Study of a Parity Check Based Fault-Detection Countermeasure for the AES Key Schedule

Christophe Clavier, Julien Francq, Antoine Wurcker

Study of a Parity Check Based Fault-Detection Countermeasure for the AES Key Schedule

Christophe Clavier[1], Julien Francq[2], and Antoine Wurcker[1]

[1] Université de Limoges, XLIM-CNRS
Limoges, France
christophe.clavier@unilim.fr
antoine.wurcker@xlim.fr
[2] Airbus Defence & Space - Cybersecurity
Élancourt, France
julien.francq@airbus.com

Abstract. In this paper we study a parity check based countermeasure proposed by Chen et al. that thwarts their attack by detecting byte fault injection during the AES key schedule process.
We provide a generalization of their approach that allows to derive parity equations for every AES sizes not given by the authors. We analyze why Chen et al. countermeasure does not properly works. Doing so we are able to extend the coverage of the fault detection to the full expanded key. Finally we suggest optimizations that reduce memory and computation costs, and propose an adaptation to a more general fault model.

Keywords: side-channel analysis, fault attacks, parity check countermeasure, AES key schedule

1 Introduction

Beside Side Channel Analysis (SCA) originally revealed by Kocher in 1996 [15] and later improved in different ways [16, 5, 11], Differential Fault Analysis (DFA) introduced in 1997 [2, 3] is another powerful means to jeopardize implementations of embedded cryptography. In DFA an attacker provokes faults during the execution of a cryptographic algorithm in order to extract information about the secret by analyzing the differential effect on the outputs.

The Advanced Encryption Standard (AES) is the current symmetric encryption standard since it has been adopted by the NIST[3] in 2001 [17]. First DFA applied on AES [8, 12, 19] essentially consisted in modifying the value of a state byte near the end of the encryption path (typically in the 8^{th} round of AES-128). In another type of DFA on AES the attacker injects a fault during the key schedule while the expanded key is computed on-the-fly.

Fault attacks on the AES key schedule were first introduced by Giraud [12] where random byte faults are injected on K_9, K_8 and M_8 (the state after the 8^{th} round). If the locations are correctly chosen, 31 faulty ciphertexts are required and 2^{16} candidates are remaining at the end of the process (a brute-force is then feasible to deduce the key). Later Chen et al. [6] also use random byte fault injections at chosen locations but only on K_9 and K_8. Here, 22 faulty ciphertexts are needed to retrieve the key, where 2^{24} candidates are remaining. In [18] Peacham et al. need to inject random word faults on chosen locations on only K_9. Only 12 faulty ciphertexts are required, and at the end of the fault process, there is only one candidate left (the key is then retrieved without uncertainty). In [20] Takahashi et al. propose a powerful DFA where only two faults on a random word (column) of K_8

[3] National Institute of Standards and Technology

are required and where a brute-force can be used to retrieve the correct key among 2^{40} candidates. This number can be decreased if the attacker can induce four faults instead of two, and then, the number of candidates can be decreased up to 2^{16}. A more powerful DFA on the key schedule has been presented by Kim et al. [14]. Like in [20] only two faults are required but on three bytes of K_9 and with a computation complexity of only 2^{32}. Moreover, four faults lead to the key without uncertainty. Finally, the ultimate goal of a DFA on AES key schedule requiring only a single faulty ciphertext has been reached by Ali et al. [1]. They exploit a fault that must be injected in the first column of K_8 and only 2^8 key candidates remain to be exhausted.

Very few attacks in literature are considering higher versions of AES key schedule: AES-192 and AES-256. In [9] Floissac et al. manage to adapt [14] to these two sizes with 16 faulty ciphertexts. This attack has been improved to around 4 faulty ciphertexts by Kim in [13].

In [6] Chen et al. propose a DFA on AES key schedule that improves the original attack of Giraud [12]. Interestingly, they also provide a parity check based countermeasure supposed to protect implementations from their own attack. The Chen et al.'s paper is cited more than hundred times including at least [20, 10] where authors recommends this countermeasure as protection against fault injections. As far as the authors know, this countermeasure has not been investigated for the 192 and 256 bits versions of the AES. Furthermore, no security proof has been provided by the authors. This implies that this flawed countermeasure may have been actually implemented. The motivations of this paper are threefold: extend – and evaluate this extension – Chen et al.'s countermeasure to the AES-192 and 256, assess the real security of the initial method of Chen et al. and correct its flaws by providing right parity formulae.

Chen et al.'s countermeasure and the notations used in this paper are introduced in Sect. 2. We then provide a study of the countermeasure security in Sect. 3. Finally we propose corrections, optimizations and an evaluation of the cost for this countermeasure in Sect. 4 and conclude the paper in the last section.

2 Notations and Background

2.1 Notations

Throughout this paper we use the following notations that apply for all 128-, 192- and 256-bit versions of the AES:

- K_r the r^{th} 128-bit round key with $r = 0, \ldots, 10|12|14$
- $K_{r,i}$ the i^{th} byte of K_r with $i = 0, \ldots, 15$
- w_i the i^{th} 4-byte column of the expanded key with $i = 0, \ldots, 43|51|59$
- $w_{i,j}$ the j^{th} byte of w_i with $j = 0, \ldots, 3$
- RW the operation `RotWord`
- SW the operation `SubWord`
- RC the operation `Rcon`

We also denote by N_i and N'_i the 4-byte non-linear vectors defined as:

AES-128 $N_i = \texttt{Rcon}(\texttt{SubWord}(\texttt{RotWord}(w_{4*i-1})))$ with $i = 1, \ldots, 10$

AES-192 $N_i = \texttt{Rcon}(\texttt{SubWord}(\texttt{RotWord}(w_{6*i-1})))$ with $i = 1, \ldots, 8$

AES-256 $N_i = \texttt{Rcon}(\texttt{SubWord}(\texttt{RotWord}(w_{8*i-1})))$ with $i = 1, \ldots, 7$
$N'_i = \texttt{SubWord}(w_{8*i+3})$ with $i = 1, \ldots, 6$

2.2 AES Key Schedule

We now give a brief description of AES key schedule for all key sizes. We refer the reader to the NIST standard [17] for more detailed information.

The 128-bit key schedule uses a 128-bit ciphering key as a 4×4 byte matrix and successively derives new 4-byte columns by combining the previous column and the column at the same place in previous matrix as can be seen in Fig. 1. For the first column of each matrix a prior transformation is applied on the previous column which is constituted by `RotWord` (RW) that vertically rotates the column, `SubWord` (SW) that replaces every byte by its image through the AES S-Box and `Rcon` (RC) that XOR-es the column with a round-dependent constant. The other columns combination is simply a XOR operation. Eleven 128-bit round keys K_0 to K_{10} are generated.

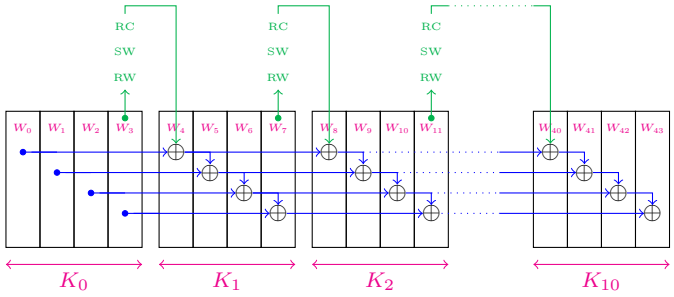

Fig. 1. The AES-128 key schedule.

The 192-bit key schedule uses a 192-bit ciphering key as a 4×6 byte matrix and generates thirteen 128-bit round keys K_0 to K_{12} by a similar process as for the 128-bit version as can be seen in Fig. 2. Each column is obtained by XOR-ing the previous column with the column at the same position in previous matrix. Here also, there is a prior transformation that applies when computing the first column of each matrix. All 4×6 matrices contain one 128-bit round key and a half. One can remark that the two last columns of the last matrix do not need to be computed.

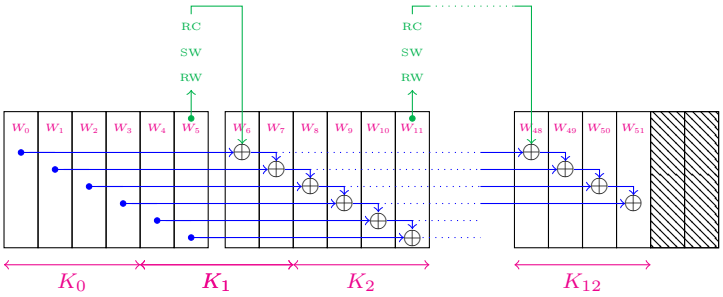

Fig. 2. The AES-192 key schedule.

The 256-bit key schedule uses a 256-bit ciphering key as a 4×8 byte matrix and also generates fifteen 128-bit round keys K_0 to K_{14} by a similar process than for other AES versions as can be seen in Fig. 3. One can note a difference though, which is an extra non-linear transformation that applies when computing each fifth column of a matrix by application of a `SubWord` step on the previous column before XOR-ing it. All 4×8 matrix

contain two 128-bit round keys. One can remark that the four last columns of the last matrix do not need to be computed.

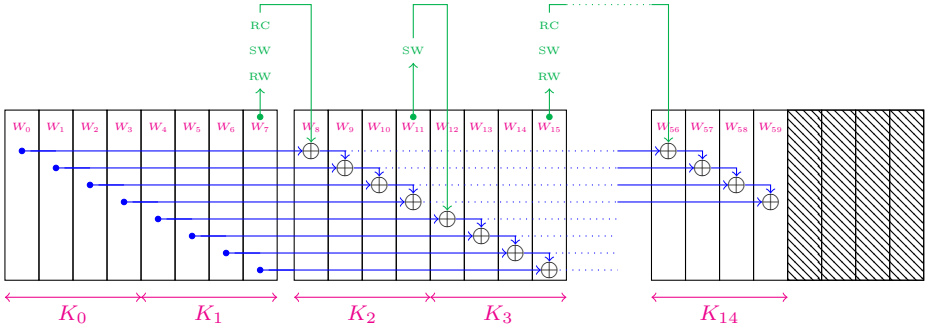

Fig. 3. The AES-256 key schedule.

2.3 Chen and Yen's Countermeasure

As DFA on AES also concerns the key schedule, countermeasures are required for this process. In order to protect from such attacks, countermeasure like duplication (computing the key schedule twice and compare the results) or parity checking can be implemented.

Chen et al.'s countermeasure is a parity check based countermeasure which detects byte fault injections, and consists in:

(i) separating the linear part and the non-linear part of the key schedule,
(ii) storing – or preferably accumulating – the values of the non-linear 4-byte vectors during round keys derivation,
(iii) checking that the last round key verifies particular relations involving the first round key and the non-linear vectors.

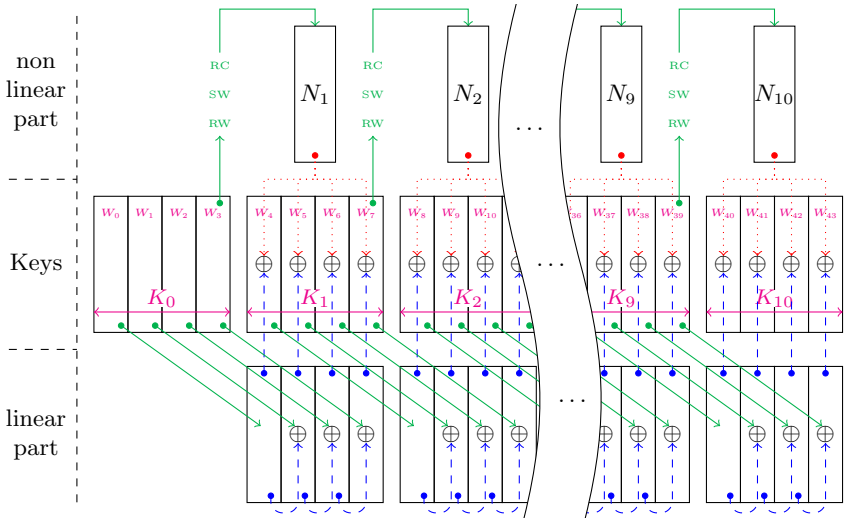

Fig. 4. Illustration of countermeasure on AES-128 key schedule.

The separation of the linear and non-linear parts is illustrated on Fig. 4 that describes the countermeasure applied on 128-bit AES key schedule. The non-linear process induced

by the `SubWord` and the `Rcon` steps generates ten 4-byte vectors N_1 to N_{10} that are stored for future usage. During the computation of K_i the vector N_i is obtained by successively applying `RotWord`, `SubWord` and `Rcon` operations on the last column of previous key:

$$N_i = RC(SW(RW(w_{4*i-1})))$$

The linear part consists in cumulatively XOR-ing columns of the previous matrix.

Finally the new key columns are obtained by XOR-ing columns of the linear part with the column N_i of the non-linear part. Here are the equations for K_i columns:

$$\begin{aligned}
w_{4*i+0} &= N_i \oplus \{ \quad w_{4*(i-1)} \\
w_{4*i+1} &= N_i \oplus \begin{cases} w_{4*(i-1)} \\ \oplus w_{4*(i-1)+1} \end{cases} \\
w_{4*i+2} &= N_i \oplus \begin{cases} w_{4*(i-1)} \\ \oplus w_{4*(i-1)+1} \\ \oplus w_{4*(i-1)+2} \end{cases} \\
w_{4*i+3} &= N_i \oplus \begin{cases} w_{4*(i-1)} \\ \oplus w_{4*(i-1)+1} \\ \oplus w_{4*(i-1)+2} \\ \oplus w_{4*(i-1)+3} \end{cases}
\end{aligned}$$

Chen et al. base their countermeasure on the fact that storing the N_i values allows to check parity relations between the first and the last key. They propose the following equations to check columns and rows parity.

– for every row $i = 0, \ldots, 3$ the following equation holds:

$$\begin{aligned}
&K_{10,i} \oplus K_{10,4+i} \oplus K_{10,8+i} \oplus K_{10,12+i} \\
&\quad = K_{0,8+i} \oplus K_{0,12+i} \oplus N_{3,i} \oplus N_{7,i}
\end{aligned} \tag{1}$$

– for the last column of the last key the following equation holds:

$$\bigoplus_{i=0}^{3} K_{10,12+i} = \bigoplus_{i=0}^{3} (K_{0,4+i} \oplus K_{0,12+i} \oplus N_{2,i} \oplus N_{6,i} \oplus N_{10,i}) \tag{2}$$

A method is given in order to get other columns equations and authors claim that row checks are able to detect any fault induced into K_8 to K_{10}.

3 Study of the Countermeasure

3.1 Deriving Equations

Chen et al. obtain their equations by computing from the first to the last key. They do not give equations for AES-192 nor AES-256. We detail hereafter a generalization allowing to derive equations from the last key to the first one so that it can be adapted for all AES key sizes. All the processes are illustrated on Fig. 5, 7 and 9.

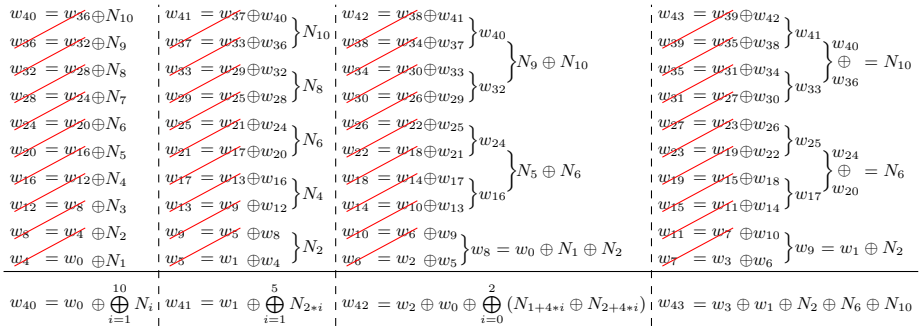

Fig. 5. Method to obtain equations between last key and first key for AES-128.

Equations for AES-128 First of all we have to point out some relations inherited from AES-128 key schedule construction for all rounds $r = 1, \ldots, 10$:

$$w_{4*r} = w_{4*(r-1)} \oplus N_r \tag{3}$$

$$w_{4*r+1} = w_{4*(r-1)+1} \oplus w_{4*r} \tag{4}$$

$$w_{4*r+2} = w_{4*(r-1)+2} \oplus w_{4*r+1} \tag{5}$$

$$w_{4*r+3} = w_{4*(r-1)+3} \oplus w_{4*r+2} \tag{6}$$

Equation (3) shows the relation that exists between the first column of a key, the first column of the previous key and a non-linear vector. Equations (4) to (6) show the relation that exists between others columns of a key, the corresponding column of the previous key and the previous column.

For the first column of the last key w_{40} we combine together all round equations of type (3). After simplification it remains:

$$w_{40} = w_0 \oplus \bigoplus_{i=1}^{10} N_i \tag{7}$$

For the second column of the last key w_{41} we combine together all round equations of type (4). After simplification it still remains ten first columns residues that we combine pairwise – by using equations of type (3) – to obtain five non-linear vectors:

$$w_{41} = w_1 \oplus \bigoplus_{i=1}^{5} N_{i*2} \tag{8}$$

A similar process is applied for the two last columns w_{42} and w_{43} of the last key. After simplification and combination using Equations (3) to (6) as depicted on Fig. 5 we obtain relations between only the first key, the last key and non-linear vectors:

$$w_{42} = w_2 \oplus w_0 \oplus \bigoplus_{i=0}^{2} (N_{1+4*i} \oplus N_{2+4*i}) \tag{9}$$

$$w_{43} = w_3 \oplus w_1 \oplus N_2 \oplus N_6 \oplus N_{10} \tag{10}$$

Note that Chen et al. have already found Equations (7) to (10). The method described here aims at simplifying the derivation of these equations and presents the advantage to be straightforwardly applicable for other key sizes.

Equations for AES-192 The AES-192 key schedule is very similar to the AES-128 one. The main difference is that it involves 6-column matrices to generate 13 round keys in only 8 steps producing non-linear vectors N_1 to N_8. The countermeasure is illustrated on Fig. 6.

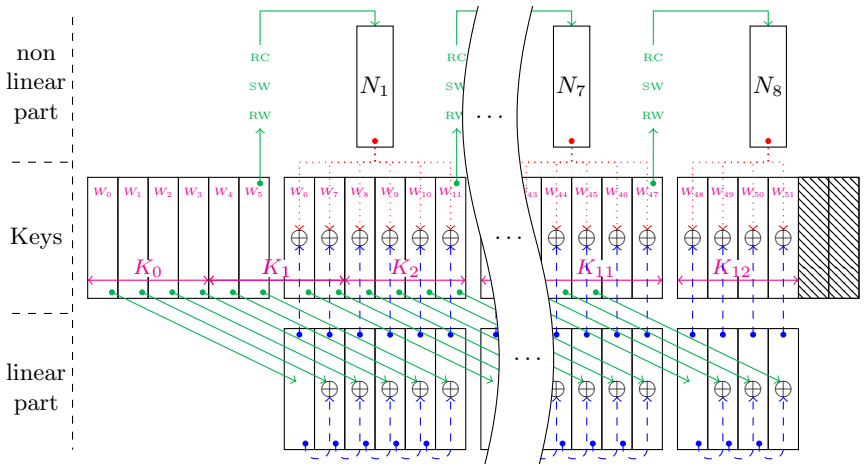

Fig. 6. Illustration of countermeasure on AES-192 key schedule.

The method to derive equations for AES-128 can still be applied to relate last key columns to first key ones. Equations (3) to (6) become:

$$w_{6*i} = w_{6*(i-1)} \oplus N_i \tag{11}$$
$$w_{6*i+1} = w_{6*(i-1)+1} \oplus w_{6*i} \tag{12}$$
$$w_{6*i+2} = w_{6*(i-1)+2} \oplus w_{6*i+1} \tag{13}$$
$$w_{6*i+3} = w_{6*(i-1)+3} \oplus w_{6*i+2} \tag{14}$$

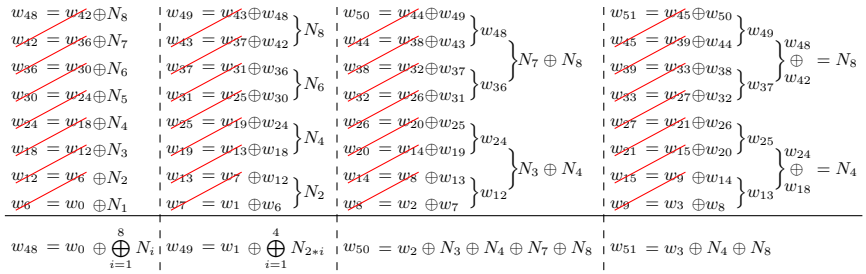

Fig. 7. Method to obtain equations between last key and first key for AES-192.

Step by step description and resulting equations are given in Fig. 7.

Equations for AES-256 The AES-256 key schedule is different from the two other versions. There are 7 steps in key schedule that generate 15 round keys using two different non-linear transformations at each step. The computation of each 8-column matrix involves two types of non-linear vectors: N_1 to N_7 for the first half of each matrix, and N'_1 to N'_6 for the second half (except for the last key). The countermeasure is illustrated on Fig. 8.

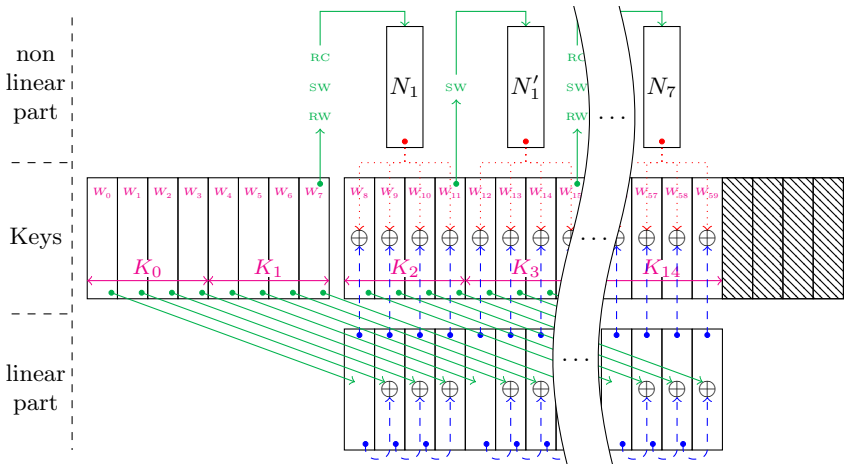

Fig. 8. Illustration of countermeasure on AES-256 key schedule.

Here also we can apply the same method as for other key sizes to derive equations that relate last key columns to first key ones:

$$w_{8*i} = w_{8*(i-1)} \oplus N_i \tag{15}$$

$$w_{8*i+1} = w_{8*(i-1)+1} \oplus w_{8*i} \tag{16}$$

$$w_{8*i+2} = w_{8*(i-1)+2} \oplus w_{8*i+1} \tag{17}$$

$$w_{8*i+3} = w_{8*(i-1)+3} \oplus w_{8*i+2} \tag{18}$$

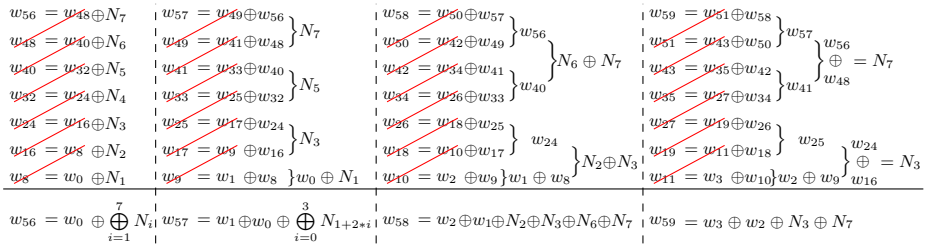

Fig. 9. Method to obtain equations between last key and first key for AES-256.

One can remark that these equations are independent from $\{N'_i\}_{i=1..6}$ non-linear vectors because the last round key is made of only the first 4 columns of the last matrix. Step by step description and resulting equations are given in Fig. 9.

3.2 Countermeasure Behavior Facing Faults

We point out that every term of previously obtained equations are 4-byte columns. We can thus decompose them into four equations that put in relation bytes from the last key, bytes from the first key and bytes from non-linear vectors. For every key size, we thus have 16 so-called *atomic* equations, one per last key byte.

Chen et al. propose to check parity of rows by using equations obtained by XOR-ing the four atomic equations from a same row (cf. Equation (1)). They claim that "The row parity check can detect the fault induced on the eighth to tenth round keys.". We will show that it is partially false. Indeed this rows parity check protects from the attack

described in their paper where byte faults must occur in the last column of K_9 and the two last columns of K_8. But it does not detect every fault injection. They also evoke that a column parity check can be done by using some equations obtained by XOR-ing four atomic equations from a same column (cf. Equation (2)).

In order to evaluate Chen et al. countermeasure we performed simulations that provoke byte fault injections at every location of the expanded key and test if the differentials that propagate to the last key are detected or not by the different equations. For the sake of completeness, we also simulated fault injections into the countermeasure registers, namely the linear and the non-linear parts.

Remark that a fault induced into the ciphering key K_0 is never detected. This is because it is equivalent to encrypt with another key and thus with a fully valid expanded key. In the following sections we only consider byte faults occurring on round keys from K_1 onward.

3.3 Analysis of AES-128 Countermeasure

Fault Propagation Due to a lack of diffusion the fault propagation in AES key schedule is tightly related to the location of the fault, as described by Clavier et al. in [7]. Thus when the location of the fault is known the positions of bytes hit by fault propagation are also known. Furthermore we can predict separately the fault propagation pattern on the linear part and that on the non-linear part, whose superposition gives the propagation pattern on the expanded key.

The countermeasure principle lies in accumulating non-linear vectors during process, in order to verify that combined with last key K_{10} they fit with first key K_0 and then try to detect a differential inserted by fault.

Note that this method can only detect a small part of a differential propagation. Indeed, the progressive accumulation of non-linear vectors during the key calculation process makes that some differentials propagate simultaneously on both expanded key and non-linear vectors. As a result, only the propagation of the original differential on the linear part can be detected by checking parity equations.

This is illustrated on the example given on Fig. 10. An original differential δ_1 is induced by a fault occurring on $K_{4,4}$[4]. This differential propagates on the linear part according to a pattern specific to the fault location. Only this propagation will be noticeable by the countermeasure whose parity check equations do not involve the linear steps. This is due to the fact that other propagated differentials through non-linear steps simplify as they appear twice in the equations, both in the last key and in non-linear vectors. A differential δ_2 is created when δ_1 passes from the last column of K_5 through the non-linear process to create N_6 so we can not relate it to δ_1 without knowing key bytes values. New differentials are created each time a differential passes into a non-linear step, while the linear steps propagate differentials that remain unchanged. At the end the last key contains many differentials, but columns C_0 to C_3 (see Figure 10), that are the non-linear vectors combinations used in countermeasure equations, also contain almost the same differential pattern. When combining the last key and the non-linear accumulators C_i the only differentials that remain are the two occurrences of δ_1 propagated through the linear part.

We can make some remarks about the propagation of the original differential through linear steps. As one can see, the following properties are verified by δ_1 on Fig. 10:

[4] We voluntarily choose a fault injection in middle rounds of the key schedule, even if it may not be exploitable by an attacker, in order to explicit more clearly the propagation and compensation phenomenons.

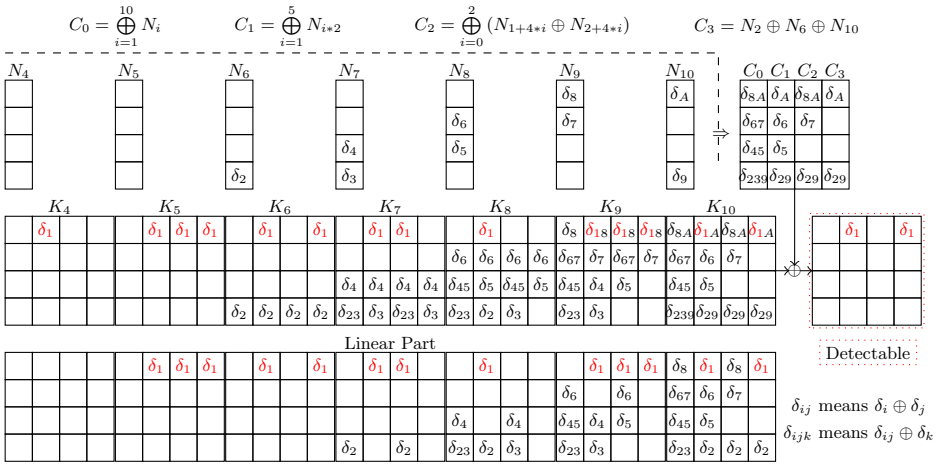

Fig. 10. Differential propagation from injection in $K_{4,4}$ illustrating that Equations (7) to (10) can detect only original differential δ_1 propagated by the linear part.

1. As the linear process only consists in XOR-ing columns together, the original differential always stays on the same row – e.g. δ_1 never leaves the first row.
2. For each matrix computation, the linear step always consists in accumulating columns from the first one to the last one of the matrix. As a consequence, the original differential never appears earlier in a matrix than in the previous ones – e.g. δ_1 injected into a second column of a matrix will never appear in any first column.
3. For a similar reason than for the previous property, the original differential that has been injected in some column of a matrix will always appear in the same column of subsequent matrices, and the appearance pattern is cyclic for the next columns. As an example, δ_1 injected into a second column will always be present in the second column of each subsequent matrix and have a cyclic appearance of period 4 keys, restarting in K_8 the same sequence.

Rows Parity Check Our simulations show that 64 out of the 160 key bytes positions[5] of K_1 to K_{10} produce differential propagations that are not detected by the rows parity check method proposed by Chen et al. We previously remarked that only the original differential can possibly propagate up to the parity check equation without cancellation. We also noticed that this propagation always stays on the same row. As a consequence, the countermeasure which consists in checking whether the XOR of all four bytes of a same row is null or not does not detect faults on positions that produce an even number of occurrences of the original differential in a same row of the last key. This is illustrated on Fig. 10 where δ_1 occurs twice on the same row in the detectable square on the right. Only the positions that produce an odd number of occurrences are safe.

Columns Parity Check Contrarily to the rows parity check, our simulations show that the columns parity check detects any single byte fault that occurs on a byte of any K_1 to K_{10}.

This observation matches with the theory since any single byte fault creates detectable differentials that are on the same row, and there is at least one occurrence of the original differential on every matrix following the one at which the fault occurred (c.f. the third remark above). As only the injection row may contain active differentials that are

[5] More details about the patterns of non-detection are provided in Appendix. A.

detectable, checking the parity by XOR-ing all bytes of a column always detects the fault at least in the column of injection.

The last part of this analysis, that was not evaluated by Chen et al., is about injections of faults into countermeasure registers We observed that faults injected into the linear part are always detected while some are not for the non-linear part. Both phenomenons can be explained by the construction of this countermeasure that checks a differential between the key obtained and the non-linear part. On the first hand, a differential into the linear part propagates into the keys and not directly into the non-linear part, so that it is always detected. On the other hand a fault injected into non-linear part may propagate, under certain circumstances, by the same way into keys and non-linear part leading to a non-detection at the end of the process. More precisely, this non-detection effect is produced when the fault is injected in a non linear vector before it is used into the key calculation. By this way, both non-linear vector and round key are infected leading to the same infection on both sides of parity equations (see Figure 11). If the non-linear vector has been used without error at least one time, the fault is detected because only a part of the round key is infected, leading to a different infection on the respective parts of parity equations (see Figure 12). As this undesirable effect is dependant to the fact that the error must infect all columns of the round key under injection, it can be simply avoided by re-computing each N_i vector during K_i calculation after that at least one column was already calculated. For example the Table 1 details the difference between the computation of K_1 with (right) and without (left) recalculation of N_1 vector after two column calculations. We could have chosen to recalculate N_1 before w_5 or w_7 instead of w_6.

Table 1. Successive operations of calculation of K_1 vectors with (right) and without (left) recalculation of N_1

N_i re-computation	
No	Yes
$N_1 = T(w_3)$	$N_1 = T(w_3)$
$L = w_0$	$L = w_0$
$w_4 = N_1 \oplus L$	$w_4 = N_1 \oplus L$
$L = L \oplus w_1$	$L = L \oplus w_1$
$w_5 = N_1 \oplus L$	$w_5 = N_1 \oplus L$
$L = L \oplus w_2$	$L = L \oplus w_2$
	$N_1 = T(w_3)$
$w_6 = N_1 \oplus L$	$w_6 = N_1 \oplus L$
$L = L \oplus w_3$	$L = L \oplus w_3$
$w_7 = N_1 \oplus L$	$w_7 = N_1 \oplus L$
$T(X) = RC(SW(RW(X)))$	

3.4 Analysis of AES-192 and AES-256 Countermeasure

The study realized on AES-128 in Sect. 3.3 is still valid concerning the role of non-linear and linear parts for AES-192 and AES-256.

Interestingly our simulations for these key sizes show that there are faults that are not detected by the columns parity check. Precisely, for AES-192 (resp. AES-256) any fault occurring on one of the 2 (resp. 4) last columns of a matrix is not detected. This is explained by the second remark above which states that differentials on the linear part never propagate in previous matrix columns. They can only propagate to the right, never

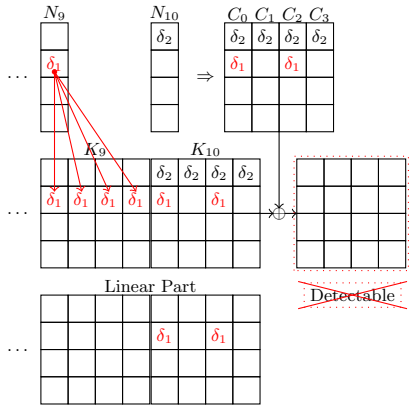

Fig. 11. Injection of differential δ_1 into non-linear vector N_9 before it was used into K_9 calculation leading to an undetectability of the fault by the parity checks.

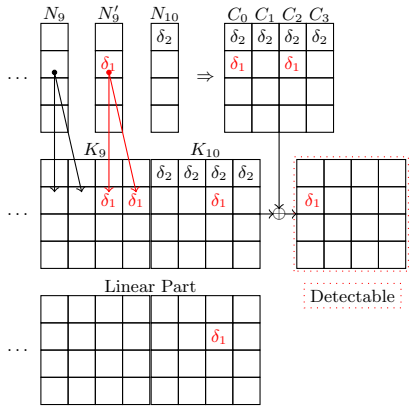

Fig. 12. Injection of differential δ_1 into non-linear vector N_9 after two proper K_9 columns calculation leading to a detectability of the fault by the parity checks.

to the left. Since the proposed countermeasure checks parity only on the first 4 columns of the last matrix which do contain the last round key (c.f. Fig. 6 and 8), faults occurring on one of the 2 (resp. 4) last columns of a matrix are not detected.

4 Corrections, Optimizations, Cost Evaluation and General Discussions

4.1 Correction for AES-192 and AES-256

The problem of columns parity check non-detection detailed in Sect. 3.4 can be corrected by checking the whole last matrix values instead of only the last key. As the last columns of last matrix are not calculated we propose instead to perform extra parity checks on the 2 (resp. 4) last columns of the penultimate matrix.

Here are given the 2 extra equations for AES-192. They have been obtained by using the same method than in Sect. 3.1:

$$w_{46} = w_3 \oplus w_4 \oplus \bigoplus_{i=4}^{7} N_i \qquad\qquad w_{47} = w_4 \oplus w_5 \oplus N_5 \oplus N_7$$

Here are the 4 extra equations for AES-256. Note that in those equations N'_i are involved instead of N_i:

$$w_{52} = w_4 \oplus \bigoplus_{i=1}^{6} N'_i \qquad\qquad w_{54} = w_4 \oplus w_6 \oplus N'_1 \oplus N'_2 \oplus N'_5 \oplus N'_6$$

$$w_{53} = w_5 \oplus \bigoplus_{i=1}^{3} N'_{2*i} \qquad\qquad w_{55} = w_5 \oplus w_7 \oplus N'_2 \oplus N'_6$$

Those equations were integrated into our simulator which validated the detection for any byte fault injected into K_1 to K_{12} (resp. K_{14}).

4.2 Optimization for the Byte Fault Model

We showed that the columns parity check detects every single byte fault[6] propagation, even on AES-192 and AES-256 if the proposed correction (extra parity equations) is applied. As an optimization we recommend to use only columns parity check and not rows parity check in order to do only 4 (resp. 6 or 8) tests instead of 8 (resp. 10 or 12) ones. In that configuration defender can also reduce the countermeasure cost by accumulating in memory the 8-bit XOR of the 4 bytes of the non-linear vectors instead of the 32-bit vector itself, dividing by 4 the memory requirement.

4.3 Adaptation to the Multi-Byte Fault Model

While the columns parity check is able to detect every single byte fault injection, it may not detect a multi-byte fault. It is notably the case if the XOR of differentials in the injection column is equal to 0. Although such fault effect can be difficult to produce, it is still possible to protect from it by using atomic parity check equations instead of columns parity check. We remind that the atomic equations are obtained by the decomposition of 4-byte columns equations into four 1-byte equations. i.e. four atomic equations are combined to create one column parity check equation. Checking atomic equations requires 16 (resp. 24 or 32) checks but avoids non-detection in cases of differential compensations.

[6] Except possibly faults on K_0. Notice however that in the case were the fault occurs while reading K from NVM to RAM the parity equations still detect the fault if the checksum computation involves bytes from original K (in NVM) instead of bytes of read K_0 (in RAM).

4.4 Cost Evaluation

We have implemented three AES-128 countermeasures in 8-bit software in order to compare their respective extra costs with respect to a non protected implementation. The results are summarized in Table 2 which also provides the extra costs between the countermeasures.

First we implement the classic doubling countermeasure that executes the whole[7] key schedule twice and checks the equality of the two obtained keys. We then implement two corrected versions of Chen et al.'s countermeasure, the one with atomic equations checking and the one – that we call *compressed* – that refers to the reduced memory version detailed in Section 4.2. We compiled with the tool avr-gcc for an 8-bit AVR microcontroller (ATmega328) without optimization option. The size evaluation is realized using avr-size and the time evaluation is done using oscilloscope measurement of executions.

The results show that those three versions are more or less close in term of extra costs. Depending on the device used and the implementation choices Chen et al.'s countermeasure – strengthened by our corrections – may be a good candidate as countermeasure.

Table 2. Results of software extra costs of three secured implementations with respect to two reference implementations

		Countermeasures (CM)		
		Doubling	Chen et al. Compressed	
Reference			Yes	No
Without CM	size	+86%	+71%	+96%
	time	+111%	+155%	+179%
Doubling	size	+0%	-8%	+5%
	time	+0%	+21%	+32%

4.5 Remark on Chen and Yen Proposition

Chen et al. are wrong saying that the rows parity check covers every positions of K_8, K_9 and K_{10}, but they are right when they claim that it can protect from their attack. Indeed, their attack requires to inject faults only in the last column of K_9 and the two last ones of K_8, which are positions covered by the rows parity check. If the defender is satisfied by the rows parity check coverage – which we consider risky –, then Chen et al.'s proposition reduces the computational cost of the countermeasure because Equation (1) involves only 2 non-linear vectors (for AES-128) to accumulate in memory instead of 24 ones for Equation (7) to (10) of the columns parity check.

4.6 Sensitivity to Double Faults

While we do not provide any proof, we have the intuition that the parity checking countermeasure may provide more resistance then the doubling countermeasure against double fault injections attacks. The idea behind this intuition is that the doubling countermeasure seems particularly vulnerable when the attacker can produce the same fault effect twice[8].

[7] We choose to implement a doubling of the whole key schedule in order to be able to detect the same area of injection than the corrected countermeasure of Chen et al.

[8] This is true for the encryption-encryption version, but may not but true for the encryption-decryption version.

On the contrary, it does not appear obvious why the parity check based countermeasure would be more vulnerable to such *same-repeated-faults* than to any other type of double fault. We let the study of the comparative vulnerability of both countermeasures as an open problem.

5 Conclusion

In this paper we deeply study the AES key schedule fault detection countermeasure proposed by Chen et al. that has been cited as countermeasure but never evaluated, implying a risk that may have been implemented with flaws. We provide a generalization of their approach and give a constructive method to obtain parity equations for all columns and for every key sizes, where the original paper only recommends rows parity check (which is flawed) and only for AES-128. We analyze the underlying reasons why and when the countermeasure works or do not work. In particular, we have evaluated the original proposal and showed that it is not as good as expected by the authors as it is far from covering every byte fault positions. We point out detection holes of the straightforward application of the countermeasure to AES-192 and AES-256, and provide extra parity equations to fix this problem. We also suggest optimizations that reduce the memory and computational costs of the countermeasure, and propose an adaptation to a more general fault model.

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.

This work is part of ICT COST ACTION IC1204 TRUDEVICE (Trustworthy Manufacturing and Utilization of Secure Devices).

References

1. Subidh Ali and Debdeep Mukhopadhyay. A Differential Fault Analysis on AES Key Schedule Using Single Fault. In Breveglieri et al. [4], pages 35–42.
2. Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In Burton S. Kaliski, Jr, editor, *Advances in Cryptology – CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer-Verlag, 1997.
3. Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer-Verlag, 1997.
4. Luca Breveglieri, Sylvain Guilley, Israel Koren, David Naccache, and Junko Takahashi, editors. *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC '11, Tokyo, Japan, September 29, 2011, Proceedings*. IEEE Computer Society Press, 2011.
5. Éric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES '04*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer-Verlag, 2004.
6. Chien-Ning Chen and Sung-Ming Yen. Differential Fault Analysis on AES Key Schedule and Some Countermeasures. In Reihaneh Safavi-Naini and Jennifer Seberry, editors, *Information Security and Privacy, 8th Australasian Conference, ACISP '03*, volume 2727 of *Lecture Notes in Computer Science*, pages 118–129. Springer-Verlag, 2003.
7. Christophe Clavier, Damien Marion, and Antoine Wurcker. Simple Power Analysis on AES Key Expansion Revisited. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES '14*, volume 8731 of *Lecture Notes in Computer Science*, pages 279–297. Springer, 2014.

8. Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential Fault Analysis on AES. In Jianying Zhou, Moti Yung, and Yongfei Han, editors, *Applied Cryptography and Network Security – ACNS '03*, volume 2846 of *Lecture Notes in Computer Science*, pages 293–306. Springer-Verlag, 2003.
9. Noémie Floissac and Yann L'Hyver. From AES-128 to AES-192 and AES-256, How to Adapt Differential Fault Analysis Attacks on Key Expansion. In Breveglieri et al. [4], pages 43–53.
10. Behnam Ghavami, Hossein Pedram, and Mehrdad Najibi. An EDA Tool for Implementation of Low Power and Secure Crypto-Chips. *Computers & Electrical Engineering*, 35(2):244–257, 2009.
11. Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual Information Analysis. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES '08*, volume 5154 of *Lecture Notes in Computer Science*, pages 426–442. Springer, 2008.
12. Christophe Giraud. DFA on AES. In Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa, editors, *AES 4 Conference*, volume 3373 of *Lecture Notes in Computer Science*, pages 27–41. Springer-Verlag, 2004.
13. Chong Hee Kim. Improved Differential Fault Analysis on AES Key Schedule. *IEEE Transactions on Information Forensics and Security*, 7:41–50, 2012.
14. Chong Hee Kim and Jean-Jacques Quisquater. New Differential Fault Analysis on AES Key Schedule: Two Faults Are Enough. In Gilles Grimaud and François-Xavier Standaert, editors, *Smart Card Research and Advanced Application – CARDIS '08*, volume 5189 of *Lecture Notes in Computer Science*, pages 48–60. Springer, 2008.
15. Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 1996.
16. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael J. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.
17. National Institute of Standards and Technology. Advanced Encryption Standard (AES). Federal Information Processing Standard #197, 2001.
18. David Peacham and Byron Thomas. DFA against AES Key Expansion. CHES '06 Rump Session, 2006.
19. Gilles Piret and Jean-Jacques Quisquater. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES '03*, volume 2779 of *Lecture Notes in Computer Science*, pages 77–88. Springer-Verlag, 2003.
20. Junko Takahashi, Toshinori Fukunaga, and Kimihiro Yamakoshi. DFA Mechanism on the AES Key Schedule. In Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography – FDTC '07*, pages 62–74. IEEE Computer Society Press, 2007.
21. Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. In Claudio Agostino Ardagna and Jianying Zhou, editors, *Workshop on Information Security Theory and Practice – WISTP '11*, volume 6633 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 2011.

A Remarkable Patterns in Propagation

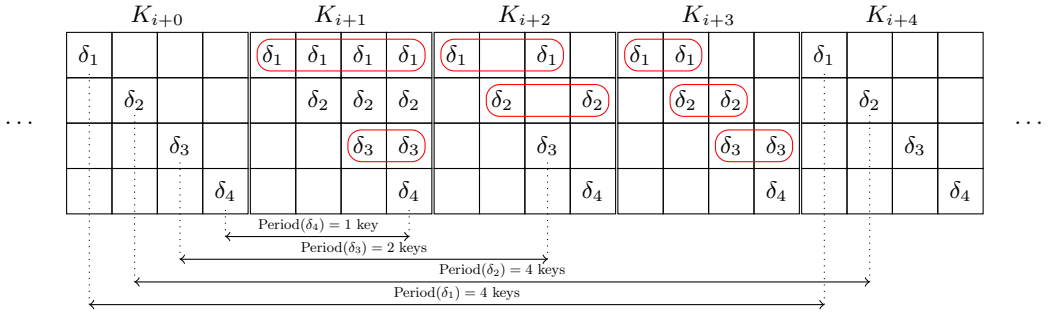

Fig. 13. The 4 differential propagation patterns depending on the injection column.

We describe how patterns can be inferred from the propagation of an original differential through AES-128 key schedule. Fig. 13 shows an example of propagation of four differentials, one per column (the differential propagation pattern does not depend on the row).

Remark that the fault propagation patterns have a period that depends on the injection column. First and second columns show a period of 4 round keys while the third column shows a period of 2 keys, and the fourth column a period of 1 key.

We have surrounded in red the row segments that contain an even number of differentials. They are not detected by the rows parity check if the last key lies at these positions. In order to derive the 16 columns (i.e. 64 bytes) that are concerned by non-detection of faults, we write, for each injection column, equations that relate the round r of the fault injection, the offset of the non-detection segment, and the period:

Fault injection in a 1^{st} column:

$$\begin{aligned}(r+1) &\equiv 10 \pmod 4 \;\Rightarrow\; r \in \{1,5,9\}\\ (r+2) &\equiv 10 \pmod 4 \;\Rightarrow\; r \in \{4,8\}\\ (r+3) &\equiv 10 \pmod 4 \;\Rightarrow\; r \in \{3,7\}\end{aligned}$$

Fault injection in a 2^{nd} column:

$$\begin{aligned}(r+2) &\equiv 10 \pmod 4 \;\Rightarrow\; r \in \{4,8\}\\ (r+3) &\equiv 10 \pmod 4 \;\Rightarrow\; r \in \{3,7\}\end{aligned}$$

Fault injection in a 3^{rd} column:

$$(r+1) \equiv 10 \pmod 2 \;\Rightarrow\; r \in \{1,3,5,7,9\}$$

Fault injection in a 4^{th} column:

no non-detection