Dogo Rangsang Research Journal
ISSN : 2347-7180

UGC Care Group I Journal
Vol-08 Issue-14 No. 01 February : 2021

Access-Based Cache Attacks on AES Practiced with Cache Games

Mr.ALOK KUMAR PATTNAIK*, Mrs.PRAGYAN PARAMITA PANDA
Dept. OF Computer Science and Engineering, NIT , BBSR
alokkumar@thenalanda.com*, pragyanparamita@thenalanda.com

Abstract— **Instead of relying on a scheme's theoretical flaws, side channel attacks on cryptographic systems take advantage of information obtained via physical implementations. Much progress was made, in particular, during the past few years for the class of access-driven cache-attacks. The locations of memory accesses made by a victim process are the source of information leaking for such attacks. In this study, we evaluate the AES situation and provide an attack that can practically instantly decrypt AES-128 with just a handful of observed encryptions while still retrieving the entire secret key. Unlike the majority of prior assaults, ours doesn't require any knowledge of the plaintext or the ciphertext (such as its distribution, etc.). Additionally, we demonstrate for the first time how to recover the plaintext without having access to the ciphertext. Furthermore, a non-privileged user account can be used to run our spy process. It is the first functional attack for implementations employing compressed tables, where it is no longer possible to determine the start of AES rounds. for all successful prior assaults. Our attack's outcomes are all supported by a fully functional implementation, not just by theoretical arguments or computer simulations. A contribution that is possibly of independent relevance is a denial-of-service assault on the present Linux scheduler (CFS), which enables innovative, high-precision monitoring of memory accesses. Lastly, we provide some generalisations of our assault and make some suggestions for potential defences against it.**
***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 avoid design weaknesses and bugs. However, theoretical soundness of a scheme is not necessarily 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, e.g., [1]–[15]. These cache based side-channel attacks (or *cache attacks* for short) split into three types: *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 [8]–[11]. From these timings he can make inferences about the overall number of cache hits and misses during an encryption. On the other hand, 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 [12]–[15]. Finally, access-driven attacks additionally enable the adversary to determine the cache sets accessed by the cipher [4]–[6]. Therefore, he can infere which elements of, e.g., a lookup table have been accessed by the cipher.

Using the fact that accessing data which has already been copied into the cache is up to two orders of magnitude faster than accessing data in the main memory, access-driven attacks roughly work as follows: assume two concurrently running processes (a *spy process* S and a cryptographic *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) [16], [17], one of the most popular symmetric-key block-ciphers. On a high-level the main properties of our attack are two-fold: first, our attack works under very weak assumptions, and is thus the strongest working access-driven attack currently known. Second, we provide a concrete and practically usable implementation of the attack, based on new techniques. We also resolve a series of so far open issues and technicalities.

Let us describe these properties and the underlying technical contributions in more detail. In fact, for our attack to work we only need to assume that the attacker has a test

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

Copyright @ 2021 Authors

machine at his disposal prior to the attack, which is identical to the victim machine. The test machine is used to carry out an offline learning phase which consists of about 168′000 encryptions. Further, two artificial neural network have to be trained on an arbitrary platform.

To carry out the attack all we need to be able to is 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 don't require any explicit interactions, such as interprocess communication or I/O. Osvik et al. [6], [7] refer to attacks in this setting as *asynchronous attacks*.

Our attack technique has the following features:

- In contrast to previous work [5]–[7], 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 [18]. When using compressed tables, the first, respectively 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 implementation of OpenSSL 0.9.8n on Linux. It is highly efficient and is able to recover keys in "realtime". More precisely, it consists of two phases: in an oberservation phase, which lasts about 2.8 seconds on our test machine, about 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 novelly 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.
- Finally, we also describe a novel approach to reconstruct the AES key from the leaked key bits leaked.

To emphasize the practicability of our attack, let us sketch how it could be used to mount a targeted malware attack (which are currently regularly seen in practice). In a targeted attack we may assume that the attacker knows the configuration of the victim machine and can thus obtain an identical machine, which is one of our preconditions. Next, standard malware infection techniques can be used to compromise a non-privileged account and to deploy our custom payload on the victim machine. Since we only need to observe around 100 encryptions, which correspond to 1.56 kilobytes of ciphertext, we are able to recover the secret-key almost immediately whenever AES is being used on the victim machine after a successful infection.

B. Related Work

It was first mentioned in [19], [20] that the cache behavior potentially poses a security threat. The first formal studies of such attacks were given by Page [21], [22].

First practical results for time-driven cache attacks on the Data Encryption Standard (DES) were given in [2], and an adoption for AES was mentioned without giving details. Differently efficient time-driven attacks on AES can be found in [6]–[11], some of which require that the first respectively last round of AES can be identified. In [23], an analytical model for forecasting the security of symmetric ciphers against such attacks is proposed.

Trace-driven cache attacks were first described in [21]. Other such attacks on AES can be found, e.g., in [12]–[15]. Especially, [12] proposes 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 pioneered by Osvik et al. [6], [7]. They describe various attack techniques and implementations in what they call the *synchronous model*. The synchronous model makes rather *strong* assumptions on the capabilities on an attacker, i.e., it assumes that an attacker has the ability to trigger an encryption 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 multi-threaded CPU. Also, they implement and perform some measurements on multi-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. [5] picks up. They make advances towards realizing asynchronous attacks. To this end they describe and implement a spy process that

Copyright @ 2021 Authors

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.

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 mitigations strategies for AES have been proposed, e.g., [24]–[26].

C. Organization of this Document

In §II we briefly recapitulate the structure of a CPU cache, and the logics 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 was able to perfectly observe the single cache-accesses performed by the victim process. We drop this idealization in §IV and show how by combining a novel attack on the task scheduler and neural networks sufficiently good measurements can be obtained in practice. We give some real measurements and extensions of our attack in §V. In particular, we there sketch how to obtain the plaintext without knowing the ciphertext. Finally, we propose some countermeasures in §VI.

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

We next 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 [27]. Its size typically ranges from some hundred kilobytes up to a few megabytes.

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 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, which are accompanied by 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 $A = (A_{max}, \dots, A_0)$, 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., $(A_{a+b-1}, \dots, A_b)$ identify the cache-set. The remaining bits, i.e., $(A_{max}, \dots, A_{a+b})$ serve as a tag. Now, when requesting data from some address A, the cache logic compares the tag corresponding to A with all tags of the (unique) possible 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 $max = 63$ onx64-architectures (however, the usage of physical address extension techniques may increase the value of max [28]).

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 cacheline 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 multi-tasking 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.

[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 pathological and 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.

Copyright @ 2021 Authors

- 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 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) [16] is a symmetric block cipher, and has been adopted as an encryption standard by the U.S. government [17]. For self-containment, and to fix notation, we next recapitulate the steps of the AES algorithm [29, §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 [17] 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 byte-wise 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* given in [17]. 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 ith 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:

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

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 the basic ideas how to efficiently implement AES. These tricks are corner stones for 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 [30] reformulates AES to only need basic

Copyright @ 2021 Authors

operations on 32-bit machine words. The implementation of OpenSSL [18] 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, using X and Y for the state matrices before and after the round, and K to denote the round key. That is, we have the following relation:

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

Consequently, we have

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

and similarly for $\underline{\mathbf{y}}_1, \underline{\mathbf{y}}_2, \underline{\mathbf{y}}_3$, where $\underline{\mathbf{k}}_0$ denotes the first column of K when interpreting K as a matrix of 4×4 bytes, 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

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

This allows one to rewrite (3) to the following form which only uses 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 4kB of memory. However, they are highly redundant. For instance, we have that

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

where $\gg$ 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 2kB 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 $M \cdot s(\tilde{X})$. Alternatively, one can reuse the existing lookup table(s) by accessing them at positions that have been multiplied by a 1-entry of M. 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, the attacks of, e.g., [5], [6] cannot be executed any more. In particular this is the case for the compressed tables implementation of OpenSSL 0.9.8.

D. *Test Environment*

All our implementations and measurements have been obtained on a Intel Pentium M 1.5GHz (codename "Banias") processor, in combination with an Intel ICH4-M (codename "Odem") chipset using 512MB 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.

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 (over-restrictive) 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, we 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 $b > 6$, as less information about the secret index leaks through the cache accesses in this case.

With this notation, and (2), it is now easy to see that the following equations are satisfied:

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

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

Copyright @ 2021 Authors

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

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 $\mathbf{x}_i^*$ and $\mathbf{y}_i^*$ are known.

(4). These sets can be computed by enumerating all 2^{32-4l} possible values of $\tilde{\mathbf{x}}_i$.

The cardinality of K_i turns out not to be constant for all $\tilde{\mathbf{x}}_i^*$. However, we will need to argue about the probability that some $\tilde{\mathbf{x}}_i^*$ is a partial key-column candidate. We therefore compute the expected cardinality of the K_i by assuming that the $\tilde{\mathbf{x}}_i^*$ are equally distributed in $\{0,1\}^{4l}$. 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 K_i for all possible values of l. The last column of the table gives the probability p_l that a random $\mathbf{k}_i^* \in \{0,1\}^{4l}$ is a partial key-column candidate for a random $\tilde{\mathbf{x}}_i^*$. One can see that for $1 \leq l \leq 3$ every $\tilde{\mathbf{x}}_i^*$ can be completed to a $\tilde{\mathbf{x}}_i$ satisfying (4). Thus, in this case, this approach does not yield any information about the secret key K. The contrary extreme happens if we have $l = 8$, i.e., if we can monitor the exact entries of the lookup table accessed by the victim process. In this case, we can recover the key only from the states X, Y of two consecutive rounds. The interesting case is where $3 < l < 8$ where we can 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, as for implementations of AES using compressed tables it is hard to precisely determine where one round stops 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 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 K_i of partial key-column candidates for $i = 0, \ldots, 3$, and count how often each possible $\mathbf{k}_i^* \in \{0,1\}^{4l}$ also satisfies $\mathbf{k}_i^* \in K_i$. We denote this frequency by $f_i(\mathbf{k}_i^*)$. By the former remark, and because of the 31 last monitored memory accesses, we have enough observations to complete (4) for any of these N offsets.

A simple observation yields that $\mathbf{k}_i^*$ is an element of K_i at least $z_{\mathbf{k}_i^*}M$ times, if $\mathbf{k}_i^*$ is the truncated part of the correct i^{th} column of a round key in $z_{\mathbf{k}_i^*}$ different rounds. Put differently, we have $f_i(\mathbf{k}_i^*) \geq z_{\mathbf{k}_i^*}M$. For each of the remaining $N - z_{\mathbf{k}_i^*}M$ wrong starting points we may assume that $\mathbf{k}_i$ occurs in K_i with probability p_l. This is, because solving (4) for wrong values of $\mathbf{x}_i^*, \mathbf{y}_i^*$ should not leak any information about the correct key, even if $\mathbf{x}_i^*, \mathbf{y}_i^*$ are not fully random, but overlapping parts of correct values from subsequent rounds. As in the previous section, this assumption experimentally 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 $f_i(\mathbf{k}_i^*)$ as follows:

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

From these equations one can see that every $\mathbf{k}_i^*$ 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 $\mathbf{k}_i^*$ which do not occur in a round key. This difference grows linearly in the number M of observed encryptions. On the other hand, the standard deviation $\sigma[f_i(\mathbf{k}_i^*)] = \sqrt{V[f_i(\mathbf{k}_i^*)]}$ only grows like the square root of M (note here, that N is a fixed constant times M). Thus, for

Copyright @ 2021 Authors

an increasing number of encryptions, the peaks will become better to separate from the floor.

Using the $f_i(\underline{\mathbf{k}}_i^*)$ and the Bayes Theorem [31], [32] 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 the victim process.

C. Key Search

As stated in the previous section, key candidates $\underline{\mathbf{k}}_i^*$ will experience a peak in the frequency table if they are part of the correct key schedule, while all other $\underline{\mathbf{k}}_i^*$ will not do so with high probability. Next, we are therefore concerned with assigning scores to sets of partial key-column candidates, and to search for sets with high scores, which then are very likely to stem from the original key.

We chose the mean log-probability to assign a score to a set of candidates. That is, for a set S of candidates, the score is computed as follows:

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

We assume that every element in such a set S is tagged with the position of the key schedule it is a candidate for.

We now iteratively 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. Then, whenever adding a further part of the key, the key schedule of AES also forces one to fix some other parts, cf. Figure 2 (there, the $\underline{\mathbf{t}}^i$ denote temporary variables without further semantics). Our algorithm now iteratively repeats these steps until a K implying a 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 ith round key. Therefore, we initialize a heap containing singletons for all possible values of $\underline{\mathbf{k}}_3^{i*}$, sorted by their score $h(\{\underline{\mathbf{k}}_3^{i*}\})$.
- 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 $\underline{\mathbf{t}}_2^{i+1}$ 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 S of the heap already contains a candidate for $\underline{\mathbf{k}}_3^{i+3*}$, we compute the combinations $S \uplus \{\underline{\mathbf{k}}_3^{i+4*}\}$ 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 S in Figure 2 (there, we indicated the case $i = 2$). If this is not the case, we discard $S \uplus \{\underline{\mathbf{k}}_3^{i+4*}\}$, 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 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 $\underline{\mathbf{k}}_j^i$ in a line allows to fill the complete key schedule. Given S, we already fixed $4 \cdot 4 \cdot l = 80$ bits of such a "line". Further, the solving the nonlinearities in the key schedule yields 24 more bits thereof. 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 S in the key schedule. Note here that typically there is at most 1 solution of the non-linearities for a given position. In the rare case that there are more than 1 solutions, we perform the following steps for either of them.
  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.
- Now, we continue processing the heap until the topmost element of the heap has a smaller score than the stored full key. In this case, we output the stored key as the secret key and quit. This is, because typically the score of sets will decrease when extending the set, because even when adding a candidate with very high score, most often other parts with worse score also have to be added because of 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 S with some fixed "base line" has the same score $h(S)$ 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 this over-restrictive assumption can be dropped. We therefore first describe the way the task scheduler of modern Linux kernel works, and explain how its behavior can be exploited for our

Copyright @ 2021 Authors

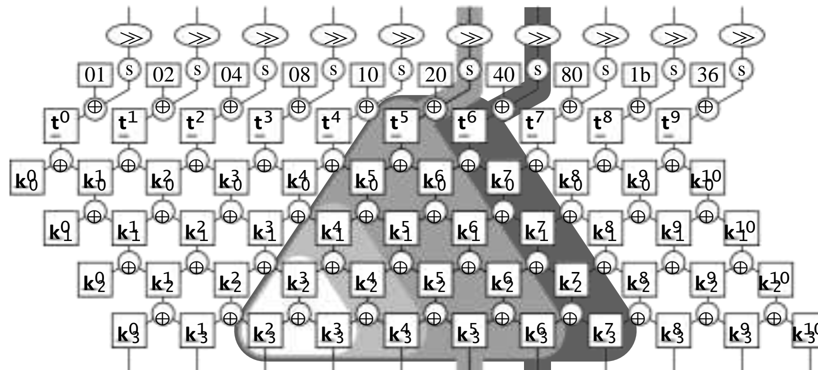


Figure 2. Key schedule of the 128 bit variant of AES. By $\underline{\mathbf{k}}^n_m$ we denote the mth column of the nth round key. The $\mathbf{t}^n$ are temporary variables without any further semantics. The bottommost elements, i.e., $\underline{\mathbf{k}}^i_3$, are passed as inputs to the nonlinearity 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.

purposes. We then briefly recapitulate the concept of neural networks, and show how they can serve an attacker to treat inaccurate measurements.

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 processes (i.e. transisition from running to ready) and which process is next to be *activated* when a CPU becomes idle (i.e. transition 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. not waste time on overhead like context switching and scheduling decisions) and
- *minimizing latency* of reactions to external events.

Starting from Linux kernel 2.6.23, all Linux systems are equipped with the *Completely Fair Scheduler* (or *CFS*) [33], 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 truely 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: only the running

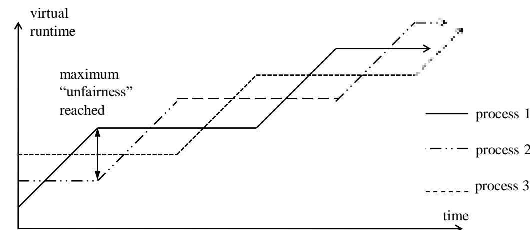


Figure 3. Functioning of the Completely Fair Scheduler. Here, three process are running concurrently. After process 1 was the assigned the CPU for some time, process 2 is the next to be activated to keep the unfairness among the different processes smaller than some threshold.

process' 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 the difference between the rightmost and leftmost entries be $\Delta\tau = \tau_{right} - \tau_{left}$. This difference in virtual runtime of the most and least favourable 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: $\Delta\tau \leq \Delta\tau_{max}$. It always selects the leftmost process to be activated next and preempts the running rightmost process when further execution would exceed this maximum unfairness $\Delta\tau_{max}$.

This logic is illustrated in Figure 3. There, 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 $\Delta\tau_{max}$. 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 the virtual runtime $\tau_{unblock}$ of unblocking processes is initialized. It is desirable that such a process is activated as soon as possible and given

Copyright @ 2021 Authors

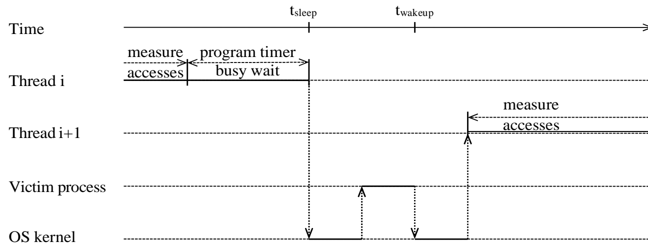


Figure 4. Sequence diagram of the denial of service attack on CFS. Multiple threads run alternatingly and only leave very small periods of time to the victim process.

enough time to react to the event it was waiting for with low latency. The concept of treating a process very favourably by the scheduler in this situation is called *sleeper fairness*. In CFS terms this means assigning it the lowest possible virtual runtime while not violating CFS' invariants: To not exceed the maximum unfairness $\tau_{unblock} \leq \tau_{right} - \Delta\tau_{max}$ must hold. Also, virtual runtime must not decrease across blocking and unblocking to prevent a trivial subversion of CFS' strategy. Therefore the virtual runtime τ_{block} a process had when it blocked needs to be remembered and serves as another lower bound. Finally, we get

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

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

B. A Denial of Service Attack for CFS

We will now show how the fairness conditions, and in particular the sleeper fairness, of CFS can be exploited by an attacker. On a high level, the idea is the following: the spy process S requests most the available CPU time, and only leaves very small intervals to the victim process V . By choosing the parameters of S appropriately, the victim process will, on average, only be able to advance by one memory access 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. After that V is again allowed to run for "a few" CPU cycles, and V measures again, etc.

In this section, we will describe the underlying denial of service (DoS) attack on CFS. The procedure for measuring cache access can be found in §V-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 sufficiently long. 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 access were performed by V since the previous measurement.
- It then computes t_{sleep} and t_{wakeup}, which designate the points in time when thread i should block and thread $i+1$ should unblock, respectively. 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 yield the CPU.

During the time when no spy thread runs, the kernel first activates the victim process (or some other process running concurrently on the system). This process is allowed to execute until the timer expires which unblocks thread $i+1$. Because of the large number of threads and the order they run, their virtual runtimes will only increase very slowly. While a thread's virtual runtime is kept sufficiently low in this way, it will be the leftmost in the timeline when it unblocks and immediately preempt the currently running process. This mechanism ensures that S immediately regains control of the CPU after V ran.

Typically, $t_{wakeup} - t_{sleep}$ 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 operating system (in ns 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 0.6672366819ns per CPU cycle. At the start of the attack, the offset of the time-stamp-counter to the OS time is measured, so we are able to convert time measured by `rdtsc` to OS time with very high accuracy. Note that since newer Linux versions change the cpu clock to save power when the idle thread runs, a dummy process with very low priority is lauched to prevent the idle thread from changing the linear relationship between OS time and time-stamp-counter.

Even with exact computation 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 $t_{wakeup} - t_{sleep}$ according to the rate of observed memory accesses. And second, the fact that the clock and timer devices don't actually operate with nanosecond accuracy, as their API may suggest. In our hardware setting, the actual time when the timer expires lies in an interval of about ± 100

Copyright @ 2021 Authors

```
#define CACHELINESIZE 64
#define THRESHOLD 200
unsigned measureflush(void *table,
            size_t tablesize, uint8_t *bitmap) {
 size_t i;
 uint32_t t1, t2;
 unsigned bit, 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" :
           "=a"(t2),
           "=D"(t1) :
           "S"((const char *)table +
                        CACHELINESIZE * i) :
           "ebx", "ecx", "edx", "cc");
  bit = (t2 - t1 < THRESHOLD) ? 1 : 0;
  n_hits += bit;
  bitmap[i/8] &= ~(1 << (i%8));
  bitmap[i/8] |= bit << (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.

machine cycles around t_{wakeup}. In theory, this could also be compensated with a more complex computational model of the hardware. In practice, just assuming a linear relationship between TSC and OS time is sufficient for our purposes.

To hide the spy process from the user t_{wakeup} t_{sleep} is 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 the spy waits for a victim to start and after the victim terminates.

Remark: Note that in spirit our DoS attack is similar to that in [34]. 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 the attack in [34].

C. Testing for Cache Accesses

While in the previous section 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 will now 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 want to discuss in detail. On a high level, it measures the time needed for each memory access into the lookup table, and infers whether this data had already been in the cache before or not.

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 seperated 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 used 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 sideeffect of the latter: namely, `cpuid` is a serializing instruction, i.e., it logically seperates 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 part of this time stamp is discarded, and the least significant part (which is stored in `%eax`) is moved to `%edi` to preserve it 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 timestamp is stored to `%eax`, and the accessed data is flushed from the cache again by flushing its cacheline using `clflush(%%esi)`.

Outputs: In each iteration, two outputs are handed back to the routine. 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 implicitely 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 (`bit=1`) or not (`bit=0`). 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.

Copyright @ 2021 Authors

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 respectively misses occured, 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 by noise. This is because not only the victim and the spy process, but also other processes running concurrently on the same system, perform memory accesses, which can cause wrong 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 logics 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 (neural networks, ANNs), the functioning and training of which we explain next.

1) Introduction to Artificial Neural Networks: On a very high level, an artifical neural network [35]–[38] is a computational model processing data. Typical applications are the approximation of probability distributions, pattern recognition, and classification of data.

A neural network can be conceived as a directed graph with a value being attached to each of its nodes. Some of the nodes are labeled as *input* respectively *output* nodes. Except for the input nodes, the values of all nodes are computed from the values attached to its predecessors.

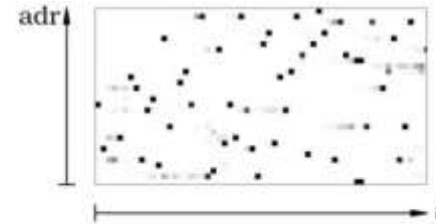
For a node v, let $u_1, \ldots, u_m$ be its predecessors, and let X_v, X_{u_i} denote the variables attached to these nodes. Then X_v is computed as

$$X_v = \sigma(w_0 + \sum_{i=1}^{m} w_i X_{u_i}) \ .$$

for some $w_i \in \mathbb{R}$ and a (typically nonlinear) *activation function* σ.

Before being deployed in practice, artificial neural networks typically have to undergo a *training phase*, where the weights w_i are adjusted using *back-propagation* techniques. This can be done by first assigning randomly distributed values to the w_i, and by then testing the neural network on some inputs samples, for which a target output is known. In the simplest case, the error of the output is then numerically differentiated with respect to the weights, and the weights are brought into the opposite direction of the obtained gradient by subtracting a ratio of it. These steps are iteratively executed until the quality of the neural network is good enough, i.e., until the error is sufficiently small.

(a) Input of the neural network.



(b) Output of the neural network.

Figure 6. Input and output of our artificial neural networks. The input is given by a bitmap, where white squares indicate observed cache hits. The ANN filters out noise, and outputs probabilities that memory accesses were actually performed by the victim process. The darker the square, the higher is the probability that at the given point in time the specified memory location was accessed.

More advanced techniques than just subtracting a ratio of the gradient contain, e.g., the so-called L-BFGS method [39].

We refer to [37], [38] for detailed discussions of artificial neural networks.

2) Inputs and Outputs of Our ANN: The first of your two neural networks outputs the probability that at some given point t_0 in time, a memory access was performed by the victim process at memory location adr_0. It therefore takes as input a segment of the bitmap obtained using the techniques presented in §IV-C. This bitmap contains a 1 whenever a cache hit was observed for a certain memory location and point in time, and a 0 otherwise.

For (t_0, adr_0), this segment of the bitmap consists of all observations $(t_{-11}, \mathsf{adr}_{-11})$ up to $(t_{11}, \mathsf{adr}_{11})$. That is, all cache hits/misses monitored up to 11 activations of the victim process before and after the questionable point in time are given as inputs to the ANN for the memory locations close to the questionable one. If parts of the resulting square of monitored cache hits/misses do not exist, e.g., because the associated addresses are outside of the lookup table, they are filled with all zeros.

Figure 6 shows the inputs and outputs of a well-trained artificial neural networks. The inputs are given by a bitmap which indicates at which points in time which memory accesses resulted in cache hits or misses. This bitmap is obtained by using the algorithm described in §IV-C. Because of the size of the lookup table (2kB) and the size of each cache line (2^6 = 64B), 32 addresses have to be considered.

Copyright @ 2021 Authors

Now, the neural network processes all cache hits in any of these 32 addresses for any given point in time. The output is shown in Figure 6(b): there, a dark square at (t_0, adr_0) indicates that the victim process accessed adr_0 at t_0 with very high probability, whereas light squares indicate that the corresponding memory location has not been accessed with high probability. As one can see, on average the victim process only advances by 1 memory access on each activation (there is one black square per output column), which emphasizes the high precision of the denial of service attack presented in §V-B.

The example shows 61 activations, corresponding to nearly 4 rounds, of AES. Among the results, there are 2 wrong positives, and 3 undetected memory accesses. However, this precision experimentically turned out to be sufficient for the ideal attack described in §III.

We then use a second ANN to estimate how many memory access the victim process performed at activation t_0. This is important for accurately estimating the timeline of memory accesses. In particular, two sources of inexactness are detected. First, as always 8 table entries are loaded into the cache at a time, the victim process may advance by more than 1 table lookup in the case that subsequent lookup indices map into the same cache line. This phenomenon is not detectable by the techniques from §V-C, as it only causes one single cache hit. Second, sometimes some other process than the victim process may be activated between two activations of the spy process and thus no table lookup is performed at all. Also, in rare cases, more than one memory access performed by the victim process can be retired because of inaccuracies in the parameters of the scheduler DoS attack.

The input of this second ANN is the sum of outputs for one column of the first ANN. That is, for a column corresponding to some time t_0, the input is given by

$$x_{t_0} = \sum_{i=0}^{31} y_{(t_0,\mathsf{adr}_i)},$$

and the output is given by the expected number of lookup table accesses at time t_0. By $y_{(t_0,\mathsf{adr}_i)}$ we denote the output of the first ANN at time t_0 and memory location adr_i.

3) Parameters of the Neural Network:
We now briefly describe the parameters of the ANN we use to obtain whether or not a cache hit occurred at (t_0, adr_0).

Besides the input and the output layer, our neural network has one more hidden layer. In a first step, the $(11+1+11)^2 = 529$ bits input bits of the first ANN are first combined linearly, and then transformed into 23 hidden nodes using the nonlinear function

$$\sigma_{11}(x) = \frac{1}{1+e^{-x}} - \frac{1}{2} = \frac{1}{2}\tanh\left(\frac{x}{2}\right) .$$

These intermediate results are again combined linearly, and then manipulated using the function

$$\sigma_{12}(x) = \frac{1}{1+e^{-x}},$$

which is interpreted as an approximation of the target probability distribution.

Let now be given a set V of test inputs which consists of $32\,n$ observed memory accesses, where 32 is the number of addresses, and n is the number of activations of the victim process. Let further V_1 be the subset of time-address-pairs (t, adr), at which a cache hit occurred (i.e., the victim process accessed address adr at activation t), and let $V_0 = V \setminus V_1$. Let further $y_v \in (0, 1)$ be the output of the neural network for position $v = (t, \mathsf{adr})$. We then define the error function which we are aiming to minimize as

$$e_1(V) = \frac{1}{32n}\left(\sum_{v\in V_1} \log y_v + \sum_{v\in V_0} \log(1-y_v)\right) .$$

For the second neural network, the error function is just the mean square error, i.e., we have

$$e_2(W) = \frac{1}{n}\sum_{w\in W} (t_w - y_w)^2 ,$$

where W consists of n columns of 32 observations each, t_w is the target number of memory accesses in this column, and y_w is the output of the neural network. In a first step, 23 hidden nodes are created using the non-linear function

$$\sigma_{21}(x) = \tanh(\log x) .$$

In a second step, the inner nodes are combined into one node by only computing a weighted sum (i.e., $\sigma_{22}(x) = x$). Finally, somewhat non-standard, the output of our neural network is the variable attached to this node times the original input to the ANN. This real number is then interpreted as the number of table lookups the victim process performed at time t_0. The consequential timing reconstruction turns out to be sufficiently precise to serve as input for the theoretical attack described in §III-B.

4) Sources of Noise:
As can be seen in Figure 6, 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 linear cache prediction logics of the CPU, which informally also fetches data from the memory locations next to the required ones into the cache. Thus, when the encryption process access some address adr_0 in the cache, sometimes adr_1 and adr_2 will be loaded into the cache 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 list, and already computes results in advance. Thus, the results are already available when the according instruction is actually reached, or they are dismissed if they are not needed because, e.g., a context

Copyright @ 2021 Authors

switch happens before. 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.

V. Results and Discussion

In the following we give some timings obtained from real measurements, and discuss extensions of our attack.

A. Results

The following numbers were obtained on our test platform specified in §II-D. Our spy process was specified to start 250 threads and to monitor 100 encryptions of the victim process.

- *Running time*: Performing 100 AES encryptions (i.e., encrypting 1.56kB) takes about 10ms on our platform. Now when monitoring its memory accesses, this blows up to 2.8 seconds. We believe that this delay is sufficiently small for a user not to become sceptical, as such a delay could also be explained by 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 searched 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. §V-C. For each encryption 160 memory accesses are monitored. Thus, $160 \cdot 100 \cdot 32$ bits = 62.5kB would have to be downloaded.

B. 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. As stated there, 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., 4.69kB of encrypted plaintext) are sufficient for this approach.

C. 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.

D. 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 ith column of the state matrix X before respectively 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:

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

Enumerating all possibilities shows that this equation typically has 0 or 1 solutions, where is 0 is dominating.

Copyright @ 2021 Authors

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 16B 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 some possible mitigation strategies to avoid information leakage, or at least to limit it to an extent which renders our attack impossible. An extensive list of countermeasures against access-driven cache attacks can be found in [6].

A. Academical Generic Mitigation Strategies

The probably most obvious generic countermeasure against cache-based attacks is to avoid the usage of CPU caches at all, or to flush the whole cache on any context switch, i.e., whenever the scheduler preempts one process and activates a different one. However, these strategies are only of academical interest, as fetching memory from the RAM is between 10 and 100 times slower than accessing data which has already been copied into the cache. Another solution is to avoid key-dependent table lookups, which may be appropriate for certain security sensitive applications, but typically results in a high computational overhead.

B. Generic (Semi-Efficient) Countermeasures

Several generic countermeasures against access-based cache attacks, which still seem to be sufficiently efficient (at least for client machines), are conceivable:

- One natural way to hamper our attack is to make all high-resolution timers (such as `rdtsc`) inaccessible to processes. Although this does not fully prevent our attack, to the best of our knowledge testing for cache hits would become too inefficient to remain unsuspicious to the user in this case. However, many software packages (e.g., runtime linkers, multimedia applications) extensively use `rdtsc`, and thus only very few scenarios, where this approach is appropriate, are conceivable.
- 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 $T[x]$ is defined to be such obligatorily available data, 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. This might not be the case for systems where, e.g., a multiexponentiation of the form $y = g^x h^r$ in a 2048 bit group has to be evaluated [40]. Also, realizing this feature might require substantial work on the kernels of current operating systems.
- Another option, especially for large lookup tables, is to mark them as *uncachable*, which implies that a victim process will only see cache misses.
- 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.

C. Countermeasures for AES

One concrete mitigation strategy has been realized in OpenSSL 1.0 [18]. There, only the substitution table S is stored, and then the required multiplications within $GF(2^8)$ are computed by using the following relations, which can be realized in a highly efficient way using the `PCMPGTB` instruction:

$$\begin{aligned} 2 \cdot x &= (x \ll 1) \oplus \left(1b \wedge \begin{cases} ff & (\texttt{int8_t})x > 0 \\ 0 & (\texttt{int8_t})x \le 0 \end{cases}\right) \\ &= (x \ll 1) \oplus (1b \wedge \texttt{PCMPGTB}(x, 0)) \\ 3 \cdot x &= 2 \cdot x \oplus x \end{aligned}$$

In this case, the required table contains $2^8 = 256$ entries of $2^0 = 1$ bytes each, and on standard x86-architectures with a cache-line size of $2^6 = 64$ 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.

Because of the large prevalence of AES another mitigation strategy is currently embarked by software vendors. Namely, they are increasingly often offering hardware support of AES in their chips, e.g., [25], rendering access-driven cache attacks impossible.

Copyright @ 2021 Authors

REFERENCES

[1] J.-F. Gallais, I. Kizhvatov, and M. Tunstall, "Improved trace-driven cache-collision attacks against embedded AES implementations," in *WISA 2010 (to appear)*,

[2] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, "Cryptanalysis of DES implemented on computers with cache," in *CHES 2003*, ser. LNCS, C. D. Walter, Ç. Koç, and C. Paar, Eds., vol. 2779. Springer, 2003,

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

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

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

[6] 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,

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

[8] O. Aciiçmez, W. Schindler, and Ç. Koç, "Cache based remote timing attack on the AES," in *CT-RSA 2007*, ser. LNCS, M. Abe, Ed., vol. 4377. Springer, 2007,

[9] D. J. Bernstein, "Cache-timing attacks on AES," http://cr.yp.to/papers.html, 2004,

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

[11] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *CHES 2006*, ser. LNCS, L. Goubin and M. Matsui, Eds., vol. 4249. Springe, 2006,

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

[13] 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,

[14] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo, "AES power attack based on induced cache miss and countermeasure," in *ITCC 2005*. IEEE Computer Society, 2005,

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

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

[17] FIPS, *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology, 2001,

[18] OpenSSL, "OpenSSL: The Open Source toolkit for SSL/TSL," http://www.openssl.org/,

[19] 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,

[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,

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

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

[23] K. Tiri, O. Aciiçmez, M. Neve, and F. Andersen, "An analytical model for time-driven cache attacks," in *FSE 2007*, ser. LNCS, A. Biryukov, Ed., vol. 4593. Springer, 2007,

[24] 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,

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

[26] R. Könighofer, "A fast and cache-timing resistant implementation of the AES," in *CT-RSA 2008*, ser. LNCS, T. Malkin, Ed., vol. 4964. Springer, 2008,

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

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

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

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

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

[32] J. Bernardo and A. Smith, *Bayesian Theory*.

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

[34] D. Tsafrir, Y. Etsion, and D. Feitelson, "Secretly monopolizing the cpu without superuser privileges," in *USENIX Security 2007*. USENIX, 2007,

Copyright @ 2021 Authors

[35] M. Jordan and C. Bishop, “Neural networks,” *ACM Computing Surveys*, vol. 28, no. 1, pp. 73–75,

[36] 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,

[37] C. Bishop, *Neural Networks for Pattern Recognition*.

[38] P. Simard, D. Steinkraus, and J. Platt, “Best practices for convolutional neural networks applied to visual document analysis,” in *ICDAR 2003*. IEEE Computer Society, 2003,

[39] R. Byrd, P. Lu, J. Nocedal, and C. Zhu, “A limited memory algorithm for bound constrained optimization,” *SIAM Journal on Scientific and Statistical Computing*, vol. 16, no. 5, pp. 1990–1208,

[40] B. Möller, “Algorithms for multi-exponentiation,” in *SAC 2001*, ser. LNCS, vol. 2259. Springer, 2001,

Copyright @ 2021 Authors