AES Power Attack Based on Induced Cache Miss and Countermeasure

Guido Bertoni, Vittorio Zaccaria
STMicroelectronics, Advanced System Technology
Agrate Brianza - Milano, Italy,
{guido.bertoni, vittorio.zaccaria}@st.com

Luca Breveglieri, Matteo Monchiero, Gianluca Palermo
Politecnico di Milano, Dipartimento di Elettronica e Informazione
Milano, Italy
{breveglieri, monchiero, gpalermo}@elet.polimi.it

Abstract

This paper presents a new attack against a software implementation of the Advanced Encryption Standard. The attack aims at flushing elements of the SBOX from the cache, thus inducing a cache miss during the encryption phase. The power trace is then used to detect when the cache miss occurs; if the miss happens in the first round of the AES then the information can be used to recover part of the secret key. The attack has been simulated using the Wattch simulation framework and a simple software implementation of AES (using a single table for the SBOX). The attack can be easily extended to more sophisticated versions of AES with more than one table. Eventually, we present a simple countermeasure which does not require randomization.

Keywords: AES, Cache, Power Analysis, Cache Miss, Block Cipher.

1 Introduction

Current processor systems are characterized by a core speed which is an order of magnitude higher than traditional memory speed [5]. This results in a main memory access penalty that can be of hundreds of cycles. To increase the computational power, processors are generally equipped with a cache memory which decreases the memory access latency. Unfortunately caches contain only a small portion of the application data and can introduce additional latency to the memory transaction in the case of a miss. This involves also additional power consumption which is due to the activation of memory devices down in the memory hierarchy.

The miss penalty has been already used to attack symmetric encryption algorithms, especially DES [2, 4, 7], while no such attacks have been published for AES so far. Note that previous attacks based on cache miss used to purge the cache completely, and used to count the number of cache miss executed in an encryption call based on cold cache status. So the attacks can be classified as timing attacks.

In this paper we present an attack which does not involve the complete cache invalidation before the encryption. The aim is to force a limited number of selective cache misses following a fixed rule and identify when the miss appears during the encryption call. We then use power trace to identify if and when a cache miss really occurs (remember that cache misses are abrupt event on the dynamic power profile).

The proposed method is based on the idea of performing a selective cache miss, i.e. replacing a single block of the cache with useless data for the encryption function. In order to do so, the software attack routine declares an array as large as the cache. Every time an element of the array is accessed by the attacker routine, an element of the cache is allocated to it and (most probably) will involve the invalidation of an element of the SBOX. As we will show after, this is due to the fact that different addresses can have the same 'alias' cache block (see Section 3 for a detailed description).

If a miss occurs at a specific index triggered by the attacker routine, then it means that, during the accesses for computing the SBOX, that index has been used. During the computation of the first round the index is a bitwise xor between the secret key and the plain text and since the plain text is known, the secret key can be inferred partially or completely, depending on the configuration of the cache.

As a matter of fact, the secret key cannot be completely reconstructed since, in general, the cache block contains more than one SBOX value; this means that for larger cache blocks, fewer secret key bits are reconstructed. As an example, performing the attack on target architectures with a cache block size equal to 2 words (8 bytes) or 4 words (16 bytes), the secret key can be reconstructed respectively for 80 bits or 64 bits.

The paper is organized as follow. Section 2 presents the AES encryption algorithm, while in Section 3 an introduction of the cache architectures is provided. A detailed description of the proposed attack based on cache misses is presented in the Section 4. In Section 5, the simulation setup used to validate the attack is presented, while in Section 6 the countermeasure is described. Finally Section 7 presents some conclusion and future works.

2 AES

The Rijndael algorithm has been selected to be the Advanced Encryption Algorithm (AES); it is a secret-key (symmetric) block cipher crypto-system [6] which encrypts (or decrypts) one block of data at a time. The encryption algorithm accepts one data block (or plain text) and the key, and produces the encrypted data block (the input and output data blocks are of identical size). The decryption algorithm accepts one encrypted data block and the key, and outputs the plain text. Both encryption and decryption use the same secret key.

Internally, the AES encryption algorithm can be partitioned into two independent processes: Encryption and Key Schedule. In most cases where the AES Encryption process is executed in software, the key schedule is performed in advance and the expanded key is stored in memory. The decryption algorithm is similarly partitioned into the Decryption and Inverse Key Schedule processes. If a small amount of RAM is available, it is possible to generate round keys on the fly, this possibility is not considered in the paper.

The Rijndael algorithm permits all the combinations of key sizes and data block sizes among 128, 160, 192, 224 and 256 bits. NIST however, has restricted the length of the data blocks for the AES algorithm to 128 bits only, while the key size can be chosen as 128, 192 or 256 bits. AES has an iterative structure and works in rounds: a fixed set of operations, called a round, is iteratively applied to the input block of data, called state, the elements of which are bytes. After iterating a predefined number of rounds the state is output.

The number of rounds to be executed is determined by each key size. The use of a key of 128 bits requires 10 rounds, 12 rounds are applied in the case of 192 bits key and 14 rounds are required for the 256 bits key. The input block of data is encrypted by applying the 10, 12 or 14 rounds. Except for the first and the last rounds, the other rounds are identical and consist of four transformations each. A brief overview of the algorithm is given below, but the reader should refer to [6] for a complete description. The four transformations forming each round are called SubBytes, ShiftRows, MixColumns and AddRoundKey. The SubBytes transformation receives as input each element (one byte) of the state and calculates as output its multiplicative inverse in the finite field $GF(2^8)$ followed by an affine transformation. The ShiftRows transformation applies a left rotation to the four rows of the state: no rotation for the 1st row, a rotation of one position (one element) for the 2nd row, of two positions for the 3rd, and of three positions for the 4th. The MixColumns transformation treats each column as a 4-term polynomial, the coefficients of which are defined over the field $GF(2^8)$, and multiplies each column by a constant polynomial, again defined over the field $GF(2^8)$. The AddRoundKey transformation adds the round key to the state. The algorithm starts with an initial transformation composed only by a key addition. The difference between the middle rounds and last one is that the last wound is similar to a normal round except that the MixColumns transformation is omitted. To decrypt an enciphered block, the inverse transformation should be applied, and of course the secret key must be known. The round key is derived from the secret key by means of a special algorithm, called Key Schedule. The basic idea of Key Schedule is that eleven different round keys of 128 bits each, are derived from a single secret key of 128 bits, for a total of 1408 bits of so-called ”key material” or expanded key.

When the AES is implemented in software there are different possibilities, depending on the type of CPU and memory structure available. The diversity of the implementation comes from the SubBytes and MixColumns transformations. The calculation of the multiplicative inverse in $GF(2^8)$ that is part of SubBytes is never computed, but the complete SubBytes transformation is precomputed and stored in memory in a table generally names SBOX. The SBOX requires 256 bytes to be stored. This is the basic software implementation of the AES, that compute all the other operations with the available instructions. But if memory is available it is possible to precompute the T-table, which store the possible values of the SubBytes and MixColumns transformation, thus avoid the computation of part of the MixColumns at the price of a larger memory requirement. In our case, the basic version is the worst case for performing the attack, thus we will consider this case and demonstrate how the attack works in the worst case and commenting the other cases.

To understand how the attack work we summarize the basic operation performed during the first and the beginning of the second round. In the initial transformation the input data are bitwise xored to the secret key, after this every bytes of the state is substituted applying the SubBytes transformation. Practically every byte of the state is intended as the address for reading the SBOX table. Due to the structure of the initial transformation, the address is equal to the xor between each byte of the plain text and each byte of the key.

3 Cache Model

Current processor systems are characterized by a core speed which is an order of magnitude higher than traditional memory speed [5]. This results in a main memory access penalty that can be of hundreds of cycles. Caches are local memories used to decrease the main memory access penalty by providing faster access time to data (or instructions) which are more frequently used. This is achieved by using more expensive but highly performance memory cells coupled with a memory structure which minimizes the amount of wiring (an thus delay).

Cache size is usually much smaller (in terms of capacity) than usual dynamic memory; this is due to the cost which can be a significant part of the overall processor system. For this reason caches contain only a small portion of the application data.

Processor memory access is usually split in two phases. The first phase consists in accessing the cache for retrieving data; this is done usually in one or two clock cycles, but can be more if the processor has a very short clock cycle time. When a processor does not find data within a cache (which is technically called 'miss'), it accesses lower memory levels with an additional cycle penalty which is called 'miss penalty'. Depending on the miss management policy, the processor can decide to discard some of the current data in the cache and refill it with 'missing' data from the lower memory.

More formally, a cache provides an access structure which exploits temporal and spatial locality of the application data. Temporal locality property holds when there is a high probability that if data at a specific address is used at time t then it will be used also at time $(t+\delta)$, where δ is a small amount of time.

Spatial locality property holds when there is a high probability that if data at address a is accessed at time t then data at address $(a+\omega)$ is accessed in $(t+\delta)$ where ω and δ are respectively a small offset and a small amount of time.

Caches store data by means of blocks (also called lines) whose size is in the order of tens of bytes. Each cache block is accessed by means of an index which a subportion of the address of the block. Caches can store, for the same index, a number of blocks which is indicated by the associativity of the cache. If we assume a direct mapped cache (associativity equal to 1) then the index is computed as $i = b_{address} MOD(C_{size}/C_{blocksize})$ where $b_{address}$ is the block number within the memory, C_{size} is the cache size and $C_{blocksize}$ is the block size . Since many blocks can have the same index, when a cache is operating on data it should check if the address of the block positioned at the requested index is equal to the address of the requested block. This is done by means of a companion information which is called 'tag' and contains the remaining part of the address of the block currently stored in the cache. The tag structure is an additional memory which is accessed at the same time as the block. When the tag of the block within the cache does not correspond to the tag of the requested block (i.e., a 'miss' happens) then the block is to be replaced before the data can actually be used by the processor. This transaction usually lasts for several cycles depending on the frequency gap between the core speed and the main memory speed. If the data block within the cache was modified (or 'dirty') then it should be written back before reading the new block. For this reason, every time we have a miss, we can have several transactions between the cache subsystem and the memory subsystem which, a part from performance penalty, produce also electrical energy loss.

4 Cache Miss Attack

The basic idea proposed in this paper is to force a cache miss while the processor is executing the AES encryption algorithm on a known plain text.

In order to perform the cache miss attack, we make the following assumption: the attacker knows the cache architecture of the target processor in terms of cache size, block size and associativity. This is a reasonable hypothesis since this information is relatively simple to be gathered from the public processor datasheet.

In Subsection 4.1 we present an outline of the proposed attack. For the moment we assume that the cache is direct mapped and the block size is equal to one byte. Later (see sections 4.2 and 4.3) we will remove these constraints by showing how the attack can be performed with higher associativity and larger block sizes.

4.1 Attack Outline

The attack configuration is composed by an attack software routine (running in user mode) which is able to perform a series of calls to a cryptographic API (running in supervisor mode, or in another process).

Before starting the attack, the attacker routine needs to call once the encryption routine and then initialize an array A_m as large as the cache size. Since the cache is direct mapped, a successive read of an element of the array involves a replacement of the corresponding cache block.

The attacker then calls again the cryptographic API. If a miss occurs during the first 16 memory accesses for computing the SBOX, then it means that the cache block index triggered by the attacker routine has been used also for accessing an element of the SBOX. If the attacker knows the

Table 1. Simplescalar configuration parameters used for the simulations

Parameter	Value
Fetch width	1
Branch penalty	3
Branch predictor	Bimodal (2048 entries)
Decode width	1
Issue width	1
Inorder	true
L1 DCache	4 KB (One-way, 8-byte block)
L1 ICache	16 KB (One-way, 32-byte block)
Unified L2 Cache	256 KB (4-way, 64-byte block)

offset of the SBOX table within the cache then he can infer the SBOX index from the cache block index. Since the index is a bitwise xor between the secret key and the plain text and since the plain text is known, the secret key can be inferred partially or completely, depending on the configuration of the cache.

The attack is organized as follow:

1. The encryption function is called without monitoring the power consumption. This first call is used to fill the cache with the data structure involved in the attack. At the end of the first encryption the cache contains all the elements of the SBOX.

2. The attacker reads one element of the array A_m to replace a single element of the SBOX in a known position.

3. The encryption is called again executed using the same plain text of pass 1. During this phase, power consumption is monitored. If a cache miss occurs during the first round, this indicates that a byte of the secret key xored to the plain text has accessed the SBOX element replaced by the attacker.

In order to discover all the bits of the secret key, these three steps should be iterated for all the SBOX elements (256). Therefore, the number of encryptions required to perform the attack is equal to 2×256, while the number of power trace is 256.

As can be intuitively noted, if the cache block is one byte then this attack allows to reconstruct the complete secret key.

4.2 Cache Block Size

In previous section we assumed that the cache block size was equal to one byte. This is a restrictive hypothesis because typically data cache block size is 8 bytes or 16 bytes [5]. Larger cache blocks reduce the amount of information that a miss attack can recover from a cache miss.

As an example, consider an SBOX is implemented as an array of bytes and a target processor supporting byte-wise operation and a cache block of 8 bytes. In this case, the cache block contains 8 SBOX elements and a cache miss indicates that the encryption algorithm has accessed one of those 8 elements. This scenario does not permit to understand which element has been read out of the elements in the block. In this case, the attack is capable to reconstruct only the 5 most significant bits of each byte of the secret key. The attack reduces the key space considerably, from 128 bits to 48 bits. There is even an advantage in term of complexity of the attack, with a cache block size equal to 8 bytes it is necessary 2*32 encryption calls and 32 power traces.

Some platforms supports only word-wise operation so the SBOX is declared as an array of 32-bit words. In this case the proposed attack performs well since more bits of the secret key can be recovered. As an example, if the block size is equal to 8 bytes, the number of bits that can be discovered is 112 out of 128.

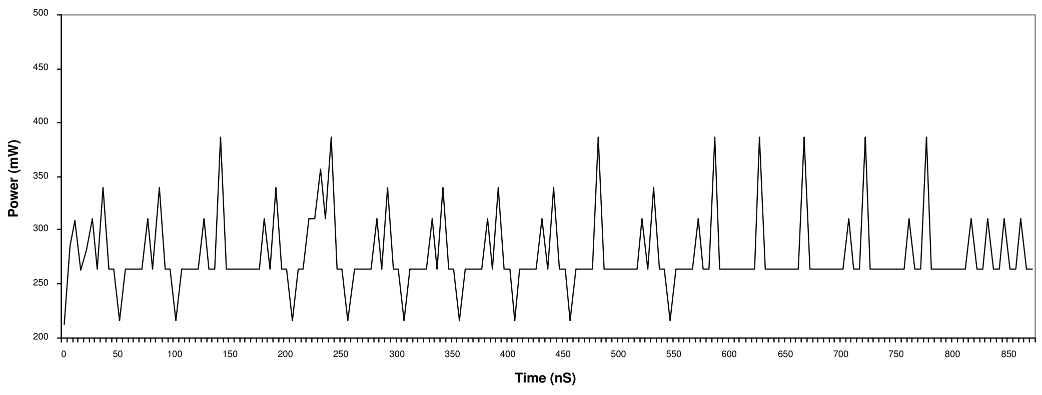
4.3 Cache Associativity

The removal of the direct mapped cache constraint does not affect the attack capability (number of key-bits that can be discovered). In order to perform the attack for set associative caches, the attacker should use a number of arrays A_m equal to the cache ways. With respect to the outline presented in the Subsection 4.1, the attack has to be modified. After the first encryption call, the attack routine should trigger a cache miss by reading the element in position j within all the arrays.

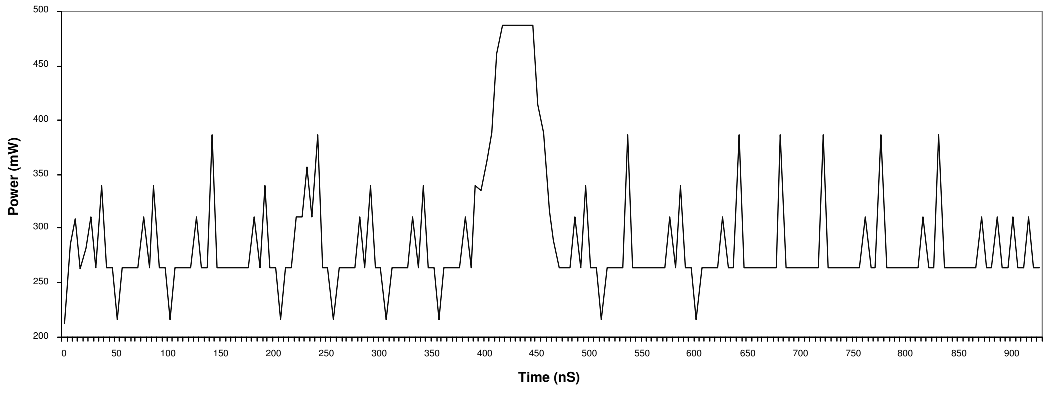
If the cache replacement policy is different than LRU or is unknown, the attacker should read two or more times the target elements of the arrays A_m, in order to raise the probability of purging an SBOX element. The number of encryption calls and power trace to be collected is not increased by the cache associativity.

4.4 More Practical Problems

To perform the attack the location of the SBOX should be known. During an encryption call the AES access only four types of data: the input data, the output data, the expanded key and the SBOX. To discover the offset at which these structures are stored in the cache, the attacker can induce selective cache misses in the same way as presented for the attack. The size of the four data structures used by the algorithm varies with the data structure and the size of the SBOX is the biggest one.



(a) Without cache miss



(b) With cache miss during the 8-th cache access

Figure 1. Power dissipation profile of the AES first round

It can also happen that two bytes of the secret key are equal. This situation can be identified by observing that the number of induced miss in the first round is lower than the number of key-bytes. To detect the secret key, it is enough to change the plain text and run again the attack.

5 Test conducted

In order to validate the attack methodology, we conducted several experiments using the Simplescalar-3.0 toolset [1]. Simplescalar is a framework for cycle-accurate microarchitecture modeling and simulation. We configured the simulator to model a MIPS-like Instruction Set and a 5-stage single-issue microarchitecture. Main parameters of the model are indicated in Table 1.

Software power is determined using power models carried out from Power Analyzer [3], a library of power models designed to be integrated in Simplescalar. The model allowed us to estimate the cycle-by-cycle power consumption associated of the target architecture while it is running the AES algorithm.

The simulation setup, adopted for the validation of the attack, is a framework widely used to analyze and optimize microprocessor power dissipation at the architectural level. The experimental results has been carried out using the configuration described in Table 1. We configure the Data Cache parameter as an 8-byte for block cache. In this way, by using the proposed attack methodology, we can discover 80 bits out of the 128 bits of the secret key. However, the validity of the method does not depend on a particular configuration. We have tested the method on different cache organizations, obtaining the results described in the previous section.

Figure 1 shows two different power dissipation profiles of the target processor while it is running the first round of the AES algorithm. In Figure 1(a), 16 peaks corresponding to the memory load instructions which read Sbox can be observed. In this scenario, the Sbox has been already stored in the Data Cache, so the power dissipated by each load operation is relatively small. It is due only to a cache access.

Figure 1(b), shows the power profile of the first round when previously a block of the cache, where the SBox was stored, has been overwritten. In this scenario we can discover which element of the Sbox has got dirty and so its address. As previously described this information provides us to know some bits of the secret key. A miss can be detected by observing the power dissipation peak when the 8-th load occurs. In fact, the power value of the 8-th load is due not only to the L1 cache but also the the L2 cache access.

6 Countermeasure

In this section we propose a simple countermeasure against the proposed attack. Since the proposed attack focuses attention on the presence of cache miss, the countermeasure has to avoid cache misses related to the computation of the encryption algorithm. In this case, no information about the secret key can be discovered and the attack does not perform.

The countermeasure is based on the insertion of a set of SBOX accesses in the encryption function before the real computation starts. The minimum number of cache accesses needed to avoid the attack depends on the knowledge that the developer has of the target architecture. If he is not aware of the target cache structure, he has to read all the elements of the SBOX. Otherwise, he can force to read only one element of the SBOX for each cache block where the SBOX table is stored. As an example, having a cache block of 8 bytes, the countermeasure requires only 32 memory accesses.

It can be noted that the cost of this countermeasure consist only in the number of memory access. The countermeasure insert no additional cache miss since, when the AES algorithm is executed, the SBOX has been already totally stored in the cache. Also note that this countermeasure is not specific to AES, it can be used to protect all block ciphers against timing or power attacks based on cache miss.

7 Conclusion and Further Research

The paper presents a novel methodology for attacking encryption algorithm implemented on a system with cache memory. The attack is quite simple and allows to recovery part of the secret key of the AES. the amount of bits extracted depend on the system configuration. The attack has been presented and tested against the AES but it can be used against other block ciphers and in a future development even against public key algorithm. A simple countermeasure against the attack has been proposed, there is no need of randomization.

References

[1] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, 1996.

[2] D. W. J. Kelsey, B. Schneier and C. Hall. Side channel cryptanalysis of product ciphers. In *In 5th European Symposium on Research in Computer Security*, page 97110, 1998.

[3] N. S. Kim, T. Austin, T. Mudge, and D. Grunwald. Challenges for architectural level power modeling. In R. Melhem and R.Graybill, editors, *Power-Aware Computing*. 2001.

[4] 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.

[5] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[6] U.S. Department of Commerce/National Institute of Standard and Technology. *FIPS PUB 197, Specification for the Advanced Encryption Standard (AES)*, November 2001. Available at http://csrc.nist.gov/encryption/aes.

[7] T. S. M. S. Y. Tsunoo, T. Saito and H. Miyauchi1. Cryptanalysis of DES Implemented on Computers with Cache. In *Workshop on Cryptographic Hardware and Embedded Systems*, page 6269, 2003.