Simple Power Analysis on AES Key Expansion Revisited

Christophe Clavier, Damien Marion, and Antoine Wurcker

Université de Limoges, XLIM-CNRS
Limoges, France
{christophe.clavier,damien.marion}@unilim.fr,
antoine.wurcker@xlim.fr

Abstract. We consider a simple power analysis on an 8-bit software implementation of the AES key expansion. Assuming that an attacker is able to observe the Hamming weights of the key bytes generated by the key expansion, previous works from Mangard and from VanLaven et al. showed how to exploit this information to recover the key from unprotected implementations.

Our contribution considers several possible countermeasures that are commonly used to protect the encryption process and may well be adopted to protect the computation and/or the manipulation of round keys from this attack. We study two different Boolean masking countermeasures and present efficient attacks against both of them. We also study a third countermeasure based on the computation of the key expansion in a shuffled order. We show that it is also possible to attack this countermeasure by exploiting the side-channel leakage only. As this last attack requires a not negligible computation effort, we also propose a passive and active combined attack (PACA) where faults injected during the key expansion are analyzed to derive information that render the side-channel analysis more efficient. These results put a new light on the (in-)security of implementations of the key expansion with respect to SPA.

As a side contribution of this paper, we also investigate the open question whether two different ciphering keys may be undistinguishable in the sense that they have exactly the same set of expanded key bytes Hamming weights. We think that this problem is of theoretical interest as being related to the quality of the diffusion process in the AES key expansion. We answer positively to this open question by devising a constructive method that exhibits many examples of such ambiguous observations.

Keywords: side-channel analysis, simple power analysis, passive and active combined attacks, AES key expansion.

1 Introduction

Side channel analysis is an effective means to derive secrets stored in a security device like a smart card from measurements of a leaking physical signal such as the execution duration, the power consumption or the electromagnetic emanation. Since the first publication of a timing attack by Kocher [6] many side-channel analysis methods have been presented that exploit a large number of leakage traces by a statistical method: Differential Power Analysis [7], Correlation Power Analysis [2], Mutual Information Analysis [4] and Template Analysis [3] are few such well known methods.

Simple Power Analysis (SPA) also permits to infer information in a more direct manner by "visually" inspecting a single (in the most favorable cases) trace. Two kinds of information can be retrieved by SPA. At a high level it allows to recognize different instructions or blocks of instructions that are executed on the device. This capability is typically exploited either to recover a sequence of arithmetic operations of a modular exponentiation used in public key cryptography, or for a rough reverse engineering and/or a first characterization phase of an implementation or of the leakage behavior of the device. At a lower level SPA informs about the values of the operands involved in each elementary instruction particularly for load and store operations when this data is read from or written to the bus. The dependency between the value of a data and that of the power consumption that leaks when it is manipulated has early been studied [11,9,10] and in the classical models the power consumption is tightly linked either with the Hamming weight of the data or with the Hamming distance between this data and the value it replaces on the bus.

In this paper we consider an attacker that is able to infer the Hamming weights of the data manipulated by targeted instructions of a software AES implementation on an 8-bit microprocessor. Specifically the targeted data are the different bytes of the different round keys, while the targeted instructions may be located either in the AES key expansion process which computes these round keys, or in the `AddRoundKey` function which XOR the round keys with the current state of the encryption process. While the problem of inferring an AES key from the Hamming weights and the expanded key bytes has first been mentioned in [1], Mangard [8] was the first to describe such an attack which has later been improved by VanLaven et al. [13]. While the SPA on the AES key expansion described in these works only apply on naive unprotected implementations, we study in this paper to which extent this attack may be adapted to implementations featuring side-channel countermeasures. We consider three different scenarios where either a Boolean masking is applied to the round keys or the order of computation of the expanded key bytes is randomly shuffled. The masking countermeasure prevents the attacker from obtaining the Hamming weight of actual key bytes, while the shuffling countermeasure prevents him to precisely know to which key byte an observation is related.

The paper is organized as follows: The problem statement and a background on the related previous works are presented in Sect. 2. This section also considers the open problem whether two expanded keys may have the same of Hamming weights. Section 3 presents our main contribution where we describe attacks on three countermeasures. In the light of these results we give implementation recommendations in Sect. 4 while Sect. 5 concludes this work.

2 Problem Statement and Previous Work

Given a 16-byte ciphering key K, the AES key expansion derives eleven 16-byte round keys K_r ($r = 0, \dots, 10$) with $K_0 = K$ and where individual bytes of K_r are denoted $k_{r,i}$ ($i = 0, \dots, 15$).

The expanded key $\overline{K} = \{K_0, \dots, K_{10}\}$ is computed column by column by means of two types – a linear and a non-linear – of relations:

$$k_{r,i} = k_{r-1,i} \oplus k_{r,i-4} \quad \text{(for } i = 4, \dots, 15\text{)} \tag{1}$$

$$k_{r,i} = k_{r-1,i} \oplus \mathrm{S}(k_{r-1,12+((i+1) \bmod 4)}) \oplus c'_r \quad \text{(for } i = 0, \dots, 3\text{)} \tag{2}$$

where S is the S-Box substitution and c'_r is a round specific constant equal to $\{\mathtt{02}\}^{r-1}$ if $i = 0$ and equal to 0 if $i \in \{1, 2, 3\}$. We refer the interesting reader to the AES specifications [12] for further details on the AES ciphering process.

The problem considered in this paper is how to identify the ciphering key K based on a set $\{\mathrm{HW}(k_{r,i})\}_{r,i}$ of part or all Hamming weights of the expanded key bytes.

Mangard [8] was the first to give a solution to this problem. He proposed to build lists of values of 5-byte key parts which are both compatible with the observed Hamming weights of these bytes, and also compatible with the Hamming weights of 9 other key bytes (and several other intermediate bytes) that can be computed from the 5-tuple.

In [13] VanLaven et al. also consider the same problem and give an elegant analysis of the key byte links which allows them to derive an efficient guess-compute-and-backtrack algorithm where a sequence of key bytes are successively guessed in an optimal order that maximizes the number of other bytes that can be computed and checked with respect to their Hamming weight. Once an inconsistency with respect to the observations is found the algorithm considers the next possible value for the current guessed byte and eventually backtracks one level back in the sequence of key bytes when all values for the current guessed byte have been considered. Interestingly the last contribution of this work shows that their algorithm can cope with (slightly) erroneous observation at the price of a more demanding computational work in the key space exploration process.

Undistinguishable Keys. We study the open question whether there exist key pairs – or more generally key sets – which are undistinguishable for having the same Hamming weights signatures[1]. We are thus concerned by the existence or non-existence of two different keys K and K' such that $\overline{K}$ and $\overline{K'}$ have exactly the same 176 Hamming weights.

If the AES key expansion was deriving round keys K_1 to K_{10} with an ideal random behavior, the probability that there exist two keys having the same signature would be overwhelming low. Indeed the probability that two random bytes have same Hamming weight is $p = 2^{-2.348}$ so that the probability that the

[1] By *Hamming weights signature* of a key K we mean the set of all the Hamming weights of its expanded key $\overline{K}$.

signatures of two random keys are the same is $q = p^{176} \simeq 2^{-413.3}$. It follows that the probability that at least one collision of signatures occurs among the whole key space is about $1 - e^{-\frac{q}{2} \cdot 2^{2*128}} \simeq 2^{-158.3}$.

While the AES key expansion is far from having a random behavior, it was considered in [8] that so-called twin keys probably do not exist or should be very rare[2]. We show in this paper that this belief is wrong by proposing a constructive method that can easily generate millions of them. We refer the reader to Appendix A for the description of this method and just provide here an example of such key pair:

$$\begin{cases} K = \texttt{B3 65 58 9D B4 EB 57 72 1F 51 F7 58 02 0C 00 17} \\ K' = \texttt{F2 65 19 DC B4 EB 57 33 5E 51 F7 19 02 0C 00 56} \end{cases}$$

Note that the existence of twin keys is of theoretical interest as it gives a new demonstration of the quite non-ideal behavior of the diffusion process of the AES key expansion. Nevertheless, it has no practical impact on the attacks considered in this paper since the only consequence is that when attacking a key belonging to such pair, the attack process ends with two possible keys instead of a unique one. The correct key can then be identified thanks to a known plaintext/ciphertext pair.

3 Key Recovery on Protected Implementations

In this section we study three different countermeasures that may be implemented to protect the key expansion function against simple power analysis.

The first two countermeasures are natural ways to apply a Boolean masking on the expanded key. They make use of 11-byte and 16-byte masks respectively in order to cope with limited RAM resources and/or small random entropy generation capacity that usually prevail on embedded devices. The third countermeasure is a columnwise shuffling of the expanded key computation.

3.1 11-byte Entropy Boolean Masking

We consider here that at each execution all round keys are masked by 11 specific random bytes m_r so that the attacker has no longer access to the leakages of individual bytes $k_{r,i}$ of each K_r but rather to those of masked versions $K'_r = (k'_{r,i})_i$ with $k'_{r,i} = k_{r,i} \oplus m_r$. Figure 1 depicts the mask pattern that applies on the expanded key bytes.

The basic attack does not apply directly since the measured Hamming weights are related to masked bytes that do not verify neither linear nor non-linear links of the key expansion process.

In order to apply the guess-compute-and-backtrack strategy of the basic attack we now have to make also guesses about the values of the masks of all key

[2] The exact sentence of the author was: *The high diffusion of the AES key expansion suggests that there are only very few keys of this kind, if there are such keys at all.*

K_0			
m_0	m_0	m_0	m_0
m_0	m_0	m_0	m_0
m_0	m_0	m_0	m_0
m_0	m_0	m_0	m_0

K_1			
m_1	m_1	m_1	m_1
m_1	m_1	m_1	m_1
m_1	m_1	m_1	m_1
m_1	m_1	m_1	m_1

K_2			
m_2	m_2	m_2	m_2
m_2	m_2	m_2	m_2
m_2	m_2	m_2	m_2
m_2	m_2	m_2	m_2

$\cdots$

Fig. 1. Part of the 8-byte masking scheme

bytes involved in the key search. As each extra mask that must be guessed induces a multiplication by 2^8 of the searched space, we use two tricks to contain the necessary computing work factor.

The first idea is to exploit some extra information that can be inferred about the key bytes $k_{r,i}$ by considering measured Hamming weights from multiple traces. More precisely, consider two bytes x and y masked by the same random value m. The respective Hamming weights of the masked bytes $x' = x \oplus m$ and $y' = y \oplus m$ verify the two following properties:

$$|\,\mathrm{HW}(x') - \mathrm{HW}(y')| \leqslant \mathrm{HD}(x, y) \leqslant \min(8, \mathrm{HW}(x') + \mathrm{HW}(y')) \tag{3}$$

$$\mathrm{HD}(x, y) \equiv \mathrm{HW}(x') + \mathrm{HW}(y') \pmod 2 \tag{4}$$

Both equations give information about the Hamming distance between unmasked values x and y. For example, suppose that $x = 30$ and $y = 121$ (i.e. $\mathrm{HD}(x, y) = 5$). With a first trace for which $m = 70$, we measure $\mathrm{HW}(x') = \mathrm{HW}(30 \oplus 70) = 3$ and $\mathrm{HW}(y') = \mathrm{HW}(121 \oplus 70) = 6$. From Eq. (3) we infer that $3 \leqslant \mathrm{HD}(x, y) \leqslant 8$, and due to the odd parity given by Eq. (4) we learn that $\mathrm{HD}(x, y) \in \{3, 5, 7\}$. With a second trace for which $m = 24$, we measure $\mathrm{HW}(x') = \mathrm{HW}(30 \oplus 24) = 2$ and $\mathrm{HW}(y') = \mathrm{HW}(121 \oplus 24) = 3$. This second measure allows to further constrain $\mathrm{HD}(x, y)$ which now belongs to $\{3, 5\}$. By exploiting more and more traces we can decrease the number of possible candidates and ultimately expect to identify the Hamming distance between the unmasked bytes. Interestingly we notice that the parity equation may be used to detect erroneous measurements. For example, if the measurements from ten traces give an odd parity for $\mathrm{HW}(x') + \mathrm{HW}(y')$ eight times and an even parity only twice, then one may conclude that either $\mathrm{HW}(x')$ or $\mathrm{HW}(y')$ has not been correctly measured on these two last traces.

In a first phase of the attack, multiple traces are analysed in order to get as much possible information about the Hamming distance $\mathrm{HD}(k_{r,i}, k_{r,i'})$ of each couple of bytes belonging to the same round key. Then in a second phase a smart exploration of the key space is performed based on the Hamming weights measured from a unique trace, and on the Hamming distances constraints obtained in the first phase.

The second idea to reduce to computational effort is to limit the process of guessing and computing key bytes to only two adjacent round keys K_r and K_{r+1}. That way we have to guess only two mask bytes. For each (m_r, m_{r+1}) candidate we perform a key search where we guess successive bytes of K_r and derive the values of successive bytes of K_{r+1}. For example, consider that we start the search by guessing $k_{r,12}$ (equivalently we could start at positions 13, 14 or 15). In a first step we guess $k_{r,3}$ and compute $k_{r+1,3}$. In a second step we guess $k_{r,7}$ and compute $k_{r+1,7}$. Then we guess $k_{r,11}$ and compute $k_{r+1,11}$, and so on. Figure 2 shows the order in which successive bytes of K_r and K_{r+1} are respectively guessed and computed. As in the basic attack, each time a key byte is guessed or computed we check the consistency with the measured Hamming weights of its masked values. A more efficient consistency check consists in verifying that each newly guessed or computed byte has compatible Hamming distances with all already known key bytes belonging to the same round key. For example, when $k_{r,11}$ is guessed in the third step four constraints on $\mathrm{HW}(k_{r,11} \oplus m_r)$, $\mathrm{HD}(k_{r,11}, k_{r,7})$, $\mathrm{HD}(k_{r,11}, k_{r,3})$ and $\mathrm{HD}(k_{r,11}, k_{r,12})$ are verified, and when $k_{r+1,11}$ is computed three checks imply $\mathrm{HW}(k_{r+1,11} \oplus m_{r+1})$, $\mathrm{HD}(k_{r+1,11}, k_{r+1,7})$ and $\mathrm{HD}(k_{r+1,11}, k_{r+1,3})$. As we can see, the more deeper we are in the exploration process, the more opportunities we have to invalidate wrong guess sequences and backtrack.

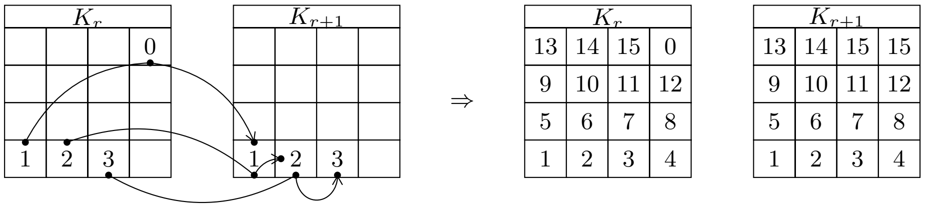


Fig. 2. Guess order of the 11-byte masking scheme

We have extensively simulated our attack by generating perfect measurement sets of Hamming weights. For different numbers T of exploited traces ($T \in \{5, 10, 15, 20, 30\}$) – this number influences the tightness of the bounds derived for the Hamming distances – we ran N simulations ($N = 1000$ in most cases) of the attack. For each run we picked a key at random, and for each T executions we computed a masked expanded key based on an execution specific set of masks $(m_0, \ldots, m_{10})$, from which we derived the set of Hamming weights assumed to be available to the attacker. Given a round r we computed the sets of possible Hamming distances between each couple $(k_{r,i}, k_{r,i'})$ and $(k_{r+1,i}, k_{r+1,i'})$. Then we choose one particular trace (actually a set of Hamming weights) among the T available ones and a starting position of the guess sequence[3], and executed the second phase of the attack (exploration process).

Table 1 shows the simulation results obtained on a classical PC equipped with an 2.4 MHz I5 core processor and 4 GB of RAM. For each number of exploited traces we give the average computation time as well as the average

[3] Note that an attacker can freely choose both the trace which is exploited for the key search, the round r from 0 to 9 and the starting position from 12 to 15. We took this opportunity to select those parameters that minimize the number of possible values of the starting triplet – i.e. $(k_{r,12}, k_{r,3}, k_{r+1,3})$ in the example above – that are compatible with the measured Hamming weights.

residual entropy of the key (the $\log_2$ of the number of compatible keys returned by the attack). Because of a large variance in the attack computation time, we choose to limit the exploration with a given timeout. The value of this timeout as well as the percentage of simulations that terminated within this limit are also presented. Note that the average figures in the second and third columns are computed over the set of terminating simulations.

Table 1. Simulation results of the attack on the 11-byte masking countermeasure

Number of traces (T)	Average time (s)	Average residual entropy (bits)	Simulation timeout (s)	Percentage of terminating runs	Number of runs (N)
5	398	5.9	1800	47.0	83
10	40.6	0.66	300	93.4	500
15	10.0	0.29	60	94.7	1000
20	5.9	0.24	60	98.2	1000
30	3.0	0.24	60	100.0	1000

The proposed attack is quite efficient, even for a number of exploited traces reduced to five. In this case about 45% of runs terminate in less than 30 minutes and the average entropy of the key set that remains to exhaust is only about five bits.

Remark 1. From a practical point of view related to the ability for the attacker to infer Hamming weight from the leakage traces, we notice that in this attack not all 176 Hamming weights are needed per trace but only 32 ones. Also, the opportunity that the attacker has to choose which round key he wants to attack may be exploited to select the portion of the traces where he is the more confident about the measured Hamming weights.

3.2 16-byte Entropy Boolean Masking

The second countermeasure that we consider consists in masking all bytes of a round key with a different random byte, while repeating these 16 masks for all round keys. Precisely, each masked round key is defined as $K'_r = (k'_{r,i})_i$ with $k'_{r,i} = k_{r,i} \oplus m_i$ $(i = 0, \ldots, 15)$. Figure 3 depicts the mask pattern that applies on the expanded key bytes.

As in the attack on the 11-byte masking scheme, we will first exploit several traces in order to obtain information on Hamming distances between key bytes sharing a same mask. We also want to limit to two the number of mask values that must be simultaneously guessed in the most explosive (less constrained) part of the key space exploration. It follows from this that the sequence of guesses should extend horizontally on a same byte position i rather than on a same round key r.

K_0			
m_{00}	m_{04}	m_{08}	m_{12}
m_{01}	m_{05}	m_{09}	m_{13}
m_{02}	m_{06}	m_{10}	m_{14}
m_{03}	m_{07}	m_{11}	m_{15}

K_1			
m_{00}	m_{04}	m_{08}	m_{12}
m_{01}	m_{05}	m_{09}	m_{13}
m_{02}	m_{06}	m_{10}	m_{14}
m_{03}	m_{07}	m_{11}	m_{15}

K_2			
m_{00}	m_{04}	m_{08}	m_{12}
m_{01}	m_{05}	m_{09}	m_{13}
m_{02}	m_{06}	m_{10}	m_{14}
m_{03}	m_{07}	m_{11}	m_{15}

$\dots$

Fig. 3. Part of the 16-byte masking scheme

Given a starting position $a \in \{0, 1, 2, 3\}$, we define the related position $b = 12 + ((a+1) \bmod 4)$. For each guess on the couple of masks (m_a, m_b), we perform an exploration of the key space as follows. First we guess $k_{0,a}$. Then repeatedly for $r = 0, \dots, 9$ we guess $k_{r,b}$ and derive $k_{r+1,a}$. As in the attack described in Sect. 3.1, each newly guessed or computed key byte is checked against available information about the Hamming weight of its masked value and the Hamming distances with other already known bytes at the same position. We have now performed the most demanding part of the exploration since we had to make a new guess for each byte $k_{r,b}$. At this point we have a reasonably small number of compatible key candidates for which we know all key bytes at positions a and b except $k_{10,b}$. We now guess $k_{10,b}$ which is quite constrained by the Hamming distances at position b and so does not increase much the exploration size. Knowing $k_{10,b}$, we can now successively compute key bytes at position $c = b - 4$ backward from $k_{10,c}$ to $k_{1,c}$. Note that m_c is the only value that we must guess to compute this line up to $k_{1,c}$. We terminate the line c by guessing the quite constrained last byte $k_{0,c}$. Now, guessing the mask m_d at position $d = c - 4$ we can compute in the same way all the line d from $k_{10,d}$ to $k_{1,d}$, and terminate the line by guessing $k_{0,d}$. We can pursue the same process with one more line at position $e = d - 4$ and then the next line is located at position $f = 12 + ((b+1) \bmod 4)$ and is computed forward from $k_{0,f}$ to $k_{9,f}$ terminating with a guess on $k_{10,f}$. Successively we determine all the expanded key, line after line, at positions whose sequence $a, b, c, \dots$ is presented on Fig. 4.

Interestingly, we can notice a property that stands for the first line a and which allows to dramatically speed up the attack. For each solution found on

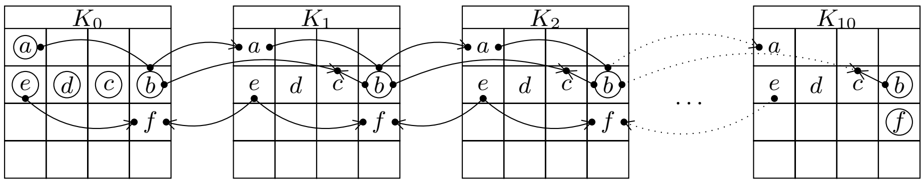


Fig. 4. Guess order of the 16-byte masking scheme

lines a and b by assuming the couple of masks (m_a, m_b) we would have found a companion solution with any other value m'_a where all $k_{r,b}$ are the same and where each $k_{r,a}$ is replaced by $k_{r,a} \oplus (m_a \oplus m'_a)$. As key bytes at position a do not influence those recovered on the successive lines b, c, d,... we do not have to know the exact value of m_a and can fix it arbitrary. At the end of the attack we are able to compute the correct value of the line a by inferring the error made on m_a based for example on the difference between the assumed value of $k_{10,a}$ and its exact value which can be computed as $k_{9,p} \oplus k_{10,p}$ where $p = a + 4$. Doing so, the first part of the exploration, which results in knowing values at positions a and b, can be done by guessing virtually only one mask byte (m_b). A speed-up factor of 2^8 is achieved which results in a particularly efficient attack.

Table 2 presents simulation results for this attack in a similar manner than in Sect. 3.1. Surprisingly, the key recovery in the presence of a 16-byte masking is much more efficient than with the 11-byte masking despite the higher mask entropy. For example the key is recovered within 1 second on average when 10 traces are exploited against 40 seconds for the 11-byte masking. Also, it is possible to use only 3 traces with still small computation time and residual key entropy in a significant proportion of cases.

Table 2. Simulation results of the attack on the 16-byte masking countermeasure

Number of traces (T)	Average time (s)	Average residual entropy (bits)	Simulation timeout (s)	Percentage of terminating runs	Number of runs (N)
3	77.3	7.3	600	60.7	28
5	25.3	4.2	300	88.5	1000
10	1.09	1.7	60	100.0	1000
15	0.24	0.93	60	100.0	1000
20	0.12	0.55	60	100.0	1000
30	0.07	0.24	60	100.0	1000

3.3 Column-Wise Random Order Countermeasure

The third countermeasure consists in calculating independent bytes in a random order. Due to the column based structure of the key schedule the four bytes of each column can be calculated independently. Figure 5 gives an example of a possible sequence of permutation.

This countermeasure is hiding a part of information. We still assume that the attacker is able to correctly identify all 176 Hamming weights but for every column he only obtains a non-ordered set of 4 values. For example, given the example key represented in Figure 6 where key bytes Hamming weights are indicated in the corner, the information that an attacker has access to is shown on Figure 7. The key bytes of each column have been involved in a random order so that the attacker can only infer non-ordered quadruplets of Hamming weights.

K_0			
$k_{0,2}$	$k_{0,4}$	$k_{0,9}$	$k_{0,14}$
$k_{0,3}$	$k_{0,6}$	$k_{0,11}$	$k_{0,12}$
$k_{0,1}$	$k_{0,5}$	$k_{0,10}$	$k_{0,13}$
$k_{0,0}$	$k_{0,7}$	$k_{0,8}$	$k_{0,15}$

K_1			
$k_{1,0}$	$k_{1,7}$	$k_{1,11}$	$k_{1,15}$
$k_{1,1}$	$k_{1,6}$	$k_{1,10}$	$k_{1,14}$
$k_{1,3}$	$k_{1,5}$	$k_{1,9}$	$k_{1,12}$
$k_{1,2}$	$k_{1,4}$	$k_{1,8}$	$k_{1,13}$

K_2			
$k_{2,3}$	$k_{2,7}$	$k_{2,8}$	$k_{2,13}$
$k_{2,2}$	$k_{2,5}$	$k_{2,9}$	$k_{2,15}$
$k_{2,0}$	$k_{2,6}$	$k_{2,11}$	$k_{2,14}$
$k_{2,1}$	$k_{2,4}$	$k_{2,10}$	$k_{2,12}$

...

Fig. 5. Part of an example of effect of random order countermeasure

K_0			
2B $\boxed{4}$	28 $\boxed{2}$	AB $\boxed{5}$	09 $\boxed{2}$
7E $\boxed{6}$	AE $\boxed{5}$	F7 $\boxed{7}$	CF $\boxed{6}$
15 $\boxed{3}$	D2 $\boxed{4}$	15 $\boxed{3}$	4F $\boxed{5}$
16 $\boxed{3}$	A6 $\boxed{4}$	88 $\boxed{2}$	3C $\boxed{4}$

K_1			
A0 $\boxed{2}$	88 $\boxed{2}$	23 $\boxed{3}$	2A $\boxed{3}$
FA $\boxed{6}$	54 $\boxed{3}$	A3 $\boxed{4}$	6C $\boxed{4}$
FE $\boxed{7}$	2C $\boxed{3}$	39 $\boxed{4}$	76 $\boxed{5}$
17 $\boxed{4}$	B1 $\boxed{4}$	39 $\boxed{4}$	05 $\boxed{2}$

K_2			
F2 $\boxed{5}$	7A $\boxed{5}$	59 $\boxed{4}$	73 $\boxed{5}$
C2 $\boxed{3}$	96 $\boxed{4}$	35 $\boxed{4}$	59 $\boxed{4}$
95 $\boxed{4}$	B9 $\boxed{5}$	80 $\boxed{1}$	F6 $\boxed{6}$
F2 $\boxed{5}$	43 $\boxed{3}$	7A $\boxed{5}$	7F $\boxed{7}$

...

hexadecimal key byte value

corresponding Hamming weight

Fig. 6. Three first round keys derived from an example key with their corresponding Hamming weights

Since all 24 permutations on the quadruplet can be considered as valid a priori[4], the computational effort for considering every permutation on each column makes the key search computationally unfeasible. In order to reduce the exploration cost we use what we call a booking system. During our attack we will book Hamming weights at fixed positions, either by choice when the byte value is guessed, or by constraint when it is computed from a key byte relation. Once it is booked a Hamming weight is no more available in its column until a backtrack releases it due to a modification of the last guessed byte.

For instance, when we have to guess a value for $k_{1,15}$ we first guess its Hamming weight among the list {2,3,4,5} of available Hamming weights. If we guess that $\mathrm{HW}(k_{1,15}) = 4$, then the guess on $k_{1,15}$ itself ranges over all values having an Hamming weight equal to 4, and the list of available Hamming weights for that column is now reduced to {2,3,5}. When another byte of the same column will be also guessed (or computed) at a deeper step of the exploration process its Hamming weight will necessarily have to belong to this reduced set. If at some point a backtrack occurs on $k_{1,15}$ then the Hamming weight value 4 is released and will be possibly available for other bytes of this column.

We describe here two versions of this attack, one using information given by one acquisition, which can take non-negligible time, and faster version which exploits faulty executions in order to gather more information.

[4] Due to possible Hamming weight duplicates, some columns may have a reduced number of possible permutations.

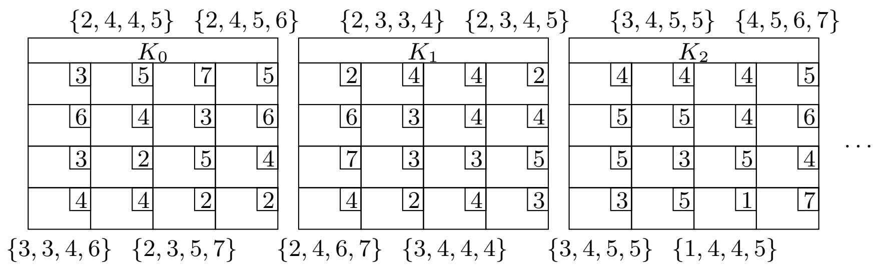


Fig. 7. Information gained by the attacker reduces to quadruplets of Hamming weights of each column

Basic Attack. In a basic version of our attack we follow an equivalent exploration pattern than the one used in [13] for a non-protected implementation. The only difference is that the guess may have different possible Hamming weights. As explained above, before guessing a byte value at a current position we have to guess which un-booked Hamming weight value will be used at this position and book it while it's corresponding values are exhausted. When we have guessed bytes at enough positions to compute key bytes from others we check that the Hamming weight of the computed values are available for their columns and we book these Hamming weights also. If the Hamming weight of a computed byte is not available then this solution is not valid and we backtrack from the previous guessed byte. Note that if a same Hamming weight value is available n times in a column it can be booked n times too.

We simulated this attack by considering random keys and corresponding non-ordered quadruplets for each column. Table 3 presents the number of executions over 100 runs that ended before a time limit which ranges from 30 minutes to 6 hours. As it can take undefined long time we choose to interrupt a run if it takes more than 6 hours (27 % of cases). Note that the average time for the non-interrupted executions is about 2 hours, so that average time over all executions could possibly be quite larger.

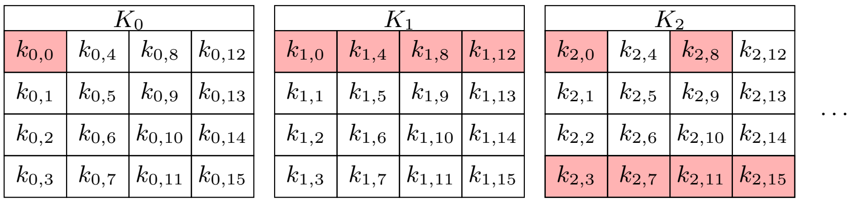
Faulting Attack. We describe here a more efficient version of the attack which uses fault injections in order to significantly reduce the execution time of the key search.

We assume that the attacker can induce a fault in a random byte of a chosen column, and we take the example of the first column in the following explanations. The fault model assumes a random modification of the faulted byte value.

Table 3. Results of non-faulted attack against random order counter-measure

Time Elapsed	$\leqslant$ 30 min	$\leqslant$ 1h	$\leqslant$ 2h	$\leqslant$ 3h	$\leqslant$ 4h	$\leqslant$ 5h	$\leqslant$ 6h	+ 6h
# over 100 runs	6	25	41	55	66	71	73	27

The key observation used in this attack is that a differential induced at some key byte of the first column propagates following a fixed pattern of active bytes. For example, if the fault modifies the value of $k_{0,0}$ then Figure 8 shows the positions of all active bytes in the first three round keys[5]. Due to the shuffling counter-measure, the attacker does not know which of $k_{0,0}$, $k_{0,1}$, $k_{0,2}$ or $k_{0,3}$ has been modified by the fault, but what is important is that the vertical relative positions of the active bytes are fixed (given by the pattern of Figure 8) and known from the attacker.



K_0			
🟥 $k_{0,0}$	$k_{0,4}$	$k_{0,8}$	$k_{0,12}$
$k_{0,1}$	$k_{0,5}$	$k_{0,9}$	$k_{0,13}$
$k_{0,2}$	$k_{0,6}$	$k_{0,10}$	$k_{0,14}$
$k_{0,3}$	$k_{0,7}$	$k_{0,11}$	$k_{0,15}$

K_1			
🟥 $k_{1,0}$	🟥 $k_{1,4}$	🟥 $k_{1,8}$	🟥 $k_{1,12}$
$k_{1,1}$	$k_{1,5}$	$k_{1,9}$	$k_{1,13}$
$k_{1,2}$	$k_{1,6}$	$k_{1,10}$	$k_{1,14}$
$k_{1,3}$	$k_{1,7}$	$k_{1,11}$	$k_{1,15}$

K_2			
🟥 $k_{2,0}$	$k_{2,4}$	🟥 $k_{2,8}$	$k_{2,12}$
$k_{2,1}$	$k_{2,5}$	$k_{2,9}$	$k_{2,13}$
$k_{2,2}$	$k_{2,6}$	$k_{2,10}$	$k_{2,14}$
🟥 $k_{2,3}$	🟥 $k_{2,7}$	🟥 $k_{2,11}$	🟥 $k_{2,15}$

...

Fig. 8. Part of the pattern induced by a fault on first byte of first column of K_0

As in the basic attack described above, the attacker can exploit a non-faulted execution to infer the reference quadruplets of Hamming weights for each column.

When exploiting a faulted execution, the attacker can compare, for each column, the possibly modified quadruplet of Hamming weights with the original one. He is thus able to identify which Hamming weights have been modified and thus concern active bytes. Let's consider an example where the faulted byte is $k_{0,2}$ which value has been modified from `0x15` to `0xB1`. This example case is depicted on Figure 9 where one can see all subsequent active bytes. Note that in this example, some active bytes ($k_{2,5}$, $k_{2,9}$ and $k_{2,13}$) have been modified while their Hamming weights remained unchanged.

Due to the shuffling counter-measure, the attacker faces round keys where each column has been shuffled as shown on Figure 10. Remind that the attacker does not know neither the byte values nor the active bytes positions (colored in red on the figure), but only the quadruplets of Hamming weights. Comparing for example the original ({2,4,6,7} on Figure 7) and faulted ({2,4,4,6} on Figure 10) quadruplets of column 4, he can infer that 7 is the Hamming weight of the only active cell in this column. Similarly, he can also infer that the Hamming weight of the only active cell in the column 7 is 5. Considering column 10, the attacker infers the partial information that one of the two active bytes Hamming weights is equal to 1.

Even if the information retrieved about the Hamming weights of the active bytes of each column is only partial, we can nevertheless exploit them in the key search algorithm. For example, in the guess-compute-and-backtrack process, when one guesses that the value of e.g. $k_{1,3}$ has an Hamming weight of 7 (so that

[5] Obviously, the pattern is not limited to the three round keys, it extends on all 11 round keys.

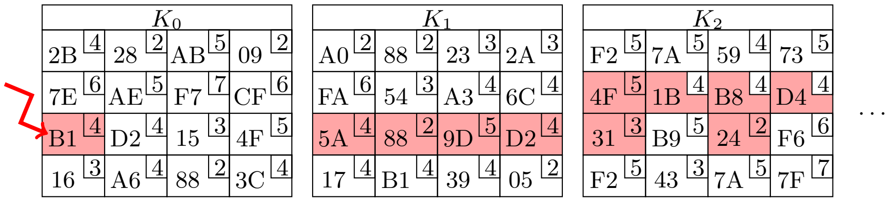


Fig. 9. Details of fault effect without considering countermeasure Red/darkgrey Active, green/lightgrey Active but remains unchanged

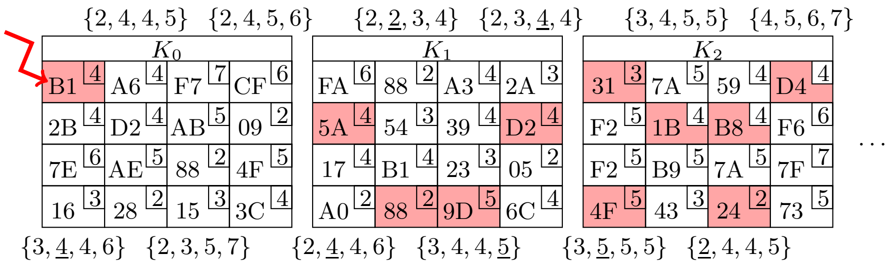


Fig. 10. Attacker point of view of faulted execution, underlined values in sets are thoses detected by the attacker as modified by the fault

$k_{1,3}$ would be an active byte), then in column 7 the active byte is necessarily located in the bottom cell also (cf. the active bytes pattern of Figure 8), so that we know that $\mathrm{HW}(k_{1,15}) = 5$.

As one can see, the principle of the faulting attack is to exploit in the key search phase information about Hamming weights of active bytes (whose relative vertical positions is fixed) which have been acquired by comparing Hamming weight quadruplets of faulted executions from original ones. While the detailed explanations are quite intricate, it is though possible to infer more information from successive faulty executions to further reduce the execution time of the key search.

We have simulated the faulting attack by exploiting as much information given by faults as possible. We give in Table 4 average execution times of the key search phase as a function of the number of exploited faulty executions. Note that even with only one faulty execution the average attack time is dramatically reduced from several hours to only 20 minutes.

Table 4. Results of faulted attack against random order counter-measure

fault number	time (min)
1	20
5	5
10	3
20	2
30	2

Remark 2. It is interesting to notice that if the fault did not occur in the first column then the attack is still possible while possibly less efficient. Indeed the pattern of active bytes induced by a fault in any column is always a subset of the pattern induced by a fault in the first column. Consequently this shorter pattern has the same shape as the pattern starting from the first column and can then be exploited in the same way but will provide information only for rightmost columns. This allows to perform this attack even when the attacker do not have a precise control on the timing of the fault.

4 Recommendations for Secure Implementations

Considering the problem of recovering a key by analysing the Hamming weights of the key bytes computed during the key expansion process, several countermeasures are proposed in the seminal contribution [8] among which the Boolean masking of the key expansion. We showed that two versions of this countermeasure with 11 and 16 bytes of mask entropy are not sufficient to prevent the key recovery when the attacker can precisely infer the Hamming weights. Our attacks on the Boolean masking also apply if the expanded key is computed once for all and there is no key expansion process computed by the device. In that case the Hamming weights can still be measured, not while the key bytes are computed but rather when they are transferred into RAM and/or used in the `AddRoundKey` function.

Using an hardware or an 16- or 32-bit AES implementation prevents our attacks which only apply on 8-bit software implementations. On these later devices we recommend either to implement (if ever possible) a full 176-byte key masking where all key bytes are masked by independent random values, or to combine a weaker masking with other countermeasures that reinforce its security. For example, combining one of the two masking methods considered in this paper together with the column-wise shuffling should be sufficient to prevent the attacker from obtaining enough exploitable information from the computation of the round keys itself. As for the manipulation of the key bytes in the encryption process, the combination of masking and shuffling should also be sufficient with the advantage here that the entropy of the shuffling is higher in this later case since all 16 bytes may be shuffled together instead of per chunks of four bytes. Obviously, on top of these fundamental countermeasures, any means to make it difficult to find the relevant points of interest on the side-channel trace – e.g. random delays – or to interpret the leakage in terms of Hamming weight – added signal noise – would add extra security to the AES implementation.

5 Conclusion

In this paper we have revisited a simple power analysis on the AES key expansion. While previous works only apply on unprotected implementations, we have considered three different countermeasures and presented efficient attacks in each scenario. In two Boolean masking cases (11-byte and 16-byte mask entropy) our attacks recover the key in a matter of seconds when a few power traces are exploited. In the case of a column-wise shuffling of the key expansion process, we have devised an attack which takes several hours on average and proposed an improved version that takes advantage of extra information provided by fault analysis so that the computation time is reduced to a few minutes.

Our attacks assume that the attacker is able to obtain correct values of the Hamming weights of the key bytes. As a future work it may be interesting to study how more difficult it would be to cope with erroneous observations.

Acknowledgments. Simulations presented in this paper have been partly performed on the CALI computing cluster of university of Limoges, funded by the Limousin region, XLIM, IPAM and GEIST institutes, as well as the university of Limoges.

References

1. Biham, E., Shamir, A.: Power Analysis of the Key Scheduling of the AES Candidates. In: Second AES Candidate Conference – AES2, Rome, Italy (1999)
2. Brier, E., Clavier, C., Olivier, F.: Correlation Power Analysis with a Leakage Model. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer, Heidelberg (2004)
3. Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 13–28. Springer, Heidelberg (2003)
4. Gierlichs, B., Batina, L., Tuyls, P., Preneel, B.: Mutual Information Analysis. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 426–442. Springer, Heidelberg (2008)
5. Koç, Ç.K., Paar, C.: CHES 2000. LNCS, vol. 1965. Springer, Heidelberg (2000)
6. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
7. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
8. Mangard, S.: A Simple Power-Analysis (SPA) Attackon Implementations of the AES Key Expansion. In: Lee, P.J., Lim, C.H. (eds.) ICISC 2002. LNCS, vol. 2587, pp. 343–358. Springer, Heidelberg (2003)
9. Mayer-Sommer, R.: Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In: Koç, Paar (eds.) [5], pp. 78–92
10. Messerges, T.S.: Using Second-Order Power Analysis to Attack DPA Resistant Software. In: Koç, Paar (eds.) [5], pp. 238–251
11. Messerges, T.S., Dabbish, E.A., Sloan, R.H.: Investigations of Power Analysis Attacks on Smartcards. In: WOST 1999: Proceedings of the USENIX Workshop on Smartcard Technology, pp. 151–162. USENIX Association, Berkeley (1999)
12. National Institute of Standards and Technology. Advanced Encryption Standard (AES). Federal Information Processing Standard #197 (2001)
13. VanLaven, J., Brehob, M., Compton, K.J.: Side Channel Analysis, Fault Injection and Applications - A Computationally Feasible SPA Attack on AES via Optimized Search. In: Sasaki, R., Qing, S., Okamoto, E., Yoshiura, H. (eds.) SEC 2005. IFIP AICT, vol. 181, pp. 577–588. Springer, Heidelberg (2005)

A Generating Undistinguishable Keys Pairs

The core idea of our method comes from the observation that given a permutation τ of $\{0,\ldots,7\}$ and the byte transformation $\pi : b = (b_7 \ldots b_0) \mapsto \pi(b) = (b_{\tau(7)} \ldots b_{\tau(0)})$ we have $\mathrm{HW}(\pi(b)) = \mathrm{HW}(b)$. Thus, a sufficient condition for K and K' to form a twin pair is that $k'_j = \pi(k_j)$ for all $j = 0,\ldots,175$. Our goal is to find K such that defining K' by $k'_{0,i} = \pi(k_{0,i}), i = 0,\ldots,15$ the sufficient condition propagates up to (near) the end of the expansion. As π is linear the sufficient condition propagates well on all linear relations. The only difficult task is to ensure the propagation of the condition also for non-linear relations. Denoting $c_r = \{02\}^{r-1}$ the constant involved in the first non-linear relation at round $r = 1,\ldots,10$, and assuming that the sufficient conditions hold up to round key K_{r-1}, they propagate to K_r provided that:

$$k'_{r,0} = \pi(k_{r,0}) \Leftrightarrow \mathrm{S}(\pi(k_{r-1,13})) \oplus c_r = \pi(\mathrm{S}(k_{r-1,13})) \oplus \pi(c_r) \quad (5)$$
$$k'_{r,1} = \pi(k_{r,1}) \Leftrightarrow \mathrm{S}(\pi(k_{r-1,14})) = \pi(\mathrm{S}(k_{r-1,14})) \quad (6)$$
$$k'_{r,2} = \pi(k_{r,2}) \Leftrightarrow \mathrm{S}(\pi(k_{r-1,15})) = \pi(\mathrm{S}(k_{r-1,15})) \quad (7)$$
$$k'_{r,3} = \pi(k_{r,3}) \Leftrightarrow \mathrm{S}(\pi(k_{r-1,12})) = \pi(\mathrm{S}(k_{r-1,12})) \quad (8)$$

The first task is to find a suitable bit permutation which maximizes the probability that these conditions hold by chance. Interestingly the probability that any condition (6) to (8) holds is as large as about $\frac{1}{4}$ when τ permutes only 2 bits[6]. This is due to the fact that $\mathrm{S}(\pi(x)) = \pi(\mathrm{S}(x))$ as soon as $\pi(x) = x$ and $\pi(y) = y$ for $y = \mathrm{S}(x)$ where both fixed-point conditions hold with probability $\frac{1}{2}$. Finding a twin pair only necessitates that all $k_{r-1,i}$ ($r = 1,\ldots 10$ and $i = 12,\ldots,15$) belong the following sets:

$$\Omega_r = \{x : \mathrm{S}(\pi(x)) \oplus c_r = \pi(\mathrm{S}(x)) \oplus \pi(c_r)\} \quad (\text{for } i = 13)$$
$$\Omega = \{x : \mathrm{S}(\pi(x)) = \pi(\mathrm{S}(x))\} \quad (\text{for } i \in \{12, 14, 15\})$$

It is important that either Ω or Ω_1 contains some value x which satisfies the condition without being a fixed point for π otherwise K' would be equal to K. We have chosen τ which permutes bits 0 and 6. Note that it is the only bit transposition having a non fixed point for Ω.

The second task is to generate many key candidates which verify by construction as many sufficient conditions as possible. We devised a method that efficiently generates a large number of candidates that systematically fulfill sufficient conditions for all $r \leqslant 5$. First we make vary the twelve key bytes $k_{1,12+n}$, $k_{2,12+n}$ and $k_{3,12+n}$ ($n = 0,\ldots,3$) which are free except that they must all belong

[6] This is also true for condition (5) for a similar reason.

to their respective relevant Ω, Ω_2, Ω_3 or Ω_4 set. Due to the previous remark the number of possible choices for these bytes is lower bounded by $(256/4)^{12} = 2^{72}$. We also make use of the following relations among the key bytes

$$k_{4,12+n} = k_{0,12+n} \oplus \mathrm{S}(k_{3,12+(n+1) \bmod 4}) \oplus c'_4 \quad (9)$$

$$k_{0,8+n} = k_{0,12+n} \oplus \mathrm{S}(k_{2,12+(n+1) \bmod 4}) \oplus c'_3 \oplus k_{3,12+n} \quad (10)$$

$$k_{0,4+n} = k_{0,8+n} \oplus \mathrm{S}(k_{1,12+(n+1) \bmod 4}) \oplus c'_2 \oplus k_{2,12+n} \oplus \mathrm{S}(k_{2,12+(n+1) \bmod 4}) \oplus c'_3 \oplus k_{3,12+n} \quad (11)$$

$$k_{0,0+n} = k_{0,8+n} \oplus \mathrm{S}(k_{0,12+(n+1) \bmod 4}) \oplus c'_1 \oplus k_{1,12+n} \oplus \mathrm{S}(k_{1,12+(n+1) \bmod 4}) \oplus c'_2 \oplus k_{2,12+n} \quad (12)$$

where c'_r is defined to be c_r if $n = 0$ and 0 otherwise. The proofs of these relations are provided in Appendix B. Considering equation (9), and knowing that $k_{3,12...15}$ have been chosen in their respective Ω set, one can choose values for $k_{0,12+n}$ that belong to its Ω set such that $k_{4,12+n}$ also belongs to its own Ω set. For example, given $k_{3,14} \in \Omega$ one can find two values $k_{0,13}$ and $k_{4,13}$ which respectively belong to Ω_1 and Ω_5. There always exists several such choices that we have tabulated though only one choice was sufficient in our implementation. Choosing $k_{0,12+n}$ this way ensures that the sufficient conditions will be verified even for the non-linear relations involved in the computation of K_5.

The process to generate the key candidates resumes as follow: choose arbitrary value for $k_{1,12+n}$, $k_{2,12+n}$ and $k_{3,12+n}$ $(n = 0, \ldots, 3)$ that belong to their respective relevant Ω set, then choose values for $k_{0,12+n}$ as explain above, and terminate the valuation of $K = K_0$ by using equations (10) to (12) successively. For each such key K we compute K' by applying the bit transposition π to all its bytes. Our construction method ensures that $k'_{r,i} = \pi(k_{r,i})$ – and so $\mathrm{HW}(k'_{r,i}) = \mathrm{HW}(k_{r,i})$ – for all $r = 0, \ldots, 5$.

Generating sufficiently many key candidates, one can expect to find one for which the sufficient conditions propagate by chance over the non-linear relations up to the end of the expansion.

After having found a first winning key pair – the one given in Sect. 2 – we explored in its neighborhood and we surprisingly generated many other undistinguishable pairs much more easily that it was to find the first one. For example, keeping the values of $k_{1,13}$, $k_{1,14}$, $k_{2,12}$ and $k_{2,13}$ involved in the first key pair, we have been able to generate more than 23 millions of other undistinguishable key pairs in a few days of computation. This tend to demonstrate that pairs of keys having same Hamming weight signatures are far from being uniformly distributed, but we have not studied this behavior in more detail.

B Proofs of Equations (9) to (12)

B.1 Equation (9)

Proof.

$$\begin{aligned}
k_{4,12+n} &= k_{4,8+n} \oplus k_{3,12+n} \\
&= k_{4,4+n} \oplus k_{3,12+n} \oplus k_{3,8+n} \\
&= k_{4,0+n} \oplus k_{3,12+n} \oplus k_{3,8+n} \oplus k_{3,4+n} \\
&= \mathrm{S}(k_{3,12+(n+1) \bmod 4}) \oplus c'_4 \oplus k_{3,12+n} \oplus k_{3,8+n} \oplus k_{3,4+n} \oplus k_{3,0+n} \\
&= \mathrm{S}(k_{3,12+(n+1) \bmod 4}) \oplus c'_4 \oplus k_{3,4+n} \oplus k_{3,0+n} \oplus k_{2,12+n} \\
&= \mathrm{S}(k_{3,12+(n+1) \bmod 4}) \oplus c'_4 \oplus k_{2,12+n} \oplus k_{2,4+n} \\
&= \mathrm{S}(k_{3,12+(n+1) \bmod 4}) \oplus c'_4 \oplus k_{2,8+n} \oplus k_{2,4+n} \oplus k_{1,12+n} \\
&= \mathrm{S}(k_{3,12+(n+1) \bmod 4}) \oplus c'_4 \oplus k_{1,12+n} \oplus k_{1,8+n} \\
&= \mathrm{S}(k_{3,12+(n+1) \bmod 4}) \oplus c'_4 \oplus k_{0,12+n}
\end{aligned}$$

□

B.2 Equation (10)

Proof.

$$\begin{aligned}
k_{3,12+n} &= k_{3,8+n} \oplus k_{2,12+n} \\
&= k_{3,4+n} \oplus k_{2,12+n} \oplus k_{2,8+n} \\
&= k_{3,0+n} \oplus k_{2,12+n} \oplus k_{2,8+n} \oplus k_{2,4+n} \\
&= \mathrm{S}(k_{2,12+(n+1) \bmod 4}) \oplus c'_3 \oplus k_{2,12+n} \oplus k_{2,8+n} \oplus k_{2,4+n} \oplus k_{2,0+n} \\
&= \mathrm{S}(k_{2,12+(n+1) \bmod 4}) \oplus c'_3 \oplus k_{2,4+n} \oplus k_{2,0+n} \oplus k_{1,12+n} \\
&= \mathrm{S}(k_{2,12+(n+1) \bmod 4}) \oplus c'_3 \oplus k_{1,12+n} \oplus k_{1,4+n} \\
&= \mathrm{S}(k_{2,12+(n+1) \bmod 4}) \oplus c'_3 \oplus k_{1,8+n} \oplus k_{1,4+n} \oplus k_{0,12+n} \\
&= \mathrm{S}(k_{2,12+(n+1) \bmod 4}) \oplus c'_3 \oplus k_{0,12+n} \oplus k_{0,8+n}
\end{aligned}$$

□

B.3 Equation (11)

Proof.

$$\begin{aligned}
k_{3,12+n} &= k_{3,8+n} \oplus k_{2,12+n} \\
&= k_{3,4+n} \oplus k_{2,12+n} \oplus k_{2,8+n} \\
&= k_{3,0+n} \oplus k_{2,12+n} \oplus k_{2,8+n} \oplus k_{2,4+n} \\
&= \mathrm{S}(k_{2,12+(n+1) \bmod 4}) \oplus c'_3 \oplus k_{2,12+n} \oplus k_{2,8+n} \oplus k_{2,4+n} \oplus k_{2,0+n} \\
&= \mathrm{S}(k_{2,12+(n+1) \bmod 4}) \oplus c'_3 \oplus k_{2,12+n} \oplus k_{2,0+n} \oplus k_{1,8+n} \\
&= \mathrm{S}(k_{2,12+(n+1) \bmod 4}) \oplus c'_3 \oplus k_{2,12+n} \oplus \mathrm{S}(k_{1,12+(n+1) \bmod 4}) \oplus c'_2 \oplus k_{1,8+n} \oplus k_{1,0+n} \\
&= \mathrm{S}(k_{2,12+(n+1) \bmod 4}) \oplus c'_3 \oplus k_{2,12+n} \oplus \mathrm{S}(k_{1,12+(n+1) \bmod 4}) \oplus c'_2 \oplus k_{1,4+n} \oplus k_{1,0+n} \oplus k_{0,8+n} \\
&= \mathrm{S}(k_{2,12+(n+1) \bmod 4}) \oplus c'_3 \oplus k_{2,12+n} \oplus \mathrm{S}(k_{1,12+(n+1) \bmod 4}) \oplus c'_2 \oplus k_{0,8+n} \oplus k_{0,4+n}
\end{aligned}$$

□

B.4 Equation (12)

Proof.

$$\begin{aligned}
k_{2,12+n} &= k_{2,8+n} \oplus k_{1,12+n} \\
&= k_{2,4+n} \oplus k_{1,12+n} \oplus k_{1,8+n} \\
&= k_{2,0+n} \oplus k_{1,12+n} \oplus k_{1,8+n} \oplus k_{1,4+n} \\
&= \mathrm{S}(k_{1,12+(n+1) \bmod 4}) \oplus c'_2 \oplus k_{1,12+n} \oplus k_{1,8+n} \oplus k_{1,4+n} \oplus k_{1,0+n} \\
&= \mathrm{S}(k_{1,12+(n+1) \bmod 4}) \oplus c'_2 \oplus k_{1,12+n} \oplus k_{1,0+n} \oplus k_{0,8+n} \\
&= \mathrm{S}(k_{1,12+(n+1) \bmod 4}) \oplus c'_2 \oplus k_{1,12+n} \oplus \mathrm{S}(k_{0,12+(n+1) \bmod 4}) \oplus c'_1 \oplus k_{0,8+n} \oplus k_{0,0+n}
\end{aligned}$$

□