Analysis of countermeasures against access driven cache attacks on AES

Johannes Blömer and Volker Krummel [*]
{bloemer,krummel}@uni-paderborn.de

Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn, Germany

Abstract. Cache attacks on implementations of cryptographic algorithms have turned out to be very powerful. Progress in processor design, e.g., like hyperthreading, requires to adapt models for tampering or side-channel attacks to cover cache attacks as well. Hence, in this paper we present a rather general model for cache attacks. Our model is stronger than recently used ones. We introduce the notions of information leakage and so called resistance to analyze the security of several implementations of AES. Furthermore, we analyze how to use random permutations to protect against cache attacks. By providing a successful attack on an AES implementation protected by random permutations we show that random permutations used in a straightforward manner are not enough to protect against cache attacks. Hence, to improve upon the security provided by random permutations, we describe the property a permutation must have in order to prevent the leakage of some key bits through cache attacks. Using a permutation having this property forces an adversary to consider several rounds of the cipher. This increases the complexity of any cache attack considerably. We also describe how to implement our countermeasure efficiently. The method to do so is of independent interest, since it alone can also be used to protect against cache attacks. Moreover, combining both countermeasures allows for a trade-off between security and efficiency.

Keywords: cache attacks, AES, threat model, countermeasures, random permutations

1 Introduction

Since Kocher published his work about timing attacks [12] in 1996 it is well known that observing the temporal behavior of an encryption algorithm may reveal information about the secret key. During the selection process of AES Koeune and Quisquater [13] showed that a careless implementation of Rijndael is susceptible to timing attacks. They used the fact that the time for the MixColumns operation depends on the values of the intermediate results. At that time table lookups were regarded as constant time operations and hence were not considered to be susceptible to timing attacks. However, due to the hierarchical organization of memory into fast cache and slow main memory, the assumption that table lookups use constant time required revision. In 2002 Page [19] presented a theoretical attack on DES that exploited timing information to deduce information about cache hits and misses, which in turn reveal information about secret keys being used. In the sequel we call attacks that exploit information about the cache behavior *cache based attacks* or CBAs. Tsunoo et al. [23] published a practical cache based attack against DES[1]. Further publications of Page [20], Percival [21], Bernstein [4], Osvik et al. [18] and Brickell et al. [8] disclosed the full power of cache based attacks. See [7, 5, 14, 17, 1, 2] for further improvements of cache based attacks. In particular, the fast AES implementation of Barreto [3] is susceptible to cache attacks. Note that Barreto's implementation is used in virtually all crypto libraries. It is susceptible to cache based attacks since it depends heavily on the usage of 5 large sboxes each of the size of 1024 bytes.

As was pointed out by Bernstein in [4], the threat model that is often implicitly used for cache attacks may not be strong enough. In particular, often it is assumed that the adversary $\mathcal{A}$ only can extract information from the cache before and after the encryption. This assumption is wrong from the theoretical point of view due to the process switching of the operating system. Moreover, it also has been practically disproved in [16]. Hence, several of the countermeasures proposed in the literature so far may not be effective. In this paper we present a stronger model to analyze cache attacks. We take into account powerful adversaries $\mathcal{A}$ that are able to obtain cache information even during the encryption. Within this model we show that using random permutations to mitigate the leakage of information as proposed in [8] is not an effective countermeasure. On one hand, we present a CBA that shows that random permutations do not increase the complexity of CBAs as much as one might expect. On the other hand, the same attack shows that a random permutation does

[*] This work was partially supported by grants from Intel Corporation, Portland

[1] See [8] for a more detailed description of the evolution of cache based side channel attacks.

not prevent the leakage of the complete secret key. We also consider a modified countermeasure based on random permutations. This countermeasure is quite general even though we present it in detail only for AES. Although we use permutations, we do not use arbitrary permutations. Instead we only use permutations that hedge a certain number of bits of the last round key in AES. By this we mean, that using our countermeasure a cache attack on the last round of AES, say, will only reveal about half the bits of the last round key. As one can see, this is the least amount of leaking information that can be provably protected by permutations. To determine the remaining bits, an attacker has to combine the cache attack with another attack, for example a cache attack on the next to last round. We give a mathematical precise description and analysis of the property of permutations that we need for our countermeasure. This analysis also sheds some light on the difference between the cache attack by Osvik et al. on the first two rounds of AES [18] and the attack of Brickell et al. on the tenth round of AES [8].

Furthermore, we analyze the security of several implementations of AES against cache attacks. One of these implementations is provably secure within our model. Suppose you want to compute a function like the AES sbox $S : \{0,1\}^8 \to \{0,1\}^8$ via table look-ups. Cache attacks become a threat if the complete table for S does not fit into a single so called cache line. In this case, we simply break the function S into four function $S_1, \ldots, S_4 : \{0,1\}^8 \to \{0,1\}^2$, say, such that $S(x) = (S_1(x), \ldots, S_4(x))$. If the smaller tables for the functions S_i fit into a single cache line, cache attacks are impossible. Of course, this countermeasure can be used with almost all tables by choosing suitable parameters for the size of the range of the S_i. In particular, this idea can also be used to protect the applications of permutations that are realized as table lookups. How to apply permutations securely has not been considered before.

The paper is organized as follows. In Section 2 we provide the technical background. After that we introduce our threat model in Section 3. In Section 4 we give our formal outline of a cache attack. We show that our formal cache attack is quite general. I.e., it covers the cache attacks on the first round of [18] and on the last round of [8]. In Section 5 we introduce our main security measures, information leakage and resistance. We use our security measures to analyze the security of several different implementations of AES in Section 6. In Section 7, first we consider random permutations as a countermeasure and describe a cache attack on this countermeasure. Then, we present and discuss an improved countermeasure using so called distinguished permutations. We finish with some extensions and remarks about future research.

2 Notation and technical preliminaries

In this section we give a short description of the technical background of cache based attacks, i.e., the memory management of modern computers. The memory of a computer should have two properties. It should be big and fast. However, with recent technologies these are contradictory properties. To achieve a fast data management, commonly used computers have a hierarchy of different types of memory with different sizes and different access times. They are separated into the *main memory* and different levels of *cache memories* (level1 cache, level2 cache etc.). The difference between these kinds of memory is their size and their speed. Generally speaking, smaller memory is faster. For a thorough treatment of computer architecture we refer to [10]. For this paper we simplify the situation by assuming that we have a slow main memory and only a single cache memory.

Cache memory is a fast but small memory that is placed between the processor and the main memory. It constitutes a trade-off between speed and size. On one hand, it is much larger than processor registers but much smaller than the main memory. On the other hand, it is much faster than main memory but slower than processor registers. Every memory transfer from main memory to the processor is redirected through the cache. Every time data is loaded from main memory it is checked whether the data is already in the cache. If that is true the data is loaded directly from the cache into the processor, avoiding access to the slow main memory. We call this a *cache hit*. If the data is not in the cache it is first loaded from the main memory into the cache and then transferred into the processor. We call this a *cache miss*. Hence, if a process wants to access the same data more than once within a short period of time, after the first access the data can be quickly reloaded from the cache.

In order to keep the administration of the cache simple, it is organized in so called *cache lines* of fixed size $|CL|$ bits, e.g., 512 bits. The main memory is partitioned into pieces of $|CL|$ bits. A simple function maps each of these pieces to a certain cache line. Every time a single byte of the main memory is accessed, the piece of main memory containing this byte is transferred to the cache. That means that data is always mapped to the same cache line[2]. Since the cache is much smaller than the main memory data will be pushed out of the cache from time to time.

In this paper we focus on exploiting the behavior of the cache mechanism for attacking cryptographic algorithms such as AES that use sboxes for encryption. In fast implementations, applications of the sboxes are realized via table lookups. However, one of the main security problems[3] of using table lookups for cryptographic purposes on common processors is that the processors cache is much faster than the main memory. Therefore, an encryption of a plaintext that uses cached data more often should be faster than an encryption of a plaintext that uses more accesses to the main memory. Hence, time measurements may reveal cache contents which in turn may leak information about the secret key. The situation with modern processors is even worse since the cache is shared between different processes. Although a process cannot read cached data of another process, it is able to push that data out of the cache. In [18] the authors give a detailed description of how these properties of modern processors can be exploited to attack fast implementations of AES with the help of cache timings (see also below).

In this paper we deal with two different implementations of AES. The first one is the *standard* implementation as described in FIPS 197 [15] and [9] which only uses the so called *standard sbox* $\mathbf{S}$ having 256 entries each of size 8 bit. The other implementation is the *fast* implementation [9, 3] that uses 5 larger sboxes $\mathbf{T}_0, \ldots, \mathbf{T}_3$ in rounds $1, \ldots, 9$ and the sbox $\mathbf{T}_4$ in round 10 of the encryption. Each $\mathbf{T}_i$ maps an element of $\{0,1\}^8$ to an element of $\{0,1\}^{32}$ and hence has size 2^{13} bits. See [3] for a detailed description.

3 Threat Model

We consider computers with a single processor, fast but small cache memory and large but slow main memory. Every time a process wants to read a word from the main memory a portion of data in the size of a cache line is transferred to the cache. An AES encryption or decryption process (AES process) is running on that computer that takes as input a plaintext (or ciphertext) and computes the corresponding AES ciphertext (or plaintext) with a fixed secret key k. To define our threat model we make several assumptions. We explain and justify each assumption with regard to our focus on cache based attacks. We start with a general assumption of cryptanalysis since it allows to simplify descriptions and analysis of attacks.

Assumption 1

1. *$\mathcal{A}$ knows all technical details about the underlying cryptographic algorithm and its implementation.*
2. *$\mathcal{A}$ can feed the AES process with chosen plaintexts (or ciphertexts) and gets the corresponding ciphertexts (or plaintexts).*

In particular, since we focus on information leakage due to table lookups, we assume that $\mathcal{A}$ knows the position of the sboxes in the memory and the possible cache lines they can be mapped to. Since variations of implementations are rather limited, the security of the implementation should not rely on keeping implementational aspects secret. This is the natural extension of Kerckhoffs' principle to implementation attacks. In order to give simple descriptions of the attacks, we model the interaction of $\mathcal{A}$ with the AES process such that $\mathcal{A}$ can use chosen plaintexts and chosen ciphertexts. However, every attack and countermeasure presented in this paper also works if we restrict $\mathcal{A}$ to use only known plaintexts.

Assumption 2 *$\mathcal{A}$ gets the indices of the cache lines that were accessed during the encryption (decryption) (*cache information*).*

Since $\mathcal{A}$ has access to the computer we assume that he can measure the timings of encryptions (decryptions) with reasonable precision. He can use this information to determine the accessed cache lines in a similar way as described in [11]. To build a strong model we simplify the determination of accessed cache lines in the following way. We assume that $\mathcal{A}$ simply gets the *correct* partition of the set of all cache lines M into the sets of accessed cache lines D_0 and the set D_1 of cache lines that were not accessed during the encryption. We call this partition *cache information.* The plaintext / ciphertext pair together with the cache information is called a *measurement.*

Assumption 3 *$\mathcal{A}$ can restrict the cache information to certain rounds of the encryption.*

[2] Modern processors are able to map data to a fixed number of different cache lines. This property is called associativity

[3] See Bernstein [4] for a thorough treatment of the technical problems.

This restriction is justified by the property of modern multitasking operating systems to change the active process after a constant amount of running time[4]. Hence, it is possible that the encryption process is interrupted by the attackers process, allowing $\mathcal{A}$ to access the cache during an encryption (decryption). In [4] Bernstein already warned that this property may be exploitable and the authors of [8] managed to exploit it to determine arbitrary cache information on a real PC with some reasonable precision.

Assumption 4 *$\mathcal{A}$ cannot distinguish between the elements of a single cache line.*

This assumption is justified because up to now it is not clear if it is technically possible to distinguish access times of elements within the same cache line. None of our attacks requires this somewhat difficult and unlikely ability of the adversary $\mathcal{A}$. Obviously, the ability to distinguish elements within the same cache line would allow more powerful cache attacks than the attacks published so far. Distinguishing elements that reside in the same cache line implies that the adversary gets the value of an intermediate result. To counteract such powerful attacks effectively requires expensive randomization techniques like the one proposed in [6]. All efficient countermeasures that were designed to counteract cache attacks so far rely essentially on this assumption. Likewise, the countermeasures presented in this paper are effective only under this assumption.

4 Access driven CBAs on AES

Under the assumptions of our thread model we can give the formal outline of a cache based attack. An attacker $\mathcal{A}$ who uses cache information to derive information about the secret key performs the following two steps:

1. $\mathcal{A}$ gets $n \in \mathbb{N}$ measurements $m^{(1)}, \ldots, m^{(n)}$ of encryptions of plaintexts $p^{(1)}, \ldots, p^{(n)}$ with the secret key k.
2. For each measurement $m^{(j)}$ the attacker $\mathcal{A}$ computes a set of possible values of an intermediate result $x^{(j)}$ of the encryption that only depends on the plaintext (or ciphertext), the ith byte k_i of the key k, and the obtained cache information. Depending on this set of values $\mathcal{A}$ computes a set $\widehat{K}_i^{(j)}$ of candidates for k_i. Finally, $\mathcal{A}$ combines the information of all measurements $m^{(1)}, \ldots, m^{(n)}$ by computing
$$\widehat{K}_i := \bigcap_{j=1}^{n} \widehat{K}_i^{(j)}.$$

Fig. 1. Formal outline of an access driven CBA

To illustrate the general structure of cache attacks let us briefly recall the CBAs on AES based on the first round of [18] and on the last round of [8]. For simplicity we assume that a cache line has size $|CL| = 512$ bits. Hence, an sbox $\mathbf{T}_0, \ldots, \mathbf{T}_4$ fits into 16 cache lines.

4.1 CBA on the first round of AES

We describe the CBA of [18] based on intermediate results of the first round. To be more precise, $\mathcal{A}$ focus on the result of the first application of an sbox in the first round. Since the involved sbox depends on the index i of the key byte we only consider the output $x_i = \mathbf{T}_{(i \mod 4)}[p_i \oplus k_i]$ of the sbox $\mathbf{T}_{(i \mod 4)}$. To simplify notation we simply write $x_i = \mathbf{T}[p_i \oplus k_i]$. For $0 \leq \ell \leq 15$ the sbox is mapped into the cache lines CL_ℓ as follows: $CL_\ell = \{\mathbf{T}[x] | x = \ell \cdot 16, \ldots, \ell \cdot 16 + 15\}$. To derive information about the ith byte of the secret key k an attacker performs the following operations according to the general structure shown in Figure 1:

1. $\mathcal{A}$ chooses $n \in \mathbb{N}$ plaintexts $p^{(1)}, \ldots, p^{(n)}$ that are fixed in byte $p_i^{(j)}$ and vary in the other bytes.
2. $\mathcal{A}$ obtains measurements $m^{(j)} = (D_0^{(j)}, D_1^{(j)}, p^{(j)})$ for $1 \leq j \leq n$.
3. $\mathcal{A}$ concludes that
$$x_i \in \widehat{X}^{(j)} = \bigcup_{\ell \in D_0^{(j)}} \{\ell \cdot 16, \ldots, \ell \cdot 16 + 15\}$$

[4] For further details see [22].

4. $\mathcal{A}$ computes the sets

$$\widehat{K}_i^{(j)} = \left\{ p_i^{(j)} \oplus \widehat{x}_i^{(j)} \mid \widehat{x}_i^{(j)} \in \widehat{X}^{(j)} \right\}$$

for all $1 \leq j \leq n$.

5. $\mathcal{A}$ computes the set

$$\widehat{K}_i = \bigcap_{j=1}^{n} \widehat{K}_i^{(j)}$$

of candidates for k_i.

In [18] the authors show that in this way, $\mathcal{A}$ is able to compute the 4 most significant bits of every key byte. They also show, that one can combine this attack with an attack on the second round to compute the complete key even if the cache information is taken over all 10 rounds.

4.2 CBA on the last round of AES

Next, we describe the CBA on the fast implementation of AES mentioned in [8] that is based on intermediate results of the last round of the encryption. Basing the attack on the last round has advantages over the attack on the first rounds of [18]. First, cache information of the last round is sufficient to determine all bits of the secret key. So $\mathcal{A}$ does not need to attack different rounds. Another advantage occurs if the encryption process uses the fast implementation of AES [3]. Here the sbox $\mathbf{T}_4$ of the last round is special and is only used in that round. This helps the attacker because cache information is never perturbed by cache accesses of other rounds. We show how an attacker can use cache information to determine bytes of the last round key k^{10}. Knowing all key bytes of the last round key allows to revert the key schedule and compute the cipher key k. We denote the ℓ-th cache line used for the table look-ups for $\mathbf{T}_4$ by $CL_\ell, \ell = 0, \ldots, 15$. Hence, CL_ℓ contains the tuples $\{\mathbf{T}_4[x] | x = 16 \cdot \ell, \ldots, 16 \cdot \ell + 15\}$. The structure of this CBA fits into the general structure shown in Figure 1. To derive information about the ith byte of the last round key k_i^{10} an attacker performs the following operations:

1. $\mathcal{A}$ chooses $n \in \mathbb{N}$ plaintexts $p^{(1)}, \ldots, p^{(n)}$ uniformly at random.
2. $\mathcal{A}$ obtains the ciphertexts and the measurements $m^{(j)} = (D_0^{(j)}, D_1^{(j)}, c^{(j)})$ for $1 \leq j \leq n$.
3. $\mathcal{A}$ concludes that

$$x_i^{(j)} \in \widehat{X}_i^{(j)} = \bigcup_{\ell \in D_0^{(j)}} \{\ell \cdot 16, \ldots, \ell \cdot 16 + 15\}$$

4. $\mathcal{A}$ computes the sets

$$\widehat{K}_i^{(j)} = \left\{ c_i^{(j)} \oplus \mathbf{S}\left[\widehat{x}_i^{(j)}\right] \mid \widehat{x}_i^{(j)} \in \widehat{X}_i^{(j)} \right\}$$

for all $1 \leq j \leq n$.

5. $\mathcal{A}$ computes the set

$$\widehat{K}_i = \bigcap_{j=1}^{n} \widehat{K}_i^{(j)}$$

of candidates for $k_i^{(10)}$.

If there is a single byte with this property, the adversary has determined k_i^{10}. Now it is not hard to see that the intersection of sets in step (5) eventually will contain only a single element iff for every $\delta \in \{0,1\}^8 \setminus \{0\}$ the following property holds

$$\exists j \in \{0, \ldots, 15\} \exists a \in CL_j : a \oplus \delta \notin CL_j. \tag{1}$$

We verified that the cache lines CL_j as defined above actually have this property. We will consider this property more closely when we consider countermeasures based on permutations in Section 7. Moreover, experiments show that on average approximately 15 pairs (p^j, c^j) together with the cache information D_0^j suffice to determine the key byte k_i^{10} uniquely.

5 Information leakage and resistance

CBAs are very powerful attacks. Although they seem to be unrealistic and hypothetical on first sight they were proven to be a real threat for implementations of cryptographic algorithms on computers with cache. Hence, a strong threat model is essential for a thorough security analysis. The threat model described above is stronger than the threat models published so far. The adversary is more powerful because $\mathcal{A}$ can restrict the cache information to a smaller interval of encryption operations. This reduces the number of accessed cache lines per measurement and increases the efficiency of cache based attacks. The main questions when analysing the security against CBAs are information leakage and complexity of a CBA. After giving a formal definition of information leakage we introduce the notion of the so called *resistance* of an implementation as a measure that allows to estimate the complexity of a CBA.

Information leakage The most important aspect of an implementation regarding the security against access driven CBAs is to determine the maximal amount of information that leaks via access driven CBAs. As we will see, the amount of leaking information about the secret key varies depending on the details of the CBA and the implementation of the cryptographic algorithm. We make the following definition:

Definition 1 (information leakage). *We consider an adversary who can mount a CBA using an arbitrary number of measurements as described in Assumption 2. Let $\widehat{\mathcal{K}}_i$ be the set of remaining key candidates for a key byte k_i^{10} at the end of the attack. Then the leaking information is $8 - \log_2\left(|\widehat{\mathcal{K}}_i|\right)$ bits.*

The amount of leaking information allows to estimate the uncertainty of an attacker about the secret key that remains after a successful access driven CBA. To quantify the maximal amount of information $\mathcal{A}$ can obtain about the secret key by access driven CBAs, we define $|CL|$ to be the size of a cache line in bits, $|S|$ the number of entries of the sbox and s the size of a single sbox element in bits. Hence, the number of elements that fits into a cache line is $\frac{|CL|}{s}$ and the cache information of a single measurement leaks at most $\log_2(|S|) - \log_2\left(\frac{|CL|}{s}\right) = \log_2\left(\frac{|S|}{|CL|} \cdot s\right)$ bits. Depending on the exact nature of an attack, the sets of measurements let the attacker reduce the number of remaining key candidates after the attack. The information leakage varies between 0 and 8 bits of information per byte. For example, the attack on the first round of [18] mounted on the fast implementation can determine at most 4 bits of every key byte regardless of the number of measurements. In contrast, the attack of [8] based on the last round allows an adversary to determine all key bits. Furthermore, in Section 6 we present an implementation that does not leak any information in our model.

Complexity of a CBA The information leakage as defined above measures the maximal amount of information a CBA can provide using an arbitrary number of measurements. Determining the expected number of measurements an attacker needs to obtain the complete leaking information depends on the details of the implementation and on details of the CBA. For simplification we introduce the notion of so called resistance. The resistance focuses on the general structure of a CBA as shown in Figure 1 and does not consider details of certain CBAs. It is a general measure to estimate the complexity of CBAs on different implementations.

Definition 2 (Resistance). *The resistance of an implementation is the expected number E_r of key candidates that are proven to be wrong during a single measurement that is based on r rounds of the encryption.*

The larger E_r the more susceptible is the implementation to access driven CBAs. In particular, if an implementation does not leak any information then an adversary cannot rule out key candidates and hence the resistance is 0. To compute E_r we assume that all sbox lookups are independently and uniformly distributed. This assumption is justified because an attacker $\mathcal{A}$ usually does not have any information about the distribution of the sbox lookups. Hence, the best he can do in an attack is to choose the parts of the plaintexts/ciphertexts that are not relevant for the attack uniformly at random.

Let m be the number of cache lines needed to store the complete sbox. Each cache line can store v elements of an sbox. Furthermore, let w be the number of sbox lookups per round and let r be the number of rounds the attack focuses on. In an access driven CBA a key candidate is proven to be incorrect if it causes an access of a cache line that was not accessed during a measurement. Assuming that all sbox lookups are uniformly distributed the probability that a cache line is not accessed in all $r \cdot w$ sbox lookups is

$$p_{\text{miss}} := \left(\frac{m-1}{m}\right)^{r \cdot w}.$$

Hence,

$$E_r := \left(\frac{m-1}{m}\right)^{r\cdot w} \cdot m \cdot v \tag{2}$$

is the expected number of key candidates that can be sorted out after a single measurement. However, the maximal amount of information an arbitrary number of measurements can reveal is limited by the information leakage. Further measurements will not reveal further information. We verified by experiments that the number of measurements needed to achieve the full information leakage only depends on E_r.

In the sequel, we focus on methods to counteract CBAs. In general, there are two approaches to counteract such a side channel. The first approach is to use some kind of randomization to ensure that the leaking information does not reveal information about the secret key. Using randomizing is a general strategy that protects against several kinds of side channel attacks, see for example [6]. In Section 7 we analyze a more efficient method based on random permutations. Before that, we consider the second approach that is to reduce the bandwith of the side channel. We present several implementations of AES and examine their information leakage and their resistance.

6 Countermeasure 1: Modify implementation

As Bernstein pointed out in [4] to thwart cache attacks it is not sufficient to load all sbox entries into the cache before accessing the sbox in order to compute an intermediate result because $\mathcal{A}$ can get cache information at all times. Hence, loading the complete sbox into the cache does not suffice to hide all cache information. Therefore, he advises to avoid the usage of table lookups in cryptographic algorithms. Computing the AES SubBytes operation according to its definition $f : \{0,1\}^8 \to \{0,1\}^8$, $x \mapsto a \cdot \mathrm{INV}(x) \oplus b$ would virtually cause no cache accesses and hence seems to be secure against CBAs. However, implementing SubBytes like this would result in a very inefficient implementation on a PC. To achieve a high level of efficiency people prefer to use precomputed tables. In the sequel, we analyze the security of some well known and some novel variations of implementations of AES. First, we explain the different implementations and after that examine the information leakage and the resistance as defined in (2) against CBAs:

the standard implementation as described in [9].
the fast implementation of Barreto as described in [3, 9].
fastV1 is based on the fast implementation. The only difference is that the sbox $\mathbf{T}_4$ of round 10 is replaced by the standard sbox as proposed in [8].
fastV2 is also based on the fast implementation but uses only sbox $\mathbf{T}_0$. The description of the fast implementation of AES shows that the ith entry of the sboxes $\mathbf{T}_1, \ldots, \mathbf{T}_3$ is equal to the ith entry of the sbox $\mathbf{T}_0$ shifted by $1, 2$ and 3 bytes to the right respectively (see [9, 3]). Hence, we propose to use only sbox $\mathbf{T}_0$ in the encryption and shift the result as needed to compute the correct AES encryption. E.g., to compute the sbox lookup $\mathbf{T}_1[i]$ using the sbox $\mathbf{T}_0$ we simply cyclically shift the value $\mathbf{T}_0[i]$ by 1 byte to the right.
small-n A simple but effective countermeasure to counteract cache attacks is to split the sbox $\mathbf{S}$ into n smaller sboxes $\mathbf{S}_0, \ldots, \mathbf{S}_{n-1}$ such that every small sbox $\mathbf{S}_i$ fits completely into a single cache line[5]. An application $\mathbf{S}_i[x]$ of sbox $\mathbf{S}_i$ yields d_i bits of the desired result $\mathbf{S}[x]$. Hence, the correct result can be calculated by computing all bits separately and shift them into the correct position.
We construct the small sboxes $\mathbf{S}_i$ for $0 \le i \le n-1$ as follows:

$$\mathbf{S}_i : \{0,1\}^8 \to \{0,1\}^{d_i}$$

mapping

$$x \mapsto \lfloor \mathbf{S}[x] \rfloor_{(\sum_{j=0}^{i-1} d_j, (\sum_{j=0}^{i} d_j)-1)}$$

where $\lfloor y \rfloor_{(b,e)}$ are the bits $y_b \ldots y_e$ of the binary representation of $y = (y_0, \ldots, y_7)$. Instead of applying the sbox $\mathbf{S}$ to x directly each $\mathbf{S}_i$ is applied.
The result is computed as

$$\mathbf{S}[x] = \sum_{i=0}^{n-1} \mathbf{S}_i[x] \cdot 2^{\sum_{j=0}^{i-1} d_j}.$$

[5] Each sbox should fit into a single cache line at every cache level.

In the sequel, we assume that the size of the sbox is a multiple of the size of a cache line and that all d_j are equal. Depending on the number n of required sboxes we call this implementation small-n. E.g., let $|CL| = 512$ and for $0 \leq i \leq 3$ let each $\mathbf{S}_i$ store the bits $\langle \mathbf{S}[x] \rangle_{2i,2i+1}$. The result $\mathbf{S}[x]$ is then computed as

$$\mathbf{S}[x] = \mathbf{S}_0[x] \oplus \mathbf{S}_1[x] \cdot 4 \oplus \mathbf{S}_2[x] \cdot 16 \oplus \mathbf{S}_3[x] \cdot 64.$$

We call this implementation *small*-4. Obviously, the performance depends on the number of involved sboxes and shifts to move bits into the right position. To estimate the efficiency we used the small-n variants in the last round of the fast implementation. Due to the inefficient bit manipulations on 32 bit processors our ad hoc implementation of using small-4 only in the last round shows that the penalty is about 60%. We expect that a more sophisticated implementation reduces this penalty significantly. However, we stress that access driven CBAs are very powerful attacks. Hence, it is not astonishing that secure implementations are not that efficient.

Table 1 in the appendix shows a summary of timing measurements of the implementations described above. The measurements were done on a Pentium M (1400MHz) running linux kernel 2.6.18, gcc 4.1.1.

Next, we consider CBAs based on different sboxes and examine the information leakage and the resistance of each of the implementations described above. The standard implementation uses only a single sbox. Hence, a CBA as described above is based on that sbox. We verified by experiments that measurements taken over ≤ 3 rounds of the standard implementation leak all key bits. Although the small probability p_{miss} prevents performing further experiments we assume that even more rounds will leak all key bits. The resistance for all numbers of rounds is listed in column 1 of Table 2 in the appendix.

The second implementation is the fast implementation. The CBA on the first round of [18] on one of the sboxes $\mathbf{T}_0, \ldots, \mathbf{T}_3$ shows that in this case the fast implementation will reveal half of the key bits, even with an arbitrary number of measurements. The resistance of the fast implementation against such an attack is shown in column 2 of Table 2. The CBA of [8] as described in Section 4.2 based on the sbox $\mathbf{T}_4$ show that in this case the fast implementation leaks all key bits. Since this sbox is only used in the last round the resistance as shown in column 3 of Table 2 does not change for a different number of rounds.

The implementation called fastV1 also leaks all key bits. The resistance against CBAs based on sboxes $\mathbf{T}_0, \ldots, \mathbf{T}_3$ remains the same as listed in column 2 of Table 2. The resistance against CBAs based on the standard sbox is shown in column 4 of Table 2. It remains constant over the number of rounds because the standard sbox is only used in the last round.

Like the fast implementation, the variation called fastV2 also leaks all key bits. It uses only the large sbox $\mathbf{T}_0$ in every round. The resistance for all possible numbers of rounds is listed in column 5 of Table 2.

Last, we consider the variants small-2, small-4 and small-8 that use smaller sboxes than the standard sboxes. Computing $\mathbf{S}[x]$ using variant small-4 or small-8 leaks 0 bits of information having cache lines of size 512 bits because of two reasons:

1. Every $\mathbf{S}_i$ fits completely into a single cache line.
2. For every x each $\mathbf{S}_i$ is used exactly once to compute $\mathbf{S}[x]$.

Hence, the cache information remains constant for all inputs. The only assumption that is involved is that $\mathcal{A}$ cannot distinguish between the accesses on different elements within the same cache line (Assumption 4). The variant small-2 presumably leaks all key bits in our setting. As we have shown above, the variants small-4 and small-8 leak no key bit and hence have resistance 0 (see column 7 and 8 of Table 2). The resistance of small-2 is listed in column 6 of Table 2.

Comparison of implementations As Table 2 shows, the standard implementation provides rather good resistance against CBAs but only has low efficiency. The fast implementation provides the lowest resistance against CBAs but is very efficient. Its variants fastV1 and fastV2 are almost as efficient on 32 bit platforms but provide better resistance against CBAs. The variants using small sboxes provide the best resistance. Especially small-4 and small-8 prevent the leakage of information. For high security applications we propose to use one of the variants using small sboxes and adapt the number of sboxes to the actual size of cache lines of the system.

7 Countermeasure 2: Random Permutation

Another class of countermeasure that was already proposed but not analyzed in [8] is to use secret random permutations to randomize the accesses to the sbox. In this section we present a CBA against an implementation of AES secured by a random permutation that needs roughly 2300 measurements to reveal the complete key. This shows that the increase of the complexity of CBAs induced by random permutations is not as high as one would expect. In particular, the uncertainty of the permutation is not a good measure to estimate the gain of security. A random permutation has uncertainty of $\log_2(256!) \approx 1684$ bits and the uncertainty of the induced partition on the cache lines is $\log_2(256!/(16!)^{16}) \approx 976$ bits.

On the other hand, we present a subset of permutations, so called distinguished permutations, that reduce the information leakage from 8 bits to 4 bits per key byte. Hence, the remaining bits must be determined by an additional attack thereby increasing the complexity. In our standard scenario this is the best one can achieve.

We focus only on the protection of the last round of AES and we assume that the output x of the 9th round is randomized using some secret random permutation π. To be more precise, each byte x_i of the state $x = x_0, \dots, x_{15}$ is substituted by $\pi(x_i)$. To execute the last round of AES a modified sbox $\mathbf{T}'_4$ that depends on π fulfilling $\mathbf{T}'_4[\pi(x_i)] = \mathbf{T}_4[x_i]$ is applied to every byte x_i. This ensures that the resulting ciphertext $c = c_0, \dots, c_{15}$ is correct. We denote the ℓ-th cache line used for the table look-ups for $\mathbf{T}'_4$ by $CL_\ell, \ell = 0, \dots, 15$. Hence, CL_ℓ contains the values $\{\mathbf{S}[\pi^{-1}(x)] | x = 16\ell, \dots, 16\ell + 15\}$. Using a permutation π, information leaking through accessed cache lines does not depend directly on x_i but only on the permuted value $\pi(x_i)$. Since π is unknown to $\mathcal{A}$ the application of π prevents him to deduce information about the secret key $k^{10} = k_0^{10}, \dots, k_{15}^{10}$ directly. However, in the sequel we will show how to bypass random permutations by using CBAs.

7.1 An access driven CBA on a permuted sbox

We assume that we have a fast implementation of AES that is protected by a random permutation π as described above. We also assume that the adversary $\mathcal{A}$ has access to the AES decryption algorithm. This assumption can be avoided. However, the exposition becomes easier if we allow $\mathcal{A}$ access to the decryption. We show how an adversary $\mathcal{A}$ can compute the bytes $k_0^{10}, \dots, k_{15}^{10}$ of the last round key.

Let $\widehat{k}_0$ denote a candidate for byte k_0^{10} of the last round key. In a first step for each possible value $\widehat{k}_0$ the adversary $\mathcal{A}$ determines the assignment $P_{\widehat{k}_0}$ of bytes to cache lines induced by π under the assumption that $\widehat{k}_0 = k_0^{10}$. To be more precise $\mathcal{A}$ computes a function

$$P_{\widehat{k}_0} : \{0,1\}^8 \to \{0, \dots, 15\}$$

such that if $\widehat{k}_0$ is correct then for all x:

$$\pi(x) \in \{16P_{\widehat{k}_0}(x), \dots, 16P_{\widehat{k}_0}(x) + 15\}.$$

I.e., if $\widehat{k}_0$ is correct then $P_{\widehat{k}_0}$ is the correct partition of values $\pi(x)$ into cache lines.

Let us fix some x and a candidate $\widehat{k}_0$ for k_0^{10}. We set $c_0 = \mathbf{S}[x] \oplus \widehat{k}_0$ and $\widehat{M}_0 = \{0, \dots, 15\}$. The adversary repeats the following steps for $j = 1, 2, \dots,$ until $\widehat{M}_0$ contains a single element.

1. $\mathcal{A}$ chooses a ciphertext c^j, whose first byte is c_0, while the remaining bytes of c^j are chosen independently and uniformly at random.
2. Using his access to the decryption algorithm, $\mathcal{A}$ computes the plaintext p^j corresponding to the c^j.
3. By encrypting p^j, the adversary $\mathcal{A}$ determines the set D_0^j of indices of cache lines accessed for the table look-ups for T'_4 during the encryption of p^j.
4. $\mathcal{A}$ sets $\widehat{M}_0 := \widehat{M}_0 \cap D_0^i$.

If $\widehat{M}_0 = \{y\}$, then $\mathcal{A}$ sets $P_{\widehat{k}_0}(x) = y$. Repeating this process for all x yields the function $P_{\widehat{k}_0}$ which has the desired property.

Under the assumption that the guess $\widehat{k}_0$ was correct, the function $P_{\widehat{k}_0}$ is the correct partition of values $\pi(x)$ into cache lines. Moreover, it is not difficult to see that the information provided by $P_{\widehat{k}_0}$ enables the adversary to mount an attack similar to the one described in Section 4.2. This attack can be used to determine for each possible $\widehat{k}_0$ a set of vectors $\widehat{k}_1, \dots, \widehat{k}_{15}$ of hypotheses for the other key bytes. For the time being, we assume that π has the property that for each $\widehat{k}_0$ there remains only a single vector of hypotheses for the other key bytes. Hence, in the end there are only 256 AES keys left and a simple brute force attack reveals the correct one. In general, a random permutation has this property. For a mathematical precise definition and analysis of that property see Section 7.2.

Cost Analysis Experiments show that in the first step of the attack $\mathcal{A}$ needs on average 9 measurements consisting of a pair (p^i, c^i) and the corresponding cache information D_0^i such that the intersection $\widehat{M}_0 := \bigcap D_0^i$ contains only a single element $y = P_{\widehat{k}_0}(x)$. We need to determine the mapping $P_{\widehat{k}_0}(x)$ for every key candidate $\widehat{k}_0$ and every argument $x \in \{0,1\}^8$. Hence, a straightforward implementation of the attack needs roughly $256 \cdot 256 \cdot 9$ measurements to determine the function $P_{\widehat{k}_0}(x)$ for all arguments $x \in \{0,1\}^8$ and all key candidates $\widehat{k}_0 \in \{0,1\}^8$. However, one can reuse measurements for different key candidates $\widehat{k}_0, \widehat{k}'_0$ to reduce the number of measurements to roughly $256 \cdot 9 = 2304$. To determine the vector of hypothesis based on the candidate $\widehat{k}_0$ we can reuse the measurements obtained by determining the function $P_{\widehat{k}_0}$. Hence, the expected number of measurements of this attack is 2304.

7.2 Separability and distinguished permutations

From a security point of view, it is desirable to reduce the information leakage. E.g., a cache attack alone should reveal as few information as possible, in particular it should not reveal the complete key. Then the adversary is forced to either mount a refined and more complex CBA based on other intermediate results or combine the cache attack with some other method to determine the key bytes uniquely. In this case, the situation is similar to the attack of [18], where a cache attack on the first round only reveals 4 bits of each key byte. Hence Osvik et al. combine cache attacks on the first and second round of AES.

First, we present the property a permutation applied to the result of the 9-th round should have such that $\mathcal{A}$ cannot determine the key bytes uniquely using only a cache attack on the last round. We denote the ℓth cache line by CL_ℓ and the elements of CL_ℓ by $a_0^{(\ell)}, \ldots, a_{15}^{(\ell)}$. Hence, the underlying permutation used to define this cache line is given by

$$\pi^{-1}(16\ell + j) = \mathbf{S}^{-1}[a_j^{(\ell)}]. \tag{3}$$

We say that a key candidate $\widehat{k}_0$ is *separable* from the first key byte k_0 of the last round if there exists a measurement that proves $\widehat{k}_0$ to be wrong. Conversely, a key candidate $\widehat{k}_0$ is *inseparable* from the key k_0 if there does not exist a measurement that proves $\widehat{k}_0$ to be wrong. More precisely, writing $\widehat{k}_0 = k_0 \oplus \delta$ the bytes $\widehat{k}_0$ and k_0 are inseparable if and only if

$$\forall \ell \in \{0, \ldots, 15\} \forall a \in CL_\ell : a \oplus \delta \in CL_\ell. \tag{4}$$

Notice that this property only depends on the difference δ and not on the value of k_0. Since there are 16 elements of the sbox in every cache line property (4) can only be satisfied by at most 16 differences. It turns out that for $|\Delta| = 16$ the set

$$\Delta := \{\delta \mid \text{for all } k_0 \in \{0,1\}^8 \text{ the bytes } k_0 \text{ and } k_0 \oplus \delta \text{ are inseparable}\}$$

forms a 4 dimensional subspace of $\mathbb{F}_{2^8}$ viewed as a 8 dimensional vector space over $\mathbb{F}_2$. It is obvious that the neutral element 0 is an element of Δ and that every $\delta \in \Delta$ is its own inverse. It remains to show that Δ is closed with respect to addition. Consider $\delta, \delta' \in \Delta$ and an arbitrary $a \in CL_\ell$. Then $a' = a \oplus \delta \in CL_\ell$ implies that $a' \oplus \delta' = a \oplus \delta \oplus \delta' \in CL_\ell$ because of (4) and $\delta \oplus \delta' \in \Delta$ holds.

Hence, any partition that has the maximal number of inseparable key candidates must generate a subspace of dimension 4. Using this observation we describe how to efficiently construct permutations such that the set Δ of inseparable differences has size 16. In the sequel, we will call any such permutation a *distinguished permutation.*

Construction of the subspace We first construct a set Δ of 16 differences that is closed with respect to addition over $\mathbb{F}_{256}$. We can do this in the following way

1. set $\Delta := \{\delta_0 := 0\}$, choose δ_1 uniformly at random from the set $\{1, \ldots, 255\}$, set $\Delta := \Delta \cup \{\delta_1\}$
2. choose δ_2 uniformly at random from $\{1, \ldots, 255\} \setminus \Delta$, set $\Delta := \Delta \cup \{\delta_2, \delta_3 := \delta_1 \oplus \delta_2\}$
3. choose δ_4 uniformly at random from $\{1, \ldots, 255\} \setminus \Delta$, set $\Delta := \Delta \cup \{\delta_4, \delta_5 := \delta_4 \oplus \delta_1, \delta_6 := \delta_4 \oplus \delta_2, \delta_7 := \delta_4 \oplus \delta_3\}$
4. choose δ_8 uniformly at random from $\{1, \ldots, 255\} \setminus \Delta$, set $\Delta := \Delta \cup \{\delta_8, \delta_9 := \delta_8 \oplus \delta_1, \delta_{10} := \delta_8 \oplus \delta_2, \delta_{11} := \delta_8 \oplus \delta_3, \delta_{12} := \delta_8 \oplus \delta_4, \delta_{13} := \delta_8 \oplus \delta_5, \delta_{14} := \delta_8 \oplus \delta_6, \delta_{15} := \delta_8 \oplus \delta_7\}$

This construction ensures that Δ is closed with respect to addition and hence Δ forms a subspace as desired.

Construction of the permutation Now we can compute the function P that maps $\mathbf{S}[x] \in \mathbb{F}_2^8$ to a cache line. We use the fact that 16 proper translations of a 4 dimensional subspace form a partition of a 8 dimensional vector space $\mathbb{F}_2^8$. A basis $\{b_0, \dots b_3\}$ of the subspace Δ can be expanded by 4 vectors $b_4, \dots b_7$ to a basis of $\mathbb{F}_2^8$. The 16 translations of Δ generated by linear combinations of $b_4, \dots, b_7$ form the quotient space $\mathbb{F}_2^8/\Delta$ that is a partition of $\mathbb{F}_2^8$. To construct the function P we do the following:

1. for every cache line CL_ℓ do
2. choose $a^{(\ell)}$ uniformly at random from $\mathbb{F}_{256}/\{a^{(j)} \oplus \delta \mid j < \ell, \delta \in \Delta\}$
3. fill CL_ℓ with the values of the set $\{a^{(\ell)} \oplus \delta \mid \delta \in \Delta\}$

Using (3) this partition into cache lines defines the corresponding permutation.

Analysis of the countermeasure The security using a distinguished permutation as defined above rests on two facts.

1. Using a distinguished permutation where the set Δ of inseparable differences has size 16, a cache attack on the last round of AES will reveal only four bits of each key byte k_i^{10}. Overall 64 of the 128 bits of the last round key remain unknown. Therefore, the adversary has to combine his cache attack on the last round with some other method to determine the remaining 64 unknown bits. For example, he could try a modified cache attack on the 9-th round exploiting his partial knowledge of the last round key. Or he could use a brute force search to determine the last round key completely.
2. There are several distinguished permutations and each of these permutations leads to 16! different functions mapping elements to 16 lines. If we choose randomly one of these functions, before an adversary can mount a cache attack on the last round as described in Section 4.2, he first has to use some method like the one described in Section 7.1 to determine the function P that is actually used.

We stress that we consider the first fact to be the more important security feature. We saw already in Section 7.1 that determining a random permutation used for mapping elements to cache lines is not as secure as one might expect. Since we are using permutations of a special form the attack described in Section 7.1 can be improved somewhat. In the remainder of this section we briefly describe this improvement. To do so, first we have to determine the number of subspaces leading to distinguished permutations.

As before view $\mathbb{F}_2^n := \{0,1\}^n$ as an n-dimensional $\mathbb{F}_2$ vector space. For $0 \leq k \leq n$ we define $D_{n,k}$ to be the number of k-dimensional subspaces of $\mathbb{F}_2^n$. To determine $D_{n,k}$ for V an arbitrary m-dimensional subspace of $\mathbb{F}_2^n$ we define

$$N_{m,k} := |\{(v_1, \dots, v_k) | v_i \in V, v_1, \dots v_k \text{ are linearly independent}\}|.$$

The number $N_{m,k}$ is independent of the particular m-dimensional subspace V, it only depends on the two parameters m and k. Then

$$D_{n,k} = \frac{N_{n,k}}{N_{k,k}}.$$

Next we observe that

$$N_{m,k} = \prod_{j=0}^{k-1} (2^m - 2^j) = 2^{k(k-1)/2} \prod_{j=0}^{k-1} (2^{m-j} - 1).$$

Hence, we obtain that

$$D_{n,k} = \frac{\prod_{j=0}^{k-1} (2^{n-j} - 1)}{\prod_{j=0}^{k-1} (2^{k-j} - 1)}.$$

In our special case we have $n = 8$ and $k = 4$ and hence the number of 4 dimensional subspaces is

$$D_{8,4} = \frac{255 \cdot 127 \cdot 63 \cdot 31}{15 \cdot 7 \cdot 3 \cdot 1} = 200787.$$

As mentioned above, each subspace leads to 16! different distinguished permutations. Hence, overall we have $200787 \cdot 16! \approx 2^{60}$ distinguished permutations. On the other hand, because of the special structure of our permutations, to determine the function P by cache attacks can be done more efficiently than determining an arbitrary function mapping elements to cache lines (see Section 7.1). In particular, $\mathcal{A}$ only needs to observe about 7 accesses of a single but arbitrary cache line. With high probability this will be enough to determine a basis of the subspace being used. In addition, $\mathcal{A}$ needs at least one access for every other cache line in order to determine the function P. The corresponding probability experiment follows the multinomial distribution. We did not calculate the expected number of tries exactly. Experiments show that if we can determine the accessed cache line exactly, on average 62 measurements suffice to compute the function P exactly. However, a single measurement only yields a set of accessed cache lines. But arguments similar to the ones used for the first part of the attack in Section 7.1 show that we need on average 9 measurements to uniquely determine an accessed cache line. Therefore, on average we need $62 \cdot 9 = 558$ experiments to determine the function P.

Hence, compared to the results of Section 7.1 we have reduced the number of measurements used to determine the function P by a factor of 3. However, we want to stress again, that the main security enhancement of using distinguished permutations instead of arbitrary permutations is the fact, that distinguished permutations have a lower information leakage. To improve the security, one can choose larger key sizes such as 192 bits or 256 bits. Since distinguished permutations protect half of the key bits, the remaining uncertainty about the secret key after cache attacks can be provably increased from 64 bits to 96 bits or 128 bits, respectively.

Separability and random permutations In our CBA on an implementation protected by a random permutation (Section 7.1) we assumed that fixing a candidate $\widehat{k}_0$ determines the candidates for all other key bytes. With sufficiently many measurements for a fixed $\widehat{k}_0$ we can determine the function $P_{\widehat{k}_0}$ as defined in Section 7.1. Furthermore, we saw that the separability of candidates $\widehat{k}, \widehat{k}'$ depends only on their difference $\delta = \widehat{k} \oplus \widehat{k}'$. Hence, to be able to rule out all but one candidate $\widehat{k}_i$ at position i for a fixed $\widehat{k}_0$ the permutation π must have the following property:

$$\forall \delta \neq 0 \exists j \in \{0, \ldots, 15\} \exists a \in CL_j : a \oplus \delta \notin CL_j .$$

There are less than 2^{129} of the $256! \approx 2^{1684}$ permutations that do not have this property. Hence, a random permutation satisfies this condition with probability $1 - \frac{2^{129}}{2^{1684}}$.

8 Summary of countermeasures and open problems

In this paper we presented and analyzed the security of several different implementations of AES. Moreover, we analyzed countermeasures based on permutations: random permutations and distinguished permutations. We give a short overview over the advantages and disadvantages of the countermeasures:

countermeasure	# measurements	information in bits /security	efficiency
small-4	∞	0 / high	slow
random permutation	2300	128 / low	fast
distinguished permutations	560	64 / medium	fast

The second column shows the expected number of measurements an attacker has to perform in order to get the amount of information shown in the third column.

Small-4 (see Section 6) prevents information leakage in a cache attack. However, the efficiency depends on the size of a cache line and is rather low. In contrast, random permutations (see Section 7) provide only low security. About 2300 measurements are enough to reveal the complete 128 bit AES key. If realized via table lookups, random permutations are fast. But to increase the security offered by random permutations they have to be changed frequently. Changing a permutation may cause problems with respect to efficiency and security. So far, we have no precise analysis of these issues.

Distinguished permutations (see Section 7.2) protect half of the key bits and hence provide a medium level of security. Using distinguished permutations, no frequent changes of permutations are required to achieve a medium level of security. Hence, they do not suffer from the above mentioned problems of random permutations. Therefore, distinguished permutations provide a better ratio of efficiency and security as random permutations but still leak half of the key bits.

Random permutations and distinguished permutations have to be realized as tables for efficiency reasons. Hence, a straightforward implementation of the applications of a permutation would render the whole implementation susceptible to cache attacks. A possible solution to this problem is to realize permutations via small sboxes that completely fit into a cache line. Following the description of the small variant of Section 6, π is split into smaller tables $\pi_0, \ldots, \pi_3$ each of which is applied to the input x. Obviously, this does not make sense if the standard sbox $\mathbf{S}$ is used because both π and $\mathbf{S}$ map from $\{0,1\}^8$ to $\{0,1\}^8$. Hence, it takes as many table lookups to apply π realized with small sboxes as it takes to apply $\mathbf{S}$ realized with small sbox directly. Moreover, realizing $\mathbf{S}$ via small tables has the advantage of not leaking information via the cache behavior.

The situation is different if the large sboxes of the fast implementation are used. Again π maps from $\{0,1\}^8$ to $\{0,1\}^8$ but a large sbox maps from $\{0,1\}^8$ to $\{0,1\}^{32}$. Therefore, it takes 4 times as many table lookups to realize the large sbox via small sboxes than to realize π via small tables. Hence, first applying π to an input via small tables and then applying a large permuted sbox, as shown in Figure 2, makes sense if this technique is faster than realizing the standard sbox $\mathbf{S}$ via small sboxes. Here, one has to take into account the technical problem that on 32-bit platforms the byte oriented structure of the standard sbox $\mathbf{S}$ leads to a time consuming post processing to incorporate the output of the sbox into the encryption state.

Note that realizing π via small tables does not leak any information in cache attacks. Only the application of the permuted sbox leaks information about intermediate states. Hence, this scenario is exactly the scenario of our attack in Section 7.1 where we assumed that only the application of the sbox leaks information.

As mentioned in Section 6 one can scale the sizes of the smaller tables to improve efficiency. But it is essential to determine whether the amount of information that leaks with this method is acceptable or not. Summing up, the analysis given above shows that permutations as a countermeasure to thwart cache based attacks do not provide as much security as one would expect. However, we have shown that using distinguished permutations one can reduce the information leakage via CBAs. That means that even with an arbitrary number of measurements a CBA based on the last round cannot determine certain bits of the secret key. Since we consider the reduction of information leakage as a preferred goal distinguished permutations constitute an interesting way to improve the security gain of permutations.

References

1. Onur Aciiçmez and Çetin Kaya Koç. Trace-driven cache attacks on AES (short paper). In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS*, volume 4307 of *Lecture Notes in Computer Science*, pages 112–121. Springer, 2006.
2. Onur Aciiçmez, Werner Schindler, and Çetin Kaya Koç. Cache based remote timing attack on the AES. In Masayuki Abe, editor, *CT-RSA*, volume 4377 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2007.
3. Paulo Barreto. The AES block cipher in C++, 2003. http://planeta.terra.com.br/informatica/paulobarreto/EAX++.zip.
4. D. J. Bernstein. Cache-timing attacks on AES, 2005. http://cr.yp.to/papers.html#cachetiming, Document ID: cd9faae9bd5308c440df50fc26a517b4.
5. Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES power attack based on induced cache miss and countermeasure. In *ITCC (1)*, pages 586–591. IEEE Computer Society, 2005.
6. Johannes Blömer, Jorge Guajardo, and Volker Krummel. Provably secure masking of AES. In Helena Handschuh and M. Anwar Hasan, editors, *Selected Areas in Cryptography*, volume 3357 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2004.
7. Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In Louis Goubin and Mitsuru Matsui, editors, *CHES*, volume 4249 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2006.
8. Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. Cryptology ePrint Archive, Report 2006/052, 2006. http://eprint.iacr.org/.
9. Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer Verlag, 2002.
10. John Hennesey and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd edition, 2002.
11. Wei-Ming Hu. Lattice scheduling and covert channels. In *IEEE Symposium on Security and Privacy*, pages 52–61. IEEE Press, 1992.
12. Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.
13. Francois Koeune and Jean-Jacques Quisquater. A timing attack against Rijndael. Technical Report CG-1999/1, Universit Catholique de Louvain, 1999.
14. Cédric Lauradoux. Collision attacks on processors with cache and countermeasures. In Christopher Wolf, Stefan Lucks, and Po-Wah Yau, editors, *WEWoRC*, volume 74 of *LNI*, pages 76–85. GI, 2005.
15. National Institute of Standards and Technology. *Advanced Encryption Standard (AES) (FIPS PUB 197)*, 2001.
16. Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *Proceedings of Selected Areas in Cryptography 2006*, 2006.
17. Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. A refined look at Bernstein's AES side-channel analysis. In Ferng-Ching Lin, Der-Tsai Lee, Bao-Shuh Lin, Shiuhpyng Shieh, and Sushil Jajodia, editors, *ASIACCS*, page 369. ACM, 2006.
18. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *CT-RSA*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.
19. D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive, Report 2002/169, 2002. http://eprint.iacr.org/.

20. D. Page. Partitioned cache architecture as a side-channel defence mechanism. Cryptology ePrint Archive, Report 2005/280, 2005. http://eprint.iacr.org/.
21. Colin Percival. Cache missing for fun and profit. In *BSDCan '05*, 2005.
22. William Stallings. *Operating Systems: Internals and Design Principles.* Prentice Hall, 5th edition, 2005.
23. Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2779 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2003.

A Appendix

# sboxes	fast	standard	fastV1	fastV2	small-2	small-4	small-8
time factor	1	~ 3	~ 1	~ 1	1.32	1.6	1.95

Table 1. Timings for different implementations of AES

	1	2	3	4	5	6	7	8
	standard	fast	fast T_4	fastV1	fastV2	small-2	small-4	small-8
	$\mathbf{S}$	$\mathbf{T}_0,\ldots,\mathbf{T}_3$	$\mathbf{T}_4$	$\mathbf{S}$	$\mathbf{T}_0$	$\mathbf{S}_0,\mathbf{S}_1$	$\mathbf{S}_0,\ldots,\mathbf{S}_3$	$\mathbf{S}_0,\ldots,\mathbf{S}_7$
E_1	2.57	198.0	91.2	2.57	91.2	$3.91\cdot 10^{-3}$	0	0
E_2	$2.57\cdot 10^{-2}$	153.0	91.2	2.57	32.5	$5.96\cdot 10^{-8}$	0	0
E_3	$2.58\cdot 10^{-4}$	118.0	91.2	2.57	11.6	$9.09\cdot 10^{-13}$	0	0
E_4	$2.58\cdot 10^{-6}$	91.2	91.2	2.57	4.12	$1.39\cdot 10^{-17}$	0	0
E_5	$2.59\cdot 10^{-8}$	70.4	91.2	2.57	1.47	$2.12\cdot 10^{-22}$	0	0
E_6	$2.59\cdot 10^{-10}$	54.4	91.2	2.57	$5.22\cdot 10^{-1}$	$3.23\cdot 10^{-27}$	0	0
E_7	$2.60\cdot 10^{-12}$	42.0	91.2	2.57	$1.86\cdot 10^{-1}$	$4.93\cdot 10^{-32}$	0	0
E_8	$2.61\cdot 10^{-14}$	32.5	91.2	2.57	$6.62\cdot 10^{-2}$	$7.52\cdot 10^{-37}$	0	0
E_9	$2.61\cdot 10^{-16}$	25.1	91.2	2.57	$2.36\cdot 10^{-2}$	$1.15\cdot 10^{-41}$	0	0
E_{10}	$2.62\cdot 10^{-18}$	25.1	91.2	2.57	$8.39\cdot 10^{-3}$	$1.75\cdot 10^{-46}$	0	0

Table 2. The resistance E_r of AES implementations as defined in (2)

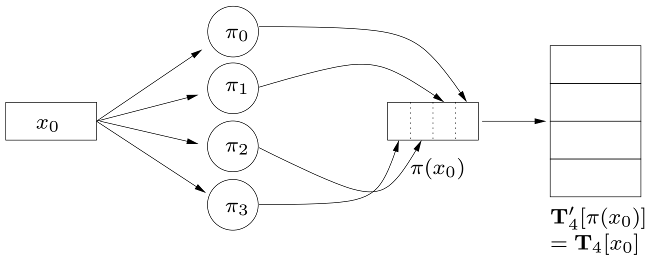

Fig. 2. Combining small tables with permutation π