



**HAL**  
open science

## Generalized Code-Abiding Countermeasure

Pierre-Antoine Tissot, Lilian Bossuet, Vincent Grosso

► **To cite this version:**

Pierre-Antoine Tissot, Lilian Bossuet, Vincent Grosso. Generalized Code-Abiding Countermeasure: Applying Lightweight Fault Resistant Countermeasure to Word-Oriented Block Cipher. *Electronics*, 2023, 12 (4), pp.976. 10.3390/electronics12040976 . hal-04225001

**HAL Id: hal-04225001**

**<https://hal.science/hal-04225001>**

Submitted on 2 Oct 2023

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Generalized Code-Abiding Countermeasure

## Applying Lightweight Fault Resistant Countermeasure to Word-Oriented Block Cipher

Pierre-Antoine TISSOT, Lilian BOSSUET, Vincent GROSSO

No institute given.

**Abstract.** An adversary can inject errors during the encryption process performed by a software or hardware implementation by using fault injection attacks. The widely used countermeasures to these attacks are based on spatial, temporal or information redundancy. This type of solution is very efficient regarding to the detection of the fault, but can be very expensive in terms of implementation cost. Try to propose a secure-efficient countermeasure for a lightweight cipher is a hard challenge. This paper considers information redundancy based on parity bit code, with code-abiding transformations of the cipher's operations. This error detection code, with the code-abiding notion added, is very efficient against single fault injection and has a small overcost. The solution is tested on the LED lightweight cipher to measure its overhead. A bitslice version of the cipher is used with the parity bit code applied to be robust against all the single-word fault injections with 16 additional parity bits on the original 64-bit state.

**Keywords:** Fault Attack, Error Detection, Code-Abiding, Overcost

## 1 Introduction

Cryptographic implementations are prone to physical attacks. Physical attacks take advantage of physical properties of a device while running cryptographic algorithm to break the security. Most popular physical attacks are fault attacks [BS97] (taking advantage of the circuit's tend to perturbations) and side-channel attacks [KJJ99] (taking advantage of the circuit's leakage). This work focuses on fault attacks. The principle of fault attacks is to use means such as laser injection or clock glitching in order to inject faults during an encryption and extract information by analyzing the circuit's behavior after the injection.

To counter fault attacks, various countermeasures have been developed, using mainly redundancy [BBB<sup>+</sup>21, BHL21, KMD<sup>+</sup>20, LJK21, PYGS16]. Redundancy allows to create multiple information sources. These multiple sources are compared at the end of the computation to detect fault injection. Redundancy can be applied at three different levels: temporal, spacial and informational. Temporal redundancy is based on the multiple encryptions of a plaintext by the same physical cipher (same circuit) and the comparison between the resulting ciphertexts. Spacial redundancy is based on the multiple encryptions of a plaintext by different physical ciphers (different circuits). Moreover, an additional information can be added to the plaintext to create an information redundancy. In all the cases, the potential leakages of the cipher are more numerous and the side-channel attacks (SCA) are thus more efficient [REB<sup>+</sup>08]. Therefore, when designing a countermeasure against fault attacks, the designer should then take into account vulnerabilities that his countermeasure can bring for side channel adversary. One rule of the thumb in that case is to make the overcost of the countermeasure as small as possible, especially when the countermeasure is implemented on a lightweight cipher.

## 1.1 Related Work

Simon et al. [SBD<sup>+</sup>20] presented a solution of error detection that hardly increase the SCA vulnerability. However, this solution is restrictive for any designer that would like to apply code-abiding to an existing cipher, and more particularly work-oriented block ciphers. Our goal is then to generalize the code abiding method to any existing or new word-oriented block cipher.

Bertoni et al. [BBK<sup>+</sup>02] used the parity bit code to detect fault injected on AES. Bertoni et al. describe a modification of the algorithm with the addition of the parity bit matrix, when our objective is to use a bitslice version of an algorithm to add a bitwise countermeasure to a word-oriented cipher. Then, the S-Box used in [BBK<sup>+</sup>02] with half of its entries set to 00..001 is efficient in a software way, but in a hardware implementation, a big number of logical gates would be used. Moreover, the protected cipher would be robust against 1-bit fault injection, when our solution prevent any 1-word fault injection.

Lac et al. [LCFS17] used an Internal Redundancy Countermeasure: every data block is duplicated  $k$  times and surrounded by  $n$  reference blocks. The  $k$  copies allow to detect up to  $k$  fault injections by comparing the results. Moreover, the reference blocks would detect a fault even if it affects all the copies of the data block. Indeed, reference blocks are known pairs of plaintext/ciphertext, and a check is done of the cipher reference blocks to detect an injected fault. The blocks are randomly distributed in the register. Thus, for each data block, we have  $k + n$  blocks of overhead. In our paper, the solution adds 1 bit for each data block. The overhead is then much lighter in our solution.

## 1.2 Our contributions

Our first contribution is an exhibition of a fault injection realized in precise conditions during a computation of a Friet operation [SBD<sup>+</sup>20] that results into an undetected error. The conditions are presented, with two countermeasures that can be applied to allow the detection of the error.

The second contribution is the application of the code-abiding method to an existing cipher with an example on the lightweight LED cipher [GPPR11]. The countermeasure is designed to get the smallest overcost possible. The secured solution presented in our work should be 25% more expensive than the original LED implementation, as only one parity bit is added for each nibble. Our work thus focuses on the cost optimization of the countermeasure, in terms of number of gates, memory space needed and power consumption.

This work should allow implementations to be robust against single injection fault with an optimization of the overcost brought by the countermeasure.

## 2 Background

In this section we briefly introduce notions on coding theory that are useful for the countermeasure presented. We also recall the operation of LED block cipher [GPPR11] on which we apply our countermeasure as a proof of efficiency of our method.

### 2.1 Error Detection

The solutions presented in this paper use code abiding concept introduced in [SBD<sup>+</sup>20]. This solution is based on computation over data encoded with error detection. In the following, we give the goal and the principle of error detection, which is a set of techniques that makes possible to detect errors during the transmission of information.

**Definition 1** (Error detection code). Let  $E$  a set and  $\mathcal{C} \subset E$ . We denote  $\bar{\mathcal{C}} = E \setminus \{\mathcal{C}\}$ .  $\mathcal{C}$  is an error detection code if and only if:

- $\forall x \in \mathcal{C}, \forall y \in \bar{\mathcal{C}}, x + y \in \bar{\mathcal{C}}$
- $\forall x \in \mathcal{C}, \forall z \in \mathcal{S} \subsetneq \mathcal{C}, x + z \in \mathcal{C}$

In this case,  $+$  is the addition operator, according to the set  $E$ .

An error detection code allows to divide a set into two different subsets with a minimal Hamming distance between a word of a subset and a word from the other.

**Definition 2** (Parity bit). Let  $x$  a  $n$ -bit word. We denote  $x_i$  the  $i$ -th bit of  $x$ , then we have  $x = x_{n-1} || x_{n-2} || \dots || x_1 || x_0$  where  $||$  is the concatenation operator. The parity bit  $x_p$  is the sum of all the  $x_i$  (using XOR operator):  $x_p = x_{n-1} \oplus x_{n-2} \oplus \dots \oplus x_1 \oplus x_0$ . We use the even parity in our case, so the XOR of all the bits (including the parity bit) is 0. Its purpose is to detect an odd number of fault in the output.

**Example 1** (Parity bit). Let  $x = 011010$ . The parity bit  $x_p = 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 = 1$ .

The parity bit method is the error detection scheme that is used during this work. The two subsets are composed by the words with an even parity for the first one and the words with an odd parity for the second one.

**Definition 3** (Check function). The `CheckFunction` applied to a word verifies its parity characteristic. The function returns a Boolean with `TRUE` when an even parity is verified and `FALSE` when odd parity.

**Example 2.** Check function

- `CheckFunction(011011) = TRUE`
- `CheckFunction(110010) = FALSE`

## 2.2 Code abiding

We now want to implement the error detection scheme into an encryption algorithm. Then, we need to use functions that keep the parity characteristic of the words. With this intention, we use the code abiding notion. Code abiding was introduced in [SBD<sup>+</sup>20], in this work the idea was to build permutation over the space  $E$ . In order to detect fault, the permutation built must respect separation of the space. In other words, the permutation over  $E$  can be seen as two permutations, one over  $\mathcal{C}$ , the other over  $\bar{\mathcal{C}}$ . The separation between the two spaces should allow detection of every single fault injection.

**Definition 4** (Code abiding function).  $f$  is a  $\mathcal{C}$  code abiding function if and only if:

- $\forall x \in \mathcal{C}, f(x) \in \mathcal{C}$
- $\forall x \notin \mathcal{C}, f(x) \notin \mathcal{C}$

The algorithm has to be composed by code abiding functions to keep the parity property of the words and propagate the error injected until the detection of the fault.

## 2.3 LED Cipher

The LED Cipher, presented in [GPPR11], is a lightweight block cipher. Its purpose is to offer a very small silicon footprint in comparison with other block ciphers, and to be secure against related-key attacks, by using AES-like security proofs.

This cipher is a 64-bit block cipher using mostly 64-bit keys and 128-bit keys. But any length between 64 bits and 128 bits can be used, if the length is divisible by 4. In this sense, 80-bit keys are also often used. In our work, we focus on the 64-bit key length. However, the results presented are valid for any key length and are not limited to the LED cipher.

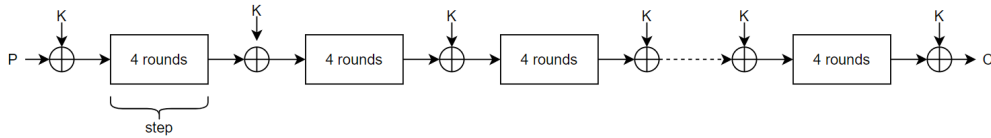
A 64-bit state  $St$  is conceptually divided into sixteen 4-bit nibbles ( $St = st_0 || st_1 || \dots || st_{15}$ ) and arranged in a square array as described in Matrix  $state$ .

$$state = \begin{pmatrix} st_0 & st_1 & st_2 & st_3 \\ st_4 & st_5 & st_6 & st_7 \\ st_8 & st_9 & st_{10} & st_{11} \\ st_{12} & st_{13} & st_{14} & st_{15} \end{pmatrix}$$

Using the same process, the key  $K$  is divided into subkeys  $k_i$  (Matrix  $K$ ).

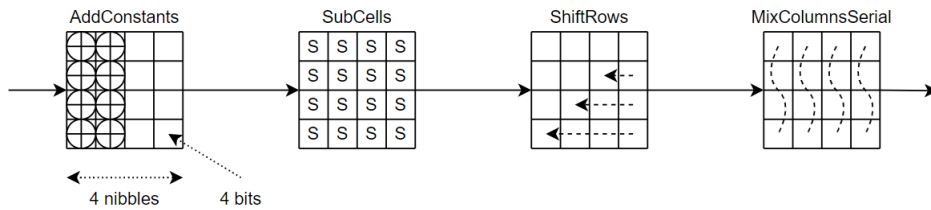
$$K = \begin{pmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \\ k_{12} & k_{13} & k_{14} & k_{15} \end{pmatrix}$$

The cipher process is the combination of two operations: **AddRoundKey** and **step** (see Figure 1). The **step** operation is computed  $s$  times while the **AddRoundKey**  $s + 1$  times. This value depends on the key length:  $s = 8$  for a 64-bit key and  $s = 12$  for a 128-bit key.



**Figure 1:** Representation of the LED encryption.

The **step** operation is composed by four rounds themselves composed by four operations, **AddConstants**, **SubCells**, **ShiftRows** and **MixColumnsSerial** (See Figure 2), while the **AddRoundKey** operation is the combination of the state and the subkeys using XOR.



**Figure 2:** A round of LED composed by the functions **AddConstants**, **SubCells**, **ShiftRows** and **MixColumnsSerial**.

After the  $s$  **step** and  $s + 1$  **AddRoundKey**, the state becomes the ciphertext.

Another approach of the cipher is its bitslice version [BLL15] and this approach brings some important properties for this work. Let suppose that the machine used has 64-bit length registers. Then the 64-bit state is stored in a single register. The bitslice transformation of the cipher stores the state in 64 registers containing each 1 bit of data. This approach allows to have a bit-oriented cipher rather than a word-oriented. This is very important for the implementation of the parity scheme.

Another advantage of the bitslice version is the parallel encryptions. In the same conditions than the previous point, instead of using 64 registers containing only 1 bit of useful data, we can encrypt  $n$  plaintexts in parallel and then store 64  $n$ -bit useful data. As the bitslice version is bitwise, the cipher cannot interfere between the different states. It is this method that induces detection of any 1-word fault injection. Indeed, as every machine word is seen as the concatenation of a single bit of  $n$  states and as any 1-bit injection in a state would be detected, then up to 1-word fault injection could be detected here.

### 3 Error Compensation Issue

In this section we exhibit a potential fault attack against Friet [SBD<sup>+</sup>20]. Indeed, in particular scenarios, we show that a fault injected can create an several errors that can be compensated during the parity check so the error is not detected. The scenario has small probability of success depending on the attack model and some requirements about implementation characteristics. We then present a countermeasure to prevent this kind of attack in a strong model where the adversary can inject the fault of his choice at the position and time he chooses.

#### 3.1 Issue example

We bring out the vulnerabilities with an example, we next generalize it. We assume that the attacker can add a fault to one value during the computation.

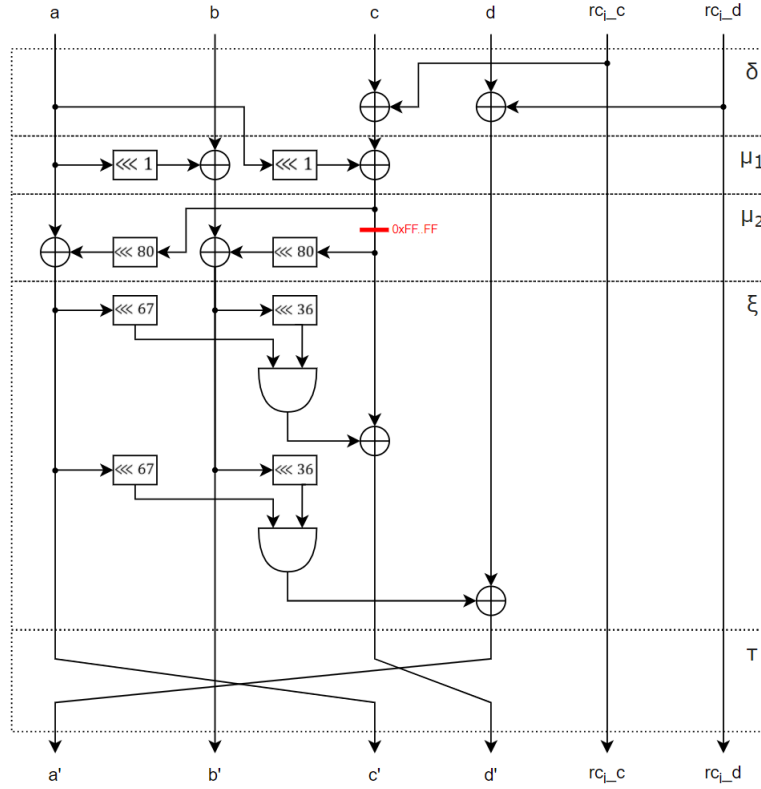
Let  $\epsilon = 2^{128} - 1$  as Friet manipulates 128-bit data. For the sake of simplicity, we call  $a, b, c$  and  $d$  the inputs of the  $\mu_2$  operation. For the same reason,  $a', b', c'$  and  $d'$  are the outputs of the operation. During these operations, the parity equation followed is  $d = a \oplus b \oplus c$ . This parity equation is checked to ensure that no fault is injected. We inject the additive fault  $\epsilon$  to the word  $c$  during the  $\mu_2$  operation, after that the rotate word  $c$  is added to  $a$  and to  $b$  before the addition. The fault injection is illustrated in Figure 3, with the red line showing the modification.

When such a fault is injected, the output of two branches are modified: the second  $b$  and the third  $c$ . We denote  $b'$  and  $c'$  the second and third words of the output of the faulty  $\mu_2$  operation. Then we have two equations:

$$\begin{aligned} b' &= b \oplus ((c \oplus 0\text{xFF..FF}) \lll 80) = b \oplus (c \lll 80) \oplus (0\text{xFF..FF} \lll 80) = b \oplus (c \lll 80) \oplus 0\text{xFF..FF} \\ c' &= c \oplus 0\text{xFF..FF}. \end{aligned}$$

At the output of the faulty  $\mu_2$ , we have  $(a', b', c', d') = (a, b \oplus (c \lll 80) \oplus 0\text{xFF..FF}, c \oplus 0\text{xFF..FF}, d)$ . Then the check subroutine does not detect the injected fault since the fault on the two branches cancel out when applying a XOR operation on the values.

In the previous example, we saw that the fault  $0\text{xFF..FF}$  is not detected, because it remains unchanged with the shift by 80 bits (that is the shift of  $\mu_2$ ). However, this is not the only fault that is not detected with this shift. Indeed, all the fault that remain unchanged with the 80-bit shift have this property. With the Algorithm 1 we can identify all the valid fault that are undetected. Indeed, we begin with the word  $i = 1$ , we shift this word by 80 bits and we test when the word come back to the initial value 1. The cycle length found is 8 and the number of cycle is then  $\frac{\text{word\_length}}{\text{cycle\_length}} = \frac{128}{8} = 16$ . That means



**Figure 3:** Round of FRIET-P

that every fault composed by 8 same 16-bit concatenated words is not detected. Thus we have  $2^{16} - 1$  undetected faults (the value  $0x00..00$  is not a fault) over  $2^{128}$  different faults. In terms of probability, we have a probability around  $2^{-112}$  to have an undetected fault. As this probability is very tiny then with some random faults it difficult to identify such a weakness. Moreover, the undetected fault is a 128-bit injection and when the registers are strictly smaller than 128-bit long, then the necessary fault would affect two registers so two faults would be needed. This constraint places the error outside of the study.

---

**Algorithm 1** Find the length of a cycle

---

**Require:** Size of the shift (here 80)

**Ensure:** Length of a cycle (how many shifts to recover the former value)

$i \leftarrow 1$

$t_{cycle} \leftarrow 1$

**while**  $(i \lll 80) \% 128 \neq 1$  **do**

$t_{cycle} \leftarrow t_{cycle} + 1$

$i \leftarrow (i \lll 80) \% 128$

**return**  $t_{cycle}$

---

The same analysis can be done for  $\mu_1$ , it is easy to see that the only undetected fault, is the all 1 fault. For the  $\chi$  the bitwise and between two branches make the fault non-detection probabilistic in function of the data in the second branch.

### 3.2 Countermeasures

We assume that the registers are wide enough to ensure that the undetected faults are still in the limits of the study. An obvious solution to this issue is to increase the cycle length to limit the number of undetected faults. With a shift of 1 bit, the cycle length is maximum with a value of 128. Indeed, only the words composed by 128 same 1-bit words are undetected. However, the fault `0xFF..FF` remains undetected (`0x00..00` is still not a fault) and we have to modify the former operation to implement our solution.

Another solution must be found to detect all the faults without changing the cryptographic primitives of the cipher. We copy every variable used more than once and check if the copies are equal. A Boolean `flag` is used to express the error detection (`flag` gets the value 0 when an error is detected). With this principle, any fault injected during an operation only affects one copy and is detected before using the copies. The following Algorithm 3 presents the copies and the checks on the operation  $\mu_2$  of the FRIET-P round and shows the overcost of this solution in comparison with the classical FRIET-P presented in Algorithm 2.

---

**Algorithm 2** Classical  $\mu_2$  operation of the FRIET-P round

---

**Require:** Four 128-bit words  $a, b, c$  and  $d$

**Ensure:** Four 128-bit words  $a', b', c'$  and  $d'$  computed by the original  $\mu_2$  operation

$a' \leftarrow a \oplus (c \lll 80)$

$b' \leftarrow b \oplus (c \lll 80)$

$c' \leftarrow c$

$d' \leftarrow d$

**return**( $a', b', c', d'$ )

---



---

**Algorithm 3** Protected  $\mu_2$  operation of the FRIET-P round

---

**Require:** Four 128-bit words  $a, b, c$  and  $d$

**Ensure:** Four 128-bit words  $a', b', c'$  and  $d'$  computed by the protected  $\mu_2$  operation

$c_0 \leftarrow c$

$c_1 \leftarrow c$

$c_2 \leftarrow c$

`flag`  $\leftarrow$  `flag` & ( $c_0 == c_1$ ) & ( $c_0 == c_2$ )

$a' \leftarrow a \oplus (c_0 \lll 80)$

$b' \leftarrow b \oplus (c_1 \lll 80)$

$c' \leftarrow c_2$

$d' \leftarrow d$

**return**( $a', b', c', d'$ )

---

The overcost of the countermeasure lies on the three copies of the value  $c$  and the comparison of these three copies. This solution is used in the rest of the paper to avoid undetected fault injection.

## 4 Code Abiding on Existing Cipher

In this section, we present a generic method to apply code-abiding countermeasure to word oriented block ciphers and illustrate this technique on LED cipher [GPPR11]. Word-oriented ciphers are often implemented with tables (S-boxes for the substitution layer and multiplicative tables for the diffusion layer) and with XOR operation on words in the same column. We then need to consider error detection code at word, column and state level.



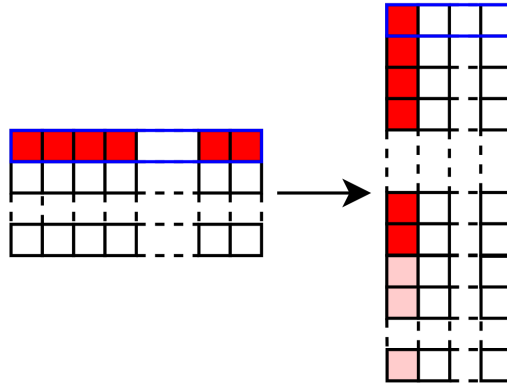
The basic code is defined at word level and is simply extended to the column and state level. Indeed, the columns and the state are only a concatenation of the words. Thus if we have a code  $\mathcal{C}$  of parameters  $[n, k, d]$  at word level, the concatenation of  $l$  word is a code  $\mathcal{C}'$  of parameters  $[ln, lk, d]$  at column level.

The principle of code abiding protection is to apply permutation on different codes. Thus we have two cases if we apply always permutation on full state then either we are in the code and stay in the code or we are not in the code and stay outside the code. Since we target only 1 fault injection we can at most change once of set and due to our construction, any single fault injection forces to change from a word of the code to a word outside the code. The last property is obtain thanks to bitslice representation and check of non-modification when we use the same variable in different place. (Note that we can hope for security, and fault detection, for multiple random fault with high probability, thanks to parallelism we use). We next present in more details the adaptation made for each operation.

#### 4.1 State modification

Let  $S$  be the 64-bit state of the unprotected LED cipher. In order to detect fault, we need to add a redundant part. In our case we select the 5-bit parity check code. Thus we need to add a parity bit for each 4-bit nibble of the state. Indeed, if we denote  $S_i$  the  $i^{th}$  bit of  $S$ , we have:  $S_{64+i} = S_{4 \times i} \oplus S_{4 \times i + 1} \oplus S_{4 \times i + 2} \oplus S_{4 \times i + 3}$ , where  $S_{4 \times i}, S_{4 \times i + 1}, S_{4 \times i + 2}, S_{4 \times i + 3}$  are the bits of the nibble  $i$ . Eventually, we have a 80-bit state composed of 64 data bits and 16 parity bits.

In the Section 2, we presented the bitslice version of LED. This is the version that is used in this work, and we thus have to add the parity property by adding 16 registers. The Figure 4 illustrates this step with in blue a register of the machine, in red the data bits of a state and in pink the parity bits of the red state. The code abiding notion is more bit-oriented than word-oriented and then the bitslice approach allows to use the code abiding notion on a classical word-oriented cipher.



**Figure 4:** State after Bitslice and Parity transformations.

These transformation functions are summarized in the following Algorithms 4 and 5.

#### 4.2 Key and Constant

We assume that the key and the constant are stored in an encoded manner.

We copy the key and the constant at the beginning of the computation and use the copy for the all computation at the end we check that the copy used stayed unchanged. Thus any change of the key during the encryption is detected. Since the attacker can only

---

**Algorithm 4** State  $S$  transformed into its bitslice version.

---

**Require:** State  $S$   
**Ensure:** State  $S$  in its bitslice version  
  **for**  $j$  in range(64) **do**  
     $Sj_i \leftarrow Si_j$   
  **return**  $S$

---



---

**Algorithm 5** Parity bits added to the bitslice state  $S$ .

---

**Require:** Bitslice state  $S$   
**Ensure:** State  $S$  with parity bits  
  **for**  $i$  in range(16) **do**  
     $S_{64+i} \leftarrow S_{4 \times i} \oplus S_{4 \times i + 1} \oplus S_{4 \times i + 2} \oplus S_{4 \times i + 3}$   
  **return**  $S$

---

inject one fault, modification of the key is detected. An adversary that modifies the key may inject fault at each key addition. However by using copy and checking at the end and thanks to the absence of key schedule in LED-64, the attacker cannot use this method for multiple fault injection.

The only method should be to modify the stored key. However, LED is known for resistance against related key attacks and thus no exploitable information can be obtained by the attacker.

If a fault is injected on the key, the XOR operation with the state propagates the error on the state until the parity check of the state.

### 4.3 AddConstant

We calculate the constant presented in Section 2. This constant is a 64-bit value that we transform into 80  $n$ -bits values (bitslice + parity transformations) that are computed to the state using XOR operation. If  $n$  encryptions are performed in parallel, the constant must fit the  $n$ -bit length of the registers, then the 80 bits have to be duplicated  $n$  times. This is illustrated in Algorithm 6 (0xFF..FF is composed by  $\frac{n}{4}$  F).

---

**Algorithm 6** Constant  $c$  bitsliced and duplicated.

---

**Require:** 64-bit constant  $c$   
**Ensure:** 80  $n$ -bit (with duplication of the bit) constants  $c_i$  with parity and bitslice transformations.  
  **for**  $i$  in range(64) **do**  
     $c_i \leftarrow ((c \gg (63 - i)) \& 1) \times 0xFF..FF$   
  **return**  $c_i$

---

If a fault is injected on the constant, the error is propagated on the state until the check parity function.

### 4.4 ShiftRows

This function is the same operation as the former **ShiftRows** operation. The parity bits are shifted among the nibble from which they have been computed. The bit  $S_{64+i}$  is shifted  $\frac{i}{4}$  bits to the left. This is illustrated in Figure 5. During this operation, a fault can be injected on the state and stays on it until its detection. Moreover, as the state is only shifted, its value remains the same, then a fault injected before the operation is propagated on the output.

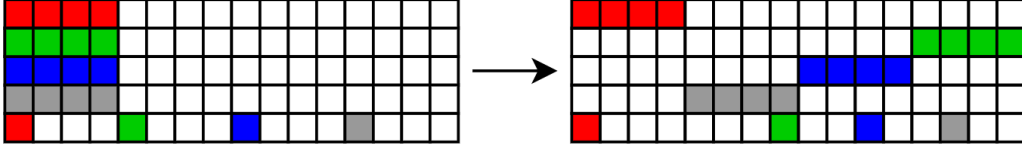


Figure 5: ShiftRows on the bitslice parity state

#### 4.5 SubCells

The substitute operation brings confusion and non linearity to encryption. It is then a critical function of the block cipher construction. The extension from the code  $\mathcal{C}$  to the code  $\mathcal{C}'$  requires to represent the 4-bit S-box by a 5-bit S-box, and this projection brings a choice of the 5-bit S-box.

We present in section 5 a way to construct the protected S-box. Here we present the results on the PRESENT S-box represented by the Table 2. But we use an alternative form of the S-box composed only by logical gates, the Algebraic Normal Form. This form gives 5 equations where  $x_i$  is the  $i^{th}$  bit of the input and  $y_i$  the  $i^{th}$  bit of the output ( $x_0..x_3$  are the data bits and  $x_4$  is the parity bit).

$$\begin{aligned}
 y_0 &= x_3x_2x_1 \oplus x_3x_2x_0 \oplus x_3x_1x_0 \oplus x_3 \oplus x_2x_1 \oplus x_2 \oplus x_0 \oplus 1 \\
 y_1 &= x_3x_2x_0 \oplus x_3x_2 \oplus x_3x_1x_0 \oplus x_3x_0 \oplus x_2x_0 \oplus x_1 \oplus x_0 \oplus 1 \\
 y_2 &= x_3x_2x_1 \oplus x_3x_2x_0 \oplus x_3x_1x_0 \oplus x_2x_0 \oplus x_2 \oplus x_1x_0 \oplus x_0 \\
 y_3 &= x_4x_3x_2x_1x_0 \oplus x_4x_3x_1x_0 \oplus x_3x_2x_1x_0 \oplus x_3x_1x_0 \oplus x_3 \oplus x_2x_1 \oplus x_1 \oplus x_0 \\
 y_4 &= x_4x_3x_2x_1x_0 \oplus x_4x_3x_1x_0 \oplus x_4 \oplus x_3x_2x_1x_0 \oplus x_3x_2x_0 \oplus x_3x_2 \oplus x_3x_0 \oplus x_3 \oplus x_2 \oplus x_1x_0 \oplus x_1 \oplus x_0
 \end{aligned}$$

This function is presented in Algorithm 7. We can denote the copies of the values used more than once to avoid error compensation presented in Section 3, and as the bit  $x_4$  is used only in the last two equations, this bit is copied only twice. Indeed, the output  $S_{4 \times i+m}$  only lies on the values  $x_m..$  then a fault injected on a copy only affect one output, and the parity characteristic allows the error detection.

---

**Algorithm 7** SubCells function.

---

**Require:** State  $S$ ,  $i$ , flag

**Ensure:** State  $S$  after the SubCells operation and the flag detection flag

**for**  $j$  in range(5) **do**

$$x_{j0} \leftarrow S_{4 \times i+0}$$

$$x_{j1} \leftarrow S_{4 \times i+1}$$

$$x_{j2} \leftarrow S_{4 \times i+2}$$

$$x_{j3} \leftarrow S_{4 \times i+3}$$

**if**  $j > 2$  **then**

$$x_{j4} \leftarrow S_{4+i}$$

**for**  $j$  in range(5) **do**

$$\mathbf{flag} \leftarrow \mathbf{flag} \ \& \ (x_{0j} == x_{1j}) \ \& \ (x_{0j} == x_{2j}) \ \& \ (x_{0j} == x_{3j}) \ \& \ (x_{0j} == x_{4j})$$

$$S_{4 \times i+0} \leftarrow x_{03}x_{02}x_{01} \oplus x_{03}x_{02}x_{00} \oplus x_{03}x_{01}x_{00} \oplus x_{03} \oplus x_{02}x_{01} \oplus x_{02} \oplus x_{00} \oplus 1$$

$$S_{4 \times i+1} \leftarrow x_{13}x_{12}x_{10} \oplus x_{13}x_{12} \oplus x_{13}x_{11}x_{10} \oplus x_{13}x_{10} \oplus x_{12}x_{10} \oplus x_{11} \oplus x_{10} \oplus 1$$

$$S_{4 \times i+2} \leftarrow x_{23}x_{22}x_{21} \oplus x_{23}x_{22}x_{20} \oplus x_{23}x_{21}x_{20} \oplus x_{22}x_{20} \oplus x_{22} \oplus x_{21}x_{20} \oplus x_{20}$$

$$S_{\times i+3} \leftarrow x_{34}x_{33}x_{32}x_{31}x_{30} \oplus x_{34}x_{33}x_{31}x_{30} \oplus x_{33}x_{32}x_{31}x_{30} \oplus x_{33}x_{31}x_{30} \oplus x_{33} \oplus x_{32}x_{31} \oplus x_{31} \oplus x_{30}$$

$$S_{64+i} \leftarrow x_{44}x_{43}x_{42}x_{41}x_{40} \oplus x_{44}x_{43}x_{41}x_{40} \oplus x_{44} \oplus x_{43}x_{42}x_{41}x_{40} \oplus x_{43}x_{42}x_{40} \oplus x_{43}x_{42} \oplus x_{43}x_{40} \oplus x_{43} \oplus x_{42} \oplus x_{41}x_{40} \oplus x_{41} \oplus x_{40}$$


---

If a fault is injected before or during the function, the separation of the codes in the

S-box representation keeps the word out of the code  $\mathcal{C}$  and the fault is propagated into the space.

## 4.6 MixColumnsSerial

This operation is composed by the four post-multiplications with the matrix  $A$  (see Section 2). The state is decomposed into 4 columns of 4 5-bit nibbles each. These nibbles are the 4-bit data and the parity bit associated. In our operation, only a multiplication by 2 is used (a multiplication by 4 is just two multiplications by 2). The Algorithm 8 show the multiplication by 2 operation. This operation is a shift of the bits and a XOR with the LSB of the data word on the second bit of the nibble. The computation on the parity bit is thus only a XOR with this LSB.

The Algorithm 9 presents the state divided into columns and the multiplication with the matrix  $A$ . Same as in the `SubCells` function, we create a copy of each element used more than once to avoid a future error compensation.

---

**Algorithm 8** Multiplication by 2.

---

**Require:** Nibble  $nibble$

**Ensure:** Nibble  $nibble \times 2$

**function** `mc2`( $nibble$ )

$nib_{30} \leftarrow nibble[3]$

$nib_{31} \leftarrow nibble[3]$

$nib_{32} \leftarrow nibble[3]$

$flag \leftarrow flag \ \& \ (nib_{30} == nib_{31}) \ \& \ (nib_{30} == nib_{32})$

$nibble[0], nibble[1], nibble[2], nibble[3], nibble[4] \leftarrow nib_{30}, nibble[0] \oplus nib_{31}, nibble[1],$   
 $nibble[2], nibble[4] \oplus nib_{32}$

---

With the same observations than the previous operations, if a fault is injected before or during the `MixColumnsSerial` operation, this error is propagated through the operation on the state.

## 5 5-bit Representation

In the protected version of LED, the `SubCells` function uses a 5-bit representation of the PRESENT S-box. This section presents how to create a 5-bit representation from a 4-bit permutation and which representation is the best in terms of cost optimization.

### 5.1 Score function

In the last section, the `SubCells` function requires a representation on 5 bits of the 4-bit PRESENT. The former 4-bit S-box must remain the same with the parity bit added at the end of the words. Indeed, the 5-bit representation is already half filled with the words with an even parity (see Table 1). Then we have  $16^{16}$  candidates to represent a 4-bit S-box. We must find a way to compare one candidate from another.

First, we only consider the S-boxes that correspond to permutations (each output has one and only one related input). Indeed, the parity code used is the 5-bit parity code  $\mathcal{C} = [5, 4, 2]$  but as we want to consider this code at the state level, the resulting code  $\mathcal{C}' = [80, 64, 2]$  is selected.  $\mathcal{C}'$  is only a concatenation of 16 codes  $\mathcal{C}$ . This concatenation brings the constrains of the permutations on the S-boxes.

---

**Algorithm 9** MixColumnsSerial function.

---

**Require:** Column  $col$  composed by five 4-bit nibbles

**Ensure:** Column  $col$  composed by five 4-bit nibbles after post-multiply with the matrix  $M$

```

function MixSingleColumn( $col$ )
   $nibble[0] \leftarrow [col[0], col[1], col[2], col[3], col[16]]$ 
   $nibble[1] \leftarrow [col[4], col[5], col[6], col[7], col[17]]$ 
   $nibble[2] \leftarrow [col[8], col[9], col[10], col[11], col[18]]$ 
   $nibble[3] \leftarrow [col[12], col[13], col[14], col[15], col[19]]$ 
  for  $i$  in range(4) do
     $nibble[0], nibble[1], nibble[2], nibble[3] \leftarrow nibble[1], nibble[2], nibble[3],$ 
     $mc2(mc2(nibble[0])) \oplus nibble[1] \oplus mc2(nibble[2]) \oplus mc2(nibble[3])$ 

```

**Require:** State  $RS$

**Ensure:** State  $RS$  after the MixColumnsSerial operation

```

function MixColumnsSerial( $RS$ )
   $col0 = [RS[0], RS[1], RS[2], RS[3], RS[16], RS[17], RS[18], RS[19], RS[32], RS[33],$ 
   $RS[34], RS[35], RS[48], RS[49], RS[50], RS[51], RS[64], RS[68], RS[72], RS[76]]$ 
   $col1 = [RS[4], RS[5], RS[6], RS[7], RS[20], RS[21], RS[22], RS[23], RS[36], RS[37],$ 
   $RS[38], RS[39], RS[52], RS[53], RS[54], RS[55], RS[65], RS[69], RS[73], RS[77]]$ 
   $col2 = [RS[8], RS[9], RS[10], RS[11], RS[24], RS[25], RS[26], RS[27], RS[40], RS[41],$ 
   $RS[42], RS[43], RS[56], RS[57], RS[58], RS[59], RS[66], RS[70], RS[74], RS[78]]$ 
   $col3 = [RS[12], RS[13], RS[14], RS[15], RS[28], RS[29], RS[30], RS[31], RS[44], RS[45],$ 
   $RS[46], RS[47], RS[60], RS[61], RS[62], RS[63], RS[67], RS[71], RS[75], RS[79]]$ 

  MixSingleColumn( $col0$ )
  MixSingleColumn( $col1$ )
  MixSingleColumn( $col2$ )
  MixSingleColumn( $col3$ )

```

---

To compare the candidates, we must give a score to the S-boxes and select the best score among the candidates. In this work, a focus on the implementation cost is realized. Then, the score of a candidate is the number of logic gates needed to construct the S-box. The Algebraic Normal Form (ANF) of the S-box is used to count the number of AND and XOR gates. With the score function presented in Algorithm 10, the best representation on 5 bits is the S-box with the lowest number of logical gates. The next subsection is the application of the score function to every representation on 5 bits of a 4-bit S-box.

**Table 1:** 5-bit S-box derived from PRESENT to fill.

$x$	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
$S'[x]$	18	...	...	0A	...	0C	17	...	...	12	00	...	14	...	...	1B
$x$	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
$S'[x]$	...	06	1D	...	1E	...	...	11	09	...	...	0F	...	03	05	...

---

**Algorithm 10** Score of a S-box  $S$

---

**Require:** S-box  $S$

**Ensure:** Score of  $S$  (number of logical gates in the ANF)

```

function score( $S$ )
   $anf \leftarrow \text{ANF}(S)$ 
  return count(&) + count( $\wedge$ )

```

---

## 5.2 Exhaustive list

To fill the 5-bit S-box, a candidate must be selected among all the  $16!$  permutations. The obvious way to choose the best S-box is to score every candidate and to keep the one with the lowest score. This process is summarized in Algorithm 11 and is the most precise way to find the lowest score. Indeed, we would have the score of each function and then select the best one according to the criteria of implementation cost. However, it requires to browse all of the  $16!$  permutations, and this can be a very long task. A new solution based on the construction of the 5-bit representation can be as efficient and very easier to achieve.

---

**Algorithm 11** Selection of the S-box with the lowest number of gates

---

**Require:** List of all the 5-bit permutations derived from a 4-bit S-box  $\text{Permutation}_{\text{LIST}}$

**Ensure:** Permutation with the lowest score and its score

```

function score_selection( $\text{Permutation}_{\text{LIST}}$ )
   $\text{low\_score} \leftarrow 1000$ 
  for  $S \in \text{Permutation}_{\text{LIST}}$  do
     $s \leftarrow \text{score}(S)$ 
    if  $s < \text{low\_score}$  then
       $\text{low\_score} \leftarrow s$ 
       $\text{selected} \leftarrow S$ 
  return  $\text{low\_score}, \text{selected}$ 

```

---

## 5.3 Construction

A new selection method is introduced with a construction approach instead of an exhaustive approach. In this paragraph, an *even* word denotes a word that verifies the parity characteristic and an *odd* word a word which does not. Every even word is only 1 bit away

from an odd word. The LSB is used to separate an even from an odd word (0x18 and 0x19 are only 1 bit away from each other, and this bit is the LSB). Each even input is substituted by an even output and each odd input is substituted by an odd output. The 5-bit S-box is constructed with the following rule: an odd input is substituted by the odd word 1-bit away from the even output linked to the even input 1-bit away from the odd input. Indeed, each even pair of input/output have a 1-bit away odd pair of input/output. This construction is explained in the Algorithm 12. With this method, the representation of PRESENT is shown in Table 2 and consists of 62 logical gates. Several S-Boxes with good cryptographic properties were tested and none has an ANF constructed with less than 94 logical gates (the biggest one was created with 124 logical gates). We now have to test the robustness of the cipher.

---

**Algorithm 12** Construction of a code abiding 5-bit representation from a 4-bit S-box.

---

**Require:** S-box S half-filled

**Ensure:** S-box S full-filled

```

for  $i$  in range(32) do
  if  $i$  is odd then
     $S[i] \leftarrow S[i \oplus 1] \oplus 1$ 

```

---

**Table 2:** 5-bit code abiding representation constructed from PRESENT

$x$	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
$S'[x]$	18	19	0B	0A	0D	0C	17	16	13	12	00	01	14	15	1A	1B
$x$	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
$S'[x]$	07	06	1D	1C	1E	1F	10	11	09	08	0E	0F	02	03	05	04

## 6 Experimental results

This section presents the various tests done on the protected LED to determine its robustness against 1-bit fault injection.

### 6.1 Scenarios

To test the robustness of the protected LED cipher, four scenarios are tested. The detection of a fault injected simply sets a variable `flag` to 0, but other actions can be done when an error is detected. During the tests, the fault injection is simulated, so there is no case where a fault does not create an error.

**Scenario 1** A bit of the state is toggled at a random place of the state and at a random moment of the encryption. This bit-flip induces a change on the parity characteristic of the nibble where it belongs. With the code abiding properties of the functions used during the encryption, the error persists until the parity check function and thus is always detected.

**Scenario 2** A bit of the key or the constant of the `AddConstant` function is toggled at a random place and a random round of the encryption. As the XOR operation is a code abiding operation, the fault is transmitted from the constant to the state and persists until the parity check. The error is thus always detected.

**Scenario 3** A fault is injected on a data used more than once during a function at a random place and a random round of the encryption. The copies done before the use of the data are then not equal and the test sets the `flag` to 0. The fault is thus always detected.

In all the scenarios, the fault is always detected, then the code abiding solution is robust against 1-bit fault attack.

In these tests, 100 000 faults have been injected for each scenarios and all the fault have been detected. The results are presented in Table 3.

**Table 3:** Robustness results of the secured implementation.

Fault injections	Scenario 1	Scenario 2	Scenario 3
100 000	100% detected	100% detected	100% detected

In our work, we just decide to raise a flag to show that the fault is detected. It is the simplest action that can be done when a fault is detected but this solution has some drawbacks: it shows that an error is detected and if a fault is injected on this precise flag bit, then no matter if an other error is injected into the state, it is not detected. However, several other decisions can be taken according to the error detection:

- **No cipher output.** If an error is detected, the algorithm does not display the faulty ciphertext
- **Zero value output.** If an error is detected, the ciphertext displayed is 0.
- **Random cipher output.** If an error is detected, the ciphertext displayed is totally randomized.

All of these solutions remove the link between the ciphertext and the key, as the ciphertext, if it is displayed, is created without using the key. However, the two first schemes allow the adversary to use a statistical analysis. Indeed, a method to statistically analyze the effectiveness of the fault attack can be applied to recover the key [DEK<sup>+</sup>18].

## 6.2 Overcost of the countermeasure

Adding the parity scheme to the LED cipher has a cost. Indeed, we convert an encryption algorithm working on 4-bit words to an encryption algorithm working on 5-bit words. Thus, the new round functions have a bigger price than the former ones. Moreover, our  $n$  states are 80-bit long instead of 64-bit long (we encrypt  $n$  plaintexts in parallel, and in the tests, we fix  $n = 64$ ). Thus, our implementation takes a bigger place in the memory and one secure encryption takes longer than an unprotected encryption. We differentiate several implementations: the *classical* implementation refers to the soft implementation using lookup tables; the *bitslice* is the bitslice version of LED without any protection; the *code abiding* implementation is the addition of the parity bit during the encryption; and the *code abiding + copies* implementation combines the code abiding properties with copies of values used more than once. The cost can be summarized in the Table 4. The compiler used was the GNU GCC Compiler without any optimization to be closer to a physical implementation. The CPU used is the Intel Core i5 CPU. The results are presented as a ratio to have a better understanding of the overcost of countermeasures from one implementation to another. The results a must be put into perspective as the classical implementation only one plaintext at the time when the other implementations can encrypt up to 64 plaintexts at the same time (on a 64-bit length machine). The overcost of the code abiding countermeasure is then better than expected. Indeed, in terms of time overcost, a 25% rise was expected (25% more bits are computed) when an only 12% is measured. However, with the copies countermeasure, an overcost of 79% is reached.



**Table 4:** Implementation results and cost comparison of the encryptions.

	Ratio <i>classical</i>	Ratio <i>bitslice</i>	Ratio <i>code abiding</i>
<i>classical</i>	1	-	-
<i>bitslice</i>	1,83	1	-
<i>code abiding</i>	2,04	1,12	1,00
<i>CA + copies</i>	3,28	1,79	1,6

Moreover, another comparison on each round function allows to precisely understand where the countermeasure has the biggest impact (see Table 5). The heaviest functions from the classical implementation to the other ones are clearly the *subCells* as the function does not use any lookup table and the *addConstant* as the constant used must be transformed into a bitslice and parity constant. However, as mentioned before, it is more interesting to compare the bitslice versions as they encrypt the same number of plaintexts and are based on the same principles. With these comparisons, the biggest overcost is the *mixColumns* function.

**Table 5:** Implementation results and cost comparison of the round functions.

<b>Ratio classical</b>	<i>bitslice</i>	<i>CA</i>	<i>CA + copies</i>
<i>addConstant</i>	6,6	8,6	8,6
<i>subCells</i>	6,0	7,2	9,6
<i>shiftRows</i>	0,8	0,9	0,9
<i>mixColumns</i>	1,9	2,1	3,6

<b>Ratio bitslice</b>	<i>CA</i>	<i>CA + copies</i>
<i>addConstant</i>	1,3	1,3
<i>subCells</i>	1,2	1,6
<i>shiftRows</i>	1,2	1,2
<i>mixColumns</i>	1,1	1,9

<b>Ratio CA</b>	<i>CA + copies</i>
<i>addConstant</i>	1,0
<i>subCells</i>	1,3
<i>shiftRows</i>	1,0
<i>mixColumns</i>	1,7

## 7 Conclusion

The principle used in this work to prevent fault injections is to detect them using an error detecting code, the parity bit code. This code relies on a redundancy of the information contained in a word. The parity bit code used is the 5-bit parity code, with 4 data bits and 1 parity bit. This method allows to detect a 1-bit fault injection on a value during an operation.

This work lightens an issue induced by an error compensation. Indeed, depending on the operation performed, an error injected on a value can be propagated into several computed outputs and with the parity bit code, this error may compensate with its multiple occurrences. The first step is then to present the conditions on the fault and on the operation to reach the compensation, and then to propose a countermeasure to this error compensation that lies on copying the values used more than once and check for equality of the copies.

In addition to this first measure, a method is presented to apply code-abiding notion to word-oriented ciphers. An example on the LED cipher shows the transformations of the state and the round functions to include the parity bit code to the operations. A protected version of the existing LED cipher is then created. Its robustness against 1-bit fault injection is tested and the results validate its security. Moreover with the bitslice method, the robustness reaches 1-word fault injection detection.

The next step is to extend this method to a generic one to include code abiding to new cryptographic primitives. A critical operation is the S-box used and the projection of this S-box into a larger space to add the parity bit brings many candidates. A way to differentiate them is to give them a score based on their implementation cost and select the cheapest S-box.

Eventually, future works could focus on applying the code-abiding method to a larger cipher such as AES rather than lightweight ciphers, and evaluate the overcost of the countermeasure compared to other error detecting solutions. Moreover, 1-bit error detection has its limitations and a work on multiple faults detection and correction would be interesting.

## References

- [BBB<sup>+</sup>21] Anubhab Bakshi, Shivam Bhasin, Jakub Breier, Anupam Chattopadhyay, and Vinay B. Y. Kumar. Feeding three birds with one scone: A generic duplication based countermeasure to fault attacks. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*, pages 561–564. IEEE, 2021.
- [BBK<sup>+</sup>02] Guido Bertoni, Luca Breveglieri, Israel Koren, Paolo Maistri, and Vincenzo Piuri. A parity code based fault detection for an implementation of the advanced encryption standard. In *17th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2002), 6-8 November 2002, Vancouver, BC, Canada, Proceedings*, pages 51–59. IEEE Computer Society, 2002.
- [BHL21] Jakub Breier, Xiaolu Hou, and Yang Liu. On evaluating fault resilient encoding schemes in software. *IEEE Trans. Dependable Secur. Comput.*, 18(3):1065–1079, 2021.
- [BLL15] Zhenzhen Bao, Peng Luo, and Dongdai Lin. Bitsliced implementations of the prince, LED and RECTANGLE block ciphers on AVR 8-bit microcontrollers. In Sihang Qing, Eiji Okamoto, Kwangjo Kim, and Dongmei Liu, editors, *Information and Communications Security - 17th International Conference, ICICS 2015, Beijing, China, December 9-11, 2015, Revised Selected Papers*, volume 9543 of *Lecture Notes in Computer Science*, pages 18–36. Springer, 2015.
- [BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997.
- [DEK<sup>+</sup>18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: exploiting ineffective fault inductions on symmetric cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):547–572, 2018.

- [GPPR11] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The LED block cipher. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2011.
- [KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
- [KMD<sup>+</sup>20] Pantea Kiaei, Darius Mercadier, Pierre-Évariste Dagand, Karine Heydemann, and Patrick Schaumont. Custom instruction support for modular defense against side-channel and fault attacks. In Guido Marco Bertoni and Francesco Regazzoni, editors, *Constructive Side-Channel Analysis and Secure Design - 11th International Workshop, COSADE 2020, Lugano, Switzerland, April 1-3, 2020, Revised Selected Papers*, volume 12244 of *Lecture Notes in Computer Science*, pages 221–253. Springer, 2020.
- [LCFS17] Benjamin Lac, Anne Canteaut, Jacques J. A. Fournier, and Renaud Sirdey. Thwarting fault attacks using the internal redundancy countermeasure (IRC). *IACR Cryptol. ePrint Arch.*, page 910, 2017.
- [LJK21] Seungkwang Lee, Nam-Su Jho, and Myungchul Kim. Table redundancy method for protecting against fault attacks. *IEEE Access*, 9:92214–92223, 2021.
- [PYGS16] Conor Patrick, Bilgiday Yuce, Nahid Farhady Ghalaty, and Patrick Schaumont. Lightweight fault attack resistance in software using intra-instruction redundancy. In Roberto Avanzi and Howard M. Heys, editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, volume 10532 of *Lecture Notes in Computer Science*, pages 231–244. Springer, 2016.
- [REB<sup>+</sup>08] Francesco Regazzoni, Thomas Eisenbarth, Luca Breveglieri, Paolo Ienne, and Israel Koren. Can knowledge regarding the presence of countermeasures against fault attacks simplify power attacks on cryptographic devices? In Cristiana Bolchini, Yong-Bin Kim, Dimitris Gizopoulos, and Mohammad Tehranipoor, editors, *23rd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2008), 1-3 October 2008, Boston, MA, USA*, pages 202–210. IEEE Computer Society, 2008.
- [SBD<sup>+</sup>20] Thierry Simon, Lejla Batina, Joan Daemen, Vincent Grosso, Pedro Maat Costa Massolino, Kostas Papagiannopoulos, Francesco Regazzoni, and Niels Samwel. Friet: An authenticated encryption scheme with built-in fault detection. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EURO-CRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 581–611. Springer, 2020.