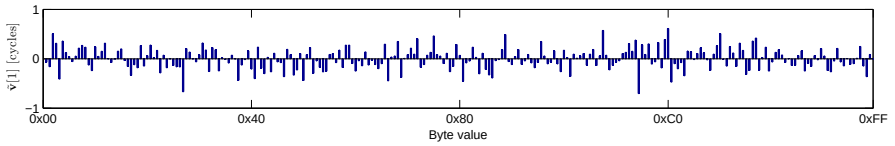
We conclude that it is possible to launch Bernstein's time-driven attack against all three mobile devices. Of course, the remaining key space is still too large for a brute-force key search, but we clearly observe the leakage of timing information. Generally, we

# candidates	Key byte	Possible values								
106	0	f4	f5	6d	2f	31	1b	...	**00**	...
132	1	71	32	2f	43	fa	1c	...	**10**	...
9	2	23	2f	21	2c	22	2e	**20**	2d	...
4	3	33	31	**30**	32					
37	4	84	a0	dc	38	6e	f4	...	**40**	...
12	5	53	**50**	51	52	4c	49	4d	fd	...
6	6	**60**	61	62	6f	6e	63			
4	7	71	**70**	73	72					
130	8	fc	84	88	**80**	19	94	b8	91	...
7	9	93	9d	**90**	91	8c	3d	9e		
9	10	a2	a3	bd	**a0**	ac	a1	ae	ad	...
4	11	b3	b2	b1	**b0**					
117	12	20	fa	**c0**	c8	22	c4	18	b8	...
4	13	**d0**	d3	d2	d1					
8	14	e3	ec	ed	e2	e1	**e0**	ee	ef	
4	15	f3	f1	f2	**f0**					

Table 5.12: Sample output of the correlation phase on the Samsung Galaxy SIII.

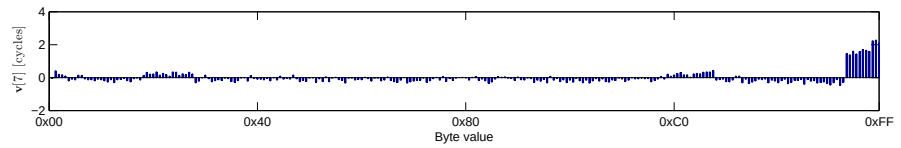


(a) Sample signature plot of byte 1 in the study phase on the Samsung Galaxy SIII.

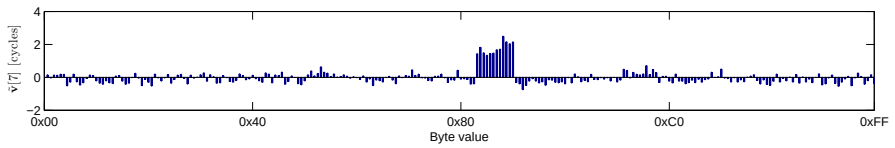


(b) Sample signature plot of byte 1 in the attack phase on the Samsung Galaxy SIII.

Figure 5.9: Sample signature plot of byte 1 on the Samsung Galaxy SIII.



(a) Sample signature plot of byte 7 in the study phase on the Samsung Galaxy SIII.



(b) Sample signature plot of byte 7 in the attack phase on the Samsung Galaxy SIII.

Figure 5.10: Sample signature plot of byte 7 on the Samsung Galaxy SIII.

observed the same behaviour as on the desktop machines. For some runs the correlation phase leaked the correct key byte immediately, for others the correct key byte was not present anymore, and yet for others the key spaces was not reduced at all.

However, in order for this attack to be successful an attacker might carefully adapt the implementation in order to generate the required cache evictions, such that the encryption function leaks enough timing information. Indeed, this sounds like an awful lot of work since the attacker in general does not have any information about the location of the T-tables within the cache. Hence, the attacker does not know where the noise should be generated. Due to the impressive number of required measurement samples to be generated in the two online phases (study phase and attack phase), this attack might drastically drain the battery on mobile devices. Hence, one might consider launching this attack only if the battery is charging. Supposing that the average user charges a mobile device during the night this is a rather reasonable assumption. Another scenario might be an attacker who wants to attack the disk encryption. In this case the attacker is in possession of the device and is willing to do anything in order to get the secret key. The online phases take about six hours on our Google Nexus S in order to generate $2 \cdot 2^{30}$ measurement samples. On the *Acer Iconia A510* and the *Samsung Galaxy SIII* the same number of measurement samples takes just four hours. We consider the brute-force key-search phase to be carried out on a remote machine. In this case the mobile device transfers the gathered information to a remote server and the server continues the attack, which might potentially lead to a successful key recovery.

5.5.3 Complexity

In contrast to access-driven attacks, time-driven attacks feature an enormous online complexity. Table 5.13 summarizes the complexity of Bernstein's timing attack in terms of AES encryptions. The range of the brute-force key search is given in terms of the best and the worst result of our attacks. Since the correlation phase only involves the computation of correlation values based on the aggregated statistical data computed in the two online phases, we ignore this complexity here. Note that only the study phase and the attack phase must be carried out on the mobile device. The brute-force key search might be carried out on a more powerful device, possibly even supporting AES New Instructions (AES-NI).

Phase	
Study phase	2^{25} – 2^{32}
Attack phase	2^{25} – 2^{32}
Brute-force key search	2^{58} – 2^{128}

Table 5.13: Complexity of Bernstein's attack given in AES encryptions.

5.5.4 Summary

The following paragraphs summarize our main observations of the analysis of Bernstein's time-driven attack.

Observation 1. It might be hardly possible to produce stable results, i.e., that two successive runs yield the same amount of key bits. Bernstein's attack requires cache manipulations due to memory accesses of other processes and even the process performing the encryption itself performs cache evictions of already loaded T-table elements. If the cache would not be manipulated by other memory accesses, the encryption might run in constant time, since all data might be provided from the L1 data cache in constant time. Neve [37] claims that the exploitable timing information leaks through the manipulation of constant cache sets by different processes running on the same machine. The source code published by Bernstein [15] indicates that the imaginary server function, which does the encryption, manipulates a temporary array. Of course, these array manipulations lead to constant cache manipulations and hence also lead to exploitable timing differences. We conclude that possibly uncontrollable noise might corrupt the timing measurements and hence lead to wrong key bytes or the key space is not reduced at all. On server machines, which commonly do the same work all the time, this might lead to more successful attacks. In contrast, on mobile devices the user might frequently change the used applications and hence this attack might not succeed due to frequently changing memory accesses.

Observation 2. For analysis purposes, we correlate all possible combinations of measurement samples in the study phase with the ones from the attack phase. However, in practice one would define the number of samples to be generated and the resulting output would be correlated. As our investigations revealed, the initially defined number of measurement samples only rarely yields the correct key bytes. This might exacerbate the attack since we do not know in advance how many samples to generate.

Observation 3. According to our investigations, the best runs yield a remaining brute-force complexity of approximately 2^{60} AES encryptions. Though the usage of AES New Instructions on modern Intel processors might drastically speed up the AES encryption, 2^{60} AES encryptions are still unfeasible.

Observation 4. Sometimes the key space is not reduced at all and sometimes the key space is reduced too much. While the former implies that the correlation phase returns all possible values for a specific key byte, the latter implies that some of the correct key bytes are not present anymore. More formally, if the key space is reduced in order to bring this attack in the range of a possible brute-force key search, then the correct key-byte is probably not present anymore. However, if all correct key bytes are still present after

the correlation phase, then the remaining complexity is far too large for a brute-force key search. For instance, we observed runs where the remaining brute-force complexity was 2^{30} AES encryptions. Unfortunately, nearly half of the correct key bytes were not present anymore, which rendered this run of the attack completely useless.

Observation 5. Due to the huge number of samples to be generated within the study phase and the attack phase, this attack might drastically drain the battery on mobile devices. Hence, one might consider launching this attack only if the battery is charging. Having the "standard user" in mind, who charges the mobile device during the night, this might be a reasonable assumption. Even if the online phase takes about six hours as in case of the *Nexus S*.

5.6 Cache-Collision Timing Attack

Bogdanov et al. [17] suggest the exploitation of so called *wide collisions* between two consecutive encryptions of chosen plaintexts. The main idea behind their attack is to choose pairs of plaintexts $(\mathbf{P}_1, \mathbf{P}_2)$ in a specific manner, such that five S-Box lookups collide when encrypting such a pair of plaintexts. Supposing an empty cache, the encryption of $\mathbf{P}_1$ loads the corresponding table elements into the cache. Obviously, the encryption of $\mathbf{P}_2$ is computed faster if table lookups collide, i.e., if intermediate state bytes between $\mathbf{P}_1$ and $\mathbf{P}_2$ are equal. However, due to the fact that one cache line contains multiple table elements, the lookup of two state bytes might cause a cache collision though the two bytes are not necessarily equal. As we will see later in this section, this imposes a problem.

Besides the concept of *wide collisions*, the following section describes the cache-collision attack in more detail. We also state the main findings of the cache-collision attack on our three test devices and, last but not least, we analyze the attack complexity. Though S-Box collisions are considered for the explanation of *wide collisions*, it should be clear that this attack also works for the common T-table implementation of the AES.

5.6.1 Attack

Before we cover the details of the cache-collision attack of Bogdanov et al. [17], we outline the notion of a *wide collision* as follows. Figure 5.11 visualizes the initial AES states of $\mathbf{P}_1$ and $\mathbf{P}_2$, represented as a matrix of four columns and four rows, respectively. Due to reasons of simplicity we stick to their notation: Bytes which are equal in both states are shown brighter, whereas bytes which are different in both states are shown darker. Suppose we choose the diagonal pair $(A,\ E)$, i.e., $A = \{a_0, a_1, a_2, a_3\}$ and $E = \{e_0, e_1, e_2, e_3\}$, independently and randomly, such that $a_i \neq e_i$ for $0 \leq i < 4$. Furthmore, suppose we also choose the remaining bytes randomly, but equal for both plaintexts. Afterwards, we apply the transformations labeled in Figure 5.11 for the first round and inspect the resulting output state. Due to the nature of the chosen plaintext, the resulting state bytes are pairwisely equal between the intermediate state of $\mathbf{P}_1$ and $\mathbf{P}_2$ within the columns 2, 3, and 4. Due to the *ShiftRows* transformation the pairwisely different bytes, i.e., the bytes of the diagonal pair $(A,\ E)$, are aligned within the first column. However, due to the *MixColumns* transformation it might be possible that some bytes — at most 3 at the same time — of the first column are equal between the first and the second state, i.e., $a'_i = e'_i$ for some $0 \leq i < 4$. Within the sample transformations outlined in Figure 5.11 we assume that $a'_1 = e'_1$. If a collision between any a'_i and e'_i occurs, then the colliding bytes are

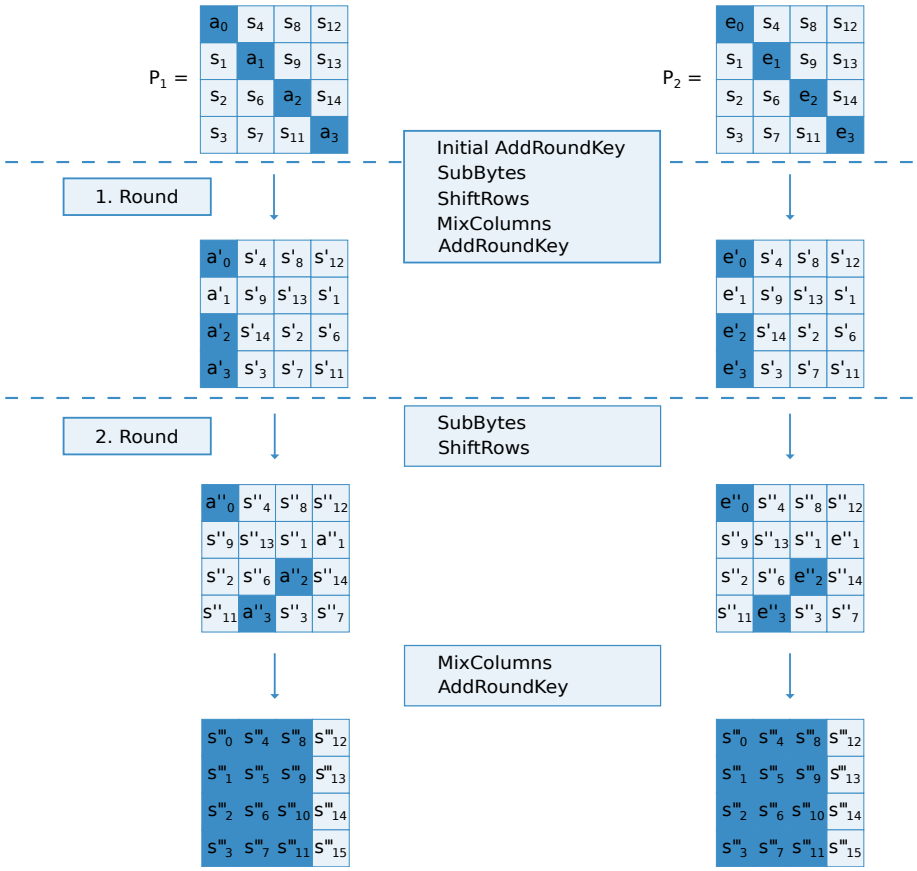


Figure 5.11: Intermediate states of a plaintext pair ($\mathbf{P}_1$, $\mathbf{P}_2$) producing a wide collision, assuming $a'_1 = e'_1$ and hence $a''_1 = e''_1$ also holds true.

shifted to the same column by the *ShiftRows* transformation of the second round. In case of this example the colliding bytes $a'_1 = e'_1$ are shifted to the fourth column. Consequently, the *MixColumns* and *AddRoundKey* transformations output pairwisely equal columns for both states, which leads to four additional S-Box collisions in the third round and thus to lower encryption times compared to encryptions where no wide collision occurs. Of course, the pairwisely equal columns depend on where the colliding bytes $a'_i = e'_i$ are shifted to.

Now that we are familiar with the notion of a *wide collision*, we outline the cache-collision attack according to Bogdanov et al. [17], which consists of the following three phases: *online phase*, *collision-detection phase*, and *key-search phase*. Though we describe these phases only for one out of the four possible diagonals, i.e., bytes of the plaintext which are shifted to the same column after the *ShiftRows* transformation, these steps must be performed for all four diagonals in order to recover the whole secret key. For the sake of clarity, all four possible diagonals are outlined in Figure 5.12.

Online Phase. The purpose of the online phase is to gather measurement samples for the subsequent collision-detection phase. Hence, the attacker randomly chooses N different pairs of diagonals (A, E), such that $a_i \neq e_i$, for $0 \leq i < 4$. For each of these diagonal pairs (A, E), the remaining plaintext bytes are chosen randomly but equal for both plaintexts I times. In order to achieve stable measurement results, R iterations are performed for each plaintext pair ($\mathbf{P}_1$, $\mathbf{P}_2$), which simply means that

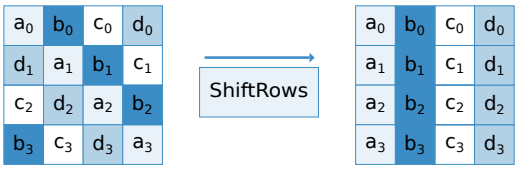


Figure 5.12: All four possible diagonals.

the chosen pair of plaintexts is encrypted R times. Actually, within each iteration the attacker clears the cache[3], encrypts plaintext $\mathbf{P}_1$, and subsequently measures the encryption time of plaintext $\mathbf{P}_2$. Hence, the output of the online phase consists of a list of diagonal pairs (A, E) along with the encryption time of the second plaintext. While Bogdanov et al. [17] suggest using the average encryption time of the $j \in \{2, 5, 10, R\}$ fastest out of the R encryptions, we compute multiple statistical values in the online phase for analysis purposes.

$$\begin{aligned} S^1 = \begin{bmatrix} a'_0 \\ a'_1 \\ a'_2 \\ a'_3 \end{bmatrix} &= \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} \text{SBox}(s_0) \\ \text{SBox}(s_5) \\ \text{SBox}(s_{10}) \\ \text{SBox}(s_{15}) \end{bmatrix} \\ &= \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} \text{SBox}(a_0 \oplus k_0) \\ \text{SBox}(a_1 \oplus k_5) \\ \text{SBox}(a_2 \oplus k_{10}) \\ \text{SBox}(a_3 \oplus k_{15}) \end{bmatrix} \end{aligned} \tag{5.11}$$

Collision-Detection Phase. In this phase the output of the online phase, i.e., the list of diagonal pairs (A, E) along with the corresponding encryption times, is analyzed in order to determine which diagonals lead to wide collisions. As already mentioned above, this is done based on the encryption time of the second plaintext $\mathbf{P}_2$. Based on this decision there might be *false positives*, i.e., diagonal pairs which are supposed to lead to wide collisions due to their encryption time, but in fact do not lead to wide collisions. As we will see in the next phase, at least 4 diagonal pairs leading to wide collisions are necessary in order to recover the correct sub key. Hence, the higher the expectation rate of *false positives*, the more diagonal pairs are necessary for the next phase. If only 4 diagonal pairs are chosen and one pair is a *false positive*, the evaluation of the next phase will not yield the correct sub key.

Key-Search Phase. The first step of this phase is to determine which sub keys lead to a wide collision between any four diagonal pairs (A, E) output in the previous phase. Due to the above mentioned problem of false positives, the collision-detection phase might output more than four pairs of diagonals. The attacker has to consider all possible combinations of four diagonal pairs out of the suggested diagonal pairs. Then, the attacker exhaustively iterates over all possible sub keys of the corresponding diagonal (2^{32} combinations) and computes, for each of these 4 diagonal pairs, the intermediate AES state according to Equation 5.11.

Equation 5.11 outlines the round transformations of the first round for column one, excluding the *AddRoundKey* operation. The *AddRoundKey* transformation of round

[3]For further details regarding the eviction of cache sets we refer to Section 5.2.

one is omitted since it is a simple XOR operation and if two bytes collide before the XOR operation, then these bytes also collide after the XOR operation with the same round key. Nevertheless, the initial *AddRoundKey* ($s_i = a_j \oplus k_i$), with $j \equiv i \mod 4$, for the investigated diagonal must be considered and is computed as follows: $s_0 = a_0 \oplus k_0$, $s_5 = a_1 \oplus k_5$, $s_{10} = a_2 \oplus k_{10}$, and $s_{15} = a_3 \oplus k_{15}$. The *SubBytes* transformation is denoted by the *SBox()* operation and the *ShiftRows* transformation is done implicitly since the diagonal $A = \{a_0, a_1, a_2, a_3\}$ is considered to be aligned within the first column after the *ShiftRows* transformation. The same equation can be formed for plaintext $\mathbf{P}_2$, except that $\mathbf{a}_i$ and $\mathbf{a}'_i$ are replaced with $\mathbf{e}_i$ and $\mathbf{e}'_i$ respectively. Due to the fact that the matrix multiplication stated in Equation 5.11 leads to a system of equations ($a'_i = e'_i$ for $0 \leq i < 4$) with four unknowns (k_0, k_5, k_{10}, and k_{15}), four diagonal pairs (A, E) leading to wide collisions are necessary in order to recover the corresponding sub key of this diagonal.

By iterating over all possible sub keys of the corresponding diagonal, i.e., k_0, k_5, k_{10}, and k_{15}, and computing the intermediate state S^1 according to Equation 5.11 for both diagonals A and E, one simply checks which key candidates lead to a collision of any byte value $a'_i = e'_i$ between all four pairs of diagonals A and E. However, the colliding byte positions might differ between two diagonal pairs, e.g., for the first pair of diagonals the colliding bytes are $a'_i = e'_i$ and for the second pair of diagonals the colliding bytes are $a'_j = e'_j$, with $i \neq j$. If a collision occurs between all four pairs of diagonals A and E, then the corresponding choice of the sub key is treated as a possible sub-key candidate and stored for the following brute-force key search.

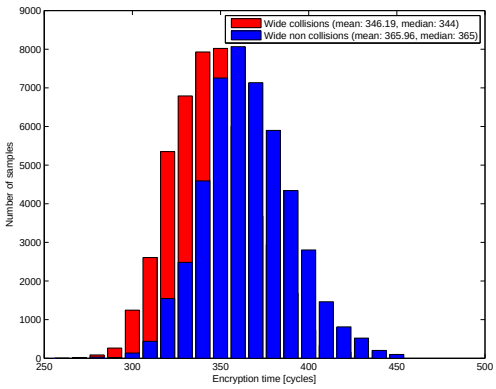
After computing the list of possible sub keys for all 4 diagonals the final brute-force key search can be performed. Therefore all possible combinations of 4-byte sub keys are checked against a known plaintext-ciphertext pair.

5.6.2 Analysis

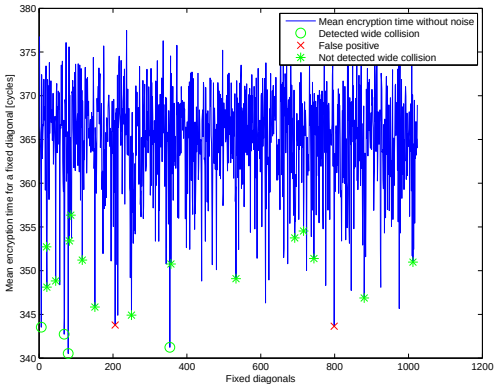
The following section states the main results of the cache-collision attack on the ARM Cortex-A8 and the ARM Cortex-A9 processor. The attack of the full 10-round AES implementation on the ARM Cortex-A8 revealed that a reliable detection of wide collisions seems to be a challenging task. Hence, we start by attacking a reduced version of the AES, with only 3 rounds. The histogram in Figure 5.13(a) visualizes the encryption times of a 3-round AES implementation on an ARM Cortex-A8 processor. Five diagonal pairs (A, E) which result in a wide collision and five diagonal pairs (A, E) which do not result in a wide collision are encrypted. Due to reasons of noise each encryption is performed $I \cdot R$ times ($I = 400$, $R = 20$). The lower encryption times of plaintexts which lead to wide collisions are clearly observable. Hence, the execution time can be used to distinguish diagonals which lead to wide collisions from diagonals which do not lead to wide collisions. For the ARM Cortex-A9 processor the encryption times of wide collisions and non wide collisions are also clearly separable. The plots are similar to the one shown for the ARM Cortex-A8 processor.

With the separable encryption times of these two categories of plaintexts in mind, Figure 5.13(b) visualizes an approach to determine diagonals which lead to wide collisions. The plot shows the mean encryption time without noise, i.e., averaged over $I \cdot R$ encryption times below a predefined threshold[4] of a fixed diagonal pair (A, E). This was done since our

[4]The threshold might be determined in a preprocessing stage by averaging over multiple encryptions.

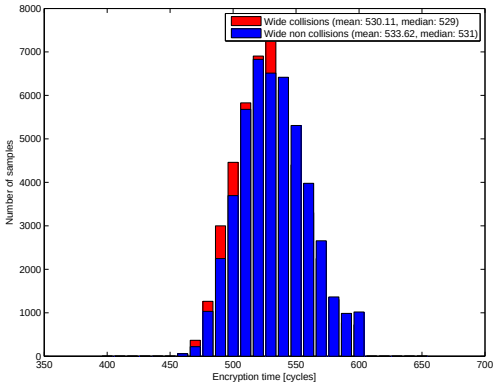


(a) Histogram of encryption times.

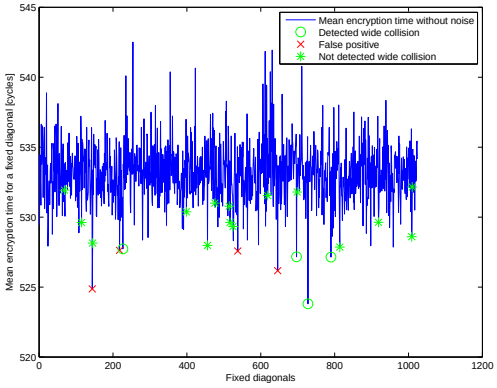


(b) Detection of wide collisions with $n = 6$.

Figure 5.13: Analysis of wide-collision detection for 3-round AES on a Cortex-A8.



(a) Histogram of encryption times.

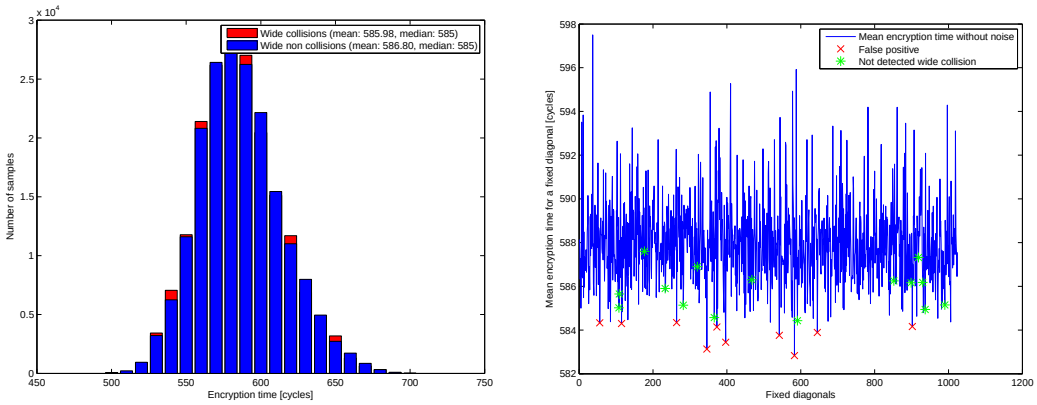


(b) Detection of wide collisions for $n = 8$.

Figure 5.14: Analysis of wide-collision detection for 6-round AES on a Cortex-A8.

analysis exposed this statistical value as a quite eligible decision criteria for the detection of wide collisions. Now we simply consider the $n = 6$ lowest encryption times as possible wide collisions. As Figure 5.13(b) indicates, among the 6 chosen diagonals there are 4 real wide collisions and 2 false positives. Note that 4 real wide collisions are required for a successful attack. In order to visualize the detected wide collisions as well as false positives, we stored all occurring wide collisions during this run.

Experiments on the ARM Cortex-A8 processor revealed that wide collisions can be detected for AES implementations of up to 6 rounds. Though the histogram in Figure 5.14(a) shows slightly lower encryption times for plaintexts where a wide collision occurs, the two categories of plaintexts are harder to distinguish than for the 3-round AES. This in turn increases the number of false positives, which can be observed in Figure 5.14(b). For more than 6 rounds of the AES we are not able to reliably detect wide collisions on the ARM Cortex-A8. As Figure 5.15(a) indicates, the encryption times of plaintexts which lead to wide collisions and plaintexts which do not lead to wide collisions cannot be distinguished anymore. Figure 5.15(b) visualizes the collision detection phase of a run where not even a single wide collision was detected among 10 chosen diagonal pairs. Though



(a) Histogram of encryption times.

(b) Detection of wide collisions for $n = 10$.

Figure 5.15: Analysis of wide-collision detection for 7-round AES on a Cortex-A8.

bad collision-detection mechanisms might be overcome by taking more diagonal pairs into consideration, this drastically increases the complexity of the brute-force key-search phase. Due to the fact that 4 diagonal pairs are necessary in order to solve Equation 5.11, the attack requires to test all combinations of 4 diagonal pairs from the set of diagonal pairs returned from the collision-detection phase. Hence, $\binom{n}{4}$ combinations of diagonal pairs need to be investigated in the first part of the key search phase and for larger values of n this drastically increases the complexity of the remaining brute-force key-search phase.

The main reason for this attack to fail on the Cortex-A8 processor seems to be the additional noise due to the larger cache-line size. Bogdanov et al. [17] attacked an ARM920T processor with a cache-line size of 32 bytes, whereas the ARM Cortex-A8 has a cache-line size of 64 bytes. Simply spoken, on a cache miss the ARM920T only loads 8 table elements into the cache, whereas the Cortex-A8 loads 16 elements into the cache. By changing the data type of the AES table elements, from 4-byte integers to 8-byte integers, we are able to simulate a 32 byte cache-line. Indeed, this setting decreases the number of false positives in the collision-detection phase for the full 10-round AES implementation on the ARM Cortex-A8 processor. Moreover, our investigations on the ARM Cortex-A9 processor, which also has a cache-line size of 32 bytes, confirmed our assumption regarding the larger cache-line size.

Taking probability theory into consideration clarifies the problem of detecting wide collisions on processors with a larger cache-line size. There is a total of $4 \cdot 9$ lookup operations into the same T-table in the rounds 1 to 9. The last round of the AES encryption uses a different T-table, at least in case of the AES implementation shipped with *OpenSSL 0.9.7a*. Equation 5.12 states the probability of δ consecutive table elements of a T-table still not being present in the CPU cache after an entire encryption, where δ denotes the number of table elements per cache line. In case of processors with a cache-line size of 32 bytes each of these lines holds 8 table elements. Hence, the probability of a specific element still not being present after encrypting the first byte of the second plaintext $\mathbf{P}_2$ is 0.3189. However, the ARM Cortex-A8 holds 16 table elements per cache line and hence the probability is 0.092, which is significantly lower. The corresponding probabilities for a specific element already being cached are $1 - 0.3189 = 0.6811$ and $1 - 0.082 = 0.918$, respectively. In other words, the probability for additional cache collisions, besides the required wide collisions, is far greater for the ARM Cortex-A8 than for the ARM920T

and the ARM Cortex-A9. This in turn decreases the overall encryption time of $\mathbf{P}_2$ and makes wide-collisions and non wide collisions nearly indistinguishable according to their encryption time. The above considerations assume that the cache is large enough in order to hold all T-tables, i.e., table elements of T-tables do not evict themselves. In case of the ARM Cortex-A8 and the ARM Cortex-A9 processor this holds true.

$$\begin{aligned} \mathrm{A} &:= \textit{Block of table elements still not in cache after one encryption.} \\ \delta &:= \textit{Number of table elements per cache line.} \\ P(\mathrm{A}) &= \left(1 - \frac{\delta}{256}\right)^{36} \end{aligned} \tag{5.12}$$

Table 5.14 lists the specific parameters, according to our investigations, for a successful detection of wide collisions with a high probability. Note that we do not consider the Google Nexus S in this table since the larger cache-line size of the ARM Cortex-A8 exacerbates the cache-collision attack, i.e., a successful detection of wide collisions seems to be impossible without changing the AES T-table implementation. The parameter X determines the maximum number of cycles an encryption is supposed to take. Encryptions consuming more than X cycles are ignored for the computation of the statistical timing value S_1 (*mean encryption time without noise*), which will be introduced below. The parameters N, I, and R correspond to the parameters mentioned in the attack description above. N determines the number of different diagonal pairs (A, E) chosen. I denotes how often the remaining bytes are chosen and R the number of times each resulting plaintext P_2 is encrypted in order to achieve stable measurement results.

The second and the third part of Table 5.14 deal with the success probabilities regarding the detection of wide collisions. The corresponding statistical values (S_1, ..., S_5) are listed in Table 5.16. These statistical values have been determined empirically and might differ for other devices. Based on the encryption time of plaintext $\mathbf{P}_2$ these statistical values are computed and output in the online phase. Later on these values act as a decision criteria in order to distinguish wide collisions from non wide collisions. n denotes the number of chosen values in the *collision-detection phase*, $n-4$ denotes the number of *false positives* accepted. For the *Acer Iconia A510* we observe a success probability of 75 % if considering $n = 9$ values in the *collision-detection phase*. However, we only observe this success probability if considering nearly all the statistical values mentioned in Table 5.16. Since this would increase the overall complexity of the attack, the more realistic approach would be to choose only one statistical value. In our case, for the Acer Iconia A510, the best results were achieved using the *median of the minimum encryption time* (S_3). Under the assumption that $n = 9$ values are chosen, we achieve a success probability of 38 % in the *collision-detection phase*. For the *Samsung Galaxy SIII* the success probability is even worse.

The problem concerning the detection of wide collisions is clearly indicated in Table 5.15, which stems from a specific run with parameters chosen exactly as stated in Table 5.14. For each of the four diagonals this table shows x/y, where x denotes the number of detected wide collisions among the y diagonal pairs chosen. For the diagonals 2 and 4 there is not even a single false positive among the first 4 diagonal pairs chosen. However, for diagonal 3 we have to choose at least 8 values in order to detect enough wide collisions and, even worse, in case of diagonal 1 we are not able to detect 4 wide collisions among $n = 10$ chosen diagonal pairs. Hence, for this specific run this attack will not succeed.

	Acer Iconia A510	**Samsung Galaxy SIII**
X	1800 – 2000 cycles	1200 – 1800 cycles
N	1024	1024
I	800	1000
R	20/40	20
	$\mathbf{P}_{success}$ considering multiple statistical values.	
	S_1, S_2, S_3, S_4, S_5	S_1, S_2, S_5
$n = 6$	-	-
$n = 7$	13 %	8 %
$n = 8$	25 %	8 %
$n = 9$	75 %	33 %
$n = 10$	75 %	33 %
	$\mathbf{P}_{success}$ considering only one statistical value.	
	S_3	S_5
$n = 6$	-	-
$n = 7$	-	-
$n = 8$	-	-
$n = 9$	38 %	17 %
$n = 10$	38 %	17 %

Table 5.14: Parameters and success probabilities for the devices under attack.

Diagonal	**n = 4**	**n = 5**	**n = 6**	**n = 7**	**n = 8**	**n = 9**	**n = 10**
1	1/4	1/5	1/6	1/7	1/8	2/9	3/10
2	4/4	5/5	6/6	6/7	7/8	7/9	7/10
3	2/4	3/5	3/6	3/7	4/8	5/9	5/10
4	4/4	5/5	5/6	5/7	5/8	6/9	6/10

Table 5.15: Number of detected wide collisions among the number of chosen diagonal pairs.

	Statistical values
S_1	***Mean encryption time without noise.*** For a fixed diagonal pair (A, E) we total the $I \cdot R$ encryption times of P_2, *iff* the encryption time is below a predefined threshold (parameter X in Table 5.14). Afterwards we divide the overall encryption time by the number of considered encryptions.
S_2	***Mean/median of median encryption time.*** For a fixed diagonal pair (A, E) we compute the median of R encryptions of P_2. Later on we consider the mean/median of these I median values as decision criteria.
S_3	***Mean/median of minimum encryption time.*** For a fixed diagonal pair (A, E) we compute the minimum of R encryptions of P_2. The decision criteria is the mean/median of the I minimum encryption times.
S_4	***Mean/median of encryption times without noise.*** For a fixed diagonal pair (A, E) we total the R encryption times of P_2, *iff* the encryption time is below a predefined threshold (parameter X in Table 5.14). The mean/median of these I intermediate values is used to detect wide collisions.
S_5	***Mean/median of the mean of the r-fastest encryptions.*** For a fixed diagonal pair (A, E) we compute the mean of the $r = 5$ fastest encryptions of P_2. The decision criteria is the mean/median of these I intermediate values.

Table 5.16: Most promising statistical values for the detection of wide-collisions.

Phase	$n = 4$	$n = 6$	$n = 9$
Online phase	$\sim 2^{27}$	$\sim 2^{27}$	$\sim 2^{27}$
Key-search phase (find all possible sub-key candidates)	$\sim 2^{29}$	$\sim 2^{33}$	$\sim 2^{36}$
Key-search phase (test all possible sub-key candiates	$\sim 2^{32}$	$\sim 2^{48}$	$\sim 2^{60}$

Table 5.17: Attack complexity given in AES encryptions.

5.6.3 Complexity

Obviously, the more diagonal pairs one considers, the higher becomes the probability to detect four wide collisions among the chosen ones. However, by choosing $n \geq 6$ the complexity of the brute-force step increases drastically. The complexity of the online phase for both plaintexts ($\mathbf{P}_1$ and $\mathbf{P}_2$) is given by $N \cdot I \cdot R \cdot 2$ AES encryptions for each diagonal. Hence, the overall complexity of the online phase for all four diagonals is denoted as $N \cdot I \cdot R \cdot 2 \cdot 4$. In our case, for a successful run — $N = 1\,024$, $I = 1\,000$, $R = 20$ — this yields a complexity of approximately 2^{27} AES encryptions. The complexity of the collision-detection phase can be ignored since one simply chooses n diagonal pairs with the lowest encryption times. The complexity of the last phase, namely the key-search phase, mainly depends on the number of chosen diagonal pairs, denoted as n. Again, this parameter depends on the expectation rate of false positives. If we expect many false positives, n must be higher. Otherwise, n can be reduced to a minimum of $n = 4$ diagonal pairs per diagonal. As Equation 5.11 indicates, only one out of the ten AES rounds is computed, which reduces the complexity by a factor of $\frac{1}{10}$. Furthermore, Bogdanov et al. [17] claim that a key candidate only rarely survives the check of the first diagonal pair and hence they reduce the complexity by a factor of $\frac{1}{4}$. The complexity for the first part of the key-search phase is given by $\binom{n}{4} \cdot 2^{32} \cdot \frac{1}{10} \cdot \frac{1}{4} \cdot 4$. As our experiments indicate, at least $n = 9$ possible wide collisions must be taken into consideration in oder for this attack to be realistic. Hence, the first part of the key-search phase consists of approximately 2^{36} AES encryptions. According to Bogdanov et al. [17] each of the four diagonals yields $256 \cdot \binom{n}{4}$ sub-key candidates. For a choice of $n = 9$ we retrieve 32 256 possible sub-key candidates per diagonal. Since these sub-key candidates have to be enumerated exhaustively this yields a remaining complexity of $32\,256^4 \approx 2^{60}$ AES encryptions. In contrast, considering only $n = 6$ diagonal pairs we retrieve 3 840 possible sub keys, which leads to a remaining complexity of $3\,840^4 \approx 2^{48}$ AES encryptions. Table 5.17 summarizes the complexity of the cache-collision timing attack for different values of n.

Since only the measurement samples must be gathered directly on the device under attack, the remaining attack steps might be done on a more powerful machine. With the advent of AES New Instructions (AES-NI) in the Intel Westmere family a new era in terms of encryption performance began for the AES. According to [7] an Intel Core i7 processor with 6 cores, each clocked at 3.3 GHz, and hyper-threading technology is capable of performing an AES encryption in 3.84 cycles. This results in about 2^{41} AES encryptions per hour, which makes the complexity mentioned above less awkward, at least for $n = 6$.

5.6.4 Summary

Since the *Acer Iconia A510* and the *Samsung Galaxy SIII* both employ the same processor architecture, and even the same operating-system version we would expect both devices to have the same success probability. However, our investigations showed that the *Samsung*

Galaxy SIII has a worse success rate than the *Acer Iconia A510*. In order to get a brief overview of the running tasks and processes on both devices we used the *adb shell* (Android Debug Bride), which is part of the Android SDK. This analysis revealed that there are far more processes running on the *Samsung Galaxy SIII* than on the *Acer Iconia A510*.

Under certain circumstances, i.e., considering multiple statistical values in parallel, we are able to detect wide collisions with $n \geq 6$. However, the brute-force complexity for the remaining key space is too large.

Chapter 6

Conclusions

In this master's thesis we investigated the applicability of cache attacks on today's mobile devices. The need for this work arose from the popularity of mobile devices and their ubiquitous usage scenarios. Since cache attacks have been claimed to be launched successfully on desktop computers and an ARM9 board, the aim of this work was to analyze such attacks according to their applicability on state-of-the-art mobile devices. We strongly focused on real-world environments rather than the commonly stated laboratory constraints. Our investigations are based on mobile devices featuring a fully-functioning operating system. The only requirement for the investigated attacks to work is a rooted smartphone or tablet computer. We analyzed three different approaches aiming at recovering a secret AES key.

In particular, we proposed a new attack approach based on the access-driven attack by Osvik et al. [46]. This approach investigates memory accesses of the AES software implementation. We refer to this approach as *cache-access pattern analysis* since we visualize memory accesses into precomputed AES T-tables. By comparing the gathered memory-access patterns with precomputed access patterns we are able to extract the used secret key. The analysis of cache-access patterns has been shown to be a rather promising attack. Most importantly, the fact that AES T-tables are usually not properly aligned allows us to recover the whole secret key without a subsequent brute-force attack. A proper alignment of T-tables might be achieved by a single statement within the source code. This simple countermeasure is sufficient in order to prevent us from recovering the whole key immediately. However, depending on the cache-line size we are still able to reduce the key space at least from 128 bits to 64 bits. In case of smaller cache-line sizes the key space might be reduced even further, i.e., 48 bits on systems with a cache-line size of 32 bytes. With the introduction of AES New Instructions on modern Intel processors a key space of 48 bits might be searched exhaustively within a few days. Thus, a remaining key space of 48 bits and the fact that we are able to gather the necessary cache-access patterns within just a few minutes imposes a serious threat for today's mobile devices. Therefore, we heavily stress the importance of countermeasures.

Secondly, we investigated the time-driven attack suggested by Bernstein [15]. This attack assumes that an attacker gathers statistical information for many encryptions and organizes these encryptions in two sets. The first set represents the gathered information of encryptions under a known key and the second set represents the gathered information of encryptions under an unknown key. Given enough measurement samples one searches for correlations between the statistical information of these two sets of encryptions and tries to deduce the secret key. We consider this attack to be heavily susceptible to noise

related to processes running in parallel on the system. Thus in our scenarios it did not yield reliable results.

Finally, we also analyzed the time-driven attack suggested by Bogdanov et al. [17]. Though this attack was initially launched on an ARM9 board we analyzed the applicability of this attack on current mobile devices featuring a full operating system. The main idea of their attack is to search for two plaintexts, such that specific state bytes collide, i.e., access the same cache-line. For a cache-line size of 64 bytes we were able to reproduce their attack only for a reduced version of the AES, i.e., 6 rounds. However, for a cache-line size of 32 bytes we were able to reproduce it for the full 10-round AES. Due to the remaining brute-force complexity of approximately 2^{60} AES encryptions we do not consider this attack as a real threat. Nevertheless, it might serve as a strong basis for subsequent attacks.

Though time-driven attacks require far more measurement samples in order to exploit the leaked timing information these kinds of attacks are considered far more realistic and might be applicable for a wide range of mobile devices. The manifold attack scenarios on mobile devices do not necessarily require attacks to run in a minimum of time. For instance, an adversary might be in possession of the mobile device and hence does not care whether the attack takes a few hours or just a few seconds. Supposing a scenario in which one attacks the disk encryption seems to be reasonable.

Overall we conclude that though time-driven attacks require far less knowledge of the device under attack these attacks are more prone to noise, generated by other processes, than access-driven attacks. This results from the fact that time-driven attacks rely on the encryption time alone.

Future research in this area might investigate the applicability of trace-driven attacks on mobile devices. As suggested by Bertoni et al. [16] the usage of performance counters might facilitate an attacker by gathering the necessary memory-access traces. Given the fact that ARM Cortex-A series processors indeed employ such performance monitors this seems to be a rather reasonable approach.

Another important issue might be the implementation of countermeasures. For instance, Cortex-A series processors support a feature called cache lockdown, which prevents the eviction of specific data within the cache. Such techniques might be employed in order to prevent AES T-table elements from being evicted, thus resulting in constant encryption times.

Appendix A

Definitions

A.1 Abbreviations

ADB	Android Debug Bridge
AES	Advanced Encryption Standard
CPU	Central-Processing Unit
DVM	Dalvik Virtual Machine
JNI	Java Native Interface
LRU	Least-Recently Used
MMU	Memory-Management Unit
NDK	Native Development Kit
PMCCNTR	Cycle Count Register
PMCNTENCLR	Count Enable Clear Register
PMCNTENSET	Count Enable Set Register
PMCR	Performance Monitor Control Register
PMUSERENR	User Enable Register
SDK	Software Development Kit
TSC	Intel's Time-Stamp Counter

A.2 Used Symbols

$L1\_S$	L1 cache size
$L1\_B$	L1 cache-line size
$L1\_W$	L1 cache associativity
δ	Maximum number of T-table elements per cache line
γ	Number of cache sets a T-table is supposed to consume

Appendix B

Device Specifications

Acer Iconia A510

Processor	Cortex-A9
Processor implementation	Nvidia Tegra 3 1.4 GHz quad core
Instruction-set architecture	ARMv7
Out-of-order execution	yes
L1 cache size	32 KB
L1 cache associativity	4 way
L1 cache-line size	32 byte
L1 cache sets	256
Operating system	Android 4.0.4

Google Nexus S

Processor	Cortex-A8
Processor implementation	Exynos 3 Single 1 GHz
Instruction-set architecture	ARMv7
Out-of-order execution	no
L1 cache size	32 KB
L1 cache associativity	4 way
L1 cache-line size	64 byte
L1 cache sets	128
Operating system	Android 2.3.4

Samsung Galaxy SIII

Processor	Cortex-A9
Processor implementation	Exynos 4 Quad 1.4 GHz
Instruction-set architecture	ARMv7
Out-of-order execution	yes
L1 cache size	32 KB
L1 cache associativity	4 way
L1 cache-line size	32 byte
L1 cache sets	256
Operating system	Android 4.0.4

Appendix C

Source Code

C.1 Kernel Module

This section outlines the necessary basics in order to compile a kernel module for the Android platform. Sylve [45] provides a great tutorial on how to compile kernel modules for Android. We followed his instructions in order to build our kernel module successfully.

For the purpose of cache attacks a kernel module, as outlined in Listing C.1, is required in order to grant user-space applications access to the cycle-count register. Compiling kernel modules requires the appropriate kernel sources of the device under attack. Fortunately, the kernel sources of Android-based mobile devices can be found on the website of the manufacturer. Otherwise, the use of any state-of-the-art search engine should reveal the location of the correct kernel sources.

Figure C.1 outlines the basic steps in order to load a kernel module on an Android device. In case `insmod` returns an error the `dmesg` command might reveal further information about the problem. The next few paragraphs briefly outline the necessary steps for compiling the kernel module for different mobile devices. Furthermore, we outline encountered problems and their solutions.

Listing C.1: Kernel module to enable the cycle-counter register within the user space on single-core devices.

```
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_DESCRIPTION("Enables the ccnt register within user-space applications");
MODULE_LICENSE("GPL");

int init_module()
{
  /* Enable user-mode access to the performance counter */
  asm volatile("mcr p15, 0, %0, c9, c14, 0" :: "r"(1));
  return 0;
}

void cleanup_module()
{
  /* Disable user-mode access to the performance counter */
  asm volatile("mcr p15, 0, %0, c9, c14, 0" :: "r"(0));
}
```

Google Nexus S. In case of the *Google Nexus S* the kernel source can be found at `https://android.googlesource.com/kernel/samsung.git`. After checking out the required kernel version, in our case *android-samsung-2.6.35-gingerbread*, one should be able to compile kernel modules for the *Google Nexus S* as outlined in Listing C.2.

```
user@system:~/android-sdk-linux/platform-tools$ ./adb push /path/to/kernel-module/enableccnt.ko /sdcard/enableccnt.ko
558 KB/s (24932 bytes in 0.043s)
user@syste:~/android-sdk-linux/platform-tools$ ./adb shell
shell@android:/ $ su
shell@android:/ # insmod /sdcard/enableccnt.ko
shell@android:/ # lsmod
enableccnt 504 0 - Live 0xbf028000
btlock 1584 0 - Live 0xbf024000
Si4709_driver 16195 0 - Live 0xbf01c000
exfat_fs 17079 0 - Live 0xbf013000 (P)
exfat_core 59615 1 exfat_fs, Live 0xbf000000 (P)
shell@android:/ #
```

Figure C.1: Load kernel module on an Android device.

Trying to load the kernel module on the *Google Nexus S* yielded the following error message: *insmod: init_module '/sdcard/enableccnt.ko' failed (Exec format error)*. Furthermore, the `dmesg` output returned the error message: *enableccnt: version magic '2.6.39.4 SMP preempt mod_unload ARMv7' should be '2.6.39.4+ SMP preempt mod_unload ARMv7'*. As suggested by Hommey [27], we solved this problem by creating a file `.scmversion` with the content `-ge382d80` within the kernel-source folder. The appropriate content can be extracted from `/proc/version` on the *Google Nexus S*. Afterwards we compiled and loaded the kernel module again. This time it worked properly.

Listing C.2: Building the kernel module for the Google Nexus S.

```
#!/bin/bash -
SDK_ROOT='%path to your android sdk root folder%'
NDK_ROOT='%path to your android ndk root folder%'
KERNEL_SOURCE='%path to your Google Nexus S kernel source folder%'
KERNEL_MODULE='%path to kernel module%'

# locate the arm crosscompiler toolchain
TOOLCHAIN=$(find ${NDK_ROOT} -name "*androideabi-gcc" | sed 's/gcc$//')

# create configuration for the Google Nexus S
cd $KERNEL_SOURCE
make ARCH=arm herring_defconfig CROSS_COMPILE=${TOOLCHAIN} modules_prepare

# build the kernel module
make ARCH=arm CROSS_COMPILE=${TOOLCHAIN} -C ${KERNEL_SOURCE} M=${KERNEL_MODULE} modules

# copy the kernel module to the smartphone
$SDK_ROOT/platform-tools/adb push ${KERNEL_MODULE}/enableccnt.ko /sdcard/enableccnt.ko
```

Acer Iconia A510. In case of the *Acer Iconia A510* we found the corresponding source code at `http://support.acer.com`. Since this mobile device employs a multi-core processor we use a slightly different kernel module. As outlined in Listing C.3 we ensure that the cycle-count register is enabled on a all active CPUs. Since usually only the CPU 0 is active the cycle-count register will be enabled only on this CPU. Obviously, the attack application must be pinnend to the same CPU in order to prevent this process from being executed on a different core and thus from failing to read the cycle-count register. In order to pin the attack application to a specific CPU we use the *syscall* `sched_setaffinity`.

Listing C.4 outlines the basic steps in order to compile the kernel module. Preparing the kernel source for building the kernel module requires a `.config` file. In case of the *Acer Iconia A510* this file can be found on the device (`/proc/config.gz`). Hence, we simply copy this file to the extracted kernel-source folder and continue by calling the make target `module_prepare` with the right path to the toolchain used for the Android platform, followed by the corresponding target used to compile the kernel module itself. Compiling and loading the kernel module on the *Acer Iconia A510* did not cause any problems.

Listing C.3: Kernel module to enable the cycle-counter register within the user space on multi-core devices.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/smp.h>

MODULE_DESCRIPTION("Enables the ccnt register within userspace");
MODULE_LICENSE("GPL");

static void enable(void *unused)
{
  asm volatile("mcr p15, 0, %0, c9, c14, 0" :: "r"(1));
}

int init_module()
{
  unsigned int value;

  /* Call function that enables ccnt register on all active CPUs. */
  on_each_cpu(enable, NULL, 0);
  return 0;
}

void cleanup_module()
{
  /* Disable user-mode access to the performance counter */
  asm volatile("mcr p15, 0, %0, c9, c14, 0" :: "r"(0));
}
```

Listing C.4: Building the kernel module for the Acer Iconia A510.

```
#!/bin/bash -
SDK_ROOT='%path to your android sdk root folder%'
NDK_ROOT='%path to your android ndk root folder%'
KERNEL_SOURCE='%path to your Acer Iconia A510 kernel source folder%'
KERNEL_MODULE='%path to kernel module%'

# locate the arm crosscompiler toolchain
TOOLCHAIN=$(find ${NDK_ROOT} -name "*androideabi-gcc" | sed 's/gcc$//')

# retrieve the configuration file directly from the tablet
$SDK_ROOT/platform-tools/adb pull /proc/config.gz
gunzip config.gz
cp config $KERNEL_SOURCE/.config

# prepare the kernel sources
cd $KERNEL_SOURCE
make ARCH=arm CROSS_COMPILE=${TOOLCHAIN} modules

# finally build the kernel module itself
make ARCH=arm CROSS_COMPILE=${TOOLCHAIN} -C ${KERNEL_SOURCE} M=${KERNEL_MODULE} modules

# and copy the kernle module to the tablet computer
$SDK_ROOT/platform-tools/adb push ${KERNEL_MODULE}/enableccnt.ko /sdcard/enableccnt.ko
```

Samsung Galaxy SIII. The appropriate kernel sources for the *Samsung Galaxy SIII* can be found at `http://opensource.samsung.com`. After extracting the kernel-source archive one simply uses the bash script outlined in Listing C.5 in order to prepare the kernel source and compile the kernel module outlined in Listing C.1.

In case of the *Samsung Galaxy SIII* the appropriate `.config` file could not be located on the device. Fortunately, the `README` file shipped with the kernel source outlines how to generate the `.config` file for the *Samung Galaxy SIII* kernel. Though in general it is possible to compile kernel modules without compiling the whole kernel source, we ran into troubles when trying to load the kernel module on the device. The command `insmod enableccnt.ko` returned the following error message: *insmod: ini_module '/sdcard/enableccnt.ko' failed (Exec format error)*. Furthermore, the `dmesg` command returned *enableccnt: no symbol version for module_layout*. In our case building the whole kernel source, as outlined in Listing C.5, did the trick and we were able to load the resulting kernel module successfully.

Listing C.5: Building the kernel module for the Samsung Galaxy SIII.

```
#!/bin/bash -
SDK_ROOT='%path to your android sdk root folder%'
NDK_ROOT='%path to your android ndk root folder%'
KERNEL_SOURCE='%path to your Samsung Galaxy SIII kernel source folder%'
KERNEL_MODULE='%path to kernel module%'

# locate the arm crosscompiler toolchain
TOOLCHAIN=$(find ${NDK_ROOT} -name "*androideabi-gcc" | sed 's/gcc$//')

# create configuration for Samsung Galaxy SIII
cd $KERNEL_SOURCE
make ARCH=arm m0_00_defconfig

# build complete kernel
make CROSS_COMPILE=${TOOLCHAIN}

# build the kernel module
make ARCH=arm CROSS_COMPILE=${TOOLCHAIN} -C ${KERNEL_SOURCE} M=${KERNEL_MODULE} modules

# and copy the kernel module to the smartphone
$SDK_ROOT/platform-tools/adb push ${KERNEL_MODULE}/enableccnt.ko /sdcard/enableccnt.ko
```

Bibliography

[1] ARM Ltd. `http://www.arm.com/`, 2012.

[2] ARM Ltd.: ARM Cortex A8. `http://www.arm.com/products/processors/cortex-a/cortex-a8.php`, 2012.

[3] ARM Ltd.: ARM Cortex A9. `http://www.arm.com/products/processors/cortex-a/cortex-a9.php`, 2012.

[4] Qualcomm Inc. `http://www.qualcomm.com/`, 2012.

[5] O. Aciiçmez and Çetin Kaya Koç. Trace-Driven Cache Attacks on AES. *IACR Cryptology ePrint Archive*, 2006:138, 2006.

[6] D. Agrawal, B. Archambeault, J. Rao, and P. Rohatgi. The EM SideChannel(s). In *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 29–45. Springer Berlin / Heidelberg, 2003.

[7] K. Akdemir, M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, and R. Zohar. Breakthrough AES Performance with Intel AES New Instructions. *Intel Corporation*, 2010. White Paper.

[8] Android. Android Developer's Guide. `http://developer.android.com/`, 2012.

[9] Android. Dalvik Technical Information. `http://source.android.com/tech/dalvik/index.html`, 2012.

[10] ARM. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R ed., ARM DDI 0406 A*, Apr. 2007.

[11] ARM. *ARM Technical Reference Manual, Cortex-A8, Revision: r3p2, ARM DDI 0344K*, May 2010.

[12] ARM. *Cortex-A Series Programmer's Guide, Version: 2.0*, Aug. 2011.

[13] ARM. *ARM Technical Reference Manual, Cortex-A9, Revision: r4p0, ARM DDI 0388H*, 2012.

[14] A. Becker and M. Pant. *Android: Grundlagen und Programmierung*. dpunkt Verlag, 1 edition, May 2009.

[15] D. J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.

[16] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo. AES Power Attack Based on Induced Cache Miss and Countermeasure. In *ITCC (1)*, pages 586–591, 2005.

[17] A. Bogdanov, T. Eisenbarth, C. Paar, and M. Wienecke. Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs. In J. Pieprzyk, editor, *Topics in Cryptology - CT-RSA 2010*, volume 5985 of *Lecture Notes in Computer Science*, pages 235–251. Springer Berlin / Heidelberg, 2010.

[18] P. Brady. Android Anatomy and Physiology. `http://sites.google.com/site/io/anatomy--physiology-of-an-android`. Google I/O Session 2008, Presentation Slides.

[19] Cryptome. The Complete, Unofficial TEMPEST Information Page. `http://cryptome.org/#NSA--TS`, 2012.

[20] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Berlin Heidelberg, 1 edition, Feb. 2002.

[21] A. Davison. *Killer game programming in Java*. O'Reilly Media, Inc., 2005.

[22] W. V. Eck and N. Laborato. Electromagnetic Radiation from Video Display Units: An Eavesdropping Risk? *Computers & Security*, 4:269–286, 1985.

[23] R. Grisenthwaite. Presentation Slides: Cortex A8 Processor. `esd.et.ntust.edu.tw/downloads/2012_embeddedApplication/ARMArchiRichardGrisenthwaite.pdf`, 2012. ARM Ltd.

[24] J. Handy. *The Cache Memory Book*. Morgan Kaufmann, Jan. 1998.

[25] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4 edition, Sept. 2006.

[26] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 5 edition, Sept. 2011.

[27] M. Hommey. Building a custom kernel for the Nexus S. `http://glandium.org/blog/?p=2214`, 2012.

[28] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*, Dec. 2011.

[29] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z*, Dec. 2011.

[30] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide*, Dec. 2011.

[31] B. Jacob. Cache design for embedded Real-Time systems. June 1999.

[32] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in CryptologyCRYPTO99*, page 789789, 1999.

[33] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. 1109:104–113, 1996.

[34] F. Koeune and F.-X. Standaert. A Tutorial on Physical Security and Side-Channel Attacks. In *Foundations of Security Analysis and Design III : FOSAD 2004/2005*, volume 3655 of *Lecture Notes in Computer Science*, pages 78–108, 11 2006.

[35] M. G. Kuhn. Compromising emanations: eavesdropping risks of computer displays. Technical Report UCAM-CL-TR-577, University of Cambridge, Computer Laboratory, Dec. 2003.

[36] N. Lawson. Side-Channel Attacks on Cryptographic Software. *IEEE Security & Privacy*, 7(6):65–68, Dec. 2009.

[37] M. Neve. *Cache-based Vulnerabilities and SPAM Analysis*. PhD thesis, UCL, June 2006.

[38] NIST. *Advanced Encryption Standard (AES) (FIPS PUB 197)*. National Institute of Standards and Technology, Nov. 2001.

[39] M. O'Hanlon and A. Tonge. Investigation of Cache Timing Attacks on AES, 2005.

[40] Open Handset Alliance. `http://www.openhandsetalliance.com/`, 2012.

[41] OpenSSL Software Foundation. OpenSSL Project. `http://www.openssl.org/`, 2012.

[42] G. Paoloni. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. *Intel Corporation*, 2010. White Paper.

[43] J.-J. Quisquater and D. Samyde. ElectroMagnetic Analysis (EMA): Measures and Countermeasures for Smart Cards. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer Berlin / Heidelberg, 2001.

[44] S. P. Skorobogatov and R. J. Anderson. Optical Fault Induction Attacks. In *CHES*, pages 2–12, 2002.

[45] J. Sylve. Guide to Compiling Custom Kernel Modules in Android. `http://yatsec.blogspot.co.at/2011/01/guide-to-compiling-custom-kernel.html`, Jan. 2011.

[46] E. Tromer, D. A. Osvik, and A. Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 23:37–71, July 2009.

[47] R. Williams. *Computer Systems Architecture: A Networking Approach*. Prentice Hall, 2 edition, Aug. 2006.

[48] P. Wright. *Spycatcher*. William Heinemann, 1987.