Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs

Andrey Bogdanov[1], Thomas Eisenbarth[2], Christof Paar[2], Malte Wienecke[2]

[1] Dept. ESAT/SCD-COSIC, Katholieke Universiteit Leuven, Belgium
`andrey.bogdanov@esat.kuleuven.be`
[2] Horst Görtz Institute for IT Security
Ruhr University Bochum, Germany
{`thomas.eisenbarth, christof.paar, malte.wienecke`}`@rub.de`

Abstract. This paper proposes a new type of cache-collision timing attacks on software implementations of AES. Our major technique is of differential nature and is based on the internal cryptographic properties of AES, namely, on the MDS property of the linear code providing the diffusion matrix used in the MixColumns transform. It is a chosen-plaintext attack where pairs of AES executions are treated differentially. The method can be easily converted into a chosen-ciphertext attack. We also thoroughly study the physical behavior of cache memory enabling this attack.
On the practical side, we demonstrate that our theoretical findings lead to efficient real-world attacks on embedded systems implementing AES at the example of ARM9. As this is one of the most wide-spread embedded platforms today [7], our experimental results might make a revision of the practical security of many embedded applications with security functionality necessary. To our best knowledge, this is the first paper to study cache timing attacks on embedded systems.

1 Introduction

Side-channel attacks and cache timing leakage. Though side-channel leakage seems to have been extensively used by state security agencies for decades to obtain secret information [9], the idea of applying side-channel attacks to implementations of cryptographic algorithms appeared in the scientific literature rather recently. The first side-channel attack published was timing analysis proposed by Kocher [9] in 1996 where he observes the execution time of keyed cryptographic algorithms to recover the key and points out the usefulness of timing analysis applied to software implementations of block ciphers.

Probably the most widely known timing attacks on symmetric algorithms belong to the class of cache timing attacks on block ciphers with S-boxes. This is not least due to the literally ubiquitous usage of block ciphers, and first of all, of the U.S. encryption standard AES [2] in the overwhelming majority of security applications, both in PCs and embedded systems.

As the name of cache timing attacks suggests, they utilize the particularities of microcontrollers and microprocessors with cache memory which frequently exhibit key-dependent timing. Cache timing attacks on many block ciphers with S-boxes become possible since S-box invocations in software are often implemented as indexed table look-up operations that can require different execution times for different inputs due to RAM cache hits and misses. When the inputs to S-boxes are key-dependent, this timing information frequently turns out sufficient to recover the entire key. Today, several variants of cache timing attacks on AES are known [4], [12], [5], [1].

Generally speaking, side-channel analysis methods strongly depend on a concrete implementation of the attacked cryptographic algorithm. There is also no exception for cache timing attacks on AES where the choice of the optimal attack method can greatly vary depending on the implementation at hand. For AES, one can basically distinguish between first round and final round approaches: While first round attacks [1], [5] tend to be applicable to large classes of implementations at the cost of more encryption samples required, final round attacks [5] target only the T-box based 32-bit implementation [6] of AES having the advantage of being more efficient in this particular case. In this paper, we pursue and expand the more generic approach of first round attacks.

Main idea of our attack. The major idea behind our new cache timing attack is to choose pairs of plaintexts in a specific way, so that five AES S-boxes (one in round 2 and four in round 3) process either pairwisely equal or pairwisely distinct values in two adjacent AES executions. If for a plaintext pair the five S-boxes process pairwisely equal values, it is called a *wide collision*.

In our attack, we measure the average time of every second AES execution from each pair. We are interested in the average number c of S-box collisions (S-box pairs processing equal values) between the two AES runs in a pair. If a wide collision has occurred for a pair of plaintexts, c will be by 5 higher compared to c when there is no wide collision. We hope to detect enough wide collisions against the background if there are enough samples available. After this, we construct four systems of nonlinear equations with respect to parts of the key which are then resolved by brute force for the key recovery.

This technique becomes possible due to the fact that AES uses a maximum distance separable (MDS) code to construct its diffusion matrix for MixColumns operation. MDS codes are known to provide linear transforms with the maximum possible branch number [6], which is 5 for the parameter choice of AES. Interestingly enough, it is precisely the excellent cryptographic properties of AES, due to the optimal selection of the diffusion matrix making it resistant to differential and linear cryptanalysis, that enable our cache timing attack techniques.

Cache timing attacks and embedded security. Security in embedded systems is constantly and quickly becoming more crucial with the spread of embedded devices in one's everyday life. It is getting even more important in the age of pervasive computing.

Side-channel attacks have been known to impose a serious threat to embedded systems such as smart cards or other embedded microcontrollers for the last decade. However, as applied to symmetric key algorithms, the toolbox of the attacker was mainly limited to techniques based on information leakage via power consumption and electromagnetic radiation of the devices [11], [10]. At the same time, timing analysis have been only very rarely utilized to analyze embedded implementations of block ciphers, being mainly applied in the domain of desktop and server PCs. This is partially due to the fact that many lightweight and low-cost embedded systems have been providing hardware implementations of symmetric key algorithms. Besides that, many lightweight platforms based on 8-bit or 16-bit CPUs run at such low frequencies that microarchitectural performance optimization such as caches are not necessary.

This apparent disregard of and disbalance against cache timing attacks in the context of embedded security does not seem justified anymore though, since the embedded landscape is rapidly changing nowadays. As the computing world goes pervasive, a steadily growing number of embedded applications require more computing power. 32-bit RISC ARM-type CPUs have become a standard choice in many embedded applications such as banking and payment terminals, mobile communications, JavaCard applications, mobile TV, multimedia, toll collect systems, smart phones, electronic tachographs, PDAs etc. More and more security-related functionality is being put into software instead of hardware. Even some smart card microcontrollers are migrating towards powerful and universal computing architectures based on an ARM core [16]. At the same time, ARM microprocessors do have cache memory with nontrivial behavior and are as a rule operated under multi-process operating systems such as Linux or Windows Embedded/-Mobile. As opposed to almost all lightweight 8-bit microcontrollers, these two facts make many embedded systems vulnerable to cache timing attacks, first of all those based on ARM-type CPUs.

Aiming to close this gap, we tackle the problem of applying cache timing attacks to embedded devices at the example of ARM9 microprocessors. Our findings presented in Table 2 show that our cache timing based techniques of new type apply well to the OpenSSL software implementation of AES on ARM9. This indicates that numerous real-world embedded applications relying on AES and using ARM-type CPUs can turn out vulnerable to cache-timing attacks. This might force us to reconsider the practical security of many embedded systems currently in use. Furthermore, based on our results, we recommend to take the threat of cache timing attacks into account when designing and evaluating new embedded systems with security functionality.

Organization of this paper. The remainder of the paper is organized as follows. In Section 2, the most relevant previous work is briefly outlined including the advanced methods of expanded second-round attacks. Section 3 presents our new differential attack technique based the diffusion properties of AES. We deal with the physical cache behavior enabling our attack in Section 4. The attacked embedded platform, its impact as well as our experiments and practical results are provided in Section 5. We conclude in Section 6.

2 Previous Work on Cache-Collision Timing Attacks

In 1998 J. Kelsey *et al.* [8] analyzed the cache behavior of modern processors as a side channel against ciphers with large lookup tables like S-boxes. This proposal was established by D. Page [14] in the year 2002, who described and simulated a theoretical attack on DES. The first real-world implementation of such an attack was developed by Y. Tsunoo *et al.*[17] against DES and Triple-DES.

In general, cache attacks can be divided into three basic classes: trace driven, access driven, and time driven attacks. In trace driven attacks, the adversary is allowed to observe every single memory and cache access. Therefore, he knows when and where a collision occurs [14]. The access driven attacks provide the information which set of the cache is accessed by the cryptographic progress. For this, the cache is filled with data of the attacker. After the encryption the attacker checks which data is still present in the cache [13].

Attacks presented in this paper belong, however, to the class of the time driven attacks. Here, information is obtained by observing the execution time which is influenced by cache hits and cache misses. In this case, the attacker can only capture the total execution time of the encryption and then make a statistical evaluation to extract key-related information. The basic idea of timing attacks was introduced by Kocher [9]. A considerably higher number of encryption samples is needed compared to trace driven attacks. However, time-driven attacks correspond to an attacker with most restricted attack potential and are typically much more realistic, thus, being valid for numerous real-world applications, especially on embedded systems.

The cache based timing techniques developed in this paper target implementations of the Advanced Encryption Standard (AES)[6]. AES is a symmetric block cipher standardized by the National Institute of Standards and Technology (NIST). Nowadays, it is the most used cipher. The algorithm behind AES, Rijndael [6], was designed by J. Daemen and V. Rijmen. The most common ways to implement the cipher are the straightforward implementation, which is used on 8-bit microprocessors, and the 32-bit transformation table implementation. The latter combines different round functions to five transformation tables, or T-tables. During an encryption one table is used for the last round, the remaining rounds are processed with the other four lookup tables. Since cache attacks exploit the cache hits of lookup tables, the 32-bit T-box implementation is a well suited target, because it offers five large lookup tables.

2.1 First-Round Attack

The first round attack is a basic attack which takes advantage of cache line collisions evoked in the first round of the encryption. A cache line collision appears if two entries of the same cache line are accessed. Since the first round of the 32-bit implementation is realized with four tables, only four input values of the round p'_i access the same table. For example, the values $p'_0, p'_4, p'_8,$

and p'_{12} are processed by the first transformation table $\mathbf{T}_0$. The first round input p'_i itself is computed by an XOR combination of the plaintext p_i and the corresponding key value k_i:

$$p'_i = p_i \oplus k_i \qquad \text{for } 0 \leq i < 16. \tag{1}$$

With a cache line collision the adversary can create a relation between two different key bytes. Such a collision is evoked if, for example, $\langle p'_i \rangle = \langle p'_j \rangle$, for $i, j \in \{0, 4, 8, 12\}$ and $i \neq j$, i.e., the most significant bits of the values are equal, ignoring the $(\log_2 \gamma)$ least significant bits, where γ indicates the number of table entries in one cache line. The resulting relation is $\Delta_{i,j} = \langle k_i \oplus k_j \rangle = \langle p_i \oplus p_j \rangle$. Using such relation the size of the key space is reduced from 2^{128} to 2^{68} possible keys, if $\gamma = 8$.

Since the execution time is influenced by cache hits, the cache line collisions can be detected using statistic methods, like calculating the the average encryption time of a sample with the same relation $\Delta_{i,j}$.

2.2 Second-Round Attack

The second round attack is based on the first round attack, but also considers collisions between the first and the second round of the encryption. To do so, the input values of the second round p'' with access to the same transfomation table are analyzed. For the first table $\mathbf{T}_0$ the following equations describe how the input values are computed:

$$p''_0 = 2 \bullet \mathbf{S}[p'_0] \oplus 3 \bullet \mathbf{S}[p'_5] \oplus \mathbf{S}[p'_{10}] \oplus \mathbf{S}[p'_{15}] \oplus k_{16} \tag{2}$$
$$p''_4 = 2 \bullet \mathbf{S}[p'_4] \oplus 3 \bullet \mathbf{S}[p'_9] \oplus \mathbf{S}[p'_{14}] \oplus \mathbf{S}[p'_3] \oplus k_{20} \tag{3}$$
$$p''_8 = 2 \bullet \mathbf{S}[p'_8] \oplus 3 \bullet \mathbf{S}[p'_{13}] \oplus \mathbf{S}[p'_2] \oplus \mathbf{S}[p'_7] \oplus k_{24} \tag{4}$$
$$p''_{12} = 2 \bullet \mathbf{S}[p'_{12}] \oplus 3 \bullet \mathbf{S}[p'_1] \oplus \mathbf{S}[p'_6] \oplus \mathbf{S}[p'_{11}] \oplus k_{28} \tag{5}$$

where $\mathbf{S}[x]$ and $\bullet$ stand for the AES S-box lookup for the value x and the finite field multiplication in $GF(2^8)$ as used in the AES. The key values k_i are the key bytes generated by the key scheduling algorithm. These values depend on the initial key. For instance, the value k_{24} is equivalent to $(\mathbf{S}[k_{13}] \oplus k_0 \oplus 01_{(16)} \oplus k_4 \oplus k_8)$ for AES-128. If a first round look up collides with a second round lookup, e.g., if $\langle p'_0 \rangle = \langle p''_8 \rangle$, we gain the following equation:

$$\begin{aligned}\langle p_0 \oplus k_0 \rangle = \langle 2 \bullet \mathbf{S}[p_8 \oplus k_8] \oplus 3 \bullet \mathbf{S}[p_{13} \oplus k_{13}] \oplus \mathbf{S}[p_2 \oplus k_2] \\ \oplus \mathbf{S}[p_7 \oplus k_7] \oplus \mathbf{S}[k_{13}] \oplus k_0 \oplus 01_{(16)} \oplus k_4 \oplus k_8 \rangle\end{aligned} \tag{6}$$

which leads to

$$\begin{aligned}\langle p_0 \rangle = \langle 2 \bullet \mathbf{S}[p_8 \oplus k_8] \oplus 3 \bullet \mathbf{S}[p_{13} \oplus k_{13}] \oplus \mathbf{S}[p_2 \oplus k_2] \\ \oplus \mathbf{S}[p_7 \oplus k_7] \oplus \mathbf{S}[k_{13}] \oplus 01_{(16)} \oplus \Delta_{4,8} \rangle.\end{aligned} \tag{7}$$

The adversary can now divide a large sample of plaintexts and encryption times into 2^{32} sets considering every combination for the key values k_2, k_7, k_8 and k_{13}, so that (7) is solved. The set with the correct key values should have the lowest encryption time. In a similar way, the complete key can be extracted.

2.3 Expanded Second-Round Attack

O. Acıiçmez *et al.*[1] improved the idea of the second round attack and created a chosen plaintext attack. For the expanded second round attack collisions between the first and the second round are considered, e.g., as described in (6). The difference in this attack is that the plaintext values p_0, p_2, p_7, and p_{13} are fixed for the entire sample. By combining all invariable parameters into one constant c, (6) is simplified to:

$$\langle p_0 \rangle = \langle 2 \bullet \mathbf{S}[p_8 \oplus k_8] \oplus c \rangle. \tag{8}$$

Since p_0 is a fixed value as well, the appearance of a cache line collision depends only on the value of $\langle p_8 \rangle$, i.e., γ values of p_8 evoke a cache collision. The adversary takes advantage of this fact by collecting a sample of encryption time and plaintext, where the fixed values remain the same. This sample is divided into 2^8 sets according to the value of p_8 of each plaintext. The sets which evoke the cache line collision have a lower average encryption time. To confirm the results a reference sample can be taken, where one fixed plaintext byte has another value. Using a similar procedure each key value can be reconstructed separately.

3 Differential Cache-Collision Attack Using Diffusion

Our cache timing technique is based on the notion of a wide collision where five pairs of AES S-boxes process pairwisely equal values. Though it can be made applicable to all AES versions, we will introduce it here at the example of AES-128. In order to be able to recover the key, the adversary needs to detect such wide collisions. Correspondingly, the attack flow consists of an online stage, a collision detection stage and a key recovery stage (the latter two being offline stages):

– In the *online stage*, pairs of chosen 16-byte plaintexts (P_1, P_2) are sent to the AES encryption routine. The adversary measures the time t required by the CPU to encrypt P_2, that is, the second plaintext in each pair.
  The output of the online stage to the next stages consists of the set of plaintext pairs (P_1, P_2) and the corresponding execution time values t.
– In the *collision detection stage*, the time values t are used to tell which plaintext pairs (P_1, P_2) lead to a wide collision. It is expected that if a wide collision occurs, t will be lower (results of five table lookups already in the cache memory). Otherwise, we expect t to be higher. Thus, the collision detection stage accepts sets of plaintext pairs and times output by the online stage and returns the set of plaintext pairs (P_1, P_2) that most probably lead to wide collisions.
– In the key recovery stage, one reconstructs AES key candidates from the list of plaintext pairs (P_1, P_2) which most probably result in wide collisions. These key candidates are then checked using a known plaintext-ciphertext pair.

Now, having realized the importance of wide collisions for our attack, we will first introduce this notion more formally. Then we will return to the online as well as key recovery stages afterwards. Collision detection is dealt with in Section 4.

3.1 Wide Collisions

In the attack, we always consider plaintexts pairwisely. More precisely, the pairs of plaintexts (P_1, P_2) are divided into pairs of main diagonals of the 4×4-byte AES state. A diagonal of P_1 is paired with the corresponding diagonal of P_2. In this way, four pairs are formed, marked with the same coloring:

$$P_1 = \begin{bmatrix} a_0 & d_1 & c_2 & b_3 \\ b_0 & a_1 & d_2 & c_3 \\ c_0 & b_1 & a_2 & d_3 \\ d_0 & c_1 & b_2 & a_3 \end{bmatrix} \qquad P_2 = \begin{bmatrix} e_0 & h_1 & g_2 & f_3 \\ f_0 & e_1 & h_2 & g_3 \\ g_0 & f_1 & e_2 & h_3 \\ h_0 & g_1 & f_2 & e_3 \end{bmatrix} \tag{9}$$

One of these pairs is the pair (A, E), where $A = \{a_i\}$ and $E = \{e_i\}$, for $0 \leq i < 4$. In the following description, we show how to extract a subset of key bytes at the example of the diagonal pair (A, E). The remaining key bytes can be extracted in a similar way using the other three diagonal pairs.
Let us form the plaintexts P_1 and P_2 in the following way:

- Byte values on the main diagonals A and E are chosen randomly and independently of each other with the only restriction that $A \neq E$ (four byte positions should not collide simultaneously).
- The remaining bytes of P_1 and P_2 are pairwisely equal but randomly chosen as well.

Then one obtains[3]:

$$P_1 = \begin{bmatrix} a_0 & x_4 & x_8 & x_{12} \\ x_1 & a_1 & x_9 & x_{13} \\ x_2 & x_6 & a_2 & x_{14} \\ x_3 & x_7 & x_{11} & a_3 \end{bmatrix} \qquad P_2 = \begin{bmatrix} e_0 & x_4 & x_8 & x_{12} \\ x_1 & e_1 & x_9 & x_{13} \\ x_2 & x_6 & e_2 & x_{14} \\ x_3 & x_7 & x_{11} & e_3 \end{bmatrix} \tag{10}$$

Now we will follow the propagation of this difference on the main diagonal up to the S-box layer of round 3. So, after the first round of the encryption, the plaintexts P_1 and P_2 are transformed into:

$$P_1' = \begin{bmatrix} a_0' & x_4' & x_8' & x_{12}' \\ a_1' & x_9' & x_{13}' & x_1' \\ a_2' & x_{14}' & x_2' & x_6' \\ a_3' & x_3' & x_7' & x_{11}' \end{bmatrix} \qquad P_2' = \begin{bmatrix} e_0' & x_4' & x_8' & x_{12}' \\ e_1' & x_9' & x_{13}' & x_1' \\ e_2' & x_{14}' & x_2' & x_6' \\ e_3' & x_3' & x_7' & x_{11}' \end{bmatrix} \tag{11}$$

Note that P_1' and P_2' differ only in the first column.

The MixColumns transform of the first column in the first round can provide collisions in up to three byte positions (four collisions would imply the non-bijectivety of AES which is not the case):

$$a_i' = e_i' \qquad \text{for some } i\text{'s in } 0 \leq i < 4. \tag{12}$$

Consider the first byte position with $i = 0$ as an example. Here we have two possibilities: either $a_0' = e_0'$ or $a_0' \neq e_0'$.

If the byte values collide $a_0' = e_0'$, which occurs with probability 1/256, one obtains 4 more byte collisions in the second round, as P_1' and P_2' are transformed by SubBytes and ShiftRows into P_1'' and P_2'':

$$P_1'' = \begin{bmatrix} a_0'' & x_4'' & x_8'' & x_{12}'' \\ x_9'' & x_{13}'' & x_1'' & a_1'' \\ x_2'' & x_6'' & a_2'' & x_{14}'' \\ x_{11}'' & a_3'' & x_3'' & x_7'' \end{bmatrix} \qquad P_2'' = \begin{bmatrix} e_0'' & x_4'' & x_8'' & x_{12}'' \\ x_9'' & x_{13}'' & x_1'' & e_1'' \\ x_2'' & x_6'' & e_2'' & x_{14}'' \\ x_{11}'' & e_3'' & x_3'' & x_7'' \end{bmatrix} \tag{13}$$

[3] In equations (10), (11), (13), and (14), the grey values mark the differing values of both states.

and the MixColumns operation of the second round outputs two equal columns:

$$P_1''' = \begin{bmatrix} x_0''' & y_4''' & y_8''' & y_{12}''' \\ x_1''' & y_5''' & y_9''' & y_{13}''' \\ x_2''' & y_6''' & y_{10}''' & y_{14}''' \\ x_3''' & y_7''' & y_{11}''' & y_{15}''' \end{bmatrix} \qquad P_2''' = \begin{bmatrix} x_0''' & z_4''' & z_8''' & z_{12}''' \\ x_1''' & z_5''' & z_9''' & z_{13}''' \\ x_2''' & z_6''' & z_{10}''' & z_{14}''' \\ x_3''' & z_7''' & z_{11}''' & z_{15}''' \end{bmatrix} \tag{14}$$

Only the values of the first columns are pairwisely equal, which leads to 4 S-boxes in the SubBytes layer of the third round to process pairwisely equal byte values and 5 S-box collisions in total. This is called a *wide collision.*

However, if $a_0' \neq e_0'$, which occurs with probability 255/256, one has $a_0'' \neq e_0''$ in (13). That is, only one byte position differs in the first columns of P_1'' and P_2''. Due to the MDS property of the MixColumns matrix acting on 4 byte values, all elements in the first column of P_1''' and P_2''' will be pairwisely different, since the matrix has branch number 5. This leads to 5 S-box non-collisions in total and is called a *wide non-collision.*

The average difference between the numbers of colliding and non-colliding S-boxes for wide collision and wide non-collision is 5. This discrepancy in the number of S-box collisions makes wide collisions much easier to detect against the background of wide non-collisions.

The intuition behind our cache timing collision attack is then that the average AES encryption time is detectably lower in the presence of a wide collision. We will deal with this kind of statistics in Section 4. In this section, we will discuss how the online stage is arranged and describe the procedure of key recovery based on a set of detected wide collisions for each diagonal.

3.2 Online Phase

In the online phase, the plaintexts P_1 and P_2 are generated in the way described above. A pair of diagonals[4] (A, E) is randomly chosen with the property $A \neq E$. For each of the 4 diagonals, this random choice of 8 byte values (4 for P_1 and 4 for P_2) is performed n times. That is, altogether, the online procedure described below is performed $4 \cdot n$ times.

For a fixed choice of A and E, the remaining state values of P_1 and P_2 are randomly chosen as well, but they are pairwisely equal for both plaintexts P_1 and P_2. That is, P_1 and P_2 are equal up to the main diagonals. For a fixed (A, E), this random choice is performed I times: For each of the $4 \cdot n$ choices, we run I such iterations. Each of these I iterations is repeated r times to ensure the stability of time measurements: We say that each iteration has r rounds.

In each of the r rounds, both plaintexts P_1 and P_2 are sent to the consecutive encryption. The time t required by the encryption of P_2 is captured. To improve the resolution of the time measurements, one can clear the cache memory before encrypting P_1. Moreover, the time interval between the two encryptions should be possibly short to avoid numerous cache accesses by other processes that might clear parts of the cache memories. With the cache cleared, the first encryption behaves like a random encryption in terms of cache usage and fills the cache memory with some lookup entries. Physically, to detect a wide collision, we want to observe how many entries on average are added to the cache by the encryption of P_2 after encrypting P_1. Therefore, t contains information about wide collisions and is stored together with the corresponding diagonal values (A, E) for the offline analysis.

Note that it is also possible to work with the complete encryption time for both plaintexts, but then more measurements are necessary to cancel out added by the encryption of P_1.

Thus, the major parameters influencing the complexity of the online stage (also referred to as *online complexity*) in our attack are n, I and r. The total number of AES encryptions required by the attack in the online phase will be $8 \cdot n \cdot I \cdot r$.

[4] Again, we explain the online phase at the example of the main diagonals A and E without loss of generality. All the other diagonals are attacked in a similar way.

3.3 Key Recovery

Algebraically, if a wide collision is detected, at least one byte at the end of the first AES round collides. So we have $a'_i = e'_i$ for some $0 \le i < 4$ (see also (11) and (12)). Every such expression binds the four key bytes on the same main diagonal. For instance, for $a'_0 = e'_0$ we will have the following equation:

$$\begin{gathered} 02 \cdot S(k_0 \oplus a_0) \oplus 03 \cdot S(k_5 \oplus a_1) \oplus 01 \cdot S(k_{10} \oplus a_2) \oplus 01 \cdot S(k_{15} \oplus a_3) \\ = \\ 02 \cdot S(k_0 \oplus e_0) \oplus 03 \cdot S(k_5 \oplus e_1) \oplus 01 \cdot S(k_{10} \oplus e_2) \oplus 01 \cdot S(k_{15} \oplus e_3), \end{gathered} \tag{15}$$

where k_0, k_5, k_{10} and k_{15} form the 4-byte subkey of the main diagonal. One obtains similar nonlinear equations with respect to 4-byte key chunks for all other possible byte collisions after the first round.

To recover each 4-byte subkey corresponding to each diagonal, we need at least four equations of type (15) and, thus, at least 16 in total for the full key recovery. For each diagonal, parameter n is chosen in a way that more than four collision candidates will be normally proposed by the collision detection stage. Assume that the collision detection stage proposes $4 + m$ collisions, $m \in \{0, 1, 2, \dots\}$. As a rule, the detection error probability will be nonzero, so that some of the proposed $4+m$ collision candidates will be non-collisions. Therefore, the key recovery procedure has to deal with this type of errors. We propose to do that in two steps as follows.

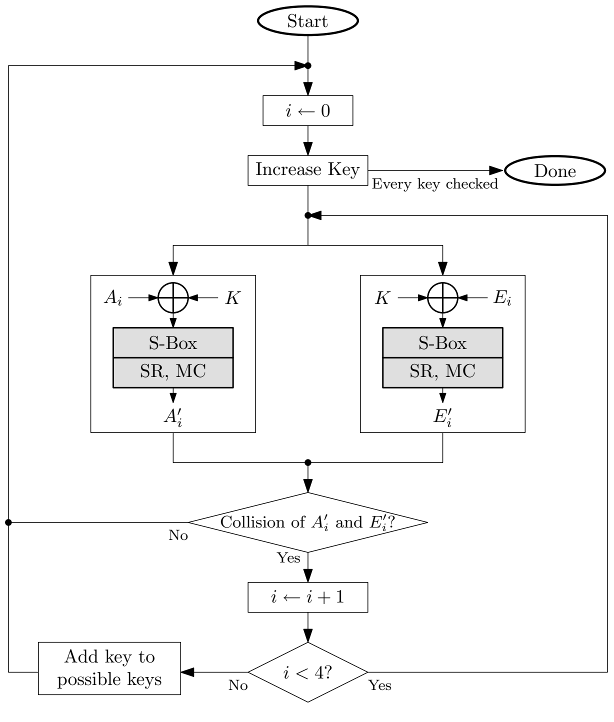

Fig. 1. Finding candidate subkeys from four diagonal pairs (A_i, E_i) for $0 \le i < 4$. K is a 4-byte subkey corresponding to some diagonal to be tested.

In the first step, we consider all possible $\binom{4+m}{4}$ choices of 4 collisions out of suggested $4 + m$ collisions. We also consider all possible 2^{32} subkey candidates for this diagonal. For each choice of 4 pairs (A_i, B_i), $0 \le i < 4$, and for each subkey candidate, we perform AddRoundKey, SubBytes, ShiftRows and MixColumns transforms as applies to the target diagonal. If the current choice of pairs (A_i, B_i) leads to collisions between all four pairs in some position[5] in the output column, the 4-byte subkey candidate survives and is added to the short list of subkey candidates. This process is visualized in Figure 1. On average, for each subkey test, one has to perform an amount of operations roughly comparable to 25% of an AES round, as a key candidate will only rarely survive the check with the first pair of diagonals. The complexity of the first step is approximately $4 \cdot \frac{1}{10} \cdot \frac{1}{4} \cdot \binom{4+m}{4} \cdot 2^{32} \approx \binom{4+m}{4} \cdot 2^{28.7}$ AES encryptions and can be optimized by taking into account that the values of subkey candidates are adjacent.

For the second step, consider how many subkey candidates survive for each of the four diagonals after the first step of key recovery. Since the positions of collisions after the first round are unknown, for each of the four diagonals there will be $4^4 = 256$ subkey candidates if $m = 0$. If $m > 0$, we expect to have $256 \cdot \binom{4+m}{4}$ subkey candidates.

In the second step of key recovery, all partial keys from the four short lists (one list for each diagonal) are concatenated to perform a final key test by computing a full AES encryption using a known plaintext-ciphertext pair. This final key test is executed for each key candidate. Having the estimated number of surviving subkey candidates after the first step in mind, the complexity of the second step of key recovery can be computed as $2^{32} \cdot \binom{4+m}{4}^4$ as one has to inspect each combination of subkeys.

Thus, the offline complexity of our attack is dominated by the second step of key recovery and can be estimated as $2^{32} \cdot \binom{4+m}{4}^4$ AES encryptions. See Section 5 for our experimental results.

4 Physical Behavior of Cache Hits on Embedded Platforms

Classical timing attacks [9, 21] are applicable in cases where the implemented algorithm features a data-dependent runtime. Hence, resistance is easily achievable by building constant run-time code.

The microarchitecture of modern CPUs contains several measures to speed up the execution of programs by methods such as instruction level parallelism, several caches and branch prediction units. The behavior of these microarchitectural measures is usually not considered by implementers, since most code written for modern embedded systems is supposed to be portable to different platforms. Even if an implementer would like to take the behavior of the underlying platform into account, this is often not possible since the processor interacts with different threads in an unpredictable manner. Furthermore, in some cases timing relevant behavior of a CPU is not documented [4]. Hence, code with a constant execution time is desirable, but not always possible.

Cache Behavior. A very common microarchitectural feature found on almost all modern 32-bit CPUs is the data cache. Cache is a small, but fast memory between the processor and the RAM. This is due to the fact that the processing speed of modern CPUs exceeds the access time of RAM by far. Caches are intended to overcome this bottleneck. The storage capacity of a cache is smaller compared to the main memory, but the cache can be accessed at a much higher speed than RAM. For the CPU, the cache is transparent. When a value from RAM is queried, the cache simply returns it if a copy of that value is currently in the cache. This is called a *cache hit*. If the value is not in the cache (a so-called *cache miss*), the cache queries the value from the larger RAM, passes it to the CPU and stores it in the cache. Of course the latter takes additional time, resulting in an increased runtime of the executed program for each cache miss. The cache itself is arranged in 2^l cache lines. Each of these lines can hold 2^b bytes. This leads to a complete cache size of $2^{(b+l)}$ bytes. For every queried value, a full cache line is loaded from the RAM, hence a few adjacent values to the queried one are also prefetched.

Several techniques to improve the basic operating mode of a cache have been proposed in order to improve the ratio of the cache hits and cache misses. In direct-mapped cache every

[5] Note that this position does not have to be the same for all four pairs of diagonals.

data from the main memory can only be stored in one specific cache line. This allows a very simple and fast verifying method to check if the data is cached at the cost of a rather high number of cache misses. In a fully associative cache the data can be stored in every cache line. To determine, if the needed data is cached, all entries must be checked. This takes a long time compared to the direct-mapped cache, but has the advantage that the amount of cache misses is very low. A combination of the advantages of both models is the n-way set associative cache. An entry can be stored in n possible cache lines. These cache lines are combined into one cache set. The n-way set associative cache is quite common in practice, as it provides a good tradeoff between cache hit time and cache miss ratio.

Target Platform. Since our goal is to evaluate the threat of cache timing attacks to modern embedded platforms, we chose a rather powerful ARM9 processor as target. Modern ARM 16/32 bit processors are used for many embedded applications where the demand for computing power is high and the power consumption is restricted. ARM cores can be found in most modern smart phones, portable game consoles and PDA's, but also in various other embedded electronics [7].

The hardware used as target is the Embest SBC2440-II single board computer. The board hosts a Samsung S3C2440A [15] microprocessor featuring an ARM9 core, namely an ARM920T [3]. Besides the ARM, the CPU provides functionality to handle the boards interfaces, such as an LCD controller. It is a typical chip to be found in modern PDAs, such as Nokia N810, palmOne Treo 600, etc. The CPU can be clocked at up to 400 MHz. We operated the Embest SBC2440II board with an open source ARM-Linux with a 2.6.13 linux kernel. The board alternatively features WindowsCE.

Cache Architecture of the ARM. The ARM920T core features a Harvard memory architecture with separate data and instruction cache. The caches have a size of 16 KB each and are divided into 512 cache lines of 8 four-byte words. Both caches are arranged in 64-way set-associative caches with each having eight sets of 64 cache lines. Each entry can be located in just one set, but in this set it can be stored in any of the set's cache lines. The bits 7 to 5 of the address define the set where the entry is located. The cache line itself can be determined by comparing the tag of the address, bits 31 to 8, with the tags stored in the cache. The bits 4 to 2 specify the word in the cache line and the bytes in a word can be addressed with the bits 1 to 0.

Attack Conditions. As a target for the attack we used the T-box implementation of the AES provided by the openSSL package [20]. Although our attack is not limited to the T-box implementation, we analyzed the cache behavior for this case, as the T-box implementation is the most common in practice. We assume that the attacker can encrypt two consecutive chosen plaintexts (or decrypt two consecutive chosen ciphertexts). The attacker can also measure the execution time of the second encryption only. Hence our attack is applicable in cases where the AES output can not be accessed. Other attacks such as all final round attacks are not possible in this case. The attack can be easily performed in cases where the attacker has full access to the system; a realistic assumption for many embedded applications. On many PDAs, smart phones etc., the user is able to execute own code directly or after jailbreaking the device. Hence, depending on the specific application, the adversary is able to query a commercial application protected by an AES and to overcome the security by measuring its execution time. Cache timing attacks can also be a viable measure to circumvent security protection mechanisms on the operating system layer such as sandboxing.

Cache Behavior of the ARM. As described earlier, wide collisions have a much stronger influence on the execution time than normal collisions. Figure 2 presents a histogram over the encryption time of the encryption of the second plaintext. The encryption time is visibly decreased in the case a wide collision occurs (shown in light gray). If no wide collision occurs, the execution time is slightly higher (dark gray bars). The difference between the two sets can be exploited by the attacker to perform the previously described attack.

Fig. 2. Histogram comparing the execution time of AES encryptions with and without wide collisions on the target platform

Them measurement setup should try to minimize the risk of non-collisions being detected as collisions. One big source of noise is structure of the cache, namely the existence of cache lines. In case of the T-box implementation up to eight table entries are loaded into a cache line for each cache miss. Consequently, even though no wide cache collision occurs, several unexpected collisions may occur. Please keep in mind that difference in Figure 2 is taken from real measurements. Effects like the non-perfect description of the cache behavior mentioned above are already included.

5 Experimental Results

All measurements were performed on the ARM platform described in Section 4. The ARM Board is set up as a server running the AES implementation of openSSL 0.9.8K and queried via the Ethernet interface of the board. Every challenge queries two consecutive AES encryptions.

As described earlier, four wide collisions per column are sufficient to extract the corresponding four bytes of the AES key with a remaining uncertainty of 2^8 key candidates. Hence, after finding four wide collisions for each of the four columns, a full AES-128 key can be recovered with a remaining computational complexity of 2^{32} AES computations. The amount of computations has to be increased if we do not assume a perfect collision detection, i.e., in the case we accept a certain number of false positives.

Our test setup has been optimized to minimize the number of false positives in the collision detection process. We evaluated different test settings by trying different test parameters. Besides the number different diagonals n, we tested I different plaintexts per diagonal. The tested parameters are $n = 256$, $n = 512$, and $n = 1024$ diagonals with $I = 200$, $I = 400$ and $I = 800$ different plaintexts each. To increase the reliability of the timing measurements, we repeat each measurement r times, where $r = 20$ or $r = 40$. Since measurement noise mostly increases the measured computation times, we calculated the average encryption time for one plaintext by using only the j fastest out of the r timing samples. Chosen parameters for j were the 2, 5, 10 and $j = r$ fastest measurements.

Table 1. Number of false positives m for one column with a success probability of 65%

	Parameters			**False positives**			
Test	$\boldsymbol{n}$	$\boldsymbol{I}$	$\boldsymbol{r}$	$j = 2$	$j = 5$	$j = 10$	$j = r$
1	800	600	40	1.78	1.56	1.33	1.56
2	1024	400	20	2.11	1.22	1.22	1.67
3	1024	400	40	0.89	0.89	0.67	0.89
4	1024	600	15	1.67	1.89	2	1.78
5	1024	800	40	0.13	0.13	0.38	0.38

Table 1 shows the expected number of false positives for one column (hence, four collisions and four revealed key bytes). All non-collisions that have a timing lower than or within the group of the four fastest real collisions are considered false positives. If we accept one false positive per column (i.e. a total of four false positives), the remaining key space is increased to about 2^{41} key candidates for the entire AES key. If two non-collisions are accepted per column, the complexity of the key search rises to approximately $2^{47.6}$, for three to $2^{52.5}$, and for four false positives to $2^{56.5}$ possibilities. With a highly optimized AES implementation such as the one of Hamburg [19], up to 2^{38} AES encryptions can be performed per hour on a modern core2duo desktop processor, resulting in a feasible attack, even in the case of one false positive per column.

Table 2. Number of measurements N and the size of the remaining key space K for our differential wide collision attack and the expanded second round attack by Aciiçmez *et. al.* [1]

Attack	**Test**	**N**	**Remaining key space K [2^x]**			
			$j = 2$	$j = 5$	$j = 10$	$j = r$
Our attack	**2**	65,536,000	49.7	43.8	43.8	47.4
	3	131,072,000	41.6	42.2	41.2	42.2
	5	262,144,000	36.9	36.9	38.7	37.5
Aciiçmez *et. al.* [1]		128,000,000				32

Table 2 summarizes the complexity of the key recovery step as number of remaining key candidates and complexity of the online phase as the needed number of traces for the different test cases described in Table 1. The results of our differential wide collision attack are compared to the expanded second round attack of Aciiçmez *et. al.* [1] on the same target platform, revealing that our attack has an increased complexity for the key recovery, but can successfully be performed with a lower number of measurements. All parameters of both attacks are for an expected success rate of 90%. Depending on the chosen attack parameters, we can approximately halve the number of needed measurements when compared to the expanded second round attack. In test setup 2, 66 million measurements suffice, with a remaining key space of $2^{43.8}$, resulting in a more realistic attack.

6 Conclusion

We presented a novel differential collision attack making use of the MDS properties of the AES algorithm. The attack outperforms previous attacks in the same adversarial scenario in terms of needed measurements and decreases the remaining key space far enough to be easily computable on a modern desktop PC.

We furthermore presented the first evaluation of the vulnerability of embedded platforms to cache timing attacks and showed that cache attacks are feasible in practical setups. We want to stress that cache attacks pose a serious threat, especially on embedded platforms. Developers of embedded software solutions relying security functionality should consider the threat of cache timing attacks when designing their systems.

Acknowledgements. During the work on this paper, Andrey Bogdanov was supported partially by the Fund for Scientific Research - Flanders (FWO) and partially by the Chair for Embedded Security at the Ruhr-University of Bochum.

References

1. O. Aciiçmez, W. Schindler, and Ç. K. Koç. Cache based remote timing attack on the AES. In *Topics in Cryptology — CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, pages 271–286. Springer-Verlag, 2007.

2. Advanced Encryption Standard. FIPS. Publication 197. National Bureau of Standards, U.S. Department of Commerce, 2001.
3. ARM Limited. *ARM920T Technical Reference Manual*, 1 edition.
4. D. J. Bernstein. Cache-timing attacks on AES. Technical report, Department of Mathematics, Statistics and Computer Science, The University of Illinois at Chicago, 2005. Available from `cr.yp.to/antiforgery/cachetiming-20050414.pdf`.
5. J. Bonneau and I. Mironov. Cache-collision timing attacks against AES. Technical report, Computer Science Department, Stanford University and Microsoft Research, Mountain View, CA, 2006.
6. J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag, 2002.
7. ARM INC. ARM Powered Products. `http://www.arm.com/markets/mobile_solutions/app.html`.
8. J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side channel cryptanalysis of product ciphers. In *Journal of Computer Security*, pages 97–110. Springer-Verlag, 1998.
9. P.C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems. CRYPTO'96, Springer-Verlag, 1996.
10. P.C. Kocher, J. Jaffe, B. Jun. Differential Power Analysis. CRYPTO'99, LNCS, Springer-Verlag, 1999.
11. S. Mangard, E. Oswald, T. Popp. Power Analysis Attacks and Countermeasures for Cryptographic Smart Cards: Revealing the Secrets of Smart Cards. Springer-Verlag, 2007.
12. M. Neve, J. Seifert, and Z. Wang. Cache time-behavior analysis on AES. Available from `http://www.cryptologie.be/document/Publications/AsiaCSSfull06.pdf`, 2006.
13. D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006*, pages 1–20. Springer-Verlag, 2006.
14. D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical report, 2002.
15. Samsung Electronics. *S3C2440A 32-Bit CMOS Microcontroller User's Manual*, 1 edition.
16. ST33F1M. Smartcard MCU with 32-bit ARM. Available from `http://www.st.com/stonline/books/pdf/docs/15066.pdf`
17. Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. Cryptanalysis of DES implemented on computers with cache. In C. D. Walter, Çetin Kaya Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, volume 2779 of *LNCS*, pages 62–76. Springer, 2003.
18. Y. Tsunoo, E. Tsujihara, M. Shigeri, H. Kubo, K. Minematsu. Improving cache attacks by considering cipher structure. Int. J. Inf. Secur. 5(3), pp. 166—176, 2006.
19. M. Hamburg. Accelerating AES with Vector Permute Instructions. In *CHES*, pages 18–32, 2009.
20. OpenSSL 0.9.8.K. Openssl: The open source toolkit for ssl/tls. `http://www.openssl.org/`. [Online; accessed 18-June-2009].
21. J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems. A Practical Implementation of the Timing Attack. In *Smart Card Research and Applications, CARDIS '98*, pages 167–182, 1998.