Pinpointing Cache Timing Attacks on AES

Chester Rebeiro, Mainack Mondal, and Debdeep Mukhopadhyay

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur, India

{*chester, mainack, debdeep*}*@cse.iitkgp.ernet.in*

Abstract

The paper analyzes cache based timing attacks on optimized codes for Advanced Encryption Standard (AES). The work justifies that timing based cache attacks create hits in the first and second rounds of AES, in a manner that the timing variations leak information of the key. To the best of our knowledge, the paper justifies for the first time that these attacks are unable to force hits in the third round and concludes that a similar third round cache timing attack does not work. The paper experimentally verifies that protecting only the first two AES rounds thwarts cache based timing attacks.

1 Introduction

The biggest threat to the security of embedded systems is from side channel attacks. These attacks were initially discovered by Kocher [8] and were then used to retrieve the secret key of a cryptographic algorithm running on small embedded devices such as smart cards. An embedded device could have several side channels which leak information about the key. These channels include power consumption, electro magnetic radiation, timing etc. Of these, timing attacks are the cheapest to perform as they do not require sophisticated measuring instrumentation. In this paper we focus on preventing timing attacks for software implementations of cryptographic algorithms.

The main source of leakages in a cryptographic algorithm implementation is from the processor's cache memory. Cache memory is present on almost every computing system and makes use of the temporal and spatial locality of the code to improve the overall execution time of the program. The time required to access data present in the cache is much lesser than the time required to access data stored in memory. This differential timing for memory access is used by attackers to gain knowledge of the secret key of a cryptographic algorithm. Such attacks are known as *cache attacks*. Algorithms such as AES [7] that use large key dependent lookup tables are most vulnerable to such attacks.

Based on the capabilities of the adversary cache attacks can be categorized into two, namely cache-time and cache-trace attacks. In a *cache-trace* attack, the adversary is assumed to be able to capture the profile of the cache activity during the encryption. The profile includes the outcomes of every memory access the cipher issues in terms of cache hits and misses. In a *cache-time* attack, the attacker derives the key from just the knowledge of the time required for encryptions. No additional information about the individual hit-miss pattern in cache access is required. This makes time-driven attacks more deadly compared to trace-driven attacks. Using a time-driven attack, it has been shown that a remote server running an encryption algorithm can be broken over the network [1, 2].

The easiest way to prevent cache attacks is by avoiding table lookups. This requires sboxes to be computed using functions. The only efficient method to implement such sboxes is by using bitslicing. In a bitsliced implementation [9, 13], several encryptions can be done in parallel on a single processor. However bitslicing is limited to operating modes that do not require chaining. For operating modes requiring chaining, table based implementations are a better solution although they are vulnerable to cache attacks. There are several countermeasures that have been proposed for table based implementations such as in [2, 6, 11]. All these countermeasures suffer from a similar drawback which is the overhead on the performance of the algorithm. This overhead drastically slows down encryptions making such methods impractical for high speed applications. Hence it is desirable to incorporate countermeasures with minimum overhead. However cache accesses being statistical and dependent on data is extremely hard to analyze. In fact there has been insignificant research to exactly pinpoint the cause of cache attacks. In this paper we justify the cause of cache based timing attacks on AES by analyzing the cipher structure. We then argue that to prevent timing attacks on AES, it is sufficient to just protect the first two rounds of the cipher instead of protecting all rounds.

In order to experimentally verify the result, we implement the first two rounds using composite field sboxes [5]. We justify our claim by showing that such an implementation cannot be attacked by observing the timing of the cipher.

The paper is structured as follows : Section 2 has the background for the work. Section 3 has related works on countermeasures for cache attacks. Section 4 analyzes the AES structure and presents the causes for information leakage in AES. It also reasons why a third round cache timing attack is infeasible. Section 5 proposes a new countermeasure based on an analysis of the AES structure. Section 6 has experimental results showing the overhead of a popular countermeasure (composite field sboxes) on the cipher's performance and compares it with the countermeasure proposed in the paper. The final section has the conclusion.

2 Background

The present section gives an overview of a popular implementation technique for the AES block cipher [12]. Subsequently we brief an existing work on cache based timing attacks against AES [2].

2.1 AES Block Cipher

AES-128 [7] is a 10 round cipher which takes as input a 16 byte plaintext $P = (p_0, p_1, \cdots, p_{15})$ and a 16 byte secret key $K = (k_0, k_1, \cdots, k_{15})$. The most widely used software implementation of AES is based on Barreto's code [12]. This performance optimized implementation uses four $1KB$ lookup tables T_0, T_1, T_2, and T_3 for the first 9 rounds of the algorithm, and an additional $1KB$ lookup table T_4 for the final round. The structure of each round is shown in Equation 1 and encapsulates the four basic AES operations of *SubByte*, *ShiftRow*, *MixColumn*, and *AddRoundKey*. The input to round i ($1 \leq i \leq 9$) is the state S^i comprising of 16 bytes $(s_0^i, s_1^i, \cdots, s_{15}^i)$ and round key K^i split into 16 bytes $(k_0^i, k_1^i, \cdots, k_{15}^i)$. The output of the round is the next state S^{i+1}. The first round S^1 comprises of inputs $(P \oplus K)$ and round key K^1.

$$\begin{aligned} S^{i+1} = \{&T_0[s_0^i] \oplus T_1[s_5^i] \oplus T_2[s_{10}^i] \oplus T_3[s_{15}^i] \oplus \{k_0^i, k_1^i, k_2^i, k_3^i\}, \\ &T_0[s_4^i] \oplus T_1[s_9^i] \oplus T_2[s_{14}^i] \oplus T_3[s_3^i] \oplus \{k_4^i, k_5^i, k_6^i, k_7^i\}, \\ &T_0[s_8^i] \oplus T_1[s_{13}^i] \oplus T_2[s_2^i] \oplus T_3[s_7^i] \oplus \{k_8^i, k_9^i, k_{10}^i, k_{11}^i\}, \\ &T_0[s_{12}^i] \oplus T_1[s_1^i] \oplus T_2[s_6^i] \oplus T_3[s_{11}^i] \oplus \{k_{12}^i, k_{13}^i, k_{14}^i, k_{15}^i\}\} \end{aligned} \quad (1)$$

2.2 Bernstein's Cache Timing Attack

In 2005, Bernstein demonstrated a cache timing attack on a remote server running Barreto's implementation of AES [2]. The attack required $2^{27.5}$ plaintext samples generated randomly from a client machine. Initially all the bytes of key (K_T) on the server are set to zero, and for every key byte k_{T_i} the average time for encryption is obtained for all possible values (0 to 255) of the plaintext byte p_i. This forms the template for the known key byte k_{T_i}. Next, the process is repeated with the unknown key K and a similar timing profile for every key byte k_i is obtained. The two profiles are statistically correlated to reveal the unknown key byte k_i.

3 Related Work

Various countermeasures have been proposed over the years to provide protection against cache timing attacks [2, 6, 11]. All proposed countermeasures are based on the primary motivation which is to stop or randomize the information leaked by the cache access patterns of the AES implementation. These countermeasures include:

- Using random permutations to mask the information about actual cache access patterns [3, 4].
- Using smaller and compact tables for encryption.
- Ensuring constant time for AES encryptions by using dummy functions of known latency [6].
- Architectural and operating system modifications.

The common drawback of the first three countermeasures is the overhead on the encryption time. This is generally not acceptable especially for high speed embedded applications. The fourth countermeasure cannot be applied to current operational systems.

All the strategies proposed in literature to prevent cache timing attacks have one thing in common: the trade off between the performance and security of the implementation. Evidently maximizing both parameters is a definite goal for any countermeasure. This is considered in [11] where the protection of just the first two and the last two rounds are considered. This countermeasure is for the cache trace attacks proposed in the same paper. In [4], the authors pointed out that protecting the first and last rounds is sufficient. However we will show in the following section that the second round also causes information leakage and hence has to be protected.

4 Pinpointing Cache Timing Attacks

For an AES encryption each table T_0, T_1, T_2, and T_3 is accessed four times in every round for the first 9 rounds, while table T_4 is accessed 16 times in the final round. In all there are 160 table accesses. If n_h is the number of cache

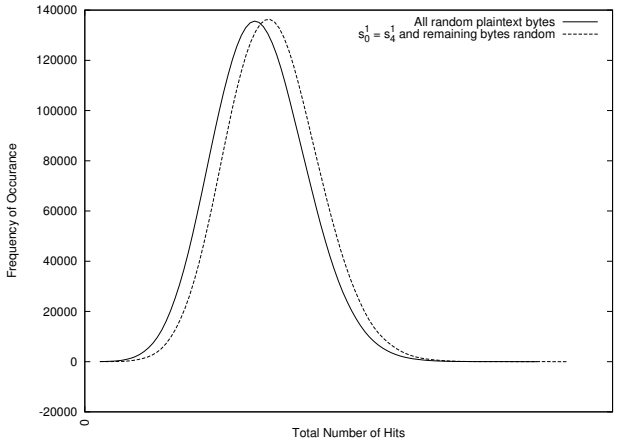

Figure 1. Normally Distributed Hit Count Shifted Left when Hit in First Round is Present

hits and n_m is the number of misses then the encryption time can be approximated by Equation 2.

$$\begin{aligned} T &= n_h \cdot T_h + n_m \cdot T_m + T_k \\ &= n_h \cdot T_h + (160 - n_h) \cdot T_m + T_k \end{aligned} \tag{2}$$

where T_h and T_m are the time required for a hit and a miss respectively, and T_k is a constant.

The time difference between two encryptions is represented by Equation 3.

$$\begin{aligned} \Delta T &= \Delta n_h \cdot T_h + \Delta n_m \cdot T_m \\ &= \Delta n_h (T_h - T_m) \end{aligned} \tag{3}$$

In the equation Δn_h and Δn_m are the difference in the number of hits and misses in the two encryptions respectively. The difference in the number of hits (or misses) occur because of the different patterns in accessing tables T_0 to T_4 as a result execution time for AES depends on its inputs (the plaintext and key), therefore in order to analyze the timing variation it is sufficient to analyze the number of hits and misses that occur during the program execution. In the remainder of this section we use Equation 2, with $T_h = 1$, $T_m = 4$ to approximate the actual timing of AES.

A First Round Distinguisher : For a given key if all bytes of the plaintext are varied randomly, the frequency distribution of the number of hits is normal (Figure 1). Further, if a single cache hit in the first round is forced, for example by setting $s_0^1 = s_4^1$ (ie. $p_0 \oplus k_0 = p_4 \oplus k_4$), then the normal distribution is still obtained but is shifted to the right with respect to the earlier distribution. This essentially is the first round distinguisher, and the reduced time obtained when there is a hit in the first round is used in first round cache timing attacks.

An attack based on this first round distinguisher can be carried out as follows. For every possible value for byte p_0, average hits are determined by randomly varying at least one byte of the plaintext for each row of Equation 1 except the bytes p_4, p_8, and p_{12}. It may be noted that the bytes p_0, p_4, p_8, and p_{12} access the same table (T_0) in the first round. The average hits for different values of p_0 is shown in Figure 2. There are three glitches, these correspond to hits that occur when $p_0 \oplus k_0 = p_4 \oplus k_4$, $p_0 \oplus k_0 = p_8 \oplus k_8$, and $p_0 \oplus k_0 = p_{12} \oplus k_{12}$. From this, the exclusive-or of the keys can be determined as shown in Equation 4.

$$\begin{aligned} k_0 \oplus k_4 &= p_0 \oplus p_4 \\ k_0 \oplus k_8 &= p_0 \oplus p_8 \\ k_0 \oplus k_{12} &= p_0 \oplus p_{12} \end{aligned} \tag{4}$$

In a similar way ex-or differences of other key bytes can be determined.

Figure 2 assumes a *hypothetical cache* in which a miss causes just a single data to be loaded into cache. However a

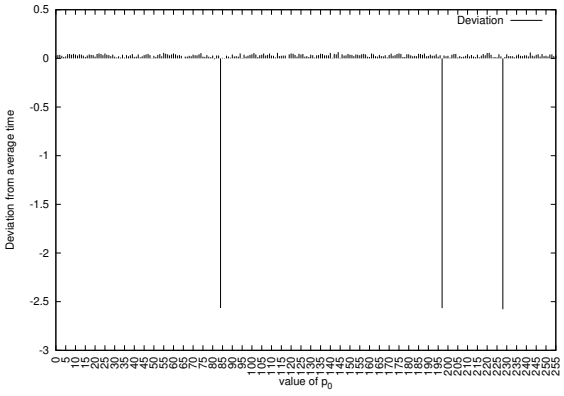

Figure 2. First Round with Plaintext Byte p_0 Varied

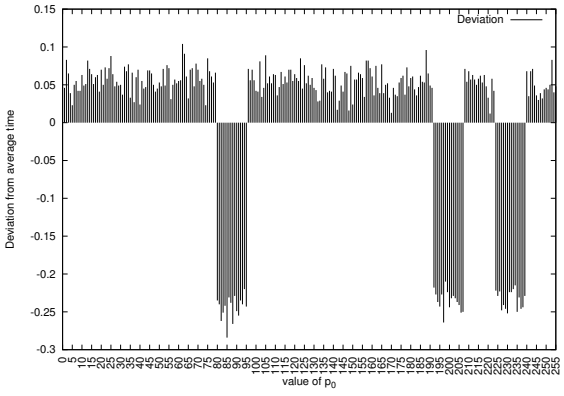

Figure 3. First Round with plaintext byte p_0 varied and cache line of 64 bytes

miss in a *real cache memory* results in an entire cache line to be loaded into cache. A cache line of size l bytes (which is generally a power of 2) contains contiguous memory locations, therefore a cache hit occurs when two table indices i and j are related by Equation 5, where $>>$ is the right shift operator.

$$i >> \log_2 l = j >> \log_2 l \tag{5}$$

For a real cache it is only possible to retrieve the higher order bits of the key bytes, therefore Equation 4 can be modified as shown in Equation 6.

$$\begin{aligned}(k_0 \oplus k_4)_{(7\cdots log_2 l)} &= (p_0 \oplus p_4)_{(7\cdots log_2 l)} \\ (k_0 \oplus k_8)_{(7\cdots log_2 l)} &= (p_0 \oplus p_8)_{(7\cdots log_2 l)} \\ (k_0 \oplus k_{12})_{(7\cdots log_2 l)} &= (p_0 \oplus p_{12})_{(7\cdots log_2 l)}\end{aligned} \tag{6}$$

Figure 3 shows the same attack as in Figure 2 except that it is simulated with for a cache having a cache line equal to 16 thirty two bit words (64 bytes). It may be noted that the three glitches present in Figure 2 have broadened in Figure 3. The extent to which it has broadened is equal to the cache line size (16 words).

A Second Round Distinguisher : In the first round if all bytes in a row (Equation 1) are kept constant while the remaining bytes are varied randomly then four bytes in the second round will be constant. For example, if s_0^1, s_5^1, s_{10}^1, s_{15}^1 are kept constant then s_0^2, s_1^2, s_2^2, and s_3^2 in the second round are constant. This can be used to force hits in the second round as explained below.

The plaintext byte p_0 is varied from 0 to 255 just as in the first round attack. This implies s_0^1 also takes on all possible bytes. The bytes s_5^1, s_{10}^1, s_{15}^1 are kept constant by keeping the corresponding plaintexts constant. The remaining bytes in the AES state are varied randomly and the average time taken for each value of p_0 is obtained. Figure 4 plots the average time taken for all possible values of p_0. There are at most 9 glitches that can occur in the graph. Figure 4 shows only 8 glitches instead of 9 because the ninth glitch coincides with the y axis. The 9 glitches occur when $s_0^2 = s_0^1$, $s_1^2 = s_5^1$, $s_2^2 = s_{10}^1$, $s_3^2 = s_{15}^1$ and when s_0^2, s_1^2, s_2^2, s_3^2, or s_0^1 are zero. This distinguisher can be exploited in cache attacks targeting the second round.

Third Round Distinguishers are Not Possible : In the previous descriptions of distinguishers on one and two rounds of AES, we have seen that the **average** number of hits are correlated to the plaintext value. To increase the average number of hits, at least one hit has to be forced in one of the rounds in each encryption. In order to obtain the complete normal curve (Figure 1) we need to observe the number of hits (in the form of encryption time) over large number of samples. This means that we need to have some randomly varying bytes in the plaintext. Further to inflict at least one hit, the tables should be accessed at some constant locations in one round while being accessed at all possible locations in the next round. This is possible only in the first and second round but not for the third round due to the AES structure. For example if there are two plaintext bytes varying, then the variation at the input of the third round is completely unpredictable. Moreover if only one plaintext byte is varied, then we obtain only one point in the normal curve and thus we are unable to obtain the required statistics. Figure 5 shows the plot of deviation from average time when the plaintext byte p_0 is varied while the rest are constant. We can observe that unlike the first and second rounds (Figures 2 and 4) no distinguishable glitches are present.

5 The Proposed Counter Measure

From the discussions in the previous section it may be concluded that significant variation in encryption timings are obtained whenever a hit is forced during the encryption.

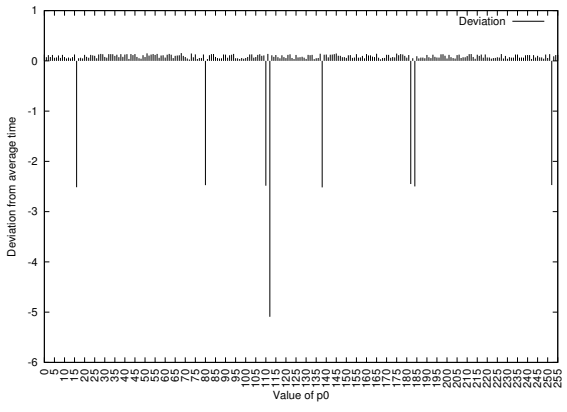

Figure 4. Second Round with Plaintext Byte p_0 Varied

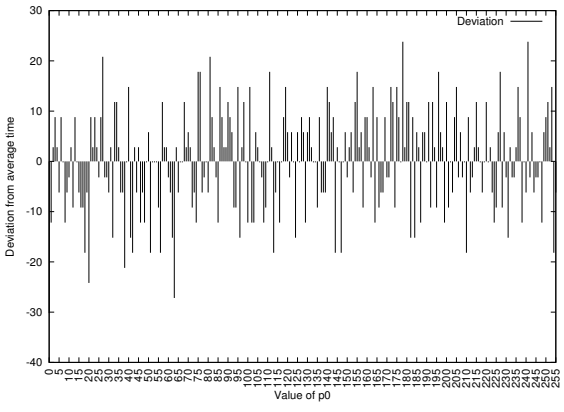

Figure 5. Third Round with Plaintext Byte p_0 Varied

In the first round distinguisher three plaintext bytes in a column (Equation 1) are maintained constant while the fourth byte is varied from 0 to 255 thus forcing hits in the first round. The second round distinguisher keeps three bytes in a row constant and varies the fourth byte from 0 to 255 thus forcing hits in the second round. In encryptions that have forced hits in the first two rounds, the encryption time is lesser than the average. The position in the graph (Figures 2 and 4) where the timing is significantly lower than average reveals information about key. These are used either directly or indirectly in all cache timing attacks.

It is impossible to force a hit in the third round (or for that matter any round greater than three) by just manipulating the input plaintext. Our proposed technique to counter cache attacks is to prevent information leakages in just the first and second rounds of the cipher. The preventive measure for the first and second round could be any of the previously used techniques. The advantage of our proposed method is that the overhead on the performance is reduced considerably, thus an AES implementation with our countermeasure implemented will not only prevent cache attacks but will also be able to do encryptions faster than regular protected AES implementations.

6 Experimental Validation and Results

To test our proposal we used an implementation of Bernstein's attack [10]. In this attack the client server architecture from [2] is replaced by a single system without need for any networking. In order to aid the attacker by reducing the attack time required, we flush the cache at every iteration to ensure that every encryption starts with a fresh cache.

We first executed the attack on an unprotected OpenSSL library[14] which uses Barreto's code [12]. The test platform was a laptop having Intel Core 2 Duo with $32KB$ L1 data cache. The attack was successful with 2^{25} randomly generated plaintexts. To determine k_0, the plaintext byte p_0 was varied from 0 to 255 and the graph of deviation from average is plotted. The graph (Figure 6) clearly shows for some values of p_0 the execution time was significantly different from the average. These correspond to the hits obtained in the initial rounds of the cipher.

Table 1. Comparison of Timings for Protected and Unprotected Implementations

Implementation of AES	Time (in clock cycles)	Overhead
Unprotected	712	-
All rounds protected	47500	66.71
2 rounds protected	7674	10.79

We then modified OpenSSL's AES code so that the first two rounds used composite field [5] implementations for sbox instead of table lookup. The attack was carried out on the same platform. After 2^{28} randomly generated plaintexts the attack was unsuccessful. The graph (Figure 7) clearly shows no pattern in the waveform.

Table 6 compares the timing for AES with two rounds protected and with all rounds protected. The two round protected AES has an overhead of 10.79 compared to the unprotected implementation, while the implementation with all rounds protected has an overhead of 66.71 times that of the unprotected AES implementation. This shows about 6 times improvement in the performance of our countermeasure.

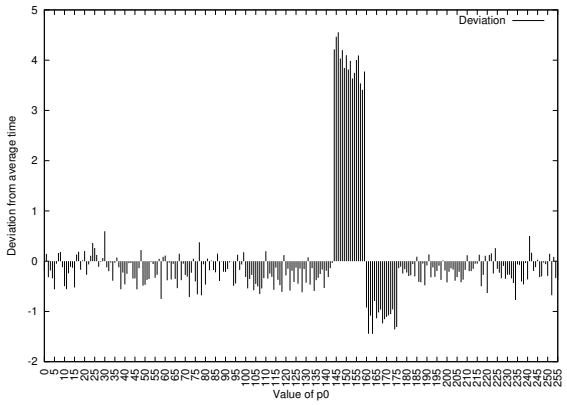

Figure 6. Deviation from average for p_0 on an Unprotected AES code

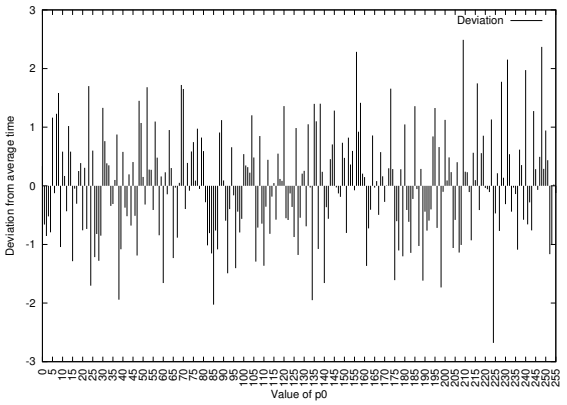

Figure 7. Deviation from average for p_0 on a Protected AES code

7 Conclusion

The paper pinpoints that the vulnerability of software implementations of AES due to cache timing is because of the ability of the attacker to control cache hits in the first and second rounds. It is explained that the attacker is not able to control the hits in the third round, thus making a third round cache attack infeasible. The paper proposes that providing countermeasures for the first two AES rounds are sufficient to prevent timing attacks. This leads to the implementation of AES resistant against cache timing attacks with minimum overhead. Detailed analysis and supported experimentations have been provided to justify the claim.

References

[1] O. Aciiçmez, W. Schindler, and Çetin Kaya Koç. Cache Based Remote Timing Attack on the AES. In M. Abe, editor, *CT-RSA*, volume 4377 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2007.

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

[3] J. Blömer and V. Krummel. Analysis of Countermeasures Against Access Driven Cache Attacks on AES. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *Selected Areas in Cryptography*, volume 4876 of *Lecture Notes in Computer Science*, pages 96–109. Springer, 2007.

[4] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert. Software mitigations to hedge aes against cache-based software side channel vulnerabilities. Cryptology ePrint Archive, Report 2006/052, 2006.

[5] D. Canright. A Very Compact S-Box for AES. In J. R. Rao and B. Sunar, editors, *CHES*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2005.

[6] A. Canteaut, C. Lauradoux, and A. Seznec. Understanding cache attacks. Research Report RR-5881, INRIA, 2006.

[7] Federal Information Processing Standards Publication 197. Announcing the Advanced Encryption Standard (AES), 2001.

[8] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113, London, UK, 1996. Springer-Verlag.

[9] M. Matsui and J. Nakajima. On the Power of Bitslice Implementation on Intel Core2 Processor. In P. Paillier and I. Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 121–134. Springer, 2007.

[10] M. Neve. *Cache-based Vulnerabilities and SPAM Analysis*. PhD thesis, Thesis in Applied Science, UCL, 2006.

[11] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In D. Pointcheval, editor, *CT-RSA*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[12] Paulo S. L. M. Barreto. The AES Block Cipher in C++. URL: http://www.larc.usp.br/˜pbarreto/AES++.zip.

[13] C. Rebeiro, A. D. Selvakumar, and A. S. L. Devi. Bitslice Implementation of AES. In D. Pointcheval, Y. Mu, and K. Chen, editors, *CANS*, volume 4301 of *Lecture Notes in Computer Science*, pages 203–212. Springer, 2006.

[14] The OpenSSL Project. http://www.openssl.org.