





Article

Cache Misses and the Recovery of the Full AES 256 Key

Samira Briongos ^{1,2,*} , Pedro Malagón ^{1,2,*}  Juan-Mariano de Goyeneche ^{1,2} 
and Jose M. Moya ^{1,2} 

¹ Integrated Systems Lab, Universidad Politécnica de Madrid, ETSI Telecomunicación, 28040 Madrid, Spain; goyeneche@die.upm.es (J.-M.d.G.); josem@die.upm.es (J.M.M.)

² Center for Computational Simulation, Universidad Politécnica de Madrid, Campus de Montegancedo, 28660 Madrid, Spain

* Correspondence: samirabriongos@die.upm.es (S.B.); malagon@die.upm.es (P.M.)

Received: 8 February 2019; Accepted: 27 February 2019; Published: 6 March 2019



Featured Application: This work introduces a new approach to exploit the information gained from cache attacks. Contrary to previous approaches, we focus on cache misses to retrieve the secret AES encryption key more advantageously: we target the OpenSSL AES T-table-based implementation and show that our approach greatly reduces the number of samples and, consequently, the time required to obtain a 128-bit key. Moreover, we demonstrate a practical recovery of a 256-bit AES key.

Abstract: The CPU cache is a hardware element that leaks significant information about the software running on the CPU. Particularly, any application performing sequences of memory access that depend on sensitive information, such as private keys, is susceptible to suffer a cache attack, which would reveal this information. In most cases, side-channel cache attacks do not require any specific permission and just need access to a shared cache. This fact, combined with the spread of cloud computing, where the infrastructure is shared between different customers, has made these attacks quite popular. Traditionally, cache attacks against AES use the information about the victim to access an address. In contrast, we show that using non-access provides much more information and demonstrate that the power of cache attacks has been underestimated during these last years. This novel approach is applicable to existing attacks: *Prime+Probe*, *Flush+Reload*, *Flush+Flush* and *Prime+Abort*. In all cases, using cache misses as source of information, we could retrieve the 128-bit AES key with a reduction in the number of samples of between 93% and 98% compared to the traditional approach. Further, this attack was adapted and extended in what we call the encryption-by-decryption cache attack (EBD), to obtain a 256-bit AES key. In the best scenario, our approach obtained the 256 bits of the key of the OpenSSL AES T-table-based implementation using fewer than 10,000 samples, i.e., 135 milliseconds, proving that AES-256 is only about three times more complex to attack than AES-128 via cache attacks. Additionally, the proposed approach was successfully tested in a cross-VM scenario.

Keywords: side-channel cache attacks; cache misses; AES; cloud computing

1. Introduction

Cloud computing aims to provide its users with compute resources as they are required, eliminating the need of acquiring and maintaining expensive computing infrastructure. Its low cost, ease of use and on demand access to computing resources have made both government and industry increasingly adopt these technologies. Major companies such as Amazon, Google and Microsoft have

become Cloud Service Providers. Cloud providers offer computing capabilities at low prices because of economies of scale: by achieving high utilization of their servers, they can divide costs between more customers. This means that multiple virtual machines (VMs) are co-hosted on a single physical host relying on a virtual machine manager (VMM) to provide logical isolation between them.

However, physical isolation between tenants does not exist, and, as a consequence, shared hardware can create covert channels between different VMs. CPU caches are one of these shared resources, which can be exploited to recover fine grain information from co-resident tenants. Well known attack techniques, such as *Evict+Time*, *Prime+Probe* [1], *Flush+Reload* [2], and *Flush+Flush* [3], or the recently introduced *Prime+Abort* [4], allow inferring instructions or data accessed by a victim program. Thus, when this memory access depends on sensitive information, this information is leaked.

The most impressive demonstrations of the ability of such attacks to extract private information from their victims target both symmetric [1,5,6] and asymmetric [7–9] cryptographic implementations. Intel CPU caches have traditionally been victims of cache attacks, due to their inclusive architecture and replacement policies. However, several researchers [10–12] have also proven that AMD and ARM processors, which have different architectures and replacement policies, are also vulnerable to cache attacks.

Cache attacks require the victim and the attacker to share the cache. In cloud environments this means that, prior to the attack, the attacker must achieve co-residency with the victim. As first shown by Ristenpart et al. [13], co-residency is achievable and detectable in well-known cloud platforms (Amazon EC2). After this work was published, cloud providers obfuscated the techniques they employed. However, recent works [9,14] have shown that it is still possible to determine if two VMs are allocated in the same physical machine. Once a VM is allocated in the same host as a target victim, its owner is ready to gain information about what the neighbors are doing.

Researchers have also become concerned that these attacks represent a serious threat. For this reason, many approaches have been proposed that try to avoid the leakage [15–17] or to detect attacks [18–21]. To the best of our knowledge, no hardware countermeasure is implemented in real cloud environments and no hardware manufacturer has changed the cache architecture. We believe that the main reason for this absence of countermeasures is the performance penalty they introduce. As a result, an attacker wishing to exploit side-channel cache attacks will only have to take into the account the countermeasures based on detection that can be implemented by users.

In this work, we consider the T-table-based implementation of AES. This implementation is known to be vulnerable to cache attacks. However, it is commonly used for comparison, and to demonstrate different attack techniques. In this work, it serves to our purpose of showing the accuracy of the information gained from the non-access to memory and to quantify the improvement that this approach represents compared to the traditionally used approach based on access.

We could improve the results of previously published side channel attacks, decreasing the number of samples required to retrieve the whole key by multiple orders of magnitude. Every known technique (*Flush+Reload*, *Flush+Flush*, *Prime+Probe* and *Prime+Abort*) can benefit from this approach. Moreover, the presented approach is less prone to consider false positives memory accesses. Regarding non-access, a false positive is considered if the data have been loaded into the cache and removed from the cache in a short period of time, which is unlikely to happen. Whereas a false positive in the traditional approach is considered whenever the data are loaded into the cache in any other round of the AES encryption, rather than the target round, or whenever these data are predicted to be used, thus are speculatively loaded into the cache, being both frequent options.

In summary, the main contributions of this work are the following

- We present a non-access attack, a novel approach to exploit information gained from the cache misses.
- We show that our approach improved the performance of previous cache attacks and demonstrate its effectiveness for *Flush+Reload*, *Flush+Flush*, *Prime+Probe* and *Prime+Abort*. We could reduce

the amount of encryptions required to derive the key between 93% and 98%. That is, if an access attack requires 100,000 samples, our approach requires fewer than 3000, performed in a similar and real experimental setup.

- We extend the non-access attack to gain information from more rounds of the algorithm, introducing EBD, a practical attack implementation that provides the full 256-bit AES encryption key using cache side channel attacks.
- We show that the complexity of the attack does not depend on the size of the key but on the number of rounds of each encryption. Passing from 10 rounds (AES-128) to 14 rounds (AES-256) increases the complexity by a factor of around 3.

The remainder of this paper is organized as follows. In Section 2, we provide information on the relevant concepts to understand the attacks. Section 3 describes AES implementations and the attacks we performed to extract the keys. In Section 4, we provide a discussion of our results. Finally, in Section 5, we draw some conclusions.

2. Background and Related Work

To achieve a better understanding of side channel cache-attacks, we give a basic introduction to some required architectural concepts followed by a description of existing cache attacks and their principles.

2.1. Architectural Concepts

We explain the fundamental ideas of three features that have been key to existing cache attacks: CPU cache memories, shared memory and transactional memory.

2.1.1. CPU Caches

CPU caches are small and fast memories located between the CPU and main memory, specially designed to hide main memory access latencies. As the cost of memory is related to the speed, the cache memory is structured in a hierarchy of typically three levels. Level 1 is the smallest, fastest cache memory and similar to level 2 is usually core private. Level 3, the biggest and slowest cache memory, is commonly shared among processing cores. They hold a copy of recently used data, which will likely be requested by the processor in a short period of time. If the data requested by the processor are not present in any level of the cache, they will be loaded from main memory and stored in the cache. If a cache level is completely filled at that point, some of the data currently contained in it will be replaced (evicted) to store the loaded data instead. The selection of the evicted data depends on the processor architecture. Processors, including AMD, Intel and some ARM high-performance processors, typically use a policy related to least recently used (LRU). However, the replacement policy of most ARM processors is based in random replacement or round-robin replacement.

In Intel processors, such as the one used in our experiments, caches are inclusive memories. This means that high level caches, such as level 3 (L3) cache, have a copy of the data of lower level caches. In addition, what is more important and relevant to most attacks, L3 cache is shared among all cores. Consequently, a process being executed in a core, related to L3 cache data, may produce side effects in other core processes. Other processors use exclusive caches, where a memory block is present in only one level of the cache hierarchy.

Caches are usually set associative: they are organized as S sets of W lines (also called ways) each holding B bytes. It is common to name them as W -way set associative cache. Given a memory address, the less significant $\log_2 B$ bits locate the byte on the line, the previous $\log_2 S$ bits do the same for the set and the remaining high-order bits are used as a tag for each line. The tag will be used to discern whether a line is already loaded or not in the cache. In modern processors, the last level cache (LLC) is divided into slices. Thus, it is necessary to know the slice, and the set in which a datum will be placed. The slice is computed as a hash function of the remaining bits of the address. If the number of cores is a power of two, the hash will be a function of the tag bits. Otherwise, it will use all the bits of the

address [22,23] Note that, as main memory is much larger than CPU caches, multiple memory lines map to the same cache set.

2.1.2. Shared Memory

Programs are loaded from hard disk drives to RAM to be executed, using RAM for both instructions and data, because the access to RAM is much faster than to hard disk. RAM memory is a limited resource, so when there is no free memory available, some data and instructions are moved to virtual memory, which is a copy of RAM stored in the hard disk. Using virtual memory in an application severely affects its performance. It is natural that operating systems employ mechanisms such as memory sharing to reduce memory utilization. Whenever there are two different processes using the same library, instead of loading it twice into RAM memory, sharing memory capabilities allow mapping the same physical memory page into the address spaces of each process.

Deduplication is a concrete method of shared memory, which was originally introduced to improve the memory utilization of VMMs and was later applied to non-virtualized environments. The hypervisor or the operating system scans the physical memory and recognizes processes that place the same data in memory; that is, pages with identical content. When several pages happen to include the same content, all the mappings to these identical pages are redirected to one of them, and the other pages are released. However, if any change is performed by any process in the merged pages, memory is duplicated again.

The Linux memory deduplication feature implementation is called KSM (Kernel Same-page Merging) and appeared for the first time in Linux kernel version 2.6.32. KSM is used as a page sharing technique by the Kernel-based Virtual Machine (KVM), which we used as hypervisor within our experiments. KSM scans the user memory for potential pages to be shared, scanning only potential candidates instead of the whole memory continuously [24].

The deduplication optimization saves memory allowing more virtual machines to run on the host machine. To exemplify this statement, we refer to onl [25], who stated that it is possible to run over 50 Windows XP VMs with 1 GB of RAM each on a machine with just 16 GB of RAM. In terms of performance, deduplication is an attractive feature for cloud providers. However, after several demonstrations of side-channel attacks exploiting page sharing, they are advised to disable this feature and no cloud provider is ignoring this advice.

2.1.3. Transactional Memory

When multiple threads are running in parallel and try to access a shared resource, a synchronization mechanism is required to avoid conflicts. Transactional memory is an attempt to simplify concurrent programming, avoiding common problems with mutual exclusion and improving the performance of other locking techniques. Transactional memory enables optimistic execution of the transactional code regions specified by the programmer. The processor executes the specified sections assuming that it is going to be possible to complete them without any interference. The programmer is no longer responsible for identifying the locks and the order in which they are acquired; he only needs to identify the regions that are going to be defined as part of the “transaction”.

During the execution of a transaction, the variables and the results of the operations are only visible for that thread; that is, any update performed in these regions is not visible to other threads. If the execution ends successfully, the processor commits all the changes as if they had occurred instantaneously, making them visible to any process. If, on the other hand, there is a conflict with another process, the transaction is unsuccessful and the processor aborts its execution. Consequently, the processor requires a mechanism to undo all the updates, discard all the changes and restore the architectural state to pretend that the execution never happened.

Intel Transactional Synchronization Extensions (TSX) are the Intel’s implementation of hardware transactional memory. TSX provides two software interfaces: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM).

2.2. Cache Attacks

Cache memories create covert channels that can be exploited to extract sensitive information. When a process tries to access some data, if it is already loaded into the cache (namely, a cache hit), the time required to recover these data is significantly lower than the access time in the case the data have to be retrieved from main memory (cache miss). Therefore, if the execution time of a cryptographic process depends on the previous presence (or not) of the accessed data on the cache memory, this time information can be exploited to gain private information (such as secret keys) from cryptographic processes. Traditionally, cache attacks have been classified as:

- *Time driven attacks*: The information is learned by observing the timing profile for multiple executions of a target cipher.
- *Trace driven attacks*: The information is gained by monitoring the cache directly, considering that the attacker has access to the cache profile when running the target process.
- *Access driven attacks*: The information is retrieved from the sets of the cache accessed during the execution of the target process

The cache memory was first mentioned as a covert channel to extract sensitive information by Hu [26] in 1992. In 1996, Kocher [27] introduced the first theoretical attacks, as did Kelsey [28] in 1998 describing the possibility of performing attacks based on cache hit ratios. In 2002, Page [29] studied a theoretical example of cache attacks for DES, which was also used by Tsunoo [30] to study timing side-channels created due to table lookups. In 2004, Bernstein [5] proposed the first time-driven attack after observing non-constant times when executing cryptographic algorithms. Although the attack he presented was not practical, his correlation attack has been investigated extensively and even retried between VMs recently [31]. Soon after, another attack was proposed by Percival [32], who suggested that an attacker could determine cache ways occupied by other processes, measuring access times to all ways of a cache set. He found that these times are correlated with the number of occupied ways.

In 2006, Osvik et al. [1] proposed two techniques, which have been widely used since then, named *Evict+Time* and *Prime+Probe*. These techniques are intended to allow an attacker to determine the cache sets accessed by a victim process. A significantly more powerful attack that exploits shared memory and the completely fair scheduler (CFS) was proposed by Gullasch [33]. The same principles of the attack were later exploited by Yarom and Falkner [2], who named the attack *Flush+Reload*. The target of this attack was the L3 cache, as it is inclusive and shared among cores. From those, the *Flush+Reload* and the *Prime+Probe* attacks (and their variants) stand out over the rest due to their higher resolution.

2.2.1. Evict+Time

This technique consists of three steps: First, an encryption is triggered and its execution time measured. Second, an attacker evicts some line. Third, the encryption time is measured again. By comparing the second time with the first measure, an attacker can decide whether the cache line was accessed, as higher times will be related to the use of the mentioned line.

2.2.2. Flush+Reload

This technique relies on the existence of shared memory: thus, when an attacker flushes the desired lines from the cache, he can be sure that, if the victim process needs to retrieve the flushed line, it will have to load it from main memory. *Flush+Reload* also works in three stages: First, the desired lines from the cache are flushed (reverse engineering may be required to determine the addresses that can leak information). Second, the victim runs its process or a fragment of it. Third, the flushed line is accessed by the attacker, measuring the time required to do it. Depending on the reload time, the attacker decides whether the line was accessed.

This attack uses the `clflush` instruction to remove the target lines from the cache in the initialization stage. It is easy to implement and quite resistant to micro-architectural noise, thus it has become popular. However, its main drawback is that it requires memory deduplication to

be enabled. Deduplication is an optimization technique designed to improve memory utilization by merging duplicate memory pages. Consequently, it can only recover information coming from statically allocated data. Shared memory also implies that the attacker and the victim have to be using the same cryptographic library. To eliminate the need of knowing the version of the algorithm attacked, Gruss et al. presented the cache template attack [34], where they enforced the existence of shared memory between the attacking and the attacked processes.

The attack was first introduced by Gullasch et al. [33], and later extended to target the LLC to retrieve cryptographic keys, TLS protocol session messages or keyboard keystrokes across VMs [2,34,35]. It has also demonstrated its power against AES T-table based implementations [36], RSA implementations [2], or ECDSA [37], among others. It can also detect cryptographic libraries [38]. Further, Zhang et al. [39] showed that it is applicable in several commercial PaaS clouds, where it is possible to achieve co-residency with a victim [13].

Relying on the `clflush` instruction and with the same requirements as *Flush+Reload*, Gruss et al. [3] proposed the *Flush+Flush* attack. It was intended to be stealthy and bypass attack monitoring systems [19,20]. The main difference with *Flush+Reload* is that this variant recovers the information by measuring the execution time of the `clflush` instruction instead of the reload time, thus avoiding direct cache accesses. This was the key fact to avoid detection. However, recent works have demonstrated that it is detectable [21,40].

2.2.3. *Prime+Probe*

Shared memory is not always available through deduplication in virtual environments as most cloud providers turned off this feature after several attack demonstrations. However, *Prime+Probe* [9,41] still works, and, by targeting the L3 cache, an attacker can still extract sensitive information. Since *Prime+Probe* is agnostic to special OS features in the system, it can be applied in virtually every system.

Prime+Probe consist of three steps: First, the attacker fills a cache set with known data. Second, the attacker triggers (or waits for) the victim to perform an encryption. Third, after the encryption has finished, the attacker tries to access the known data place during the first step and measures the time it takes to determine if the previously loaded data have been evicted by the victim process. If so, the attacker discovers which lines the victim process has used.

This attack was first proposed for the L1 data cache by [1] and later expanded to the L1 instruction cache [42]. These approaches require both victim and attacker to share the same core, which diminishes practicality. However, it has been recently shown to be applicable to LLC. Researchers have bypassed several difficulties to target the LLC, such as retrieving its complex address mapping [22,23,43], and recovered cryptographic keys or keyboard typed keystrokes [11,41,44]. Furthermore, the *Prime+Probe* attack was used to retrieve a RSA key in the Amazon EC2 cloud [45].

These attacks highly rely on precise timers to retrieve the desired information about the victim memory access. If a defense system tries to either restrict access to the timers or to generate noise that could hide timing information, the attack is less likely to succeed. Once again, attackers have been able to overcome this difficulty. The *Prime+Abort* attack [4] exploits Intel's implementation of Hardware Transactional Memory (TSX) so it does not require timers to retrieve the information. It first starts a transaction to prime the targeted set, waits and finally it may or may not receive and abort depending on whether the victim has or has not accessed this set. That is, no need for timers.

To summarize, the presented attacks target the cache, selecting one memory location that is expected to be accessed by the victim process. They consist of three stages: *initialization* (the attacker prepares the cache somehow), *waiting* (the attacker waits while the victim executes) and *recovering* (the attacker checks the state of the cache to retrieve information about the victim). They differ in the implementation, requirements and achievable resolution.

3. Attacks on AES

In this section, we describe the fundamentals of AES encryption and decryption algorithms. We explain the insides of the attacked T-table-based OpenSSL AES implementation. We present the non-access cache attack against AES-128, which outperforms previously published cache attacks. Afterwards, we explain how the approach followed in this non-access attack can be extended to perform a practical attack on AES-256. We name it encryption-by-decryption cache attack, as we use the information from the encryption to obtain a decryption round key, which can be transformed into an encryption round key. This way, we are able to obtain information from two different and consecutive encryption rounds.

3.1. AES Fundamentals

The AES algorithm is explained in [46]. It is a symmetric block cipher that operates with data in blocks of 16 bytes. Both encryption and decryption behave similarly. They repeatedly apply a round transformation on the state, denoted S . The number of iteration rounds, N_r , depends on the size of the key: 10 rounds for 128-bits, 12 rounds for 192-bits and 14 rounds for 256-bits. The encryption process is depicted in Figure 1. In each round, the algorithm uses a different round key K^r , which is obtained from the algorithm key using a known and fixed scheme. Once a round key is known, there is a straight forward algorithm to recover the same amount of bits of the encryption key. Consequently, to obtain a 256-bit AES key, we need to obtain at least two consecutive round keys.

AES arranges both round key and the data in form of a 4×4 matrix of byte elements. If we follow the terms used in [46], we can denote $S^r, 0 \leq r \leq N_r$ the input state to round r . Each of its byte elements is then denoted as $S^r_{i,j}, 0 \leq i < 4, 0 \leq j < 4$. i indicates the row of the state and j the columns. We use only one subindex j to refer to a column of the state S_j . Considering a plaintext block $p_0 p_1 p_2 \dots p_{15}$, the elements of the initial state S^0 of the algorithm are $S^0_{i,j} = p_{i+4j}$.

The normal round transformation in the encryption process consists of four steps denoted: SubBytes, ShiftRows, MixColumns and AddRoundKey, being SubBytes the only non-linear transformation. SubBytes is an S-box S_{RD} applied to the elements of the state. ShiftRows is an element transposition that rotates the rows of the state i positions to the left. MixColumns operates on the state column by column. The operation is a matrix multiplication of a 4×4 known matrix and each of four the original columns. AddRoundKey is the addition (bitwise XOR in $GF(2)$) of the elements of the state with the corresponding element of the round key.

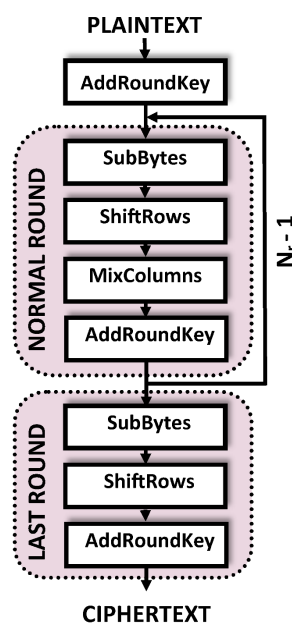


Figure 1. Diagram of the AES encryption process.

All these steps have an inverse step, which are used in the decryption process: `InvSubBytes` (using S-box S_{RD}^{-1}), `InvShiftRows` (rotating to the right), `InvMixColumns` (multiplying by the inverse matrix) and `AddRoundKey`. The normal round transformation is applied in all but the first and last rounds. The first round applies only the `AddRoundKey` step. The last round is a normal round without the `MixColumns` step.

The decryption algorithm can be implemented by applying the inverses of the steps in a reverse order, being `InvSubBytes` the last step of each round excepting the last one. It is preferable to perform the non-linear step first in typical implementations. In [46], the authors presented the *Equivalent Decryption Algorithm*, with a sequence of steps equal to the encryption, by switching the order of application of `InvSubBytes` and `InvShiftRows` (the order is indifferent) and `InvMixColumns` and `AddRoundKey`. The round key of all but the first and last rounds needs to be adapted for this purpose, by applying the `InvMixColumns` step to it.

The key fact to understand the attack against AES-256 is that the state S^r can be reached in the r round of the plaintext encryption process or in the $N_r + 1 - r$ round of the ciphertext decryption process.

3.2. AES T-Table Implementation

The steps of the round transformation include a non-linear S-box substitution and a matrix multiplication. To reduce the number of operations performed during each round and to improve performance, AES software implementations without special assembly instructions use tables (T-tables) with precalculated values for each of the possible 256 values of an input element. In this work, we evaluate the T-table-based implementation available in OpenSSL version 1.0.1f or newer versions when compiled with the `no-asm` flag.

In the aforementioned implementation, the state is represented by four 32-bit variables starting with s or t , one for each column of the state. The name of the variable starts with s for odd rounds and with t for even rounds, considering the first round to be 0. The Most Significant Byte (MSB) of the variable represents the element in Row 0, while the Least Significant Byte (LSB) represents the element in Row 3. Encryption and decryption algorithms use four tables, each containing the result of S-box substitution and part of the matrix multiplication. The tables start with T_e for encryption and with T_d for decryption. Generically, we denote each of the four tables T_i (T_{e_i} and T_{d_i}), where i is the column of the multiplication matrix considered in the table. Table T_i contains the contribution of the element in row i to each row of the resulting column state. That is, each of the 256 positions of the table (2^8) contains a 32-bit value, where each byte is the contribution to each row, aligned to perform the addition of the different contributions. The size of each table is 1024 bytes. The column state is calculated by adding (bitwise XOR in $GF(2)$) the contributions from the different rows and the round key. Therefore, the four tables are accessed for each output column state. An appropriate selection of the input variable s_j for the different rows is used to perform the `ShiftRows` step; for the destination variable t_j , variable s_{i+j} is used for row i (index of table T_i). In this example, and for the entire work, the addition in subindexes i and j is done modulus 4. For example, in Round 1, we can obtain t_0 as follows:

$$t_0 = T_{e_0}[s_0 \gg 24] \oplus T_{e_1}[(s_1 \gg 16) \& 0xff] \oplus T_{e_2}[(s_2 \gg 8) \& 0xff] \oplus T_{e_3}[s_3 \& 0xff] \oplus rk[4];$$

The last round does not include the matrix multiplication. A table for S_{RD}^{-1} (only `InvSubBytes` step) is required for decryption algorithm (table T_{d_4}). As there are coefficients with value 1 in the multiplication matrix of step `MixColumns`, the table is not needed for the encryption algorithm. It is implicitly available in the T_e tables. In this case, $T_{e_{(i+2) \bmod 4}}$ is used for row i , because it has coefficient 1 for that row.

3.3. Attacking AES

As it can be inferred from the previous description of the T-table implementation, the accesses to each T_{e_i} depend on the key. Thus, cache hits and misses depend on the key. In Section 3.3.1, we explain how to retrieve the 128 bits key using information about cache misses and the ciphertext.

We show that, by using cache misses, we are not only able to recover an AES 128 bit key using significantly fewer samples than previous approaches, but we can also recover the 256 bits of an AES key. To this end, we need to recover information about two consecutive rounds. As we explain below, we perform this attack in two steps. The first one targets the last round and the second one the penultimate round. A direct cache attack to the penultimate round key is not feasible, so we use the equivalences between encryption and decryption to transform our data and to be able to use cache misses to retrieve the key.

When describing the attacks, we use a nomenclature that is consistent with the terms used when describing AES. The iteration variables, subindexes and superindexes follow this rules:

- i is used to refer to the rows of the state or round key. As a subindex, it indicates the row of the element. As an iteration value, it is used for accessing elements related to a row (the T-tables).
- j is used to iterate the columns of the state or round key. As a subindex, it indicates the column of an element.
- t is the iteration variable for traversing the different encryptions of the experiment.
- l is the iteration variable for traversing the values of a Cache line.
- r is used as superindex to indicate the round of a variable (state, round key).

The elements used in the algorithm are represented by capital letters.

- S^r represents the state of an AES encryption algorithm before round r . The 16 bytes of the state can be accessed using i and j subindexes. The state is different for each encryption performed, and the array $\overline{S^r}$ represents the whole set.
- D^r is the equivalent to S for the decryption algorithm.
- K^r represents the encryption round key for round r . Each of the 16 bytes can be accessed using i and j subindexes.
- Kd^r represents the decryption round key for round r . It can be calculated from the corresponding encryption key K^{N_r+1-r} by applying the `InvMixColumns` step.
- X_i is information on access (1) or non-access (0) to a subset of table Te_i . The subset contains 16 values (64 bytes), which is the cache line length. $\overline{X_i}$ represents the array containing the access information for each encryption performed in the experiment.
- CK^r is a set of arrays with 256 positions (the possible values), one for each round key byte. Each position represents the amount of discarding votes received by the candidate for that concrete key byte. The position with the minimum value after the analysis (we call the function `argmin`) is the correct key byte value. Its elements can be accessed using i and j subindexes.
- CKd^r is the equivalent to CK^r for a decryption round key.

3.3.1. Attack on AES-128

We assume that the attacker shares the cache with the victim, which means he can monitor accesses to the cache with line granularity. We also assume that the attacker has access to the ciphertext. The attacker needs to recover information from each of the four T-tables. To do so, he can monitor one line of one table, one line of each T-table or even try to monitor the 16 lines of each of the four tables at the same time (64 lines). The number of samples required to retrieve all the bits of the key will vary in each case as will the effect of the attack observed by the victim [18]. The more lines monitored at a time, the more noticeable the attack will be. While a slight increase in the encryption time can be assumed to have been caused by other programs or virtual machines running on the same CPU, a higher increase in the encryption time would be quite suspicious.

For each of the evaluated techniques, i.e., *Prime+Probe*, *Prime+Abort*, *Flush+Reload* and *Flush+Flush*, we use a similar process to retrieve the desired information. The process followed during the attack involves three steps and is the same in all cases:

1. **Setup:** Prior to the attack, the attacker has to retrieve the necessary information about where the T-tables are located in memory; that is, their virtual addresses or the cache set in which they are

- going to be loaded. This way he ensures that the cache is in a known state, ready to perform the attack.
2. **Measurement collection:** In this step, the attacker monitors the desired tables, applying each technique between the requested encryptions and stores the gathered information together with the ciphertext. As explained in Section 2.2, this step can be subdivided into three stages: initialization, waiting and recovering.
 3. **Information processing:** Finally, the attacker uses the information recovered in the previous steps and the information about the T-tables (the contents of the monitored line) to retrieve the secret key.

These main steps are equivalent to previous proposals [36,44]. Other proposals are also applicable to our non-access attack, such as the one suggested by Gülmezoglu et al. [47], which aims to detect the bounds of the encryptions instead of triggering them. However, this approach also depends on the accuracy of the detection, which introduces a new variable to be considered when comparing the effectiveness of the attack approaches. In contrast, the aforementioned setup can be generalized for all the considered techniques, allowing a fair comparison.

In the following, we describe each of the steps for each of the considered algorithms and their particularities in our experimental setup. We consider two scenarios. In the first one, the attack is performed from a spy process running in the same OS as the victim. In the second one, the attacker runs in a VM and the victim runs in a different VM and they are both allocated in the same host. Table 1 includes the details of the machine in which the experiments are performed. The steps are equal in the two considered scenarios.

Table 1. Experimental platform details.

| Processor | Cores | Frequency | OS | LLC Slices | LLC Size | LLC Ways | VMM | Guest OS |
|---------------------|-------|-----------|------------|------------|----------|----------|-----|--------------------|
| Intel core i5-7600K | 4 | 3.8 GHz | CentOS 7.6 | 8 | 6 MB | 12 ways | KVM | CentOS 7.6 minimal |

Setup

The tasks required to perform *Flush+Reload* and *Flush+Flush* are different from those required for *Prime+Probe* and *Prime+Abort*. Since *Flush+Something* attacks rely on shared memory, an attacker wishing to exploit this feature needs to do some reverse engineering on the library used by the victim to retrieve the target virtual addresses. Note that the offset between the addresses of the beginning of the library and the target symbol (the table) is constant. Thus, once this offset is known, the attacker can easily get the target address by adding this offset to the virtual address where the library is loaded.

In contrast, *Prime+Something* attacks require the physical address of the target; more accurately, it is necessary to know the set and the slice in which the data will be loaded. This information can be extracted from the physical address. Part of the physical address can be directly inferred from the virtual address. Indeed, the bits of the address that points to the elements within a cache page are part of their physical address. For example, if the size of a cache page is 4 KB, the 12 lowest significant bits of the address will keep the same when translating it from virtual to physical. However, both virtual and physical addresses of the data that the victim is processing are unknown to the attacker. To overcome this difficulty, an attacker needs to create its own eviction set (a group of W elements that map exactly to one set) and profile the whole cache looking for accesses being carried out by the victim.

Instead of recovering the complex address function of our processor, we create the eviction sets dynamically using the technique summarized in the Algorithm 1 of the work of Liu et al. [41]. We have also enabled *hugepages* of 2 MB in our system to work with 21 known bits. Since the number of cores of our processor is a power of two, we only need to obtain the mapping of one of the sets for each slice (the function that determines the slice in which the data will be placed only uses the *tag* bits). The remaining sets that will be used during the profiling phase and the attack itself can be constructed using this retrieved mapping.

Measurement collection

The measurement collection process is somehow similar for all the considered techniques. Algorithm 1 summarizes the general method employed to acquire the necessary information about the victim accesses to the tables. This information will be later used to obtain the full key.

Algorithm 1 Generic attack algorithm for cache attacks against T-Table AES implementations.

Input: $\text{Address}(\text{Te}_0), \text{Address}(\text{Te}_1), \text{Address}(\text{Te}_2), \text{Address}(\text{Te}_3)$ ▷ Addresses of the T-Tables
Output: $\bar{X}_0, \bar{X}_1, \bar{X}_2, \bar{X}_3, \bar{S}^{N_r+1}$ ▷ Information about the accesses and ciphertext

```

1: for  $t = 0$  to number_of_encryptions do
2:   for  $m = 0$  to 4 do ▷ INITIALIZATION
3:     Evict from cache ( $\text{Te}_m$ ); ▷ The attacker prepares the cache for the attack
4:   end for
5:   ▷ WAITING
6:    $\bar{S}^{N_r+1}[t] = \text{encrypt}(\text{random plaintext});$  ▷ The victim performs one encryption
7:   for  $m = 0$  to 4 do
8:     Infer victim accesses to( $\text{Te}_m$ ) ▷ RECOVERING
9:     if hasAccessed( $\text{Te}_m$ ) then ▷ The attacker Reloads, Flushes, Probes the target or gets the
Abort
10:       $\bar{X}_m[t] = 1;$  ▷ The attacker decides if the victim has used the data.
11:    else
12:       $\bar{X}_m[t] = 0;$ 
13:    end if
14:  end for
15: end for
16: return  $\bar{X}_0, \bar{X}_1, \bar{X}_2, \bar{X}_3, \bar{S}^{N_r+1};$ 

```

In the initialization, the attacker has to evict the data of one line of each Te_i table from the cache. That is, the attacker ensures he knows the state of the cache before the victim uses it. Since each cache line holds 64 bytes, and each entry of the table has 4 bytes, it holds 16 of the 256 possible values of the table. Then, in the waiting stage, he triggers an encryption. Once it has finished, in the recovering stage, the attacker checks the state of the cache. If the cache state has changed, this means that it is likely that the victim had used the data of that T-Table. The different approaches considered in this work differ in the way they evict the data from the cache during the initialization and in the way they recover the information about utilization of the evicted table line.

In the initialization stage of *Flush+Something*, the target lines are evicted from the cache using the *clflush* instruction available in Intel processors. In *Prime+Something*, the attacker evicts the target lines by accessing the data of the created eviction sets that map to the same region as the target lines. Intel implements a pseudo LRU eviction policy. As a result, accessing W elements that map to a set implies that any older data in the cache will be replaced with the attacker's data. In *Prime+Abort*, this operation is performed inside a transactional region defined using the Intel TSX extension.

In the recovering stage of *Flush+Reload*, the attacker accesses each of the evicted lines measuring the access times. If the time is below a threshold, it means the line was in the cache, so the algorithm must have accessed it in any of the rounds. *Flush+Flush* decides if the data have been used by measuring the time it takes to flush the line again. The flushing time depends on the presence of the line on the cache; it takes longer to flush data that are located in the cache. The *Prime+Probe* recovers information on access by accessing the data from the eviction set and measuring the access time. As recommended in previous works [41], we access the data within the eviction set in reverse order in the Probe step. That is, the eviction set is read following a zigzag pattern. If the data of the table are used by the encryption process, part of the attacker's data will be evicted; thus, the time it takes to "Probe" the set will be higher. In the *Prime+Abort*, as the eviction set was accessed inside a transaction, any eviction

of the attacker's data from the cache causes an abort. The attacker defines a handler for this abort that evaluates its cause. There are different abort causes which are identified with an abort code [48]. This abort code is loaded into the *eax* register and it is read in the handler function to check if the abort was due to an eviction.

Figure 2 shows the distribution of the measured times during the Reload step (Figure 2a) and during the Probe step (Figure 2b). Note that it is easy to establish a threshold to distinguish between accesses and not-accesses to the monitored Te table.

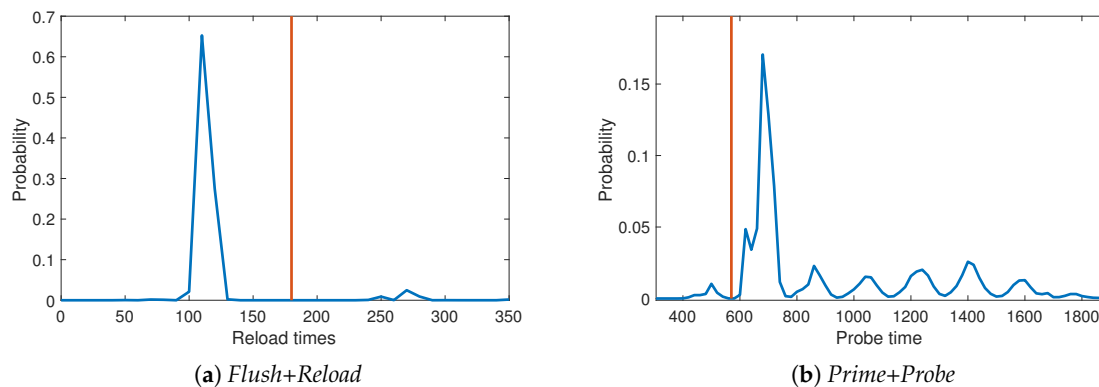


Figure 2. Timing information obtained when attacking AES in the Reload and Probe steps in our test machine. The threshold is shown in red.

Information Processing

This final step is common for all the attacks. In this stage, our approach differs from previous attack strategies. Traditional approaches for the considered attacks take into account if the data stored in the monitored cache line has been used by the victim. In contrast, we consider the opposite situation, i.e., the victim has not used that data. Our approach is based on the following observations:

- Whenever the retrieved information shows that the victim has used the considered data, they could have been used during the last round or during any of the remaining ones. Thus, whenever the data are used in a different round, this approach introduces “noise”. On the other hand, if a monitored cache line has not been accessed during the encryption, neither has it been accessed in the last round nor in any other round.
- Speculative execution of pieces of code, or prefetching of the data that the processor believes are going to be used in the near future, may involve that the attacker sees in the cache data that have not been used by the victim (false positive). Nonetheless, to obtain a false positive in the non-access approach, the data have to be used during the last round of the encryption and then flushed away by other process during the short period of time after the end of the encryption and before the recovering step.
- Each non-access sample reduces the key search space discarding up to n^4 key values (n equals to 16 table entries stored in the cache line, which are used to obtain 4 output bytes). Thus, we obtain information from each of the samples. In contrast, when considering accesses, only when one candidate value towers over the rest, the retrieved information is useful to retrieve the secret key.
- The information obtained this way is more likely to be applicable to all the rounds of an encryption, or at least to the penultimate round, as it refers to the whole encryption, so we can use this information to retrieve longer keys (192 or 256 bits).

We have seen in Section 3.2 that each table *Te* is accessed four times during each round. Therefore, the probability of not accessing a concrete cache line within an encryption is provided by Equation (1), where n represents the number of table entries a cache line can hold, and N_r the number of rounds of the algorithm. Particularly, in our case, where the size of a cache line is 64 bytes, n is 16 (each entry of

Te has 32-bits) and the number of rounds (N_r) for AES-128 is 10. Consequently, the probability of not accessing the cache line is 7.5%. This means that one of each fourteen encryptions performed gives us useful information.

$$Pr[no\ access\ Te_i] = \left(1 - \frac{n}{256}\right)^{N_r * 4} \tag{1}$$

As stated before, we focus on the last round of the encryption process. This round includes SubBytes, ShiftRows and AddRoundKey operations. Referred to the last round input state and the T-tables, we can express an output element by $S_{i,j}^{N_r+1} = K_{i,j}^{N_r} \oplus Te_{i+2} [S_{i,j+i}^{N_r}]$. When a cache line has not been used during the encryption, we can discard all the key values which, given an output byte, would have had to access any of the Te-table entries hold in the mentioned cache line. For example, given a ciphertext byte "0F" and a cache line holding "01", "AF", "B4", and "29", if this line remains unused after performing the encryption, we discard the key byte values $0F \oplus 01$, $0F \oplus AF$, $0F \oplus B4$ and $0F \oplus 29$. As shown in Section 3.2, the same table is used to obtain all the elements of the row of the output state. This means each Te_i table also discards the $K_{i,j}$ values with the same i value independently of the j index.

A general description of the key recovery algorithm for the last round key is provided in Algorithm 2. Key bytes of the last round, $K_{i,j}^{N_r}$, are recovered from the output state \bar{S}^{N_r+1} (ciphertext) and the information about the accesses to each table Te_i , which is stored in \bar{X}_i and recovered using Algorithm 1. The output state (\bar{S}^{N_r+1}) is obtained by arranging the known ciphertext vector as a matrix, as indicated in Section 3.1. The algorithm first initializes an array of 256 elements for each of the 16 bytes of the key ($K_{i,j}$). This array ($CK_{i,j}^{N_r}$) will contain the discarding votes for each key byte candidate. The candidate with less negative votes is the value of the secret key.

Algorithm 2 Recovery algorithm for key byte $K_{i,j}^{N_r}$.

Input: $\bar{X}_0, \bar{X}_1, \bar{X}_2, \bar{X}_3, \bar{S}^{N_r+1}$ \triangleright Information about the accesses and ciphertext collected in the previous step.

Output: $K_{i,j}^{N_r}$ \triangleright Secret key values

```

1: for  $l = 0$  to 256 do
2:    $CK_{i,j}^{N_r} [l] = 0;$   $\triangleright$  Initialization of the candidates array
3: end for
4: for  $t = 0$  to number_of_encryptions do
5:   if  $\bar{X}_{i+2} [t] == 0$  then
6:     for  $l = 0$  to  $n$  do  $\triangleright n$  stands for the number of entries that a cache line holds
7:        $CK_{i,j}^{N_r} [\bar{S}_{i,j}^{N_r+1} [t] \oplus Te_{i+2} [l]] ++;$   $\triangleright$  Vote for discard key candidate
8:     end for
9:   end if
10: end for
11: return  $\text{argmin}(CK_{i,j}^{N_r});$   $\triangleright$  The candidate with the fewest votes for discard is the secret key.
```

Figure 3 shows the values of one of the 16 $CK_{i,j}^{N_r}$ array, which contains the discarding votes each possible key value has received during the attack for a concrete byte of the key. The x-axis represents each possible key value, and the y-axis the number of times each option has been discarded. The minimum is well distinguishable from the rest of possible key values and is highlighted in red. The index with the minimum score represents the secret key. This approach allows recovering the 128 bits of the key with fewer than 3000 encryptions on average in the best case. We present further results and provide further details in Section 4.

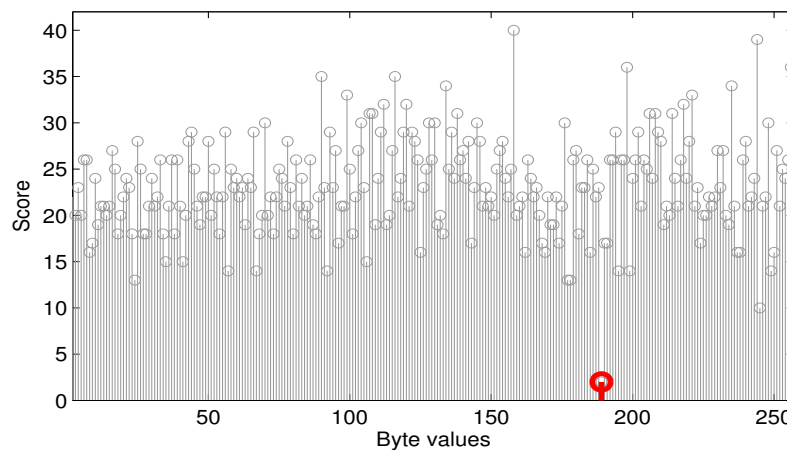


Figure 3. Key byte possible values with their associated discarding scores.

3.3.2. Attack on AES-256

The attack against AES-256 comes as an extension of the previously explained attack on the last round of an encryption. There are no further assumptions than in the previous case. The setup and measurement collection steps are exactly the same for both attacks, and only the information processing phase is different since it includes an extra step. We retrieve 128 bits of the key using Algorithm 2 and use the guessed 128 bits of the key and the same information that has already been used (non-accesses and ciphertext) to derive the remaining 128 unknown bits of the key. We similarly use information about the non-used cache lines to discard key values until we obtain the key. This is possible because this information refers to the whole encryption and, as previously stated, if a cache line has not been accessed within an encryption, it has not been accessed in any of its rounds.

Note that the probability of non-accessing one line when performing an AES-256 encryption, which is a function of the number of rounds, and is given by Equation (1), is 2.69% (14 rounds). Consequently, the attack on AES-256 would require more samples to successfully guess the whole key. In this case, about 1 out of 37 samples carries useful information. This means we need to collect at least the same number of useful samples than in the previous case.

The key to understand the attack is to understand the equivalence between encryption and decryption. Using the decryption algorithm, we transform the data referring to the encryption process in a way that it is possible to easily derive the round key discarding its possible values. For this reason, we call this approach encryption-by-decryption attack (EBD).

If we analyze the AES encryption process, we see that the output of each round is used as an index to access the T-tables in the next round after a ShiftRows operation. Taking into account the valuable information of the unused cache lines, it is possible to figure out which are the state values that are not possible before the penultimate round, S^{Nr-1} . That is, if a cache line has not been accessed, all the state values that would have forced an access to this line can be discarded.

Once we know the last round key, it is straightforward to obtain the input to that round (S^{Nr-1}) only taking into the account the encryption function. However, because of the MixColumns operation which is performed in all the previous rounds (except for the initial one), the certainty about the non-usage of a cache line of one table and the knowledge of the encryption function are not enough to obtain information about the key of these rounds. However, if we use the decryption function and take into the account the equivalence between encryption and decryption, we can transform our data so we can derive information about a state of the decryption function, which is equivalent to the S^{Nr-1} state in the encryption function.

Figure 4 shows with dotted lines the instants where equivalent decryption and encryption algorithms have the same values. According to the figure, we establish a bidirectional relation between an encryption state S^r and a decryption state D^{Nr+1-r} using SubBytes and ShiftRows steps.

The relations are defined in Equations (2a) and (2b). These relations and the knowledge of the values which are not possible in the state S^{Nr-1} give us the values which are not possible in the state D^2 . Since the value of what we call intermediate value IV^1 can be easily obtained from the decryption function, our scenario is similar to the previous one ($D^2_{i,j} = IV^1_{i,j} \oplus CKd^1_{i,j}$).

$$D^{Nr+1-r} = \text{ShiftRows}(S_{RD}(S^r)) \tag{2a}$$

$$S^r = S_{RD}(\text{InvShiftRows}(D^{Nr+1-r})) \tag{2b}$$

Algorithm 3 shows how to obtain the correct candidates for the round key of decryption algorithm Round 1 (the second round). The key of the first decryption round is exactly the same as the key used in the last round of the encryption, previously obtained using the non-access cache attack and targeting the last round (Algorithm 2). As we know both input data (ciphertext) and round key, we can perform the first round of the decryption algorithm and obtain the state of round 1, D^1 . Applying the Td-tables on the state, we calculate an intermediate value of Round 1, IV^1 . At this point, if we perform the AddRoundKey, the next step of the decryption algorithm, the result would be D^2 , which can be transformed into the encryption S^{13} state, as explained using Equation (2b).

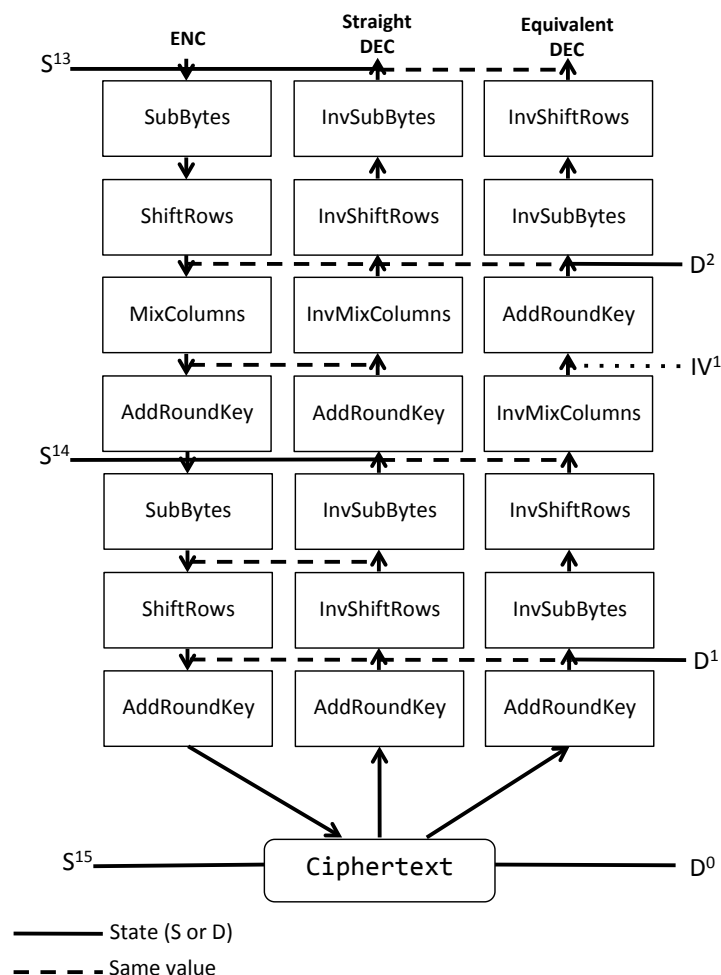


Figure 4. Relation between encryption, straight decryption and equivalent decryption.

In the T-table implementation, the state (including S^{Nr-1} or S^{13} for AES256) is used as index to access the T-tables. Note that not accessing a cache line holding one entry of the table also indicates which indexes of the table are not being used, which is similar to indicating which values of the state can be discarded. Then, we apply Equation (2a) to transform the discarded values for state S^{Nr-1} to discarded values for state D^2 . After this transformation, we have information on both sides of the

AddRoundKey in round 1 of the decryption sequence. We can vote to discard possible values of the decryption round key. We vote to those values, which, once the decryption state is transformed into the encryption state, would lead to a memory access on the unused lines.

The least voted value for each byte is the decryption round key. Both the encryption and decryption keys are related and one can be obtained from the other applying the InvMixColumns operation, or an equivalent operation depending on the direction of the conversion. As a result of this information processing, we know two consecutive round keys; that is, the full AES-256 key. This algorithm could be applied recursively for the whole encryption/decryption process, as the cache information is applicable to the whole encryption.

Using this encryption-by-decryption cache attack on AES-256, we obtain the key with fewer than 10,000 samples in the best scenario. We provide further details in Section 4.

Algorithm 3 Recovery algorithm for key byte $Kd_{i,j}^1$.

Input: $\bar{X}_0, \bar{X}_1, \bar{X}_2, \bar{X}_3, \bar{S}^{N_r+1}$ and \mathbf{K}^{N_r} ▷ Information about the accesses, ciphertext and last round key.
Output: $\mathbf{Kd}_{i,j}^1$ ▷ Secret decryption key values

- 1: $\mathbf{Kd}^0 = \mathbf{K}^{N_r}$
- 2: **for** $l = 0$ **to** 256 **do**
- 3: $\text{CKd}_{ij}^1[l] = 0;$ ▷ Initialization of the candidates array
- 4: **end for**
- 5: **for** $t = 0$ **to** *number_of_encryptions* **do**
- 6: **if** $\bar{X}_i[t] == 0$ **then**
- 7: $D^0 = \bar{S}^{N_r+1}[t];$
- 8: $D^1 = D^0 \oplus \mathbf{Kd}^0;$
- 9: **for** $j = 0$ **to** 4 **do**
- 10: $\mathbf{IV}_j^1 = \mathbf{Td}_0[D_{0,j}^1] \oplus \mathbf{Td}_1[D_{1,j-1}^1] \oplus \mathbf{Td}_2[D_{2,j-2}^1] \oplus \mathbf{Td}_3[D_{3,j-3}^1]$ ▷ Obtain intermediate value
- 11: **end for**
- 12: **for** $l = 0$ **to** n **do** ▷ n stands for the number of entries that a cache line holds
- 13: $\text{CKd}_{i,j}^1 = [\mathbf{IV}_{ij}^1 \oplus S_{RD}[l]] ++;$ ▷ Vote for discard key candidate
- 14: **end for**
- 15: **end if**
- 16: **end for**
- 17: **return** $\text{argmin}(\text{CKd}_{ij}^1);$ ▷ The candidate with the fewest votes for discard is the **secret key**.

4. Discussion of the Results

Our experimental setup included an Intel Core i5-7600K processor (3.80 GHz) with 4 CPUs and two core exclusive L1 caches of 32 KB (one for data and other for instructions), a L2 cache of 256 KB and, finally, a L3 cache of 6 MB shared among all cores whose line size is 64 bytes. It has CentOS 7.6 installed while the Virtual Machines in our experiments have CentOS 7.6 minimal and both have one virtual core. The hypervisor was KVM. This information is summarized in Table 1. The target of our experiments was the 1.0.1f OpenSSL release, which includes the T-Table implementation of AES.

In the previous section, we present attacks that work for different lengths of AES keys (128 or 256 bits). These attacks assume the knowledge of unused cache lines to discard possible key values and that the ciphertexts are known by the attacker. We performed the experiments in two different scenarios (virtualized and non-virtualized environments) considering different techniques to retrieve information about the cache utilization (*Prime+Probe*, *Prime+Abort*, *Flush+Reload* and *Flush+Flush*) and different key lengths. Table 2 shows the mean number of encryptions that had to be monitored to obtain the full key in non-virtualized environments (the victim and the attacker processes run in the

same OS). Similarly, Table 3 shows the results for virtualized environments (the victim runs in one VM and the attacker in a different one sharing the same host).

Table 2. Mean number of samples required to retrieve the whole key in non-virtualized environments.

| | <i>Flush+Reload</i> | <i>Flush+Flush</i> | <i>Prime+Probe</i> | <i>Prime+Abort</i> |
|----------------|---------------------|--------------------|--------------------|--------------------|
| AES 128 | 3000 | 15,000 | 14,000 | 3500 |
| AES 256 | 8000 | 35,000 | 38,000 | 8000 |

Table 3. Mean number of samples required to retrieve the whole key in virtualized environments (cross-VM).

| | <i>Flush+Reload</i> | <i>Flush+Flush</i> | <i>Prime+Probe</i> |
|----------------|---------------------|--------------------|--------------------|
| AES 128 | 10,000 | 40,000 | 45,000 |
| AES 256 | 28,000 | 100,000 | 110,000 |

When applying the *Prime+Abort* technique, we wanted to monitor four different lines. We needed to define four transactional regions to distinguish between them, which also means we used more resources. Note that we do not include the results for the *Prime+Abort* technique in virtualized environments. The TSX instructions were virtualized, thus it should be possible to perform *Prime+Abort* in a cross-VM scenario. However, with our experimental setup consisting on an attacker preparing the cache, triggering an encryption and waiting for the result, we got too many aborts. Since all of them have similar causes, distinguishing between the aborts that were due to the encryption and the ones that were noise was difficult. For this reason, and since we considered that our point was already proved, we did not evaluate the data collected with the *Prime+Abort* technique in the cross-VM scenario.

For a fair comparison between the access and our non-access approach, we also retrieved the key for the access approach considering the key of 128 bits. Table 4 shows the results for both approaches and the percentage of improvement regarding the number of traces required to successfully retrieve the secret key. In all cases, the number of samples was reduced significantly.

Table 4. Mean number of samples required to retrieve the whole 128 bits key with the access approach (first column), our approach (second column) and the improvement our approach means. (V) means virtualized scenario.

| | Access | Non-Access | Improvement |
|------------------------|---------------|-------------------|--------------------|
| <i>Flush+Reload</i> | 100,000 | 3000 | 97% |
| <i>Flush+Flush</i> | 250,000 | 16,000 | 94% |
| <i>Prime+Probe</i> | 190,000 | 14,000 | 93% |
| <i>Prime+Abort</i> | 110,000 | 3500 | 97% |
| <i>Flush+Reload(V)</i> | 400,000 | 10,000 | 98% |
| <i>Flush+Flush(V)</i> | 1,120,000 | 40,000 | 96% |
| <i>Prime+Probe(V)</i> | 1,250,000 | 45,000 | 96% |

We noticed that, when monitoring more than one line at a time, each table provided a different amount of information. This fact can be observed in Figure 5, where we represent a color map for each byte of the key candidates array for that byte. Dark colors represent fewer discarded values (correct key values) while light colors represent most discarded values. With a uniform distribution of the input plaintexts, the minimum score was more remarkable as we increased the number of encryptions. The figure shows that there is a pattern for each byte associated with the same table. We retrieved a different number of useful samples for each table. This was due to the prefetcher and our uniform access pattern that seemed to trigger it. This meant that more encryptions were required to get the full key.

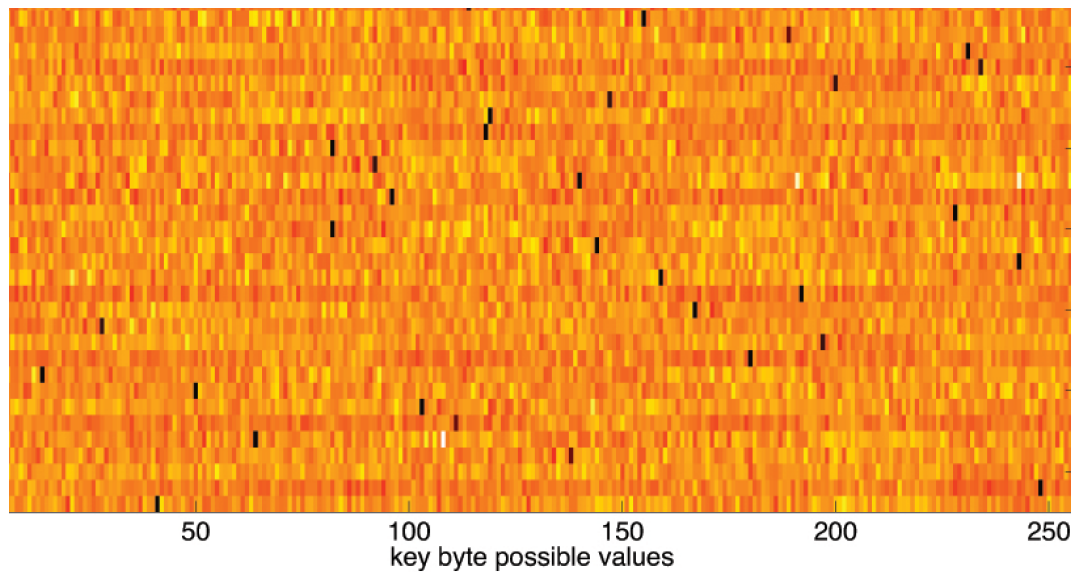


Figure 5. Color map of discarding scores distribution for the 32 bytes of the key retrieved with the *Flush+Reload* technique. Correct key values are the darkest ones.

We could completely recover the full AES 256 key using the EBD attack. Comparing the number of traces required to obtain the 128 bit key and the number of traces required to obtain the 256 bit key, AES-256 happened to be only three times harder to attack with our approach. The different probabilities of non-accessing a line explained our results: 2.69% for AES-256 and 7.5% for AES-128, which in theory indicates $2.8 \times$ more encryptions being needed to get the same information. In practice, we measured 5.29%, 3.87%, 4.55% and 4.31% of cache misses within our samples for AES-128 and each of the lines, respectively, and 1.52%, 0.93%, 1.31% and 1.02% for AES-256 for *Flush+Reload*. These results allow us to discuss the results in [49]. They state that, although larger key lengths translate into an exponential increase in the complexity of a brute force approach, this cannot be applied to side-channel attacks. In addition, using the two last rounds of the encryption algorithm (a version that uses a different table for computing the last round of the encryption), they arrive at the conclusion that AES-256 only provides an increase in complexity of 6–7 compared to cache-based attacks on AES-128. We, in contrast, demonstrate that this factor is equal to 3.

In the case of the EBD attack, we did not find similar results or experimental setups to compare with. All studied attacks can be described as eminently theoretical. In the literature, Acici mez et al. [50] stated that their attack can be applied to the second round of the encryption, but they did not provide experimental results. In 2009, Biryukov et al. [51] presented a distinguisher and related key attack on AES-256 with 2^{131} data complexity. Later, they slightly improved their results [52], achieving 2^{119} data complexity. In [53], they described an attack using two related keys to recover the 256-bit key, but their target cipher was the nine-round version of AES-256 and used chosen-plaintexts. Another attack [54] runs in a related-subkey scenario with four related keys targeting the 13-round AES-256; it has 2^{76} time and data complexity. Bogdanov et al. [55] presented a technique of cryptanalysis with bicliques, also leading to a non-practical complexity recovering the key. Kim [56] explained also theoretically how to apply differential fault analysis to reduce the required number of faults in AES-192 and AES-256.

As stated above, each retrieved sample that carries information about non-accesses to the T-Tables reduces the key search space. This means that, if the attack is interrupted for some reason before it finishes and the attacker decides to apply brute force to retrieve the key, the number of keys he has to check is much smaller. In Figure 6, we present the base 2 logarithm of the key search space vs. the total number of samples retrieved, i.e., the number of bits that remain unknown after each performed encryption. Note that, in the case of the encryption with the 256 bit key, we also begin with 128 unknown bits; as explained above, we get the full key in two round using the same data

and in each round we recover 128 bits. Thus, the figure represents the mean number of unknown bits per round.

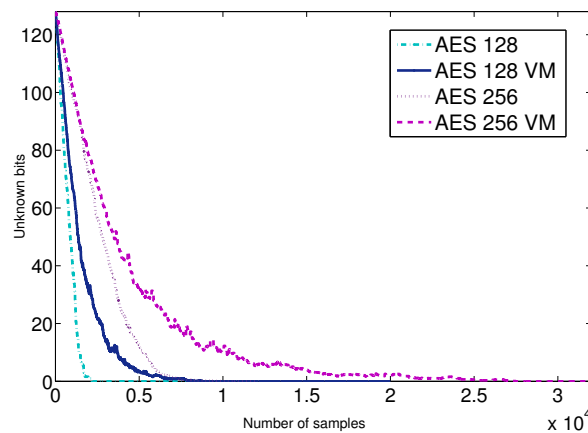


Figure 6. Key bits that remain unknown after each encryption for the *Flush+Reload* technique.

To put these data in context, we also measured the time the two processes (victim and attacker) need to interact until we can obtain the full key. This time is referred to the measurement collection step, since the information processing is performed offline. Table 5 shows these mean times.

Table 5. Mean times required for gathering the data of Figure 6.

| Attack Duration | |
|-----------------|---------|
| AES-128 | 43 ms |
| AES-128 (VM) | 558 ms |
| AES-256 | 135 ms |
| AES-256 (VM) | 1732 ms |

Assuming the number of collected samples varies linearly with the time, these results show that, if we want to detect such attacks before they succeed, it is necessary to act quickly. Detection times have to be in the order of milliseconds. Chiappetta et al. [19] let the victim run for 100 ms, which means our approach would succeed before they detect the attack. Even detection times of around 10 ms imply that around half of the key would be leaked. In CloudRadar [20], detection times are approximately 5 ms, then the hypervisor would need to migrate the machine. CacheShield [21] triggers an alarm after 4.48 ms, and would have to trigger later a countermeasure. It is realistic to assume that, even with any of these countermeasures enabled, victim and attacker can interact for at least 8–10 ms. This means part of the key would be leaked anyway, whereas there would be no leakage with the access approach. Note that we are assuming that the victim is encrypting random plaintext. If the attacker can make the victim encrypt chosen plaintext in such a way the number of non-accessed lines increases, our approach would succeed much more quickly and trigger almost no cache misses, rendering previous countermeasures ineffective.

We demonstrated that our non-access approach retrieves the AES key using fewer samples than the traditional access approach in Intel processors that mount an inclusive cache. However, AMD and ARM processors [10–12] that are vulnerable to cache attacks can also benefit from our attack. We believe it is possible to achieve a similar reduction in the number of samples required to succeed in these processors. It was also recently demonstrated that non-inclusive caches are also vulnerable to cache attacks [57]. As a result, our approach is feasible for non-inclusive caches. As long as the attacker is able to evict the victim's data from the cache, he will be able to use our approach to improve the efficiency of the attack against AES.

The T-Table implementation has been replaced by the S-Box implementation in software. In this case, the encryption is performed using a unique table holding 256 byte values. An attacker can also use our approach to target this implementation. If he is able to flush a line in the last round of the encryption, and given that the 16 accesses to the table per round represent a non-accessing probability of 1% (the attack is feasible), each non-accessed line would allow us to discard 64 values out of 256 possible of each byte of the key. Note that our presented attack against the T-table implementation discards 16 values of 256 possible for 4 bytes of the key. Thus, the estimated leakage of the theoretical attack to the last round of the new implementation is about 75 times higher than the presented attack to the T-table implementation.

5. Conclusions

The massive adoption of cloud computing has introduced numerous advantages, but also new challenging security and privacy risks. One of these risks is the information leakage derived from resource sharing among VMs and, specifically, from the shared cache memory. This risk is also present in non-virtualized environments where all the running processes share the cache.

This work introduces a novel approach to exploit the information gained from the cache by means of existing cache attacks: *Flush+Reload*, *Flush+Flush*, *Prime+Probe* and *Prime+Abort*. We focus on the cache lines that have not been used and target the AES T-Table implementation to prove our point. Even when the T-table version of the AES algorithm is known to be vulnerable to side-channel attacks, it is still available in newer OpenSSL versions (compiling with the no-asm flag). Using the non-access attack, we demonstrate a reduction that varies from 93% to 98% in the amount of encryptions required to obtain an AES-128 key. This improvement is crucial to develop a practical attack on real servers, where the AES key is usually a session key; i.e., it is periodically renewed using key-exchange algorithms. Besides, we demonstrate that, if there is a detection system trying to detect cache attacks running, the amount of leaked information will depend on the detection time, but our approach recovers information from the very beginning of the attack.

Moreover, we present a practical attack in both time and complexity against the 256-bit version of AES: the encryption-by-decryption (EBD) cache attack. We implemented the attack on a real experimental setup and successfully obtained the encryption key in a matter of milliseconds, requiring only 10,000 encryptions in the best scenario. EBD retrieves the key by breaking the attack into two 128-bit stages. In the first stage, non-accessed lines are used to obtain the last 128 bits of the key by discarding key values when the data has not been used. In the second stage, EBD takes the 128-bit subkey obtained in the first stage and uses it with the decryption function (performing part of an AES decryption) to reach an state equivalent to an encryption state from which the non-accessed lines can serve to obtain the remaining 128 bits of the key. Our results show that AES-256 is only three times harder to attack than AES-128.

Author Contributions: Conceptualization, S.B., P.M., J.-M.d.G. and J.M.M.; Data curation, S.B.; Funding acquisition, S.B. and J.M.M.; Investigation, S.B.; Methodology, S.B. and P.M.; Software, S.B.; Supervision, J.M.M.; Writing—original draft, S.B. and P.M.; and Writing—review and editing, J.-M.d.G.

Funding: This research was funded by Spanish Ministry of Economy and Competitiveness grant numbers TIN-2015-65277-R, AYA2015-65973-C3-3-R and RTC-2016-5434-8.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Osvik, D.A.; Shamir, A.; Tromer, E. Cache Attacks and Countermeasures: The Case of AES. In Proceedings of the Topics in Cryptology—CT-RSA, San Jose, CA, USA, 13–17 February 2006; Springer: Heidelberg/Berlin, Germany, 2006; Volume 3860, pp. 1–20.
2. Yarom, Y.; Falkner, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In Proceedings of the USENIX Security Symposium, San Diego, CA, USA, 20–22 August 2006; USENIX Association: Berkeley, CA, USA, 2014; pp. 719–732.

3. Gruss, D.; Maurice, C.; Wagner, K. Flush+Flush: A Stealthier Last-Level Cache Attack. *CoRR* **2015**.₁₄. [[CrossRef](#)]
4. Disselkoe, C.; Kohlbrenner, D.; Porter, L.; Tullsen, D. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada, 16–18 August 2017; USENIX Association: Berkeley, CA, USA, 2017; pp. 51–67.
5. Bernstein, D.J. Cache-Timing Attacks on AES 2005. Available online: <http://palms.ee.princeton.edu/system/files/Cache-timing+attacks+on+AES.pdf> (accessed on 28 February 2019).
6. Weiß, M.; Heinz, B.; Stumpf, F. A Cache Timing Attack on AES in Virtualization Environments. In Proceedings of the International Conference on Financial Cryptography and Data Security—FC, Kralendijk, Bonaire, 27 February–2 March 2012; Springer: Heidelberg/Berlin, Germany, 2012; Volume 7398, pp. 314–328.
7. Brumley, B.B.; Hakala, R.M. Cache-Timing Template Attacks. In Proceedings of the Advances in Cryptology—ASIACRYPT, Tokyo, Japan, 6–10 December 2009; Springer: Heidelberg/Berlin, Germany, 2009; Volume 5912, pp. 667–684.
8. Chen, C.; Wang, T.; Kou, Y.; Chen, X.; Li, X. Improvement of trace-driven I-Cache timing attack on the {RSA} algorithm. *J. Syst. Softw.* **2013**, *86*, 100–107. [[CrossRef](#)]
9. Inci, M.S.; Gulmezoglu, B.; Irazoqui, G.; Eisenbarth, T.; Sunar, B. Seriously, Get off My Cloud! Cross-VM RSA Key Recovery in a Public Cloud. Cryptology ePrint Archive, Report 2015/898. 2015. Available online: <http://eprint.iacr.org/> (accessed on 28 February 2019).
10. Irazoqui, G.; Eisenbarth, T.; Sunar, B. Cross Processor Cache Attacks. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16), Xi'an, China, 30 May–3 June 2016; ACM: New York, NY, USA, 2016; pp. 353–364.
11. Lipp, M.; Gruss, D.; Spreitzer, R.; Mangard, S. ARMageddon: Last-Level Cache Attacks on Mobile Devices. *arXiv* **2015**, arXiv:1511.04897.
12. Green, M.; Rodrigues-Lima, L.; Zankl, A.; Irazoqui, G.; Heyszl, J.; Eisenbarth, T. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada, 16–18 August 2017; USENIX Association: Berkeley, CA, USA, 2017; pp. 1075–1091.
13. Ristenpart, T.; Tromer, E.; Shacham, H.; Savage, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In Proceedings of the ACM Conference on Computer and Communications Security (CCS), Chicago, IL, USA, 9–13 November 2009; ACM: New York, NY, USA, 2009; pp. 199–212.
14. Zhang, Y.; Juels, A.; Oprea, A.; Reiter, M.K. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In Proceedings of the IEEE Symposium on Security and Privacy, S&P, Berkeley, CA, USA, 22–25 May 2011; pp. 313–328.
15. Wang, Z.; Lee, R.B. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In Proceedings of the 34th Annual International Symposium on Computer Architecture, ISCA '07, San Diego, CA, USA, 9–13 June 2007; ACM: New York, NY, USA, 2007; pp. 494–505.
16. Kim, T.; Peinado, M.; Mainar-Ruiz, G. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In Proceedings of the 21st USENIX Security Symposium (USENIX Security 12), Bellevue, WA, USA, 8–10 August 2012; USENIX Association: Berkeley, CA, USA, 2012; pp. 189–204.
17. Liu, F.; Ge, Q.; Yarom, Y.; Mckeen, F.; Rozas, C.; Heiser, G.; Lee, R.B. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), Barcelona, Spain, 12–16 March 2016; pp. 406–418.
18. Briongos, S.; Malagón, P.; Risco-Martín, J.L.; Moya, J.M. Modeling Side-channel Cache Attacks on AES. In Proceedings of the Summer Computer Simulation Conference, SCSC '16, Montreal, QC, Canada, 24–27 July 2016; Society for Computer Simulation International: San Diego, CA, USA, 2016; pp. 37:1–37:8.
19. Chiappetta, M.; Savas, E.; Yilmaz, C. Real time detection of cache-based side-channel attacks using hardware performance counters. *Appl. Soft Comput.* **2016**, *49*, 1162–1174. [[CrossRef](#)]
20. Zhang, T.; Zhang, Y.; Lee, R.B. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In Proceedings of the 19th International Symposium on Research in Attacks, Intrusions, and Defenses, RAID 2016, Paris, France, 19–21 September 2016; Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J., Eds.; Springer International Publishing: Cham, Switzerland, 2016; pp. 118–140.

21. Briongos, S.; Irazoqui, G.; Malagón, P.; Eisenbarth, T. CacheShield: Detecting Cache Attacks Through Self-Observation. In Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY '18, Tempe, AZ, USA, 19–21 March 2018; ACM: New York, NY, USA, 2018; pp. 224–235.
22. Maurice, C.; Le Scouarnec, N.; Neumann, C.; Heen, O.; Francillon, A. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In Proceedings of the Research in Attacks, Intrusions, and Defenses, Kyoto, Japan, 2–4 November 2015; Bos, H., Monrose, F., Blanc, G., Eds.; Springer International Publishing: Cham, Switzerland, 2015; pp. 48–65.
23. Irazoqui, G.; Eisenbarth, T.; Sunar, B. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In Proceedings of the 2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, 26–28 August 2015; pp. 629–636.
24. Arcangeli, A.; Eidus, I.; Wright, C. Increasing memory density by using KSM. In Proceedings of the Linux Symposium, Montreal, QC, Canada, 13–17 July 2009; pp. 19–28.
25. Kernel Samepage Merging. 2015. Available online: http://kernelnewbies.org/Linux_2_6_32#head-d3f32e41df508090810388a57efce73f52660ccb/ (accessed on 28 February 2019).
26. Hu, W. Lattice scheduling and covert channels. In Proceedings of the IEEE Symposium on Research in Security and Privacy, Oakland, CA, USA, 4–6 May 1992; IEEE Computer Society: Washington, DC, USA, 1992; pp. 52–61.
27. Kocher, P.C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Proceedings of the Advances in Cryptology—CRYPTO, Santa Barbara, CA, USA, 18–22 August 1996; Springer: Heidelberg/Berlin, Germany, 1996; Volume 1109, pp. 104–113.
28. Kelsey, J.; Schneier, B.; Wagner, D.; Hall, C. Side Channel Cryptanalysis of Product Ciphers. *J. Comput. Secur.* **2000**, *8*, 141–158. [\[CrossRef\]](#)
29. Page, D. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. Cryptology ePrint Archive, Report 2002/169. 2002. Available online: <http://eprint.iacr.org/> (accessed on 28 February 2019).
30. Tsunoo, Y.; Saito, T.; Suzaki, T.; Shigeri, M.; Miyauchi, H. Cryptanalysis of DES Implemented on Computers with Cache. In Proceedings of the Cryptographic Hardware and Embedded Systems—CHES, Cologne, Germany, 8–10 September 2003; Springer: Heidelberg/Berlin, Germany, 2003; Volume 2279, pp. 62–76.
31. Irazoqui, G.; Inci, M.S.; Eisenbarth, T.; Sunar, B. Fine grain Cross-VM Attacks on Xen and VMware Are Possible! In Proceedings of the 2014 IEEE Fourth International Conference on Big Data and Cloud Computing, (BD-CLOUD '14), Sydney, Australia, 3–5 December 2014; IEEE Computer Society, Washington, DC, USA, 2014; pp. 737–744.
32. Percival, C. Cache missing for fun and profit. In Proceedings of the BSDCan 2005, Ottawa, ON, Canada, 13–14 May 2005.
33. Gullasch, D.; Bangerter, E.; Krenn, S. Cache Games—Bringing Access-Based Cache Attacks on AES to Practice. In Proceedings of the IEEE Symposium on Security and Privacy, S&P, Berkeley, CA, USA, 22–25 May 2011; pp. 490–505.
34. Gruss, D.; Spreitzer, R.; Mangard, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In Proceedings of the USENIX Security Symposium, Washington, DC, USA, 12–14 August 2015; USENIX Association: Berkeley, CA, USA, 2015; pp. 897–912.
35. Irazoqui, G.; Inci, M.S.; Eisenbarth, T.; Sunar, B. Lucky 13 Strikes Back. In Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, 14–17 April 2015; ACM: New York, NY, USA, 2015; pp. 85–96.
36. Irazoqui, G.; Inci, M.S.; Eisenbarth, T.; Sunar, B. Wait a Minute! A fast, Cross-VM Attack on AES. In Proceedings of the Research in Attacks, Intrusions and Defenses Symposium, RAID, Gothenburg, Sweden, 17–19 September 2014; Springer: Heidelberg/Berlin, Germany, 2014; Volume 8688, pp. 299–319.
37. Yarom, Y.; Benger, N. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-Channel Attack; Cryptology ePrint Archive, Report 2014/140. 2014. Available online: <http://eprint.iacr.org/> (accessed on 28 February 2019).
38. Irazoqui, G.; Inci, M.S.; Eisenbarth, T.; Sunar, B. Know Thy Neighbor: Crypto Library Detection in Cloud. In Proceedings of the Privacy Enhancing Technologies 2015, Philadelphia, PA, USA, 30 June–2 July 2015; pp. 25–40.

39. Zhang, Y.; Juels, A.; Reiter, M.K.; Ristenpart, T. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14, Scottsdale, AZ, USA, 3–7 November 2014; ACM: New York, NY, USA, 2014; pp. 990–1003.
40. Kulah, Y.; Dincer, B.; Yilmaz, C.; Savas, E. SpyDetector: An approach for detecting side-channel attacks at runtime. *Int. J. Inf. Secur.* **2018**. [[CrossRef](#)]
41. Liu, F.; Yarom, Y.; Ge, Q.; Heiser, G.; Lee, R.B. Last-Level Cache Side-Channel Attacks are Practical. In Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Jose, CA, USA, 17–21 May 2015; pp. 605–622.
42. Aciçmez, O.; Schindler, W. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and its Demonstration on OpenSSL. In Proceedings of the Topics in Cryptology—CT-RSA 2008, San Francisco, CA, USA, 8–11 April 2008; Springer: Heidelberg/Berlin, Germany, 2008; pp. 256–273.
43. Yarom, Y.; Ge, Q.; Liu, F.; Lee, R.B.; Heiser, G. Mapping the Intel Last-Level Cache. Cryptology ePrint Archive, Report 2015/905. 2015. Available online: <http://eprint.iacr.org/> (accessed on 28 February 2019).
44. Apecechea, G.I.; Eisenbarth, T.; Sunar, B. S\$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing—And Its Application to AES. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, 17–21 May 2015; pp. 591–604.
45. İnci, M.S.; Gulmezoglu, B.; Irazoqui, G.; Eisenbarth, T.; Sunar, B. Cache Attacks Enable Bulk Key Recovery on the Cloud. In Proceedings of the Cryptographic Hardware and Embedded Systems—CHES 2016: 18th International Conference, Santa Barbara, CA, USA, 17–19 August 2016.
46. Daemen, J.; Rijmen, V. *The Design of Rijndael: AES—The Advanced Encryption Standard*; Information Security and Cryptography; Springer: Heidelberg/Berlin, Germany, 2002.
47. Gülmezoglu, B.; İnci, M.S.; Irazoqui, G.; Eisenbarth, T.; Sunar, B. A Faster and More Realistic Flush+Reload Attack on AES. In Proceedings of the International Workshop on Constructive Side-Channel Analysis and Secure Design—COSADE, Berlin, Germany, 13–14 April 2015; Springer: Heidelberg/Berlin, Germany, 2015; Volume 9064, pp. 111–126.
48. Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual (Section 2.1.1.2). 2017. Available online: <https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf> (accessed on 28 February 2019).
49. Neve, M.; Tiri, K. On the Complexity of Side-Channel Attacks on AES-256—Methodology and Quantitative Results on Cache Attacks. Cryptology ePrint Archive, Report 2007/318. 2007. Available online: <http://eprint.iacr.org/> (accessed on 28 February 2019).
50. Aciçmez, O.; Schindler, W.; Koç, Ç.K. Cache Based Remote Timing Attack on the AES. In Proceedings of the Topics in Cryptology—CT-RSA, San Francisco, CA, USA, 5–9 February 2007; Springer: Heidelberg/Berlin, Germany, 2007; Volume 4377, pp. 271–286.
51. Biryukov, A.; Khovratovich, D.; Nikolić, I. Distinguisher and Related-Key Attack on the Full AES-256. In Proceedings of the Annual International Cryptology Conference on Advances in Cryptology, Santa Barbara, CA, USA, 16–20 August 2009; Springer: Heidelberg/Berlin, Germany, 2009; pp. 231–249.
52. Biryukov, A.; Khovratovich, D. Related-Key Cryptanalysis of the Full AES-192 and AES-256. Cryptology ePrint Archive, Report 2009/317. 2009. Available online: <http://eprint.iacr.org/> (accessed on 28 February 2019).
53. Biryukov, A.; Dunkelman, O.; Keller, N.; Khovratovich, D.; Shamir, A. Key Recovery Attacks of Practical Complexity on AES-256 Variants with up to 10 Rounds. In Proceedings of the Advances in Cryptology—EUROCRYPT, French Riviera, France, 30 May–3 June 2010; Springer: Heidelberg/Berlin, Germany, 2010; Volume 6110, pp. 299–319.
54. Biryukov, A.; Khovratovich, D. Feasible Attack on the 13-round AES-256. Cryptology ePrint Archive, Report 2010/257. 2010. Available online: <http://eprint.iacr.org/> (accessed on 28 February 2019).
55. Bogdanov, A.; Khovratovich, D.; Rechberger, C. Biclique Cryptanalysis of the Full AES. In Proceedings of the Advances in Cryptology—ASIACRYPT, Seoul, Korea, 4–8 December 2011; Springer: Heidelberg/Berlin, Germany, 2011; Volume 7073, pp. 344–371.

56. Kim, C.H. Differential fault analysis of AES: Toward reducing number of faults. *Inf. Sci.* **2012**, *199*, 43–57. [[CrossRef](#)]
57. Yan, M.; Sprabery, R.; Gopireddy, B.; Fletcher, C.; Campbell, R.; Torrellas, J. Attack directories, not caches: Side channel attacks in a non-inclusive world. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP) (2019), San Fransisco, CA, USA, 20–22 May 2019.



© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).