On the Applicability of Time-Driven Cache Attacks on Mobile Devices

(Extended Version*)

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 known to be sophisticated attacks against cryptographic implementations on desktop computers. Recently, also investigations of such attacks on testbeds with processors that are employed in mobile devices have been done. In this work we investigate the applicability of Bernstein's [4] timing attack and the cache-collision attack by Bogdanov *et al.* [6] in real environments on three state-of-the-art mobile devices. These devices are: an *Acer Iconia A510*, a *Google Nexus S*, and a *Samsung Galaxy SIII*. We show that T-table based implementations of the Advanced Encryption Standard (AES) leak enough timing information on these devices in order to recover parts of the used secret key using Bernstein's timing attack. We also show that systems with a cache-line size larger than 32 bytes exacerbate the cache-collision attack by Bogdanov *et al.* [6].

Keywords: AES, ARM Cortex-A series processors, time-driven cache attacks, cache-collision attacks.

1 Introduction

Cache attacks are a specific form of implementation attacks that focus on the exploitation of variations within the execution time of a cryptographic algorithm. Variations in the execution time occur due to different access times within the memory hierarchy. For instance, the central-processing unit (CPU) is able to access data within the CPU cache an order of magnitude faster than data within the main memory. Today's mobile devices also employ CPU caches and investigations of implementation attacks—and cache attacks in particular—are necessary in order to ensure the user's privacy and security on these devices. However, until recently these attacks mainly focused on desktop machines [4, 8, 10, 14]. Only minor efforts have been made towards the investigation of these attacks on mobile devices [12], where mainly testbeds simulating specific mobile-device configurations [6, 7, 15] have been used. Due to the wide-spread usage of mobile devices, *e.g.*, smartphones and tablet computers, and their manifold application scenarios, security and privacy issues on these devices are of utmost importance. Additional applications and widgets allow for further enhancements of capabilities on these devices and might contain security-relevant algorithms. Since these algorithms might be

* This paper is an extended version of a short paper accepted at NSS 2013 [13].

vulnerable to implementation attacks, the investigation of such attacks shall raise the awareness of implementation attacks among developers, leading to more secure systems in general. The aim of this work is to analyze whether T-table based implementations of the Advanced Encryption Standard (AES) on state-of-the-art Android-based mobile devices, *i.e.*, featuring a full-blown operating system, leak enough timing information to deduce the used secret keys. Therefore, we launch the attack of Bernstein [4] as well as the attack of Bogdanov *et al.* [6] on three mobile devices, namely an *Acer Iconia A510*, a *Google Nexus S*, and a *Samsung Galaxy SIII*.

The presented paper is organized as follows. In Section 2 we outline related work and give a short motivation for the analysis of cache attacks on mobile devices. Section 3 briefly introduces the required preliminaries for the investigation of cache attacks on mobile devices. We state the main concepts of the two investigated cache attacks in Section 4 and the main findings regarding the analysis of these two attacks on mobile devices in Section 5. Finally, we conclude this work in Section 6.

2 Related Work and Motivation

Cache attacks can be separated into three main categories: (1) *time-driven attacks*, (2) *access-driven attacks*, and (3) *trace-driven attacks*. Time-driven attacks [4] focus on the exploitation of the overall encryption time and, thus, require many measurement samples. In contrast, access-driven attacks [8, 14] and trace-driven attacks [5] focus on more fine-grained information leakage and require far less measurement samples than time-driven attacks. However, access-driven attacks and trace-driven attacks require sophisticated knowledge about the hardware as well as the software under attack.

Cache attacks have been analyzed extensively in desktop environments and the wide-spread usage of mobile devices stresses the importance of the investigation of cache attacks also on these devices. Though, only minor efforts have been made towards the analysis of cache attacks on mobile devices so far. In 2010, Bogdanov *et al.* [6] proposed a cache-collision attack by exploiting collisions between consecutive encryptions of pairs of chosen plaintexts. The attack environment was an ARM9 board configured as a server that runs an AES implementation of OpenSSL [11]. This implementation was queried by the authors via an Ethernet interface. Gallais and Kizhvatov [7] investigated trace-driven cache attacks on an ARM7 microcontroller.

In 2012, Weiß *et al.* [15] investigated the applicability of Bernstein's [4] time-driven cache attack on a Beagleboard employing an ARM Cortex-A8 processor, running the *Fiasco.OC microkernel* and the *L4Re runtime environment* on top. Given this setting they considered the attack of a specifically designed authentication protocol between an application running in a *Trusted Execution Environment* (TEE) and a remote back-end server. The untrusted world acts as a relay in order to forward messages from the TEE to the outside world and vice versa. They claim that by overtaking the untrusted world with specific malware, it is possible to break the authentication protocol using Bernstein's timing attack. Additionally, they use this scenario to compare the vulnerability of different AES implementations. Nevertheless, Weiß *et al.* [15] claim that noise makes the attack difficult and, hence, further research regarding the impact of real noise is necessary.

Recent investigations of cache attacks on mobile devices and embedded devices have been mainly done in laboratory constraints and environments. In this work, we focus on the investigation of such time-driven cache attacks in more realistic environments by analyzing the applicability of the attack by Bernstein [4] and the attack by Bogdanov *et al.* [6] on three Android-based mobile devices.

3 Requirements

In this section we briefly outline the main concept of the Advanced Encryption Standard and the ARM Cortex-A series architecture.

3.1 Advanced Encryption Standard

The Advanced Encryption Standard (AES) [9], designed by J. Daemen and V. Rijmen as Rijndael, is a block cipher operating on a 128-bit state. We denote the state as a series of bytes $\mathbf{S} = \{\mathbf{s}_0, \ldots, \mathbf{s}_{15}\}$, which are usually represented as a matrix of four rows and four columns, respectively. The AES consists of four round transformations: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. Due to performance reasons *SubBytes* usually uses precomputed S-Boxes in order to substitute each byte of the state with a precomputed byte value. Since *MixColumns* also performs complex mathematical operations, software implementations usually operate on look-up tables ($\mathbf{T}_0, \ldots, \mathbf{T}_4$) which hold precomputed values for these two round transformations. With these look-up tables the encryption can be performed by look-up operations in combination with XOR operations. In the following we denote the initial state as $\mathbf{s}_i = \mathbf{p}_i \oplus \mathbf{k}_i$, where $\mathbf{P} = \{\mathbf{p}_0, \ldots, \mathbf{p}_{15}\}$ represents the plaintext and $\mathbf{K} = \{\mathbf{k}_0, \ldots, \mathbf{k}_{15}\}$ the initial round key.

The fact that these look-up tables—each consisting of 256 4-byte values—are partially cached during the encryption and the fact that the look-up indices are key dependent leads to AES implementations which are highly susceptible to cache attacks.

3.2 ARM Architecture

The ARM Cortex-A series processors [3] are employed in many modern mobile devices, *e.g.*, smartphones and tablet computers. Like common desktop processors, these processors utilize CPU caches in order to hold close data used recently and data probably to be used in the near future. The Cortex-A8 and the Cortex-A9 processors employ a 4-way set-associative data cache with a cache-line size of either 32 or 64 bytes and a total size of 32 KB. Since the cache memory is smaller than the main memory, multiple locations from the main memory are mapped to the same location within the cache memory. In case of so-called set-associative caches a specific location within the memory is mapped to a specific cache set and can be placed within any cache line of this specific cache set. Thus, a mechanism is required in order to evict a cache line from such a specific cache set to free up space for new data to be cached. Due to reasons of simplicity ARM processors usually evict a cache line from a cache set randomly. In contrast, most desktop CPU caches employ a deterministic replacement policy. Since

time-driven cache attacks rely on statistical analysis of measurement samples, the random replacement policy might have a negative impact on the number of required measurement samples.

The ARM Cortex-A series also features performance-monitor registers [2], implemented within a special coprocessor. These registers include a so called *Cycle Count Register*, which is a 32-bit register capable of counting clock cycles. This register is used to measure execution times on a cycle-accurate basis, which allows distinguishing main-memory accesses from cache-memory accesses.

4 Time-Driven Cache Attacks

The basic idea of time-driven cache attacks is to exploit the overall encryption time of cryptographic algorithms employing precomputed look-up tables. The main problem of such implementations is that specific look-up operations leak different timing information based on where the corresponding data is located within the memory hierarchy. Since the cache is a shared resource and many different locations from the main memory are mapped to the same location within the cache memory, manipulations of the cache memory happen rather frequently.

In the following subsections we briefly outline the basic concept of Bernstein's [4] time-driven cache attack and the concept of the cache-collision attack suggested by Bogdanov *et al.* [6].

4.1 Timing Attack

In 2005, Bernstein [4] suggested a time-driven cache attack against the AES T-table implementation of OpenSSL [11]. Bernstein's idea is based on the assumption that the overall encryption time correlates with the timing leakage of specific T-table look-up operations, *e.g.*, $\mathbf{T}_0[\mathbf{p}_0 \oplus \mathbf{k}_0]$. He claims that by gathering the encryption times of many samples of plaintexts it is possible to recover the secret key. Therefore, he gathers many measurement samples of encryptions under a known key $\mathbf{K}$ and correlates this timing information with measurement samples under an unknown key $\tilde{\mathbf{K}}$. Initially, Bernstein aimed at recovering the used secret key of a remote server. However, Neve [10] performed further investigations of this time-driven attack and suggested to launch it against a local encryption function. Neve furthermore claims that cache evictions do not happen completely random. Instead, most of the cache evictions occur at specific locations within the cache. Thus, these cache evictions induce timing variations which allow attackers to deduce the secret key. In this work we stick to the approach suggested by Neve and perform the attack on a local machine, more precisely on Android-based mobile devices.

Attack Concept. The basic concept of Bernstein's time-driven cache attack consists of four different phases: (1) *study phase*, (2) *attack phase*, (3) *correlation phase*, and (4) *exhaustive key-search phase*. These four phases are briefly described within the following paragraphs. For further details about this attack we refer to [4, 10].

Study Phase. In this phase the attacker gathers the encryption times of multiple random plaintexts $\mathbf{P}$ under a known secret key $\mathbf{K}$. The resulting timing information, *i.e.*, the encryption time of a specific plaintext $\mathbf{P}$, is stored in a data structure $\mathbf{t}[i][b]$. More formally, $\mathbf{t}[i][b]$ holds the total of all encryption times where the plaintext byte $\mathbf{p}_i = b$. In addition, $\mathbf{n}[i][b]$ counts the number of encrypted plaintexts where $\mathbf{p}_i = b$. For reasons of simplicity, we assume the known key $\mathbf{k}_i$ to be $\texttt{0x00}$ for $0 \leq i < 16$. According to Neve [10], given this information the attacker computes the so called *plaintext-byte signature* as outlined in Equation 1.

$$\mathbf{v}[i][b] = \frac{\mathbf{t}[i][b]}{\mathbf{n}[i][b]} - \frac{\sum_i \sum_b \mathbf{t}[i][b]}{\sum_i \sum_b \mathbf{n}[i][b]} \tag{1}$$

Attack Phase. In this phase the attacker sends multiple random plaintexts $\tilde{\mathbf{P}}$ to the encryption function, which encrypts these plaintexts under an unknown key $\tilde{\mathbf{K}}$. Again, the corresponding information is stored in $\tilde{\mathbf{t}}[i][b]$ as well as $\tilde{\mathbf{n}}[i][b]$, and the attacker computes the *plaintext-byte signature* $\tilde{\mathbf{v}}$ as outlined above, except that $\mathbf{t}$ and $\mathbf{n}$ are replaced by $\tilde{\mathbf{t}}$ and $\tilde{\mathbf{n}}$, respectively.

Correlation Phase. Within this phase the attacker correlates the gathered timing profiles from the study phase $\mathbf{v}$ with the timing profiles from the attack phase $\tilde{\mathbf{v}}$. Neve [10] introduced the following heuristic: *If pairs of plaintext bytes and key bytes in the study phase $(\boldsymbol{p}_i, \boldsymbol{k}_i)$ and the attack phase $(\tilde{\boldsymbol{p}}_i, \tilde{\boldsymbol{k}}_i)$ have the same difference, i.e., $\boldsymbol{p}_i \oplus \boldsymbol{k}_i = \tilde{\boldsymbol{p}}_i \oplus \tilde{\boldsymbol{k}}_i$, then the timing profiles are similar*. Thus, computing the correlations for all indices i and all possible byte values b as outlined in Equation 2 reveals potential key candidates. The resulting correlations are sorted in a decreasing order and a threshold based on the computed variance is used to determine the most likely key candidates. The higher the correlation $c[i][b]$ the more likely the corresponding byte value b represents the real key byte. Note that we assume the known key $\mathbf{K}$ to be zero and hence we do not have to consider $\mathbf{K}$ here.

$$c[i][b] = \sum_{j=0}^{255} \mathbf{v}[i][j] \cdot \tilde{\mathbf{v}}[i][j \oplus b] \tag{2}$$

Exhaustive Key-Search Phase. Since the correlation phase usually yields multiple key candidates per key byte $\tilde{\mathbf{k}}_i$ an exhaustive key search on the remaining key space has to be performed.

4.2 Collision Attack

As the name already suggests, cache-collision attacks focus on the exploitation of collisions between look-up indices of intermediate state bytes. Given information about such collisions one tries to infer relations between key bytes. Bogdanov *et al.* [6] suggested to focus on cache collisions between the encryption of two plaintexts which are chosen in a specific manner. Their intention is to choose pairs of plaintexts $(\mathbf{P}_1, \mathbf{P}_2)$ such that five

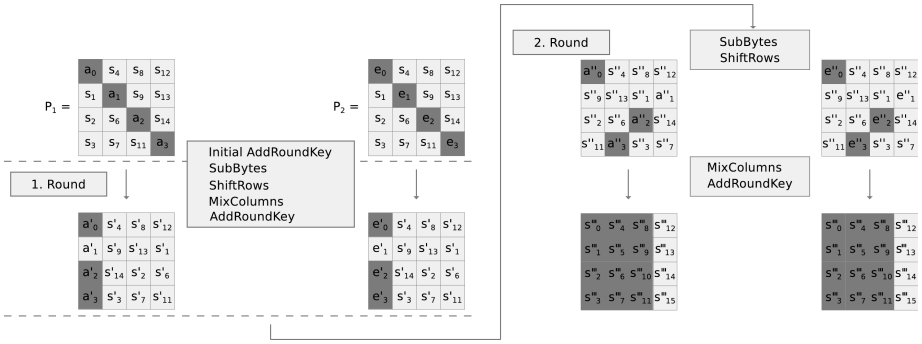

Fig. 1. Intermediate states of a plaintext pair ($\mathbf{P}_1$, $\mathbf{P}_2$) leading to a wide collision, assuming $a'_1 = e'_1$ and hence $a''_1 = e''_1$ also holds true.

S-Box look-ups[1] (*SubBytes* transformations) within the encryption of $\mathbf{P}_2$ collide with S-Box look-ups of $\mathbf{P}_1$. This is what they call a *wide collision*. Supposing an initially empty cache the encryption of plaintext $\mathbf{P}_1$ loads the necessary S-Box elements into the cache. The encryption of the second plaintext $\mathbf{P}_2$ will be performed faster if look-up indices collide, *i.e.*, are equal between the first and the second encryption and thus are already located within the CPU cache. Hence, the encryption time of the plaintext $\mathbf{P}_2$ is used as an indicator to determine whether such a *wide collision* occurred.

Considering the plaintext as a matrix of 4×4 bytes, we investigate the main diagonal from the top left to the bottom right for the explanation. For the actual attack all four diagonals, *i.e.*, state bytes which are shifted to the same column after the first *ShiftRows* transformation, must be considered because one diagonal can only reveal four key bytes. Figure 1 outlines the two plaintexts $\mathbf{P}_1$ and $\mathbf{P}_2$, with the main diagonals $A = \{a_0, a_1, a_2, a_3\}$ and $E = \{e_0, e_1, e_2, e_3\}$ being pairwisely different. According to the notation of Bogdanov *et al.* [6] we illustrate bytes which are pairwisely equal between $\mathbf{P}_1$ and $\mathbf{P}_2$ brighter, whereas bytes which are pairwisely different are illustrated darker. In order to provoke *wide collisions* the attacker chooses two diagonals (A, E) randomly and pairwisely different, such that $a_i \neq e_i$ for $0 \leq i < 4$. The rest of both plaintexts is also chosen randomly but pairwisely equal. Now, we simply follow the round transformations labeled within Figure 1. At the end of round 1, after performing the *MixColumns* and the *AddRoundKey* transformation, one might observe collisions of the state bytes within the first column, *i.e.*, $a'_i = e'_i$ for some i. If such a collision occurs, then these bytes are shifted to the same column within the second round and hence after the round transformations of the second round pairwisely equal columns occur. The example in Figure 1 assumes that the state bytes $a'_1 = e'_1$, and thus the last column contains pairwisely equal state bytes after the second round. In this case 5 additional S-Box collisions occur during the encryption of the second plaintext $\mathbf{P}_2$, *i.e.*, one for the look-up of e'_1 within the first round and four collisions for the look-up of the values within the last column in the third round. If such a *wide collision* occurs, the average

[1] Though S-Box collisions are considered for the explanation of wide collisions, this attack also works for the common T-table implementation.

encryption time of the second plaintext is assumed to be faster than in case such a *wide collision* does not occur.

Attack Concept. The attack consists of three different phases: (1) *online phase*, (2) *collision-detection phase*, and (3) *key-search phase*, which are outlined within the following paragraphs. For further details regarding this attack we refer to [6].

Online Phase. Basically, the attacker performs three loops. Within the outermost loop one chooses the diagonal pairs (A, E) as outlined above N times. Within the second loop the attacker chooses the rest of the two plaintexts pairwisely equal I times, and the innermost loop encrypts the same two plaintexts $(\mathbf{P}_1, \mathbf{P}_2)$ R times in order to lower the impact of noise. Additionally, one might also consider to evict all data from the cache before encrypting $\mathbf{P}_1$ to initialize the cache, *i.e.*, to evict all T-table elements from previous runs. The output of this phase consists of the diagonal pairs (A, E) along with a statistical value related to the encryption time of the second plaintext $\mathbf{P}_2$.

Collision-Detection Phase. Given the output of the previous phase the intention of this phase is to determine which pairs of diagonals lead to *wide collisions*. Therefore, we take n diagonal pairs (A, E) with the lowest encryption times—according to the selected statistical metric—of plaintext $\mathbf{P}_2$, computed in the previous phase, and consider these diagonal pairs to lead to *wide collisions*.

Key-Search Phase. For each possible collision of state bytes between the encryption of $\mathbf{P}_1$ and $\mathbf{P}_2$ after the first round one can form equations which hold for specific subkeys. For instance, consider a collision between $a'_1 = e'_1$, as illustrated in Figure 1, then Equation 3 must hold for specific subkeys $(k_0, k_5, k_{10}, k_{15})$. Thus, the attacker establishes a system of four equations ($a_i = e_i$ for $0 \leq i < 4$) in order to evaluate for which 4-byte subkeys $a'_i = e'_i$ holds. Since this leads to a system of equations with four unknowns, at least four diagonal pairs (A, E) that lead to *wide collisions* are necessary. The evaluation of this system of equations is done by iterating over all possible 4-byte subkeys. Subkeys that lead to a collision are stored for the following exhaustive key search.

$$\begin{aligned} &01 \cdot Sbox(a_0 \oplus k_0) \oplus 02 \cdot Sbox(a_1 \oplus k_5) \oplus 03 \cdot Sbox(a_2 \oplus k_{10}) \oplus 01 \cdot Sbox(a_3 \oplus k_{15}) \\ &\qquad = \\ &01 \cdot Sbox(e_0 \oplus k_0) \oplus 02 \cdot Sbox(e_1 \oplus k_5) \oplus 03 \cdot Sbox(e_2 \oplus k_{10}) \oplus 01 \cdot Sbox(e_3 \oplus k_{15}) \end{aligned} \tag{3}$$

After performing these four steps for all four diagonals, the subsequent exhaustive key search on the 4-byte subkeys of each diagonal should reveal the whole secret key.

5 Analysis and Practical Results

Instead of using specific testbeds, as done in [6, 15], we launch these attacks on state-of-the-art Android-based mobile devices[2]: (1) an *Acer Iconia A510 tablet computer*, (2) a *Google Nexus S*, and (3) a *Samsung Galaxy SIII*. These devices feature fully-functioning

[2] For a detailed specification of these devices see Table 4 in Appendix A.

Table 1. Sample output of Bernstein's time-driven cache attack on a *Samsung Galaxy SIII*.

# of key candidates	Key byte	Possible values									
3	0	**b5**	b4	b8							
125	1	00	a2	be	c2	b8	1d	f6	...	**93**	...
165	2	87	03	51	17	1b	1f	c7	...	**11**	...
2	3	**66**	67								
104	4	59	1d	10	a5	34	06	50	...	**af**	...
6	5	**bc**	bd	b9	b8	ba	bb				
8	6	cc	**cd**	ca	cf	cb	c8	ce	c9		
2	7	**8d**	8c								
115	8	1e	ea	c9	ee	e6	11	**12**	cc	02	...
2	9	b8	**b9**								
153	10	76	7d	56	b3	5b	4b	3c	...	**55**	...
2	11	**12**	13								
23	12	83	9f	82	96	94	97	**92**	9d	98	...
2	13	**4a**	4b								
40	14	6a	7a	7b	74	61	7c	64	**6b**	78	...
2	15	9c	**9d**								

operating systems and the only restriction is that these devices must be rooted in order to load a specific kernel module. This is necessary since we use a kernel module to set the appropriate bit within a control register to grant unprivileged applications access to the cycle-count register. However, for the sake of clarity we have to mention that we only need to load this kernel module once after powering up the device. The attack itself runs in unprivileged mode and the attacked AES implementation is the standard C implementation of OpenSSL [11] employing T-tables.

The purpose of this work is to analyze whether realistic environments on mobile devices leak enough timing information such that the used secret AES key can be recovered, at least partially. We performed both attacks within a single application, as already suggested by Neve [10] for the analysis of Bernstein's [4] attack. This means that we simply call an encryption function that was implemented within the attack application itself. More formally, the attack application performs the encryptions, gathers the required information, and computes the relevant information as outlined above. In this section we briefly state the main findings of the conducted attacks on the three mobile devices.

5.1 Timing Attack

Since time-driven cache attacks perform statistical analysis on gathered measurement samples, a large number of measurement samples is required. Performing the timing attack with 2^{30} measurement samples in both phases—study and attack phase—takes about 6 hours on the *Google Nexus S* smartphone and about 4 hours on the *Acer Iconia A510* and the *Samsung Galaxy SIII*. In Table 1 we illustrate a sample output of the correlation phase on the *Samsung Galaxy SIII*. The first column outlines the number of remaining key candidates for each possible key byte. The second column denotes the index of the key byte and the third column states all possible key bytes, with the correct key marked in bold. The possible key candidates are sorted according to the computed

correlation and, thus, the position also indicates the probability of the corresponding key candidate being the correct key byte. A series of dots illustrates omitted key bytes. One clearly observes that timing information is leaking. However, the number of remaining key bits is still too large for an exhaustive key search. In this case the complexity of the remaining key-search phase corresponds to 2^{58} AES encryptions, which is still impractical. Considering only the first round of the AES the key space cannot be reduced enough in order to perform an exhaustive key search. Note that only the study phase as well as the attack phase must be executed on the mobile device itself, the following exhaustive key-search phase might be executed on a more powerful machine, perhaps supporting *AES New Instructions* [1] (AES-NI). However, even with the introduction of *AES New Instructions* in Intel's most recent Westmere platform, 2^{58} AES encryptions are still impractical. In order to retrieve more key bits one might apply the second round attack as suggested by Neve [10]. Nevertheless, we did not investigate the second round since the only purpose of this analysis was to determine whether timing information also leaks in more realistic scenarios on mobile devices and whether we can exploit this timing leakage.

From Table 1 we also observe that for some key bytes the number of possible key candidates has been reduced significantly, *e.g.*, to only 2 key candidates. However, some key bytes have not been reduced significantly. In contrast, we also observed runs where the key space was reduced but the correct key byte was not present among the possible key candidates anymore. Weiß *et al.* [15] also presented such an output where the correct key byte is shown within the first position[3] of all key bytes. Though we also observed runs where the correct key byte is at the first position for most of the 16 key bytes, we do not consider this the usual case. For the sake of completeness we have to mention that Weiß *et al.* [15] presented an output of a different AES implementation, which is based on Bernstein's Poly1305-AES message-authentication code. This implementation uses one large look-up table. Weiß *et al.* consider this implementation to be the most vulnerable AES implementation regarding time-driven cache attacks.

In order to generate the required cache evictions memory accesses at constant cache locations must be performed. These cache evictions induce timing variations within subsequent encryptions and thus lead to exploitable information leakage. Bernstein [4] simply generated the required cache evictions by sending data of different length to the server and the server in turn performed memory accesses on the transmitted data. Obviously, these memory accesses result in cache evictions at constant cache sets. We also launched this attack in a more realistic scenario where we mounted the attack while watching videos or while watching an image slideshow on the mobile devices. Nevertheless, running external applications on purpose did not leak more information and, hence, did not further reduce the key space. We conclude that these external applications either affected the wrong cache sets or lead to uncontrollable noise and hence corrupted the timing measurements. Furthermore, on multi-core devices, *e.g.*, the *Acer Iconia A510* and the *Samsung Galaxy SIII*, the two applications might be executed on different cores. Thus, a fairly realistic approach would be to wrap the attack in a fine-

[3] The possible key candidates are sorted according to the computed correlation values. Thus, the first position represents the most likely key candidate.

Table 2. Results of Bernstein's time-driven attack on the three mobile devices.

Device	Samples in		Remaining key space
	study phase	attack phase	
Acer Iconia A510	2^{30}	2^{27}	73 bits
	2^{30}	2^{29}	78 bits
Google Nexus S	2^{30}	2^{29}	65 bits
	2^{29}	2^{28}	69 bits
Samsung Galaxy SIII	2^{30}	2^{29}	58 bits
	2^{30}	2^{30}	61 bits

grained application and to control the memory accesses and potentially also the number of active cores within this application.

In Table 2 we state two of the best runs, *i.e.*, runs with the lowest number of remaining key bits, of Bernstein's attack for each of the three mobile devices. The number of measurement samples within the study phase as well as the attack phase are also listed in this table. We consider the number of generated measurement samples to be a crucial part. For the same number of measurement samples we observed runs where the key space was not reduced significantly and runs where the key space was reduced too much, *i.e.*, the correct key byte was not present anymore. Thus, more measurement samples do not necessarily lead to better results in terms of remaining key bits.

According to our observations noise must be generated carefully and might even disturb the measurement results and thus lead to non-recoverable key bytes. Generating noise within external applications, *e.g.*, launching a web browser or an image slideshow, did not improve our attacks. We conclude that either the wrong cache sets are affected or that noise disturbs our measurement results completely. Thus, this attack is heavily susceptible to noise generated by other applications running in parallel on the same CPU. Another problem we observed is that sometimes the key space is reduced too much and sometimes the key space is not reduced at all. While the former means that the correct key byte is not present within the list of key candidates anymore, the latter means that the key space still contains nearly all possible key bytes.

We also have to mention that time-driven attacks in general require a huge number of measurement samples and this in turn might drain the battery drastically. As already mentioned above, generating $2 \cdot 2^{30}$ measurement samples on the *Google Nexus S* takes about 6 hours. However, for scenarios where the attacker is in possession of the mobile device[4] time might not be a critical factor. Another reasonable assumption might be to launch the attack while the user charges the mobile device.

5.2 Collision Attack

The critical part of this attack is the detection of diagonal pairs (A, E) that lead to *wide collisions*. Since there must be at least 4 real *wide collisions* among the n chosen

[4] In some scenarios the attacker might even be the owner of the mobile device.

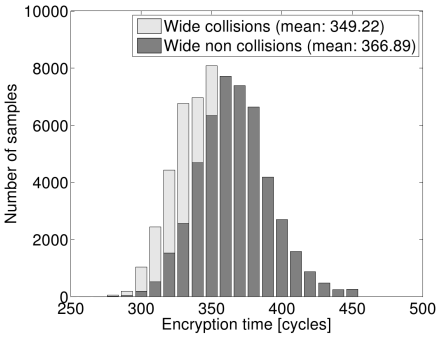

Fig. 2. Histogram of encryption times for a 3-round AES implementation. Encryption times for wide collisions and wide non collisions are visualized in light gray and dark gray, respectively.

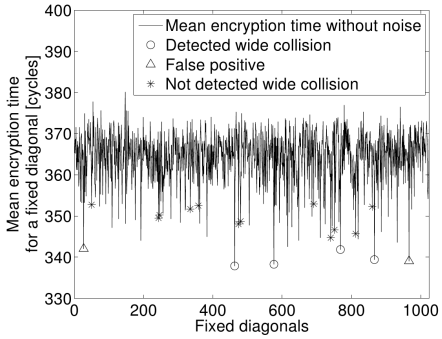

Fig. 3. Detection of wide collisions for a 3-round AES implementation.

diagonal pairs (A, E), a high expectation rate of *false positives*[5] must be overcome by taking more of these diagonal pairs into consideration.

On the ARM Cortex-A8 processor we observed that the detection of diagonals (A, E) which lead to *wide collisions* is a challenging task, at least for the full 10 rounds of the AES. Thus, we started with a reduced 3-round AES implementation. Figure 2 illustrates the histogram of encryption times of five diagonal pairs which lead to *wide collisions* in light gray and five diagonal pairs which do not lead to wide collisions in dark gray. Due to reasons of noise each chosen diagonal pair (A, E) is encrypted $I \cdot R$ times ($I = 400, R = 20$). We clearly observe that the encryption times of plaintexts which lead to *wide collisions* and encryption times of plaintexts which do not lead to *wide collisions* can be separated easily and by considering, *e.g.*, $n = 6$ diagonal pairs (A, E) with the lowest average encryption time we detected 4 diagonal pairs which indeed lead to *wide collisions*. Figure 3 visualizes this approach. For each of the N randomly chosen diagonal pairs (A, E) this plot shows the mean encryption time without noise, *i.e.*, averaged over $I \cdot R$ encryption times below a predefined threshold[6]. These figures look similar for the ARM Cortex-A9 processors, at least for the 3-round AES implementation.

In contrast, if we investigate the encryption times of a 7-round AES implementation as outlined in Figure 4, we observe that the encryption times of plaintexts that lead to *wide collisions* and plaintexts that do not lead to wide collisions cannot be distinguished anymore. Obviously, as outlined in Figure 5, the detection of *wide collisions* fails. Among the $n = 10$ chosen diagonal pairs, which are supposed to lead to *wide collisions* according to their encryption time, there is not even a single diagonal pair which

[5] False positives are diagonal pairs which are supposed to lead to *wide collisions* due to their encryption time, but in fact do not lead to wide collisions.

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

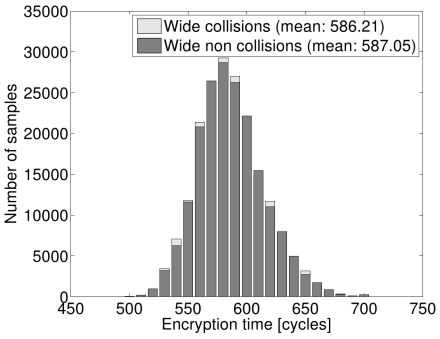

Fig. 4. Histogram of encryption times for a 7-round AES implementation. Encryption times for wide collisions and wide non collisions are visualized in light gray and dark gray, respectively.

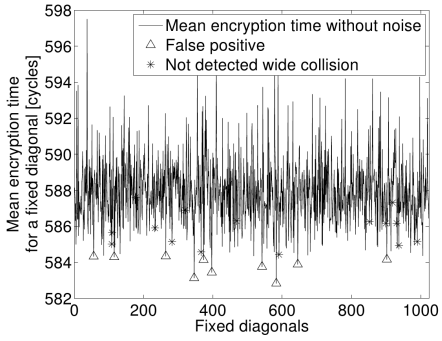

Fig. 5. Detection of wide collisions for a 7-round AES implementation.

indeed resulted in a wide collision. As already outlined above, a system of four equations with four unknowns must be solved. Hence, among the n chosen diagonal pairs there must be at least four real *wide collisions* in order for the attack to be successful.

The reason for this attack to fail on the ARM Cortex-A8 processor seems to be the larger cache-line size of 64 bytes. In contrast, Bogdanov *et al.* [6] launched the attack on an ARM9 board with a cache-line size of 32 bytes. Given the fact that each T-table is composed of 256 4-byte elements, a 32-byte cache line holds 8 T-table elements, whereas a 64-byte cache line holds 16 T-table elements. This means that in case of a cache miss the Cortex-A8 loads 16 consecutive T-table elements into the cache, whereas the ARM9 board loads only 8 elements into the cache at once. If we take probability theory into consideration the problem becomes clear. Since the last round of the AES T-table implementation usually employs a different T-table, there are $4 \cdot 9$ look-up operations into the same T-table within the rounds 1–9. Equation 4 outlines the probability for a specific block of T-table elements, *i.e.*, δ consecutive elements, still not being cached after one encryption.

$$\begin{aligned} A &:= \textit{Block of table elements still not in cache after one encryption.} \\ \delta &:= \textit{Number of table elements per cache line.} \\ P(A) &= \left(1 - \frac{\delta}{256}\right)^{4 \cdot 9} \end{aligned} \tag{4}$$

In case of $\delta = 16$ this yields a probability of 0.098 that a specific block of T-table elements is still not being cached after one encryption. In case of $\delta = 8$ this yields a probability of 0.319. Hence, the probability for a specific T-table element already being cached after the first encryption is $1 - 0.098 = 0.902$ and $1 - 0.319 = 0.681$. This in turn means that the probability for additional cache collisions, besides the required *wide collisions*, is far greater on systems with a cache-line size of 64 bytes. Thus, the overall encryption time of $\mathbf{P}_2$ decreases and this makes *wide collisions* nearly indistinguishable from non wide collisions. We conclude that the larger cache-line size of 64 bytes on the

Table 3. Sample result of Bogdanov *et al.*'s cache-collision attack on two mobile devices.

Device	Diagonal pairs	Parameters
Acer Iconia A510	$n = 7$	$C = 2000$, $N = 1024$, $I = 800$, $R = 20$
	$n = 8$	$C = 2000$, $N = 1024$, $I = 800$, $R = 40$
Samsung Galaxy SIII	$n = 7$	$C = 1200$, $N = 1024$, $I = 1000$, $R = 20$
	$n = 9$	$C = 1800$, $N = 1024$, $I = 1000$, $R = 20$

ARM Cortex-A8 exacerbates the detection of *wide collisions* and thus the applicability of this attack in general. Due to this observation we do not consider the ARM Cortex-A8 processor within the following analysis.

There are many different dimensions along which we can consider the detection of *wide collisions*. Firstly, we need to find the appropriate parameters, *i.e.*, C, I, and R, where C denotes the maximum number of cycles one encryption is supposed to take. Encryptions consuming more than C cycles are ignored. I denotes how often the remaining bytes, which do not belong to the chosen diagonal, of the plaintext are chosen randomly. Finally, R denotes how often a specific pair of plaintexts $(\mathbf{P}_1, \mathbf{P}_2)$ is encrypted in order to achieve stable measurement results. Secondly, there are different statistical values (metrics) which might be used to distinguish *wide collisions* from non wide collisions.

Table 3 shows two runs of Bogdanov *et al.*'s cache-collision attack with the corresponding parameters on the two Cortex-A9 devices. According to our observations we did not find any parameter configuration to detect enough *wide collisions* among $n \leq 6$ chosen diagonal pairs on both Cortex-A9 devices. One might overcome a high expectation rate of false positives with a larger number n of chosen diagonal pairs (A, E) within the *collision-detection phase*, but this drastically increases the complexity of the following 4-byte subkey search and the final exhaustive key search. This results from the fact that from all n chosen diagonal pairs the key-search phase considers all possible combinations of 4 diagonal pairs out of the n chosen diagonal pairs. According to Bogdanov *et al.* [6] the number of expected 4-byte subkeys per diagonal is $\binom{n}{4} \cdot 256$. Since these subkeys must be enumerated exhaustively for all four diagonals this yields an overall complexity of $\left(\binom{n}{4} \cdot 256\right)^4$ AES encryptions. For $n \leq 6$, *i.e.*, $\left(\binom{6}{4} \cdot 256\right)^4 \approx 2^{47.6}$ AES encryptions, this is feasible in a few days by employing *AES New Instructions* [1]. However, for $n = 7$ this results in about 2^{52} AES encryptions which is impractical.

Our main observation regarding the applicability of Bogdanov *et al.*'s cache-collision attack is that a cache-line size of 64 bytes exacerbates this attack. Furthermore, the detection of at least four *wide collisions* among a small number of chosen diagonal pairs, *e.g.*, $n \leq 6$, is a challenging task and a larger number of n drastically increases the remaining brute-force complexity.

6 Conclusion

Recent investigations of cache attacks on mobile devices mainly focused on specific testbeds and stressed the importance of analyzing these attacks in more realistic en-

vironments. Thus, we investigated the applicability of two time-driven cache attacks on state-of-the-art Android-based mobile devices. We observed that timing information also leaks on these devices and can be used to reduce the key space of cryptographic algorithms significantly. Though, time-driven cache attacks usually require an enormous number of measurement samples which might drain the battery drastically, we consider the attack of Bernstein [4] a threat for cryptographic implementations on mobile devices. In addition, we analyzed the attack of Bogdanov *et al.* [6] according to its applicability on mobile devices in more realistic environments. We showed that a cache-line size of 64 bytes exacerbates this attack and even on systems with a cache-line size of 32 bytes the detection of *wide collisions* seems to be a challenging task. Our observations revealed that, in practice, encryptions where *wide collisions* occur and encryptions where no wide collisions occur are hardly distinguishable. Even though a high number of false positives might be overcome by taking more diagonal pairs into consideration, this drastically increases the complexity of the following key-search phase.

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), and the Austrian Research Promotion Agency (FFG) under grant number 836628 (SeCoS).

References

[1] 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, Whitepaper, 2010.

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

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

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

[5] 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.

[6] A. Bogdanov, T. Eisenbarth, C. Paar, and M. Wienecke. Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs. In *CT-RSA*, pages 235–251, 2010.

[7] J.-F. Gallais and I. Kizhvatov. Error-Tolerance in Trace-Driven Cache Collision Attacks. In *COSADE*, pages 222–232, Darmstadt, 2011.

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

[9] National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard, November 2001.

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

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

[12] R. Spreitzer and T. Plos. Cache-Access Pattern Attack on Disaligned AES T-Tables. In *COSADE 2013*, LNCS. Springer, 2013. In press.
[13] R. Spreitzer and T. Plos. On the Applicability of Time-Driven Cache Attacks on Mobile Devices. In *NSS 2013*, LNCS. Springer, 2013. In press.
[14] E. Tromer, D. A. Osvik, and A. Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.
[15] M. Weiß, B. Heinz, and F. Stumpf. A Cache Timing Attack on AES in Virtualization Environments. In *FC*, pages 314–328. Springer Berlin Heidelberg, 2012.

A Device Specifications

Table 4. Detailed device specifications for the three mobile devices under attack.

	Acer Iconia A510	**Google Nexus S**	**Samsung Galaxy SIII**
Processor	Cortex-A9	Cortex-A8	Cortex-A9
Processor implementation	Nvidia Tegra 3 Quad 1.4 GHz	Exynos 3 Single 1 GHz	Exynos 4 Quad 1.4 GHz
Out-of-order execution	yes	no	yes
L1 cache size	32 KB	32 KB	32 KB
L1 cache associativity	4 way	4 way	4 way
L1 cache-line size	32 byte	64 byte	32 byte
L1 cache sets	256	128	256
Operating system	Android 4.0.4	Android 2.3.4	Android 4.0.4