Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2018) 26: 1953 – 1966

doi:10.3906/elk-1711-359

Cache-timing attacks without a profiling phase

Ali Can ATICI*, Cemal YILMAZ, Erkay SAVAŞ
Department of Science and Engineering, Faculty of Engineering and Natural Sciences, Sabancı University, Orhanlı, Tuzla, Turkey

Received: 26.11.2017 • **Accepted/Published Online:** 03.05.2018 • **Final Version:** 27.07.2018

Abstract: Theoretically secure cryptographic algorithms can be vulnerable to attacks due to their implementation flaws. Bernstein's attack is a well-known cache-timing attack that uses execution times as the side-channel. The major drawback of this attack is that it needs an identical target machine to perform its profiling phase where the attacker models the cache timing-behavior of the target machine. This assumption makes the attack unrealistic in many circumstances. In this work, we present an effective method to eliminate the profiling phase. We propose a methodology to model the cache timing-behavior of the target machine by trying hypothetical cache behaviors exhaustively. Our implementation results show that the proposed nonprofiled Bernstein's attack has comparable (and better in some test instances) performance to the original attack with the profiling phase.

Key words: Cryptography, side-channel analysis, cache-timing attacks

1. Introduction

Cryptographic algorithms that are secure against known theoretical attacks can still be vulnerable to side-channel analysis if they are not cautiously implemented. Execution time, power consumption, electromagnetic emission, etc. can be used as side-channel information [1–4]. In unprotected cryptographic implementations, the secret key directly affects the emitted side-channel information. Thus, observations made on these leaked data can eventually lead to the revelation of the secret key.

Side-channel attacks, which exploit the fact that microarchitectural resources, such as cache memory and branch prediction unit, are shared, are widely studied in the literature [4–6]. The cache access patterns of cryptographic programs can be exploited by the cache-based side-channel attacks. Bernstein's attack [7] is a well-known cache-timing attack, which is applied in a client-server setting. The attack tries to infer the secret key, which resides in a server that employs the Advanced Encryption Standard (AES) [8] to encrypt incoming messages, by using the variations in the encryption times of randomly generated messages. A major drawback of Bernstein's attack is the necessity of having a computer system that is identical to the target system as the profiling phase of the attack needs to construct a model of the cache timing-behavior of the latter.

In this work we propose a methodology based on hypothetical modeling of the cache timing-behavior of a computer system and demonstrate that Bernstein's attack successfully recovers the key using one of the models, which best represents its cache timing-behavior. In the proposed approach, all possible cache timing-behaviors of a computer system are analytically extracted and the one that gives the highest correlation to the measured attack data is chosen to be used in the cache-timing attack. This approach eliminates the need of a profiling

*Correspondence: alicana@sabanciuniv.edu

phase (i.e. the need for an identical target machine), which makes the attack more realistic and feasible in practice.

The rest of the paper is organized as follows: Section 2 presents related work; Section 3 provides brief background information on AES, Bernstein's attack, and a variation of Bernstein's attack on the last round of AES; Section 4 discusses the details of the profiling phase; Section 5 outlines the proposed approach; Section 6 explains how we conduct Bernstein's attack without the profiling phase in order to validate our hypothetical modeling methodology, and presents discussions on the results; Section 7 concludes the paper.

2. Related work

The first findings on cache-based side-channel attacks are reported in [1] and [9]. We can divide cache-based side-channel attacks into 3 categories: i) access-driven, ii) trace-driven, and iii) time-driven cache attacks.

Access-driven attacks exploit the information as to whether a cache line (or set) is accessed (or not) during a cryptographic operation to infer the secret key. In their proposed approach [10], Osvik et al. employ a spy process to determine the cache lines/sets that are used by a cryptographic application in a known plaintext and/or ciphertext setup. In trace-driven cache attacks (i.e. the second category of attacks), it is assumed that the adversary has full control over the target device and that they can determine whether a particular memory access is a miss or hit during the cryptographic operation by monitoring electromagnetic or power emissions of a cryptographic device. Trace-driven attacks are also investigated in detail [11]. In the last category of cache attacks, time-driven attacks measure the execution time of a cryptographic operation and exploit the timing variations in different runs with different plaintexts. The assumption is that the execution time of the operation is heavily affected by the memory access times due to cache misses. Thus, the variations in different runs of the cryptographic operation occur because of different number of cache hits and misses that are dependent on the secret keys and the plaintext. In [12], Tsunoo et al. use the time variations that occur during encryptions due to the cache misses as a result of s-box table lookup operations.

The majority of the cache attacks rely on the so-called *cache cleaning* operation via a spy process, which can be detected easily [13], that evicts all or a part of data of cryptographic process from the cache before the start of an encryption operation. Bernstein's attack, which can be categorized as a timing-based attack, is the only exception. In his experiments, Bernstein runs the attack on an AES server locally and reduces the key space considerably after measuring the execution times for 2^{30} sample plaintexts. The reasons that lead Bernstein's attack to succeed are further investigated in [14,15].

3. Preliminaries

3.1. Advanced encryption standard (AES)

AES is a symmetric-key block cipher algorithm with a block size of 128 bits. AES computations are performed in rounds. An initial AddRoundKey is the first computation of the encryption process. The following rounds perform SubBytes, ShiftRows, MixColumns, and AddRoundKey computations in the given order. However, MixColumns computation is not involved in the last round of AES.

There exists a fast AES implementation which does not perform the round computations separately. Instead, it combines 3 steps in a round, namely the SubBytes, ShiftRows, and MixColumns, into a lookup operation in AES acceleration (or lookup) tables. MixColums computations are not involved in the AES final round; thus a different table is required. The final round is implemented by only one separate table. In this work, we employ such a fast AES implementation from the OpenSSL library (v0.9.7a Feb 19, 2003).

3.2. Bernstein's cache-timing attack

Bernstein presents a cache-based timing attack, which targets the lookup table based OpenSSL implementation of AES. Two entities are involved in this attack: an *AES server* and an *AES client*. The AES server waits for the incoming encryption requests. When a request is received, it encrypts the message and sends back the ciphertext. The AES client sends randomly generated messages to the server and gets the corresponding ciphertext and measures the elapsed timing. The attack has 2 main phases: *profiling phase* and *attack phase*. In the profiling phase, the attacker uses an AES server, which is identical to the target server, to encrypt a large number of randomly generated plaintexts with a known key. The attacker obtains the execution time of each encryption and saves it along with the plaintext. Then, in the attack phase, the same operation is repeated, but this time on the target AES server using an unknown key. In the attack phase, the key is not known therefore, for each access in the first round, all possible key byte values are tried and a timing profile of indexes are obtained for each key candidate. Then each of the timing profiles in the attack phase is correlated with the timing profile obtained in the profiling phase. The key value giving the highest correlation is the most likely key candidate. The details of the attack can be found in [7,14].

In Bernstein's attack, the profiling phase tries to model the cache timing-behavior of the target system. The attack needs no spy process to artificially evict cache lines holding lookup table entries, but rather relies on naturally occurring evictions, if any [15]. Moreover, no specific knowledge about the target system is required, since the attack needs nothing other than the timing information. Thus, Bernstein's attack is generic and can be applied to all similar systems.

3.3. Applying Bernstein's attack to the last round of AES

There exists a simpler attack on the final round of AES, which can be applied on the AES implementation employed in [7] and [14] (OpenSSL v0.9.7a). With this version of the attack, the whole key can be extracted. In this paper, we employ the last round attack, which allows us to extract the 128 bits of the key [15].

In the last round of AES, a separate table, namely $T4$, is used, which basically implements the AES SubBytes operation. The outputs of $T4$ lookup operations (i.e. $T4[s_i^9]$, where s_i^9 is the lookup index of round 10 and $i = 0, 1, \ldots, 15$), are used as indexes to obtain the aforementioned statistical models. In the profiling phase, the outputs of $T4$ lookups used in the last round can be computed using the formula

$$InvShiftRows\left(c_i \oplus k_i^{10}\right), \tag{1}$$

where c_i and k_i^{10} stand for the i^{th} bytes of the ciphertext and the 10^{th} round key, respectively, and $i = 0, 1, \ldots, 15$. As both the key and the ciphertext are known in the profiling phase, we can obtain a timing profile based on the output values of $T4$ lookup operations. Timing profiles can be represented as an array of $T_i^p[256]$, where i is the order of the $T4$ access and $i = 0, 1, \ldots, 15$.

In the attack phase, the secret round key byte $\tilde{k}_i^{10}$ is unknown; thus we obtain one timing profile for each candidate of the corresponding key byte using $InvShiftRows(c_i^{'} \oplus k)$ for $k = 0, 1, \ldots, 255$, namely $T_{i,k}^a[256]$. Then the timing profiles in the attack phase $T_{i,k}^a$ are correlated to the timing model of the profiling phase T_i^p. The key value k that yields the highest correlation is chosen as the most likely candidate for the key byte $\tilde{k}_i^{10}$.

4. A closer look at the profiling phase

In Section 3, we have outlined 2 attacks. However, they have both needed a profiling phase. It is now crucial to understand what we achieve after the profiling phase. Unintentional collisions in cache lines holding the AES lookup tables cause variations in access times due to cache misses. The profiling phase aims to obtain data about cache timing-behavior of AES by registering the variations in cache line access times. Cache timing-behavior of AES can be expressed as a timing model for each of the 16 $T4$ accesses in the last round. Since we know the secret key in the profiling phase, the timing model for the i^{th} access in the last round is simply a histogram of average execution times of an AES encryption indexed by output bytes of $T4$ as computed in (1). Figures 1a and 1b illustrate actual timing models for 10^{th} and for 12^{th} accesses to $T4$ respectively, when the profiling phase is executed on a computing platform with Intel Pentium P6200 CPU running Ubuntu 3.0.0-12 kernel. Here, the inverse s-box operation is also applied to the models to enhance visual clarity. Thus, the x-axis shows the byte indexes (s_i^9) used in accesses to table $T4$.

In Figures 1a and 1b, the timing measurements are either above or below the average execution time. Here the measurements above the average can be attributed to cache misses in the corresponding cache lines. Furthermore, the execution times tend to remain above or below the average line for a group of consecutive index values. This particular pattern can be explained by the fact that a cache line holds 16 of the $T4$ entries. Thus a collision in a cache line will naturally affect the access times of 16 entries. To summarize, at the end of the profiling phase, we obtain a timing model T_i^p for the i^{th} access in the last round, which is just an array of 256 average execution times of AES.

In the attack phase, the timing measurements are obtained, grouped, and averaged depending on the values of the ciphertext byte involved in the i^{th} output of the $T4$ lookup operation, as the corresponding key byte value is unknown. The result is cache timing-behavior model $\tilde{T}_i^a$, which is again an array of 256 average execution times. Then, the 2 timing models, namely T_i^p and $\tilde{T}_i^a$, are correlated. As T_i^p is indexed by $T4$ output values and $\tilde{T}_i^a$ is indexed by ciphertext byte values, we transform the latter into 256 timing models, $T_{i,k}^a$ indexed by the $T4$ output values by applying an exhaustive search on the key space of $k \in [0, 255]$. Actual correlations are computed between T_i^p and $T_{i,k}^a$, and the key values with low correlations are eliminated. The remaining keys, sorted from highest to lowest correlation, are expected to be few, resulting in a significant reduction in the key space if the attack is successful. The essential steps of the last round attack with profiling phase are given in Algorithm 1, where $T^p = \bigcup_{i=0}^{15} T_i^p$ and $T^a = \bigcup_{i=0}^{15} \tilde{T}_i^a$.

The last round attack can reveal the entire key, although it still needs a profiling phase. It is not an easy task for an attacker to setup an identical platform and to run the profiling phase. It is also pointed out in [10] and [16] that accessing an identical machine or reproducing the machine-specific cache effects may not be feasible.

To increase the feasibility and applicability of the attack, we present a novel methodology that needs neither an identical target system nor a profiling phase. We use hypothetical modeling to obtain the timing-behavior of the cache and need only the size of the lookup tables and the cache line size of the computing platform.

Algorithm 1 Attack with profiling phase

Require: T^p and T^a
Ensure: K_R: Ordered reduced key space

$K \leftarrow \emptyset$
$K_R \leftarrow \emptyset$
for $i = 0$ to 15 **do**
    for $k = 0$ to 255 **do**
        $T^a_{i,k} \leftarrow$ Transform ($\tilde{T}^a_i$, k)
        $\gamma \leftarrow$ Correlate ($T^a_{i,k}$, T^p_i)
        $\nu \leftarrow$ Variance ($T^a_{i,k}$, T^p_i)
        $K[i] \leftarrow K[i] \cup (k, \gamma, \nu)$
    end for
    $K[i] \leftarrow$ Sort ($K[i]$) ▷ Descending on γ
    $\delta \leftarrow$ Threshold ($K[i]$)
    $K_R[i] \leftarrow$ Reduce ($K[i]$, δ)
end for

5. Simplified cache timing model

In this section, we introduce a methodology to model the timing characteristics of the data cache for a running program on a CPU. As far as the cache attacks are concerned, a simple model that is based on variations in cache line/set access times can be used to capture the cache timing-behavior. The proposed methodology does not aim to capture all the complicated structural properties of a modern cache. On the contrary, it aims to extract a generic cache timing-behavior model based on a simplified set of assumptions. The proposed model requires minimum knowledge (i.e. cache line size) about the target system. Although the timing model is obtained using simplified assumptions, it is shown by our experimental results that it can still be used effectively to conduct attacks. This implies that our simplified model can be extended and improved to cover real-world computing platforms. Next, we provide a more formal explanation of our model for data cache timing-behavior:

Definition 1 ***Data*** *in the data cache of a CPU are composed of individual bytes. Elements of* ***data*** *are individually accessible by data indexes.*

Assumption 1 ***Data*** *in the cache are aligned and occupy a number of consecutive cache lines (unfragmented). The first byte of* ***data*** *is always placed in a new cache line.*

Assumption 2 *The direct-mapping is used as the cache placement strategy, where a single cache line can be considered as a cache set.*

Assumption 3 *Parts of* ***data****, essentially a sequence of bytes, can be accessed simply by indexing. Each index value points to an equal number of bytes.*

Assumption 4 ***Data*** *are the relevant part of the code that cause cache hits/misses when accessed. Accessing* ***data*** *in the cache (i.e. a cache hit) and* ***data*** *not in the cache (i.e. a cache miss), take* t *and* $\mathrm{t} + \Delta$*, respectively, and we always have* $\Delta > 0$*.*

Assumption 5 *Cache collisions may occur between 2 different programs or within the same program; i.e. data sharing the same cache lines/sets can evict each other. During the run of a program, collisions occur always on the same cache lines/sets.*

Assumption 6 *A cache collision in a cache line evicts the entire block from the cache and brings a new block from the memory.*

Assumption 7 *During a single run of a program,* ***data*** *are accessed only once, which means the program observes only one hit or one miss during the run.*

Assumption 8 *The execution times of a program in the presence of hits and misses are* $\mathrm{t_h}$ *and* $\mathrm{t_m}$*, respectively. And,* $\mathrm{t_h}$ *and* $\mathrm{t_m}$ *have equal probability to occur.*

Based on these assumptions, we obtain several immediate results, captured as propositions.

Proposition 1 *Following Assumption 1 and Assumption 3, the total number of cache lines occupied by* ***data*** *can be calculated as*

$$\left\lceil \frac{|data|}{b} \right\rceil , \tag{2}$$

where $|data|$ *and* b *stand for the number of bytes in* ***data*** *and in a cache line, respectively.*

Proposition 2 *Following Assumptions 4, 7, and 8, we can approximate* $\mathrm{t_h}$*,* $\mathrm{t_m}$*, and* $\mathrm{t_a}$ *of a program with*

$$t_h = t + t_f \, , \tag{3}$$

$$t_m = (t + \Delta) + t_f \, , \tag{4}$$

$$t_a = (t_h + t_m)/2 \, , \tag{5}$$

where t_a *is the average execution time of a program,* t_f *is the execution time of instructions that do not require memory access. As* $\Delta > 0$*, we have* $t_h < t_a < t_m$*. This result implies that a particular execution time of a program will tend to be higher than the average execution time of that program* (t_a)*, when the program accesses the cache lines that are subject to collisions, and vice versa.*

Proposition 3 *Let the cache line index range* $[c_1, c_2]$*, where* $c_2 > c_1$ *and* $c_1 c_2 \geq 0$*, represent the indexes where cache lines are in collision. Taking the Assumptions 2, 5, and 6 into account, we can calculate the range of* ***data*** *indexes that maps to the colliding cache lines. Let* κ *be the number of bytes accessed by each* ***data*** *index. Then all* ***data*** *indexes within the following range are mapped to the cache lines that are in collision:*

$$\left[\frac{c_1 . b}{\kappa}, \frac{c_2 . b}{\kappa} + \frac{b}{\kappa} - 1 \right] \tag{6}$$

Here the cache line and ***data*** *indexes start from 0 (i.e. the first* b *bytes of the* ***data*** *reside in the* 0^{th} *cache line, second* b *bytes reside in the* 1^{st} *cache line, etc.).*

Algorithm 2 Modeling the Cache Timing-Behavior

Require: $data$, m, S_c, b, κ
Ensure: T: Cache timing-behavior model

1: I_c = MissDataIndex(S_c, b, κ) ▷ Proposition (3)
2: **for** $s = 0$ to $m - 1$ **do** ▷ for each **data** index
3: **if** $s \in I_c$ **then**
4: $T[s] = 1$
5: **else**
6: $T[s] = -1$
7: **end if**
8: **end for**

Based on these assumptions and propositions, an algorithm can be given to extract a timing model of the cache memory. Algorithm 2 describes the steps to obtain a model for given **data**.

In Algorithm 2, m is the number of indexes that are used to access **data** parts of κ bytes, b is the number of bytes in one cache line, and S_c denotes the subset of cache lines subject to collisions (i.e. contention set). The algorithm returns a timing model T, where each value of the index used to access **data** is matched with a timing value. Line 1 of Algorithm 2 calculates the set of **data** indexes that result in cache misses and Line 3 of Algorithm 2 checks whether a data index is in set I_c. In the case the referenced index causes a miss, the access to the corresponding data part will take longer. In this model, we assume $t_a = 0$, $t_h = -1$, and $t_m = 1$ for simplicity.

6. Practical application of the proposed methodology and validation results

6.1. Cache-timing attacks without a profiling phase

The attack without the profiling phase needs only the knowledge of the cache line size of the target computer and the sizes of the lookup tables. A typical cache line size is 64 B in the majority of contemporary computers and the AES lookup tables and their sizes can be obtained by examining the source code of the implementation.

Since we perform the last round attack, the **data** are table $T4$ of 1024 B, which is used only in the last round of AES encryption. It has 256 indexes and each index is used to access a 4 B entry. As all our target platforms have cache line sizes of 64 B, table $T4$ occupies 16 cache lines. The correct timing model of the cache can be obtained only if we know the cache lines subject to eviction due to collisions. However, without an identical computer system on which AES runs with a known key, we infer no information about the contention set and therefore the cache timing-behavior cannot be obtained.

On the other hand, in our simplified approach, we have only a total of 2^{16} simplified models as $T4$ occupies 16 cache lines. Thus, a brute-force approach, based on trying all simplified models exhaustively, is feasible.

To form our simplified models, we need to find the cache contention sets. As there are 2^{16} simplified models (i.e. 2^{16} cache configurations of hits and misses), we can use 16-bit integers that take values in $\left[0, 2^{16} - 1\right]$ to represent these models. For instance, the index value of 0x7FFF in hexadecimal representation indicates that the first cache line is in the contention set, assuming that each bit of an index stands for a cache line and the bit value of 0 indicates a collision in the corresponding cache line. Algorithm 3 explains how the cache contention sets are derived. It takes an index and iterates through its bits starting from the rightmost bit, which corresponds to the last cache line.

Algorithm 3 Obtaining a Cache Contention Set

Require:
l : index of simplified cache timing model
n : number of cache lines occupied by **data**
Ensure: S_c : Cache contention set for index l

1: $S_c \leftarrow \emptyset$
2: **for** $i = n - 1$ to **do**
3: **if** ($l \, mod \, 2 == 0$) **then**
4: $S_c \leftarrow S_c \cup i$
5: **end if**
6: $l \leftarrow l/2$
7: **end for**

Finally, Algorithm 4 gives us the most possible cache timing-behavior model given the timing measurement data from the attack phase (i.e., T^a).

Algorithm 4 Searching For Cache Timing-Behavior Model

Require:
$T4$: Lookup table
m : Index count of $T4$
κ : Size of each $T4$ entry in number of bytes
b : Size of each cache line in number of bytes
T^a : Timing model in attack phase
n : Number of cache lines occupied by $T4$
δ : Correlation threshold
Ensure: $\tilde{T}^h$: Correct cache timing-behavior model

1: $M \leftarrow \emptyset$
2: **for** $l = 0$ to 15 **do**
3: $S_c \leftarrow$ Algorithm 3 (l, n)
4: **for** $j = 0$ to 15 **do**
5: $T_l^h[j][:] \leftarrow$ Algorithm 2 ($T4$, m, S_c, b, κ)
6: **for** $i = 0$ to 255 **do**
7: $x \leftarrow$ AES-sbox(i)
8: $T_{tmp}^h[j][x] \leftarrow T_l^h[j][i]$
9: **end for**
10: $T_l^h[j][:] \leftarrow T_{tmp}^h[j][:]$
11: **end for**
12: $K_R \leftarrow$ Algorithm 1 (T_l^h, T^a)
13: $M \leftarrow M \cup (T_l^h, |K_R|)$
14: **end for**
15: $M \leftarrow$ Sort (M) ▷ Ascending on $|K_R|$
16: $\tilde{T}^h \leftarrow M[0][0]$

Algorithm 4 iterates through all simplified cache timing models; it first finds the corresponding contention set in line 3, then calculates the corresponding simplified model in line 5, and applies the AES s-box operation on the model in lines 6–10 since we perform the last round attack using the outputs of table T4. Then the attack phase in Algorithm 1 is applied to find the size of the reduced key space in line 12. The sizes of the reduced key space for simplified models are saved as described in line 13. Finally, they are sorted from smallest

to largest (line 15) and the simplified model with the smallest reduced key space size is chosen as the most probable cache timing-behavior model (line 16). Once we obtain the model, we can run Algorithm 1 and find the key bytes.

In order to test Algorithm 4, we ran it for the example data set[1] in Figures 1a and 1b, and obtained the index 14433 as the most probable cache timing model (i.e. the cache contention set). When we plot this model according to Algorithm 2, we obtain Figure 2. A closer look at Figure 2 reveals that our simplified model resembles the real model previously depicted in Figure 1b. Figure 2 shows behavior similar to that of Figure 1b. As explained for Figures 1a and 1b in Section 4, we can observe that consecutive groups of cache lines are either in the contention set or not, which is consistent with the real model that is based on the measurements.

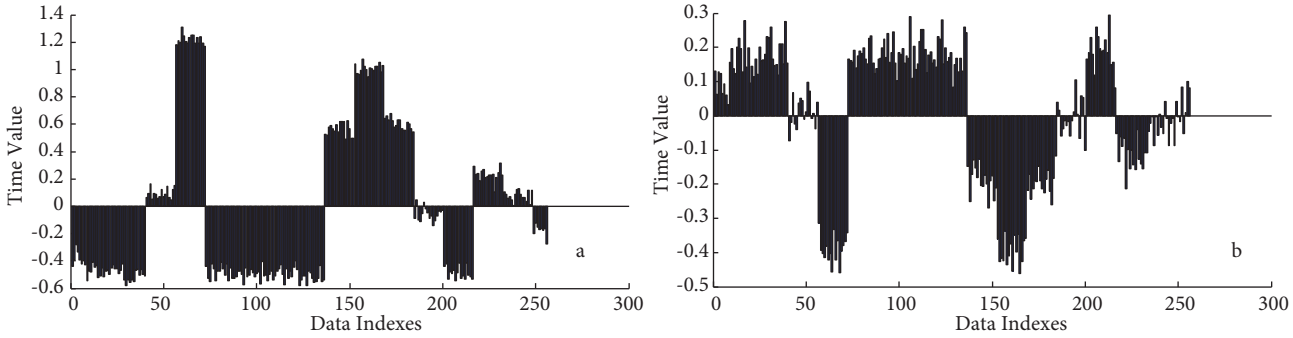

Figure 1. Profiling phase models. (a) Byte 10 Model, (b) Byte 12 Model.

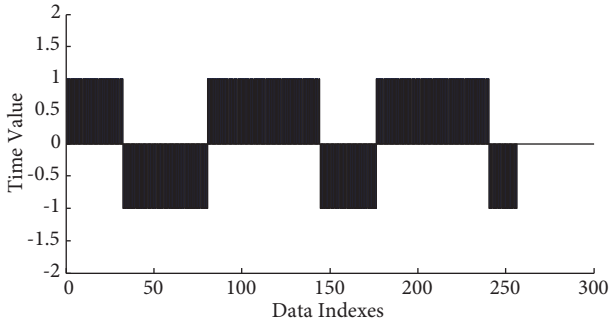

Figure 2. Calculated profiling phase model.

6.2. Implementation results

We executed the last round attack with and without a profiling phase on various software and hardware setups with different client-server deployment configurations. Each attack is conducted by the AES client with 2^{30} randomly generated messages, where the size of each message is 600 B. The target of the attack, the AES server, runs the table-based OpenSSL (v0.9.7a) implementation of AES, which is employed originally by Bernstein. In the attacks with a profiling phase, 2 separate measurements are used while in the attacks without a profiling phase only attack phase measurements are used. The attacks generated a candidate set for each key byte value,

[1] The C and MATLAB codes as well as the example data set used in the paper can be found at https://github.com/alicanatici/cacheAttackWithoutProfilingPhase

which are sorted by their likelihood. By multiplying the sizes of the candidate sets we obtained the reduced key space size for the whole AES key, which necessitated an exhaustive search. For each attack setup, we conducted a number of attacks and calculated the average of the results. In the Table we prefer to list only the average values after we conducted a number of attacks for each case as enumerating the results of each attack individually would result in a table which is difficult to interpret. Furthermore, in some of our setups, we observed large discrepancies between the best and worst case results, which we think would misguide the reader. In the first 2 columns of the Table, the attack configurations (i.e. the CPU type and the client-server deployment setup) are given. The last 2 columns present the average sizes of the reduced key spaces obtained after the attacks have been applied. Ubuntu (Linux kernel 3.0.0) is used as the operating system in the first 4 rows' experiments and CentOS (Linux kernel 2.6.18) is used for the last 2 rows.

Table. Attack results.

Processor	AES client-server deployment	Reduced key space with a profiling phase	Reduced key space without a profiling phase
Intel Pentium P6200	same core	2^{32}	2^{32}
Intel Pentium P6200	different core	2^{49}	2^{37}
Intel C2Duo P8400	same core	1	2^{12}
Intel C2Duo P8400	different core	2^{24}	2^{29}
Intel Xeon E5405	same core	2^{34}	2^{19}
Intel Xeon E5405	different core	2^{51}	2^{16}

In our experiments, we always found the unknown AES key in the reduced key space. We also observed that for all of the attack configurations the sizes of the reduced key spaces were always within feasible limits for an exhaustive search. An in-depth analysis of the results further reveals that the attack performed better in the majority of the cases when the client and server were located in the same core. Since cache collisions are caused by the program itself [15] and by other programs [14] sharing the same cache lines, when the 2 programs reside in the same core, more cache collisions occur and the attack performs better. The results also show that the performance of the proposed attack without a profiling phase is comparable to that of the original attack. We further observed performance gains in some of the cases, specifically in rows 2, 5, and 6. If we check the sizes of the "reduced key space with a profiling phase" in these rows, we see that they are larger compared to the rest of the results. This implies that the profiling and/or attack phase measurements are noisy and this situation degrades the performance of the attack with a profiling phase. However, when we apply our proposed approach on these cases, we observe a performance gain. This gain is possibly due to the fact that our simplified models are noise free and specifically selected for the attack phase data. Thus, we can claim that our proposed methodology gives better results under noisy conditions.

Experiment results show that it is possible to conduct successful cache-timing attacks with the proposed simplified model. On the other hand, how such successful results are achieved with this simplified model may not be obvious. If the assumptions captured all the structural properties of a cache (which is most probably not possible), our model could mimic exactly the same behavior of a cache and we would possibly need fewer number of measurements to conduct a successful attack. However, in our experiments without a profiling phase, we used 2^{30} measurements, which is the same number that we used in the attack with a profiling phase. This high number of measurements is required in order to compensate for the unknown architectural and behavioral complexities of cache memory hierarchy, which cannot be fully captured by our simplified set of assumptions.

For example, in Assumption 2, what would happen if we assumed the cache were 4-way set associative? There would not be any cache evictions in a related AES table until the relevant cache line in the set were accessed by the process. However, the experiments show that cache evictions eventually occur. Thus, we would just need to wait until a cache eviction occurs, which means taking more measurements. Consequently, we compensate for the deficiencies due to the simplifications of the cache behavior by increasing the measurement count and using statistical techniques.

A further improvement introduced by this new method is that it does not need the modification of the address space layout randomization (ASLR) flag as mentioned in [15]. The ASLR technique randomizes the address space of the executables, stack, heap, and the libraries. Since the original Bernstein's attack needs 2 separate runs (i.e. profiling and attack phase), the address spaces may be different at each run due to ASLR. Thus, the timing models of these 2 phases may not correlate and the attack may fail. However, in our methodology there is no need for a separate profiling phase that may cause a mismatch between profiling and attack phase timing models. Consequently, this result demonstrates that the use of ASLR is not an effective countermeasure against the cache attack.

The applicability of the proposed attack on CPUs that are from different vendors is an important topic that needs further investigation, but during our research we did not have the opportunity to conduct the attack on CPUs from different vendors. Nevertheless, in modern CPUs cache architectures share common characteristics, such as multilevel cache hierarchies and moving of data in blocks in case of cache conflicts. Furthermore, in [17,18], the authors implement Bernstein's attack on ARM CPU and, in [19–22], the authors successfully apply known cache attacks on embedded ARM CPU platforms. We think that these publications and the similarities of the modern cache architectures provide strong evidence that our attack would also work on different CPU platforms.

It may be argued that the table-based AES implementation is outdated and many improvements, such as AES-New Instructions (NI) support and side-channel resistant implementations have been added to the OpenSSL library since then. Nevertheless, we think, as many others in the scientific community, that vulnerabilities enabling cache attacks are important artifacts that require further study. Therefore, in this study we investigated the feasibility of an improved attack based on a generalized cache timing-behavior modeling approach, and in order to verify our claims we used this specific implementation as a test bed since it is easier to observe leakages, which simplifies the verification process. In addition, there have been many recent works based on this table-based AES implementation. In [17,18], Spreitzer et al. perform Bernstein's cache-timing attack on modern ARM CPU architectures, demonstrating that it is possible to perform this attack on ARM architectures. In [19–22], several authors conduct other known cache attacks such as *prime+probe*, *flush+reload*, *evict+reload*, *flush+flush*, *cache access pattern*, and *cache collision* attacks, which show these attacks are also applicable on embedded platforms. In [23,24], Gülmezoğlu et al. conduct successful *flush+reload* and *prime+probe* cache attacks on cross-virtual machine (VM) environments. In [25], Irazoqui et al. show it is possible to successfully conduct Bernstein's cache-timing attack on Xen and VMware by using popular crypto libraries. Again Irazoqui et al. show in [26,27] that it is possible to perform *flush+reload* and *prime+probe* cache attacks on virtualized environments. In [28], Weiβ et al. perform Bernstein's cache-timing attack on a virtualization environment, which runs on an ARM CPU platform. In a recent research that claims a novel attack [29], the stalling delay caused by cache bank conflicts is exploited to infer the secret key. In [30], Moghimi et al. conduct a cache attack on a Software Guard eXtensions (SGX)[2] supported Intel platform with different AES implementations.

[2] See https://software.intel.com/en-us/sgx which was accessed on March 17, 2018.

In [31], the authors provide improvements over existing cache attacks and provide experimental results. In [32], the authors show a cross processor cache attack that targets high efficiency CPU interconnects. Moreover, in [33,34], the authors share their findings on cache attacks experimented on general purpose CPUs. All of the works mentioned above share a common point, which is that they all use the same or similar table-based AES implementations.

7. Conclusion

In this work, we present a methodology to extract a simplified model of the cache timing-behavior of a computer. Furthermore, we also present a variant of Bernstein's cache-timing attack without a profiling phase on the last round of AES and demonstrate that it can be successfully applied in many experimental settings. The implementation results demonstrate that the method can be used to extract realistic timing models and the nonprofiled attack has a comparable performance to the original attack with a profiling phase. The proposed methodology can also be applied on other CPU architectures and on other cryptographic algorithms, as long as cache conflicts (e.g., cache misses) occur on sensitive data (e.g., AES lookup tables), which leaks information about the secret key. In summary, the new method allows to apply Bernstein's attack in a more realistic and practical context by eliminating the profiling phase and its associated difficulties.

References

[1] Kocher PC. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In: The 16th Annual International Cryptology Conference on Advances in Cryptology; 18–22 August 1996; Santa Barbara, CA, USA. Berlin, Germany: Springer. pp. 104-113.

[2] Kocher PC, Jaffe J, Jun B. Differential power analysis. In: The 19th Annual International Cryptology Conference on Advances in Cryptology; 15–19 August 1999; Santa Barbara, CA, USA. Berlin, Germany: Springer. pp. 388-397.

[3] Gandolfi K, Mourtel C, Olivier F. Electromagnetic analysis: concrete results. In: Cryptographic Hardware and Embedded Systems - CHES; 14–16 May 2001; Paris, France. Berlin, Germany: Springer. pp. 251-261.

[4] Aciicmez O, Koc CK, Seifert J. Predicting secret keys via branch prediction. In: Topics in Cryptology - CT-RSA; 5–9 February 2007; San Francisco, CA, USA. Berlin, Germany: Springer. pp. 225-242.

[5] Mowery K, Keelveedhi S, Shacham H. Are AES x86 cache timing attacks still feasible? In: Cloud Computing Security Workshop; 16–18 October 2012; Raleigh, NC, USA. New York, NY, USA: ACM. pp. 19-24.

[6] Reberio C, Mukhopadhyay D. Micro-architectural analysis of time-driven cache attacks: quest for the ideal implementation. IEEE T Comput 2015; 64: 778-790.

[7] D. J. Bernstein. Cache timing attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[8] AES Standard. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[9] Kelsey J, Schneier B, Wagner D, Hall C. Side channel cryptanalysis of product ciphers. J Comput Secur 2000; 8: 141-158.

[10] Tromer E, Osvik DA, Shamir A. Efficient cache attacks on AES, and countermeasures. J Cryptol 2009; 23: 37-71.

[11] Aciicmez O, Koc CK. Trace-driven cache attacks on AES (short paper). In: 8th International Conference on Information and Communications Security; 4–7 December 2006; Raleigh, NC, USA. Berlin, Germany: Springer. pp. 112-121.

[12] Tsunoo Y, Saito T, Suzaki T, Shigeria M, Miyauchi H. Cryptanalysis of DES implemented on computers with cache. In: Cryptographic Hardware and Embedded Systems - CHES; 8–10 September 2003; Cologne, Germany. Berlin, Germany: Springer. pp. 62-76.

[13] Chiappetta M, Savaş E, Yılmaz C. Real time detection of cache-based side-channel attacks using hardware performance counters. Appl Soft Comput 2016; 49: 1162-1174.

[14] Neve M. Cache-based vulnerabilities and SPAM analysis. PhD, Université Catholique de Louvain, Louvain, Belgium, 2006.

[15] Atici AC, Yilmaz C, Savas E. An approach for isolating the sources of information leakage exploited in cache-based side-channel attacks. In: 7th International Conference on Software Security and Reliability - Companion (SERE-C); 18–20 June 2013; Washington, DC, USA. Washington, DC, USA: IEEE. pp. 74-83.

[16] Bonneau J, Mironov I. Cache-collision timing attacks against AES. In: Cryptographic Hardware and Embedded Systems - CHES; 10–13 October 2006; Yokohama, Japan. Berlin, Germany: Springer. pp. 201-215.

[17] Spreitzer R, Gerard B. Towards more practical time-driven cache attacks. In: Information Security Theory and Practice. Securing the Internet of Things - WISTP; 30 June–2 July 2014; Heraklion, Crete, Greece. Berlin, Germany: Springer. pp. 24-39.

[18] Spreitzer R, Plos T. On the applicability of time-driven cache attacks on mobile devices. In: 7th International Conference, Network and System Security; 3–4 June 2013; Madrid, Spain. Berlin, Germany: Springer. pp. 656-662.

[19] Lipp M, Gruss D, Spreitzer R, Maurice C, Mangard S. ARMageddon: cache attacks on mobile devices. In: 25th USENIX Security Symposium; 10–12 August 2016; Austin, TX, USA. Berkeley, CA, USA: USENIX Association. pp. 549-564.

[20] Spreitzer R, Plos T. Cache-access pattern attack on disaligned AES T-tables. In: 4th International Workshop on Constructive Side-Channel Analysis and Secure Design, COSADE; 6–8 March 2013; Paris, France. Berlin, Germany: Springer. pp. 200-214.

[21] Bogdanov A, Eisenbarth T, Paar C, Wienecke M. Differential cache-collision timing attacks on AES with applications to embedded CPUs. In: Topics in Cryptology - CT-RSA; 1–5 March 2010; San Francisco, CA, USA. Berlin, Germany: Springer. pp. 235-251.

[22] Gallais JF, Kizhvatov I, Tunstall M. Improved trace-driven cache-collision attacks against embedded AES implementations. In: 11th International Workshop on Information Security Applications, WISA; 24–26 August 2010; Jeju Island, Korea. Berlin, Germany: Springer. pp. 243-257.

[23] Gülmezoğlu B, İnci MS, Irazoqui G, Eisenbarth T, Sunar B. Cross-VM cache attacks on AES. IEEE T Comput Syst 2016; 2: 211-222.

[24] Gülmezoğlu B, İnci MS, Irazoqui G, Eisenbarth T, Sunar B. A faster and more realistic flush+reload attack on AES. In: 6th International Workshop on Constructive Side-Channel Analysis and Secure Design - COSADE; 13–14 April 2015; Berlin, Germany. New York, NY, USA: Springer-Verlag New York. pp. 111-126.

[25] Irazoqui G, İnci MS, Eisenbarth T, Sunar B. Fine grain cross-VM attacks on Xen and VMware. In: IEEE Fourth International Conference on Big Data and Cloud Computing; 3–5 December 2014; Sydney, Australia. Washington, DC, USA: IEEE. pp. 737-744.

[26] Irazoqui G, Eisenbarth T, Sunar B. 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; 18–20 May 2015; San Jose, CA, USA. Washington, DC, USA: IEEE. pp. 591-604.

[27] Irazoqui G, İnci MS, Eisenbarth T, Sunar B. Wait a Minute! A fast, cross-VM attack on AES. In: 17th International Symposium on Research in Attacks, Intrusions and Defenses - RAID; 17–19 September 2014; Gothenburg, Sweden. Switzerland: Springer. pp. 299-319.

[28] Weiß M, Heinz B, Stumpf F. A cache timing attack on AES in virtualization environments. In: Financial Cryptography and Data Security Conference; 27 February–March 2 2012; Kralendijk, Bonaire. Berlin, Germany: Springer. pp. 314-328.

[29] Jiang ZH, Fei Y. A novel cache bank timing attack. In: IEEE/ACM International Conference on Computer-Aided Design - ICCAD; 13–16 November 2017; Irvine, CA, USA. Piscataway, NJ, USA: IEEE. pp. 139-146.

[30] Moghimi A, Irazoqui G, Eisenbarth T. CacheZoom: How SGX amplifies the power of cache attacks. In: Cryptographic Hardware and Embedded Systems - CHES; 25–28 September 2017; Taipei, Taiwan. Berlin, Germany: Springer. pp. 69-90.

[31] Ashokkumar C, Giri RP, Menezes B. Highly efficient algorithms for AES key retrieval in cache access attacks. In: IEEE European Symposium on Security and Privacy; 21–24 March 2016; Saarbrücken, Germany. Washington, DC, USA: IEEE. pp. 261-275.

[32] Irazoqui G, Eisenbarth T, Sunar B. Cross processor cache attacks. In: 11th Asia Conference on Computer and Communications Security; 30 May–03 June 2016; Xi'an, China. New York, NY, USA: ACM. pp. 353-364.

[33] Gullasch D, Bangerter E, Krenn S. Cache games - bringing access-based cache attacks on AES to practice. In: 32nd IEEE Symposium on Security and Privacy; 22–25 May 2011; Oakland, CA, USA. Washington, DC, USA: IEEE. pp. 490-505.

[34] Reberio C, Mondal M, Mukhopadhyay D. Pinpointing cache timing attacks on AES. In: 23rd International Conference on VLSI Design; 3–7 January 2010; Bangalore, India. Washington, DC, USA: IEEE. pp. 306-311.