To Infect Or Not To Infect: A Critical Analysis Of Infective Countermeasures In Fault Attacks

Anubhab Baksi[1], Dhiman Saha[2], and Sumanta Sarkar[3]

[1] School of Computer Science & Engineering,
Nanyang Technological University, Singapore

[2] Department of Electrical Engineering & Computer Science,
IIT Bhilai, India

[3] TCS Innovation Labs, Hyderabad, India

anubhab001@e.ntu.edu.sg, dhiman@iitbhilai.ac.in, sumanta.sarkar1@tcs.com

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 their ability in inhibiting differential fault attacks without explicitly detecting the fault. We observe that despite being adopted over a decade ago, a systematic study 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, propose new design based on already established standards, refute a security claim made by a scheme proposed in CHES'14 and re-instantiate another scheme which is deemed broken by proposing a simple patch.

Keywords: fault countermeasure, infection, differential fault analysis

1 Introduction

Fault attacks are becoming a real threat to small scale devices performing a cryptographic operation. This type of attack forces a certain device to work under suboptimal condition resulting in erroneous calculations, which is then exploited. *Differential fault analysis* or *differential fault attack* (DFA) [7], 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 of 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 gives 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 [17]; or *passive*, where a shield to block external interference [3] is used.
(ii) Using redundancy in computation. These type of countermeasures commonly duplicate (fully/partially) the circuit, followed by a certain procedure which dictates what to do in case a fault is sensed.
(iii) Using protocol level technique. Here the underlying protocol ensures that the conditions required for a successful fault happens with low probability, e.g., [12,1].

Our interest lies in the second category of fault protection (further elaborated in Section 2). In particular, we focus on 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 non-zero 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 is proposed to counter the shortcomings of the previously proposed *detective countermeasures* (also referred to as *detection based countermeasures*, see Section 2.2 for more details). Incidentally however, most of the infective countermeasures proposed in the literature are broken quite soon (it is even stated in [4] 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.

Since we perform a critical analysis on DFA, we choose to ignore the *side channel* adversarial model (such as power or electro-magnetic analysis). Existing SCA countermeasures can be applied on top of our proposals, hence no special treatment is necessary.

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. For convenience, we call them *type I* (Section 2.3) and *type II* (Section 2.3) countermeasures. The basic difference between them is that, the type I schemes allow the full computations of the actual and redundant ciphers to run, then computes Δ; in contrast, type II schemes computes the difference on the fly – the difference is computed after each round. Type I countermeasures are further divided into two sub-categories based on existing literature; *multiplication based* (Section 3.1) and *derivative based* (Section 3.2). Type II countermeasures are also referred to as *cipher level* countermeasures (described in Section 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 [24] (Section 3.1). Also, we propose software friendly options for the scheme proposed in [15], (Section 3.2) based on existing ciphers.
- For the type II schemes, we show (in Section 4.1) the CHES'14 infective countermeasure proposed in [28] is not any infective countermeasure at all; thereby refuting its security claim and also showing its weakness against DFA. This also refutes the security claim made by the modified infective countermeasure in [26]. The basic principle of this paper is same as that of [28]; certain modifications are done on [28] 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 [16] (LatinCrypt'12), which is broken in [4,28]. We propose a simple patch (in Section 4.2) on the [16] scheme that resists the attacks presented in [4,28].

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. This fault, in 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*.

Now, flipping one/few bit(s) of a register works as a simplified version of the classical differential attack (DA). In DA, the difference is inserted through the chosen plaintexts (hence, works at the beginning of

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

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 is in contrast to the *stuck-at/hard* fault model where particular bit(s) are permanently stuck with the fault value (one may refer to [9] for an example). The stuck-at model assumes more control for the attacker, and is outside the scope of DFA. Similar to this, modification of operation is also a strong attacker model (such as instruction skip in [23]), 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 [19], non-linear $[n, k]$ codes [22], etc.

2.3 Rise 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., [21]). 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 [25] that, one bit flip is an incidence which may happen "by chance".

The idea of *infection* is first proposed in the context of public key cryptography [29]. The concept is later adopted in symmetric key setting [18,13,16,24,28,26,14]. The infective countermeasures proposed in [18,13] do not involve randomness, and are attacked in [24]. The authors in [24] also speculates, 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.

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

- *Actual & 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 $C = E^1_K(P)$ and $C' = E^2_K(P)$, respectively (E is the underlying cipher parametrized by the secret key K with input P). Output from the actual cipher is later infected, and made available to the attacker. The notations, C and C', are used in this document 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 repeat the exactly same fault, in exactly the same location and during exactly the same round on one particular device; as many times she wants (as long as it is

[2]Not to be confused with the case where the fault flips only one bit.

practicable). Thus, we give the attacker to repeat exactly 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 j^{th} 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`;
  $$\texttt{RoundFunction}_1(p) = p;$$
  $$\texttt{RoundFunction}_j(p) = \texttt{MixColumns}(\texttt{ShiftRows}(\texttt{SubBytes}(p))) \text{ for } j = 2, \ldots, 10;$$
  and
  $$\texttt{RoundFunction}_{11}(p) = \texttt{ShiftRows}(\texttt{SubBytes}(p)).$$
  Basically, when the j^{th} round key k_j is inserted with $\texttt{RoundFunction}_j(\cdot)$, it gives the actual j^{th} 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} [20]. 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.

Type I This type of countermeasures allow the two computations, $E^1_K(\cdot)$ and $E^2_K(\cdot)$ 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.

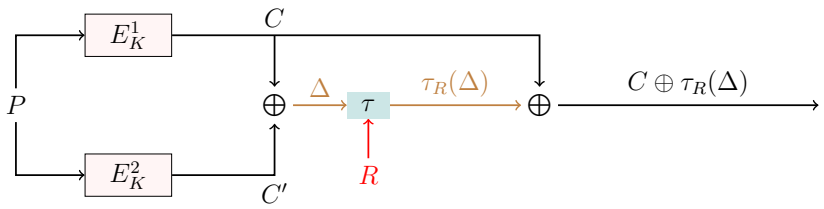

Fig. 1: Type I infective countermeasures

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 $\mathbb{F}_2^n$.

[3]We assume the most common fault attacker 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.

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 [24] ($\tau_R(\Delta) = R \cdot \Delta$ over GF(2^n) multiplication) or [15] ($\tau_R(\Delta) = N(R) \oplus N(R \oplus \Delta)$, where N is defined as a non-linear hybrid cellular automata).

Type II (Cipher Level) This new type of infection is introduced in [16], we also use the alias of cipher level countermeasure. 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. In the *actual round*, S_0 is updated only; whereas in the *redundant round*, S_1 is only updated; and 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 $\mathbf{0}$. Note that, unlike the $\tau_{(\cdot)}(\cdot)$ function in type I schemes, it does not take any random input. As for the choice of $\sigma(\cdot)$, inversion in GF(2^8) is proposed per SBox (8-bit) of `AES` in [16]; whereas [28,26] propose the $n : 1$ OR gate. 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 Section 4 for more details).

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 S_2 is updated (with influence from Δ) by a clever rule such that:

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

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 are not deviated from its actual computation – this ensures a proper execution of the cipher. In our patched version of LatinCrypt'12, we do not update S_0 and S_1 during meaningful rounds (see Section 4.2 for more details).

To determine whether a dummy or a meaningful round will take place, the authors in [16] 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, ($1 \le j \le \eta$) 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 [16,**?**], i is initialized by 1 (odd); which means, a redundant round always precedes the corresponding actual round. In the modified scheme in [26], however, the order of execution of actual and redundant rounds are not predetermined.

2.4 Necessity and Sufficiency of Randomness

It is well-known from [24] 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`.

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:

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

[4] In our patched version of [16] (Section 4.2), we do not explicitly compute δ.

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 uniform 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 defective 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 non-zero mutual information regarding the secret key) to reduce the mutual information to zero.

2.5 Scope of Infective Countermeasures

The working procedure of infective countermeasures is, it relies on the (non-zero) 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 *double fault*.

However, repeating the identical fault in spatial domain would require a very strong adversary model. Such models, although used in literature (the only case we know is, [27]), 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 [2] 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* [9], where knowledge of the faulty output is not required.

Another type of fault attack, known as *collision fault attack* (CFA) [8], 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 (Section 4.2) can indeed protect against such an attack.

In a very recent paper, the authors propose a new block cipher named `CRAFT` that inherently helps to mount one particular type of detection mechanism in hardware [6]. When implemented with a particular error detecting code together with the duplicated cipher, the overall design takes less area in hardware; compared to common block ciphers when duplicated and implemented with that particular error detecting code. Hence, the duplicated cipher along with the code can detect faults up to a certain extend. However, detecting a fault is only one part of the story; the real problem lies in what will be done if a fault is detected — which is not addressed in the paper. In a real life scenario, this may turn out to be a vital problem; in part because there is a common belief that the detection mechanism can be bypassed if no external randomness is used [24]. We believe, an infective countermeasure may be used with this fault detection mechanism to make the overall design resistant to DFA.

3 Type I Constructions

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 [24] (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).

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.

Algorithm 1: Multiplication based infective countermeasure: FDTC'12

Input: C, C'; R_0, R_1, R_2 ▷ None of R_0, R_1, R_2 equals $\mathbf{0}$ or $\mathbf{1}$
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** $g \oplus R_0$

While the original proposal remains unbroken so far; this lightweight alternative is broken soon afterwards in [4], we describe the attack here. Assume 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 block (C), whereas the other is the faulty output block ($C \oplus \Delta$). 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 sixteen times). For the original GF(2^{128}) multiplication for `AES`, it would require 2^{128} repeats of the same fault and storage; making it impractical.

One may note that, the authors [24] 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 non-zero 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. In this regard, we observe and argue that if the attacker has access to side channels 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 totally cutting off the need for a DFA. It seems futile to protect only Δ against such kind of combined analysis which, we feel, unnecessarily complicates the scheme. In any case, a full SCA protection is needed (not just Δ). Following this, we focus our analysis keeping in mind protection against exclusively differential fault and other fault attacks; and keep side channel protection out of scope. SCA protection, if deemed necessary, can be implemented on top of our schemes by existing countermeasures.

Our Hardware Friendly Alternatives As GF(2^n) multiplication consumes much resources in hardware (where n is generally 128), here 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; therefore, 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 $\boldsymbol{a} = (a_0, a_1, a_2, \ldots, a_{n-1})$, recursively as:

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

For example, with the vector $\boldsymbol{a} = (a_0, a_1, a_2, a_3)$; we have $\rho^1(\boldsymbol{a}) = (a_3, a_0, a_1, a_2)$; $\rho^2(\boldsymbol{a}) = \rho^1(\rho^1(\boldsymbol{a}))$ $= \rho^1(a_3, a_0, a_1, a_2) = (a_2, a_3, a_0, a_1)$; 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 i^{th} row is the i^{th} 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).

Algorithm 2(a): Multiplication based infective countermeasure: Our first variant

Input: $C; C'$ ▷ $|C|, |C'| = n$
Output: C if no fault; random, otherwise

$\Delta \leftarrow C \oplus C'$ ▷ Δ is represented as an n-bit vector
for $i \leftarrow 0; i < n; i \leftarrow i+1$ **do**
  $R_i \xleftarrow{\$} \mathbb{F}_2^n$
$M \leftarrow \begin{bmatrix} R_0 \\ R_1 \\ \vdots \\ R_{n-1} \end{bmatrix}$
$a \leftarrow M \cdot \Delta$ ▷ Multiplication is over GF(2)
return $C \oplus a$

Algorithm 2(b): Multiplication based infective countermeasure: Our second variant

Input: $C; C'$ ▷ $|C|, |C'| = n$
Output: C if no fault; random, otherwise

$\Delta \leftarrow C \oplus C'$ ▷ Δ is represented as an n-bit vector
$R_0, R_1 \xleftarrow{\$} \mathbb{F}_2^n$
$M \leftarrow \begin{bmatrix} R_0 \\ \rho^1(R_0) \\ \vdots \\ \rho^{n-2}(R_0) \\ R_1 \end{bmatrix}$
$b \leftarrow M \cdot \Delta$ ▷ Multiplication is over GF(2)
return $C \oplus b$

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 is same as a circulant matrix, except the last row (it would be a circulant matrix if last row would be equal to $(n-1)^{\text{th}}$ 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:

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

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,

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

So, just by XORing all the bits, attacker is able to deduce the parity of Δ.

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 non-linear 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 [14], the authors propose such a construction by choosing a quadratic N. However, this scheme is subsequently broken in [2]. Recall from Section 3 that $\tau_R(\cdot)$ has to be non-invertible. This claim is shown to be incorrect in [2], 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 [14] opt for more complex and high degree function $N(\cdot)$ based on non-linear cellular automata in [15]; and this is the only yet unbroken proposal in this category.

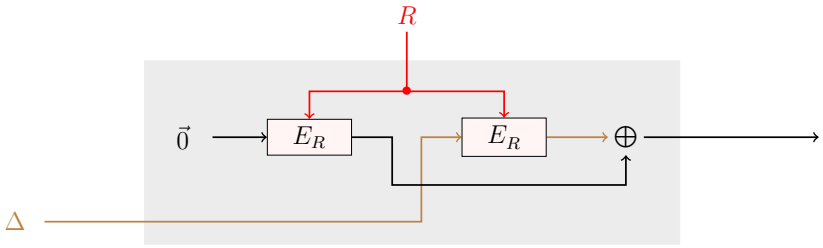

Fig. 2: Derivative based construction of $\tau_R(\Delta)$: Our design

Our Construction 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 2 shows the construction.

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

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 [15] 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 or MAC can be used in place of E. For a hash function H (which does not involve a key), we propose to use $\tau_R(\Delta) = H(R) \oplus H(R \oplus \Delta)$.

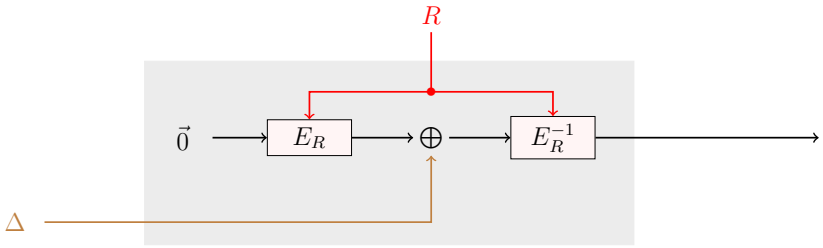

Fig. 3: Encrypt-XOR-decrypt based design of $\tau_R(\Delta)$

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	1.00	1.00	1.00	–
GF(2^{128}) Multiplication		2.28	2.57	0.28	0.47	[24], Section 3.1
Algorithm 2(b)		41.86	34.06	0.28	0.35	Section 3.1
Cellular Automata		> 2700	> 3400	0.82	0.88	[15], Section 3.2
`AES` Encryption Derivative	$z = 11$	1.65	1.81	1.42	1.35	Section 3.2
	$z = 4$	0.64	0.63	1.06	1.36	

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 Figure 3. This may, however, cost more due to inverse key schedule, inverse SBox etc. during decryption.

3.3 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 [1]). We consider `AES` encryption as the underlying cipher (which is deemed to be protected), so we implement it to get a perspective of overhead (so, $n = 128, \eta = 11$).

For the GF(2^{128}) multiplication in [24], 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 Section 3.1), we perform this field multiplication only once (instead of 3 times, as in Algorithm 1). We only implement Algorithm 2(b) (second hardware friendly alternative, see Section 3.1), as the other alternative (Algorithm 2(a)) can be considered a part of it. We also implement the non-linear hybrid cellular automata based design in [15] (Section 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 `Round Function`$_0(\cdot)$, with corresponding `AddRoundKeys`[5]. 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 $1.00\times$ 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 [10] for AVR and MSP architectures.

As it can be seen, the derivative scheme based on `AES` (with $z = 4$) outperforms other schemes in software ($0.64\times$ in AVR and $0.63\times$ 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 [24].

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 microcontroller, 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 comes in handy, e.g., for `AES`; but causes extra overhead for bit-oriented constructions. For the bit-oriented

[5]Not to be confused with notation: z is a parameter, whereas ξ is a constant for a given cipher (e.g., $\xi = 4$ for `AES`). So, z can take any value from $[\xi, \eta]$.

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.

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†	1402 (3)	–
GF(2^{128}) Multiplication	8398 (63)	–	16496 (61)	–	7264 (15)	[24] Section 3.1
Cellular Automata	214 (1)	136†	406 (1)	136†	210†	[15] Section 3.2
Algorithm 2(b)	91†	130†	88†	130†	56†	Section 3.1

(·) indicates % resource utilization † indicates negligible utilization

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 variant, 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

Now, we focus on the type II constructions. At first, the n-bit random vectors β and k_0[6] are so chosen that,

$$\texttt{RoundFunction}_0(\beta) = \beta \oplus k_0.$$

Finding such pairs is easy in SPN ciphers – one can set a β randomly, then compute, $k_0 = \beta \oplus \texttt{RoundFunction}_0(\beta)$. Notice that, β uniquely identifies k_0; so once a k_0 is fixed (based on a β), for β' ($\neq \beta$), $\texttt{RoundFunction}_0(\beta') \oplus k_0$ will not give β. Both β and k_0 are (re-)generated at each invocation of the countermeasure, but are fixed during one invocation. They 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 key can be non-linearly inserted (in contrast to SPN ciphers, where they are always XORed); in which 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. We thus keep the incorporation of 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, $x \cdot \boldsymbol{y}$ for scalar x multiplication with the vector $\boldsymbol{y}$ over GF(2), $\sharp x(\boldsymbol{y})$ for number of occurrence(s) of element x in vector $\boldsymbol{y}$.

We now briefly describe how infection in [16] works (see Algorithm 3). 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.

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:

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

[6] Not to be confused with notation: k_0 is not a round key.

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

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 outputting $\mathbf{0}$ 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 non-zero 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 AES as $\sigma(\cdot)$[7].

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 more. In turn, this will infect subsequent S_0 and S_2 more. Finally, each of $S_0, S_1, S_2, T_0, T_1, T_2$ will contain random values.

Algorithm 3: Infective countermeasure: Latin-Crypt'12 (for SPN)

Input: P; $\beta; k_0$; round keys $k_j; j = 1, \ldots, \eta$ derived from K
▷ $\texttt{RoundFunction}_0(\beta) = \beta \oplus k_0$

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** $i \leq 2\eta$ **do**
5: $\quad \lambda \xleftarrow{\$} \mathbb{F}_2^1$
6: $\quad a \leftarrow (i \wedge \lambda) \oplus ((\neg\lambda) \times 2)$
7: $\quad b \leftarrow \lceil i/2 \rceil \times \lambda$
8: $\quad S_a \leftarrow \texttt{RoundFunction}_b(S_a) \oplus k_b$
9: $\quad T_a \leftarrow S_a \oplus T_2 \oplus \beta$
10: $\quad c \leftarrow (\lambda \wedge (\neg(i \wedge 1))) \cdot \sigma(T_0 \oplus T_1)$
11: $\quad S_2 \leftarrow S_2 \oplus c$
12: $\quad S_0 \leftarrow S_0 \oplus c$
13: $\quad i \leftarrow i + \lambda$
14: $S_0 \leftarrow S_0 \oplus \texttt{RoundFunction}_0(S_2) \oplus k_0 \oplus \beta$
15: **return** S_0

Algorithm 4: Infective countermeasure: CHES'14 (for SPN)

Input: P; $\beta; k_0$; security level t ($\geq 2\eta$); round keys $k_j; j = 1, \ldots, \eta$ derived from K
▷ $\texttt{RoundFunction}_0(\beta) = \beta \oplus k_0$

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** $q \leq t$ **do**
5: $\quad \lambda \leftarrow rstr[q]$
6: $\quad a \leftarrow (i \wedge \lambda) \oplus ((\neg\lambda) \times 2)$
7: $\quad b \leftarrow \lceil i/2 \rceil \times \lambda$
8: $\quad S_a \leftarrow \texttt{RoundFunction}_b(S_a) \oplus k_b$
9: $\quad c \leftarrow (\lambda \wedge (\neg(i \wedge 1))) \wedge \sigma(S_0 \oplus S_1)$
10: $\quad d \leftarrow (\neg\lambda) \wedge \sigma(S_2 \oplus \beta)$
11: $\quad S_0 \leftarrow (\neg(c \vee d) \cdot S_0) \oplus ((c \vee d) \cdot S_2)$
12: $\quad i \leftarrow i + \lambda$
13: $\quad q \leftarrow q + 1$
14: **return** S_0

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.

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

Despite its promise, this first cipher level protection is attacked soon afterwards in [4], and later in [28]. The basic observation that leads to the attack is, when 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 [28] (se Algorithm 4), is much alike 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 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 4 depicts the work-flow. We next show why this is a problem.

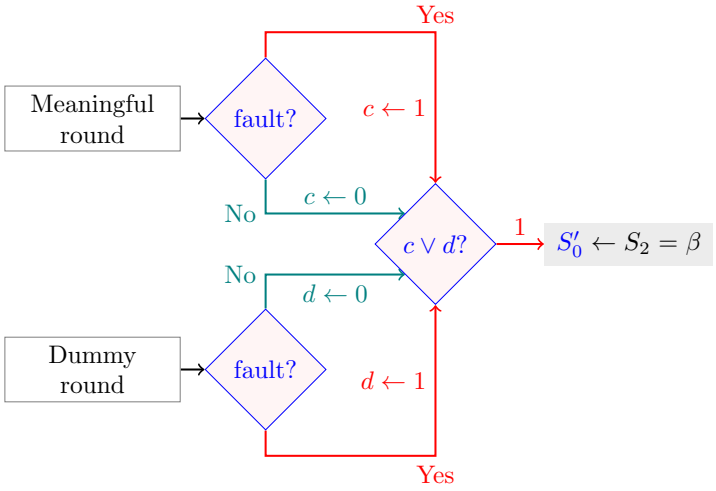

Fig. 4: Fault detection work-flow in CHES'14 countermeasure

4.1 CHES'14 Countermeasure Is Not Infective

As the heading suggests, our claim is that the scheme proposed in CHES'14 [28] does not fall under the purview of infective countermeasures. As already noted, the authors in [28] propose their scheme as a fix for the LatinCrypt'12 countermeasure [16]; which is, indeed, infective (albeit weak). It appears that the authors of the CHES'14 paper, while attempting to fix the weaknesses of [16], end up doing away with the infection mechanism. It is further worth mentioning that there have been follow up works on [28] which claim to further analyze and improve it [26] or scrutinize in the light of non-DFA attacks [5,11]. However, none of them uncovers this fact that this scheme is actually not infective. Below, we reveal our analysis after a detailed inspection of [28] in contrast to the scheme they try to fix [16].

Before proceeding further, 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 and unintelligible way. This philosophy is already described in the existing literature, e.g., [24,15]. Even the CHES'14 paper [28, page 94] itself says, "The infection countermeasure ... aims to destroy the fault invariant by diffusing the effect of a fault in such a way that it renders the faulty ciphertext ucnexploitable". This is in sharp contrast with non-infective schemes which generally suppress the output or return random output upon explicitly/implicitly detecting the fault. Our claim is that CHES'14 countermeasure implicitly detects the fault while returning random output and does not actually diffuse the fault in the state; thereby not following the infection philosophy.

In order to do a comparative analysis, one may recall their descriptions from Algorithm 3 and Algorithm 4. We start by doing a dry run of the algorithms for two main cases (namely, for the final and the penultimate rounds). This analysis can be easily extended for early rounds.

- Initial conditions: $i = 2\eta; \lambda = 1$
- Assume fault injection in Line 8 of Algorithm 3 and in Line 8 of Algorithm 4

Dry Run 1: Algorithm 3, Lines 6 – 15

1: $a \leftarrow 0$ ▷ i is even
2: $b \leftarrow \eta$
3: $S'_0 \xleftarrow{\text{fault}} \texttt{RoundFunction}_\eta(S_0) \oplus k_\eta$
4: $T_0 \leftarrow S'_0$ ▷ Infection begins
5: $c \nleftarrow 0$
6: $S'_2 \leftarrow S_2 \oplus c$
7: $S'_0 \leftarrow S'_0 \oplus c$
8: $i \leftarrow 2\eta + 1$
9: $S'_0 \leftarrow S'_0 \oplus \texttt{RoundFunction}_{k_0}(S'_2) \oplus \beta$
10: **return** S'_0 ▷ S'_0 is the infected state

Dry Run 2: Algorithm 4, Lines 6 – 14

1: $a \leftarrow 0$ ▷ i is even
2: $b \leftarrow \eta$
3: $S'_0 \xleftarrow{\text{fault}} \texttt{RoundFunction}_\eta(S_0) \oplus k_\eta$
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

- Initial conditions: $i = 2\eta - 2; \lambda = 1$
- Assume fault injection in Line 8 of Algorithm 3 and in Line 8 of Algorithm 4

Dry Run 3: Algorithm 3, 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: $T_0 \leftarrow S'_0$ ▷ Infection begins
6: $c \nleftarrow 0$
7: $S'_2 \leftarrow S_2 \oplus c$
8: $S'_0 \leftarrow S'_0 \oplus c$
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: $c \nleftarrow 0$
16: $S'_2 \leftarrow S_2 \oplus c$
17: $S'_0 \leftarrow S'_0 \oplus c$
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: $S'_0 \leftarrow \texttt{RoundFunction}_\eta(S'_0) \oplus k_\eta$
23: $T_0 \leftarrow S'_0$ ▷ Infection continues
24: $c \nleftarrow 0$
25: $S'_2 \leftarrow S'_2 \oplus c$
26: $S'_0 \leftarrow S'_0 \oplus c$
27: $i \leftarrow 2\eta + 1$
28: $S'_0 \leftarrow S'_0 \oplus \texttt{RoundFunction}_0(S'_2) \oplus k_0 \oplus \beta$
29: **return** S'_0 ▷ S'_0 is the infected state

Dry Run 4: Algorithm 4, 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

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 3. 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 4) 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 3 and 4, by targeting the penultimate actual round for fault injection.

Like before, it can be observed from Dry Run 3 (tracing steps of Algorithm 3) and 4 (tracing steps of Algorithm 4) 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 well understood that detection countermeasures can be rendered useless if the bit that senses the presence of fault is altered. Even though CHES'14 countermeasure is branch-less; effectively, the security depends on whether one particular bit is 1 or 0. At the very end, all infective countermeasures replace the state by a random value (so does this countermeasure). Still, it is necessary that the replacement subroutine does not rely on one particular bit. That is not the case in [28], and it suffers from the same vulnerability as detection (viz., it is susceptible to single bit flip). This invalidates the claim that this countermeasure is 'infective'. We believe this is a serious security flaw, and this countermeasure may not be recommended. The same argument works for the modified countermeasure in [26].

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 [16] 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 (Section 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 3). So, our patch for this scheme works basically by retaining the preceding part of Algorithm 3 (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:

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

It is easy to check, independent of z, $\texttt{RoundFunction}^z(\beta, k_0) = \beta$. Hence, we propose to substitute Line 14 of Algorithm 3:

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

by

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

where z ($\geq \xi$) is predetermined.

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 then 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[8]. 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 [16], 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. As mentioned earlier in Section 3.1, we do not consider Eve to have the capability to do both fault and side channel attack. As such, we do not claim additional side channel security from the patched scheme. When side channel protection is solicited, we recommend to use existing SCA countermeasures: Our patched scheme can be implemented easily even when the cipher implementations are protected by some SCA countermeasure. In summary, we emphasize that; making the meaningful/dummy round computation deterministic do not incur additional vulnerability from the perspective of DFA.

Algorithm 5: 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

In Algorithm 5, 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

[8] It is recommended in [16] to use a non-linear function as $\sigma(\cdot)$. However, we believe, this is not a necessary condition.

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 (Section 2.5). 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 utilize), 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 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.

Algorithm 6: Our patched version of LatinCrypt'12 countermeasure (for SPN): Protected up to $\mu - 1$ instruction(s) skip

Input: P; $\beta; k_0$; round keys $k_j; j = 1, \ldots, \eta$ derived from K; z ($\geq \xi$); μ (> 1)

▷ $\texttt{RoundFunction}_0(\beta) = \beta \oplus k_0$

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: **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: | **for** $m = 1; m \leq \mu + z; m \leftarrow m + 1$ **do**
6: | | $S_2 \leftarrow S_0 \oplus S_2$
7: | | $S_2 \leftarrow S_1 \oplus S_2$
8: | | $S_2 \leftarrow \texttt{RoundFunction}_0(S_2) \oplus k_0$
9: | $S_2 \leftarrow \texttt{RoundFunction}_0(S_2) \oplus k_0$ ▷ Dummy round
10: **for** $j \leftarrow 1; j \leq z; j \leftarrow j + 1$ **do**
11: | $S_2 \leftarrow \texttt{RoundFunction}_0(S_2) \oplus k_0$ ▷ Dummy round
12: **for** $m = 1; m \leq \mu; m \leftarrow m + 1$ **do**
13: | $S_0 \leftarrow S_0 \oplus S_2 \oplus \beta$ ▷ $S_2 = \texttt{RoundFunction}^z(S_2, k_0)$
14: **return** S_0

In the light of [26] (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 5 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: | $S_0 \leftarrow S_0 \oplus S_2 \oplus \beta$ ▷ $S_2 = \texttt{RoundFunction}^z(S_2, k_0)$

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. The complete algorithm

is given in Algorithm 6. This can pave the pathway to an infective countermeasure which is guaranteed to have a certain protection against instruction skips.

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 5), for AVR and MSP architectures. We take `AES` encryption as the underlying cipher. As before, we use the source codes from the `FELICS` [10] 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). For the 11-nested rounds, the relative clock cycles is slightly bigger than 3×; this is due to the fact that now `AES` is running 3 times together with few other computations[9]. The relative code sizes are less than 3×, as part of the same code can be reused (such as the SBox). For the 4-nested round, the relative clock cycles is less than 3 for AVR as now less than 3× `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× in 11-nested to 2.58× 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.

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 5)	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

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× unprotected `AES`, whereas 4-nested would be around 3.12× 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 [28] 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.

[9]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.

Fourth, the function $\tau_{(\cdot)}(\cdot)$ in type I is recommended to be highly non-linear; in contrast, the non-linearity in type II countermeasures come from the non-linearity 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 considered within the scope (similar to [15]); and we do not claim any SCA security against our designs in general. As such, they may be vulnerable to side channel attacks; but in that case, they can be easily protected by incorporating standard SCA countermeasures (no specialized countermeasure will not be necessary), if/when needed.

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 IFA, can be an interesting direction to pursue. Finally, a cipher may be constructed which is more suitable to deploy together with an infective countermeasure.

Acknowledgments

The authors would like to thank the anonymous reviewers of TCHES 2019 (issue 2) for their helpful comments.

References

1. 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, April 30 - May 4, 2018. (2018) 191–194 1, 10
2. 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. (2015) 241–252 6, 8
3. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. IACR Cryptology ePrint Archive **2004** (2004) 100 1
4. 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. (2013) 101–107 2, 7, 13
5. Battistello, A., Giraud, C.: A note on the security of CHES 2014 symmetric infective countermeasure. In: Constructive Side-Channel Analysis and Secure Design - 7th International Workshop, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers. (2016) 144–159 13
6. Beierle, C., Leander, G., Moradi, A., Rasoolzadeh, S.: Craft: Lightweight tweakable block cipher with efficient protection against dfa attacks. IACR Transactions on Symmetric Cryptology **2019**(1) (Mar. 2019) 5–45 6
7. Biham, E., Shamir, A.: Differential Fault Analysis of Secret Key Cryptosystems. In Kaliski, BurtonS., J., ed.: Advances in Cryptology - CRYPTO '97. Volume 1294 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1997) 513–525 1
8. Blömer, J., Krummel, V.: Fault based collision attacks on AES. In: Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2006, Yokohama, Japan, October 10, 2006, Proceedings. (2006) 106–120 6
9. Clavier, C.: Secret external encodings do not prevent transient fault analysis. In: Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings. (2007) 181–194 3, 6
10. 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) 10, 18
11. 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) (2018) 547–572 13
12. Dobraunig, C., Koeune, F., Mangard, S., Mendel, F., Standaert, F.: Towards fresh and hybrid re-keying schemes with beyond birthday security. In: Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers. (2015) 225–241 1
13. 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) (December 2011) 328–347 3

14. Ghosh, S., Saha, D., Sengupta, A., Chowdhury, D.R.: Preventing fault attacks using fault randomization with a case study on AES. In: Information Security and Privacy - 20th Australasian Conference, ACISP 2015, Brisbane, QLD, Australia, June 29 - July 1, 2015, Proceedings. (2015) 343–355 3, 8, 9
15. Ghosh, S., Saha, D., Sengupta, A., Chowdhury, D.R.: Preventing fault attacks using fault randomisation with a case study on AES. IJACT **3**(3) (2017) 225–235 2, 5, 9, 10, 11, 13, 19
16. 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. (2012) 305–321 2, 3, 5, 11, 13, 15, 16
17. He, W., Breier, J., Bhasin, S.: Cheap and cheerful: A low-cost digital sensor for detecting laser fault injection attacks. In: Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings. (2016) 27–46 1
18. Joye, M., Manet, P., Rigaud, J.B.: Strengthening hardware aes implementations against fault attacks. IET Information Security **1**(3) (2007) 106 3
19. Karri, R., Kuznetsov, G., Gössel, M.: Parity-based concurrent error detection of substitution-permutation network block ciphers. In: Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings. (2003) 113–124 3
20. Keliher, L., Sui, J.: Exact maximum expected differential and linear probability for two-round advanced encryption standard. IET Information Security **1**(2) (2007) 53–57 4
21. 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, Berlin, Heidelberg, Springer Berlin Heidelberg (2007) 215–228 3
22. Kulikowski, K., Karpovsky, M., Taubin, A.: Robust codes for fault attack resistant cryptographic hardware. In: Fault Diagnosis and Tolerance in Cryptography, 2nd International Workshop. (2005) 1–12 3
23. 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. (2017) 33–40 3
24. 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. (2012) 85–94 2, 3, 5, 6, 7, 10, 11, 13
25. Otto, M.: Fault Attacks and Countermeasures. PhD thesis, Institut für Informatik, Universität Paderborn (2004) 3
26. Patranabis, S., Chakraborty, A., Mukhopadhyay, D.: Fault tolerant infective countermeasure for AES. In: Security, Privacy, and Applied Cryptography Engineering - 5th International Conference, SPACE 2015, Jaipur, India, October 3-7, 2015, Proceedings. (2015) 190–209 2, 3, 5, 13, 15, 17
27. 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). (Aug 2016) 36–46 6
28. Tupsamudre, H., Bisht, S., Mukhopadhyay, D.: Destroying fault invariant with randomization - A countermeasure for AES against differential fault attacks. In: Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings. (2014) 93–111 2, 3, 5, 13, 15, 18
29. Yen, S., Joye, M.: Checking before output may not be enough against fault-based cryptanalysis. IEEE Trans. Computers **49**(9) (2000) 967–970 3