Journal of Cryptographic Engineering (2020) 10:355–374
https://doi.org/10.1007/s13389-020-00224-9

REGULAR PAPER

To infect or not to infect: a critical analysis of infective countermeasures in fault attacks

Anubhab Baksi[1] · Dhiman Saha[2] · Sumanta Sarkar[3]

Received: 27 June 2019 / Accepted: 22 March 2020 / Published online: 23 May 2020

Abstract
As fault-based cryptanalysis is becoming more and more of a practical threat, it is imperative to make efforts to devise suitable countermeasures. In this regard, the so-called "infective countermeasures" have garnered particular attention from the community due to its ability in inhibiting differential fault attacks without explicitly detecting the fault. We observe that despite being adopted over a decade ago, a systematic study of infective countermeasures is missing from the literature. Moreover, there seems to be a lack of proper security analysis of the schemes proposed, as quite a few of them have been broken promptly. Our first contribution comes in the form of a generalization of infective schemes which aids us with a better insight into the vulnerabilities, scopes for cost reduction and possible improvements. This way, we are able to propose lightweight alternatives of two existing schemes. Further, we analyze shortcomings of LatinCrypt'12 and CHES'14 schemes and propose a simple patch for the former.

Keywords Fault countermeasure · Infection · Differential fault analysis

1 Introduction

Fault attacks are becoming a real threat particularly to small scale devices performing a cryptographic operation. This type of attack forces a certain device to work under sub-optimal condition resulting in erroneous calculations, which is then exploited. *Differential Fault Analysis* or *Differential Fault Attack* (DFA) [11], one type of fault attack, is predominantly used against symmetric key ciphers. Most, if not all, ciphers which are considered secure against classical attacks are shown to have severe weaknesses against DFA. This attack works by injecting a difference (*fault*) during the cipher execution, which normally results in flipping one or more bits of a register. Then, after analyzing the output difference of the non-faulty and the faulty outputs of the cipher computations; the attacker, Eve, is often able to deduce information on the secret key.

Success of DFA also gave rise to a series of works attempting to protect ciphers from this attack. Various types of countermeasures are proposed in the literature. All of these countermeasures rely on full or partial redundancy either in device, cipher implementation or the protocol. Broadly, the state-of-the-art schemes can be classified into three categories:

(i) Using a separate, dedicated device. They can be either *active* which uses a sensor to detect any potential fault, such as [24]; or *passive*, where a shield to block external interference [4] is used.
(ii) Using redundancy in computation. These types of countermeasures commonly duplicate (fully/ partially) the circuit, followed by a certain procedure which dictates what to do in case a fault is sensed directly or indirectly.
(iii) Using protocol level technique. Here, the underlying protocol ensures that the conditions required for a successful fault happens with low probability [2,18].

Our interest lies in the second category of fault protection (further elaborated in Sect. 2). In particular, we focus on

✉ Anubhab Baksi
anubhab001@e.ntu.edu.sg

Dhiman Saha
dhiman@iitbhilai.ac.in

Sumanta Sarkar
sumanta.sarkar1@tcs.com

1 School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore

2 Department of Electrical Engineering and Computer Science, IIT Bhilai, Raipur, India

3 TCS Innovation Labs, Hyderabad, India

the so-called *infective countermeasures* (or, *infection-based countermeasures*). These countermeasures work by doing two computations of the same cipher (which we call, *actual* and *redundant*); thereafter computing the difference (XOR) between them (we denote it by Δ). This difference serves the purpose of implicitly detecting the fault—a nonzero difference implies a fault injection. This difference is further processed to produce a random vector in an unintelligible manner. This random vector is then used to corrupt (*infect*) output from the actual computation, which is then made available to Eve. Hence, the attacker gets the original (non-faulty) output if no fault is sensed by the countermeasure; or gets a random output otherwise—thus she has no meaningful information on Δ. This makes DFA impossible to mount as it requires the knowledge of Δ.

This idea of infection was proposed to counter the shortcomings of the previously proposed *detective countermeasures* (also referred to as *detection-based countermeasures*, see Sect. 2.2 for more details). Incidentally however, most of the infective countermeasures proposed in the literature have been broken quite soon (it is even stated in [7] that, “it is very difficult to design a secure infective countermeasure”). In fact, after more than a decade of their first introduction, we only have a handful of the schemes which are not broken—and yet, they generally have heavy implementation cost. Our observation is that almost all of the schemes proposed in this context are ad hoc solutions, instead of utilizing already established design standards. This motivates us to look further down into the topic to gain better understanding of the solution that the infective countermeasures intend to provide. To do a more systematic and complete study on the designs proposed, we first categorize them. Following this, we revisit the design choices to explore vulnerabilities and/or improvements.

Our contributions

Here, we mention the key contributions.

– We categorize the existing countermeasures into two types, so that a more systematic and comprehensive study is possible. This categorization is done from a cipher designer’s point of view, which is missing in the existing literature. For convenience, we call these *Type I* (Sect. 2.4.1) and *Type II* (Sect. 2.4.2) countermeasures. The basic difference between them is that Type I schemes allow the full computations of the actual and redundant ciphers to run, then computes Δ; in contrast, Type II schemes compute the difference after each round. Type I countermeasures are further divided into two sub-categories based on existing literature; *multiplication based* (Sect. 3.1) and *derivative based* (Sect. 3.2). Type II countermeasures are also referred to as *cipher level* countermeasures (described in Sect. 4).
– Among the schemes in the Type I category, we find two schemes which are not broken. Although we do not find any attack to those schemes, we remark that both of them incur a substantial hardware or software overhead. In this regard, we propose two hardware efficient alternatives to the countermeasure described in [33] (Sect. 3.3.1). Also, we propose software friendly options for the scheme proposed in [22], (Sect. 3.3.3) based on existing ciphers.
– For the Type II schemes, we show (in Sect. 4.1) that CHES’14 infective countermeasure proposed in [44] is not an infective countermeasure at all; thereby refuting its security claim and showing its weakness against DFA. This also refutes the security claim made by the modified infective countermeasure in [36]. The basic principle of this paper is the same as that of [44]; certain modifications are done on [44] to make it better resilient against instruction skip attacks on a microcontroller.[1]
– Interestingly, though, this CHES’14 scheme is proposed as an improvement on the first cipher level infective countermeasure presented in [23] (LatinCrypt’12), which is broken in [7,44]. We propose a simple patch (in Sect. 4.2) on the [23] scheme that resists the attacks presented in [7,44].

In context of the security proofs of our countermeasures/patch, we would like to note that no separate security proof would be required. Since we reuse the existing cipher design paradigm, the security proof is the same as the cipher itself.

As for the practical evaluation of our schemes, we note the following. Our proposed countermeasures are designed at the algorithmic level. Hence, anyone with the access to an experimental setup can validate. Any hidden vulnerability in the implementation which is transparent at the algorithmic level is not within the scope.

2 Background

2.1 Context of differential fault analysis

As mentioned earlier, DFA works by injecting a difference (fault) during cipher computation. This fault injection can be done by various methods; clock/power glitch, LASER shot, to name a few. More information on such fault injecting equipment can be found, for example, in [5]. This fault, in

[1] An instruction skip is considered a separate (non-DFA) type of fault.

effect, results in a few bit(s) flip of the cipher at a round. Normally, attacker can choose the round, but unable to precisely target the words; as a result, she does not know the actual fault value (the value which is effectively XORed with the cipher state). This is a very commonly employed model and termed as *random fault model* [38,43].

Now, flipping one/few bit(s) of a register works as a simplified version of the classical *Differential Attack* (DA) [10]. In DA, the difference is inserted through the chosen plaintexts (hence, works at the beginning of the cipher execution). In contrast, in DFA, the difference can be inserted at any point of time during execution (normally it is inserted near the end of the cipher execution). Now, since the difference passes through a small number of rounds of the cipher (can be even one round), the resistance against differential attack is not very strong. At the end of the cipher execution, when attacker gets the corresponding output difference, an analysis similar to DA may reveal secret information.

Although DA and DFA work very similarly, one may notice that the same idea used to thwart DA cannot be potentially used to thwart DFA. The DA protection arises from many iterations of the cipher, which is meaningless in DFA, as attacker is able to attack any round near the end. Hence, the solutions proposed to protect against DFA require certain assumption on the underlying device/communication protocol, rather than completely relying on the cipher description (which is the case for DA).

We assume that Eve can inject faults temporarily (the fault values are not permanent—the device goes back to its normal situation once the source of fault is revoked). This contrasts to the *stuck-at/hard* fault model where particular bit(s) are permanently stuck with the fault value (one may refer to [14] for an example). The stuck-at model assumes more control for the attacker and is outside the scope of DFA. Like this, modification of operation is also a strong attacker model (such as an instruction skip on a software implementation, as in [32]), and also not a DFA. In short, DFA assumes the transient fault model where only the operands are subject to alteration.

2.2 Early countermeasures: detection based

One of the earliest countermeasures proposed against DFA is known as detective countermeasure (or detection-based countermeasure). Conceptually, it checks whether any fault is injected by explicitly checking whether $\Delta = 0$. If a fault is detected (i.e., $\Delta \neq 0$), it blocks the device from producing the faulty output (either the output from the cipher is suppressed, or an invalid signal ($\perp$) is generated, or a random output is generated). This stops the attacker from getting any meaningful information regarding the faulty state, which makes DFA impossible. Ideas in this direction are commonly generated from coding & information theory, such as linear parity [27], nonlinear (n, k) codes [31], etc.

2.3 Evolution of infective countermeasures

It is commonly argued that the concept of explicit equality checking in detection-based countermeasures is subject to bypassing that step (e.g., [29] or [45, Section 3]). Generally, such comparisons rely only on one bit (like the zero flag in a microcontroller); any attack that is able to flip this bit renders the whole countermeasure useless.[2] Although injecting two faults in one invocation of cipher is not a model commonly used; it is commented in [35] that, one-bit flip is an incidence which may happen "by chance." If the attacker is able to flip this particular bit, even with low probability, the entire security provided by the detection-based countermeasure would be nullified. In general, the motivation for the improved version of the detection-based countermeasures came from removing the single point of failure. It is stated in [45, Section 3]: "... the usage of a decision procedure violates the guarantee of developing a highly reliable hardware fault immune cryptographic protocol." More discussion on the philosophy of infection can be found at [35, Chapter 3.2.1].

The idea of *infection*, which avoids the dependency of security of one bit, was first proposed in the context of public key cryptography [45]. This proposal was claimed to be weak (see [46]), though this idea was followed soon after in the public key cryptography such as [13,25].

The concept was adopted in symmetric key setting [20,21, 23,26,33,36,44]. The infective countermeasures proposed in [20,26] did not involve randomness and were attacked in [33]. The authors in [33] also speculate, randomness may be required in such countermeasures, although they do not present any formal proof. Anyway, all infective countermeasures proposed thereafter adopt this idea and use randomness. Still, most schemes proposed in the literature are broken soon. In fact, in our literature survey we observe that basically 3 different schemes are proposed which are not considered broken by DFA.

2.4 Notations and terminologies

Before proceeding further, we define terms and notations that we use the subsequent parts.

– *Actual and redundant computations* As mentioned earlier, infective countermeasures require two computations of the same cipher. These two are referred to as actual and redundant computations, and symbolically denoted as $C = E_K^1(P)$ and $C' = E_K^2(P)$, respectively (E is the

[2] This case is different from the fault injection flipping only one bit.

underlying cipher parameterized by the secret key K with input P, and the superscript denotes the two separate executions). Output from the actual cipher is later infected and made available to the attacker. The notations, C and C', are used in this work to denote the output from the actual and redundant computations, respectively. When Δ is computed after the full iteration of both the actual and redundant computations are finished; i.e., in Type I countermeasures; $\Delta = C \oplus C'$.

Also, we assume that the attacker is able to exactly repeat the same fault, in same location and during the same round on one particular device; as many times she wants (as long as it is practical). Thus, we give the attacker to exactly repeat the same fault over temporal domain, and to make Δ constant.[3]

- η We denote by η the total number of rounds of the cipher; e.g, for `AES`, $\eta = 11$ (counting the initial `AddRoundKey` as a separate round).
- n We use n to denote the block size of the cipher; e.g., $n = 128$ for `AES`.
- $\texttt{RoundFunction}_j(\cdot)$, $j = 1, \ldots, \eta$. We use $\texttt{RoundFunction}_j(p)$ to denote the jth round function of the underlying cipher with the corresponding input p, $j \in \{1, \ldots, \eta\}$. Note that, it does not involve the round key insertion.
  For example, in case of `AES`; $\texttt{RoundFunction}_1(p) = p$; $\texttt{RoundFunction}_j(p) = \texttt{MixColumns ShiftRows(SubBytes(p)))}$ for $j = 2, \ldots, 10$; and $\texttt{RoundFunction}_{11}(p)$=`ShiftRows(SubBytes(p))`. Basically, when the jth round key k_j is inserted with $\texttt{RoundFunction}_j(\cdot)$, it gives the actual jth round of the cipher. In other words, $\texttt{RoundFunction}_j(\cdot)$ is the internal diffusion within the state of a cipher (not involving the round key).
- $\texttt{RoundFunction}_0(\cdot)$ By $\texttt{RoundFunction}_0(p)$; we denote the standard (most frequent) round of a cipher, with input p (without involving the round key). In case of `AES`, $\texttt{RoundFunction}_0(p) \equiv \texttt{RoundFunction}_2(p)$.
- ξ Generally, cipher designers are conservative in the sense that the number of rounds (η) in a cipher is kept more than that of what would be required to reach a practical security. Often, we may not need full η rounds of iteration to ensure a practical security. Instead, with less than η iterations of $\texttt{RoundFunction}_0(\cdot)$, together with corresponding round key insertions; one can achieve good resistance against classical cryptanalysis techniques such as differential or linear attacks. We denote, by ξ, the minimum number of iterations required for a cipher to offer a practical security.
  For an example, consider a reduced version of `AES` with 4 $\texttt{RoundFunction}_0(\cdot)$ rounds, together with corresponding `AddRoundKeys`. It can be proven to have no differential path with probability better than 2^{-113} [28]. Hence, we take $\xi = 4$ for `AES`.
- R. By R, we denote an n-bit random vector. When more than one random vectors are used, we denote them by $R_0, R_1, \ldots$, respectively. These vectors are generated using entropy external to the cipher computation, and hence uncontrollable & unknown to the attacker. The numeric values of R_i's change at every invocation of the countermeasure but fixed during the course of one invocation of the countermeasure.
- $\mathbf{0}$. We use $\mathbf{0}$ to denote an n-bit vector of all 0 bits.
- $\mathbf{1}$. Similarly, we use $\mathbf{1}$ to denote an n-bit vector of which most significant $n-1$ bits are 0 and the least significant bit is 1.

[3] We assume the most common fault attack model, where the fault value is constant but unknown to the attacker; but the same fault can be repeated over time by keeping the source of the fault unchanged. Such model is used, for example, in [3].

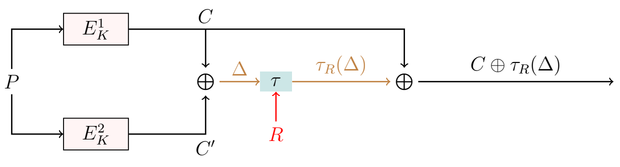

Fig. 1 Type I infective countermeasures

2.4.1 Type I

This type of countermeasures allows the two computations, $E_K^1(\cdot)$ and $E_K^2(\cdot)$ to finish their iterations full (η) iterations; the difference Δ is then obtained by XORing the outputs from the actual (C) and the redundant cipher computations (C'). Figure 1 shows a pictorial view.

Then, Δ is passed through a function $\tau(\cdot)$, which is parametrized by R. This $\tau_R(\Delta)$ is such that, $\tau_R(\mathbf{0}) = 0 \ \forall R$; and for any $\Delta \neq 0$, the distribution of $\{\tau_R(\Delta)\}$ is indistinguishable from the uniform distribution over $\mathbb{F}_2^n$.

Notice that, the attacker should not be able to deduce the input of $\tau_R(\cdot)$ except the case when output of $\tau_R(\cdot) = 0$. If, somehow, the attacker is able to do so, then she can find out Δ and hence can perform a DFA. So, one intrinsic property of $\tau_R(\cdot)$ is that, it is not invertible, except $\tau_R(\mathbf{0}) = 0$.

The output from $\tau_R(\cdot)$ is XORed with C (the output from the actual computation), and this is made available. Hence, the attacker gets, $C \oplus \tau_R(\Delta)$, which can be interpreted as the infected output from the actual cipher.

Examples of this type of countermeasures include the schemes in [33] ($\tau_R(\Delta) = R \cdot \Delta$ over GF(2^n) multiplication) or [22] ($\tau_R(\Delta) = N(R) \oplus N(R \oplus \Delta)$, where N is defined as nonlinear hybrid cellular automata).

2.4.2 Type II (cipher level)

This new type of infection is introduced in [23] and also referred to as the cipher level countermeasure. An schematic view is given in Fig. 2. The schemes belonging to this type rely on the diffusion property of the underlying cipher to spread the infection. Instead of letting both the ciphers to run full (η) rounds, these schemes try to identify the effect of a fault at an intermediate round.

Say, the registers S_0 and S_1 are computing the actual and redundant computations, respectively. In the *actual round* (respectively, *redundant round*), S_0 (respectively, S_1) is updated only. A *meaningful round* refers to either an actual or a redundant round. Then, the XOR difference, δ, is computed ($\delta = S_0 \oplus S_1$);[4] and this δ goes through a function $\sigma(\cdot)$. This function, $\sigma(\cdot)$ is such that, it outputs $\mathbf{0}$ only when its input is $\mathbf{0}$. As for the choice of $\sigma(\cdot)$, inversion in GF(2^8) is proposed per SBox of AES in [23]. The choice of $\sigma(\cdot)$ in [44] is the $n:1$ OR gate, although the authors introduce/use the term BLFN instead of OR gate. The same choice for $\sigma(\cdot)$ with the same term is later used in [36] as well as in its journal extension [37]. The scheme also takes a random n-bit vector β such that, $\exists k_0$ for which, $k_0 \oplus \texttt{RoundFunction}_0(\beta)$ gives β. See Sect. 4 for more details on Type II countermeasures.

One special register, called the *dummy register*, denoted by S_2 here, is initialized with β. In the so-called *dummy rounds*, this register S_2 is updated only (so, this is not a meaningful round). This register S_2 is updated (with influence from δ) by the following rule:

$$S_2 \leftarrow \begin{cases} \beta & \text{if no fault is injected,} \\ \beta' \ (\neq \beta) & \text{if fault is injected.} \end{cases}$$

During the subsequent meaningful rounds, S_0 and S_1 are updated with influence from S_2. So, in case of a fault, the contents of S_0 and S_1 are infected, which propagates further in the subsequent rounds. This ideally makes the final output, S_0, random in case of a fault. In case of no fault, the content of S_0 and S_1 is not deviated from its actual computation—this ensures a proper execution of the cipher. In our patched version of the LatinCrypt'12 countermeasure (proposed in [23]), we do not update S_0 and S_1 during meaningful rounds.

To determine whether a dummy or a meaningful round will take place, the authors in [23] propose to use a random bit, λ: $\lambda = 0$ means a dummy round, $\lambda = 1$ means a meaningful round. Also, one counter i is looped from 1 to 2η; within this loop, $\texttt{RoundFunction}_j(\cdot)$'s, $(1 \leq j \leq \eta)$ are performed, together with the corresponding round key insertions. Further, when this is a meaningful round (i.e., $\lambda = 1$), i is even implies an actual round takes place, and i is odd implies a redundant round takes place. We make the ordering of actual, redundant and dummy rounds deterministic in the patched version, so we do not use λ. In both [23,44], i is initialized by 1 (odd), which means, a redundant round always precedes the corresponding actual round. In the modified scheme in [36], however, the order of execution of actual and redundant rounds is not predetermined.

[4] In our patched version of [23] (Sect. 4.2), we do not explicitly compute δ.

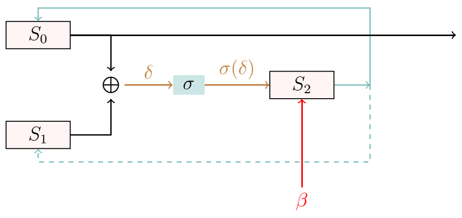

Fig. 2 Type II infective countermeasures (schematic)

2.5 Necessity and sufficiency of randomness

It is well known from [33] that randomness is required in infective countermeasures. However, they do not present any formal proof, rather their comment is more of an informal case-study with AES-128 encryption.

The attacker basically exploits the information she gains from DFA to derive the key K. If Z is the information obtained from DFA, then the amount of information available about the key can be measured by the mutual information between random variables K and Z:

$$I(K; Z) = \sum_{k \in K} \sum_{z \in Z} \Pr(k, z) \log \left(\frac{\Pr(k, z)}{\Pr(k) \Pr(z)} \right).$$

It is easy to see that $I(K; Z) = 0$ if and only if K and Z independent. Therefore, it is sufficient that Z is uniformly distributed to make DFA fail.

For instance, in Type I infective countermeasures, the attacker gets a $Z = C \oplus \tau_R(C' \oplus C)$ as an output. Therefore, Z must be random in order to be independent of key K. Thus for infective countermeasure, it is necessary and sufficient to have $\tau_R(C' \oplus C)$ random. On the other hand, in the detective countermeasure, once we detect the faulty cipher, we can simply output Z as constant. Here, randomness of Z is not necessary (random Z works too).

Detection-based countermeasures set the mutual information zero by suppressing the (faulty) output. In contrast, infection-based countermeasures apply one-way functions on the output difference (which contains nonzero mutual information regarding the secret key) to reduce the mutual information to zero.

Nonzero mutual information between K and Z could imply an information leakage. For example, the mutual information between the XOR of the non-faulty and the faulty ciphertext, with the key is 32-bits in case of the diagonal fault attack on AES [39]. For the design choice of Remark 1, the mutual information is of 1-bit in case of a fault; hence, we modify it to Algorithm 2(b) (Sect. 3.3). More quantitative analysis in this direction can be found in [30], where the authors used CLEFIA as the benchmark cipher for various DFA approaches.

In all our proposed countermeasures/patch, the mutual information between the key K and the observation Z which is available to the attacker is zero (in case of a fault). Therefore, the attacker does not get any information regarding the key in case of a fault.

2.6 Scope and applicability

The working procedure of infective countermeasures relies on the (nonzero) difference between the actual and redundant computations of the cipher. Hence, if both the actual and the redundant computations are infected by identical faults, which result in the same output in both cases, the corresponding difference will be zero. In this case, the countermeasure will treat this as non-faulty; and make the faulty output (from the actual cipher) available to the attacker (without infection). Hence, this type of repeated faults can be used to make the countermeasure invalid. We call this type of faults a *double fault*.

However, repeating the identical fault in spatial domain would require a very strong adversary model. Such model, although shown to be effective through experiment (the only case we know is, [41]), is not common. In a more common model, the authors assume that the attacker can repeat the fault in time domain (such as, the model used in [3] to break an infective countermeasure). Dealing with double faults is rather tricky in infective countermeasure, and we leave this problem open for future research.

Besides double faults, infective countermeasures cannot provide safeguard against few other fault models, for example, *Ineffective Fault Attack* [14]. Another type of fault attack, known as *Collision Fault Attack* (CFA) [12], injects transient fault near the start of the cipher (E) execution. This attack works very similarly to DFA. Suppose, one computation is allowed to run as-is, where the other computation is injected with a fault (near the beginning of the cipher execution). If, it happens that, both the faulty and non-faulty outputs are equal, then the situation is similar to attacking the inverse of the cipher (E^{-1}) by DFA (as the fault can be thought to be injected near the end of execution of E^{-1}). The attacker may be able to deduce information regarding the early rounds of the cipher (or equivalently, the later rounds of E^{-1}). This may eventually help her to find information on the secret key (by an analysis similar to DFA on E^{-1}). We note that, while such model cannot be protected by Type I countermeasures (as both the actual and redundant computations produce equal output); our patched version of the LatinCrypt'12 countermeasure (Sect. 4.2) can indeed protect against such an attack.

2.6.1 Impeccable circuits/CRAFT

The authors in [1] presented a novel idea, called impeccable circuits, which allows fault detection using error detecting codes. The idea is later extended to a full-fledged cipher named CRAFT [9]. Impeccable circuits/CRAFT can detect faults up to some extent (the fault coverage can be changed by changing the underlying error detection code). Also, both the constructions can detect double fault.

In comparison with that, an infective countermeasure works on a different principle. It does not rely on an error detecting/correcting code, but relies on the difference of the final/intermediate output. While the impeccable circuits/CRAFT cannot detect faults beyond the permissible coverage of the error detecting code, an infection countermeasure can take action any such fault, except double fault. However, we note that an infection scheme can be made protected against double faults by employing a similar concept (different encodings for the actual and the redundant computations).

2.6.2 Statistical ineffective fault attack (SIFA)

SIFA [17] is a type of IFA, which also makes use of statistical information. Infective countermeasures do not have any inherent protection against SIFA (or other IFA).

At the same time, it is worth noticing that SIFA does not necessarily phase out infective countermeasures in general. We point out the following use-case for infective countermeasure in the presence of a SIFA model. SIFA requires a large number of faults (typically of the order of thousands), implying an extensive control over the target device. DFA, on the other hand, has been shown to work with as low as only one fault. Hence, SIFA is more complex in nature compared to DFA. So, there could be scenarios where the attacker is unable to apply SIFA may be able to mount DFA, implying the necessity of a DFA protection.

Basically, the applicability for our countermeasures can be compared to that of CRAFT [9]. It may be noted that, although CRAFT is published after SIFA, it is not protected against SIFA (this is acknowledged by the authors of CRAFT in [9, Section 1]).

In a nutshell, we highlight the fault attacks that are generally considered outside the scope of an infective countermeasure, together with corresponding countermeasure(s):

– *Stuck-at/hard* Since the algorithmic description of the countermeasure can be altered in such cases, we recommend to employ some dedicated device that is able to thwart a fault injection (see Sect. 1).

– *Double fault* Injecting identical faults at both the actual and the redundant computation can skip the infection mechanism. This can be avoided by using different encodings for the two computations (e.g., similar to `CRAFT`). If the attacker cannot inject identical faults to more than two computations, then more redundant computations (such as, triplication [6]) can be used.
– *IFA (SIFA)* Any ineffective fault analysis, including SIFA, can bypass the infective mechanism as an ineffective fault does not result in a change of the target state. In such a case, one can employ some form of error correction and/or a *masking* countermeasure as used in the recent SIFA countermeasures [15,40,42].

It may be noted that, such countermeasures either are beyond the control of a cipher designer, or can be non-trivial/costly to design or implement compared to an infective countermeasure. Hence, in cases where only DFA protection is solicited, infective countermeasures can be used.

2.7 Connection with side channel countermeasures

Side channel countermeasures are not directly considered in this work, as we focus on DFA. Here, we would like to note that SCA is already known quite powerful. If the attacker has access to side channel information then she can choose the degree to which she can use it to obtain further information—she should not be limited to recover only Δ. More precisely, she can target other potentially useful registers as well; which may allow her to recover the secret key directly, thereby a separate DFA would not be required. So, if a side channel adversarial model is considered within the scope, the infection component as well as both the actual and the redundant computations are required to be protected by some SCA countermeasure. Hence, we implicitly assume the actual and the redundant computations are already protected, and comment on protecting the infection component here.

In general, the *hiding* countermeasures [34, Chapter 7] that aim at decreasing the SNR (by increasing noise, or decreasing signal) require minimal changes to the underlying cipher, as the countermeasures are applied at the implementation level. Hence, we believe adopting such countermeasure would be relatively straightforward on top of our countermeasures.

On the other hand, the masking countermeasures [34, Chapter 9] rely on randomizing the intermediate variables so that the power leakage is independent of those variables. Masking is applied on the algorithmic level, so applying it to the infective countermeasures could be of interest. In this regard, we note the following observations:

– For the multiplication based schemes (Sect. 3.3.1), masking can be applied by following the approach similar to [33]. This is done in Algorithms 3(a) and 3(b) (Sect. 3.3.2).
– For the derivative-based scheme (Sect. 3.3.3), the underlying cipher is reused as the building block. Therefore, only the difference (Δ) needs to be protected by masking. This can be done, for example, by computing $\Delta' = C \oplus C' \oplus R'$ and redefining $\tau_R(\Delta)$ as $\tau_R(\Delta', R') = E_R(R') \oplus E_R(\Delta')$ where $R' \xleftarrow{\$} \mathbb{F}_2^n$ with $R' \neq C \oplus C', \mathbf{0}$. The additional cost incurred for this protection would be that of an n-bit $\oplus$ operation and generation of an n-bit random vector R'.
– For the patched version of the LatinCrypt' 12 countermeasure (Sect. 4.2), Eve can only target those steps where S_2 is updated (Lines 5, 6, 7, 9 of Algorithm 6), as S_0 and S_1 are presumably SCA protected. However, one may notice that S_2 is initialized with β (which is randomly generated) (Line 1). Hence, β inherently works as a mask to S_2, and no other masking would be necessary.

3 Type I constructions

We describe the Type I schemes here with adequate details. As already mentioned, the existing schemes in Type I category can be classified in two sub-categories; we refer to them as multiplication based (Sect. 3.1) and derivative based (Sect. 3.2). We then propose our constructions of Type I constructions (Sect. 3.3) followed by the corresponding benchmarking results (Sect. 3.4).

3.1 Multiplication based constructions

To the best of our knowledge, the earliest infective countermeasure, which is still unbroken, is based on the GF(2^n) multiplication proposed in [33] (see Algorithm 1). In our terminology, here $\tau_R(\Delta) = R \cdot \Delta$, where $(\cdot)$ refers to a GF(2^n) multiplication and given $R \neq \mathbf{0}, \mathbf{1}$. If $R = \mathbf{0}$, then C is available without infection to the attacker. On the other hand, if $R = \mathbf{1}$, then she gets $C \oplus \Delta$ as the output; from where she can compute Δ (as she knows C).

Algorithm 1 Multiplication based infective countermeasure: FDTC' 12

```
Input: $C, C'; R_0, R_1, R_2$                    ▷ $R_0, R_1 \neq \mathbf{0}; R_2 \neq \mathbf{0}, \mathbf{1}$
Output: $C$ if no fault; random, otherwise
1: $a \leftarrow R_2 \cdot (C \oplus R_0)$          ▷ $\cdot$ refers to GF($2^n$) multiplication
2: $b \leftarrow R_2 \cdot (C' \oplus R_1)$
3: $c \leftarrow a \oplus b$
4: $d \leftarrow R_0 \oplus R_1$
5: $e \leftarrow R_2 \cdot d$
6: $f \leftarrow (C \oplus R_0) \oplus c$
7: $g \leftarrow f \oplus e$                        ▷ $g = (C \oplus R_0) \oplus R_2 \cdot (C \oplus C')$
8: return $g \oplus R_0$
```

Hence, for AES, one has to implement a GF(2^{128}) multiplication. However, the authors acknowledge that the GF(2^{128}) multiplication is costly in hardware, although they do not provide any benchmarking result. So, they come up with the idea of substituting the GF(2^{128}) multiplication by sixteen independent GF(2^8) multiplications (which replace the GF(2^{128}) multiplications in Lines 1, 2, 5 of Algorithm 1). The authors claim, the security of the scheme will remain unchanged, given those sixteen random multipliers are independent.

While the original proposal remains unbroken so far, this lightweight alternative is broken soon afterwards [7]. We describe the attack here. Assume the attacker is able to replicate the same fault value in the temporal domain (i.e., the fault value is constant in all the injections). Under this assumption, the lightweight variant will restrict two particular values to output per GF(2^8) multiplication; corresponding to the cases when $R = \mathbf{0}$ or $\mathbf{1}$. So, per GF(2^8) multiplication, it only outputs 254 values. One of the missing values is the non-faulty output (C), whereas the other is the faulty output ($C \oplus \Delta$). As explained earlier, leaking $C \oplus \Delta$ can lead to a successful DFA. This breaks the security claim of the lightweight proposal, as attacker can exhaust 254 cases (then repeat the procedure till all the key bytes are recovered). For the original GF(2^{128}) multiplication for AES, it would require 2^{128} repeats of the same fault and storage; making it impractical.

3.2 Derivative-based constructions

The generic schemes of the derivative-based category follow the concept of the Boolean derivative. More precisely, they use a nonlinear function $N(\cdot)$ and compute $\tau_R(\cdot)$ as the derivative of $N(\cdot)$ at Δ:

$$\tau_R(\Delta) = N(R) \oplus N(R \oplus \Delta).$$

In [21], the authors proposed such a construction by choosing a quadratic N. However, this scheme was subsequently broken in [3]. Recall from Sect. 3 that $\tau_R(\cdot)$ has to be non-invertible. This claim was shown to be incorrect in [3], under the assumption that attacker can keep Δ a constant. Note that derivative of a quadratic function is affine; further, the attack generates a sequence where in each bit, there is only one term of the form $\Delta_i r_i$, and rest of terms are independent of R. So for a constant Δ, it easy to find the value of r_i by seeing the distribution of these bits. Later, the authors of [21] opt for more complex and high degree function $N(\cdot)$ based on nonlinear cellular automata in [22]; and this is the only yet unbroken proposal in this category, to the best of our knowledge.

3.3 New Type I schemes proposed in this work

With the backdrop presented, we now introduce our proposals. The novelty of our proposals lies in the simplicity and lower implementation cost.

3.3.1 Multiplication based

As GF(2^n) multiplication has a significant hardware (where n is generally 128), we propose two lightweight alternatives. First, we present a scheme that requires a significant amount of randomness in Algorithm 2(a). It generates n fresh random vectors (each of which is of n bits) $R_0, \ldots, R_{n-1}$ (none of which is equal to $\mathbf{0}, \mathbf{1}$). They are used to generate a random $n \times n$ binary matrix M, which is then multiplied (over GF(2)) with Δ to constitute $\tau_{(\cdot)}(\cdot)$.

Since this scheme requires a total of n^2 bits of entropy, it may not appear suitable where frequent random number generations is not easy. Hence, instead of using n^2 random bits; we next propose another scheme that uses $2n$ bits of entropy, which is described in Algorithm 2(b). We define the i^{th} *cyclic rotation*, $\rho^i(\cdot)$, on an n-bit vector $\mathbf{a} = (a_0, a_1, a_2, \ldots, a_{n-1})$, recursively as:

$$\rho^0(\mathbf{a}) = \mathbf{a};$$
$$\rho^1(\mathbf{a}) = (a_{n-1}, a_0, a_1, \ldots, a_{n-2});$$
$$\rho^i(\mathbf{a}) = \rho^{i-1}(\rho^1(\mathbf{a})) \quad \text{for } i = 2, \ldots, n-1.$$

For example, with the vector $\mathbf{a} = (a_0, a_1, a_2, a_3)$, we have $\rho^1(\mathbf{a}) = (a_3, a_0, a_1, a_2)$; $\rho^2(\mathbf{a}) = \rho^1(\rho^1(\mathbf{a})) = \rho^1(a_3, a_0, a_1, a_2) = (a_2, a_3, a_0, a_1)$ and so on. Once we generate an n-bit random vector R_0 ($\neq \mathbf{0}, \mathbf{1}$), we create an $(n-1) \times n$ binary matrix M whose ith row is the ith cyclic rotation of R_0 (for $i = 0, \ldots, n-2$). Following this, we augment another randomly generated n-bit binary vector R_1 ($\neq \mathbf{0}, \mathbf{1}$) as the last row to M (to make it an $n \times n$ binary matrix). Then, we multiply M and Δ over GF(2).

In both the algorithms, if the attacker can guess M, then she will be able to deduce information about Δ. However, guessing M succeeds with probability $\frac{1}{2^{n^2}}$ for Algorithm 2(a), and $\frac{1}{2^{2n}}$ for Algorithm 2(b).

Remark 1 One may notice, the structure of M in our second alternative (Algorithm 2(b)) is same as a circulant matrix, except the last row (it would be a circulant matrix if last row would be equal to $n-1$th cyclic rotation of R_0). However, we observe that circulant M reveals one bit of entropy of Δ. Suppose, $\Delta = (\Delta_0, \Delta_1, \Delta_2, \ldots, \Delta_{n-1})^\top$ and $R_0 = (r_0, r_1, r_2, \ldots, r_{n-1})$. So, we have:

Algorithm 2(a) Multiplication based infective countermeasure (1st variant)

```
Input: $C; C'$                                   ▷ $|C|, |C'| = n$
Output: $C$ if no fault; random, otherwise
1: $\Delta \leftarrow C \oplus C'$
                          ▷ $\Delta$ is represented as an $n$-bit vector
2: for $i \leftarrow 0; i < n; i \leftarrow i+1$ do
3:     $R_i \xleftarrow{\$} \mathbb{F}_2^n$
4: $M \leftarrow \begin{bmatrix} R_0 \\ R_1 \\ \vdots \\ R_{n-1} \end{bmatrix}$
5: $a \leftarrow M \cdot \Delta$
                               ▷ Multiplication is over GF(2)
6: return $C \oplus a$
```

Algorithm 2(b) Multiplication based infective countermeasure (2nd variant)

```
Input: $C; C'$                                   ▷ $|C|, |C'| = n$
Output: $C$ if no fault; random, otherwise
1: $\Delta \leftarrow C \oplus C'$
                          ▷ $\Delta$ is represented as an $n$-bit vector
2: $R_0, R_1 \xleftarrow{\$} \mathbb{F}_2^n$
3: $M \leftarrow \begin{bmatrix} R_0 \\ \rho^1(R_0) \\ \vdots \\ \rho^{n-2}(R_0) \\ R_1 \end{bmatrix}$
4: $a \leftarrow M \cdot \Delta$
                               ▷ Multiplication is over GF(2)
5: return $C \oplus a$
```

$$\begin{aligned}
\tau_R(\Delta) &= M \cdot \Delta \\
&= \begin{bmatrix} r_0 & r_1 & r_2 & \dots & r_{n-1} \\ r_{n-1} & r_0 & r_1 & \dots & r_{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r_1 & r_2 & r_3 & \dots & r_0 \end{bmatrix} \begin{bmatrix} \Delta_0 \\ \Delta_1 \\ \vdots \\ \Delta_{n-1} \end{bmatrix} \\
&= \Bigg[\bigoplus_{i=0}^{n-1} r_i \Delta_i, r_{n-1}\Delta_0 \oplus \left(\bigoplus_{i=0}^{n-2} r_i \Delta_{i+1} \right), \dots, r_0 \Delta_{n-1} \\
&\quad \oplus \left(\bigoplus_{i=1}^{n-1} r_i \Delta_{i-1} \right) \Bigg]^\top
\end{aligned}$$

XORing all the bits of $\tau_{R_0}(\Delta)$ will give: $\bigoplus_{i=0}^{n-1} \bigoplus_{j=0}^{n-1} r_i \Delta_j = \left(\bigoplus_{j=0}^{n-1} \Delta_j\right) \left(\bigoplus_{i=0}^{n-1} r_i\right)$ = (parity of Δ) AND (parity of R_0). Now, since we assume that the attacker is able to replicate the same Δ over multiple runs; hence Δ is constant, so is parity of Δ. Also, R_0 is random, which means; its parity is 0 with probability $\frac{1}{2}$. Hence,

$$\begin{aligned}
&\text{(parity of } \Delta\text{) AND (parity of } R_0\text{)} \\
&= \begin{cases} 0, \text{ with probability } 1 & \iff \text{ parity of } \Delta \text{ is } 0 \\ 0, \text{ with probability } \frac{1}{2} & \iff \text{ parity of } \Delta \text{ is } 1 \end{cases} .
\end{aligned}$$

So, just by XORing all the bits, attacker is able to deduce the parity of Δ. This gives the attacker 1-bit of mutual information between the observation $C \oplus a$ and the key, in case of a fault. For this reason, this idea is modified to Algorithm 2(b) which is free from this shortcoming.

3.3.2 Side channel protection (masking)

One may note that, the authors [33] do not explicitly compute Δ from C and C'. This is because they are careful to avoid what they call *combined attacks*. Such attacks work by first injecting the fault to cause a nonzero Δ, then to recover Δ by side channel attacks (such as the power leakage). In this way, attacker can solve for Δ from the side channel information, the knowledge of Δ further helps to recover the secret key by utilizing DFA. The authors first mask C and C' by XORing them with two random vectors R_0 and R_1 respectively; then applying the GF(2^n) multiplications (Lines 1, 2 in Algorithm 1); and later canceling the effect of unwanted masks. As our focus is on DFA in Algorithms 2(a) and 2(b), we do not consider masking Δ. In situation, where SCA protection is needed, the following Algorithms 3(a) and 3(b) can be used (as replacement for Algorithms 2(a) and 2(b), respectively). As noted in Sect. 2.7, the implicit assumption to apply Algorithms 3(a) and 3(b) is that the rest (the actual and redundant computations) is already protected against SCA. As for the overhead of the masking, n-bit $\oplus$ (eight times) and $\cdot$ (two times) are to be performed, apart from generating two n-bit random vectors.

3.3.3 Derivative based

Since the constructions in this category use the derivative of a function, our idea is to use any existing cipher, rather than the usual approach of using an ad hoc solution. With our usual notations, this construction can be described as, $\tau_R(\Delta) = E_R(\mathbf{0}) \oplus E_R(\Delta)$, E being any standard cipher. Figure 3 shows the construction.

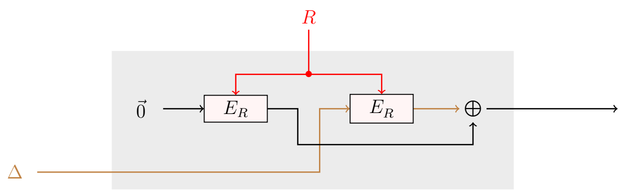

Fig. 3 Derivative-based construction of $\tau_R(\Delta)$: Our design

Algorithm 3(a) Multiplication based infective countermeasure (1st variant) with masking

```
Input: $C; C'$                                   ▷ $|C|, |C'| = n$
Output: $C$ if no fault; random, otherwise
1: for $i \leftarrow 0; i < n+2; i \leftarrow i+1$ do
2:     $R_i \xleftarrow{\$} \mathbb{F}_2^n$                    ▷ $R_0, R_1 \neq \mathbf{0}$
3: $M \leftarrow \begin{bmatrix} R_2 \\ R_3 \\ \vdots \\ R_{n+1} \end{bmatrix}$
4: $a \leftarrow M \cdot (C \oplus R_0)$        ▷ $\cdot$ refers to GF($2^n$) multiplication
5: $b \leftarrow M \cdot (C' \oplus R_1)$
6: $c \leftarrow a \oplus b$
7: $d \leftarrow R_0 \oplus R_1$
8: $e \leftarrow M \cdot d$
9: $f \leftarrow (C \oplus R_0) \oplus c$
10: $g \leftarrow f \oplus e$
                         ▷ $g = (C \oplus R_0) \oplus M \cdot (C \oplus C')$
11: return $g \oplus R_0$
```

Algorithm 3(b) Multiplication based infective countermeasure (2nd variant) with masking

```
Input: $C; C'$                                   ▷ $|C|, |C'| = n$
Output: $C$ if no fault; random, otherwise
1: $\Delta \leftarrow C \oplus C'$
2: $R_0, R_1, R_2, R_3 \xleftarrow{\$} \mathbb{F}_2^n$        ▷ $R_0, R_1 \neq \mathbf{0}$
3: $M \leftarrow \begin{bmatrix} R_2 \\ \rho^1(R_2) \\ \vdots \\ \rho^{n-2}(R_2) \\ R_3 \end{bmatrix}$
4: $a \leftarrow M \cdot (C \oplus R_0)$        ▷ $\cdot$ refers to GF($2^n$) multiplication
5: $b \leftarrow M \cdot (C' \oplus R_1)$
6: $c \leftarrow a \oplus b$
7: $d \leftarrow R_0 \oplus R_1$
8: $e \leftarrow M \cdot d$
9: $f \leftarrow (C \oplus R_0) \oplus c$
10: $g \leftarrow f \oplus e$
                         ▷ $g = (C \oplus R_0) \oplus M \cdot (C \oplus C')$
11: return $g \oplus R_0$
```

We do not impose any restriction on E, so it can be taken as the underling cipher or can be another cipher. However, to keep overhead low, we recommend to reuse the underlying cipher or use an optimized implementation/lightweight cipher.

3.3.4 Advantages

This strategy gives us a few advantages over the usual ad hoc approaches, such as:

1. The underlying cipher is already analyzed thoroughly for weakness. This gives us more confidence (as breaking such a countermeasure would probably imply breaking the cipher used); particularly when compared to the new and ad hoc approaches.
2. The cipher to be protected can be reused. This avoids the necessity to build a new hardware/code afresh for a new component. In that case, we only have performance penalty in terms of throughput.
   Further, as already mentioned, instead of the full cipher; ξ rounds of $\texttt{RoundFunction}_0(\cdot)$, together with the corresponding round key insertion can be used with minimal compromise to security.
3. One may notice, for example, the derivative-based construction in [22] is only defined for block size, $n = 128$. Although, in theory, the concept can be generalized for other block sizes (e.g., for a 64-bit block), one has to go through the lengthy design process, and has to guarantee its security from scratch. In our construction, one can easily choose from a pool of already analyzed ciphers (including the underlying cipher itself).

Remark 2 Since this type of construction does not require inversion, any hash function can be used in place of E. For a hash function H (which does not involve a key), we propose to use $\tau_R(\Delta) = H(R) \oplus H(R \oplus \Delta)$.

Remark 3 If the cipher E is invertible, then a new construction of $\tau_{(\cdot)}(\cdot)$ can be given. The cipher, E, takes a random vector R as its key and $\mathbf{0}$ as input. Then, it XORs Δ with $E_R(\mathbf{0})$. This is then decrypted using the same key R. So, $\tau_R(\Delta)$ is given by: $\tau_R(\Delta) = E_R^{-1}(E_R(\mathbf{0}) \oplus \Delta)$. We call this type of construction as encrypt-XOR-decrypt-based design. A pictorial description is given in Fig. 4. This may, however, cost more due to inverse key schedule, inverse SBox etc. during decryption.

3.4 Benchmarking results for Type I schemes

Here, we present benchmarking results for Type I schemes. The extra component needed to provide external randomness is not considered, following the previous papers (such as [2]). We consider AES encryption as the underlying cipher, so we implement it to get a perspective of overhead (so, $n = 128$, $\eta = 11$).

For the GF(2^{128}) multiplication in [33], we choose the irreducible polynomial as, $x^{128} + x^7 + x^2 + x + 1$ (the same polynomial used in AES-GCM). Also, since we do not consider the combined attacks (i.e., the fault attacks coupled with side channel attacks; see Sect. 3.1), we perform

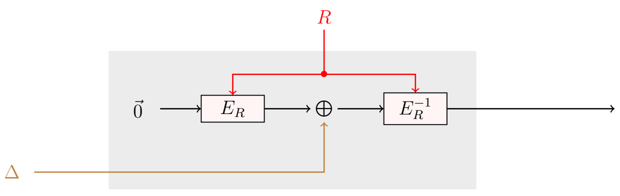

Fig. 4 Encrypt-XOR-decrypt-based design of $\tau_R(\Delta)$

Table 1 Software benchmarking results (clock cycles, code size) for Type I schemes

Construction		Clock cycles		Code size		Reference
		AVR	MSP	AVR	MSP	
AES Encryption (unprotected)		1.00 (14775)[a]	1.00 (14842)[a]	1.00 (2444)[b]	1.00 (3234)[b]	–
GF(2^{128}) Multiplication		2.28	2.57	0.28	0.47	[33], Sect. 3.1
Algorithm 2(b)		41.86	34.06	**0.28**	**0.35**	Section 3.3.1
Cellular Automata		≈ 2700	≈ 3400	0.82	0.88	[22], Sect. 3.2
AES Encryption Derivative[c]	$z = 11$	1.65	1.81	1.42	1.35	Section 3.2
	$z = 4$	**0.64**	**0.63**	1.06	1.36	

Bold indicates the least overhead in the category
[a],[b] Parenthesized values are the benchmarking costs in clock cycles (a) or bytes (b)
[c] z is a dummy parameter that can take any integer from $[\xi, \eta]$

this field multiplication only once (instead of 3 times, as in Algorithm 1). We only implement Algorithm 2(b) (second hardware friendly alternative, see Sect. 3.3.1), as the other alternative (Algorithm 2(a)) can be considered a part of it. We also implement the nonlinear hybrid cellular automata-based design in [22] (Sect. 3.2). For the derivative-based construction that relies on a standard cipher, we take AES encryption.

All Type I schemes are implemented as stand-alone module, which means; one has to account for the additional clock cycles/circuitry needed to run the actual and the redundant cipher, as well as other subsidiary modules (such as, XORing the outputs of the actual and the redundant ciphers etc.).

For software, we implement two versions of the derivative and encrypt-XOR-decrypt based implementations; corresponding to $z = 4$ and $z = 11$, where we consider the cipher constituted by z-rounds of $\mathtt{RoundFunction}_0(\cdot)$, with corresponding AddRoundKeys. Note that z is a parameter, whereas ξ is a constant for a given cipher (e.g., $\xi = 4$ for AES). So, z can take any integer value from $[\xi, \eta]$. Such reduced round version helps to reduce latency. In Table 1, we present the software performance results (both the clock cycles and code size) of our Type I proposals, along with the existing ones. These are given relative to 1.00× that of the unprotected AES encryption. The data presented here are taken as the average of multiple runs. We use open source codes available in FELICS tool [16] for AVR and MSP architectures.

As it can be seen, the derivative scheme based on AES (with $z = 4$) outperforms other schemes in software (0.64× in AVR and 0.63× in MSP) in terms of clock cycles. Technically, these are equivalent to 8 rounds of AES, minus the key schedule. In terms of relative code size, the Algorithm 2(b) works similar to, if not better, than a single GF(2^{128}) multiplication [33].

The data in the previous table may appear counter-intuitive; particularly noticing the GF(2^{128}) multiplication takes less clock cycles than the matrix multiplication in Algorithm 2(b). However, one should keep in mind that:

1. All implementations are quite basic. There are scopes to optimize the codes keeping clock cycles/code size in mind. The reason behind this is to get a fair comparison among the constructions.
   For example, the matrix multiplication in Algorithm 2(b) is done on bit-by-bit basis. In the 8-bit AVR micro-controller, this can be sped up by using byte-by-byte operations.
2. The FELICS framework takes the input and output test vectors in (arrays of) bytes. This is helpful, e.g., for AES; but causes extra overhead for bit-oriented constructions. For the bit-oriented matrix multiplication in Algorithm 2(b), one has to go through additional conversion from byte to bit (at the beginning) and from bit to byte (at the end); which adds unnecessary costs.

As for the hardware benchmark, we choose the Spartan 3 (3s1500fg676-4) and Virtex 6 (6vcx75tff484-1) FPGA families, and report the results in Table 2. Here, our hardware friendly approach, Algorithm 2(b) outperforms all other designs in both the families. In fact, the relative amount of resource utilization in the devices are negligibly small.

4 Type II (cipher level) constructions

Here, we focus on the Type II constructions for infective countermeasures. At first, the n-bit random vectors β and k_0[5] are so chosen that, $\mathtt{RoundFunction}_0(\beta) = \beta \oplus k_0$. Finding such pairs is easy in SPN ciphers—one can set a β randomly, then compute, $k_0 = \beta \oplus \mathtt{RoundFunction}_0(\beta)$. Notice that, β uniquely identifies k_0; so once a k_0 is fixed (based on a β), for β' ($\neq \beta$), $\mathtt{RoundFunction}_0(\beta') \oplus k_0$

[5] It is to be observed that k_0 is not a round key.

Table 2 Hardware benchmarking results (FPGA) for Type I schemes

Construction	Spartan 3			Virtex 6		Reference
	Slices	Slice F/Fs	4-input LUTs	Slice registers	Slice LUTs	
AES Encryption (unprotected)	1931 (14)	785 (2)	3551 (13)	540[a]	1402 (3)	–
GF(2^{128}) Multiplication	8398 (63)	–	16496 (61)	–	7264 (15)	[33] Sect. 3.1
Cellular Automata	214 (1)	136[a]	406 (1)	136[a]	210[a]	[22] Sect. 3.2
Algorithm 2(b)	**91**[a]	**130**[a]	**88**[a]	**130**[a]	**56**[a]	Section 3.3.1

Bold indicates the least overhead in the category
$(\cdot)$ indicates % resource utilization, [a] indicates negligible utilization

will not give β. Both β and k_0 are (re-)generated at each invocation of the countermeasure, but are fixed during one invocation. These are kept secret from the attacker.

However, finding such (β, k_0) pair for Feistel constructions can be tricky in general. In a Feistel cipher, the f function can map n_1 bits to n_2 ($\neq n_1$) bits—in such a case, such k_0 does not exist (as length of β and k_0 are different). Further, the round keys can be nonlinearly mixed (in contrast to the SPN ciphers, where these are commonly XORed). In this case, one has to go through a rigorous computation to get k_0 for a given β, which may turn out to be quite costly. Hence we keep the Type II schemes for Feistel ciphers out of scope for this work. There can be Feistel ciphers, where these countermeasures are applicable at ease, but it has to be possibly checked on a case by case basis, and may not be generalized.

We use the following notations: $\neg$ for logical negation, $\wedge$ for logical AND, $\vee$ for logical OR, $+$ for arithmetic addition, $\times$ for arithmetic multiplication, $\lceil\cdot\rceil$ for the ceiling function, $x \cdot \mathbf{y}$ for scalar x multiplication with the vector $\mathbf{y}$ over GF(2), $\sharp x(\mathbf{y})$ for number of occurrence(s) of element x in vector $\mathbf{y}$.

We now briefly describe how infection in [23] works (see Algorithm 4). A variable, i, loops from 1 until it reaches 2η. At each round, a randomly generated bit, λ, determines whether this round will be a dummy round (i is not incremented) or a meaningful (actual or redundant) round (i is incremented by 1). When $\lambda = 0$, a dummy round occurs. It sets the value β to the register S_2 (S_2 is also initialized with β). When $\lambda = 1$, an actual or a redundant round occurs, depending on whether i is even or odd, respectively.

Algorithm 4 Infective countermeasure: LatinCrypt'12 (for SPN)

```
Input: { P
       { β; k_0
       { round keys k_j; j = 1,...,η
       {   derived from K
                                      ▷ RoundFunction_0(β) = β ⊕ k_0
Output: { E_K(P)   if no fault
        { random   otherwise
 1: $S_0 \leftarrow P$                                  ▷ Actual state
    $S_1 \leftarrow P$                                  ▷ Redundant state
    $S_2 \leftarrow \beta$                              ▷ Dummy state
 2: $T_0 \leftarrow 0$; $T_1 \leftarrow 0$; $T_2 \leftarrow \beta$
 3: $i \leftarrow 1$
 4: while $i \le 2\eta$ do
 5:     $\lambda \xleftarrow{\$} \mathbb{F}_2^1$
 6:     $a \leftarrow (i \wedge \lambda) \oplus ((\neg\lambda) \times 2)$
 7:     $b \leftarrow \lceil i/2 \rceil \times \lambda$
 8:     $S_a \leftarrow \texttt{RoundFunction}_b(S_a) \oplus k_b$
 9:     $T_a \leftarrow S_a \oplus T_2 \oplus \beta$
10:     $c \leftarrow (\lambda \wedge (\neg(i \wedge 1))) \cdot \sigma(T_0 \oplus T_1)$
11:     $S_2 \leftarrow S_2 \oplus c$
12:     $S_0 \leftarrow S_0 \oplus c$
13:     $i \leftarrow i + \lambda$
14: $S_0 \leftarrow S_0 \oplus \texttt{RoundFunction}_0(S_2) \oplus k_0 \oplus \beta$
15: return $S_0$
```

Algorithm 5 Infective countermeasure: CHES'14 (for SPN)

```
Input: { P
       { β; k_0
       { security level t (≥ 2η)
       { round keys k_j; j = 1,...,η
       {   derived from K
                                      ▷ RoundFunction_0(β) = β ⊕ k_0
Output: { E_K(P)   if no fault
        { random   otherwise
 1: $S_0 \leftarrow P$                                  ▷ Actual state
    $S_1 \leftarrow P$                                  ▷ Redundant state
    $S_2 \leftarrow \beta$                              ▷ Dummy state
 2: $i \leftarrow 1$; $q \leftarrow 1$
 3: $rstr \leftarrow \mathbb{F}_2^t \ni \sharp 1(rstr) = 2\eta$
 4: while $q \le t$ do
 5:     $\lambda \leftarrow rstr[q]$
 6:     $a \leftarrow (i \wedge \lambda) \oplus ((\neg\lambda) \times 2)$
 7:     $b \leftarrow \lceil i/2 \rceil \times \lambda$
 8:     $S_a \leftarrow \texttt{RoundFunction}_b(S_a) \oplus k_b$
 9:     $c \leftarrow (\lambda \wedge (\neg(i \wedge 1))) \wedge \sigma(S_0 \oplus S_1)$
10:     $d \leftarrow (\neg\lambda) \wedge \sigma(S_2 \oplus \beta)$
11:     $S_0 \leftarrow (\neg(c \vee d) \cdot S_0) \oplus ((c \vee d) \cdot S_2)$
12:     $i \leftarrow i + \lambda$
13:     $q \leftarrow q + 1$
14: return $S_0$
```

Since, $\lambda = 1$ makes i from even to odd, or vice versa, both the actual and redundant rounds are carried out η times. The two registers, S_0 and S_1 are used to compute the actual and redundant rounds, respectively. The variables a and b are updated such that:

$$a = \begin{cases} 0 & \text{if } \lambda \text{ is 1 and } i \text{ is even (actual round),} \\ 1 & \text{if } \lambda \text{ is 1 and } i \text{ is odd (redundant round),} \\ 2 & \text{if } \lambda \text{ is 0 (dummy round);} \end{cases}$$

$$b = \begin{cases} 0 & \text{if } \lambda \text{ is 0 (dummy round),} \\ \lceil i/2 \rceil & \text{otherwise (meaningful round).} \end{cases}.$$

Effectively, a determines which register (among S_0, S_1, S_2) to update; and b determines which $\texttt{RoundFunction}_{(\cdot)}(\cdot)$ and which round key/k_0 to use ($S_a \leftarrow \texttt{RoundFunction}_b(S_a) \oplus k_b$, Line 8).

Since we start i from 1 (odd), the redundant round always precedes the actual round. Three back-up registers, T_0, T_1, T_2 are also updated based on a. Basically, T_a holds a copy of S_a. Then Δ is computed by $T_0 \oplus T_1$, which is passed through a function $\sigma(\cdot)$ which has the property of returning $\mathbf{0}$ as output if the input is $\mathbf{0}$. The variable c is updated with the following rule:

$$c = \begin{cases} \sigma(T_0 \oplus T_1) & \text{if } \lambda \text{ is 1 and } i \\ & \text{ is even (actual round),} \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

Basically, c determines whether the output of $\sigma(T_0 \oplus T_1)$ (which is $\mathbf{0}$ for no fault, or nonzero for a fault) can be infected to S_2 (Line 11) and S_0 (Line 12). The authors propose to use inversion in GF(2^8) per SBox of $\texttt{AES}$ as $\sigma(\cdot)$.[6] Finally, to infect the fault which is injected at the last round (when $i = 2\eta - 1$ or 2η); the authors propose to update S_0 one last time based on S_2 (Line 14: $S_0 \leftarrow S_0 \oplus \texttt{RoundFunction}_0(S_2) \oplus k_0 \oplus \beta$). This ensures, any infection passes through at least one $\texttt{RoundFunction}_0(\cdot)$.

If no fault is injected, then both the actual and redundant states (S_0, S_1) as well as T_0, T_1 contain the same computation of the cipher; and the dummy state S_2 as well as T_2 contain β. Injecting a fault during a dummy round ($\lambda = 0$) will not give attacker any meaningful information. If a fault is injected during an actual round (S_0), then it will infect both S_0 and S_2 (Lines 12, 13). Note that, this infection will spread in S_0 in the subsequent actual rounds. Also, notice that, the c is zero for a redundant round, that is due to the fact that the corresponding actual round is not computed (but the redundant round is computed). Now, the infection in S_2 will change the subsequent update of S_2 from β (Line 8); this will change T_2 from β (Line 9). Now that $T_2 \neq \beta$, in all the subsequent rounds (actual, redundant and dummy alike) T_0, T_1 and T_2 will be infected further. In turn, this will infect subsequent S_0 and S_2 further. Finally, each of $S_0, S_1, S_2, T_0, T_1, T_2$ will contain random values.

If the attacker chooses a redundant round (when S_1 is updated) to inject a fault, then it will cause infection to S_0 and S_2 in the next actual round. Then, by a similar process, all three registers will be infected.

Despite its promise, this first cipher level protection was attacked soon afterwards in [7], and later in [44]. The basic observation that leads to the attack is, when a fault is injected at the last round ($i = 2\eta - 1$ or 2η), infection passes through only one $\texttt{RoundFunction}_0(\cdot)$ (Line 14). One round of diffusion is not generally sufficient to resist Eve to recover information on the faulty state. Hence, attacker can still perform DFA by injecting at the last round of the cipher computation.

The CHES'14 countermeasure, given in [44] (see Algorithm 5), is similar to that of the LatinCrypt'12. The variable, c is still updated the same way, but it now returns a bit instead of an n-bit vector; as $\sigma(\cdot)$ now returns 1-bit as output (the $n : 1$ OR gate, denoted by BLFN in the paper, is chosen as $\sigma(\cdot)$). Another major difference is, this scheme computes another 1-bit variable d, which is updated such that (Line 8):

$$d = \begin{cases} \sigma(S_2 \oplus \beta) & \text{if } \lambda \text{ is 0 (dummy round),} \\ 0 & \text{otherwise (meaningful round).} \end{cases}$$

Then, if $(c \vee d)$ is 1, then S_0 is substituted by S_2 (Line 11: $S_0 \leftarrow (\neg(c \vee d) \cdot S_0) \oplus ((c \vee d) \cdot S_2)$. This overwrites the content of S_0, and makes it random, as content of S_2 $(= \beta)$ is random. Figure 5 depicts the work-flow. We next show why this is a problem.

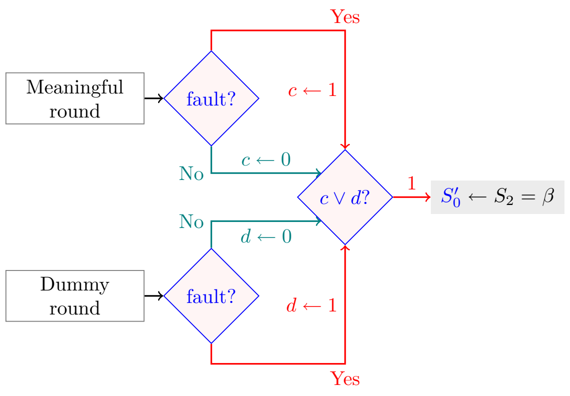

Fig. 5 Fault detection work-flow in CHES'14 countermeasure

[6] However, they keep the choice for $\sigma(\cdot)$ relaxed, so other $\sigma(\cdot)$ can also be used with their scheme.

4.1 Critical look at CHES'14 countermeasure

We would like to point out that the scheme proposed in CHES'14 [44] does not fit in the philosophy of infective countermeasures, rather it is a fault detection technique. Being a detective countermeasure, [44] does protect a cipher from DFA. However, at the same time, it is susceptible to 1-bit judgment condition, as discussed in Sect. 2.3. The work of [44] is well-cited as an infective countermeasure, such as [8,17,36,37]. For better understanding of both the infective and detective countermeasures, we feel to throw light on the proper categorization of the [44] countermeasure.

In the subsequent parts, we reveal our analysis after a detailed inspection of [44]. At this point, it would be helpful to recall the philosophy of infective schemes: To diffuse the fault to the entire state of the cipher in a non-deterministic. This philosophy is already described in the existing literature, e.g., [22,33]. We show that the CHES'14 countermeasure does not actually diffuse the fault in the state. Thus, it does not follow the infection philosophy.

In order to do a comparative analysis, one may recall their descriptions from Algorithm 4 and Algorithm 5. We start by doing dry runs of the algorithms for two main cases (namely, for the final and the penultimate rounds). This analysis can be easily extended for earlier rounds.

Case 1: last (actual) round fault

We start by analyzing a fault injection in the last actual round. Dry Run 1 traces the relevant steps of Algorithm 4. It can be seen that, once the fault is injected in the round; it is diffused. The faulty state is stored in T_0 and continues further using c before the last call to the round function, after which, the infected state is returned.

The CHES'14 scheme (Algorithm 5) is traced by Dry Run 2. It can be noticed that the faulty state S_0' has no contribution to the final state that is returned (this is merely a random state β). The 1-bit variables c and d implicitly detect the fault in meaningful or dummy rounds, respectively. If detected, the scheme simply replaces the faulty state by a random state, thereby deviating from the basic notion of infection. We use red to indicate infection, and blue to indicate substitution.

Case 2: penultimate (actual) round fault

Here, we investigate how a fault propagates through multiple rounds of Algorithms 4 and 5, by targeting the penultimate actual round for fault injection.

Like before, it can be observed from Dry Run 3 (tracing steps of Algorithm 4) and 4 (tracing steps of Algorithm 5) that, while for the LatinCrypt'12 countermeasure, the induced fault is diffused across the actual rounds; the CHES'14 countermeasure ends up substituting the faulty state with a random state ($= \beta$).

It is understood from the existing literature that detection countermeasures can be rendered useless if the bit that senses the presence of fault is altered. In fact, the main motivation for adopting the infective countermeasures (instead of the detective counterpart) was to protect the first order fault plus one-bit fault, as stated in [45, Section 3] (note that this paper introduced the concept of infection). If we only consider first order faults (i.e., the attacker can only target once during the cipher execution), then detective countermeasures would be sufficient (no need for infection). Hence, although the CHES'14 countermeasure protects against first order DFA; it does not inherently imply the infective nature.

It may be noted that, the same weakness has been pointed out in an independent work [19]. More precisely, it is mentioned in [19, Section VIII.A], "similar to the consistency check in the detection countermeasure, BLFN brings about

– Initial conditions: $i = 2\eta; \lambda = 1$
– Assume fault injection in Line 8 of Algorithm 4 and in Line 8 of Algorithm 5

Dry Run 1 Algorithm 4, Lines 6 – 15

```
1: $a \leftarrow 0$                                          ▷ $i$ is even
2: $b \leftarrow \eta$
3: $\underline{S_0'} \xleftarrow{\text{fault}}$ RoundFunction$_\eta(S_0) \oplus k_\eta$
4: $\underline{T_0} \leftarrow \underline{S_0'}$                  ▷ Infection begins
5: $\underline{c} \not\leftarrow 0$
6: $S_2' \leftarrow S_2 \oplus \underline{c}$
7: $\underline{S_0'} \leftarrow \underline{S_0'} \oplus \underline{c}$
8: $i \leftarrow 2\eta + 1$
9: $\underline{S_0'} \leftarrow \underline{S_0'} \oplus \underline{\text{RoundFunction}_{k_0}(S_2')} \oplus \beta$
10: return $\underline{S_0'}$                   ▷ $\underline{S_0'}$ is the infected state
```

Dry Run 2 Algorithm 5, Lines 6 – 14

```
1: $a \leftarrow 0$                                          ▷ $i$ is even
2: $b \leftarrow \eta$
3: $S_0' \xleftarrow{\text{fault}}$ RoundFunction$_\eta(S_0) \oplus k_\eta$
4: $c \leftarrow 1$
5: $d \leftarrow 0$
6: $S_0' \leftarrow S_2 = \beta$                             ▷ No infection
7: $i \leftarrow 2\eta + 1$
8: $q \leftarrow t + 1$
9: return $S_0'$
                                        ▷ $S_0'$ is just a random state
```

a 1-bit judgment condition. Therefore, the malicious modification on the result of BLFN may lead to infection failure." This observation further reinforces our claim.

Hence, the inability to thwart a single-bit judgment condition invalidates the claim that this countermeasure is infective. A similar argument works for the modified countermeasure in [36,37], rendering it outside the domain of the infective countermeasures.

4.2 Our patch for LatinCrypt'12 countermeasure

One may observe that if a fault is injected in a sufficiently early round, then it passes through sufficient rounds of diffusion, thus making it random to the attacker. This is valid regardless of the nature (actual, redundant or dummy) of the round. Hence, to make the [23] scheme sound, what we need is the assurance that no matter which round attacker chooses; the countermeasure goes through sufficient rounds of diffusion.

This observation leads us to propose a simple patch for the aforementioned scheme. Recall (Sect. 4) that attacker can choose the last meaningful round as the target for fault injection, which goes through only one round of diffusion (Line 14 of Algorithm 4). So, our patch for this scheme works basically by retaining the preceding part of Algorithm 4 (with few amendments), mainly to insert more diffusion at Line 14. In fact, we make use of the following interesting property:

$$\begin{aligned}\beta &= \texttt{RoundFunction}_0(\beta) \oplus k_0\\ &= \texttt{RoundFunction}_0(\texttt{RoundFunction}_0(\beta) \oplus k_0) \oplus k_0.\end{aligned}$$

We now define *z-nested round function*, denoted by $\texttt{RoundFunction}^z(\cdot\,,\cdot)$; $z = 1, 2, \ldots, \eta$; recursively as:

– Initial conditions: $i = 2\eta - 2; \lambda = 1$
– Assume fault injection in Line 8 of Algorithm 4 and in Line 8 of Algorithm 5

Dry Run 3 Algorithm 4, Lines 5 – 15

```
    Iteration 1: $(i = 2\eta - 2) < 2\eta$
1:  $\lambda = 1$                                              ▷ (Say)
2:  $a \leftarrow 0$                                          ▷ Actual Round
3:  $b \leftarrow \eta - 1$
4:  $S_0' \xleftarrow{\text{fault}} \texttt{RoundFunction}_{\eta-1}(S_0) \oplus k_{\eta-1}$
5:  $\underline{T_0} \leftarrow \underline{S_0'}$                 ▷ Infection begins
6:  $\underline{c} \nleftarrow 0$
7:  $\underline{S_2'} \leftarrow S_2 \oplus \underline{c}$
8:  $\underline{S_0'} \leftarrow \underline{S_0'} \oplus \underline{c}$
9:  $i \leftarrow 2\eta - 2 + 1$
    Iteration 2: $(i = 2\eta - 1) < 2\eta$
10: $\lambda = 1$                                              ▷ (Say)
11: $a \leftarrow 1$                                          ▷ Redundant Round
12: $b \leftarrow \eta$
13: $S_1 \leftarrow \texttt{RoundFunction}_{\eta}(S_1) \oplus k_{\eta}$
14: $T_1 \leftarrow S_1$
15: $\underline{c} \nleftarrow 0$
16: $\underline{S_2'} \leftarrow S_2 \oplus \underline{c}$
17: $\underline{S_0'} \leftarrow \underline{S_0'} \oplus \underline{c}$
18: $i \leftarrow 2\eta - 1 + 1$
    Iteration 3: $(i = 2\eta)$
19: $\lambda = 1$                                              ▷ (Say)
20: $a \leftarrow 0$                                          ▷ Actual Round
21: $b \leftarrow \eta$
22: $\underline{S_0'} \leftarrow \underline{\texttt{RoundFunction}_{\eta}(S_0')} \oplus k_{\eta}$
23: $\underline{T_0} \leftarrow \underline{S_0'}$                 ▷ Infection continues
24: $\underline{c} \nleftarrow 0$
25: $\underline{S_2'} \leftarrow \underline{S_2'} \oplus \underline{c}$
26: $\underline{S_0'} \leftarrow \underline{S_0'} \oplus \underline{c}$
27: $i \leftarrow 2\eta + 1$
28: $\underline{S_0'} \leftarrow \underline{S_0'} \oplus \underline{\texttt{RoundFunction}_0(S_2')} \oplus k_0 \oplus \beta$
29: return $\underline{S_0'}$                         ▷ $\underline{S_0'}$ is the infected state
```

Dry Run 4 Algorithm 5, Lines 5 – 14

```
    Iteration 1: $(q = t - 2) < t$
1:  $\lambda = 1$                                              ▷ (Say)
2:  $a \leftarrow 0$                                          ▷ Actual Round
3:  $b \leftarrow \eta - 1$
4:  $S_0' \xleftarrow{\text{fault}} \texttt{RoundFunction}_{\eta-1}(S_0) \oplus k_{\eta-1}$
5:  $c \leftarrow 1$
6:  $d \leftarrow 0$
7:  $S_0' \leftarrow S_2 = \beta$                              ▷ No infection
8:  $i \leftarrow 2\eta - 1$
9:  $q \leftarrow t - 2 + 1$
    Iteration 2: $(q = t - 1) < t$
10: $\lambda = 1$                                              ▷ (Say)
11: $a \leftarrow 1$                                          ▷ Redundant Round
12: $b \leftarrow \eta$
13: $S_1 \leftarrow \texttt{RoundFunction}_{\eta}(S_1) \oplus k_{\eta}$
14: $c \leftarrow 1$
15: $d \leftarrow 0$
16: $S_0' \leftarrow S_2 = \beta$                              ▷ No infection
17: $i \leftarrow 2\eta$
18: $q \leftarrow t - 1 + 1$
    Iteration 3: $(q = t)$
19: $\lambda = 1$                                              ▷ (Say)
20: $a \leftarrow 0$                                          ▷ Actual Round
21: $b \leftarrow \eta$
22: $S_0' \leftarrow \texttt{RoundFunction}_{\eta}(S_0' = \beta) \oplus k_{\eta}$
23: $c \leftarrow 1$
24: $d \leftarrow 0$
25: $S_0' \leftarrow S_2 = \beta$                              ▷ No infection
26: $i \leftarrow 2\eta + 1$
27: $q \leftarrow t + 1$
28: return $S_0'$
                                          ▷ $S_0'$ is just a random state
```

$$\texttt{RoundFunction}^z(x, y) = \begin{cases} \texttt{RoundFunction}_0(x) \oplus y \\ \quad \text{if } z = 1, \\ \texttt{RoundFunction}_0(\texttt{RoundFunction}^{z-1}(x, y)) \oplus y \\ \quad \text{otherwise.} \end{cases}$$

It can be checked, independent of z, $\texttt{RoundFunction}^z(\beta, k_0) = \beta$. Hence, we propose to substitute Line 14 of Algorithm 4:

$$S_0 \leftarrow S_0 \oplus \texttt{RoundFunction}_0(S_2) \oplus k_0 \oplus \beta$$

by

$$S_0 \leftarrow S_0 \oplus \texttt{RoundFunction}^z(S_2, k_0) \oplus \beta$$

where z ($\geq \xi$) is predetermined. It may be noted that, the overhead for nested round function is on the temporal domain and not on the spatial domain.

In a nutshell, we incorporate the following amendments:

– We observe the back-up registers, T_0, T_1, T_2 are indeed redundant—the same functionality can be obtained by using actual (S_0), redundant (S_1) and dummy registers (S_2) only. So, we do not use them.
– We remove λ altogether. Instead of running the meaningful/dummy rounds in a random order, we simplify our scheme by running them in a deterministic order. Within a loop of i from 1 to η, we run the redundant round, actual round and the dummy round (in this order). After this, we compute the nested round function on a loop of j, counting from 1 to z ($\geq \xi$).
– We only infect the dummy register within loop over i, that goes from 1 to η (neither S_0, nor S_1 is infected within this loop). This makes sure that S_2 contains β (in case of no fault) or something other than β (in case of a fault). Introduction of nested round function ensures S_2 gets sufficient diffusion, no matter which round attacker targets. At the end, we infect S_0 by $S_0 \leftarrow S_0 \oplus S_2 \oplus \beta$. Since, $S_2 \oplus \beta$ works like a one time pad (similar to the case of Type I countermeasures), in case of a fault, attacker gets no information on S_0.
– The reason we do not infect S_0 within the loop of i is, attacker can simply bypass any infection in S_0 within the loop of i by injecting fault at the last actual round.
– We do not infect the redundant state S_1 at all. This is inspired from the observation that, attacker actually gets the infected S_0 (and not S_1).
– Instead of computing δ (i.e., $\delta \leftarrow S_0 \oplus S_1$; $S_2 \leftarrow S_2 \oplus \delta$) explicitly; we do the computation implicitly ($S_2 \leftarrow S_0 \oplus S_2$; $S_2 \leftarrow S_1 \oplus S_2$). This modification helps to prevent any attack on δ (such as resetting it to $\mathbf{0}$ before it updates S_2; or skipping the XOR of δ with S_2).
– We choose $\sigma(\cdot)$ as the identity function for simplicity.[7] Although we do not compute δ ($= S_0 \oplus S_1$) explicitly; technically speaking, we XOR ($S_0 \oplus S_1$) directly to S_2.

One motivation for keeping random ordering of dummy/meaningful round (through λ), based on [23], is to have a somewhat protection against side channel analysis. The authors speculate that, since the operations in dummy rounds mimic that of a meaningful round; attacker cannot identify a meaningful round (with probability $> \frac{1}{2}$). However, this does not amount for secure enough protection against side channel analysis. For example, say, the attacker can get λ itself by side channel analysis; thus the assumed SCA security does not hold anymore. At the same time, we emphasize that; making the meaningful/dummy round computation deterministic does not incur additional vulnerability from the perspective of DFA. Another reason for keeping the random ordering would be to reduce the success rate of the fault injecting attacker [36,37]. In this regard, we would like to note that our infection schemes are independent on the success rate of the attacker. Even if the attacker succeeds to inject a fault with probability 1, the security claims of our schemes would still hold. Hence, we believe making the execution sequence random would not add any extra protection.

In Algorithm 6, we present an algorithmic description of our patched scheme. As already described, here we iterate over a loop of i, from 1 through η. Within this loop, we deterministically compute the actual, redundant and the dummy round, in this order. After the actual and the redundant round computations are done, we update the dummy register S_2 ($S_2 \leftarrow S_0 \oplus S_1 \oplus S_2$: Lines 5, 6). S_2 is also initialized with β (Line 1). Hence, any fault in actual or redundant round will result S_2 to contain a value different from β. This change in S_2 will affect the dummy round computation, ensuring S_2 does not contain β (Line 7). In case of no fault, S_2 will continue to contain β. When this loop is over; the following loop over j, from 1 to z ($\geq \xi$), will ensure the diffusion is spread sufficiently on S_2 (in case of fault)/S_2 still contains β (otherwise) by additional dummy rounds. Finally, S_0 is returned after it is XORed with $S_2 \oplus \beta$ (Line 10). It can be noted that, injecting fault anywhere during the loop over j will not yield useful information to the attacker.

Now let us revisit the patched scheme for its usability against CFA (Sect. 2.6). When Eve injects fault at any meaningful round, it sets a β' ($\neq \beta$) at S_2; so S_2 is now infected (Lines 5, 6). The infection spreads on S_2 over subsequent rounds. Even after several rounds of iteration, if S_0 becomes equal to S_1 (which is the case CFA wants to uti-

[7] It is recommended in [23] to use a nonlinear function as $\sigma(\cdot)$. However, we believe this is not a necessary condition.

lize), S_2 will still be infected (i.e., S_2 will contain something $\neq \beta$). The next loop over j will also spread the infection to S_2. Hence, S_2 will be random (Line 10); this means, S_0 will be XORed by a random vector. Thus, in case of a CFA, attacker will always get a random (infected) output and cannot mount the attack. To the best of our knowledge, this is the first CFA countermeasure reported in the literature.

In the light of [36] (i.e., resilience of infective countermeasures against the instruction skip attack), one may notice the interesting observations: Under non-faulty situation, Lines 5–7 of Algorithm 6 are idempotent. So, it does not affect the non-faulty computation of the cipher if these lines are iterated, say, μ (> 1) times:

```
1: for $m = 1; m \leq \mu + z; m \leftarrow m + 1$ do
2:     $S_2 \leftarrow S_0 \oplus S_2$
3:     $S_2 \leftarrow S_1 \oplus S_2$
4:     $S_2 \leftarrow \texttt{RoundFunction}_0(S_2) \oplus k_0$
```

In case of a fault, this will help the infection to propagate further in S_2. Similarly, the Line 10 is still vulnerable to one instruction skip. As this line is idempotent under non-faulty situation, we can substitute it by:

```
1: for $m = 1; m \leq \mu; m \leftarrow m + 1$ do
2:     $S_0 \leftarrow S_0 \oplus S_2 \oplus \beta$
                                  ▷ $S_2 = \texttt{RoundFunction}^z(S_2, k_0)$
```

Table 3 Software benchmarking results (clock cycles, code size) for our patch

Construction	Countermeasure	Architecture	Clock cycles	Code size
AES Encryption (unprotected)	–	AVR	1.00	1.00
		MSP	1.00	1.00
AES Encryption (protected by Algorithm 6)	11-nested round	AVR	3.25	2.94
		MSP	3.69	2.31
	4-nested round	AVR	2.90	2.94
		MSP	3.13	2.58

This ensures that the Line 10 is protected against $\mu - 1$ instruction skip attack. Altogether, it can be claimed that the overall construction can withstand up to $\mu - 1$ instruction skips. This can pave the pathway to an infective countermeasure which is guaranteed to have a certain protection against instruction skips. We believe the arguments presented in algorithmic terms are still valid when the proposed strategy is expressed in terms of instructions instead. The underlying property that is used here still applies when the algorithm is expressed as an equivalent sequence of instructions. Hence, the implementation-level idempotency condition can be ascertained from the algorithmic description.

Benchmarking

In Table 3, we present the software benchmarking results (clock cycles and code size) for the basic patched version of the LatinCrypt'12 scheme (Algorithm 6), for AVR and

Algorithm 6 Our patched version of LatinCrypt'12 countermeasure (for SPN)

```
Input: $\begin{cases} P \\ \beta; k_0 \\ \text{round keys } k_j; j = 1, \ldots, \eta \text{ derived from } K \\ z\ (\geq \xi) \end{cases}$
                                          ▷ $\texttt{RoundFunction}_0(\beta) = \beta \oplus k_0$
Output: $\begin{cases} E_K(P) & \text{if no fault} \\ \text{random} & \text{otherwise} \end{cases}$
1:  $S_0 \leftarrow P$                                      ▷ Actual state
    $S_1 \leftarrow P$                                      ▷ Redundant state
    $S_2 \leftarrow \beta$                                  ▷ Dummy state
2:  for $i \leftarrow 1; i \leq \eta; i \leftarrow i + 1$ do
3:      $S_0 \leftarrow \texttt{RoundFunction}_i(S_0) \oplus k_i$          ▷ Actual round
4:      $S_1 \leftarrow \texttt{RoundFunction}_i(S_1) \oplus k_i$          ▷ Redundant round
5:      $S_2 \leftarrow S_0 \oplus S_2$
6:      $S_2 \leftarrow S_1 \oplus S_2$
7:      $S_2 \leftarrow \texttt{RoundFunction}_0(S_2) \oplus k_0$          ▷ Dummy round
8:  for $j \leftarrow 1; j \leq z; j \leftarrow j + 1$ do
9:      $S_2 \leftarrow \texttt{RoundFunction}_0(S_2) \oplus k_0$          ▷ Dummy round
10: $S_0 \leftarrow S_0 \oplus S_2 \oplus \beta$          ▷ $S_2 = \texttt{RoundFunction}^z(S_2, k_0)$
11: return $S_0$
```

MSP architectures. We take AES encryption as the underlying cipher. As before, we use the source codes from the FELICS [16] tool. However, unlike the benchmarking in Type I schemes, which are done as standalone implementations; this one is done together with AES actual and redundant encryptions (because of the intrinsic nature of this scheme). We choose two different nested round functions; corresponding to 11 and 4, respectively. As before, we do not consider the cost due to generation of external entropy. Also, we assume β and k_0 are provided to the algorithm. The figures given here are taken after averaging multiple runs with different test vectors; and are relative to unprotected AES in $\times$ 1.00 unit (the numeric figures for this implementation can be found in Table 1). For the 11-nested rounds, the relative clock cycles is slightly bigger than $3\times$; this is due to the fact that now AES is running 3 times together with few other computations.[8] The relative code sizes are less than $3\times$, as part of the same code can be reused (such as the SBox). For the 4-nested round, the relative clock cycle is less than 3 for AVR as now less than $3\times$ AES is computed, but the relative code size remains roughly the same as 11-nested rounds. However, for MSP architecture, although the relative clock cycles reduces from the 11-nested rounds, the relative code size has a slight increase ($2.31\times$ in 11-nested to $2.58\times$ in 4-nested). This is probably due to some optimization done by the compiler, which is more efficient when almost three identical computations are running, compared to the case where one of them is slightly different.

Since the same hardware (that is used to design the underlying cipher) can be reused to build this scheme (with degraded throughput), we do not provide any separate hardware benchmarking here. Roughly, it can be estimated that 11-nested round countermeasure is comparable to $4\times$ unprotected AES, whereas 4-nested would be around $3.12\times$ unprotected AES. So, the throughputs can be estimated to be downgraded roughly by this scale.

5 Conclusion

In this work, we study the infective countermeasures, which are used to protect ciphers against certain classes of fault attacks (including the most common differential fault attacks in symmetric key cryptography). This is the first work studying these countermeasures in details. Apart from providing a systematic classification, we show several new results. These results range from proposing new type of countermeasures (that relies on already established standards instead of the usual ad hoc approaches), to proposing new lightweight schemes (in both software and hardware). We also show a flaw in the scheme from [44] published at CHES'14. Moreover, we fix a broken scheme with a little amendment. Our work underlies the need for more rigorous analysis of not only the standard ciphers, but also the fault countermeasures.

The main differences between the two types of infective countermeasures are listed here. First, unlike Type I countermeasures, cipher level countermeasures do not construct a separate diffusion function $\tau_R(\cdot)$, rather they rely on the round function of the underlying cipher to infect the actual computation. Second, cipher level countermeasures rely on sensing the infection round by round, whereas Type I countermeasures let both the actual and redundant executions to finish before infection. Third, Type II countermeasures compute a so called dummy round, which is idempotent in case of no fault, which is missing in Type I. Fourth, the function $\tau_{(\cdot)}(\cdot)$ in Type I is recommended to be highly nonlinear; in contrast, the nonlinearity in Type II countermeasures come from the nonlinearity of $\texttt{RoundFunction}_0(\cdot)$, (not from $\sigma(\cdot)$); hence, $\sigma(\cdot)$ can be linear. Fifth, a Type I construction can be adopted to non-SPN designs, such as a Feistel network-based block cipher or a stream cipher easily; whereas it may be rather non-trivial to do the same with a Type II construction.

Focusing only on DFA countermeasures gives us the advantage to explore the domain more extensively. For this purpose, the side channel protection is not directly considered within the scope. Our schemes can be protected by incorporating standard SCA countermeasures, refer to Sect. 2.7 for more details.

Multiple works can be considered in the future scope, here we list a few. First, one may look into constructing an integrated DFA and side channel countermeasure. Second, protecting against double faults (that identically affect both the actual and the redundant ciphers) is an interesting problem. Third, designing infective countermeasures which are more resilient against other types of faults, such as instruction skip or double fault or IFA (SIFA, in particular), can be an interesting direction to pursue. Finally, a cipher may be constructed which is more suitable to deploy together with an infective countermeasure (possibly equipped with an error detecting code).

References

1. Aghaie, A., Moradi, A., Rasoolzadeh, S., Shahmirzadi, A.R., Schellenberg, F., Schneider, T.: Impeccable circuits. Cryptology ePrint archive, report 2018/203 (2018). https://eprint.iacr.org/2018/203. Accessed 18 Jan 2020
2. Baksi, A., Bhasin, S., Breier, J., Khairallah, M., Peyrin, T.: Protecting block ciphers against differential fault attacks without re-keying. In: 2018 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2018, Washington, DC, USA,

[8] The actual and the redundant computations are basically identical except near the very end; hence, the compiler possibly optimizes the code that reduces the clock cycles.

April 30–May 4, 2018, pp. 191–194 (2018). https://doi.org/10.1109/HST.2018.8383913

3. Banik, S., Bogdanov, A.: Cryptanalysis of two fault countermeasure schemes. In: Progress in Cryptology—INDOCRYPT 2015—16th International Conference on Cryptology in India, Bangalore, India, December 6–9, 2015, Proceedings, pp. 241–252 (2015). https://doi.org/10.1007/978-3-319-26617-6_13
4. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. IACR cryptology ePrint archive , vol. 2004, p. 100 (2004). http://dblp.uni-trier.de/db/journals/iacr/iacr2004.html#Bar-ElCNTW04
5. Barenghi, A., Breveglieri, L., Koren, I., Naccache, D.: Fault injection attacks on cryptographic devices: theory, practice, and countermeasures. Proc. IEEE **100**(11), 3056–3076 (2012). https://doi.org/10.1109/JPROC.2012.2188769
6. Barenghi, A., Breveglieri, L., Koren, I., Pelosi, G., Regazzoni, F.: Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In: Proceedings of the 5th Workshop on Embedded Systems Security, WESS 2010, Scottsdale, AZ, USA, October 24, 2010, p. 7 (2010). https://doi.org/10.1145/1873548.1873555
7. Battistello, A., Giraud, C.: Fault analysis of infective AES computations. In: 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013, pp. 101–107 (2013). https://doi.org/10.1109/FDTC.2013.12
8. Battistello, A., Giraud, C.: A note on the security of CHES 2014 symmetric infective countermeasure. In: 7th International Workshop on Constructive Side-Channel Analysis and Secure Design, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers, pp. 144–159 (2016). https://doi.org/10.1007/978-3-319-43283-0_9
9. Beierle, C., Leander, G., Moradi, A., Rasoolzadeh, S.: Craft: lightweight tweakable block cipher with efficient protection against DFA attacks. IACR Trans. Symmetric Cryptol. **2019**(1), 5–45 (2019). https://doi.org/10.13154/tosc.v2019.i1.5-45
10. Biham, E., Shamir, A.: Differential cryptanalysis of des-like cryptosystems. In: 10th Annual International Cryptology Conference on Advances in Cryptology—CRYPTO '90, Santa Barbara, California, USA, August 11–15, 1990, Proceedings, pp. 2–21 (1990). https://doi.org/10.1007/3-540-38424-3_1
11. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Kaliski Jr., B.S. (ed.) Advances in Cryptology-CRYPTO '97. Lecture Notes in Computer Science, vol. 1294, pp. 513–525. Springer, Berlin (1997). https://doi.org/10.1007/BFb0052259
12. Blömer, J., Krummel, V.: Fault based collision attacks on AES. In: Third International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2006, Yokohama, Japan, October 10, 2006, Proceedings, pp. 106–120 (2006). https://doi.org/10.1007/11889700_11
13. Blömer, J., Otto, M.: Wagner's attack on a secure CRT-RSA algorithm reconsidered. In: Third International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2006, Yokohama, Japan, October 10, 2006, Proceedings, pp. 13–23 (2006). https://doi.org/10.1007/11889700_2
14. Clavier, C.: Secret external encodings do not prevent transient fault analysis. In: 9th International Workshop on Cryptographic Hardware and Embedded Systems—CHES 2007, Vienna, Austria, September 10-13, 2007, Proceedings, pp. 181–194 (2007). https://doi.org/10.1007/978-3-540-74735-2_13
15. Daemen, J., Dobraunig, C., Eichlseder, M., Gross, H., Mendel, F., Primas, R.: Protecting against statistical ineffective fault attacks. Cryptology ePrint archive, Report 2019/536 (2019). https://eprint.iacr.org/2019/536. Accessed 18 Jan 2020
16. Dinu, D., Biryukov, A., Großschädl, J., Khovratovich, D., Perrin, L., Corre, Y.L., Biryukov, A.: Triathlon of lightweight block ciphers for the internet of things (2015). http://csrc.nist.gov/groups/ST/lwc-workshop2015/papers/session7-dinu-paper.pdf. Accessed 18 Jan 2020
17. Dobraunig, C., Eichlseder, M., Korak, T., Mangard, S., Mendel, F., Primas, R.: SIFA: exploiting ineffective fault inductions on symmetric cryptography. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(3), 547–572 (2018). https://doi.org/10.13154/tches.v2018.i3.547-572
18. Dobraunig, C., Koeune, F., Mangard, S., Mendel, F., Standaert, F.: Towards fresh and hybrid re-keying schemes with beyond birthday security. In: 14th International Conference on Smart Card Research and Advanced Applications, CARDIS 2015, Bochum, Germany, November 4–6, 2015. Revised Selected Papers, pp. 225–241 (2015). https://doi.org/10.1007/978-3-319-31271-2_14
19. Feng, J., Chen, H., Li, Y., Jiao, Z., Xi, W.: A framework for evaluation and analysis on infection countermeasures against fault attacks. IEEE Trans. Inf. Forensics Secur **15**, 391–406 (2020). https://doi.org/10.1109/TIFS.2019.2903653
20. Fournier, J., Rigaud, J.B., Bouquet, S., Robisson, B., Tria, A., Dutertre, J.M., Agoyan, M.: Design and characterisation of an aes chip embedding countermeasures. Int. J. Intell. Eng. Inform. **1**(3/4), 328–347 (2011). https://doi.org/10.1504/IJIEI.2011.044101
21. Ghosh, S., Saha, D., Sengupta, A., Chowdhury, D.R.: Preventing fault attacks using fault randomization with a case study on AES. In: 20th Australasian Conference on Information Security and Privacy , ACISP 2015, Brisbane, QLD, Australia, June 29–July 1, 2015, Proceedings, pp. 343–355 (2015). https://doi.org/10.1007/978-3-319-19962-7_20
22. Ghosh, S., Saha, D., Sengupta, A., Chowdhury, D.R.: Preventing fault attacks using fault randomisation with a case study on AES. IJACT **3**(3), 225–235 (2017). https://doi.org/10.1504/IJACT.2017.10007295
23. Gierlichs, B., Schmidt, J., Tunstall, M.: Infective computation and dummy rounds: fault protection for block ciphers without check-before-output. In: Progress in Cryptology—LATINCRYPT 2012—2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7–10, 2012. Proceedings, pp. 305–321 (2012). https://doi.org/10.1007/978-3-642-33481-8_17
24. He, W., Breier, J., Bhasin, S.: Cheap and cheerful: a low-cost digital sensor for detecting laser fault injection attacks. In: 6th International Conference on Security, Privacy, and Applied Cryptography Engineering, SPACE 2016, Hyderabad, India, December 14–18, 2016, Proceedings, pp. 27–46 (2016). https://doi.org/10.1007/978-3-319-49445-6_2
25. Joye, M., Ciet, M.: Practical fault countermeasures for Chinese remaindering based RSA. In: Third International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC (2005)
26. Joye, M., Manet, P., Rigaud, J.B.: Strengthening hardware aes implementations against fault attacks. IET Inf. Secur. **1**(3), 106 (2007)
27. Karri, R., Kuznetsov, G., Gössel, M.: Parity-based concurrent error detection of substitution-permutation network block ciphers. In: 5th International Workshop on Cryptographic Hardware and Embedded Systems—CHES 2003, Cologne, Germany, September 8–10, 2003, Proceedings, pp. 113–124 (2003). https://doi.org/10.1007/978-3-540-45238-6_10
28. Keliher, L., Sui, J.: Exact maximum expected differential and linear probability for two-round advanced encryption standard. IET Inf. Secur. **1**(2), 53–57 (2007)
29. Kim, C.H., Quisquater, J.J.: Fault attacks for CRT based RSA: new attacks, new results, and new countermeasures. In: Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.J. (eds.) Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems, pp. 215–228. Springer, Berlin (2007)

30. Krämer, J., Stüber, A., Kiss, Á.: On the optimality of differential fault analyses on CLEFIA. IACR Cryptology ePrint Archive, vol. 2014, p. 572 (2014). http://eprint.iacr.org/2014/572. Accessed 18 Jan 2020
31. Kulikowski, K., Karpovsky, M., Taubin, A.: Robust codes for fault attack resistant cryptographic hardware. In: 2nd International Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 1–12 (2005)
32. Kumar, S.V.D., Patranabis, S., Breier, J., Mukhopadhyay, D., Bhasin, S., Chattopadhyay, A., Baksi, A.: A practical fault attack on arx-like ciphers with a case study on chacha20. In: 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2017, Taipei, Taiwan, September 25, 2017, pp. 33–40 (2017). https://doi.org/10.1109/FDTC.2017.14
33. Lomné, V., Roche, T., Thillard, A.: On the need of randomness in fault attack countermeasures—application to AES. In: 2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012, pp. 85–94 (2012). https://doi.org/10.1109/FDTC.2012.19
34. Mangard, S., Oswald, E., Popp, T.: Power analysis attacks—revealing the secrets of smart cards. Springer, Berlin (2007)
35. Otto, M.: Fault attacks and countermeasures. Ph.D. thesis, Institut für Informatik, Universität Paderborn (2004). https://pdfs.semanticscholar.org/36e0/5d6a59a8b2abc9ef21098e404fc47a81d622.pdf
36. Patranabis, S., Chakraborty, A., Mukhopadhyay, D.: Fault tolerant infective countermeasure for AES. In: 5th International Conference on Security, Privacy, and Applied Cryptography Engineering, SPACE 2015, Jaipur, India, October 3–7, 2015, Proceedings, pp. 190–209 (2015). https://doi.org/10.1007/978-3-319-24126-5_12
37. Patranabis, S., Chakraborty, A., Mukhopadhyay, D.: Fault tolerant infective countermeasure for AES. J. Hardw. Syst. Secur. **1**(1), 3–17 (2017). https://doi.org/10.1007/s41635-017-0006-1
38. Piret, G., Quisquater, J.: A differential fault attack technique against SPN structures, with application to the AES and KHAZAD. In: 5th International Workshop on Cryptographic Hardware and Embedded Systems—CHES 2003, Cologne, Germany, September 8–10, 2003, Proceedings, pp. 77–88 (2003). https://doi.org/10.1007/978-3-540-45238-6_7
39. Saha, D., Mukhopadhyay, D., Chowdhury, D.R.: A diagonal fault attack on the advanced encryption standard. IACR Cryptology ePrint Archive **2009**, 581 (2009). http://eprint.iacr.org/2009/581
40. Saha, S., Jap, D., Roy, D.B., Chakraborti, A., Bhasin, S., Mukhopadhyay, D.: A framework to counter statistical ineffective fault analysis of block ciphers using domain transformation and error correction. IEEE Trans. Inf. Forensics Secur. (2019). https://doi.org/10.1109/TIFS.2019.2952262
41. Selmke, B., Heyszl, J., Sigl, G.: Attack on a DFA protected AES by simultaneous laser fault injections. In: 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 36–46 (2016). https://doi.org/10.1109/FDTC.2016.16
42. Shahmirzadi, A.R., Rasoolzadeh, S., Moradi, A.: Impeccable circuits II. Cryptology ePrint Archive, Report 2019/1369 (2019). https://eprint.iacr.org/2019/1369. Accessed 18 Jan 2020
43. Song, L., Hu, L.: Differential fault attack on the PRINCE block cipher. In: Lightweight Cryptography for Security and Privacy - Second International Workshop, LightSec 2013, Gebze, Turkey, May 6-7, 2013, Revised Selected Papers, pp. 43–54 (2013). https://doi.org/10.1007/978-3-642-40392-7_4
44. Tupsamudre, H., Bisht, S., Mukhopadhyay, D.: Destroying fault invariant with randomization—a countermeasure for AES against differential fault attacks. In: 16th International Workshop on Cryptographic Hardware and Embedded Systems—CHES 2014, Busan, South Korea, September 23–26, 2014. Proceedings, pp. 93–111 (2014). https://doi.org/10.1007/978-3-662-44709-3_6
45. Yen, S.-M., Kim, S., Lim, S., Moon, S.-J.: RSA speedup with Chinese remainder theorem immune against hardware fault cryptanalysis. IEEE Trans. Comput. **52**(4), 461–472 (2003). https://doi.org/10.1109/TC.2003.1190587
46. Yen, S.M., Kim, D.: Cryptanalysis of two protocols for RSA with CRT based on fault infection. In: Third International Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC (2004)