Cache Based Power Analysis Attacks on AES

Jacques Jean-Alain Fournier, Michael Tunstall

Cache Based Power Analysis Attacks on AES

Jacques Fournier[1,2] and Michael Tunstall[3]

[1] University of Cambridge, Computer Laboratory,
William Gates Building, JJ Thomson Avenue,
Cambridge CB3 0FD, UK
jacques.fournier@cl.cam.ac.uk
[2] Gemplus Card International, Security Technologies Department,
Avenue des Jujubiers, La Ciotat, F-13705, France.
jacques.fournier@gemplus.com
[3] Smart Card Centre, Information Security Group,
Royal Holloway, University of London,
Egham, Surrey TW20 0EX, UK.
m.j.tunstall@rhul.ac.uk

Abstract. This paper describes possible attacks against software implementations of AES running on processors with cache mechanisms, particularly in the case of smart cards. These attacks are based on side-channel information gained by observing cache hits and misses in the current drawn by the smart card. Two different attacks are described. The first is a combination of ideas proposed in [2] and [11] to produce an attack that only requires the manipulation of the plain text and the observation of the current. The second is an attack based on specific implementations of the **xtime** function [10]. These attacks are shown to also work against algorithms using Boolean data masking techniques as a DPA countermeasure.

1 Introduction

Several attacks have been published on using cache access events as a side-channel [2, 3, 11, 16] on DES and AES. These are predominately timing attacks taking into account the total number of cache misses in the algorithm to determine information on the secret key being used. The use of a side-channel to analyse the pattern of cache accesses is described in [12].

More recently, an attack was published using the cache lines accessed at each table look-up in the ByteSub function of an AES implemented on a PC to derive the secret key [11]. This involves detecting what cache lines are used for every table look-up to derive the secret key. This was done by having a separate process running in parallel to observe the change in the cache after each table look-up. As each process is sharing the same cache, the changes in the lines could be directly observed by the attacking process.

Attacks using the change in current signature caused by a cache miss have also been published [2] but only part of the key could be obtained. The amount of

information on the key that is derived is determined by the size of the cache lines i.e. the larger the cache lines the smaller the amount of information available.

In this paper, an extension to the attack presented in [2] is proposed. This improvement allows the entire AES key to be derived, and focuses on the cache hit event rather than the cache misses. The attack is somewhat similar to the attack described in [11], but less information is retrieved in the initial steps as the exact cache lines used are unknown. The resulting attack requires no manipulation of the cache as required in [2, 11] but involves manipulating the plain text and observing the corresponding patterns of cache hits generated. Furthermore, another attack is described based on optimisations (for performance and security reasons) used for the **xtime** function [10]. Both attacks are extended to show that these attacks are a realistic threat to DPA resistant algorithms that just use Boolean data masking to protect against DPA [6]. These attacks are described in the context of smart cards, as smart card chips are available that use a cache for data and code accesses e.g. [7, 15], and the power consumption is a readily available side-channel in smart cards.

The remainder of the paper is structured as follows: Section 2 provides some details about cache mechanisms, while Section 3 explains how the latter mechanisms influence the current to briefly describe the side-channel model. Section 4 describes the first part of the attack where roughly half of the secret key can be derived. Section 5 shows how the rest of the key can be derived by two separate methods. Section 6 illustrates how these attacks can be adapted so that they can be applied to DPA resistant algorithms. Section 8 describes some suitable countermeasures, which is followed by a conclusion.

Notation: Throughout this paper the algorithm under attack will be AES where a plaintext $P = (p_1, p_2, p_3 \dots p_{16})_{256}$ is enciphered with a secret key $K = (k_1, k_2, k_3 \dots k_{16})_{256}$. Where the subscript 256 means that the values are to this base. This notation is used throughout this paper e.g. $\texttt{F0}_{16}$ is 240 written in base 16.

2 Cache Description

The shrink of technologies along with the growing need for more sophisticated applications is currently generating a significant shift in the hardware platforms used in smart cards, which have traditionally been based on 8-bit CISC-like CPUs. More sophisticated smart cards are emerging based on 32-bit CPUs containing dedicated peripherals (cryptographic co-processors, memory managers, large memories ...) [7, 15]. Such CPUs are optimised to achieve high performance involving dedicated mechanisms that are implemented to compensate for time consuming operations or long data paths. Details about those sophisticated mechanisms can be obtained from [5, 8]. In order to understand the attacks presented in this paper, we focus on two of these mechanisms, namely pipelining and caching.

Pipelining: Pipelining is a technique whereby the execution of each instruction is decomposed into elementary and independent steps. Each step is implemented as a separate hardware block that can work in parallel. Typically, a 3-stage pipeline can be decomposed into an Instruction Fetch (IF), an Execute (EX) stage and a Write Back (WB) stage. More sophisticated 5-stage pipelines like [8] can involve an IF stage, a DC (Decode) stage, an EX stage, a MEM (Memory access) stage and a WB stage. Each stage is designed to be completed within one clock cycle, which means that even if each instruction takes 5 clock cycles, as in the case of a 5-stage pipeline, an instruction can be issued at every clock cycle.

Caching: Smart card architectures include embedded Non-Volatile Memories (NVM) like EEPROM or Flash to store code or data. The memories usually have high read latencies where, for example, reading one byte involves reading a whole line that takes several clock cycles.This would mean that the IF stage and the MEM stage would take more than one clock cycle, which would stall the pipeline. This would considerably reduce the rate in which instructions can be issued. To compensate for these 'slow' memories, cache mechanisms are implemented. A cache is a small, fast RAM memory whose role is to buffer the lines of NVM being fetched. Due to their technology and small size (leading to faster decode and access times) caches allow a word to be fetched in one clock cycle.

When the data or instruction word is to be fetched from the NVM, the CPU will first check whether this particular word is already in the cache: if yes (this is a *cache hit*), the word is fetched directly from the cache. If, on the contrary, this particular word is not cached this is a *cache miss*. The CPU will then fetch a whole line (e.g. 16 bytes) within which the targeted word is found. This means that even if fetching this word takes more than one clock the other words of this line will already be in the cache when required.

This mechanism considerably increases the instruction issue rate and therefore performance. On Harvard architectures, the cache is applied to both the instruction and data memories in separate caches. Detailed studies of the performance enhancements of cache mechanisms can obtained from [5]. In order to keep power consumption low smart card CPUs usually only implement one level of cache with a granularity in the order of 8 to 16 bytes.

3 The Side Channel

Given the above description of the cache mechanism, we can easily see that in the case of a cache hit the pipeline is not stalled and normal execution occurs. In the case of a cache miss the pipeline flow is stalled and the NVM is accessed. In terms of side-channel information leakage (namely the power consumption) when reading data from memory:

- In the case of a cache miss, the instruction takes more cycles than a cache hit.

– In the case of a cache miss, the power consumed by the execution is significantly higher than in the case of a cache hit because NVM accesses should consume more power than a normal CPU.

With these observations we can build a power analysis attack based on the distinctive signatures of cache hits and cache misses.

The rest of the paper details a method of using this model to build an attack on AES based on cache hits and cache misses.

In our description, the first assumption is that we have a pipelined CPU embedding a one level cache mechanism for both the instructions and data. An example of a hardware simulation of this side-channel is given in [2]. To simplify our illustration, we suppose that on the architecture being attacked the NVM is accessed by lines of 16 bytes i.e. each cache miss will mean that 16 bytes are loaded into the cache.

4 The First ByteSub Function

Our attack is implemented against the AES algorithm as described in [10]. The first step of the attack targets the ByteSub function of the first round. Just before entering this function the input data is XORed with the secret key. The resulting 16 bytes enter the ByteSub function that is usually implemented as a look-up on a table of 256 entries.

4.1 The Power Consumption

An attack on this function is already described in [2]; a slightly modified version is stated here. The main difference is this attack relies purely on the observation of the side-channel described in Section 3, whereas the attack described in [2] manipulates the cache. Less information is generated but the attack is more powerful as it only needs to manipulate the messages being ciphered and observe the cache access pattern generated.

Key information can be derived from the cache access events during the table look-up depending on the order in which the look-up table is loaded into the cache. It is assumed that for each acquisition the cache has been flushed, which can easily be provoked by resetting the smart card under observation.

The first byte of the message is fixed to a value, p_1, and different values of the second byte of the message, p_2 can be tried until a cache hit occurs. At which point it is known that $p_1 \oplus k_1 \approx p_2 \oplus k_2$, which is only an approximation due to the size of the cache granularity. In the case under study (i.e. we have a cache with a granularity of 16 bytes) we can only be sure of the high nibble of the approximation given. Therefore $(p_1 \oplus p_2) \wedge \mathtt{F0}_{16}$ will give $(k_1 \oplus k_2) \wedge \mathtt{F0}_{16}$ with at most sixteen different messages i.e. all sixteen possible values for the high nibble of p_2 can be tried until a cache hit is observed.

Once $(k_1 \oplus k_2) \wedge \mathtt{F0}_{16}$ is found, $(k_2 \oplus k_3) \wedge \mathtt{F0}_{16}$ can be found using the same method by choosing p_1 and p_2 so that a cache hit is always generated between

the first two look-ups, and varying p_3 until another cache hit is generated. It is important to have a cache hit between the first two look-ups, as otherwise it is not known which cache line corresponds to the observed cache hit and some information is lost. If this process is repeated for each subsequent key byte, the high nibble of each byte will be known as a function of the high nibble of the first byte. With at most 240 acquisitions the exhaustive search to find the key of an AES implementation can be reduced from 2^{128} to 2^{68}.

In practice, this will only be true if the implementation is known. The ByteSub function can be implemented before or after the ShiftRow function, as the ShiftRow function is a bytewise permutation. A permutation is sometimes also used on the message and key on entry to the algorithm to convert the array format to the grid format used in the specification [10]. This is an optional bytewise permutation that will change addressing during the algorithm. Both permutations will change the order in which the data is treated by the ByteSub function.

In the following sections we will assume that the implementation details are known, as the added complexity due to these permutations is negligible. The grid permutation will be ignored and the ShiftRow function will be assumed to take place after the ByteSub function.

5 Finding the Rest of the Key

The first step described in Section 4 reduces the keyspace to 2^{68} and is theoretically trivial. There are two ways to continue the attack to derive the rest of the key using the same side-channel. These two independent methods are described below.

5.1 The Second ByteSub Function

The second ByteSub function (i.e. the ByteSub of the second round of AES) can be used to determine the rest of the key in a similar manner to that described in [11], and the same notation has been used for clarity. Plain texts are chosen such that there are no cache misses in the first ByteSub function, except for the first table look-up. The plain text bits that are XORed with the unknown bits of the key (i.e. the first byte and the lower nibbles of the rest of the plain text) are randomised for each acquisition. If the first look-up in the second ByteSub function is a cache hit then information on the unknown key bits can be derived. In this case the following relationship is known:

$$\begin{aligned}(2 \bullet s(p_1 \oplus k_1) \oplus 3 \bullet s(p_6 \oplus k_6) \oplus s(p_{11} \oplus k_{11}) \\ \oplus s(p_{16} \oplus k_{16}) \oplus k_1 \oplus s(k_8) \oplus 1) \wedge \mathtt{F0}_{16} = (k_1 \oplus k_2) \wedge \mathtt{F0}_{16}\end{aligned}$$

Where the function $s(\cdot)$ represents the look-up table used in the ByteSub function and $\bullet$ represents multiplication over GF(2^8).

The value of $(k_1 \oplus k_2) \wedge \texttt{F0}_{16}$ is known from the first part of the attack described in Section 4. The value of k_1 is unknown but given k_1 the high nibbles of k_6, k_{11}, k_{16} and k_8 can be derived. This means that there are 24 unknown bits in the equation. The evaluation of the 2^{24} possible combinations of the left hand side of the equation will be equal to $(k_1 \oplus k_2) \wedge \texttt{F0}_{16}$ with a frequency of 1 in 16. One plaintext that produces a cache hit in the second ByteSub function will therefore reduce the unknown bits in the equation from 2^{24} to 2^{20}.

A second cache hit with a different plain text can then be analysed, the correct key values will be in the intersection of the two sets of 2^{20} values produced. With 6 evaluations of the above equation all the unknown bits can be derived. This corresponds to 96 acquisitions, as the cache hit occurs with a probability of 1/16 given that the plaintext input is mostly random. This reduces the unknown key bits from 2^{68} to 2^{44}.

The cache misses could also be used as they would reduce the keyspace by 15/16, but given the small amount of acquisitions required this should not be necessary.

Any acquisition with two successive cache hits can then be used to derive information on another 5 key bytes. If the second look up in the second ByteSub function is also a cache hit, the following equation holds.

$$(2 \bullet s(p_2 \oplus k_2) \oplus 3 \bullet s(p_7 \oplus k_7) \oplus s(p_{12} \oplus k_{12}) \\ \oplus s(p_{13} \oplus k_{13}) \oplus k_2 \oplus k_1 \oplus s(k_8) \oplus 1) \wedge \texttt{F0}_{16} = (k_1 \oplus k_2) \wedge \texttt{F0}_{16}$$

It is faster to search through the possible values of this equation as there are 20 unknown bits, the values of k_1 and k_8 being provided by the previous step. As previously, the evaluation of this equation reduces the unknown values by a factor of 16. It is expected that 5 such equations need to be evaluated, taking the intersection as before, to provide one value for all of the key bytes in the equation. This event occurs with a probability of 1/256 so the acquisition phase will be lengthier than the previous step. A total of 1280 acquisitions should be required.

If all of the key bytes in the above equations are derived, the key can then be found by an exhaustive search of the remaining unknown key bits. This will be a search in a keyspace of size 2^{24} (i.e. 9 complete key bytes are given by the formulae above, for the remaining six the high nibble is known, leaving 24 unknown bits), which can easily be exhausted on a PC. This is fortunate as continuing the attack for three successive cache hits would be difficult as the probability of seeing such an event is 1/4096, which would make the attack excessively time consuming.

The last set of equation evaluations are time consuming, which means that it can be advantageous to acquire less data and let the exhaustive search complete the key search. If, for example, an attacker takes 768 acquisitions the expected exhaustive key search would be around 2^{32}. Another means of speeding up the evaluation of the possible key values for the second equation would be to use the event of a cache hit followed by a cache miss, which occurs with a probability of 15/256, each of which will reduce the unknown keyspace by 15/16.

5.2 The xtime Function

A second method to reduce the key search space is to focus on the **xtime** function. The **xtime** function is a multiplication by 2 over GF(2^8) and is used in the MixColumn function as shown in Algorithm 1.

Algorithm 1: The MixColumn function

Input: $X = (x_0, x_1, \ldots, x_{15})_{256}$
Output: $Y = (y_0, y_1, \ldots, y_{15})_{256}$

for $i \leftarrow 0$ **to** 15 **do**
    $y_i \leftarrow \mathbf{xtime}(x_i) \oplus \mathbf{xtime}(x_{(i+4) \mod 16}) \oplus x_{(i+4) \mod 16}$
    $y_i \leftarrow y_i \oplus x_{(i+8) \mod 16} \oplus x_{(i+12) \mod 16}$
end

return Y

The **xtime** function is a bit shift followed by a conditional XOR (as shown in Algorithm 2). This is difficult to implement securely in smart cards as there is a danger that the result of the conditional test can be leaked through the power consumption as the two branches will take different amounts of time to complete. Even if this is implemented so that the calculation always takes the same amount of time, there is still a risk of a partitioning attack [14].

Algorithm 2: The **xtime** function

Input: $x = (x_7, x_6, \ldots, x_0)_2$
Output: $y = \mathbf{xtime}(x)$

$y \leftarrow (x << 1) \wedge \mathrm{FF}_{16}$
if $x_7 = 1$ **then**
    $y \leftarrow y \oplus \mathrm{1B}_{16}$
end

return y

In smart cards a possible replacement for this function is with a look-up table of 256 bytes to avoid any conditional testing. This protects the implementation against Simple Power Analysis but the table will be in Non-Volatile Memory so will be accessed via the cache as with the look-up table used in the ByteSub function. The pattern of cache hits and misses can therefore be analysed in a similar way to the first phase of the attack described in ection 4. The first look-up to the **xtime** table will be a cache miss, if this is followed by a cache hit then:

$$s(p_1 \oplus k_1) \wedge \mathrm{F0}_{16} = s(p_6 \oplus k_6) \wedge \mathrm{F0}_{16}$$

Where, as previously, the $s(.)$ represents the look-up table in the ByteSub function. The right hand side of the equation uses $p_6 \oplus k_6$ rather than $p_4 \oplus k_4$, as defined in Algorithm 1, due to the ShiftRow function. In this equation there are 2^{12} possible combinations given that the high nibble of k_6 is known as a function of k_1, from the first part of the attack described in section 4. Searching through all the combinations will give 2^8 possible values for the pair (k_1, k_6). Due to the non-linear nature of the $s(\cdot)$ function another cache hit can be found with a different message that will provide a different set of 2^8 values. The correct key will be in the intersection between the two sets of possible values. After three cache hits with three different messages are found there should only be one hypothesis for both k_1 and k_6. Each cache hit will occur with a probability of 1/16, so 48 acquisitions should be enough to find the value of k_1 and k_6.

The next cache access is the first **xtime** function call for the next output byte. The values for p_1 and p_6 can be fixed so that a cache hit is always generated between the first two **xtime** look-ups. If a cache hit occurs for the next **xtime** look-up then:

$$s(p_6 \oplus k_6) \wedge \mathrm{F0}_{16} = s(p_2 \oplus k_2) \wedge \mathrm{F0}_{16}$$

In this case k_6 is known and the high nibble of k_2 is known, as k_1 has been determined the high nibble of all the key bytes are known. The 4 unknown bits of k_2 in the equation can be exhausted for the value of p_2 that provokes a cache hit. One cache hit of this nature would be enough to determine the 4 unknown bits. This process can be continued with the following equations:

$$\begin{aligned}
s(p_2 \oplus k_2) \wedge \mathrm{F0}_{16} &= s(p_7 \oplus k_7) \wedge \mathrm{F0}_{16} \\
s(p_7 \oplus k_7) \wedge \mathrm{F0}_{16} &= s(p_3 \oplus k_3) \wedge \mathrm{F0}_{16} \\
s(p_3 \oplus k_3) \wedge \mathrm{F0}_{16} &= s(p_8 \oplus k_8) \wedge \mathrm{F0}_{16} \\
s(p_8 \oplus k_8) \wedge \mathrm{F0}_{16} &= s(p_4 \oplus k_4) \wedge \mathrm{F0}_{16} \\
s(p_4 \oplus k_4) \wedge \mathrm{F0}_{16} &= s(p_5 \oplus k_5) \wedge \mathrm{F0}_{16}
\end{aligned}$$

This can determine the first 8 bytes of the key with 192 acquisitions, leaving an exhaustive search of 2^{32} possible keys. An exhaustive search of 2^{32} is prohibitive so further analysis would be advantageous. The next set of possible cache hits follow the equations:

$$\begin{aligned}
s(p_6 \oplus k_6) \wedge \mathrm{F0}_{16} &= s(p_{11} \oplus k_{11}) \wedge \mathrm{F0}_{16} \\
s(p_7 \oplus k_7) \wedge \mathrm{F0}_{16} &= s(p_{12} \oplus k_{12}) \wedge \mathrm{F0}_{16} \\
s(p_8 \oplus k_8) \wedge \mathrm{F0}_{16} &= s(p_9 \oplus k_9) \wedge \mathrm{F0}_{16} \\
s(p_5 \oplus k_5) \wedge \mathrm{F0}_{16} &= s(p_{10} \oplus k_{10}) \wedge \mathrm{F0}_{16}
\end{aligned}$$

There will be no need to compare $s(p_{11} \oplus k_{11})$ with $s(p_7 \oplus k_7)$, as if a cache hit is generated between $s(p_6 \oplus k_6)$ and $s(p_{11} \oplus k_{11})$ a cache hit will also be generated with $s(p_7 \oplus k_7)$ due to the selected message.

Acquiring data from these formulae requires a further 64 acquisitions (for a total of 256 acquisitions) and reduces the amount of unknown key bits to 16.

As an exhaustive search of 2^{16} is trivial, no further acquisitions are required to derive the key.

6 Application to DPA Resistant Implementations

In smart cards implementations of cryptographic algorithms like AES are implemented with countermeasures to protect against Differential Power Analysis (DPA) [6]. One of the techniques used to protect the AES is by masking the data being manipulated with a random value. The data is then manipulated in such a way that the value present in memory is always masked with the same random. This mask is then removed at the end of the algorithm to produce the ciphertext. The most common form of masking is Boolean masking where all data manipulated is treated after being XORed with a random, such that the result is also XORed with the same random value. An example of this sort of implementation can be found in [1].

The size of the random is generally limited as look-up tables need to be randomised before the execution of the algorithm so that the input and output values of the s-box leak no information. An example of how this is done is given in Algorithm 3. As illustrated in the latter, the random used for masking the input data can be no larger than n, and the random used for the output value can be no larger that x.

Algorithm 3: Randomising S-Box Values

Input: $S = (s_0, s_1, s_2, \ldots, s_n)_x$ containing the s-box, $\mathbf{R}$ a random $\in [0, n]$, and r a random $\in [0, x)$.

Output: $RS = (rs_0, rs_1, rs_2, \ldots, rs_n)_x$ containing the randomised s-box.

for $i \leftarrow 0$ **to** n **do**
    $rs_i \leftarrow s_{(i \oplus \mathbf{R})} \oplus r$
end

return RS

In the case of AES both $\mathbf{R}$ and r are on one byte, which means that the random mask during the calculation of AES will also be on one byte.

6.1 Implementing the Attack

The described attack can be implemented as described in the above sections, as the random will provide one byte of variation. In all the equations used to test key hypotheses, the values generated are always compared with the neighbouring byte. If, for example, all bytes in the algorithm are masked with the random $\mathbf{R}$ the first phase of the attack described in section 4 will give $(k_1 \oplus \mathbf{R} \oplus k_2 \oplus \mathbf{R}) \wedge \mathtt{F0}_{16}$. The $\mathbf{R}$'s will cancel leaving $(k_1 \oplus k_2) \wedge \mathtt{F0}_{16}$ as with the approach detailed in section 4. The random will just change the order of the cache lines and the order

of the bytes within them, but the same plaintext values will give the same cache access pattern.

This does not mean that a DPA resistant algorithm is as easy to attack as a naive implementation. There will be an initialisation phase during the algorithm execution where the look-up table for the ByteSub function is randomised and written into RAM, as described in Algorithm 3. In order for the cache to reveal information as described above, enough time needs to have passed between the execution of Algorithm 3 and the ciphering algorithm so that the cache no longer contains the randomised look-up table. In theory, it may be possible to apply the attack in [3] but it is necessary to know the cache lines that no longer contain the randomised look-up table.

6.2 The xtime Function

The attack described in Sections 4 and 5 can work against a DPA resistant algorithm assuming the randomised look-up table is no longer present in the cache, but this assumption is probably not reasonable. It would be simpler to directly attack the **xtime** function instead of the ByteSub function. The **xtime** function has the property that if $y = \mathbf{xtime}(x)$ then $y \oplus \mathbf{R} = \mathbf{xtime}(x \oplus \mathbf{R})$ for $\mathbf{R} \in [0, 255]$ i.e. the data mask will carry across the **xtime** function. This means that there is no need to load the **xtime** table into RAM in a DPA resistant implementation of AES.

In this case the attack described in Section 5.2 can be extended to recover all of the key data rather than just the first byte and the lower nibbles. The first equation for a cache hit between the first and second **xtime** look-up becomes:

$$(s(p_1 \oplus k_1) \oplus \mathbf{R}) \wedge \mathrm{F0}_{16} = (s(p_6 \oplus k_6) \oplus \mathbf{R}) \wedge \mathrm{F0}_{16}$$
$$s(p_1 \oplus k_1) \wedge \mathrm{F0}_{16} = s(p_6 \oplus k_6) \wedge \mathrm{F0}_{16}$$

In this case there are 16 unknown bits and an evaluation will reduce the keyspace by a factor of 16. After four evaluations of this equation a single solution can be found for the pair (k_1, k_6). This cache hit event occurs with a probability of 1/16 for a random plain text. An attack therefore requires a maximum of 64 acquisitions before being able to derive the key byte.

The attack can continue in the same manner as the attack described in Section 5.2 but the total attack will require around 480 acquisitions and an exhaustive search of 2^{16} to derive the entire key.

7 Countermeasures

Several countermeasures can provide a protection against this attack in smart cards. These are:

Programming Instructions: On some architectures the caching of data can be avoided by fetching data without caching it. Such instructions do incur performance penalties but they have the advantage of always taking the same amount of time to execute.

Random Delay: The use of dummy code in cryptographic algorithms is a common countermeasure used to prevent side-channel attacks. Such mechanisms lower the signal-to-noise ratios of such side-channels, thus adding another level of difficulty to the implementation of this attack. A discussion of this effect is given in [4], further discussion in the specific context of side-channel attacks on cache access patterns appears in [13].

Random Order: If all the functions are conducted in a random order it will not be possible to determine any relationship between a cache hit/miss and the actual values being manipulated, which can either be implemented in hardware [13] or software [9].

An example of this is given in [9] for copying 256 bytes from buffer A to buffer B and is detailed in Algorithm 4. The same principle can be applied to the loop in the ByteSub and MixColumn function so that an attacker does not know which of the 16! possible combinations have been acquired.

In an actual DPA resistant implementation this countermeasure would be expected, as it renders power attacks exceedingly difficult especially when combined with data masking.

Algorithm 4: Random Order Data Copying

Input: $A = (a_0, a_1, a_2, \ldots, a_{255})_{256}$, $\{x, y, z, w\}$ four random bytes (x odd).
Output: $B = (b_0, b_1, b_2, \ldots, b_{255})_{256}$.

for $n \leftarrow 0$ **to** 255 **do**
  $i \leftarrow (x \times (n \oplus w) + y \pmod{256}) \oplus z$
  $b_i \leftarrow a_i$
end

return B

Calculating the xtime function: On a 32-bit architecture, the **xtime** operation can be computed without a performance penalty compared to the table look-up implementation. On an assembly instruction level, the table look-up implementation of the **xtime** would be as illustrated by Algorithm 5 where the implementation takes $4 \times 16 = 64$ instruction cycles.

On a 32-bit architecture Algorithm 6 can be implemented, which not only avoids any memory accesses but may be faster on a 32-bit platform as the operation would take $8 \times 4 = 32$ instruction cycles. The side-channel issues concerning the visibility of the most significant bit of each byte is less of an issue as four bytes are being manipulated separately.

A more complete discussion of the countermeasures for protecting algorithms against attacks using a side-channel to observe cache accesses is given in [13].

Algorithm 5: Table Look-up implementation of **xtime**

Input: $A = (a_0, a_1, a_2, \ldots, a_{15})_{256}$, $X = (x_0, x_1, x_2, \ldots, x_{255})_{256}$ table for **xtime** look-up.
Output: $B = (b_0, b_1, b_2, \ldots, b_{15})_{256}$.

for $i \leftarrow 0$ **to** 15 **do**
    LOAD a_i
    $j \leftarrow X_{a_i}$
    LOAD x_j
    STORE $b_i \leftarrow x_j$
end

return B

Algorithm 6: Calculating **xtime**

Input: $A = (a_0, a_1, a_2, \ldots, a_{15})_{256}$.
Output: $B = (b_0, b_1, b_2, \ldots, b_{15})_{256}$.

for $i \leftarrow 0$ **to** 3 **do**
    LOAD $R_1 \leftarrow (a_{4i}, a_{4i+1}, a_{4i+2}, a_{4i+3})$
    $R_2 \leftarrow R_1 \wedge 80808080_{16}$
    $R_2 \leftarrow R_2 >> 7$
    $R_3 \leftarrow R_2 * 1\mathrm{B}_{16}$
    $R_1 \leftarrow R_1 << 1$
    $R_1 \leftarrow R_1 \wedge \mathrm{FEFEFEFE}_{16}$
    $R_1 \leftarrow R_1 \oplus R_3$
    STORE $(b_{4i}, b_{4i+1}, b_{4i+2}, b_{4i+3}) \leftarrow R_1$
end

return B

8 Conclusion

In this paper we propose an attack against software AES implemented on a smart card with cache mechanisms. Our attack is based on the observation of the power consumption information leakage generated by the different mechanisms behind the caching techniques. We first explain how cache events generate different side-channel signatures, before showing how varying the input message on the first round can be combined with this observation to reduce the AES key search space from 2^{128} to 2^{68}.

We propose two alternatives to find the remaining key bits either by focussing on cache events during a ByteSub operation of the second AES round, or by targeting the **xtime** of the MixColumn operation in the first round. Furthermore, we argue that these attacks are also valid against implementations where masking techniques are implemented as a countermeasure against DPA-like attacks.

This shows that when implementing cryptography on a given processor, the specificities of this processor must be taken into account in order to have a secure implementation. Caches are highly important features in high performance embedded processors but they need to be carefully used when executing cryptographic algorithms like AES.

References

1. M.-L. Akkar and C. Giraud. An implementation of DES and AES secure against some attacks. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptogaphic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 309–318. Springer-Verlag, 2001.
2. D. J. Bernstein. Cache timing attacks on AES, 2004. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.
3. G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo. AES power attack based on induced cache miss and countermeasures. In *International Symposium on Information Technology: Coding and Computing – ITCC 2005*, pages 586–591. IEEE Computer Society, 2005.
4. C. Clavier, J.-S. Coron, and N. Dabbous. Differential power analysis in the presence of hardware countermeasures. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 252–263. Springer-Verlag, 2000.
5. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.
6. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.
7. MIPS-Technologies. SmartMIPS ASE. `http://www.mips.com/content/Products/`.
8. MIPS-Technologies. MIPS$^{\text{TM}}$architecture for programmers volume I: Introduction to the MIPS32$^{\text{TM}}$architecture. Technical Report MD00082, Revision 0.95, March 2001.

9. D. Naccache, P. Q. Nguyễn, M. Tunstall, and C. Whelan. Experimenting with faults, lattices and the DSA. In S. Vaudenay, editor, *Public Key Cryptography – PKC 2005*, volume 3386 of *Lecture Notes in Computer Science*, pages 16–28. Springer-Verlag, 2005.
10. National Institute of Standards and Technology. Advanced encryption standard (AES) (FIPS–197), 2001.
11. D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. `http://eprint.iacr.org/2005/271`, 2005.
12. D. Page. Theoretical use of cache memory as a cryptanalytic side–channel. Cryptology ePrint Archive, Report 2002/169, 2002. `http://eprint.iacr.org/`.
13. D. Page. Defending against cache based side-channel attacks. *Information Security Technical Report*, 8(1):30–44, April 2003.
14. J. R. Rao, P. Rohatgi, H. Scherzer, and S. Tinguely. Partitioning attacks: or how to rapidly clone some gsm cards. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 31–41, 2002.
15. Infineon Technologies AG Secure and Mobile Solutions Security Group. Security & chip cards ICs SLE88Cx4000P, preliminary short product information 04.03, 2003.
16. Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. Cryptanalysis of DES implemented on computers with cache. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 62–76. Springer-Verlag, 2003.