Implementation and improvement of the Partial Sum Attack on 6-round AES

Francesco Aldà (francesco.alda@rub.de)
Fakultät für Mathematik, Ruhr-Universität Bochum, Germany

Riccardo Aragona (riccardo.aragona@unitn.it)
Department of Mathematics, University of Trento, Italy

Lorenzo Nicolodi (lo@hidden-bits.com)
Freelancer, University of Trento, Italy

Massimiliano Sala (maxsalacodes@gmail.com)
Department of Mathematics, University of Trento, Italy

Abstract

The Partial Sum Attack is one of the most powerful attacks developed in the last 15 years against reduced-round versions of AES. We introduce a slight improvement to the basic attack which lowers the number of chosen plaintexts needed to successfully mount it. Our version of the attack on 6-round AES can be carried out completely in practice, as we demonstrate providing a full implementation. We also detail the structure of our implementation, showing the performances we achieve.

Keywords: Symmetric Cryptography, Cryptanalysis, Advanced Encryption Standard, Partial Sum Attack

1 Introduction

Some cryptanalysis research on block ciphers deals with studying and proposing attacks on their reduced-round versions. Results on reduced versions are interesting, since they can help to better understand the behavior of a cipher, pointing out weaknesses in its structure, which can eventually lead to attacks on the full version or at least characterize the security margin of the cipher.

In 2001, Ferguson et al. [FKL+01] introduced one of the most important attacks developed in the last 15 years against reduced-round versions of the Advanced Encryption Standard [DR02, DR98], the *Partial Sum Attack*. Specifically, they developed attacks against AES reduced to 6, 7 and 8 rounds. The

attack on 6-round is powerful and apparently it can be carried out in practice. It improves a previous attack already described in [DR98]. The latter is based on the *integral cryptanalysis*, a general technique which is applicable to a large class of SPN block ciphers. This technique was originally designed by Lars Knudsen in the paper presenting the block cipher Square [DKR97], as a specific attack against its byte-oriented structure. This is the reason why this class of attacks is commonly known as the *Square Attack*. Since AES inherits many properties from Square, this attack can be easily extended to reduced-round versions of the Advanced Encryption Standard.

In this paper we slightly improve the *Partial Sum Attack* on 6-round reduced AES and we describe the structure of our implementation. After examining the literature which was developed and produced after the publication of [FKL+01], we are not aware of any effective implementation of this attack. Therefore we believe that our implementation is the first and, mostly, we show that it is completely practicable. Moreover, we believe that our effort can allow a deeper knowledge of the attack workflow and can point out some other weaknesses or potentialities not already discovered or exploited.

The rest of the paper is organized as follows: in Section 2 we give an overview on the *Square Attack* and its extensions and we subsequently describe the *Partial Sum Attack* in details. In Section 3 we present our results. First we explain our (slight) theoretical improvement. We then detail our implementation and provide the results of our computations. In particular, we have been able to recover a (full) 6-round key in less than 12 days with 25 cores.

2 Preliminaries

We recall that the AES-128, -192, -256 encryption process consists of an initial key addition, followed by the application of 10, 12 and 14 round transformations, respectively. The initial key addition and every round transformation take as input an intermediate result, called the *state*, and a round key, which is derived from the cipher key through the key schedule. The output of any round is another state. The round transformation is a sequence of four processing steps: *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*. The final round differs from the others since the *MixColumns* step is removed. For further details on the structure of AES we refer to [DR02, DR98].
In the following sections, we first give an overview on the Square Attack on 4-round AES and briefly introduce its extensions. We then describe the Partial Sum Attack.

2.1 Square Attack

The Square Attack is a chosen plaintext attack, which is independent of the specific choices of the S-box of the *SubBytes* function, the multiplication polynomial of the *MixColumns* transformation and the key schedule.
In order to explain how this attack can be performed, we first introduce the following definition.

Definition 2.1. *A Δ-set is a set of 256 AES states that differ in one of the state bytes (called* active *byte) and are equal in the other state bytes (called* passive *bytes). In other words, for every $x, y \in \Delta$ we have*

$$\begin{cases} x_{i,j} \neq y_{i,j} & \text{if } (i,j) \text{ is active} \\ x_{i,j} = y_{i,j} & \text{if } (i,j) \text{ is passive} \end{cases}$$

where $i, j \in \{0,1,2,3\}$.

As it is explained in [DR02], the Square Attack on 4-round AES is heavily based on the following property.

Proposition 2.2. *Let $b_{i,j}^{(l)}$ be the byte in position (i,j), $i, j \in \{0,1,2,3\}$, of the lth state of a Δ-set after the application of three AES rounds. Then*

$$\sum_{l=1}^{256} b_{i,j}^{(l)} = 0. \tag{1}$$

In other words, the state at the end of the third round is *balanced*, i.e. all bytes at the input of the fourth round sum to zero.

Note that the initial key addition is implicitly assumed and not counted in the number of rounds.

Let us consider a 4-round reduced AES, in which the fourth round is a final round, i.e. it does not include *MixColumns*. Every byte of the ciphertext only depends on one input byte of the fourth round. The Square Attack on 4-round AES can then be mounted as follows. For any l^{th} state of a Δ-set, $1 \leq l \leq 256$, let $c_{i,j}^{(l)}$, where $0 \leq i, j \leq 3$, be the ciphertext byte in position (i,j). Let $k_{i,j}^{(4)}$ be a guess for the byte in position (i,j) of the 4th round key (which is the last key used). For any (i,j), if the value of $k_{i,j}^{(4)}$ is correct, the following equation holds:

$$\sum_{l=1}^{256} SubBytes^{-1}(c_{i,j}^{(l)} + k_{i,j}^{(4)}) = \sum_{l=1}^{256} b_{i,(j+i) \bmod 4}^{(l)} = 0, \tag{2}$$

where $b_{i,j}^{(l)}$ is the byte in position (i,j) of the lth state of a Δ-set after the application of three AES rounds.

If Equation (2) does not hold, the assumed value for the key byte must be wrong. This check eliminates all wrong key bytes, except for one value that could satisfy (2) by chance. To be more precise, the following result holds.

Proposition 2.3. *If $(X^{(l)})_{1 \leq l \leq 256}$, is a sequence of independent uniformly distributed random variables with values in $\mathbb{F}_{2^8}$, then the probability*

$$\mathbb{P}\left(\sum_{l=1}^{256} X^{(l)} = 0\right) = 2^{-8}.$$

Proof. Let X and Y be two discrete independent random variables, with density functions $f_1(x)$ and $f_2(x)$ respectively. The convolution $f_3(x) = [f_1 * f_2](x) = \sum_y f_1(y) f_2(x-y)$ is the density function of the random variable $Z = X + Y$. Since X and Y take values in $\mathbb{F}_{2^8}$, their sum Z takes values in $\mathbb{F}_{2^8}$ too. Therefore the density function of Z is an uniformly distributed random variable since it is the circular convolution of two independent uniformly distributed random variables. This result can be easily extended to the sum of an arbitrary number of random variables. □

Since checking Equation (2) for a single Δ-set leaves only 1 over 256 of the wrong key assumptions as a possible candidate, the 4th round key can be found with a sufficently large confidence using two different Δ-sets. Henceforth, this crosscheck will be called *verification step*.
In order for the attack to be successful, one needs to use 2 sets of 256 chosen plaintexts which form a Δ-set. Moreover, all 16 bytes of the key need to be recovered. Therefore, the working factor consists of 2^9 encryptions and $2^9 \cdot 2^4 = 2^{13}$ evaluations of Equation (2).

In [DR02], Daemen et al. describe how this attack can be extended adding one round at the end or one round at the beginning. Combining the basic attack on 4 rounds with both extensions yields to the Square Attack on 6-round AES. We can sketch this attack as follows. For the extension by one round at the end, the attacker has to perform a partial decryption of two rounds instead of only one and therefore she has to guess four more bytes of the final round key. The idea for the extension by one round at the beginning consists of choosing a set of plaintexts which, at the end of the first round, results in a Δ-set with a single active byte. This requires the guessing of four bytes of the initial round key $k^{(0)}$. We refer to [DR02] for further details on these two extensions. In both cases, we need to guess five key bytes instead of one. By combining the two methods, we would need to guess nine bytes.

2.2 Partial Sum Attack

Without considering the verification steps, the Square Attack on 6-round AES requires 2^{32} chosen plaintexts, 2^{32} memory for storing the corresponding ciphertexts and $(2^8)^9 = 2^{72}$ steps for guessing the nine bytes, when it is applied

to recover 4 bytes of the 6th round key. Therefore it is completely out of reach for current computing resources.

The Partial Sum Attack [FKL$^+$01] significantly improves the Square Attack on 6-round AES. Ferguson et al. introduced two main ideas. First, instead of guessing four bytes of the initial round key $k^{(0)}$, one can use all 2^{32} plaintexts. For any value of the initial round key, the corresponding ciphertexts consist of 2^{24} groups of 2^8 encryptions that vary in a single active byte at the end of the first round. In order to retrieve a Δ-set with a single active byte at the end of the first round, one should indeed impose a particular linear combination which ranges over all possible values of $\mathbb{F}_{2^8}$ and three other linear combinations which are constant for all 256 states. Hence one has 2^{24} ways to choose the values for these three linear combinations. Therefore, all an attacker has to do is guess four bytes of the 6th round key and one byte of the 5th round key, perform a partial decryption to a single state byte at the end of the 4th round, sum this value over all 2^{32} encryptions and check if the result is zero. Compared to the Square Attack on 6 rounds, the attacker needs to guess 40 bits instead of 72.
The further idea behind the improvement introduced by Ferguson et al. consists in organizing the partial decryption on *partial sums*. Throughout the rest of the paper, we denote by $\bar{\Delta}$-set a group of 2^{32} plaintexts such that one state column of bytes at the input of *MixColumns* of the first round ranges over all possible values of $(\mathbb{F}_{2^8})^4$ and all other bytes are constant.
In order to properly understand what *partial sums* are and how one can use them, we introduce the following notation, where the pair (i, j) is used to denote the state entry (with $0 \leq i, j \leq 3$) and the index l (with $1 \leq l \leq 2^{32}$) denotes that we are considering the lth element of the $\bar{\Delta}$-set:

$b_{i,j}^{(l)}$ is the byte at the end of the 4th round;

$a_{i,j}^{(l)}$ is the byte before the *MixColumns* application of the 5th round;

$c_{i,j}^{(l)}$ is the byte at the end of the 6th round, that we call the ciphertext byte;

$a_s^{(l)}$ is the sth column of the lth state before the *MixColumns* application of the 5th round, so $a_j^{(l)}$ contains $a_{0,j}^{(l)}$, $a_{1,j}^{(l)}$, $a_{2,j}^{(l)}$, $a_{3,j}^{(l)}$;

$k^{(h)}$ is the hth round key and $\bar{k}^{(h)} = \mathit{MixColumns}^{-1}(k^{(h)})$;

$\bar{k}_{i,j}^{(h)}$ is the byte of $\bar{k}^{(h)}$.

It is possible to show that, in order to compute the partial decryption to a state byte at the end of the 4th round, we need to consider a configuration of four bytes in each ciphertext and guess the corresponding bytes of the 6th round key, according to one of the configurations shown in Figure 1. Observe that each configuration has exactly one byte per state row and one byte per state column.

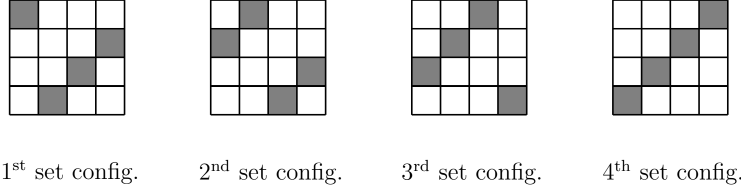

Figure 1. The set of 4 bytes of the 6th round key (resp. ciphertexts) for the Partial Sum Attack on 6-round AES

In the following computations, with abuse of notation, we denote by *MixColumns*$^{-1}$ and *SubBytes*$^{-1}$ the inverse of *MixColumns* and *SubBytes* applied to a single column of the state. The relations between the $a^{(l)}$'s, the $c^{(l)}$'s and the $k^{(h)}$'s are easily established:

$$a_j^{(l)} = \begin{bmatrix} a_{0,j}^{(l)} \\ a_{1,j}^{(l)} \\ a_{2,j}^{(l)} \\ a_{3,j}^{(l)} \end{bmatrix} = \mathit{MixColumns}^{-1}\left(\mathit{SubBytes}^{-1}\begin{pmatrix} c_{0,j}^{(l)} + k_{0,j}^{(6)} \\ c_{1,(j-1) \bmod 4}^{(l)} + k_{1,(j-1) \bmod 4}^{(6)} \\ c_{2,(j-2) \bmod 4}^{(l)} + k_{2,(j-2) \bmod 4}^{(6)} \\ c_{3,(j-3) \bmod 4}^{(l)} + k_{3,(j-3) \bmod 4}^{(6)} \end{pmatrix}\right),$$

where $j \in \{0, 1, 2, 3\}$. When j is understood, we will remove it; for example we denote by

$$\xi^{(l)} = \begin{bmatrix} \xi_0^{(l)} \\ \xi_1^{(l)} \\ \xi_2^{(l)} \\ \xi_3^{(l)} \end{bmatrix} := \mathit{SubBytes}^{-1}\begin{pmatrix} c_{0,j}^{(l)} + k_{0,j}^{(6)} \\ c_{1,(j-1) \bmod 4}^{(l)} + k_{1,(j-1) \bmod 4}^{(6)} \\ c_{2,(j-2) \bmod 4}^{(l)} + k_{2,(j-2) \bmod 4}^{(6)} \\ c_{3,(j-3) \bmod 4}^{(l)} + k_{3,(j-3) \bmod 4}^{(6)} \end{pmatrix},$$

for $1 \leq l \leq 2^{32}$. Let N be the byte matrix of *MixColumns*$^{-1}$. Working out the product, we have

$$a_j^{(l)} = \begin{bmatrix} N_0 \cdot \xi_0^{(l)} + N_1 \cdot \xi_1^{(l)} + N_2 \cdot \xi_2^{(l)} + N_3 \cdot \xi_3^{(l)} \\ N_3 \cdot \xi_0^{(l)} + N_0 \cdot \xi_1^{(l)} + N_1 \cdot \xi_2^{(l)} + N_2 \cdot \xi_3^{(l)} \\ N_2 \cdot \xi_0^{(l)} + N_3 \cdot \xi_1^{(l)} + N_0 \cdot \xi_2^{(l)} + N_1 \cdot \xi_3^{(l)} \\ N_1 \cdot \xi_0^{(l)} + N_2 \cdot \xi_1^{(l)} + N_3 \cdot \xi_2^{(l)} + N_0 \cdot \xi_3^{(l)} \end{bmatrix},$$

where

$$N_0 = \alpha^3 + \alpha^2 + \alpha$$
$$N_1 = \alpha^3 + \alpha + 1$$
$$N_2 = \alpha^3 + \alpha^2 + 1$$
$$N_3 = \alpha^3 + 1$$

and α is a defining element of $\mathbb{F}_{2^8} = \mathbb{F}_2[x] / < x^8 + x^4 + x^3 + x + 1 >$, i.e. α is such that $\alpha^8 = \alpha^4 + \alpha^3 + \alpha + 1$.
Thus we can compute a state byte at the end of the 4th round as follows:

$$b^{(l)}_{i,(j+i) \bmod 4} = \gamma^{-1}\left(a^{(l)}_{i,j} + \bar{k}^{(5)}_{i,j}\right), \tag{3}$$

where $i \in \{0,1,2,3\}$ and γ^{-1} is the S-box of *SubBytes*$^{-1}$. Observe that in (3) γ^{-1} is applied to $a^{(l)}_{i,j} + \bar{k}^{(5)}_{i,j}$ rather than to $a^{(l)}_{i,j} + k^{(5)}_{i,j}$. The latter would be wrong, since $k^{(5)}_{i,j}$ is added *after* the application of *MixColumns*.

In order to identify a possible right guess, we have to check if $\sum_{l=1}^{2^{32}} b^{(l)}_{i,(j+i) \bmod 4} = 0$. This sum can be expressed as

$$\sum_{l=1}^{2^{32}} \gamma^{-1}\left(N_{-i} \cdot \xi^{(l)}_0 + N_{1-i} \cdot \xi^{(l)}_1 + N_{2-i} \cdot \xi^{(l)}_2 + N_{3-i} \cdot \xi^{(l)}_3 + \bar{k}^{(5)}_{i,j}\right) \tag{4}$$

where the indices $-i, 1-i, 2-i, 3-i$ are all meant to be reduced modulo 4, giving a remainder in $\{0,1,2,3\}$.
If we trivially execute this summation, given 2^{32} ciphertexts and 2^{40} possible key guesses, we have to sum 2^{72} different values, which does not significantly improve the basic Square Attack. As it is pointed out in [FKL$^+$01], Expression (4) can be organized in a more efficient manner. Once the row i is fixed, for each $t \in \{0,1,2,3\}$, it is possible to associate a partial sum $x^{(l)}_t$ to each set $\{\xi^{(l)}_0, \ldots, \xi^{(l)}_t\}$, defined as follows:

$$x^{(l)}_t := \sum_{z=0}^{t} N_{z-i} \cdot \xi^{(l)}_z.$$

In particular,

$$x^{(l)}_2 = x^{(l)}_1 + N_{2-i}\xi^{(l)}_2 \text{ and } x^{(l)}_3 = x^{(l)}_2 + N_{3-i}\xi^{(l)}_3.$$

In order to simplify the notation, let $(c^{(l)}_0, c^{(l)}_1, c^{(l)}_2, c^{(l)}_3)$ be the 4-tuple formed by the lth ciphertext's bytes, extracted according to one of the configurations described above. Guessing the key values and using the partial sums, we can define some maps

$$(c^{(l)}_0, c^{(l)}_1, c^{(l)}_2, c^{(l)}_3) \longmapsto (x^{(l)}_1, c^{(l)}_2, c^{(l)}_3) \longmapsto (x^{(l)}_2, c^{(l)}_3) \longmapsto x^{(l)}_3.$$

Using a similar notation, let (k_0, k_1, k_2, k_3) be four bytes of the 6th round key which we want to guess, arranged in the same configuration chosen for the ciphertexts, and let k_4 be a guess for the 5th round key byte $\bar{k}_{i,j}^{(5)}$. The Partial Sum Attack is organized as follows.

- We start with the list of 2^{32} 4-tuples $(c_0^{(l)}, c_1^{(l)}, c_2^{(l)}, c_3^{(l)})$. Guessing k_0 and k_1, we can compute $x_1^{(l)}$ and count how many times each triple $(x_1^{(l)}, c_2^{(l)}, c_3^{(l)})$ appears during this computation. As there are only 2^{24} possible values for these three bytes, we do not have to list all the triples, but it is sufficient to count how often each one occurs (and actually it is enough to record the triples that appear an odd number of times).
- We then guess k_2 and compute how often each pair $(x_2^{(l)}, c_3^{(l)})$ occurs.
- Similarly, we guess k_3 and compute how often each value of $x_3^{(l)}$ occurs.
- Finally, guessing the value of k_4, we can compute Expression (4) and check if the result is zero.

Complexity

In the first phase one guesses 2 bytes and processes 2^{32} ciphertexts bytes. For each choice of k_0 and k_1, one more byte has to be guessed, but only 2^{24} triples have to be processed. In the third phase k_3 has to be guessed but one has only to process 2^{16} pairs. This holds similarly for the other two phases. Summing up all the contributions we obtain that 2^{50} operations are required for a single $\bar{\Delta}$-set of 2^{32} elements. In [FKL+01], it is shown that 6 different $\bar{\Delta}$-sets are necessary in order to determine the correct 5-tuple $(k_0, k_1, k_2, k_3, k_4)$ with overwhelming probability. This can be further improved as it will be explained in Section 3.2.

3 Implementation and improvement

The results described in this work started from Francesco Aldà's Master's thesis [Ald13], where he developed a C++ code of the Partial Sum Attack and introduced the improvement specified in Section 3.2.

3.1 High-level scheme of the implementation

After examining the literature which was developed and produced after the publication of the Partial Sum Attack, we believe that an effective implementation of this attack had not already been presented before Francesco Aldà's thesis and this work.
In this section we introduce and explain the main ideas we used in our implementation of the Partial Sum Attack on 6-round AES. We refer to Section 3.3 for further technical details on our implementation.

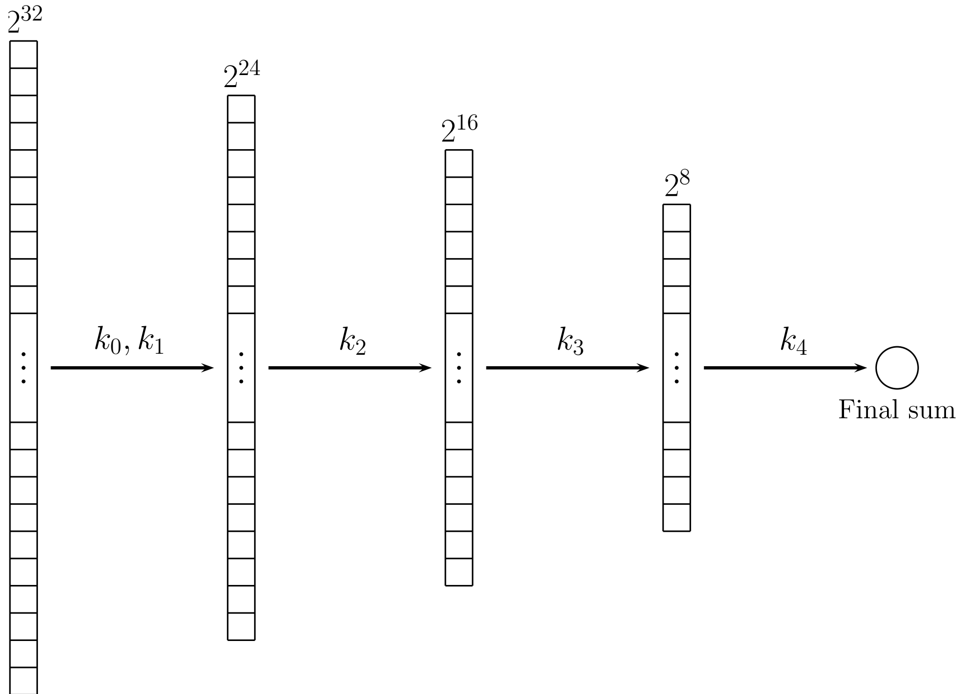

Figure 2. Partial Sum Attack workflow

As it is displayed in Figure 2, the attack's steps are very simple. At the beginning of the attack, a $\bar{\Delta}$-set with 2^{32} elements has to be encrypted. In this way, we can obtain and store the 4-tuples $(c_0^{(l)}, c_1^{(l)}, c_2^{(l)}, c_3^{(l)})$, formed by the lth ciphertext's bytes, extracted according to one of the configurations described in Section 2.2.
Following the idea explained in [FKL+01], it is sufficient to count how many times each 4-tuple appears during the computation. As there are only 2^{32} possible 4-tuples, we do not have to store all $(c_0^{(l)}, c_1^{(l)}, c_2^{(l)}, c_3^{(l)})$ values. Since all the operations we are dealing with are computed in a field of characteristic 2, it suffices to only count modulo 2. In fact, only the elements which appear an odd number of times in Expression (4) give a non-zero contribution. Hence, a single bit suffices for each count and it is possible to store our list of 4-tuples into a 2^{32}-bit vector. Therefore, the space requirement for 2^{32} counters is just 2^{32} bits, which correspond to 0.5 *GigaBytes*.
We have then to start a loop over 2^{16} possible values of k_0, k_1. For each pair (k_0, k_1), we are able to compute the partial sums $x_1^{(l)}$ and store the triples $(x_1^{(l)}, c_2^{(l)}, c_3^{(l)})$. Using the same rationale, it suffices to count the parity of the times each triple occurs. Again, we store all the parities in a 2^{24}-bit vector.
Moreover, using an appropriate sorting, it suffices to compute the value $x_1^{(l)}$ every 2^{16} elements: in fact, this value only depends on $c_0^{(l)}, c_1^{(l)}, k_0$ and k_1. Thus,

if $1 \leq l, h \leq 2^{32}$, we have

$$\begin{cases} c_0^{(l)} = c_0^{(h)} \\ c_1^{(l)} = c_1^{(h)} \end{cases} \Longrightarrow x_1^{(l)} = x_1^{(h)}.$$

The same ideas can be similarly applied to the second step. For each value k_2, one can compute the partial sums $x_2^{(l)}$, count the parity of the times each pair $(x_2^{(l)}, c_3^{(l)})$ occurs and store it into a 2^{16}-bit vector. As before, it suffices to compute the value $x_2^{(l)}$ every 2^8 elements: in fact, this value only depends on $x_1^{(l)}, c_2^{(l)}$ and k_2.
In the third step, for each value k_3, we can compute the partial sums $x_3^{(l)}$, count how many times each $x_3^{(l)}$ occurs and store its parity into a 2^8-bit vector. Unlike the previous steps, this must be done scanning every entry of the 2^{16}-bit vector.
Finally, looping over the value k_4, it is possible to compute the final sum and check if the result is zero (*verification step*).
As it was explained for the Square Attack, checking this sum for a single $\bar{\Delta}$-set is expected to eliminate 255 of the wrong key assumptions $(k_0, k_1, k_2, k_3, k_4)$. It is therefore necessary to verify their correctness using different $\bar{\Delta}$-sets. At each positive verification, the key space is reduced by a factor 2^{-8}. Apparently this implies that 6 different $\bar{\Delta}$-sets (or more) are need to find the correct 5-tuple $(k_0, k_1, k_2, k_3, k_4)$ with overwhelming probability. This result can be improved, as it is explained in the following section.

3.2 Improvement

As it was already underlined, when it was published, the Partial Sum Attack represented one of the best cryptanalytic results on reduced-round versions of the Advanced Encryption Standard. After its publication, many other researchers worked on the integral cryptanalysis of AES, finding new extensions or improvements for this class of attacks (see for example [GM08, LW11, Tun12]).
Our approach started from performing a full implementation of the attack as it is described in Section 3.1, trying to understand where some other potentialities could be exploited. We believe that this effort can lead to better understand the dynamic of an attack.

In the original paper [FKL+01], it is claimed that at least 6 sets of 2^{32} plaintexts, which form a $\bar{\Delta}$-set, are necessary in order to find the correct 5-tuple $(k_0, k_1, k_2, k_3, k_4)$. We claim that only two $\bar{\Delta}$-sets are necessary in order to find the correct $(k_0, k_1, k_2, k_3, k_4)$. In fact, fixing one configuration according to which the four ciphertexts bytes $(c_0^{(l)}, c_1^{(l)}, c_2^{(l)}, c_3^{(l)})$ are extracted, one can compute the sum in four different state bytes at the end of the 4th round. In Figure 3 we provide an example.

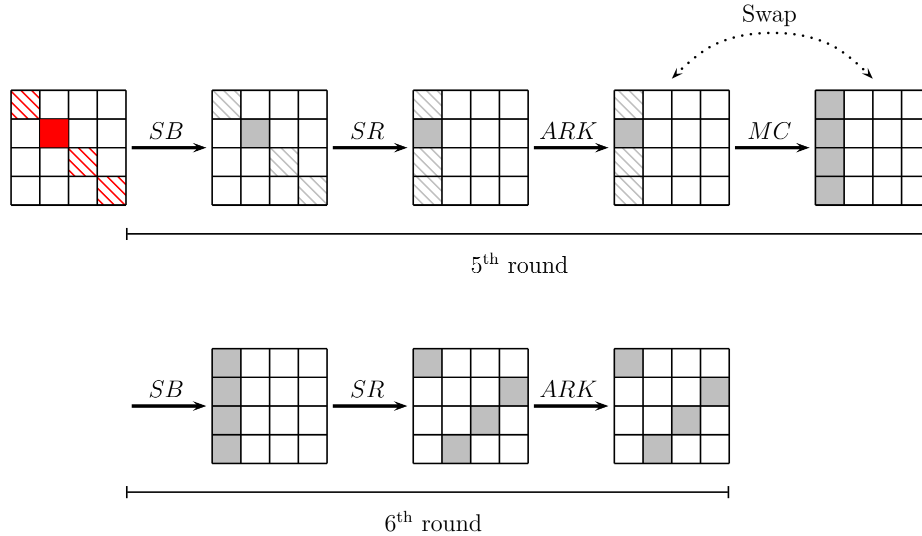

Figure 3. The state bytes at the end of the 4th round (in red) which can be computed for a configuration according to which four ciphertexts bytes $(c_0^{(l)}, c_1^{(l)}, c_2^{(l)}, c_3^{(l)})$, for $1 \leq l \leq 2^{32}$, are extracted

If we consider each sum as indipendent, using only two $\bar{\Delta}$-sets, the probability that for a 4-tuple (k_0, k_1, k_2, k_3) exists for each row a value k_4 which gives a zero sum for both $\bar{\Delta}$-sets is

$$\left(\frac{1}{256}\right)^8.$$

Therefore checking the value of the sum for at least three bytes at the end of the 4th round, for which there exist values k_4 (dependent only on the row) which give zero sums for both $\bar{\Delta}$-sets, is expected to eliminate all but one 4-tuple (k_0, k_1, k_2, k_3), that is, the correct one.
Note that the values of the 5th round key bytes k_4, which produce zero sums, may be different for each row, but, as for (k_0, k_1, k_2, k_3), their correctness follows by the crosschecking between the two $\bar{\Delta}$-sets.
The hypothesis which bears our result consists of considering the sums on four rows as independent. We believe that this hypothesis substantially holds even in practice. In fact, even though the bytes involved in the sums belong to the same state and their correlation is hence nonzero, the diffusion and confusion introduced by the round transformations should have made it negligible.
Therefore we observed that the number of chosen plaintexts which are necessary in order to mount the attack can be reduced from 6 $\bar{\Delta}$-sets of 2^{32} elements to only 2.
This theoretical result states that the number of chosen plaintexts which are needed to successfully mount the attack can be significantly reduced. In general, this reduction implies that more verification steps are needed, since one can check only two sums instead of six, which might increment the number of false positive key bytes, which are discarded with very high probability by

the crosschecking on other rows. Therefore in our implementation we used 3 $\bar{\Delta}$-sets of 2^{32} elements and checked the value of the sum for four bytes at the end of the 4th round, in order to lower as much as possible the probability of false positives. We also made this choice since using only 2 $\bar{\Delta}$-sets involves more verification steps (there are more wrong key candidates which give a zero sum), which are time consuming operations in our implementation. We simply observed that using 3 $\bar{\Delta}$-sets resulted slightly faster and easier to manage.

This improvement has been reached indipendently from [Tun12], where a quite similar result is described. In fact, our result is based on different observations and reaches different conclusions.

3.3 Implementation's details

First of all, we ported Francesco Aldà's code to C, to reduce the overhead of C++ abstractions, which are useful but not essential for this kind of application. During this phase, we decided to map every Boolean vector's element to a bit inside an `unsigned char`'s array. This also allowed us to save space and time while writing and reading the encrypted arrays to and from the disk, storing every 2^{32} array into a 512MB file. We decided to save on disk the encrypted vectors as binary files, to save time while testing the attack; on one hand, this process forced us to create some ancillary functions, to toggle and mask bits as necessary but, on the other hand, it had the side effect of accelerating some functions where shifting and masking were required, because we did it byte by byte, instead of bit by bit. After completing the porting and introducing the new memory management concepts, we started focusing on how the memory management operations could be accelerated and we ended up managing every group of 8 `unsigned char` array elements as an `unsigned long long int` array, where possible. This allowed us to deal with the allocated memory as a set of 64 bit blocks, reducing the time needed to complete, for example, some `XOR` operation between these arrays. The resulting implementation was satisfactory.

We also decided to allow the parallelization of the attack on multiple core systems and we needed to exchange information between each process. We chose `OpenMPI` [GFB+04, GWS06] because we appreciated its documentation and the maturity of the open source project supporting it. Porting the code from a linear to a parallel paradigm presents no real difficulties because the attack is mainly composed by loops, repeated for values from 0 to 255, so we decided to execute the 5 most inner loops on each worker (a worker is a parallel process running the attack), assigning to each of them a range of values of k_1 to go through in the outer of these 5 loops. Moreover, we shared the encrypted vectors, using `NFSv4`, on every system running the attack, and using the same share storage to save the guessed partial keys and to check the status of the attack from each worker. The master process coordinating the attack distributes the values to each worker using the `Round-Robin algorithm` [SGG08] and then waits for replies from each of them. After finish-

ing the attack with one of the assigned values, every worker reports the result to the master, if successful. If the attack with that value was not successful, the worker asks the master if any other was successful and, if so, it stops the attack, otherwise it starts the attack with the next assigned value.

To retrieve the whole 16 bytes key, the attack has to be run 4 times, according to the four configurations shown in Figure 1. The master writes 4 files that contain the partial keys guessed and it also writes the whole key in another file, when the attack is completed for every configuration.

The final outcome of this effort was interesting in terms of memory and time. The attacks have been launched on 6 desktop PC, with 4 cores (Intel Pentium CPU G640 @2.80GHz) and 8GB of RAM each, using 25 processes. The first process coordinated the attacks, while the 24 workers performed the attacks in 11,5 days on average. Based on this result, we estimate that, on average, the 128-bit 6th round key can be guessed in 25,8 hours, using 256 workers with less than 2GB of RAM used by each worker.

Acknowledgements

Most of the results shown in this work were developed in the first author's Master's thesis and he would like to thank the other authors, especially his supervisor (the last author). For interesting discussions, the authors would like to thank Anna Rimoldi.

References

[Ald13] F. Aldà, *The Partial Sum Attack on 6-round reduced AES: implementation and improvement*, Master's thesis (laurea magistrale), University of Trento, Department of Mathematics, 2013.

[DKR97] J. Daemen, L. Knudsen, and V. Rijmen, *The block cipher Square*, Proc. of FSE 1997, LNCS, vol. 1267, 1997, pp. 149–165.

[DR98] J. Daemen and V. Rijmen, *AES proposal: Rijndael*, Tech. report, NIST, 1998.

[DR02] J. Daemen and V. Rijmen, *The design of Rijndael*, Information Security and Cryptography, Springer-Verlag, 2002, AES - the Advanced Encryption Standard.

[FKL+01] N. Ferguson, J. Kesley, S. Lucks, B. Schneier, M. Stay, D. Wagner, and D. Whitinf, *Improved cryptanalysis of Rijndael*, Proc. of FSE 2001, LNCS, vol. 1978, Springer, 2001, pp. 213–230.

[GFB+04] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*, LNCS, vol. 3241, Springer, 2004, pp. 97–104.

[GM08] S. Galice and M. Minier, *Improving Integral Attacks against Rijndael-256 up to 9 rounds*, Proc. of AFRICACRYPT 2008, LNCS, vol. 5023, Springer, 2008, pp. 1–15.

[GWS06] R. L. Graham, T. S. Woodall, and J. M. Squyres, *Open MPI: A Flexible High Performance MPI*, LNCS, vol. 3911, Springer, 2006, pp. 228–239.

[LW11] Y. J. Li and W. L. Wu, *Improved Integral Attacks on Rijndael*, Journal of Information Science and Engineering **27** (2011), no. 6, 2031–2045.

[SGG08] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, John Wiley & Sons, 2008.

[Tun12] M. Tunstall, *Improved "Partial Sums"-based Square Attack on AES*, Proc. of SECRYPT 2012, INSTICC Press, 2012, pp. 25–34.