Improved Trace-Driven Cache-Collision Attacks against Embedded AES Implementations*

Jean-François Gallais[1], Ilya Kizhvatov[1], and Michael Tunstall[2]

[1] Université du Luxembourg
6, rue Richard Coudenhove-Kalergi, L-1359 Luxembourg
{jean-francois.gallais,ilya.kizhvatov}@uni.lu

[2] Department of Computer Science, University of Bristol
Merchant Venturers Building, Woodland Road
Bristol BS8 1UB, United Kingdom
tunstall@cs.bris.ac.uk

Abstract. In this paper we present two attacks that exploit cache events, which are visible in some side channel, to derive a secret key used in an implementation of AES. The first is an improvement of an adaptive chosen plaintext attack presented at ACISP 2006. The second is a new known plaintext attack that can recover a 128-bit key with approximately 30 measurements to reduce the number of key hypotheses to 2^{28}. This is comparable to classical Differential Power Analysis; however, our attacks are able to overcome certain masking techniques. We also show how to deal with unreliable cache event detection in the real-life measurement scenario and present practical explorations on a 32-bit ARM microprocessor.

Keywords: Side channel attacks, power analysis, cache attacks, AES

1 Introduction

Fetching data from the random access memory or non-volatile memory in embedded microprocessors can take a significant number of clock cycles and a processor is unable to perform any further instructions while it waits. The use of cache memory aims to decrease the cost of memory accesses. Cache memory is a memory held within the core of the microprocessor that can be accessed rapidly. When data is accessed the line of data holding this address is moved to the cache, where the amount of data moved is dictated by the architecture of the cache. This is based on the assumption that when a certain address is accessed it is likely that the data around this address is also likely to be accessed in the near future.

It has been noted that the power consumption of a microprocessor is dependent on the instruction being executed and on any data being manipulated [9, 14]. An attacker can, therefore, observe where functions, and sequences of functions,

* An extended abstract of this paper will appear at WISA 2010. This is the full version.

occur in a power consumption trace. This could, potentially, allow an attacker to derive information on cryptographic keys if an observed sequence is affected by the value of the key. It has also been observed that the electromagnetic field around a microprocessor also has this property [12, 23].

In this paper we consider the effect of a cache on an instantiation of AES. Given the above observation, cache accesses should be visible in the power consumption or electromagnetic emanations. The location of these cache accesses during the computation of AES has been shown to reveal information on the secret key used [4, 11]. In this paper we present an attack that represents a significant improvement over the adaptive chosen plaintext trace-driven attack [11] with a new adaptive algorithm for choosing plaintexts for recovering of 60 bits of the key from an expected 14.5 acquisitions. We also present a new known-plaintext attack requiring only 30 traces and an exhaustive search in 2^{28} hypotheses. Both attacks can tolerate uncertainties in observing a sequence of cache events, and a partially preloaded cache, as described in [8]. We described some experiments on a 32-bit ARM microcontroller, while all the previous works on trace driven attacks considered simulations of cache accesses [16, 1, 8, 25] or of the power consumption [4].

We consider the implementation of an AES that would be used in a secure microprocessor. That is, an implementation that only uses one lookup table to 256 bytes that can be written to RAM as a masked random ordered table. This would prevent an attacker from being able to apply differential power analysis [17]. This would not be practical in an implementation that uses so-called T-tables that allow a fast implementation on x86 microprocessors. Previous work on observing traces of cache events has primarily been involved in attacking implementations that use T-tables [16, 1, 8, 25]. Our approach assumes that lookup tables used are aligned with the cache, which will be the case for an optimized implementation, as opposed to the recent work [25] that presented a trace-driven attack exploiting cache misalignment. We present our attacks for the cache organized in 16-byte lines, however the attacks are easily adaptable to other cache line sizes.

The rest of this paper is organized as follows. In Section 2 we describe the cache mechanism and previous work in analyzing cache access. We present an improved adaptive attack in Section 3, and extend this to a known plaintext attack in Section 4. In Section 5 we then present the results of practical explorations on a 32-bit ARM microcontroller, and explain how to conduct our attacks where a detected sequence of cache hits and misses may be incorrect.

2 Generalities and Previous Work

2.1 Caching and Performance

The gap between the increased speed at which modern microprocessors treat data and the comparatively slow latencies required to fetch the data from the Non-Volatile Memories to the registers raise performance issues. To reduce the

"distance" between the CPU and the NVM, i.e. the number of wasted clock cycles for which the CPU has to wait for the data, the solution is to keep them quickly accessible in a faster memory. Faster, however, typically means more expensive, hence this choice affects the size of available fast storage memory, the so-called *cache memory*. Examples of embedded devices with cache memory are the microprocessors of the widespread ARM9 family and of the subsequent ARM families [2].

Concretely, modern microprocessors typically come with a SRAM cache memory. When a byte of data must be paged in during the computation, the processor first looks for in the cache. If present in the cache, this results in a **cache hit**, the data is brought to the registers within a single clock cycle without stalling the pipeline. If not present in the cache, this results in the **cache miss**, and the desired data fetched from Non-Volatile Memory (NVM), and the entire line containing the desired data is loaded into the cache. As suggested by the different technologies used in the cache and main memory, a cache miss typically takes more clock cycles and consumes more energy than a cache hit.

2.2 Cache-based Attacks against AES

Following the pioneering articles of Kelsey *et al.* [13] and Page [21], several notorious attacks have been published involving the cache mechanism and targeting AES. Cache-based attacks fall into three different types. *Time-driven attacks* exploit the dependence of the execution time of an algorithm on the cache accesses. Bernstein described a simple cache-timing attack leading to a complete key recovery on a remote server [3]. In *access-driven attacks* presented in [18, 20], an attacker learns which cache lines were accessed during the execution by pre-loading the cache with the chosen data.

Here, we elaborate on *trace-driven attacks*. In this type of cache attacks an adversary derives information from individual cache events from the side-channel trace of an execution, such as registered power consumption or electromagnetic emanations. Trace-driven attacks pose a particular threat to embedded devices since the latter are exposed to a high risk of power or electromagnetic analysis, as opposed to desktop and server implementations that are a usual target in access- and time-driven cache attacks (however, a cache-timing attack on embedded AES implementation was presented recently in [6]).

Previous work on trace-driven attacks was described in [4, 16, 1, 11, 25]. However, most of these works target an optimized AES implementation that uses large lookup tables, as described in the original Rijndael proposal [10]. Here we focus on a conventional 256-byte lookup table since this would often be the choice in a constrained device, e.g. in a smart card, for the reasons outlined above in the Introduction. Also, previous works did not tackle the unreliable cache event detection, at most considering the setting when a lookup table is partially preloaded into the cache [8]. In [4, 16, 1, 8], the effect of cache organization, and in particular the cache line size, on the attack was considered. Here we develop our attacks assuming the cache line size is 16 bytes, but they can be easily adapted to other sizes. Another popular cache line size is 32 bytes; in this case our attack

will be more complex (however, the dependency of the attack complexity on the cache organization is not straightforward, as detailed in [8]).

We also note that there is a similarity between cache attacks and side channel collision attacks [24, 5], as already observed in [16], hence the name *cache-collision* attacks.

2.3 Notation

We denote the most significant nibble of a byte b with $\widehat{b}$. In the same manner, the least significant nibble of b is denoted $\check{b}$. We denote the input of the `SubByte` function in the first AES round as x_i, equal to $p_i \oplus k_i$, where p_i and k_i respectively represent a byte of plaintext and key in blocks of 16 bytes. We index the bytes row-wise and *not* column-wise as in the AES specification [26, 10], *i.e.* in our notation p_0, p_1, p_2, p_3 is the first row of a 16-bit plaintext. We assume that in an embedded software AES implementation S-Box lookups are performed row-wise, and indexing bytes in the order of S-Box computation simplifies description of our algorithms. We denote addition and multiplication over $\mathrm{GF}(2^8)$ by $\oplus$ and $\bullet$ respectively.

2.4 Adaptive Chosen Plaintext Attack

In this section we recall the trace-driven cache-collision attack presented in [11]. It uses an adaptive chosen plaintext strategy, i.e. each plaintext is chosen according to the result of the analysis done beforehand.

The method presented in [11] targets the AES block cipher with the `SubByte` function implemented as a single lookup table. It is assumed that the cache contains no AES data before the encryption. This can easily be done by resetting the device. During the AES encryption, the `AddRoundKey` adds the bytes of the key $K = (k_0, k_1, \ldots, k_{15}) \in (\mathbb{F}_{2^8})^{16}$ and the plaintext $P = (p_0, p_1, \ldots, p_{15}) \in (\mathbb{F}_{2^8})^{16}$. The state produced is $(x_0, x_1, \ldots, x_{15}) = (k_0 \oplus p_0, k_1 \oplus p_1, \ldots, k_{15} \oplus p_{15})$. The `SubByte` function denoted $S(\cdot)$ is a permutation over $\mathbb{F}_{2^8}$ and its elements are pre-computed and stored in Non-Volatile Memory (NVM). It is performed using a lookup table containing 16 lines with 16 entries, each line being associated to the most significant nibble of the input byte (as detailed in [26]). Because the cache is assumed to contain no AES data, the entire line indexed by the upper nibble of $S(x_0)$ is loaded from the NVM to the cache, inducing a first cache miss. The second lookup, indexed by x_1, will be a cache hit with probability $\frac{1}{16}$, as there is 1 chance over 16 that the values $S(x_0)$ and $S(x_1)$ belong to the same line. If a cache hit occurs, then we have :

$$\widehat{k_0 \oplus p_0} = \widehat{k_1 \oplus p_1}$$

By rearranging the terms in the equation, we obtain :

$$\widehat{k_0 \oplus k_1} = \widehat{p_0 \oplus p_1}$$

An attacker can try to search among the 16 possible values for the upper nibble of p_1 and find the one inducing a cache hit within an expected number of $\sum_{i=1}^{16} \frac{i}{16} = 8,5$ acquisitions. Once the correct value for the second lookup is found, she can reiterate the process for the 14 other lookups. She will end up with the trace MHH...H and thus with the actual values for $\widehat{k_0 \oplus k_1}, \widehat{k_0 \oplus k_2}, \ldots, \widehat{k_0 \oplus k_{15}}$ which reduces the key search space by 60 bits. Expected number of plaintexts (traces) is $15 \times 8,5 = 127,5$, the worst-case complexity is $16 \times 15 = 240$ traces. Algorithm 1 presents this adaptive strategy.

Algorithm 1: Adaptive Chosen Plaintext Trace-Driven Cache-Collision Attack [11]

Input: AES
  Plaintext $\leftarrow (0, 0, 0, \ldots, 0)$
  $i \leftarrow 1$
  while $i < 16$ **do**
    CacheTrace $\leftarrow$ AES(Plaintext)
    if CacheTrace[i] == Miss **then**
      $\widehat{\text{Plaintext}}[i]$++
    else
      i++
    end if
  end while
Output: Plaintext=$(p_0, p_1, p_2, \ldots, p_{15})$

In Section 3 we show how to improve this algorithm to significantly reduce the number of required traces.

3 Improved Chosen Plaintext Attack

In this section, we show that the adaptive chosen plaintext attack described in [11] does not optimally exploit the adaptive scenario. We present an improved adaptive strategy that significantly reduces the number of required plaintexts.

We observe that in Algorithm 1 the plaintexts at each step are chosen *independently* of the plaintexts in the previous steps. More precisely, ignored are the events located in the cache trace to the right of the current event (for which the plaintext nibble is being chosen). Below we show that by observing these events we can drastically reduce the total number of plaintexts required to achieve the desired trace MHH...H.

We make use of the fact that a miss at position i, $0 < i < 16$, indicates that $\widehat{p_j \oplus k_j} \neq \widehat{p_i \oplus k_i}$ for all j such that $0 \leqslant j < i$ and event at position j is a miss. This means that any plaintext with the particular difference $\widehat{p_j \oplus p_i}$ between the nibbles in positions j and i will not lead to the desired trace MHH...H. So the

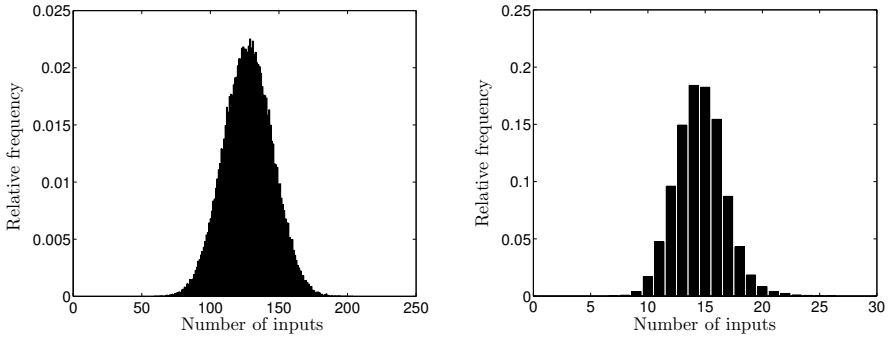
plaintexts with this difference can be omitted from the subsequent queries. On the other hand, if there is a hit at position i, the plaintext nibble in this position may already be the one which we are searching for, i.e. satisfying $\widehat{p_0 \oplus k_0} = \widehat{p_i \oplus k_i}$. So we cannot do better than keeping it for the next query, changing it only if the event in position i becomes a miss in subsequent queries.

Algorithm 2: Improved Chosen Plaintext Trace-Driven Cache-Collision Attack

Input: AES
  Plaintext $\leftarrow \{0\}_{16}$
  Constraints $\leftarrow \{0\}_{16,16}$
  $i \leftarrow 1$
  while $i < 16$ **do**
    Plaintext $\leftarrow$ SelectNextPlaintext(i,Plaintext,Constraints)
    CacheTrace $\leftarrow$ AES(Plaintext)
    for j from i to 15 **do**
      if CacheTrace[j] == Miss **then**
        for k from 0 to $j-1$ **do**
          if CacheTrace[k] == Miss **then**
            Constraints[j][k] $\longleftarrow$ Constraints[j][k] $\cup$ $\widehat{\text{Plaintext}}[j] \oplus \widehat{\text{Plaintext}}[k]$
          end if
        end for
      end if
    end for
    while (CacheTrace[i] == Hit) AND ($i < 16$) **do**
      i++
    end while
  end while
Output: Plaintext=$\{p_l\}_{l=0}^{15}$

The formal description of our improved strategy is given in Algorithm 2. The algorithm terminates with a plaintext $(p_0, p_1, \ldots, p_{15})$ yielding the desired cache trace MHH...H and thus the full chain for $k_0, \ldots, k_{15}$ reducing the key search space by 60 bits. Due to space limitations we omit the details of selecting the next plaintext based on the constraints, denoting this part as SelectNextPlaintext routine. In every miss position j (except for the initial miss in position zero), this routine chooses a plaintext nibble satisfying all the constraints with j preceding nibbles, or, in case there no such plaintext nibble, the next value of the nibble.

We have simulated this attack with 10^5 random keys both for the original Algorithm 1 [11] and our improved Algorithm 2. The results are shown in Figure 1. The improvement is drastic: on average 14.5 plaintexts for our improved attack to obtain a 60-bit reduction against 127.5 for the original attack. These figures are for the case of absolutely reliable cache event detection. In Section 5.3 we show that our improved algorithm has a good error tolerance.



(a) Original chosen plaintext attack [11] (b) Improved chosen plaintext attack

Fig. 1. Distribution of the number of plaintexts required to obtain a 60-bit reduction of the key search space.

4 Known Plaintext Attack

The improved attack in Section 3 requires adaptively chosen plaintexts. In this section, we present a known plaintext trace-driven cache-collision attack on AES-128 that enables full key recovery. The attack consists of two steps, namely, analyses of the first and second round cache access patterns.

4.1 Analysis of the First AES Round

We first analyze the cache events occurring in the first encryption round of AES with a sieve. The inputs to the sieve are N plaintexts and the corresponding cache traces that are obtained from N acquisitions. In a q-th acquisition, a plaintext $P^{(q)}$ is a 16-byte array and a cache trace $(CT)^{(q)}$ is the array of the cache accesses observed in the first round of AES, while encrypting $P^{(q)}$ under the unknown key K. The output of the sieve is a set of linear equations in the high nibbles of k_i, $i \in \{0, 15\}$ that decreases the entropy of the key search space by 60 bits.

In our strategy, the unknowns are defined as high nibbles of the XOR-difference between the first key byte and the 15 other key bytes. Hence, we aim to recover $\widehat{k_0 \oplus k_i}$ for which we define the set of possible initial values: $\boldsymbol{\kappa}_{0,i} = \{0, \ldots, 15\},\ 1 \leqslant i \leqslant 15$.

We recall from Sections 2.4 and 3 that the cache events observed in a power trace allow an attacker to determine whether at a certain lookup the S-Box input belongs to a previously loaded line of the lookup table or not.

If a cache hit occurs at the i-th lookup, one can state that the high nibble of the input of the S-box is equal to the high nibble of one and only one of the previous inputs that caused a cache miss. Hence the following statement holds:

$$CT_i = H \Longrightarrow \exists! j \in \Gamma,\ \widehat{k_i \oplus p_i} = \widehat{k_j \oplus p_j}$$

where Γ denotes the set of indices where a cache miss previously occurred in a trace.

Similarly, if a cache miss occurs at the i-th lookup, the high nibble of the input of the S-box is not equal to the high nibble of any of the previous inputs that caused a cache miss. Thus the statement:

$$CT_i = M \Longrightarrow \forall j \in \Gamma,\ \widehat{k_i \oplus p_i} \neq \widehat{k_j \oplus p_j}$$

Since $\widehat{k_i \oplus k_j} = \widehat{k_i \oplus k_0} \oplus \widehat{k_0 \oplus k_j}$, the terms in the above equations and inequations can be rearranged and we obtain:

$$CT_i = H \Longrightarrow \exists! j \in \Gamma,\ \widehat{k_i \oplus k_0} = \widehat{p_i \oplus p_j} \oplus \widehat{k_j \oplus k_0}$$

and

$$CT_i = M \Longrightarrow \forall j \in \Gamma,\ \widehat{k_i \oplus k_0} \neq \widehat{p_i \oplus p_j} \oplus \widehat{k_j \oplus k_0}$$

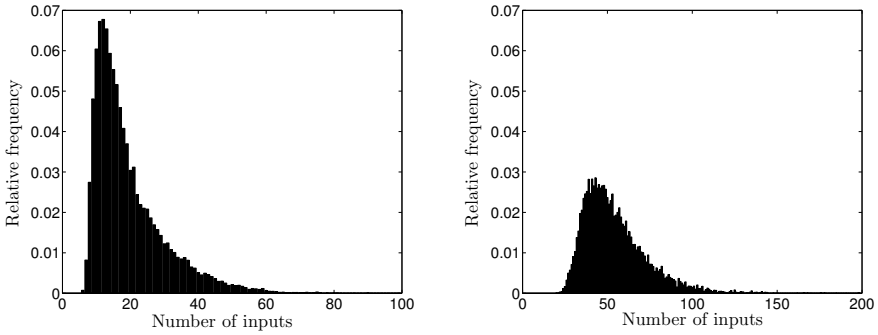
Algorithm 3: Known plaintext analysis of the first round

Input: $(P^{(q)}, CT^{(q)})\ q \in [1, N]$
  $\boldsymbol{\kappa}_{0,i} \leftarrow \{0, \ldots, 15\},\ 1 \leqslant i \leqslant 15$
  for $i \leftarrow 1$ to 15 **do**
    $q \leftarrow 0$
    while $|\boldsymbol{\kappa}_{0,i}| > 1$ **do**
      $q \leftarrow q + 1$
      $\boldsymbol{\kappa}' \leftarrow \varnothing$
      for $j \leftarrow 0$ to $i - 1$ **do**
        if $CT_j^{(q)} = Miss$ **then**
          $\boldsymbol{\kappa}' \leftarrow \boldsymbol{\kappa}' \cup \{\widehat{p_i^{(q)} \oplus p_j^{(q)}} \oplus \boldsymbol{\kappa}_{0,j}\}$
        end if
      end for
      if $CT_i^{(q)} = Miss$ **then**
        $\boldsymbol{\kappa}_{0,i} \leftarrow \boldsymbol{\kappa}_{0,i} \backslash \boldsymbol{\kappa}'$
      else if $CT_i^{(q)} = Hit$ **then**
        $\boldsymbol{\kappa}_{0,i} \leftarrow \boldsymbol{\kappa}_{0,i} \cap \boldsymbol{\kappa}'$
      end if
    end while
  end for
Output: $\boldsymbol{\kappa}_{0,i},\ i \in [1, 15]$

The sieve we developed uses the above statements to reduce the possibilities for $\widehat{k_i \oplus k_0}$. This is executed particularly efficiently if the right-hand sides of the equations are known, while the left-hand sides are the unknowns. This suggests to fix the lookup i and gain through the analysis of the plaintexts and cache events indexed from 0 to i the most information available on $\widehat{k_i \oplus k_0}$ so that only one possibility remains for this nibble. This can be achieved for a large enough

number of traces (see Figure 2). Once done, one can continue the analysis for the next values of i until 15. At the end, an attacker is left with only one possibility for each $\widehat{k_i \oplus k_0}$. The sieve is explicitly detailed in Algorithm 3.

To estimate the number of traces required to determine 60 bits, we ran the sieve for 10^5 simulated attacks, each with a random key. Figure 2(a) presents the results of this simulation. On average 19.43 acquisitions are required to reduce the entropy of the key to 68 bits. We note that this is less than for the original adaptive *chosen plaintext* attack of [11].



(a) Misses and hits (b) Misses only

Fig. 2. Distribution of the required number of plaintexts to obtain a 60-bit reduction of the key search space in the known plaintext attack.

Our strategy also works if we do not take into account the information available from the cache hits, that is, if $\boldsymbol{\kappa}_{0,i}$ is not modified when a cache hit is observed in a cache trace at the i-th lookup. The resulting analysis is less efficient than the one of the misses and the hits, as depicted in the plotted distribution of the number of required inputs (Figure 2(b)). Indeed, the average number of inputs required to perform a 60-bit reduction of the entropy of the key is in this case 54.19. However, we explain in Section 5.5 how this 'Misses-only' analysis can be useful if we assume a scenario where the cache already contains AES data before the encryption.

4.2 Analysis of the Second AES Round

After the first round analysis with our sieve we are left with 68 unknown key bits. In this section we show how to recover these remaining bits by analyzing the cache events occurring during the second AES encryption round. Our approach in general resembles that of [11], however we exploit the cache events much more efficiently. Similar approach was briefly sketched in [1], however they did not present the analysis for the number of traces required, whereas we perform theoretical analysis in Section 4.3. We assume that the round keys are

pre-computed and pre-stored, thus no access to the S-Box lookup table occurs between the encryption rounds[3].

The analysis of the second round consists of 3 phases:

1. From the first lookup, recover the following nibbles of the key: $\widehat{k}_0$, $\check{k}_0$, $\check{k}_5$, $\check{k}_7$, $\check{k}_{10}$, $\check{k}_{15}$, 24 bits in total.
2. From the second lookup, recover the following nibbles of the key: $\check{k}_1$, $\check{k}_6$, $\check{k}_{11}$, $\check{k}_{12}$, 16 bits in total.
3. Recover the remaining 28 bits by an exhaustive search over.

These phases re-use the known inputs from the first round analysis (we assume that a 2-round cache trace is acquired for each input) but in most cases require additional known inputs.

For simplicity and due to space limitations we will describe the analysis exploiting misses only, but hits can be exploited similarly, leading to an even more efficient attack. We describe our algorithm and then analyze its complexity in terms of required traces and computational workload.

First Lookup of the Second Round. In this step we will exploit the traces of the form M**...*|M, i.e. having a miss in the first lookup of the second round. This lookup is indexed by

$$y_0 = 2 \bullet s(x_0) \oplus 3 \bullet s(x_5) \oplus s(x_{10}) \oplus s(x_{15}) \oplus s(k_7) \oplus k_0 \oplus 1.$$

The fact that this lookup is a miss leads to the following system of inequations:

$$\begin{cases} \widehat{y}_0 \neq \widehat{x}_{j_1} \\ \dots \\ \widehat{y}_0 \neq \widehat{x}_{j_L} \end{cases}, \quad j_1, \dots, j_L \in \Gamma,$$

where Γ is set of indices of misses observed in the 16 previous lookups (i.e. in the first round), $|\Gamma| = L$. After rearranging the terms the system becomes

$$2 \bullet s(x_0) \oplus 3 \bullet s(x_5) \widehat{\oplus s(x_{10})} \oplus s(x_{15}) \oplus s(k_7) \neq \begin{cases} \widehat{\delta}_{j_1} \\ \dots \\ \widehat{\delta}_{j_L} \end{cases}, \quad j_1, \dots, j_L \in \Gamma, \quad (1)$$

where $\widehat{\delta}_j$ are some known values depending on the plaintext bytes and the key byte nibbles recovered in the first part of the analysis.

We have only 24 unknown bits in the left part of (1) since from the first round analysis we know the high nibble of the XOR difference between any two bytes of the key. Solving (1) for a single trace by exhaustive search over 2^{24}

[3] Meanwhile, the strategy presented here would be straightforward to adapt to an AES implementation with an on-the-fly key schedule, and a similar strategy can be applied using `xtimes` operation of AES `MixColumns` transform in case the former is implemented as a lookup table (see [11] for using `xtimes` in an adaptive attack).

candidates for these bits will leave us with some fraction of these candidates. The next trace will result in a different system of the form (1) and thus further reduce the amount of candidates. After several traces of the form M**...*|M we will remain with the $k_0, k_5, k_{10}, k_{15}, k_7$ completely recovered. We perform the analysis of the required number of traces in Sect. 4.3.

Second Lookup of the Second Round. Having finished with the analysis of the first lookup, we can exploit traces of the form M**...*|*M, i.e. having a miss in the second lookup of the second round. This lookup is indexed by

$$y_1 = 2 \bullet s(x_1) \oplus 3 \bullet s(x_6) \oplus s(x_{11}) \oplus s(x_{12}) \oplus s(k_7) \oplus k_0 \oplus k_1 \oplus 1$$

The fact that this lookup is a miss leads to the following system of inequations (after rearranging the terms):

$$2 \bullet s(x_1) \oplus 3 \bullet \widehat{s(x_6) \oplus s(x_{11})} \oplus s(x_{12}) \neq \begin{cases} \widehat{\delta}_{j_1} \\ \dots \\ \widehat{\delta}_{j_R} \end{cases}, \quad j_1, \dots, j_R \in \Gamma, \qquad (2)$$

where Γ is set of indices of misses observed in the 17 previous lookups (i.e. in the first round and in the first lookup of the second round), $|\Gamma| = R$, and $\widehat{\delta}_j$ are some known values depending on the plaintext bytes and the previously recovered nibbles of key bytes. Note that if the first lookup of the second round is a miss, one of the inequations in (2) emerges from $\widehat{y}_1 \neq \widehat{y}_0$. From the analysis of the first lookup we already know y_0 and thus can consider this inequation here.

We have only 16 unknown bits in the left part of (2), namely the nibbles $\check{k}_1$, $\check{k}_6$, $\check{k}_{11}$ and $\check{k}_{12}$, the rest having been recovered in the previous steps. Solving (2) for several traces of the form M**...*|*M, we will get a single candidate for these unknown nibbles. Analysis of the required number of traces for this step is performed in Sect. 4.3.

Brute Force over 2^{28} Key Candidates. After the analysis of the first and second lookups of the second round the remaining unknown key chunks are $\check{k}_2$, $\check{k}_3$, $\check{k}_4$, $\check{k}_8$, $\check{k}_9$, $\check{k}_{13}$, $\check{k}_{14}$. They comprise 28 bits in total and therefore can be recovered by exhaustive search.

4.3 Theoretical Analysis of the Second Round Attack Complexity

Here we calculate an estimate for the number of traces required for the second round attack. We note that in the analysis we will implicitly assume that the inputs to the second round lookups are statistically independent of the inputs to the first round lookups. Strictly speaking, this is not true, however the statistical dependency is not significant and so can be omitted for practical reasons.

First lookup. We have to remain with 1 candidate out of 2^{24}. Let us denote by F_1 an average fraction of candidates left by the system (1) for a given trace of the form M**...*|M. To remain with one candidate out of 2^{24} with N_1 traces of the form M**...*|M the following must hold:

$$2^{24} \cdot F_1^{N_1} \leqslant 1,$$
$$N_1 \geqslant \log_{F_1} 2^{-24}.$$

Since the left part of (1) can be viewed as a random mapping from a set of 2^{24} elements to a set of 2^4 elements, the probability for an element out of 2^{24} not to map to L elements given by the right part is $\frac{16-L}{16}$. Considering L as a random variable, $F_1 = \mathbf{E}\left(\frac{16-L}{16}\right) = 1 - \frac{1}{16}\mathbf{E}L$.

So to obtain the number of traces N_1 we have to calculate the distribution for L, i.e. for the average number of lines in cache after 16 lookups given the condition that 17-th lookup is a miss. Let T denote the (unconditional) number of S-Box lines in cache after 16 lookups. We can obtain the probability mass function $\Pr(L) = \Pr(T = s|CT_{17} = M)$ following Bayes' law:

$$\Pr(T = s|CT_{17} = M) = \frac{\Pr(CT_{17} = M|T = s) \cdot \Pr(T = s)}{\Pr(CT_{17} = M)}.$$

For a given s we have

$$\Pr(CT_{17} = M|T = s) = \frac{16 - s}{16}.$$

Next,

$$\Pr(CT_{17} = M) = 16 \cdot \left(\frac{15}{16}\right)^{16} \cdot \frac{1}{16} = \left(\frac{15}{16}\right)^{16}.$$

The distribution $\Pr(T = s)$ is a classical allocation problem [15]. We have

$$\Pr(T = s) = \binom{16}{16 - s} \left(\frac{s}{N}\right)^{16} \Pr_0(s),$$

where

$$\Pr_0(s) = \sum_{l=0}^{s} \binom{s}{l} (-1)^l \left(1 - \frac{l}{s}\right)^{16}.$$

Carrying out the numerical computations we obtain $\mathbf{E}L = 10.0263$, therefore $F_1 = 0.3734$ and the number of required traces of the form M***|M is $N = 16.885$. The total number of known plaintexts required for the analysis of the first lookup is then $N_1 / \Pr(CT_{17} = M) = 47.42$.

Second lookup. In this case the analysis is similar to the first round. We deal with the traces M***|*M that lead to systems of the form (2) with 16 unknown bits. The average fraction remaining after each trace is $F_2 = 1 - \frac{1}{16}\mathbf{E}R$ where R is number of lines in cache after 17 lookups given the condition that 18-th

lookup is a miss. Denoting here by T the unconditional number of lines in cache after 17 lookups, we have

$$\Pr(R) = \Pr(T = s | CT_{18} = M) = \frac{\Pr(CT_{18} = M | T = s) \cdot \Pr(T = s)}{\Pr(CT_{18} = M)},$$

where, as above,

$$\Pr(CT_{18} = M | R = s) = \frac{16 - s}{16}$$

and

$$\Pr(CT_{18} = M) = 16 \cdot \left(\frac{15}{16}\right)^{17} \cdot \frac{1}{16} = \left(\frac{15}{16}\right)^{17}.$$

The distribution $\Pr(T = s)$ is computed here for 17 lookups similarly to the above case. Numerical computations yield $\mathbf{E}R = 10.3579$ and $F_2 = 0.3526$. The number of required traces of the form M***|*M is

$$N_2 \geqslant \log_{F_2} 2^{-16} = 10.64.$$

The total number of known plaintexts required for the analysis of the second lookup is then $N_2 / \Pr(CT_{18} = M) = 31.87$. Note that the traces used in the analysis of the first lookup may be re-used here.

Computational complexity. The first phase of the second round analysis takes $O(2^{24})$ checks, while the second one takes $O(2^{16})$ checks. The final exhaustive search takes at most 2^{28} AES encryptions. The overall second round analysis complexity is hence about 2^{28} AES encryptions.

As already mentioned earlier, second round attack can exploit hits in a similar way as described above for misses, which leads to a reduction in the number of traces. Our estimations for the second round attack exploiting both hits and misses, done like in Section 4.3, show that an average of 28.15 traces is required in the analysis of the first lookup, and 19.59 traces (re-using the available ones) in the analysis of the second lookup.

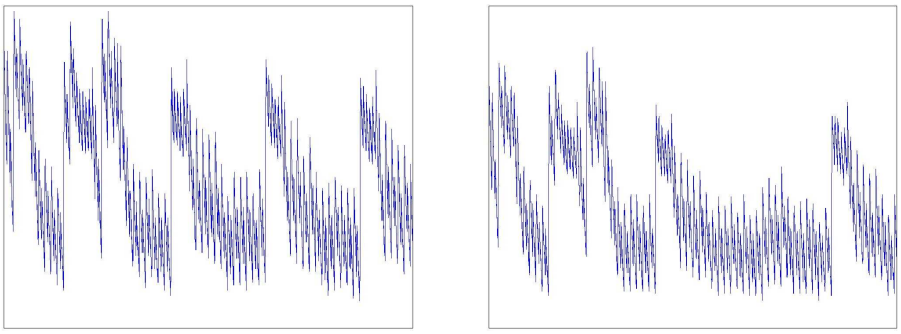
Thus, for the full AES-128 key recovery we require about 30 known plaintexts with the corresponding side-channel traces and an exhaustive search of 2^{28}. The attack will work in the same way for AES decryption. In the next section we demonstrate that our attacks can be performed in a real-life noisy environment.

5 Dealing with Detection Errors

In this section, we address the issue of detecting cache events in a real-life noisy environment. First, we show a practical example of distinguishing cache events in a power consumption trace of a 32-bit ARM microcontroller. Then we outline our general approach of dealing with detection errors and propose error-tolerant versions of our two attacks presented in Sections 3 and 4.

5.1 Practical Explorations

In order to demonstrate that one could observe a series of cache hits and misses in the power consumption, the AES was implemented on NXP LPC2124 [19], an ARM7TDMI microprocessor. Though ARM7 family devices do not normally feature a cache, this particular microprocessor features a Memory Acceleration Module (MAM). The MAM is in fact a cache that increases the efficiency of accesses to the flash memory. A series of power consumption traces were acquired during the computation of the `SubByte` function. Two examples of these acquisitions are shown in Figure 3.



(a) Power consumption trace showing cache misses for every memory access.

(b) Power consumption trace showing a cache hit on second memory access.

Fig. 3. Power consumption during the first three memory accesses of the `SubByte` function.

Each trace shows the first three memory accesses required to compute the `SubByte` function. In Figure 3(a) one can observe three memory accesses, as peaks in the power consumption, in the right hand side of the figure. In Figure 3(b), the power consumption during the same three memory accesses in the `SubByte` function is shown. One can see that the second peak in the power consumption is not visible, which corresponds to a cache hit. Note also that the amount of clock cycles is distinctly different, though not that clear from the picture as the difference in the power consumption.

From this we can observe that one can distinguish a cache hit from a cache miss in a straightforward manner. Recording a series of cache hits and misses could be automated by something as simple as setting a threshold and observing whether the power consumption passes this threshold.

Unfortunately, our ARM7TDMI microprocessor is not suitable for implementing the attack described in this paper, since the cache (MAM) consists of one line of 16 bytes. An attack would be possible where cache hits show where two adjacent memory accesses load the same cache line, but this is beyond the scope of this paper.

5.2 General Approach to Distinguishing Cache Events

We assume that we can measure some statistic in a side-channel trace like the height of the peak in the cycles corresponding to the table look-up, the value of the statistic being larger in case of a cache miss and smaller in case of a cache hit. In Section 5.1 we have shown that this is a sound assumption that holds in practice. We further assume that the statistic for hits and misses will follow the distributions that are close to normal (due to the noise that is usually Gaussian). Distinguishing between hits and misses is then a task of distinguishing between the two distributions.

A simple distinguishing solution would be in fixing a single threshold for the value of the statistic, like in practical collision detection of [7]. This will result in an unavoidable trade-off between Type I and Type II errors. However, in our algorithms both taking a miss for a hit and a hit for a miss will lead to the incorrect key recovery. Therefore, our approach is in fixing 2 thresholds t_H and t_M, $t_H \leqslant t_M$. In this setting, we distinguish between three types of events.

1. If the statistic is smaller than t_H we consider the event to be a hit.
2. If the statistic is larger than t_M, we consider the event to be a miss.
3. If the value of the statistic falls between the thresholds, we consider the event to be "uncertain".

We assume that the thresholds t_H and t_M are chosen such that it is highly unlikely for a miss to be misinterpreted as a hit and vice versa. We denote the probability of the "uncertain event" by error probability p. Below we show how the additional "uncertain" category helps in making our algorithms resistant to errors when p is non-zero. Obviously, our error-tolerant attacks require more traces in order to succeed in the presence of errors.

5.3 Error-Tolerant Improved Adaptive Chosen Plaintext Attack

To make our attack of Section 3 error-tolerant, we keep the plaintext nibbles unchanged if the event is "uncertain" since we cannot do anything better than wait for another trace. In the forward positions, this means that the errors are treated just as hits. In the current position, where a hit leads to the desired equation and thus to proceeding to the next position, in case of an "uncertain" event we keep the current plaintext nibble and proceed with the next trace. In fact, Algorithm 2 does not need to be changed: it automatically implements the described strategy when the cache trace includes 3 event types: Miss, Hit, Uncertain.

We performed 10^4 simulated attacks with random keys in the presence of detection errors. The results show that our Algorithm 2 tolerates errors very well. For the error probability 0.2 it requires 22.6 traces on average, and for the the error probability 0.5 – 47.2 traces on average. Note that this is better than for the original adaptive algorithm of [11] without detection errors.

5.4 Error-Tolerant Known Plaintext Attack

The analysis presented in Section 4.1 can also be adapted in order to find the linear dependencies between the upper nibbles of the key in the presence of uncertain cache events. The cache traces are now considered as arrays of 16 events, among the misses M, the hits H and the uncertain accesses U (being in the reality either M or H).

At the i-th lookup, if an uncertain event occurs in the q-th trace, $\boldsymbol{\kappa}_{0,i}$ is not modified and the analysis continues for i with the next input.

Then for another lookup $i' > i$ and the same trace q, $\boldsymbol{\kappa}'$ is computed and eventually will miss the value $\widehat{p_{i'} \oplus p_i} \oplus \boldsymbol{\kappa}_{0,i}$ if the uncertain event $CT_i^{(q)}$ was actually a cache miss. For the uncertain cache accesses in the q-th trace, another set $\boldsymbol{\kappa}*$ is computed, containing the value $\widehat{p_{i'} \oplus p_i} \oplus \boldsymbol{\kappa}_{0,i}$ involving the uncertain lookup i.

- If $CT_{i'}^{(q)} = M$, the sieve proceeds with subtracting $\boldsymbol{\kappa}'$ from $\boldsymbol{\kappa}_{i'}$. Since $\boldsymbol{\kappa}'$ is eventually smaller than it should be, there are no chances of evicting the correct value for $\widehat{k_{i'} \oplus k_0}$ from $\boldsymbol{\kappa}_{0,i'}$.
- If $CT_{i'}^{(q)} = H$, the sieve intersects $\boldsymbol{\kappa}_{i'}$ with $\boldsymbol{\kappa}' \cup \boldsymbol{\kappa}*$, such that the correct value, if ever present in $\boldsymbol{\kappa}*$ will not be evicted from $\boldsymbol{\kappa}_{0,i'}$.
- If $CT_{i'}^{(q)} = U$, no action is performed on $\boldsymbol{\kappa}_{0,i'}$, like previously done at the i-th lookup in the q-th input.

Our known plaintext strategy for the first AES round analysis in the presence of uncertain cache events is explicitly written in Algorithm 4. Simulated attacks were conducted for 10^4 random keys when the probability of uncertain events is 0.2 and 0.5. The average required numbers of traces are respectively 24.63 and 39.82. Note that for a probability equal to 0.5 the figure is less than that for the adaptive known plaintext attack reported above in Section 5.3.

The analysis of the second AES round can be adapted to be tolerant to uncertainties by treating them in the same manner.

5.5 Attacks With Partially Pre-Loaded Cache

Our attacks can tolerate the setting when the cache already contains some S-Box lines at the beginning of the first AES round in a manner similar to [8]. If the lookup table is partially loaded in the cache prior to the encryption, the cache trace will result in having more H than one could have got with a clean cache.

Our adaptive Algorithm 2 from Section 3 straightforwardly tolerates this setting because it exploits only misses, and a partially preloaded cache means that some misses will not be observed. This does not lead to an incorrect key recovery since we will not exclude the correct key hypotheses from our set but only leave some additional incorrect key hypotheses.

In the case of our known plaintext attack, the claims from Section 4.1 when a hit occurs at the i-th lookup are no longer true: there does not necessarily exist an index $j \in \Gamma$ such that $\widehat{x_i} = \widehat{x_j}$. If one applies the sieve described in

Algorithm 4: Known plaintext analysis of the first round with uncertain cache events

Input: $(P^{(q)}, CT^{(q)})$ $q \in [1, N]$
  $\boldsymbol{\kappa}_{0,i} \leftarrow \{0, \ldots, 15\},\ 1 \leqslant i \leqslant 15$
  for $i \leftarrow 1$ to 15 **do**
    $q \leftarrow 0$
    while $|\boldsymbol{\kappa}_{0,i}| > 1$ **do**
      $q \leftarrow q + 1$
      $\boldsymbol{\kappa}', \boldsymbol{\kappa}* \leftarrow \varnothing$
      for $j \leftarrow 0$ to $i - 1$ **do**
        if $CT_j^{(q)} = M$ **then**
          $\boldsymbol{\kappa}' \leftarrow \boldsymbol{\kappa}' \cup \{\widehat{p_i^{(q)} \oplus p_j^{(q)}} \oplus \boldsymbol{\kappa}_{0,j}\}$
        else if $CT_j^{(q)} = U$ **then**
          $\boldsymbol{\kappa}* \leftarrow \boldsymbol{\kappa} * \cup \{\widehat{p_i^{(q)} \oplus p_j^{(q)}} \oplus \boldsymbol{\kappa}_{0,j}\}$
        end if
      end for
      if $CT_i^{(q)} = M$ **then**
        $\boldsymbol{\kappa}_{0,i} \leftarrow \boldsymbol{\kappa}_{0,i} \backslash \boldsymbol{\kappa}'$
      else if $CT_i^{(q)} = H$ **then**
        $\boldsymbol{\kappa}_{0,i} \leftarrow \boldsymbol{\kappa}_{0,i} \cap (\boldsymbol{\kappa}' \cup \boldsymbol{\kappa}*)$
      end if
    end while
  end for
Output: $\boldsymbol{\kappa}_{0,i},\ i \in [1, 15]$

Section 4.1 to such inputs, when $CT_i = H$, the set $\boldsymbol{\kappa}_{0,i}$ will be intersected with a set $\boldsymbol{\kappa}'$ possibly not containing the correct value for $\widehat{k_i \oplus k_o}$, thus evicting the latter from $\boldsymbol{\kappa}_{0,i}$. However, when $CT_i = M$, one can subtract $\boldsymbol{\kappa}'$ from $\boldsymbol{\kappa}_{0,i}$ because the former contains only incorrect values for $\widehat{k_0 \oplus k_i}$, although $\boldsymbol{\kappa}'$ may be smaller than if the cache did not contain any lines prior to the encryption. This suggests, in order to avoid a failure in the key recovery, that the sieve should be adapted as mentioned in Section 4.1, i.e. to perform an action on $\boldsymbol{\kappa}_{0,i}$ only when misses occur. The analysis of the second round can exploit misses only in the same manner to tolerate the partially preloaded cache.

We performed simulated attacks with 10^4 random keys, when the cache is filled with 4, 8 and 12 lines of the lookup table, before the encryption starts. The average numbers of known inputs required for a 60-bit reduction of the key entropy are respectively 92.03, 158.02 and 271.10. We finally mention that in case a noisy environment is combined with a partially pre-loaded cache, our solutions described in Sections 5.4 and 5.5 are perfectly compatible, though requiring a higher number of inputs.

6 Conclusion

In this paper, we describe side channel analysis that can be applied to implementations of AES on embedded devices featuring a cache mechanism. We have improved the adaptive chosen plaintext attack described in [11] and presented a new known plaintext attack that recovers a 128-bit AES key with approximately 30 measurements and with an exhaustive search with 2^{28} remaining hypotheses. We have shown that both our attacks can tolerate the errors in determining cache events from a side channel trace that occur in a noisy environment, as well as the partially pre-loaded cache.

We stress that the complexity of our attacks is comparable to that of the first order Differential Power Analysis on unprotected software implementations. At the same time, cache-collision attacks are resistant to Boolean masking in the case where all S-Boxes share the same random mask, as detailed in [11]. When such a masking scheme is used, our attacks will outperform higher order DPA attacks that typically require thousands of traces.

The countermeasures against trace-driven cache-collision attacks have been discussed in the previous works on the subject [4, 16, 11, 8] and are similar to the countermeasures against cache attacks in general [22]. They include pre-fetching the lookup table into the cache prior to encryption and shuffling the order of table lookup computations.

Acknowledgements

The authors would like to thank the anonymous reviewers from CHES 2010 for their thorough comments and valuable suggestions. The work described in this paper has been supported in part by the European Commission IST Programme under Contract ICT-2007-216676 ECRYPT II and EPSRC grant EP/F039638/1 "Investigation of Power Analysis Attacks".

References

1. Acıiçmez, O., Koç, Ç.K.: Trace-driven cache attacks on AES (short paper). In: Ning, P., Qing, S., Li, N. (eds.) ICICS'06. LNCS, vol. 4307, pp. 112–121. Springer, Heidelberg (2006)
2. ARM Ltd.: Processors. http://www.arm.com/products/processors/ (2010)
3. Bernstein, D.J.: Cache-timing attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf (2004)
4. Bertoni, G., Zaccaria, V., Breveglieri, L., Monchiero, M., Palermo, G.: AES power attack based on induced cache miss and countermeasure. In: International Conference on Information Technology: Coding and Computing (ITCC'05). vol. 1, pp. 586–591. IEEE (2005)
5. Bogdanov, A.: Improved side-channel collision attacks on AES. In: Adams, C., Miri, A., Wiener, M. (eds.) SAC'07. LNCS, vol. 4876, pp. 84–95. Springer (2007)
6. Bogdanov, A., Eisenbarth, T., Paar, C., Wienecke, M.: Differential cache-collision timing attacks on AES with applications to embedded CPUs. In: CT-RSA'10. LNCS, vol. 5985, pp. 235–251. Springer (2010)

7. Bogdanov, A., Kizhvatov, I., Pyshkin, A.: Algebraic methods in side-channel collision attacks and practical collision detection. In: Chowdhury, D.R., Rijmen, V., Das, A. (eds.) INDOCRYPT'08. LNCS, vol. 5365, pp. 251–265. Springer (2008)
8. Bonneau, J.: Robust final-round cache-trace attacks against AES. Cryptology ePrint Archive, Report 2006/374 (2006), http://eprint.iacr.org/2006/374
9. Brier, E., Clavier, C., Olivier, F.: Correlation power analysis with a leakage model. In: Joye, M., Quisquater, J.J. (eds.) Cryptographic Hardware and Embedded Systems — CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer (2004)
10. Daemen, J., Rijmen, V.: The Design of Rijndael: AES – The Advanced Encryption Standard. Springer (2002)
11. Fournier, J., Tunstall, M.: Cache based power analysis attacks on AES. In: Batten, L.M., Safavi-Naini, R. (eds.) ACISP'06. LNCS, vol. 4058, pp. 17–28. Springer (2006)
12. Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic analysis: Concrete results. In: Koç, C.K., Naccache, D., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems — CHES 2001. LNCS, vol. 2162, pp. 251–261. Springer (2001)
13. Kelsey, J., Schneier, B., Wagner, D., Hall, C.: Side channel cryptanalysis of product ciphers. Journal of Computer Security 8, 141–158 (2000)
14. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M.J. (ed.) Advances in Cryptology — CRYPTO '99. LNCS, vol. 1666, pp. 388–397. Springer (1999)
15. Kolchin, V.F., Sevastyanov, B.A., Chistyakov, V.P.: Random Allocations. V. H. Winston & Sons, Washington, D.C. (1978)
16. Lauradoux, C.: Collision attacks on processors with cache and countermeasures. In: Wolf, C., Lucks, S., Yau, P.W. (eds.) WEWoRC 2005. Lecture Notes in Informatics, vol. P-74, pp. 76–85. Gesellschaft für Informatik, Bonn (2005)
17. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks: Revealing the Secrets of Smart Cards. Springer Verlag (2007)
18. Neve, M., Seifert, J.P.: Advances on access-driven cache attacks on AES. In: INDOCRYPT'08. LNCS, vol. 4356, pp. 147–162. Springer (2007)
19. NXP B.V.: LPC2114/2124 single-chip 16/32-bit microcontrollers. http://www.nxp.com/documents/data_sheet/LPC2114_2124.pdf (2007)
20. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: Pointcheval, D. (ed.) CT-RSA'06. LNCS, vol. 3860, pp. 1–20. Springer (2006)
21. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. Technical report CSTR-02-003, University of Bristol (2002)
22. Page, D.: Defending against cache-based side-channel attacks. Information Security Technical Report 8(P1), 30–44 (2003)
23. Quisquater, J.J., Samyde, D.: Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In: Attali, I., Jensen, T.P. (eds.) Smart Card Programming and Security, International Conference on Research in Smart Cards — E-smart 2001. LNCS, vol. 2140, pp. 200–210. Springer (2001)
24. Schramm, K., Leander, G., Felke, P., Paar, C.: A collision-attack on AES: Combining side channel- and differential-attack. In: Joye, M., Quisquater, J.J. (eds.) CHES'04. LNCS, vol. 3156, pp. 163–175. Springer (2004)
25. Zhao, X.J., Wang, T.: Improved cache trace attack on AES and CLEFIA by considering cache miss and S-box misalignment. Cryptology ePrint Archive, Report 2010/056 (2010), http://eprint.iacr.org/2010/056
26. FIPS PUB 197: Specification for the Advanced Encryption Standard (2001), http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf