Advances on Access-Driven Cache Attacks on AES*

Michael Neve[1] and Jean-Pierre Seifert[2,3]

[1] Intel Corporation, CTG STL Trusted Platform Laboratory,
2111 NE 25th Avenue, Hillsboro Oregon 97124, USA
michael.neve.de.mevergnies@intel.com
[2] Applied Security Research Group
The Center for Computational Mathematics and Scientific Computation
Faculty of Science and Science Education
University of Haifa
Haifa 31905, Israel
[3] Institute for Computer Science
University of Innsbruck
6020 Innsbruck, Austria
jeanpierreseifert@yahoo.com

Abstract. An access-driven attack is a class of cache-based side channel analysis. Like the time-driven attack, the cache's timings are under inspection as a source of information leakage. Access-driven attacks scrutinize the cache behavior with a finer granularity, rather than evaluating the overall execution time. Access-driven attacks leverage the ability to detect whether a cache line has been evicted, or not, as the primary mechanism for mounting an attack. In this paper we focus on the case of AES and we show that the vast majority of processors suffer from this cache-based vulnerability. Our best results are indeed performed on a processor without the multi-threading capabilities — in contrast to previous works in this area that had suggested that multi-threading actually improved, or even made possible, this class of attack.

Despite some technical difficulties required to mount such attacks, our work shows that access-driven cache-based attacks are becoming easier to understand and analyze. Also, when such attacks are mounted against systems performing AES, only a very limited number of encryptions are required to recover the whole key with a high probability of success, due to our last round analysis from the ciphertext.

1 Introduction

Side channels have been studied for many years in the context of smart cards and embedded systems. Recently some researches demonstrated that microprocessors are also vulnerable to side channels [2, 13, 14, 17], by showing that the

* This work has first been presented during the rump session of Crypto 05 by E. Brickell.

E. Biham and A.M. Youssef (Eds.): SAC 2006, LNCS 4356, pp. 147–162, 2007.
© Springer-Verlag Berlin Heidelberg 2007

cache mechanism induces a variability in the execution time due to the different memory accesses. This represents a threat for cryptographic software, since the cache accesses are dependent on the inputs of the software, namely the plaintext and the key. Hence, the analysis of the execution time provides information about the key.

The cache has been mentioned earlier as a potential vulnerability regarding covert channels [6,9,19] and side channels [7,8]. Tsunoo *et al.* demonstrated in [17] the first practical results on DES. They also mentioned results on AES but did not provide further details. In [2], Bernstein showed results of side channel analysis against AES, based on the first round. [10] and [11,12,13] independently provided an analysis of the second round of AES. These attacks belong to the class of *time-driven* cache-based attacks as they analyze the overall execution time.

Moreover [11,12,13] detailed also techniques to perform *access-driven* attacks on AES, where a process *spies* on another one using the cache accesses. [14] used a similar technique against RSA implementations. [3] recently provided software mitigation strategies for AES that reduce the cache leakage.

In this paper, we detail a new and very efficient access-driven cache-based side channel attack. We focus on 128-bit AES implementations that uses four 1KB precomputed SBox tables (such as OpenSSL [1]) and we show that an analysis of the ciphertexts can lead to the recovery of the entire secret key.

Previous attacks [14,12,11,13] exploited the hardware-assisted multi-threading capability of some microprocessors, cf. [15], in order to run a spy process quasi parallel to a crypto process. However, today most processors are single-threaded, therefore, this paper investigates and demonstrates that one can successfully perform such attacks also on this common class of processors.

In practical cases, many processes are quasi-parallel executed at the same time as our crypto and spy processes. Those other processes generate noise in the measurements of the spy process. In this paper however, we are interested in deriving expected numbers of measurements necessary to disclose the full key with perfect measurements, *i.e.* without noise. Nevertheless, we will discuss the consequence of noise on our strategies. Moreover, we will also elaborate on the number of snapshots that the spy process can take per encryption. In addition we will discuss the impact of the measurement resolution upon the quality of the attack.

The present paper is organized as follows. The next Section briefly recalls some facts about AES and cache-based side-chanel attacks. In Section 3, we detail how access-driven attacks can be mounted on single-threaded processors and we demonstrate our practical success by showing a snapshot of a cache activity on such a processor. In common implementations of AES the last round uses a separate SBox table from the other rounds. We show in Section 3 how this information can be combined with the ciphertexts in order to deduce the key. In Section 4 we compute the expected number of cache lines accessed during the last round. We discuss the different cases of attack resolution in Section 5, and two different analysis methods are described in Section 6. Finally we provide our conclusions in Section 7.

2 Definitions and Preliminaries

AES. AES is a popular and commonly used block cipher. We only recall here the particular features of AES that we use in this paper. Refer to [4] for full details. AES operates in a succession of identical rounds, where four operations are performed on the state (*i.e.* the temporary value): an SBox permutation `SubBytes`, a byte transposition `ShiftRows`, a collumn by column permutation `MixColumn` and a sub key addition `AddRoundKey`. The last round however is slightly different since the `MixColumn` operation is skipped.

The key schedule `ExpandedKey` derives the sub keys $K^{(i)}$ from the secret key k. The non-linearity is given by the mean of `SubBytes`. `ExpandedKey` is invertible and, in the case of 128-bit AES, it is possible to derive the secret key from any single sub key. We will use this property in our attack.

Efficient software implementations take advantage of precomputed SBox tables to reach high performances. In OpenSSL for example there are five 1KB tables (T_0, T_1, T_2, T_3, T_4) necessary for the encryption part. All rounds but the last one use 4 of them (T_0 to T_3) whereas the last round and the key schedule use the special fifth one (T_4).

Cache-Based Side Channel Attacks. Access-driven side channels consider that two (or more) processes are executed quasi-parallel on the processor. One process (called here the crypto process) is performing a cryptographic function (*i.e.* AES in this case) involving a secret key. As aforementioned, precomputed values are involved in the execution of the crypto process and their accesses are done through the memory hierarchy. On each data request, the cache checks whether it holds the data, or not. If it does, a cache-hit occurs and the data is immediately transmitted to the processor. Otherwise, a cache-miss occurs and the data must be fetched from a higher memory level, with a longer access time.

A second process, called a spy process, *spies* on the cache accesses of the crypto process. It continuously loads a table S of the size of the cache. From time to time, the crypto process is executed and it inevitably evicts some parts of S by accessing particular data. Therefore, the next time that the spy process is executed, the access time of each part of S (*i.e.* the time necessary to reload a given part of S) indicates which part has been evicted by the crypto process during the last execution of the crypto process.

Thus, the cache is leaking information about the crypto process's memory accesses. Since the software implementation is known, an attacker can infer partial knowledge of the secret key. It is however worth underlining the fact that the spy process cannot diretly access the data of the crypto process; it only observes the cache activity generated by the crypto process and deduces (partial) information from this activity.

3 Exploiting OS Scheduling Instead of Simultaneous Multithreading

Recall that previously described attacks [12, 11, 13, 14] take advantage of the multi-threading capacity of certain processors. It allows them to have two

processes running *quasi* parallel on the same processor, as if there were two logical processors [15,16]. In this manner some logical elements are shared, while the quasi parallelism enables one process to *spy* on the other through the use of the shared logic elements. The cache architecture is one such example of a shared element. Although hardware-assisted multi-threading seems to be mandatory at first sight, we show in the rest of this section that it is not.

Although single-threaded processors run threads/processes serially, the OS manages to execute several programs also in a quasi parallel way, only at a coarser resolution, cf. [16]. The OS basically decomposes an application into a series of short threads that are ordered with other application threads. The processor's resources are thus temporally shared according to the OS's ascribed prioritization.

In order to transfer the (hardware-assisted) multi-threaded processor attacks from [12, 11, 13, 14] to single-threaded processors, one has to leave the comfort of hardware-assistance and exploit subtle OS particularities — which may vary from OS to OS. While this seems quite possible for attacks such as [14], the very fast execution time of AES seems to require the aforementioned hardware-assistance in order to efficiently switch between the spy and the crypto process. Indeed, the objective is to ensure is that the crypto thread runs only for a small amount of time between any two runs of the spy thread, or in other words we are able to implement the following strategy:

spy: Continuously watches the cache usage of the parallel crypto thread.
crypto: Runs only for a small amount of time between any two runs of spy.

Interestingly enough, the basic idea is already pointed out in one of the fundamental papers on cache-based side channel attacks, cf. Hu [6], and can adapted to today's OS to stretch the AES execution time over several OS quantums, cf. [6, 16]. According to cf. [6], the so called *preemptive scheduling* property, cf. [16], "...allows a process to control when it yields the CPU to another process without waiting until the end of the quantum." Therefore, using the Linux command `sleep` instead of the VAX security kernel command `WAIT` and the following repetitive spy process paradigm we are able to achieve an implementation of the above attack strategy:

– Watch the cache usage.
– Spend most of the OS quantum.
– Yield the CPU to another process via an appropriate sleep near to the quantum end.

This paradigm uses the fact that the OS will reschedule the (very short) remaining quantum part to the crypto thread which will be able to execute a few instructions, after which the OS will quickly reschedule to the spy thread, allowing him to spy on the recently used memory accesses. As the above paradigm and all its subtle implementation details heavily depend on the underlying OS, CPU type and frequency, etc. we will not deepen further this technical details here.

Figure 1 shows the successful implementation of the above strategy and was actually created by observing an unmodified AES implementation through the cache accesses. In this Figure, there are 80 columns. Each column represents a single cache line. The 80 columns are divided into five tables of 16 cache lines, each table representing an SBox (starting with T_0 at the left and continuing to T_4 at the right). Each row in this figure indicates a different measuring time (the uppermost being the first measurement). Each point in a row displays the activity of the particular cache line that it represents. The brighter the point, the longer the time it takes to access an element in the cache line. It is important to understand that we get no information about the order of the accesses within one measurement. By using this kind of picture, an attacker can follow the activity of one or more specific cache lines. We obtained similar patterns on another OS and we believe that any multitasking OS could lead to the same access results.

Figure 1 depicts 4 successive AES encryptions. In this particular example, each encryption is repeated 5 times. The time resolution enables us to perform a few measurements per encryption. However, we do not have any distinction between the AES rounds. We only know that they are interrupted several times by the spy program at some points during the encryption.

SBox T_4 [1] plays a particular role here, as it is invoked only on the last round. Therefore, the SBox T_4 accesses indicate the end of an encryption, and all lines within the SBox T_4 accesses are then linked to a single encryption.

4 Analysis of the Last Round

Previously cited attacks use the information about the first or the first and second rounds of one encryption[2]. However, we focus here on the accesses of the last round. Indeed, if the time resolution of the spy process enables us to see the accesses of one encryption, SBox T_4 will also appear clearly.

The ciphertext is now under investigation in order to take advantage of the last round accesses. Recall from our introduction, the last round of AES is particularly of interest in the sense that the `MixColumns` operation is never applied. And for that particular reason, OpenSSL uses SBox T_4, especially for the last round. With $c := E_{AES}(p, k)$ being the ciphertext, we have the following relations linking c and the last round:

$$c = K^{(10)} \oplus \texttt{ShiftRows}[\texttt{SubBytes}[x^{(9)}]],$$

where $x^{(9)}$ is the initial state of round 10 (*i.e.* the output of round 9 and input to round 10). Since round 10 uses SBox T_4, we denote the actual access to T_4 by [SBox T_4 outputs]. The relation becomes:

$$c = K^{(10)} \oplus [\text{SBox } T_4 \text{ outputs}].$$

[1] Recall that T_4 has a size of 1KB and therefore it is represented by the last set of 16 points in each row, with 64-byte cache lines.

[2] But the accesses contain the cache activity of all rounds and the analysis of the first round(s) is disturbed by the access of the other rounds.

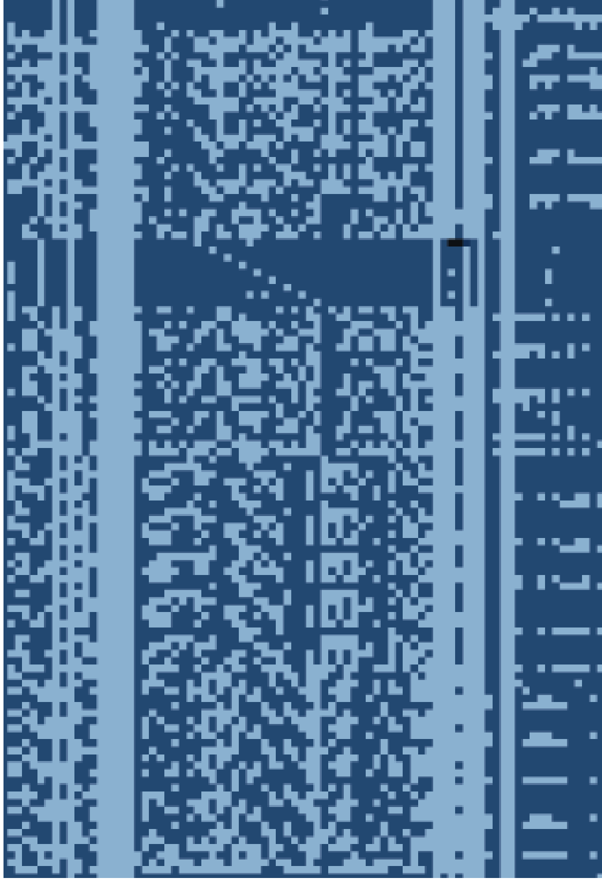

Fig. 1. Evolution of the cache *versus* time, displaying several AES encryptions. Each horizontal line represents the state of the cache lines (represented by a point) at a given time. The brighter, the longer the time to access its corresponding cache line.

Therefore, we derive a relation defining $K^{(10)}$:

$$K^{(10)} = c \oplus [\text{SBox } T_4 \text{ outputs}],$$

from which it is easy to deduce the value of k from $K^{(10)}$ — see our brief recall of AES in section 2.

However, [SBox T_4 outputs] represents the result of *all* the accesses of the last round, *i.e.* for all bytes of the T_4 input $x^{(9)}$. Moreover, the cache accesses only point out the accessed cache lines, but not the individual elements in those lines. Although the next section will give more details on the last points, we refer the reader to Handy [5] for a thorough review of cache architectures.

Table 1. Expected number of cache lines of SBox T_4 accessed in t last rounds for $t > 1$ and $m = 16$

t	2	3	4	5	6	7	> 7
$\mathbb{E}(P(t \cdot 16))$	13.97	15.28	15.74	15.91	15.97	15.99	≈ 16

5 Average Number of Accesses for the Last Round

Let us first introduce some notations. Let $\delta = 2^o$ be the cache line size (in byte) and $m = 2^l$ be the number of cache lines of SBox T_4. Let also $p(b)$ be the probability that one specific cache line is accessed in b T_4 accesses, and $P(b)$ its corresponding random variable. Likewise, $p_n(b)$ the probability that one specific cache line is *not* accessed during b T_4 accesses, and $P_n(b)$ its corresponding random variable. Also, let us assume that the accesses to T_4 are independent and uniformly distributed. We now want to compute the expected number of different cache accesses into T_4. Using that $p(1) = 1/m$, or $p_n(1) = 1 - 1/m$ and the last assumption yields

$$p_n(16) = \left(1 - \frac{1}{m}\right)^{16}.$$

Therefore, the expected number of cache lines *not* accessed in a last round is given by

$$\mathbb{E}(P_n(16)) = 16 \cdot \left(1 - \frac{1}{m}\right)^{16}.$$

In the case of caches with 64 bytes per cache line (*i.e.* $\delta = m = 16$), we get $\mathbb{E}(P_n(16)) = 5.70$ and thus $\mathbb{E}(P(16)) = 10.30$ as the expected number of cache lines accessed during a last round.

6 Resolution

On Figure 1, the T_4 accesses are all visible within a few vertical lines. Let the resolution factor t be defined as

$$t := \frac{\#\text{ of ciphertexts}}{\#\text{ of measurements}},$$

which yields the following different resolution cases:

- *low resolution*: One measurement covers t encryptions (with $t > 1$) and therefore several last round accesses are overlaid. Then $\mathbb{E}(P(t \cdot 16)) = 16 \cdot (1 - 1/m)^{t \cdot 16}$. Table 1 shows that $\mathbb{E}(P(t \cdot 16))$ rapidly gets close to its limits.
- *one line resolution*: The frequency of measurements isolates one last round per measurement, *i.e.* $t = 1$. We already computed this case. Then $\mathbb{E}(P(16))$ equals 10.30 for $m = 16$.

– *high resolution*: There are several measurements $(1/t)$ occurring during the last round, *i.e.* $t < 1$. The observation of the evolution of the accesses gives a notion of the order in which the accesses have taken place and therefore narrows down the possible accesses per byte.

For now, we consider *one line resolution* to detail the analysis of the accesses. We return to this in Section 8 and discuss the impact of the resolution in the analysis's results.

7 Non-elimination and Elimination Methods

We detail here how to deduce the secret key from cache accesses of SBox T_4 and the ciphertexts.

The first method is directly inferred from the relation obtained above:

$$\mathbf{K}^{(10)} = \mathbf{c} \oplus [\text{SBox } T_4 \text{ outputs}].$$

This states that $\mathbf{K}^{(10)}$ is computed with the ciphertext $\mathbf{c}$ and *some* SBox outputs resulting from the SBox T_4 accesses. Each access to a particular line outputs one out of 16 values and we try to discover which one it is, from many ciphertext/accesses pairs. This finally leads to the value of $\mathbf{K}^{(10)}$, when applied in a byte-wise fashion.

The second method is based on the inverse relation:

$$\mathbf{K}^{(10)} \neq \mathbf{c} \oplus \neg[\text{SBox } T_4 \text{ outputs}],$$

where $\neg$[SBox T_4 outputs] refers to the non-accessed cache lines. The relation simply means that the bytes obtained by the addition of $\mathbf{c}$ and the non-accessed cache lines can be discarded as candidates for $\mathbf{K}^{(10)}$. This method, as we are about to see, requires less ciphertext/accesses pairs than the first one.

Let us call those methods respectively *Non-elimination* and *Elimination* methods, since they share the same philosophy as Tsunoo's methods [18]. Let us further suppose that we have a large number of clear measurements of the cache accesses over the last round and the corresponding ciphertexts. We will now detail each method individually.

7.1 Non-elimination Method

This method is separated into three steps. All three steps must be applied for all of the 16 bytes of the key. Suppose we attack byte i, $0 \leq i \leq 15$.

1. *Selection of the ciphertext*: The ciphertext/accesses pairs are sorted according to the value of byte i of the ciphertext. Since the key is constant, it is clear that if the ith byte of different ciphertexts have the same value, all the accesses corresponding to those ciphertexts must contain an access to one common cache line[3].

[3] Since the ciphertexts can be considered random, the other bytes will have random accesses to T_4. We seek the constant access among the random ones.

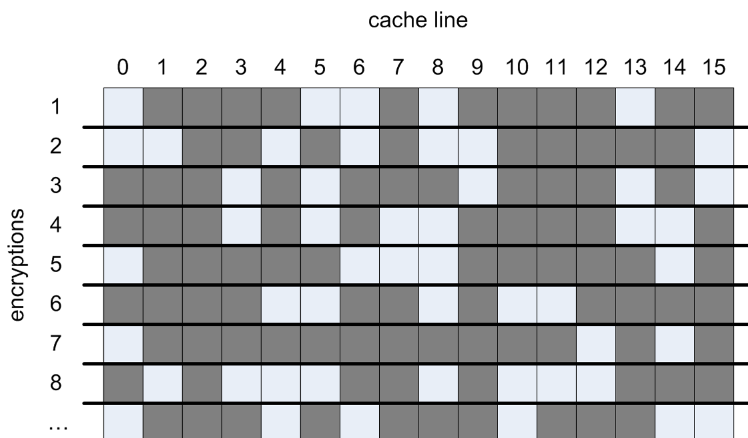

Fig. 2. Cache line accesses for ciphertexts with a constant value for byte i. The dark boxes represent accessed cache lines.

Consider for example Figure 2 as the accesses for a constant value of byte i (say `x00`).

2. *Discovery of the correct access*: The access corresponding to the value of byte i is found by taking the unique access present on *every* encryption (cfr. Figure 3 where byte $i =$ `x00` is found to be linked to cache line 2. We define in this case false positives as the wrong candidates present along with the correct candidate: *e.g.* the number of false positives on this example is
   - 10 on encryption 1,
   - 6 on encryption 2,
   - 5 on encryption 3,
   - 3 on encryptions 4 and 5,
   - 1 on encryption 6 and further
   - 0 on encryption 7 and further.

   The probability of a false positive accessed for k successive encryptions is $\left(1 - ((m-1)/m)^{15}\right)^k$ and this gives less than 4 percents when $k = 7$.
3. *Application of the difference*: The bitwise difference of the selected values of byte i must also link two elements in the corresponding access of T_4. Operation (2) showed that byte $i =$ `x00` is linked to cache line 2. Let us assume that the same operation was being executed on a different value of byte i (*e.g.* `x01`) and the corresponding cache line was 5. Therefore the bitwise difference of the values for byte i is `x00` $\oplus$ `x01` $=$ `x01` $= 1$. Hence we only need to find, in the cache lines 2 and 5, output values presenting the same difference. The two lines are shown below:

```
...
2    b7 fd 93 26 36 3f f7 cc 34 a5 e5 f1 71 d8 31 15
...
5    53 d1 00 ed 20 fc b1 5b 6a cb be 39 4a 4c 58 cf
...
```

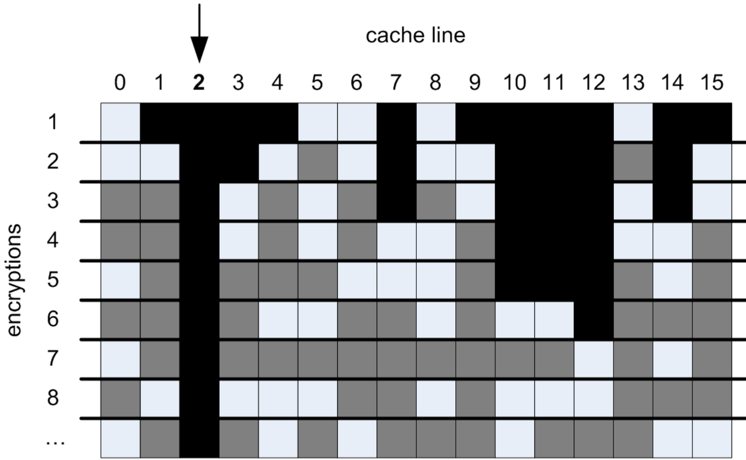

Fig. 3. Highlight of the constant access. The dark boxes represent accessed cache lines and the black boxes show the evolution of the possible candidates.

> The only pair having a bitwise difference of 1 are `x fd` and `x fc`, when byte i is equal to respectively `x 00` and `x 01`. Therefore, the byte of $\mathbf{K}^{(10)}$ corresponding to byte i has a value of `x fd` $\oplus$ `x 00` $=$ `x fc` $\oplus$ `x 01` $=$ `x fd`. In the unlikely event of more than one match, the operation (2) must be repeated to establish other bitwise differences.

The expected number of pairs to find the correct key byte is

$$\sum_{n=2}^{\infty} p_{fp}(n) \cdot N(n) \approx 186,$$

with $p_{fp}(n)$ and $N(n)$ respectively being the probability of having a false positive after n pairs and the average number of pairs necessary to get two values repeated. The other bytes of $\mathbf{K}^{(10)}$ are found the same way, by considering another byte number.

7.2 Elimination Method

Here, all bytes can be treated at the same time. We consider the case of byte i for the sake of clarity; it is straightforward to apply the method to the other ones. Let $\mathcal{V}$ be the set of all possible key byte values. Initially, $\mathcal{V}$ is composed of all 256 values a byte can take: $\mathcal{V} = \{j : 0 \leq j \leq 255|j\}$. At the end, we want that $\mathcal{V} = \{k_i^{(10)}\}$. Consider for example that the ciphertext's byte i c_i equals `x 2c` and the corresponding accesses are the ones displayed in Figure 4.

The accessed cache lines are

$$1\ 2\ 3\quad 5\ 6\ 7\quad 9\ 10\quad 12\quad 14\ 15$$

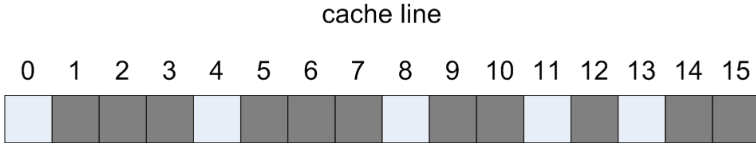

Fig. 4. Example of accessed cache lines. The dark boxes represent accessed cache lines.

and the non-accessed ones are

$$0\ 4\ 8\ 11\ 13.$$

This method focuses on the latter list of cache lines. Let $\widetilde{\mathcal{A}}$ represent this subset of the cache lines and $n_{\widetilde{\mathcal{A}}}$ be the number of elements of $\widetilde{\mathcal{A}}$:

$$\widetilde{\mathcal{A}} = \{0, 4, 8, 11, 13\}, \qquad n_{\widetilde{\mathcal{A}}} = \left|\widetilde{\mathcal{A}}\right| = 5.$$

By the elimination relation, $\mathbf{K}^{(10)} \neq \mathbf{c} \oplus \neg[T_4 \text{ outputs}]$, each non-accessed cache line enables us to remove all key candidates corresponding to this access. In our example, this means that for the first element of $\widetilde{\mathcal{A}}$ we have:

$$\begin{aligned} K_i^{(10)} &\neq c \oplus [\text{cache line } 0] \\ &\neq \mathtt{x\,2c} \oplus {}_{\mathtt{x}}\{\mathtt{63,7c,77,7b,f2,6b,6f,c5,30,01,67,2b,fe,d7,ab,76}\} \\ &\neq {}_{\mathtt{x}}\{\mathtt{4f,50,5b,57,de,47,43,e9,1c,2d,4b,07,d2,fb,87,5a}\} \\ &= \mathcal{V}_e, \end{aligned}$$

where $\mathtt{x}\ldots$ and ${}_{\mathtt{x}}\{\ldots\}$ represent hexadecimal values. All values of $\mathcal{V}_e$ can then be eliminated from $\mathcal{V}$:

$$\mathcal{V} \leftarrow \{j : 0 \leq j \leq 255\} \backslash \mathcal{V}_e$$

Then we go to the next element of $\widetilde{\mathcal{A}}$ (*i.e.* cache line 4) and apply the same technique. The cache line bytes are

$${}_{\mathtt{x}}\{\mathtt{09,83,2c,1a,1b,6e,5a,a0,52,3b,d6,b3,29,e3,2f,84}\}$$

added with $\mathtt{x\,2c}$ gives new candidates to eliminate. Then $\mathcal{V}$ is updated as:

$$\mathcal{V} \leftarrow \ \mathcal{V} \backslash \{\mathtt{25,af,00,36,37,42,76,8c,7e,17,fa,9f,05,cf,03,a8}\}.$$

This is then repeated for the three other cache lines in $\widetilde{\mathcal{A}}$.

For one given ciphertext/accesses pair, each cache line ends up eliminating 16 different values from the byte candidates: the ciphertext byte is constant and the SBox outputs are all different from each other. In our example 80 candidates have been eliminated with the pair under consideration. Then, another ciphertext/access pair is analyzed and the same technique is applied with the non-accessed cache lines of that pair. The ciphertext's byte i and the non-accessed cache lines are probably different from the previous analyzed pair. Therefore the

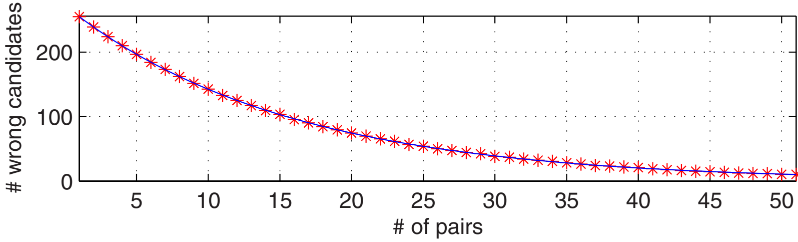

Fig. 5. Experimental verification of the reduction formula, yielding the number of wrong key candidates per pairs. The continuous line is the theoretical formula and the stars are the simulated data.

subset of wrong candidates deduced from this pair should only present a few collisions with the one of previous pairs.

However, there will be more and more collisions as the number of wrong key candidates becomes closer to one. Consider for example the following table showing the reduction of the number of wrong key candidates, for a practical case.

#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\|\mathcal{V}\|$	255	175	119	77	55	33	21	15	6	6	2	2	2	1	1	0

Then other pairs are analyzed until there is only one key byte remaining[4]. It is important to note that this method allows to work on all the bytes, at the same time. In this case, we need 16 subsets (one per byte) keeping the count of all candidates:

$$\mathcal{V}_0, \dots, \mathcal{V}_{15}.$$

Let us first compute the number of pairs needed to distinguish the right key candidate, for one byte. We can now define s, the number of candidates eliminated by the analysis of one ciphertext/access pair. At the beginning, we have 255 wrong candidates. With the first pair, we eliminate s of them and the number of wrong candidates is 255-s. However, starting at the second pair, collisions can occur. Therefore, we approximate the number of remaining wrong key candidates after n pairs by $255 \cdot (1 - s/255)^n$. This formula is validated by simulation with $s = 16$ (see Figure 5).

We then apply the formula with the expected number of non-accessed cache lines (*i.e.* 5.70, for 64-byte cache lines). Hence, substituting $s = 5.70 \cdot 16$ into $255 \cdot (1 - s/255)^n$ we get the following results[5] (cfr. Table 2).

After approximatively 14 pairs, the number of wrong candidates for one byte should be close to 0. This shows that less than 20 ciphertext/accesses pairs are

[4] Note that one can stop anytime and run an exhaustive search on the remaining key byte candidates.

[5] The data differ from the ones in the practical case because there are 5 non-accessed cache lines whereas 5.70 are considered in Table 2.

Table 2. Theoretical results of wrong key candidates, per pair ciphertext/accesses, for $m = 5.70 \cdot 16$

# pairs	$\|\mathcal{V}\|$	# pairs	$\|\mathcal{V}\|$
0	255	8	7
1	164	9	5
2	105	10	3
3	68	11	2
4	43	12	1
5	28	13	0.8
6	18	14	0.5
7	12	15	0.3

needed to recover the whole $\mathbf{K}^{(10)}$ subkey and therefore also the secret key $\mathbf{k}$. This method gives a much better performance than the non-elimination one.

8 Practical Considerations

Let us now re-elaborate the question of the measurement resolution.

- *Low resolution*: Table 1 highlighted that the expected number of accessed cache lines rapidly approaches to 16, when the number of encryptions between two measurements increases. However, even if the leakage gets smaller, every ciphertext/accesses pair with at least one non-accessed cache line carries information. Moreover, low resolution implies multiple ciphertexts for a single cache information (*i.e.* one line combines all the accesses corresponding to the ciphertexts). In this case the analysis must integrate the different possible ciphertext values and statistically derive the most likely key bytes.
- *One line resolution*: As detailed above, 5.70 cache lines are not accessed. The analysis does not need to deal neither with the multiple ciphertexts issue nor with the order inside the accesses.
- *High resolution*: Both methods are still possible. But the leakage also gives some information about the order of the accesses. One can then increase the performances of the analysis and therefore reduce the number of required pairs, by correlation of the byte accesses in the AES program and the accesses visible in the measurements. For $t \leq 1/16$, one can clearly identify the byte accesses. Two to three pairs only makes it possible to find the correct candidates for all key bytes[6].

Finally, we considered in this paper that the cache accesses were exempted of any measurements noise. However practical attacks must deal with noise in the measurements. Consider for example Figure 1 which presents vertical stripes and a diagonal line in the upper half. The presence of noise in the measurements

[6] The elimination and non-elimination methods then presents the same performances.

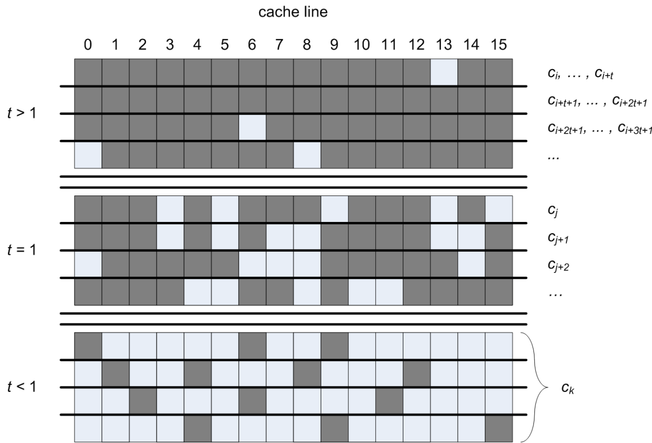

Fig. 6. Different resolutions for access-driven cache-based attacks. The resolution factor t defines the ratio *# of ciphertexts / # of measurements*.

increases the number of accessed cache lines. However, the techniques that we detailed here can still be exploited, by taking into account the noise[7].

We gave above the minimum expected number of measurements to perform the attacks for $t = 1$. As this boundary is precious and has been practically confirmed it should used to evaluate the efficiency and security of current software implementations which are hardened by corresponding countermeasures.

9 Summary

In this paper, we detailed advances on recent processor-oriented side channels. Our contribution is two-fold: we detailled a software method to achieve snapshots of cache accesses on single-threaded processors and we showed that the analysis of the last round of AES enables the full disclosure of an 128-bit AES key with less than 20 encryptions. Where previous studies focused exclusively on a minority of processors, we investigated the access-driven cache-based attacks on single-threaded processors. We explained our strategy and why it is solely depending on software engineering. Moreover, we chose the challenging case of AES: its short execution time (compared to RSA's) demonstrates the fine granularity of our cache accesses' snapshots. Our software strategy can easily be adapted and combined with previously reported access-driven attacks on any single-threaded processor. Moreover, on common implementations the last round is performed with the help of a special precomputed table. Through this feature, we achieved to infer more information than with other strategies. We gave expected numbers of measurements, depending on the granularity and

[7] Also, the location of the vertical stripes is variable between different runs of the setup.

noise of the access-driven measurements. This contribution sets new boundaries for countermeasures against cache-based attacks. For example, some software mitigations proposed to apply masking techniques and to renew the mask every 256 encryptions. We showed in this paper that this number might have to be reconsidered.

Acknowledgment

We would like to thank the anonymous reviewers for their useful comments and also for this sentence "Figure 1 should be framed on the wall in front of every crypto software programmer".

References

1. Openssl: the open-source toolkit for ssl / tls. Available online at `http://www.openssl.org/`
2. Bernstein, D.J.: Cache-timing attacks on AES (2004), Available onlineat `http://cr.yp.to/papers.html#cachetiming`
3. Brickell, E., Graunke, G., Neve, M., Seifert, J.-P.: Software mitigations to hedge aes against cache-based software side channel vulnerabilities. Cryptology ePrint Archive, Report, 2006/052 (2006), Available online at `http://eprint.iacr.org/`
4. Daemen, J., Rijmen, V.: The design of Rijndael, AES - The Advanced Encryption Standard. In: Information Security and Cryptology, Springer, Heidelberg (2001)
5. Handy, J.: The cache memory book (2nd ed.): the authoritative reference on cache design. Academic Press, Inc., Orlando, FL, USA (1998)
6. Hu, W.-M.: Lattice scheduling and covert channels. In: Proceedings of the IEEE Symposium on Security and Privacy, vol. 25, pp. 52–61 (1992)
7. Kelsey, J., Schneier, B., Wagner, D., Hall, C.: Side channel cryptanalysis of product ciphers. Journal of Computer Security 8(2/3) (2000)
8. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
9. Lampson, B.W.: A note on the confinement problem. Communications of the ACM 16(10), 613–615 (1973)
10. Neve, M., Seifert, J.-P., Wang, Z.: A refined look at Bernstein's AES side-channel analysis. In: Proceedings of AsiaCCS 2006 (2006)
11. Osvik, D.A., Shamir, A., Tromer, E.: Cache atacks and countermeasures: the case of AES (extended version) (2005), Available online at `http://www.wisdom.weizmann.ac.il/ tromer/`
12. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of aes. Cryptology ePrint Archive, Report, 2005/271, (2005) Available online at `http://eprint.iacr.org/2005/271.pdf`
13. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of aes. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006)
14. Percival, C.: Cache missing for fun and profit (2005), Available online at `http://www.daemonology.net/hyperthreading-considered-harmful/`

15. Shen, J., Lipasti, M.: Modern Processor Design: Fundamentals of Superscalar Processors. McGraw-Hill, New York (2005)
16. Silberschatz, A., Gagne, G., Galvin, P.B.: Operating system concepts, 7th edn. John Wiley and Sons, Inc., USA (2005)
17. Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., Miyauchi, H.: Cryptanalysis of des implemented on computers with cache. In: D.Walter, C., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 62–76. Springer, Heidelberg (2003)
18. Tsunoo, Y., Tsujihara, E., Minematsu, K., Miyauchi, H.: Cryptanalysis of block ciphers implemented on computers with cache. In: Proceedings of International Symposium on Information Theory and Its Applications, pp. 803–806 (2002)
19. Wray, J.C.: An analysis of covert timing channels. Journal of Computer Security 1(3-4), 219–232 (1992)