Breach the Gate: Exploiting Observability for Fault Template Attacks on Block Ciphers

Sayandeep Saha[1], Debapriya Basu Roy[1], Arnab Bag[1], Sikhar Patranabis[1], and Debdeep Mukhopadhyay[1]

Indian Institute of Technology, Kharagpur
{sahasayandeep, dbroy, arnab, sikhar, debdeep}@iitkgp.ac.in

Abstract. Fault attacks (FA) are one of the potent practical threats to modern cryptographic implementations. Over the years the FA techniques have evolved, gradually moving towards the exploitation of device-centric properties of the faults. In this paper, we exploit the fact that activation and propagation of a fault through a given combinational circuit (i.e. observability of a fault) is data dependent. Next, we show that this property of combinational circuits leads to powerful fault attacks even for implementations having dedicated and provably secure protections against both power and fault-assisted vulnerabilities. The attacks found in this work are applicable even if the fault injection is made at the middle rounds of a block cipher, which are out of reach for most of the other existing fault analysis strategies. Quite evidently, they also work for a known plaintext scenario. Moreover, the middle round attacks are entirely blind in the sense that no access to the ciphertexts (correct/faulty) or plaintexts are required. The adversary is only assumed to have the power of repeating an unknown plaintext several times. Experimental validation over software implementations of PRESENT and AES proves the efficacy of the proposed attacks.

Keywords: Fault Attack · Fault Propagation · Masking.

1 Introduction

Implementation-based attacks are one of the most practical threats to modern cryptography. With the dramatic increase in the usage of embedded devices for IoT and mobile applications, such attacks have become a real concern. Most of the modern embedded devices carry cryptographic cores and are physically accessible by the adversary. Therefore, suitable countermeasures are often implemented to protect the cryptographic computations from exploitation.

Side-channel attacks (SCA) [7] and Fault attacks (FA) [4, 6] are the two most widely explored implementation attack classes till date. The main idea behind the first one is to passively exploit the operation dependency (simple-power-analysis) or data-dependency (differential/correlation power analysis) of the cryptographic computation to infer the secret key by measuring power or electromagnetic (EM) signals. In contrast, fault attacks are active in nature

as they work by corrupting the intermediate computation of the device in a controlled manner. Intentionally injected faults usually create a statistical bias in some of the intermediate computation states, which are exploited by the adversary either analytically or statistically to reduce the entropy of the unknown key and thereby recover it [4].

The protection mechanisms found in modern devices mostly try to mitigate the two above-mentioned classes of attacks. In this context, hardening the cipher algorithm itself with countermeasures is often preferred than the sensor and shield-based physical countermeasures. This is due to the fact that algorithm-level countermeasures are flexible and low cost. Moreover, they often provide provable security guarantees. Masking is the most prominent and widely deployed countermeasure so far, against passive SCA [12, 15, 20, 23]. Masking is a class of techniques which implement secret sharing at the level of cryptographic circuits. Each cipher variable x is split into a certain number (say d) of shares in masking which are statistically independent by their own, and also while considered in groups of size up to d. Each underlying function of the cipher is also shared into d component functions (respecting the correctness) to process the shared variables. The order of protection d intuitively means that an adversary has to consider SCA leakages for $d + 1$ points, simultaneously, to gain some useful information about the intermediate computation. In the context of FA, detection-type countermeasures are the most common one. The main principle of these FA countermeasures is to detect the presence of a fault via some redundant computation (time/space redundancy or information redundancy), and then react by either muting or randomizing the corrupted output [13, 18]. Another alternative way is to avoid the explicit detection step altogether and perform the computation in a way so that it gets deliberately randomized if there is any error in computation [30].

Symmetric key primitives (such as block ciphers) are the most widely analyzed class of cryptographic constructs in the context of implementation-based attacks. Quite evidently, the current evaluation criteria for a block cipher design takes the overhead due to SCA and FA protections directly into account. In other words, countermeasures are nowadays becoming an essential part of a cipher. In practice, there exist proposals which judiciously integrate these two countermeasures for block ciphers [27]. Whether such hardened algorithms are actually secured or not is, however, a crucial question to be answered.

Recent developments in FA show that the answer to the above-mentioned question is negative. Although combined countermeasures are somewhat successful in throttling passive attacks, they often fall prey against active adversaries. In [8, 9], it was shown that if an adversary has the power of injecting a sufficient number of faults, even the correct ciphertexts can be exploited for the attack. The attack in [8], also known as Statistical Ineffective Fault Analysis (SIFA), changed the widely regarded concept that fault attacks require faulty ciphertexts to proceed. Most of the existing FA countermeasures are based on this belief and thus were broken. In a slightly different setting, the so-called Persistent Fault Analysis (PFA) [21,32] presented a similar result. The main reason

behind the success of SIFA and PFA is that they typically exploit the statistical bias in the event when a fault fails to alter the computation. However, this seemingly simple event can be exploited in several other ways too, which may lead to more powerful attacks on protected implementations. This paper reports some of such newly found vulnerabilities. Our contributions here are as follows:

Our Contributions:

In this paper, we propose a new attack strategy for protected implementations which exploit fundamental principles of digital gates to extract the secret. The main observation we exploit is that *the output observability of a fault, injected at one input of an AND gate depends on the values of the other inputs.* In general, the activation and propagation of a wire-fault inside a circuit depend upon the value under process which is indeed a side channel leakage. Based on this simple albeit fundamental observation we devised attacks which can break masking schemes of any arbitrary order, even if it is combined with FA countermeasures. *The strongest feature of this attack strategy is that it can enable attacks in the middle round of a cipher without requiring any explicit access to the ciphertexts even if they are correct. Just knowing whether the outcome of the encryption is faulty or not would suffice. The plaintexts are also needed not to be known explicitly in all scenarios, but the adversary should be able to repeat them any number of times.* One should note that the attacks like SIFA requires ciphertext access and are also not applicable to the middle rounds. [1] In some sense, the proposed attack strategy further challenges the belief that practical fault attacks at least require access to the correct ciphertexts.

The fault model utilized in this attack is similar to the wire-fault model of digital circuits. A similar fault model was exploited for SIFA [8]. However, the exploitation methodology of the faults is entirely different from SIFA. While SIFA uses statistical analysis based on the correct ciphertexts, we propose a novel strategy based on *fault templates.* The template-based attack strategy, abbreviated as FTA, efficiently exploit fault characteristics from different fault locations for constructing distinguishing fault patterns, which enable key/state recovery. In principle, FTA is closer to SCA than FAs and hence, evaluation of masking against this new class of attack becomes extremely important.

The attacks proposed in this paper require multiple fault locations to extract the entire key. *Note that, we do not require multiple fault locations to be corrupted at the same time, but injections can be made one location at a time in different independent experiments.* While faulting intermediate wires of a combinational net is challenging, it is still practically feasible with modern fault injection setups like laser-based injection [24, 28]. In particular, it was shown

[1] Several modern symmetric-key protocols do not expose the ciphertexts. One prominent example is the Message Authentication Codes (MAC) in certain application scenarios. Furthermore, for many existing Authenticated Encryption schemes, direct access to the plaintext is not available for the block ciphers used within the scheme. However, fixing the plaintext value may be achieved. Also, in real devices, the accessibility of plaintexts cannot be assumed in every scenario. One typical example is the shared root key usage in UTMS [19].

in [28] that for FPGA-based implementations, each gate can be targeted with laser-induced faults resulting in stuck-at faults at their inputs. Further, in [24] it was shown that targeting microcontrollers with laser results in bit set/reset faults with reasonably high probability. Both of these results are of high relevance in the present context as they clearly establish the practicality of our fault models for both software and hardware implementations. One advantage of the proposed FTA attack strategy is that the target implementation can be extensively profiled before attack and parameters for obtaining the desired faults can be accurately identified. In summary, the fault model in our attacks is practically achievable, which establishes the potency of these attacks.

The idea of FA without direct access to the plaintext and ciphertext has been explored previously. The closest to our proposal are so-called Blind Fault Analysis (BFA) and the Safe-Error-Attack (SEA). However, none of these attacks exploit the inherent circuit properties as we do in our case. Finally, both BFA and SEA can be throttled by masking. The greatest advantage of our proposal lies at this point that our attacks are applicable for masking countermeasures even while combined with a state-of-the-art FA countermeasure. Although SIFA and PFA work on masking, both of them require ciphertext access to attack.

In order to validate our idea, we choose the block cipher PRESENT as a test case [5]. PRESENT is a fairly well-analyzed design and a potential contestant in the ongoing NIST competition for standardizing lightweight block ciphers [2]. The choice of a lightweight cipher is also motivated by the fact that countermeasures are extremely crucial for such ciphers as they are supposed to be deployed on low cost embedded devices. However, the attacks are equally applicable to larger block ciphers like AES. We shall present a practical example to justify this claim.

The rest of the paper is organized as follows. We begin by explaining the fundamental principles behind the attacks in Sec. 2 through interpretable examples. Feasibility of the attacks for unmasked but FA protected implementations are discussed in Sec. 3 taking PRESENT as an example. Attacks on combined countermeasures are proposed in Sec. 4 (on PRESENT), followed by an evaluation of a publicly available masked AES implementation in Sec. 5. Sec. 6 presents a brief discussion on possible countermeasures. We conclude in Sec. 7.

2 The Fundamental Principle

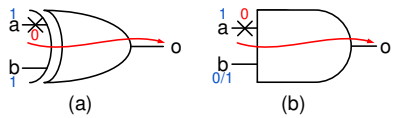

Fig. 1: Fault propagation: a) XOR gate; b) AND gate. The inputs for activation and propagation are in blue and the value of the stuck-at fault is in red.

2.1 Fault Activation and Propagation

The concept of fault activation and propagation is instrumental for structural fault testing of digital circuits. Almost every Automatic test pattern generation (ATPG) algorithm relies on these two events. Consider a combinational circuit $\mathcal{C}$ and an internal net i in this circuit. The problem is to test if the net has been stuck at a value 0 or 1. A test pattern for exposing this fault to the output is required to perform the two following events in sequence:

1. **Fault Activation:** The test pattern is required to set the net i to a value x such that i carries the complement of x (i.e., $\overline{x}$) in the presence of a fault and x, otherwise.
2. **Fault Propagation:** The test pattern has to ensure that the effect of the fault propagates to the output of the circuit $\mathcal{C}$.

Both the activation and propagation events strongly depend upon the structure of the circuit graph, and the gates present in the circuit. However, understanding the fault activation and propagation property of each gate is the very first step to have an insight of the attacks we are going to propose. Let us first consider a linear 2-input XOR gate as shown in Fig. 1(a). Without loss of generality, we consider a stuck-at-0 fault at the input a, while the input b may take values independently. In order to activate the fault at a, one must set $a = 1$. The next step is to propagate the fault at the output. One may observe that setting the input b to either 0 or 1 will expose the fault at a to the output o. A similar phenomenon can be observed for an n-input XOR gate. This observation is summarized in the following statement:

Given an n-input XOR gate having an input set I, ($|I| = n$), an output O, and a faulted input $i \in I$, the fault propagation to O does not depend upon the valuations of the subset $I \setminus \{i\}$.

An exactly opposite situation is observed for the nonlinear gates like AND/OR. For the sake of illustration let us consider the two input AND gate in Fig. 1(b). Here a stuck-at fault (either stuck-at-0 or stuck-at-1) at input a can propagate to the output o if and only if the input b is set to the value 1[2]. An input value of 1 for an AND gate is often referred to as *non-controlling value*[3]. The activation and propagation property of the AND gates, thus, can be stated as follows:

For an n-input AND gate with input set I, output O, and one faulty input $i \in I$, the fault propagation takes place if an only if every input in the subset $I \setminus \{i\}$ is set to its non-controlling value.

[2] The fault activation takes place if a is set to 1 (stuck-at-1) or 0 (stuck-at 1).

[3] A controlling input value of a gate is defined as a value, which, if present for at least one input, sets the output of the gate to a known value. Non-controlling value is the complement of the controlling value.

2.2 Information Leakage Due to Fault Propagation

Now we describe how information leaks due to the propagation of faults. Once again we consider the AND and the XOR gate for our illustration. Let us assume that the gates are processing secret data and an active adversary $\mathcal{A}$ can only have the information whether the output is faulty or not. The adversary can, however, inject a stuck-at fault at one of the inputs of the gate[4]. We also consider that the adversary has complete knowledge about the type of the gate she is targeting. With these adversary model now we can analyze the information leakage due to the presence of faults.

First, we consider the XOR gate. Without loss of generality, let us assume the fault to be stuck-at-0, and the injection point is a. Then the fault will propagate to the output whenever it gets activated. In other words, just by observing whether the output is faulty $\mathcal{A}$ can determine the value of a. More precisely, if the output is fault-free $a = 0$ and $a = 1$, otherwise.

The situation is slightly different in the case of AND gates. Here the output becomes faulty only if the fault is activated at a and b is set to its non-controlling value. In this case, the adversary can determine the values of both a and b. However, one should note that the fault will only propagate if both a and b are set to unity. For all other cases the output will remain uncorrupted and $\mathcal{A}$ cannot determine what value is being processed by the gate. Putting it in another way, the adversary can divide the value space of (a, b) into two equivalence classes. The first class contains values $(0, 0), (0, 1)(1, 0)$, whereas the second class contains only a single value $(1, 1)$. One should note that the intra-class values cannot be distinguished from each other.

One general trend in FA community is to quantify the leakage in terms of entropy loss. The same can be done here for both the gates. Without the fault the entropy of (a, b), denoted as $H((a, b))$, is 2. In the case of XOR gate, the entropy reduces after the first injection event. Depending on the value of the observable O_{f_1}, which we set to 1 if the fault is observed at the output (and 0, otherwise), the actual input value at the fault location can be revealed. More formally, we have $H((a, b)|O_{f_1} = 0) = 1$ and $H((a, b)|O_{f_1} = 1) = 1$. Therefore, the remaining entropy $H((a, b)|O_{f_1}) = \frac{1}{2} \times H((a, b)|O_{f_1} = 0) + \frac{1}{2} \times H((a, b)|O_{f_1} = 1) = 1$. In other words, the entropy of (a, b) reduces to 1 after one fault injection. The situation is slightly different in the case of AND gate. Here the remaining entropy can be calculated as $H((a, b)|O_{f_1}) = \frac{3}{4} \times \log_2 3 + \frac{1}{4} \times \log_2 1 = 1.18$. Although the leakage here is slightly less compared to the XOR gate, once should note that it is conditional on the non-faulty inputs of the gate too. In other words, partial information regarding both a and b are leaked, simultaneously. In contrast, XOR completely leaks one bit but does not leak anything about the other inputs.

As we shall show later in this paper, both of AND and XOR gate leakages can be cleverly exploited to mount extremely strong FAs on block ciphers. In the next subsection, we extend the concept of leakage for larger circuits.

[4] Although for simplicity we are considering stuck-at faults here, our arguments are also valid for single bit toggle faults

2.3 Fault Propagation in Combinational Circuits

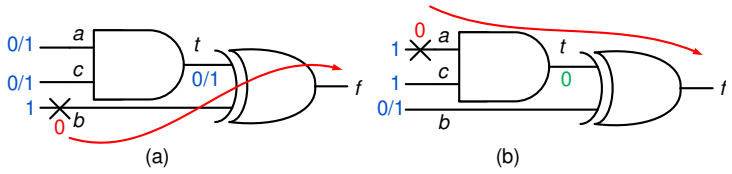

Fig. 2: Fault propagation through combinational circuits: a) Injection at XOR gate input; b) Injection at AND gate input. The inputs for activation and propagation are shown in blue and the nature of the stuck-at fault is shown in red. The propagated faulty intermediate value is shown in green.

One convenient and general way of realizing different sub-operations of a block cipher is by means of algebraic expressions over $GF(2)$ also known as Algebraic Normal Form (ANF). For the sake of explanation, we also use the ANF representation of the circuits throughout this paper. ANF representation is also common while implementing masking schemes. Therefore, a good starting point would be to analyze the effect of faults on an ANF expression. For example, let us consider the following ANF expression and its corresponding circuit in Fig. 2.[5]

$$f = b + ca \tag{1}$$

As in the previous case, we assume that the adversary $\mathcal{A}$ can only observe whether the output is faulty or not, but cannot observe the actual output of the circuit. Also, the inputs are not observable but can be kept fixed. With this setting the adversary injects a stuck-at-0 fault in b (see Fig. 2(a)). Now, since the input is fixed, a fault at the output would imply that $b = 1$. On the other hand, the output will be correct only if $b = 0$. The property of the XOR get mentioned in the previous subsection ensures that the other input coming from the product term does not affect the recovery of the bit b. In a similar fashion, one can recover the output of the product term ca.

Let us now consider recovery of the bits a and c, with the fault injected at a. From the properties of an AND gate, the fault will propagate to the wire t (see Fig. 2(b)) if and only if $c = 1$ and $a = 1$. This fault, on the other hand, will directly propagate to the output as the rest of the circuit only contain XOR gates. However, from adversary's point of view, entropy reduction due to a non-faulty output is not very significant (non-faulty output may occur for (c, a) taking values $(0, 0)$, $(0, 1)$ and $(1, 0)$). Moreover, no further information is leaked even if the attacker now targets the input c with another fault. It may seem that the AND gates are not very useful as leakage sources. However, it is not true if we can somehow exploit the fact that it leaks information about more than

[5] Note that the "+" represents XOR operation here

one bits. In the next subsection, we shall show the impact of this property for attacking block ciphers.

2.4 Propagation Characteristics of S-Boxes

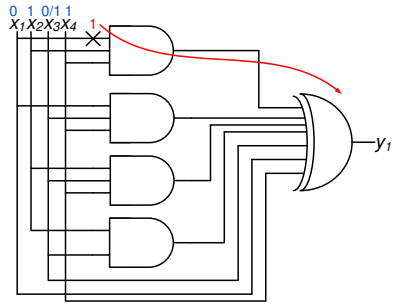

Fig. 3: Fault propagation through S-Box Polynomials. The input pattern causing the propagation is shown in blue. The stuck-at fault type is shown in red numbers.

The S-Boxes one of the most common constituents of modern block ciphers. In most of the cases, they are the only non-linear function within a cipher. Mathematically, they are vectorial Boolean functions consisting of high degree polynomials over $GF(2)$. Such polynomials contain high degree monorails which are nothing but several bits AND-ed together. As a concrete example, we consider the S-Box polynomials for PRESENT as shown in Eq. (2). This S-Box has 4 input bits denoted as x_1, x_2, x_3, x_4 and 4 output bits y_1, y_2, y_3, y_4.

$$\begin{aligned} y_1 &= x_1x_2x_4 + x_1x_3x_4 + x_1 + x_2x_3x_4 + x_2x_3 + x_3 + x_4 + 1 \\ y_2 &= x_1x_2x_4 + x_1x_3x_4 + x_1x_3 + x_1x_4 + x_1 + x_2 + x_3x_4 + 1 \\ y_3 &= x_1x_2x_4 + x_1x_2 + x_1x_3x_4 + x_1x_3 + x_1 + x_2x_3x_4 + x_3 \\ y_4 &= x_1 + x_2x_3 + x_2 + x_4 \end{aligned} \tag{2}$$

Let us consider the first polynomial in this system without loss of generality. Also, we consider a stuck-at-1 fault at x_1 during the computation of this polynomial. The exact location of this fault in the circuit is depicted in Fig. 3. Given this fault location, the fault propagates to the output only if $(x_1 = 0,\ x_2 = 1,\ x_3 = 0,\ x_4 = 1)$ or $(x_1 = 0,\ x_2 = 1,\ x_3 = 1,\ x_4 = 1)$. For the rest of the cases, the output remains unaltered. *Consequently, if the S-Box inputs are changing and the value is inaccessible for the adversary, she can still detect when the S-Box processes the input* $(0, 1, 0, 1)$ *or* $(0, 1, 1, 1)$*, as compared to other inputs.* [6] In the next subsection, we shall show how this simple observation results in key leakage for an entire cipher.

[6] One should note that the faults need not be always injected at the input of a combinational block, but they can also be injected at internal wires too. Such a fault model is easier to realize in case of software implementations as shown in [8]

3 Fault Observability Attacks

In this section, we describe how information leakage from gates eventually results in key leakage for so-called FA resilient block cipher implementations. For the sake of simplicity, we begin with implementations having redundancy-based detection-type FA countermeasures. Implementations having both masking and FA countermeasures will be considered in the subsequent sections. The detection-type FA countermeasures under consideration may use any form of redundancy (space, time or information redundancy) [13, 18]. However, the attacks we are going to describe are equally applicable to any member of this classical countermeasure class. For the sake of simplicity, we, therefore, consider the most trivial form where the redundancy check happens at the end of the computation before outputting the ciphertexts.

3.1 Template-based Fault Attacks

Before going to the actual attack instances, let us first describe our general attack strategy, which is based on constructing templates. Although the concept of fault template attack (FTA) has previously been addressed in the literature, (first proposed in [10]) their applicability was extremely limited. Similar to the template attacks in SCA, fault template attacks also consist of two phases, namely:

1. **Template Building (offline):** This is an offline phase where the target implementation is profiled extensively to construct an informed model for the attack. The aim of this informed modeling is to directly reason about some unknown in the online phase of the attack (mounted on a similar implementation), based on some observables from the online experiment[7]. Formally, a template $\mathcal{T}$ for fault attack can be represented as a mapping $\mathcal{T} : \mathcal{F} \rightarrow \mathcal{X}$, where an $a \in \mathcal{F}$ is constructed by computing some function on the observables. Alternatively, a can also be a probability distribution defined over the observables. The location for a fault injection can be used as an auxiliary information while computing the function from the observable set to the set $\mathcal{F}$. The range set $\mathcal{X}$ of the template $\mathcal{T}$ either represents a part of an intermediate state, (for example, the value of a byte/nibble) or a part of the secret key.

2. **Template Matching (online):** In this online phase, an implementation (identical to one profiled in the offline phase) with an unknown key is targeted with fault injection. The injection locations may be pre-decided from the template construction phase. The unknown is supposed to be discovered by first mapping the observables from this experiment to a member of the set $\mathcal{F}$ and then by finding out the corresponding value of the unknown from the set $\mathcal{X}$ using the template $\mathcal{T}$.

[7] The observable, for example, can be the knowledge that whether the output of the encryption is faulty or not.

Unlike differential or statistical fault attacks, the key recovery algorithms in fault template attacks are fairly straightforward in general. The fault complexity of the attacks is somewhat comparable with that of the statistical fault attacks. However, one great advantage over statistical or differential fault attacks is that access to ciphertexts or plaintexts is not essential in certain instances of FTA. The attacker only requires to know whether the outcome is faulted or not. More precisely, FTA can target the middle rounds of block ciphers, which are otherwise inaccessible by statistical or differential attacks due to extensive computational complexity. Apart from that, the FTA differs significantly from all other classes of fault attacks in the way it exploits the leakage. While attacks differential or statistical attacks use the bias in the state due to fault injection as a key distinguisher, template-based attacks directly recover the intermediate state values. From this aspect, this attack is closer to the SCA attacks. However, there are certain dissimilarities with SCA as well, in the sense that SCA template attacks try to model the noise from the target device and measurement equipment. In contrast, FTA goes beyond noise modeling and build templates over the fault characteristics of the underlying circuit.

Algorithm 1 *BUILD_TEMPLATE*

Input: Target Implementation C, Fault fl
Output: Template $\mathcal{T}$

$\mathcal{T} := \emptyset$
$w :=$ `GET_SBOX_SIZE`() ▷ Get the width of the S-Box
for $(0 \leq k \leq 2^w)$ **do** ▷ Vary one key word
  $F_t := \emptyset$
  for $(0 \leq p \leq 2^w)$ **do** ▷ Vary one plaintext word
    $x := p \oplus k$
    $y_f := C(x)^{fl}$ ▷ Inject fault in one of the S-Boxs for each execution
    $y_c := C(x)$
    if `DETECT_FAULT(`y_f`,` y_c`) == 1` **then** ▷ Fault detection function
      $F_t := F_t \cup \{1\}$
    else
      $F_t := F_t \cup \{0\}$
    end if
  end for
  $\mathcal{T} := \mathcal{T} \cup \{(F_t, k)\}$
end for
Return $\mathcal{T}$

Algorithm 2 *MATCH_TEMPLATE*

Input: Protected cipher with unknown key C_k, Fault fl, Template $\mathcal{T}$
Output: Set of candidate correct keys k_c

```
k_cand := ∅                                          ▷ Set of candidate keys
w := GET_SBOX_SIZE()
F_t := ∅
for (0 ≤ p ≤ 2^w) do                                ▷ Vary a single w bit word of the plaintext
    O := (C_k(p))^fl                                 ▷ Inject fault for each execution
    if (O == 1) then                                 ▷ Fault detected
        F_t := F_t ∪ {1}
    else
        F_t := F_t ∪ {0}
    end if
end for
k_cand := k_cand ∪ {T(F_t)}
Return k_cand
```

3.2 Attacks on Unmasked Implementations: Known Plaintext

In this subsection, we present the first concrete realization of FTA. The first attack we present requires the plaintexts to be known and controllable. However, explicit knowledge of the ciphertexts is not expected. The adversary is only provided with the information whether the outcome of encryption is faulty or not. One practical example of such attack setup is a block-cipher based Message-Authentication Code (MAC), where the authentication tag might not be exposed to the adversary, but the correctness of the authentication is available. We also assume a stuck-at-1 fault model for simplicity. However, the attack also applies to stuck-at-0 and bit-flip models. For the sake of illustration, we mainly consider the PRESENT block cipher as our target. The attack consists of two phases:

3.2.1 Offline Phase: Template Building

Perhaps the most important aspect of the attacks we describe is the fault location. As elaborated in Sec. 2 leakage from the non-linear or the linear gates can be exploited. For this particular case, we choose an AND gate for fault injection respecting the fact that information regarding multiple bits is leaked, simultaneously. For the sake of simplicity, the same fault location as in Sec. 2.3 is utilized. *The observables, in this case, are the 0,1 patterns, from the protected implementation where 0 represents a correct outcome and 1 represents a faulty outcome.* The domain set $\mathcal{F}$ of the template consists of patterns called *fault patterns* (denoted as F_t in the algorithm) constructed from the observables. The fault location, in this case, is fixed. The process of transforming the observables to fault patterns and then mapping them to the set $\mathcal{X}$ is outlined in Algorithm 1[8]. For each choice of the key nibble (which is a member from set $\mathcal{X}$), all 16 possible plaintext nibbles are fed to the S-Box equations according to a predefined sequence, and the stuck-at-1 fault is injected for each of the cases. Consequently,

[8] Note that, in this attack in all our subsequent attacks, constructing the template for one S-Box is sufficient. The same template can be utilized for extracting all key nibbles of a round one by one.

for each choice of the key one obtains a bit-string of 16 bits which is the desired fault pattern (F_t). The fault patterns are depicted in Table 1. It can be observed that corresponding to each fault pattern there can be two candidate key suggestions. One should also note that changing the fault location might change the fault patterns and the mapping $\mathcal{T} : \mathcal{F} \rightarrow \mathcal{X}$.

3.2.2 Online Phase: Template Matching

The online phase of the attack is fairly straightforward. The attacker now targets an actual implementation (similar to that used in the template building phase) with an unknown key and constructs the fault patterns. The fault patterns are constructed for each S-Box at a time, by targeting the first round [9]. Next, the template is matched and the key is recovered directly. The algorithm for the online phase is outlined if Algorithm 2 for each nibble/byte. Although, the overall attack procedure is fairly simple several intricacies are still associated with it. We address them in the following paragraphs.

Table 1: Template-1 for attacking the first round of PRESENT by varying the plaintext nibble. The black cells represent 1 (faulty output) and the gray cells represent 0 (correct output).

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Key
								■		■						13, 15
												■		■		9, 11
	■		■													4, 6
■		■														5, 7
										■		■				12, 14
				■		■										1, 3
					■		■									0, 2
													■		■	8, 10

Table 2: Template-2 for attacking the first round of PRESENT. The black cells represent 1 (faulty output) and the gray cells represent 0 (correct output).

0	Key
	2, 3, 6, 7, 10, 11, 14, 15
■	0, 1, 4, 5, 8, 9, 12, 13

Unique key recovery: The template used in the proposed attack reduces the entropy of each key nibble to 1-bit. The obvious question is whether the entropy can be reduced to zero or not. In other words, is it somehow possible to create a template which provides unique key suggestions for each fault pattern? Unfortunately, the answer is negative for this particular example. This is because, with the chosen fault location, no leakage happens for the variable x_3. In fact, there

[9] Extraction of round keys in a per-nibble/byte basis is done for all the attacks described in this paper.

is no such location in the S-Box equations which can simultaneously leak information regarding all the bits. Therefore, one-bit uncertainty will always remain for the given template and for all other similar templates. However, the key can still be recovered uniquely if another template, corresponding to a different fault location is utilized. The choice of this fault location should be such that it leaks about x_3. The main challenge in this context is to keep the number of injections as low as possible for the second template. Fortunately, it was observed that the second template can be constructed in a way so that it only requires a single fault injection. The trick is to corrupt a linear term x_3 (The template is depicted in Table. 2). Due to the activation-propagation property of the XOR gates, a single injection would reveal the value of the bit x_3. In practice, we take the intersection between the key suggestions obtained from two different templates and can identify the key uniquely. As a concrete example for why it works consider that the key suggestion from the first template is $(13, 15)$ for some specific nibble. The second template will provide either of the two suggestion sets described in it. Now, since 13 and 15 only differ by the bit x_3, the suggestion set returned by template-2 is suppose to contain only one of them. Hence taking the intersection of this second key suggestion set with the first one would uniquely determine the key.

Required number of faults: The key recovery process in our proposed attack is nibble-wise. For each nibble, we require total 17 fault injections (16 for the first template matching and 1 for the second template matching). In order to retrieve a total round key, we thus require $17 \times 16 = 272$ fault injections. A general trick for minimizing the number of faults is to first choose the highest degree monomial for injection so that the maximum number of bits can be leaked at once. The remaining bits can then be leaked by choosing lower degree terms and constructing templates for them. The injections in our attack are supposed to be precise, repeatable and at different locations. However, it is still practically feasible as pointed out in the introduction. Moreover, one should also note that all fault locations within a single higher degree monomial are equivalent in terms of leakage. This fact gives the attacker extra flexibility while choosing the fault locations during practical evaluation.

It should be observed that although the attack described here requires at most two fault locations to be corrupted to uniquely recover the key, the corruptions need not be simultaneous. In practice, one can run independent fault campaigns on the target implementation and combine the results to finally recover the key. A similar attack is also applicable for AES (see Appendix B for a brief description of this attack). In the next subsection, we will explore the situations where the fault is injected in the middle round of the cipher. As we shall see, the attack methodology of ours still allows the recovery of the key within reasonable computational and fault complexity.

Algorithm 3 *BUILD_TEMPLATE_MIDDLE_ROUND*

Input: Target implementation C, Faults $fl_0, fl_1, ..., fl_h$
Output: Template $\mathcal{T}$

```
T := ∅
w := GET_SBOX_SIZE()                                   ▷ Get the width of the S-Box
for (0 ≤ x ≤ 2^w) do                                   ▷ The key is known and fixed here and x = p + k
    F_t := ∅
    for each fl ∈ {fl_0, fl_1,..., fl_h} do
        y_f := C(x)^fl                                 ▷ Inject fault in one copy of the S-Box for each execution
        y_c := C(x)
        if DETECT_FAULT(y_f, y_c) == 1 then            ▷ Fault detection function
            F_t := F_t ∪ {1}
        else
            F_t := F_t ∪ {0}
        end if
    end for
    T := T ∪ {(F_t, x)}
end for
Return T
```

3.3 Attacks on Unmasked Implementations: Middle Rounds

Classically FAs target the outer rounds of block ciphers. Attacking middle rounds are not feasible due to the extensive exhaustive search complexity involved, which becomes equal to the brute force complexity. However, the proposed template-based attack techniques do not suffer from this limitation. In this subsection, we shall investigate the feasibility of template-based attacks on the middle rounds of a given block cipher.

The main challenge in a middle round attack is that the round inputs are not accessible. Therefore, the attacks described in the last subsections cannot be directly applied in this context. However, template construction is still feasible. A single attack location, in this case, cannot provide sufficient exploitable leakage. The solution here is to corrupt multiple chosen locations and to construct a single template combining the information obtained. Unlike the previous case, where the plaintext was varying during the attack phase, in this case, it is required to be kept fixed. Formally, the mapping from the set of observables to the set $\mathcal{F}$, in this case, is a function of fault locations. Also, the range set $\mathcal{X}$ of the template, in this case, contains byte/nibble values from an intermediate state instead of keys (more precisely, the inputs of the S-Boxes).

Algorithm 4 *MATCH_TEMPLATE_MIDDLE_ROUND*

Input: Protected cipher with unknown key C_k, Faults $fl_0, fl_1, ..., fl_h$, Template $\mathcal{T}$
Output: Set of candidate correct keys k_c

```
k_cand := ∅                                          ▷ Set of candidate keys
w := GET_SBOX_SIZE()
F_t := ∅
for each fl ∈ {fl_0, fl_1, ...fl_h} do
    O := (C_k(p))^fl                                 ▷ Inject fault for each execution
    if (O == 1) then                                 ▷ Fault detected
        F_t := F_t ∪ {1}
    else
        F_t := F_t ∪ {0}
    end if
end for
k_cand := k_cand ∪ {T(F_t)}
Return k_cand
```

One critical aspect of this attack is to select the fault locations which would lead to maximum possible leakage. In contrast to the previous attack, where corrupting the highest degree monomials leak the maximum number of bits, in this new attack we observed that linear monomials are better suited as fault locations. This is because linear monomials leak information irrespective of the value of their input or the other inputs of the gate and as a result minimum number of injections would be required for them. Considering the example of PRESENT, one bit is leaked per fault location and hence 4 different locations have to be tried to extract a complete nibble of an intermediate state. The template construction and the attack algorithm (in per S-Box basis) are outlined in Algorithm 3 and 4.

The template for the middle round attack on PRESENT is shown in Table. 3. Since the linear terms are corrupted, each intermediate can be uniquely classified. in the online phase of the attack, the plaintext is held fixed. The specified fault locations are corrupted one at a time, and the fault patterns are constructed. An intermediate state can be recovered with this approach immediately (by applying the Algorithm 4 total 16 times). However, one should notice that recovering a single intermediate state does not allow the recovery of the round key. At least, two consecutive states must be recovered for the actual key recovery. Fortunately, recovery of any state with the proposed attack strategy is fairly straightforward. Hence, one just need to recover the states corresponding to two consecutive rounds and extract one round of key in a trivial manner. In essence, the round key corresponding to any of the middle rounds can be recovered. The number of faults required for entire round key recovery is 128 in this case for PRESENT.

3.4 Discussion

The attack technique outlined for the middle rounds requires the fault to be injected at many different locations. Although the SEA attacks would also require a similar number of fault injections[10], as we show in the next section, the

[10] In fact, one can perform the same attack at the key addition stages to recover the key directly.

proposed attack strategy still works when masked implementations are targeted. This is an advantage over SEA or BFA or as they are not applicable on masking implementations [17].

Table 3: Template for attacking the middle rounds of PRESENT

f_0	f_1	f_2	f_3	State		f_0	f_1	f_2	f_3	State
■	■	■	■	0		▒	■	■	■	8
■	■	▒	■	1		▒	■	▒	■	9
■	▒	■	■	2		▒	▒	■	■	a
■	▒	▒	■	3		▒	▒	▒	■	b
■	■	■	▒	4		▒	■	■	▒	c
■	■	▒	▒	5		▒	■	▒	▒	d
■	▒	■	▒	6		▒	▒	■	▒	e
■	▒	▒	▒	7		▒	▒	▒	▒	f

It is interesting to observe that a trade-off is involved regarding the required number of fault locations with the controllability of the plaintext. If the plaintext is known and can be controlled, the number of required fault locations are low. On the other hand, the number of different fault locations increases if the plaintext kept fixed. This can be directly attributed to the leakage characteristics of the gates. The leakage from AND gets are more useful while its inputs are varying and it is exactly opposite for the XOR gates. It is worth mentioning that the middle round attacks can also be realized by corrupting several higher order monomials in the S-Box polynomials. However, due to the relatively low leakage from AND gates for one fault, the number of injections required per location is supposed to be higher.

From the next section onward, we shall focus on attacking masked implementations. Although, masking is not meant for fault attack prevention, in certain cases it may aid the fault attack countermeasures [17]. The study on masking becomes more relevant in the present context because our attacks, in principle, are close to SCA attacks (in the sense that both tries to recover values of some intermediate state).

4 Attack on Masked Implementations

Masking is a provably secure countermeasure for SCA attacks. Loosely speaking, masking implements secret sharing at the level of circuits. Over the years, several masking schemes have been proposed, the most popular one being the Threshold-Implementation (TI) [20]. For illustration purpose, in this work, we shall mostly use TI implementations.

Before going into the details of our proposed attack on masking, let us briefly comment on why SEA does not work on masking. Each fault injection in the SEA reveals one bit of information. However, each actual bit of a cipher is shared in multiple bits in the case of masking, and in order to recover the actual bit,

all three bits have to be recovered, simultaneously. Moreover, the mask changes at each execution of the cipher. Hence, even if a single bit is recovered with SEA, it becomes useless as the next execution of the cipher is suppose to change this bit with probability $\frac{1}{2}$. By the same argument, attacking linear terms in the masked S-Box polynomials would not work for key/state recovery, as attacking linear monomials typically imply faulting a XOR gate input. As a XOR gate only leaks about the faulted input bit, in this case, the attacker will end up recovering a uniformly random masked bit.

4.1 Leakage from Masking

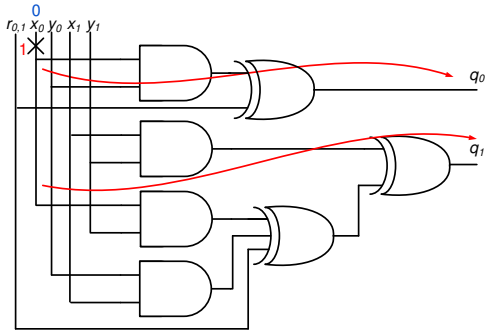

Fig. 4: Fault propagation through masked AND gate.

Table 4: Output status for faulted masked AND gate for different input values. The variables x and y are used for representing the unshared variables (i.e. $x_0 + x_1 = x$ and $y_0 + y_1 = y$). C and F denote correct and faulty outputs.

x_0	x	y_0	y	$r_{0,1}$	C/F	x_0	x	y_0	y	$r_{0,1}$	C/F
0	0	0	0	0	C	0	0	0	0	1	C
0	0	0	1	0	F	0	0	0	1	1	F
0	1	0	0	0	C	0	1	0	0	1	C
0	1	0	1	0	F	0	1	0	1	1	F
0	0	1	1	0	F	0	0	1	1	1	F
0	0	1	0	0	C	0	0	1	0	1	C
0	1	1	1	0	F	0	1	1	1	1	F
0	1	1	0	0	C	0	1	1	0	1	C
1	1	0	0	0	C	1	1	0	0	1	C
1	1	0	1	0	C	1	1	0	1	1	C
1	0	0	0	0	C	1	0	0	0	1	C
1	0	0	1	0	C	1	0	0	1	1	C
1	1	1	1	0	C	1	1	1	1	1	C
1	1	1	0	0	C	1	1	1	0	1	C
1	0	1	1	0	C	1	0	1	1	1	C
1	0	1	0	0	C	1	0	1	0	1	C

Let us recall the unique property of AND gates that they leak about multiple bits, simultaneously. We typically exploit this property for breaching the security of masked implementations. To illustrate how the leakage happens we start with a simple example. Consider the circuit depicted in Fig. 4, which corresponds to the first-order masked AND gate. The corresponding ANF equations are given as follows:

$$\begin{aligned} q_0 &= x_0y_0 + r_{0,1} \\ q_1 &= x_1y_1 + (r_{0,1} + x_0y_1 + x_1y_0) \end{aligned} \tag{3}$$

Here (q_0, q_1) represents the output shares and (x_0, x_1), (y_0, y_1) represents the input shares. We assume that actual unmasked input to the gate (denoted as x and y) remains fixed. However, all the shares vary randomly due to the property of masking. Consequently, all the inputs to the constituent gates of the masked circuit also vary randomly. Without loss of generality, let us now consider that a stuck-at-1 fault is induced at the input share x_0 during the computation of both the output shares. Now, from the ANF expression, it can be observed that x_0 is AND-ed with y_0 and y_1 in two separate shares. So, faulting x_0 would leak information about both y_0 and y_1. From the properties of the AND gate, the stuck-at-1 fault will propagate to the output only if $x_0 = 0$ and $y_i = 1$ with $i \in \{0, 1\}$. However, it should also be noted that faults from both of the gates should not propagate, simultaneously. This is because in that case, the faults will cancel each other. The actual output of the masked AND circuit (i.e., $q_0 + q_1$) will be faulty only if one of the constituent AND gates propagate the fault effect. More specifically, the effective fault propagation requires either $(y_0 = 0, y_1 = 1)$ or $(y_0 = 1, y_1 = 0)$. In other words, *the fault will propagate if and only if the actual unshared bit y ($y = y_0 + y_1$) equals to 1. There will be no fault propagation if $y = 0$.* The fact is illustrated in the truth table presented in Table. 4.

The above-mentioned observation establishes the fact that *a properly placed fault can leak the actual unshared input bits from a masked implementation.* This observation is sufficient for bypassing masking countermeasures as we shall show subsequently in this paper. However, to strongly establish our claim we go through several examples before describing a complete attack algorithm.

4.2 Leakage from TI AND Gates

The second example of ours involves a TI implemented AND gate. We specifically focus on a four-share realization of a first-order masked AND gate proposed by Nikova et al. [20]. The ANF representation of the implementation is given as:

$$\begin{aligned} q_0 &= (x_2 + x_3)(y_1 + y_2) + y_1 + y_2 + y_3 + x_1 + x_2 + x_3 \\ q_1 &= (x_0 + x_2)(y_0 + y_3) + y_0 + y_2 + y_3 + x_0 + x_2 + x_3 \\ q_2 &= (x_1 + x_3)(y_0 + y_3) + y_1 + x_1 \\ q_3 &= (x_0 + x_1)(y_1 + y_2) + y_0 + x_0 \end{aligned} \tag{4}$$

Let us consider a fault injection at the input share x_3 which sets it to 0. An in-depth investigation of the ANF equations reveal that x_3 is multiplied with

y_1+y_2 and y_0+y_3. The leakage due to this fault will reach the output only when $y_0+y_1+y_2+y_3 = y = 1$. One may notice that x_3 also exists as linear monomial in the ANF expressions. However, the effect of this linear monomial gets canceled out in the computation of the actual output bit. Hence the fault effect of this linear term does not hamper the desired fault propagation. In essence, the TI AND gate is not secured against the proposed attack model.

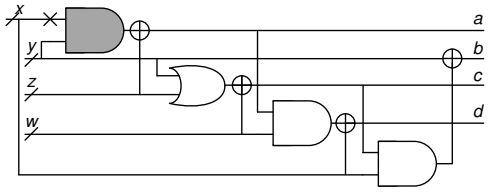

Fig. 5: Fault propagation through an S-Box having TI gates. The fault location is marked with the cross. Note that each wire is of 4-bit width.

TI AND gates are often utilized as constituents for Masked S-Boxes. One prominent example of this is a compact 4-bit S-Box from [31]. The circuit diagram of the S-Box is depicted in Fig. 5. We specifically target the highlighted AND gate in the structure, which is TI implemented. If we inject the same fault as the TI AND gate, the fault effect propagates to the output with exactly the same probability as the TI AND. This is because there is no non-linear gate in the output propagation path of this fault. As a result, we can conclude that even this S-Box leaks. It is worth mentioning that the choice of the target AND gate is arbitrary and, in principle, any of the TI AND gates depicted in the circuit can be targeted. One may also target the OR gate based on the same principle. However, the non-controlling input or OR being 1, the leakage will happen for the input value 0 instead of input value 1.

One important practical question is *how many of such desired fault locations may exist for a masked implementation.* It turns out there are plenty of such locations even for the simple TI AND gate implementation. It is apparent that any of the input shares from (x_0, x_1, x_2, x_3) or (y_0, y_1, y_2, y_3) can be used for causing leakage. In fact, changing the target input share will enable recovery of both x and y separately. Another critical question here is that *whether there will always exist such favorable situations where faulting a share will lead to the leakage of an unmasked bit.* We argue that it will always be the case because the output of any masking scheme must always satisfy the property of correctness. Putting it in a different way, the output of the masked AND gate must always result in $q = xy = (x_0 + x_1 + x_2 + x_3)(y_0 + y_1 + y_2 + y_3)$. Although shares are never supposed to be combined during the masked computation, ensuring correctness always requires that the monomials x_3y_0 x_3y_1, x_3y_2 and x_3y_3 are computed at some share during the masked computation (considering x_3 to be the fault location). Hence, irrespective of the masking scheme used, we are supposed to get fault locations which are exploitable for our purpose (i.e. leaks

$(y_0+y_1+y_2+y_3)=y$). Finding out such locations becomes even easier with our template based setup where extensive profiling of the implementation is feasible for known key values.

So far we have discussed regarding the feasibility of leakage for AND gates and indirectly shared masked implementations of S-Boxes. The obvious next step is to verify our claim for explicitly shared S-Boxes which we elaborate in the next subsection. As it will be shown, attacks are still possible for such S-Boxes.

Table 5: The GIFT S-Box

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$S(x)$	1	a	4	c	6	f	3	9	2	d	b	7	5	0	8	e

4.3 Leakage from Shared S-Boxes

There are numerous examples of TI S-Boxes in the literature. For the sake of illustration, we choose the 4×4 S-Box from the GIFT block cipher [3]. The actual mapping of the S-Box is shown in Table. 5. For our purpose, we select the 3-share TI implementation of this S-Box proposed in [14]. One should note that the GIFT S-Box is originally cubic. In order to realize a 3-shared TI, the original S-Box function $S : GF(2)^4 \rightarrow GF(2)^4$ is broken into two bijective sub-functions $F : GF(2)^4 \rightarrow GF(2)^4$ and $G : GF(2)^4 \rightarrow GF(2)^4$, such that $S(X) = F(G(X))$. Both F and G are quadratic functions for which 3-share TI is feasible. In [14], it was found that for the most optimized implementation in terms of Gate Equivalence (GE), F and G should be constructed as follows:

$$
\begin{aligned}
&G(x_3,x_2,x_1,x_0) = (g_3,g_2,g_1,g_0) \\
&g_3 = x_0 + x_1 + x_1x_2 \\
&g_2 = 1 + x_2 \\
&g_1 = x_1 + x_2x_0 \\
&g_0 = x_0 + x_1 + x_1x_0 + x_2 + x_3
\end{aligned}
\tag{5}
$$

$$
\begin{aligned}
&F(x_3,x_2,x_1,x_0) = (f_3,f_2,f_1,f_0) \\
&f_3 = x_1x_0 + x_3 \\
&f_2 = 1 + x_1 + x_2 + x_3 + x_3x_0 \\
&f_1 = x_0 + x_1 \\
&f_0 = 1 + x_0
\end{aligned}
\tag{6}
$$

Both G and F are shared into three functions each denoted as G_1, G_2, G_3 and F_1, F_2, F_3, respectively. Details of these shared functions can be found in [14]. For our current purpose, we only focus on the shares corresponding to the bit g_0 of G. The ANF equations corresponding to this bit are given as follows:

$$
\begin{aligned}
g_{10} &= x_0^3 + x_1^3 + x_2^3 + x_3^3 + x_0^2x_1^2 + x_0^2x_1^3 + x_0^3x_1^2 \\
g_{20} &= x_0^1 + x_1^1 + x_2^1 + x_3^1 + x_0^1x_1^3 + x_0^3x_1^1 + x_0^3x_1^3 \\
g_{30} &= x_0^2 + x_1^2 + x_2^2 + x_3^2 + x_0^1x_1^1 + x_0^1x_1^2 + x_0^2x_1^1
\end{aligned}
\tag{7}
$$

Here $x_i = x_i^3 + x_i^2 + x_i^1$ for $i \in \{0,1,2,3\}$, and $g_0 = g_{10} + g_{20} + g_{30}$.

We now search for suitable fault locations for our purpose. One such feasible location is x_0^2. One should observe that the leakage due to this fault injection actually depends upon $(x_1^1 + x_1^2 + x_1^3 + 1) = x_1 + 1$. Hence the fault propagation will take place in this case while x_1 is equal to zero. Similarly, it can be shown that a fault injection at x_1^2 will leak the actual value of x_0. *One interesting observation here is that fault injection at any of the shares of an input bit x_i is equivalent to the injection at any other share of the same input. This is because all of them actually cause the leakage of the other unshared input bit associated.* This is, in fact, extremely useful from an attacker's point of view as she may select any one of them for leaking information.

Table 6: Output status for faulted masked AND gate for different input values with bit-flip fault. The variables x and y are used for representing the unshared variables (i.e. $x_0 + x_1 = x$ and $y_0 + y_1 = y$).

x_0	x	y_0	y	$r_{0,1}$	C/F	x_0	x	y_0	y	$r_{0,1}$	C/F
0	0	0	0	0	C	0	0	0	0	1	C
0	0	0	1	0	F	0	0	0	1	1	F
0	1	0	0	0	C	0	1	0	0	1	C
0	1	0	1	0	F	0	1	0	1	1	F
0	0	1	1	0	F	0	0	1	1	1	F
0	0	1	0	0	C	0	0	1	0	1	C
0	1	1	1	0	F	0	1	1	1	1	F
0	1	1	0	0	C	0	1	1	0	1	C
1	1	0	0	0	C	1	1	0	0	1	C
1	1	0	1	0	F	1	1	0	1	1	F
1	0	0	0	0	C	1	0	0	0	1	C
1	0	0	1	0	F	1	0	0	1	1	F
1	1	1	1	0	F	1	1	1	1	1	F
1	1	1	0	0	C	1	1	1	0	1	C
1	0	1	1	0	F	1	0	1	1	1	F
1	0	1	0	0	C	1	0	1	0	1	C

4.4 Different Fault Models

So far in this paper, we have mostly utilized stuck-at faults for all our illustrations. The attacks are equivalent to stuck-at-0 and stuck-at-1 fault models. Interestingly, they are also equally applicable while the fault flips the value of the target bit. To show why it works we recall the concept of fault activation and propagation described at the beginning of this work. Fault reaches the output of a gate from its input only while these two events are satisfied, simultaneously. Considering AND gates (and other non-linear gates), the fault activation depends on specific values at the target input for stuck-at faults (value 0 for stuck-at 1, and value 1 for stuck-at 0). However, for bit flip faults the fault is always active. In other words, in the case of stuck-at faults the fault activation event happens with probability $\frac{1}{2}$, whereas for toggle faults it happens with probability 1. The fault propagation, however, still depends on the occurrence of a non-controlling value at other inputs of the gate. Hence, the main property we exploit for attacking masking schemes (that is, the fault propagation to the output depends on the value of unmasked bits) still holds for bit flip fault models, and attacks are still feasible. As concrete proof of our claim, we present the

truth table corresponding to the simple first order masked AND gate once again in Table 6, this time for a bit flip fault at x_0.

4.5 Template Attack on Masked PRESENT: Main Idea

In this subsection, we utilize the concepts developed in the previous subsections for attacking a complete block cipher implementation. Once again we choose PRESENT as our target. A three-share TI implementation of PRESENT, with simple redundancy countermeasure, is considered for our experiments. As for the three-shared TI, we implemented the lightweight scheme proposed by Poschmann et al. [22]. The implementation technique for the S-Box closely resembles the GIFT S-Box discussed in Sec. 4.3. Considering the fact that PRESENT S-Box is also cubic, it is first represented as a combination of two quadratic bijective mappings F and G. Each of these mappings is then converted to three-shared TI implementations. The implementation of the linear mappings is straightforward. For the sake of completeness, the keys are also masked. As for the fault countermeasure is concerned, we implemented the most common form of redundancy, where the redundancy check happens at the final stage just before outputting the ciphertext. Two separate copies of the masked PRESENT with different masks are instantiated as two redundant branches of computation. Upon detection of a fault, the output is muted or randomized[11].

The three-shared ANF equations for F and G functions can be found in [22]. For our purpose, it is sufficient to focus only on the shared implementation of F which is given below. For the sake of illustration, we first present the unshared version of F (Eq. (8)), and then the shares corresponding to it (Eq. (9)).

$$\begin{aligned} &F(x_3,x_2,x_1,x_0) = (f_3,f_2,f_1,f_0) \\ &\quad f_3 = x_2 + x_1 + x_0 + x_3x_0; f_2 = x_3 + x_1x_0; f_1 = x_2 + x_1 + x_3x_0; \\ &\quad f_0 = x_1 + x_2x_0. \end{aligned} \tag{8}$$

$$\begin{aligned} f_{10} &= x_1^2 + x_2^2x_0^2 + x_2^2x_0^3 + x_2^3x_0^2 & f_{11} &= x_2^2 + x_1^2 + x_3^2x_0^2 + x_3^2x_0^3 + x_3^3x_0^2 \\ f_{20} &= x_1^3 + x_2^3x_0^3 + x_2^1x_0^3 + x_2^3x_0^1 & f_{21} &= x_2^3 + x_1^3 + x_3^3x_0^3 + x_3^1x_0^3 + x_3^3x_0^1 \\ f_{30} &= x_1^1 + x_2^1x_0^1 + x_2^1x_0^2 + x_2^2x_0^1 & f_{31} &= x_2^1 + x_1^1 + x_3^1x_0^1 + x_3^1x_0^2 + x_3^2x_0^1 \end{aligned} \tag{9}$$

$$\begin{aligned} f_{12} &= x_3^2 + x_1^2x_0^2 + x_1^2x_0^3 + x_1^3x_0^2 & f_{13} &= x_2^2 + x_1^2 + x_0^2 + x_3^2x_0^2 + x_3^2x_0^3 + x_3^3x_0^2 \\ f_{22} &= x_3^3 + x_1^3x_0^3 + x_1^1x_0^3 + x_1^3x_0^1 & f_{23} &= x_2^3 + x_1^3 + x_0^3 + x_3^3x_0^3 + x_3^1x_0^3 + x_3^3x_0^1 \\ f_{32} &= x_3^1 + x_1^1x_0^1 + x_1^1x_0^2 + x_1^2x_0^1 & f_{33} &= x_2^1 + x_1^1 + x_0^1 + x_3^1x_0^1 + x_3^1x_0^2 + x_3^2x_0^1 \end{aligned}$$

4.6 Middle Round Attacks

The most interesting question in the current context is how to attack the middle rounds of a cipher without direct access to the plaintexts or ciphertexts. The

[11] Actually, our attacks do not depend on this choice and would equally apply for any detection-type countermeasure

attacks in the first round with known plaintext will become trivial once the middle round attacks are figured out. Note that, *in all of these attacks (even for the known plaintext case), we assume the plaintext to be fixed, whereas the masks vary randomly.* The attacker is only provided with the information whether the outcome is faulty or not and nothing else. For the case of middle round attacks, the value of the fixed plaintext is unknown to the adversary.

4.6.1 Template Construction

Algorithm 5 *BUILD_TEMPLATE_MASK*

Input: Masked cipher S, Faults $fl_0, fl_1, \cdots, fl_h$, Number of masked executions per input M
Output: Template $\mathcal{T}$

```
T := ∅
w := GET_SBOX_SIZE()                                   ▷ Get the width of the S-Box
for (0 ≤ x ≤ 2^w) do
    F_t := ∅
    for each fl ∈ {fl_0, fl_1, ·fl_h} do
        V := ∅
        for m_ind ≤ M do
            m := GEN_MASK()                            ▷ Generate fresh mask for each execution
            y_f := C(x, m)^fl                          ▷ Inject fault in one copy of the S-Box for each execution
            m := GEN_MASK()
            y_c := C(x, m)
            if DETECT_FAULT(y_f, y_c) == 1 then        ▷ Fault detection function
                V := V ∪ {1}
            else
                V := V ∪ {0}
            end if
        end for
        if V ∼ D_1 then
            F_t := F_t ∪ {1}
        else
            F_t := F_t ∪ {0}
        end if
    end for
    T := T ∪ {(F_t, x)}
end for
Return T
```

Table 7: Template for attacking TI PRESENT (middle round). The black cells indicate a faulty outcome and yellow cells represent correct outcome.

$fl_0 = x_0^2$ (f_{10}, f_{20}, f_{30})	$fl_1 = x_0^2$ (f_{11}, f_{21}, f_{31})	$fl_2 = x_0^2$ (f_{12}, f_{22}, f_{32})	$fl_3 = x_3^2$ (f_{13}, f_{23}, f_{33})	State		$fl_0 = x_0^2$ (f_{10}, f_{20}, f_{30})	$fl_1 = x_0^2$ (f_{11}, f_{21}, f_{31})	$fl_2 = x_0^2$ (f_{12}, f_{22}, f_{32})	$fl_3 = x_3^2$ (f_{13}, f_{23}, f_{33})	State
■	▒	■	■	0		■	▒	▒	■	8
■	■	■	▒	1		■	■	▒	▒	9
▒	■	▒	■	2		▒	■	■	▒	a
▒	▒	■	▒	3		▒	▒	▒	■	b
▒	■	■	■	4		▒	■	▒	▒	c
▒	▒	▒	▒	5		▒	▒	■	■	d
■	▒	▒	▒	6		■	▒	■	▒	e
■	■	▒	■	7		■	■	■	■	f

The very first step of the attack is template-building. The attacker is assumed to have complete knowledge of the implementation and key, and also can figure out suitable locations for fault injection. One critical question here is how many different fault locations will be required for the attack to happen. Let us take a closer look at this issue in the context of the shared PRESENT S-Box. Without loss of generality, let us assume the input share x_0^2 as the fault injection point during the computation of the shares (f_{10}, f_{20}, f_{30}). It is easy to observe that this fault leaks about the expression $(x_2^2+x_2^3+x_2^1) = x_2$. In a similar fashion the fault location x_0^2 during the computation of the shares (f_{11}, f_{21}, f_{31}) leaks about x_3; the location x_0^2 during the computation of the shares (f_{12}, f_{22}, f_{32}) leaks about x_1; and the location x_3^2 during the computation of (f_{13}, f_{23}, f_{33}) leaks about x_0. Consequently, we obtain the template shown in Table 7 for independent injections at these selected locations.

The template construction algorithm is outlined in Algorithm 5. The aim is to characterize each S-Box input (denoted as x in the Algorithm 5) with respect to the fault locations. The plaintext nibble is kept fixed in this case during each fault injection campaign, while the mask varies randomly. One important observation at this point is that the fault injection campaign has to be repeated several times with different random mask for each valuation of an S-Box input. To understand why this is required, once again we go back to the concept of fault activation and propagation. Let us consider any of the target fault locations; for example, x_0^2. The expression which leaks information is $(x_2^2 + x_2^3 + x_2^1)$. Now, for the fault to be activated in a stuck-at fault scenario, x_0^2 must take a specific value (0 or 1 depending on the fault). Since all the shared values change randomly at each execution of the cipher, we can expect that the fault activation happens with probability $\frac{1}{2}$ [12]. Once the fault is activated, the propagation happens, depending on the value of the other input of the gate which actually causes the leakage. In order to let the fault activate, the injection campaigns have to run several times, corresponding to a specific fault location for both the template building and online attack stage. Given the activation probability of $\frac{1}{2}$, 2 executions (injections) with different valuations at x_0^2, would be required on average.

As a consequence of performing several executions of the cipher corresponding to one fault location, we are supposed to obtain a set of suggestions for the valuation of the bit to be leaked. For example, for two separate executions we may get two separate suggestions for the value of $(x_2^2+x_2^3+x_2^1)$. If the fault at x_0^2 is not activated, the suggestion will always be 0. However, if the fault is activated the suggestion reflects the actual value of x_2. There is no way of understanding when the fault at x_0^2 gets activated. So, a suitable technique has to be figured out to discover the actual value of x_2 from the obtained set of values. Fortunately, the solution to this problem is simple. Let us consider the set of observables corresponding to a specific fault location as a random variable $\mathcal{V}$ taking values 0 or 1. The value of $\mathcal{V}$ is zero if no fault propagates to the output and 1, otherwise. Mathematically, $\mathcal{V}$ can be assumed as a Bernoulli distributed random variable. Now, it is easy to observe that *if the actual value to be leaked is* 0*,* $\mathcal{V}$ *will never*

[12] for bit-flip faults, however, the fault activation will happen with probability 1

take a value 1 *(that is, the fault never propagates to the output).* Therefore, the probability distribution of $\mathcal{V}$ for this case can be written as:

Algorithm 6 *MATCH_TEMPLATE_MASK*

Input: Protected cipher with unknown key C_k, Faults $fl_0, fl_1, ..., fl_h$, Template $\mathcal{T}$
Output: Set of candidate correct keys k_c

```
k_cand := ∅                                          ▷ Set of candidate keys
w := GET_SBOX_SIZE()
F_t := ∅
for each fl ∈ {fl_0, fl_1, ·fl_h} do
    V := ∅
    for m_ind ≤ M do
        O := (C_k(P))^fl                             ▷ Inject fault for each masked execution
        if (O == 1) then                             ▷ Fault detected
            V := V ∪ {1}
        else
            V := V ∪ {0}
        end if
    end for
    if V ~ D_1 then
        F_t := F_t ∪ {1}
    else
        F_t := F_t ∪ {0}
    end if
end for
k_cand := k_cand ∪ {T(F_t)}
Return k_cand
```

$$\mathcal{D}_0 : \mathbb{P}[\mathcal{V} = 0] = 1 \text{ and } \quad \mathbb{P}[\mathcal{V} = 1] = 0 \tag{10}$$

In contrast, if the value to be leaked is 1, the probability distribution of $\mathcal{V}$ becomes:

$$\mathcal{D}_1 : \mathbb{P}[\mathcal{V} = 0] = \frac{1}{2} \text{ and } \quad \mathbb{P}[\mathcal{V} = 1] = \frac{1}{2} \tag{11}$$

The template construction procedure becomes easy after the identification of these two distributions. *More precisely, if $\mathcal{V} \sim \mathcal{D}_0$ the corresponding location in the template takes a value 0. The opposite thing happens for $\mathcal{V} \sim \mathcal{D}_1$.*

4.6.2 Online Phase

The online phase of the attack algorithm is outlined in Algorithm 6. Fundamentally it is similar to the template construction phase. We keep the plaintext fixed and run the fault campaigns at pre-decided locations. The templates are decided by observing the output distributions of the random variable $\mathcal{V}$ as described in the previous section. At the end of this step, one round of the cipher is completely recovered. However, in order to recover the complete round key, recovery of two consecutive rounds is essential. Recovery of another round is trivial with this approach, and therefore a round key can be recovered uniquely.

Number of Faults: In the case of PRESENT, we use 4 fault locations, and each of them requires several fault injections with the mask changing randomly. The number of injections required for each of these locations depends upon the

number of samples required to accurately estimate the distribution of the variable $\mathcal{V}$. In an ideal case, two fault injections on average should reveal the actual leakage. Experimentally, we found that 4-5 injections on average are required to reveal the actual distribution of $\mathcal{V}$. The increased number is probably caused by the fact that an entire mask of 128-bit is generated randomly in our implementation and the activation of an injected fault happens with a slightly different probability than expected. Assuming, 5 injections required per fault location the total number of fault requirements for a nibble becomes roughly 20. Therefore, around $32 \times 20 = 640$ faults are required to extract the entire round key of the PRESENT cipher [13]. Note that, in practical experiments, this number may rise given the fact that some of the injections may be unsuccessful. However, given the fact that the accuracy of modern fault injection setups like laser stations are almost 100%, this should not be of significant concern. It is somewhat apparent that the attack outlined above will also work for situations where the plaintext is known. In fact, the required number of injections reduces 320 in that case.

5 Practical Validation

In this section, we evaluate the applicability of the proposed FTA strategy for a publicly available masked implementation. One should note that profiling of the target implementation to detect desired fault locations is an important factor in FTA attacks. This section demonstrates how to perform such profiling for a relatively less understood public implementation. The target implementation of ours is a masked implementation of AES targeted for 32-bit Cortex M4 platform with Thumb-2 instruction set. Since the original implementation lacks fault countermeasure, we added simple temporal redundancy, that is the cipher is executed multiple times, and the ciphertexts are matched before output. In all of our experiments, the observable is a string of $0, 1$ bits with its corresponding interpretations.

5.1 Simulated Experiments on a Public Implementation of AES

So far in this paper, we have mainly demonstrated the attacks in the context of PRESENT block cipher. However, it is interesting to analyze whether AES is also susceptible to the proposed attacks. Although in principle the answer should be yes, it is always important to analyze the attacks for third-party implementations. With this viewpoint, we choose one publicly available implementation of optimized, bit-sliced, 1st-order masked AES from [1]. The masked S-Box in this implementation utilizes Trichina gates [29] for SCA protection.

[13] Given the fact, that PRESENT uses an 80-bit master key, and 64-bit round keys, the remaining keyspace after one round key extraction would be of size 2^{16}, which is trivial to search exhaustively.

Analyzing the S-Box: The main concern of analyzing third-party implementations is that the high-level structure is not very well-understood during profiling. This being a practical issue, we decide to handle it with a simple trial-and-error based profiling of the S-Box. We target each instruction at once and simulate a bit stuck-at or bit-flip fault for one of its operands. One should note that in this experiment, we do not restrict ourselves to the faults in the input shares during the shared execution of a single bit. The faults can now happen at any intermediate wire, and we accept them as long as they are found useful for constructing templates. The compiled code in Thumb-2 of the S-Box is found to have 2621 instructions in total. It was found that a total of 1102 among them results in exploitable fault locations in our case. The exploitability was decided based on the fact whether the fault location can reduce the entropy of the S-Box input. The result of this experiment is summarized in Table. 8 and it indicates that one can have plenty of exploitable fault locations to run practical FTA attacks.

Table 8: Summary of exploitable instructions.

Total Instruction Count	# Exploitable Instructions	% Vulnerable Instructions
2621	1102	42.6

Table 9: Summary of the templates for different fault models.

Fault Model	#Fault Locations	#Distinct Patterns	#Patterns with 2 value suggestions	#Patterns with 1 value suggestion
Stuck-at	16	200	56	144
Bit-flip	15	198	58	140

Different Fault Models: The next step is to construct templates and use them to perform full-scale attacks. We specifically consider two different fault models for template construction: 1) stuck-at fault; 2) bit flip fault. The corresponding templates are summarized in Table 9. For the first model, we found 200 distinct patterns in the template having 16 different fault locations. Each pattern maps to either one or two suggestions for the intermediate state byte value. The result for the other case is very similar. During the online phase, it was found that roughly $7 - 8$ fault injections per location with different mask values are sufficient for template matching. One should note that none of the templates constructed can uniquely identify a complete state. In the worst case, we may get 2^{16} equally likely suggestions for one single intermediate round[14]. As two consecutive states are required in the case of middle round attack, the total number of key suggestions become $2^{16} \times 2^{16} = 2^{32}$. However, one should note that this is simply a worst case estimate and in practice, the attack complexity is supposed to be lower than this. Even if the complexity reaches the worst-case estimate, the exhaustive search complexity of 2^{32} is fairly reasonable. It is worth mentioning that the choice of this AES implementation was entirely random and subject to the availability of public codes. To summarize, the FTA attacks work fairly well for masked AES implementations having fault countermeasures and suitable measures should be considered.

The attack applies to popular design styles for both software and hardware implementations of ciphers with TI protection. To illustrate, consider the TI implementation as shown in [26] , where table lookups have been utilized for

[14] Although, in our experiments, we got several states with single suggestions.

processing the input shares of an S-Box. This specific implementation performs the share computations in a per-bit manner via table look-up (bits corresponding to different independent S-Boxes are packed in one register). For our purpose, it is sufficient to inject a bit fault during this table look-up which is a common fault model used for FAs [16]. In the next section, we discuss some of the state-of-the-art FA countermeasures in the context of FTA.

6 Possible Countermeasures

In this section, we discuss the applicability of some of the well-known fault attack countermeasures for preventing the proposed attacks. Both the middle round and known plaintext attacks on the masking schemes are taken into consideration.

6.1 Device-Level Countermeasures:

Self-destruction is one of the most radical steps that can be taken to prevent against FAs. However, given the fact that most of the cryptographic devices in the modern day is supposed to operate in an open environment, self-destruction can be extremely costly and will have a very low yield. This is because small embedded devices cannot afford to have extremely efficient methods to handle power-spikes and electromagnetic radiation effects. As a result, deciding between malicious fault and accidental fault becomes almost impossible. One reasonable trade-off could be to destroy the device after a certain number of faults has been encountered. However, a resourceful attacker may always try to bypass it by first corrupting the fault counter, which is reasonable with any standard lab setup and may not even require precise faults. Another option to prevent FA is to use tamper resilient shielding. However, this is not cost-effective for most of the embedded devices and can also be bypassed by careful de-packaging of the chip.

6.2 Infection Countermeasures

Infection countermeasures were mainly proposed to get rid of the explicit check operation often used in detection countermeasures. The explicit check operation has been shown to have serious consequences in the context of security [11, 30]. Another distinct property of infection countermeasures is that they try to make the faulty ciphertexts unexploitable by somehow destroying (often called infecting) the useful patterns within them. Usually, an infection function is called upon detection (not via explicit check) of a fault to infect the computation. In the present context, we consider the infection countermeasure proposed by Tupsamudre et al. in [30]. The infection function is fairly simple albeit effective in this case. The idea is to output a uniformly random string upon the detection of a fault. Additionally, the countermeasure involves random dummy round computation to make a targeted fault injection difficult.

The infection function and the fault detection mechanism do not found to have any significant effect on the proposed attacks. However, the dummy rounds have some interesting effects. The presence of dummy rounds makes the observable distributions noisy thus making the template matching difficult at the attack phase. However, the attacks cannot be fully mitigated. In particular, we found that the middle round attacks were throttled in the case of PRESENT due to an excessive number of key suggestions returned during template matching. However, the known plaintext version of the attack can still work with slightly higher complexity. Further analysis of infection countermeasures can be found in the Appendix D.

6.3 Code-based Error-Detection

Code-based error detection is one of the lightweight alternatives for throttling fault attacks. The low resource overhead comes at the cost of limited fault coverage. The simplest example of code-based error detection is simple single-bit parity checking which can detect 50% of the injected faults. The error-detection capability can be improved further by using non-linear codes [18]. The proposed attack strategy remains unaltered at the presence of such countermeasures. This can be explained by the fact that even if some of the errors remain undetected, the distribution $\mathcal{D}_0$ in Eq. 10 remains unaltered. Although the distribution $\mathcal{D}_1$ might get affected slightly, it still remains distinguishable from $\mathcal{D}_0$. On the other hand, code-based detection schemes with high error detection rate behave almost identically with standard time/space redundancy countermeasures. Hence, the proposed attacks would not get throttled with such detection schemes.

6.4 Error Correction

Error correction is an alternative countermeasure strategy, which is relatively less explored compared to other countermeasure classes. Error correction can be somewhat effective in the present context. However, with a little more power given to the adversary, the effectiveness of error correction may fall short. One should recall that the attacks described in this paper only need to know whether the fault has happened or not. An adversary having the power of measuring side-channel information may still get this information even in the presence of error correction. This is because the error correction logic is supposed to make more transitions while it has to correct a bit, than the situations while nothing has to be corrected. Also, an error would make a valid code-word deviate from its predefined structure. It is not very difficult for a side-channel adversary to detect such deviation via side-channel. The vulnerability of such check operations have already been shown as exploitable in [25] in the context of detection countermeasures. Availability of such information is sufficient to make the FTA attacks work.

7 Conclusion

Modern cryptographic devices incorporate special algorithmic trick to throttle both SCA and FA. In this paper, we propose a series of attacks which can efficiently bypass most of the state-of-the-art countermeasures against SCA and FA even if they are incorporated together. The attacks, abbreviated as FTA, are template-based and exploit the characteristics of basic gates under the influence of faults for information leakage. Although the fault model is similar to the SIFA attacks, the exploitation mechanism is entirely different from SIFA. Most importantly, FTA enables attack on the middle round of a protected cipher implementation, which is beyond the capability of SIFA or any other existing FA technique proposed so far. Middle round attacks without explicit knowledge of plaintexts and ciphertexts may render many well-known block cipher-based cryptographic protocols vulnerable. Practical validation of the attacks has been shown for PRESENT and a publicly available implementation of AES. One interesting future application would be to make these attacks work for secured public key implementations. Another potential future work in this direction is to figure out a suitable countermeasure against FTA attacks.

References

1. Masked-aes-implementation, `https://github.com/Secure-Embedded-Systems/Masked-AES-Implementation`
2. Lightweight Cryptography (2017), `https://www.nist.gov`
3. Banik, S., et. al.: GIFT: a small PRESENT. In: CHES. pp. 321–345. Springer (2017)
4. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. CRYPTO pp. 513–525 (1997)
5. Bogdanov, A., et. al.: PRESENT: An ultra-lightweight block cipher. In: CHES. pp. 450–466. Springer (2007)
6. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: EUROCRYPT. pp. 37–51. Springer (1997)
7. Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In: CHES. pp. 13–28. Springer (2002)
8. Dobraunig, C., et. al.: SIFA: exploiting ineffective fault inductions on symmetric cryptography. TCHES pp. 547–572 (2018)
9. Dobraunig, C., et. al.: Statistical ineffective fault attacks on masked aes with fault countermeasures. In: ASIACRYPT. pp. 315–342. Springer (2018)
10. Ghoshal, A., Patranabis, S., Mukhopadhyay, D.: Template-based fault injection analysis of block ciphers. In: SPACE. pp. 21–36 (2018)
11. Gierlichs, B., Schmidt, J., Tunstall, M.: Infective computation and dummy rounds: fault protection for block ciphers without check-before-output. In: LatinCrypt. pp. 305–321. Springer (2012)
12. Groß, H., Mangard, S., Korak, T.: An efficient side-channel protected aes implementation with arbitrary protection order. In: CT-RSA. pp. 95–112. Springer (2017)

13. Guo, X., Mukhopadhyay, D., Jin, C., Karri, R.: Security analysis of concurrent error detection against differential fault analysis. Journal of Cryptographic Engineering **5**(3), 153–169 (Sep 2015)
14. Gupta, N., et. al.: Threshold implementations of gift: A trade-off analysis. IACR Cryptology ePrint Archive **2017**, 1040 (2017)
15. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: Securing hardware against probing attacks. In: CRYPTO. pp. 463–481. Springer (2003)
16. Joye, M., Tunstall, M.: Fault analysis in cryptography, vol. 147. Springer (2012)
17. Korkikian, R., Pelissier, S., Naccache, D.: Blind fault attack against spn ciphers. In: FDTC. pp. 94–103. IEEE (2014)
18. Kulikowski, K., Karpovsky, M., Taubin, A.: Robust codes for fault attack resistant cryptographic hardware. In: FDTC. pp. 1–12 (2005)
19. Niemi, V., Nyberg, K.: UMTS security. John Wiley & Sons (2006)
20. Nikova, S., Rechberger, C., Rijmen, V.: Threshold implementations against side-channel attacks and glitches. In: ICICS. pp. 529–545. Springer (2006)
21. Pan, J., et. al.: One fault is all it needs: Breaking higher-order masking with persistent fault analysis. Cryptology ePrint Archive, Report 2019/008 (2019), https://eprint.iacr.org/2019/008
22. Poschmann, A., et. al.: Side-channel resistant crypto for less than 2,300 ge. Journal of Cryptology **24**(2), 322–345 (2011)
23. Reparaz, O., et. al.: Consolidating masking schemes. In: CRYPTO. pp. 764–783. Springer (2015)
24. Roscian, C., et. al.: Fault model analysis of laser-induced faults in SRAM memory cells. In: FDTC. pp. 89–98. IEEE (2013)
25. Saha, S., et. al.: Breaking redundancy-based countermeasures with random faults and power side channel. In: FDTC. pp. 15–22 (2018)
26. Sasdrich, P., Bock, R., Moradi, A.: Threshold implementation in software. In: COSADE. pp. 227–244. Springer (2018)
27. Schneider, T., Moradi, A., Güneysu, T.: ParTI–towards combined hardware countermeasures against side-channel and fault-injection attacks. In: CRYPTO. pp. 302–332. Springer (2016)
28. Tajik, S., et. al.: Laser fault attack on physically unclonable functions. In: FDTC. pp. 85–96. IEEE (2015)
29. Trichina, E.: Combinational logic design for aes subbyte transformation on masked data. IACR Cryptology ePrint Archive **2003**, 236 (2003)
30. Tupsamudre, H., Bisht, S., Mukhopadhyay, D.: Destroying fault invariant with randomization. In: CHES. pp. 93–111. Springer (2014)
31. Ullrich, M., et. al.: Finding optimal bitsliced implementations of 4× 4-bit s-boxes. In: SKEW 2011 Symmetric Key Encryption Workshop, Copenhagen, Denmark. pp. 16–17 (2011)
32. Zhang, F., et. al.: Persistent fault analysis on block ciphers. TCHES pp. 150–172 (2018)

Supplementary Material

A Impact of Fault Propagation on Pseudorandomness

In this paper, we demonstrate the impact of fault propagation characteristics on the security of block ciphers. However, conceptually, one could view the observation of fault propagation as a means to compromise the security of any pseudorandom function (PRF) in general. In this section, we shed light on how one can exploit the key-dependent fault propagation behavior exhibited by any PRF to distinguish it from a uniformly random function. We begin by formally defining a PRF.

Pseudorandom Functions. Informally, an efficiently computable function [15] is called pseudorandom if there exists no probabilistic polynomial-time (PPT) adversary that can distinguish it from a truly random function. More formally, a PRF family is an efficiently computable function family $\{F(k,\cdot) : \mathcal{Y} \to \mathcal{Z}\}_{k\in\mathcal{K}}$ (where K, $\mathcal{Y}$ and $\mathcal{Z}$ are indexed by the security parameter λ) such that for all PPT adversaries $\mathcal{A}$ we have

$$\left|\mathbb{P}[\mathcal{A}^{F(k,\cdot)}(1^\lambda) = 1] - \mathbb{P}[\mathcal{A}^{f(\cdot)}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda),$$

where k is a uniformly sampled key from the keyspace $\mathcal{K}$ and $f : \mathcal{Y} \to \mathcal{Z}$ is a (truly) random function.

It follows from this definition that any pair of uniformly sampled functions from a given PRF family should be computationally indistinguishable. More formally, for all PPT adversaries $\mathcal{A}$ we should have

$$\left|\mathbb{P}[\mathcal{A}^{F(k_1,\cdot)}(1^\lambda) = 1] - \mathbb{P}[\mathcal{A}^{F(k_1,\cdot)}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda),$$

where k_1 and k_2 are uniformly sampled keys from the keyspace $\mathcal{K}$. The proof of this claim follows from the definition of a PRF and a simple hybrid argument.

Fault Observability Oracle. Now assume that in addition to the PRF oracle, the adversary $\mathcal{A}$ has black-box access to a second oracle called a "fault observability oracle" (or $\mathcal{O}_{\mathrm{FO}}$ in short). This oracle takes as input a PRF key $k \in \mathcal{K}$ and an input $y \in \mathcal{Y}$, and a fault description fl [16], and outputs a bit $b \in \{0,1\}$. The output bit b is supposed to indicate whether a fault with description fl, upon injection into the PRF evaluation algorithm, propagates to the eventual PRF output. In particular, the adversary $\mathcal{A}$ *does not* have access to the *actual output* of the faulty evaluation algorithm.

In the presence of this additional oracle, the advantage of the adversary $\mathcal{A}$ in distinguishing any pair of uniformly sampled functions from a given PRF family is given by

$$\epsilon := \left|\mathbb{P}[\mathcal{A}^{F(k_1,\cdot),\mathcal{O}_{\mathrm{FO}}(k_1,\cdot,\cdot)}(1^\lambda) = 1] - \mathbb{P}[\mathcal{A}^{f(\cdot),\mathcal{O}_{\mathrm{FO}}(k_2,\cdot,\cdot)}(1^\lambda) = 1]\right|,$$

[15] A function is said to be efficiently computable if there exists a poly-time algorithm that can evaluate this function on any arbitrary valid input.

[16] The fault description fl is assumed to include all information relevant to a fault injection, such as the nature, timing and precise location of the fault

where k_1 and k_2 are uniformly sampled keys from the keyspace $\mathcal{K}$.

Exploiting Key-Dependent Fault Observability. Now suppose that there exists a class of faults for which the corresponding fault observability behavior depends on the PRF key. In this case, the output of the oracle $\mathcal{O}_{\text{FO}}$ leaks some information about the information about the underlying key k, which an adversary $\mathcal{A}$ can exploit to distinguish any pair of uniformly sampled functions from a given PRF family.

More formally, suppose there exists a set of fault descriptions $\mathcal{FS}$ such that for any fault with description $fl \in \mathcal{FS}$, we have

$$|\mathbb{P}[\mathcal{O}_{\text{FO}}(k_1, y, fl) \neq \mathcal{O}_{\text{FO}}(k_2, y, fl)] - 1/2| > \epsilon',$$

where k_1 and k_2 are uniformly sampled keys from the keyspace $\mathcal{K}$, y is any arbitrarily chosen input from the input space $\mathcal{Y}$, and ϵ' is *non-negligible* in the security parameter λ. It is easy to see that by querying the $\mathcal{O}_{\text{FO}}$ on arbitrary inputs fault descriptions in the set $\mathcal{FS}$, the adversary $\mathcal{A}$ can distinguish between $F(k_1, \cdot)$ and $F(k_2, \cdot)$ with advantage at least ϵ'.

In summary, the existence of a fault observability oracle and a set of faults $\mathcal{FS}$ with key-dependent fault behavior poses a threat to the security of any family of PRFs. In this paper, we develop this idea into a full-fledged key-recovery attack on block ciphers. Fundamentally, our approach is based on the following observations:

1. First of all, it is practically feasible to realize fault observability oracle with respect to block ciphers by simply injecting faults into an implementation of the block cipher under test, and observing whether the fault propagates to the ciphertext.
2. Secondly, as pointed out in Section 2, it is possible to inject faults into the circuitry of block cipher sub-components such as S-Boxes, where the fault observability at the output of the S-Box depends on the input. Since the input to an S-Box is typically a function of the secret key and the cipher internal state, Section 2 implies the existence a class of faults $\mathcal{FS}$ as described above.

B More on the Attack from Sec. 3.2: AES Example

It would be interesting to see how the first round attack described in Sec. 3.2 works in the case of AES. The S-Box polynomials for AES has the highest degree of 7. As the first injection location, we choose the highest degree monomial $x_1x_2x_3x_4x_5x_7x_8$ from the polynomial corresponding the 0'th output bit of the AES S-Box. Very similar to the PRESENT case, the fault template here also suggests two keys per pattern. However, the total number of injections required are 256 for the first template and 257 in total to recover a single byte of the key. The first template for the AES attack contains a total of 128 distinct patterns with two key suggestion per pattern. The second template, which is based on fault injection at a linear term, contains two patterns.

C Alternative Fault Template for Masked PRESENT

In this section, we present an alternative fault template for the 3-share implementation of PRESENT. In this case, the fault is injected in the shares of the G function. In fact, concentrating on the shares corresponding to any two bits of G are sufficient. For the sake of illustration, we first present the unshared version of G, and then the shares corresponding to two of the actual output bits of it. The unshared G, and the shares corresponding to g_0 and g_1 are presented in Eq. (12), Eq. (13) and Eq. (14), respectively.

$$\begin{aligned} G(x_3,&x_2,x_1,x_0) = (g_3,g_2,g_1,g_0) \\ & g_3 = x_2 + x_1 + x_0,; g_2 = 1 + x_2 + x_1; g_1 = 1 + x_3 + x_1 + x_2x_0 + x_1x_0; \\ & g_0 = 1 + x_0 + x_3x_2 + x_3x_1 + x_2x_1 \end{aligned} \tag{12}$$

$$\begin{aligned} g_{10} &= 1 + x_0^2 + x_3^2x_2^2 + x_3^2x_2^3 + x_3^3x_2^2 + x_3^2x_1^2 + x_3^2x_1^3 + x_3^3x_1^2 + x_2^2x_1^2 + x_2^2x_1^3 + x_2^3x_1^2 \\ g_{20} &= x_0^3 + x_3^3x_2^3 + x_3^1x_2^3 + x_3^3x_2^1 + x_3^3x_1^3 + x_3^1x_1^3 + x_3^3x_1^1 + x_2^3x_1^3 + x_2^1x_1^3 + x_2^3x_1^1 \\ g_{30} &= x_0^1 + x_3^1x_2^1 + x_3^1x_2^2 + x_3^2x_2^1 + x_3^1x_1^1 + x_3^1x_1^2 + x_3^2x_1^1 + x_2^1x_1^1 + x_2^1x_1^2 + x_2^2x_1^1 \end{aligned} \tag{13}$$

$$\begin{aligned} g_{11} &= 1 + x_3^2 + x_1^2 + x_2^2x_0^2 + x_2^2x_0^3 + x_2^3x_0^2 + x_1^2x_0^2 + x_1^2x_0^3 + x_1^3x_0^2 \\ g_{21} &= x_3^3 + x_1^3 + x_2^3x_0^3 + x_2^1x_0^3 + x_2^3x_0^1 + x_1^3x_0^3 + x_1^1x_0^3 + x_1^3x_0^1 \\ g_{31} &= x_3^1 + x_1^1 + x_2^1x_0^1 + x_2^1x_0^2 + x_2^2x_0^1 + x_1^1x_0^1 + x_1^1x_0^2 + x_1^2x_0^1 \end{aligned} \tag{14}$$

Without loss of generality, let us consider the input x_3^2 as the first fault injection point. It is easy to observe that this fault leaks about the expression $(x_1^1 + x_1^2 + x_1^3 + x_2^1 + x_2^2 + x_2^3) = x_1 + x_2$. Further investigation reveals that fault injection at any share of x_1 in Eq. (13) leaks information regarding $x_3 + x_2$, and a similar injection in one of the shares of x_2 reveals about $x_3 + x_1$. Independent injections at these locations thus reduces the entropy of the three actual bits (x_3, x_2, x_1) to 1 bit. However, no information regarding the bit x_0 can be revealed from Eq. (13) as the shares of x_0 are only present as linear monomials. In order to extract this bit we have to consider the shares from Eq. (14). Corrupting any single share of x_2 or x_1 exposes x_0 in this case. However, one should note that even after the extraction of x_0 the overall entropy of the entire state (x_3, x_2, x_1, x_0) still remains as 1. Consequently, this template provides two suggestions for each S-Box input at an intermediate round. The template is depicted in Table. 10.

D Infection Countermeasures Against FTA Attacks

One of the most prominent infection countermeasure is due to Tupsamudre et. al. [30]. In order to detect a fault, this countermeasure performs two executions of each round (denoted as cipher and redundant rounds). There can be arbitrary number of dummy rounds happening between a cipher and a redundant round.

Table 10: Alternative Template for attacking TI PRESENT (middle round). The black cells indicate a faulty outcome and yellow cells represent correct outcome.

fl_0	fl_1	fl_2	fl_3	State
■	■	■	■	3, 13
■	■	■	■	5, 11
■	■	■	■	2, 12
■	■	■	■	1, 15
■	■	■	■	6, 8
■	■	■	■	0, 14
■	■	■	■	4, 10
■	■	■	■	7, 9

The whole computation is controlled by a random bit-string of fixed length (denoted as *rstr*). A bit zero in *rstr* denotes a dummy round and a bit value of one denotes a cipher or redundant round.

Let us consider FTA for infection countermeasures. To validate the robustness of this countermeasure we implemented it on a 3-share masked PRESENT implementation. In order to prevent SCA-based identification of individual rounds the final key addition operation of PRESENT is converted into a complete round by adding a dummy S-Box layer and pLayer. In other words, the implementation processes 32 cipher and 32 redundant rounds, and a predefined number of dummy rounds. The security against the proposed fault-template attacks was evaluated for 16, 32 and 64 dummy round computations.

The proposed attacks does not get affected by the fact that the ciphertext is randomized upon the detection of a fault. However, the dummy round computations are found to have significant effects. Basically, the existence of dummy rounds add noise to the observables as the target fault location cannot be determined exactly. Let us now have a closer look on these noisy observables. Let the cipher processes total R rounds among which there are total n cipher and redundant rounds and $R-n$ dummy rounds. As already pointed out, an arbitrary number of dummy round computation may take place between any cipher round and its corresponding redundant round. Let us further assume that the fault location is set at loop iteration t.[17] Depending upon the random *rstr* string, the fault injection may either hit the desired cipher round r (or its corresponding redundant round), or some arbitrary cipher or dummy round. The event that fault injection happens at round r (cipher or redundant) is considered as *signal*, whereas injection at any other round or at a dummy round is considered as *noise*.

As mentioned previously, the noise directly affects the observables in this case. This is because even if the fault affects an undesired location, the fault effect would propagate to the output. This fact influences the decision mak-

[17] For simplicity, we assume that the attacker can deterministically identify the target S-Box and fault location within a round

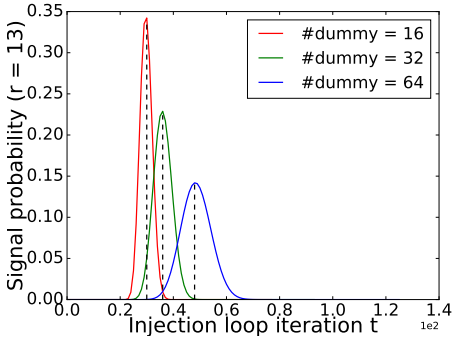

Fig. 6: Variation of signal probability with targeted loop iterations for injection(for different number of dummy rounds).

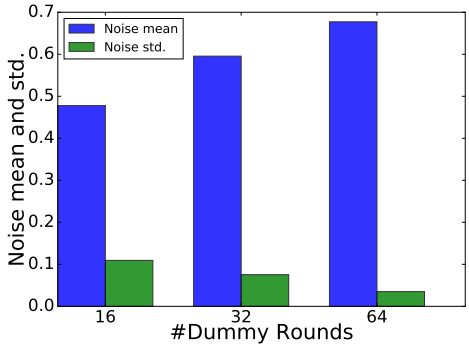

Fig. 7: Mean and standard deviation of noise for different count of dummy rounds.

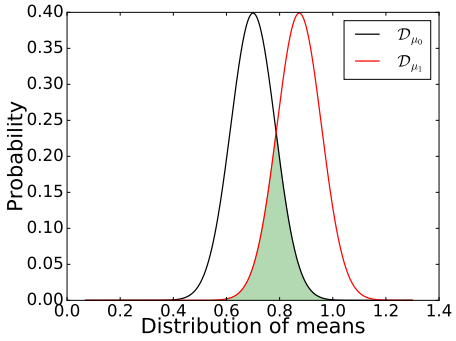

Fig. 8: Overlapping distributions for $\mathcal{D}_{\mu_0}$ and $\mathcal{D}_{\mu_1}$. The shaded region indicates the threshold window.

ing procedure of our attacks. In other words, the fault patterns for template matching cannot be constructed properly during the online phase. As described in Sec. 4.6.1, for a specific fault location inside the S-Box the observable is a Bernoulli distributed random variable ($\mathcal{V}$). The aforementioned noise actually affects this distribution. The random variable corresponding to this noisy distribution is denoted as $\mathcal{V}^{'}$. In order to make the attacks happen, we need to decide the actual fault patterns by nullifying effect of noise. As already shown in Eq. (10), and (11), the noise-free distributions $\mathcal{D}_0$ and $\mathcal{D}_1$ only depend upon the leaked values. The main task there was to decide whether the noise-free random variable for the observable $\mathcal{V}$ is distributed according to $\mathcal{D}_0$ or $\mathcal{D}_1$.

Let us now characterize the noisy distribution. For convenience, let us define another random variable $\mathcal{V}_n$ denoting the distribution of the noise. The noisy random variable $\mathcal{V}^{'}$ is then distributed as follows:

$$\begin{aligned} P[\mathcal{V}^{'} = 1] &= p_{sig} \times P[\mathcal{V} = 1] + (1 - p_{sig}) \times P[\mathcal{V}_n = 1] \\ P[\mathcal{V}^{'} = 0] &= 1 - P[\mathcal{V}^{'} = 1] \end{aligned} \tag{15}$$

Here p_{sig} represents the signal probability. Given the fault is injected at the loop iteration t, with the aim of affecting the r-th cipher or redundant round p_{sig} can be given as follows:

$$p_{sig} = \frac{\binom{t-1}{2r-2}\binom{R-t}{n-2r+1} + \binom{t-1}{2r-1}\binom{R-t}{n-2r}}{\binom{R}{n}} \tag{16}$$

The decision making procedure for fault pattern recovery now takes the following form. Consider two noisy distributions, $\mathcal{D}_0^{'}$ and $\mathcal{D}_1^{'}$, given as follows:

$$\begin{aligned} \mathcal{D}_0^{'} : P[\mathcal{V}^{'} = 1] &= p_{sig} \times P[\mathcal{V} = 1 | x = 0] + (1 - p_{sig}) \times P[\mathcal{V}_n = 1] \\ P[\mathcal{V}^{'} = 0] &= 1 - P[\mathcal{V}^{'} = 1] \end{aligned} \tag{17}$$

and

$$\begin{aligned} \mathcal{D}_1^{'} : P[\mathcal{V}^{'} = 1] &= p_{sig} \times P[\mathcal{V} = 1 | x = 1] + (1 - p_{sig}) \times P[\mathcal{V}_n = 1] \\ P[\mathcal{V}^{'} = 0] &= 1 - P[\mathcal{V}^{'} = 1] \end{aligned} \tag{18}$$

Here x denotes the leaking intermediate. The decision making process then can be stated as:

Decide the outcome(one component of the target fault pattern) to be 0 *(no fault propagation) if* $\mathcal{V}^{'} \sim \mathcal{D}_0^{'}$*, and to be* 1*, otherwise.*

In order to nullify the effect of the noise in decision making process, we need to ensure two things; firstly, fault injection must happen at a loop iteration where the probability of signal p_{sig} is the highest. This can be achieved easily using the expression for p_{sig} given in Eq. (16). The variation of signal probabilities for

a specif choice of r with different counts of dummy rounds is depicted in Fig. 6. It can be observed that p_{sig} achieves its highest value corresponding to a given r only at certain loop iterations.

The second factor which can nullify the noise effect is an accurate estimation of the distribution of $\mathcal{V}_n$. Unlike the p_{sig}, estimation of $\mathcal{V}_n$ was found to be tricky. One may observe that noise in this case comes from two points: 1) injection at a dummy round; 2) injection at arbitrary cipher or redundant rounds. Note that the propagation of the fault in our case typically depends upon the actual unshared value. For an injection at a dummy round this value dependent propagation effect becomes random (as the data inside the dummy rounds are random) and can be estimated properly. However, for the second case, the noise is somewhat correlated with the signal. This is attributed to that fact that the plaintext in our attacks are typically held fixed, which also fixes the unshared values processed in all cipher and redundant computation rounds. Although the desired round of injection is r, some of its neighbouring cipher and dummy rounds get hit by the fault with significantly high probability. In essence, the noise in our case is correlated with the signal, which makes the detection of signal significantly challenging.

One option to reduce the correlation of the noise with the signal component is to increase the number of dummy rounds. Fig. 7, presents the mean and variances for the expected value of the noise distribution (V_n) for different number of dummy round computations. The expected value of V_n is normally distributed, in general. The distributions were estimated during the profiling (template-building) phase by changing the plaintext values. It can be observed that the variance of noise is significantly high while the number of dummy rounds are low and it gradually improves with the increased number of dummy rounds[18]. This observation also indicates that may be a better estimation of the noise distribution would be possible if the number of dummy rounds are increased arbitrarily. However, having a huge number of dummy rounds is impractical as the overhead will be extremely high.

Let us now try to see if the signal components can be recovered for a reasonable count of dummy rounds. Without loss of generality, we set the number of dummy rounds to be 64 for this specific experiment. The expected value $\mu_{\mathcal{V}_n}$ of $\mathcal{V}_n$ (which is nothing but $P[\mathcal{V}_n = 1]$) is normally distributed. This makes the mean of $\mathcal{D}_0^{'}$ (denoted as $\mathcal{D}_{\mu_0}$) and $\mathcal{D}_1^{'}$ ($\mathcal{D}_{\mu_1}$) normally distributed as well. In order to make the abovementioned decision process work with high confidence, both the means should be accurately estimated and their distributions should overlap as less as possible.

With a proper mathematical model, we are now at the position to state our detection procedure for the fault patterns. Corresponding to each fault location we perform the fault injection campaign for several different mask values and

[18] One should note that even a noise standard deviation of 0.07 is high in this context, as the p_{sig} is in the range of 0.1-0.2 for any reasonable count of dummy rounds. In other words, the contribution of noise component is so high that even a small standard deviation value can distort the fault pattern detection mechanism.

gather sufficient number of observations for the noisy observable random variable $\mathcal{V}^{'}$. The mean of $\mathcal{V}^{'}$ is next estimated as $\mu_{\mathcal{V}'}$. In the next step, we simply estimate the probability of $\mu_{\mathcal{V}'}$ belonging to any of the two distributions $\mathcal{D}_{\mu_0}$ or $\mathcal{D}_{\mu_1}$. More precisely, we calculate the following:

$$P[\mathcal{D}_{\mu_0} \mid \mu_{\mathcal{V}'}] \text{ and } \text{ and } P[\mathcal{D}_{\mu_1} \mid \mu_{\mathcal{V}'}] \tag{19}$$

The outcome (one component of the target fault pattern) is assumed to take the value for which the probability is the highest.

Unfortunately, the abovementioned detection strategy is found to have some accuracy issues. To understand the reason we refer to the distributions of $\mathcal{D}_{\mu_0}$ and $\mathcal{D}_{\mu_1}$ depicted in Fig. 8. The highly overlapped patterns of these distributions are the sole cause behind the inaccuracies in the detection. To deal with these inaccuracies, we set a threshold window in the detection mechanism which gives an indication if the detection confidence is sufficiently high or not (Typically, some part of the shaded region in Fig. 8 is selected as threshold window). The threshold is set simply based on the observed value of $\mu_{\mathcal{V}'}$. If the value is within the low confidence region, the detection process raises a flag indicating the uncertainty in the detection. *Having this threshold at place, it is observed that in the case of PRESENT, two components of the fault pattern vector (which is of length 4) may remain undecided on average.* Given there are total 16 possible fault patterns for the fault location we chose, the template-matching will now return 4 suggestions on average for each intermediate value. As a result we would get total $4^{16} = 2^{32}$ suggestions for an intermediate state. Note that, one may further filter these suggestions by performing the same experiment for another set of fault locations and taking the intersection between the value suggestions corresponding to each nibble returned from these two experiments. In our case, we tried with the fault locations at the G function (presented in Appendix. C) and found that taking the intersection leaves us with $2-3$ suggestions for each intermediate nibble, with three suggestions occurring rarely. The size of the suggestion set now becomes roughly 2^{20}.[19]

In the known plaintext scenario, where the target intermediate round is the first round, the abovementioned complexity figure is still reasonable for recovering a round key. However, for middle round attacks one need to estimate two consecutive intermediate states to recover a complete round key. In the present context, the number of key suggestions for a middle round key recovery would become 2^{40} (and 2^{56} for the entire master key), which, although, is less then brute force complexity, but still impractical. It is worth mentioning that the results we consider in this case are specific for the attack on PRESENT (however, the attack procedure is generic). There is always a chance that changing the TI equations or the fault locations result in an attack with better accuracy and complexity figures. In nutshell, although infection countermeasures are somewhat promising as protections against the proposed attacks, they cannot be considered as an ultimate solution against FTA.

[19] For the entire master key it becomes 2^{36}.