BALoo: First and Efficient Countermeasure dedicated to Persistent Fault Attacks

Pierre-Antoine Tissot[1], Lilian Bossuet[1] and Vincent Grosso[1]
[1]*Université Jean Monnet Saint-Etienne, CNRS, Institut d Optique Graduate School, Laboratoire Hubert Curien UMR 5516, F-42023, SAINT-ETIENNE, France*

***Abstract*—Persistent fault analysis is a novel and efficient cryptanalysis method. The persistent fault attacks take advantage of a persistent fault injected in a non-volatile memory, then present on the device until the reboot of the device. Contrary to classical physical fault injection, where differential analysis can be performed, persistent fault analysis requires new analyses and dedicated countermeasures. Persistent fault analysis requires a persistent fault injected in the S-box such that the bijective characteristic of the permutation function is not present anymore. In particular, the analysis will use the non-uniform distribution of the S-box values: when one of the possible S-box values never appears and one of the possible S-box values appears twice.**

In this paper, we present the first dedicated protection to prevent persistent fault analysis. This countermeasure, called BALoo for *Bijection Assert with Loops*, checks the property of bijectivity of the S-box. We show that this countermeasure has a 100% fault coverage for the persistent fault analysis, with a very small software overhead (memory overhead) and reasonable hardware overhead (logical resources, memory and performance). To evaluate the overhead of BALoo, we provide experimental results obtained with the software and the hardware (FPGA) implementations of an AES-128.

***Index Terms*—Fault attacks, persistent fault analysis, fault countermeasure, permutation properties.**

I. Introduction

In our everyday life, we use more and more connected devices which need to exchange secret data, thus, they need to establish secure communications. One main component of secure communication protocol is block ciphers, *e.g.* AES [1], PRESENT [2]. Block ciphers are composed of three main layers: key mixing, linear layer and non-linear layer. The non-linear part of these block ciphers lies on a permutation function, represented by an S-box. This function must bring confusion and non-linearity to the state. Its security is then crucial.

When implemented on physical devices, these ciphers are vulnerable against physical attacks such as side-channel attacks [3] (taking advantage of the circuit's leakages) and fault attacks [4] (taking advantage of the device's sensitivity to perturbations).

The principle of fault attacks is to use a physical threat (laser injection [5], clock glitching [6], ...) in order to inject faults during an encryption and extract information by analyzing the impact of the injected fault. Most of the studies already done are based on transient faults: faults injected in run time, with a unique and immediate impact [7]. The adversary studies the difference between a faulty ciphertext and a fault-free ciphertext, both encrypted from the same plaintext and same key, to get information about the secret key. With transient faults, the adversary can still get fault-free ciphertext. However, the drawback of this type of injection is that the adversary must inject a fault every time he wants the data to be affected, and these injections can be hard to perform and to success.

A novel type of fault analysis has been introduced by Zhang *et al.* in [8]. The analysis is based on faults called *persistent injections*, in opposition to transient faults. These faults are performed to keep a data value faulty whenever this data is used. This type of faults are more detailed in the following background section. The analysis performed by Zhang *et al.* is based on a persistent fault injected in the block cipher S-box. This analysis is also detailed in the following section. However, no countermeasure against this precise type of injection has been presented in the state of the art and persistent fault were designed to bypass transient countermeasures. Indeed, the attack has been studied several times and improved in some aspects of the analysis [9] (reduces the constraints on the injection), [10] (persistent fault analysis performed with deep learning), but no countermeasures specifically designed against persistent fault analysis have been presented. Indeed, some on-line tests already exist [11], [12], but no specific test against PFA.

Our contribution is the proposition of the first countermeasure dedicated to persistent fault analysis. The solution is based on the length of the cycles found with the loop construction of any permutation function. We call this solution BALoo for *Bijection Assert with Loops*. In this paper, we explain how to construct loops from any permutation function (with some results on widely used block ciphers, as well as on the NIST Lightweight Cryptography candidates that use S-boxes in their implementation and the AES). Then, we evaluate the overhead of BALoo with an AES-128 software implementation and AES-128 hardware implementation (FPGA).

The paper is divided as follows: Section II presents all the background notions useful in the rest of the work (persistent fault attack functioning and limitation and fault model considered). Section III is the BALoo presentation, this section introduces and explains the solution and its fault coverage against persistent fault injection. Section IV answers several general questions on the loop construction of the permutations and Section V gives the software and hardware overhead of the BALoo solution.

II. Background

This section presents the different useful notions for the rest of the study: the persistent fault analysis and the fault models targeted.

A. Persistent Fault Analysis

Fault attacks are threats against cryptographic implementation. Since their presentation in the late 90's [4], [13], transient faults (faults injected in run time that affect data during a limited and short time) were used, and differential analysis allows to recover some sensitive data. The main idea is to compare the output of the algorithm with and without fault injection, the difference helps the adversary to reduce the guessing entropy of the sensitive data. Various methods have been presented on different target, *e.g.* [7], [14]–[16].

Recently, persistent fault injection analysis has been introduced [8]. The persistent fault injection affects the value of a data stored in FPGA memory using the rowhammer attack [17]. The target data remains faulty until the reboot of the memory. With this principle, the adversary can inject the fault at any moment of the device's life, and the fault will remain active until the reboot. With this type of fault, the adversary is then immune to temporal redundancy. Indeed, temporal redundancy is a widely used countermeasure based on multiple encryptions of a plaintext and a comparison of the obtained ciphertexts. As presented in [8], with a persistent fault on an AES S-box, no matter how many times the same plaintext is encrypted, the faulty ciphertexts are always the same. A statistical study of the ciphertexts distribution can lead the adversary to recover bytes of the last round subkey. The probability distributions and their employment during the analysis are shown in Figure 1. While non-invertible S-boxes as been used in the past, *e.g.* DES [18], nowadays, with better knowledge of Boolean vector function, permutations are usually used for S-boxes, in the rest of the paper we assume that the S-box is a permutation.

The adversary uses a divide-and-conquer approach, he attacks each S-box byte independently. For each output word, he captures the output of several encryptions $\{c_j\}$, we denote by C_j the random variable of the output of the cipher for the word considered (j refers to the byte j of the word). From these observations he can determine the probability of each possible output c_j (v represents the value of the AES S-box that never appears and v^* the value that appears twice):

- $P_r(C_j = c_j) = 0 \Rightarrow c_j = v \oplus k_j$,
- $P_r(C_j = c_j) = max_{c'_j}(P_r(C_j = c'_j)) \Rightarrow c_j = v^* \oplus k_j$,
- $0 < P_r(C_j = c_j) < max_{c'_j}(P_r(C_j = c'_j)) \Rightarrow c_j \neq v \oplus k_j$ and $c_j \neq v^* \oplus k_j$.

In this analysis, the adversary knows which fault was injected (v and v^* are known). Thus, from the distribution of the $\{c_j\}$, the adversary can use the two first relations to recover the value of the secret subkey, *e.g.* $P_r(C_j = c_j) = 0 \Rightarrow k_j = c_j \oplus v$. With enough ciphertexts (Zhang *et al.* give a result of around 2 000 faulty ciphertexts needed to recover the key byte with v and v^* known [8]), the probability distribution of the ciphertext will get close to the one shown in Figure 1, in the faulty encryption frame. Thanks to the fault injection knowledge (v and v^* known), the adversary can then recover k. Note that here is presented the analysis with the best knowledge about the injection, but Zhang *et al.* also present several analyses with less constraints [9] (v and v^* not known, some injections not successful, ...).

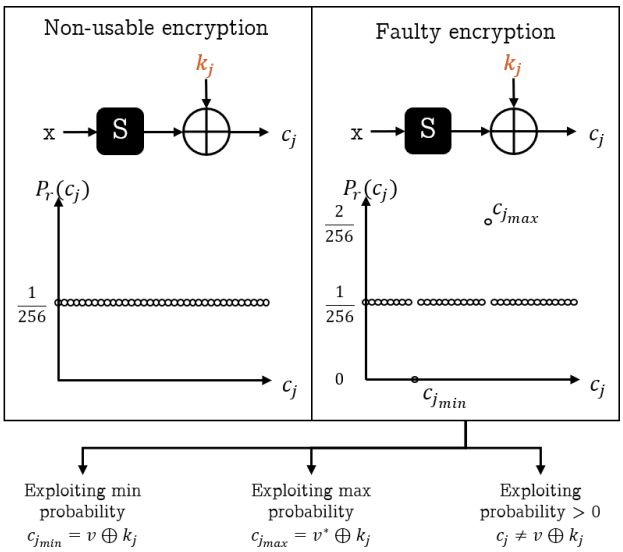

Fig. 1. Overview of persistent fault analysis [8]. v represents the value of the AES S-box that never appears and v^* the value that appears twice.

The fault model only takes into account faults that delete at least one value of the S-box. Indeed, the rearrangements of the S-box are not considered as usable S-boxes as the adversary cannot exploit a faulty probability of distribution if the distribution is the same before and after the fault injection. The persistent fault analyses presented so far exploit a value distribution that is different from the non-faulty one.

Contrary to transient analysis, the adversary does not need to have control on the plaintexts. He just requires random ciphertexts.

B. Multiple Fault Injections

When analyzing a persistent fault injection, the main information acquired by the adversary is the probability of distribution of the ciphertexts. The goal of the attack is to highlight the value that never appears or the value that appears twice. If multiple faults are injected in the S-box, several values would be deleted and several other values would have a higher probability of occurrence: $Pr(C_j = c_j) = 0$ is true for several c_j, and $max_{c_j}(Pr(C_j = c_j))$ is also reached for multiple values (see Figure 2). Thus, the attack would be less efficient with several faults injected. Clearly, if two values c_{j_1}, c_{j_2} never appear, the adversary cannot decide if $k_j = v \oplus c_{j_1}$ or $k_j = v \oplus c_{j_2}$, hence he has 2 candidates for each subkey, he thus needs to test all full key candidates 2^{16} to recover the good one, when considering AES-128 case study.

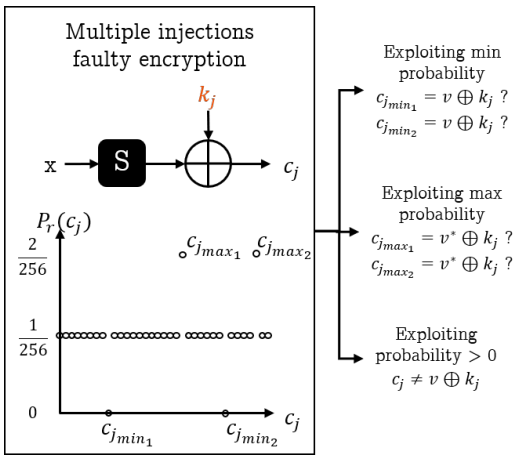

Fig. 2. Probability Distribution in case of a Multiple faults injection

However, a fault that affects only one byte brings high constrains on the injection: the adversary has to be able to attack the memory with a very precise threat.

C. Fault Models

To describe the fault models that we use in this work, some notations must be introduced. The injected faults will be indexed by the integer r, and each fault has its own value, denoted f_r. Then, the i-th value of the S-box is S_i, and its faulty value by the fault f_r is $S_i^{f_r}$.

Moreover, an injected fault will have three different impacts on the data, depending on the injection technique ($\vee$ denote a `Or` operation, $\wedge$ a `And` operation and $\oplus$ a `Xor` operation). Usually, only one kind of modification is considered possible, thus the operation is given by the context.

- SET model: $S_i^{f_r} = S_i \vee f_r$
- $RESET$ model: $S_i^{f_r} = S_i \wedge f_r$
- $FLIP$ model: $S_i^{f_r} = S_i \oplus f_r$

An example of a fault injection with the SET model ($f_0 = 010$ and $S_5^{f_0} = S_5 \vee f_0$) is presented in Table I. $S_x^{f_0}$ denotes a faulty S-box with a value (3) that appears twice and a value (1) that never appears. In another case, $S_x^{f_1,f_2}$ is a faulty S-box with the $FLIP$ model ($f_1 = f_2 = 110$ with $S_3^{f_1} = S_3 \oplus f_1$ and $S_7^{f_2} = S_7 \oplus f_2$) but is still a permutation and then it is not usable by the adversary.

TABLE I
FAULT INJECTION EXAMPLE ON A SIMPLE S-BOX.

x	0	1	2	3	4	5	6	7
S_x	3	5	0	2	7	1	6	4
$S_x^{f_0}$	3	5	0	2	7	3	6	4
$S_x^{f_1,f_2}$	3	5	0	4	7	1	6	2

In the $S_x^{f_0}$ situation, let's suppose that in the ciphertexts, the value 7 never appears, the value 5 appears twice as much and the other values appear with a $\frac{1}{8}$ probability. With enough ciphertexts, all values but the value 7 are found, then the adversary knows that $c_{j_{min}} = 7$. With the fault f_0 ($f_0 = 3$) injected, and the value $v = 1$ corresponding to the output of the S-box that disappeared, the adversary can compute $k_j = c_{j_{min}} \oplus v = 7 \oplus 1 = 6$. The adversary has found the last round key $k_j = 6$.

The robustness of the solutions proposed is detailed according to the different fault models.

III. Proposed Countermeasure

The main goal of the proposed countermeasure is to detect if an usable (from the adversary point of view) permanent fault has been injected in the S-box. Indeed, a check of all the values of the S-box must reveal that they all emerge exactly once. To do it, the first naive idea would be to use redundancy as used in usual faults injection countermeasures. Nevertheless, persistent fault is naturally resistant to temporal redundancy as the fault is permanent: for a fixed plaintext, the ciphertext will always be the same no matter when the encryption is done. According to the fault model, persistent fault analysis can also resist to spatial redundancy because the adversary is able to inject the same fault in both S-boxes (being able to inject one fault in one S-box is as difficult as injecting a fault in another S-box). Another approach is to use the fact that the S-box is a permutation function and to find intrinsic properties of this function that allow to detect fault injections. This last approach is detailed in the next subsection.

A. Permutation properties

Three properties of a permutation that could detect fault injections are detailed.

The first one is that the sum of all the elements i of the permutation function is a known constant. This constant can be pre-computed and compared to the actual sum.

Since we consider permutations of n-bit words, we also know that the `Xor` of the 2^n possible values of words should be 0. Using `Xor` operation instead of `Add` operation is also a solution to detect an injected fault.

The last property is that any permutation function can be decomposed as a set of cycles. For example, the S-box presented in Table II, can be decomposed in $(0, 3, 2)(1, 5)(4, 7)(6)$. Where $(0, 3, 2)$ means that $S_0 = 3, S_3 = 2$ and $S_2 = 0$ and thus, these three values form a permutation loop.

TABLE II
S-BOX EXAMPLE WITH THE FOLLOWING SET OF LOOPS $(0, 3, 2)(1, 5)(4, 7)(6)$.

x	0	1	2	3	4	5	6	7
S_x	3	5	0	2	7	1	6	4

For each specific S-box, the loops have a precise length. The calculation of these lengths (Algorithm 1) can be used to detect faults as we propose for the BALoo countermeasure (Algorithm 2). Indeed, a modification in a loop length can only happen if at least a faulty value of the S-box is present.

We can see that the permutation function properties are easy to check. However, it can happen that some very precise injections sometimes lead to undetectable faults, as we will see in the next section.

Algorithm 1 `length` function

Require: S-box S, first index i, target length l
Ensure: length calculated = length targeted
  $t = 1$ ▷ loop length
  $j = i$ ▷ index
  while $(S[j] \neq i) \& (t < l + 1)$ **do** ▷ first index not reach + length < target
    $j = S[j]$ ▷ next step
    $t++$ ▷ actual length + 1
  return $t == l$ ▷ length computed = target ?

Algorithm 2 $BALoo$ countermeasure

Require: An n-bit S-box S and the list of loop length L_l and start index L_i and their number s.
Ensure: Fault detection
  $FaultD = \texttt{False}$
  $j = 0$
  while $j < s$ **do**
    $FaultD = FaultD \vee \neg \texttt{length}(S, L_i(j), L_l(j))$
    $j++$
  return $FaultD$

B. Fault coverage

1) Add property: This solution is based on the addition of all the values of the S-box that should give a constant result. With the SET (resp. $RESET$) fault model, no matter how many faults are injected, their index targeted, and their value, the sum is always higher (resp. lower) to the pre-computed value and the faults are always detected.

However, with the $FLIP$ fault model, some faults remain undetected. Indeed, if some bits are flipped to 1, other bits can be flipped to 0 to compensate and keep the same result of the sum.

2) Xor property: This solution is based on the `Xor` of all the values of the S-box that should give 0. If some faults modify the final result of the `Xor`, then the faults are detected. However, no matter the fault model used, some combinations of fault injections are still undetected for $FLIP$, SET and $RESET$ models, as several faults can offset each other and lead to an exclusive sum of 0.

3) BALoo countermeasure (Algorithm 2) : As presented before, any S-box can be represented by a set of loops. A loop begins with a value (any value of the loop), and contains all the steps until recovering the original value. For example, the AES S-box is composed of 5 loops. The loop which contains the value 11 is 27-step length: $S_{11} = 43$, $S_{43} = 241$, ..., $S_{158} = 11$ (this loop is illustrated on the first loop on Figure 3). The important information about loops is their length. For example, with the AES S-box, the length of the loop in which we can find the 11 value is exactly 27. If a fault affects a value of this loop, the length becomes either lower or higher to the former length. Then, by measuring the loop lengths, we can detect fault injections. However, in the fault model in which any type of fault is possible, some precise fault selection leads to undetected injections. A simple example of this fault type is presented in the second loop of Figure 3. The length of this loop is still 27 but three faults are injected in the S-box. Yet, these faults modify the S-box, but the faulty S-box remains a permutation function (the faulty values are just swapped with other faulty values), then the injected fault cannot be used by any persistent fault analysis as presented in [8] because this type of fault that does not create a modification in the probability distribution of the S-box values. As a consequence, the BALoo countermeasure does not admit any undetected faults that could be exploited by an adversary who applies a Persistent Fault Analysis [8].

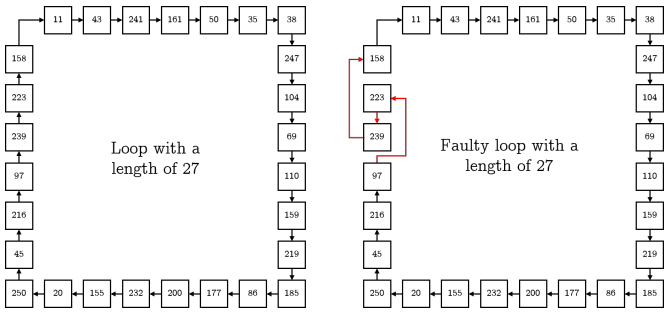

Fig. 3. Example of an AES loop with a length of 27.

C. Robustness

To validate the theoretical fault coverage results, the different countermeasures are implemented and tested. For each fault model and each number of injected faults, 10 000 000 tests are performed. The random fault model is the worst case for the adversary and thus gives the upper bound of the fault coverage. Indeed, if the fault coverage is different than 100%, that means that with an adversary in a stronger model (chosen fault), the fault coverage can be as low as 0.

Figure 4 shows the fault coverage of the BALoo countermeasure in the SET or $RESET$ fault model. Indeed, both give the same results as the models are similar. The fault coverage of BALoo is presented in comparison with the other solutions (`Xor`, `Add` and Duplication).

Figure 5 shows the fault coverage of the BALoo countermeasure in the $FLIP$ fault model, with the same comparison with the other solutions.

In each fault model, the spatial duplication countermeasure (with the S-box duplicated and a vote between the values of the S-boxes) leads to undetected faults. Indeed, for an even number of faults, half of the faults are injected in the first S-box, and the other half in the other S-box. The faults must have the same impact on both S-boxes to be undetected: same S-box value and same impact of this value. This reasoning can also be applied to a n-time duplication, with $(n + 1)$ faults injected in the $(n + 1)$ duplicated S-boxes.

The `Add` solution gives a 100% fault coverage in the $SET/RESET$ model and a lower coverage in the $FLIP$ model. The `Xor` solution leads to undetected fault in both models.

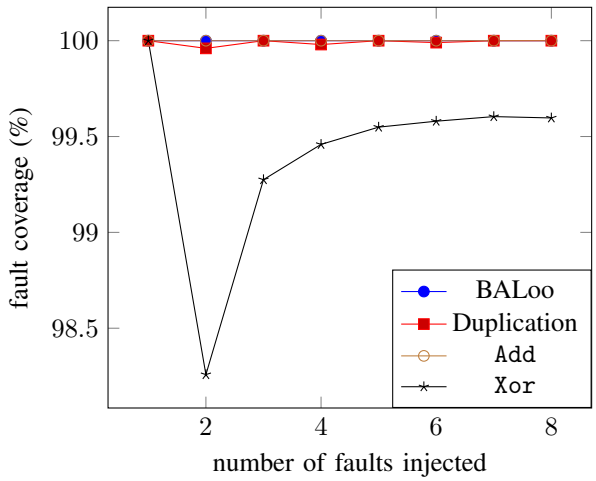

Fig. 4. Fault coverage for the SET and $RESET$ models.

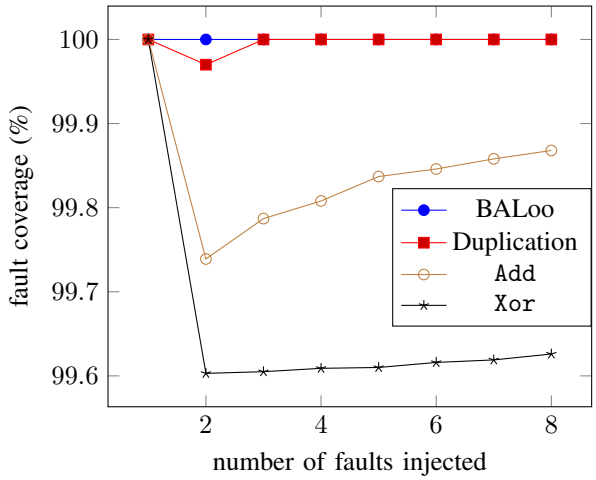

Fig. 5. Fault coverage $FLIP$ model.

The BALoo countermeasure always gives 100% fault coverage as no fault included in the fault model can be undetected. Indeed, a function is a permutation if and only if a loop construction can be extracted from the function. In a formal loop definition, each value is the only input of a value and the only output of another value (note that both values can be the same and both can also be the original value). A function constructed from loops is then always a permutation and a permutation can always be interpreted as a set of loops. With this property, the exploitable faults lead to functions that are not permutations and then the BALoo countermeasure always detects them.

The next step is to see how to compute the length of loops to implement the BALoo countermeasure for any block cipher with bijective S-box.

IV. Loops on S-boxes in a general case

The loop decomposition of a S-box is unique. To use the BALoo countermeasure, the loops must be previously found, and their length computed in off-line. Algorithm 3 shows how to find the different loops from an S-box. This algorithm can be processed for any S-box, no matter its size.

Algorithm 3 Find the loops of an S-box.

Require: S-box S
Ensure: Loops of S
  $loops = [\,]$ ▷ List of the loops
  while $len(loops) < len(S)$ **do**
    $i = 0$
    while $i \in loops$ **do** ▷ First index of the next loop
      $i++$
    $l = [i]$
    while $S[l[-1]] \neq i$ **do** ▷ The loop l is not complete
      $l.append(S[l[-1]])$
    $loops.append(l)$ ▷ l is complete and added to $loops$
  return $loops$

The application of the algorithm on several widely used cipher S-boxes is presented in Table III with the number of loops and the longest loop (in case of a parallel verification, the longest loop determines the countermeasure duration). The block ciphers are: LOW MC [19], PRESENT [2], PRINCE [20], GIFT [21], NOEKEON [22], PRIMATE [23], ELEPHANT [24], GIFT-COFB [21], AES [1] and ROMULUS [25].

For the GIFT-COFB cipher, several S-boxes are used in the same substitution layer. We add an index to differentiate the different S-boxes of this cipher.

TABLE III
Determination of the loops length on several used S-boxes

Cipher	Block size	Number of loops	Longest length
LOW MC	3	3	6
PRESENT	4	4	7
PRINCE	4	4	8
GIFT	4	2	9
NOEKEON	4	10	2
PRIMATE	4	5	11
ELEPHANT	4	2	13
GIFT-COFB$_1$	5	2	31
GIFT-COFB$_2$	5	4	10
GIFT-COFB$_3$	5	2	31
GIFT-COFB$_4$	5	8	5
AES	8	5	87
ROMULUS	8	12	140

For example, the length of the AES loops found are:

- 87 for the loop with the value 4
- 81 for the loop with the value 1
- 59 for the loop with the value 0
- 27 for the loop with the value 11
- 2 for the loop with the value 115

V. BALoo Software and Hardware Overhead

This section presents the overhead of the BALoo countermeasure with a software implementation and an hardware implementation (FPGA) of an AES-128.

A. Software memory overhead

The software overhead measured is the number of S-box memory accesses of the BALoo countermeasure. A single AES SubBytes operation needs 16 S-box bytes read instructions and a full AES-128 encryption is composed by 10 rounds which call the SubBytes function once. That means that for every encryption, 160 bytes are accessed in memory. The persistent fault analysis [8] shows that even with the strongest adversary, around 2000 ciphertexts are needed to recover the key (guessing entropy close to 0). With this hypothesis, we choose to perform a fault injection check every 1000 encryptions (until a fault detection) to stay robust against the persistent fault analysis (guessing entropy higher than 40 bits). The BALoo countermeasure browses all the S-box values once, which means 256 memory bytes read instructions. 256 countermeasure calls are thus added every 160 000 encryption accesses (a memory calls overhead of 0.16%). If a developer wants to keep a higher guessing entropy he can apply the BALoo countermeasure every 500 encryption and keep more than 80 bits of guessing entropy for an overhead of 0.32%.

B. Hardware overheads

The hardware overhead of the BALoo countermeasure is measured according to three resources: the number of logical elements, the number of registers and the memory bits used. The device on which the AES and its protection are implemented is the FPGA Intel Cyclone V. We use Quartus II with best effort for performance and area synthesis options.

The AES implementation is composed by 16 S-boxes used in parallel to encrypt a plaintext in only 11 clock cycles. However, this means that the BALoo countermeasure must be applied on all of the 16 S-boxes. Instead of applying the countermeasure on all the 16 S-boxes (this solution was tested and results are very costly as 16 S-boxes must be protected, giving an overhead of 356% in terms of logical elements), we add another S-box to reach 17 S-boxes. These 17 S-boxes allow to verify one S-box in the same time as the other 16 are used.

The verification of one S-box is at most 256 clock cycles long, and one encryption takes 11 cycles. That means that a check is around 24 encryptions and the check of all the 17 S-boxes takes 408 encryptions. With the limit of 1000 encryptions to avoid Persistent Fault Analysis, the AES implementation with this protection is secured.

This countermeasure adds 1 S-box but also some multiplexers in order to select which S-boxes are used and which S-box is verified. The implementation costs results are presented in Table IV. The overhead of the protection is quite similar to the naive comparison solution, but with the advantage of detecting all the faults injected. Moreover, to accelerate the detection, another S-box can be added to check two S-boxes at the same time. The overhead of this 18th S-box is small in comparison with the 17 S-boxes implementation.

Moreover, an estimation of the theoretical maximum frequency of the FPGA is given, as well as the power consumption of the circuit. Note that those data are only estimations provided by the Quartus-II tool.

TABLE IV
IMPLEMENTATION RESULTS ON FPGA AND PROTECTION OVERHEAD.

Performances with Cyclone V 5CGXFC9E7F35C8

	FPGA resources				
	Logic (ALM)	Registers	Memory bits	Power (mW)	Fmax (MHz)
AES	339	13	32 768	566,32	94
Secure AES with 17 S-boxes	608	34	34 816	568,78	58
Secure AES with 18 S-boxes	763	43	36 864	568,48	51

	Overhead %				
	Logic (ALM)	Registers	Memory bits	Power (mW)	Fmax (MHz)
Secure AES with 17 S-boxes	79	162	6	0,43	-38
Secure AES with 18 S-boxes	125	231	13	0,38	-46

The logic overhead is $+79\%$ with one verification S-box and $+125\%$ with two verification S-boxes. With both check configuration, the memory overhead is very low.

As the critical path is increased, the maximum frequency is then reduced, but with no optimization on the implementation, the reduction is reasonable (around -35%). This result depends on many synthesis parameters, but it gives a general idea of the overhead.

This solution has a slightly smaller overhead than spatial redundancy, but with the advantage of fault coverage and the possibility to accelerate detection, with low-cost additional permutations.

VI. CONCLUSION

This paper presents the first countermeasure dedicated to permanent fault analysis of symmetric block cipher. Our proposed countermeasure, called BALoo, achieves a 100% fault coverage in any fault model with a very small memory overhead (around 0.1% for a software AES implementation) and a limited hardware overhead. This hardware overhead is measured on a FPGA and with a hardware AES implementation which uses 16 different S-boxes. BALoo protects all of the 16 different S-boxes used in this implementation, with the addition of a 17th S-box to verify the integrity of one S-box when the 16 others can keep on encrypting, and cycling on the S-box checked.

REFERENCES

[1] J. Daemen and V. Rijmen, "The block cipher rijndael," in *Smart Card Research and Applications, This International Conference, CARDIS '98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings* (J. Quisquater and B. Schneier, eds.), vol. 1820 of *Lecture Notes in Computer Science*, pp. 277–284, Springer, 1998.

[2] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: an ultra-lightweight block cipher," in *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings* (P. Paillier and I. Verbauwhede, eds.), vol. 4727 of *Lecture Notes in Computer Science*, pp. 450–466, Springer, 2007.

[3] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings* (M. J. Wiener, ed.), vol. 1666 of *Lecture Notes in Computer Science*, pp. 388–397, Springer, 1999.

[4] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings* (B. S. K. Jr., ed.), vol. 1294 of *Lecture Notes in Computer Science*, pp. 513–525, Springer, 1997.

[5] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers* (B. S. K. Jr., Ç. K. Koç, and C. Paar, eds.), vol. 2523 of *Lecture Notes in Computer Science*, pp. 2–12, Springer, 2002.

[6] J. Balasch, B. Gierlichs, and I. Verbauwhede, "An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011* (L. Breveglieri, S. Guilley, I. Koren, D. Naccache, and J. Takahashi, eds.), pp. 105–114, IEEE Computer Society, 2011.

[7] B. Colombier, P. Grandamme, J. Vernay, É. Chanavat, L. Bossuet, L. de Laulanié, and B. Chassagne, "Multi-spot laser fault injection setup: New possibilities for fault injection attacks," in *Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers* (V. Grosso and T. Pöppelmann, eds.), vol. 13173 of *Lecture Notes in Computer Science*, pp. 151–166, Springer, 2021.

[8] F. Zhang, X. Lou, X. Zhao, S. Bhasin, W. He, R. Ding, S. Qureshi, and K. Ren, "Persistent fault analysis on block ciphers," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 3, pp. 150–172, 2018.

[9] F. Zhang, Y. Zhang, H. Jiang, X. Zhu, S. Bhasin, X. Zhao, Z. Liu, D. Gu, and K. Ren, "Persistent fault attack in practice," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, p. 172–195, Mar. 2020.

[10] Y. Cheng, C. Ou, F. Zhang, and S. Zheng, "Dlpfa: Deep learning based persistent fault analysis against block ciphers." Cryptology ePrint Archive, Paper 2023/021, 2023. https://eprint.iacr.org/2023/021.

[11] G. D. Natale, M. Doulcier, M. Flottes, and B. Rouzeyre, "Self-test techniques for crypto-devices," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 18, no. 2, pp. 329–333, 2010.

[12] *13th IEEE International On-Line Testing Symposium (IOLTS 2007), 8-11 July 2007, Heraklion, Crete, Greece*, IEEE Computer Society, 2007.

[13] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults (extended abstract)," in *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding* (W. Fumy, ed.), vol. 1233 of *Lecture Notes in Computer Science*, pp. 37–51, Springer, 1997.

[14] G. Piret and J. Quisquater, "A differential fault attack technique against SPN structures, with application to the AES and KHAZAD," in *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings* (C. D. Walter, Ç. K. Koç, and C. Paar, eds.), vol. 2779 of *Lecture Notes in Computer Science*, pp. 77–88, Springer, 2003.

[15] T. Fukunaga and J. Takahashi, "Practical fault attack on a cryptographic LSI with ISO/IEC 18033-3 block ciphers," in *Sixth International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2009, Lausanne, Switzerland, 6 September 2009* (L. Breveglieri, I. Koren, D. Naccache, E. Oswald, and J. Seifert, eds.), pp. 84–92, IEEE Computer Society, 2009.

[16] A. Baksi, S. Bhasin, J. Breier, D. Jap, and D. Saha, "A survey on fault attacks on symmetric key cryptosystems," *ACM Comput. Surv.*, vol. 55, no. 4, pp. 86:1–86:34, 2023.

[17] Y. Kim, R. Daly, J. S. Kim, C. Fallin, J. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pp. 361–372, IEEE Computer Society, 2014.

[18] National Institute of Standards and Technology, "Data encryption standard (des)." FIPS Publication 46-3, October 1999.

[19] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner, "Ciphers for MPC and FHE," in *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I* (E. Oswald and M. Fischlin, eds.), vol. 9056 of *Lecture Notes in Computer Science*, pp. 430–454, Springer, 2015.

[20] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçin, "PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract," in *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings* (X. Wang and K. Sako, eds.), vol. 7658 of *Lecture Notes in Computer Science*, pp. 208–225, Springer, 2012.

[21] S. Banik, S. K. Pandey, T. Peyrin, Y. Sasaki, S. M. Sim, and Y. Todo, "GIFT: A small present - towards reaching the limit of lightweight encryption," in *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings* (W. Fischer and N. Homma, eds.), vol. 10529 of *Lecture Notes in Computer Science*, pp. 321–345, Springer, 2017.

[22] J. Daemen and V. Rijmen, "Rijndael for AES," in *The Third Advanced Encryption Standard Candidate Conference, April 13-14, 2000, New York, New York, USA*, pp. 343–348, National Institute of Standards and Technology,, 2000.

[23] E. Andreeva, B. Bilgin, A. Bogdanov, A. Luykx, B. Mennink, N. Mouha, and K. Yasuda, "APE: authenticated permutation-based encryption for lightweight cryptography," *IACR Cryptol. ePrint Arch.*, p. 791, 2013.

[24] T. Beyne, Y. L. Chen, C. Dobraunig, and B. Mennink, "Multi-user security of the elephant v2 authenticated encryption mode," in *Selected Areas in Cryptography - 28th International Conference, SAC 2021, Virtual Event, September 29 - October 1, 2021, Revised Selected Papers* (R. AlTawy and A. Hülsing, eds.), vol. 13203 of *Lecture Notes in Computer Science*, pp. 155–178, Springer, 2021.

[25] T. Iwata, M. Khairallah, K. Minematsu, and T. Peyrin, "Duel of the titans: The romulus and remus families of lightweight AEAD algorithms," *IACR Cryptol. ePrint Arch.*, p. 992, 2019.