METHODS FOR FINDING THE SOURCES OF LEAKAGE IN CACHE-TIMING ATTACKS AND REMOVING THE PROFILING PHASE

by

ALİ CAN ATICI

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Sabancı University
December 2018

METHODS FOR FINDING THE SOURCES OF LEAKAGE IN CACHE-TIMING ATTACKS AND REMOVING THE PROFILING PHASE

APPROVED BY:

Prof. Erkay Savaş ....
(Dissertation Supervisor)

Prof. Albert Levi ....

Asst. Prof. Murat Kaya ....

Asst. Prof. Ahmet Onur Durahim ....

Asst. Prof. Cengiz Örencik ....

DATE OF APPROVAL: 27-12-2018

ABSTRACT

METHODS FOR FINDING THE SOURCES OF LEAKAGE IN CACHE-TIMING ATTACKS AND REMOVING THE PROFILING PHASE

ALİ CAN ATICI

Ph.D. Dissertation, December 2018

Supervisor: Prof. Erkay Savaş

Keywords: Side-Channel Analysis, Leakage Sources, Hardware Performance Counters, Cache-Timing Attacks, Cache Modeling, Profiling Phase

Cryptographic algorithms are widely used in daily life in order to ensure data confidentiality and privacy. These algorithms are extensively analyzed by scientists against a theoretical deficiency. However, these theoretically verified algorithms could still posses security risks if they are not cautiously implemented. Side-channel analysis can infer the secret key by using the information leakage due to implementation flaws. One of the most studied side-channel attack is the Bernstein's cache-timing attack. This attack owes its reputation to its ability to succeed without a spy process, which is needed to create intentional cache contentions in other cache attacks. However, the exact leakage sources of the Bernstein's attack remained uncertain to a large extent. Moreover, the need for an identical target system to perform its profiling phase makes the attack unrealistic for real world computing platforms. In this dissertation we address these two problems. Firstly, we propose a methodology to reveal the exact sources of the information leakage. The proposed methodology makes use of hardware performance counters to count the number of cache misses, to which the code blocks in the program are subject. Our methodology can help the developers analyze their implementations and fix their code in the early phases of the development. Secondly, we present an approach to extract simplified cache timing-behavior models analytically and propose to use these generated models instead of a profiling phase. The fact that the attack can be accomplished without a profiling phase will lead the attack to be considered a more realistic threat than the attack originally proposed by Bernstein. We believe that, this improved version of the attack will encourage the cryptographic system designers to take further precautions against the attack.

ÖZET

ÖNBELLEK-ZAMANLAMA SALDIRILARINDA SIZINTI KAYNAKLARINI BULMAK VE AYRIMLAMA FAZINI KALDIRMAK İÇİN METOTLAR

ALİ CAN ATICI
Doktora Tezi, Aralık 2018
Danışman: Prof. Dr. Erkay Savaş

Anahtar Sözcükler: Yan-Kanal Analizi, Sızıntı Kaynakları, Donanım Performans Sayaçları, Önbellek-Zamanlama Saldırıları, Önbellek Modelleme, Ayrımlama Fazı

Kriptografik algoritmalar günlük hayatımızda yaygın olarak kullanılmaktadır. Bu algoritmalar, olası açıklara karşı, biliminsanlari tarafından matematiksel olarak analiz edilmektedir. Bu algoritmalar matematiksel olarak doğrulanmış olmalarına rağmen, özenli bir şekilde gerçeklenmezlerse, güvenlik riskleri barındırmaya devam edebilirler. Yan-kanal analizi ile gerçekleme hatalarından kaynaklı bilgi sızıntıları kullanılarak gizli anahtar elde edilebilmektedir. En çok çalışılmış yan-kanal ataklarından birisi Bernstein'ın atağıdır. Bu atak, başarılı sonuçlar elde etmek için kasıtlı olarak önbellek çakışmaları yaratan bir casus sürece ihtiyaç duymaması ile bilinmektedir. Bununla birlikte, atağı başarıya ulaştıran sızıntı kaynakları net bir şekilde ortaya çıkarılamamıştır. Ayrıca ayrımlama fazı için, hedef sistemin bire bir kopyasına ihtiyaç duyması atağın gerçek hayattaki uygulanabilirliği üzerinde soru işaretleri uyandırmıştır. Bu tezde, bu iki sorun üzerinde çalışmalar yapılmıştır. İlk olarak, bilgi sızıntısının kesin kaynağını bulmak için bir metodoloji öneriyoruz. Önerilen metodoloji, program içerisindeki kod bloklarının maruz kaldığı önbellek ıskalarını, donanım performans sayaçları ile saymaya dayanmaktadır. Program geliştiricileri, metodolojimizi kullanarak kodlarını analiz edebilir ve olası hataları erken bir aşamada düzeltebilirler. İkinci olarak, önbellek zaman-davranışı modellerini analitik olarak çıkarmaya yarayan bir yaklaşım sunuyor ve oluşturulan bu modelleri ayrımlama fazı yerine kullanmayı öneriyoruz. Saldırının bir ayrımlama aşaması olmadan gerçekleştirilebilmesinin, saldırının daha gerçekçi bir tehdit olarak görülmesini sağlayacağını ve kriptografik sistem tasarımcılarını, atağa karşı ek önlemler almaya teşvik edeceğini düşünüyoruz.

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Erkay Savaş for his valuable support and guidance throughout my studies. This thesis would not be written without his encouragement. I would also like to thank Cemal Yılmaz for his guidance and valuable contributions to this thesis. I am also grateful to the other members of the jury, Prof. Dr. Albert Levi, Asst. Prof. Murat Kaya and Asst. Prof. Cengiz Örencik, for reviewing my dissertation and providing valuable suggestions.

The last but not least, I would like to thank my wife for her continuous support during my thesis studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1

INTRODUCTION

Cryptography, the art of protecting secrets, has been being used for centuries. Even though it might be considered as an art in its early ages, it has been being studied as a science for more than half a century. Thus, the science of cryptography has advanced tremendously since the era of simple ciphers. Now, the modern cryptography, rises upon complex mathematical foundations.

The desire to conceal secrets from other people has led to the rise of a counter-desire: breaking the secret messages. Thus, in modern cryptographic studies, the cryptographic algorithm design and the cryptanalysis (i.e. breaking the ciphers) have developed in constant interaction with each other. New encryption techniques have stimulated the efforts to find new cryptoanalytical methods to break the ciphers and obtain the secret messages; the new cryptanalytical methods are, in turn, considered to design new algorithms that withstand them. In the early times of cryptanalysis, researchers were relying exclusively on the mathematical analysis of cryptographic algorithms in order to find theoretical weaknesses to break the ciphers. However, in 1996 Kocher [1] introduced a new type of attack, which uses the timing measurements taken during the execution of the private key operations on a real hardware. His attack leveraged an implementation flaw instead of a theoretical weakness of the underlying mathematical algorithm. And he was able to break the cipher. With his study, Kocher introduced a new field in cryptanalysis: side-channel analysis (or side-channel attacks).

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, execution footprints in the micro-architectural structure of underlying microprocessor, 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.

1.1 Motivation

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 [4–7]. The cache access patterns of cryptographic programs are exploited by the cache-based side-channel attacks. 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 guesses to infer the secret key or to reduce the size of the key space.

The majority of the cache attacks rely on the so-called *cache cleaning* operation executed by a spy process, which can be detected easily [8] that evicts all or a part of data of cryptographic process from the cache before the start of an encryption operation. Bernstein's attack [9], which can be categorized as a timing-based attack is the only exception. Bernstein's attack is applied in a client-server setting. The attack tries to infer the secret key, which resides in a server that employs a software implementation of the Advanced Encryption Standard (AES) [10] to encrypt incoming messages, by using the variations in the encryption times of randomly generated messages.

The attack consists of two main phases. In the first phase, known as the profiling 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 (attack) phase extracts a similar model on the target machine, where the secret key is unknown, and correlates these two obtained models to make inferences about the secret key, i.e. reduces the key space 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 [11], Neve explains the reasons that lead the Bernstein's attack to success. 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 (i.e. eviction of the cache lines used by the AES server), which in turn makes the attack possible.

Bernstein's original attack is very interesting and needs for further study to understand its main cause and remedy the implementation accordingly. Bernstein's attack is powerful because it does not count on any spy process; yet it is also weak as it needs an identical target machine to perform its profiling phase. It is powerful, but the sources of the internal leakage is neither understood nor investigated well, which may pose a risk for the similar systems that can have the same leakage sources. It is weak, since obtaining an exact replication of the target system and emulating all its machine specific cache effects can be very difficult if not impossible. This dissertation aims to investigate and produce solutions on these two open areas: i.e. (i) to devise a methodology to find the internal leakage sources, (ii) to remove the profiling phase to make the attack more realistic.

1.2 Contributions

This thesis presents two novel solutions which address the open areas of the Bernstein's cache-timing attack. The first solution presents a methodology to find the code blocks that are in contention with each other by using Hardware Performance Counters (HPC). And the second solution proposes a new methodology to model the cache timing-behavior of an application, where the inferred cache timing-behavior model is used to remove the profiling phase of the Bernstein's cache-timing attack.

The major contributions of this thesis are summarized as follows:

- We propose to use the HPCs in order to count the L1 data cache misses and to use this information the detect which code blocks are in contention with each other (Section 4.1).

- We demonstrate by experiments that the proposed methodology is able to find the colliding code blocks (Section 4.2.2).
- We propose a methodology to model the cache timing-behavior of an application. The proposed method does not need to know all the architectural properties of the underlying system. Only the cache line size is enough to extract the proposed model (Section 5.2).
- We utilize the proposed cache timing-behavior model to remove the profiling phase of the Bernstein's attack, which makes the attack more realistic for the real computing platforms (Section 5.3.1).
- We validate the proposed model by conducting Bernstein's cache-timing attack successfully without a profiling phase (Section 5.3.2).

1.3 Outline

The organization of this thesis is as follows: Chapter 2 presents the related work on cache-timing attacks; Chapter 3 provides the background information on AES, Bernstein's attack, a variation of Bernstein's attack on the last round of AES, CPU caches and HPCs; Chapter 4 outlines our methodology to find the colliding code blocks of a program that cause leakage and presents the experimental results; Chapter 5 explains how we extracted a cache timing-behavior model from the cache line size, how this model is used to remove the profiling phase of the Bernstein's attack and presents the validation results; Chapter 6 concludes the thesis.

Chapter 2

RELATED WORK

In the literature there are many works that use the shared micro-architectural resources as side-channels. In [4, 12] Aciicmez et al. leverage the branch prediction unit to infer the secret keys. Again, Aciicmez et al. show that the instruction cache can also be used as a side-channel to obtain the secret keys [13–15]. Side-channel attacks which exploit the data caches are also widely studied in the literature [5, 9, 16–24]. In this thesis we focus on the data cache based side-channel attacks. The first notions on data cache based side-channel attacks are reported in [1] and [25]. In [1], Kocher states that cryptographic applications may take different times for varying inputs. He also explains that the reasons behind these variations include cache hits, performance optimizations etc. He performs a timing attack against private key operations and remarks that the tables used in the implementations of algorithms such as Blowfish [26], SEAL [27] and DES [28] can produce timing variations due to cache hits and misses. Kelsey et al. [25] propose a timing attack against IDEA [29] and state that ciphers such as Blowfish [26] and CAST [30] can be vulnerable to cache attacks due to their large S-boxes.

Cache-based side-channel attacks can be divided into three main 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 generate intentional cache conflicts with the cryptographic application to obtain the cache access patterns of the latter. In their proposed approach [16], Tromer 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 the attack Tromer et al. describe two cases: synchronous and asynchronous. In the synchronous case, the attacker is capable of starting an encryption operation at will and he has two different measurement methodologies to detect the colliding cache lines: *Evict+Time* and *Prime+Probe*. By using *Evict+Time*, the attacker first starts an encryption operation, then accesses to some specific memory addresses (i.e. AES table addresses) to cause evictions and finally starts another encryption and measures the execution time. Here the first encryption brings the AES tables into the cache, the memory access in the second step causes evictions if the same cache lines are used by the two processes and the final step extracts which cache lines are accessed since the execution time is directly effected by the cache hits and misses. The *Prime+Probe* method works in a different way. The attacker first accesses to every memory location of the AES tables to bring them into the cache. Then he triggers an encryption and afterwards he performs another memory access. Again by measuring its memory access times he deduces which cache lines are accessed by the cryptographic process. In the synchronous attack, they show that the full AES key is recoverable after 300 encryptions on a Athlon 64 system by *Prime+Probe* measurements. The asynchronous attack has extra limitations, such that the spy process is not allowed to interact with the target process. Thus, the attacker will not be able to trigger an encryption. The attack is conducted only with the memory access patterns of the spy process. The attacker accesses to a set of memory blocks repeatedly and measures the access times. If the spy process conflicts with the target process, memory access will be longer than usual. This timing information is used to detect the conflicting cache lines and to conduct the attack. In the asynchronous case, one minute of monitoring of encryption operations with the same secret key reveals 45.7 bits of the AES key.

Trace-driven cache attacks (i.e., the second category of attacks) were, on the other hand, first introduced in [5]. 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 monitoring electro-magnetic or power 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 [5], 2^{10} encryptions are performed to collect traces and then an offline analysis phase is run with a computational complexity of around 2^{32} steps. As a result, 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 [20, 31, 32].

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. This dependency can be exploited to extract the secret key. In [18] 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. They propose two methods to infer the secret key: non-elimination table attack and elimination table attack. In the non-elimination attack, by using the shorter encryption timings the probable key guesses are extracted and a search for the secret key is executed. However, in the elimination attack, they use the longer encryption timings to guess the improbable keys and they reduce the size of the key space to search. When they apply the elimination table attack approach on DES algorithm, they show that the algorithm is broken after 2^{23} known plaintexts. They also state a success rate greater than 90% after 2^{24} operations.

Bernstein's attack, which is the main focus of this thesis, is also a time-driven attack. Neve gives an explanation as to why Bernstein's attack works [11]. Neve examines the behavior of the cache accesses during the AES encryption process and observes that variations in AES execution times are due to deterministic and naturally occurring system dependent cache evictions. However, Neve does not identify the exact root cause creating these cache contentions. In this thesis, we take one step further and propose a methodology to discover which process(es) or code segments cause these deterministic and system dependent cache evictions [33]. Bernstein's attack is a profiled timing attack. In the literature there are works that aim to improve the profiling phase and chose the best attack strategy by mathematical analysis [34]. But, to the best of our knowledge, there are no studies that try to execute the profiling phase analytically without performing any physical measurements. As a part of this thesis, we present a methodology to extract the profiling phase model analytically and propose conducting Bernstein's attack with these analytically extracted models [35].

Chapter 3

PRELIMINARIES

In this chapter, we provide the necessary background information about the basic properties of CPU caches and hardware performance counters, the details of the AES algorithm and its table based implementation, 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

In today's modern processors, the memory system itself is a hierarchy of memories, where each level in the hierarchy has different speeds of access times. The memory hierarchy is mainly comprised of registers, several levels of cache, main memory and secondary memory. The memory access times of these different levels are summarized in Table 3.1 [36].

Table 3.1: Memory hierarchy access times

Memory level	**Access time**
Register	0.3ns - 2ns
Level 1 cache	3ns - 8ns
Level 2 cache	6ns - 10ns
Main memory	10ns - 70ns
Fixed rigid disk	3ms - 15ms

As seen from the Table 3.1, the closer the memory to the processor, the faster it is. Faster memory means more advanced technology, thus being more costly. Because of the cost, the size of the memory gets smaller, as it comes closer to the processor.

In this memory hierarchy, cache is a fast and small memory, which stores the data that the processor uses and will probably use again in the near future. Figure 3.1 shows an example of memory hierarchy which has two levels of cache [37]. 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.

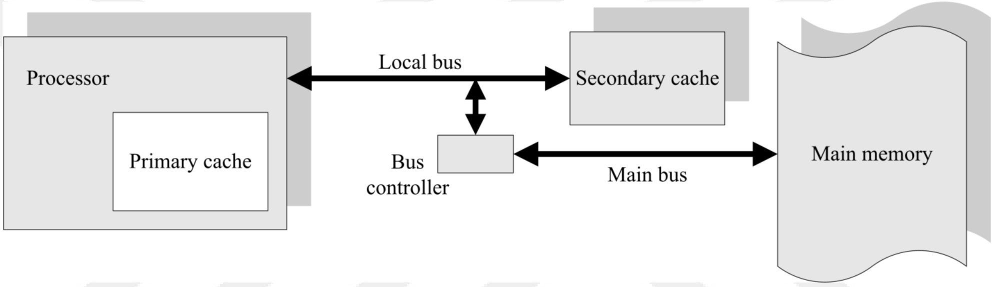


Figure 3.1: An example of a memory system

Processors tend to execute the codes sequentially, which means after executing instruction x, most probably instruction $x + 1$ will be executed. The same rationale is valid also for the data. In a typical program flow, data is accessed sequentially most of the time (e.g. arrays). Also, in a program, same variables are used again and again. This situation is called as *principle of locality* and can be examined under three categories:

- Temporal locality: Used variables tend to be used again.
- Spatial locality: Data is accessed sequentially as in arrays.
- Sequential locality: Most of the time instructions are fetched sequentially.

Thus, according to the spatial locality, if we fetch a piece of data from memory, most likely the processor will also need the data which is adjacent to the formerly fetched data. So, when a miss occurs in the cache, not only the missing data is fetched, an entire block which contains the missing data is fetched to the cache. The fetched block is 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.

A typical cache consists of *sets*, where each set contains one or more cache lines. The method of placing a data in the memory to a cache location is called *mapping* [38]. Mapping is done according to some specific address bits and each address maps to a single set. We can name three groups of caches: (i) *direct mapped* cache, (ii) *N-way set associative* cache, (i) *fully associative* cache.

In a direct mapped cache, each set contains one cache line. Figure 3.2 shows which fields of the address is used to map the cache location.

Memory Address	Tag	Set	Byte Offset
	27-bits	3-bits	2-bits

Figure 3.2: Cache fields for a direct mapped cache

The cache fields in Figure 3.2 belong to a 32-bit processor, which has an 8-set direct mapped cache, where each cache line is one word. The least significant two bits determine the byte offset in the relevant word. The following three bits are used to select the cache line that will accommodate the main memory data. Since different addresses map to a single cache line, the actual data address must be also kept. Thus, the remaining significant bits are also stored in the cache as *tag* to indicate the real address of the data in the cache line. Figure 3.3 depicts the basic architecture of an 8-set direct mapped cache.

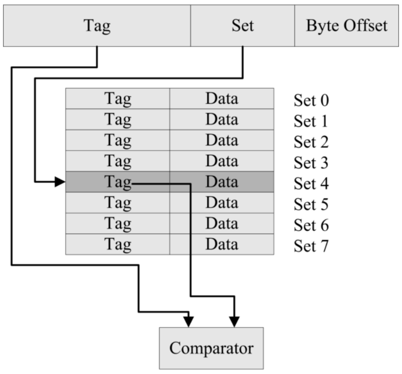


Figure 3.3: Basic direct mapped cache architecture

If any two physical addresses map to the same cache location, a cache conflict occurs and the data currently resides in the cache is evicted. In the case of direct mapped caches, since there is only one cache line in a set, two addresses that map to the same cache set always cause a conflict. Figure 3.4 shows an example of how the physical addresses are mapped to a direct cache.

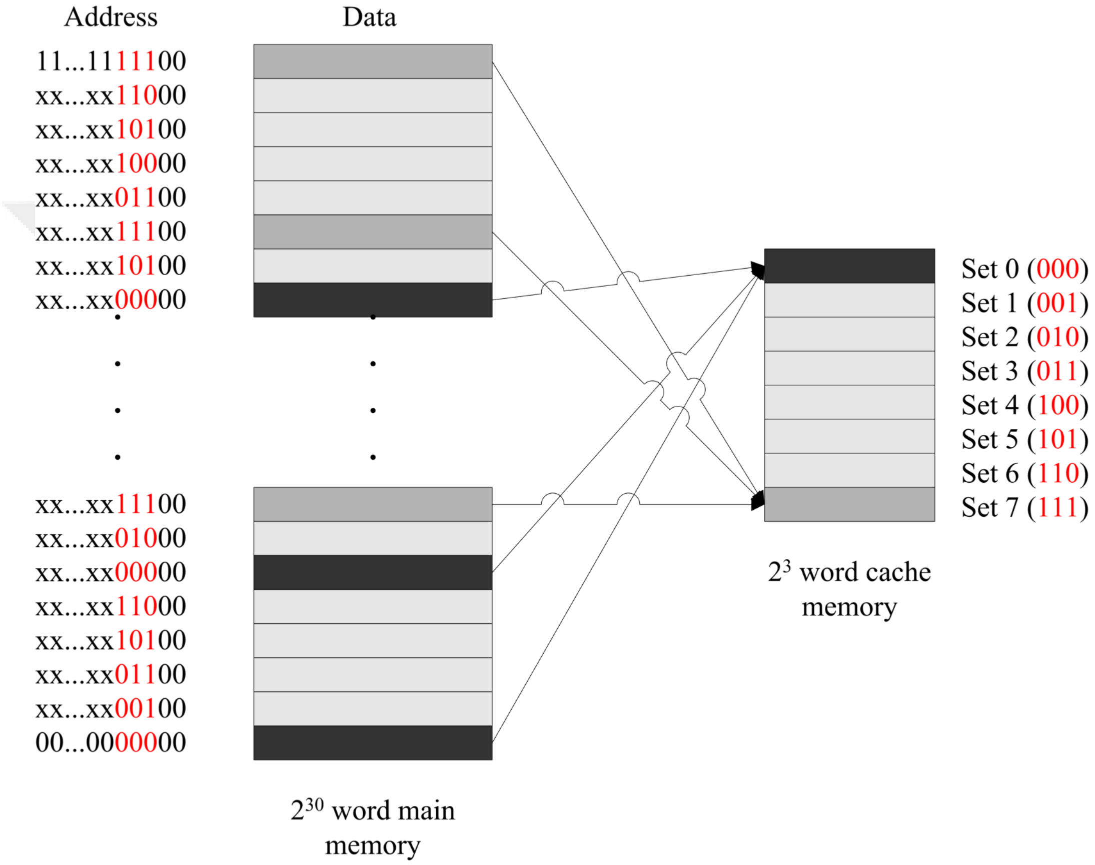


Figure 3.4: Direct mapped cache address mapping example

However, the situation is different in N-way set associative caches. In this cache architecture each set has N cache lines and an address that maps to a set can be accommodated in any one of these cache lines. Hence, this architecture reduces the number of cache misses. Similarly, a fully associative cache is a B-way associative cache, where B is the number of total cache lines in the cache. Further information about the cache architectures and operating principles can be found in [36–38].

3.2 Hardware Performance Counters (HPCs)

Hardware performance counters are hardware-resident counters that record various events occurring on a processor. Today's general-purpose CPUs include a fair number of such counters, which are capable of recording events, such as the number of instructions executed, the number of branches taken, the number of cache hits and misses experienced, etc [39,40].

Hardware performance counters are mainly used for software profiling. Engineers use them to improve the performance of their software. The actual hardware performance counter numbers in a CPU varies among the vendors and varies even in the different models by the same vendor. For example, Intel CPU families support a set of architectural and non-architectural performance events [39]. Non-architectural events are model specific and not compatible among the CPU families. However, architectural events are same among the CPU families and they provide a uniformity in different classes of CPUs. The required information about the hardware performance counters of a specific processor can easily be found among the documentation of the relevant CPU.

Even though not all the HPCs are same, their working principle is similar. Hardware performance counters are by default inactivated. To activate them, a code indicating the type of event to be counted and the physical counter to be used for counting is written to a register and then the CPU is instructed to start the counting process. There are often a number of physical counters present in a CPU and these counters can individually be paired with any event known to the CPU. Once activated, hardware counters count the events of interest during program executions and store the counts in a set of special purpose registers. These registers can then be programmatically read and reset, and hardware counters can be deactivated as needed. In most cases, kernel must be patched to access these counters and privileged user rights are required to program the performance counter control registers. However, there are many libraries such as PAPI [41] that provide a simple interface to program and read the performance counters.

Hardware performance counters have been traditionally used (i) for performance debugging to identify hotspots in the program code [42], (ii) for functional correctness debugging, where they serve as an abstraction mechanism [43–45], and (iii) for carrying out side channel attacks [24]. This work differs in that we use hardware performance counters to identify the sources of cache contentions, as well.

3.3 Advanced Encryption Standard (AES)

AES [10] is a symmetric-key block cipher algorithm. AES is announced by the U.S. National Institute of Standards and Technology (NIST) as FIPS PUB 197 in 2001. The original algorithm, which was submitted to the AES competition, is called Rijndael. The AES standard is a variant of the Rijndael algorithm which has a fixed block length of 128 bits. To provide different security levels, AES can work with key sizes of 128, 192 or 256 bits. It accepts plaintext inputs as blocks of 128 bits and outputs 128 bits ciphertexts for each input block.

AES operates on a 4×4 matrix of bytes which is called as the state. Before beginning of the encryption or decryption, the input is copied to the state matrix, computations are performed on this state matrix and in the end, the result is copied to the output array. This process is depicted in Figure 3.5.

$$\begin{array}{|c|c|c|c|}\hline i_0 & i_4 & i_8 & i_{12} \\ \hline i_1 & i_5 & i_9 & i_{13} \\ \hline i_2 & i_6 & i_{10} & i_{14} \\ \hline i_3 & i_7 & i_{11} & i_{15} \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|c|c|}\hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ \hline s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ \hline s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ \hline s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|c|c|}\hline o_0 & o_4 & o_8 & o_{12} \\ \hline o_1 & o_5 & o_9 & o_{13} \\ \hline o_2 & o_6 & o_{10} & o_{14} \\ \hline o_3 & o_7 & o_{11} & o_{15} \\ \hline \end{array}$$

Figure 3.5: AES input, state and output transition

AES computations are done in rounds. The number of rounds depends on the key size. AES has 10, 12 and 14 rounds for the key sizes 128, 192 and 256, respectively. Every round uses a different 128-bit round key which is generated from the master key according to the AES key expansion algorithm.

AES starts encryption with an initial `AddRoundKey` operation and each following round consists of the `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` operations in the given order. However, the last round of the AES does not involve the MixColumns operation. AES performs decryption by computing the inverse operations in the reversed order (note that all round operations in AES algorithm are reversible). AES rounds are shown in Figure 3.6. In the figure N_r denotes the number of rounds.

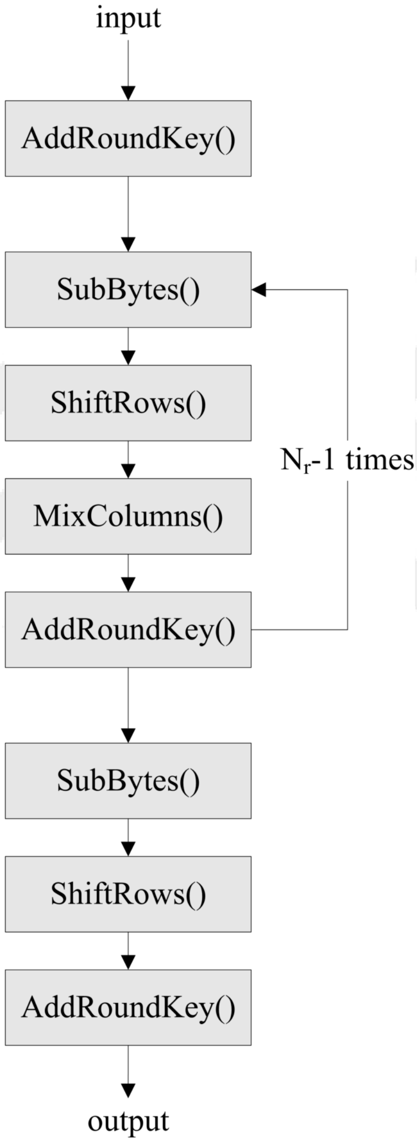


Figure 3.6: AES rounds

AddRoundKey operation performs bitwise XOR between the bytes of the state and the bytes of the round key. SubBytes provides a non-linear substitution of the state bytes. Each byte of the state is replaced with another byte according to a nonlinear transformation that is derived from the Rijndael S-Box. SubBytes operation is depicted in Figure 3.7.

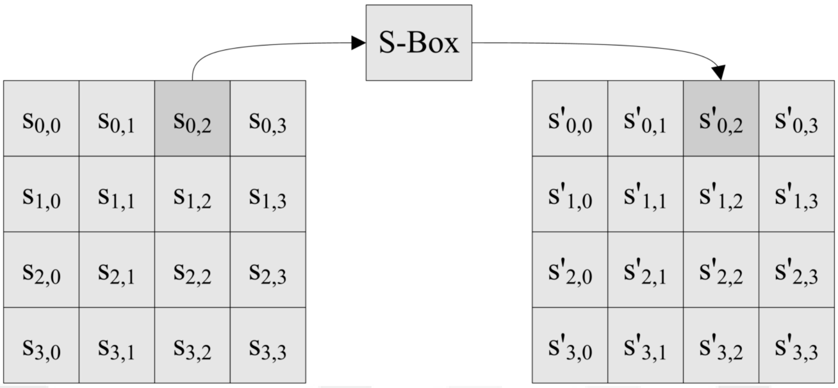

S-Box
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,1}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$

Figure 3.7: AES SubBytes operation

ShiftRows operation cyclic left shifts each row by a predefined offset. Row number 0 is not shifted, row number 1 is shifted once, row number 2 is shifted twice and row number 3 is shifted three times. Figure 3.8 shows how the state changes after the ShiftRows operation.

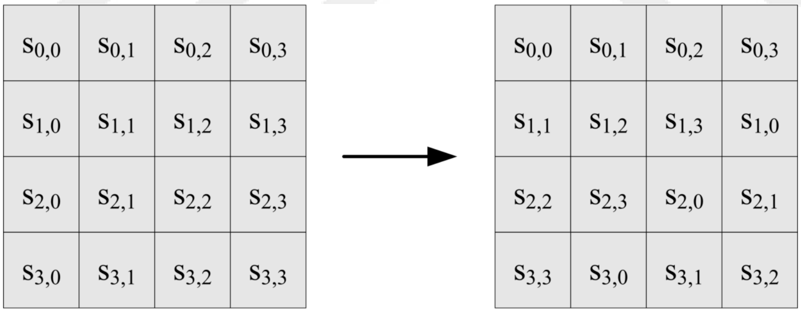

$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,1}$ $s_{1,2}$ $s_{1,3}$ $s_{1,0}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,1}$
$s_{3,3}$ $s_{3,0}$ $s_{3,1}$ $s_{3,2}$

Figure 3.8: AES ShiftRows operation

MixColumns performs a linear transformation on the columns of the state. Each column of the state constitutes a polynomial over $GF(2^8)$ and a modular polynomial multiplication is executed between each column and the fixed polynomial $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$ in modulo $x^4 + 1$. Figure 3.9 shows the MixColumns operation.

The straightforward AES implementations perform all the round computations sequentially. However, AES can be implemented faster on platforms which have a 32-bit or larger word size. Instead of calculating each round operation separately, SubBytes,

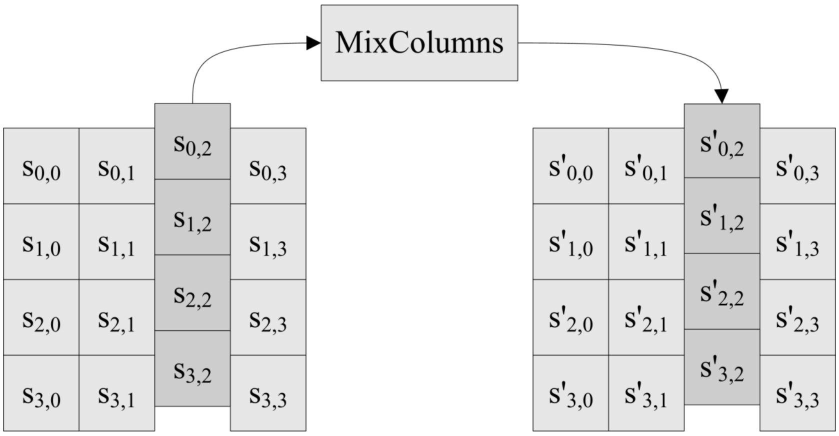

MixColumns
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,1}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$

Figure 3.9: AES `MixColumns` operation

`ShiftRows` and `MixColumns` operations can be combined in a lookup table, which is named as AES T-table. In such an implementation, except for the last round, four tables ($T0$, $T1$, $T2$, $T3$) are needed where each table has 256 entries of 32-bit entry size. Thus, each table requires 1024 bytes and in total four tables require 4 KB in the memory. Since the last round does not perform `MixColums` operation another table ($T4$) is needed. One table with 256 entries of 32-bit entry size suffices to implement the last round. With this implementation, one round can be computed by 16 table lookups and 12 32-bit `XOR` operations, followed by the four 32-bit `XOR` operations of the `AddRoundKey` step. Figure 3.10 depicts how next state s^{i+1} is computed from current state s^i and T-tables. The only difference in the last round is the usage of $T4$ table instead of $T0$, $T1$, $T2$ and $T3$.

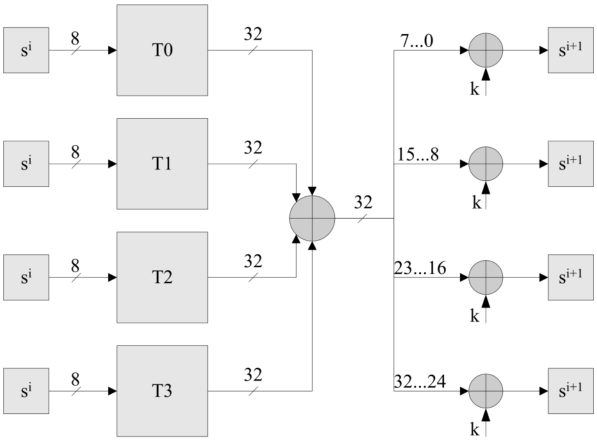


Figure 3.10: Table-based AES implementation

3.4 Bernstein's Cache-Timing Attack

Bernstein presents a cache based timing attack, which targets the lookup table based OpenSSL implementation of AES [9]. In his attack, he aims the first `AddRoundKey` operation of the AES encryption. He exploits the timing differences of the AES execution times that are caused by the T-table accesses. He claims that the T-table lookup indexes $k[j] \oplus p[j]$, where $j = 0, 1, \ldots, 15$, k denotes the first round key bytes and p denotes the plaintext bytes, affect the table access time and this time is highly correlated with the AES execution time.

In his attack he defines two different roles: 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 the elapsed timing. After gathering sufficient number of timing measurements, the client performs a statistical computation and obtains the secret key. The attack consists of four phases: *profiling*, *attack*, *correlation* and *key search*.

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. 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 $i = 0, 1, \ldots, 15$. The attacker obtains the execution time of each encryption and saves it along with the indexes. The attacker stores the average timing information for each byte and each value of that byte in a two dimensional array $t[16][256]$. The attacker also keeps track of the overall average execution time. At the end of the profiling phase, the attacker computes a model for each byte. This model basically shows the timing difference of the average individual byte execution time from the overall average execution time. Figure 3.11 shows such a model. In the figure, x-axis shows all the possible values from 0 to 255 that a byte can take and y-axis gives the timing difference according to the overall mean execution time. At the end of the profiling phase, a total of sixteen models, one for each byte, are constructed.

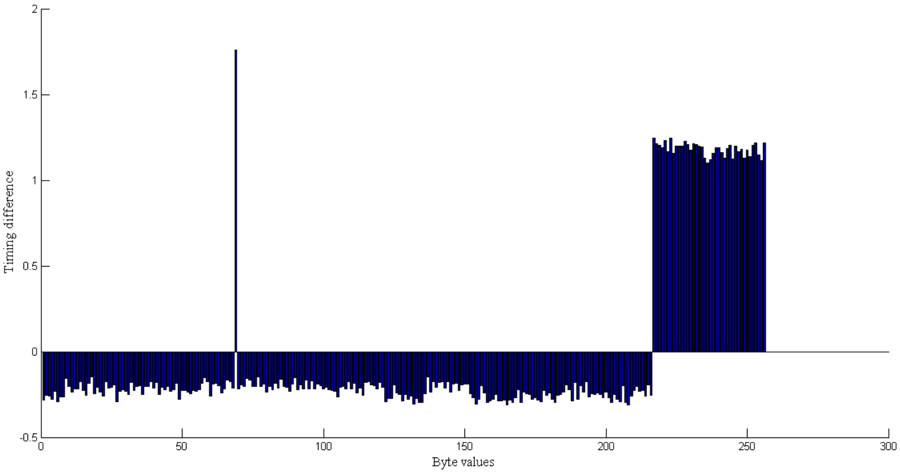


Figure 3.11: Example byte model obtained after the profiling phase

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, the timing profiles are constructed according to the plaintext values p_i' instead of lookup table indexes $s_i^{0'} = p_i^{'} \oplus k_i^{0'}$. Again, the attacker obtains the execution time of each encryption and saves it along with the indexes (i.e. this time plaintext byte values). The attacker stores the average timing information for each byte and each value of that byte in another two dimensional array $t'[16][256]$. The model obtained after the attack step is shown in Figure 3.12.

Figure 3.12 belongs to the same byte model as in Figure 3.11. As can be seen from the figure, since the secret key is not known, it looks like a scrambled model and does not provide any useful information.

Correlation phase is executed after the attack phase. In this phase, for each model found in the attack phase all possible key byte values are tried and a timing profile of indexes ($s_i^{0'} = p_i^{'} \oplus k_i^{0}$) are obtained for each key candidate where $i = 0, 1, \ldots, 15$ and $k_i = 0, 1, \ldots, 255$. Then, each of the timing models in the attack phase is correlated with the timing model obtained in the profiling phase. The key byte candidates are sorted in a decreasing order according to their correlation scores. Also a selection threshold is

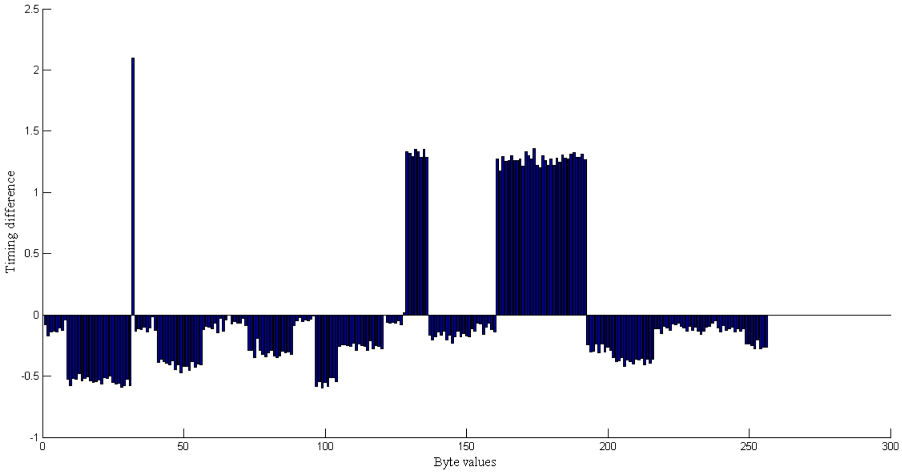


Figure 3.12: Example byte model obtained after the attack phase

applied on the correlation results of the key byte candidates. Hence, only the most probable key candidates appear on the results. In summary, correlation phase tries to find the model which is the closest to the profiling phase model by trying all the possible byte values. Figure 3.13 shows the model, which belongs to byte 15, that gives the highest correlation for the attack phase byte model depicted in Figure 3.12. One can easily notice the resemblance between Figure 3.11 and Figure 3.13 which is consistent with the obtained high correlation value.

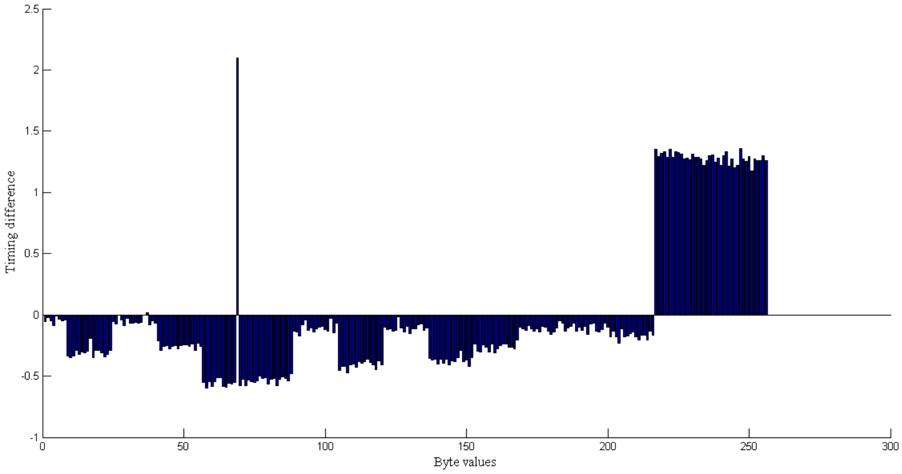


Figure 3.13: Example byte model obtained after the correlation phase

Table 3.2 also gives the correlation results for the example data used in Figures 3.11 and 3.12, where the secret key is: {ce fb 74 0a a9 55 d3 1d ed 29 81 4c 25 72 5b 93}. In Table 3.2 first column denotes the number of possible values, second column denotes the byte number of the key and the last column denotes the possible values for that key byte in the descending order.

Table 3.2: Correlation phase results for the example data set

240	00	f8 ff fa f9 fc fd fb fe e3 e7 e2 e1 e0 e6 e4 81 ... (ce at rank 66)
255	01	bc bd bb b8 ba a5 bf a2 a7 be a6 e0 e3 a0 b9 e5 ... (fb at rank 57)
016	02	37 32 31 34 35 36 33 30 2c 2f 2d 2b 2e 29 28 2a (missing 74)
016	03	0e 0b 0f 09 0a 0c 08 0d 13 12 17 14 10 11 15 16
016	04	b6 b0 b3 b7 b5 b2 b1 a9 af aa ac b4 a8 ae ab ad
096	05	4e 4d 51 48 4a 4c 53 49 4b 56 55 4f 54 50 57 52 ...
001	06	d3
016	07	1c 1d 1a 19 1f 18 1b 1e 07 00 04 06 03 01 02 05
119	08	ee eb ed ec e8 e9 ea ef f5 f0 f3 f6 f2 f1 f7 f4 ...
256	09	34 32 37 36 33 7d 31 0f 63 35 7a 30 66 78 65 0d ... (29 at rank 50)
008	10	81 87 83 84 86 82 80 85
017	11	4e 4c 4b 49 4d 4a 48 4f 50 56 57 51 54 53 52 55 ...
080	12	26 23 21 24 27 20 22 25 3f 3e 3c 3d 39 38 3b 3a ...
110	13	73 71 77 75 70 72 76 74 33 37 31 34 35 32 36 30 ...
001	14	5b
008	15	96 93 94 92 91 90 97 95

Finally, the key search phase simply performs a brute force key search on the remaining key space, using a known plaintext and ciphertext pair.

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. 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. For more information about the attack, the interested reader can profitably refer to [9, 11].

3.5 Applying Bernstein's Cache-Timing Attack to the Last Round of AES

Bernstein's original attack can recover at most half of the 128-bit AES key (when 64 B cache blocks are used) since only the upper nibble of each key byte is used to determine the cache set that holds the corresponding entry. Neve [11] demonstrated that the other half of the secret key can be obtained if the Bernstein's attack is combined with a similar attack targeting the second round. However, the version of the AES algorithm used in both works [9, 11] (OpenSSL v0.9.7a) allows an easier attack on the last round of AES that has the potential of recovering the entire key. Similar to the original Bernstein attack, the AES client (i.e. adversary) performs the *profiling*, *attack*, *correlation* and *key search* phases by aiming the last round table lookup operations. This new improved attack allowed us to recover 128-bits of the key in some cases.

The profiling phase is executed with a known key, similarly as in the original attack case. AES client sends randomly chosen plaintexts and receives the resulting ciphertext and the timing measurement data. During the last round of the AES [10], a separate table, namely $T4$, which basically implements the AES `SubBytes` operation in the last round, is used. The number of cache hits and misses that occur during the accesses to this table affect the overall execution time of the encryption. 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 byte models. In the profiling phase, the outputs of $T4$ lookups used in the last round can be computed using the formula

$$\texttt{InvShiftRows}(c_i \oplus k_i^{10}), \tag{3.1}$$

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. 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 $\mathcal{T}_i^p[256]$ where i is the order of the $T4$ access and $i = 0, 1, \ldots, 15$. An example byte model obtained after this phase is given in Figure 3.14.

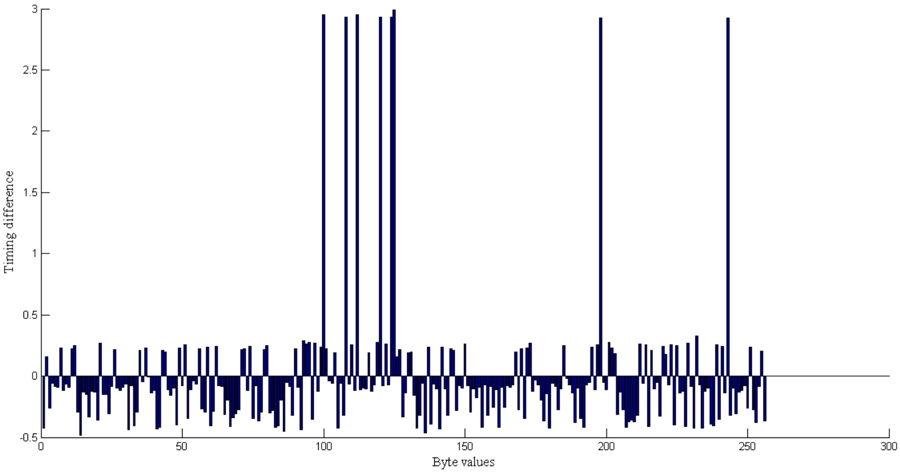


Figure 3.14: Example $T4$ output byte model obtained after the profiling phase

After the profiling phase, the attack phase is executed similarly. In this phase, the secret round key byte $\tilde{k}_i^{10}$ is unknown, hence we obtain one timing profile for each candidate of the corresponding key byte using the following equation

$$\texttt{InvShiftRows}(c_i'), \tag{3.2}$$

where c_i' stands for the i^{th} bytes of the ciphertext. Thus, at the end of attack phase we obtain a timing profile, namely the array $\tilde{\mathcal{T}}_i^a[256]$. Figure 3.15 shows an example byte model of an attack phase data.

In the correlation phase, for each $T4$ output byte model, all possible timing profiles, namely $\mathcal{T}_{i,k}^a[256]$, are constructed by using $\texttt{InvShiftRows}(c_i' \oplus k)$, where $k = 0, 1, \ldots, 255$. Then, the timing profiles in the attack phase $\mathcal{T}_{i,k}^a$ are correlated to the timing model of the profiling phase $\mathcal{T}_i^p$. 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., $\tilde{k}_0^{10}$, ..., $\tilde{k}_{15}^{10}$. Figure 3.16

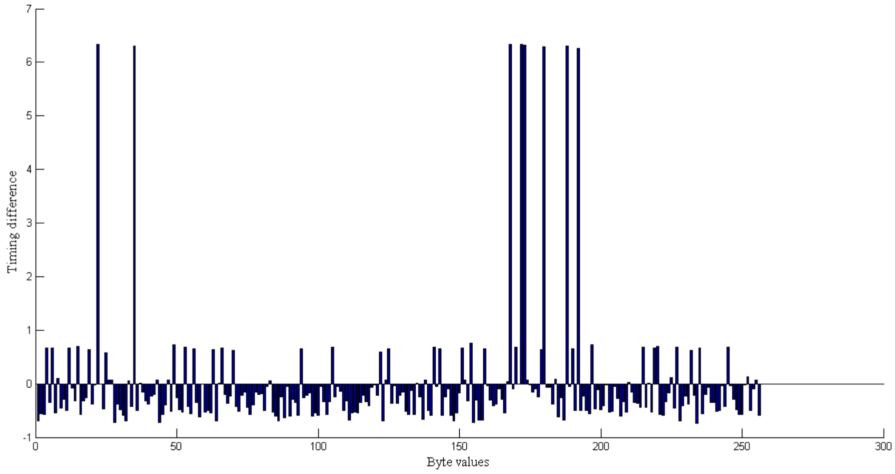


Figure 3.15: Example $T4$ output byte model obtained after the attack phase

shows the model, which belongs to byte 1, that gives the highest correlation for the attack phase byte model depicted in Figure 3.15.

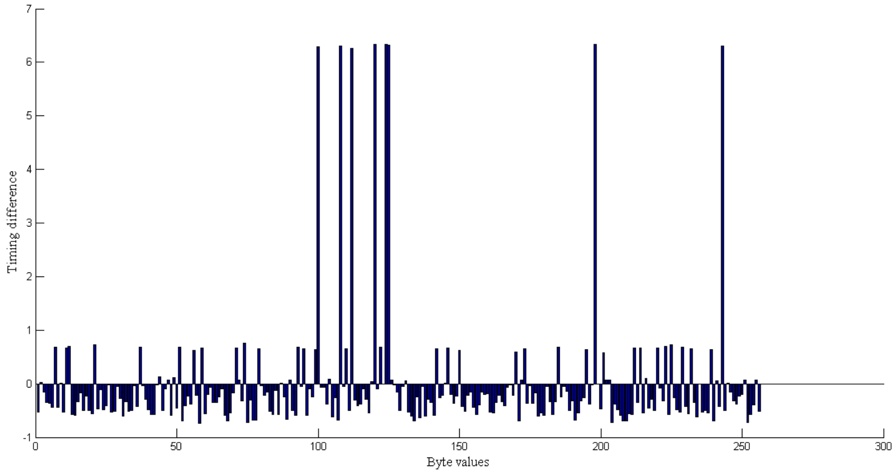


Figure 3.16: Example $T4$ output byte model obtained after the correlation phase

After the correlation phase, a brute force key search is applied to fully recover the key. Table 3.3 presents the correlation results for the example data used in Figures 3.14 and 3.15, where the secret key is: {2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c} and the last round key is: {d0 14 f9 a8 c9 ee 25 89 e1 3f 0c c8 b6 63 0c a6}. Again, in Table 3.3 first column denotes the number of possible values, second column denotes the byte number of the key and the last column denotes the possible values for that key byte in the descending order.

Table 3.3: Last round attack correlation phase results for the example data set

001	00	d0
001	01	14
001	02	f9
001	03	a8
001	04	c9
001	05	ee
001	06	25
001	07	89
001	08	e1
001	09	3f
001	10	0c
001	11	c8
256	12	71 70 36 2e 33 cb 62 fc 60 ea ... (b6 at rank 251)
256	13	04 59 a5 2f ef a3 63 ...
256	14	ea 23 25 16 67 77 ae 67 8d 33 14 dc 0c ...
256	15	fb 84 f3 78 cc 3f 01 5b 82 a6 ...

An important point to note here is that, 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 (i.e. every bit affects the relevant byte model), the last round attack can reveal the entire key as opposed to the half of the key in the first round attack.

Chapter 4

ISOLATING the LEAKAGE SOURCES in CACHE-TIMING ATTACKS

Bernstein [9], exploits the differences between the encryption times of randomly generated messages to extract secret keys processed by an OpenSSL implementation of AES (Advanced Encryption Standard [10]). Neve [11] demonstrates that the exploitable timing differences in the Bernstein's attack occur due to some conflicting L1 data cache accesses with the AES cache accesses, causing some table entries used by AES to be evicted from the cache, e.g., messages that experience cache misses take longer to encrypt than those messages that don't experience any cache misses.

An important observation we make in the Bernstein's attack is that, unlike many other side channel attacks, in which the attack is performed by using a spy process that intentionally creates cache contentions, the Bernstein's attack operates without using any spy processes. This observation strongly suggests that the cache contentions exploited in the Bernstein's attack are unintentionally caused by some entities in the cryptographic system. We value this observation, because such inadvertent contentions should be considered as a flaw in the implementation of cryptographic systems. This, consequently, necessitates a software analysis framework to identify the primary sources of these contentions and check the effectiveness of proposed countermeasures, to further improve the reliability of cryptographic applications.

In this chapter, we present an approach to identify code segments in cryptographic applications that are unintentionally in cache contentions with each other, thus leaking information that can be exploited in side-channel attacks to extract secret keys. This is important because locating the sources of information leakage in software implementation of cryptographic applications provides developers an invaluable opportunity to fix them, thus prevent side-channel attacks. An integral part of the proposed approach is to quantify the contentions in cache memory. For that we use hardware performance counters to count the number of misses occurring in cache memory. We furthermore evaluate our approach by conducting a series of experiments on the well-known Bernstein's attack. The results of these experiments demonstrate, (to the best of our knowledge) for the first time, that the part of the OS kernel that handles socket communications is in cache contention with the AES encryption code and that these contentions are the primary source of information leakage, making the Bernstein's attack a success.

The remainder of the chapter is organized as follows: Section 4.1 introduces the proposed approach; Section 4.2 presents a case study in which we applied the proposed approach to the Bernstein's attack; Section 4.3 discusses the current state of the work and provides ideas for future work; Section 4.4 provides concluding remarks on the chapter.

4.1 Approach

We, in particular, present an approach to detect code segments in cryptographic algorithms, which are in cache contention with each other, thus possibly leaking information that can be exploited in cache-based side-channel attacks to extract secret keys. This is important because locating the root causes of information leakage in software implementation of cryptographic applications provides developers an invaluable opportunity to fix them, thus prevent side-channel attacks.

In this work we divide a cryptographic application under analysis into a set of nonoverlapping *code blocks*.

Definition 1 *A code block is a code segment that takes a number of inputs and produces a number of outputs. Any data, which is not locally defined in a code block, but accessed from inside the block, is considered to be an input to the code block. Furthermore, any program state update made inside a code block, which is visible outside the block (including the side effects), is considered to be an output of the code block.*

In theory, a code block is in cache contention with another code block, when one block evicts at least one cache line used by the other block. One way to detect contentions is to use a cache simulator to determine such cache evictions. However, developing cache simulators is a non-trivial task as one may need to simulate a large portion of the underlying hardware platform, including the CPU and the RAM. Furthermore, it is often undesirable (if not infeasible) to execute an application together with its environment, such as the underlying OS, in a hardware simulator due to the runtime overhead introduced by the simulator.

We, therefore, developed an approach to empirically (i.e., via conducting experiments) determine code blocks that are in cache contention with each other. In this approach, to determine whether a code block A causes cache contentions in another code block B in program P, we first replace A in P with a mock code block A'. The output is a new program P'.

The mock block A' is constructed by using a simple capture-reply approach. In the capture mode, the original program P is fed with a set of concrete inputs I, the program executions are monitored, and the inputs to A together with the respective outputs of A are captured and recorded. In the reply mode, i.e., in P', the mock code block A' mimics the behavior of the original code block A by simply replying pre-recorded outputs for the respective inputs, without performing any actual computations.

We, then, feed P and P' with I and measure the number of cache misses experienced by B in both programs.

Definition 2 *In the original program P, the code block A is considered to be in con-*

tention with the code block B, if one can find a set of program inputs I such that the number of cache misses experienced by B in P' is statistically significantly *less than that of experienced by B in P; computing the same output in A in a different way significantly reduces the cache misses observed in B.*

Note that the same approach can readily be applicable to determine the contentions between two sets of code blocks.

To count the number of cache misses occurring in a code block, we use hardware performance counters. One challenge we encountered when first using hardware performance counters for this purpose was that the counters do not distinguish between the instructions issued by different processes. To deal with this, we used a kernel driver, called `perfctr` (http://linux.softpedia.com), which implements virtual hardware counters that can track hardware events on a per-process basis.

At a high level we count cache misses occurring in a code block as follows: 1) a hardware performance counter that counts L1 data cache misses is activated at the beginning of a program execution, 2) the value of the counter is read before and after the execution of the block and the difference is attributed to the block, and 3) the counter is deactivated at the end of the execution. If the code block is executed more than once, the number of cache misses are aggregated over all the executions.

4.2 Case Study

We evaluated the proposed approach by conducting a series of experiments. In these experiments, we applied the approach on the Bernstein's AES server to determine the code blocks in the server that cause cache contentions with the AES round tables, thus making the Bernstein's attack a success.

An important observation we make in the Bernstein's attack is that, unlike many other side channel attacks in which the attack is performed by using a spy process that intentionally creates cache contentions, the Bernstein's attack operates without using any spy process. This observation strongly suggests that the cache contentions in the AES server are unintentionally caused by some entities in the system.

We distinguish between two types of entities: external entities and internal entities. External entities are those entities that the AES server has no control over, such as the CPU architecture of the underlying platform and the other processes sharing the same hardware and software resources with the AES server. Internal entities, on the other hand, refer to those entities employed by the AES server, such as the server source code and the libraries used by the server.

4.2.1 Analyzing External Entities

In our case study, we first evaluated the effects of external entities on the success of cache-based side-channel attacks. To this end, we conducted a series of experiments, in which we carried out a slight variation of the Bernstein's attack, called the *last-round Bernstein attack*. This attack variation uses the same AES server and the same AES client as with the original Bernstein's attack. The only difference is that the last-round attack targets the last round of AES, rather than the first round of AES as is the case in the original Bernstein's attack.

We conducted the last-round attack on different hardware and software platforms with varying client-server deployment configurations. In each experiment setup, to carry out the attack, the malicious AES client used 2^{30} randomly generated messages, each of which was of size 600 bytes. The AES server, in turn, used the OpenSSL v0.9.7a (Feb 19, 2003) implementation of AES with a 128-bit secret key, the same implementation used by the original Bernstein's attack (http://www.openssl.org/source/), for the encryptions.

Table 4.1 presents the results we obtained. In this table, the first two columns depict the CPU and the operating system 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 they are on different cores. In the experiment platforms, each core had its separate L1 data cache.

For each setup, we carried out many experiments. For each key byte, the attack produced a set of candidate key values, prioritized by their likelihood. The cross product of

Table 4.1: Results obtained from the last-round Bernstein attack.

Processor	Operating System	AES Client-Server Deployment Configuration	Size of the Reduced Key Space
Intel Pentium P6200	Ubuntu 3.0.0-12 kernel	same core	2^{32}
Intel Pentium P6200	Ubuntu 3.0.0-17 kernel	different cores	2^{49}
Intel Core 2 Duo P8400	Ubuntu 3.0.0-12 kernel	same core	1
Intel Core 2 Duo P8400	Ubuntu 3.0.0-17 kernel	different cores	2^{24}
Intel Xeon E5405	CentOS 2.6.18 kernel	same core	2^{34}
Intel Xeon E5405	CentOS 2.6.18 kernel	different cores	2^{51}

the candidate sets for all the key bytes constitutes the reduced key space that needs to be exhaustively searched for full key extraction.

The last column in Table 4.1 presents the average size of the reduced key spaces obtained at the end of the attacks. We first observed that the secret key was always in the reduced key space. We then observed that in all the experiment setup, the size of the reduced key space was in feasible bounds for an exhaustive search [33].

We, therefore, reached the conclusion that the attack succeeded in extracting the full key, regardless of the underlying hardware and software platforms, and the client-server deployment configurations. This result strongly suggests that the unintentional cache contentions that make the attack a success, are caused primarily internally by the AES server. We reached this conclusion by noting that varying the external entities did not render the attack useless.

4.2.2 Analyzing Internal Entities

We then focused on the implementation of the AES server and applied our approach to identify the code segments in the server source code that can potentially cause the unintentional cache contentions with the AES round tables.

To perform the analysis, we divided the server code depicted in Algorithm 1 into 3 code blocks, namely *RECEIVE*, *ENCRYPT*, and *SEND*, each containing a single source code line. Note that the details of the actual server code are abstracted away in Algorithm 1 for the sake of the discussion. However, in the experiments, we used the actual source code of the Bernstein's AES server.

Algorithm 1 AES server

```
while true do
    $Message\ \ m \leftarrow receive()$
    $Message\ \ c \leftarrow encrypt(m, key)$
    $send(c)$
end while
```

Our first goal was to quantify the extent to which each code block suffers from cache misses. This is important because we use the original AES server code as our control group and the outcomes obtained from it as the basis for comparison.

To this end, we randomly generated $10,000$ messages for encryption, each of which is of size 600 bytes. In the experiments, these messages were always used in the same order for encryption, i.e., in the order they were generated. Furthermore, the first $1,000$ messages were used to warm up the AES server, where no measurements were made. In the remainder of the chapter, these messages are referred to as the *test suite*.

We then executed our test suite on the AES server; the AES client sent every message included in the test suite one after another to the AES server for encryption. All the experiments reported in this section were carried out on an Intel Xeon E5405 platform with 4 GB of RAM, running CentOS with kernel v2.6.18. Furthermore, the server and the client (when present) were executed on two different cores each having its separate L1 data cache.

For each message received at the server, we counted the number of cache misses experienced by each of the code blocks. Figure 4.1 visualizes the data we obtained. In this figure, the horizontal axis denotes the code blocks and the vertical axis denotes the number of L1 data cache misses observed in the blocks. Furthermore, the total cache miss counts are itemized to reflect the misses observed in the OS kernel space and in the user space. Each box in the figure illustrates the distribution of miss counts obtained in the respective scenario. The lower and upper bar of a box represents the first and third quartiles and the horizontal bar inside represents the median value.

We observed that *RECEIVE* and *SEND* code blocks experienced significantly more

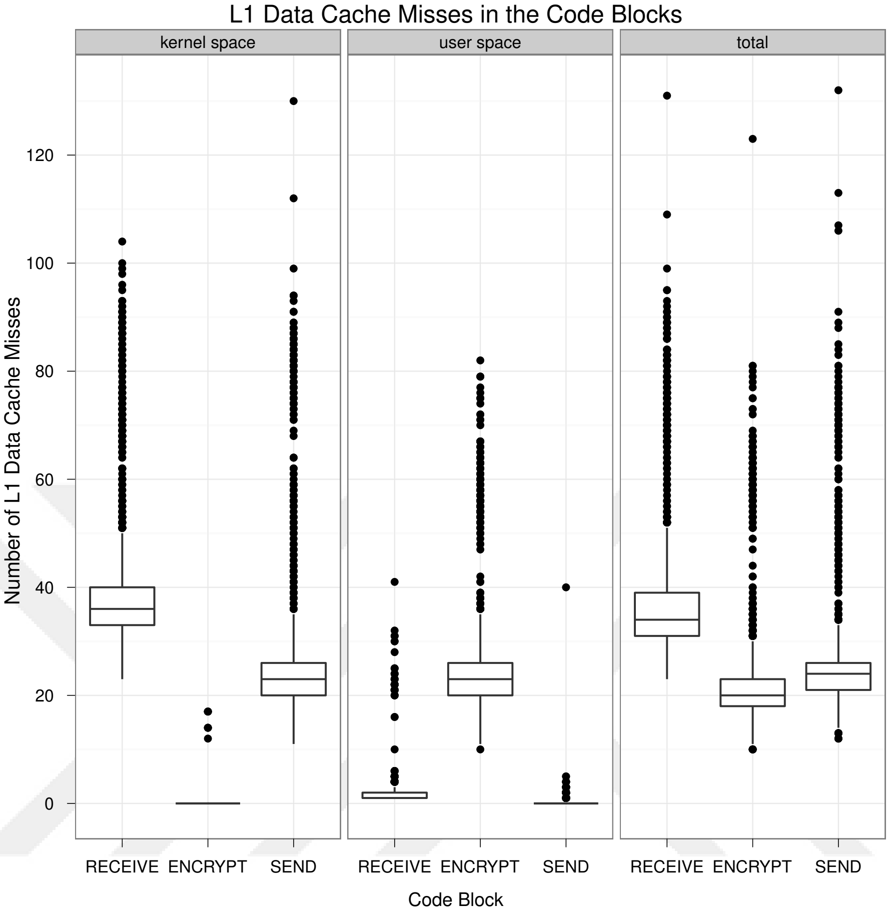


Figure 4.1: L1 data cache misses in the original AES server [33]

cache misses in the kernel space than in the user space, while *ENCRYPT* was experiencing significantly more misses in the user space than in the kernel space. The average number of cache misses in *RECEIVE*, *SEND*, and *ENCRYPT* were about 36, 24, and 0.01 in the kernel space and about 1.7, 0.2, and 24 in the user space, respectively.

As socket communications are mostly handled by the kernel, it is possible for *RECEIVE* and *SEND* to have more cache misses in the kernel. Furthermore, as AES encryption code runs in the user space for the most part, it is possible for *ENCRYPT* to have more cache misses in the user space. However, if the cache misses observed in the kernel space and in the user space are correlated, then it strongly suggests that the kernel is in contention with the (user space) AES encryption code.

To check for correlations, we first replaced the *ENCRYPT* code block in the AES

Algorithm 2 AES server with *MOCK_ENCRYPT*

```
while true do
    $Message\ \ m \leftarrow receive()$
    $Message\ \ c \leftarrow mockEncrypt(m, key)$
    $send(c)$
end while
```

server with a *MOCK_ENCRYPT* block as in Algorithm 2.

Given a message for encryption, *MOCK_ENCRYPT* simply returns the pre-recorded ciphertext of the message without performing any encryption. The rest of the AES server code was not changed. We then executed our test suite with the modified AES server, i.e., the AES client sent the messages in the test suite one at a time to the server for encryption.

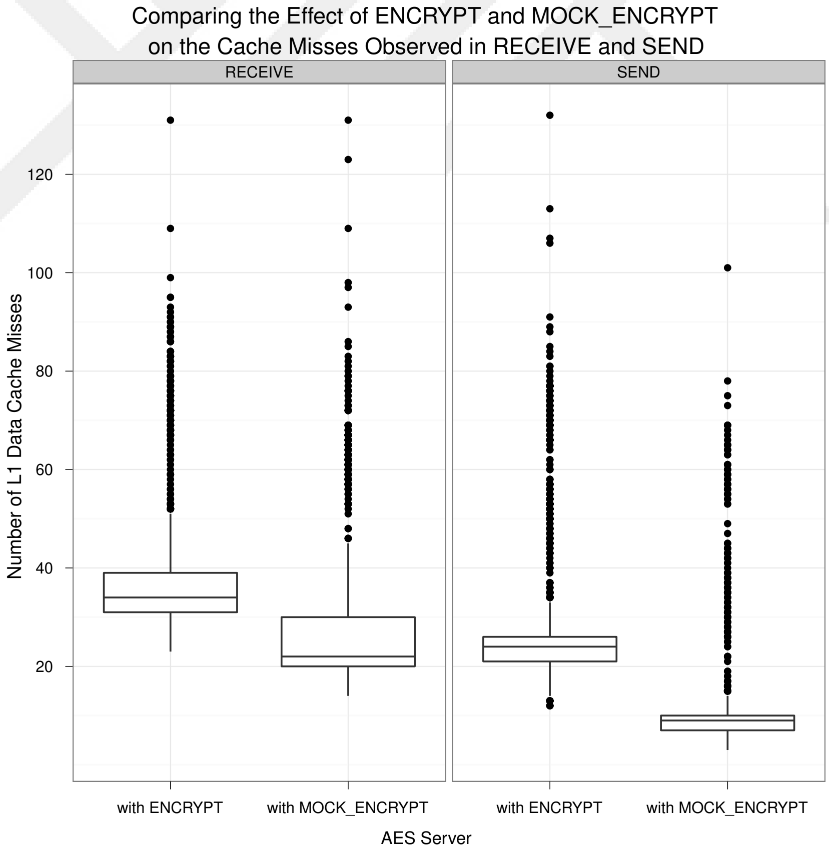


Figure 4.2: Comparing the effect of *ENCRYPT* and *MOCK_ENCRYPT* on the L1 cache misses observed in *RECEIVE* and *SEND* [33]

Figure 4.2 compares the number of cache misses experienced by the *RECEIVE* and

SEND code blocks in the original server and in the modified server. We observed that using *MOCK_ENCRYPT* reduced the number of L1 data cache misses observed in *RECEIVE* and *SEND* by 31% and 63% on average, respectively, compared to that of using the original code block *ENCRYPT*. Kruskal-Wallis test revealed that these differences were statistically significant with a p-value of less than 2.2e-16, suggesting that the AES encryption code is in contention with the kernel.

In an additional experiment, rather than replacing the code block *ENCRYPT* with a mock block, we replaced the *RECEIVE* and *SEND* code blocks with their mock blocks, namely *MOCK_RECEIVE* and *MOCK_SEND*, and kept the original *ENCRYPT* block as in Algorithm 3. *MOCK_RECEIVE* simply replays the pre-recorded messages arriving at the AES server without creating a network socket. Note that the AES client is not used in this setup, thus no messages are exchanged over the network. As there is no client to reply to, *MOCK_SEND* does not perform any operation.

Algorithm 3 AES server with *MOCK_RECEIVE* and *MOCK_SEND*

```
while true do
    $Message\ \ m \leftarrow mockReceive()$
    $Message\ \ c \leftarrow encrypt(m, key)$
    $mockSend(c)$
end while
```

Figure 4.3 depicts the results we obtained. We observed that removing the socket communications reduced the average number of L1 data cache misses observed in the AES encryption code by 83%. The reduction was statistically significant with a p-value of less than 2.2e-16.

To validate our results, we carried out the attack, this time without using any network sockets. To this end, we combined the source code of the AES client and the AES server in a single stand-alone application and removed all the code segments responsible for socket communications. That is, the attack code directly called the AES encryption function, instead of sending and receiving packages over the network. As is the case with the experiments reported in Section 4.2.1, we used 2^{30} randomly generated messages of 600

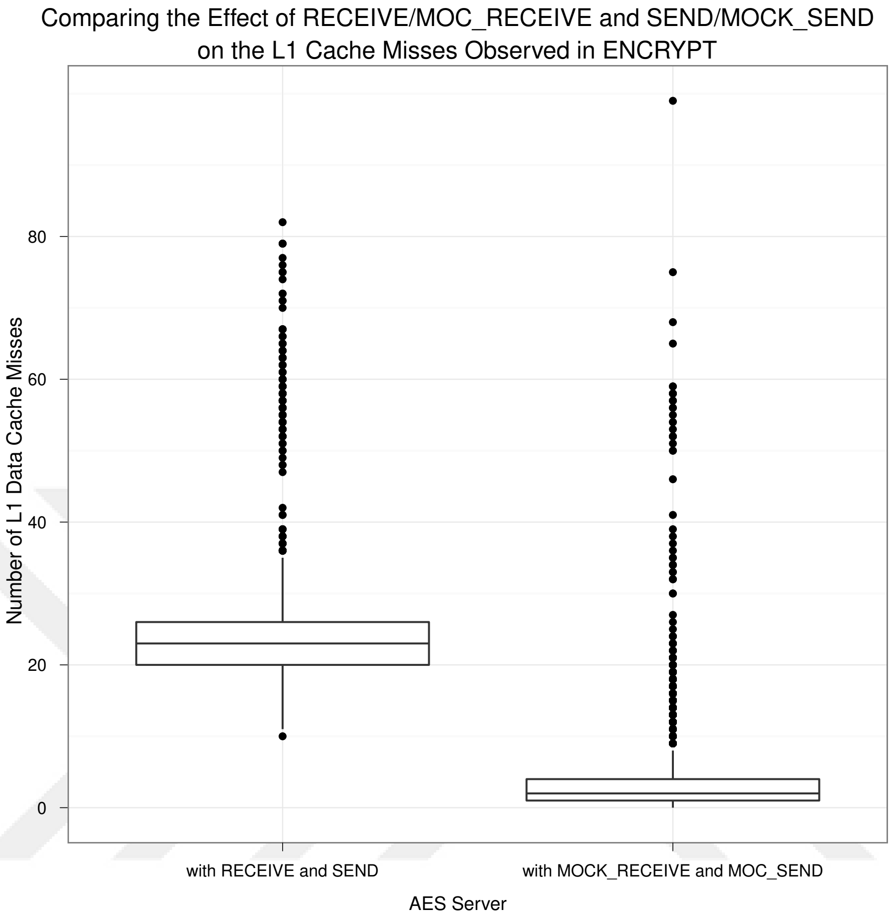


Figure 4.3: Comparing the effect of *RECEIVE/MOC_RECEIVE* and *SEND/MOCK_SEND* on the L1 cache misses observed in *ENCRYPT* [33]

bytes each in the attack. Furthermore, the attack was repeated 5 times.

At the end of the attacks, the size of the reduced key space was 2^{107} on average. As it is not feasible to exhaustively search this space, the attack failed to extract the secret key in the experiment setup used in the study. For comparison, in the same experiment setup and with the socket communications are in place, the size of the reduced key space was 2^{51} in the worst case (Table 4.1).

We also observe that, although the attack failed when the socket communications were removed from the source code, the attack was still be able to reduce the key space from 2^{128} to 2^{107}. This can point to the presence of other unintentional cache contentions, which requires further analysis.

With all these in mind, the results of our experiments strongly suggest that the part of the OS kernel that handles socket communications is in cache contention with the AES encryption code and that these contentions are the primary source of information leakage, making the Bernstein's attack a success.

4.3 Discussion

In this chapter, we used the proposed approach to determine code segments that are in cache contention with each other. All empirical studies suffer from threats to their internal and external validity. One potential threat is that we have only studied one cryptographic system, namely the Bernstein's AES server with the OpenSSL implementation of AES. This may impact the generality of our results. However, this cryptographic system has been extensively used in other related works in the literature [11,16,22,31,46–55], which emphasizes the importance and impact of the work. A related issue is that the code blocks used in the experiments were relatively simple code blocks with well-defined code and functionality boundaries. While these issues pose no theoretical problems, we need to apply our approach to larger, more realistic scenarios in future work to understand how well it scales.

Moreover, the proposed approach can readily be applicable to determine contentions between code segments on any types of hardware and software resources as long as the contentions can be quantified. To this end, we believe that using hardware performance counters is a perfect match as they enable us to quantify contentions on almost all hardware resources, such as CPUs, CPU pipelines, instruction and data caches, branch prediction/target buffers, TLBs, RAMs, memory buses, and I/O buses. Aciicmez et al., for instance, describe micro-architectural side-channel attacks that exploit contentions on instruction cache and branch prediction/target buffers [12,15]. Both attacks rely on the fact that contentions on these resources, increase the execution time of cryptographic applications. Although the contentions leveraged in these attacks are intentionally introduced by a spy process, unintentional contentions on these micro-architectural resources can also be exploited in a similar way. We conjecture that the proposed approach can be used to isolate such unintentional contentions on instruction cache and branch prediction/target buffers, thus further improve the reliability of cryptographic applications.

Another interesting avenue for future work is to fully automate the proposed approach. In the experiments, although the mock code blocks were created automatically, we manually decided on the combinations of mock and actual code blocks to be tested. One challenge towards fully automating the proposed approach is to automatically select the combinations in such a way that the choice of combinations does not introduce any bias in the analysis.

A related concern is how best to divide the code base into code blocks. As this division is application-specific, one way to address this concern is to provide developers with tools that enable them to annotate the code blocks in programs' source code. Another way is to automatically perform the division at a given granularity level, such as at the level of functions, classes, and libraries, as is the case in performance profiling tools.

4.4 Chapter Summary

Many modern CPUs contain hardware performance counters (HPCs) in their architecture. These HPCs allow us to count many specific run-time events such as, the number of instructions executed, the number of branches taken, the number of cache hits and misses. Even though these counters are mainly purposed to be used in performance profiling, they can be also used to conduct side channel attacks [24]. In this chapter, we presented a novel usage of HPCs that allows us to detect the source of information leakage. The proposed approach is able to detect code segments in cryptographic applications that are in cache contentions with each other, thus leaking information that can be exploited in cache-based side-channel attacks to extract secret keys.

In order to evaluate the approach, we conducted a series of experiments by using the well-known Bernstein's attack. The results of these experiments helped in pinpointing the primary source of the exploitable side channel in the Bernstein's AES server. To be more specific, to the best of our knowledge, we demonstrated for the first time, that the part of the OS kernel that handles socket communications is in cache contention with the AES encryption code and that these contentions are the primary source of information leakage, making the Bernstein's attack a success [33]. In Section 4.2.2, to further validate our results, we removed the socket communication from the code and observed that the attack is not feasible anymore.

It is inevitable for a cryptographic processes not to share or to share hardware and software resources of the underlying platform (i.e. personal computers, workstations, and servers), on which they run. It is also known that the shared resources may cause an information leakage which enables side-channel attacks. Thus, the way the cryptographic processes are written requires utmost care and attention to prevent such attacks. We believe that, the proposed approach can help developers to write more secure programs by providing them valuable analysis data regarding the possible information leakage sources.

Chapter 5

REMOVING the PROFILING PHASE in CACHE-TIMING ATTACKS

In the case of cache attacks, as mentioned in Chapter 2, a profiling phase and/or a spy process is essential in order to conduct a successful attack. The strength of Bernstein's timing attack comes from the fact that it does not require a spy process. As we have shown in Chapter 4, it depends on the naturally occurring cache contentions between the code blocks. Albeit it does not need a spy process, a major drawback of Bernstein's attack is the necessity of having a computer system which 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. Exact replication of the target system, with running processes on it, with the same input-output events and all its machine specific cache effects can be very difficult, which causes the attack to be considered unrealistic in many contexts [16, 21].

In this chapter we propose a methodology, based on hypothetical modeling of the cache timing-behavior of a computer system and demonstrate that the Bernstein's attack successfully recovers the key using one of the models that best represents its cache timing-behavior. In our hypothetical modeling 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 phase (i.e. the need for an identical target machine), which makes the attack more realistic and feasible in practice.

The rest of the chapter is organized as follows: Section 5.1 discusses the details of the profiling phase; Section 5.2 outlines our proposed approach; Section 5.3 explains how we conduct the Bernstein's attack without the profiling phase in order to validate our hypothetical modeling methodology; Section 5.4 discusses the importance and validity of the results in the context of recent studies; Section 5.5 concludes the chapter.

5.1 Analysis of the Profiling Phase

In Chapter 3, we outline the cache-timing attack of Bernstein briefly and then explain a modified version of the attack which focuses on the final round, instead of the first round of AES. Both attacks need a profiling phase. It is now crucial to understand what we achieve after the profiling phase is successfully applied. In [11] and in Chapter 4 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 profiling phase helps to obtain data cache timing-behavior of AES process by registering the variations in cache line access times. In this section we will further investigate the profiling phase of the last round attack.

Cache timing-behavior of an 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 profiling phase, the timing model for the i^{th} access in the last round is simply a histogram of the deviation of average execution times of an AES encryption indexed by output bytes of $T4$ as computed in Equation 3.1. Figure 5.1 and Figure 5.2 shows such a cache timing-behavior model where the figures depict the models of bytes 0-7 and bytes 8-15 respectively.

The profiling phase data used in the Figures 5.1 and 5.2 are collected from a computing platform with Intel Pentium P6200 CPU running Ubuntu 3.0.0-12 kernel. In these figures, 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 $T4$.

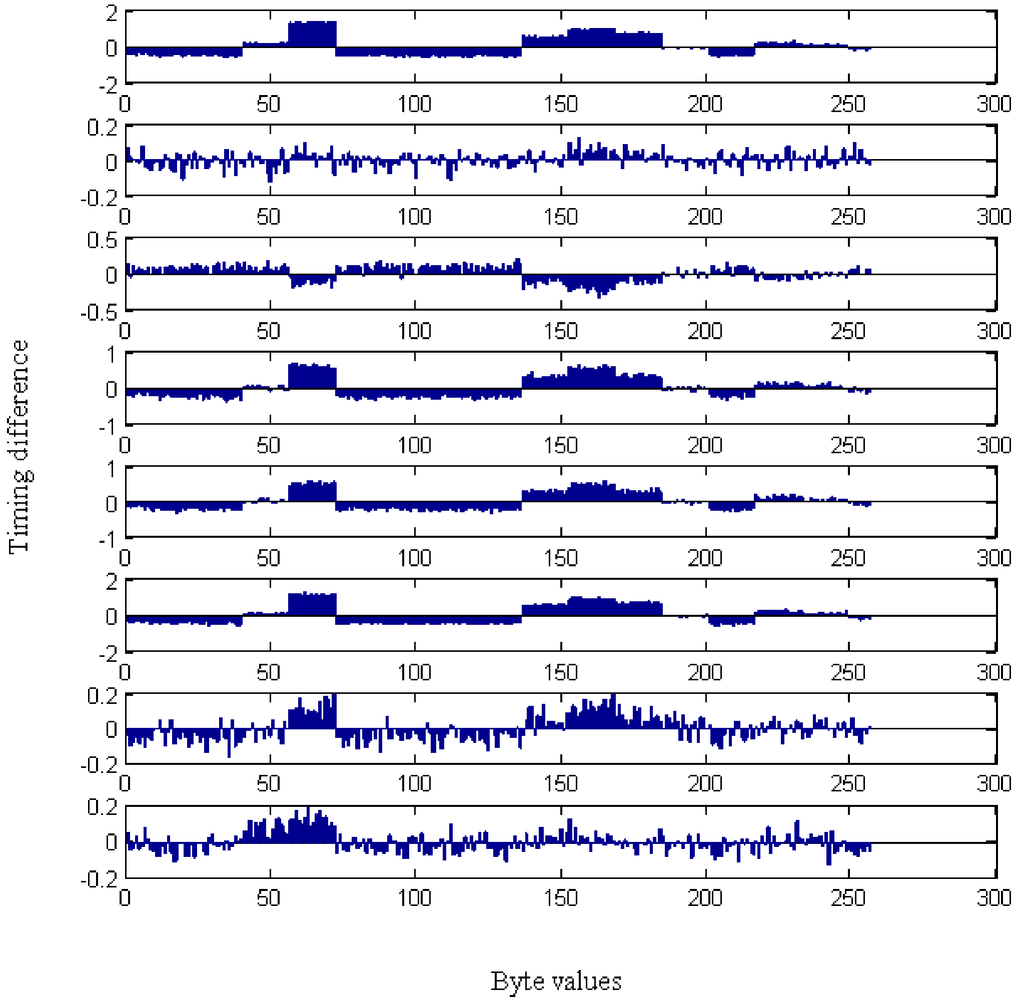


Figure 5.1: Example profiling phase cache timing-behavior, bytes 0-7

In Figures 5.1 and 5.2, the mean value is the average execution time of the all encryption operations. One can notice that the timing measurements are either above or below the average execution time. Here, the measurements above the average are attributed to cache misses in the corresponding cache lines due to the extra time required to fetch the missing data from the main memory. Similarly, the measurements below the average are attributed to cache hits in the corresponding cache lines.

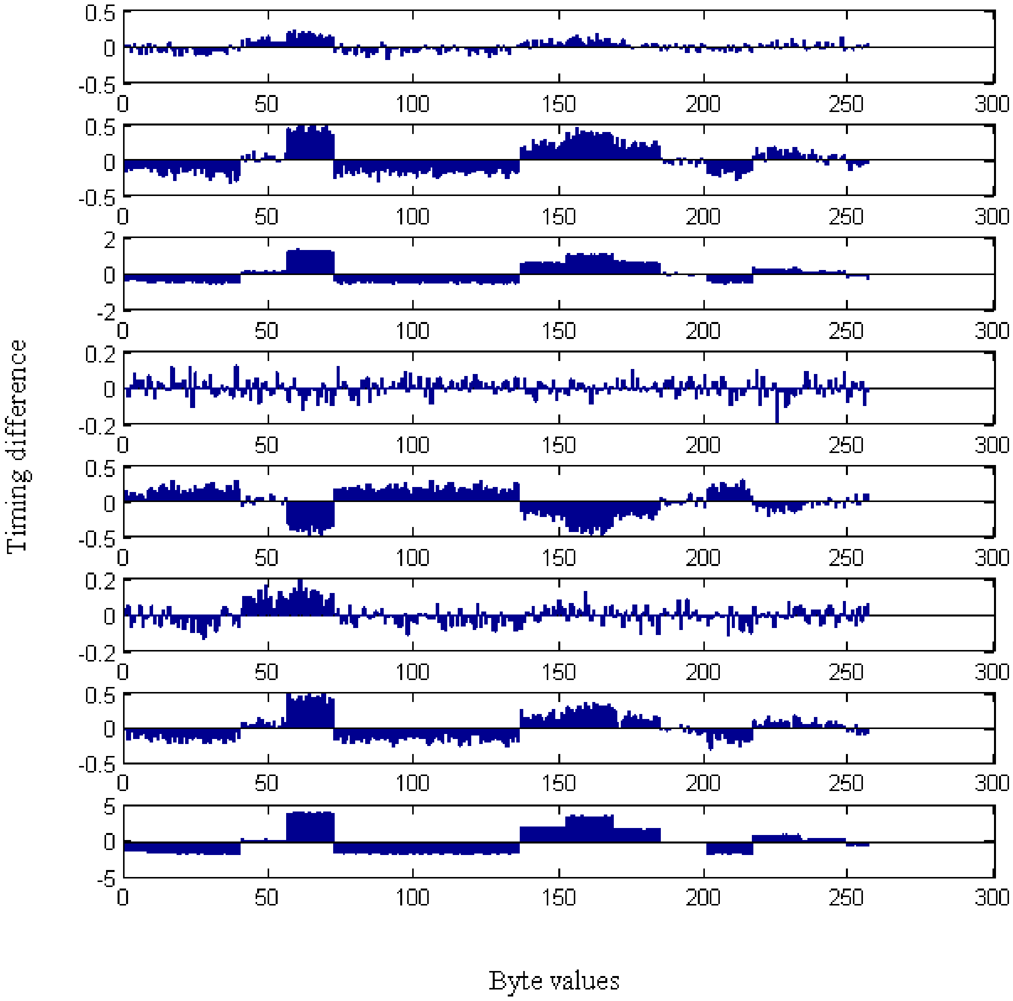
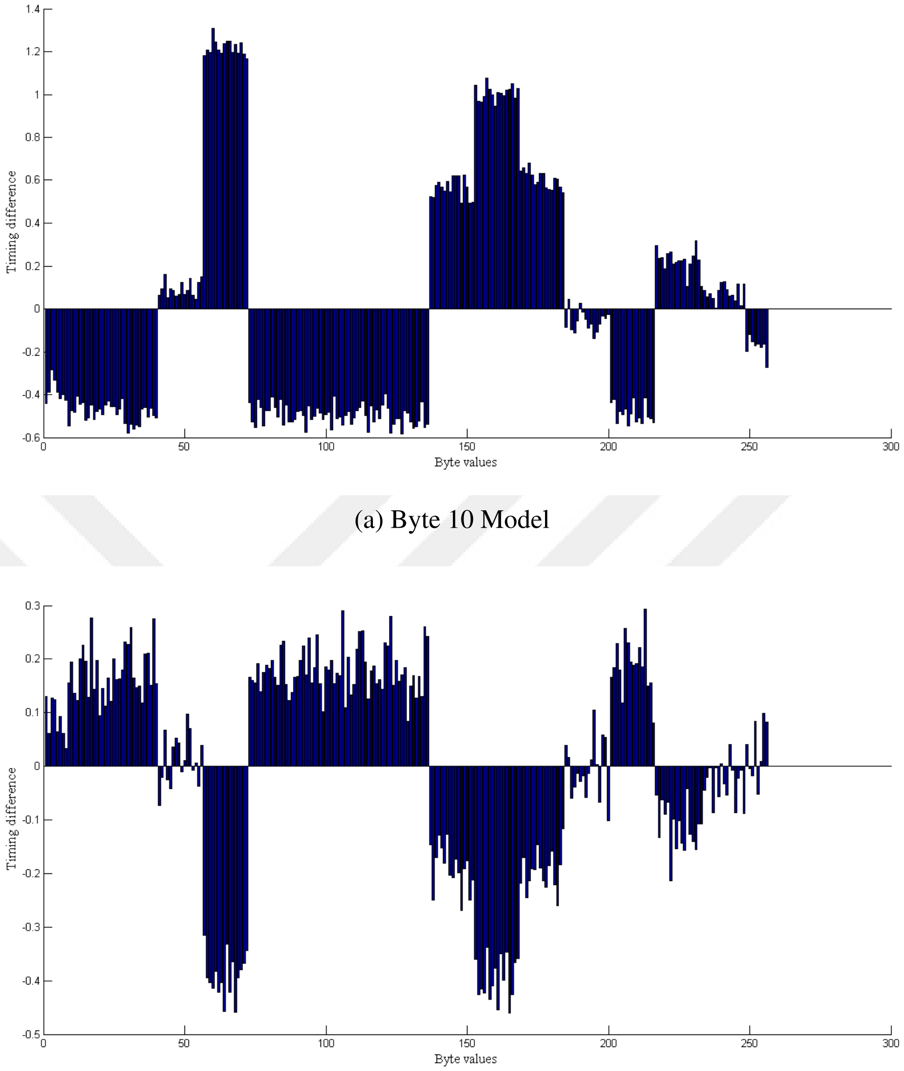


Figure 5.2: Example profiling phase cache timing-behavior, bytes 8-15

Furthermore, if all the 16 byte models are examined, it is seen that the execution times tend to remain above or below the average line for a group of consecutive index values. This particular pattern can be observed in Figure 5.3 in more detail.

Figure 5.3 just zooms in the models of byte 10 and byte 12 in the Figure 5.2. The observed pattern is 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 due to the principle of locality.

In Figure 5.3, 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



(a) Byte 10 Model

(b) Byte 12 Model

Figure 5.3: Example profiling phase byte 10 and byte 12 model

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. An important point to note here is that the indexes that experience collisions does not change between different bytes (i.e. always the same indexes cause misses or hits). This observation suggests that the cache lines which are in contention does not change throughout the execution of the program. Cache conflicts are consistently occur on the same group of cache lines.

To summarize, at the end of the profiling phase, we obtain a timing model $\mathcal{T}_i^p$ for the i^{th} access in the last round, which is just an array of 256 values where each value represents the deviation from the average of the total 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 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 values where each value represents the deviation. Then, the two timing models, namely $\mathcal{T}_i^p$ and $\tilde{\mathcal{T}}_i^a$ are correlated. As $\mathcal{T}_i^p$ 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 $\mathcal{T}_i^p$ and $\mathcal{T}_{i,k}^a$, 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 4, where $\mathcal{T}^p = \cup_{i=0}^{15} \mathcal{T}_i^p$ and $\mathcal{T}^a = \cup_{i=0}^{15} \tilde{\mathcal{T}}_i^a$ are the sets of timing models in the profiling and attack phases, respectively.

Algorithm 4 Attack with profiling phase

```
Require: $\mathcal{T}^p$ and $\mathcal{T}^a$
Ensure: $\mathcal{K}_R$: Ordered reduced key space
$\mathcal{K} \leftarrow \emptyset$
$\mathcal{K}_R \leftarrow \emptyset$
for $i = 0$ to $15$ do
    for $k = 0$ to $255$ do
        $\mathcal{T}_{i,k}^a \leftarrow$ Transform($\tilde{\mathcal{T}}_i^a, k$)
        $\gamma \leftarrow$ Correlate($\mathcal{T}_{i,k}^a, \mathcal{T}_i^p$)
        $\nu \leftarrow$ Variance($\mathcal{T}_{i,k}^a, \mathcal{T}_i^p$)
        $\mathcal{K}[i] \leftarrow \mathcal{K}[i] \cup (k, \gamma, \nu)$
    end for
    $\mathcal{K}[i] \leftarrow$ Sort($\mathcal{K}[i]$)            ▷ Descending on $\gamma$
    $\delta \leftarrow$ Threshold($\mathcal{K}[i]$)
    $\mathcal{K}_R[i] \leftarrow$ Reduce($\mathcal{K}[i], \delta$)
end for
```

The last round attack can reveal the entire key, although it still needs a profiling phase. We already mentioned that it is not an easy task for an attacker to setup an identical platform and to run the profiling phase. To increase the feasibility and applicability of the attack, we present a novel methodology which 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. The details are provided in the following sections.

5.2 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. Highly complex and optimized cache implementations and lack of details thereof, render an accurate modeling of cache timing-behavior an involved task. The used models in the cache-timing attack are, in fact, based on the time variations in the access times of cache lines/sets. Indeed, every memory access results in accessing a cache line or set whether it is a cache hit or miss. Therefore, 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 successful 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 3 ***Data*** *in the data cache of a CPU are composed 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 ***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 a cache placement strategy, where a single cache line can be considered as a cache set. 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 Assumption 1.*

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 $(t + \Delta)$, respectively, and we always have $\Delta > 0$.*

Assumption 5 *Cache collisions may occur between two 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 *A program's execution time varies with its input depending on the cache hits and misses occurred during its execution. The execution times of a program in the presence of hits and misses are t_h and t_m, respectively. And t_h and 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*

$$n = \left\lceil \frac{|data|}{b} \right\rceil, \tag{5.1}$$

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 t_h, t_m and t_a of a program with*

$$t_h = t + t_f, \tag{5.2}$$

$$t_m = (t + \Delta) + t_f, \tag{5.3}$$

$$t_a = (t_h + t_m)/2, \tag{5.4}$$

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 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:*

$$\mathcal{I}_c = \left[\frac{c_1 \cdot b}{\kappa}, \frac{c_2 \cdot b}{\kappa} + \frac{b}{\kappa} - 1 \right]. \tag{5.5}$$

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 5 describes the steps to obtain a model for a given **data**.

Algorithm 5 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($\mathcal{S}_c, b, \kappa$)        ▷ ($Proposition$ 3)
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
```

In Algorithm 5, 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 $\mathcal{S}_c$ 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 5 calculates the set of **data** indexes which results in cache misses and Line 3 of Algorithm 5 checks whether a data index is in set $\mathcal{I}_c$. 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.

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 Preposition 3, we can find which **data** indexes will cause a cache miss. If we use Algorithm 5, our model will have a total of* 40 *indexes and the indexes in ranges* $[8, 15]$ *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 5.4.*

When Figures 5.3 and 5.4 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

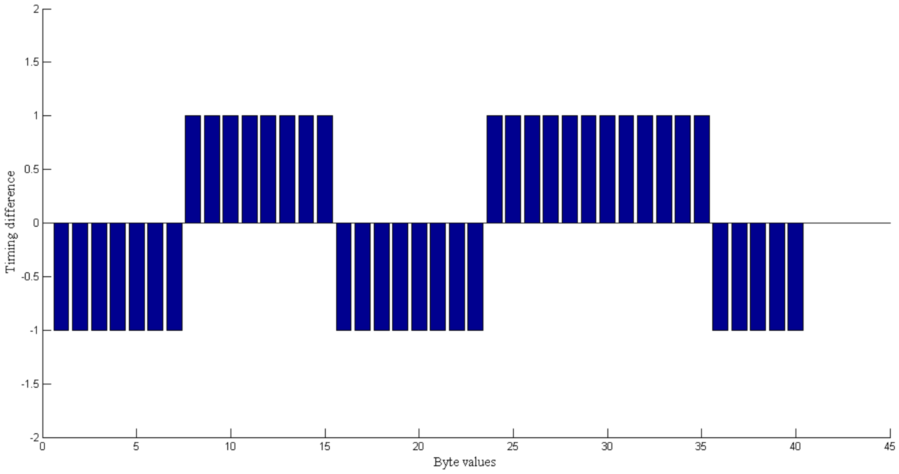


Figure 5.4: Example cache timing-behavior model

timing models obtained in the profiling phase of the Bernstein's attack, table $T4$ is **data** and lookup bytes are the indexes as defined by the terminology introduced in Section 5.2. The measured timing values in the profiling phase of the Bernstein's attack are noisy and obtained by averaging excessively many AES execution times. All the same, the simplified timing model captures essentially the same behavior.

5.3 Practical Application of The Proposed Methodology and Validation Results

In this section, we give a formal description of the non-profiled Bernstein attack, which we use to validate our hypothetical modeling methodology and present our experimental results.

5.3.1 Cache Timing Attacks without a Profiling Phase

In order for an attacker to model the cache timing-behavior of the server in the profiling phase, the attacker must produce an identical system the cache timing-behavior of which must exactly be the same as the target computer. Exact replication of the target system, with running processes on it, with the same input output events and all its machine specific cache effects can be very difficult, if not impossible. This is a major drawback in the Bernstein's original cache-timing attack, which is also mentioned in [21] and [16]. Using hypothetical modeling as suggested here, however, eliminates the need for an identical system, hence the profiling phase. The attack without the profiling 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** 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. 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 $[0, 2^{16} - 1]$ 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 the bit value of 0 indicates a collision in the corresponding cache line. Algorithm 6 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 6 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 $l$
$\mathcal{S}_C \leftarrow \emptyset$
for $i = n - 1$ to $0$ do
    if ($l \bmod 2 == 0$) then
        $\mathcal{S}_C \leftarrow \mathcal{S}_C \cup i$
    end if
    $l \leftarrow l/2$
end for
```

Finally, Algorithm 7 gives us the most possible cache timing-behavior model given the timing measurement data from the attack phase (i.e. $\mathcal{T}^a$).

Algorithm 7 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 4 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). As the Bernstein attack tends to calculate higher correlation values for the simplified models which are closer to the measurements in the attack phase, and higher correlations yield smaller reduced key space sizes, we propose to select the model with the smallest reduced key space. Once we obtain the model, we can run Algorithm 4 and find the key bytes.

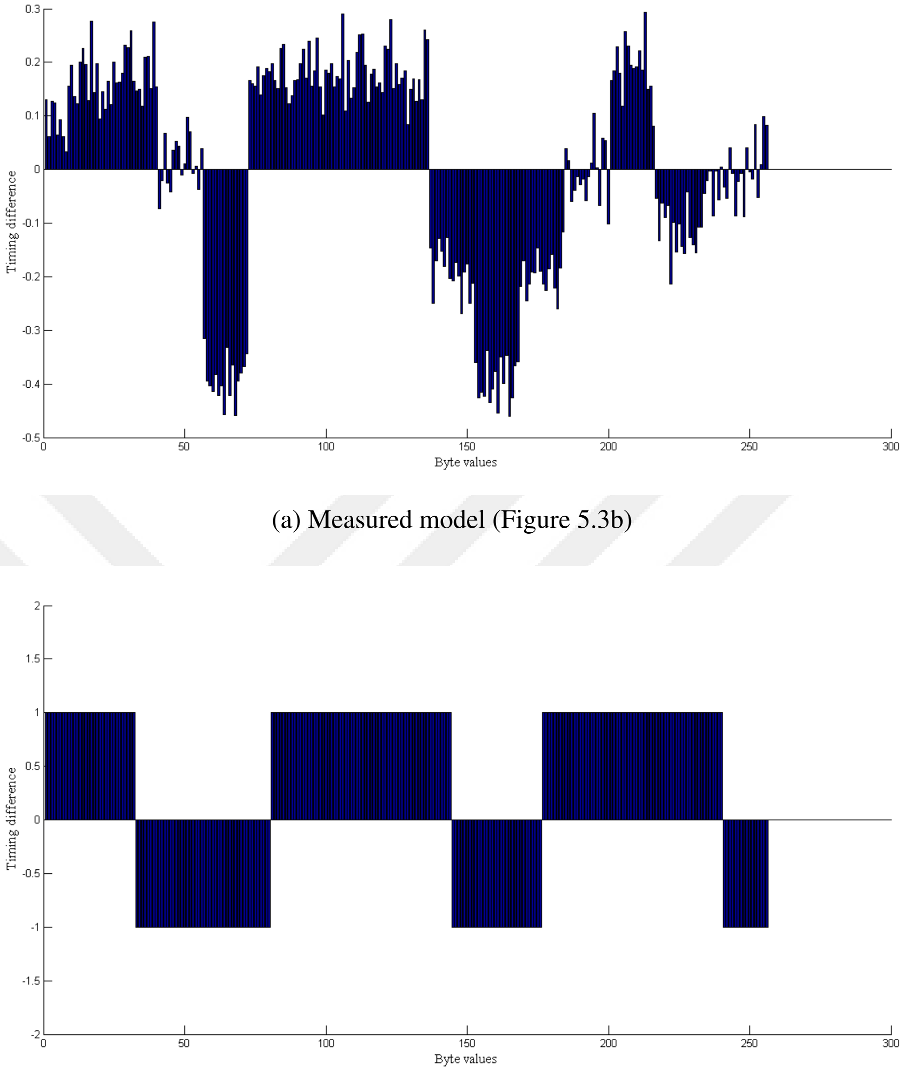
In order to test Algorithm 7, we ran it for the example data set in Figures 5.1, 5.2 and 5.3 and obtained the index 14433 as the most probable cache timing model. When we plot this model, we obtain Figure 5.5. A closer look at Figure 5.5 reveals that our simplified model (Figure 5.5b) resembles to the real model (Figure 5.5a) previously depicted in

Algorithm 7 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
$\mathcal{M} \leftarrow \emptyset$
for $l = 0$ to $2^n - 1$ do
    $\mathcal{S}_C \leftarrow$ Algorithm 6$(l, n)$
    for $j = 0$ to $15$ do
        $\mathcal{T}_l^h[j][:] \leftarrow$ Algorithm 5$(T4, m, \mathcal{S}_C, b, \kappa)$
        for $i = 0$ to $255$ do
            $x \leftarrow$ AES-sbox$(i)$
            $\mathcal{T}_{tmp}^h[j][x] \leftarrow \mathcal{T}_l^h[j][i]$
        end for
        $\mathcal{T}_l^h[j][:] \leftarrow \mathcal{T}_{tmp}^h[j][:]$
    end for
    $\mathcal{K}_R \leftarrow$ Algorithm 4$(\mathcal{T}_l^h, \mathcal{T}^a)$
    $\mathcal{M} \leftarrow \mathcal{M} \cup (\mathcal{T}_l^h, |\mathcal{K}_R|)$
end for
$\mathcal{M} \leftarrow$ Sort$(\mathcal{M})$                              ▷ Ascending on $|\mathcal{K}_R|$
$\tilde{\mathcal{T}}^h \leftarrow \mathcal{M}[0][0]$
```

Figure 5.3b.

An important point to note here is that, in Algorithm 7, we assume that all the lookup index bytes will have the same timing model. Actually this is not always the case. In Section 5.1 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 5.3. Thus, for the indexes used in lookup operations where the real model is the symmetric of our simplified



(a) Measured model (Figure 5.3b)

(b) Calculated model

Figure 5.5: Calculated profiling phase model vs. measured profiling phase model

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 4. In Algorithm 4, 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, the correct key byte will appear in the first ranks in the sorted list.

5.3.2 Experimental Results

We used the same data set used in Chapter 4 for our experiments. It is important to remind that, 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 his attack, to encrypt the incoming messages. In the attacks with a profiling phase, two separate measurements are used (i.e. profiling and attack phase measurements) 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, which are sorted by their likelihood. By multiplying the sizes of the candidate sets we obtain the reduced key space size for the whole AES key, which necessitated exhaustive search.

For each attack setup, we conducted a number of attacks and calculated the average of the results. In the Table 5.1 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 three columns of the Table 5.1, the attack configurations (i.e. the CPU type, operating system and the client-server deployment setup) are given. The fourth and fifth columns present the average sizes of the reduced key spaces obtained after the last round attack with a profiling phase and without a profiling phase, respectively.

Table 5.1: Attack results without profiling phase

Processor	Operating System	AES Client-Server Deployment Configuration	Reduced Key Space with a Profiling Phase	Reduced Key Space without a Profiling 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}

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 and by other programs sharing the same cache lines, when the two 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. Our results show that performance of the proposed attack varies in the experiment setups (i.e. it increases, decreases and stays same in comparison with the attack with profiling phase). We should note that no correlations are observed between these test setups. The performance of the attack depends solely on the configuration of underlying software and hardware platform during execution of the 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 separate CPUs. Since both programs run on the same PC, we minimize the effects of network delay on the measurements. In the classical attack (i.e. using both profiling and attack phases) we reduced the key space to 2^{67} from 2^{128} with 2^{34} measurements, while the non-profiled attack reduced the key space to 2^{60} with 2^{34} measurements. These results demonstrate that our methodology can also be used 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.

5.4 Discussion

In this chapter we experimentally showed 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 [33]. The ASLR technique randomizes the address space of the executables, stack, heap, and the libraries. Since the original Bernstein's attack needs two 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 two 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 [46, 47], the authors implement Bernstein's attack on ARM CPU and, in [31, 48–50], 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 [46, 47], 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 [31, 48–50], 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 [51, 52], Gulmezoglu et al. conduct successful flush+reload and prime+probe cache attacks on cross-virtual machine (VM) environments. In [55], 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 [53, 54] that it is possible to perform flush+reload and prime+probe cache attacks on virtualized environments. In [56], Weiß 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 [57], the stalling delay caused by cache bank conflicts is exploited to infer the secret key. In [58], Moghimi et al. conduct a cache attack on a Software Guard eXtensions (SGX) [59] supported Intel platform with different AES implementations. In [60], the authors provide improvements over existing cache attacks and provide experimental results. In [61], the authors show a cross processor cache attack that targets high efficiency CPU interconnects. Moreover, in [23, 62], 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.

5.5 Chapter Summary

Bernstein's cache-timing attack has been widely studied in the literature. The reason behind its attraction is that the attack does not need a spy process like the other cache-timing attacks. However, the attack still has a major drawback: It needs a profiling phase where an identical target system is a necessity. In this chapter, we provide a solution to this problem. We present a methodology to extract a simplified model of the cache timing-behavior of a computer, which eliminates the need for a profiling phase [35].

In the presented methodology, 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. The simplified model can be extracted with two easily obtainable information: Cache-line size and the size of the sensitive data that causes cache conflicts.

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 non-profiled 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.

Chapter 6

CONCLUSIONS and FUTURE WORK

Throughout the history, sharing of sensitive information securely has been of great importance for human beings, governments, armies and corporations, etc. People developed a multitude of methods to hide the secret information from their adversaries. These efforts gave rise to the science of cryptography. Many mathematical algorithms were developed to encrypt sensitive information. And these algorithms were mathematically analyzed to be proven secure. However, side channel analysis changed the game. It showed that mathematical robustness is not enough. To be truly secure, cryptographic algorithms should also be implemented securely. Bernstein's cache-timing attack is one of the most studied side channel attacks, which uses the timing information as a side channel. Bernstein's attack drew a great amount of attention, due to the fact that it does not need a spy process to create cache conflicts. The attack works thanks to some internal cache conflicts within a program. There are studies which showed that the L1 data cache conflicts allow the Bernstein's attack to be successful, even though there has not been a study in the literature to provide an in-depth analysis to find out the main source of the information leakage. Moreover, the attack had one drawback, that it needs an identical target machine to perform its profiling phase. This requirement caused the attack to be considered unrealistic. In this dissertation we provide solutions to these problems.

To address the former problem, we propose to use hardware performance counters to quantify the cache misses as a result of cache conflicts within the code blocks in a program. The proposed approach divides the program into code blocks. Just before the beginning and right after the end of each code block, L1 data cache miss counter is read. The difference of the two values gives the number of cache misses that the code block observed. Then, a code block (e.g. B) is replaced with its mockup counterpart (e.g. $B^{'}$) to see the code block's effect on the other code blocks. If the other code blocks observe fewer number of misses when $B^{'}$ is in the program, then we deduce that B causes cache conflicts on the other code blocks. By using our proposed approach, we showed that the socket communication code blocks cause cache conflicts on the AES encryption code block. In order to further validate our approach, we removed the socket communication from the program and conducted Bernstein's attack. In the end, we could not get any meaningful results from the attack, which is due to the removal of the contention sources. We believe that the proposed approach can be used by the developers to produce more robust implementations for the cryptographic algorithms.

Secondly, we propose a methodology to eliminate the requirement of the profiling phase in Bernstein's attack. In our methodology we propose to use simplified cache timing-behavior models instead of a profiling phase. The simplified models can be constructed using few generic and easily obtainable information about the target system such as cache line size. Furthermore, we used the simplified models in order to conduct the Bernstein's attack without a profiling phase. The improved attack, exhaustively tries all the possible simplified models and choses the best one to use in the attack. Experimental results showed that the attack without the profiling phase has comparable performance to the original attack with the profiling phase. By removing the profiling phase of the Bernstein's attack, we are able to demonstrate that the attack can be applied in a more realistic setting increasing its importance.

6.1 Future Directions

The experiments which we performed for isolating the leakage sources were conducted on relatively simple code blocks. This work can be extended to more complex programs to evaluate the approach on real life scenarios. Moreover, automating the code block selection at a predefined granularity level would be an interesting topic to study. We believe that the proposed methodology is a valuable tool to detect code blocks which are in contention with cryptographic code blocks handling secret information. It, therefore, can be used as a part of a secure software generation toolset in order to analyze the critical code blocks against possible information leaks.

The improved cache-timing attack was experimented on general purpose CPUs. As future study, this work can be extended on embedded platforms as well as multi-tenant cloud computing environments. Moreover, evaluating the applicability of the improved attack on other table-based cryptographic algorithms would be an other interesting topic to study. Our improvements on the Bernstein's attack made it easily applicable for real life systems. Thus, apart from attacking cryptographic systems, the improved attack can be used to investigate possible vulnerabilities of cryptographic systems. And depending on the assessment results, administrators of such systems can take required actions.

BIBLIOGRAPHY

[1] 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*, CRYPTO ’96, (London, UK), pp. 104–113, Springer-Verlag, 1996.

[2] P. C. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” in *Advances in Cryptology - CRYPTO ’99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pp. 388–397, 1999.

[3] K. Gandolfi, C. Mourtel, and F. Olivier, “Electromagnetic analysis: Concrete results,” in *Cryptographic Hardware and Embedded Systems — CHES 2001* (Ç. K. Koç, D. Naccache, and C. Paar, eds.), (Berlin, Heidelberg), pp. 251–261, Springer Berlin Heidelberg, 2001.

[4] O. Aciiçmez, Ç. K. Koç, and J. Seifert, “Predicting secret keys via branch prediction,” in *Topics in Cryptology - CT-RSA 2007, The Cryptographers’ Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings*, pp. 225–242, 2007.

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

[6] 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*, CCSW ’12, (New York, NY, USA), pp. 19–24, ACM, 2012.

[7] 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.

[8] M. Chiappetta, E. Savas, and C. Yilmaz, “Real time detection of cache-based side-channel attacks using hardware performance counters,” *Appl. Soft Comput.*, vol. 49, pp. 1162–1174, Dec. 2016.

[9] D. J. Bernstein, “Cache Timing Attacks on AES.” http://cr.yp.to/antiforgery /cachetiming-20050414.pdf, 2005. Last accessed on October, 2018.

[10] "AES Standard." http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001. Last accessed on October, 2018.

[11] M. Neve, *Cache-based Vulnerabilities and SPAM analysis*. PhD thesis, Universite catholique de Louvain, 2006.

[12] O. Aciiçmez, S. Gueron, and J. Seifert, "New branch prediction vulnerabilities in openssl and necessary software countermeasures," in *Cryptography and Coding, 11th IMA International Conference, Cirencester, UK, December 18-20, 2007, Proceedings*, pp. 185–203, 2007.

[13] O. Aciiçmez, "Yet another microarchitectural attack: : exploiting i-cache," in *Proceedings of the 2007 ACM workshop on Computer Security Architecture, CSAW 2007, Fairfax, VA, USA, November 2, 2007*, pp. 11–18, 2007.

[14] O. Aciiçmez and W. Schindler, "A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on openssl," in *Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings*, pp. 256–273, 2008.

[15] O. Aciiçmez, B. B. Brumley, and P. Grabher, "New results on instruction cache attacks," in *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, pp. 110–124, 2010.

[16] 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.

[17] K. Tiri, O. Aciiçmez, M. Neve, and F. Andersen, "An analytical model for time-driven cache attacks," in *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, pp. 399–413, 2007.

[18] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeria, and H. Miyauchi, "Cryptanalysis of DES Implemented on Computers with Cache," in *CHES 2003 LNCS* (C.D. Walter et al., ed.), vol. 2279, pp. 62–76, 2003.

[19] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo, "AES power attack based on induced cache miss and countermeasure," in *International Symposium on Information Technology: Coding and Computing (ITCC 2005), Volume 1, 4-6 April 2005, Las Vegas, Nevada, USA*, pp. 586–591, 2005.

[20] O. Aciiçmez and Ç. K. Koç, "Trace-driven cache attacks on AES (short paper)," in *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, pp. 112–121, 2006.

[21] J. Bonneau and I. Mironov, "Cache-Collison Timing Attacks Against AES," in *CHES 2006 LNCS* (L. Goubuin and M. Matsui, ed.), vol. 4249, pp. 201–215, 2006.

[22] O. Aciiçmez, W. Schindler, and Ç. K. Koç, "Cache based remote timing attack on the AES," in *Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings*, pp. 271–286, 2007.

[23] C. Rebeiro, M. Mondal, and D. Mukhopadhyay, "Pinpointing cache timing attacks on aes," in *VLSI Design, 2010. VLSID '10. 23rd International Conference on*, pp. 306–311, January 2010.

[24] L. Uhsadel, A. Georges, and I. Verbauwhede, "Exploiting hardware performance counters," in *Fault Diagnosis and Tolerance in Cryptography, 2008. FDTC '08. 5th Workshop on*, pp. 59–67, August 2008.

[25] 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, 2000.

[26] B. Schneier, "Description of a new variable-length key, 64-bit block cipher (blowfish)," in *Fast Software Encryption* (R. Anderson, ed.), (Berlin, Heidelberg), pp. 191–204, Springer Berlin Heidelberg, 1994.

[27] P. Rogaway and D. Coppersmith, "A software-optimized encryption algorithm," in *Fast Software Encryption* (R. Anderson, ed.), (Berlin, Heidelberg), pp. 56–63, Springer Berlin Heidelberg, 1993.

[28] "Data Encryption Standard." https://csrc.nist.gov/csrc/media/publications/fips/46/3 /archive/1999-10-25/documents/fips46-3.pdf, 1977. Last accessed on October, 2018.

[29] X. Lai, J. L. Massey, and S. Murphy, "Markov ciphers and differential cryptanalysis," in *Advances in Cryptology — EUROCRYPT '91* (D. W. Davies, ed.), (Berlin, Heidelberg), pp. 17–38, Springer Berlin Heidelberg, 1991.

[30] C. M. Adams, "Constructing symmetric ciphers using the cast design procedure," *Designs, Codes and Cryptography*, vol. 12, pp. 283–316, Nov 1997.

[31] J.-F. Gallais, I. Kizhvatov, and M. Tunstall, "Improved trace-driven cache-collision attacks against embedded aes implementations," in *Information Security Applications* (Y. Chung and M. Yung, eds.), (Berlin, Heidelberg), pp. 243–257, Springer Berlin Heidelberg, 2011.

[32] X. jie ZHAO and T. WANG, "Improved cache trace attack on aes and clefia by considering cache miss and s-box misalignment," 2010. zhaoxinjieem@163.com 14645 received 2 Feb 2010, last revised 5 Feb 2010.

[33] 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 2013, Gaithersburg, Maryland, USA, 18-20 June 2013 - Companion Volume*, pp. 74–83, 2013.

[34] C. Rebeiro and D. Mukhopadhyay, “Boosting profiled cache timing attacks with A priori analysis,” *IEEE Trans. Information Forensics and Security*, vol. 7, no. 6, pp. 1900–1905, 2012.

[35] A. C. Atici, C. Yilmaz, and E. Savas, “Cache-timing attacks without a profiling phase,” *Turkish Journal of Electrical Engineering & Computer Sciences*, vol. 26, pp. 1953–1966, 2018.

[36] L. Null and J. Lobur, *The Essentials of Computer Organization and Architecture*. Jones & Bartlett Learning, 5 ed., 2018.

[37] G. Blanchet and B. Dupouy, *Computer Architecture*. John Wiley & Sons, Incorporated, 1 ed., 2013.

[38] D. Harris and S. Harris, *Digital Design and Computer Architecture*. Morgan Kaufmann, 2 ed., 2012.

[39] “Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3B: System Programming Guide, Part 2.” https://software.intel.com/sites/default/files/managed/7c/f1/253669-sdm-vol-3b.pdf, 2018. Last accessed on October, 2018.

[40] “Preliminary Processor Programming Reference (PPR) for AMD Family 17h Models 00h-0Fh Processors.” https://www.amd.com/system/files/TechDocs/54945_PPR_Family_17h_Models_00h-0Fh.pdf, 2017. Last accessed on October, 2018.

[41] “Performance Application Programming Interface (PAPI).” http://icl.utk.edu/papi/, Last accessed on October, 2018.

[42] N. Smeds, “Openmp application tuning using hardware performance counters,” in *Proceedings of the OpenMP applications and tools 2003 international conference on OpenMP shared memory parallel programming*, WOMPAT’03, (Berlin, Heidelberg), pp. 260–270, Springer-Verlag, 2003.

[43] C. Yilmaz and A. Porter, “Combining hardware and software instrumentation to classify program executions,” in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE ’10, (New York, NY, USA), pp. 67–76, 2010.

[44] C. Yilmaz, “Using hardware performance counters for fault localization,” in *Proceedings of International Conference on the Advances in System Testing and Validation Lifecycle*, VALID ’10, pp. 87–92, 2010.

[45] B. Ozcelik, K. Kalkan, and C. Yilmaz, “An approach for classifying program failures,” in *Proceedings of International Conference on the Advances in System Testing and Validation Lifecycle*, VALID ’10, pp. 93–98, 2010.

[46] R. Spreitzer and B. Gérard, "Towards more practical time-driven cache attacks," in *Information Security Theory and Practice. Securing the Internet of Things - 8th IFIP WG 11.2 International Workshop, WISTP 2014, Heraklion, Crete, Greece, June 30 - July 2, 2014. Proceedings*, pp. 24–39, 2014.

[47] R. Spreitzer and T. Plos, "On the applicability of time-driven cache attacks on mobile devices," in *NSS*, vol. 7873 of *Lecture Notes in Computer Science*, pp. 656–662, Springer, 2013.

[48] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *USENIX Security Symposium*, pp. 549–564, USENIX Association, 2016.

[49] R. Spreitzer and T. Plos, "Cache-access pattern attack on disaligned AES t-tables," in *COSADE*, vol. 7864 of *Lecture Notes in Computer Science*, pp. 200–214, Springer, 2013.

[50] A. Bogdanov, T. Eisenbarth, C. Paar, and M. Wienecke, "Differential cache-collision timing attacks on AES with applications to embedded cpus," in *Topics in Cryptology - CT-RSA 2010, The Cryptographers' Track at the RSA Conference 2010, San Francisco, CA, USA, March 1-5, 2010. Proceedings*, pp. 235–251, 2010.

[51] 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*, pp. 111–126, 2015.

[52] B. Gülmezoglu, M. S. Inci, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross-vm cache attacks on AES," *IEEE Trans. Multi-Scale Computing Systems*, vol. 2, no. 3, pp. 211–222, 2016.

[53] G. I. Apecechea, T. Eisenbarth, and B. Sunar, "S$a: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pp. 591–604, 2015.

[54] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! A fast, cross-vm attack on AES," in *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, pp. 299–319, 2014.

[55] G. I. Apecechea, M. S. Inci, T. Eisenbarth, and B. Sunar, "Fine grain cross-vm attacks on xen and vmware," in *2014 IEEE Fourth International Conference on Big Data and Cloud Computing, BDCloud 2014, Sydney, Australia, December 3-5, 2014*, pp. 737–744, 2014.

[56] M. Weiß, B. Heinz, and F. Stumpf, "A cache timing attack on AES in virtualization environments," in *Financial Cryptography*, vol. 7397 of *Lecture Notes in Computer Science*, pp. 314–328, Springer, 2012.

[57] Z. H. Jiang and Y. Fei, "A novel cache bank timing attack," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 139–146, Nov 2017.

[58] G. Irazoqui, T. Eisenbarth, and B. Sunar, "MASCAT: preventing microarchitectural attacks before distribution," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY 2018, Tempe, AZ, USA, March 19-21, 2018*, pp. 377–388, 2018.

[59] "Software Guard eXtensions (SGX)." https://software.intel.com/en-us/sgx, Last accessed on October, 2018.

[60] A. C., R. P. Giri, and B. L. Menezes, "Highly efficient algorithms for AES key retrieval in cache access attacks," in *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pp. 261–275, 2016.

[61] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pp. 353–364, 2016.

[62] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games - bringing access-based cache attacks on AES to practice," in *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pp. 490–505, 2011.