Cache-Access Pattern Attack on Disaligned AES T-Tables

Raphael Spreitzer and Thomas Plos

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria
{raphael.spreitzer,thomas.plos}@iaik.tugraz.at

Abstract. Cache attacks are a special form of implementation attacks and focus on the exploitation of weaknesses in the implementation of a specific algorithm. We demonstrate an access-driven cache attack, which is based on the analysis of memory-access patterns due to the T-table accesses of the Advanced Encryption Standard (AES). Based on the work of Tromer *et al.* [20] we gather the cache-memory access patterns of AES T-table implementations and perform a pattern-matching attack in order to recover the used secret key. These T-tables usually do not start at memory addresses which are mapped to the beginning of a specific cache line. Thus, focusing on disaligned AES T-tables allows us to recover the whole secret key by considering only the first round of the AES. We apply the presented cache attack on a Google Nexus S smartphone, which employs a Cortex-A8 processor and runs a fully-functioning operating system. The attack is purely implemented in software and the only requirement is a rooted mobile device. To the best of our knowledge, we are the first to launch an access-driven attack on an ARM Cortex-A processor. Based on our observations of the gathered access patterns we also present an enhancement, which in some cases allows us to recover the secret key without a subsequent brute-force key search.

Keywords: AES, ARM Cortex-A8, disaligned AES T-tables, memory-access pattern attack.

1 Introduction

Mobile devices have become an ubiquitous part of our everyday life. The wide-spread usage of these devices inevitably leads to private and sensitive information being stored on them. In order to protect these data against adversaries cryptographic primitives and cryptographic protocols are implemented. Security and privacy enhancing methodologies include, for instance, disk encryption and the encrypted communication over the Internet. Standardized cryptographic algorithms, *i.e.*, the Advanced Encryption Standard (AES), are applied to cope with these challenges.

Though, the AES is considered to be mathematically secure, a specific implementation of this algorithm does not necessarily have to be secure. Implementation attacks explicitly focus on the exploitation of implementation weaknesses by considering information leaking through side channels, *e.g.*, variations of the execution time or the power consumption based on different inputs. A special form of these implementation attacks

E. Prouff (Ed.): COSADE 2013, LNCS 7864, pp. 200–214, 2013.
The final publication is available at link.springer.com.

are cache attacks, which aim at the exploitation of different access times resulting from the fact that data within the central-processing unit (CPU) cache can be accessed an order of magnitude faster than data located within the main memory.

Cache attacks can be classified into three main categories: (1) *time-driven attacks*, (2) *access-driven attacks*, and (3) *trace-driven attacks*. All three types aim at recovering the secret key of a specific cryptographic implementation and only differ in the amount of information available to recover the key. While time-driven attacks focus on the investigation of the overall encryption time, access-driven and trace-driven attacks require a more fine-grained knowledge about memory accesses and the corresponding cache hits and misses. Cache attacks have been launched successfully on a variety of desktop computers [6, 9, 11, 20, 25] and, recently, also the investigation on mobile and embedded devices has started [8, 10, 23]. However, these investigations focus either on microcontrollers or on devices that do not feature a full-blown operating system.

The rising popularity of mobile devices in our everyday life clearly states the need for the investigation of such cache attacks on modern mobile devices in a realistic scenario. In this work, we close this gap and present an access-driven cache attack based on the analysis of cache-access patterns. We also practically apply the presented attack on a *Google Nexus S* smartphone, which is an Android-based smartphone with an ARM Cortex-A8 processor. Moreover, we focus on the exploitation of information leaked through disaligned AES T-tables. According to our knowledge, we are the first to present an access-driven attack on ARM Cortex-A series processors, which employ a random-replacement policy.

The presented paper is organized as follows. In Section 2 we outline related work in the field of cache attacks. We introduce the required preliminaries and notations in Section 3. Section 4 denotes the main part of this paper and presents our attack approach. In Section 5 we state an observation which allows an attacker to further reduce the remaining key space. Section 6 illustrates the results of the proposed attack on the ARM Cortex-A8 processor. Finally we conclude this work in Section 7.

2 Related Work

In 1996, Kocher [13] demonstrated the exploitation of timing information in order to recover secret keys of cryptographic implementations. He also claimed that—due to compiler optimizations, RAM cache hits, and many other factors—timing-independent implementations are extremely difficult to achieve. Since Kocher's groundbreaking work implementation attacks evolved enormously. One specific form of implementation attacks are cache attacks, which can be separated into three categories: *time-driven attacks*, *access-driven attacks*, and *trace-driven attacks*.

Time-Driven Attacks. These attacks require only minor knowledge of the implementation and the hardware architecture under attack. Depending on the provided input the implementation might leak different timings. Thus, the basic idea of time-driven attacks is to gather timing information of many encryptions and to perform statistical correlations in order to recover the used secret key. Attacks in this category typically require far more measurement samples than attacks within the following two categories.

Access-Driven Attacks. The purpose of these attacks is to determine which cache lines or cache sets have been accessed during the encryption. Hence, knowledge of the location of the precomputed S-Boxes or T-tables within the memory as well as information about the cache architecture is necessary. However, fewer measurement samples are necessary than in case of time-driven attacks.

Trace-Driven Attacks. For these attacks a detailed cache profile based on the information of every single memory access is necessary, *i.e.*, for every look-up operation an attacker knows whether it resulted in a cache hit or a cache miss. As suggested by Aciiçmez and Koç [1] performance counters of modern CPUs might be used to establish such a memory-access profile.

The exploitation of information leaked through cache-memory access times started in the year 2000. Kelsey *et al.* [12] explicitly suggested the exploitation of cache-hit ratios of cryptographic implementations that employ large S-boxes. Followed by Page [18] and Tsunoo *et al.* [21, 22] the exploitation of cache-based side-channel information of the Data Encryption Standard (DES) began. With the introduction of the AES increased attention has been given to the development of cache attacks against this symmetric cipher. For instance, Bertoni *et al.* [7] simulated a first-round attack on the AES by inducing cache misses and using power traces to determine when and where these cache misses occurred. Lauradoux [14] presented a collision attack on the AES based on power traces. In 2006, Bonneau and Mironov [9] also presented a cache-collision attack. However, their approach was based on timing information. The aim of cache-collision attacks is to deduce linear relations between look-up indices due to collisions between these indices. Bernstein [6] followed a similar approach and investigated the overall encryption time. Therefore, he correlated the encryption times of plaintexts under a known AES key with the encryption times under an unknown AES key.

Tromer *et al.* [20] (first presented in 2005) paved the way for access-driven attacks. They suggested two approaches in order to gather cache-memory accesses: (1) *Prime and Probe* as well as (2) *Evict and Time*. The approach of *Prime and Probe* is to detect accessed cache sets through the investigation of memory accesses within the attacker's address space. In contrast, *Evict and Time* tries to determine the accessed cache sets through the encryption time. Neve and Seifert [16] suggested the investigation of ciphertexts and the corresponding cache accesses related to the AES T-table used within the last round. They adapted the elimination and non-elimination method suggested by Tsunoo *et al.* [21]. Zhao *et al.* [25] also presented an access-driven attack on the first and the second round of the AES encryption, respectively. Based on the *Prime and Probe* approach of Tromer *et al.* [20] they gathered unaccessed cache sets and reduced the set of possible keys. Overall, they claimed to reduce the key space from 128 bits to 18 bits with 350 AES encryptions. Though, they state that disaligned T-tables leak more information about the key bits, they also consider the disalignment of AES T-tables as a complication. Furthermore, Zhao *et al.* investigated the disalignment of S-Boxes in an access-driven attack on *Camellia* [26] and a simulated trace-driven attack on *CLEFIA*, and AES [24]. In 2011, Gullasch *et al.* [11] suggested the exploitation of the Linux scheduler to gather memory accesses of a victim process that performs AES encryptions. ARM7 microcontrollers and ARM Cortex-A8 processors have been attacked recently by Bogdanov *et al.* [8], Gallais and Kizhvatov [10], and Weiß *et al.* [23].

However, these attacks represent trace-driven as well as time-driven approaches and do not consider a full-blown operating system. In this paper we focus on ARM Cortex-A processors running a fully-functioning operating system.

3 Requirements and Preliminaries

In this section we briefly outline the required preliminaries in order to launch the presented cache-access pattern attack on the ARM Cortex-A8 processor. Firstly, we introduce the ARM architecture with a focus on the performance-monitor registers and the cache-memory architecture. Secondly, we outline the basics of the AES and finally we cover necessary notations which are used throughout this paper.

3.1 ARM Cortex-A Series Processors

Currently, most mobile platforms employ 32-bit ARM processors. For instance, the ARM Cortex-A series processor [5] was designed explicitly for mobile devices with limited power resources and hence this processor architecture is the most commonly used architecture in today's smartphones and tablet computers. In order to overcome the gap between the high CPU clock frequencies and the slow main-memory access times, ARM Cortex-A processors also employ CPU caches. CPU caches are used to hold recently used data and data probably to be accessed in the near future close to the CPU. Since the main memory is usually larger than the cache memory, a mapping has to be established. For instance, the most commonly employed cache-mapping technique is the set-associative mapping. A set-associative cache is divided into equally sized cache sets, each consisting of 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. An algorithm decides which data is to be replaced, *i.e.*, evicted from the cache in order to free up a cache line for new data. ARM processors usually implement a random-replacement policy, which means that data is evicted randomly from the cache. In contrast, most modern desktop processors implement a deterministic-replacement policy.

The investigated ARM Cortex-A8 processor is a single-core processor with a clock frequency of 1 GHz. It employs a 4-way set associative L1 cache with a cache-line size of 64 bytes and a total size of 32 KB. According to the *ARM Architecture Reference Manual* [2], the Cortex-A series also features performance-monitor registers, implemented within a special coprocessor. These registers are capable of counting different types of events. For our purposes, the only register of concern is the *Cycle Count Register (PMCCNTR)*, which is a 32-bit register that counts core clock cycles. This register allows us to measure the execution times of AES encryptions with a sufficiently high resolution in order to distinguish T-table accesses within the cache memory from T-table accesses within the main memory. Unfortunately, it is accessible in privileged mode only, except for the case that access to this register is explicitly granted to unprivileged processes. Hence, a kernel module might be loaded by the attack application in order to allow unprivileged applications to access this register. For further information about the *Cycle Count Register* and how to enable this register to be accessible by unprivileged applications, we refer to [2, 3, 4].

3.2 Advanced Encryption Standard

In 2000, the National Institute of Standards and Technology (NIST) announced Rijndael, designed by J. Daemen and V. Rijmen, as the Advanced Encryption Standard (AES) [15]. The AES is a block cipher operating on a 128-bit state, denoted as a series of bytes $\mathbf{s} = \{\mathbf{s}_0, \ldots, \mathbf{s}_{15}\}$, which is usually represented as a matrix of four columns and four rows, respectively. Basically, the AES consists of four round transformations: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. After the initial *AddRoundKey* transformation $n_r - 1$ rounds of the before mentioned round transformations are applied, followed by the final round n_r that simply omits the *MixColumns* transformation. The number of rounds n_r to be performed depends on the key length and in case of AES-128 the number of rounds is defined to be $n_r = 10$.

Software implementations of the AES usually employ look-up tables in order to overcome the computation of the complex round transformations and to improve the performance of this block cipher. These precomputed look-up tables combine the round transformations, excluding *AddRoundKey*, and facilitate the implementation of the encryption and decryption algorithm through simple look-up operations in combination with XOR operations. Equation 1 illustrates the software implementation of the AES using look-up tables. The initial state bytes are computed as $\mathbf{s}_i^0 = \mathbf{p}_i \oplus \mathbf{k}_0^0$, with $\mathbf{p}$ representing the plaintext and $\mathbf{k}^0$ the initial round key. The look-up tables are denoted as $\mathbf{T}_0, \ldots, \mathbf{T}_3$ and consist of 256 4-byte elements, resulting in a total size of 1 KB for each T-table. Starting with the initial state byte $\mathbf{s}_i^0$ the computations outlined in Equation 1 are performed $n_r - 1$ times. Due to the fact that the *MixColumns* transformation is omitted within the last round, other T-tables are used within the last round. The resulting state $\mathbf{s}$, after performing the last round, represents the ciphertext. The combination of large look-up tables and key-dependent look-up indices makes the AES highly prone to cache attacks. Due to reasons of simplicity we refer to the initial state bytes as $\mathbf{s}_i = \mathbf{p}_i \oplus \mathbf{k}_i$ within the following sections.

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

We launched the presented attack against the T-table implementation of *OpenSSL 0.9.7a* [17], though newer versions, for instance, *OpenSSL 1.0.1c* might neither resist this attack since the T-table implementation is typically the same. The only difference is that the last round might either use the T-tables $\mathbf{T}_0, \ldots, \mathbf{T}_3$ in a slightly adapted way or a separate T-table $\mathbf{T}_4$.

3.3 Notations and Definitions

In this subsection we introduce the used notations and definitions. In particular, we outline the notion of disaligned AES T-tables.

Definition 1. According to Tromer *et al.* [20] we denote the maximum number of T-table elements per cache line as δ. Supposing a cache-line size of 64 bytes and a table-element size of 4 bytes $\delta = \frac{64}{4} = 16$.

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

We consider an AES T-table to be aligned if it starts at a memory address which is mapped to the beginning of a cache line. In this case the first cache line related to a specific AES T-table holds exactly the first δ table elements, which is the optimum. However, in practice we observe disaligned T-tables. This means that the memory address of a precomputed look-up table does not correspond to the beginning of a cache line. Hence, the first cache line related to a specific T-table holds less than δ table elements. The investigated mobile device revealed that even a disalignment of $\delta - 1$ is possible, *i.e.*, the first or the last cache line holds only one element, which might (ideally) reveal the whole secret key without a subsequent brute-force attack. As opposed to the statement by Rebeiro *et al.* [19] that such a disalignment of T-tables must be explicitly forced by the programmer, we observed disaligned T-tables though not explicitly forced. This might be a result of the used compiler, *i.e.*, *arm-linux-androideabi-gcc (GCC) 4.4.3* in our case. A proper alignment of AES T-tables can be achieved by declaring the T-tables as `__attribute__((aligned(64))) static const uint32_t Te0[256]`. In general aligning T-tables appropriately can be done by specifying a number which is a multiple of the cache-line size.

In case the T-table is properly aligned the number of recoverable key bits per key byte is limited to the upper $8 - \log_2 \delta$ bits, *i.e.*, the upper 4 bits in case of a cache-line size of 64 bytes. Observe that the look-up indices within the first round of the AES are denoted as $\mathbf{s}_i = \mathbf{p}_i \oplus \mathbf{k}_i$ for byte i. In case of $\delta = 16$ the first cache set contains the corresponding T-table elements of the look-up indices $\mathbf{s}_i \in \{\texttt{0x00}, \ldots, \texttt{0x0F}\}$. 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 at least the upper four bits of the secret key $\mathbf{k}_i$. In practice we usually observed disaligned T-tables, which means that the first cache line related to a specific T-table holds less than δ elements and hence more key bits might be recovered.

Definition 3. We denote the number of T-table elements within the first cache line related to a specific T-table as $\alpha \in \{1, \ldots, \delta\}$.

4 Attack Concept

As the name already suggests, the presented attack belongs to the class of access-driven attacks. Our attack approach focuses on the exploitation of disaligned AES T-tables and we demonstrate that disaligned T-tables are especially vulnerable, *i.e.*, in some cases allow an attacker to recover the whole secret key without a single brute-force computation. The attack is purely implemented in software and the only prerequisite for our attack to work is a rooted mobile device to load the kernel module, which permits access

to the *Cycle Count Register* to unprivileged applications by setting the appropriate bit within a control register. Observe that the attack itself does not require root access since setting the appropriate bit—in order to allow unprivileged applications to access the *Cycle Count Register*—might be done within a separate application. Furthermore, this bit only needs to be set once after powering up the device. The basic idea is to gather memory-access patterns of AES encryptions and to match them against precomputed patterns in order to gain knowledge of the used secret key.

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. Due to reasons of simplicity we implemented the AES encryption directly within the attack application. The chosen plaintext ensures the encryption of a specific byte value when attacking the corresponding key byte. Furthermore, we assume to be able to measure the encryption time with a sufficiently high resolution, *i.e.*, to distinguish main-memory accesses from cache-memory accesses, and to be able to gather the memory-access patterns in an appropriate way. Tromer *et al.* [20] suggest two different approaches in order to gather the memory-access patterns: (1) *Prime and Probe*, and (2) *Evict and Time*. Though they state *Prime and Probe* as being extremely efficient, we employ the *Evict and Time* approach, for we consider it more appropriate for caches with a random-replacement policy. Recall that the random-replacement policy makes it more difficult to initialize a whole cache set to specific elements, which is necessary for the *Prime and Probe* approach (accessing a specific element might evict a previously loaded one). Thus, further investigations might be done to reveal whether the *Prime and Probe* approach also leads to more efficient attacks on systems with a random-replacement policy. The idea of the *Evict and Time* approach is to observe cache evictions based on the encryption time itself. Therefore, the attacker allocates a data structure which is as large as the L1 data cache[1]. The attacker starts by triggering the encryption of a plaintext $\mathbf{p}$ and afterwards evicts a specific cache set by accessing the appropriate elements within the allocated data structure. Finally, by measuring the encryption time of the same plaintext $\mathbf{p}$ again, the attacker might determine whether a cache set required for the encryption of plaintext $\mathbf{p}$ has been evicted or not.

The attack is composed of the following steps: *(1) gather cache-access patterns*, *(2) extract a pattern vector*, *(3) compute possible access patterns*, *(4) extract possible key candidates*, and optionally *(5) perform a brute-force key search on the remaining key space*. The only step which must be executed on the attacked device is step (1). Hence, we also refer to this step as online phase. The following subsections outline these steps in more detail.

4.1 Gather Cache-Access Patterns

In order to gather the memory-access patterns for all key bytes $\mathbf{k}_{i|i\in\{0,\ldots,15\}}$ we set $\mathbf{p}_i = \texttt{0x00}$, choose the rest of the plaintext randomly, and perform the following steps. First,

[1] In order to ensure the eviction of a specific cache set with an appropriate probability we use a data structure which is 3 times the size of the L1 cache. In this case the probability for a specific cache line still being present within a cache set after accessing all 12 elements which map to this cache set is $\left(\frac{3}{4}\right)^{12} = 0.0318$.

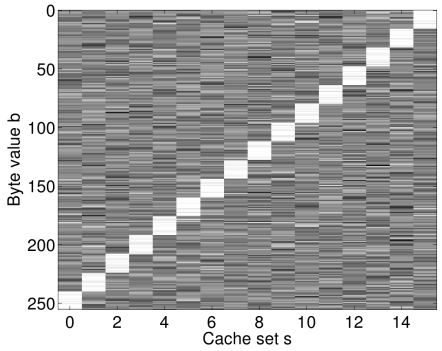

Fig. 1. Gathered measurement score (original) for $\mathbf{k}_5 = \texttt{0xF3}$. Brighter areas represent slower encryptions after the eviction of the corresponding cache set.

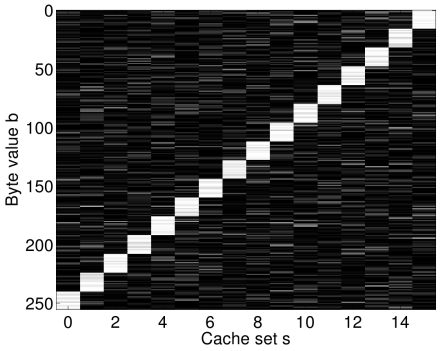

Fig. 2. Normalized measurement score for $\mathbf{k}_5 = \texttt{0xF3}$. Brighter areas represent slower encryptions after the eviction of the corresponding cache set.

we encrypt the chosen plaintext $\mathbf{p}$ to load the necessary T-table elements into the data cache and the corresponding instructions into the instruction cache. This step is referred to 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[2]. Note 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 $b \in \{\texttt{0x00}, \ldots, \texttt{0xFF}\}$ representing all possible byte values $\mathbf{p}_i$ might take, and $s \in \{0, \ldots, \gamma - 1\}$ representing the evicted cache set of T-Table $\mathbf{T}_j$. 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 $\mathbf{p}_i = \texttt{0xFF}$.

More formally, for each possible plaintext byte $\mathbf{p}_{i|i\in\{0,\ldots,15\}} \in \{\texttt{0x00}, \ldots, \texttt{0xFF}\}$ of the plaintext $\mathbf{p}$ we establish a data structure $\mathbf{t}_i[b][s]$. The purpose of this data structure is to store 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 decreased, *i.e.*, a cache miss occurred. Figure 1 illustrates an example of such a data structure after performing the above outlined steps for a specific key byte, *e.g.*, $\mathbf{k}_5$ with $R = 150$ iterations in this case. Figure 2 illustrates the normalized measurement score with the mean of the corresponding column subtracted from each value. The vertical axis shows all possible bytes

[2] We assume this information to be known in order to simplify the explanation. Basically, this information might be retrieved in a simple pre-processing stage.

the plaintext byte $\mathbf{p}_5$ might take and the horizontal axis illustrates the evicted cache set s. 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. On our test device we have 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 corresponding index. Figure 1 illustrates that given a plaintext byte $\mathbf{p}_5 \in \{\texttt{0xF0}, \ldots, \texttt{0xFF}\}$, and a key byte $\mathbf{k}_5 = \texttt{0xF3}$ 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 1 only reveals the upper 4 bits of the key byte, *e.g.*, $\mathbf{k}_5 \in \{\texttt{0xF0}, \ldots, \texttt{0xFF}\}$.

4.2 Extract a Pattern Vector

From the measurement scores $\mathbf{t}_i[b][s]$ gathered in the previous phase we extract a pattern vector. Therefore, we compute the mean and the standard deviation of each cache set s, *i.e.*, the $mean_{i,s}$ and the $std_{i,s}$ of the columns within the matrix $\mathbf{t}_i[b][s]$. We apply the following empirically detected threshold: If the measurement score of a specific byte value b within a specific cache set $\mathbf{s}$ is greater than the $mean_{i,s}$ plus the standard deviation $std_{i,s}$, we assume that this index has been accessed during the encryption. Consequently, if the measurement score is below this threshold, we assume that this index has not been accessed during the encryption. Equation 2 outlines the extraction of a pattern vector for a specific cache set s and an attacked byte i.

$$\text{pattern\_vector}_i[b][s] = \begin{cases} 1, & \text{iif } \mathbf{t}_i[b][s] > mean_{i,s} + std_{i,s} \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

4.3 Compute Possible Access Patterns

As we have seen in Subsection 4.1, 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 compute the memory-access patterns within a specific cache set. Recall that the lower $\log_2 \delta$ bits are used as index bits, *i.e.*, these bits determine a specific element within a 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 3 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 $s \in \{0, \ldots, \gamma - 1\}$. 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.

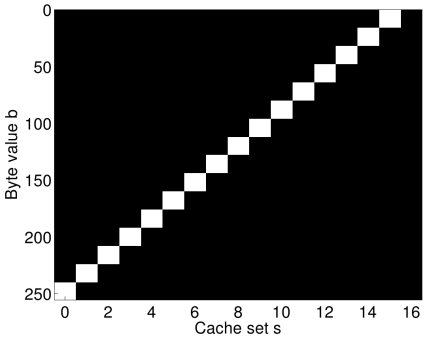

Fig. 3. Generated pattern for $\mathbf{h} = \texttt{0xF3}$, a disalignment $\mathbf{d} = 0$, and a cache-line size of 64 bytes

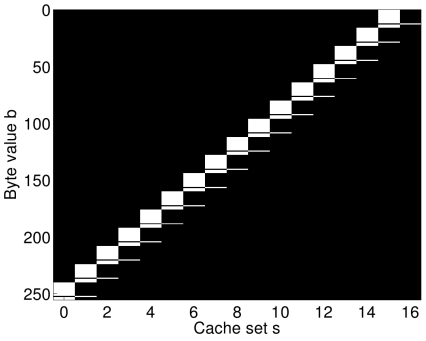

Fig. 4. Generated pattern for $\mathbf{h} = \texttt{0xF3}$, a disalignment $\mathbf{d} = 1$, and a cache-line size of 64 bytes

$$
\begin{aligned}
& b \in \{\texttt{0x00}, \ldots, \texttt{0xFF}\} \\
& s \in \{0, ..., \gamma - 1\} \\
& \text{pattern}_{\mathbf{h},\mathbf{d}}\,[b]\,[s] = \begin{cases} 1, & \text{iif shift\_right}(\,(b \oplus \mathbf{h}) + \mathbf{d},\ \log_2 \delta) == s \\ 0, & \text{otherwise} \end{cases}
\end{aligned} \tag{3}
$$

Considering all possible disalignments and all possible key hypotheses we deduce that there are $\delta \cdot 256$ patterns per cache set. For $\delta = 16$ this results in 4096 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. More specifically, an empirical analysis revealed that there are only 1376 unique patterns for one specific cache set, except for the first and the last cache set for which there are 2736 and 2720 unique patterns[3], respectively. Due to this observation we store the patterns in a way, such that a pattern maps to a list of pairs $(\mathbf{h}, \mathbf{d})$ that generate this specific pattern. As can be seen in Equation 3 the computed patterns depend on the key hypotheses and the possible disalignments. Thus, if the disalignment is known in advance, the number of unique patterns reduces even further. Figure 3 and Figure 4 illustrate the plots of the computed patterns, as outlined in Equation 3, corresponding to the hypothetical key byte $\mathbf{h} = \texttt{0xF3}$ and two different disalignments $\mathbf{d} \in \{0, 1\}$. For visualization purposes we computed the pattern for all cache sets a specific T-table $\mathbf{T}_j$ can take. For the actual attack one might even recover the secret key by computing the patterns for only one cache set. Figure 3 visualizes the generated pattern for an aligned T-table, *i.e.*, $\mathbf{d} = 0$. In this case the T-table consumes exactly 16 cache sets and each cache set holds 16 table elements. In contrast, Figure 4 visualizes a generated pattern for a disaligned T-table, *i.e.*, $\mathbf{d} = 1$. The first cache set, *i.e.*, cache set 0, holds only

[3] The discrepancy of $2736 - 2720 = 16$ patterns between the first and the last cache set results from the fact that in case of a disalignment $\mathbf{d} = 0$ the last cache set does not contain any T-table elements.

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 consumed by this T-table. Though, cache set 16 only holds the last table element with the index $\mathbf{s}_i = 0\text{xFF}$, indicated at byte 0x0C. An interesting property of the single element within cache set 17 is that $0\text{xFF} \oplus 0\text{x0C} = 0\text{xF3}$, since $\mathbf{s}_i \oplus \mathbf{p}_i = \mathbf{k}_i$, and thus yields the correct key byte immediately. We will exploit this specific property later. Of course, for other cache-line sizes this attack approach works analogously.

4.4 Extract Possible Key Candidates

In this step, we compare the pattern vector of a specific cache set s against all possible pattern vectors computed for the same cache set s. If they match, we retrieve a list of possible key candidates **h** that yield this specific pattern. If the access patterns are clearly visible among the first set, an attacker might consider exploiting the access pattern of the first cache set only. However, a possibly noisy pattern might hinder the extraction of the pattern within specific cache sets. Thus, we extract the pattern of multiple cache sets and match them against the precomputed patterns. The best results, *i.e.*, where the largest number of key bits is recovered, might be achieved by computing the intersection of the returned key hypotheses **h** of all investigated cache sets. As already outlined before, a noisy pattern might yield the wrong key candidate for a specific cache set and hence computing the intersection of the returned key candidates might prevent a successful key extraction. Thus, we count the number of cache sets that consider **h** as a possible key candidate. The more cache sets report a possible key candidate **h**, the more likely it might be the real key byte.

4.5 Brute-Force Key Search on the Remaining Key Space

In case the pattern-matching approach yields more than one key candidate per key byte a subsequent brute-force key search with a known plaintext-ciphertext pair might reveal the correct secret key.

5 Improvements of the Attack Concept

Further investigations of the extracted cache-access patterns revealed that the generated patterns leak even more information, at least in case of disaligned T-tables. Recall that the look-up indices into the T-tables within the first round are computed as $\mathbf{s}_i = \mathbf{p}_i \oplus \mathbf{k}_i$. 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 T-table entry corresponding to the correct key byte is always within the largest block of the first cache set as well as the largest block of the last cache 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 key candidate in order to invert all bits.

The observation that the larger block always contains the correct key byte might be clarified as follows. 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 at $\alpha = 1$ and increasing it continuously leads to a change within the lower $\log_2 \alpha$ bits of $\mathbf{s}_i$, with the remaining upper bits of $\mathbf{s}_i$ staying constant for a given α. Considering only the upper $8 - \lceil \log_2 \alpha \rceil$ bits of these α look-up indices $s_i \in \{0, \ldots, \alpha - 1\}$, these elements form a contiguous group. Now, consider the inverse operation of the key addition $\mathbf{p}_i = \mathbf{s}_i \oplus \mathbf{k}_i$, with the key value $\mathbf{k}_i$ being irrelevant for this explanation. Within such a group, which is composed of the look-up 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. In case the plaintext byte $\mathbf{p}_i$ equals the correct key byte $\mathbf{k}_i$ the resulting look-up index is $\mathbf{p}_i \oplus \mathbf{k}_i = 0$. Thus, the correct key byte is always within the largest block.

Another possible enhancement of this attack might be to consider only key candidates with the same disalignment within the *pattern-matching phase*. Since the T-tables are usually located contiguously within the memory the disalignment should be the same for all T-tables. Even in case the T-tables are not located contiguously within the memory this approach might be considered for key bytes $\mathbf{k}_i$ related to the same T-table $\mathbf{T}_j$, such that $i \equiv j \mod 4$. Thus, for some runs this might even further reduce the remaining key space.

6 Practical Application

Depending on the number of iterations R within the online phase, and the number of exploited cache sets N, $16 \cdot 256 \cdot 3 \cdot N \cdot R$ AES encryptions are required in order to gather the cache-access patterns for all possible key-byte indices $0 \leq i < 16$ and all possible plaintext byte values $0 \leq \mathbf{p}_i < 256$ of the corresponding index. The factor 3 represents the number of encryptions per plaintext, *i.e.*, *warm-up* phase and two encryptions in order to gather the measurement score. Considering the exploitation of all $N = 17$ cache sets on systems with a cache-line size of 64 bytes and $R = 10$ iterations this yields a total number of 2^{21} AES encryptions. In case of $R = 150$ iterations this yields 2^{25} AES encryptions. As already outlined above, in case the number of exploited cache sets is reduced the complexity of the attack in terms of AES encryptions decreases. Excluding the brute-force key search, which might be done on a separate machine, the whole attack can be performed within 40 to 80 seconds on a Cortex-A8 processor. If only the online phase, *i.e.*, gathering the cache-access patterns, is performed on the mobile device under attack and the remaining steps are performed on a more powerful machine, then this might be even reduced to a few seconds.

Gathering cache-access patterns has been observed to be successful already for $R = 10$ runs per attacked key byte $\mathbf{k}_i$. Though the pattern might not be visible by visual inspection immediately, the attack procedure outlined in this paper still might

Table 1. Results of several access-driven attacks on the ARM Cortex-A8 processor

		Pattern		Pattern + Largest Block	
R	α	$\mathbf{P}_{success}$	**Average Remaining Bits**	$\mathbf{P}_{success}$	**Average Remaining Bits**
20	10	90 %	32	90 %	18
20	14	80 %	32	50 %	25
150	1	100 %	16	80 %	9
150	2	100 %	32	100 %	16
150	3	100 %	16	100 %	8
150	4	100 %	40	100 %	40
150	5	100 %	16	100 %	0
150	6	100 %	32	90 %	21
150	7	100 %	16	80 %	6
150	8	80 %	64	80 %	48
150	9	100 %	16	100 %	0
150	10	100 %	32	100 %	21
150	11	100 %	16	70 %	5
150	12	100 %	48	90 %	37
150	13	100 %	16	80 %	4
150	14	100 %	32	50 %	20
150	15	100 %	16	70 %	7
150	16	100 %	64	100 %	64

recover the key successfully. In practice, one might consider exploiting the cache-access patterns of only one specific cache set per key byte $\mathbf{k}_i$. Furthermore, we observed that in case of disaligned T-tables the implemented enhancement, *i.e.*, attacking the largest block within the first or the last set, improves the result of this attack in terms of remaining key bits. Table 1 summarizes the main results of our attack on the Google Nexus S, which employs an ARM Cortex-A8 processor. Moreover, our test device operates a fully-functioning *Android 2.3.4* operating system. *Pattern* refers to the proposed attack employing the pattern-matching approach and *Pattern + Largest Block* refers to the proposed improvement of the attack. We observed that some disalignments, *e.g.*, where only 5 or 9 elements are located within the first set, are highly vulnerable to the presented attack. In this case our enhanced attack is able to recover the whole secret key without a single brute-force encryption. On average we observe a success probability of 87 % and less than 20 bits need to be searched exhaustively.

The presented results are based on the investigation of all odd cache sets, plus the first and the last one for the proposed improvement of the attack. We chose the odd cache sets in order to eliminate possible noise which might affect multiple contiguous cache sets. Nevertheless, it might be possible that the investigation of other cache sets, *e.g.*, $s \in \{0, 1, 2, 3, 4, 16\}$, yields better results. Such investigations might be subject to future work. Furthermore, Table 1 also shows that in case of a disalignment of 1 or $\delta - 1$ the key might not be revealed directly. Though, a visual inspection by a human being usually yields the correct key byte directly, a programmatic detection of the correct key byte within the first or the last set might be difficult due to reasons of noise.

7 Conclusion

In this paper we demonstrated an attack based on the analysis of memory-access patterns. According to our knowledge we are the first to perform an access-driven attack on ARM Cortex-A series processors, which employ a random-replacement policy. We have shown that given the memory-access patterns of disaligned T-tables it might be possible to reveal the secret key without a single brute-force computation. In addition, we emphasize that under the assumption that the cache-access pattern extraction of specific cache sets works reliably, then computing the intersection of the returned key candidates of the investigated cache sets might even leak more key bits. Concluding the investigation of the proposed access-driven attack we heavily stress the importance of aligned AES T-tables. Specifying the `aligned` attribute when declaring the T-table ensures such a proper alignment. Though this does not prevent timing information from being leaked it prevents an attacker from recovering the whole key immediately. In case of properly aligned T-tables only half of the key bits can be recovered on systems with a cache-line size of 64 bytes.

Future work related to the investigation of cache attacks on the ARM Cortex-A series might be to investigate the applicability of this attack without using the PMCCNTR register, and thus removing the requirement for a rooted mobile device. Furthermore, the utilization of other performance-monitor registers might be considered to launch even more sophisticated attacks. For instance, employing these performance-monitor registers might allow the implementation of trace-driven attacks purely in software.

Acknowledgements. This work has been supported by the Austrian Science Fund (FWF) under grant number TRP 251-N23 (Realizing a Secure Internet of Things - ReSIT).

References

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

[2] ARM Ltd. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R ed., ARM DDI 0406 A*, April 2007.

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

[4] ARM Ltd. *Cortex-A Series Programmer's Guide, Version: 2.0*, August 2011.

[5] ARM Ltd. Cortex-A Series. Available online at `http://www.arm.com/products/processors/cortex-a/index.php`, 2012.

[6] D. J. Bernstein. Cache-timing attacks on AES. Available online at `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, April 2005.

[7] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo. AES Power Attack Based on Induced Cache Miss and Countermeasure. In *Information Technology: Coding and Computing*, volume 1 of *ITCC 2005*, pages 586–591. IEEE Computer Society, 2005.

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

[9] J. Bonneau and I. Mironov. Cache-Collision Timing Attacks Against AES. In *Cryptographic Hardware and Embedded Systems - CHES 2006*, volume 4249 of *LNCS*, pages 201–215, 2006.

[10] J.-F. Gallais and I. Kizhvatov. Error-Tolerance in Trace-Driven Cache Collision Attacks. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, COSADE 2011, pages 222–232, 2011.

[11] D. Gullasch, E. Bangerter, and S. Krenn. Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy*, SP 2011, pages 490–505. IEEE Computer Society, 2011.

[12] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security*, 8(2–3):141–158, 2000.

[13] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO 1996*, volume 1109 of *LNCS*, pages 104–113. Springer Berlin / Heidelberg, 1996.

[14] C. Lauradoux. Collision attacks on processors with cache and countermeasures. In *Western European Workshop on Research in Cryptology*, WEWoRC 2005, pages 76–85, 2005.

[15] National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard, November 2001. Available online at `http://www.itl.nist.gov/fipspubs/`.

[16] M. Neve and J.-P. Seifert. Advances on Access-Driven Cache Attacks on AES. In *Selected Areas in Cryptography*, volume 4356 of *LNCS*, pages 147–162. Springer Berlin / Heidelberg, 2007.

[17] OpenSSL Software Foundation. OpenSSL Project. Available online at `http://www.openssl.org/`, 2012.

[18] D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. Technical Report CSTR-02-003, University of Bristol, Department of Computer Science, June 2002. Available online at `http://www.cs.bris.ac.uk/Publications/Papers/1000625.pdf`.

[19] C. Rebeiro, R. Poddar, A. Datta, and D. Mukhopadhyay. An Enhanced Differential Cache Attack on CLEFIA for Large Cache Lines. In *Progress in Cryptology - INDOCRYPT 2011*, volume 7107 of *LNCS*, pages 58–75. Springer Berlin / Heidelberg, 2011.

[20] E. Tromer, D. A. Osvik, and A. Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[21] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. Cryptanalysis of DES Implemented on Computers with Cache. In *Cryptographic Hardware and Embedded Systems, CHES*, volume 2779 of *LNCS*, pages 62–76. Springer, 2003.

[22] Y. Tsunoo, E. Tsujihara, K. Minematsu, and H. Miyauchi. Cryptanalysis of Block Ciphers Implemented on Computers with Cache. In *International Symposium on Information Theory and Its Applications, ISITA*, October 2002.

[23] M. Weiß, B. Heinz, and F. Stumpf. A Cache Timing Attack on AES in Virtualization Environments. In *Financial Cryptography and Data Security*, volume 7397 of *LNCS*, pages 314–328. Springer Berlin Heidelberg, 2012.

[24] X. Zhao and T. Wang. Improved Cache Trace Attack on AES and CLEFIA by Considering Cache Miss and S-box Misalignment. *IACR Cryptology ePrint Archive*, 2010:56, 2010.

[25] X. Zhao, T. Wang, D. Mi, Y. Zheng, and Z. Lun. Robust First Two Rounds Access Driven Cache Timing Attack on AES. In *International Conference on Computer Science and Software Engineering*, volume 3 of *CSSE 2008*, pages 785–788. IEEE Computer Society, 2008.

[26] X. Zhao, T. Wang, and Y. Zheng. Cache Timing Attacks on Camellia Block Cipher. *IACR Cryptology ePrint Archive*, 2009:354, 2009.