Remote Cache-Timing Attack without Learning Phase

Ali Can Atici, Cemal Yilmaz, Erkay Savas
Faculty of Engineering and Natural Sciences
Sabanci University
Istanbul 34956, Turkey
{alicana, cyilmaz, erkays}@sabanciuniv.edu

Abstract—Theoretically secure cryptographic algorithms can be vulnerable to attacks due to their implementation flaws, which disclose side-channel information about the secret key. Bernstein's attack is a well known cache-timing attack which uses execution time as the side-channel. The major drawback of this attack is that it needs an identical target machine to perform its learning 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 learning phase. We propose a methodology to model the cache timing-behavior of the target machine by hypothetical modeling. To test the validity of the proposed method, we performed the Bernstein attack and showed that, in majority of the cases, the new attack is actually superior to the original attack which uses a learning phase.

Index Terms—Cache-timing attacks, side-channel, cache model, timing model.

1 Introduction

Cryptographic algorithms that are secure against known theoretical attacks can still be vulnerable to side-channel attacks because of the flaws in their implementations [1]. Execution time, power consumption, electromagnetic emission, execution footprints in the micro-architectural structure of underlying microprocessor, etc., can be used as side-channel information. Since in cryptographic implementations, the secret key directly affects the emitted side-channel information, observations made on this leaked data can eventually lead to the revelation of the secret key.

Side-channel attacks, which exploit the fact that micro-architectural resources, such as cache memory and branch prediction unit, are shared, are widely studied in the literature [2], [3], [4], [5]. Cache-based side channel attacks, which are also the main focus of this paper, exploit the cache access patterns of cryptographic applications. These attacks typically operate by inferring if a cache access is a hit or a miss, or if a certain cache line is accessed or not, mostly by measuring the access time. If the inference is accurate, the access patterns can be associated with im/probable key values to extract the secret key or to reduce the size of the key space.

Bernstein's cache-timing attack [6] is a well-known cache-based side-channel attack, which is applied remotely in a client-server setting. The attack exploits the differences in encryption times of randomly generated messages to recover the secret key used by an OpenSSL implementation of AES (Advanced Encryption Standard [7]). It is demonstrated that exploitable timing differences in Bernstein's attack occur due to L1 data cache accesses by other processes [8] or by the process itself [9], which result in conflicts with the AES cache accesses, causing some table entries used by AES to be evicted from the cache.

A major drawback of Bernstein's attack is the necessity of having a computer system which is identical to the target system as the learning phase of the attack needs to construct a model of the cache timing-behavior of the latter. Exact replication of the target system and all its machine specific cache effects can be very difficult, which causes the attack to be considered unrealistic in many contexts [10], [11].

In this work we show that Bernstein's attack does not really need a specific learning phase. We propose a methodology, based on hypothetical modeling of the cache timing-behavior of a system and demonstrate that the Bernstein's attack successfully recovers the key using one of the models that best represents the cache timing-behavior of the system. Our proposed methodology eliminates the need for an identical target machine, which makes the attack more realistic. Furthermore, since the learning phase is eliminated, countermeasures such as ASLR [9], which changes the cache timing-behavior of a system between learning and attack phases, can not be effective anymore.

The rest of the paper is organized as follows: Section 2 presents related works; Section 3 provides brief background information on CPU caches, AES, Bernstein's attack, and the last round Bernstein attack; Section 4 discusses the details of cache-timing attacks with learning phase; Section 5 outlines our proposed approach; Section 6 explains how we conduct the Bernstein's attack without the learning phase by using hypothetical modeling; and Section 7 concludes the paper.

2 Related Work

Cache-based side-channel attacks were first mentioned in [12] and later in [13]. We can divide cache-based side-channel attacks into three 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 an access-driven attack, the adversary is generally assumed to be able to run a *so-called* spy process to create intentional cache contentions with the cryptographic process to monitor the cache access patterns of the latter. Osvik *et al.* [11], [14] propose an approach, in which a spy process is employed to identify the cache lines/sets that are accessed during a cryptographic operation, where plaintext and/or the ciphertext are known. In their attack Osvik *et al.* state two cases: synchronous and asynchronous. In the synchronous case, the attacker is capable of starting an encryption operation at will. They show that a 128-bit AES key can be recovered after 300 encryptions on Athlon 64 platform with the synchronous attack. The asynchronous case is more restrictive, where the spy process has no interaction with the target process. It is only allowed to use its own memory access measurements to infer the secret key. In this case, they are able to recover 45.7 bits of the AES key after one minute of observation time of encryption operations with the same secret key.

Trace-driven cache attacks (i.e., the second category of attacks) were, on the other hand, first introduced in [2]. In these attacks, it is assumed that the adversary has full control over the target device and that she can determine whether a particular cache access is a miss or hit during the cryptographic operation by observing power or electromagnetic emissions of the cryptographic device. Thus, theoretically the attacker will obtain a trace of cache access outcomes during the cryptographic operation. In a simulation of the attack reported in [2], it was shown that a 56-bit DES key actually provides only 32-bit key security if the attack is successfully applied. Trace-driven attacks are also investigated in detail in [15].

Finally, 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 which are dependent on the secret keys and the plaintext. This dependency can eventually be used to infer the secret key. In [16] Tsunoo *et al.* use the time variations that occur during encryptions due to the cache misses as a result of table lookups for s-box operations. The proposed attack is applied to the DES algorithm, which is shown to be broken using 2^{23} known plaintexts and at a success rate greater than 90% after 2^{24} operations.

Most of the cache attacks are originally designed to be applied locally but they can be converted into remote attacks, where cryptographic operations are executed in a remote server. Works such as [17], [11], [14] demonstrate how a particular cache attack can be applied remotely.

The majority of the cache attacks rely on the so-called *cache cleaning* operation via a spy process, which evicts all data of cryptographic process from the cache before the start of an encryption operation. Bernstein's attack [6], which can be categorized as a remote timing-based attack is the only exception. In this attack, there is an AES server and an AES client. The AES client sends random plaintexts to the AES server. The AES server encrypts the plaintexts and sends the resulting ciphertexts back to the client. The client uses the execution times of these encryption operations to infer the secret key. The attack consists of two main phases. In the first phase, known as the learning phase, which is run on an identical platform to the target with a known key, a statistical model is extracted depending on the timing variations of the encryptions.

The second (i.e., attack) phase extracts a similar model on the target machine, where the secret key is unknown, and correlates the obtained two models to make inferences about the secret key (i.e., reduces the key space expectedly for a feasible exhaustive search). 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. In [8], Neve gives an explanation as to why Bernstein's attack works. The most interesting point in the attack is that it does not need a spy process. An intrinsic flaw in the implementation of AES server naturally causes cache contentions, which in turn make the attack possible. These flaws are further investigated in [9].

3 Preliminaries

In this section we provide information about the basic properties of CPU caches, the details of the AES algorithm, how the original Bernstein attack is conducted, and how we modified the Bernstein attack for the last round of the AES algorithm.

3.1 CPU Caches

Cache is a fast memory between RAM and CPU, which exploits the principle of locality [18]. The memory system itself is a hierarchy of memories of different speeds. A data item requested by the CPU is searched first in the topmost (level 1) and also the fastest cache level; and in case it is not found therein, the next level in the hierarchy is tried. If the data is found in a cache level, it is a *cache hit* for this level of the cache hierarchy. Any data item missing in a level leads to a *cache miss* which in turn results in a delay in the access time as the next levels need to be accessed.

Data transfers in memory hierarchy are allowed only between two adjacent levels. Each transfer involving a cache memory moves a *block* of data that will be accommodated in a *cache line*. A cache line size is usually a multiple of the CPU word length and therefore it is dependent on the architecture of the underlying platform. A typical cache line size in modern computers is 64 B.

3.2 Advanced Encryption Standard (AES)

AES [7] is a symmetric-key block cipher algorithm with a block size of 128 bits. AES can work with 128, 192 or 256-bit keys, each of which provides a different level of security.

AES computations are performed in rounds, in which computations are performed on a 4×4 matrix of bytes, which is referred as the AES state. AES has 10, 12, and

14 rounds for the key sizes of 128-bit, 192-bit, and 256-bit, respectively. Every round uses a different key which is generated from the master key according to the AES key generation procedure. AES starts an encryption with an initial `AddRoundKey` operation and each subsequent round consists of the `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey` operations in the given order. However, the last round of AES does not involve the `MixColumns` operation. AES performs a decryption by applying the inverse of the round operations in the reverse order (note that all round operations in the AES algorithm are reversible).

There exists a fast AES implementation which does not perform the round operations separately. Instead, it combines three steps in a round, namely the `SubBytes`, `ShiftRows`, and `MixColumns`, into a lookup operation to AES acceleration (or lookup) tables. In such an implementation, four lookup tables are precomputed, saved and used in all AES rounds except for the last round. Since the last round does not perform the `MixColums` operation, another table is needed. One separate table suffices to implement the last round. An AES round can be computed using 16 table lookup and 12 32-bit XOR operations, followed by the `AddRoundKey` step, which basically consists of four additional 32-bit `XOR` operations.

In this paper, we use an OpenSSL implementation (v0.9.7a Feb 19, 2003) of AES, which employs five static round tables (4 tables for the first rounds and an additional table for the last round) as explained above.

3.3 Bernstein's Cache-Timing Attack

Bernstein presents a cache based timing attack, which targets the lookup table based OpenSSL implementation of AES [6]. In Bernstein's attack there are two separate parties: an *AES server*, which is the victim of the attack, and an *AES client*, which is the adversary applying the attack. The AES server waits for the incoming encryption requests from the network. 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 two main phases: *learning phase* and *attack phase*. In the learning 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 measures the execution time of each encryption and saves it along with the plaintext. In the first round of AES, the indexes to the lookup tables are computed as $s_i^0 = p_i \oplus k_i^0$, where p_i and k_i^0 are ith bytes of the plaintext and the first round key, respectively, and $i = 0, \ldots, 15$. In the learning phase, since we know both the plaintext and the key bytes, we can obtain a timing profile of the indexes, which captures the access times of table lookups.

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 ($s_i' = p_i' \oplus k$) are obtained for each key candidate where $k = 0, 1, \ldots, 255$. Then each of the timing profiles in the attack phase is correlated with the timing profile obtained in the learning phase. The key value giving the highest correlation is the most likely key candidate. For more information about the attack, the interested reader can profitably refer to [6], [8].

In Bernstein's attack, the learning 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 [9]. Also, 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.

3.4 Applying Bernstein's Attack to the Last Round of AES

In Bernstein's original attack only the upper nibbles of key bytes are used in indexes to access lookup tables, thus it can recover at most half of a 128-bit AES key (when 64 B cache blocks are used) [8]. Other half of the key can be obtained, if the attack is extended to the second round of AES [8]. However, the version of the AES implementation used in both works [6] and [8] (OpenSSL v0.9.7a) allows an easier attack on the last round of AES that has the potential of recovering the entire key. In this work, we use this attack methodology that allows us to recover 16 bytes of the key [9].

In the last round of AES [7], 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 learning phase, the outputs of $T4$ lookups used in the last round can be computed using the formula

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

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 learning phase, we can obtain a timing profile based on the output values of $T4$ lookup operations. As a result, we obtain a total of 16 timing profiles for $T4$ lookup operations in the last round, in each of which 256 average execution times of AES are stored. Namely, timing profiles can be represented as an array of $\mathcal{T}_i^l[256]$ 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 $\texttt{InvShiftRows}(c_i' \oplus k)$ for $k = 0, 1, \ldots, 255$, namely $\mathcal{T}_{i,k}^a[256]$. Then, the timing profiles in the attack phase $\mathcal{T}_{i,k}^a$ are correlated to the timing profile of the learning phase $\mathcal{T}_i^l$. The key value k yielding the highest correlation is chosen as the most likely candidate for the key byte $\tilde{k}_i^{10}$. The operation is repeated 16 times for each byte of the round key used in the last round, i.e., $\tilde{k}_0^{10}$, ..., $\tilde{k}_{15}^{10}$. Since timing profiles are extracted according to the $T4$ outputs and every bit of the 10^{th} round key is used to infer the $T4$ outputs, the last round attack can reveal the entire key as opposed to the half of the key in the first round attack.

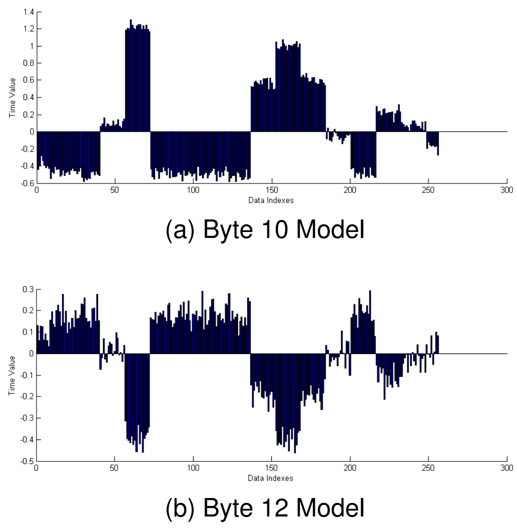
4 A Closer Look At The Attack With Learning Phase

In Section 3, we outlined the cache-timing attack of Bernstein briefly and then explained a modified version of the attack which focuses on the final round, instead of the first round of AES. Both attacks need a learning phase. It is now extremely crucial to understand what we achieve after the learning phase is successfully applied. In [9] and [8] the sources of the unintentional collisions in cache lines holding the AES lookup tables are investigated. These unintentional collisions cause variations in access times due to cache misses. The learning phase helps to obtain data cache timing-behavior of AES process by registering the variations in cache line access times.

Cache timing-behavior of AES process can be expressed as a timing model for each of 16 $T4$ accesses in the last round. Since we know the secret key in the learning phase, the timing model for the ith access in the last round is simply a histogram of average execution times of AES indexed by $T4$ output bytes as computed in Eq. 1.

Figure 1 illustrates two of the 16 actual timing models for 10th and 12th accesses to $T4$ when the learning phase is applied 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, hence the x-axis shows the byte indexes (s_i^9) used in accesses to table $T4$.

In Figure 1, 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. In Figure 1, we also see a symmetry between the two models. They actually suggest that the same group of consecutive indexes (i.e. cache lines) behave the same way while accessing table $T4$ (i.e., all hits or all misses). Since these models belong to two different $T4$ lookup accesses, it is quite normal to see such a symmetry. To summarize, at the end of the learning phase, we obtain a timing model $\mathcal{T}_i^l$ for the i^{th} access in the last round, which is just an array of 256 average execution times of AES.



(a) Byte 10 Model

(b) Byte 12 Model

Fig. 1. Learning Phase Models

In the attack phase, the timing measurements are obtained, grouped and averaged depending on the values of the ciphertext byte involved in i^{th} output of the $T4$ lookup operation, as the corresponding key byte value is unknown. The result is cache timing-behavior model $\tilde{\mathcal{T}}_i^a$, which is again an array of 256 average execution times. Then, the two timing models, namely $\mathcal{T}_i^l$ and $\tilde{\mathcal{T}}_i^a$ are correlated. As $\mathcal{T}_i^l$ is indexed by $T4$ output values and $\tilde{\mathcal{T}}_i^a$ by ciphertext byte values, we transform the latter into 256 timing models, $\mathcal{T}_{i,k}^a$ 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 $\mathcal{T}_i^l$ and $\tilde{\mathcal{T}}_i^a$, 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 key space if the attack is successful. The essential steps of the last round attack with learning phase are given in Algorithm 1, where $\mathcal{T}^l = \cup_{i=0}^{15} \mathcal{T}_i^l$ and $\mathcal{T}^a = \cup_{i=0}^{15} \tilde{\mathcal{T}}_i^a$ are the sets of timing models in the learning and attack phases, respectively.

Algorithm 1 Attack with learning phase

```
Require: $\mathcal{T}^l$ and $\mathcal{T}^a$
Ensure: $\mathcal{K}_R$: Ordered reduced key space
 1: $\mathcal{K} \leftarrow \emptyset$
 2: $\mathcal{K}_R \leftarrow \emptyset$
 3: for $i = 0$ to 15 do
 4:     for $k = 0$ to 255 do
 5:         $\mathcal{T}_{i,k}^a \leftarrow$ Transform($\tilde{\mathcal{T}}_i^a, k$)
 6:         $\gamma \leftarrow$ Correlate($\mathcal{T}_{i,k}^a, \mathcal{T}_i^l$)
 7:         $\nu \leftarrow$ Variance($\mathcal{T}_{i,k}^a, \mathcal{T}_i^l$)
 8:         $\mathcal{K}[i] \leftarrow \mathcal{K}[i] \cup (k, \gamma, \nu)$
 9:     end for
10:     $\mathcal{K}[i] \leftarrow$ Sort($\mathcal{K}[i]$)            ▷ Descending on $\gamma$
11:     $\delta \leftarrow$ Threshold($\mathcal{K}[i]$)
12:     $\mathcal{K}_R[i] \leftarrow$ Reduce($\mathcal{K}[i], \delta$)
13: end for
```

The last round attack can reveal the entire key, although it still needs a learning phase. It is not an easy task for an attacker to setup an identical platform and to run the learning phase. It is also pointed out in [10] and [11] that, neither to access an identical machine nor to recreate the machine-specific cache effects may be feasible. To increase the feasibility and applicability of the attack, we present a novel methodology which needs neither an identical target system nor a learning 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. The details are provided in the following sections.

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 the CPU. Although the timing model will be obtained by certain assumptions, it can still be used effectively even for the cases where these assumptions are overly simplistic for a real world computing platform as shown in our experimental results.

Highly complex and optimized cache implementations and lack of details thereof, render an accurate modeling of cache timing-behavior an involved task. Nevertheless, so far as the cache attacks are concerned, we need a simplified model requiring only the basic understanding of cache organizations. Here, we provide a more formal explanation of our simplified model for a data cache timing-behavior based on the following definitions and assumptions:

Definition 1. ($D1$) *Data* in the data cache of a CPU is comprised of individual bytes. *Data* can be a complex structure or a simple array. Either way, elements of *data* are individually accessible by data indexes. An AES lookup table is an example for *data*, where an index is an 8-bit number.

Assumption 1. ($A1$) *Data* in the cache is aligned and occupy a number of consecutive cache lines (unfragmented). The bytes of *data* are never colocated with other data in the same cache line and its first byte is always placed in a new cache line.

Assumption 2. ($A2$) The direct-mapping is used as a cache placement strategy. While the exact location of data is unknown and not needed, relative locations of its elements and the number of cache lines they occupy can be easily obtained under $A1$.

Assumption 3. ($A3$) 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. ($A4$) Accessing *data* in the cache (i.e., a cache hit) and *data* not in the cache (i.e., a cache miss), take t and $(t + \Delta)$, respectively, and we always have $\Delta > 0$.

Assumption 5. ($A5$) Cache collisions may occur between two different programs, or within the same program; i.e., data sharing the same cache lines evicting each other. During the run of a program, collisions occur always on the same cache lines.

Assumption 6. ($A6$) A cache collision in a cache line evicts the entire block from the cache and brings a new block from the memory.

Assumption 7. ($A7$) A program can observe only hits or only misses in a single run, and the number of hits and misses are equal.

Assumption 8. ($A8$) A program's execution time varies with its input depending on the cache hits and misses occurred during its execution. Execution time of a program can be, t_h or t_m, when it observes hits or misses, respectively. Execution times t_h and t_m have equal probability to occur.

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

Proposition 1. ($P1$) Following $A1$ and $A3$, 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 numbers of bytes in *data* and in a cache line, respectively.

Proposition 2. ($P2$) Following $A4$, $A7$ and $A8$, we can approximate t_h, t_m and t_a of a program with

$$t_h = n_h \cdot t + t_f \tag{3}$$

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

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

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 and n_h and n_m are the number of cache hits and misses, respectively. As $\Delta, n_h, n_m > 0$, we have $t_h < t_a < t_m$. This result implies that program inputs causing accesses to cache lines subject to collisions (see $A5$) result in an execution time, which will tend to be higher than the average execution times of all inputs, t_a, and vice versa.

Proposition 3. ($P3$) 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 $A2, A5$, and $A6$ into account, we can calculate the range of *data* indexes which maps to the colliding cache lines. Let κ denotes the number of bytes accessed by each *data* index. Then, all *data* indexes within the following range are mapped to the cache lines which are in collision:

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

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

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 a 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 finally $\mathcal{S}_c$ denotes the subset of cache lines subject to collisions (i.e., contention set). The algorithm returns a timing model $\mathcal{T}$ 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 which results in cache misses and Line 3 of Algorithm 2 checks whether a data index is in set $\mathcal{I}_c$. In case the referred index causes a miss, the access to the corresponding data part will take longer. In this model we assume $t_a = 0$ to model the timing behavior according to the timing differences from the average.

Example 1. Suppose that *data* is an array of 80 bytes and each data index points to 2 bytes in the memory. If the cache block size is $b = 8$, then *data* will use $m = 40$ indexes and fits in 10 cache lines. Further assume that cache line indexes $[2, 3]$ and $[6, 8]$ are in the contention set. Then using $P3$, we can find which *data* indexes will cause a cache miss. If we use Algorithm 2, our model will have a total of 40 indexes and the indexes in ranges

Algorithm 2 Modeling the Cache Timing-Behavior

```
Require: data, $m$, $\mathcal{S}_c$, $b$, $\kappa$
Ensure: $\mathcal{T}$: Cache timing-behavior model
1: $\mathcal{I}_c$ = MissDataIndex($\mathcal{S}_c, b, \kappa$)          ▷ ($P3$)
2: for $s = 0$ to $m - 1$ do                 ▷ for each data index
3:     if $s \in \mathcal{I}_c$ then
4:         $\mathcal{T}[s] = 1$
5:     else
6:         $\mathcal{T}[s] = -1$
7:     end if
8: end for
```

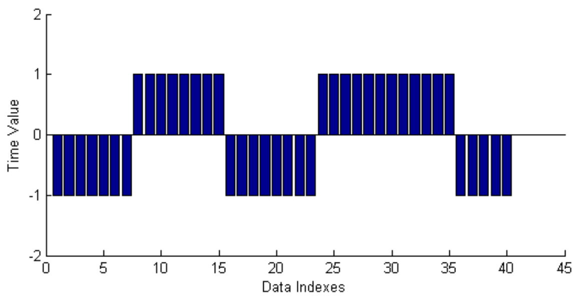


Fig. 2. Example Cache Timing-Behavior Model

$[8, 15]$ and $[24, 35]$ will have the value of 1, while the rest of the indexes will have the value of -1 as illustrated in Figure 2.

When Figures 1 and 2 are compared, one can easily see the similarity between the patterns in the execution times in the actual and the simplified timing models. In the timing models obtained in the learning phase, table $T4$ is *data* and lookup bytes are the indexes as defined by the terminology introduced in Section 5. The measured timing values in the learning phase are noisy and obtained by averaging excessively many AES execution times. All the same, the simplified timing model captures essentially the same behavior.

6 The Proposed Attack Methodology and Implementation Results

In this section, we give a formal description of the proposed attack using hypothetical modeling without the learning phase and present our experiment results.

6.1 Cache-Timing Attacks Without Learning Phase

In order for an attacker to model the cache timing-behavior of the server in the learning phase, the attacker must produce an identical system the cache timing-behavior of which must exactly be the same as the target computer. This is a major drawback in the Bernstein's original cache-timing attack, which is also mentioned in [10] and [11], as this can be infeasible in many circumstances. Using hypothetical modeling as suggested here, however, eliminates the need for an identical system, hence the learning phase. The attack without the learning phase needs only the knowledge of the cache line size of the target computer and the size of the lookup tables. A typical cache line size is 64 B in 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* (see $D1$) is 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. But, 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 $[0, 2^{16} - 1]$ 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 that 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 leftmost bit, which corresponds to the first 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: $\mathcal{S}_C$: Cache contention set for index $l$
1: $\mathcal{S}_C \leftarrow \emptyset$
2: for $i = 0$ to $n - 1$ do
3:     if ($l \bmod 2 = 0$) then
4:         $\mathcal{S}_C \leftarrow \mathcal{S}_C \cup i$
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., $\mathcal{T}_a$).

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 lines 4-6, and applies the AES s-box operation to the model in line 7. Then, the attack phase in Algorithm 1 is applied to find the size of the reduced key space in line 8. The sizes of the reduced key space for simplified models are saved as shown in line 9. Finally, they are sorted from smallest to largest (Step 11) and the simplified model with the smallest reduced key space size is chosen as the most probable cache timing-behavior model (Step 12). Since we perform the last round attack using the outputs of table $T4$, the AES s-box operation is applied to the model in line 7. Once we obtain the model, we can run Algorithm 1 and find the key bytes.

Algorithm 4 Searching For Cache Timing-Behavior Model

```
Require:
    $T4$ : Lookup table
    $m$ : Index count of $T4$
    $\kappa$ : Size of each $T4$ entry in number of bytes
    $b$ : Size of each cache line in number of bytes
    $\mathcal{T}_a$ : Timing model in attack phase
    $n$ : Number of cache lines occupied by $T4$
    $\delta$ : Correlation threshold
Ensure: $\tilde{\mathcal{T}}_h$: Correct cache timing-behavior model
 1: $\mathcal{M} \leftarrow \emptyset$
 2: for $l = 0$ to $2^n - 1$ do
 3:     $\mathcal{S}_C \leftarrow$ Algorithm 3$(l, n)$
 4:     for $j = 0$ to 15 do
 5:         $\mathcal{T}_h[j] \leftarrow$ Algorithm 2$(T4, m, \mathcal{S}_C, b, \kappa)$
 6:     end for
 7:     $\mathcal{T}_h \leftarrow$ AES-sbox$(\mathcal{T}_h)$
 8:     $\mathcal{K}_R \leftarrow$ Algorithm 1$(\mathcal{T}_h, \mathcal{T}_a)$
 9:     $\mathcal{M} \leftarrow \mathcal{M} \cup (\mathcal{T}_h, |\mathcal{K}_R|)$
10: end for
11: $\mathcal{M} \leftarrow$ Sort$(\mathcal{M})$                ▷ Ascending on $|\mathcal{K}_R|$
12: $\tilde{\mathcal{T}}_h \leftarrow \mathcal{M}[0][0]$
```

In order to test Algorithm 4, we ran it for the example in Figure 1 and obtained the index 14433 as the most probable cache timing model. When we plot this model, we obtain Figure 3. A closer look at Figure 3 reveals that our simplified model resembles to the real model depicted in Figure 1b.

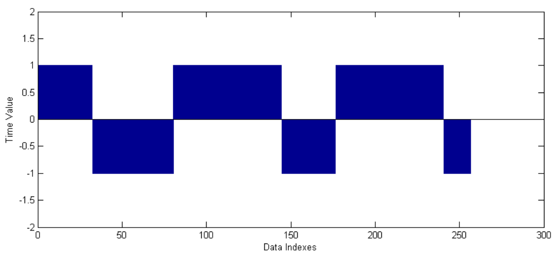


Fig. 3. Calculated Learning Phase Model For Intel Pentium P6200

An important point to note here is that, in Algorithm 4, we assume that all the lookup index bytes will have the same timing model. Actually this is not always the case. In Section 4 we mention that a cache timing model actually gives us the cache lines which are in contention, thus a model can take two forms as seen in Figure 1. Thus, for the indexes used in lookup operations where the real model is the symmetric of our simplified model, the correct key value will tend to appear in the bottom of the sorted list of reduced key spaces. The problem can be solved with a small modification in Algorithm 1. In Algorithm 1, key guesses are sorted depending on their correlation values, which can be positive or negative. The key value with the maximum correlation becomes the most possible candidate. In case of a symmetry between the real and the simplified model, this correlation grows in the negative direction for the correct key guesses. Thus, if we take the absolute values of the correlations before sorting, correct key byte will appear in the first ranks in the sorted list.

6.2 Implementation and Results of Proposed Attack

We conducted the last round attack with and without learning phase on different hardware and software platforms with varying client-server deployment configurations. In each experiment setup, to carry out the attack, the AES client uses 2^{30} randomly generated messages, each of which is of size 600 B. The AES server, in turn, uses the OpenSSL v0.9.7a (Feb 19, 2003) implementation of AES (http://www.openssl.org/source/), the same implementation used by the original Bernstein's attack, for the encryptions. In the attacks with learning phase, two separate measurements are used (i.e. learning and attack phase measurements) while in the attacks without learning phase only attack phase measurements are used.

For each experiment setup, we carried out a number of attacks and calculated the average of the results. For each key byte, the attacks produced a set of candidate key values, sorted by their likelihood. By multiplying the sizes of the candidate sets we obtain the reduced space for the whole AES key, which necessitates exhaustive search.

Table 1 presents the results we obtained, in which the first two columns summarize the CPU and the operating system configurations used in the attacks, respectively. The third column depicts the deployment configuration of the AES client and server, i.e., whether the client and the server are on the same core or on different cores. The last two columns present the average sizes of the reduced key spaces obtained after the attacks are applied.

In our experiments, we first observed that the correct secret key is always in the reduced key space. We then observed that in all the experiment setups, the sizes of the reduced key spaces are always within feasible limits for an exhaustive search. To put into a perspective, the expected time of a brute-force search for a 56-bit DES is less than a day by using the latest version (RIVYERA[1]) of the specialized cryptanalytic engine COPACOPANA [19]. Although both attack methods are successful, the results clearly show that the proposed attack without the learning phase outperforms the original attack in majority of the cases. Since our simplified models are noise free and specifically selected for the attack phase data, a performance gain in the results can be expected. A further improvement introduced by the new attack method is that it does not need the modification of the ASLR flag as mentioned in [9], as there is no need for a separate learning phase which may cause a mismatch between learning and attack timing models. We can also state that ASLR, which randomizes the location of the lookup tables in the memory (and cache), is not an effective countermeasure against the proposed attack.

Furthermore, we also tried a more realistic setup where we measured the execution timings from the client side. In this setup we used a PC which hosts two separate Intel Xeon E5405 CPUs, where AES server and client run on different CPUs. Since both programs run in the same PC, we minimize the effects of network delay on the measurements. In the classical attack (i.e., using both learning and attack phases) we reduced the key space to 2^{67} from 2^{128} with 2^{34} measurements, while the proposed attack (i.e., without

1. See (http://www.sciengines.com/company/news-a-events/74-des-in-1-day.html), which was accessed on June 21, 2015.

TABLE 1
Results obtained from the last-round attack with and without learning phase

Processor	Operating System	AES Client-Server Deployment Configuration	Reduced Key Space with Learning Phase	Reduced Key Space without Learning Phase
Intel Pentium P6200	Ubuntu 3.0.0-12 kernel	same core	2^{32}	2^{32}
Intel Pentium P6200	Ubuntu 3.0.0-17 kernel	different cores	2^{49}	2^{37}
Intel Core 2 Duo P8400	Ubuntu 3.0.0-12 kernel	same core	1	2^{12}
Intel Core 2 Duo P8400	Ubuntu 3.0.0-17 kernel	different cores	2^{24}	2^{29}
Intel Xeon E5405	CentOS 2.6.18 kernel	same core	2^{34}	2^{19}
Intel Xeon E5405	CentOS 2.6.18 kernel	different cores	2^{51}	2^{16}

learning phase) reduced the key space to 2^{60} with 2^{34} measurements. These results demonstrate that the attack can be applied in multi-processor and multi-core platforms. This finding is especially important in cloud computing environments as different virtual machines can be co-located in different cores of the same computer. Several works in the literature successfully demonstrate that it is possible to co-locate a spy process and cryptographic application in the same computer and apply a cache-attack [20], [21].

7 Conclusion

In this work, we present a variant of Bernstein's remote cache attack without a learning phase against AES and demonstrate that it can be successfully applied in many experimental settings. The attack assumes a tractable, simplified model of cache timing-behavior, in which the cache is partitioned into two sets of cache lines: the cache lines in one set take longer to access due to persistent collisions and those in the other that are faster to access as the collisions in them are absent, few or sporadic. By exhaustively trying all possible simplified models and correlating them to the real timing measurements of AES taken in the attack phase, the proposed method is used to recover secret key of AES. We demonstrate the effectiveness of the method by using Bernstein's attack on the last round of AES. The attack is implemented in several settings featuring different hardware, software and client-server deployments. The results prove that the method can be used to extract realistic timing models. The experimental results show that the proposed attack outperforms the original attack by Bernstein in majority of the cases. In summary, the new method allows to apply Bernstein's attack in more realistic and practical settings by eliminating the learning phase.

References

[1] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *CRYPTO*, ser. Lecture Notes in Computer Science, M. J. Wiener, Ed., vol. 1666. Springer, 1999, pp. 388–397.

[2] D. Page, "Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel," Department of Computer Science,University of Bristol, Tech. Rep. CSTR-02-03, June 2002.

[3] O. Aciiçmez, Çetin Kaya Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *CT-RSA*, ser. Lecture Notes in Computer Science, M. Abe, Ed., vol. 4377. Springer, 2007, pp. 225–242.

[4] K. Mowery, S. Keelveedhi, and H. Shacham, "Are aes x86 cache timing attacks still feasible?" in *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, ser. CCSW '12. New York, NY, USA: ACM, 2012, pp. 19–24. [Online]. Available: http://doi.acm.org/10.1145/2381913.2381917

[5] C. Rebeiro and D. Mukhopadhyay, "Micro-architectural analysis of time-driven cache attacks: Quest for the ideal implementation," *IEEE Trans. Computers*, vol. 64, no. 3, pp. 778–790, 2015. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TC.2013.212

[6] D. J. Bernstein, "Cache Timing Attacks on AES," http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[7] "AES Standard," csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001.

[8] M. Neve, "Cache-based vulnerabilities and spam analysis," Ph.D. dissertation, Universite catholique de Louvain, 2006.

[9] A. C. Atici, C. Yilmaz, and E. Savas, "An approach for isolating the sources of information leakage exploited in cache-based side-channel attacks," in *Seventh International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, Maryland, USA, 18-20 June 2013 - Companion Volume*, 2013, pp. 74–83.

[10] J. Bonneau and I. Mironov, "Cache-Collison Timing Attacks Against AES," in *CHES 2006 LNCS*, L. Goubuin and M. Matsui, Ed., vol. 4249.

[11] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES (extended version)," http://tau.ac.il/ tromer/, 2005.

[12] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '96. London, UK, UK: Springer-Verlag, 1996, pp. 104–113. [Online]. Available: http://dl.acm.org/citation.cfm?id=646761.706156

[13] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," *J. Comput. Secur.*, vol. 8, no. 2,3, pp. 141–158, Aug. 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=1297828.1297833

[14] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, 2009.

[15] O. Aciiçmez and Çetin Kaya Koç, "Trace-driven cache attacks on aes (short paper)," in *ICICS*, ser. Lecture Notes in Computer Science, P. Ning, S. Qing, and N. Li, Eds., vol. 4307. Springer, 2006, pp. 112–121.

[16] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeria, and H. Miyauchi1, "Cryptanalysis of DES Implemented on Computers with Cache," in *CHES 2003 LNCS*, C.D. Walter et al., Ed., vol. 2279.

[17] O. Aciiçmez, W. Schindler, and Çetin Kaya Koç, "Cache based remote timing attack on the aes," in *CT-RSA*, ser. Lecture Notes in Computer Science, M. Abe, Ed., vol. 4377. Springer, 2007, pp. 271–286.

[18] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, The Hardware/Software Interface*. Morgan Kaufmann Publishers, 2009.

[19] T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp, "Cryptanalysis with copacobana," *IEEE Trans. Computers*, vol. 57, no. 11, pp. 1498–1513, 2008.

[20] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, E. Al-Shaer, S. Jha, and A. D. Keromytis, Eds. ACM, 2009, pp. 199–212. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653687

[21] B. Gülmezoglu, M. S. Inci, G. I. Apecechea, T. Eisenbarth, and B. Sunar, "A faster and more realistic flush+reload attack on AES," in *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany,*

April 13-14, 2015. Revised Selected Papers, ser. Lecture Notes in Computer Science, S. Mangard and A. Y. Poschmann, Eds., vol. 9064. Springer, 2015, pp. 111–126. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21476-4_8

[22] M. Abe, Ed., *Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4377. Springer, 2006.