2011 IEEE Symposium on Security and Privacy

Cache Games – Bringing Access-Based Cache Attacks on AES to Practice

David Gullasch
Bern University of Applied Sciences, Dreamlab Technologies
david.gullasch@bfh.ch

Endre Bangerter
Bern University of Applied Sciences
endre.bangerter@bfh.ch

Stephan Krenn
Bern University of Applied Sciences, University of Fribourg
stephan.krenn@bfh.ch

***Abstract*—Side channel attacks on cryptographic systems exploit information gained from physical implementations rather than theoretical weaknesses of a scheme. In recent years, major achievements were made for the class of so called access-driven cache attacks. Such attacks exploit the leakage of the memory locations accessed by a victim process.**

In this paper we consider the AES block cipher and present an attack which is capable of recovering the full secret key in almost realtime for AES-128, requiring only a very limited number of observed encryptions. Unlike previous attacks, we do not require any information about the plaintext (such as its distribution, etc.). Moreover, for the first time, we also show how the plaintext can be recovered without having access to the ciphertext at all. It is the first working attack on AES implementations using compressed tables. There, no efficient techniques to identify the beginning of AES rounds is known, which is the fundamental assumption underlying previous attacks.

We have a fully working implementation of our attack which is able to recover AES keys after observing as little as 100 encryptions. It works against the OpenSSL 0.9.8n implementation of AES on Linux systems. Our spy process does not require any special privileges beyond those of a standard Linux user. A contribution of probably independent interest is a denial of service attack on the task scheduler of current Linux systems (CFS), which allows one to observe (on average) every single memory access of a victim process.

***Keywords*-AES; side channel; access-based cache attacks;**

I. Introduction

Cryptographic schemes preventing confidential data from being accessed by unauthorized users have become increasingly important during the last decades. Before being deployed in practice, such schemes typically have to pass a rigorous reviewing process to eliminate design weaknesses. However, theoretical soundness of a scheme is necessary but not sufficient for the security of concrete implementations of the scheme.

Side channel attacks are an important class of implementation level attacks on cryptographic systems. They exploit, for instance, the leakage of information from electromagnetic radiation or power consumption of a device, and running times of certain operations. Especially, side channel attacks based on cache access mechanisms of microprocessors represented a vivid area of research in the last few years [1]–[16]. These cache based side channel attacks (or *cache attacks* for short) fall into the categories of time-driven, trace-driven, and access-driven attacks.

In time-driven attacks an adversary is able to observe the overall time needed to perform certain computations, such as whole encryptions [9]–[12]. These timings leak information about the overall number of cache hits and misses during an encryption. In trace-driven attacks, an adversary is able to obtain a profile of the cache activity during an encryption, and to deduce which memory accesses issued by the cipher resulted in a cache hit [13]–[16]. Finally, access-driven attacks additionally enable the adversary to determine the cache sets accessed by the cipher [4]–[7]. Therefore, he can infer, e.g., which elements of a lookup table have been accessed by the cipher.

All these three types of attacks exploit the fact that accessing cached data is up to two orders of magnitude faster than accessing data in the main memory. The attack scenario underlying such attacks is a follows: Consider two concurrently running processes (a *spy process* S and a security sensitive *victim process* V) using the same cache. After letting V run for some small amount of time and potentially letting it change the state of the cache, S observes the timings of its own memory accesses, which depend on the state of the cache. These measurements allow S to infer information about the memory locations previously accessed by V.

A. Our Contributions

In a nutshell, we present a novel, practically efficient access-driven cache attack on the Advanced Encryption Standard (AES) [17], [18], which is the most widely used symmetric-key block cipher today. On a high level the main features of our attack are as follows: First, our attack works under very weak assumptions, and thus is the strongest working access-driven attack currently known. Second, we provide a concrete and practically usable implementation of the attack. It uses new techniques and also resolves a series of so far open issues and technicalities.

Let us discuss our results in more detail. For our attack to work we need to assume that the attacker has a test machine

This work was in part funded by the European Community's Seventh Framework Programme (FP7) under grant agreement no. 216499 and the Swiss Hasler Foundation.

1081-6011/11 $26.00 © 2011 IEEE
DOI 10.1109/SP.2011.22

at his disposal prior to the attack, which is identical to the victim machine. The test machine is used to generate training samples for two artificial neural networks from 168 000 encryptions. These then have to be trained on an arbitrary platform.

To carry out the attack all we need to be able to execute a non-privileged spy process (e.g., our spy process does not need to have access to the network interface) on the victim machine. We do not require any explicit interactions, such as inter-process communication or I/O. Osvik et al. [7], [8] refer to attacks in this setting as *asynchronous attacks*.

Our attack technique has the following features:

- In contrast to previous work [6]–[8], our spy process neither needs to learn the plain- or ciphertexts involved, nor their probability distributions in order recover the secret key.
- For the first time, we describe how besides the key also the plaintext can be recovered without knowing the ciphertexts at all.
- Our attack also works against AES implementations using so called *compressed tables*, which are typically used in practice, e.g., in OpenSSL [19]. When using compressed tables, the first and the last round of an encryption typically cannot be identified any more, which renders previous attacks impossible.
- We have a fully working implementation of our attack techniques against the 128-bit AES It is highly efficient and is able to recover keys in "realtime". More precisely, it consists of two phases: In an observation phase, which lasts about 2.8 seconds on our test machine, approximately 100 encryptions have to be monitored. Then an offline analysis phase lasting about 3 minutes recovers the key. The victim machine only experiences a delay during the observation phase. This slowdown is sufficiently slight to not raise suspicions, since it might as well be caused by high network traffic, disk activity, etc. To the best of our knowledge, this is the first fully functional implementation in the asynchronous setting.
- At the heart of the attack is a spy process which is able to observe (on average) every single memory access of the victim process. This extremely high granularity in the observation of cache hits and misses is reached by a new technique exploiting the behavior of the Completely Fair Scheduler (CFS) used by modern Linux kernels. We believe that this scheduler attack could be of independent interest.

B. Test Environment

All our implementations and measurements have been obtained on a Intel Pentium M 1.5 GHz (codename "Banias") processor, in combination with an Intel ICH4-M (codename "Odem") chipset using 512 MB of DDR-333 SDRAM. On this system, we were running Arch Linux with kernel version 2.6.33.4. As a victim process we used the OpenSSL 0.9.8n implementation of AES, using standard configurations.

C. Related Work

It was first mentioned by Kocher [20] and Kelsey et al. [21] that cache behavior potentially poses a security threat. The first formal studies of such attacks were given by Page [22], [23].

First practical results for time-driven cache attacks on the Data Encryption Standard (DES) were given by Tsunoo et al. [2], and an adoption for AES was mentioned without giving details. Various time-driven attacks on AES were given in the subsequent [7]–[12], some of which require that the first or the last round of AES can be identified. Tiri et al. [24] proposed an analytical model for forecasting the security of symmetric ciphers against such attacks.

Trace-driven cache attacks were first described by Page [22], and various such attacks on AES exist [13]–[16]. Especially, Aciiçmez et al. [13] also propose a model for analyzing the efficiency of trace-driven attacks against symmetric ciphers.

Percival [4] pioneered the work on access-driven attacks and described an attack on RSA. Access-driven attacks on AES were first investigated by Osvik et al. [7], [8]. They describe various attack techniques and implementations in what they call the *synchronous model*. This model makes rather strong assumptions on the capabilities of an attacker, i.e., it assumes that an attacker has the ability to trigger encryptions for known plaintexts and know when an encryption has begun and ended. Their best attack in the synchronous model requires about 300 encryptions.

Osvik et al. also explore the feasibility of asynchronous attacks. They refer to asynchronous attacks as an "extremely strong type of attack", and describe on a rather high level how such attacks could be carried out, assuming that the attacker knows the plaintext distribution and that the attack is carried out on a hyper-threaded CPU. Also, they implement and perform some measurements on hyper-threaded CPUs which allow to recover 47 key bits. However, a description (let alone an implementation) of a full attack is not given and many open questions are left unresolved. Further, the authors conjecture that once fine-grained observations of cache accesses are possible, the plaintext distribution no longer needs to be known. Loosely speaking, one can say that Osvik et al. postulate fully worked and practical asynchronous attacks as an open problem.

This is where the work of Neve et al. [6] picks up. They make progress towards asynchronous attacks. To this end they describe and implement a spy process that is able to observe a "few cache accesses per encryption" and which works on single threaded CPUs. They then describe a theoretical known ciphertext attack to recover keys by analyzing the last round of AES. The practicality of their attack remains unclear, since they do not provide an

implementation and leave various conceptual issues (e.g., quality of noise reduction, etc.) open.

Acıiçmez et al. [5] are the first to present a practical access-driven attack in the asynchronous model. Albeit targetting OpenSSL's DSA implementation via the instruction cache on a hyper-threaded CPU, they contribute a clever routine to perform timing measurement of the instruction cache in a real-world setting.

We improve over prior work by providing a first practical access-driven cache attack on AES in the asynchronous model. The attack works under weaker assumptions than previous ones as no information about plain- and ciphertext is required[1], and it is more efficient in the sense that we only need to observe about 100 encryptions. We also reach a novelly high granularity when monitoring memory accesses. Further, our attack also works against compressed tables, which were not considered before.

Finally, several hardware and software based mitigation strategies for AES have been proposed [25]–[27].

D. Document Outline

In §II we briefly recapitulate the structure of a CPU cache and the mechanisms underlying it. We also describe the Advanced Encryption Standard (AES) to the extent necessary for our attack. In §III we then explain how to recover the AES key under the assumption that one is able to perfectly observe single cache accesses performed by the victim process. We drop this idealization in §IV and show that by combining a novel attack on the task scheduler and neural networks sufficiently good measurements can be obtained to carry out the attack in practice. We also state measurement results obtained from the implementation of our attack. In §V we present extensions of our attack, and countermeasures in §VI. We conclude with a discussion of the limitations of our attack and potential future work in §VII.

II. Preliminaries

We first summarize the functioning of the CPU cache as far as necessary for understanding our attack. We then describe AES, and give some details on how it is typically implemented. We close this section by describing the test environment on which we obtained our measurements.

A. The CPU Cache and its Side Channels

Let us describe the behavior of the CPU cache, and how it can be exploited as a side channel. The CPU cache is a very fast memory which is placed between the main memory and the CPU [28]. Its size typically ranges from some hundred kilobytes up to a few megabytes.

[1]To be precise, the statement is true whenever AES is used, e.g., in CBC or CTR mode, which is the case for (all) relevant protocols and applications. In the practically irrelevant case, where the ECB mode (which is known to be insecure by design) is used we have to require that there is some randomness in the plaintext.

Typically, data the CPU attempts to access is first loaded into the cache, provided that it is not already there. This latter case is called a *cache hit*, and the requested data can be supplied to the CPU core with almost no latency. However, if a *cache miss* occurs, the data first has to be fetched via the front side bus and copied into the cache, with the resulting latency being roughly two orders of magnitude higher than in the former case. Consequently, although being logically transparent, the mechanics of the CPU cache leak information about memory accesses to an adversary who is capable of monitoring cache hits and misses.

To understand this problem in more detail it is necessary to know the functioning of an *n-way associative cache*, where each physical address in the main memory can be mapped into exactly n different positions in the cache. The cache consists of 2^a *cache sets* of n *cache lines* each. A cache line is the smallest amount of data the cache can work with, and it holds 2^b bytes of data together with tag and state bits. Cache line sizes of 64 or 128 bytes (corresponding to $b = 6$ and $b = 7$, respectively) are prevalent on modern x86- and x64 architectures.

To locate the cache line holding data from address $A = (A_{\max}, \ldots, A_0)$, the b least significant bits of A can be ignored, as a cache line always holds 2^b bytes. The next a bits, i.e., $(A_{a+b-1}, \ldots, A_b)$ identify the cache set. The remaining bits, i.e., $(A_{\max}, \ldots, A_{a+b})$ serve as a tag. Now, when requesting data from some address A, the cache logic compares the tag corresponding to A with all tags in the identified cache set, to either successfully find the sought cache line or to signal a cache miss. The state bits indicate if the data is, e.g., valid, shared or modified (the exact semantics are implementation defined). We typically have $\max = 31$ on x86 architectures and $\max = 63$ on x64 architectures. (However, the usage of physical address extension techniques may increase the value of $\max$ [29].)

Addresses mapping into the same cache set are said to *alias* in the cache. When more then n memory accesses to different aliasing addresses have occurred, the cache logic needs to evict cache lines (i.e. modified data needs to be written back to RAM and the cache line is reused). This is done according to a predetermined replacement strategy, most often an undocumented algorithm (e.g. *PseudoLRU* in x86 CPUs), approximating the eviction of the least recently used (LRU) entry.

With these mechanics in mind, one can see that there are at least two situations where information can leak to an adversary in multitasking operating systems (OS). Let's therefore assume that a *victim process* V, and a *spy process* S are executed concurrently, and that the cache has been initialized by S. After running V for some (small) amount of time, the OS switches back to S.

- If S and V physically share main memory (i.e., their virtual memories map into the same memory pages in RAM), S starts by flushing the whole cache. After

regaining control over the CPU, S reads from memory locations and monitors cache hits and misses by observing the latency. Hits mark the locations of V's memory accesses.

- If S and V do not physically share memory, then they typically have access to cache aliasing memory. In this case, S initializes the cache with some data D, and using its knowledge of the replacement strategy, it deterministically prepares the individual cache line states. When being scheduled again, S again accesses D, and notes which data had been evicted from the cache. This again allows S to infer information about the memory accesses of V.

Our target in the following is the OpenSSL library on Linux, which in practice resides at only one place in physical memory and is mapped into the virtual memory of every process that uses it. In this paper we are therefore concerned with the shared-memory scenario, where V uses lookup tables with 2^c entries of 2^d bytes each, and uses a secret variable to index into it. We will further make the natural assumption of cache line alignment, i.e., that the starting point of these lookup tables in memory corresponds to a cache line boundary. For most compilers, this is a standard option for larger structures. Exploiting the previously mentioned information leakage will allow S to infer the memory locations V accesses into, up to cache line granularity. That is, S is able to reconstruct $l = c - \max(0, b - d)$ bits of the secret index for a cache line size of 2^b bytes. Note that $l > 0$ whenever the lookup table does not entirely fit into a single cache line. Starting from these l bits, we will reconstruct the whole encryption key in our attack.

B. AES – The Advanced Encryption Standard

The *Advanced Encryption Standard* (AES) [17] is a symmetric block cipher, and has been adopted as an encryption standard by the U.S. government [18]. For self-containment, and to fix our notation, we next recapitulate the steps of the AES algorithm [30, §4.2].

AES always processes blocks $(x_0 \ \dots \ x_F)$ of 16 bytes at a time by treating them as 4×4 matrices. We will denote these matrices by capital letters, and its column vectors by bold, underlined lowercase letters:

$$X = \begin{pmatrix} x_0 & x_4 & x_8 & x_C \\ x_1 & x_5 & x_9 & x_D \\ x_2 & x_6 & x_A & x_E \\ x_3 & x_7 & x_B & x_F \end{pmatrix} = (\underline{\mathbf{x}}_0 \ \underline{\mathbf{x}}_1 \ \underline{\mathbf{x}}_2 \ \underline{\mathbf{x}}_3)$$

The single bytes x_i are treated as elements of $GF(2^8)$. We denote addition in this field by $\oplus$ and multiplication by $\bullet$. Note that the addition equals bitwise XOR. The irreducible polynomial for multiplication is given by $x^8+x^4+x^3+x+1$, see the standard [18] for details. We use these operations in the usual overloaded sense to operate on matrices and vectors.

Except for XORing the current state with a round key, the single rounds of AES makes use of three operations: *ShiftRows* cyclically shifts the rows of a matrix X, *SubBytes* performs a bytewise substitution of each entry in a matrix according to a fixed and invertible substitution rule, and *MixColumns* multiplies a matrix by a fixed matrix M.

In the first step of each round of AES, the *ShiftRows* operation performs the following permutation on the rows of a matrix X:

$$\mathrm{ShiftRows}(X) = \tilde{X} = \begin{pmatrix} x_0 & x_4 & x_8 & x_C \\ x_5 & x_9 & x_D & x_1 \\ x_A & x_E & x_2 & x_6 \\ x_F & x_3 & x_7 & x_B \end{pmatrix}$$

We will denote the columns of $\tilde{X}$ by $(\underline{\tilde{\mathbf{x}}}_0 \ \underline{\tilde{\mathbf{x}}}_1 \ \underline{\tilde{\mathbf{x}}}_2 \ \underline{\tilde{\mathbf{x}}}_3)$.

In the next step, all bytes of $\tilde{X}$ are substituted as defined by an *S-box*. We denote this substitution by $s(\cdot)$. That is, we have $SubBytes(\tilde{X}) = s(\tilde{X})$ with

$$s(\tilde{X}) = \begin{pmatrix} s(x_0) & s(x_4) & s(x_8) & s(x_C) \\ s(x_5) & s(x_9) & s(x_D) & s(x_1) \\ s(x_A) & s(x_E) & s(x_2) & s(x_6) \\ s(x_F) & s(x_3) & s(x_7) & s(x_B) \end{pmatrix},$$

or $s(\tilde{X}) = (s(\underline{\tilde{\mathbf{x}}}_0) \ s(\underline{\tilde{\mathbf{x}}}_1) \ s(\underline{\tilde{\mathbf{x}}}_2) \ s(\underline{\tilde{\mathbf{x}}}_3))$ for short.

Finally, the state matrices are multiplied by a constant matrix M in the *MixColumns* operation:

$$\mathrm{MixColumns}(s(\tilde{X})) = M \bullet s(\tilde{X}) = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \bullet s(\tilde{X})$$

As for X, we abbreviate the columns of M by bold letters. Here and in the remainder of this document, byte values have to be read as hexadecimal numbers.

Having said this, and denoting the round key of the i^{th} round by K_i, we can write AES as the following recurrence, where X_0 is the plaintext, and X_{r+1} is the ciphertext:

$$X_{i+1} = \begin{cases} X_i \oplus K_i & i = 0, \\ M \bullet s(\tilde{X}_i) \oplus K_i & 0 < i < r, \\ s(\tilde{X}_i) \oplus K_i & i = r\ . \end{cases} \quad (1)$$

For the 128-bit implementation of AES we have $r = 10$.

We will not detail the key schedule here, but only want to note that K_{i+1} can be obtained from K_i by applying a nonlinear transformation using the same S-box as the cipher itself, cyclically shifting the byte vectors, and XORing with $(2^i, 0, 0, 0)$ (where 2 has to be read as an element in $GF(2^8)$). The key schedule is illustrated in Figure 2.

C. How to Implement AES

In the following we briefly describe techniques for efficiently implementing AES. These techniques are the corner stones of our attack presented in the subsequent sections.

AES is heavily based on computations in $GF(2^8)$, whereas the arithmetic logic unit (ALU) of most CPUs only provides arithmetic on integers. The fast reference implementation of Rijmen et al. [31] reformulates AES to only use basic operations on 32-bit machine words. The implementation of OpenSSL [19] further exploits redundancies and halves the space needed for lookup tables, saving two kilobyte of memory. In the following we present these techniques on hand of one inner round of AES. We let X and Y denote the state matrices before and after the round, and K the round key. That is, we have the following relation:

$$Y = M \bullet \mathrm{s}(\tilde{X}) \oplus K. \tag{2}$$

Consequently, we have

$$\begin{aligned} \underline{\mathbf{y}}_0 &= \underline{\mathbf{m}}_0 \bullet s(x_0) \oplus \underline{\mathbf{m}}_1 \bullet s(x_5) \oplus \\ &\quad \underline{\mathbf{m}}_2 \bullet s(x_A) \oplus \underline{\mathbf{m}}_3 \bullet s(x_F) \oplus \underline{\mathbf{k}}_0, \end{aligned} \tag{3}$$

and similarly for $\underline{\mathbf{y}}_1, \underline{\mathbf{y}}_2, \underline{\mathbf{y}}_3$, where $\underline{\mathbf{k}}_0, \ldots, \underline{\mathbf{k}}_3$ denote the columns of K, indexed analogously to X and Y. To avoid the expensive multiplications in $GF(2^8)$ tables $T_0, \ldots, T_3$ containing all potential results are precomputed. That is, we have

$$T_i[x] = \underline{\mathbf{m}}_i \bullet s(x) \qquad 0 \le i \le 3.$$

This allows one to rewrite (3) in a form containing only table lookups and binary XORs:

$$\underline{\mathbf{y}}_0 = T_0[x_0] \oplus T_1[x_5] \oplus T_2[x_A] \oplus T_3[x_F] \oplus \mathbf{k}_0.$$

Each T_i has 2^8 entries of size 4 bytes each, and thus the tables require 4 kB of memory. However, they are highly redundant. For instance, we have that

$$\begin{aligned} T_1[x] &= (3\ 2\ 1\ 1)^T \bullet s(x) \\ &= (2\ 1\ 1\ 3)^T \bullet s(x) \ggg 3 \ = \ T_0[x] \ggg 3, \end{aligned}$$

where $\ggg$ denotes a bytewise rotation towards the least significant byte. Thus, it is possible to compute all T_i by rotating T_0. Yet, having to perform these rotations would cause a performance penalty. The idea thus is to use one table T the entries of which are doubled entries of T_0. That is, if T_0 has entries of the form $(abcd)$, those of T are of the form $(abcdabcd)$. We then have, e.g.,

$$T_1[x] = 32 \text{ bit word at offset } 3 \text{ in } T[x].$$

While saving 2 kB of memory and thus reducing the L1 footprint of the implementation substantially, this approach also allows to avoid the rotations by accessing T at the correct offsets.

While the above techniques work for most of AES, there are basically two ways to implement the last round which differs from the other rounds, cf. (1). First, an additional lookup table can be used for $s(\tilde{X})$ instead of $M \bullet s(\tilde{X})$. Alternatively, one can reuse the existing lookup table(s) by accessing them at appropriate positions. While the first technique can be exploited to identify where an encryption ends (and thus also to identify the first or last round of an encryption) by checking for memory accesses into this new table, this is not possible in the latter situation, as the observed memory accesses are indistinguishable from the previous rounds. Thus, in the latter situation most previous attacks [6], [7] cannot be executed. In particular this is the case for the compressed tables implementation of OpenSSL 0.9.8.

III. Breaking AES given Ideal Measurements

In this section, we show how the full secret AES key can be recovered under the assumption of ideal cache measurements. That is, we assume that a spy process can observe all cache accesses performed by a victim process in the correct order. Making this assumption considerably eases the presentation of our key recovery techniques. We will show how it can be dropped in §IV.

A. Using Accesses from Two Consecutive Rounds

As discussed in §II-A, having recorded the memory accesses into the lookup tables allows one to infer $l = c - \min(0, b - d)$ bits of the secret index, where the lookup table has 2^c entries of 2^d bytes each, and where a cache line can hold 2^b bytes. We therefore introduce the notation x^* to denote the l most significant bits of x, and also extend this notation to vectors. In our case, we have $c = 8$ and $d = 3$. If, for instance, we assume $b = 6$ we have $l = 5$ and consequently

$$\begin{pmatrix} 11001000_2 \\ 00010011_2 \\ 10010011_2 \\ 00101010_2 \end{pmatrix}^* = \begin{pmatrix} 11001_2 \\ 00010_2 \\ 10010_2 \\ 00101_2 \end{pmatrix}.$$

For ease of presentation and because of its high practical relevance on current CPUs will fix $b = 6$ for the remainder of this paper. However, the attack conceptually also works for other values of b, with a higher efficiency for $b < 6$ and a lower efficiency if $b > 6$.

From (2) it is now easy to see that the following equations are satisfied:

$$\underline{\mathbf{k}}_i^* = \underline{\mathbf{y}}_i^* \oplus (M \bullet s(\underline{\tilde{\mathbf{x}}}_i))^* \qquad 0 \le i \le 3\ . \tag{4}$$

Each of these equations specifies a set $\mathcal{K}_i \subseteq \{0,1\}^{4l}$ of *partial key column candidates*. Namely, we define $\mathcal{K}_i$ to consist of all elements $\underline{\mathbf{k}}_i^* \in \{0,1\}^{4l}$ for which the measured $\underline{\tilde{\mathbf{x}}}_i^*$ can be completed to a full four byte vector $\underline{\tilde{\mathbf{x}}}_i$ satisfying (4). These sets can be computed by enumerating all 2^{32-4l} possible values of $\underline{\tilde{\mathbf{x}}}_i$.

The cardinality of $\mathcal{K}_i$ turns out to depend on $\underline{\tilde{\mathbf{x}}}_i^*$, and so does the probability that some random $\underline{\mathbf{k}}_i^* \in \{0,1\}^{4l}$ is a partial key column candidate. As we need to argue about this probability we compute the expected cardinality of the $\mathcal{K}_i$ by assuming that the $\underline{\tilde{\mathbf{x}}}_i^*$ are equally distributed in $\{0,1\}^{4l}$.

l	$\lvert\{0,1\}^{4l}\rvert$	$\mathbb{E}[\lvert\mathcal{K}_i\rvert]$	$p_l = \mathbb{E}[\lvert\mathcal{K}_i\rvert]/\lvert\{0,1\}^{4l}\rvert$
1	2^4	2^4	1
2	2^8	2^8	1
3	2^{12}	2^{12}	1
4	2^{16}	$2^{14.661\ldots}$	$0.3955\ldots$
5	2^{20}	$2^{11.884\ldots}$	$3.6063\ldots \cdot 10^{-3}$
6	2^{24}	$2^{7.9774\ldots}$	$1.5021\ldots \cdot 10^{-6}$
7	2^{28}	2^4	$5.9604\ldots \cdot 10^{-8}$
8	2^{32}	1	$2.3283\ldots \cdot 10^{-10}$

Table 1. Depending on the number l of leaked bits, only a fraction p_l of the keys in $\{0,1\}^{4l}$ can be parts of the secret key's i^{th} column, if $\underline{\mathbf{x}}_i^*$ and $\underline{\mathbf{y}}_i^*$ are known. Here, $\mathbb{E}$ denotes the expectation value of the random variable $|\mathcal{K}_i|$.

Even though the encryption process is deterministic, this assumption seems to be natural, as otherwise the different states within an encryption would very likely be strongly biased, resulting in a severe security threat to AES.

Table 1 displays the expected sizes of $\mathcal{K}_i$ for all possible values of l. The last column of the table states the probability p_l that a random $\underline{\mathbf{k}}_i^* \in \{0,1\}^{4l}$ is a partial key column candidate for a random $\tilde{\underline{\mathbf{x}}}_i^*$. One can see that for $1 \leq l \leq 3$ every $\tilde{\underline{\mathbf{x}}}_i^*$ can be completed to a $\tilde{\underline{\mathbf{x}}}_i$ satisfying (4). Thus, in this case, this approach does not yield any information about the secret key K. On the other hand, for $l = 8$ the exact entries of the lookup table accessed by the victim process can be monitored and the key can be recovered from the states of two consecutive rounds only. In the interesting case where $3 < l < 8$ we learn a limited amount of information about the secret key. We will be concerned with this case in the following.

B. Using Accesses from Continuous Streams

The observations of the previous section typically cannot be directly exploited by an attacker. This is because for implementations of AES using compressed tables it is hard to precisely determine where one round ends and where the next one starts. Rather, an attacker is able to monitor a continuous stream of memory accesses performed by the victim process. Consequently, we will show how the key can be reconstructed from observations of multiple, say M, encryptions.

We remark that the order of memory accesses within each round is implementation dependent, but the single rounds are always performed serially, and each round always requires 16 table lookups. Thus, as (4) puts into relation states of consecutive rounds, it is always possible to complete all four equations (i.e., for $i = 0, \ldots, 3$) within the first 31 memory accesses after the first access in a round.

Assume now that an attacker is able to observe $160M + 31 = N + 31$ memory accesses. This means that quantitatively the accesses of M full encryptions are observed, but we do not require that the first observed access also is the first access of an encryption. The 31 remaining accesses belong to the $(M+1)^{st}$ encryption. On a high level, to circumvent the problem of not being able to identify round ends/beginnings, we now perform the following steps:

- We treat each of the first N observed memory accesses as if it was the beginning of an AES round.
- For each of these potential beginnings, we compute the sets of potential key column candidates. For each element of $\{0,1\}^{4l}$ we thereby count how often it lies in these sets.
- From these frequencies we derive the probability that a given element of $\{0,1\}^{4l}$ is a correct part of the unknown key.

More precisely, for any of the potential N beginnings of an AES round, we compute the sets $\mathcal{K}_i$ of partial key column candidates for $i = 0, \ldots, 3$, and count how often each $\underline{\mathbf{k}}_i^* \in \{0,1\}^{4l}$ also satisfies $\underline{\mathbf{k}}_i^* \in \mathcal{K}_i$. We denote this frequency by $f_i(\underline{\mathbf{k}}_i^*)$. Because of the 31 last monitored memory accesses, we have enough observations to complete (4) for any of these N offsets.

One can now see that $\underline{\mathbf{k}}_i^*$ is an element of $\mathcal{K}_i$ at least $z_{\underline{\mathbf{k}}_i^*} M$ times, if $\underline{\mathbf{k}}_i^*$ is the truncated part of the correct i^{th} column of a round key in $z_{\underline{\mathbf{k}}_i^*}$ different rounds. Put differently, we have $f_i(\underline{\mathbf{k}}_i^*) \geq z_{\underline{\mathbf{k}}_i^*} M$. For each of the remaining $N - z_{\underline{\mathbf{k}}_i^*} M$ wrong starting points we may assume that $\underline{\mathbf{k}}_i^*$ occurs in $\mathcal{K}_i$ with probability p_l. This is, because solving (4) for wrong values of $\underline{\mathbf{x}}_i^*, \underline{\mathbf{y}}_i^*$ should not leak any information about the correct key, even if $\underline{\mathbf{x}}_i^*, \underline{\mathbf{y}}_i^*$ are not fully random, but overlapping parts of correct values from subsequent rounds. In our experiments this assumption proved to be sufficiently satisfied for our purposes.

Denoting the binomial distribution for n samples and probability p by $\mathrm{Binomial}(n,p)$, we can now describe the properties of $f_i(\underline{\mathbf{k}}_i^*)$ as follows:

$$\begin{aligned} f_i(\underline{\mathbf{k}}_i^*) &\sim \mathrm{Binomial}(N - z_{\underline{\mathbf{k}}_i^*} M, p_l) + z_{\underline{\mathbf{k}}_i^*} M \\ \mathbb{E}[f_i(\underline{\mathbf{k}}_i^*)] &= N p_l + z_{\underline{\mathbf{k}}_i^*} M (1 - p_l) \\ \mathbb{V}[f_i(\underline{\mathbf{k}}_i^*)] &= (N - z_{\underline{\mathbf{k}}_i^*} M) p_l (1 - p_l) \ . \end{aligned}$$

From these equations one can see that every $\underline{\mathbf{k}}_i^*$ occurring in a round key causes a peak in the frequency table. We can now measure the difference of these peaks and the large floor of candidates $\underline{\mathbf{k}}_i^*$ which do not occur in a round key. This difference grows linearly in the number of observed encryptions M. On the other hand, the standard deviation $\sigma[f_i(\underline{\mathbf{k}}_i^*)] = \sqrt{\mathbb{V}[f_i(\underline{\mathbf{k}}_i^*)]}$ only grows like the square root of M (remember that $N = 160M$). Thus, the higher the number of encryptions, the better the peaks can be separated from the floor.

Using the $f_i(\underline{\mathbf{k}}_i^*)$ and the Bayes Theorem [32], [33] it is now possible to compute a posteriori probabilities $q_i(\underline{\mathbf{k}}_i^*)$ that a given $\underline{\mathbf{k}}_i^*$ really occurred in the key schedule of AES.

C. Key Search

In the previous section we assigned probabilities to partial key column candidates, such that only those occurring in the

correct key schedule have a high probability while all others do not. Before describing our key search heuristic, we now show how these probabilities can be used to assign scores to sets of partial key column candidates as well.

Let therefore $\mathcal{S}$ be a set of partial key column candidates, and let each element in $\mathcal{S}$ be tagged with the position of the key schedule it is a candidate for. The score of $\mathcal{S}$ is then given by the mean log probability of the elements in $\mathcal{S}$:

$$\mathrm{h}(\mathcal{S}) = \frac{1}{|\mathcal{S}|} \sum_{\underline{\mathbf{k}}_i^* \in \mathcal{S}} \log q_i(\underline{\mathbf{k}}_i^*) \ .$$

We now search for the correct unknown key. Loosely speaking, our technique outputs the K for which the mean log probability over all partial key columns in the whole key schedule is maximal. The algorithm stated below starts by fixing one entry of the key schedule which has a good score. It then adds another element to obtain a larger fraction of the whole schedule, which in turn forces one to fix even more entries, cf. Figure 2. Depending on the score of the resulting set it repeats this step or postpones the further investigation of the set and continues with another one with a higher score. This is repeated until a full key schedule with a high score is found.

- We start by searching for partial key column candidates for $\underline{\mathbf{k}}_3^{i^*}$, i.e., for the last column of the i^{th} round key. Therefore, we initialize a heap containing singletons for all possible values of $\underline{\mathbf{k}}_3^*$, sorted by their score $\mathrm{h}(\{\underline{\mathbf{k}}_3^*\})$.
- The topmost element of the heap, $\{\underline{\mathbf{k}}_3^{i^*}\}$ is removed, and combined with all partial key column candidates $\underline{\mathbf{k}}_3^*$, interpreted as candidates for $\underline{\mathbf{k}}_3^{i+1^*}$, i.e., as partial key column candidates for the last column of the $(i+1)^{st}$ round's key. As can be seen from Figure 2, combining $\{\underline{\mathbf{k}}_3^{i^*}\}$ with a candidate for $\underline{\mathbf{k}}_3^{i+1^*}$ also implies fixing $\underline{\mathbf{k}}_2^{i+1^*}$ because of the relation of round keys. We denote this operation of adding a partial key column candidate $\underline{\mathbf{k}}_3^{i+1^*}$ and all associated values to $\{\underline{\mathbf{k}}_3^{i^*}\}$ by $\uplus$. All the resulting sets $\{\underline{\mathbf{k}}_3^{i^*}\} \uplus \{\underline{\mathbf{k}}_3^{i+1^*}\}$ are added to the heap, according to their scores.
  This step is applied analogously whenever the topmost element of the heap does not at the same time contain candidates for $\underline{\mathbf{k}}_3^{i^*}$ and $\underline{\mathbf{k}}_3^{i+3^*}$.
- If the topmost element $\mathcal{S}$ of the heap already contains a candidate for $\underline{\mathbf{k}}_3^{i+3^*}$, we compute the combinations $\mathcal{S} \uplus \{\underline{\mathbf{k}}_3^{i+4^*}\}$ for all possible choices of $\underline{\mathbf{k}}_3^{i+4^*}$ as before. However, because of the nonlinear structure of the key schedule of AES, we are now able to put into relation $\underline{\mathbf{t}}^{i+3^*}$ with parts of $\underline{\mathbf{k}}_3^{i+3}$, and check whether the nonlinearity can be solved for any i, i.e., for any fixed position of $\mathcal{S}$ in Figure 2 (there, we indicated the case $i = 2$). If this is not the case, we discard $\mathcal{S} \uplus \{\underline{\mathbf{k}}_3^{i+4^*}\}$, otherwise we add it to the heap.
  We proceed analogously for $\underline{\mathbf{k}}_3^{i+5^*}$.
- Let now the topmost element of the heap $\mathcal{S}$ already contain candidates for $\underline{\mathbf{k}}_3^{i^*}$ up to $\underline{\mathbf{k}}_3^{i+5^*}$.
  From Figure 2 we can see that given four $\underline{\mathbf{k}}_i^j$ in a line allows to fill the complete key schedule. Given $\mathcal{S}$, we already fixed $4 \cdot 4 \cdot l = 80$ bits of such a "line". Further, solving the nonlinearities in the key schedule yields 24 more bits. That is, only 24 bits of the potential key remain unknown. We now perform a brute-force search over these 2^{24} possibilities at each possible position of $\mathcal{S}$ in the key schedule.
  For all possible completions of $\underline{\mathbf{k}}_3^{i^*}, \ldots, \underline{\mathbf{k}}_3^{i+3^*}$, we compute the whole key schedule, i.e., we compute $\underline{\mathbf{k}}_i^j$ for $i = 0, \ldots, 3$, $j = 1, \ldots, 9$ and compute the score for $\{\underline{\mathbf{k}}_i^{j^*} : i = 0, \ldots, 3, j = 0, \ldots, 9\}$. We store the key corresponding to the set with the highest score, together with its score.
  This step can be implemented efficiently because the solved nonlinearity typically only has 1 solution. In the rare case that there are more solutions, the above step is performed for either of them.
- We now continue processing the heap until its topmost element has a smaller score than the stored full key. In this case, we output the stored key and quit.
  Typically the output of our algorithm is the key schedule with a maximum score, as usually the score of a set decreases when extending it. This is because even when adding a candidate with very high score to some set of partial key column candidates, most often other parts with worse scores also have to be added due to the structure of the key schedule.

We remark that the symmetry of the key schedule can be used to increase the efficiency when actually implementing our attack in software. For instance, a triangle $\mathcal{S}$ with some fixed "base line" has the same score $\mathrm{h}(\mathcal{S})$ as the triangle flipped vertically. For this reason, the score only has to be computed for one of these triangles in the first two steps of our attack.

IV. Attacking AES in the Real World

In the previous section we showed how the full secret key can efficiently be recovered under the assumption that the cache can be monitored perfectly. That is, we assumed that an attacker is able to observe any single cache access performed by the victim process. We now show how our attack can be carried out in the real world where this idealization is no longer satisfied. We therefore first describe the way the task scheduler of modern Linux kernel works, and explain how its behavior can be exploited for our purposes. We then briefly recapitulate the concept of neural networks, and show how they can be used by an attacker to clean inaccurate measurements.

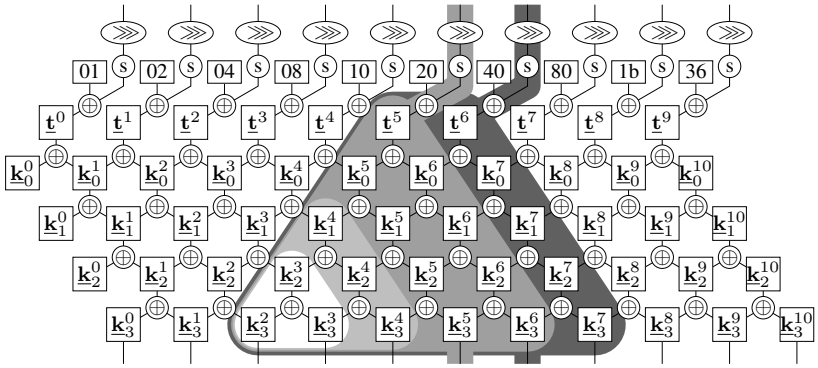


Figure 2. Key schedule of the 128 bit variant of AES. By $\underline{\mathbf{k}}_m^n$ we denote the m^{th} column of the n^{th} round key. The $\underline{\mathbf{t}}^n$ are temporary variables without any further semantics. The bottommost elements, i.e., $\underline{\mathbf{k}}_i^0$, are passed as inputs to the nonlinearity at the top. During the key search, incrementally parts of the key schedule are fixed in order to find a full key schedule with maximal score.

A. CFS – The Completely Fair Scheduler

Scheduling is a central concept in multitasking OS where CPU time has to be multiplexed between processes, creating the illusion of parallel execution. In this context there are three different states a process can possibly be in (we do not need to distinguish between processes and threads for now):

- A *running* process is currently assigned to a CPU and uses that CPU to execute instructions.
- A *ready* process is able to run, but temporarily stopped.
- A *blocked* process is unable to run until some external event happens.

The scheduler decides when to *preempt* a process (i.e., set it from running to ready) and which process to *activate* next when a CPU becomes idle (i.e., set it from ready to running). This is a difficult problem, because of the multiple, conflicting goals of the scheduler:

- *guaranteeing fairness* according to a given policy,
- *maximizing throughput* of work that is performed by processes (i.e., avoid wasting time on overhead like context switching and scheduling decisions) and
- *minimizing latency* of responses to external events.

Starting from Linux kernel 2.6.23, all Linux systems are equipped with the *Completely Fair Scheduler* (or CFS) [34], whose general principle of operation we describe in the following. Its central design idea is to asymptotically behave like an ideal system where n processes are running truly in parallel on n CPUs clocked at $1/n^{th}$ of normal speed each. To achieve this on a real system, the CFS introduces a *virtual runtime* τ_i for every process i. In the ideal system, all virtual runtimes would increase simultaneously and stay equal when the processes were started at the same time and never block. In a real system, this is clearly impossible, as only the running process's virtual runtime can increase at a time. Therefore CFS keeps a timeline (an ordered queue) of virtual runtimes for processes that are not blocked. Let

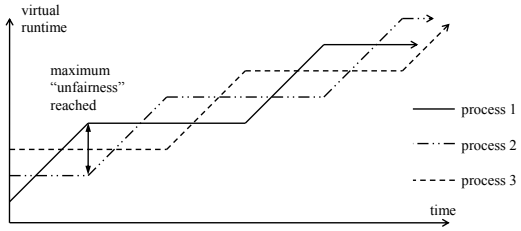


Figure 3. Functioning of the Completely Fair Scheduler. Here, three process are running concurrently. After process 1 was the assigned the CPU for some time, process 2 is the next to be activated to keep the unfairness among the different processes smaller than some threshold.

the difference between the rightmost and leftmost entries be $\Delta\tau = \tau_{\text{right}} - \tau_{\text{left}}$. This difference of the virtual runtimes of the most and least favorably scheduled processes can be interpreted as *unfairness*, which stays always zero in an ideal system. CFS lives up to its name by bounding this value above by some $\Delta\tau_{\text{max}}$. It always selects the leftmost process to be activated next and preempts the running rightmost process when further execution would result in $\Delta\tau \geq \Delta\tau_{\text{max}}$.

This logic is illustrated in Figure 3 where three processes are running on a multitasking system. At the beginning, process 1 is the next to be activated because it has the least virtual runtime. By running process 1 for some time the unfairness is allowed to increase up to $\Delta\tau_{\text{max}}$. Then CFS switches to process 2, which became the leftmost entry on the timeline in the meantime. This procedure is repeated infinitely so that every process asymptotically receives its fair share of $1/n^{th}$ CPU computing power per time.

A very important question is how to compute the virtual time of a processes which blocked at τ_{block} when it is unblocked again. We denote this computed virtual runtime by τ_{unblock}. Following a concept called *sleeper fairness* it is desirable that the process is activated as soon as possible and given enough time to react to the event it was waiting for

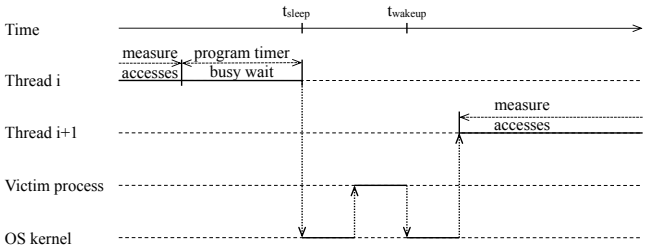


Figure 4. Sequence diagram of our denial of service attack on CFS. Multiple threads run alternatingly and only leave very small periods of time to the victim process.

with low latency. In CFS terms this means assigning it the lowest possible virtual runtime while not violating CFS's invariants: to not exceed the maximum unfairness it must hold that $\tau_{\text{unblock}} < \tau_{\text{right}} - \Delta\tau_{\text{max}}$. Also, the virtual runtime must not decrease by blocking and unblocking to prevent a trivial subversion of CFS's strategy. Therefore τ_{block} needs to be remembered and serves as another lower bound. Finally, we get

$$\tau_{\text{unblock}} = \max(\tau_{\text{block}}, \tau_{\text{right}} - \Delta\tau_{\text{max}}).$$

By blocking for a sufficiently long time, a process can ensure that it will be the leftmost entry on the timeline with $\tau_{\text{left}} = \tau_{\text{right}} - \Delta\tau_{\text{max}}$ and preempt the running process immediately.

B. A Denial of Service Attack on CFS

On a high level, our spy process S measures the memory accesses of the victim process V as follows: It requests most of the available CPU time, and only leaves very small intervals to V. By choosing the parameters of S appropriately, V will only be able to advance by one memory access on average before it is preempted again. Then, S accesses each entry of the lookup table, and checks whether a cache hit, or a cache miss occurs. Next V is again allowed to run for "a few" CPU cycles, and V measures again, etc.

In this section, we describe how the sleeper fairness of CFS can be exploited for the denial of service (DoS) attack underlying our spy process. The procedure for measuring cache accesses can be found in §IV-C.

When getting started, our spy process launches some hundred identical threads, which initialize their virtual runtime to be as low as possible by blocking for a sufficiently long time. Then they perform the following steps in a round-robin fashion, which are also illustrated in Figure 4:

- Upon getting activated, thread i first measures which memory accesses were performed by V since the previous measurement.
- It then computes t_{sleep} and t_{wakeup}, which denote the points in time when thread i should block and thread $i+1$ should unblock. It programs a timer to unblock thread $i+1$ at t_{wakeup}.
- Finally, thread i enters a busy wait loop until t_{sleep} is reached, where it blocks to voluntarily release the CPU.

In the time where no spy thread is active the kernel first activates the victim process (or some other process running concurrently on the system). This process is allowed to run until the timer unblocking thread $i+1$ expires. Because of the large number of threads and the order they run, their virtual runtimes will only increase very slowly. Thus, upon unblocking a spy thread is the leftmost element in the timeline of CFS and the currently running process is immediately preempted. This mechanism ensures that S immediately regains control of the CPU after V ran.

Typically, $t_{\text{wakeup}} - t_{\text{sleep}}$ is set to about 1500 machine cycles. Subtracting time spent executing kernel code and for context switching, this leaves less than 200 cycles for the CPU to start fetching instructions from V, decode and issue them to the execution units and finally retire them to the architecturally visible state, which is saved when the timer interrupts. When V performs memory accesses which result in cache misses, these few hundreds cycles are just enough to let one memory access retire at a time, on average.

Because of different timers used within the system, accurately setting t_{sleep} and t_{wakeup} is a challenging issue. In a first step, we have to find out the precise relation between the time stamp counter (in machine cycles), and the wall time of the OS (in nanoseconds as defined by the POSIX timer API). This can be achieved by repeatedly measuring the CPU time using the `rdtsc` instruction and the OS time, and interpolating among these values. This approximation only has to be performed once for every hardware setting. For our test environment, we got 0.6672366819 ns per CPU cycle. When starting our spy process the offset of the time stamp counter to the OS time is measured, which enables us to convert time measured by `rdtsc` to OS time with very high accuracy.

Since newer Linux versions change the CPU clock to save power when the idle thread runs, a dummy process with very low priority is launched to prevent the idle thread from changing the linear relationship between OS time and time stamp counter.

But even with exact computations of t_{wakeup} and t_{sleep} there are still other sources of inaccuracy. First, the time spent in the OS kernel stays constant for many measurements, but sometimes abruptly changes by hundreds of machine cycles. This is dynamically compensated by a feedback loop that adjusts $t_{\text{wakeup}} - t_{\text{sleep}}$ according to the rate of observed memory accesses. Second, the clock and timer devices do not actually operate with nanosecond accuracy as suggested by their APIs. As a result the actual time when the timer expires lies in an interval of about ± 100 machine cycles around t_{wakeup} for our hardware setting. In theory this could also be compensated with a more complex computational model of the hardware. However, assuming a linear relationship between the time stamp counter and OS time is sufficient for our purposes.

To hide the spy process from the user $t_{\text{wakeup}} - t_{\text{sleep}}$ is

```
#define CACHELINESIZE 64
#define THRESHOLD 200
unsigned measureflush(const uint8_t *table,
                      size_t tablesize,
                      uint8_t *bitmap)
{
        size_t i;
        uint32_t t1, t2;
        unsigned n_hits = 0;

        for (i = 0; i < tablesize/CACHELINESIZE; i++)
        {
                __asm__ (
                        "xor %%eax, %%eax    \n"
                        "cpuid               \n"
                        "rdtsc               \n"
                        "mov %%eax, %%edi    \n"
                        "mov (%%esi), %%ebx  \n"
                        "xor %%eax, %%eax    \n"
                        "cpuid               \n"
                        "rdtsc               \n"
                        "clflush (%%esi)     \n"
                : /* output operands */
                        "=a"(t2), "=D"(t1)
                : /* input operands */
                        "S"(table + CACHELINESIZE * i)
                : /* clobber description */
                        "ebx", "ecx", "edx", "cc"
                );

                if (t2 - t1 < THRESHOLD) {
                        n_hits++;
                        bitmap[i/8] |= 1 << (i%8);
                } else {
                        bitmap[i/8] &= ~(1 << (i%8));
                }
        }

        return n_hits;
}
```

Listing 5. Complete C source code for checking which parts of a lookup table `table` have been accessed by some process shortly before.

dynamically increased if no memory accesses are detected for an empirically set number of measurements. This allows the system to react to the actions of an interactive user with sufficient speed while no victim process is running.

Remark: Note that in spirit our DoS attack is similar to that of Tsafrir et al. [35]. However, while their attack is still suited for the current BSD family, it does not work any more for the last versions of the Linux kernel. This is because the logics of billing the CPU time of a process has advanced to a much higher granularity (from ms to ns) and no process can be activated without being billed by CFS any more, which was a central corner stone of their attack.

C. Testing for Cache Accesses

In the foregoing we described how the fairness condition of the CFS can be exploited to let the victim process advance by only one table lookup on average. We next show how the spy process can learn information about this lookup. That is, we show how the spy process can find the memory location the victim process indexed into, up to cache line granularity.

An implementation of this procedure in C is given in Listing 5, which we now discuss in detail. On a high level, it measures the time needed for each memory access into the lookup table and infers whether or not this data had already been in the cache before.

We start by describing the inner block of the `for` loop. The `__asm__` keyword starts a block of inline assembly, consisting of four parts: the assembly instructions, the outputs of the block, its inputs, and a list of clobbered registers. These parts are separated by colons. For ease of presentation, we describe these blocks in a different order in the following.

- *Inputs:* Only one input is given to the assembly block, namely a position in the lookup table. The given command specifies to store this position into the register `%esi`. The lookup table is traversed during the outer `for` loop, starting at the very beginning in the first iteration.
- *Assembly Instructions:* The first instruction, `xor%eax, %eax` is a standard idiom to set the register `%eax` to zero, by XORing it with its own content. Then, the `cpuid` instruction stores some information about the CPU into the registers `%eax,%ebx,%ecx,%edx`. We do not need this information in the following. The only purpose of these two instructions is the side effect of the latter: namely, `cpuid` is a serializing instruction, i.e., it logically separates the instructions before and after `cpuid`, as the CPU must not execute speculatively over such an instruction. Then, a 64 bit time stamp is stored into `%edx:%eax` by using the `rdtsc` instruction. The most significant bits of this time stamp are discarded, and the least significant bits (which are stored in `%eax`) are moved to `%edi` to preserve them during the following operations. Having this, the procedure accesses data at address `%esi` in the main memory and stores it to `%ebx`. Similar to the beginning, the CPU is forced to finish this instruction, before again a time stamp is stored to `%eax`, and the accessed data is flushed from the cache again by flushing its cache line using `clflush(%%esi)`.
- *Outputs:* In each iteration, the least significant bits of both time stamps are handed back: the content of the register `%%eax` is stored to `t2`, and that of `%%edi` is stored to `t1`.
- *Clobbered Registers:* The last block describes a list of clobbered registers. That is, it tells the compiler which registers the assembly code is going to use and modify. It is not necessary to list output registers here, as the compiler implicitly knows that they are used. The remaining `cc` register refers to the condition code register of the CPU.

Now, depending on the difference of `t1` and `t2`, the procedure decides whether the accesses resulted in a cache hit. These cache hits and misses describe whether or not the victim processes accessed the corresponding cache line in its last activation with high probability. The `THRESHOLD`

of 200 CPU cycles has been found by empirical testing. Note here that the serializing property of the `cpuid` instructions forces the CPU to always execute the same instructions to be timed between two `rdtsc` instructions, disregarding superpipelining and out-of-order instruction scheduling.

These steps are performed for the whole lookup table, starting at the beginning of the table in the memory (`i=0`) and counting up in steps of size of the cache line, as this is the highest precision that can be monitored. The number `n_hits` of cache hits and a bitmap `bitmap` containing information about where cache hits and where cache misses occurred are then handed back to the caller of the function.

D. Using Neural Networks to Handle Noise

Naturally, the measurements obtained using the techniques from §IV-B and §IV-C are not perfect, but overlaid with noise. This is because not only the victim and the spy process, but also other processes are running concurrently on the same system. They also perform memory accesses, which can cause flawed identifications of cache hits and misses. Also, sometimes the spy process will be able to advance by more than only one memory access at a time. Further, massive noise can be caused by prediction logic of the CPU, cf. §IV-D4.

Thus, filtering out noise is a core step in our attack, which we achieve by using artificial neural networks (ANNs). In the following, we describe the configuration and training of the ANNs we use.

1) Introduction to Artificial Neural Networks: An artificial neural network [36]–[39] is a computational model inspired by the workings of biological neurons. Simply speaking, neurons are interconnected nerve cells, communicating via electrical signals. They fire a signal when their summed inputs exceed an activation threshold.

An ANN can be represented as a directed graph with a value attached to each of its nodes. Some of the nodes are labeled as input or output nodes. Except for the input nodes, the value of each node is computed from the values attached to its predecessors. For a node i, let z_i denote the value associated with i and for an edge from j to i, let w_{ij} be an associated weight. The weight w_{i0} does not belong to an edge, it only serves to bias the sum that is fed into i's activation function σ, which typically is a sigmoid function. Then z_i is computed as

$$z_i = \sigma\left(w_{i0} + \sum w_{ij} z_j\right).$$

Let $\underline{\mathbf{x}}$ be the vector of values assigned to the input nodes, $\underline{\mathbf{y}}$ the vector of values computed for the output nodes and $\underline{\mathbf{w}}$ the vector of weights. With these, the network computes a function $\underline{\mathbf{y}} = f_{\underline{\mathbf{w}}}(\underline{\mathbf{x}})$. By choosing an appropriate network structure, activation function and weights, a neural network can be used to approximate an arbitrary target function. Usually, the network structure and activation function are fixed first. Finding values for the weights is then formulated as an optimization problem, i.e., one searches for a $\underline{\mathbf{w}}$ minimizing the error, which is the distance between $f_{\underline{\mathbf{w}}}$ and the target function (where difference is, e.g., the mean square error). This problem can seldom be solved algebraically and therefore nonlinear optimization techniques are used in a training phase to find sufficiently good weights. We refer to standard literature on artificial neural networks for detailed discussions [38], [39].

2) Overview of our ANNs: We use two neural networks to remove noise from our measurements: the many stray cache hits are tackled in the first and inaccuracies of the DoS on the scheduler in the second.

Measurements are made as described in §IV-C and represented by a rectangular bitmap. It is generated by using the result of each call to `measureflush()` in the spy process as one column concatenating them, so that columns are sorted chronologically from left to right. This is shown in Figure 6(a), hits and misses recorded as 1 and 0 are shown by black and white pixels in the bitmap. Because of the size of the lookup table (2 kB) and the size of each cache line ($2^6 = 64$ B), 32 addresses have to be considered. Here, 61 activations are shown and therefore the bitmap's size is 61 by 32 pixels.

The first network outputs the probability that at a given pixel in the bitmap a memory access was actually performed in the victim since the last measurement. For this, a rectangular area centered on the pixel of interest is used as input vector. We use a square of edge length 23 filled with zeros where it extends beyond the borders of the bitmap. The probabilities generated for all pixels in the input bitmap are again organized in columns.

We then use a second ANN to estimate how many memory accesses the victim performs between two measurements. This is important for accurately estimating the timeline of memory accesses. It is not guaranteed that exactly one memory access retires in the victim between two measurements because of inaccuracy of the scheduler DoS. Sometimes no memory access may retire, because the victim did not run at all or for too few instructions. At other times, several memory accesses retire, because the victim was interrupted too late. The input of this second ANN is the sum of probabilities over one column produced by the first ANN. Its output is a value in $\mathbf{R}$, which is used to resize the width of the corresponding column of probabilities. After resizing and concatenating the columns in order, the result is a map of probabilities shown in Figure 6(b). In one step in the horizontal direction, one memory access is performed in the victim.

3) Parameters and Structure of our ANNs: We now give a concise summary of the design of our neural networks, whereas we assume that the reader is familiar with ANNs.

The first ANN is a multilayer neural network. It has $23^2 = 529$ input nodes, one layer of 30 hidden nodes and one output node. Every hidden node has incoming edges

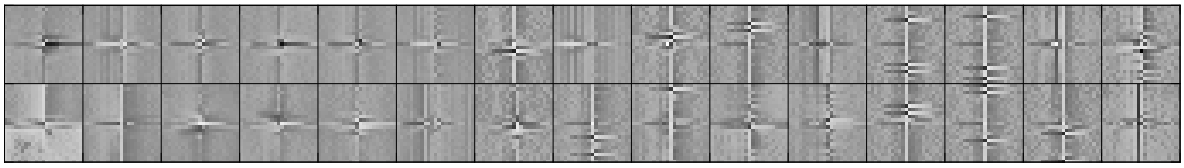
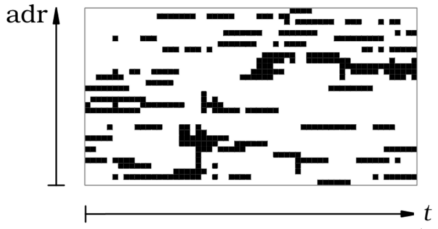
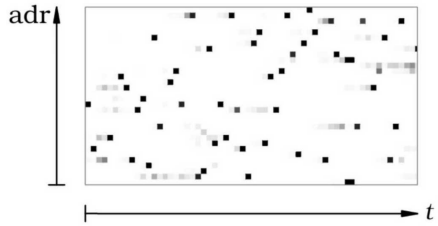


Figure 7. Map of weights for the hidden layer in the first neural network. Each square of 23 by 23 pixels represents the weights for the edges from all input nodes to one hidden node. To compute the weighted sum of inputs for a hidden node one can think of placing one of these squares on top of the bitmap of inputs, centered on the pixel of interest. Then every input value is multiplied by the value indicated by the pixel above it. Darker shades indicate negative weights and lighter shades positive weights.



(a) Input bitmap of measurements.



(b) Output map of probabilities.

Figure 6. Input and output of our artificial neural networks. The input is given by a bitmap, where black squares indicate observed cache hits. One step in the horizontal direction corresponds to the time between two measurements. The output of the first ANN are probabilities that memory accesses were actually performed by the victim process in a cache line. Higher probabilities are indicated by a darker shade. Combined with the estimation of the second network how many accesses were performed between two measurements, this gives a map where one step in the horizontal direction corresponds to one memory access in the victim.

from all input nodes and one outgoing edge to the output node. The activation functions are $\tanh$ for all hidden nodes and $1/(1+e^{-x}) - \frac{1}{2}$ for the output node. Note that the latter activation function is $\tanh$ rescaled to yield an output in the interval $[0, 1]$, which can directly be interpreted as probability. We picked the cross entropy as error function used during the training phase. This allows for a faster training than using the mean square error [39]. Further, the cross entropy also has preferable numerical properties in our specific case.

The training was done in generations. A new generation was generated by taking the best one or two networks from the previous generation (or an empty ANN at first) as a parent. Then children were created by adding a few randomly initialized hidden nodes. Sometimes also manual tweaking was necessary when weights either approached zero or very large values. Then the new generation was trained on batches from a training set of about 2^{30} samples and finally on the whole set. The L-BFGS algorithm was used for numerical optimization. This was repeated until we arrived at a network where neither manual tweaking nor adding more hidden nodes improved the error. The resulting weights for the hidden nodes are shown in Figure 7. It can be seen that the network seems to detect patterns centered on the same row or the same column as the pixel of interest. Also, the patterns most often consist of a horizontal line of pixels with the same value, starting from the left up to one distinguished pixel. This pixel is surrounded by pixels of differing value and to the right of it the line continues with pixels of this value.

The second network is used to estimate the number n of memory accesses performed between two measurements from the sum of probabilities x calculated by the first ANN for one column. For sake of brevity we skip the elaborate description of its structure, because it can easily be replaced by any other method for function approximation. It is trained to approximate the estimator $\widehat{n}(x)$ that minimizes the mean squared error $\mathbb{E}[(\widehat{n}(x) - n)^2]$ over all training samples.

4) Sources of Noise: As can be seen in Figures 6 and 7, the noise obtained from real measurements is not entirely unstructured. We now briefly explain the sources of this structure.

The vertical lines in Figure 6(a) stem from prediction logics of the cache, which detects linear memory access patterns and prefetches data accordingly. Thus, when the encryption process accesses addresses x and $x+\delta$, sometimes the cache lines containing $x + 2\delta$ and $x + 3\delta$ will be filled as well.

The horizontal lines can be explained by speculative execution. On a high level, if parts of the CPU are idle, it looks ahead in the instruction queue, and computes results in advance. Thus, the results are already available when the according instruction is to be retired and the latency of the instruction execution is hidden from the user. In this case, memory loads are issued to the cache as soon as the addresses are known and cache lines are filled. But before the data from memory can actually be used and written into an architecturally visible register, an interrupt preempts the

program. This also explains why most of the horizontal lines in Figure 6(a) end in a real memory access in Figure 6(b).

The remaining noise is due to other processes running concurrently on the same system and inaccuracy of the DoS on the scheduler.

E. Results

In the following we present measurement results that allows to assess the effectiveness of our attack in practice. Our spy process was specified to start 250 threads and to monitor 100 encryptions. The following numbers, which characterize the phases of our attack, were obtained on our test platform specified in §I-B.

- *Running time*: Performing 100 AES encryptions (i.e., encrypting 1.56 kB) takes about 10 ms on our platform. This running time blows up to 2.8 seconds, when memory accesses are monitored by the spy process. We believe that this delay is sufficiently small for the attack to go unnoticed. In fact, the user might attribute the delay to, e.g., high disk activity or network traffic.
- *Denoising*: The obtained measurements are first refined by applying our neural networks. This step approximately takes 21 seconds when running as a normal process on the target machine.
- *Preparing key search*: Next, the a posteriori probabilities of all partial key column candidates are computed by analyzing their frequencies, cf. §III-B. This step approximately takes 63 seconds.
- *Key search*: Finally, the correct key is sought as explained in §III-C. The runtime of this step varies between 30 seconds and 5 minutes, with an average of about 90 seconds.

Thus, finding the key on average takes about 3 minutes. However, if at all, the user will only notice the first few seconds, as all other processes are executed as normal processes without attacking the scheduler any more. Alternatively, the data collected in the first step could be downloaded to, and evaluated on, another machine. This data consists of one bitmap of size $2^l = 2^5 = 32$ bits for each memory access, cf. §IV-C. For each encryption 160 memory accesses are monitored. Thus, $160 \cdot 100 \cdot 32$ bits $= 62.5$ kB would have to be downloaded.

V. Extensions

In the following we show how the key search algorithm can be sped up, and how this can be used to extend our attack to other key length as well. Furthermore, we explain how the encrypted plaintext can be recovered without accessing the ciphertext at all.

A. Accelerating the Key Search

If a higher number of encryptions can be observed by the spy process, the key search of our attack can be accelerated considerably. Using the notation from §III-B this is because the peaks of the $f_i(\underline{\mathbf{k}}_i^*)$ corresponding to the true partial key column candidates become easier to separate from the floor of wrong ones. This is because the expectation value of $f_i(\underline{\mathbf{k}}_i^*)$ grows much faster than its standard deviation. Thus, after sufficiently many observations, i.e., for large N, the 9 correct candidates for each $\underline{\mathbf{k}}_i^{j^*}$ will exactly be given by the partial key column candidates with the highest frequencies.

Now, the key search algorithm from §III-C can be shortened significantly, as for each $\underline{\mathbf{k}}_i^{j^*}$ only 9 choice are left compared to $2^{4 \cdot l} = 2^{20}$ before. Having assigned values of, e.g., $\underline{\mathbf{k}}_3^{2^*}$ and $\underline{\mathbf{k}}_3^{3^*}$, there will typically be at most one possible solution for $\underline{\mathbf{k}}_2^{3^*}$ among the 9 possible values. This allows one to implement the key search in a brute force manner.

On our test environment, 300 encryptions (i.e., 4.69 kB of encrypted plaintext) are sufficient for this approach.

B. Extensions to AES-192 and AES-256

While our implementation is optimized for AES-128, the presented key search algorithm conceptually can easily be adopted for the case of AES-192 and AES-256. However, the heap used in §III becomes significantly more complex for key sizes larger than 128 bits. This problem does not occur for the key search technique presented in the previous paragraph, as its complexity is rather influenced by the number of rounds than by the size of the ciphertext. We leave it as future work to obtain practically efficient implementations of either of these two techniques.

C. Decryption without Ciphertext

In previous work it was always implicitly assumed that sniffing the network over which the ciphertext is sent is a comparatively trivial task, and thus that obtaining the key is sufficient for also recovering the plaintext. We go one step further and show how our attack can be used to also recover the plaintext without knowing the ciphertext. Because of space limitations we will only describe the plaintext recovery technique given ideal observations of the cache.

As in §III-B we assume that we have a continuous stream of cache hits/misses, without knowing where one encryption starts and the next one ends. Further, we assume that the full key K has already been recovered. We then perform the following steps to recover the plaintext without knowing the ciphertext:

- As in §III-B, we consider each of the N possible offsets in the stream of observations, and treat it as if it was the beginning of an AES round. As earlier, we use $\underline{\mathbf{x}}_i, \underline{\mathbf{y}}_i$ to denote the i^{th} column of the state matrix X before and after the round.
- For each possible number of the inner round, i.e., $j = 1, \ldots, 9$, and each column number, i.e., $i = 0, \ldots, 3$, we now solve the following equation, under the constraint that $\underline{\mathbf{x}}_i^*, \underline{\mathbf{y}}_i^*$ are equal to the observed values:

$$\underline{\mathbf{k}}_i^j = \underline{\mathbf{y}}_i \oplus M \bullet s(\underline{\tilde{\mathbf{x}}}_i) \ .$$

Enumerating all possibilities shows that this equation typically has 0 or 1 solutions, where is 0 is dominating. For each j, we consider all possibly resulting state matrices, i.e., all possible $X_j = (\underline{\mathbf{x}}_0, \underline{\mathbf{x}}_1, \underline{\mathbf{x}}_2, \underline{\mathbf{x}}_3)$.

- For each X_j, we now compute the offset at which the corresponding encryption started by just subtracting $16(j-1)$ from the current offset. Further, we compute the corresponding plaintext which can easily be done as the key is already known.
- For each of the resulting plaintexts, we now count its frequency. At some offset (namely, the correct starting point of an encryption), the correct plaintext will occur at least 9 times, whereas all other resulting plaintexts will be randomly distributed by a similar argument as in §III-A.

An ad hoc real-world implementation of this approach takes about 2 minutes to recover the plaintext of a single encryption, i.e., to reconstruct 16 B of the input. However, this must be seen as a proof of concept, which leaves much space for optimization, and which shows that it is not necessary to know the ciphertext to recover both, the key *and* the plaintext.

VI. Countermeasures

In the following we discuss mitigation strategies against our attack, which we believe to be practical. They either get rid of information leakage entirely, or at least limit leakage to an extent which renders our attack impossible. For an extensive list of countermeasures against access-driven cache attacks we refer to [7].

A. Generic Countermeasures

Let us describe two generic countermeasures against access-based cache attacks, which seem to be reasonably efficient.

First, the OS could be adapted such that it offers the possibility of pre-loading certain data each time a certain process is activated. If, in our case, the lookup table $T[x]$ would be pre-loaded, a spy process would only see cache hits, and could not infer any information about the secret key. However, such a pre-loading mechanism only seems to be reasonable if the lookup table is sufficiently small, such as 2 kB in our situation. For lookup tables used in asymmetric cryptography this is often not the case. Also, the implementation of this feature might require substantial work on the kernels of current operating systems.

Alternatively, the task scheduler could itself be hardened against our (and similar) attacks. Namely, one could limit the minimum time period between two context switches to, e.g., $500\mu s$. While such a bound is small enough to keep the system responsive, denial of service attacks on the scheduler similar to ours would no longer work.

B. Countermeasures for AES

One concrete mitigation strategy, which defeats our attack has been realized in OpenSSL 1.0 [19]. In that implementation only the substitution tables S is stored, which contains 2^8 entries of 1 byte each. Thus, on standard x86 architectures with a cache line size of 2^6 bytes we have that only $l = 2$ bits of each $\underline{\mathbf{x}}_i^*$ are leaked. Looking at Table 1 now shows that we have $p_3 = 1$, i.e., every $\underline{\mathbf{k}}_i^* \in \{0,1\}^{4\cdot 2}$ is a valid partial key column candidate for every $\underline{\mathbf{x}}_i^*$ and $\underline{\mathbf{y}}_i^*$. For this reason, our key search algorithm does not work anymore.

This mitigation strategy prevents our specific attack, but it does not eliminate the problem completely, because it still leaks information. Since AES was not designed to be secure in a threat model where an attacker is able to learn any bits of the $\underline{\mathbf{x}}_i$, the implementation remains, at least potentially, attackable. For instance, it might be possible to combine information leaking in three or more rounds to infer possible configurations of the lower, not directly leaking bits.

Finally, because of the prevalence and importance of AES, we increasingly see hardware implementation of AES, which render access-driven cache attacks impossible [26], [40], [41].

VII. Conclusion and Outlook

The most obvious limitation of our attack is that it targets a very specific software and hardware configuration (i.e., Linux OS, OpenSSL 0.9.8n, and single core x86 CPUs). In the following we discuss in how far our attack can be made more generic by extending it to a broader set of systems. In general, we believe that it is very hard to render the attack to be inherently generic, since it is sensitive to exact parameters of the target system. In fact, we have put substantial effort into fine tuning the attack, and have also experienced that minor updates of the operating system required substantial adjustments.

Yet we believe that the attack might be extended to other operating systems and multi core CPUs. To port the attack to other operating system, an intimate knowledge of task scheduler mechanics is needed to implement a successful DoS attack. According to our experience with the Linux kernel, the scheduler is a moving target, subtly changing from one release to another. With Linux understanding the scheduler and these changes is a surmountable task, as it is open source software. For a closed source operating system, one would have to reverse engineer the scheduler first. This of course tremendously increases the effort going into the attack, but does not make it impossible.

We believe our results are relevant for modern CPUs which typically have more than one core. Multiple cores clearly complicate the attack on the task scheduler, because the fundamental assumption that the victim is preempted while the spy process is running is not satisfied any longer. Yet, we believe that a dedicated attacker might overcome this problem in certain cases. For example with CPU affinity

(either by explicitly using the OS's APIs or by deceiving the OS's scheduler heuristics) the attacker can influence where the victim will run and gain nearly the same amount of control as it had in the single core scenario. The spy needs to be distributed across multiple cores and has to perform sophisticated synchronization. This is certainly a difficult but not unsurmountable task. On the other hand, multi core CPUs give the opportunity to an OS to defeat such attacks by dedicating a whole CPU core to sensitive processes. That core could even be isolated by not sharing its cache with other cores.

Finally, as cloud computing and virtualization are becoming more and more prevalent, investigating to which extent these systems are vulnerable to cache-based side channel attacks is an interesting open problem. Ristenpart et al. [42] demonstrate the existence of a cache-based side channel between virtual machines in Amazon's EC2 cloud computing service. It seems very likely that their attack can be improved significantly, e.g., by abusing mechanisms like kernel same-page mapping (KSM) in Linux KVM, where an attacker can share memory with a victim across virtual machine boundaries and mount a powerful attack through that channel.

Acknowledgments

We would like to thank Billy Brumley, the anonymous reviewers and our shepherd, Adrian Perrig, for their valuable comments and support.

REFERENCES

[1] J.-F. Gallais, I. Kizhvatov, and M. Tunstall, "Improved trace-driven cache-collision attacks against embedded AES implementations," in *WISA '10*, ser. LNCS, Y. Chung and M. Yung, Eds., vol. 6513. Springer, 2010, pp. 243–257.

[2] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, "Cryptanalysis of DES implemented on computers with cache," in *CHES '03*, ser. LNCS, C. D. Walter, Ç. Koç, and C. Paar, Eds., vol. 2779. Springer, 2003, pp. 62–76.

[3] B. Brumley and R. Hakala, "Cache-timing template attacks," in *ASIACRYPT '09*, ser. LNCS, S. Halevi, Ed., vol. 5677. Springer, 2009, pp. 667–684.

[4] C. Percival, "Cache missing for fun and profit," http://www.daemonology.net/hyperthreading-considered-harmful/, 2005.

[5] O. Acıiçmez, B. Brumley, and P. Grabher, "New results on instruction cache attacks," in *CHES 2010*, ser. LNCS, S. Mangard and F.-X. Standaert, Eds., vol. 6225. Springer, 2010, pp. 110–124.

[6] M. Neve and J.-P. Seifert, "Advances on access-driven cache attacks on AES," in *SAC '06*, ser. LNCS, E. Biham and A. M. Youssef, Eds., vol. 4356. Springer, 2006, pp. 147–162.

[7] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on AES, and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, 2010.

[8] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *CT-RSA '06*, ser. LNCS, D. Pointcheval, Ed., vol. 3860. Springer, 2006, pp. 1–20.

[9] O. Acıiçmez, W. Schindler, and Ç. Koç, "Cache based remote timing attack on the AES," in *CT-RSA '07*, ser. LNCS, M. Abe, Ed., vol. 4377. Springer, 2007, pp. 271–286.

[10] D. J. Bernstein, "Cache-timing attacks on AES," http://cr.yp.to/papers.html, 2004, University of Illinois, Chicago, US.

[11] M. Neve, J.-P. Seifert, and Z. Wang, "A refined look at Bernstein's AES side-channel analysis," in *ASIACCS '06*, F.-C. Lin, D.-T. Lee, B.-S. Lin, S. Shieh, and S. Jajodia, Eds. ACM, 2006, p. 369.

[12] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *CHES '06*, ser. LNCS, L. Goubin and M. Matsui, Eds., vol. 4249. Springe, 2006, pp. 201–215.

[13] O. Acıiçmez and Ç. Koç, "Trace-driven cache attacks on AES," Cryptology ePrint Archive, Report 2006/138, 2006.

[14] X. Zhao and T. Wang, "Improved cache trace attack on AES and CLEFIA by considering cache miss and S-box misalignment," Cryptology ePrint Archive, Report 2010/056, 2010.

[15] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo, "AES power attack based on induced cache miss and countermeasure," in *ITCC '05*. IEEE Computer Society, 2005, pp. 586–591.

[16] C. Lauradoux, "Collision attacks on processors with cache and countermeasures," in *WEWoRC '05*, ser. LNI, C. Wolf, S. Lucks, and P.-W. Yau, Eds., vol. 74. GI, 2005, pp. 76–85.

[17] J. Daemen and V. Rijmen, "AES proposal: Rijndael," AES Algorithm Submission, 1999.

[18] FIPS, *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology, 2001, http://csrc.nist.gov/publications/fips/. Federal Information Processing Standard 197.

[19] OpenSSL, "OpenSSL: The Open Source toolkit for SSL/TSL," http://www.openssl.org/, 1998–2010.

[20] P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *CRYPTO '96*, ser. LNCS, N. Koblitz, Ed., vol. 1109. Springer, 1996, pp. 104–113.

[21] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," *Journal of Computer Security*, vol. 8, no. 2/3, pp. 141–158, 2000.

[22] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel," Department of Computer Science, University of Bristol, Tech. Rep. CSTR-02-003, June 2002.

[23] ——, "Defending against cache based side-channel attacks," *Information Security Technical Report*, vol. 8, no. 1, pp. 30–44, April 2003.

[24] K. Tiri, O. Acıiçmez, M. Neve, and F. Andersen, "An analytical model for time-driven cache attacks," in *FSE '07*, ser. LNCS, A. Biryukov, Ed., vol. 4593. Springer, 2007, pp. 399–413.

[25] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert, "Software mitigations to hedge AES against cache-based software side channel vulnerabilities," Cryptology ePrint Archive, Report 2006/052, 2006.

[26] S. Gueron, "Advanced Encryption Standard (AES) instructions set," www.intel.com/Assets/PDF/manual/323641.pdf, 2008, Intel Corporation.

[27] R. Könighofer, "A fast and cache-timing resistant implementation of the AES," in *CT-RSA '08*, ser. LNCS, T. Malkin, Ed., vol. 4964. Springer, 2008, pp. 187–202.

[28] "Intel 64 and IA-32 architectures optimization reference manual," http://www.intel.com/Assets/PDF/manual/248966.pdf, 2010, Intel Corporation.

[29] "Intel 64 and IA-32 architectures software developer's manual. Volume 3A: System Programming Guide, Part 1," http://www.intel.com/Assets/PDF/manual/253668.pdf, 2010, Intel Corporation.

[30] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[31] V. Rijmen, A. Bosselaers, and P. Barreto, "Optimised ANSI C code for the Rijndael cipher," http://fastcrypto.org/front/misc/rijndael-alg-fst.c, 2000.

[32] M. Bayes, "An essay towards solving a problem in the doctrine of chances," *Philosophical Transactions*, vol. 53, pp. 370–418, 1763.

[33] J. Bernardo and A. Smith, *Bayesian Theory*. Wiley, 1994.

[34] I. Molnár, "Design of the CFS scheduler," http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt, 2007, Redhat.

[35] D. Tsafrir, Y. Etsion, and D. Feitelson, "Secretly monopolizing the cpu without superuser privileges," in *USENIX Security '07*. USENIX, 2007, pp. 1–18.

[36] M. Jordan and C. Bishop, "Neural networks," *ACM Computing Surveys*, vol. 28, no. 1, pp. 73–75, 1996.

[37] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–113, 1943.

[38] C. Bishop, *Neural Networks for Pattern Recognition*. Oxford University Press, 1996.

[39] P. Simard, D. Steinkraus, and J. Platt, "Best practices for convolutional neural networks applied to visual document analysis," in *ICDAR '03*. IEEE Computer Society, 2003, pp. 958–962.

[40] P. Ghewari, J. Patil, and A. Chougule, "Efficient hardware design and implementation of AES cryptosystem," *International Journal of Engineering Science and Technology*, vol. 2, no. 3, pp. 213–219, 2010.

[41] M. Mali, F. Novak, and A. Biasizzo, "Hardware implementation of AES algorithm," *Journal of Electrical Engineering*, vol. 56, no. 9-10, pp. 265–269, 2005.

[42] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds," in *CCS '09*, S. Jha and A. Keromytis, Eds. ACM Press, 2009, pp. 199–212.