A Countermeasure Against Statistical Ineffective Fault Analysis

Jakub Breier, Mustafa Khairallah, Xiaolu Hou and Yang Liu

Abstract—Current state-of-the-art countermeasures against Fault Injection Attacks (FIA) provide good protection against analysis methods that require the faulty ciphertext to derive the secret information, such as Differential Fault Analysis (DFA) or collision fault analysis. However, recent progress in Ineffective Fault Analysis (IFA) and Statistical IFA (SIFA) constitutes a real threat against cryptographic implementations and moreover, it cannot be thwarted by standard FIA countermeasures that focus on detecting the change in the intermediate data.

In this paper, we present a novel method based on error correcting codes that protects implementations against SIFA. We design a set of universal error-correcting gates that can be used for implementing block ciphers. We analyze a hardware implementation of protected GIFT-64 and show that our method provides 100% protection against SIFA.

Index Terms—fault injection attacks, ineffective fault analysis, countermeasures, error-correcting codes, SIFA

1 Introduction

Fault injection attacks have become a powerful tool for implementation attacks against cryptography [1]. Many different fault analysis methods have been proposed up to date, so the attacker can choose based on her capabilities and the target cipher implementation. Recently, several powerful proposals utilizing Statistical Ineffective Fault Analysis (SIFA) have been introduced [2]–[4]. SIFA performs stuck-at-fault in the intermediate value during the execution of an algorithm. Based on observing whether there is a change in the ciphertext, the attacker can gain information of the attacked intermediate value and eventually get the secret key. The main strength of (S)IFA lies in the ineffectiveness of standard fault injection countermeasures to thwart the attack. Normally, the implementation-level prevention techniques aim at detecting value changes in the computation to raise an alarm. This can be done by using a redundant computation circuits, various code-based techniques, of infective countermeasures that "infect" the entire cipher state after fault to hide the information leakage from the fault. However, in case of (S)IFA, the attacker only exploits the knowledge whether the computation was correct or not – she does not need the output value. And therefore, raising an alarm from the countermeasure is enough for her to determine the value she wants to discover.

As mentioned in [4], to prevent from these attacks, it is advised to utilize sensitive-enough physical sensors, e.g. ring oscillator based [5], that detect the physical disturbances of the circuit no matter whether the value has changed or not. However, since the sensor is not a part of the cryptographic circuit, there is always a possibility to unplug it or make it ineffective. For that reason, it is better to have multiple layers of protection, both on the circuit and the implementation level.

In this paper, we propose a countermeasure against (S)IFA that utilizes error-correcting codes. The main idea is to prevent the attacker from knowing whether the fault occured or not. We provide a set of universal error-correcting gates that can be used for implementing linear and non-linear operations of block ciphers. Our results show 100% fault coverage against the considered attacker model.

The rest of this paper is organized as follows. Section 2 presents the related work in this field. Section 3 provides the theoretical background for our method. Application of the method to ciphers is described in Section 4, followed by evaluation in Section 5. Discussion is provided in Section 6, and finally, Section 7 concludes this work.

- *Jakub Breier and Yang Liu are with School of Computer Science and Engineering, NTU Singapore. E-mail: jbreier@jbreier.com, yangliu@ntu.edu.sg*
- *Mustafa Khairallah is with School of Physical and Mathematical Sciences, NTU Singapore. E-mail: mustafam001@e.ntu.edu.sg*
- *Xiaolu Hou is with Acronis, Singapore.E-mail: ho0001lu@e.ntu.edu.sg*

2 Related Work

2.1 Ineffective Fault Attacks

Ineffective Fault Attacks (IFA) were originally proposed by Clavier in 2007 [6]. This fault analysis method exploits type of fault which changes a variable to a particular value – and in case the variable already holds this value, no change can be observed. As an example, let us assume we have a one bit variable x which is secret. This variable is being processed in a device we have previously profiled and we can assume with a high probability that we are capable of changing a certain bit in a data unit to "`1`" with a well-aimed fault injection. Now, we process x with our device, launch the fault injection, and observe the output. In case we see a difference at the output, it means the original value of x was "`0`." Otherwise, if there is no observable change at the output, we can assume with a high probability that the original value of x was "`1`."

2.2 Statistical Fault Attacks

Statistical Fault Attacks (SFA), introduced by Fuhr et al. [7] exploit the situation when the attacker is able to change an intermediate value to a biased value by injecting a fault. Three fault models were presented: 1) stuck-at-0; 2) stuck-at-0 with probability of 0.5 or logical AND with random uniform value with probability 0.5; 3) logical AND with random uniform value. The authors showed how the method works on AES, where the recovery of 4 bytes of the secret key took between 6-80 faulty ciphertexts, depending on the used model.

2.3 Statistical Ineffective Fault Attacks

Statistical Ineffective Fault Attacks (SIFA) [4] are an intersection between IFA and SFA.

In [2]–[4] authors experimentally used ineffective faults to break cryptosystems without the need of the deeper analysis of the cipher, which is normally necessary for using other methods, such as Differential Fault Analysis.

2.4 Fault Attack Countermeasures

Fault attack countermeasures mostly focus on preventing fault models that aim at altering the values during the execution, e.g. differential fault analysis. They either try to detect the change or prevent the attacker from getting information from the faulty output. On the other hand, in case of (S)IFA, the information whether there was a change during the computation is sufficient to get some information on the secret key. Currently, only device-level countermeasures can be used for preventing (S)IFA, such as sensors or special packages. However, the cipher implementer has normally no control over these countermeasures and a specialized device needs to be used to provide them. For further overview of different countermeasures, we refer the interested reader to [8].

3 Methods

3.1 Coding theory background

A *binary code*, which we denote by $\mathcal{C}$, is a subset of $\mathbb{F}_2^n$, the $n-$dimensional vector space over $\mathbb{F}_2$, where n is called the *length* of the code $\mathcal{C}$. Each element $\boldsymbol{c} \in \mathcal{C}$ is called a *codeword* of $\mathcal{C}$ and each element $\boldsymbol{x} \in \mathbb{F}_2^n$ is called a *word* [9, p.6]. Take two words $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_2^n$, the *Hamming distance* between $\boldsymbol{x}$ and $\boldsymbol{y}$, denoted by $\text{dis}\,(\boldsymbol{x}, \boldsymbol{y})$, is defined to be the number of places at which $\boldsymbol{x}$ and $\boldsymbol{y}$ differ [9, p.9]. More precisely, if $\boldsymbol{x} = x_1 x_2 \dots x_n$ and $\boldsymbol{y} = y_1 y_2 \dots y_n$, then

$$\text{dis}\,(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{n} \text{dis}\,(x_i, y_i)\,,$$

where x_i and y_i are treated as binary words of length 1 and hence

$$\text{dis}\,(x_i, y_i) = \begin{cases} 1 & \text{if } x_i \neq y_i \\ 0 & \text{if } x_i = y_i \end{cases}.$$

Furthermore, for a word $\boldsymbol{x} \in \mathbb{F}_2^n$, the *Hamming weight* of $\boldsymbol{x}$, $\text{HW}(\boldsymbol{x}) := \text{dis}\,(\boldsymbol{x}, \boldsymbol{0})$ [9, p.46]. For a binary code $\mathcal{C}$, the *(minimum) distance* of $\mathcal{C}$, denoted by $\text{dis}\,(\mathcal{C})$, is [9, p.11]

$$\text{dis}\,(\mathcal{C}) = \min\{\text{dis}\,(\boldsymbol{c}, \boldsymbol{c}') : \boldsymbol{c}, \boldsymbol{c}' \in \mathcal{C}, \boldsymbol{c} \neq \boldsymbol{c}'\}.$$

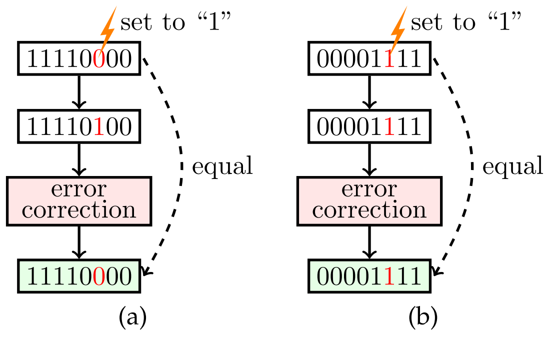


Fig. 1. Error correction against IFA when attacking the same bit position in two different codewords – 11110000 and 00001111. In case of set to "1" fault: (a) original value changes after fault and later is corrected back, (b) original value does not change after fault.

Definition 1. [9, p.39] In case $\mathcal{C}$ is a subspace of $\mathbb{F}_2^n$, $\mathcal{C}$ is called a linear code. A linear code with dimension k, length n and minimum distance d is called an $[n, k, d]$-binary code.

Definition 2. [9, p.13] Let v be a positive integer. $\mathcal{C}$ is *$v-$error-correcting* if minimum distance decoding with incomplete decoding rule is applied, v or fewer errors can be corrected.

Remark 1. $\mathcal{C}$ is $v-$error correcting if and only if $\text{dis}\,(C) \geq 2v + 1$ [9, p.13].

Considering the (S)IFA, as we are mostly dealing with 1- and 2-bit faults, the distance for the used codes should be at least 3 and 5, respectively.

3.2 Our Countermeasure Idea

Normally, it would be of no use for the attacker to affect high number of bits at the same time, since the probability of the original variable to have the exact value that is being injected gets lower with each stuck-at faulty bit. Therefore, it is safe to assume that practical attacks would aim at changing 1 or at most 2 bits of the variable.

The main idea of our countermeasure is to make the attacker unsure whether there was a change to the variable or not. For this purpose, we propose usage of error correcting codes that were thoroughly evaluated against fault injection in [10]. The working principle of the (S)IFA protection is depicted in Figure 1. The error correction ensures that in case the fault was injected, the variable will regain its original value. Therefore, the attacker is not able to distinguish whether the change due to the fault occured or not.

The crucial parameter of the code in this case is the distance d, which specifies how many bits we allow the attacker to target. To allow v bits to be targeted, we need distance to be at least $2v + 1$ to be able to correct the codeword to the original value [9].

3.3 Implementation Options

In general, there are two ways to implement encoding based countermeasures: either in a table look-up form where the address navigation is done by using codewords [10], [11]; or by computing the operations on the codewords directly, while performing integrity checks after predefined number

TABLE 1
Truth table for error-correcting AND gate, using a linear code with two codewords: $0 \mapsto 000$; $1 \mapsto 111$.

&	000	001	010	011	100	101	110	111
000	000	000	000	000	000	000	000	000
001	000	000	000	000	000	000	000	000
010	000	000	000	000	000	000	000	000
011	000	000	000	111	000	111	111	111
100	000	000	000	000	000	000	000	000
101	000	000	000	111	000	111	111	111
110	000	000	000	111	000	111	111	111
111	000	000	000	111	000	111	111	111

of operations [12]. In our work we focus on computational approach due to the fact that it is faster and has lower memory consumption.

3.4 Example

Let us consider a simple binary AND operation, taking two single bit values as inputs and one single bit value as output. To correct one bit, distance between codewords needs to be at least 3. We can construct a truth table for an implementation of error-correcting AND gate (details of such gate are explained in the next section), stated in Table 1, where the encoding is as follows: $0 \mapsto 000$ and $1 \mapsto 111$. The first column represents the input of the first operand and the first row is the second operand. We can see that the table entries contain only two values, depending on the distance between the input word and the two codewords. That also means even in the case both input values get faulted, the correction will still work. It is important to note that this particular example might leak side-channel information, since in case the Hamming weight (HW) of the word is ≤ 1, it corrects to 000, and if the HW is ≥ 2, it corrects to 111.

4 Application to Ciphers

In this section we describe a low-cost hardware implementation of our countermeasure. We use the lightweight SPN ciphers Skinny-128-128 [13] and GIFT-64-128 [14] as examples.

4.1 Single-Bit Faults

First, we consider the [3,1,3]-binary code (3-Repetition Hamming code) considered in the previous section. To reduce the cost, we use the fact that this code is a linear code. Hence, XOR/XNOR/Inversion operations can be performed on the codewords simply by applying the corresponding Boolean operations in a bitwise manner. This approach reduces the cost of implementing linear operations of the ciphers. However, by implementing these operations in that manner, the faults will propagate linearly through the gates. We assume that the correction will only be performed in the non-linear (Sbox) Layer. We study a fault model where each codeword has at most 1 faulty bit at the input of each round.

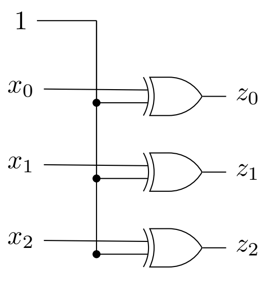
We define a set of gates that are used to operate on the codewords:

$$
\begin{aligned}
RNOT: \{z_2, z_1, z_0\} &= \{x_2 \oplus 1, x_1 \oplus 1, x_0 \oplus 1\} \\
RXOR: \{z_2, z_1, z_0\} &= \{x_2 \oplus y_2, x_1 \oplus y_1, x_0 \oplus y_0\} \\
CAND: \{z_2, z_1, z_0\} &= \{(x_2x_1 \vee x_1x_0 \vee x_0x_2) \\
&\quad \wedge(y_2y_1 \vee y_1y_0 \vee y_0y_2), (x_2x_1 \vee x_1x_0 \vee x_0x_2) \\
&\quad \wedge(y_2y_1 \vee y_1y_0 \vee y_0y_2), (x_2x_1 \vee x_1x_0 \vee x_0x_2) \\
&\quad \wedge(y_2y_1 \vee y_1y_0 \vee y_0y_2)\} \\
COR: \{z_2, z_1, z_0\} &= \{(x_2x_1 \vee x_1x_0 \vee x_0x_2) \\
&\quad \vee(y_2y_1 \vee y_1y_0 \vee y_0y_2), (x_2x_1 \vee x_1x_0 \vee x_0x_2) \\
&\quad \vee(y_2y_1 \vee y_1y_0 \vee y_0y_2), (x_2x_1 \vee x_1x_0 \vee x_0x_2) \\
&\quad \vee(y_2y_1 \vee y_1y_0 \vee y_0y_2)\}
\end{aligned}
$$

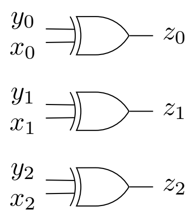
The circuit diagrams are depicted in Figure 2. As mentioned earlier, both the NOT and XOR gates have no effect on the fault value. The AND/OR gates have to be implemented at least 3 times independently to make sure that if the adversary injects a fault in one of the instances, it does not propagate to the other two bits.

Given this set of gates, we study the implementation of the GIFT cipher's Sbox, proposed in [14]. We chose the Software optimized implementation of the Sbox as a reference as it has lower number of NOT/XNOR/NAND/NOR gates, making it more suitable for our gate set. This implementation requires `5X+1N+3A+1R`, where `X,N,A,R` stand for XOR, NOT, AND and OR gates, respectively. Overall, one round of GIFT-64 needs 16 Sboxes and 32 XORs for key addition, `112X+16N+48A+16R`. Using our gate set instead, we can implement the GIFT-64 round using `336X+48N+1728A+624R`. In order to estimate the overall cost compared to the unprotected implementation of GIFT-64, we estimate `X=2.25GE,N=0.7GE,A=R=1.2GE`. We need also to take into consideration that we need to store the state, which requires 64 Flip-Flops for the unprotected case and 192 Flip Flops for the protected case. Hence, the countermeasure requires 6x area overhead for ASIC. For FPGA, we can reduce such cost to only 3x, since the CAND and COR gates can take advantage of the 6-to-1 Look Up Table structures in the modern FPGA, such that each of them can be implemented using only 3 LUTs.

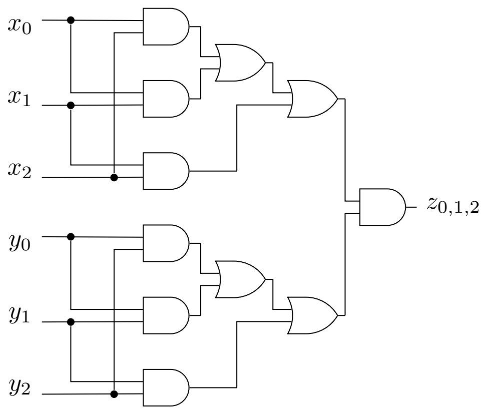
Similarly, we study the implementation of the members of the Skinny family of tweakable block ciphers. It uses two different Sboxes, one is a 4-bit Sbox and the other is an 8-bit Sbox. The 4-bit Sbox requires `4X+4R+4N`, while the 8 bit Sbox requires `8X+8R+8N`. Moreover, Skinny also uses two different diffusion layers, depending on the block size. The Round Constants require `6X+1N`, while the Key addition requires either `32X` or `64X`. Finally, the MixColumn operation requires either `64X` or `128X`. Overall, if the block size is 64 bits, one round requires `166X+65N+64R`, while if the block size is 128 bits, it requires `326X+129N+128R`. For the protected case, we need `498X+195N+2112R` and `978X+387N+4224R`, respectively. Hence, we estimate the over head for ASIC to be around 5.6x, slightly lower than the case of GIFT-64, which is because Skinny has higher `X/(A+R)` ratio, i.e. the ration between linear components and non-linear components is higher, due to the MixColumn operation, as opposed to the bit permutation in case of GIFT.



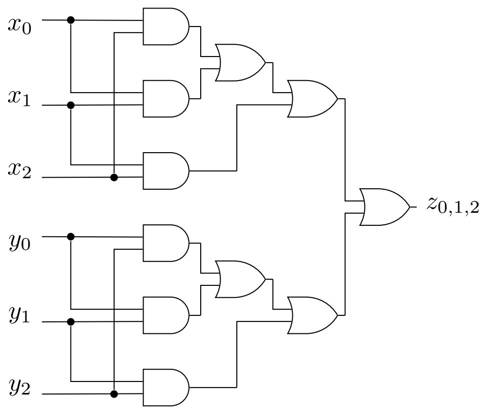
(a) $RNOT$



(b) $RXOR$



(c) $CAND$

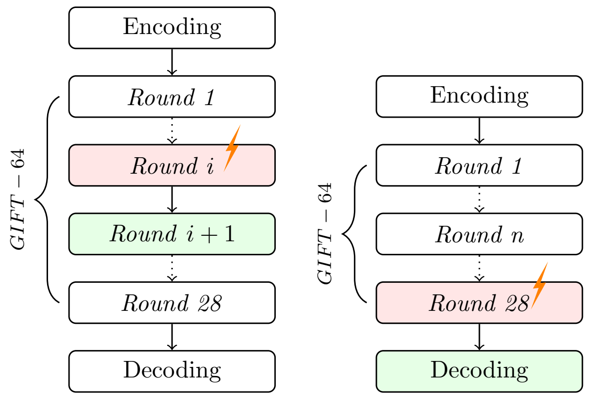


(d) COR

Fig. 2. Adjusted gates to operate on codewords (a,b), and error-correcting gates (c,d).

4.2 Double-Bit Faults

Our countermeasure can be extended to double-bit faults by using [5,1,5]-binary code instead. In this case the cost for the RNOT and RXOR gates is multiplied by 5, while CAND and COR gates can be implemented using `123A+54R` and `120A+57R` respectively. Hence, the overall cost of the implementation of GIFT-64 is `560X+80N+7824A+3504R`, in addition to 320 Flip-Flops for the state storage. Our estimate that the overhead will be around 25× for ASIC. While the cost can seem to be too expensive, given that SIFA is one of the strongest attacks on cipher implementations, we believe the cost can be justified for sensitive applications that require high security.



(a) Scenario 1 (b) Scenario 2

Fig. 3. Two evaluation scenarios: (a) fault is injected in the middle round and (b) fault is injected in the last round. Round where the fault is injected is indicated by red color, while the round/block where the fault is corrected is indicated by green color.

TABLE 2
Results on simulating ineffective fault analysis against the gates proposed in Section 4. Description of each scenario is given in Figure 3.

		Correct outputs		Faulty outputs	
Gate	#	Scenario 1	Scenario 2	Scenario 1	Scenario 2
AND	660	100%	100%	0%	0%
OR	396	100%	100%	0%	0%
XOR	276	100%	100%	0%	0%
$CAND$	28	100%	100%	0%	0%
COR	4	100%	100%	0%	0%
$RXOR$	30	100%	100%	0%	0%
$RNOT$	4	100%	100%	0%	0%

5 Evaluation

We have analyzed the ineffective fault analysis conditions of the GIFT Sbox implementation proposed in Section 4.1. We have constructed a digital logic circuit analysis tool that loops through all the possible inputs, injects a stuck-at fault at every single gate in the circuit, and checks the output for errors. We have utilized a single fault adversarial model which is the most common model used in the literature. The assumption on correcting capabilities of our proposal is that in case there is a fault that propagates through the Sbox layer, it will either be corrected at the following Sbox layer (in case of inner rounds) or at the final decoding stage. Therefore, to simulate this behavior, we have analyzed two different scenarios:

1) first/middle round fault, followed by another round;
2) last round fault, followed by an error correcting decoder.

These two scenarios are depicted in Figure 3.

As it would be computationally impractical to analyze the full GIFT state, we took advantage the properties of the permutation layer that divides the state into four 16-bit sub-states. We analyzed one 16-bit sub-state, which shows the behavior of the entire state. That means, for each gate we analyzed 2^{16} inputs for stuck-at-0 and stuck-at-1 fault. We tried to fault every bit of each input. That means, for example in case of AND gates, the total number of experiments was 660 gates $\times 2^{16}$ input values $\times 2$ fault models $\times 2$ inputs $= 173,015,040$, for each scenario.

The results are stated in Table 2. As can be seen, the circuit analysis utilizes the error-correcting properties of the linear code as described in Section 3. Every ineffective fault was captured either in the subsequent round in case of Scenario 1, or in the final decoder in case of Scenario 2. This shows that the implementation following our proposal is robust against (S)IFA that utilizes 1-bit faults.

6 Discussion

6.1 Hamming-(7,4) Code

Cheaper codes can be used to get cheaper implementations, such as the Hamming-(7,4) code, which encodes 4-bit nibbles into 7-bit codewords. This code can correct all single bit faults, as well. In this scenario, the 4-bit Sbox can be implemented as an equivalent 7-bit look-up table. Since the overall encoded state is smaller, the cost of the linear parts of the circuit is lower. However, the attacker can have more targeted attacks that aim at internal states of the Sbox. In such case the analysis of how such faults will propagate to the output of the Sbox is not clear. In other words, the attacker may be able to inject a fault in an internal gate that generates codewords with multiple faulty bits.

6.2 Faulting Encoding/Decoding Circuit

As one may have noticed, we did not consider faults into the encoding/decoding circuits that surround the cipher implementation. We will explain this in the following.

In case of the *encoding* circuit, if there is a fault in the input, it will change the value of the codeword to another codeword, effectively changing the plaintext input to the cipher. That means, the attacker would get the same situation as in case of differential cryptanalysis – she would have to cryptanalyze the entire cipher. If there is a fault in the output of the encoding circuit, the error-correcting gates in the first round will correct it the same way as in the middle of the cipher.

In case of the *decoding circuit*, the attacker would be effectively faulting the resulting ciphertext. That would not give her any additional information on either the plaintext nor the secret key.

6.3 Software Implementation

To get the same level of protection in software, one would have to use a table look-up based implementation. Similar to the implementation proposed in Section 4, to achieve the same level of protection, it would be necessary to implement the non-linear operations by error-correcting tables. While the speed overhead of such implementation is reasonable – 82.5% for PRESENT-80 implementation from [15], the memory requirements are high. For example, in case of 8-bit architectures, one binary look-up table takes 65 kB [15].

6.4 Differential Fault Analysis

A thorough analysis of error-correcting encoding scheme w.r.t. DFA was given in [10]. The work shows that it is necessary to match the assumed attacker strength with the used code. More specifically, the code distance always needs to be twice as long as the attacker's capabilities to flip certain number of bits, otherwise there is a possibility to correct one codeword into another by flipping enough bits.

7 Conclusion

In this paper we have proposed a novel method to protect cipher implementations against ineffective fault analysis. Our work is based on error-correcting codes that can be efficiently implemented in the form of error-correcting hardware gates. Attacker capabilities can be matched by the choice of proper code, e.g. for 1-bit fault models, at least a 3-bit code needs to be used, while for 2-bit fault model, at least a 5-bit code has to be used. We have evaluated a hardware implementation of protected GIFT-64 and our results show 100% fault coverage.

In the future, it would be interesting to extend the protection against side-channel attacks by utilizing adequate code, as was shown in [10].

References

[1] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *CRYPTO'97*. Springer, 1997, pp. 37–51.

[2] C. Dobraunig, M. Eichlseder, H. Gross, S. Mangard, F. Mendel, and R. Primas, "Statistical ineffective fault attacks on masked aes with fault countermeasures," in *ASIACRYPT'18*. Springer, 2018, pp. 315–342.

[3] C. Dobraunig, S. Mangard, F. Mendel, and R. Primas, "Fault attacks on nonce-based authenticated encryption: Application to keyak and ketje," in *International Conference on Selected Areas in Cryptography*. Springer, 2018, pp. 257–277.

[4] C. Dobraunig, M. Eichlseder, T. Korak, S. Mangard, F. Mendel, and R. Primas, "Sifa: Exploiting ineffective fault inductions on symmetric cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 547–572, 2018.

[5] W. He, J. Breier, S. Bhasin, N. Miura, and M. Nagata, "Ring oscillator under laser: Potential of pll-based countermeasure against laser fault injection," in *FDTC'16*. IEEE, 2016, pp. 102–113.

[6] C. Clavier, "Secret external encodings do not prevent transient fault analysis," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2007, pp. 181–194.

[7] T. Fuhr, E. Jaulmes, V. Lomné, and A. Thillard, "Fault attacks on aes with faulty ciphertexts only," in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2013, pp. 108–118.

[8] J. Breier and X. Hou, "Introduction to fault analysis in cryptography," in *Automated Methods in Cryptographic Fault Analysis*. Springer, 2019, pp. 1–10.

[9] S. Ling and C. Xing, *Coding theory: a first course*. Cambridge University Press, 2004.

[10] J. Breier and X. Hou, "Feeding two cats with one bowl: On designing a fault and side-channel resistant software encoding scheme," in *CT-RSA'17*. Springer, 2017, pp. 77–94.

[11] P. Rauzy, S. Guilley, and Z. Najm, "Formally proved security of assembly code against power analysis," *Journal of Cryptographic Engineering*, vol. 6, no. 3, pp. 201–216, 2016.

[12] T. Schneider, A. Moradi, and T. Güneysu, "Parti–towards combined hardware countermeasures against side-channel and fault-injection attacks," in *CRYPTO'16*. Springer, 2016, pp. 302–332.

[13] C. Beierle, J. Jean, S. Kölbl, G. Leander, A. Moradi, T. Peyrin, Y. Sasaki, P. Sasdrich, and S. M. Sim, "The skinny family of block ciphers and its low-latency variant mantis," in *CRYPTO'16*. Springer, 2016, pp. 123–153.

[14] S. Banik, S. K. Pandey, T. Peyrin, Y. Sasaki, S. M. Sim, and Y. Todo, "Gift: a small present," in *CHES'17*. Springer, 2017, pp. 321–345.

[15] J. Breier, X. Hou, and Y. Liu, "On evaluating fault resilient encoding schemes in software," *IEEE Transactions on Dependable and Secure Computing*, 2019.