Protecting against Statistical Ineffective Fault Attacks

Joan Daemen[1], Christoph Dobraunig[1], Maria Eichlseder[2], Hannes Gross[3], Florian Mendel[4], and Robert Primas[2] ⋆

[1] Radboud University, Netherlands ({joan,cdobrauig}@cs.ru.nl)
[2] Graz University of Technology, Austria (first.last@iaik.tugraz.at)
[3] SGS Digital Trust Services GmbH, Austria
[4] Infineon Technologies AG, Germany (florian.mendel@gmail.com)

Abstract. At ASIACRYPT 2018 it was shown that Statistical Ineffective Fault Attacks (SIFA) pose a threat for many practical implementations of symmetric cryptography. In particular, countermeasures against both power analysis and fault attacks typically do not prevent straightforward SIFA attacks that require very limited knowledge about the concrete attacked implementation. Consequently, the exploration of countermeasures against SIFA that do not rely on protocols or physical protection mechanisms is of particular interest. In this paper, we explore different countermeasure strategies against SIFA. First, we thoroughly analyze the conditions for an attack to be successful. We then show that by building the implementation from invertible building blocks rather than binary gates we can create circuits where a single fault in the computation does not cancel out. This property, when combined with a typical redundancy-based countermeasure, then results in a single-fault SIFA-secure implementation. This approach can be implemented efficiently and we show how it can be applied to 3-bit, 4-bit, and 5-bit S-boxes. Additionally, we also present an alternative countermeasure strategy based on fine-grained detection. Although this approach may lead to a higher implementation cost, it can be used to protect arbitrary circuits and can be generalized to cover multi-fault SIFA.

Keywords: Fault countermeasures · Implementation security · Fault attack · Masking · SFA · SIFA

1 Introduction

Fault attacks [10, 13] and passive side-channel attacks, like power [37] or EM analysis [40], are very powerful attacks against implementations of cryptographic algorithms. Therefore, devices like smart cards that are potentially physically accessible by an attacker typically implement corresponding countermeasures.

The common approach to counteract such attacks on an algorithmic level is to use a combination of masking (against power analysis) and some kind of

⋆ The list of authors is in alphabetical order (https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf) redundancy (against fault attacks). Masking is a secret-sharing technique that splits intermediate values of cryptographic computations into $d+1$ random shares such that the observation of up to d shares does not reveal any information about their corresponding native value [3,4,16,30–32,35,41]. Redundant computation, on the other hand, is used to detect malicious or environmental influences that could lead to exploitable erroneous cryptographic computations, and to either prevent its release or to prevent that it is exploitable by an attacker (infection) [2, 48]. Examples of recent works that propose combinations of such countermeasure techniques include ParTI [43], Private Circuits II [15,34], and M&M [23]. A quite different approach was chosen with CAPA, an actively secure MPC protocol that was adapted to cryptographic computations to provide strong protection from implementation attacks but at a cost that makes its usage of limited interest for practical applications [42].

Up until recently, implementations combining masking with fault countermeasures were typically assumed to offer protection against both power analysis and fault attacks. However, it was shown in [24] that Statistical Ineffective Fault Attacks (SIFA) [25], a combination of the principles of Ineffective Fault Attacks (IFA) [14] and Statistical Fault Analysis (SFA) [28], are applicable in typical combined countermeasure scenarios. SIFA circumvents typical redundancy/infection countermeasures since it only relies on ineffective faults, i.e., faulted computations where the attacker only observes the output if it was calculated correctly. The application to masked implementations, which was originally believed to require multiple faults in one computation, was recently demonstrated in [24].

Our contribution. While the previous work on SIFA already discussed some countermeasures that increase the robustness of implementations against the attack, they either result in significant implementation overheads, higher protocol complexity, or rely on the existence of physical protection mechanisms. Therefore, efficient algorithmic countermeasures against SIFA are of great interest and were left to future work so far.

The contribution of this paper is two-fold: First, we thoroughly analyze the general root causes that lead to successful SIFA attacks. For this purpose, we introduce a generic computation model and define a threat scenario. We then discuss why ineffective faults can be used to recover secrets from masked cryptographic implementations, even if dedicated fault countermeasures are in place.

Second, we present two different approaches that can be used to mitigate these attacks. We start by looking at the Toffoli-gate, the simplest invertible non-linear function. We then show that when the Toffoli-gate is used as the only non-linear building block in the masked implementation of a cipher, single-fault SIFA attacks can always be detected by redundancy.

Following up, we show that 3-bit, 4-bit, and 5-bit S-boxes can be built (and thus protected) using Toffoli-gates as their only non-linear component. We also give a more general criterion for SIFA-protected circuits and present a more generic countermeasure based on fine-grained detection. This countermeasure can be used to protect arbitrary circuits and can be generalized to cover multi-fault SIFA, albeit at a higher implementation cost.

To introduce the two different approaches we use a simple and generic computation model close to single core software implementations. We discuss how the model can be carried over from our computational model to actual software implementations or realized in hardware.

2 On the Effect of Faults on Masked Computations

This section serves as a preliminary section, where we define the computational model that we use and the faults that we consider on it. Then we discuss the principle of masking and show why masking does not necessarily prevent statistical ineffective fault attacks (SIFA).

2.1 Computational and Fault Model

We use a simplified generic computation model close to single core software implementations. Our model is based on Boolean functions $c \leftarrow f(a, b)$ with two input variables a and b and writing to an output variable c. We refer to the computation of such a single function as operation or instruction. Furthermore, we assume that the operations are sequentially executed, e.g., f_1 is computed before f_2:

$$\begin{aligned} c &\leftarrow f_1(a, b) \\ d &\leftarrow f_2(c, d) \end{aligned}$$

We refer to a given sequence of operations as circuit, computation, or by referring to their purpose, e.g., masked AND. In addition, we also specify the input and outputs variables of the circuit, e.g., we describe an AND followed by an XOR operation as:

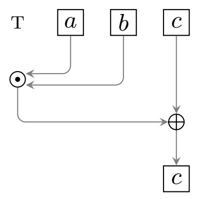
$$\begin{aligned} &\text{Input: } \{a, b, c\} \\ &\textsc{t} \leftarrow a \odot b \\ &c \leftarrow c \oplus \textsc{t} \\ &\text{Output: } \{c\} \end{aligned} \tag{1}$$

Unless stated otherwise, we assume an attacker that can induce a single fault per execution of a given circuit. A single fault is defined as a change of the content of a variable to a value chosen by the attacker, or a change of the Boolean function f to any Boolean function of the attacker's choice in a single operation. The attacker can choose the location of the fault. Furthermore, it is possible for the attacker to make the choice of the value of a variable dependent on the value it had before the fault, which allows faults like bitflips. However, an attacker cannot copy values, e.g., move the value of a variable b to a, except by manipulating the Boolean functions. Furthermore, an attacker is neither allowed to change which variables are involved in a single operation, nor manipulate the order in which operations are executed. How this model maps to real-world implementations is explained in Section 3.3 and Section 3.4.

For the following discussion, we use a graph representation in addition to the algorithmic description. For example, Figure 1a lists the algorithmic description and Figure 1b illustrates a graph-based representation for an And followed by an Xor in Equation 1.

```
Input: $\{a, b, c\}$
$\mathrm{T} \leftarrow a \odot b$
$c \leftarrow c \oplus \mathrm{T}$
Output: $\{c\}$
```

(a) Algorithmic representation



(b) Graph representation

Fig. 1: Representation of And followed by an Xor shown in Equation 1.

2.2 SIFA against Unmasked Implementations

Consider a cipher implementation that executes the cipher n times and only releases the output if all executions of the cipher match to counteract fault attacks. Assume an attacker who targets one out of the n redundant encryptions is capable of repeatedly inducing one fault in one of its intermediate operations, and observes the ciphertext unless it is suppressed by the countermeasure. The timing or location of the fault should be chosen such that the affected state bits can be computed from the ciphertext if a hypothesis is made on a small number of round key bits. This is typically the case in the penultimate round of a cipher.

Over the course of multiple such faulted encryptions, the attacker will eventually observe correct ciphertexts where the induced fault in the corresponding computation was ineffective. In these cases, the specific values used in the faulted operation rendered the induced fault ineffective, which lead to an overall correct computation. This filtered set of correct ciphertexts will, when partially decrypted using the correct partial key guess, typically show this non-uniform (biased) distribution of intermediate values in the state bits.

We can use this fact for key recovery as follows: For each key guess, we measure the distance of the distribution of the affected state bits to the uniform distribution, e.g., by using the χ^2-statistic (CHI) or the closely related Squared Euclidean Imbalance (SEI). For a sufficient number of evaluated correct ciphertexts, the key guess corresponding to the highest CHI or SEI statistic is most likely correct.

For more concrete examples, we refer to previous literature that explains attack scenarios for block ciphers like AES [25] or authenticated encryption schemes like Keyak and Ketje [27].

2.3 The Principle of Masking

In this section, we discuss the basic idea behind masking as a countermeasure against power analysis attacks. We focus on Boolean masking schemes. The resistance against power analysis gained through masking is achieved by making the individual variables in computations independent from the security-critical data that is processed. For this purpose, variables s are split into a number of so-called shares denoted by $s_0, s_1, \ldots, s_d$. To avoid ambiguity, we will denote these variables by the term *native*, so s is a native variable and $s_0, s_1, \ldots, s_d$ are its shares. Ideally, d shares would be sampled from a uniformly random distribution and we compute the last share as $s_d = s \oplus \bigoplus_{i=0}^{d-1} s_i$. This implies that in the case of a $d+1$-share masking scheme, an attacker learning d shares of a variable s would still not have any information on s.

A shared implementation of a cipher operates on the shares and ideally preserves this property: learning d shares does not give information on native variables. Hence, an attacker who can observe any d variables in a shared circuit or any d of its operations (or all shares used in the d operations) should not get any information about the native values. For linear functions, this can be ensured easily, since these can be computed without mixing the shares. For example, the computation $c \leftarrow a \oplus b$ using 2 shares is simply performed as:

```
Input: $\{a_0, a_1, b_0, b_1\}$
$c_0 \leftarrow a_0 \oplus b_0$
$c_1 \leftarrow a_1 \oplus b_1$
Output: $\{c_0, c_1\}$
```

Since many symmetric primitives allow for compact representations over $\mathbb{F}_2$, it is sufficient to find masked implementations for the addition and multiplication over $\mathbb{F}_2$ (Boolean Xor and And) in order to provide masked implementations of such a symmetric primitive. Since masking Xor is easy in Boolean masking schemes, the focus of many papers dealing with masking is to find efficient masked implementations for Boolean And [3,4,16,30–32,35,41]. If we just focus on the sharing of an And, $c \leftarrow a \odot b$, using 2 shares, such a sharing requires the addition of a resharing variable R. This is needed to ensure that the values of c_0 and c_1 are each independent of the native value of c. The resharing variable R may be derived from a dedidated random number generator or from another unrelated calculation, e.g., as shown in Changing of the Guards [19]. A possible implementation of masked AND is then:

$$\begin{aligned}
&\text{Input: } \{a_0, a_1, b_0, b_1, \mathrm{R}\}\\
&\mathrm{T}_0 \leftarrow a_0 \odot b_1\\
&\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{R}\\
&\mathrm{T}_1 \leftarrow a_0 \odot b_0\\
&c_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1\\
&\mathrm{T}_2 \leftarrow a_1 \odot b_0\\
&\mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus \mathrm{R}\\
&\mathrm{T}_3 \leftarrow a_1 \odot b_1\\
&c_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3\\
&\text{Output: } \{c_0, c_1\}
\end{aligned} \tag{2}$$

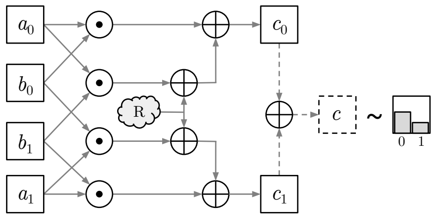
2.4 SIFA on Masked Additions and Multiplications

SIFA works on (masked) implementations that use fault countermeasures, typically some kind of redundancy, by exploiting faults that are ineffective. SIFA relies on the property that the condition if a fault is ineffective can depend on the native values of certain intermediate variables of a cryptographic implementation. This dependency is then exploited by inducing a fault over the course of multiple computations and just observing the computations where the fault does not show an effect on the output. Typically, the knowledge that some intermediate values are not distributed as expected can be exploited in key recovery attacks. We will show with an example that shared implementations may also be vulnerable to SIFA.

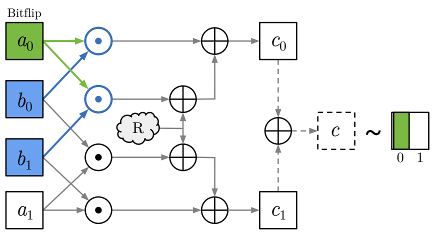
Let us again consider a Boolean sharing of s using $d+1$ shares as introduced in Section 2.3. If an attacker now only faults up to d variables, each containing one share, the condition if this fault is ineffective is completely independent of the value of s and hence, the ineffectiveness of the fault does not reveal any information about s.

Similar as for side-channel resistance, our goal is to build circuits that preserve this property. Thus, we want to build circuits where an attacker can fault d shares or d operations and still, the condition whether a fault is ineffective or not does not depend on native values.

In the case of linear functions, this is again given by the fact that all computations can be carried out independently per share. However, for non-linear functions, we have the following effect. Consider for instance a single masked AND-gate with 2 shares (cf. Figure 2). We can show that the condition whether a single fault is ineffective or not can depend on a native value. The reason is that any input share is an input variable to two of the AND-gates which are fed by both shares of the other variable. Thus, by faulting a_0, for instance, the fault will be always ineffective and does not propagate if either b_0 and b_1 are both 0 or both are 1. Hence, the condition if the fault is ineffective depends on the native value of b, namely, it is always ineffective if $b = 0$.



(a) Masked AND without fault inductions.



(b) Masked AND with a bitflip in a_0 if only correct computations of c are observed.

Fig. 2: If only correct computations are observed by the attacker, a bitflip in a_0 is only ineffective if $b_0 = b_1$, in other words, b (and thus also c) are always zero.

Assume that we have a masking scheme using two shares which is secured against power analysis attacks observing one share or one operation. One can then achieve resistance against single-fault SIFA by building an appropriate design so that throughout the entire computation, at most one operation or variable is affected by a single fault without detection. This allows that the condition whether a fault is ineffective or not depends only on an incomplete set of shares of any native variable. In the next section, we present one way to achieve this.

3 Using Building Blocks Protected Against SIFA

In this section, we investigate how we can implement ciphers so that they are protected against single-fault SIFA. We do this by moving away from seeing a circuit as sequence of additions and multiplications and thus using masked XORs and ANDs as building blocks. Instead, we take a look at the principles introduced by reversible computing [5,38,47], with the Toffoli gate [47] and related constructions as essential building blocks. We show that the masked Toffoli gate has the beneficial property that the condition whether a fault is ineffective never depends on native values. We then show that a significant fraction of 3-, 4- and 5-bit S-boxes can be built from the Toffoli gate and related constructions. Finally, we discuss how a protected circuit in our computation model can be mapped to an actual hardware or software implementation.

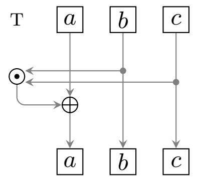
3.1 The Toffoli Gate

The Toffoli gate [47] is a non-linear 3-bit permutation. We denote it by $p_T(a, b, c)$ and illustrate it in Figure 3.

In the implementation of the Toffoli gate in Figure 3, the condition whether or not a fault is ineffective might depend on the native value. This is easy to see as, e.g., setting any of the inputs $a, b,$ or c to zero only leads to a correct computation if the faulted input was zero prior to fault induction. Alternatively,

```
Input: $\{a, b, c\}$
$\mathrm{T} \leftarrow b \odot c$
$a \leftarrow a \oplus \mathrm{T}$
Output: $\{a, b, c\}$
```

(a) Algorithmic representation



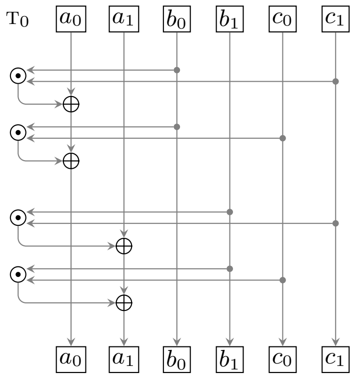
(b) Graph representation

Fig. 3: Toffoli gate (p_T).

a bitflip in one of the inputs of the AND-gate is only ineffective if the other input was zero. Hence, to protect the Toffoli gate against SIFA using single faults, we have to use at least an implementation using 2 shares as done in Figure 4, to which we refer as $p_T(a_0, a_1, b_0, b_1, c_0, c_1)$.

```
Input: $\{a_0, a_1, b_0, b_1, c_0, c_1\}$

$\mathrm{T}_0 \leftarrow b_0 \odot c_1$
$a_0 \leftarrow a_0 \oplus \mathrm{T}_0$

$\mathrm{T}_0 \leftarrow b_0 \odot c_0$
$a_0 \leftarrow a_0 \oplus \mathrm{T}_0$

$\mathrm{T}_0 \leftarrow b_1 \odot c_1$
$a_1 \leftarrow a_1 \oplus \mathrm{T}_0$

$\mathrm{T}_0 \leftarrow b_1 \odot c_0$
$a_1 \leftarrow a_1 \oplus \mathrm{T}_0$

Output: $\{a_0, a_1, b_0, b_1, c_0, c_1\}$
```

(a) Algorithmic representation



(b) Graph representation

Fig. 4: Masked Toffoli gate (p_T) using 2 shares.

As can be seen in Figure 4b, in the shared Toffoli gate only the shares a_0 and a_1 get updated in-place, by just using results involving b_0, b_1, c_0, and c_1. Furthermore, values written to the temporary register T_0 are just used once in the circuit. Hence, from Figure 4b, we can see that single faults that can influence more than one AND computation must directly change either b_0, b_1, c_0, or c_1. Since those values are present as output, such a single fault is visible at the output and is also visible in the corresponding native values of b and c. Furthermore, also single faults that impact more than one XOR have to show an effect on either a_0 or a_1. Such a single fault then will be visible at the native value of a at the output, since all operations on a_0 and a_1 are performed in-place and they do not influence each other.

In short, in the shared Toffoli gate of Figure 4, the ineffectiveness of any single fault depends only on a single share.

To turn this argument around, single fault inductions where the effect of the fault depends on all shares of a value change always at least the value of one share. Furthermore, then at least one native value is also changed. Hence, all relevant single faults can be detected by using a redundant computation of the shared Toffoli gate and only checking the native values of a, b, and c at the output of the gate. The shared Toffoli gate, together with the related non-linear permutation p_χ (Figure 5), will serve as a building block to protect ciphers against single-fault SIFA in the next section. For the sake of completeness, we note that the same properties can be achieved in a similar form for a three-share threshold implementation as shown in Appendix A.

3.2 From Protected Components to Protected Ciphers

In this section, we discuss how to protect ciphers by using protected building blocks. We focus our considerations on constructions that are built from the repeated application of bijective linear and bijective non-linear layers. Typically, those non-linear layers are built from the parallel application of smaller bijective non-linear functions called S-boxes.

The Propagation of a Fault. Let us assume we have such a construction, where each used bijective building block has the following characteristics of the shared Toffoli gate as discussed in Section 3.1. In particular, we require that an ineffective fault in such a building block never depends on any native variables. Similarly, we require effects of faults that do depend on native variables always show in at least one native value at the output of the building block. Then, due to the bijective nature of the construction, such an effect is also visible at the output of the entire layered construction. Let us now consider a redundancy-based countermeasure that only compares native values at the output of the entire primitive combined with the way of masking we present in this section. This countermeasure does not only provide protection against single fault attacks that exploit the effect of a fault, but now additionally provides protection against single-fault SIFA.

As discussed in Section 2.4, a shared linear function can be realized by applying that function to the shares separately. Hence, the behavior of a single fault on a shared linear layer never depends on native values. Furthermore, since they are bijections, a change in their inputs will always lead to a change in their outputs. Thus, we can focus on how to build bijective S-boxes if we want to protect an entire cipher against SIFA attacks. One option to do this is to construct S-boxes by the iterative application of the masked Toffoli gate of Section 3.1 and related permutations.

3-bit S-boxes. Recently, 3-bit S-boxes have become more prominent with their usage in PRINTcipher [36], LowMC [1], or Xoodoo [20]. As a representative of these S-boxes, we focus on the protection of the 3-bit χ-layer [18, 20]. This mapping χ operates on circular arrays of bits and it complements for all bits that have the pattern 01 the bits at their right. χ is bijective if and only if the length of the circular array is odd. The χ mapping in the round function of ciphers typically operates on a large set of short odd-length sub-arrays of the state in parallel. We also refer to n-bit χ as χ_n.

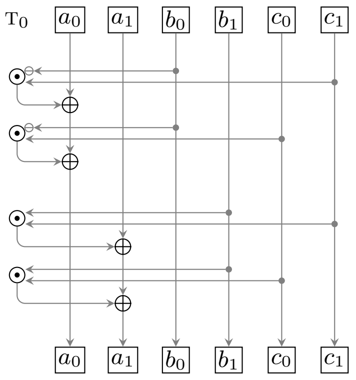
To build χ_3, we use a reversible gate similar to the Toffoli gate of Section 3.1, but using AndN instead of And (one input of the And is inverted), to which we refer as $p_\chi(a, b, c)$:

```
Input: ${a, b, c}$
$\mathrm{T} \leftarrow \overline{b} \odot c$
$a \leftarrow a \oplus \mathrm{T}$
Output: ${a, b, c}$
```

The shared version $p_\chi(a_0, a_1, b_0, b_1, c_0, c_1)$ of this gate has the same properties as discussed for the Toffoli gate in Section 3.1. The resulting shared circuit of p_χ is shown in Figure 5.

```
Input: ${a_0, a_1, b_0, b_1, c_0, c_1}$
$\mathrm{T}_0 \leftarrow \overline{b_0} \odot c_1$
$a_0 \leftarrow a_0 \oplus \mathrm{T}_0$
$\mathrm{T}_0 \leftarrow \overline{b_0} \odot c_0$
$a_0 \leftarrow a_0 \oplus \mathrm{T}_0$
$\mathrm{T}_0 \leftarrow b_1 \odot c_1$
$a_1 \leftarrow a_1 \oplus \mathrm{T}_0$
$\mathrm{T}_0 \leftarrow b_1 \odot c_0$
$a_1 \leftarrow a_1 \oplus \mathrm{T}_0$
Output: ${a_0, a_1, b_0, b_1, c_0, c_1}$
```

(a) Algorithmic representation



(b) Graph representation

Fig. 5: Masked p_χ using 2 shares.

Daemen et al. [20] pointed out that it is possible to compute χ_3 in-place in its registers. In the unshared case (and equivalently for the shared case), it is possible to compute χ_3 the following way:

```
Input: ${a, b, c}$
$p_\chi(a, b, c)$
$p_\chi(b, c, a)$
$p_\chi(c, a, b)$
Output: ${a, b, c}$
```

4-bit S-boxes. The construction and design of 4-bit S-boxes has been intensively studied in literature. De Cannière [22] sorts all 4-bit bijective S-boxes in 302 equivalence classes, where 1 class contains all affine functions, 6 classes contain quadratic functions, and 295 classes represent the cubic functions [11]. As shown by Bilgin et al. [11], 144 cubic classes can be constructed by iterating the S-boxes of the quadratic classes separated by affine layers up to 3 times. Many prominent S-boxes, e.g., the S-boxes used in Noekeon [21] and Present [12], but also several of the 16 S-boxes observed to be "optimal" by Leander and Poschmann [39] are covered. We focus on the 6 classes of quadratic functions. The variables a, b, c, and d indicate the input and output bits of the S-box, where a is the most significant bit. The operations needed to compute the 6 quadratic classes are summarized in Table 1.

Table 1: The 6 classes of quadratic 4-bit S-boxes [11] using reversible building blocks p_T and p_χ.

Q_4^4 `0123456789ABDCFE`	Q_{12}^4 `0123456789CDEFAB`	Q_{293}^4 `0123457689CDEFBA`
Input: $\{a,b,c,d\}$ $p_T(d,a,b)$ Output: $\{a,b,c,d\}$	Input: $\{a,b,c,d\}$ $p_T(b,a,c)$ $p_T(c,a,b)$ Output: $\{a,b,c,d\}$	Input: $\{a,b,c,d\}$ $p_T(d,b,c)$ $p_T(b,a,c)$ $p_T(c,a,b)$ Output: $\{a,b,c,d\}$

Q_{294}^4 `0123456789BAEFDC`	Q_{299}^4 `012345678ACEB9FD`	Q_{300}^4 `0123458967CDEFAB`
Input: $\{a,b,c,d\}$ $p_T(c,a,b)$ $p_T(d,a,b)$ $p_T(d,a,c)$ Output: $\{a,b,c,d\}$	Input: $\{a,b,c,d\}$ $p_T(b,a,c)$ $p_T(c,a,b)$ $p_T(b,a,c)$ $p_T(c,a,d)$ $p_T(d,a,c)$ Output: $\{a,b,c,d\}$	Input: $\{a,b,c,d\}$ $b \leftarrow b \oplus a$ $c \leftarrow c \oplus a$ $p_T(a,b,c)$ $p_T(b,a,c)$ $p_\chi(c,b,a)$ Output: $\{a,b,c,d\}$

With the help of Table 1, Figure 4, and Figure 5, it is possible to create masked implementations for 144 out of the 295 cubic classes of affine equivalent S-boxes [11], where the condition whether a single fault is ineffective never depends on a native value. For S-boxes which are not in these classes, we refer to results regarding the implementation of 4-bit permutations using reversible components. For instance, Golubitsky and Maslov [29] give optimal implementations (with respect to a certain set of reversible gates) for all 4-bit permutations using at most 15 reversible gates. However, please note that the set of reversible gates used may differ from the building blocks p_T and p_χ used in this section.

5-bit S-boxes. As pointed out by Shende et. al [45], every permutation (S-box) with an odd number of inputs can be implemented using reversible gates by using at most one additional variable. However, in this work we only discuss the 5-bit S-box χ_5, which has several prominent uses. For instance, it is used in the family of Keccak-f and Keccak-p permutations that are used, amongst others, in Ketje [8], Keyak [9], Kravatte [6], or most prominently in SHA-3 (Keccak [7]), but χ_5 is also the core of Ascon's S-box [26]. We provide a masked implementation of χ_5 based on the following way of implementing χ_5 [26], where a, b, c, d, and e are the input bits and r is a temporary variable:

$$\begin{aligned}
&\text{Input: } \{a, b, c, d, e\}\\
&r \leftarrow a \odot \overline{e}\\
&p_\chi(a, b, c)\\
&p_\chi(c, d, e)\\
&p_\chi(e, a, b)\\
&p_\chi(b, c, d)\\
&d \leftarrow d \oplus r\\
&\text{Output: } \{a, b, c, d, e\}
\end{aligned}$$

To provide an implementation of χ_5 that withstands single-fault SIFA, we again rely on $p_\chi(a_0, a_1, b_0, b_1, c_0, c_1)$ as a building block. We introduce additional input variables r_0 and r_1, which have to be initialized with random values, so that $r_0 \oplus r_1 = 0$. This allows us to argue the security of the following scheme:

$$\begin{aligned}
&\text{Input: } \{a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1, r_0, r_1\}\\
&p_\chi(r_0, r_1, e_0, e_1, a_0, a_1)\\
&p_\chi(a_0, a_1, b_0, b_1, c_0, c_1)\\
&p_\chi(c_0, c_1, d_0, d_1, e_0, e_1)\\
&p_\chi(e_0, e_1, a_0, a_1, b_0, b_1)\\
&p_\chi(b_0, b_1, c_0, c_1, d_0, d_1)\\
&d_0 \leftarrow d_0 \oplus r_0\\
&d_1 \leftarrow d_1 \oplus r_1\\
&\text{Output: } \{a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1\}
\end{aligned} \tag{3}$$

We end up with a construction which is the repeated application of permutation p_χ on 12 bits of the state a_0 to r_1. Due to this iterative construction, a single fault that has an effect on any native output variable of one p_χ will have an effect on the native output variables of the whole circuit. However, we truncate r_0 and r_1 at the output at the end of our circuit. Hence, we have to show that this truncation never leads to an effect of a fault disappearing.

As can be seen in Equation 3, d_0 and d_1 are only written in the operations $d_0 \leftarrow d_0 \oplus r_0$ and $d_1 \leftarrow d_1 \oplus r_1$. Furthermore, the calculation of r_0 and r_1 is independent of d_0 or d_1. As a consequence, a single fault that happens before the execution of $d_0 \leftarrow d_0 \oplus r_0$ and $d_1 \leftarrow d_1 \oplus r_1$ can never have an effect on the shares of d and r at the same time. Hence, the operations $d_0 \leftarrow d_0 \oplus r_0$ and $d_1 \leftarrow d_1 \oplus r_1$ never cancel the effect of a single fault, and effects of faults on the native value of r carry over to d.

In a similar spirit as Sugawara for AES [46], it is possible to use one share r_0 of the output of one S-box layer as input to the next layer of S-boxes. Hence, it is possible to implement ciphers which use the sharing shown in Equation 4 without the need for additional randomness, except the one needed for the initial sharing and for the first S-box layer. We have verified exhaustively that indeed, Equation 4 is a permutation on the bits $a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1$, and r_0:

$$\begin{aligned}
&\text{Input: } \{a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1, r_0\} \\
&r_1 \leftarrow r_0 \\
&p_\chi(r_0, r_1, e_0, e_1, a_0, a_1) \\
&p_\chi(a_0, a_1, b_0, b_1, c_0, c_1) \\
&p_\chi(c_0, c_1, d_0, d_1, e_0, e_1) \\
&p_\chi(e_0, e_1, a_0, a_1, b_0, b_1) \\
&p_\chi(b_0, b_1, c_0, c_1, d_0, d_1) \\
&d_0 \leftarrow d_0 \oplus r_0 \\
&d_1 \leftarrow d_1 \oplus r_1 \\
&\text{Output: } \{a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1, r_0\}
\end{aligned} \tag{4}$$

3.3 From an Abstract Model to Software Implementations

In Section 2.1, we have introduced a rather simplified computational model and explicitly restricted which faults we want to consider. Here, we want to discuss what has to be considered in real software implementations and which faults that occur in software implementations are covered by our model.

Coverage of Faults. As mentioned in Section 2.1, we consider faults that manipulate either variables directly, or change the used functions. This covers faults in software that directly manipulate values of variables stored in registers of a CPU [44], change a variable in memory before it is loaded, or even target the load of a variable from memory [17]. Furthermore, it also covers cases where a fault, like a clock glitch, changes the outcome of a computation. However, what is notably only partially covered is the case of an instruction skip, meaning that an operation is not performed and the register values are kept untouched. This is only covered if an operation in the form of $a \leftarrow a \oplus b$ is used, so that a fault could replace the $\oplus$ with a function that just uses a and neglects b. If we focus on the main building blocks p_T and p_χ used in this section, the only operations that do not follow this behavior are the AND computations to the external register T_0. Potential negative effects of the clock glitch can be mitigated by always initializing T_0 to 0 or a random value before use, so that a clock glitch just corresponds to a fault of a single output share.

What is not covered by our considerations are faults that change the execution flow of a program to a greater extent than skipping the single instructions, like manipulating the program counter. Furthermore, we do not consider the use of loops and conditional statements apart from their usage in detecting faults. What is also not considered are faults that change the operands used in operations. All these faults have in common that they may totally change the program that is executed to a point where the key is just put out in plain. Such faults have to be prevented by other means.

So far, our considerations have only explicitly covered single faults in single S-boxes that happen just once per execution of a cipher. However, modern ciphers in software implementations are usually implemented in a bit-sliced manner, meaning that usually, in a system using x-bit registers, a single computation, and thus, a single fault like a clock glitch leads to a single fault in up to x S-boxes. In ciphers that consist of layers applying many small bijective S-boxes in parallel to the state followed by bijective affine transformations of this state, putting a single fault in each of the parallel S-boxes in a single layer causes no issues with respect to our propagation-based countermeasures, since it is still ensured that single faults in all S-boxes cannot cancel each other.

Further Considerations Regarding Software. As discussed previously, our simple model gracefully extends to modern bitsliced implementations. However, one notable case that is not considered in our computational model are Load and Store instructions from memory to registers. In the simplest case, there are enough registers to store all necessary variables so that during a cryptographic computation, no Loads and Stores are needed. However, if this is not the case and a variable has to be reloaded, this might cause problems. For instance, let us consider the circuit shown in Figure 5b. Here, b_0 is just read and never written. So if we do not consider fault protection, it can be assumed that the register b_0 can just be overwritten, since the value can be reloaded from memory anyways. If we consider our fault protection mechanisms, this means that a faulted value in register b_0 might vanish, which in turn would allow SIFA again. To prevent this in general, we have to assume that values are changed and write them back to memory if their use is later required.

Furthermore, we have also abstracted the proper initialization of the temporary registers T_i before usage. The problem with uninitialized T_i is that shares can be combined, which leads to exploitable leakage, or, in the case of a clock glitch, to an unmasked use of a variable. Typically, the problem with uninitialized T_i can be easily solved by always writing a random value to them before the result of a computation is stored.

3.4 From an Abstract Model to Hardware Implementations

We note that most of our contributions regarding the protection against SIFA are also valid in hardware, but the efficient protection in hardware may require some additional considerations. One main difference between hardware and software

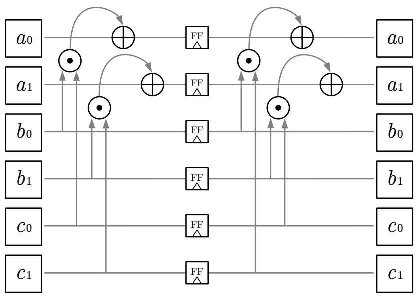


Fig. 6: Masked and single-fault SIFA-protected Toffoli gate in hardware.

implementations is that for hardware implementations one needs to take the effects of glitches into account. Glitches are the result of the behavior of the physical layout, and are thus purely parasitic, but unfortunately unavoidable. Since signals do not propagate evenly through the circuit (due to parasitic effects like differences in the capacitance of wires, different wire lengths, manufacturing imperfections, et cetera.) the output of gates could change (glitch) several times before reaching a stable logic state.

The behavior of the circuit cannot entirely be controlled by just using combinatorial logic gates but often require usage of so-called registers to gain more control over the signal timing. Registers stabilize a signal before entering the next logic gates through a separation in different clock cycles. The cost for the gained control over the signals is thus not only the increased gate count, but also the evaluation of the circuit requires more clock cycles and thus, the latency increases. Luckily when masking is used for the protection against side-channel analysis anyway, registers are already required at several places to ensure resistance to glitches. For example, TI implementations use registers after each uniformly shared function and the DOM scheme uses a register stage in each shared nonlinear gate to hinder security-critical glitches from propagating into the next shared function which could violate the security requirements.

Figure 6 shows a correctly masked Toffoli gate in hardware which already includes the required registers (FF) for a glitch resistant first-order side-channel protection. Furthermore, this variant also resists single-fault SIFA attacks because a secured sequence in which the multiplication terms are evaluated and added is provided. The overhead for the protection against SIFA over the masked variant is in this case negligible. The upper two registers are required to hinder the propagation of glitches that could violate the side-channel resistance of the implementation, and the lower four registers are usually placed for pipelining purposes. The lower four registers are the relevant ones for protection against SIFA. Again, no input variable must be used twice in non-linear operations inside the same clock cycle with different shares of the same masked variable. This would be the case when switching the order of the multiplication of b_1 and c_1 with b_1 and c_0, for instance, because a single fault of the input c_0 would affect both multiplications with the two shares of b.

4 Protecting Arbitrary Circuits

The costs of building circuits just relying on invertible gates like the Toffoli gate compared to other ways of constructing masked implementations varies depending on the S-box used. Depending on the target metric, e.g., when aiming for masked implementations with a very low latency, other approaches may be more suitable than the one introduced in Section 3. In this section, we explore how a general masked circuit can be protected against SIFA. We propose a criterion for SIFA-resistance of computation graphs and discuss how to satisfy it by either adding fault-detection checks of intermediate results or restructuring the graph.

We first refine the computation and fault model in Section 4.1 in order to introduce the general criterion for single-fault SIFA-resistance in Section 4.2. In Section 4.3, we show how to satisfy this criterion by extending a general masked implementation with local error detection checks. Then, in Section 4.4, we identify necessary steps and conditions such that global checks are sufficient. Finally, we discuss how to extend this approach to higher-order attacks, where the adversary applies multiple faults in each execution, in Section 4.5.

4.1 Extensions to the Computational and Fault Model

We first extend the computation and fault model of Section 2 and Section 3 with a few definitions.

Computation graph. We describe the computation as a directed acyclic graph (DAG) and refer to its nodes as gates $g \in \mathcal{G}$ and its edges as wires $w \in \mathcal{W}$. A gate g with n input wires $\textsc{in}(g) = \{x_1, \dots, x_n\}$ and one output wire $\textsc{out}(g) = \{y\}$ represents a Boolean function $g : \mathbb{F}_2^n \to \mathbb{F}_2$, $(x_1, \dots, x_n) \mapsto y$. In the simplest case, these are single Boolean gates, such as addition $y = x_1 \oplus x_2$ and multiplication $y = x_1 \odot x_2$ over $\mathbb{F}_2$, but they may also correspond to small circuits with several inputs. Additionally, there are some special nodes: Input and output registers, such as the shares of the input values, re-sharing inputs, and the shares of the output values, as well as branching nodes. We use a branching node with one input wire w and two output wires $(w', w'') = \top(w)$ whenever a variable w is used as input to multiple operations. We refer to w, w', w'' as one variable, but three different wires. We distinguish linear gates, whose Boolean function is affine linear over $\mathbb{F}_2$ (e.g., Xor $\oplus$, Not $\ominus$), and nonlinear gates (e.g., And $\odot$).

We remark again that this is a simplified model and not necessarily equivalent to how one would implement the logic in hardware. We refer to Section 3.3 and Section 3.4 for a discussion of how this model maps to software and hardware implementations.

Fault model. We still consider a powerful attacker that may replace any gate $y = g(x_1, \dots, x_n)$ (or wire w) by an arbitrary faulted gate $y^* = g^*(x_1, \dots, x_n)$ (or faulted wire $w^* = w^*(w)$). However, the attacker is only allowed to inject a single fault during an execution of the cryptographic primitive. We denote the

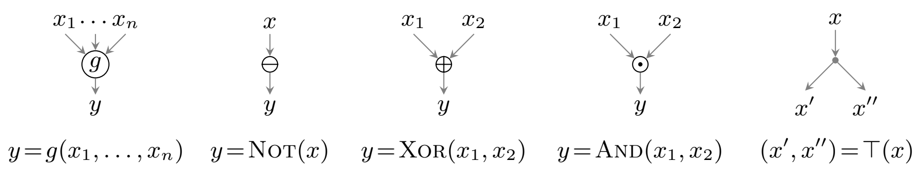


Fig. 7: Gates and branching nodes in the computation graph.

difference between the values of a wire w in the correct execution and w^* in the faulted execution by $\delta w = w \oplus w^*$, similar to differential cryptanalysis. The fault δw may propagate to other gates, and we call a gate g' active (or activated) in a faulted execution if $\delta v = 1$ for any input wire $v \in \textsc{in}(g')$. Any such fault g^* can be interpreted as a data-dependent bitflip fault on the output wire y:

$$y^* = g^*(x_1, \ldots, x_n) = g(x_1, \ldots, x_n) \oplus \delta g(x_1, \ldots, x_n)\,.$$

For this reason, we focus on wire bitflips (i.e., the case $\delta g(x_1, \ldots, x_n) = 1$) in the following discussion.

For our countermeasure, we require that the computation is masked and implemented redundantly for fault detection.

Masking. We assume that the computation implements first-order masking using $d + 1 \geq 2$ shares. In particular, an attacker who learns the value of one wire or of all input wires to one nonlinear gate must not learn any information on the native variables. The nonlinear gates may thus be, for example, the And-gates in a general masked implementation, or the entire component functions of a threshold implementation.

Redundancy and fault detection. To detect faults, we for instance duplicate the implementation and feed the same inputs to both instances. The detection is defined by an error detector Δ that only returns the result of the computation if no errors are detected in a selected set of variables $\mathcal{V}_\Delta$, i.e., if

$$\Delta := \bigvee_{v \in \mathcal{V}_\Delta} \delta v = \bigvee_{v \in \mathcal{V}_\Delta} (v \oplus v^*) = 0\,.$$

The SIFA attacker collects plaintext-ciphertext samples with $\Delta = 0$, as they receive no output if $\Delta = 1$, and uses this condition to derive information about the value of wires near the faulted gate g^* or wire w^*.

Example. As an example throughout this section, Figure 8 lists the operations of a masked implementation of the 3-bit S-box χ_3 together with its computation graph similar to [33]. When combined with the error detector Δ that checks the output variables of the circuit, this implementation is susceptible to single-fault SIFA with several possible fault locations. One of these is illustrated in Figure 8: If a bitflip is induced in a_0 as indicated (ϟ), then the condition $\Delta = 0$ implies $b_0 \oplus b_1 = b = 0$. We want to protect this implementation against single-fault SIFA by modifying either the detector or the structure of the DAG.

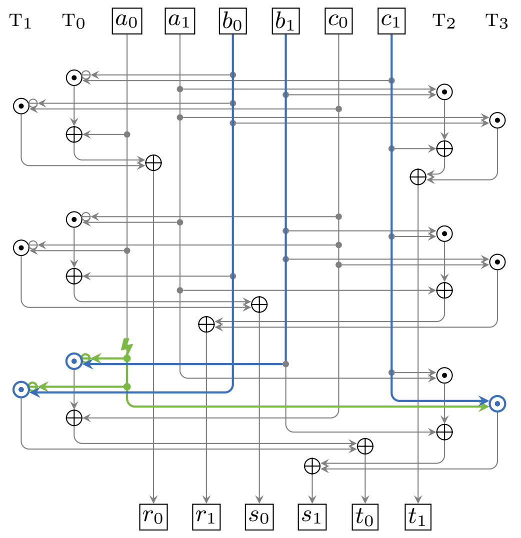
Input: $\{a_0, a_1, b_0, b_1, c_0, c_1\}$

$\mathrm{T}_0 \leftarrow \overline{b_0} \odot c_1$; $\mathrm{T}_2 \leftarrow a_1 \odot b_1$
$\mathrm{T}_1 \leftarrow \overline{b_0} \odot c_0$; $\mathrm{T}_3 \leftarrow a_1 \odot b_0$
$\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus a_0$; $\mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus c_1$
$r_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1$; $t_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3$

$\mathrm{T}_0 \leftarrow \overline{c_0} \odot a_1$; $\mathrm{T}_2 \leftarrow b_1 \odot c_1$
$\mathrm{T}_1 \leftarrow \overline{c_0} \odot a_0$; $\mathrm{T}_3 \leftarrow b_1 \odot c_0$
$\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus b_0$; $\mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus a_1$
$s_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1$; $r_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3$
ϟa_0
$\mathrm{T}_0 \leftarrow \overline{a_0} \odot b_1$; $\mathrm{T}_2 \leftarrow c_1 \odot a_1$
$\mathrm{T}_1 \leftarrow \overline{a_0} \odot b_0$; $\mathrm{T}_3 \leftarrow c_1 \odot a_0$
$\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus c_0$; $\mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus b_1$
$t_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1$; $s_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3$

Output: $\{r_0, r_1, s_0, s_1, t_0, t_1\}$

(a) Operations



(b) Computation graph

Fig. 8: Bitflip in masked χ_3 using 2 shares (resharing at the output omitted).

4.2 A General Criterion for Resistance against Single-Fault SIFA

Consider a masked implementation with a detection-based countermeasure defined by an error detector Δ that only returns the result of the computation if $\Delta = 0$, as defined in the previous section. We call the implementation *single-fault SIFA-resistant* if each possible single-bit fault is either detected by Δ or activates at most one nonlinear gate.

To see why this criterion is sufficient, consider a fault g^* and its corresponding bitflip fault on the wire y. The attacker collects plaintext-ciphertext samples with $\Delta = 0$, as they receive no output if $\Delta = 1$. The samples satisfy one of the following two conditions:

- $\delta y = 0$, i.e., no bitflip happened because $\delta g(x_1, \ldots, x_n) = 0$. The attacker learns at most these values $x_1, \ldots, x_n$. Since the implementation is masked, this information is independent of the native input and output values and thus does not allow the attacker to derive any information on the processed data or keys.
- $\delta y = 1$, but the bitflip did not propagate to Δ. The criterion implies that there is at most one nonlinear gate g' with some changed input $v^* = v \oplus \delta v$. The attacker may exploit this differential information to learn the inputs of g', which are however independent of the native inputs, and will not learn anything from the other, trivial differentials (of nonlinear gates with zero input difference or of linear gates).

4.3 Protection against SIFA using Fine-Grained Detection

We now explore how a masked implementation can be extended with a suitable detector $\mathit{\Delta}$ in order to achieve a single-fault SIFA-resistant implementation.

A straightforward, albeit not very efficient approach to satisfy the single-fault SIFA-resistance criterion follows directly from its definition: We can add local checks for inputs of nonlinear gates. Assume for instance that we duplicate the implementation and feed the same inputs to both instances. For each nonlinear gate $g \in \mathcal{G}_{\text{N}}$ and each of its input wires $w \in \textsc{in}(g)$, we add a check to update the detector $\mathit{\Delta}' = \mathit{\Delta} \vee (w \oplus w^*)$, which adds 2 gates ($\varovee$, $\oplus$), 2 branches ($\top(w)$, $\top(w^*)$), and 6 new wires ($\mathit{\Delta}'$, δw, $(w', w'') = \top(w)$, $(w^{*\prime}, w^{*\prime\prime}) = \top(w^*)$) to the DAG. We alternatively represent this as a single checking node ✓ in the DAG of a single instance of the implementation. Then, a fault may activate a single nonlinear gate $g \in \mathcal{G}_{\text{N}}$ without detection by $\mathit{\Delta}$ (if it is induced on $w^{*\prime}$ after the check and before the gate g), but it cannot activate two gates, since there are no paths without a check either from any branching to two nonlinear gates or from one nonlinear gate to another. Any single-bit fault in one of the two redundant computations or in the additional circuitry for the detector $\mathit{\Delta}$ are either detected or do not leak information to the attacker.

It is not necessary to check the inputs to all nonlinear gates separately. We are only interested in wires whose faults can activate multiple nonlinear gates. For example, in the masked implementation of an S-box or nonlinear operation, we can expect that most inputs to nonlinear gates in the DAG are directly branched from the shares of the S-box inputs, i.e., the gate input checks would check the same variable many times. Instead, we want to check only once.

In the DAG, this corresponds to a binary subtree rooted in an input variable whose inner nodes are branchings and whose leaves are other gates. We refer to edges ending in leaves as twigs and to the other, inner edges as stems. The basic approach checks each twig ending in a nonlinear gate and thus precludes a fault that activates this twig in addition to another parent or sibling edge in the DAG. Instead, it is sufficient to check only those twigs whose sibling edge is also a twig (rather than a stem), and to check only one of the two twigs. In other words, we check a variable that serves as input to multiple gates only once, right before its last use. We call this last check the *sink* of (this part of) the tree, and it will be activated if more than one twig in (this part of) the tree is active. Additionally, we also consider the output variables as sinks, since they will either be checked in the next nonlinear layer, or propagate faults deterministically to the cipher output. As a result, every wire w in the DAG is a *safe* wire that has a sink s such that there is exactly one directed path $w \to s$ and it contains at most one nonlinear gate. This implies the single-fault SIFA-resistance criterion of Section 4.2.

In the χ_3 example, by checking only such variables and only after they are used for the last time, we can reduce the number of checks to 6, i.e., once for each input variable. The result is illustrated in Figure 9.

Input: $\{a_0, a_1, b_0, b_1, c_0, c_1\}$

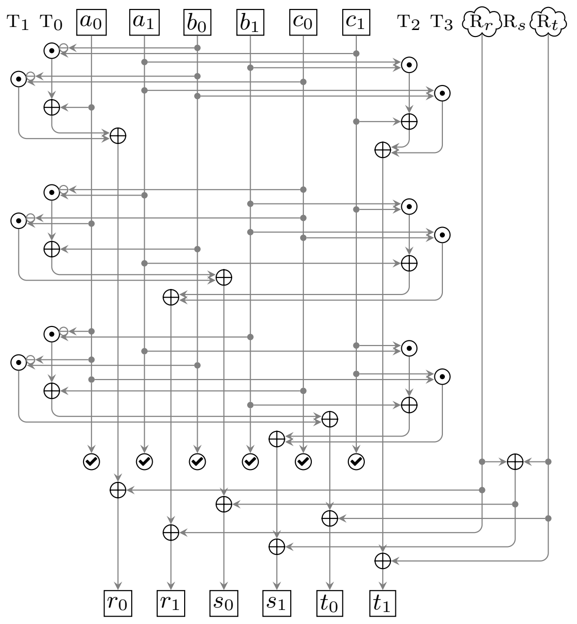
$\mathrm{T}_0 \leftarrow \overline{b_0} \odot c_1$; $\mathrm{T}_2 \leftarrow a_1 \odot b_1$
$\mathrm{T}_1 \leftarrow \overline{b_0} \odot c_0$; $\mathrm{T}_3 \leftarrow a_1 \odot b_0$
$\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus a_0$; $\mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus c_1$
$r_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1$; $t_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3$

$\mathrm{T}_0 \leftarrow \overline{c_0} \odot a_1$; $\mathrm{T}_2 \leftarrow b_1 \odot c_1$
$\mathrm{T}_1 \leftarrow \overline{c_0} \odot a_0$; $\mathrm{T}_3 \leftarrow b_1 \odot c_0$
$\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus b_0$; $\mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus a_1$
$s_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1$; $r_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3$

$\mathrm{T}_0 \leftarrow \overline{a_0} \odot b_1$; $\mathrm{T}_2 \leftarrow c_1 \odot a_1$
$\mathrm{T}_1 \leftarrow \overline{a_0} \odot b_0$; $\mathrm{T}_3 \leftarrow c_1 \odot a_0$
$\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus c_0$; $\mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus b_1$
$t_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1$; $s_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3$
$\mathrm{R}_s \leftarrow \mathrm{R}_r \oplus \mathrm{R}_t$
$r_0 \leftarrow r_0 \oplus \mathrm{R}_r$; $s_0 \leftarrow s_0 \oplus \mathrm{R}_s$
$t_0 \leftarrow t_0 \oplus \mathrm{R}_t$; $r_1 \leftarrow r_1 \oplus \mathrm{R}_r$
$s_1 \leftarrow s_1 \oplus \mathrm{R}_s$; $t_1 \leftarrow t_1 \oplus \mathrm{R}_t$

Output: $\{r_0, r_1, s_0, s_1, t_0, t_1\}$

(a) Operations



(b) Computation graph

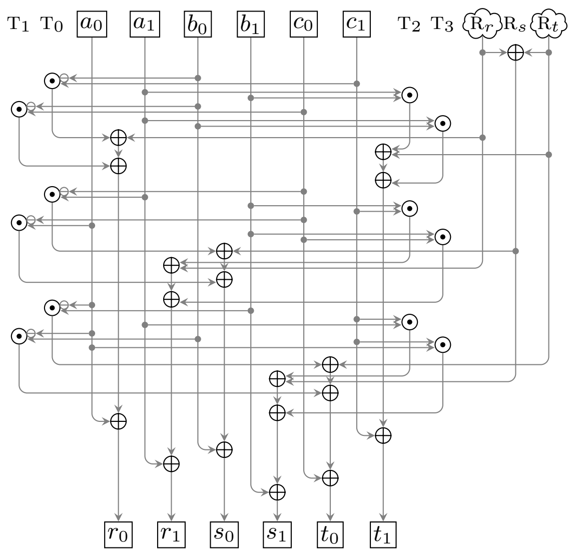
Fig. 9: Single-fault SIFA-resistant χ_3 using 2 shares, with local checks.

Input: $\{a_0, a_1, b_0, b_1, c_0, c_1\}$

$\mathrm{R}_s \leftarrow \mathrm{R}_r \oplus \mathrm{R}_t$
$\mathrm{T}_0 \leftarrow \overline{b_0} \odot c_1$; $\mathrm{T}_2 \leftarrow a_1 \odot b_1$
$\mathrm{T}_1 \leftarrow \overline{b_0} \odot c_0$; $\mathrm{T}_3 \leftarrow a_1 \odot b_0$
$r_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{R}_r$; $t_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{R}_t$
$r_0 \leftarrow r_0 \oplus \mathrm{T}_1$; $t_1 \leftarrow t_1 \oplus \mathrm{T}_3$
$\mathrm{T}_0 \leftarrow \overline{c_0} \odot a_1$; $\mathrm{T}_2 \leftarrow b_1 \odot c_1$
$\mathrm{T}_1 \leftarrow \overline{c_0} \odot a_0$; $\mathrm{T}_3 \leftarrow b_1 \odot c_0$
$s_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{R}_s$; $r_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{R}_r$
$s_0 \leftarrow s_0 \oplus \mathrm{T}_1$; $r_1 \leftarrow r_1 \oplus \mathrm{T}_3$
$\mathrm{T}_0 \leftarrow \overline{a_0} \odot b_1$; $\mathrm{T}_2 \leftarrow c_1 \odot a_1$
$\mathrm{T}_1 \leftarrow \overline{a_0} \odot b_0$; $\mathrm{T}_3 \leftarrow c_1 \odot a_0$
$t_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{R}_t$; $s_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{R}_s$
$t_0 \leftarrow t_0 \oplus \mathrm{T}_1$; $s_1 \leftarrow s_1 \oplus \mathrm{T}_3$
$r_0 \leftarrow r_0 \oplus a_0$; $t_1 \leftarrow t_1 \oplus c_1$
$s_0 \leftarrow s_0 \oplus b_0$; $r_1 \leftarrow r_1 \oplus a_1$
$t_0 \leftarrow t_0 \oplus c_0$; $s_1 \leftarrow s_1 \oplus b_1$

Output: $\{r_0, r_1, s_0, s_1, t_0, t_1\}$

(a) Operations



(b) Computation graph

Fig. 10: Single-fault SIFA-resistant χ_3 using 2 shares, with global checks.

4.4 Ensuring Fault Propagation

In this section, we discuss under which conditions the fine-grained, local detection of Section 4.3 can be replaced by global checks, similar to Section 3. We will again use the concept of sink nodes as in Section 4.3, in the sense of nodes whose activation will be detected by Δ. However, instead of implementing actual local checks in the sink nodes, these sinks are virtual nodes whose effect on Δ follows from properties of the cipher or masking approach.

First consider a uniform direct sharing of an invertible S-box. Since the sharing is uniform, the masked circuit is also invertible. As a consequence, for fixed resharing inputs, if any of the intermediate masked S-box output bits are activated by a fault, this will activate at least one bit in the masked cipher output. Thus, if the detection variables $\mathcal{V}_\Delta$ include all masked cipher output variables, then the S-box output variables can serve as sinks – what remains to be done is to ensure that each wire is a safe wire with respect to these sinks, and ideally, to get rid of the requirement to perform redundant computations for the same values of the shares. We first address the latter question.

Consider a masked implementation of an S-box which is not necessarily uniform. Instead of the individual masked S-box output bits, we can use the unmasked S-box output as sinks and add corresponding virtual nodes that compute these as sums of the masked S-box outputs to the circuit. Any fault in this unmasked S-box output would activate at least one bit in the unmasked cipher output, so the detection variables $\mathcal{V}_\Delta$ can be reduced to the unmasked values and evaluated for arbitrary resharing inputs.

Now, we still need to ensure that any wire w in the S-box circuit is a safe wire with respect to one of these sinks s, i.e., that there is exactly one directed path $w \to s$ and it contains at most one nonlinear gate. This may be violated due to branching and due to composition of nonlinear gates within an S-box. If the circuit contains such a composition of nonlinear gates, it needs to be decomposed into smaller, bijective S-boxes with nonlinear depth 1 first. For branches, we consider the branching subtree as in Section 4.3. We need to ensure that whenever two twigs in this tree activate, a sink s activates. In particular, this implies that for every branch node b, there must be a sink s such that there is a unique path $b \to s$, and this path contains only linear nodes (including branches). This may require restructuring the tree such that at least one of the last two uses of a variable (the tree root) is in a linear gate, taking care that the modifications do not invalidate the security of the masked implementation.

The approach is easy to apply to the χ_3 example. We performed the following modifications to the circuit from Figure 8 so that each wire is now a safe wire:

1. Delay $a_0 \oplus r_0$ and $c_1 \oplus t_1$ until the very end.
2. Delay $a_1 \oplus r_1$, $b_0 \oplus s_0$, $b_1 \oplus s_1$, and $c_0 \oplus t_0$ until the very end (optional).
3. Move the resharing position to preserve security of the masking.

The resulting circuit in Figure 10 shows similarities with the Toffoli-based implementation of χ_3 in Section 3.2, but there are still significant differences; most notably, the necessity for resharing variables $\mathrm{R}_r, \mathrm{R}_t$ and the lower circuit depth of the solution in Figure 10.

4.5 Towards Protection against Multiple Faults

So far, we focused only on single-fault SIFA attackers and corresponding countermeasures. Both the attack approach and the countermeasure with local checks can be generalized to a multi-fault attacker who faults up to d gates or wires.

Consider a circuit protected by dth-order masking with $d+1$ or more shares, i.e., an attacker who learns up to d shares of any variable still does not gain any information on its native value. Let the circuit be implemented with at least $d+1$ redundant computations and an error detector $\varDelta$ of at least d bits. For simplicity, assume that each of the attacker's d faults is a bitflip fault. We call the implementation *d-fault SIFA-resistant* if each possible d-bit fault is either detected by $\varDelta$ or activates at most d nonlinear gates in total.

This criterion can, for instance, be satisfied by checking all inputs to nonlinear gates with the following construction. We use $d+1$ redundant computations and an $n_\varDelta$-bit error detector $\varDelta = (\varDelta_1, \ldots, \varDelta_{n_\varDelta})$, where $n_\varDelta = d$ for odd d and $n_\varDelta = d+1$ for even d. For each relevant input wire, we branch d times to update d different error detector bits $\varDelta_i$ with the differences to all d other computations. In other words, we compute all $\binom{d+1}{2}$ differences in this bit between any two redundant computations and ensure that for each computation, each of the d comparisons activates a different detector $\varDelta_i$. Distributing the $\binom{d+1}{2}$ differences to the various $\varDelta_i$ corresponds to an edge coloring problem with $n_\varDelta$ colors in the complete graph with $d+1$ vertices, which is easy to solve. Then, activating k gates in one computation without detection by $\varDelta$ requires at least $\min(k, d+1)$ faults: each gate can either be activated without triggering $\varDelta$ by placing a fault between the checking branches and the nonlinear gate; or it can be activated while triggering d error detector bits $\varDelta_i$, each of which requires either a fault in the corresponding computation or faulting $\varDelta_i$ directly to eliminate. Thus, in summary, at least $d+1$ faults would be required in order to activate $d+1$ or more nonlinear gates and thus learn $d+1$ shares of any variable to deduce information on its native value.

Clearly, this approach is only efficient in practice for very small protection order d without further optimizations. Since the size of each masked implementation grows quadratically in d, and the checking cost per nonlinear gate in this implementation also grows quadratically in d, the construction is only of theoretic interest for larger d.

5 Conclusion

In this paper, we proposed two different approaches to counteract SIFA on an algorithmic level. First, we showed that by using reversible gates for the nonlinear operations in the implementation of the masked cipher, we can construct circuits where a single fault in the computation of the cipher will always propagate to its output. It can then be detected via redundant computations that are typically implemented to cope with other fault attacks like DFA. This approach can be implemented efficiently and its applicability was shown for 3-bit, 4-bit, and 5-bit S-boxes.

Additionally, we presented an alternative countermeasure strategy based on fine-grained detection. It can be used to protect arbitrary masked circuits, and extended to cope with multi-fault SIFA, albeit at a higher implementation cost.

Acknowledgments. Part of this work has been supported by the Austrian Science Fund (FWF): J 4277-N38 and by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. Joan Daemen is supported by the European Research Council under the ERC advanced grant agreement under grant ERC-2017-ADG Nr. 788980 ESCADA.

References

1. Albrecht, M.R., Rechberger, C., Schneider, T., Tiessen, T., Zohner, M.: Ciphers for MPC and FHE. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology – EUROCRYPT 2015. LNCS, vol. 9056, pp. 430–454. Springer (2015). https://doi.org/10.1007/978-3-662-46800-5_17
2. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proceedings of the IEEE **94**(2), 370–382 (2006). https://doi.org/10.1109/JPROC.2005.862424
3. Barthe, G., Dupressoir, F., Faust, S., Grégoire, B., Standaert, F.X., Strub, P.Y.: Parallel implementations of masking schemes and the bounded moment leakage model. In: Coron, J.S., Nielsen, J.B. (eds.) Advances in Cryptology – EUROCRYPT 2017. LNCS, vol. 10210, pp. 535–566 (2017). https://doi.org/10.1007/978-3-319-56620-7_19
4. Belaïd, S., Benhamouda, F., Passelègue, A., Prouff, E., Thillard, A., Vergnaud, D.: Private multiplication over finite fields. In: Katz, J., Shacham, H. (eds.) Advances in Cryptology – CRYPTO 2017. LNCS, vol. 10403, pp. 397–426. Springer (2017). https://doi.org/10.1007/978-3-319-63697-9_14
5. Bennett, C.H.: Logical reversibility of computation. IBM Journal of Research and Development **17**(6), 525–532 (1973). https://doi.org/10.1147/rd.176.0525
6. Bertoni, G., Daemen, J., Hoffert, S., Peeters, M., Van Assche, G., Van Keer, R.: Farfalle: parallel permutation-based cryptography. IACR Transactions on Symmetric Cryptology **2017**(4), 1–38 (2017). https://doi.org/10.13154/tosc.v2017.i4.1-38
7. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The Keccak SHA-3 submission (Version 3.0). http://keccak.noekeon.org/Keccak-submission-3.pdf (2011)
8. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., Van Keer, R.: Ketje v2. Submission to the CAESAR competition (2016), https://keccak.team/files/Ketjev2-doc2.0.pdf
9. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., Van Keer, R.: Keyak v2. Submission to the CAESAR competition (2016), https://keccak.team/files/Keyakv2-doc2.2.pdf
10. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Kaliski Jr., B.S. (ed.) Advances in Cryptology – CRYPTO '97. LNCS, vol. 1294, pp. 513–525. Springer (1997). https://doi.org/10.1007/BFb0052259
11. Bilgin, B., Nikova, S., Nikov, V., Rijmen, V., Tokareva, N.N., Vitkup, V.: Threshold implementations of small S-boxes. Cryptography and Communications **7**(1), 3–33 (2015). https://doi.org/10.1007/s12095-014-0104-7

12. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer (2007). https://doi.org/10.1007/978-3-540-74735-2_31
13. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults (extended abstract). In: Fumy, W. (ed.) Advances in Cryptology – EUROCRYPT '97. LNCS, vol. 1233, pp. 37–51. Springer (1997). https://doi.org/10.1007/3-540-69053-0_4
14. Clavier, C.: Secret external encodings do not prevent transient fault analysis. In: Paillier, P., Verbauwhede, I. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2007. LNCS, vol. 4727, pp. 181–194. Springer (2007)
15. Cnudde, T.D., Nikova, S.: More efficient private circuits II through threshold implementations. In: Fault Diagnosis and Tolerance in Cryptography – FDTC 2016. pp. 114–124. IEEE Computer Society (2016). https://doi.org/10.1109/FDTC.2016.15
16. Cnudde, T.D., Reparaz, O., Bilgin, B., Nikova, S., Nikov, V., Rijmen, V.: Masking AES with d+1 shares in hardware. In: Gierlichs, B., Poschmann, A.Y. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2016. LNCS, vol. 9813, pp. 194–212. Springer (2016). https://doi.org/10.1007/978-3-662-53140-2_10
17. Colombier, B., Menu, A., Dutertre, J.M., Moëllic, P.A., Rigaud, J.B., Danger, J.L.: Laser-induced single-bit faults in flash memory: Instructions corruption on a 32-bit microcontroller. IACR Cryptology ePrint Archive, Report 2018/1042 (2018), `https://eprint.iacr.org/2018/1042`
18. Daemen, J.: Cipher and hash function design, strategies based on linear and differential cryptanalysis. Ph.D. thesis, KU Leuven (1995), `http://jda.noekeon.org/`
19. Daemen, J.: Changing of the guards: A simple and efficient method for achieving uniformity in threshold sharing. In: Fischer, W., Homma, N. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2017. LNCS, vol. 10529, pp. 137–153. Springer (2017). https://doi.org/10.1007/978-3-319-66787-4_7
20. Daemen, J., Hoffert, S., Van Assche, G., Van Keer, R.: The design of Xoodoo and Xoofff. IACR Transactions on Symmetric Cryptology **2018**(4), 1–38 (2018). https://doi.org/10.13154/tosc.v2018.i4.1-38
21. Daemen, J., Peeters, M., Van Assche, G., Rijmen, V.: Nessie proposal: the block cipher NOEKEON. Nessie submission (2000), `http://gro.noekeon.org/`
22. De Cannière, C.: Analysis and design of symmetric encryption algorithms. Ph.D. thesis, KU Leuven (2007)
23. De Meyer, L., Arribas, V., Nikova, S., Nikov, V., Rijmen, V.: M&M: Masks and macs against physical attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(1), 25–50 (2019). https://doi.org/10.13154/tches.v2019.i1.25-50
24. Dobraunig, C., Eichlseder, M., Groß, H., Mangard, S., Mendel, F., Primas, R.: Statistical ineffective fault attacks on masked AES with fault countermeasures. In: Peyrin, T., Galbraith, S.D. (eds.) Advances in Cryptology – ASIACRYPT 2018. LNCS, vol. 11273, pp. 315–342. Springer (2018). https://doi.org/10.1007/978-3-030-03329-3_11
25. Dobraunig, C., Eichlseder, M., Korak, T., Mangard, S., Mendel, F., Primas, R.: SIFA: Exploiting ineffective fault inductions on symmetric cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(3), 547–572 (2018). https://doi.org/10.13154/tches.v2018.i3.547-572

26. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1.2. Submission to the CAESAR Competition (2016), https://ascon.iaik.tugraz.at/files/asconv12.pdf
27. Dobraunig, C., Mangard, S., Mendel, F., Primas, R.: Fault attacks on nonce-based authenticated encryption: Application to Keyak and Ketje. Cryptology ePrint Archive, Report 2018/852 (2018), https://eprint.iacr.org/2018/852
28. Fuhr, T., Jaulmes, É., Lomné, V., Thillard, A.: Fault attacks on AES with faulty ciphertexts only. In: Fischer, W., Schmidt, J.M. (eds.) Fault Diagnosis and Tolerance in Cryptography – FDTC 2013. pp. 108–118. IEEE Computer Society (2013)
29. Golubitsky, O., Maslov, D.: A study of optimal 4-bit reversible Toffoli circuits and their synthesis. IEEE Transactions on Computers **61**(9), 1341–1353 (2012). https://doi.org/10.1109/TC.2011.144
30. Groß, H., Iusupov, R., Bloem, R.: Generic low-latency masking in hardware. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(2), 1–21 (2018). https://doi.org/10.13154/tches.v2018.i2.1-21
31. Groß, H., Mangard, S.: Reconciling d+1 masking in hardware and software. In: Fischer, W., Homma, N. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2017. LNCS, vol. 10529, pp. 115–136. Springer (2017). https://doi.org/10.1007/978-3-319-66787-4_6
32. Groß, H., Mangard, S., Korak, T.: Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. IACR Cryptology ePrint Archive, Report 2016/486 (2016), https://eprint.iacr.org/2016/486
33. Groß, H., Schaffenrath, D., Mangard, S.: Higher-order side-channel protected implementations of KECCAK. In: DSD. pp. 205–212. IEEE Computer Society (2017)
34. Ishai, Y., Prabhakaran, M., Sahai, A., Wagner, D.A.: Private circuits II: Keeping secrets in tamperable circuits. In: Vaudenay, S. (ed.) Advances in Cryptology – EUROCRYPT 2006. LNCS, vol. 4004, pp. 308–327. Springer (2006). https://doi.org/10.1007/11761679_19
35. Ishai, Y., Sahai, A., Wagner, D.A.: Private circuits: Securing hardware against probing attacks. In: Boneh, D. (ed.) Advances in Cryptology – CRYPTO 2003. LNCS, vol. 2729, pp. 463–481. Springer (2003). https://doi.org/10.1007/978-3-540-45146-4_27
36. Knudsen, L.R., Leander, G., Poschmann, A., Robshaw, M.J.B.: PRINTcipher: A block cipher for IC-printing. In: Mangard, S., Standaert, F.X. (eds.) Cryptographic Hardware and Embedded Systems – CHES. LNCS, vol. 6225, pp. 16–32. Springer (2010). https://doi.org/10.1007/978-3-642-15031-9_2
37. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M.J. (ed.) Advances in Cryptology – CRYPTO '99. LNCS, vol. 1666, pp. 388–397. Springer (1999). https://doi.org/10.1007/3-540-48405-1_25
38. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development **5**(3), 183–191 (1961). https://doi.org/10.1147/rd.53.0183
39. Leander, G., Poschmann, A.: On the classification of 4 bit S-boxes. In: Carlet, C., Sunar, B. (eds.) Arithmetic of Finite Fields – WAIFI 2007. LNCS, vol. 4547, pp. 159–176. Springer (2007). https://doi.org/10.1007/978-3-540-73074-3_13
40. Quisquater, J.J., Samyde, D.: Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In: Attali, I., Jensen, T.P. (eds.) Smart Card Programming and Security – E-smart 2001. LNCS, vol. 2140, pp. 200–210. Springer (2001). https://doi.org/10.1007/3-540-45418-7_17

41. Reparaz, O., Bilgin, B., Nikova, S., Gierlichs, B., Verbauwhede, I.: Consolidating masking schemes. In: Gennaro, R., Robshaw, M. (eds.) Advances in Cryptology – CRYPTO 2015. LNCS, vol. 9215, pp. 764–783. Springer (2015). https://doi.org/10.1007/978-3-662-47989-6_37
42. Reparaz, O., De Meyer, L., Bilgin, B., Arribas, V., Nikova, S., Nikov, V., Smart, N.P.: CAPA: The spirit of beaver against physical attacks. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018. LNCS, vol. 10991, pp. 121–151. Springer (2018). https://doi.org/10.1007/978-3-319-96884-1_5
43. Schneider, T., Moradi, A., Güneysu, T.: ParTI – towards combined hardware countermeasures against side-channel and fault-injection attacks. In: Robshaw, M., Katz, J. (eds.) Advances in Cryptology – CRYPTO 2016. LNCS, vol. 9815, pp. 302–332. Springer (2016). https://doi.org/10.1007/978-3-662-53008-5_11
44. Selmke, B., Zinnecker, K., Koppermann, P., Miller, K., Heyszl, J., Sigl, G.: Locked out by latch-up? an empirical study on laser fault injection into Arm Cortex-M processors. In: Fault Diagnosis and Tolerance in Cryptography – FDTC 2018. pp. 7–14. IEEE Computer Society (2018). https://doi.org/10.1109/FDTC.2018.00010
45. Shende, V.V., Prasad, A.K., Markov, I.L., Hayes, J.P.: Synthesis of reversible logic circuits. IEEE Transactions on CAD of Integrated Circuits and Systems **22**(6), 710–722 (2003). https://doi.org/10.1109/TCAD.2003.811448
46. Sugawara, T.: 3-share threshold implementation of AES S-box without fresh randomness. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(1), 123–145 (2019). https://doi.org/10.13154/tches.v2019.i1.123-145
47. Toffoli, T.: Reversible computing. In: de Bakker, J.W., van Leeuwen, J. (eds.) Automata, Languages and Programming, 1980. LNCS, vol. 85, pp. 632–644. Springer (1980). https://doi.org/10.1007/3-540-10003-2_104
48. Tupsamudre, H., Bisht, S., Mukhopadhyay, D.: Destroying fault invariant with randomization – A countermeasure for AES against differential fault attacks. In: Batina, L., Robshaw, M. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2014. LNCS, vol. 8731, pp. 93–111. Springer (2014)

A Threshold Implementations of p_T and p_χ

Figure 11 shows the algorithmic representation of a securely masked (using three shares) and single-fault SIFA-protected Toffoli gate p_T fulfilling the three requirements for threshold implementations (TI). Namely the gate fulfills: 1) *correctness*, since the gate correctly implements the equations $a = a \oplus b \odot c$ which can be checked be adding all output shares of a (the equations $b = b$ and $c = c$ are trivial), 2) *uniformity*, which follows from the fact that for each output share a single share of a appears in additive form, and 3) *non-completeness*, because for each calculation of one output share, one share index never appears (e.g., the calculation of the output share a_0 does not use any shares with the index 1 like b_1 or c_1). The implementation of p_χ follows analogously by replacing either one of the nine AND gates with a NAND gate. A secure hardware variant of the Toffoli gate is shown in Figure 12. Again, the registers ensure that a single fault cannot influence all shares of variables that are fed into nonlinear AND gates without detecting it at the output.

```
Input: $\{a_0, a_1, a_2, b_0, b_1, b_2, c_0, c_1, c_2\}$
$\mathrm{T}_0 \leftarrow b_0 \odot c_0$
$a_0 \leftarrow a_0 \oplus \mathrm{T}_0$
$\mathrm{T}_0 \leftarrow b_0 \odot c_2$
$a_0 \leftarrow a_0 \oplus \mathrm{T}_0$
$\mathrm{T}_0 \leftarrow b_2 \odot c_0$
$a_0 \leftarrow a_0 \oplus \mathrm{T}_0$

$\mathrm{T}_0 \leftarrow b_1 \odot c_1$
$a_1 \leftarrow a_1 \oplus \mathrm{T}_0$
$\mathrm{T}_0 \leftarrow b_1 \odot c_0$
$a_1 \leftarrow a_1 \oplus \mathrm{T}_0$
$\mathrm{T}_0 \leftarrow b_0 \odot c_1$
$a_1 \leftarrow a_1 \oplus \mathrm{T}_0$

$\mathrm{T}_0 \leftarrow b_2 \odot c_2$
$a_2 \leftarrow a_2 \oplus \mathrm{T}_0$
$\mathrm{T}_0 \leftarrow b_2 \odot c_1$
$a_2 \leftarrow a_2 \oplus \mathrm{T}_0$
$\mathrm{T}_0 \leftarrow b_1 \odot c_2$
$a_2 \leftarrow a_2 \oplus \mathrm{T}_0$

Output: $\{a_0, a_1, a_2, b_0, b_1, b_2, c_0, c_1, c_2\}$
```

Fig. 11: Algorithmic representation of masked Toffoli gate (p_T) using 3 shares.

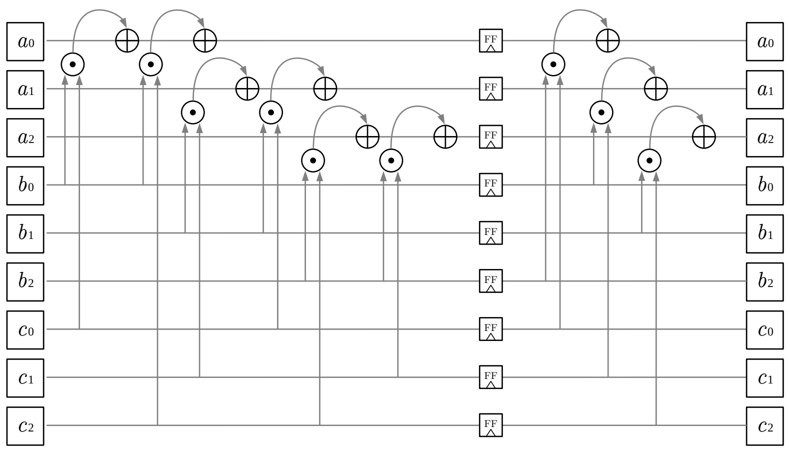


Fig. 12: 3-share TI and single-fault SIFA-protected Toffoli gate in hardware.