Software mitigations to hedge AES against cache-based software side channel vulnerabilities

Ernie Brickell[1], Gary Graunke[1], Michael Neve[1,⋆] and Jean-Pierre Seifert[1]

Intel Corporation
2111 NE 25th Avenue
Hillsboro, Oregon 97124, USA
{ernie.brickell,gary.graunke}@intel.com
{michael.neve.de.mevergnies,jean-pierre.seifert}@intel.com

Abstract. Hardware side channel vulnerabilities have been studied for many years in embedded silicon-security arena including SmartCards, SetTop-boxes, etc. However, because various recent security activities have goals of improving the software isolation properties of PC platforms, software side channels have become a subject of interest. Recent publications discussed cache-based software side channel vulnerabilities of AES and RSA. Thus, following the classical approach — a new side channel vulnerability opens a new mitigation research path — this paper starts to investigate efficient mitigations to protect AES-software against side channel vulnerabilities. First, we will present several mitigation strategies to harden existing AES software against cache-based software side channel attacks and analyze their theoretical protection. Then, we will present a performance and security evaluation of our mitigation strategies. For ease of evaluation we measured the performance of our code against the performance of the openSSL AES implementation. In addition, we also analyzed our code under various existing attacks. Depending on the level of the required side channel protection, the measured performance loss of our mitigations strategies versus openSSL (respectively best assembler) varies between factors of 1.35 (2.66) and 2.85 (5.83).

Keywords: AES, Countermeasures, Computer architecture, Computer security, Side channel attacks.

1 Introduction

Covert channels have long been recognized as a problem in designing secure software. Overt channels use the system's protected data objects to transfer information in a secure way. That is, one subject writes into a data object and another subject reads from that object. Subjects in this context are not only active users, but are also processes and procedures acting on behalf of users. The channels, such as buffers, files, shared memories, thread signals, etc. are overt because the entity used to hold the information is a data object; that is, it is an object that is normally viewed as a data container. Covert channels, in contrast, use entities or system resources not normally viewed as a data container to transfer information between subjects. These *metadata* objects, such as file locks, busy flags, execution time, disk access times, etc. are needed to register the state of the system. Covert channels involve the transfer of data across protected

⋆ Work done during an Intel internship. Affiliation: UCL Crypto Group (Belgium).

process boundaries between two cooperating processes. This observation was very early captured in the fundamental paper on the confinement problem by Lampson [Lam].

Overt channels are controlled by enforcing the access control policy of the system being designed and implemented. This policy states when and how overt reads and writes of data objects may be made. One part of the security analysis must verify the system's implementation correctly implements the stated access control policy. Recognizing and dealing with covert channels is more elusive. Objects used to hold the information being transferred are normally not viewed as data objects, but can often be manipulated maliciously in order to transfer information. In addition, the use of a covert channel requires collusion between a subject with authorization to access information which then leaks the information to an unauthorized object. Driven by strong military and government security requirements there has been a large amount of covert channel research, cf. [Lam,Kem83,Kem02]. Still it is extremely challenging to remove all covert channels from a system.

Under the *side channel* view the entity leaking the information is not intentionally cooperating in the act of leaking information, but the receiving entity is still able to obtain information through the same types of resource channels that are used for covert channels. There has been a lot of study of side channels for hardware devices, cf. Kocher's fundamental work [Koc]. He showed that the execution time of a hardware device could leak information to an unauthorized entity. But until very recently those side channel vulnerabilities had no relevance for the classical PC world and were only aimed towards embedded security devices. However, starting with Trusted Computing efforts [AFS,CEPW,EP,ELMP$^+$,Pea,Smi1,TCG] there are proposals to improve process isolation in the PC. This has created a new research-vector to examine the many covert channels in the PC and to study their effectiveness as software side channels, cf. [BB,Ber,BZBMP,HK,Per,ST,TSSSM,OST].

In this new PC software side channel arena the first shared resource to be studied is the system cache, cf. [Ber,Per,OST]. Let us clarify the history of this cache-based side channel. In the 90's, [Hu,Tro] pointed out how the classical cache behavior (miss or hit) might be used as a potential side channel. While none of them targeted the analysis of real cryptographic ciphers[1], in 2002 Page [Pag02] presented a very detailed hypothetical cache-based side channel against DES. Then, although the first experimental cache-based side channel results against DES and AES were finally presented by [TSSSM], it was again Page [Pag03] who, in 2003, explicitly described the foundations of the three recent publications [Ber,Per,OST]. Indeed, [Pag03] characterized two different cache-based side channels — the trace-driven and the time-driven cache attack methodology.

Trace-driven methodology. Trace-driven vulnerabilities, cf. [Hu,Tro,Per,OST], rely on the ability of the attacker to capture a profile of cache activity that results from running the algorithm under attack. This requires that the attacker can get access to a profile in which the cache-activity is observable and then process it to extract the cache-activity profile from other profile content. The result records if the cache produced a hit or miss for every access to memory. From such a trace it is relatively easy to relate S-box accesses for AES and DES assuming they are implemented via tables. By adapting the plaintext fed into

[1] We note that [KSWH] pointed also out that the classical cache-behavior might give raise to a side channel attack against cryptographic ciphers.

the algorithm and hence provoking different cache access patterns, the attacker can uncover the values of the key dependent variables that cause this specific cache behavior expressed in the trace profile.

Time-driven methodology. Time-driven attacks, cf. [Ber,HK,TSSSM], depend on the fact that when one runs the algorithm under attack, the execution time is effected directly by the number of cache misses. Using this relationship, the attacker can make algorithm-specific, statistically based inferences about the state during processing. Using this inference and a large number of measurements that produces the desired feature, the attacker can relate the plaintexts to the key-related variables and hence uncover their value. Because the core assumption of time-driven attacks is statistical, they generally result in higher online workload since the attack is dependent on enough statistical precision.

As time-driven cache-based side channel attacks appear to be much easier to implement, it is not surprising that the first practical results were time-driven, cf. [Ber,HK,TSSSM]. Namely, given the fundamental observations described in Hu [Hu] and Page [Pag03] it is fairly straightforward to construct a pure software cache-based side channel attack which is trace-driven. And indeed, using the core idea of [Hu], [Per] and [OST] succeeded to implement trace-driven cache-based side channels against RSA and AES.

Now, given the fact that cache-based side channel software attacks are a particular concern for modern security-efforts of all classical PC architectures it is obvious that it is important to find efficient software modifications to protect AES software against these kind of vulnerabilities. Also, this research is especially important as proposed hardware countermeasures such as those by Page [Pag05] cannot be expected to be in place within the near future.

Our software mitigations strategy is based on three principles: (1) compact tables, (2) frequently randomized tables and (3) pre-loading of relevant cache-lines. Using all three countermeasures simultaneously in an efficient manner and individually scaling them as appropriate for the power of the side channel adversary sets our mitigation strategy apart from all previously proposed mitigations.

The paper is organized as follows. The next section introduces notations and preliminaries which are used throughout the full paper. To study the effectiveness of our mitigation strategies, we will also define in section 3 the concept of a side channel adversarial threat model. Hereafter, section 4 successively develops the mitigations we are proposing, and section 5 presents the experimental efficiency and security results of the proposed mitigations. For completeness we also present in the appendix the complete x86-assembly code of our software-mitigations.

2 Notations and Preliminaries

2.1 Description of AES

AES has been extensively described in the literature (*e.g.* [DR2]). We will here only recall specific points needed throughout the present paper. We use a byte as a contiguous sequence of eight bits $\{0,1\}^8$ taking values in $\{0,\ldots,255\}$. AES deals with elements of n bytes ($n = 16$, 24 or 32) represented by $\mathbf{b} = (b_0, \ldots, b_{n-1})$. For the rest of the paper, we will use $n = 16$, but it is straightforward to extend our results to longer key sizes. Any plaintext $\mathbf{p}_i$ is represented the same way, $\mathbf{p}_i = (p_{0,i}, \ldots, p_{15,i})$, where $p_{j,i}$ is the j-th byte of $\mathbf{p}_i$. A 16-byte key $\mathbf{k} = (k_0, \ldots, k_{15})$ is expanded by `KeyExpansion` into 10 round keys $\mathbf{K}^{(r)} = (K_0^{(r)}, \ldots, K_{15}^{(r)})$ for $r = 0, \ldots, 10$; with $\mathbf{k} = \mathbf{K}^{(0)}$ (the number of round equals 10, 12 or 14, respectively, when n is 16, 24 or 32). After an initial

AddRoundKey, AES performs r successive rounds where SubBytes, ShiftRows, MixColumns and AddRoundKey are applied to a state. A state is defined as $\mathbf{x}^{(r)} = (x_0^{(r)}, \ldots, x_{15}^{(r)})$ and it is the result of the r-th AddRoundKey. The initial state is obtained by the first AddRoundKey, i.e., $x_{j,i}^{(0)} = p_{j,i} \oplus k_j$. We then introduce the r-th round of a plaintext $\mathbf{p}_i^{(r)} = (p_{0,i}^{(r)}, \ldots, p_{15,i}^{(r)})$ as input of the r-th AddRoundKey, i.e., $x_{j,i}^{(r)} = p_{j,i}^{(r)} \oplus K_j^{(r)}$. An encryption of plaintext $\mathbf{p}$ by AES with key $\mathbf{k}$ produces a ciphertext $\mathbf{c}$, denoted as $\mathbf{c} = E_{AES}(\mathbf{p}, \mathbf{k})$.

Popular software implementations of AES (OpenSSL [OpenSSL] for example) usually perform the round operations with a granularity of a word (4 bytes). Each round r state word $\mathbf{x}_i^{(r)} = (x_{4i}^{(r)}, x_{4i+1}^{(r)}, x_{4i+2}^{(r)}, x_{4i+3}^{(r)})$, $i = 0, \ldots, 4$, is generated as:

$$\begin{aligned}
\mathbf{x}_0^{(r)} &= T_0\left[x_0^{(r-1)}\right] \oplus T_1\left[x_5^{(r-1)}\right] \oplus T_2\left[x_{10}^{(r-1)}\right] \oplus T_3\left[x_{15}^{(r-1)}\right] \oplus \mathbf{K}_0^{(r)} \\
\mathbf{x}_1^{(r)} &= T_0\left[x_4^{(r-1)}\right] \oplus T_1\left[x_9^{(r-1)}\right] \oplus T_2\left[x_{14}^{(r-1)}\right] \oplus T_3\left[x_3^{(r-1)}\right] \oplus \mathbf{K}_1^{(r)} \\
\mathbf{x}_2^{(r)} &= T_0\left[x_8^{(r-1)}\right] \oplus T_1\left[x_{13}^{(r-1)}\right] \oplus T_2\left[x_2^{(r-1)}\right] \oplus T_3\left[x_7^{(r-1)}\right] \oplus \mathbf{K}_2^{(r)} \\
\mathbf{x}_3^{(r)} &= T_0\left[x_{12}^{(r-1)}\right] \oplus T_1\left[x_1^{(r-1)}\right] \oplus T_2\left[x_6^{(r-1)}\right] \oplus T_3\left[x_{11}^{(r-1)}\right] \oplus \mathbf{K}_3^{(r)}.
\end{aligned}$$

Here, T_0, T_1, T_2, T_3 are four lookup tables with 1 byte input and 1 word output and $\mathbf{K}_i^{(r)} = (K_{4i}^{(r)}, K_{4i+1}^{(r)}, K_{4i+2}^{(r)}, K_{4i+3}^{(r)})$ is the i-th word of the r-th round key. Note that the last round uses another lookup table T_4. The lookup tables are precomputed and provide a noticeable increase of performances.

2.2 Large tables and caches

Let us briefly elaborate a little bit on the vulnerability caused by careless large table AES implementations and the behavior of caches. For a thorough introduction into caches we refer the reader to [Sha,MP]. For our purposes, a simple mechanism like the following direct mapped cache suffices. With s and b being respectively the number of cache lines and the size of a cache line, the main memory is divided in contiguous blocks of b bytes. A data in main memory with address a will be mapped into the cache line $i := a \bmod s$, as shown in the following picture:

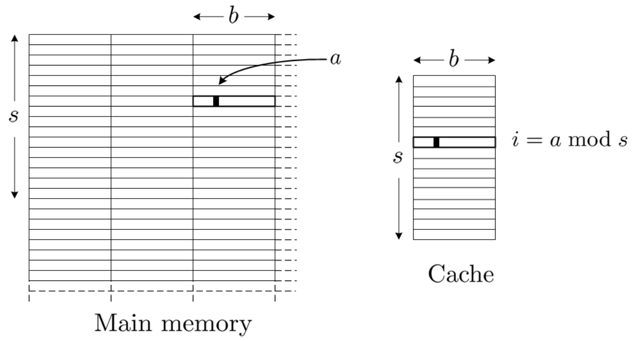

Fig. 1. Directly mapped cache.

During compilation of the AES program, the 1kB S-Box tables T_0, T_1, T_2, T_3 are given addresses in memory, from which their position in the cache will be later derived. Consider now such an access through $p_{j,i} \oplus k_j$ (for a $j \in \{0, \ldots, 15\}$). This will fetch into the cache b contiguous bytes of $T_x[p^*_{j,i} \oplus k_j]$, with $\langle p^*_{j,i} \rangle_{8-\log_2(b)} = \langle p_{j,i} \rangle_{8-\log_2(b)}$ and $x \in \{0, \ldots, 4\}$, where $\langle \cdot \rangle_m$ denotes the upper m most significant bits. That means that the side channel — cache *miss* or *hit* — cannot distinguish byte accesses within a cache-line as the bytes inside a cache-line are all "somehow" equivalent. This can also be observed through the byte time-signature where peaks are gathered in sets of $2^{8-\log_2(b)}$ values. As an example see Figure 2.

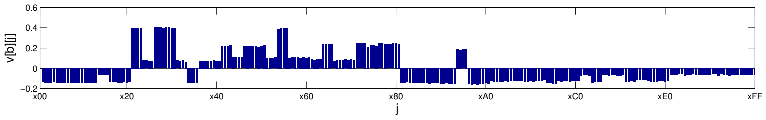

Fig. 2. Signature chart for an individual byte. The X-axis represents values from 0 to 255 a particular byte can take, while the Y-axis gives the relative timing for each individual value compared to the mean timing.

As aforementioned said, the cache can leak about the behavior of security programs and cryptographic applications. Successful cache-based side channel attacks have been recently presented in [Ber,OST,Per].

In one hand, Bernstein's time-driven attack correlates measured encryption times of a remote computer using a secret session key, with profiled encryption times of a similar computer with a known session key. This kind of attack enables the attacker to easily disclose a large percentage of the secret key.

On the other hand, access-driven attacks analyze the required time for the processor to access specific memory location. Depending whether a specific memory data has been kept in cache or has been evicted by another process, program, thread etc., a cache-hit or a cache-miss occurs on the next access. Osvik *et al.* targeted AES implementations in various scenarios, while Percival managed to spot usage of precomputed values during RSA encryption.

As the cache operates transparently for programs, preventing a variation of execution times appears unrealistic — as already observed by [Ber]. For an effective but simple example, consider how the execution time distribution (cf. Fig. 3) of AES (openSSL) indicates a potential information leakages. According to the central limit theorem, the distribution should be Gaussian; any divergence from that should be considered as potentially harmful.

3 Discussion of a side channel adversarial threat model

To study the effectiveness of our mitigation strategies, we will now introduce the notion of a side channel adversarial threat model. We do this in order to describe the effectiveness of the mitigations separately from discussing the effectiveness of an adversaries ability to exploit a software side channel vulnerability. In a software side channel, the adversary is executing a spy process on a platform that is also executing a crypto process in a multi-tasking environment. To study the cache side channel, the important ingredient is the accuracy of the information that the adversary can obtain about the cache accesses of the target process. Thus, as we describe mitigations, we will discuss how frequently the spy process would need to get data about the cache accesses of the target process in order to defeat the mitigation.

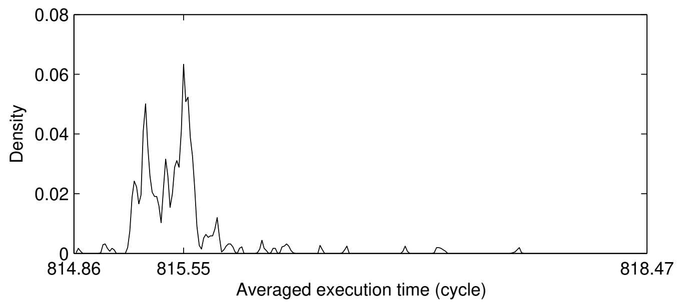

Fig. 3. Distribution of execution times for an unprotected openSLL implementation of AES. The execution times are averaged, regarding the value of one fixed byte of the plaintext, over a large number of random plaintexts (measurements have been taken according to [Ber]). In this example, two visible peaks suggest higher density around two timing accesses.

According to the above section 2.2, for AES the spy process is trying to obtain information about which cache lines are being accessed by the crypto process when the crypto process accesses the S-Box tables. Against an adversary who is able to obtain precise information about all cache accesses, a mitigation for the crypto process would be to access all cache lines of the tables each time it needed to access any entry in the table. This would be quite inefficient. However, this is probably a higher bar than is really necessary, since we do not know of any method for a spy process to obtain this precise information. Thus, we will present several different alternate methods for defending against more realistic (and demonstrated) spy processes.

The *first* method we present involves using a compact S-Box table which fills (on most modern processors) only 4 cache lines, and then accessing each of the 4 cache lines in every round. This method will defend against an adversary who is not able to observe cache access behavior more frequently than the time required by the crypto process to execute an AES round, cf. [Ber,TSSSM]

The *second* method we present does the above process only for the first and last rounds, but for the middle rounds (2 through 9), uses still larger S-box tables (which is more efficient). The reason this may still be an effective mitigation is because it is more difficult for the adversary to use information about the cache accesses of the middle rounds. Instead of accessing every cache line of these larger tables with every round, we permute the tables some number of encryption blocks, thus further obscuring the information from cache accesses. We conjecture that this method will effectively defend against an adversary who is only able to obtain information about cache access behavior every few rounds. This is perhaps conservative, since we also don't know of an attack that would be effective against this method by an adversary who was able to obtain information for individual AES rounds.

Finally, we introduce a *third* method that may be an effective mitigation even against an adversary who is able to observe cache access behavior multiple times during a round of AES. This method also uses the compact S-Box tables, and accesses each cache line in the table every round, but adds the additional step of permuting the compact S-Box tables, and changing the permutation very frequently.

In an expanded version of the present paper we will show how our various adversary models can be fitted into the formal side channel attack models of

[CJRR,BGK]. This allows formal security proofs of our presented software mitigations against side channel adversaries of specified power.

In the remainder of the paper, we will give more precise descriptions and security arguments for these mitigation methods, and discuss their performance on experimental implementations.

4 Mitigations against cache-based side channel attacks

As already mentioned before we will now explain our three individual mitigations strategies in more detail: (1) compact S-box table, (2) frequently randomized tables, (3) pre-loading of relevant cache-lines. However, due the lack of space and especially for the reason of interest we will present our mitigations in a more compact form. Namely, we will present the use of the compact S-box (first and last round) and the large-table based (inner) rounds only while making direct use of a permutation P. Therefore, assuming that a permutation P is somehow given and can be efficiently computed, we will show first how a permuted compact S-box **byte PinvS**[0 : 255] and as well a large permuted table **word T**[0 : 511] can be efficiently computed.

4.1 Constructing permuted S-box and inner round T table.

For later use, we show in figure 4 how to compute the randomized (via P) compact S-box table **PinvS** and as well the randomized (via P) large table **T**.

4.2 Permuted compact round

First, we will show how to efficiently use a single and permuted compact S-box table (256 bytes-table) in a *single* AES round. As already pointed out by [OST] (and easy derivable from our section 2.2), substituting the five big 1KB tables (as used in openSSL) by a compact S-box table (256 bytes-table) reduces dramatically the information leakage due to potential cache misses.

But, in contrast to [OST] we add additional protection mechanisms to the use of a single compact table:

- As the 256-bytes table fits in only 4 cache-lines, we can now afford it to pre-fetch all 4 cache-lines of the compact table into the cache prior to using the table.
- Frequently permuting the compacted table with a new permutation injects lots of entropy against adversaries which are relying on statistical success probabilities.

The following figure 5 shows a permuted compact round with pre-fetching of relevant cache-lines. The corresponding x86-assembler program is concluded in the appendix. Also, thanks to the x86 SSE SIMD instructions, we observe that the security-critical computations are totally constant time and are all done in parallel without any branches.

4.3 Permuted non-compact tables

After having seen how to permute (via permuation P) a compact round, we will show now, how to compute the inner rounds by using the large permuted table **T** from figure 4. The following figure 6 shows the corresponding pseudo-code, while the corresponding x86-assembler program is concluded in the appendix.

```
input: Permutation P (with parameters A and B)
output: Permuted S-box PinvS[0 : 255] and inner round table T[0 : 511]

  S-box is a 256-byte vector S[0 : 255];
  (byte [0 : 255], word [0 : 511]) function permute_S_box(S[0 : 255], P);
  begin
    byte PinvS[0 : 255], m;
    word T[0 : 511];

    for all i do parallel PinvS[i] := 0;
    for all j cache_blocks in S do
    begin
      for all i entries in S do
      begin
        p := P(i);
        m := if offset p is in cache block j then 255 else 0;
        PinvS[j * cacheblocksize + p mod cacheblocksize] : + = m ∧ S[i];
      end
    end
    /* for version 2, build expanded table T for inner rounds from PinvS */
    for all i ∈ [0 : 255] do
    begin
      word v1, v2, v3, c;

      v1 := PinvS[i];
      c := if v1 > 127 then 0x1b else 0;
      v2 := (v1 * 2) ⊕ c mod 256;
      v3 := v1 ⊕ v2;
      /* store twice so we can implement rotation by unaligned load */
      T[2 * i] := T[2 * i + 1] := v2 ⊕ (v1 <<< 8) ⊕ (v1 <<< 16) ⊕ (v3 <<< 24);
    end
    return (PinvS, T);
  end
```

Fig. 4. Constructing permuted S-box and inner round **T** table.

```
input: 16-byte vector b[0 : 15]
output: the result of applying one AES-round transformation to b

  permuted S-box is a 256-byte vector PinvS[0 : 255];
  as permutation P we simply implemented P(x) := (B + x) * A mod 256
    where A is odd;
  /* P can be efficiently computed with SSE instructions */
  r is a permutation of 16 bytes (0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12, 1, 6, 11)

  byte [0 : 15] function compact_encrypt_round(b[0 : 15], roundkey[0 : 15], A, B);
  begin
    byte v1[0 : 15], v2[0 : 15], v3[0 : 15], c[0 : 15];

    for one representative i from each cache block of S, touch S[i];
    for all i do b[i] := (b[i] + B) * A mod 256;
    for all i do v1[r[i]] := PinvS[i];
    /* complete linear transform with SSE SIMD-instructions */
    for all i do parallel c[i] := if v1[i] > 127 then 0x1b else 0;
    for all i do parallel v2[i] := (v1[i] * 2) ⊕ c mod 256;
    for all i do parallel v3[i] := v1[i] ⊕ v2[i];
    return roundkey ⊕ v2 ⊕ blrm4(v1, 1) ⊕ blrm4(v1, 2) ⊕ blrm4(v3, 3);
  end

  byte [0 : 15] function lbrm4(v[0 : 15], j);
  begin
    byte t[0 : 15];

    for all i do parallel t[4 * ⌊(i/4)⌋ + ((i mod 4 + j) mod 4)] := v[i];
    return t;
  end
```

Fig. 5. Permuted compacted round.

input: 16-byte vector $\mathbf{b}[0:15]$
output: the result of applying one AES-round transformation to $\mathbf{b}$

S-box is a 256-byte vector $\mathbf{S}[0:255]$;
$\mathbf{T}$ is a 2048-byte table for AES's linear transform and **PinvS**
to enable unaligned rotates

byte $[0:15]$ **function** inner_round_encrypt($\mathbf{b}[0:15], \mathbf{roundkey}[0:15]$);
begin
  byte $\mathbf{t}[0:15]$;
  word $\mathbf{w}[0:3]$;

  /* for performance we cannot touch all cache-lines in the inner rounds */
  for all i **do** $\mathbf{t}[i] := (\mathbf{b}[i] + B) * A \bmod 256$;
  $\mathbf{w}[0] := \mathrm{Word}(\mathbf{T}, 8*\mathbf{t}[0]) \oplus \mathrm{Word}(\mathbf{T}, 3+8*\mathbf{t}[5]) \oplus$
    $\mathrm{Word}(\mathbf{T}, 2+8*\mathbf{t}[10]) \oplus \mathrm{Word}(\mathbf{T}, 1+8*\mathbf{t}[15])$;
  $\mathbf{w}[1] := \mathrm{Word}(\mathbf{T}, 8*\mathbf{t}[4]) \oplus \mathrm{Word}(\mathbf{T}, 3+8*\mathbf{t}[9]) \oplus$
    $\mathrm{Word}(\mathbf{T}, 2+8*\mathbf{t}[14]) \oplus \mathrm{Word}(\mathbf{T}, 1+8*\mathbf{t}[3])$;
  $\mathbf{w}[2] := \mathrm{Word}(\mathbf{T}, 8*\mathbf{t}[8]) \oplus \mathrm{Word}(\mathbf{T}, 3+8*\mathbf{t}[13]) \oplus$
    $\mathrm{Word}(\mathbf{T}, 2+8*\mathbf{t}[2]) \oplus \mathrm{Word}(\mathbf{T}, 1+8*\mathbf{t}[7])$;
  $\mathbf{w}[3] := \mathrm{Word}(\mathbf{T}, 8*\mathbf{t}[12]) \oplus \mathrm{Word}(\mathbf{T}, 3+8*\mathbf{t}[1]) \oplus$
    $\mathrm{Word}(\mathbf{T}, 2+8*\mathbf{t}[6]) \oplus \mathrm{Word}(\mathbf{T}, 1+8*\mathbf{t}[11])$;
end

word function $\mathrm{Word}(x, y) \equiv$
  extracts from byte array x 4 bytes at byte offset y and returns 32-bit word;

byte $[0:3]$ **function** $\mathrm{Bytes}(x) \equiv$
  returns a byte array for a given 32-bit word x;

Fig. 6. Permuted non-compact round.

5 Practical results of our mitigations

Although our aforementioned software mitigation strategies are very flexible and allow various combinations with different security and performance strengths, we simply tested the following configurations.

V1: All rounds compact, no permutation.
V2: Outer rounds (round 1 and round 10) are compact and the inner rounds are all large (round 2 until round 9) and additionally tables are periodically permuted.
V3: All rounds compact and tables are periodically permuted.

5.1 Performance

The following figure 7 succinctly summarizes our performance results when comparing the above three variants with the performance of the openSSL and the best assembler implementation of the AES encryption/decryption algorithm.

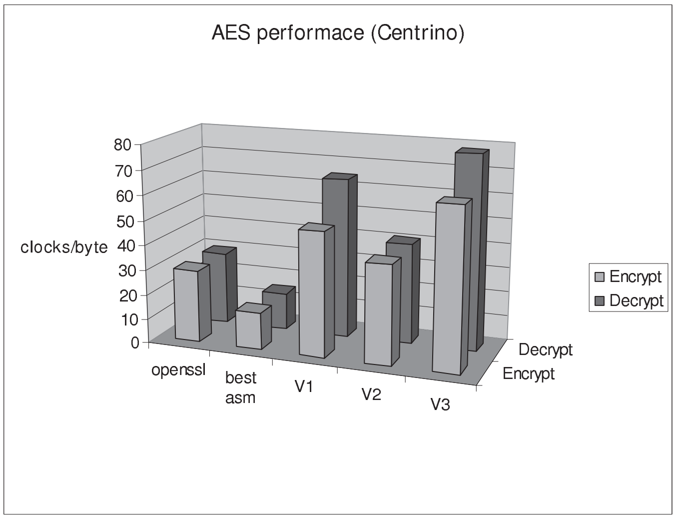

Fig. 7. Performance of mitigations for AES.

5.2 Security

We verified experimentally that all of the methods described above are effective in removing the timing vulnerabilities that were exploited by Bernstein [Ber]. This is easily visible from the following figure 8.

In the expanded (and complete) version of the present paper we will also present our successful results of hedging AES software with our mitigations against the very powerful adversaries implicitly given by [OST].

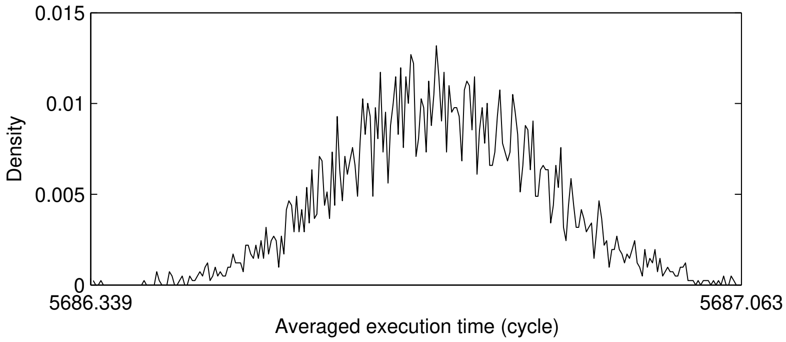

Fig. 8. Distribution of execution times for our protected implementation (version V2) of AES. The execution times are averaged, regarding the value of one fixed byte of the plaintext, over a large number of random plaintexts (measurements have been taken according to [Ber]). As expected, the distribution follows a Gaussian distribution.

6 Conclusions and recommendations for further research

In this paper, we have presented new methods and ideas to mitigate recently demonstrated software side channel attacks. We have also presented appropriate methods for discussing the effectiveness of such mitigations.

While Bernstein and OST [Ber,OST] also made numerous suggestions for software methods for implementations of AES that would protect against cache-based software side channels, the methods that we present in this paper are different from any of the ones suggested previously. For example, although [OST] suggested the possibility of using the small S-Boxes, they correctly argued that this mitigation by itself would not defeat their powerful adversaries, but would only require him to use more time. Moreover, they did also not suggest to combine this with accessing all of the cache lines in the small S-Boxes in every round or periodically permuting this compact S-box.

Additionally, we also introduced the concept of evaluating the mitigations relative to the power of the adversaries. This evaluation is useful in the search for efficient enough and secure enough mitigations to these new side channel vulnerabilities. Moreover, this paper presented also specific experimental results deducted from experiments with the mitigations proposed.

We believe also that this is only the beginning of a new research path parallel to the hardware side channel research. Indeed, we are convinced that more software side channels will be discovered, which will result in new interesting mitigation methods. Further work formalizing the power of side channel adversaries will also be useful.

Acknowledgments

We are indebted to Adi Shamir and Eran Tromer for numerous and valuable discussions about their work [OST].

References

[AFS] W. A. Arbaugh, D.J. Farber, J.M. Smith, "A secure and reliable bootstrap architecture", Proc. of IEEE Symp. On Privacy and Security, pp. 65-71, 1997.

[BZBMP] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, G. Palermo, "AES Power Attack Based on Induced Cache Miss and Countermeasure" Proc. of International Conference on Information Technology: Coding and Computing (ITCC'05), IEEE Press, pp. 586-591, 2005.

[Ber] D. J. Bernstein, "Cache-timing attacks on AES", preprint available from `http://cr.yp.to/papers.html#cachetiming`, 37 pages, 2005.

[BGK] J. Blömer, J. Guajardo and V. Krummel, "Provably Secure Masking of AES", Proc. of Selected Areas in Cryptography (SAC), Springer LNCS vol. 3357, pp. 69-83, 2004.

[BB] D. Boneh and D. Brumley, "Remote timing attacks are practical", Proc. of 12th Usenix Security Symposium, USENIX, pp. 1-14, 2003.

[CJRR] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards Sound Approaches to Counteract Power-Analysis Attacks", Proc. of Advances in Cryptology (CRYPTO '99), Springer LNCS vol. 1666, pp. 398-412, 1999.

[CEPW] Y. Chen, P. England, M. Peinado, B. Willman, "High Assurance Computing on Open Hardware Architectures", Microsoft Technical Report, MSR-TR-2003-20, March 2003.

[CNK] J.-S. Coron, D. Naccache and P. Kocher, "Statistics and Secret Leakage", *ACM Transactions on Embedded Computing Systems* **3**(3):492-508, 2004.

[DR2] J. Daemen and V. Rijmen, *The Design of Rijndael*, Springer-Verlag, Berlin, 2002.

[EP] P. England, M. Peinado, "Authenticated Operation of Open Computing Devices", ACISP '02, Springer-Verlag, LNCS vol. 2384, pp. 346-361, 2002.

[ELMP+] P. England, B. Lampson, J. Manferdelli, M. Peinado, B. Willman, "A Trusted Open Platform", *IEEE Computer*, **36(7)**:55-62, 2003.

[HK] A. Hevia and M. Kiwi, "Strength of Two Data Encryption Standard Implementations under Timing Attacks" *ACM Transactions on Information and System Security* **2**(4):416-437, 1999.

[Hu] W. M. Hu, "Lattice scheduling and covert channels", Proc. of IEEE Symposium on Security and Privacy, IEEE Press, pp. 5261, 1992.

[KSWH] J. Kelsey, B. Schneier, D. Wagner and C. Hall, "Side channel cryptanalysis of product ciphers", Proc. of 5th European Symposium on Research in Computer Security, Springer LNCS 1485, pp. 97110, 1998.

[Kem83] R. Kemmerer, "Shared resource matrix methodology: An approach to identifying storage and timing channels", *ACM Transactions on Computer Systems* **1**(3), pp. 256-277, 1983.

[Kem02] R. Kemmerer, "A practical approach to identifying storage and timing channels: Twenty years later", Proc. of 18th Annual Computer security Applications Conference (ACSAC'02), IEEE Press, pp. 109-118, 2002.

[Knu] D. E. Knuth, *The Art of Computer Programming, Vol.2: Seminumerical Algorithms*, 3rd ed., Addison-Wesley, Reading MA, 1999.

[Koc] P. C. Kocher, *Timing attacks on implementations of DH, RSA, DSS and other systems*, CRYPTO '96, Springer LNCS, pp. 104-113, 1996.

[Lam] B. W. Lampson, "A note on the confinement problem", *Communications of the ACM* **16**(10):613-615, 1973.

[MP] S. M. Mueller and W. J. Paul, *Computer Architecture*, Springer-Verlag, Berlin, 2002.

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

[OST] D. A. Osvik, A. Shamir and E. Tromer, "Cache attacks and Countermeasures: the Case of AES", Cryptology ePrint Archive, Report 2005/271, 2005.

[Pea] S. Pearson, *Trusted Computing Platforms: TCPA Technology in Context*, Prentice Hall PTR, 2002.

[Per] C. Percival, "Cache missing for fun and profit", Proc. of BSDCan 2005, Ottawa, manuscript available from `http://www.daemonology.net/hyperthreading-considered-harmful/`.

[Pag05] D. Page, "Partitioned Cache Architecture as a side Channel Defence Mechanism", Cryptology ePrint Archive, Report 2005/280, 2005.

[Pag03] D. Page, “Defending Against Cache Based side Channel Attacks”, *Information Security Technical Report* **8**(1):30-44, 2003.

[Pag02] D. Page, “Theoretical Use of Cache Memory as a Cryptanalytic side Channel”, Cryptology ePrint Archive, Report 2002/169, 2002.

[Smi1] S.W. Smith, *Trusted Computing Platforms: Design and Applications*, Springer-Verlag, 2004.

[ST] A. Shamir and E. Tromer, “Acoustic cryptanalysis”, presentation available from `http://www.wisdom.weizmann.ac.il/~tromer/`.

[Sha] T. Shanley *The Unabridged Pentium 4 : IA32 Processor Genealogy*, Addison-Wesley Professional, 2004.

[TCG] Trusted Computing Group, `http://www.trustedcomputinggroup.org`.

[Tro] J.T. Trostle, “Timing attacks against trusted path”, Proc. of IEEE Symposium on Security and Privacy, IEEE Press, pp. 125-134, 1998.

[TSSSM] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri and H. Miyauchi, “Cryptanalysis of DES implemented on computers with cache”, Proc. of CHES 2003, Springer LNCS, pp. 62-76, 2003.

Appendix

First round and second round code for our mitigation version 2.

```
/* SSE version first round code--prefetch all 4 lines for outer rounds and
use compact S-box table with LT computation and does the permutation (x+B)*A
*/

   mov edi,key // [ebp+16]
   mov esi,inptr // [ebp+8]
   movdqa xmm7,SSEmask
   push ebp
   movdqa xmm6,[SSE_MULT+edi]
   movdqa xmm5,[SSE_BIAS+edi]
   movdqu xmm0,0[esi]
   pxor xmm0,[SSE_KEY_OFFSET+0*16+edi] // key addition
// begin round 1
   paddb xmm0,xmm5 // add bias B
   add ebp,[SSE_SBOX+edi] // touch all sbox lines
   movdqa xmm1,xmm7 // lower byte mask
   pandn xmm1,xmm0 // pick off upper bytes (using not of mask)
   add ebp,[SSE_SBOX+64+edi] // touch all sbox lines
   pmullw xmm0,xmm6 // multiplier A
   add ebp,[SSE_SBOX+128+edi] // touch all sbox lines
   pand xmm0,xmm7 // lower byte mask
   pmullw xmm1,xmm6 // note LSB is zero, and will stay zero
   add ebp,[SSE_SBOX+196+edi] // touch all sbox lines
   por xmm0,xmm1 // merge bytes back
   movd eax,xmm0 // pextrw eax,xmm0,0
   pextrw ebx,xmm0,2
// try unaligned word loads with masking
   movzx esi,al // 0, 5, 10, 15
   movzx ebp,[SSE_SBOX+esi+edi]
   pextrw ecx,xmm0,5 // load 10, 11
   movzx esi,bh // 5
   mov esi,[SSE_SBOX-1+esi+edi]
   pextrw edx,xmm0,7 // load 14, 15
   and esi,0xff00
   or ebp,esi
```

```
    movzx esi,cl // 10
    mov esi,[SSE_SBOX-2+esi+edi]
    and esi,0xff0000
    or ebp,esi
    movzx esi,dh // 15
    mov esi,[SSE_SBOX-3+esi+edi]
    and esi,0xff000000
    or ebp,esi
    movd xmm2,ebp
    movzx esi,ah // do parts of last row 1 before shifting eax, ecx
    mov ebp,[SSE_SBOX-1+esi+edi]
    and ebp,0xff00
    movzx esi,ch // 11
    mov esi,[SSE_SBOX-3+esi+edi]
    pextrw ecx,xmm0,4 // pextrw ecx,xmm2,4 // load 8,9
    and esi,0xff000000
    or ebp,esi
    movd xmm4,ebp // save partial result
    movzx esi,bl // 4, 9, 14, 3
    shr eax,16 // finish load 2, 3 from initial movd
    movzx ebp,[SSE_SBOX+esi+edi]
    movzx esi,ch // 9
    mov esi,[SSE_SBOX-1+esi+edi]
    and esi,0xff00
    or ebp,esi
    movzx esi,dl // 14
    mov esi,[SSE_SBOX-2+esi+edi]
    pextrw ebx,xmm0,3 // load 6, 7
    and esi,0xff0000
    or ebp,esi
    movzx esi,ah // 3
    mov esi,[SSE_SBOX-3+esi+edi]
    and esi,0xff000000
    or ebp,esi
    pextrw edx,xmm0,6 // load 12, 13
    movd xmm3,ebp
    movzx esi,cl // 8, 13, 2, 7
    movzx ebp,[SSE_SBOX+esi+edi]
    unpcklps xmm2,xmm3 // combine first two rows
    movzx esi,dh // 13
    mov esi,[SSE_SBOX-1+esi+edi]
    and esi,0xff00
    or ebp,esi
    movzx esi,al // 2
    mov esi,[SSE_SBOX-2+esi+edi]
    and esi,0xff0000
    or ebp,esi
    movzx esi,bh // 7
    mov esi,[SSE_SBOX-3+esi+edi]
    and esi,0xff000000
    or ebp,esi
    movd xmm3,ebp
// finish last row
    movzx esi,dl // 12, 1, 6, 11
    movzx edx,[SSE_SBOX+esi+edi]
    pxor xmm1,xmm1 // preload zero
```

```
   movzx esi,bl // 6
   mov esi,[SSE_SBOX-2+esi+edi]
   and esi,0xff0000
   or edx,esi
   movd xmm0,edx
   por xmm0,xmm4 // merge partial results for last row
   unpcklps xmm3,xmm0 // combine 3rd and 4th rows
   movdqa xmm0,SSEmask1B // preload constant
   unpcklpd xmm2,xmm3 // combine all rows
// compact table linear transform
   pcmpgtb xmm1,xmm2 // < 0 means top bit is on
   movdqa xmm4,xmm2 // copy v1
   paddb xmm2,xmm2
   pand xmm1,xmm0 // masked load of 1b
   movdqa xmm0,xmm4 // copy v1
   pxor xmm2,xmm1 // v2
   movdqa xmm1,xmm4 // copy v1
   psrld xmm4,16 // v1 >> 16
   movdqa xmm0,xmm1 // continue copying v1
   pslld xmm1,8 // v1 << 8
   pxor xmm0,xmm2 // v1 ^ v2
   pxor xmm2,xmm4 // xor in v1 >> 16
   psrld xmm4,8 // continue shifting: v1 >> 24
   pxor xmm2,xmm1 // xor in v1 << 8
   pslld xmm1,8 // continue shifting: v1 << 16
   pxor xmm2,xmm4 // xor in v1 >> 24
   movdqa xmm4,xmm0 // copy v1^v2
   pslld xmm0,24 // (v1^v2) << 24
   pxor xmm2,xmm1 // xor in v1 << 24
   psrld xmm4,8 // (v1^v2) >> 8
   pxor xmm2,xmm0 // xor in (v1^v2) << 24
   pxor xmm2,xmm4 // xor in (v1^v2) >> 8
// end of linear transform
   pxor xmm2,[SSE_KEY_OFFSET+1*16+edi] // key addition
```

The inner round code for version 2 of our mitigation that uses the big (permuted) tables in the inner rounds.

```
// begin round 2
   paddb xmm2,xmm5 // add bias B
   movdqa xmm3,xmm7 // lower byte mask
   pandn xmm3,xmm2 // pick off upper bytes (using not of mask)
   pmullw xmm2,xmm6 // multiplier A
   pand xmm2,xmm7 // lower byte mask
   pmullw xmm3,xmm6 // note LSB is zero, and will stay zero
   por xmm2,xmm3 // merge bytes back
   movd eax,xmm2 // pextrw eax,xmm2,0
   pextrw ebx,xmm2,2
   movzx esi,al // 0
   mov ebp,[SSE_TABLES+8*esi+edi]
   pextrw ecx,xmm2,5
   movzx esi,bh // 5
   xor ebp,[SSE_TABLES+3+8*esi+edi]
   pextrw edx,xmm2,7
   movzx esi,cl // 10
   xor ebp,[SSE_TABLES+2+8*esi+edi]
   movzx esi,dh // 15
```

```
    xor ebp,[SSE_TABLES+1+8*esi+edi]
    movzx esi,ah // last row, 1
    movd xmm0,ebp
    mov ebp,[SSE_TABLES+3+8*esi+edi]
    movzx esi,ch // last row, 11
    xor ebp,[SSE_TABLES+1+8*esi+edi]
    pextrw ecx,xmm2,4 // pextrw ecx,xmm2,4 // load 8,9
    movd xmm4,ebp // stop last row for now
    movzx esi,bl // 4
    mov ebp,[SSE_TABLES+8*esi+edi]
    shr eax,16 // finish pextrw 2, 3 via movd
    movzx esi,ch // 9
    xor ebp,[SSE_TABLES+3+8*esi+edi]
    movzx esi,dl // 14
    xor ebp,[SSE_TABLES+2+8*esi+edi]
    pextrw ebx,xmm2,3 // load 6,7
    movzx esi,ah // 3
    xor ebp,[SSE_TABLES+1+8*esi+edi]
    pextrw edx,xmm2,6 // load 12,13
    movd xmm1,ebp // 2nd row done
    movzx esi,cl // 8
    mov ecx,[SSE_TABLES+8*esi+edi]
    unpcklps xmm0,xmm1 // combine first two rows
    movzx esi,dh // 13
    xor ecx,[SSE_TABLES+3+8*esi+edi]
    movzx esi,al // 2
    xor ecx,[SSE_TABLES+2+8*esi+edi]
    movzx esi,bh // 7
    xor ecx,[SSE_TABLES+1+8*esi+edi]
    movd xmm1,ecx // 3rd row done
    movzx esi,dl // 12
    movd xmm3,[SSE_TABLES+8*esi+edi]
    pxor xmm4,xmm3
    movzx esi,bl // 6
    movd xmm3,[SSE_TABLES+2+8*esi+edi]
    pxor xmm4,xmm3
    unpcklps xmm1,xmm4 // combine 3rd and 4th rows in xmm3
    unpcklpd xmm0,xmm1 // combine all rows in xmm2
    pxor xmm0,[SSE_KEY_OFFSET+2*16+edi] // key addition
// begin round 3
```