Cache Timing Attacks

Sebastian Banescu

16 June 2011

Abstract

This paper presents several side-channel attacks based on timing information leaked from CPU cache memory. The attacks are focused towards cryptographic ciphers that have an implementation based on lookup tables. Several attacks lead to a recovery of a major part of the secret key, such that an exhaustive search on the rest of the undetermined bits becomes computationally feasible. This attack is possible due to the data-dependent lookups performed during the encryption process. Since encryptions are performed in variable amounts of time this leads to a correlation between the time and data. By making some wise assumptions based also on the cipher structure, the attacker is able to extract the secret key from the earlier correlation. The paper also discusses the applicability of these attacks and offers some countermeasures.

1 Introduction

Cryptanalysis has been traditionally focused toward finding and exploiting weaknesses in the underlying mathematical model of cryptographic ciphers. There exist several powerful mathematical tools, like differential cryptanalysis [6], that enable breaking block ciphers, such as DES [19]. However, as standards become more mature, their underlying cryptographic cipher becomes much harder to "break" in this traditional sense. That is the main reason why researchers have started looking for other methods of performing cryptanalysis.

In the mid 1990s, Kocher [15] introduced a new approach to cryptanalysis called *side-channel attacks*. These types of attacks take advantage of information leaked by the physical implementation of cryptographic systems by using entities or system resources not normally viewed as a data container to transfer information between subjects. These metadata objects (e.g. file locks, busy flags, branch prediction tables, execution time) are needed to register the state of the system. Unfortunately, those channels can lead to illegitimate channels violating the system's security promises.

The existing types of side-channel attacks can be grouped by the type of information they are based on (e.g. power consumption, timing information, electromagnetic leaks, sound). This work focuses on attacks that arise due to timing information leaked from CPU architectures that include cache memory. The possibility of using cache behavior as an information channel was first presented in 1992 by Hu [13], however this was not targeted against cryptographic ciphers. Some timing-channel attacks in the literature tended to focus on particular implementations of ciphers that were not used in practice. An example is the timing attack on AES given by Koeune and Quisquater [16], which uses a particular implementation of multiplication in a finite field, $GF(2^8)$.

Generally it was believed that such attacks cannot be successful on optimized implementations of ciphers like AES, because the transformations described by the cryptographic algorithm are implemented in look-up tables (LUTs). However, due to the hierarchical memory model, subsequent accesses to these LUTs are performed in a variable amount of time, mostly due to CPU cache memory. Furthermore, this variation in the time delay of cache memory accesses leaks information that is used to perform a side-channel cryptanalysis attack. Throughout this paper we will regard the latter mentioned type of attack as a *cache timing attack*. More details about the operation of cache memory and how it leaks information will be given in Section 2.

Timing channels are not the only means of attacking computers with cache. Side-channel attacks based on power consumption of cache memory have also been executed against DES [22] and AES [17]. However, this paper mainly focuses on cache timing attacks, briefly referring other types of cache attacks where appropriate. The first implementations of such cache timing attacks were first implemented by Tsunoo *et al.* [30] against DES. Then, Bernstein [2] presented a successful cache timing attack on a popular implementation of AES, namely OpenSSL. Shortly afterwards, Osvik *et al* [21] introduced a couple cache-based attack using a hybrid timing & memory side-channel. These latter three works are the most heavily cited papers in the field of cache timing attacks and they will be described in detail in Section 3. Then, in Section 4 some countermeasures are proposed. Finally section 5 offers a discussion about the attacks and conclusions.

2 Problem Analysis

The perfect memory system would provide data at the working frequency of the CPU and would require constant time for each access. However, this is not the case for CPU architectures nowadays. Main memory, which is implemented using dynamic random access memory (DRAM), is the central repository for all data produced and used by the CPU. The ever increasing gap between processor computing frequencies and DRAM latency

Memory	Access Time	Capacity	Cost
Disk	10^5 clks	100s GBs	cheap
Main Memory	100s clks	GBs	affordable
L3 Cache	12 clks	MBs	expensive
L2 Cache	5 clks	100s KBs	expensive
L1 Cache	1 clks	10s KBs	more expensive
Registers	0.5 clks	KBs	most expensive

Table 1: Memory Access Time versus Capacity versus Cost

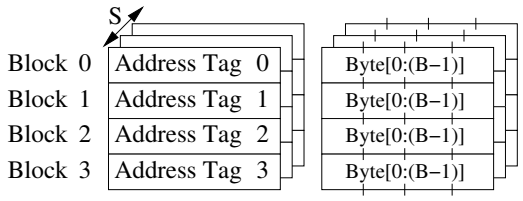

Figure 1: 4-Way Set Associative Cache Memory [8]

has lead to a bottleneck, which is a major obstacle in improving computer performance. The memory hierarchy [29] was developed by computer architecture experts as a solution for increasing performance.

The memory hierarchy comprises of several memory levels including: CPU internal, main and mass storage. Because of cost factors, the access time and capacity of memories used in a computer are inversely proportional, as can be seen from Table 1. Nevertheless, this model offers a good trade-off between all of the factors involved.

2.1 Cache Structure & Operation

Cache memory is positioned between the CPU registers and main memory, in the overall memory hierarchy, minimizing the latency of main-memory. Nowadays common processors offer 3 levels of cache memory, with the first level (L1) being the closest to the CPU registers. Generally, L1 and L2 caches are each split into instruction caches and separate data caches. L3 usually stores both instructions and data.

Regardless the purpose or level of the cache, it is structured in the same manner. Similarly to other memory types, a cache is divided into blocks (also called lines) of fixed size (B bytes). General sizes are 32, 64 or 128 bytes, but the size of the blocks can vary for each cache level. Blocks, are grouped into a number (S) of sets such that each set has an equal number of blocks. The number (W) of blocks in a set denotes the associativity of a cache, and is usually a small power of 2. The total cache size is: $S \times W \times B$ bytes. Since this size is much smaller than the number of directly addressable bytes in main memory N (e.g. 4 GB on 32-bit CPUs), a mapping strategy needs to be adopted.

The cache associativity determines how the main memory blocks map into blocks of the cache. More specifically, if we have a 4-way set associative cache like in Figure 1, then a main memory block can map to any of the 4 blocks of a given set. There exist replacement algorithms [23] that select which block will be replaced in the given set. The set corresponding to a main memory block is determined from its physical address by splitting it into:

- a tag address taking $\log(\frac{N}{S \times B})$ bits from the most significant part of the address;
- a set number taking $\log(S)$ bits from the middle part of the physical address;
- a block offset taking $\log(B)$ bits from the least significant part of the physical address.

When a memory reference is made by the CPU, the tag address of the main memory block is compared with all the tags in the corresponding set. If the tag is found then this reference qualifies as a *cache hit*. Meaning that there is no need to retrieve the data from main memory since it is already located in the cache, and the data can be immediately provided to the CPU. Conversely, if the tag is not found in the corresponding set then the memory reference qualifies as a *cache miss*. Meaning that the data needs to be retrieved from main memory. Consequently, the data will be provided to the CPU with a noticeable delay relatively to the case where a cache hit occurs. The caching mechanism operation is based on the principals of spatial and temporal locality, which help to minimize the number of cache misses. Temporal locality states that the same data blocks will likely be requested repeatedly during the execution of a process. Spacial locality states that data blocks from nearby addresses are likely to be subsequently accessed. Even though the number of cache misses is reduced by these principles, they are not eliminated. Hence, the variation in the access time of different memory addresses is the key aspect of cache timing attacks.

2.2 Attacker Model

The cache timing attacks on cryptographic ciphers aim to uncover the secret key used for encryption and decryption. The attack model is the known-plaintext attack, where the adversary can feed inputs (plaintext) to the cipher and inspect the outputs (ciphertext). Another important aspect known by the attacker is the specific cipher implementation that is used by the targeted system. This implies that the attacker has a while-box view of the cryptographic algorithm used by the cipher. The only unknown being the secret key used by the cipher. Moreover, the attacker should also be able to send an arbitrary number of plaintexts to the targeted system for encryption and also be able to obtain the timing required for this encryption.

The majority of cryptographic ciphers such as DES [19] and AES [9], make use of LUTs, as a simple method to speed-up encryption and decryption. LUTs are stored as one dimensional arrays in memory, and their location affects the execution time of a cryptographic operation. For instance a LUT is represented by an array having A elements, each having a size of E

bytes. The array is mapped into $\lceil A/B \rceil$ cache blocks, each containing at most $\lceil B/E \rceil$ elements. The LUT may not necessarily be loaded into cache before the encryption or decryption process starts. However, depending on the state of the cache different attacks may or may not be applicable. This will be discussed in the following section.

3 Types of Cache Attacks

The timing needed for encryption and decryption of a message strongly depends on where the data needed for this process resides at the moment it is referenced (e.g. one of the three cache levels, main memory). Cache misses occur at every cache level and they are categorized in [12] as:

- *cold start misses*, which occur when data is first referenced;
- *capacity misses*, which occur if the size of the LUTs used by the cipher is larger than the size of the cache; this type of cache misses do not occur for most cryptographic cipher implementations;
- *conflict misses*, which occur when at least two elements of LUTs that map to the same cache block are used.

The type of cache misses which occur during the execution of an encryption or decryption strongly depend on the initial state of the cache. Certain attacks require a certain initial state of the cache, in order to be applied successfully. Depending on the initial cache state, the attack may use either cold start misses, conflict misses or both. Capacity misses are excluded since the most popular implementations of cryptographic ciphers use LUTs that fit in the cache memory of common architectures. Based on this idea, Canteaut *et al.* [8] classify the possible attacks in three groups:

1. *Reset attacks* require that all or most LUTs used by the cryptographic cipher are not to be loaded in the cache before the attack commences. Therefore this type of attacks is mainly based on cold start misses.
2. *Initialization attacks* require the adversary to be able to set the cache into a known state before the attack commences. Therefore this type of attacks is based both on cold start misses and conflict misses.
3. *Micro-architecture attacks* require the cache to hold all or most LUTs that will be used by the cipher, before the attack commences. Therefore this type of attacks is partially based only on conflict misses and partially on other timing penalties that strongly depend on the CPU micro-architecture.

These three types of attacks are not limited to timing channels. Measuring power dissipation levels during the encryption process offers more information about each data access, while timing delays only offer an overview of the events in the same process. The works of Page [22] and Lauradoux [17] are only a couple of examples of cache attacks which exploit power channels. However, in the remainder of this paper we will focus on timing channels.

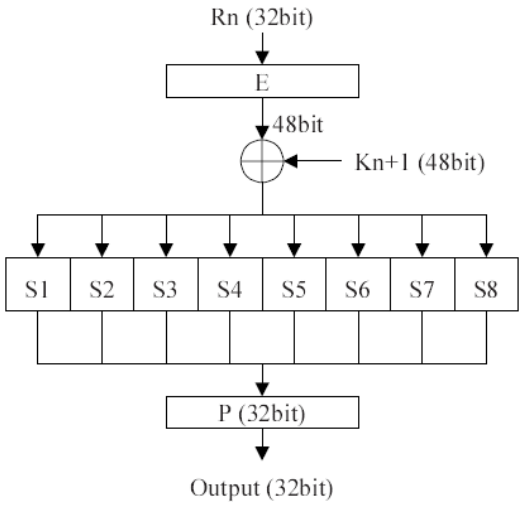

Figure 2: Round Function of DES from [30]

3.1 Reset Attacks

Reseting a cache (i.e. flushing its contents) can be done simply by removing the voltage supply. This method may be feasible on some embedded systems like smartcards, however it cannot be done on personal computers. Therefore the adversary should have a user account on the targeted machine. The attacker then performs enough memory accesses to load the cache with data blocks not used by the cryptographic cipher. Once the attack commences, many cache misses will be caused by memory references which involve LUT elements whose corresponding memory blocks have not yet been loaded into the cache.

3.1.1 Tsunoo's Attack against DES

The first practical implementation of reset attacks based on timing channels has been presented by Tsunoo *et al.* [30] against DES. The DES cipher specification [19] refers to the LUTs used in the encryption process as S-boxes. An S-box has 64 entries of 4 bits each. DES uses 8 such S-boxes per round as shown in Figure 2. The attack presented by Tsunoo is based on the fact that plaintexts with a long encryption time correspond to a high frequency of cache misses. On the other hand, a high number of different S-box input values leads to an increased number of cache misses. Therefore, if the time needed for the encryption of a given plaintext is long, one can infer that there were many different S-box input values for that plaintext.

In order to offer a simplified explanation of the attack in [30], the authors use a cipher with 2 S-boxes shown in Figure 3. It employs independent key bits K_0 and

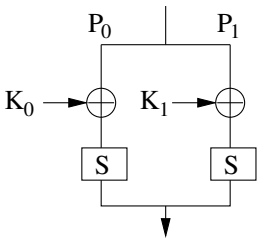

Figure 3: Cipher with 2 S-boxes [30]

K_1 XOR-ed with two bits of the plaintext, i.e. P_0, respectively P_1. The result of each XOR operation is fed into a different S-box. The *key difference* $(K_0 \oplus K_1)$ can be inferred from the plaintext using one of the following relations:

$$P_0 \oplus K_0 = P_1 \oplus K_1 \rightarrow P_0 \oplus P_1 = K_0 \oplus K_1 \quad (1)$$
$$P_0 \oplus K_0 \neq P_1 \oplus K_1 \rightarrow P_0 \oplus P_1 \neq K_0 \oplus K_1 \quad (2)$$

By finding key differences the attacker is able to reduce the key search space. There exist two methods of obtaining the key difference corresponding to each of the previous two formulas. The first method is called the *non-elimination table attack* and it is applicable for plaintexts for which equation (1) holds. In this case the inputs for the two S-boxes are equivalent, therefore the same S-box elements will be referenced resulting in a large number of cache hits. Using this fact and equation (1) it follows that the most frequent $P_0 \oplus P_1$ pair of plaintext bits, has the highest probability to be the correct key difference.

The second method is called the *elimination table attack* and it is applicable for plaintexts for which equation (2) holds. In this case the inputs for the S-boxes are different resulting in a large number of cache misses. This means that the encryption time will be longer for the chosen plaintexts. The most probable correct key difference in this case is the $P_0 \oplus P_1$ pair of plaintext bits that appears least frequently.

Since DES only specifies that one access per round is made on each of the 8 S-boxes, it means that 16 accesses are made on each S-box, during encryption. Relatively to the 64 entries of an S-box, the number of 16 accesses is rather small, resulting in a high probability of accessing different elements of the separate S-boxes. Hence, the elimination table attack can be applied on DES.

The elimination attack on DES from [30] is divided into 2 stages. In the first stage several plaintexts are generated randomly. For each plaintext the encryption time is measured after reseting the L1 cache. Time measurement accuracy is obtained by reading the time stamp counter [14] of the CPU before and after encryption. For the elimination attack the plaintexts that have shown long encryption times are relevant.

In the second stage of the attack described in [30] the authors use a particularly selected key schedule, having a single bit cyclic left shift in the first round and a total number of 28 cyclic left shifts after 16 rounds. Since the key for DES has 56 bits and in the key scheduling part it is split into two 28 bit parts, it means that the key used in the final round will be equal to the original key. Furthermore, the first round key and the final round key will differ by only 1 bit cyclic left shift.

Supposing that the input values of the S-boxes of the first round differ respectively from the inputs of the corresponding S-boxes from round 16 the following equation holds:

$$K_1 \oplus K_{16} \neq E(R_0) \oplus E(R_{15}) \quad (3)$$

where K_i represents the key in encryption round i and $E(R_{i-1})$ is the expanded output of round $i-1$. Therefore, according to the elimination table attack, the least frequent value $E(R_0) \oplus E(R_{15})$ occurring in the set of plaintexts collected in the first stage of the attack, is most likely to be the correct key difference. Note that this computation of the key difference is done separately for each of the 8 S-boxes in the round function.

However, not all the 6 input bits of the key difference can be determined in this stage because of the way cache memory loads data from main memory. Following the spatial locality principle, upon a cache miss, the referenced memory address along with its aligned adjacent addresses are loaded into the cache. This is called the cache load size and it depends on the cache size and structure which is also specific to the CPU memory addressing size.

The attack in [30] was executed on a Pentium III CPU having a cache load size of 32-bytes. Since each S-box entry was of integer type (4 bytes), it means that 8 adjacent entries could be loaded simultaneously into the cache. This would imply that the 3 LSBs of the key difference are indeterminate and the other 3 MSBs are determined successfully. However, the authors managed to determine 4 of the MSBs of each S-box input because the S-box addresses were not aligned on a 32-byte boundary, hence preventing the cache from loading 8 adjacent S-box entries upon a cache miss.

In order to completely determine the secret key of 56-bits, the first step is to prepare a random plaintext p_1 and encrypt it with the real secret key, obtaining the ciphertext c_1. In the second step the 4 determined MSBs from the 8 key differences obtained in stage 2 are used. The 32 bits corresponding to these determined MSBs are guessed for K_1 and subsequently K_{16} is obtained by XOR-ing the key differences from stage 2 and the guessed value. Afterwards an exhaustive search is done on the remaining 24 bits of the secret key, until the encryption of p_1 with the 56-bit guessed key is equal to c_1. If a matching ciphertext is not found after all the 24 bit combinations have been tried, then step 2 is repeated with a different guess of the 32 bits of K_1. If all possible values of K_1 have been tried and no matching ciphertext has been found, it means that the least frequent value $E(R_0) \oplus E(R_{15})$ occurring in the set of plaintexts collected in the first stage of the attack is not the correct key difference. In this situation one can either choose another value for the key difference that does not appear frequently or more

appropriately generate more plaintext-ciphertext pairs having long encryption times and recompute the least frequent $E(R_0) \oplus E(R_{15})$ value.

The experimental results in [30] show that the probability of successfully recovering the secret key is over 90% if 2^{17} plaintext-ciphertext pairs having longer encryption times are collected from a total of at least 2^{23} generated pairs. This success probability increases proportionally to the number of generated plaintext-ciphertext pairs. An important thing to note is that the results are specific to the particular implementation of DES and the Pentium III platform on which this attack was executed. For instance if the S-box entries would have been of character type (1 byte) then upon a cache miss, 32 adjacent S-box elements would have been loaded into the cache, greatly decreasing the differences in encryption time between different plaintexts. The previous attack was also performed on an implementation with character type S-box entries and it failed. However, the authors note that most implementations of DES use integer type S-box entries because these offer better performance on the most common platforms. The same elimination table attack can be successfully executed on other ciphers such as Triple-DES, MISTY1 [18] and Camellia [1].

The non-elimination table attack was successfully executed against the original implementation of AES [25]. Conversely to the expectation of the authors of [30], no correlation was found between the frequency of occurrence of cache misses and encryption time. However, they found that lower frequencies of cache misses implied longer encryption times. Using 2^{18} plaintexts with long encryption times, they were able to retrieve 96-bits of the AES key, by regarding the value counted most often as the correct key difference.

3.1.2 Bernstein's Attack against AES

Bernstein [2] presents a reset attack based on the speculation that the timing delay of any S-box lookup is highly correlated with the whole AES computation. This leaks information about each key byte XOR-ed with a plaintext byte $(k_j \oplus p_j)$ such that using the distribution of AES encryption timings as a function of p_j, one can compute the value of the corresponding key byte, k_j for each byte index $j \in \{0, 1, ..., 15\}$.

A View of AES

For a clear understanding of this attack a short summary of AES is initially presented. The specification of Rijndael [9] mentions four basic transformations that comprise the round function: SubBytes, ShiftRows, MixColumns and AddRoundKey. The different transformations operate on the intermediate result, called the *state*. The state can be pictured as a two-dimensional array of bytes. This array has 4 rows, the number of columns is equal to the block length divided by 32. The possible lengths of blocks and keys are: 128, 192 and 256 bits, which need 10, 12 respectively 14 rounds for encryption. The attack in [2] focuses only on AES with 128 bit blocks. It is a generic statistical timing attack, because it does not utilize any specific knowledge of why the value of a specific input affects the encryption time. Hence, there is no reason why it cannot be applied to other AES block sizes.

ShiftRows and AddRoundKey can be easily implemented on the most popular processors today since they imply simple native operations. However this is not the case for the other two transformations SubBytes and MixColumns. The SubBytes transformation is a non-linear byte substitution, operating on each of the state bytes independently. The substitution table (S) is invertible and is constructed by the multiplicative inversion in $GF(2^8)$ and the affine transform over $GF(2)$. MixColumns implies a multiplication of each column by a fixed polynomial over $GF(2^8)$. An implementation of these polynomial operations leads to relatively long encryption times. In order to improve the performance of the AES cipher, four 1024 byte LUTs are defined as follows:

$$T_0[a] = \begin{bmatrix} S[a] \bullet 02 \\ S[a] \\ S[a] \\ S[a] \bullet 03 \end{bmatrix} \quad T_1[a] = \begin{bmatrix} S[a] \bullet 03 \\ S[a] \bullet 02 \\ S[a] \\ S[a] \end{bmatrix}$$

$$T_2[a] = \begin{bmatrix} S[a] \\ S[a] \bullet 03 \\ S[a] \bullet 02 \\ S[a] \end{bmatrix} \quad T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ S[a] \bullet 03 \\ S[a] \bullet 02 \end{bmatrix}$$

where $\bullet$ denotes the multiplication in $GF(2^8)$. Furthermore, AES uses two auxiliary arrays k and y which are initialized with the key (K), respectively the key XOR-ed with the plaintext $(K \oplus P)$. Array x is viewed as four 4-byte arrays: $x = (x_0, x_1, x_2, x_3)$. In each round i a 4-byte array is computed:

$$e_i = (S[x_3[1]^{(i-1)}] \oplus M[i], S[x_3[2]^{(i-1)}], S[x_3[3]^{(i-1)}], S[x_3[0]^{(i-1)}])$$

where $M = \{1, 2, 4, 8.16, 32, 64, 128, 27, 54\}$ and $i \in \{1, 2, ..., 10\}$. Subsequently the new value of x for that round is denoted by $x^{(i)} = (e_i \oplus x_0^{(i-1)}, e_i \oplus x_0^{(i-1)} \oplus x_1^{(i-1)}, e_i \oplus x_0^{(i-1)} \oplus x_1^{(i-1)} \oplus x_2^{(i-1)}, e_i \oplus x_0^{(i-1)} \oplus x_1^{(i-1)} \oplus x_2^{(i-1)} \oplus x_3^{(i-1)})$. Similarly array y is viewed as four 4-byte arrays: $y = (y_0, y_1, y_2, y_3)$ and in each round it gets assigned the following value:

$$\begin{aligned}
&(T_0[y_0[0]^{(i-1)}] \oplus T_1[y_1[1]^{(i-1)}] \oplus T_2[y_2[2]^{(i-1)}] \oplus T_3[y_3[3]^{(i-1)}] \oplus x_0^{(i)}, \\
&T_0[y_1[0]^{(i-1)}] \oplus T_1[y_2[1]^{(i-1)}] \oplus T_2[y_3[2]^{(i-1)}] \oplus T_3[y_0[3]^{(i-1)}] \oplus x_1^{(i)}, \\
&T_0[y_2[0]^{(i-1)}] \oplus T_1[y_3[1]^{(i-1)}] \oplus T_2[y_0[2]^{(i-1)}] \oplus T_3[y_1[3]^{(i-1)}] \oplus x_2^{(i)}, \\
&T_0[y_3[0]^{(i-1)}] \oplus T_1[y_0[1]^{(i-1)}] \oplus T_2[y_1[2]^{(i-1)}] \oplus T_3[y_2[3]^{(i-1)}] \oplus x_3^{(i)}).
\end{aligned}$$

On the 10th round the MixColumns transformation is not executed. Hence, the four 4-byte arrays comprising y get assigned values directly from the S-box (S) instead of using the previous formula. The final value of array y is the output of the AES cipher.

Attack Setup and Phases

The setup described in [2] involves a client computer which is remotely connected to a server through a custom designed protocol. The client generates random

plaintexts and sends them to the server. The server performs an AES encryption of the given plaintext and returns the ciphertext together with the time difference before and after the encryption process. The time is computed using the CPU time stamp counter [14] like in the attack by Tsunoo [30] presented earlier.

Bernstein's attack [2] is divided into 3 phases called: profiling, attacking and analysis. The first phase which is executed is profiling, where a series of random 400, 600 and 800 byte packets are sent to a test server which encrypts them using AES. The test server is on a machine controlled by the attacker. It has an all zeros key and it runs the same AES software on the same type of CPU as the target server. Approximately 2^{22} packets of each size are sent to the test server and the encryption timing of each packet is recorded to determine the profile of the system. This profile can be seen as a matrix Θ whose rows represent the index in {0,...,15} of the input bytes and the columns represent all the possible 256 values of a byte. Every time the test server (having AES key K) returns the encryption time (τ) of plaintext $P = (p_0, ..., p_{15})$, the corresponding 16 matrix entries $\Theta[j][p_j], j \in \{0, ..., 15\}$ get incremented by τ.

The second phase is the attacking phase which is performed on the target server having the secret key. Initially the adversary sends a packet consisting only of zeros and obtains the corresponding ciphertext. Afterwards a similar process as in the profiling stage is executed, i.e. 2^{27} 400 byte packets, 2^{25} 600 byte packets and 2^{25} 800 byte packets are sent to the target server for profiling. Once again the profile can be viewed as a 16×256 element matrix $\tilde{\Theta}$. Every time the target server (having AES key $\tilde{K}$) returns the encryption time (τ) of plaintext $\tilde{P}$, the corresponding 16 matrix entries $\tilde{\Theta}[j][\tilde{p_j}], j \in \{0, ..., 15\}$ get incremented by τ.

In the final analysis phase a correlation is done between the two baseline profiles of the test server with the known key and the target server with the secret key. This correlation uses the fact that the look-ups in the first round of AES, (e.g. $T_0[y_0[0]^{(0)}] = T_0[k_0 \oplus p_0]$, $T_1[y_1[1]^{(0)}] = T_1[k_5 \oplus p_5]$ and so on) are dependent on both the plaintext input and the secret key. By using the heuristic that those pairs of plaintext and key bytes satisfying:

$$p_j \oplus k_j = \tilde{p_j} \oplus \tilde{k_j}$$

will also have matching profiles, naturally leads to a correlation between Θ and $\tilde{\Theta}$. Hence, possible key candidates can be derived using the following relation:

$$\tilde{k_j} = p_j \oplus k_j \oplus \tilde{p_j} \quad (4)$$

The result of this phase is therefore a series of potential keys that present large correlations. Given the ciphertext obtained after the first step in the attack phase, an exhaustive search can be performed on the remaining space of possible keys. This is facilitated by the fact that the attacker has sent an all zeros plaintext in the first step of the attack phase. Also noting that the test server has an all zeros secret AES key it means that relation (4) becomes: $\tilde{k_j} = p_j$. This way the exhaustive search for the missing key bytes can be done on the test server by enumerating all possible values of corresponding plaintext bytes. Once the ciphertext form the first step of the attack phase is obtained, the entire AES secret key has been recovered.

In [2] this attack is successfully executed on version 0.9.7a of OpenSSL [27], one of the most popular implementations of AES. The author does not indicate that the cache should be reset before each encryption, however in the source code provided in the paper, he performs a series of array operations before each encryption, therefore evicting several elements of the AES LUTs from the cache. Since this attack predominantly capitalizes on cold start misses it falls into the class of reset attacks.

3.1.3 Salembier's Investigation of Bernstein's Attack

In his paper [2], Bernstein does not give an evaluation of the total amount of time required to run the entire attack. Salembier [26] addresses this issue by repeating the attack in [2] on different AMD CPUs. More specifically the test server and the target server had different types of: CPUs and operating systems; also they were running different versions of OpenSSL and had different compilers. An important thing discovered by Slembiers' tests was that adding an unstable clock source to the target server does not thwart the attack.

Salembier performed a series of tests by gradually decreasing the number of packets sent to the server in the profiling phase. He noted that the shortest time span in which he managed to successfully retrieve the secret AES key was about 20 days. This time period is much too lengthy to be of practical use to an attacker, since most servers using SSL and SSH generally change their AES key about every hour. However, this attack can be successfully applied to systems which have fixed AES keys like mobile phones or other embedded systems.

Salembier proposes an interesting method of reducing the time required to collect the information for both the profiling and attack phase. This method would require the attacker to use three different systems to profile and attack the server simultaneously. Each system would profile and attack using a single packet size, for example, system one would profile the server with a known key using 400 byte packets, and then attack the server with 400 byte packets when attempting to determine the unknown key. The overall time would be reduced almost in half from 20.38 days to 10.58 days as shown in Table 2, based on the data collected during the tests in [26]. The profiling and attack phases would both take the maximum time of any of the individual packet size transmission segments, since all packet flows would occur in parallel with each other. The longest transformation segments in Table 2 are achieved by system 3. Although this method is promising some interesting results, the author does not imple-

	Profile time	Attack time
System 1 (400 byte)	2.50 days	1.27 days
System 2 (600 byte)	3.01 days	3.02 days
System 3 (800 byte)	3.54 days	7.04 days

Table 2: Parallel Attack Plan in [26]

ment it in [26], proposing it as future work.

3.1.4 Neve's Extension of Bernstein's Attack

Neve *et al* [20] make a thorough analysis on the attack method in [2], proving why and how this technique works. Using a special software they were able to obtain a snapshot of cache line accesses versus time, when running AES-encryptions in a real environment (having other background processes). From this snapshot they are able to observe that the exploitable different AES execution times, when averaged over many iterations are due to system-dependent cache-evictions. By varying over all possible values for the individual plaintext bytes, Bernstein's technique is implicitly searching for those cache evictions by the system. This leads then to small execution time differences for certain plaintext inputs.

The authors of [20] perform several experiments in which they gradually increase the number of samples sent to the server. The results of these experiments presented on column 5 of Table 3 show that Bernstein's attack cannot practically extract all key bits even when the sample number is very high. Moreover, the authors of [20] present an improvement of Bernstein's technique by using information from the first encryption round to extract more key bits in the second round. The results of this improvement are shown in the last column of Table 3. Thus, using a lower number of samples than Bernstein's attack, this second round analysis allows for a full AES key recovery through simple overall timing measurements.

The idea of Neve's attack is to apply Bernstein's encryption time correlation method to the internal AES state after the first round and the corresponding round key denoted by $x^{(1)}$. While applying the profiling phase for this attack is easy on the attacker's server, the attacking phase is much harder because $x^{(1)}$ is only partially known (from the cryptanalysis on the first round key). However, from the description of AES in Section 3.1.2 we know that each state word depends only on 4 out of the 16 previous state bytes. For example the the third byte of the second roung state can be expressed as: $p_2^{(1)} = x_0[2]^{(1)} \oplus y_0[2]^{(1)}$ where $y_0[2]^{(1)} = S[p_0 \oplus k_0] \oplus S[p_5 \oplus k_5] \oplus 2 \bullet S[p_{10} \oplus k_{10}] \oplus 3 \bullet S[p_{15} \oplus k_{15}] \oplus S[k_{15}] \oplus x_0[2]^{(1)}$. Therefore the attacker can use the disclosed key bits from first round to decrease the search space. This results in a more computationally feasible task of enumerating the remaining undetermined bits of the key bytes on which this state byte is dependent on. In the context of the previous example one can see that $p_2^{(1)}$ only depends on $x_0[0], x_1[1], x_2[2]$ and $x_3[3]$. Similarly all other 15 bytes

CPU	Setting	Samples	Search Time	1st round	1st+2nd round
A	Cygwin	$\geq$2 M	$\approx 1min$	96	128
		1 M	$\approx 10mins$	79	125
		400 K	$\approx 2hrs$	56	115
		200 K	$\approx 5hrs$	42	76
B	Cygwin	8 M	$\approx 1min$	101	128
		2 M	$\approx 14hrs$	62	128
C	Linux	$\geq$2 M	$\approx 12hrs$	68	112
		1 M	$\approx 12hrs$	56	80

Table 3: Results of Analysis Phase from [20]

$p_j^{(1)}, j \in \{0, 1, 3, .., 15\}$ will depend on a group of four secret key bytes. The profiling and attack phases are performed similarly to Bernstein's attack, resulting in the two profile matrices $\Theta[j][p_j^{(1)}]$ and $\tilde{\Theta}[j][p_j^{(1)}]$. After obtaining the secret key for the second round, the value of the AES secret key can be easily computed using the formulas in Section 3.1.2.

Finally, the authors of [20] conclude that Bernstein's attack cannot be easily extended into a remote side-channel analysis by simply letting the client computer measure the round trip times of the queries to the server. This is mainly due to the fact that the cache latencies are about two orders of magnitude smaller compared to network delays. Bernstein's attack would require an extensive number of measurements to extract the cache behavior from the overall time. However, they propose an advanced attack strategy that could increase the signal to noise ratio by submitting carefully chosen plaintexts. This idea originates from the fact that the profiles for each byte are deeply linked to the position of the LUTs loaded into the cache. Therefore, some bytes present similar profiles that can be combined to reduce the noise. The authors continue to state that Using such chosen plain-text amplifications, combined with the second round extension, a remote recovery eventually becomes possible in realistic times. However, no results of such an attack are presented in [20].

3.2 Initialization Attacks

This category of attacks imply that the attacker is enabled to initialize any chosen cache blocks with specific data from the LUTs used by the cryptographic cipher. This can be achieved only in multi-user systems where the attacker has access to the cache memory. Initialization can be achieved by first reseting the cache and subsequently performing some encryptions with a carefully chosen key in order to load the desired LUT blocks into the cache.

Similarly to reset attacks, this class of attacks uses the encryption delays introduced by cache misses. The main difference being that initialization attacks mainly capitalize on conflict misses instead of cold start misses mainly used by the class of attacks presented in Section 3.1.

3.2.1 Osvik's Attack against AES

The attack method proposed by Osvik *et al* [21] uses a hybrid side-channel comprising of both timing and cache memory state. It is not a pure timing attack, however since its steps involve the use of timing delays introduced by cache misses we have decided to dedicate this section to describing this attack.

The known-plaintext attack model is also employed here, however additionally to this, the adversary is able to directly observe the state of the cache (without being able to read the actual contents belonging to the target). This requirement limits the attack to multi-user systems where the attacker has legitimate access to the cache by means of a user account. Therefore the attacker can operate synchronously with the encryption on the same processor by using (or eavesdropping on) some interface that triggers encryption under an unknown key.

Using the notation from Section 2.1 the authors of [21] define δ as the cache line size (B) divided by the size of each AES LUT entry (usually 4 bytes). Further, the authors assume the worst case scenario, i.e. when the AES LUTs are aligned on the memory boundary. Non-aligned LUTs would leak an extra bit per key byte in the first round as seen in the attack by Tsunoo [30]. Another assumption made in [21] is that all LUTs are mapped into distinct cache sets which is said to hold with high probability on many systems. A *memory block* of a byte index g in an AES LUT (T_l) is denoted by $\langle g \rangle = \lfloor g/\delta \rfloor$. Using this notation we can say that two bytes g, h which are used as LUT indices in the same T_l fulfill the condition $\langle g \rangle = \langle h \rangle$ if and only if they cause access to the same cache block. This represents a collision on the most significant $\log_2(256/\delta)$ bits of the two bytes g and h.

The attack in [21] is executed on the first two rounds of AES encryption. In order to determine whether a memory block was accessed during a given encryption the predicate Q is defined as: $Q_K(P, l, g) = 1$ if and only if the encryption of plaintext P under key K accessed the memory block of index g in LUT T_l in any of the encryption rounds; otherwise $Q_K(P, l, g) = 0$.

First Round Attack

The attack on the first encryption round is presented first in an ideal form. Supposing that the attacker has obtained several samples of $Q_K(P, l, g)$ for some LUT T_l, an arbitrary index g and known but random plaintexts P. By denoting one byte of the plaintext as p_i and one byte of the secret key as k_i, the first encryption round performs accesses of the form $T_l[p_i \oplus k_i]$, where $i \equiv l (\mathrm{mod}\ 4)$. Then considering candidate values $\tilde{k_i}$ for k_i the attacker can eliminate the ones that do not satisfy the following relation: $\langle g \rangle = \langle p_i \oplus \tilde{k_i} \rangle$. The remaining $\tilde{k_i}$ values will satisfy $\langle k_i \rangle = \langle \tilde{k_i} \rangle$, meaning that these values will lead to an access of the memory block of g in LUT T_l. The eliminated $\tilde{k_i}$ values will satisfy $\langle k_i \rangle \neq \langle \tilde{k_i} \rangle$, meaning that these values will not lead to an access of the memory block of g in LUT T_l, hence having $Q_K(P, l, g) = 0$. However, from the total of 36 accesses to LUT T_l (4 in each of the first 9 encryption rounds) there still remain 35 access affected by other plaintext bytes. Therefore the probability that the encryption will not access the memory block of g in any round is $(1 - \delta/256)^{35}$. On platforms such as Athlon 64/Opteron and Pentium 4E where the cache line size is 64 bytes, $\delta = 64/4$; then the previous probability becomes appropriately 0.104. Therefore, after obtaining a few dozen samples satisfying $Q_K(P, l, g) = 1$, the correct value of $\langle k_i \rangle$ can be determined. For the afore mentioned platforms this would uncover the $\log_2(256/\delta) = 4$ most significant bits of every key byte k_i.

Second Round Attack

The attack on the second round of the AES encryption described in [21] employs four equations derived from the AES proposal [9]. These equations are a non-linear mixing of input bytes and key bytes and they express the indices $y_0[2]^{(1)}, y_1[1]^{(1)}, y_2[0]^{(1)}$ and $y_3[3]^{(1)}$, where $y^{(1)}$ represents the auxiliary array used in the second encryption round of AES presented in Section 3.1.2. An example of such an equation for the second round of encryption is: $y_0[2]^{(1)} = S[p_0 \oplus k_0] \oplus S[p_5 \oplus k_5] \oplus 2 \bullet S[p_{10} \oplus k_{10}] \oplus 3 \bullet S[p_{15} \oplus k_{15}] \oplus S[k_{15}] \oplus x_0[2]^{(1)}$.

In the context of this equation, suppose that the attacker has obtained samples of the ideal predicate $Q_K(P, l, g)$ for LUT T_2, arbitrary table indices g and known but random plaintexts P. Considering that the most significant $\log_2(256/\delta)$ bits were found in the attack against the first round, the only undetermined bits affecting the memory block accessed by $T_2[y_0[2]]$ are the least significant $\log_2 \delta$ bits of k_0, k_5, k_{10} and k_{15}. This gives a total of δ^4 possible candidate values $\tilde{k_0}, \tilde{k_5}, \tilde{k_{10}}$ and $\tilde{k_{15}}$, which are easily enumerated.

For each candidate guess, and each sample $Q_K(P, l, g)$ the attacker evaluates the previous equation while fixing the unknown low bits of $x_0[2]^{(1)}$ to an arbitrary value. The result is a predicted index $\tilde{y_0}[2]$. If the guess was correct then $\langle g \rangle = \langle \tilde{y_0}[2] \rangle = \langle y_0[2] \rangle$ and thus the lookup in T_2 causes an access to memory block of g, hence $Q_K(P, l, g) = 1$. Conversely, if the guess was incorrect then $k_i \neq \tilde{k_i}$ for some $i \in \{0, 5, 10, 15\}$ and thus $y_0[2] \oplus \tilde{y_0}[2] = c \bullet S[p_i \oplus k_i] \oplus c \bullet S[p_i \oplus \tilde{k_i}] \oplus ...$ for some $c \in \{1, 2, 3\}$. Since the plaintext is random, the remaining terms are independent of the first two presented in the previous equation. Using a differential property of the AES S-box [6] results in the probability that $T_2[y_0[2]]$ does not cause an access to the memory block of g in T_2 is at least $(1 - \delta/256)^3$. The remaining 35 accesses to T_2 performed during encryption will access the memory block of g in T_2 with probability $\delta/256$. This means the probability of $Q_K(P, l, g) = 0$ is greater than $(1 - \delta/256)^{3+35}$. Meaning that about $\log \delta^{-4} / \log(1 - (\delta/256)(1 - \delta/256)^{38})$ samples are needed to eliminate all wrong candidates $\tilde{k_0}, \tilde{k_5}, \tilde{k_{10}}$ and $\tilde{k_{15}}$. This amounts to 2056 samples when $\delta = 16$. The same procedure is executed for the other

three indices ($y_1[1], y_2[0]$ and $y_3[3]$), to obtain the least significant bits of the other undetermined key bytes.

Extracting Measurement Scores

In a practical attack the ideal predicate Q is approximated by taking the average value of a large number of samples of the form (P, g, m). Here P represents a random plaintext, g represents the LUT index and m represents the measurement score. The large number of measurement samples from a score distribution $M_K(P, l, g)$ (which approximates $Q_K(P, l, g)$).

Assuming the knowledge of the memory address of each LUT T_l, the authors of [21] can determine the cache address to which each table is mapped, denoted by $V(T_l)$. Using this address together with a given table T_l, an index byte g into that LUT and a random plaintext P, the procedure by which the measurement samples are obtained is described by the following three steps:

1. Trigger an encryption of P, which will load all the LUT memory blocks needed for the encryption of this plaintext, into the cache memory.

2. Access some W main memory addresses, at least B bytes apart, that are congruent to $V(T_l) + g \cdot B/\delta($ mod $S \cdot B)$. Since this step accesses W blocks and the CPU has a cache with associativity W, it will evict the contents of the memory block corresponding g in T_l.

3. Trigger a second encryption of P. The measurement score is then equal to the time difference before and after this encryption process.

The last step in the previous procedure uses a timing channel to extract information about the key in the following way. If the encryption of plaintext P under the encryption key K accesses the memory block of index g in T_l then this memory block will have to be re-fetched from main memory into the cache. Hence producing a cache miss with a noticeable delay with respect to the case where this memory block retrieval does not occur.

The experimental results in [21] show a successful attack against OpenSSL version 0.9.8 on an Athlon 64 platform can be performed in about half a minute of continuous measurement, using about 500,000 samples. The authors of [21] note the strong dependency of this measurement procedure on the timing channel, and add that this is a weakness with respect to attacking typical services such as a Virtual Private Network or Linux `dm-crypt` and `cryptoloop` services. The main reason being the timing noise (introduced by instruction scheduling, conditional branches, etc) generated by these services. To overcome this problem, the same paper [21] introduces another measurement procedure, less dependent on timing channels, which will not be presented here, since it is out of the scope of this paper. However, it is important to note that cache based attacks that make use of the shared memory channel are stronger than cache timing attacks. Such attacks have been also successfully executed against public key cryptographic ciphers such as RSA [24].

3.3 Micro-architecture Attacks

Reset and Initialization attacks can be thwarted for most ciphers which use small size LUTs by simply loading these LUTs before encryption starts. This countermeasure was proposed in [17, 4] and it would eliminate cold start misses and diminish the number of conflict misses in case the LUTs are much smaller than the size of the cache. However, as noted by Bernstein [2], this does not eliminate the timing variations due to the inputs of the encryption process.

Beside the rare occurrence of conflict misses, these timing variations are the result of the modern way cache memory is designed to handle multiple accesses per clock cycle on superscalar processors. For instance on X86 CPUs the cache is split into several independently addressed cache banks [31]. Multiple simultaneous references to the same cache bank produce conflict penalties which affect and are affected by the whole micro-architecture of the CPU (e.g. pipeline, load/store queue).

3.3.1 Canteaut's Attack against AES

The downside of this class of attacks is not only the fact the the micro-architecture of a CPU is a complex system which is difficult to model, but also the poor documentation offered by vendors. Nevertheless, Canteaut *et al* [8] managed to successfully execute such an attack on the original implementation of AES [25], by extending the attack from [2] to exploit timing variations of individual bits of the key instead of whole bytes.

Bernstein's attack [2] is not effective if the four LUTs are loaded into cache before encryption starts because cold start misses are eliminated and the number of conflict misses is limited. Hence the micro-architecture attack presented in [8] requires a larger number of samples (2^{30}) in the profiling phase. The LUT indices in the first round of the AES encryption are denoted by a_i. This represents a XOR-ing between the i-th plaintext byte and the corresponding key byte: $a_i = p_i \oplus k_i$. The attack investigates the correlation between the encryption time and the values of all fixed sets of the bits of $a_i = (a_{i,0}, ..., a_{i,7})$. The average encryption time is computed for all the 256 possible values of a_i. Then the bits $a_{i,l}$ with $l \in \{0, ..., 7\}$ for which the encryption timings are mostly high are determined with respect to a chosen threshold. The rest of the bits which do not pass the threshold remain undetermined.

The following part of the attack in [8] uses the non-elimination table attack described in [30] for determining the plaintext bits which provide the highest encryption time. This is done by applying the same procedure as done in the previous step for the bits of a_i, on the bits of the plaintext bytes p_i. In the final step of the attack the key bits are recovered by $k_{i,l} = a_{i,l} \oplus p_{i,l}$

Implementation	Number of predicted bits
Original AES implementation [25]	75 bits
OpenSSL [27]	20 bits

Table 4: Results of Micro-architecture Attack in [8]

CPU family/ model/ stepping	Number of predicted bits
15/4/1	17 bits
15/3/3	4 bits
15/2/4	55 bits
15/2/7	80 bits
15/2/9	75 bits
15/2/5	78 bits

Table 5: Evaluation of Micro-architecture Attack against original AES code on Pentium 4 CPUs in [8]

for the all the positions (i, l) which led to a prediction in the first stage. An exhaustive search is done on the remaining undetermined key bits.

The number of bits for which the value of $a_{i,l}$ was predicted during the first phase is presented in Table 4 for two implementations that were tested in [8] on a Pentium III processor. The significant difference between these results is due to the fact that the original AES implementation [25] uses unaligned memory accesses, which increases the number of conflict misses and conflict penalties. Hence the encryption timings are better differentiated between different input bits. The same attack was also executed against the original AES implementation [25] on different models of Pentium 4 CPUs as shown in Table 5. These results clearly show the strong dependency of the attack on the micro-architecture. Furthermore, the authors of [8] note that the performance of the micro-architecture attack strongly depends on the optimizations made by the compiler. In this direction they tested the same original implementation of AES [25] on a Pentium III CPU using different options of the gcc version 3.2.2 compiler. The results of this test are presented in Table 6 and they show a clear degradation of the number of predicted bits with respect to additional compiler optimizations.

4 Countermeasures

The vast majority of papers about cache timing attacks presented in this work offer several countermeasures. There is no generally applicable solution for preventing all types of attacks presented previously. The choice between different countermeasures strongly depends on the architecture and the application. This section presents a synthesis of these countermeasures together with their different trade-offs.

Compiler Option	Number of predicted bits
none	95 bits
-O3	80 bits
-O9 -mcpu=pentium3 -march=pentium3	45 bits

Table 6: Evaluation of Micro-architecture Attack in [8] with different compiler optimizations

Avoid LUT-based Implementations

All attacks exploit the effect of memory access on the cache and would be mitigated by cipher implementations that do not perform table lookups using key information. This countermeasure has the downside that it implies a considerable performance degradation, which is unacceptable for most applications. Moreover, eliminating LUTs does not also imply eliminating general timing attacks, as demonstrated in the attack by Koeune and Quisquater [16]. Nevertheless, there exist cipher implementations [5] which employ a description in terms of bitwise logical operations, and vectorize these operations across wide registers.

Data-Oblivious Memory Accesses

As opposed to eliminating LUTs, this solution requires that the memory access pattern is oblivious to data passing through the cipher. A naive approach would be to read all entries of the relevant table in fixed order, and use just the entries that are needed. This would imply a significant slowdown of the algorithm.

Goldreich and Ostrovsky [11] offer a generic program transformation for hiding memory accesses. It is theoretically valid, however it implies time and memory size overheads that are not acceptable for most applications. Zhuang *et al* [32] proposed a practical solution which is more efficient, however it requires complex hardware support in the processor.

Loading LUTs a priori

Since reset attacks and initialization attacks are based on time delays caused by cold start cache misses, these classes of attacks could be prevented by loading all LUTs used by the cipher in the cache before encryption starts as proposed in [21, 17]. Moreover, cache conflict misses may also be eliminated if the LUTs are carefully allocated such that none of their entries map to the same cache address. However, this does not imply that all timing variations are eliminated, as pointed out by Bernstein [2]. As presented in Section 3.3, the remaining timing variations are due to the whole micro-architecture of the CPU, which can also lead to timing attacks such as the one by Canteaut *et al* [8].

Table Storage Artifices

Another measure for preventing an attacker from learning the secret key from cache timing attacks is to disassociate the memory access patterns. A solution proposed in [21] is to use multiple copies of each LUT placed at various offsets in memory and have table lookups use a randomly chosen LUT. A similar approach is to use a single copy of each LUT which would be moved randomly in memory during encryption. A

random permutation approach can also be applied at LUT entry level. However, these methods would induce a performance overhead which might not be acceptable in some applications.

Canteaut *et al* [8] argue that the same idea can be executed without an important computational overhead if the permutation is a translation. They use a randomly chosen 4-byte mask applied to each word of the internal state of the AES at the end of the key scheduling procedure. By changing the mask once every 256 encryptions, this method allows re-mapping of the LUTs with an execution time overhead of only 5%. This performance degradation is said to be an acceptable price for the fact that this approach can eliminate cache timing attacks of all types.

Hiding the Timing

Since all the attacks presented in this paper make use of timing information, they could easily be thwarted by its absence. Osvik *et al* [21] propose adding random delays to the observed operations. Canteaut *et al* [8] suggest normalizing all operations to the worst case execution time. Both of these solutions may imply excessive time penalties which are not acceptable to most applications.

Selective Round Protection

All the attacks presented in Section 3 analyze memory accesses in specific rounds. To protect against such attacks it suffices to protect the fist and last couple of encryption rounds by any of the means described before. The other remaining rounds can use the efficient LUT based implementation. This will not cause such an excessive performance degradation as applying a countermeasure on all rounds. However, differential cryptanalysis attacks [7] can still be applied on internal rounds, but their complexity is higher.

5 Conclusions

This paper has presented several cache-based timing-channel attacks against cryptographic ciphers. These attacks take advantage of the timing information leaked during the encryption process by the cache memory behavior. This timing information may be obtained using simple software tools that utilize the CPU time stamp counter [14]. Hence, making these attacks very attractive from the financial point of view of the attacker.

An overview of the attacks presented in this paper is shown in Table 7. Every attack is characterized by some of the most important aspects such as the type of cache misses the attack benefits from. This can be seen from the type of attack. The cipher, implementation and the platform on which the attacks were executed are also important since they have a strong influence on the number of predicted bits. The running time complexity of an attack can be estimated from the number of samples it needs. The attack by Osvik *et al* [21] stands out from the performance point of view, however it is limited to multi-user systems where the attacker can trigger the cache memory. The 2 round attack presented by Neve *et al* [20] closely resembles the performance of the former attack, and its applicability is not as constrained as the former attack. The important unknown of this attack is the platform on which it was executed. The other attacks against AES need a relatively higher number of samples. Therefore they require long execution times that might impede their practicality since most symmetric key crypto systems change their secret key about every hour as pointed out in [26].

The timing variations in encryption timings exploited by cache attacks are due to cache misses, and also other micro-architectural aspects of the CPU. Cache timing attacks are mainly focused against cryptographic cipher implementations which are based on LUTs. This kind of LUT-based cipher implementations are very frequent since they offer better performance than other variants. The major downside of most of these implementations is that they use plaintext input- and key-dependent lookups. Heuristically, large LUTs increase the effectiveness of all attacks. In ciphers like the AES [9] the data-dependency lookups are very difficult to eliminate as pointed out by Bernstein [2]. Nevertheless, there exist several published ciphers [10, 3] which are built from a few simple operations that take constant time on common general-purpose CPUs. In order to completely avoid the types of attacks presented in this paper, one is urged to choose such a cryptographic cipher.

Nevertheless, the fact that a cryptographic cipher is vulnerable to cache timing attacks may not be applicable to some systems using them. If the system refuses to encrypt any plaintext given by an attacker (i.e. the attacker is not authorized to perform encryption) then none of the previously presented attacks can be executed. Furthermore the attacker also needs to be able to obtain precise timings for each encryption. This limits the remote application of such attacks as pointed out in [20]. However, in the case where users can perform and precisely time encryptions, applications like web browsing, DRM and even virtual machines are vulnerable to cache timing-attacks, as pointed out in [21].

If for any particular reason one requires the use of a LUT-based cipher implementation. The following steps should be performed to protect against timing attacks. First, one should evaluate the vulnerability of the chosen cipher implementation against time-driven cache attacks using the analytical model presented by Tiri *et al* [28]. The analytical model accurately forecasts the strength of a symmetric key crypto system based on a few simple parameters that describe the adversary's observation capabilities, the implementation, and the platform the algorithm is running on.

Based on the results from the first step the system or application designer or implementer needs to decide on possible countermeasures. A series of countermeasures

Attack Author(s)	Type of Attack	Platform	#Samples	Cipher	Implementation	Predicted bits
Tsunoo *et al* [30]	Reset	Pentium III, 16 KB cache size, 32 B cache line	2^{25}	DES	own	32/56
			2^{26}	AES	original [25]	96/128
Bernstein [2]	Reset	Pentium III, 16 KB cache size, 32 B cache line	2^{27}	AES	OpenSSL v 0.9.7a	91/128
Neve *et al* [20]	Reset	Not specified	2^{20}	AES	OpenSSL	125/128
Osvik *et al* [21]	Initialization	Athlon 64, 64 KB cache size, 64 B cache line	$< 2^{19}$	AES	OpenSSL v 0.9.8	128/128
Canteaut *et al* [8]	Micro-architecture	Pentium III, 16 KB cache size, 32 B cache line	2^{30}	AES	OpenSSL v 0.9.7i	20/128
					original [25]	75/128

Table 7: Overview of Cache Timing Attacks

were also presented, although most of them suffer from performance degradations which might not be acceptable in certain systems or applications.

References

[1] Kazumaro Aoki, Tetsuya Ichikawa, Masayuki Kanda, Mitsuru Matsui, Shiho Moriai, Junko Nakajima, and Toshio Tokita. Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms - Design and Analysis. In *Proceedings of the 7th Annual International Workshop on Selected Areas in Cryptography*, SAC '00, pages 39–56, London, UK, UK, 2001. Springer-Verlag.

[2] Daniel J. Bernstein. Cache-timing attacks on AES. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, 2005.

[3] Daniel J. Bernstein. The Salsa20 Family of Stream Ciphers. In Matthew J. B. Robshaw and Olivier Billet, editors, *The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 84–97. Springer, 2008.

[4] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES Power Attack Based on Induced Cache Miss and Countermeasure. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I - Volume 01*, pages 586–591, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Eli Biham. A Fast New DES Implementation in Software. In *Proceedings of the 4th International Workshop on Fast Software Encryption*, FSE '97, pages 260–272, London, UK, 1997. Springer-Verlag.

[6] Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. In *Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '90, pages 2–21, London, UK, UK, 1991. Springer-Verlag.

[7] Andrey Bogdanov, Thomas Eisenbarth, Christof Paar, and Malte Wienecke. Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs. In Josef Pieprzyk, editor, *Topics in Cryptology - CT-RSA 2010*, volume 5985 of *Lecture Notes in Computer Science*, pages 235–251. Springer Berlin / Heidelberg, 2010.

[8] Anne Canteaut, Cédric Lauradoux, and André Seznec. Understanding cache attacks. Research Report RR-5881, INRIA, 2006.

[9] Joan Daemen and Vincent Rijmen. AES Proposal: Rijndael. `http://www.cryptosoft.de/docs/Rijndael.pdf`, 1998.

[10] Niels Ferguson, Doug Whiting, Bruce Schneier, John Kelsey, Stefan Lucks, and Tadayoshi Kohno. Helix: Fast Encryption and Authentication in a Single Cryptographic Primitive. In Thomas Johansson, editor, *FSE*, volume 2887 of *Lecture Notes in Computer Science*, pages 330–346. Springer, 2003.

[11] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43:431–473, May 1996.

[12] Mark D Hill. Aspects of Cache Memory and Instruction. Technical report, Berkeley, CA, USA, 1987.

[13] Wei-Ming Hu. Lattice Scheduling and Covert Channels. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, SP '92, pages 52–, Washington, DC, USA, 1992. IEEE Computer Society.

[14] Intel. Using the RDTSC Instruction for Performance Monitoring. Technical report, Intel Corporation, 1997.

[15] Paul 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, pages 104–113, London, UK, UK, 1996. Springer-Verlag.

[16] Francois Koeune and Jean jacques Quisquater. A timing attack against Rijndael. Technical report, Université catholique de Louvain, 1999.

[17] C. Lauradoux. Collision attacks on processors with cache and countermeasures. In *Western European Workshop on Research in Cryptology - WEWoRC 2005*, Lecture Notes in Informatics 74, pages 76–85, Leuven, Belgique, july 2005. Bonner Kollen Verlag.

[18] Mitsuru Matsui. New Block Encryption Algorithm MISTY. In Eli Biham, editor, *FSE*, volume 1267 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 1997.

[19] National Institute of Standards and Technology. Data Encryption Standard. FIPS Publication 46-2, December 1993.

[20] Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. A refined look at Bernstein's AES side-channel analysis. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, ASIACCS '06, pages 369–369, New York, NY, USA, 2006. ACM.

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

[22] D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. Technical Report CSTR-02-003, Department of Computer Science, University of Bristol, June 2002.

[23] David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[24] Colin Percival. Cache missing for fun and profit. `http://www.daemonology.net/papers/htt.pdf`, 2005.

[25] Vincent Rijmen, Antoon Bosselaers, and Paulo Barreto. Optimised ANSI C code for the Rijndael cipher (now AES). `ftp://ftp.netbsd.org/pub/NetBSD/NetBSD-current/src/sys/crypto/rijndael`, 2000.

[26] Robert G. Salembier. Analysis of Cache Timing Attacks against AES. `http://teal.gmu.edu/courses/ECE746/project/F06_Project_resources/Salembier_Cache_Timing_Attack.pdf`, 2006.

[27] The OpenSSL Project. OpenSSL. `http://www.openssl.org`, 1998.

[28] Kris Tiri, Onur Aciiçmez, Michael Neve, and Flemming Andersen. An Analytical Model for Time-Driven Cache Attacks. In Alex Biryukov, editor, *FSE*, volume 4593 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2007.

[29] Wing N. Toy and Benjamin Zee. *Computer Hardware-Software Architecture*. Prentice Hall Professional Technical Reference, 1986.

[30] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES Implemented on Computers with Cache. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2779 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2003.

[31] K.M. Wilson and K. Olukotun. High bandwidth on-chip cache design. *Computers, IEEE Transactions on*, 50(4):292 –307, April 2001.

[32] Xiaotong Zhuang, Tao Zhang, Hsien-Hsin S. Lee, and Santosh Pande. Hardware assisted control flow obfuscation for embedded processors. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '04, pages 292–302, New York, NY, USA, 2004. ACM.