The ISC Int'l Journal of Information Security

ISeCure

January 2021, Volume 13, Number 1 (pp. 19–27)

http://www.isecure-journal.org

Evict+Time Attack on Intel CPUs without Explicit Knowledge of Address Offsets

Vahid Meraji [1], and Hadi Soleimany [2,*]

[1] *Cyberspace Research Institute, Shahid Beheshti University, Iran*

[2] *Cyberspace Research Institute, Shahid Beheshti University, Iran*

ARTICLE INFO.

Article history:

Received: November 30, 2019

Revised: June 30, 2020

Accepted: September 1, 2020

Published Online: September 5, 2020

Keywords:

Evict+Time Attack, Access-Driven Attacks, LRU Replacement Policy, Inclusive Memory

Type: Research Article

doi: `10.22042/isecure.2020.209945.500`

Abstract

Access-driven attacks are a series of cache-based attacks using fewer measurement samples to extract sensitive key values due to the ability of the attacker to evict or access cache lines compared to the other attacks based on this feature. Knowledge of address offset for the corresponding data blocks in cryptographic libraries is a prerequisite for an adversary to reload or evict cache lines in Intel processors. Preventing the access of attackers to the address offsets can potentially be a countermeasure to mitigate access-driven attacks. In this paper, we demonstrate how to perform the Evict+Time attack on Intel x86 CPUs without any privilege of knowing address offsets.

1 Introduction

Numerous studies have been conducted to present new attacks using the time difference between the processor access to main memory and cache memory. Such attacks are mostly divided into three categories of time-driven, trace-driven, and access-driven attacks. In time-driven attacks, the attacker knows the capacity of the cache memory lines and measures the cryptosystem runtime. After that, he performs statistical analyses on the measured samples to distinguish between cache miss and cache hit events. The amount of information extracted in these attacks depends on the capacity of the cache memory lines [1–5]. In trace-driven attacks, the attacker is able to create a profile in order to distinguish between the cache hit and cache miss [6]. Access-driven attacks can be divided into synchronous and asynchronous categories. The Evict+Time, Prime+Probe[7] and Flush+Reload attacks [8] are among the most important access-driven attacks. In the access-driven attacks, the attacker frequently needs to evict or reload data from the cache memory before or after performing the targeted cryptosystem which requires the knowledge about the virtual or physical addresses. In other words, it is assumed that the attacker is able to detect the address offsets of the targeted data in the libraries used by the cryptosystem.

One possible solution for preventing the success of access-driven attacks is to prevent page sharing which was mentioned for the first time in [8]. However, as it is discussed in the literature this would increase the execution time by increasing the cache-miss rate. In this paper, we study this approach from another angle. Independent of performance issues, we aim to investigate whether or not such a naive approach can be secure at all. In this paper, we propose an Evict+Time attack on AES which is applicable based on the fewer assumptions in comparison to the previous attacks. Unlike the proposed access-driven attacks in the literature, our method can retrieve all bits of the secret key without requiring any knowledge about the virtual address of the data. The attack is applicable on a processor in which the cache memory

* Corresponding author.

Email addresses: `vahmeraji@gmail.com`, `h_soleimany@sbu.ac.ir`

ISSN: 2008-2045 © 2020 ISC. All rights reserved.

has the following characteristics:

(1) The last level of cache memory uses the LRU replacement policy.
(2) The last level of cache memory is inclusive with respect to the upper levels of cache.

We should note that most of the cache memories in the modern Intel processors have the aforementioned properties

This paper is organized as follows. In Section 2, we describe background information. We introduce the access-driven attacks in Section 3. In Section 4, we describe our technique to apply Evict+Time attack without explicit knowledge of address offsets on Intel CPUs. We present the experimental results in Section 5. Finally, we conclude in Section 6.

2 Background

2.1 Software Implementations of AES

Advanced Encryption Standard (AES) is a block cipher which has been adopted as an encryption standard by the U.S. government. AES has a substitution-permutation network (SPN) structure with 128-bit block size and operates on a 4×4 order array of bytes. AES supports a key length of 128, 192 or 256 bits which are determined as AES-128, AES-192 and AES-256, respectively. In this paper, we consider an attack on AES-128 which has 10 rounds. Each round composed of four types of transformation: round key addition, Sbox, ShiftRow and MixColumn which we denote by S, `SR` and `MC`, respectively. The `MC` operation is omitted in the final round and a whitening key addition is performed after the last round.

Common libraries like `OpenSSL` use the look-up tables presented in Equation 2.1 to implement AES. In these libraries, the columns of the output matrix for the first nine rounds are computed as presented in Equation 2. The method was proposed for the first time in [9]. As the last round does not have MixColumn, T-tables cannot be utilized straightforwardly. In Equation 3, a common implementation for the last round is presented which makes use of T-tables [10]. In all equations, s_j^r refers to the j-th byte in the input of the r-th round where $0 \leq j \leq 15$ and $r \leq 10$.

$$T_0 = \begin{bmatrix} 02.S(z) \\ S(z) \\ S(z) \\ 03.S(z) \end{bmatrix}, T_1 = \begin{bmatrix} 03.S(z) \\ 02.S(z) \\ S(z) \\ S(z) \end{bmatrix}, T_2 = \begin{bmatrix} S(z) \\ 03.S(z) \\ 02.S(z) \\ S(z) \end{bmatrix},$$

$$T_3 = \begin{bmatrix} S(z) \\ S(z) \\ 03.S(z) \\ 02.S(z) \end{bmatrix} \tag{1}$$

$$\begin{aligned} &T_0[s_0^{(r)}] \oplus T_1[s_5^{(r)}] \oplus T_2[s_{10}^{(r)}] \oplus T_3[s_{15}^{(r)}] \oplus [k_0^{(r)} k_1^{(r)} k_2^{(r)} k_3^{(r)}] \| \\ &T_0[s_4^{(r)}] \oplus T_1[s_9^{(r)}] \oplus T_2[s_{14}^{(r)}] \oplus T_3[s_3^{(r)}] \oplus [k_4^{(r)} k_5^{(r)} k_6^{(r)} k_7^{(r)}] \| \\ &T_0[s_8^{(r)}] \oplus T_1[s_{13}^{(r)}] \oplus T_2[s_2^{(r)}] \oplus T_3[s_7^{(r)}] \oplus [k_8^{(r)} k_9^{(r)} k_{10}^{(r)} k_{11}^{(r)}] \| \\ &T_0[s_{12}^{(r)}] \oplus T_1[s_1^{(r)}] \oplus T_2[s_6^{(r)}] \oplus T_3[s_{11}^{(r)}] \oplus [k_{12}^{(r)} k_{13}^{(r)} k_{14}^{(r)} k_{15}^{(r)}] \| \end{aligned} \tag{2}$$

$$\begin{aligned} c_{0-3} &= (T_2(s_0^{10}) \& \mathtt{0xff000000}) \oplus (T_3(s_5^{10}) \& \mathtt{0xff0000}) \\ &\oplus (T_0(s_{10}^{10}) \& \mathtt{0xff00}) \oplus (T_1(s_{15}^{10}) \& \mathtt{0xff}) \oplus [k_0^{10} k_1^{10} k_2^{10} k_3^{10}] \\ c_{4-7} &= (T_2(s_4^{10}) \& \mathtt{0xff000000}) \oplus (T_3(s_9^{10}) \& \mathtt{0xff0000}) \\ &\oplus (T_0(s_{14}^{10}) \& \mathtt{0xff00}) \oplus (T_1(s_3^{10}) \& \mathtt{0xff}) \oplus [k_4^{10} k_5^{10} k_6^{10} k_7^{10}] \\ c_{8-11} &= (T_2(s_8^{10}) \& \mathtt{0xff000000}) \oplus (T_3(s_{13}^{10}) \& \mathtt{0xff0000}) \\ &\oplus (T_0(s_2^{10}) \& \mathtt{0xff00}) \oplus (T_1(s_7^{10}) \& \mathtt{0xff}) \oplus [k_8^{10} k_9^{10} k_{10}^{10} k_{11}^{10}] \\ c_{12-15} &= (T_2(s_{12}^{10}) \& \mathtt{0xff000000}) \oplus (T_3(s_1^{10}) \& \mathtt{0xff0000}) \\ &\oplus (T_0(s_6^{10}) \& \mathtt{0xff00}) \oplus (T_1(s_{11}^{10}) \& \mathtt{0xff}) \oplus [k_{12}^{10} k_{13}^{10} k_{14}^{10} k_{15}^{10}] \end{aligned} \tag{3}$$

where c_{i-j} represents the i-th to j-th bytes of the ciphertext and k_i^r represents the i-th byte of the key in the r-th round.

By installing `OpenSSL` libraries, the address offset of each element in the lookup tables $T_0, ..., T_3$ which are sequential can be found in the `libcrypto.so` file. To the best of our knowledge, it is assumed in all proposed access-driven attacks that the attacker can find the corresponding address offsets in the aforementioned file (look for example [7, 8, 11, 12]).

2.2 Virtual Memory

In order to isolate different processes, processes nowadays referring to virtual addresses instead of directly referring to physical addresses [13]. The space of virtual memory addresses is divided into specific-capacity and fixed-capacity pages. Physical address space is divided into the page size with a specific capacity which equals to the virtual address pages. The size of pages partitioning the address space and translation scheme to convert a virtual address into a physical address is different in processors. Depending on the capacity of these partitioning pages, a certain number of the smallest bits of virtual addresses equals to the bits of physical addresses.

If the page size is l bytes, the first $\log_2 l$ bits of a virtual address equal to the first $\log_2 l$ bits of the corresponding physical address. The page size in most Intel processors is 4kB. In other words, the first 12 bits in the virtual and physical addresses are equal. As we discussed later, this feature is an effective factor in the performance of the proposed attack in this paper

2.3 Cache Memory Structure

In this section, we describe cache memory structure and introduce basic concepts which are needed in this paper. An interested reader can refer to [14] for more details. In modern processors, cache memory is used to eliminate the delay in access to the main memory. In case the processor finds the required data in the cache memory, cache hit occurs. Otherwise, a cache miss occurs which causes a fetch from the slow main memory. The underlying basis of cache-based attacks is the time difference between these two events. Modern processors mostly use multi-level cache memory which has a variety of capacities. Each level of the cache memory is organized into sets with each set is divided into several lines.

We enumerated the bits of the address from LSB to MSB and denote the m-bit address as $(r_{m-1}, \cdots, r_0)$. The i-th to j-th bits of the address correspond to the bits $(r_j, \cdots, r_i)$ where $0 \leq i < j \leq m-1$. The bits $(r_{n+\delta-1}, \cdots, r_\delta)$ determine the proper set for transferring single-byte data with m-bit address to a level of cache memory which consists of 2^n sets. Each line of a set has a capacity of 2^δ bytes. The bits $(r_{m-1}, \cdots, r_{n+\delta})$ indicates the corresponding tag.

2.3.1 Addressing Mode

Three different methods are used for computing the host set and tag of data for transferring it to the cache memory. The VIVT method uses the virtual address of the data to calculate the host set and tag. In the PIPT method, the physical address of the data is used to compute host set and tag. The VIPT method is a combination of the two previous cases and attempts to eliminate their defects. In this method, the virtual address of data is used to compute the host set and its physical address to calculate the tag of data.

The PIPT method is used to transfer the data to cache memory in the last level cache memory in Intel processors [13].

2.3.2 Replacement Policies

When the cache memory is full, there is no cache line available for new data from the main memory to be stored in the cache memory. Consequently, a cache memory line must be chosen for replacement to open up space. Various replacement policies have been implemented in the modern processors. Least recently used (LRU) is one of the widest policies used in all levels of Intel processors and upper levels of ARM processors. Based on this policy, a line from the set which is least recently used is selected to be discarded. One should note this feature is an influential factor in the success of novel attacks on Intel processors.

2.3.3 Inclusive Cache Memory

Intel processors have three levels of cache memory. Each core of the Intel processor has its own L1 and L2 levels while L3 level is shared by all cores. If all available blocks in the higher levels (L1 and L2) also present in the L3 level, then L3 level is called inclusive. The last level of cache memory is called exclusive to the higher levels if the blocks in L3 level do not present in the upper levels simultaneously. If the late level of cache memory is neither inclusive nor exclusive, it is called non-inclusive. All proposed attacks in this paper are performed on the last level of the cache memory as we exploit the property of inclusiveness in L3.

3 Access-Driven Attacks

The access-driven attack is a class of cache-based side-channel attacks in which the attacker is able to scrutinize the cache behavior granularly in order to detect whether a specific cache line has been evicted or accessed.

In recent years, an increasing amount of literature has focused on the access-driven attack [7, 8, 11, 15, 16]. Among others, the access-driven attacks being presented on modern processors can be divided into three major categories. In what follows, we briefly introduce the general procedure for applying these attacks on the first round of the AES cryptosystem.

- **Evict+Time:** In this attack [7], the attacker first causes the victim to run the AES encryption function for encrypting a specific plaintext under the unknown secret key. Then, he guesses a value for the most significant bitsof k_i where in what follows we denote by $< k_i >$. Based on the offset in `libcrypto.so`, the attacker calculates the virtual address corresponds to $T_j(x_i \oplus k_i)$ and *evict* it from the cache memory. Finally, he triggers second encryption with the same plaintext again and measure its time. If the attackers guesses $< k_i >$ correctly, due to the incidence of cache miss, the run time of crypto system is larger than a threshold.
- **Prime+Probe:** In this attack [7], the attacker guesses the value of $< k_i >$ and computes the

address of $T_j(x_i \oplus k_i)$ using the `libcrypto.so` file. After that, the attacker fills the calculated set by the data having a congruent address to the address of the predicted block. Ultimately, he measures the runtime for accessing the data transferred to the host set after running a cryptosystem with a specific input. If the key is correctly guessed, the data transferred in the first stage of the attack will be evicted by running the cryptosystem. Consequently, the access time will be significantly larger due to the occurrence of cache-miss.

- **Flush+Reload:** In this attack [8], the attacker flushes desired memory lines which are allocated for specific data in T-tables using the `clflush` instruction. Then, he attempts to extract the key by measuring the access time of flushed data after running the cryptosystem. The evicted data is returned to the cache memory by running the cryptosystem and its access time will be small because of the cache hit if the key is properly guessed. In this attack, both flushing and accessing stages require knowledge about the addresses of these blocks in their respective libraries.

4 New Access-driven Attack without Knowledge of Address Offsets

The section starts with a description of our method which expresses how to extract the δ-th to 11-th bits of the virtual addresses of the blocks in the T-tables where δ denotes the $\log_2$ of the line capacity as described inSection 2.3. Subsequently, we demonstrate all bits of the secret key in the AES cryptosystem can be extracted by utilizing the information obtained in the first part.

4.1 Extracting Partial Bits of Virtual Address

We aim to present a method to identify all or some bits of the virtual addresses of the blocks. First, we discuss the assumptions that are considered in our method. After that, we introduce a method to evict cache memory lines which are allocated to one of the blocks in T-tables without knowing the virtual address of any of the blocks in the look-up tables $T_0, ..., T_3$. Finally, we present our method in order to extract partial bits of virtual addresses of the T-tables blocks.

4.1.1 Assumptions:

The offset addresses of the elements in the T-tables are consecutively stored in the files related to the library used by the cryptosystem. Due to the addition of a constant value to each of these offsets to produce a virtual address in the processing of encryption for these blocks, the virtual addresses of the elements of these look-up tables are sequential. As it is mentioned before, we assume that ASLR is not used and consequently it does not affect the page addresses.

The four T-tables used in the AES cryptosystem consist of 1024 consecutive elements each of which includes 4 bytes. So the four T-tables totally consists of $4096 = 2^{12}$ bytes. Consequently, the first twelve bits of the virtual address of the elements in the T-tables are not equal to each other and its value is between 0 and 4095. On the other hand, the virtual memory of the processor we use supports 4KB pages. Consequently, the 12 least significant bits of the virtual address equal to the 12 least significant bits of the physical address which means the adversary knows these bits.

We assume that the type of data transfer to the last level of cache memory is PIPT method and the storage of virtual memory pages is 4kB. If an attacker knows the 12 least significant bits of the physical address for a given data, he can consider $2^{n+\delta-12}$ sets as potential candidates for hosting this data. For example, for a data with the virtual address of $\texttt{0X}a_7a_6a_5a_4a_3\texttt{0C0}$, if we assume $\delta = 6$ then the sets that can host this data are $\{3, 67, 131, ..., 4035\}$ if and the cache memory has 4096 sets with 2^δ =64-byte lines.

4.1.2 Eviction Policy

In this part, we show that how we can evict cache memory lines which are allocated to one of the blocks in T-tables without any explicit knowledge of address offsets.

We assume that all blocks in the T-tables are placed in the last level of the cache memory. We denote the number of cache lines in each set by w. By accessing the arbitrary data such that the 12 least significant bits of their virtual addresses are equal to each other, w cache lines in each candidate cache set must be replaced to evict a specific cache line which includes one of the blocks in the T-tales. As it is mentioned before, Some other host cache sets (with the same value for δ-th to 11-th bits in their page offsets) are also evicted as a side effect which does not affect our process.

To this aim, we need to specify the number of accesses (N) required to assign a minimum of w memory lines to $2^{n+\delta-12}$ cache sets which are candidate sets for hosting the desired block from the T-tables. The lower bound for the number of required data to be loaded in order to evict a cache line can be estimated approximately by the formula given in Equation 4.

$$N = 2^{2\cdot(n+\delta-12)} \cdot w \tag{4}$$

In this formula, the size of the virtual memory pages is considered as 4kB. We provide the proof in Appendix A. One should note, Equation 4 provides an approximation of the number of required data as we make some assumption in the computation of this formula. In particular, it is assumed that physical page addresses are chosen uniformly at random which is not necessary valid in practice. In order to find the accurate number of required data, one should test the process experimentally as we did in our experience.

The replacement policy in the cache memory of Intel processor is based on least recently used (LRU). Assigning at least w memory lines to each cache set leads to the evicting of all sets which are candidates for hosting the data.

4.1.3 Profiling Process and Extracting Virtual Addresses

In what follows we describe precisely how the virtual addresses correspond to the T-tables can be retrieved partially. The process is presented in Algorithm 1.

The attacker creates a matrix M with N virtual addresses which are equal in the 12 least significant bits. The attacker makes access to all elements of the matrix M (lines 2-4 in Algorithm 1). The δ-th to 11-th bits of the virtual addresses in the matrix M equal to the δ-th to 11-th in the virtual addresses of one of the blocks in the T-tables. As it is explained in Section 4.1.2, at least w random blocks are transferred to each of the $2^{n+\delta-12}$ candidate cache sets by accessing the elements of the matrix M which leads to evict cache memory lines which are allocated to one of the blocks in T-tables.

Then the encryption function of AES for a random plaintext x and a fixed key k chosen by the adversary is performed and the time of execution is measured (lines 5-8 in Algorithm 1). In order to measure the execution time, we use `rdtsc` command. Before and after the (complete 10-rounds) encryption process the time stamp is saved in t_1 and t_2, respectively. The execution time can be computed as $t_1 - t_2$.

For each of the values of the bytes x_0, x_1, x_2 and x_3 where x_i denotes the i-th byte of the plaintext x for $0 \le i \le 3$, we compute the summation of execution times in $L_b[x_b]$ and increase the corresponding counter $counter_b[x_b]$ by one (lines 9-12 in Algorithm 1). We remind that since the plaintexts are chosen by the attacker in the profiling phase, the values of x_0, x_1, x_2, and x_3 are known for the attacker. Finally, we compute the average time of the execution of the encryption system based on the values of the bytes x_0, x_1, x_2 and x_3 by computing $D_b = L_b/counber_b$ (lines 14-18 in Algorithm 1).

Algorithm 1 Timing profile for x_0 to x_3

Input: Matrix M with N virtual addresses which are equal in the 12 least significant bits.
Output: Average time for access x_0, x_1, x_2 and x_3.

```
1: for (number=0;number<required samples;number++) do
2:    for (j=0;j<N;j++) do
3:       maccess(*$M[j]$);
4:    end for
5:    $x \leftarrow$ random plaintext
6:    $t_1$=rdtsc()
7:    $AES_k(x)$
8:    $t_2$=rdtsc()
9:    for (b=0;b<3;b++) do
10:      $counter_b[x_b]$++
11:      $L_b[x_b] = L_b[x_b] + (t_2 - t_1)$
12:   end for
13: end for
14: for (b=0;b<3;b++) do
15:   for (j=0;j<256;j++) do
16:      $D_b[j] = \frac{L_b[j]}{counter_b[j]}$
17:   end for
18: end for
```

If the evicted cache line is accessed in the process of AES encryption, a cache miss occurs. Consequently, we expect that the access to a specific (but unknown) portion of one of the T-tables will be slower which depends on exactly one of the x_0, x_1, x_2, or x_3. In other words, we expect to observe higher average time for this access in the timing profile phase presented in Algorithm 1. The leap can be observed in a specific $D_{b'}$ for $2^{\delta-2}$ consecutive values $\{j', j'+1, ..., j'+2^{\delta-2}-1\}$ in which the $(10-\delta)$ most significant bits are equal (the $(\delta-2)$-th to 7-th bits are the same). For instance, Figure 1 illustrates the results of running Algorithm 1 on the Intel Core i5-4200U processor where $\delta = 6$.

By considering $< j' > \oplus < k_{b'} >$, the attacker can identify the memory line that is evicted in Algorithm 1 where $k_{b'}$ denotes the first round key for $b' \in \{0, 1, 2, 3\}$. In other words, the attacker can conclude that $(< j' > \oplus < k_{b'} >)$-th line of $T_{b'}$ is evicted. The δ-th to 11-th bits of the virtual address of the identified memory line are equal to the δ-th to 11-th bits of elements of the matrix M. As the blocks of T-tables are consecutive, the attacker can similarly identify the δ-th to 11-th of virtual addresses for other blocks.

4.2 Key-recovery Attack

In this section, we demonstrate how all bits of the secret key can be extracted by utilizing the information obtained from the profiling phase presented in Section 4.1.

As can be seen in Equation 3, the value of each byte of the ciphertext depends only on one of the T-tables in the last round. For instance, the bytes c_0, c_4, c_8, c_{12} in the ciphertext depend only on the look-up table T_2 in the last round. A closer look at the last round of AES indicates that any byte value of the ciphertext can be written as Equation 5.

$$c_i = S(s^{10}_{[i]}) \oplus k^{10}_i \tag{5}$$

where $S()$ denotes the Sbox used in AES and $[i]$ denotes the position of the byte c_i before the `SR` operation .

Algorithm 2 Extract the byte k^{10}_i

```
Input: Matrix $M_b$.
Output: Byte $k^{10}_i$.
 1: for (number=0;number <required samples;number++) do
 2:     $x$=random()
 3:     $AES_k(x)$
 4:     for ($j$=0;$j < N$;j++) do
 5:         maccess(*$M_b[j]$)
 6:     end for
 7:     $t_1$=rdtsc()
 8:     $c = AES_k(x)$
 9:     $t_2$=rdtsc()
10:     $L[c_i]$=L[$c_i$]+($t_2 - t_1$) %$c_i$ denotes the $i$-th byte of the
        ciphertext
11:     $Counter[c_i]$++
12: end for
13: for ($c_i$=0;$c_i$<256;$c_i$++) do
14:     $D[c_i]=\frac{L[c_i]}{Counter[c_i]}$
15: end for
16: $2^{\delta-2}$ maximum values for $c_i$ are larger than others. Store
    them as $\{m_1, m_2, \cdots, m_{2^{\delta-2}}\}$.
17: for ($j$=1;$j \le 2^{\delta-2}$;$j$++) do
18:     for ($\ell$=0;$\ell < 2^{\delta-2}$;$\ell$++) do
19:         $sk_i[m_j \oplus S(\ell)]$++
20:     end for
21: end for
22: return argmax$_k(sk_i(k))$
```

The first 12 bits of the virtual address of a specific block is determined by the method described in Algorithm 1. Using this information, the first 12 bits of virtual addresses for the first block of T_b can be found as the blocks are consecutive. Consequently, we can generate Matrix M_b which contains the virtual addresses that accessing them cause to transfer at least w random blocks to the candidate cache sets which can host the first block from T_b where T_b is the lookup table applied to the state byte $s^{10}_{[i]}$.

Algorithm 2 presents the process of the key-recovery attack for obtaining an arbitrary byte of k^{10}_i. First, a randomly generated plaintext x is encrypted by the AES (lines 2-3 in Algorithm 2). After that, the elements of the matrix M_b are accessed (lines 4-6 in Algorithm 2). After that, the encryption function $AES_k(x)$ is performed again and the time of execution is measured (lines 7-9 in Algorithm 2). In order to measure the execution time, we use `rdtsc` command. Before and after the encryption process the time stamp is saved in t_1 and t_2, respectively. The execution time can be computed as $t_1 - t_2$. The summation of execution times for each value of c_i is computed as $L[c_i] = L[c_i] + (t_2 - t_1)$ and increase the corresponding counter $Counter[c_i]$ by one (lines 10-11 in Algorithm 2). In the next step, the average runtime of the encryption is computed for all 256 values of c_i (lines 13-15 in Algorithm 2).Depending on the capacity of the cache memory line, the average runtime calculated for $2^{\delta-2}$ values (out of 256 values) for c_i is notably greater than other. We denote such maximum values for c_i by $\{m_1, m_2, \cdots, m_{2^{\delta-2}}\}$. The likelihood that a specific T-table memory line does not access during the first nine rounds of the encryption process and it is accessed during the last round is $(1 - \frac{t}{256})^{9 \cdot 4} \times (\frac{4t}{256})$ where t denotes the entries per each line. If we consider $t = 16$, this probability equals to 2.44%. Consequently, we expect for same cases that the first block of T_b is reloaded by the AES encryption process during the calculation of the byte c_i when c_i equals to one of the values $\{m_1, \cdots, m_{2^{\delta-2}}\}$. The probability that a specific T-table memory line does not access during the encryption process is $(1 - \frac{t}{256})^{10 \cdot 4}$ where t denotes the entries per each line. For $t = 16$ the probability equals to 7.5%. The probability that the observed cache miss happens in the last round is 2.44/7.5=32% which is notable. This can be recognized by sufficient amount of data.

For all possible values $0 \le \ell < 2^{\delta-2}$ which correspond to the first memory line of T_b and all values $\{m_1, \cdots, m_{2^{\delta-2}}\}$, we compute the distribution of $m_j \oplus S(\ell)$. Finally we return the argument of the maximum value as the correct key for the targeted byte k^{10}_i.

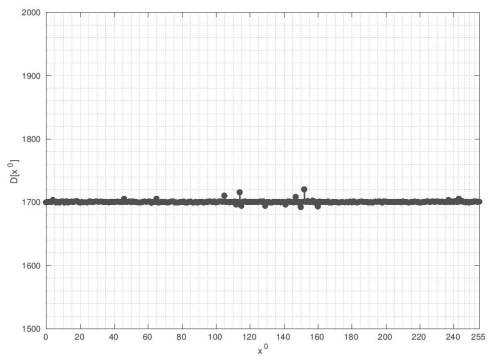
After retrieving all bytes of the last round, the attack can obtain the master key by considering the key schedule of AES. To make sure the key is correct, he should examine the obtained key for a pair of plaintext and corresponding ciphertext.

5 Experimental Verification

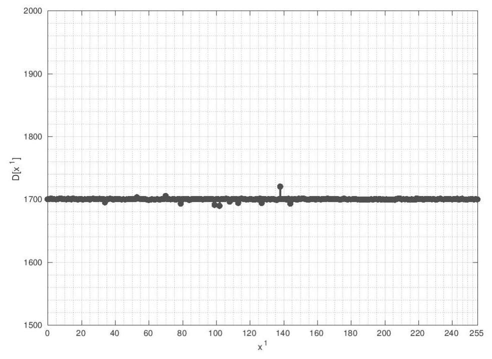
5.1 Experiment Setup and Results

To verify the theoretical model, we implemented the proposed attack on the AES-128 implemented in OpenSSL 1.1.0f library [1] . The attack was performed on a machine featuring an Intel Core i5-4200U which has a three-level cache architecture. The last level of the cache memory in this processor has $2^n = 2^{12} = 4096$ cache sets with each set is 12-way asso-

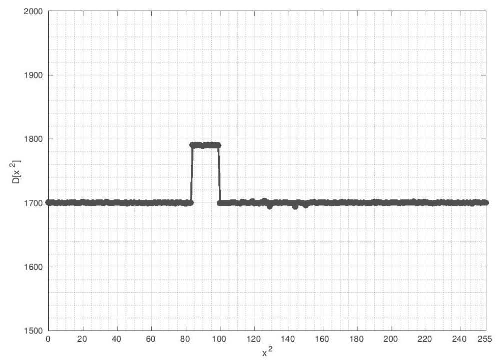
[1] Source code of the proposed attack can be found in https://github.com/V-H-M/NETT



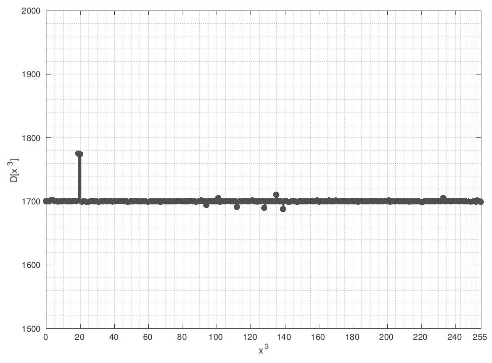
(a) Average time of execution based on x_0



(b) Average time of execution based on x_1



(c) Average time of execution based on x_2



(d) Average time of execution based on x_3

Figure 1. Timing profile phase

ciative lines with 64 bytes. The number of accesses in Algorithm 1 is considered $N = 2000$ to assign a minimum of w blocks to each candidate set. Matrix M includes the virtual addresses in the form of `0x`$a_7a_6a_5a_4a_3$`040`. The specified key which is used in the profiling phase for detecting the δ-th to 11-th bits of the corresponding blocks is selected as $k =$ `0x00,0x10,0x20,...,0xf0`. Figure 1 illustrates the experimental results obtained after performing Algorithm 1 on the processor. By considering the occurrence of the leap in $D_2[x_2]$ for $< x_2 >=$ `0x5` and $< k_2 >=$ `0x2`, the bits 6 to 11 of the virtual address of elements in the seventh block of table T_2 equal to $(000001)_2$. The attacker can calculate the bits 6 to 11 of the virtual address for the other blocks with the knowledge of bits 6 to 11 of the virtual address for the seventh block.

In order to perform the key-recovery attack presented in Algorithm 2, the attacker needs only the 6-th to 11-th bits of the virtual address of the first block in T_b. As the 6-th to 11-th of the virtual address of the seventh block in T_2 are `000001`, the virtual address of the first blocks in lookup tables T_0,...,T_3 equal to $(011011)_2$,$(101011)_2$,$(111011)_2$, and $(001011)_2$, respectively. The value N in Algorithm 2 was considered as 2000. The process was repeated for $500,000$ samples and all bits of the last round key could be extracted.

5.2 Comparison to Other Attacks

Our results and the previous results are summarized in Table 1.

It is popular in most of the literate to compare the attacks which are performed on different platforms. In order to have a better comparison, we performed the previous attacks [10, 11, 17] on the same machine that our attack is tested.

The Prime+Probe and Evict+Time attacks which are proposed in [11] targets OpenSSL's 0.9.8 version of AES. As it is mentioned in previous papers [10, 17], OpenSSL's 0.9.8 uses a separate T-table (T_4) for the implementation of the last round of AES. This is the reason that applying an attack on OpenSSl's 0.9.8 is notably easier due to the decrease of the noise. As it is discussed, our attack is applied on OpenSSl's 1.1.0f which is a harder target. Flush+Reload attack, as a more state-of-the-art approach, can be applied cross-VM environments while our attack is not applicable in VM setting. Similar to previous access-driven attacks, our attack is applicable under some circumstances which mean the attacker is permitted to measure the execution time of the victim process and also the execution time of its own. One important requirement of access-driven attacks is that the attacker is permitted to run a process in the processor of the victim [18]. However, as a main difference our attack is applicable based on the fewer assumptions in comparison to the previous attacks. We demonstrate that the Evict+Time attack can be performed on Intel x86 CPUs without any privilege of knowing address offsets.

Table 1. Comparison of cache-based side-channel attacks against AES in the only-ciphertext scenario. †: Experimental results obtained by executing the previous attacks on OpenSSl 1.1.0f and our platform (i5-4200u).

Attack	Source	Platform	Traces	OpenSSL	use of offset
Prime+Probe	[11]	Pentium 4E	16000	0.9.8a	Yes
Evict+Time	[11]	Athlon64	500000	0.9.8a	Yes
Evict+Time†	[11]	i5-4200u	500000	1.1.0f	Yes
Flush+Reload	[17]	i5-3320M	100000	1.0.1f	Yes
Flush+Reload†	[17]	i5-4200u	50000	1.1.0f	Yes
Flush+Reload	[10]	i5-2430M	25000	1.0.1g	Yes
Flush+Reload†	[10]	i5-4200u	20000	1.1.0f	Yes
Evict+Time	This paper	i5-4200u	500000	1.1.0f	No
Cross-VM Attacks:					
Flush+Reload	[17]	i5-3320M	400000	1.0.1f	Yes
Flush+Reload	[10]	i5-2430M	30000	1.0.1g	Yes

6 Conclusion

In this paper, we demonstrated how to perform Evict+Times attack on AES without explicit knowledge about the address offsets. The proposed attack is applicable on most of the Intel processors under the circumstances that the last level of cache memory uses LRU replacement policy and it is inclusive with respect to the upper levels of cache.

Acknowledgment

The authors wish to thank the reviewers for the detailed comments and insightful suggestions that significantly improved the quality of this paper.

References

[1] Onur Aciiçmez, Werner Schindler, and Çetin Kaya Koç. Cache Based Remote Timing Attack on the AES. In Masayuki Abe, editor, *CT-RSA 2007*, volume 4377 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2006.

[2] Daniel J. Bernstein. Cache-timing attacks on AES, 2005. `http://cr.yp.to/papers.html/cachetiming`.

[3] Joseph Bonneau and Ilya Mironov. Cache-Collision Timing Attacks Against AES. In Louis Goubin and Mitsuru Matsui, editors, *CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2006.

[4] Cédric Lauradoux. Collision attacks on processors with cache and countermeasures. In Christopher Wolf, Stefan Lucks, and Po-Wah Yau, editors, *WEWoRC 2005*, volume 74 of *LNI*, pages 76–85. GI, 2005.

[5] Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. A refined look at Bernstein's AES side-channel analysis. In Ferng-Ching Lin, Der-Tsai Lee, Bao-Shuh Paul Lin, Shiuhpyng Shieh, and Sushil Jajodia, editors, *ASIACCS 2006*, page 369. ACM, 2006.

[6] Onur Aciiçmez and Çetin Kaya Koç. Trace-Driven Cache Attacks on AES (Short Paper). In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS 2006*, volume 4307 of *Lecture Notes in Computer Science*, pages 112–121. Springer, 2006.

[7] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In David Pointcheval, editor, *CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[8] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In Kevin Fu and Jaeyeon Jung, editors, *23rd USENIX Security Symposium*, pages 719–732. USENIX Association, 2014.

[9] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[10] Berk Gülmezoğlu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic flush+ reload attack on aes. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 111–126. Springer, 2015.

[11] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology*, 23(1):37–71, 2010.

[12] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 549–564. USENIX Association, 2016.

[13] Michael Schwarz. Software-based side-channel attacks and defenses in restricted environments. 2019.

[14] Chester Rebeiro, Debdeep Mukhopadhyay, and Sarani Bhattacharya. *Timing channels in cryptography: a micro-architectural perspective*. Springer, 2014.

[15] Gorka Irazoqui Apecechea, Thomas Eisenbarth,

and Berk Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES. In *IEEE Symposium on Security and Privacy 2015*, pages 591–604. IEEE Computer Society, 2015.

[16] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy 2015*, pages 605–622. IEEE Computer Society, 2015.

[17] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a Minute! A fast, Cross-VM Attack on AES. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *RAID 2014*, volume 8688 of *Lecture Notes in Computer Science*, pages 299–319. Springer, 2014.

[18] Yangdi Lyu and Prabhat Mishra. A survey of side-channel attacks on caches and countermeasures. *Journal of Hardware and Systems Security*, 2(1):33–50, 2018.

Vahid Meraji is a master of science. He received his M.S. in secure communications and cryptography from Shahid Beheshti University, Tehran, Iran, in 2018. His main research interests are side-channel attacks and micro-architectural side-channel attacks.

Hadi Soleimany is an assistant professor at Cyberspace Research Institute at Shahid Beheshti University, Iran, since 2015. He received his Ph.D. in theoretical computer science from Aalto University, Finland, in 2015. He was a postdoctoral researcher at Technical University of Denmark (DTU), Denmark, in summer 2016 and 2017. His main research interests are practical aspects of cryptography.

A Appendix

Lemma 1. *The number of nonnegative integer solutions of the equation* $x_1+x_2+\cdots+x_r=n$ *is* $\binom{n+r-1}{r-1}$.

Lemma 2. *The number of nonnegative integer solutions of the equation* $x_1+x_2+\cdots+x_r=n$ *with the condition* $x_i \geq m$ *for* $1 \leq i \leq r$ *is* $\binom{n+r-m\cdot r-1}{r-1}$.

Corollary 1. *Let us assume that for nonnegative integers* $x_i \in \mathbb{Z}$ *which are chosen randomly, the equation* $x_1+x_2+\cdots+x_r=n$ *holds where* $n \in \mathbb{Z}$. *The probability that the condition* $x_i \geq m$ *for all* $1 \leq i \leq r$ *is*

$$\frac{\binom{n+r-m\cdot r-1}{r-1}}{\binom{n+r-1}{r-1}}$$

Lemma 3. *The number of accesses required to assign a minimum of* w *blocks to* $2^{n+\delta-12}$ *candidate cache sets for hosting each block from the T-tables is* $N = O(2^{2\cdot(n+\delta-12)}\cdot w)$.

Proof. The probability of transferring w blocks to $2^{n+\delta-12}$ candidate cache sets by accessing N elements equals can be deduced from corollary 1. This probability is presented in the following equation:

$$p = \frac{\binom{N+2^{n+\delta-12}-w\cdot 2^{n+\delta-12}-1}{2^{n+\delta-12}-1}}{\binom{N+2^{n+\delta-12}-1}{2^{n+\delta-12}-1}} \tag{A.1}$$

We aim to choose N large enough such that the probability p becomes high enough. In what follows, we compute the formula given for p in Equation A.1.

Stirling's formula yields the approximation $\binom{n}{r} \simeq (e\cdot\frac{n}{r})^r$ when n and r are sufficiently large. Consequently, we obtain the following equation:

$$p \simeq \frac{e\cdot(\frac{N+2^{n+\delta-12}-w\cdot 2^{n+\delta-12}}{2^{n+\delta-12}})^{2^{n+\delta-12}}}{e\cdot(\frac{N+2^{n+\delta-12}}{2^{n+\delta-12}})^{2^{n+\delta-12}}}$$

$$= (1-\frac{w\cdot 2^{n+\delta-12}}{N+2^{n+\delta-12}})^{2^{n+\delta-12}} \simeq (1-\frac{1}{\frac{N}{w\cdot 2^{n+\delta-12}}+\frac{1}{w}})^{2^{n+\delta-12}}$$

$$\simeq (1-\frac{1}{\frac{N}{w\cdot 2^{n+\delta-12}}})^{2^{n+\delta-12}}$$

As we know $(1-\frac{1}{\alpha})^\alpha \simeq e^{-1}$ for large values of α, if we consider N such that $\frac{N}{w\cdot 2^{n+\delta-12}} = 2^{n+\delta-12}$ holds then we have:

$$(1-\frac{1}{\frac{N}{w\cdot 2^{n+\delta-12}}})^{2^{n+\delta-12}} \simeq e^{-1} \tag{A.2}$$

Consequently it is sufficient to choose $N = O(2^{2\cdot(n+\delta-12)}\cdot w)$. □