IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2020, No. 3, pp. 508–543. DOI:10.13154/tches.v2020.i3.508-543

Protecting against Statistical Ineffective Fault Attacks

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

[1] Radboud University, Nijmegen, The Netherlands, {joan,cdobraunig}@cs.ru.nl
[2] Graz University of Technology, Graz, Austria, {firstname.lastname}@iaik.tugraz.at
[3] SGS Digital Trust Services GmbH, Graz, Austria, hannes.gross@sgs.com
[4] Infineon Technologies AG, Neubiberg, Germany, florian.mendel@infineon.com

Abstract. Statistical Ineffective Fault Attacks (SIFA) pose a threat for many practical implementations of symmetric primitives. Countermeasures against both power analysis and fault attacks typically do not prevent straightforward SIFA attacks, which require only very limited knowledge about the concrete implementation. Therefore, the exploration of countermeasures against SIFA that do not rely on protocols or physical protection mechanisms is of great interest. In this paper, we describe different countermeasure strategies against SIFA. First, we introduce an abstraction layer between the algorithmic specification of a cipher and its implementation in hardware or software to study and describe resistance against SIFA. We then show that by basing the masked implementation on permutations as building blocks, we can build circuits that withstand single-fault SIFA and DPA attacks. We show how this approach can be applied to 3-bit, 4-bit, and 5-bit S-boxes and the AES S-box. Additionally, we present a strategy based on fine-grained fault detection suitable for protecting any circuit against SIFA attacks. Although this approach may lead to a higher implementation cost due to the fine-grained detection needed, it can be used to protect arbitrary circuits and can be generalized to cover multi-fault SIFA. For single-fault SIFA protection, our countermeasures only have a small computational overhead compared to a simple combination of masking and duplication.

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

1 Introduction

Motivation. Fault attacks [BDL97, BS97] and passive side-channel attacks, like power or electromagnetic (EM) analysis [KJJ99, QS01], are real-world threats for implementations of cryptographic primitives. Therefore, devices like smart cards that may be physically accessible by an attacker typically implement countermeasures against these attacks.

A common countermeasure at algorithmic level is the combination of masking against side-channel attacks and some kind of redundancy against fault attacks. In masking one splits input and intermediate variables 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 [ISW03, BBP+17, RBN+15, DRB+16, GIB18, GM17, GMK16, BDF+17]. Redundant computation, on the other hand, is used to detect malicious or environmental influences that could lead to faulty cipher outputs. If a fault is detected,

*The list of authors is in alphabetical order (https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf)

Received: 2020-01-15 Accepted: 2019-03-15 Published: 2020-06-19

the cipher output is either not released ("detection") or released in a form that should prevent its exploitation ("infection") [BECN+06, TBM14]. Examples of recent works that propose combinations of such countermeasure techniques include, among others, ParTI [SMG16], Private Circuits II [IPSW06, DN16], and M&M [DAN+19]. A quite different approach was chosen with CAPA [RDB+18], where an actively secure MPC protocol was adapted for cryptographic computations to provide strong protection against implementation attacks. Due to its high cost, this approach is however of limited interest for practical applications.

Up until recently, implementations combining masking with fault countermeasures like redundancy were typically assumed to offer protection against both power analysis and fault attacks. However, recently a new type of attack was introduced that combines the principles of Ineffective Fault Attacks (IFA) [Cla07] and Statistical Fault Analysis (SFA) [FJLT13]. This attack is called Statistical Ineffective Fault Attack (SIFA) [DEK+18] and was shown to be applicable to ciphers protected with masking as well as fault detection or infection [DEG+18]. SIFA succeeds in doing this by exploiting the dependence of faults propagating to the cipher output on the value of intermediate variables. The mere presence of a fault and the non-faulty outputs of the cipher computations provide sufficient information to retrieve the value of these intermediate variables.

Our contribution. The contribution of this paper is two-fold. First, we analyze the general root causes that lead to successful SIFA attacks in more detail. To study these causes and also describe resistance against SIFA and Differential Power Analysis (DPA) [KJJ99], we introduce an abstraction layer between the algorithmic specification of a cipher and its implementation in hardware or software. In this layer, we express the cipher as a *circuit*, taking as input an array of variables and returning as output an array of variables. We split this circuit into sub-circuits where each sub-circuit takes as input variables that are either the cipher's input or the other sub-circuit's outputs. This is done recursively until we get to the level of circuits that we no longer split into sub-circuits and that we call basic circuits. In the implementation of these basic circuits it is then essential to *keep the computations and internal variables of the different basic circuits* separated. Thus, basic circuits are the natural place to define the concept of faults and their effectiveness in an unambiguous way, and it is also insightful to describe DPA resistance similar to the probing model [ISW03].

Second, we present two different approaches that, starting from the algorithmic description, allow to specify circuits that mitigate SIFA and DPA. The first approach relies on finding descriptions of ciphers that restrict basic circuits to permutations which only operate on an incomplete set of shares. In particular, those permutations are either linear or variants of the Toffoli gate [Tof80], the simplest invertible non-linear function. This strategy allows for fault detection at the end of a cipher, e.g., by means of redundant computation and comparison of the outputs. We then show that masked 3-bit, 4-bit, and 5-bit S-boxes can be built using Toffoli gates as their only non-linear component, thus offering protection against SIFA. A similar strategy can be applied to the AES S-box; however, for AES, we require fault detection at the output of the AES S-box. We verified the correctness of the masking of our Toffoli-based circuits for S-boxes of Keccak and AES using `maskVerif`, a tool for formal verification of masking schemes [BBD+15]. The second approach does not restrict basic circuits to be permutations and can thus be applied more broadly to add SIFA-protection to arbitrary existing circuits. It works by adapting the error-detection circuit for more fine-grained detection and can be generalized to cover multi-fault SIFA, albeit at a higher implementation cost.

Related Work. Potential mitigations against SIFA are already discussed in the paper introducing SIFA for masked ciphers with fault countermeasures [DEK+18]. One strategy considered there is to move from error detection to error correction, such as a majority voting. This has the effect that more than one fault is needed, since a single fault can always be corrected. This rough concept is developed further and brought into practice by several papers [BKHL19, SJR$^+$19, SRM19], which have been published around the same time or slightly after the preprint of the present paper.

The paper of Breier et al. [BKHL19] proposes a countermeasure based on error-correcting codes that can be implemented with error-correcting hardware gates. Shahmirzadi et al. [SRM19] investigate how error-correcting codes can be correctly embedded in a cryptographic implementation and show that their construction provides practical advantages over simple majority voting. Contrary to the two papers mentioned before, the Transform-and-Encode framework by Saha et al. [SJR$^+$19] proposes an instantiation that combines masking with majority voting as an error-correcting code. Their strategy has the benefit that the error correction is only applied at the output of the S-boxes.

In contrast to the above-mentioned countermeasures using error correction, our strategy only requires error detection. We propose to either use masking schemes based on the Toffoli gate or transform implementations of arbitrary masking schemes. By doing this in combination with suitable error detection, we ensure that the effect of a fault never depends on a native (unmasked) value. In most cases, the detection of the error can be done once on the primitive level (e.g., after one block cipher execution, or execution of a cryptographic permutation). Hence, it even can be implemented in a time-redundant manner, or as an encrypt-decrypt strategy. Therefore, we get an overhead of roughly two, either in space or time, compared to just a masked implementation that is usually needed for protection against side-channel attacks anyway. In contrast, the computational overhead of majority voting is at least three, with more overhead for more sophisticated error correction strategies, as shown by Shahmirzadi et al. [SRM19].

Following the preprint of this paper, Ramezanpour et al. [RAD19] proposed a different way of achieving independence of a fault effect from the native value. As an example, they give an implementation based on a threshold implementation [NRR06, NRS11] of Canright's AES S-box [Can05], where they introduce additional shares and additional computations. This results in an overhead of a factor of 2 compared to a standard threshold implementation in their FPGA implementation [RAD19]. Moreover, if protection against differential fault attacks is also needed, then this cost will increase further. In contrast to this, our scheme also works with masking based on two shares and has a low overhead, allowing much more efficient implementations in practice.

Another strategy that may protect against fault attacks, including SIFA, are approaches based on actively secure multi-party computation protocols like CAPA [RDB$^+$18], as long as a fault does not affect all parties. However, such methods are quite expensive. As pointed out by the authors of CAPA [RDB$^+$18], their implementations are to be considered as a proof-of-concept and are too costly to be used in practice. For example, an implementation of the AES S-box using three shares and one MAC key requires 156 bytes of randomness per S-box evaluation.

Outline. We start with a description of our abstraction layer and discuss how we model fault and side-channel attacks on it in Section 2. Then, we show how to describe some ciphers only relying on incomplete permutations as basic circuits in Section 3. Afterwards, we describe a circuit for the AES that can withstand single-fault SIFA and first-order DPA in Section 4. Section 5 deals with the protection of arbitrary circuits. In Section 6, we verify the correctness of the circuits for the S-boxes of AES and Keccak using `maskVerif` and discuss considerations when implementing the circuits in software or hardware.

2 The Circuit Abstraction Layer and Fault Model

In this section, we first briefly recall the basics of masking and Statistical Ineffective Fault Attacks (SIFA). Then, we define an abstract circuit model and the corresponding fault model that we use as a tool for building hardware or software cipher implementations that offer resistance against SIFA and side-channel attacks. One major purpose of the circuit abstraction layer is that it allows us to define our fault model in a clean and tidy way. In the following sections, we will make use of the term *cipher*. By the term *cipher*, we mean block ciphers, tweakable block ciphers, or cryptographic permutations.

2.1 On Masking

Masking is an algorithmic-level countermeasure against power analysis attacks but can also be relevant in preventing SIFA. A cipher takes as input an array of variables (often binary) and returns a similar array as output. An algorithm describes how to compute the latter from the former.

In a masked version of a cipher, each variable at the input is expanded to $d+1$ *shares* such that the sum of the shares (assuming the variables are elements of an additive group) equals the variable, which we call *native*. The output has the same expansion and the algorithm specifies how to compute the expanded output from the expanded input, in a deterministic way. In implementations, this allows computing the cipher on a state by encoding each of its variables in $d+1$ shares where the sum equals the native variable and summing the output shares to obtain the native output. This allows choosing d shares for each variable for each computation randomly, providing protection against differential power analysis. Namely, individual share variables are independent of native variables and if the masking is well designed, this is even the case for up to d share variables.

2.2 SIFA on Ciphers

Consider a block cipher implementation that counteracts differential fault attacks by performing each encryption n times and only releases a final ciphertext if the results of all redundant computations match up. Further, assume an attacker who targets one out of the n redundant encryptions, is capable of repeatedly injecting one fault in one of its intermediate operations and observes the final ciphertext unless it is suppressed by the countermeasure. The timing or location of the fault injection should be chosen such that the affected state bits have a non-linear dependency on a small number of round key bits; this is typically the case in the penultimate round of a block cipher.

If the attacker performs multiple such faulted encryptions for different (possibly unknown) block cipher inputs, the attacker will eventually observe correct ciphertexts where the injected fault was ineffective. In the most simple case, consider a fault injection that sets certain state bits to zero that were already zero before the fault injection. In these cases, the specific values of certain state bits rendered the injected fault ineffective, which leads to an overall correctly computed ciphertext that is observed by the attacker.

This filtered set of correct ciphertexts will, when partially decrypted using the correct partial key guess, typically show this non-uniform (biased) distribution of the affected state bits. When sticking to the simple example from before, certain state bits should show a strong bias towards zero.

We can use this property for key recovery as follows: For each possible 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, e.g., explains SIFA against the AES block cipher [DEK+18].

In a similar spirit, SIFA can also be mounted on the input of block ciphers or the initialization phase of many AEAD schemes by instead collecting the cipher inputs that lead to an ineffective fault injection in the second round. For more concrete examples, we refer to [DMMP18] who show how SIFA can be mounted on the AEAD schemes Keyak and Ketje.

In case of additionally masked implementations, the attack procedure is almost the same except the fault should now target an intermediate value, e.g., a single share, within a non-linear operation (S-box) in the targeted round. For a more detailed explanation, we refer to Section 2.6 or [DEG+18].

2.3 Definition of The Circuit Abstraction Layer

We propose an abstraction layer between the algorithmic description of a (masked) cipher or permutation and the hardware or software implementation and present our formalism here. At the circuit abstraction level, we break up the deterministic algorithmic description into a number of interconnected *circuits*.

By a circuit, we mean a fully specified deterministic function taking as input an array of input variables and returning as output an array of output variables. Trivially, the algorithm of a (masked) cipher itself defines a circuit. More interestingly, we can break up that circuit into a number of interconnected *sub-circuits* that take care of all the processing. The composite circuit that a sub-circuit is part of is called its *super-circuit*. The variables of the super-circuit and its sub-circuits are related in the following ways:

- Each input variable of a sub-circuit is either an input variable of its super-circuit or the output variable of another sub-circuit.
- Variables in a super-circuit that are neither super-circuit input nor output variables are called *intermediate* variables.
- We consider duplication a form of processing and hence, any super-circuit input or intermediate variable propagates to at most one sub-circuit input or to the super-circuit output.

This can be applied recursively, where every sub-circuit can at the same time be a super-circuit with its own sub-circuits.

Let us illustrate this with an example: non-masked AES-128. Its circuit takes as input a 128-bit key and a 128-bit plaintext and returns as output a 128-bit ciphertext. Two obvious sub-circuits are the key schedule and the datapath. The former has the 128-bit key as input and a 1408-bit expanded key as output. The latter has the 128-bit plaintext and the expanded key as input and the 128-bit ciphertext as output. The datapath circuit can be split in 11 sub-circuits: one for each round and an initial round key addition circuit. The input of each round sub-circuit is the output of another circuit and a round key taken from the expanded key. The circuits for the first 9 rounds are identical, we say they are 9 instances of the same *circuit class*. The round circuit class can be split into 4 sub-circuits: SubBytes, ShiftRows, MixColumns and AddRoundKey. One may merge ShiftRows with SubBytes or MixColumns when targeting implementations where this step does not represent processing. A SubBytes circuit splits naturally into 16 sub-circuits of the same class, representing the S-box.

In this paper, we will use the following formalism to specify circuits. We specify the input variables and their type, the output variables and their type and how to compute the output variables from the input variables. Unless stated otherwise, variables are binary. Circuits can be defined from scratch with simple operations such as addition and multiplication of variables, or in terms of sub-circuits. We call the former a basic circuit and the latter a composite circuit. Circuit 1.a provides an example of a basic circuit with three binary input variables and two binary output variables. A circuit can be used as a sub-circuit in the specification of a composite circuit. An example of a composite circuit with four binary input variables and two binary output variables is given in Circuit 1.b.

(1.a) Basic circuit	**(1.b)** Composite circuit
Name: ExampleCircuit1	Name: ExampleCircuit2
Input: (a, b, c)	Input: (a, b, c, d)
$a \leftarrow b \odot c$	$(a, b) \leftarrow$ ExampleCircuit1(a, b, c)
$b \leftarrow c \oplus a$	$(a, b) \leftarrow$ ExampleCircuit1(d, a, b)
Output: (a, b)	Output: (a, b)

The composite circuit is specified by two sub-circuits where the processing on the input variables is based on their location in the input array. When naming intermediate variables, one can make use of the fact that any variable shall be used exactly once. This implies that once a variable has occurred as the input of a sub-circuit, its name becomes available for another intermediate variable. In case we want to use a variable twice or more, we can put a cloning circuit. This is a circuit cloning variable a to b: $(a, b) \leftarrow$ Clone(a), or to a, b and c: $(a, b, c) \leftarrow$ Clone(a). In our convention, a composite circuit must use all output variables of its sub-circuits as outputs. If one wishes to omit a variable, this can be specified explicitly with a sinkhole circuit: Sinkhole(a) is a simple circuit taking one variable and returning no variables.

We refer to circuits that have as many input variables as output variables (and of the same type) as *transformative* circuits. For these circuits we have a simplified convention. We specify them operating on a *state* which is used for both input and output. When they are used in the specification of a composite circuit, we omit the arrow and output variable.

At the circuit abstraction level, we see computation as the application of a particular input at the circuit's input and the observation of the result at its output. Furthermore, such circuits can be injective, surjective, both or neither.

Definition 1. A circuit (class) is injective if for every input it returns a different output. It is surjective if for all outputs there is at least one input. We call a circuit (class) that is both injective and surjective a permutation circuit (class).

Clearly, a permutation circuit can be modeled as a transformative circuit.

2.4 Fault Model

We model faults at the level of a circuit during computation. In the absence of faults, the output of a circuit of a given class is fully determined by its input. A circuit fault is simply any deviation from this.

Definition 2. A circuit fault during a computation is a deviation of the circuit instance from its circuit class. Namely, the fault modifies the circuit in such a way that it could return for at least one input an output that does not correspond with the one prescribed by the circuit class.

This definition covers a wide range of faults, including bit flips, or set-to-0 or set-to-1 faults of input, output, or intermediate variables, but also modifications of entries in lookup tables. Circuit faults are abstract and the mapping to physical faults that occur in actual implementations is often non-trivial. The hardware implementation of a cipher circuit that has many sub-circuit instances of the same class may re-use the same combinatorial hardware for all those instances. A permanent fault in such a hardware would correspond to a circuit fault in all circuit instances that it is used to implement. Similarly, a faulted entry in the AES S-box lookup table implies circuit faults in all S-box circuits of the AES circuit. Although faults often occur in implementations of circuit classes, at circuit abstraction level we see circuit faults in circuit instances. This is more general as faults may be induced for single executions of a program sequence or combinatorial circuit, or different faults may be induced for different instances.

A circuit fault does not necessarily imply a faulty circuit output. For example, a single faulted entry in an AES lookup table only leads to a faulty circuit output if the circuit input hits that entry. A stuck-at-0 fault affecting an input variable is not visible at the output if the variable is 0 anyway. For this reason, we define the concept of fault effectiveness.

Definition 3. A circuit fault is effective during a computation if it leads to a faulty circuit output.

When protection against faults is a concern, one typically performs redundant computations. At circuit level, this can be done by feeding it with variables that satisfy some conditions. In duplication, this can be done by two circuits of the same class that operate on input variable arrays set to the same value. In the absence of faults, this will also be the case for the outputs. Another possibility is a single circuit where the input variables satisfy some linear relation and in the absence of a fault, the output variables will satisfy the same. These circuits propagate a kind of *redundancy condition* that, if not satisfied, implies a fault must have occurred. Detection of faults can be done with a circuit as well. We call this a fault detection circuit. It (typically) simply propagates the input variables unchanged to the output but has an additional binary output, called *fault alert*, which is false when the redundancy condition is satisfied and true otherwise. This fault detection circuit may opt to use only the output variables of the cipher circuit, or it may use duplicates of intermediate variables as well.

2.5 Masking in Circuits

We speak of a *masked* circuit when it corresponds to a masked cipher. It operates on share variables and preserves the property that learning d shares does not give information on native variables. In a similar way as the probing model [ISW03], we model side-channel attacks by allowing an attacker to probe all associated variables and to observe all computations of certain sub-circuits. The observation of a single sub-circuit does not give information about native variables only if the sub-circuit is *incomplete*.

Definition 4. A sub-circuit in a masked circuit is incomplete if the input variables do not include all shares of a single native variable.

For linear functions, such a partition into incomplete sub-circuits can be done quite easily as shown in Circuit 2.a.

(2.a)

```
Name: SharedXor
State: $(a_0, a_1, b_0, b_1)$
XorFirst$(a_0, b_0)$
XorFirst$(a_1, b_1)$
```

(2.b)

```
Name: XorFirst
State: $(a, b)$
$a \leftarrow a \oplus b$
```

However, if we just consider one Boolean And, such a partition into incomplete sub-circuits is more complex. Hence, many papers dealing with masking aim to find efficient masked implementations for the Boolean And [ISW03, BBP+17, RBN+15, DRB+16, GIB18, GM17, GMK16, BDF+17]. If we just focus on the sharing of an AND, $c = a \odot b$, using 2 shares, such a sharing requires the addition of a resharing variable R. This is needed to ensure that the shares c_0 and c_1 are each independent of the native value of c. The resharing variable R is a circuit input. It may be derived from a dedicated random number generator or from another unrelated calculation, e.g., as shown in Changing of the Guards [Dae17]. A possible partition of masked AND into incomplete sub-circuits is then given in Circuit 3.a. This definition requires a lot of cloning. We can alternatively use sinkholes as in Circuit 3.b.

(3.a) With cloning

```
Name: SharedAnd
Input: $(a_0, a_1, b_0, b_1, \text{R})$
$(\text{R}, \text{R}') \leftarrow$ Clone(R)
$(a_0, a_0') \leftarrow$ Clone($a_0$)
$(a_1, a_1') \leftarrow$ Clone($a_1$)
$(b_0, b_0') \leftarrow$ Clone($b_0$)
$(b_1, b_1') \leftarrow$ Clone($b_1$)
$(c_0) \leftarrow$ AndXor($a_0, b_1, \text{R}$)
$(c_0) \leftarrow$ AndXor($a_0', b_0, c_0$)
$(c_1) \leftarrow$ AndXor($a_1, b_0', \text{R}'$)
$(c_1) \leftarrow$ AndXor($a_1', b_1', c_1$)
Output: $(c_0, c_1)$
```

```
Name: AndXor
Input: $(a, b, c)$
$d \leftarrow a \odot b$
$c \leftarrow d \oplus c$
Output: $(c)$
```

(3.b) With sinkholes

```
Name: SharedAnd
Input: $(a_0, a_1, b_0, b_1, \text{R})$
$(c_0, \text{R}) \leftarrow$ Clone(R)
AndXor1($a_0, b_1, c_0$)
AndXor1($a_0, b_0, c_0$)
$(c_1, \text{R}) \leftarrow$ Clone(R)
AndXor1($a_1, b_0, c_1$)
AndXor1($a_1, b_1, c_1$)
Sinkhole($\text{R}, a_0, a_1, b_0, b_1$)
Output: $(c_0, c_1)$
```

```
Name: AndXor1
State: $(a, b, c)$
$d \leftarrow a \odot b$
$c \leftarrow d \oplus c$
```

2.6 SIFA on Masked Circuits

Implementing a masked cipher based on a circuit with incomplete sub-circuits and with fault countermeasures such as duplication at cipher level with a fault detection circuit at the end of the cipher circuit is not sufficient to prevent SIFA. In this section we explain why. We assume a SIFA attacker that can make many computations but is limited to a circuit fault in a single sub-circuit (including fault detection circuits) during each computation. The success of this attack relies on whether the behavior of the fault alert of a detection circuit depends on native variables.

To see that this is still possible in the presence of only incomplete sub-circuits, we give an example. Consider the single masked AND-gate with 2 shares (cf. Circuit 3.b). We see that every input share is an input to two ANDXOR circuits and is combined with share 0 of a native variable in one of them and share 1 in the other. For instance, faulting a_0 to $a_0 \oplus 1$ at the input of ANDXOR1(a_0, b_1, c_0) propagates to a_0 in ANDXOR1(a_0, b_0, c_0). It will flip c_0 in ANDXOR1(a_0, b_1, c_0) iff $b_1 = 1$ and c_0 in ANDXOR1(a_0, b_0, c_0) iff $b_0 = 1$. The result is that it will flip c_0 an odd number of times iff $b_0 \oplus b_1 = b = 1$. Hence it will propagate to c_0 and hence also the native variable c if $b = 1$ and not if $b = 0$. Resistance against SIFA requires us to construct circuits that ensure that the propagation of circuit faults in sub-circuits to the cipher circuit output is independent of native variables.

One condition that we use to achieve this is that each sub-circuit is incomplete. If a circuit is not incomplete, faulting such a sub-circuit might have fault effects that depend on native values. Second, we have to ensure that the circuit is built in such a way that the propagation of the fault effect does not lead to ineffective faults depending on native values. We have essentially two options to achieve this:

1. Build circuits of incomplete sub-circuits where an effective fault at the output of a single sub-circuit can never become ineffective at the output of the cipher circuit (Section 3).

2. Build a fault detection circuit that catches effective faults at the output of sub-circuits before they can become ineffective (Section 5).

Although a cipher circuit can be built from sub-circuits in a recursive way with multiple layers, in the remainder we will consider only a single level of sub-circuits. We will call these sub-circuits *basic circuits*. We then consider a single fault per execution of a cipher as a circuit fault in a single basic circuit instance. Furthermore, in a first-order side-channel attack (e.g., first-order DPA), we allow an attacker to observe a single basic circuit instance. This means that an attacker has knowledge about all variables associated to this basic circuit and the computation done within it.

2.7 On Detection Circuits

We consider detection circuits to be part of our circuit and hence, they can also be a target in fault attacks. Thus, in general, detection circuits are also incomplete, meaning that they have to operate on an incomplete set of shares. A simple solution to ensure this is to duplicate shares at the input of the cipher and to do the redundant computations on each set of shares. Then, detection circuits can check the consistency directly on duplicated shares.

In the next sections, we also present strategies protecting against single fault SIFA that only require fault detection on native values. This means that redundant computations do not have to be performed on duplicated sets of shares. Instead, only the native input values to a cipher can be duplicated and different randomness can be used to share the duplicated native values. However, when checking native values for faults, care has to be taken. If detection circuits operate on native values, faults on these circuits may leak parts of the native values. This is not a problem if these native values are not secret, e.g., when detection circuits operate on the ciphertext output of a cipher. However, if the compared native values have to be kept secret, care has to be taken that the shares representing the native value are never combined within this circuit. In addition, the detection circuits have to be placed in a manner so that faults on them do not reveal the native values.

3 Ciphers from Incomplete Permutation Circuits

In this section, we investigate how we can implement ciphers so that they are protected against single-fault SIFA. The heart of our strategy is to split a cipher into basic circuits that are permutations and that are incomplete. To do this, we use constructions common in the field of reversible computing [Lan61, Ben73, Tof80]. In particular, we use the Toffoli gate [Tof80] and related constructions as essential basic circuits.

3.1 The High-level Strategy

In this section, we aim to implement strategy 1 of Section 2.6. We do this by building a cipher circuit out of incomplete permutation basic circuits.

In this way, any single circuit fault in a basic circuit that is effective for that basic circuit is also effective for the cipher circuit. This follows from the following lemma and corollary.

Lemma 1. *Any composite circuit built from permutation sub-circuits is itself a permutation circuit.*

Proof. The circuit has at least one sub-circuit A with all its input variables also input variables of its super-circuit. Now write the super-circuit as the serial composition of two circuits:

- The first circuit applies A to the input variables and returns the corresponding output variables. The remaining variables are just copied from input to output.
- The second circuit is the super-circuit with the circuit A replaced by the identity.

The first circuit is a permutation as A is a permutation. We can iteratively apply this trick to the second circuit until it only contains a single sub-circuit. In this way we write the super-circuit as a series of invertible circuits. □

Corollary 1. *In a composite circuit built from permutation sub-circuits, any fault at the output of a sub-circuit will propagate to the output of the super-circuit.*

Proof. We can use the decomposition in circuits in the proof of Lemma 1 to split the super-circuit in two permutation circuits where the faulty output variables of a sub-circuit are input variables to one of the two permutation circuits. Therefore, the fault will propagate to the output. □

Thanks to Corollary 1, we can limit fault detection to the cipher's output.

Corollary 2. *Assume we have a composite circuit built from redundant incomplete permutation basic circuits. Further, assume that redundant circuits are implemented so that they do not influence each other. In addition, the redundant output shares are checked with the help of incomplete detection circuits. Then, the resulting composite circuit withstands single fault SIFA.*

A single fault can always only target a single incomplete basic circuit. This fault can be effective at the output of the targeted incomplete basic circuit or not. However, since the basic circuit is incomplete, the effectiveness of a fault never depends on a native value. In the case that the fault shows an effect, Corollary 1 ensures that the fault propagates through the circuit and is detected at the output by incomplete detection circuits. Also from this event, an attacker cannot infer any information, since it is also independent of native values. Thus, the effectiveness of a fault is independent of native values and hence, cannot be exploited by SIFA.

If we want to protect (tweakable) block ciphers, also the (tweak) key-schedule has to be a permutation (which it typically is), that is split in incomplete permutation basic circuits. In this case, we consider the last round-key and last tweak together with the ciphertext as output of the cipher that has to be checked for faults.

Typically, a cipher consists of a sequence of rounds and the round function has a linear and a non-linear layer. We consider only ciphers where both layers are permutations.

For the linear layer, a split in incomplete permutation basic circuits is straightforward. In particular, a linear function $y = f(x)$ can be split in $d + 1$ incomplete basic circuits that each operate on a single share of the native state variables. If f is a permutation, then so are the basic circuits computing $y_i = f(x_i)$. Furthermore, a single fault always causes a change in the native value of y.

For the non-linear layer sub-circuit, a split in incomplete permutation sub-circuits is less trivial. Typically, the non-linear layer consists of the parallel application of a non-linear S-box to subsets of the state variables. Hence the challenge is to build a circuit for the masked S-box in terms of incomplete permutation basic circuits.

We do this by constructing masked S-box circuits that are permutations using basic circuits of the inherently bijective Toffoli gate (Section 3.2) and variants. We follow a two-stage approach: first express the (unmasked) S-box in terms of Toffoli gates and then build a circuit of the masked Toffoli gate using incomplete Toffoli-gate basic circuits.

As a consequence of our design strategy, we end up with a round function circuit where each basic circuit is incomplete and a permutation on the shared state. This implies that it preserves uniformity of the sharing and hence, no fresh randomness is required during the rounds for realizing first-order DPA secure circuits.

3.2 Incomplete Permutation Basic Circuits

In the following, we present our permutation basic circuits that we will use to realize circuits for S-boxes. In essence, we need three different basic circuits. The first one is the Toffoli gate [Tof80], a non-linear 3-bit permutation. We denote it by $p_T(a, b, c)$ and define it in Circuit 4.a. For brevity in our S-box constructions, we also define a permutation $p_\chi(a, b, c)$ in Circuit 4.b that is a close variant of it. In addition, we need the basic circuit XORFirst(a, b) from Circuit 2.b to realize some S-boxes.

(4.a) Toffoli gate	**(4.b)**
Name: p_T	Name: p_χ
State: $\{a, b, c\}$	State: $\{a, b, c\}$
$\mathrm{T} \leftarrow b \odot c$	$\mathrm{T} \leftarrow \bar{b} \odot c$
$a \leftarrow a \oplus \mathrm{T}$	$a \leftarrow a \oplus \mathrm{T}$

In the first step, we build circuits out of these basic circuits. Those circuits are masked versions of the basic circuits and will be used as building blocks for the S-boxes. First, let us have a look at a circuit for the 2-share masked Toffoli gate shown in Circuit 5.a, that we will refer to as $p_{TS}(a_0, a_1, b_0, b_1, c_0, c_1)$. As can be seen in Circuit 5.a, all p_T sub-circuit instances are incomplete.

(5.a) Masked Toffoli gate	**(5.b)**
Name: p_{TS}	Name: $p_{\chi S}$
State: $\{a_0, a_1, b_0, b_1, c_0, c_1\}$	State: $\{a_0, a_1, b_0, b_1, c_0, c_1\}$
$p_T(a_0, b_0, c_1)$	$p_\chi(a_0, b_0, c_1)$
$p_T(a_0, b_0, c_0)$	$p_\chi(a_0, b_0, c_0)$
$p_T(a_1, b_1, c_1)$	$p_T(a_1, b_1, c_1)$
$p_T(a_1, b_1, c_0)$	$p_T(a_1, b_1, c_0)$

Thanks to the fact that the basic circuits are permutations on the state, any circuit fault in a single p_T sub-circuit instance that is effective at its output will also be effective at the output of the super-circuit. Moreover, any effective fault due to a single sub-circuit fault can at most affect a single share per variable, and will hence result in a faulty native variable at that point. Thanks to the correctness of sharing, this fault will propagate to the super-circuit output.

These observations are also true for the masked version $p_{\chi S}$ of p_χ (Circuit 5.b).

Next, we will show how to build circuits of two-share masked S-boxes using the basic circuits introduced here. For the sake of completeness, we note that the same properties can be achieved in a similar form for three-share threshold implementation as shown in Appendix A.

3.3 3-bit S-boxes

Recently, 3-bit S-boxes have become more prominent with their usage in PRINTcipher [KLPR10], LowMC [ARS+15], or XOODOO [DHVV18]. As a representative of these S-boxes, we focus on the protection of the 3-bit χ-layer [Dae95, DHVV18]. The mapping χ operates on circular arrays of bits and it complements all bits that have the pattern 01 in 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 will refer to n-bit χ as χ_n.

Table 1: The 6 classes of quadratic 4-bit S-boxes [BNN+15] expressed in terms of p_T and p_χ.

`0123456789ABDCFE`	`0123456789CDEFAB`	`0123457689CDEFBA`
Name: Q^4_4 State: $\{a, b, c, d\}$ $p_T(d, a, b)$	Name: Q^4_{12} State: $\{a, b, c, d\}$ $p_T(b, a, c)$ $p_T(c, a, b)$	Name: Q^4_{293} State: $\{a, b, c, d\}$ $p_T(d, b, c)$ $p_T(b, a, c)$ $p_T(c, a, b)$

`0123456789BAEFDC`	`012345678ACEB9FD`	`0123458967CDEFAB`
Name: Q^4_{294} State: $\{a, b, c, d\}$ $p_T(c, a, b)$ $p_T(d, a, b)$ $p_T(d, a, c)$	Name: Q^4_{299} State: $\{a, b, c, d\}$ $p_T(b, a, c)$ $p_T(c, a, b)$ $p_T(b, a, c)$ $p_T(c, a, d)$ $p_T(d, a, c)$	Name: Q^4_{300} State: $\{a, b, c, d\}$ XorFirst(b, a) XorFirst(c, a) $p_T(a, b, c)$ $p_T(b, a, c)$ $p_\chi(c, b, a)$

Daemen et al. [DHVV18] pointed out that it is possible to compute χ_3 in-place in its registers as a sequence of three Toffoli gates. This immediately yields a circuit for two-share masked χ_3 in terms of permutation sub-circuits:

Name: Masked_chi3
State: $\{a_0, a_1, b_0, b_1, c_0, c_1\}$
$p_{\chi S}(a_0, a_1, b_0, b_1, c_0, c_1)$
$p_{\chi S}(b_0, b_1, c_0, c_1, a_0, a_1)$
$p_{\chi S}(c_0, c_1, a_0, a_1, b_0, b_1)$

Recall from Section 3.2 that $p_{\chi S}$ is just a composite circuit and that its basic circuits are p_T or p_χ. Still, a fault effect stemming from a single basic circuit shows an effect in the native values at the S-box output.

3.4 4-bit S-boxes

The construction and design of 4-bit S-boxes has been intensively studied in literature.

Using affine equivalence, De Cannière [De 07] partitions 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 [BNN+15].

As shown by Bilgin et al. [BNN+15], 144 cubic classes can be constructed by iterating the S-boxes of the quadratic classes separated by affine layers up to 3 times. This covers many prominent S-boxes, e.g., the S-boxes used in Noekeon [DPVR00] and Present [BKL+07], but also several of the 16 S-boxes observed to be “optimal” by Leander and Poschmann [LP07]. 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.

Using Table 1, Circuit 5.a, and Circuit 5.b, it is straightforward to build circuits for two-share masked versions for 144 out of the 295 cubic classes of S-boxes [BNN+15] from incomplete permutation basic circuits. 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 [GM12] give optimal implementations (with respect to a certain set of reversible gates) for all 4-bit permutations using at most 15 reversible gates. However, note that the set of reversible gates used may differ from the basic circuits p_T and p_χ used in this section and hence, we consider the exploration of this as future work.

3.5 5-bit S-boxes

Shende et. al [SPMH03] show that every permutation (S-box) with an odd number of inputs can be implemented using reversible gates by using at most one additional variable. However, as we will see next, the need for this additional variable forces us to deviate from the strategy that each basic circuit is a permutation. In particular, we will make use of Sinkhole(r) and $(r_1, r_0) \leftarrow$ Clone(r_0) introduced in Section 2.3.

In this work, we only focus on the 5-bit S-box χ_5, which has several prominent uses. For instance, it is used in the KECCAK-p permutations inside KETJE [BDP+16a], KEYAK [BDP+16b], KRAVATTE [BDH+17], and KECCAK [BDPV11] (SHA-3). Moreover, χ_5 is also the core of ASCON's S-box [DEMS16]. We base our circuit for two-share masked χ_5 on an implementation of χ_5 [DEMS16] with input bits a, b, c, d, and e and an intermediate variable r, as shown in Circuit 6.a.

(6.a) χ_5

```
Name: $\chi_5$
State: $\{a, b, c, d, e\}$
$(r) \leftarrow$ AndNot$(a, e)$
$p_\chi(a, b, c)$
$p_\chi(c, d, e)$
$p_\chi(e, a, b)$
$p_\chi(b, c, d)$
XorFirst$(d, r)$
Sinkhole$(r)$
```

```
Name: AndNot
Input: $(b, c)$
$a \leftarrow b \odot \bar{c}$
Output: $(a)$
```

To provide an implementation of χ_5 that withstands single-fault SIFA, we again rely on $p_{\chi S}(a_0, a_1, b_0, b_1, c_0, c_1)$ as a building block. We introduce additional input variables r_0 and r_1, which have to be initialized with random values such that $r_0 \oplus r_1 = 0$. This allows us to argue the security of the following scheme in Circuit 7.a.

(7.a) Masked χ_5 with constraints

```
Name: Masked_chi5_v1
Input: $\{a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1, r_0, r_1\}$
$p_{\chi S}(r_0, r_1, e_0, e_1, a_0, a_1)$
$p_{\chi S}(a_0, a_1, b_0, b_1, c_0, c_1)$
$p_{\chi S}(c_0, c_1, d_0, d_1, e_0, e_1)$
$p_{\chi S}(e_0, e_1, a_0, a_1, b_0, b_1)$
$p_{\chi S}(b_0, b_1, c_0, c_1, d_0, d_1)$
XorFirst$(d_0, r_0)$
XorFirst$(d_1, r_1)$
Sinkhole$(r_0, r_1)$
Output: $\{a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1\}$
```

(7.b) Masked χ_5 with cloning

```
Name: Masked_chi5_v2
State: $\{a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1, r_0\}$
$(r_1, r_0) \leftarrow$ Clone$(r_0)$
$p_{\chi S}(r_0, r_1, e_0, e_1, a_0, a_1)$
$p_{\chi S}(a_0, a_1, b_0, b_1, c_0, c_1)$
$p_{\chi S}(c_0, c_1, d_0, d_1, e_0, e_1)$
$p_{\chi S}(e_0, e_1, a_0, a_1, b_0, b_1)$
$p_{\chi S}(b_0, b_1, c_0, c_1, d_0, d_1)$
XorFirst$(d_0, r_0)$
XorFirst$(d_1, r_1)$
Sinkhole$(r_1)$
```

We end up with a construction (Circuit 7.a) which is the repeated application of permutation $p_{\chi S}$ on 12 bits of the state a_0 to r_1. Due to this iterative construction, a fault that has an effect on any native output variable of one $p_{\chi S}$ would have an effect on the native output variables of the whole circuit if r_0 and r_1 would be part of the output. However, r_0 and r_1 end in Sinkhole(r_0, r_1). Hence, we have to show that this never leads to an effect of a fault disappearing.

As can be seen in Circuit 7.a, d_0 and d_1 are only written in the basic circuits XorFirst(d_0, r_0) and XorFirst(d_1, r_1). Furthermore, the calculation of r_0 and r_1 is independent of d_0 or d_1. As a consequence, a fault in a single basic circuit that happens before the execution of XorFirst(d_0, r_0) and XorFirst(d_1, r_1) can never have an effect on the shares of d and r at the same time. Hence, the basic circuits XorFirst(d_0, r_0) and XorFirst(d_1, r_1) never cancel the effect of a fault on a single basic circuit, and effects of faults on the native value of r carry over to d.

In a similar spirit as Sugawara for AES [Sug19], 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 Circuit 7.b 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 Circuit 7.b 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 and that the masking is indeed correct using `maskVerif` (cf. Section 6.1).

4 AES S-box from Incomplete Permutation Basic Circuits

So far, the main focus has been on S-boxes that have a rather simple and compact description over $\mathbb{F}_2$. However, there exist S-boxes with complex descriptions over $\mathbb{F}_2$, but more concise descriptions over larger binary fields, i.e., $\mathbb{F}_{2^n}$. Hence, we will apply our method to the representation of the S-box over $\mathbb{F}_{2^n}$. The most prominent example that falls into this category is the S-box of AES [DR02]. Building on Canright's description [Can05], we can derive a description that is better suited for our proposed countermeasure.

Figure 1 shows Canright's description of the AES S-box just using reversible computations, basically transforming the idea of Sugawara [Sug19, Figure 8] from 3-shared to 2-shared masking. This can be done by replacing all $\mathbb{F}_{2^n}$ multiplications in Canright's description by Toffoli gates operating in $\mathbb{F}_{2^n}$ using an additional input that is set to 0. To distinguish it from the binary Toffoli gate defined in Section 3.2, we denote a Toffoli gate over $\mathbb{F}_{2^n}$ by $p_T^n(a, b, c)$ with $a, b, c \in \mathbb{F}_{2^n}$. In the following, we denote multiplication and addition over $\mathbb{F}_{2^n}$ by $\cdot$ and $+$, respectively:

Name: p_T^n
State: $\{a, b, c\}$
$a \leftarrow a + b \cdot c$

We can also define a masked version of the Toffoli gate over $\mathbb{F}_{2^n}$ in a similar way as in Circuit 5.a. Here, $a_0, b_0, c_0, a_1, b_1, c_1 \in \mathbb{F}_{2^n}$ denote the shares of a, b, c and the shared version of p_T^n is denoted by $p_{TS}^n(a_0, a_1, b_0, b_1, c_0, c_1)$:

Name: p_{TS}^n
State: $\{a_0, a_1, b_0, b_1, c_0, c_1\}$
$p_T^n(a_0, b_0, c_1)$
$p_T^n(a_0, b_0, c_0)$
$p_T^n(a_1, b_1, c_1)$
$p_T^n(a_1, b_1, c_0)$

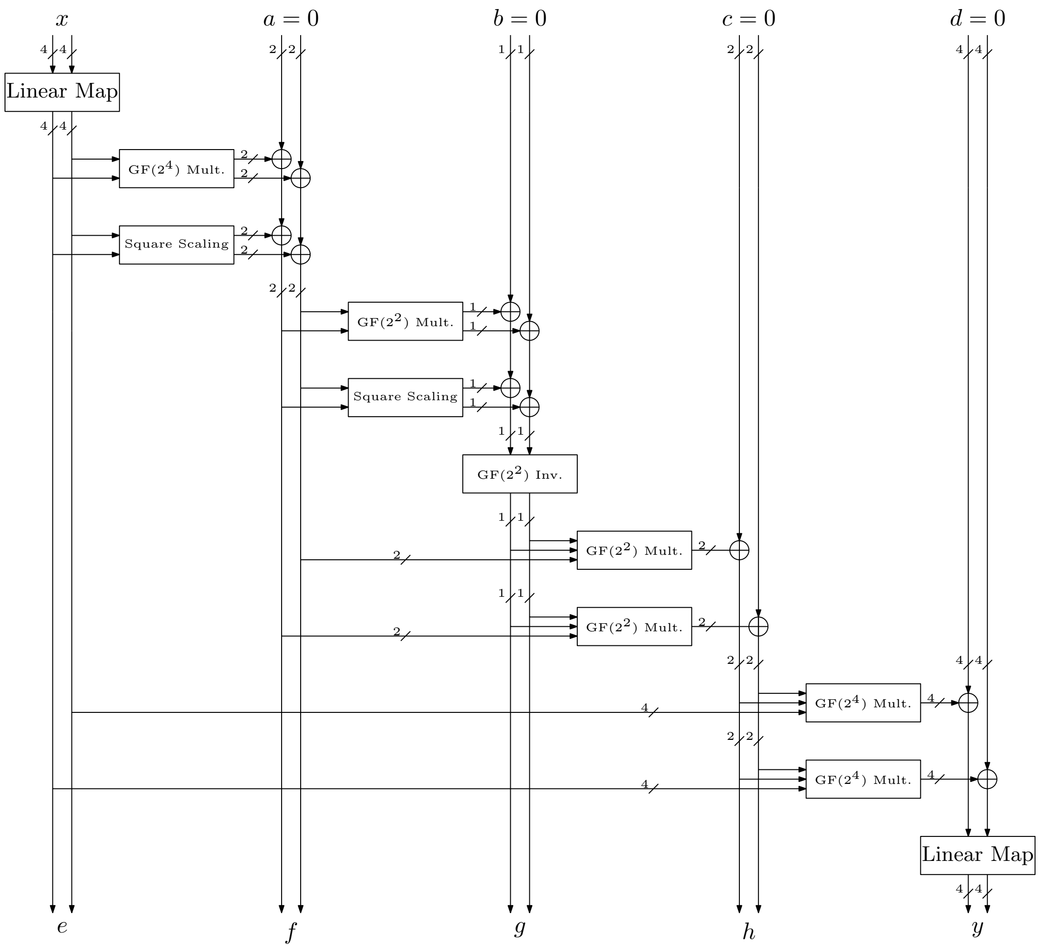


Figure 1: Description of AES S-box ($y = S(x)$) relying on invertible computations.

The arguments for the security and fault propagation of p_{TS} are analogous to Section 3.2. Again, each p_T^n is incomplete and hence, the effect stemming from faulting a single instance can never depend on a native value. Furthermore, each fault effect caused by a fault on a single instance of p_T^n is then visible in the native value. Since only the shares of a are updated dependent on the shares of b and the shares of c, a change in a native value at any point caused by a fault on a single instance of p_T^n can never become ineffective and hence, is visible at the output of p_{TS}^n.

We will now use p_{TS}^n to build a circuit for the 2-share masked AES S-box. If we take a look at the building block given in Figure 1, the multiplications in $\mathbb{F}_{2^n}$ (later encapsulated in p_T^n) are the only non-linear components of the S-box. The square scaling over $\mathbb{F}_{2^n}$ ($s_{sc}^n(a, b, c)$) is a linear reversible operation, the linear maps ($\Gamma(a)$ and $\Xi(a)$) are linear permutations, the addition of the constant (AddConstant(a)) is an affine operation, and the inversion (Inv(a)) over $\mathbb{F}_{2^2}$ corresponds to a simple bit-swap. We will construct the S-box with the help of these basic circuits: Γ, AddConstant, Inv, p_T^n, s_{sc}^n, and Ξ. A description of the basic circuits Γ, AddConstant(a), Inv(a), s_{sc}^n, and Ξ is given in Appendix B.

Having discussed all necessary building blocks, we are ready to give our shared implementation of the AES S-box. In the description (Circuit 8.a) of the AES S-box, we use variables in different fields: $x_0, x_1, d_0, d_1, e_0, e_1, y_0, y_1 \in \mathbb{F}_{2^8}$, $a_0, a_1, c_0, c_1, f_0, f_1, h_0, h_1 \in \mathbb{F}_{2^4}$, and $b_0, b_1, g_0, g_1 \in \mathbb{F}_{2^2}$. Furthermore, we require that $a_0 + a_1 = 0$, $b_0 + b_1 = 0$, $c_0 + c_1 = 0$, $d_0 + d_1 = 0$ to correctly compute the AES S-box $y_0 + y_1 = S(x_0 + x_1)$. With superscripts

H and L, we denote the higher half of coefficients and the lower half of the coefficients, respectively. For example, $x_0 \in \mathbb{F}_{2^8}$ and thus $x_0^H, x_0^L \in \mathbb{F}_{2^4}$.

(8.a) Masked AES S-box with constraints $a_0 + a_1 = b_0 + b_1 = c_0 + c_1 = d_0 + d_1 = 0$

```
Name: Masked_AES_v1
Input: $\{x_0, x_1, a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1\}$
$\Gamma(x_0)$
$\Gamma(x_1)$
$p^4_{TS}(a_0, a_1, x_0^H, x_1^H, x_0^L, x_1^L)$
$s^4_{sc}(a_0, x_0^H, x_0^L)$
$s^4_{sc}(a_1, x_1^H, x_1^L)$
$p^2_{TS}(b_0, b_1, a_0^H, a_1^H, a_0^L, a_1^L)$
$s^2_{sc}(b_0, a_0^H, a_0^L)$
$s^2_{sc}(b_1, a_1^H, a_1^L)$
$\text{Inv}(b_0)$
$\text{Inv}(b_1)$
$p^2_{TS}(c_0^H, c_1^H, a_0^L, a_1^L, b_0, b_1)$
$p^2_{TS}(c_0^L, c_1^L, a_0^H, a_1^H, b_0, b_1)$
$p^4_{TS}(d_0^H, d_1^H, x_0^L, x_1^L, c_0, c_1)$
$p^4_{TS}(d_0^L, d_1^L, x_0^H, x_1^H, c_0, c_1)$
$\Xi(x_0)$
$\Xi(x_1)$
AddConstant$(x_0)$
$e_0 \leftarrow x_0, e_1 \leftarrow x_1, f_0 \leftarrow a_0, f_1 \leftarrow a_1$
$g_0 \leftarrow b_0, g_1 \leftarrow b_1, h_0 \leftarrow c_0, h_1 \leftarrow c_1$
$y_0 \leftarrow d_0, y_1 \leftarrow d_1$
Output: $\{e_0, e_1, f_0, f_1, g_0, g_1, h_0, h_1, y_0, y_1\}$
```

(8.b) Masked AES S-box with cloning

```
Name: Masked_AES_v2
Input: $\{x_0, x_1, a_0, b_0, c_0, d_0\}$
$(a_1, a_0) \leftarrow \text{Clone}(a_0)$
$(b_1, b_0) \leftarrow \text{Clone}(b_0)$
$(c_1, c_0) \leftarrow \text{Clone}(c_0)$
$(d_1, d_0) \leftarrow \text{Clone}(d_0)$
$\Gamma(x_0)$
$\Gamma(x_1)$
$p^4_{TS}(a_0, a_1, x_0^H, x_1^H, x_0^L, x_1^L)$
$s^4_{sc}(a_0, x_0^H, x_0^L)$
$s^4_{sc}(a_1, x_1^H, x_1^L)$
$p^2_{TS}(b_0, b_1, a_0^H, a_1^H, a_0^L, a_1^L)$
$s^2_{sc}(b_0, a_0^H, a_0^L)$
$s^2_{sc}(b_1, a_1^H, a_1^L)$
$\text{Inv}(b_0)$
$\text{Inv}(b_1)$
$p^2_{TS}(c_0^H, c_1^H, a_0^L, a_1^L, b_0, b_1)$
$p^2_{TS}(c_0^L, c_1^L, a_0^H, a_1^H, b_0, b_1)$
$p^4_{TS}(d_0^H, d_1^H, x_0^L, x_1^L, c_0, c_1)$
$p^4_{TS}(d_0^L, d_1^L, x_0^H, x_1^H, c_0, c_1)$
$\Xi(x_0)$
$\Xi(x_1)$
AddConstant$(x_0)$
$e_0 \leftarrow x_0, e_1 \leftarrow x_1, f_0 \leftarrow a_0, f_1 \leftarrow a_1$
$g_0 \leftarrow b_0, g_1 \leftarrow b_1, h_0 \leftarrow c_0, h_1 \leftarrow c_1$
$y_0 \leftarrow d_0, y_1 \leftarrow d_1$
Sinkhole$(e_1, f_1, g_1, h_1)$
Output: $\{y_0, y_1, f_0, g_0, h_0, e_0\}$
```

As can be seen in Circuit 8.a, each of the basic circuits is incomplete and hence, the effect stemming from faulting a single instance is independent of native values. Next, we have to show that the effect of a fault on a single instance is always present in the native values of our circuit. If a single fault targets Γ, an effect will be visible in the native value of e. An effect caused by a fault on a p^n_T or s^n_{sc} processing shares of x and a will show an effect on the native values of x (e) and a (f), since the value of the shares of a only depend on the shares of x and p^n_T and s^n_{sc} are a permutation on the shares. The same argument is valid for p^n_T or s^n_{sc}, processing shares of a and b, since Inv(b) is just a bit-swap. Next, we have a series of p^n_T processing shares of a, b, and c. Since the shares of c only depend on the shares of a and b, again a change in a native value is always visible in the output. The same is true for the last set of p^n_T processing shares of x, c, and d, followed by a share-wise permutation Ξ.

However, in the context of using this S-box within the AES, we cannot further use, e.g., e_0, e_1 as input for d_0, d_1 for another S-box, since $e_0 + e_1$ is not 0 in general. For this reason, we remove shares from the input and the output of our S-box in a similar spirit to Sugawara [Sug19] in Circuit 8.b, which we have also formally verified using `maskVerif` (cf. Section 6.1).

The S-box in Circuit 8.b is still a permutation on its inputs as we show in Appendix C. Since we do not have any restrictions on the inputs of the S-box, we are free to reuse e_0, f_0, g_0, h_0 as inputs for another S-box calculation, and hence, do not have to always generate fresh sharings of 0. In particular, this allows for first-order side-channel secured implementations of AES without the need for additional randomness in the masked S-boxes.

However, since we discard e_1, f_1, g_1, h_1 at the output of the S-box, we hinder faults from propagating and thus, have to employ fault countermeasures on S-box level for these values (respectively for their native values e, f, g, and h). While this results only in a rather small overhead for implementations that use duplication to protect against fault attacks, this might become quite expensive for implementations that use time redundancy since one might have to store all the values that need to be checked in the time redundant computation. However, this cost can be significantly reduced by computing and storing a checksum or fingerprint of these values instead. For instance, one might only store one set of $e_0, e_1, f_0, f_1, g_0, g_1, h_0, h_1$ all initialized to 0 and always update those shares by a linear checksum with the output $e_0, e_1, f_0, f_1, g_0, g_1, h_0, h_1$ of the S-box before truncation. Note that by using a linear checksum this can be done for e_0, f_0, g_0, h_0 and e_1, f_1, g_1, h_1 independently and thus secured against first-order side-channel attacks. By computing this checksum for the original and redundant computations, a fault will be detected by checking the output of the redundant AES computations and the checksum value.

5 Protecting Arbitrary Circuits

Our approach from Section 3 works by constructing the masked circuit for a cipher in a particular way with particular basic circuits. The cost of building masked circuits this way varies depending on the specific S-box. When optimizing with respect to other metrics, such as latency, other approaches may be more suitable than the one introduced in Section 3, which is essentially serial. The AES example in Section 4 also showed that additional error checks of intermediate values are helpful in cases when rewriting the entire cipher is hard. In this section, we generalize this approach and explore how a general masked circuit can be protected against SIFA by defining a suitable error detection circuit. The goal is to take an existing masked circuit, whose basic circuits are for example individual Boolean gates, and identify the relevant intermediate values to check for errors in addition to the cipher output.

We first recall the computation and fault model in order to introduce the general criterion for single-fault SIFA-resistance in Section 5.1. In Section 5.2, we show how to satisfy this criterion by extending a general masked implementation with local error detection checks. Then, in Section 5.3, 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 5.4.

5.1 A General Criterion for Resistance against Single-Fault SIFA

We consider the directed acyclic graph (DAG) induced by a masked cipher circuit composed of basic circuits (in the sense of Section 2.5 that basic circuits are incomplete). This *computation graph* consists of nodes that represent the basic circuits $f \in \mathcal{F}$ and that are connected by edges that represent the intermediate variables $v \in \mathcal{V}$. We identify a node f with n input edges $\textsc{in}(f) = (x_1, \ldots, x_n)$ and m output edges $\textsc{out}(f) = (y_1, \ldots, y_m)$ with the corresponding vectorial Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$, $(x_1, \ldots, x_n) \mapsto (y_1, \ldots, y_m)$ of the basic circuit. We distinguish linear nodes, whose function is affine linear over $\mathbb{F}_2$, and nonlinear nodes.

As defined in Section 2, we consider a powerful single-fault attacker who may replace any node $(y_1, \ldots, y_m) = f(x_1, \ldots, x_n)$ by an arbitrary faulted node $(y_1^*, \ldots, y_m^*) = f^*(x_1, \ldots, x_n)$. We denote the difference between the values of an edge v in the correct execution and v^* in the faulted execution by $\delta v = v \oplus v^*$, similar to differential cryptanalysis, and write the resulting deviation in the output variables of a node as a function δf of the input value:

$$(y_1^*, \ldots, y_m^*) = f^*(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \oplus \delta f(x_1, \ldots, x_n)\,.$$

The fault δv may propagate to other nodes, and we call a node $f \in \mathcal{F}$ *active* in a faulted execution if either $\delta v = 1$ for any input edge $v \in \textsc{in}(f)$ or f is the faulted gate modified by the attacker.

We denote the fault alert by Δ and the set of variables it checks by $\mathcal{V}_\Delta$, i.e., $\Delta := \bigvee_{v \in \mathcal{V}_\Delta} \delta v = \bigvee_{v \in \mathcal{V}_\Delta} (v \oplus v^*)$. 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 edges near the faulted node f^*.

Example. As an example throughout this section, Figure 3 lists the operations of a masked implementation of the 3-bit S-box χ_3 together with its computation graph similar to [GSM17]. In the graph, Clone($\cdot$) nodes are represented by small bullets. In the circuit on the left-hand side, for compactness, we omit calls $(v, v') = \text{Clone}(v)$ (i.e., variables named v' are always clones) and list up to two nodes per line. 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 3: If a bitflip is induced as indicated (ϟ) in the input a_0 of the Clone(a_0) node, 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 the detector (or the structure of the DAG).

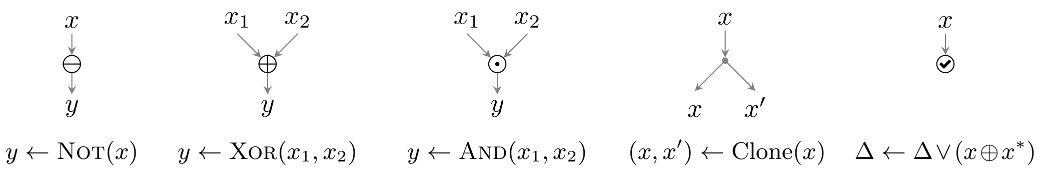

x
y
x_1
x_2
y
x_1
x_2
y
x
x
x'
x
$y \leftarrow \textsc{Not}(x)$
$y \leftarrow \textsc{Xor}(x_1, x_2)$
$y \leftarrow \textsc{And}(x_1, x_2)$
$(x, x') \leftarrow \text{Clone}(x)$
$\Delta \leftarrow \Delta \vee (x \oplus x^*)$

Figure 2: Nodes for basic circuits in the computation graph examples.

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 $\Delta = 0$. We call the implementation *single-fault SIFA-resistant* if each possible single fault is either detected by Δ or activates at most one nonlinear node.

To see why this criterion is sufficient, consider a fault f^*. 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 $\delta f(x_1, \ldots, x_n) = 0$. 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.

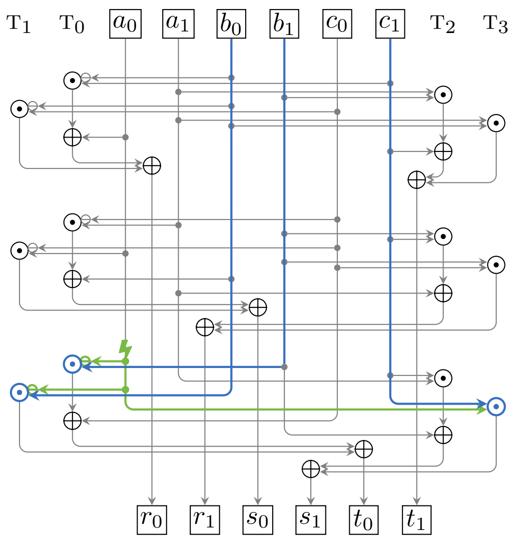
Input: $(a_0, a_1, b_0, b_1, c_0, c_1)$

$\mathrm{T}_0 \leftarrow \overline{b_0'} \odot c_1'$; $\mathrm{T}_2 \leftarrow a_1' \odot b_1'$
$\mathrm{T}_1 \leftarrow b_0' \odot c_0'$; $\mathrm{T}_3 \leftarrow a_1' \odot b_0'$
$\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus a_0'$; $\mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus c_1'$
$r_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1$; $t_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3$

$\mathrm{T}_0 \leftarrow \overline{c_0'} \odot a_1'$; $\mathrm{T}_2 \leftarrow b_1' \odot c_1'$
$\mathrm{T}_1 \leftarrow c_0' \odot a_0'$; $\mathrm{T}_3 \leftarrow b_1' \odot c_0'$
$\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus b_0'$; $\mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus a_1'$
$s_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1$; $r_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3$
ϟa_0
$\mathrm{T}_0 \leftarrow \overline{a_0'} \odot b_1'$; $\mathrm{T}_2 \leftarrow c_1' \odot a_1$
$\mathrm{T}_1 \leftarrow a_0' \odot b_0$; $\mathrm{T}_3 \leftarrow c_1 \odot a_0$
$\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus c_0$; $\mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus b_1$
$t_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1$; $s_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3$

Output: $(r_0, r_1, s_0, s_1, t_0, t_1)$

(a) Circuit, $(v, v') = \mathrm{Clone}(v)$ omitted



(b) Computation graph

Figure 3: Bitflip in masked χ_3 using 2 shares (resharing at the output omitted).

- $\delta y \neq 0$, but the resulting bitflip(s) did not propagate to Δ. The criterion implies that there is at most one active nonlinear node, i.e., either f or another nonlinear node f' with some changed input $v^* = v \oplus \delta v$. The attacker may exploit this differential information to learn the inputs of this active nonlinear node, which are however independent of the native inputs, and will not learn anything from the other, trivial differentials (of nonlinear nodes with zero input difference or of linear nodes).

5.2 Protection against SIFA using Fine-Grained Detection

We now explore how a masked implementation can be extended with a suitable detector Δ in order to achieve a single-fault SIFA-resistant implementation.

Basic Idea. 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 nodes. Assume for instance that we duplicate the implementation and feed the same inputs to both instances. For each nonlinear node f and each of its input edges $v \in \textsc{in}(f)$, we add a check to update the detector $\Delta \leftarrow \Delta \vee (v \oplus v^*)$. 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 node f without detection by Δ (if the attacker faults either the nonlinear node itself or the preceding check), but it cannot activate two nodes, since there are no paths without a check either from any node to two nonlinear nodes or from one nonlinear node to another. Thus, any single-bit fault in one of the two redundant computations or in the additional circuitry for the detector Δ are either detected or do not leak information to the attacker.

Reducing Checks. It is, however, not necessary to check the inputs to all nonlinear nodes separately. For example, in many circuits, most inputs to nonlinear nodes in the DAG are directly cloned from the shares of the inputs, i.e., the node 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 Clone$(\cdot)$

nodes and whose leaves are other nodes. 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 node 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 nodes 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 circuit 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 edge v in the DAG is a *safe* edge that has a sink s such that there is exactly one directed path $v \to s$ and it contains at most one nonlinear node. This implies the single-fault SIFA-resistance criterion of Section 5.1.

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 (instead of 24 in the naive approach), i.e., once for each input variable. The result is illustrated in Figure 4.

5.3 Ensuring Fault Propagation

In this section, we discuss under which conditions the fine-grained, local detection of Section 5.2 can be replaced by global checks, similar to Section 3. We will again use the concept of sink nodes as in Section 5.2, 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 edge is a safe edge 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. For simplicity, we assume that all nodes except cloning nodes have a single output bit.

Instead of the individual shares of the S-box output bits, we can use the native 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 native S-box output would activate at least one bit in the native 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 edge v in the S-box circuit is a safe edge with respect to one of these sinks s, i.e., that there is exactly one directed path $v \to s$ and it contains at most one nonlinear node. This may be violated due to cloning nodes (or, generally, nodes with multiple outputs) and due to composition of nonlinear nodes within an S-box. If the circuit contains such a composition of nonlinear nodes, it needs to be decomposed into smaller, bijective circuits with nonlinear depth 1 first, similar to Section 3. For cloning, we consider the cloning subtree as in Section 5.2. We need to ensure that whenever two twigs in this tree activate, a sink s activates. In particular, this implies that for every cloning 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. 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 node, 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 perform the following modifications to the circuit from Figure 3 so that each edge is now a safe edge:

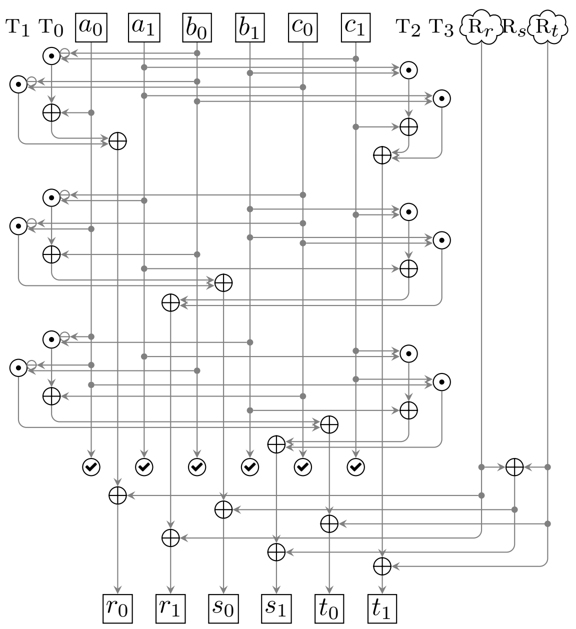
```
Input: $(a_0, a_1, b_0, b_1, c_0, c_1)$
$\mathrm{T}_0 \leftarrow \overline{b'_0} \odot c'_1$ ; $\mathrm{T}_2 \leftarrow a'_1 \odot b'_1$
$\mathrm{T}_1 \leftarrow \overline{b'_0} \odot c'_0$ ; $\mathrm{T}_3 \leftarrow a'_1 \odot b'_0$
$\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus a'_0$ ; $\mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus c'_1$
$r_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1$ ; $t_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3$

$\mathrm{T}_0 \leftarrow \overline{c'_0} \odot a'_1$ ; $\mathrm{T}_2 \leftarrow b'_1 \odot c'_1$
$\mathrm{T}_1 \leftarrow \overline{c'_0} \odot a'_0$ ; $\mathrm{T}_3 \leftarrow b'_1 \odot c'_0$
$\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus b'_0$ ; $\mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus a'_1$
$s_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1$ ; $r_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3$

$\mathrm{T}_0 \leftarrow \overline{a'_0} \odot b'_1$ ; $\mathrm{T}_2 \leftarrow c'_1 \odot a'_1$
$\mathrm{T}_1 \leftarrow \overline{a'_0} \odot b'_0$ ; $\mathrm{T}_3 \leftarrow c'_1 \odot a'_0$
$\mathrm{T}_0 \leftarrow \mathrm{T}_0 \oplus c'_0$ ; $\mathrm{T}_2 \leftarrow \mathrm{T}_2 \oplus b'_1$
$t_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{T}_1$ ; $s_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{T}_3$
$\mathrm{R}_s \leftarrow \mathrm{R}'_r \oplus \mathrm{R}'_t$ ; CHECKS
$r_0 \leftarrow r_0 \oplus \mathrm{R}'_r$ ; $s_0 \leftarrow s_0 \oplus \mathrm{R}'_s$
$t_0 \leftarrow t_0 \oplus \mathrm{R}'_t$ ; $r_1 \leftarrow r_1 \oplus \mathrm{R}_r$
$s_1 \leftarrow s_1 \oplus \mathrm{R}_s$ ; $t_1 \leftarrow t_1 \oplus \mathrm{R}_t$

Output: $(r_0, r_1, s_0, s_1, t_0, t_1)$
```

(a) Circuit, $(v, v') = \mathrm{Clone}(v)$ omitted

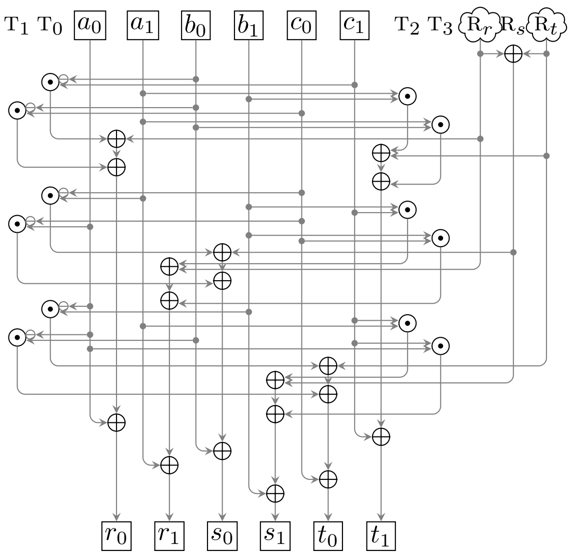


(b) Computation graph

Figure 4: Single-fault SIFA-resistant χ_3 using 2 shares, with local checks. CHECKS is short for the error detecting sub-circuits $\Delta \leftarrow \Delta \vee (v \oplus v^*)$ for $v \in \{a_0, a_1, b_0, b_1, c_0, c_1\}$.

```
Input: $(a_0, a_1, b_0, b_1, c_0, c_1)$
$\mathrm{R}_s \leftarrow \mathrm{R}'_r \oplus \mathrm{R}'_t$
$\mathrm{T}_0 \leftarrow \overline{b'_0} \odot c'_1$ ; $\mathrm{T}_2 \leftarrow a'_1 \odot b'_1$
$\mathrm{T}_1 \leftarrow \overline{b'_0} \odot c'_0$ ; $\mathrm{T}_3 \leftarrow a'_1 \odot b'_0$
$r_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{R}'_r$ ; $t_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{R}'_t$
$r_0 \leftarrow r_0 \oplus \mathrm{T}_1$ ; $t_1 \leftarrow t_1 \oplus \mathrm{T}_3$
$\mathrm{T}_0 \leftarrow \overline{c'_0} \odot a'_1$ ; $\mathrm{T}_2 \leftarrow b'_1 \odot c'_1$
$\mathrm{T}_1 \leftarrow \overline{c'_0} \odot a'_0$ ; $\mathrm{T}_3 \leftarrow b'_1 \odot c'_0$
$s_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{R}'_s$ ; $r_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{R}_r$
$s_0 \leftarrow s_0 \oplus \mathrm{T}_1$ ; $r_1 \leftarrow r_1 \oplus \mathrm{T}_3$
$\mathrm{T}_0 \leftarrow \overline{a'_0} \odot b'_1$ ; $\mathrm{T}_2 \leftarrow c'_1 \odot a'_1$
$\mathrm{T}_1 \leftarrow \overline{a'_0} \odot b'_0$ ; $\mathrm{T}_3 \leftarrow c'_1 \odot a'_0$
$t_0 \leftarrow \mathrm{T}_0 \oplus \mathrm{R}_t$ ; $s_1 \leftarrow \mathrm{T}_2 \oplus \mathrm{R}_s$
$t_0 \leftarrow t_0 \oplus \mathrm{T}_1$ ; $s_1 \leftarrow s_1 \oplus \mathrm{T}_3$
$r_0 \leftarrow r_0 \oplus a_0$ ; $t_1 \leftarrow t_1 \oplus c_1$
$s_0 \leftarrow s_0 \oplus b_0$ ; $r_1 \leftarrow r_1 \oplus a_1$
$t_0 \leftarrow t_0 \oplus c_0$ ; $s_1 \leftarrow s_1 \oplus b_1$
Output: $(r_0, r_1, s_0, s_1, t_0, t_1)$
```

(a) Circuit, $(v, v') = \mathrm{Clone}(v)$ omitted



(b) Computation graph

Figure 5: Single-fault SIFA-resistant χ_3 using 2 shares, with global checks.

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

The resulting circuit in Figure 5 shows similarities with the Toffoli-based implementation of χ_3 in Section 3, but there are still significant differences; most notably, the necessity for resharing variables $\mathrm{R}_r, \mathrm{R}_t$ and the lower depth of the circuit in Figure 5.

5.4 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 basic circuits (nodes).

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 or observes up to d basic circuits still does not gain any information on any native value. Let the circuit be implemented with at least $d+1$ redundant computations and an error detector Δ of at least d bits. For simplicity, assume that each of the attacker's d faults is a bitflip fault on one of the intermediate variables. We call the implementation *d-fault SIFA-resistant* if each possible d-bit fault is either detected by Δ or activates at most d nonlinear nodes in total.

This criterion can, for instance, be satisfied by checking all inputs to nonlinear nodes with the following construction. We use $d+1$ redundant computations and an n_Δ-bit error detector $\Delta = (\Delta_1, \ldots, \Delta_{n_\Delta})$, where $n_\Delta = d$ for odd d and $n_\Delta = d+1$ for even d. For each relevant input edge, we clone d times to update d different error detector bits Δ_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 Δ_i. Distributing the $\binom{d+1}{2}$ differences to the various Δ_i corresponds to an edge coloring problem with n_Δ colors in the complete graph with $d+1$ vertices, which is easy to solve. Then, activating k nodes in one computation without detection by Δ requires at least $\min(k, d+1)$ faults: each node can either be activated without triggering Δ by placing a fault between the check (with its cloning) and the nonlinear node; or it can be activated while triggering d error detector bits Δ_i, each of which requires either a fault in the corresponding computation or faulting Δ_i directly to eliminate. Thus, in summary, at least $d+1$ faults would be required in order to activate $d+1$ or more nonlinear nodes and thus learn $d+1$ shares of any variable to deduce information on its native value.

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

6 Implementation

In this section, we describe our experimental results on correctness and performance and discuss how well our circuit model matches the reality of hardware and software implementations.

6.1 Formally Verifying the Masking of Toffoli-based Circuits

To gain additional trust in the soundness of our masking, we verified the circuits using a tool-assisted approach. More specifically, we make use of `maskVerif`, a tool by Barthe et al. [BBD+15] for formally verifying masking schemes. `maskVerif` takes as input a (masked) circuit description that mainly consists of simple logical operations such as AND, XOR, or NOT. The interface of the circuit can consist of (shared) variables, as well as additional randomness. Given such a masked circuit, `maskVerif` can verify if the implemented masking is indeed correct in a specified leakage model and with a specified protection order. For more details about `maskVerif` we refer to the original publication [BBD$^+$15] and the tool's website [GBB].

We verified the correctness of the masking of our Toffoli-based circuits for S-boxes from AES and KECCAK using `maskVerif`. Therefore, we converted the circuits from Circuit 7.b (5-bit χ) and Circuit 8.b (AES S-box) into a `maskVerif`-compatible format and successfully verified their first-order security in the probing model and in the presence of (propagation delay) glitches[1]. Note that, for implementations in hardware and software, additional design considerations are necessary in practice, which we discuss in part in Section 6.3 and Section 6.4.

6.2 Benchmarks and Practical Evaluation

In this section, we give a preliminary evaluation of the costs of our proposed countermeasure. We start with a theoretical estimation of the costs and then we demonstrate the effectiveness and low overhead of our countermeasure by implementing KECCAK-f[200] on a low-end 8-bit AVR XMEGA128D4 microprocessor as an example. We consider both our Toffoli-based masked S-box (Circuit 7.b) and a traditionally masked S-box using Domain-Oriented Masking (DOM) [GSM17]. We then use these implementations for comparing their runtime and code size and for verifying the SIFA-protection of Circuit 7.b in a practical evaluation.

6.2.1 Cost Estimation

First, let us discuss protection against single-fault SIFA. The costs of our countermeasures can be roughly split into two parts: the cost of masking and the cost of redundancy. Let us first start with the cost of redundacy. Since we perform plain double execution, we get roughly a factor two of overhead. This overhead is in terms of space due to roughly the doubled number of registers needed to store the state and might be in time, e.g., in software implementations where everything has to be executed twice, or in hardware if designers decide to not duplicate the whole circuit, but perform time redundancy instead. So, if we consider that masking is needed anyway and our masking strategies are not worse than common strategies, we get the same overhead that would be needed for fault protection by duplication.

Now, let us take a look at the efficiency of our masking proposal. Let us start with the way of masking that we introduce in Section 3 and Section 4. This way of masking relies on describing a cipher in terms of incomplete permutation circuits, and hence, mainly relies on masked versions of the Toffoli gate and related constructions. As it can be seen in Figure 1, such a description comes even quite natural in the case of AES and does not incur a prohibitive increase in the number of operations needed to compute an S-box. In addition, those masked implementations are secure without the need of online randomness. As an example, let us have a look at KECCAK's S-box and compare a DOM implementation [GSM17] with masking the Toffoli-based description given in Circuit 7.b.

Let us start with the DOM implementation of KECCAK's S-box. To mask this with two shares, we need 5 DOM ANDNOTS plus 10 XORS. Hence, in total, we roughly need 20 2-bit ANDS/ANDNOTS, 30 2-bit XORS, and 5 random bits for calculating KECCAK's S-box. In contrast, in Circuit 7.b, we need 5 calls to $p_{\chi S}$ plus two XORS. Thus, we have a total of 20 2-bit ANDS/ANDNOTS, 22 2-bit XORS, and 0 random bits for calculating KECCAK's S-box. Hence, in terms of operations, the Toffoli-based version has a slight advantage. However, we need storage for one more share at the input. We will see how this compares in a practical implementation in Section 6.2.2.

[1]The used code is available at https://github.com/sifa-aux/countermeasures

The overhead incurred by following the strategy of Section 5 depends on a number of factors. We focus again on the DOM implementation of KECCAK's S-box (with 20 ANDs/ANDNOTs, 30 XORs, and 5 resharing bits), both as a point of comparison and as the target circuit for the countermeasure. First consider the approach with local checks from Section 5.2, Figure 4. On top of the $2 \times 20 + 2 \times 30 = 100$ gates for duplicate execution of this S-box, our countermeasure adds 10 XORs and 10 ORs for the checking circuit, plus an additional state of 1 bit globally. It is necessary to either execute both duplicate instances in parallel or invest additional space to keep track of the checked intermediate values. Other metrics, such as the circuit depth, remain essentially unchanged. With the improvements for global checks from Section 5.3, Figure 5, the countermeasure comes with zero overhead compared to the basic DOM implementation with simple redundancy.

When considering the ideas from Section 5.4 for higher-order protection, the overheads are more substantial. When considering a straightforward application without any optimizations, this can be estimated as follows when focusing only on nonlinear gates (which are responsible for the overhead). For protection against d-fault SIFA, we need a masked implementation of dth order with (at least) $d+1$ shares, as well as $d+1$-fold redundant execution with an error detector of $n_\Delta \in \{d, d+1\}$ bits. With the higher-order DOM approach [GSM17], each AND-gate of an unprotected S-box circuit corresponds to $(d+1)^2$ ANDs plus $(d+1)^2$ XORs in a masked circuit and requires up to $d \cdot (d+1)/2$ resharing bits. These gates are duplicated $d+1$ times for redundancy, resulting in a total of $2 \times (d+1)^3$ gates (ANDs and XORs). Our countermeasure adds a check (1 XOR plus 1 OR) for each of the inputs of these ANDs, for a total of $(d+1)^2 \times 2 \times \frac{(d+1)\cdot d}{2} \times 2 = 2d \times (d+1)^3$ gates. This corresponds to an overhead of a factor of d in the number of gates compared to DOM masking with redundancy (only for the nonlinear gates, the linear gates add no overhead). We expect that for concrete circuits, significant optimizations similar to Section 5 are possible. Still, we consider this approach to be primarily of theoretical interest.

6.2.2 Practical Benchmarks

To keep the comparison as fair as possible, we opted to take the compact C implementation of KECCAK-f[200] from the eXtended Keccak Code Package [BDH+] as the basis for our implementations. We then simply duplicated all linear operations and replaced the S-box by Toffoli/DOM masked assembly versions[2]. The resulting performance numbers are hence not necessarily representative for the maximum performance on 8-bit platforms but very representative for a direct comparison of the two S-box implementations. The resulting numbers for our comparison are shown in Table 2. We measured the runtime without the runtime cost of a PRNG and discuss the needed amount of random bits as a separate metric. From the presented numbers, it is easy to see that our Toffoli-based S-box is on par with the DOM variant in terms of runtime, and binary size. Please note that the AVR XMEGA128D4 has not been designed for cryptographic purposes, and hence, might allow for side-channel attacks due to violations of the seperation of the shares. However, what we show in the next section is that our proof-of-concept implementation still provides protection against single fault SIFA.

6.2.3 Evaluation of SIFA resistance

We also evaluated the SIFA resistance of our designs by means of simulated fault injections and a practical evaluation on an AVR XMEGA128D4 microprocessor using clock glitches. The evaluation methodology is the same as the one that is used by the authors of [DEG+18] (Section 4.2): We take our Toffoli masked assembly S-box implementation (the same that is used in the benchmarks), target one instruction with a fault injection, call the S-box with every possible input, and check whether the correct unmasked S-box outputs follow a

[2]The used code is available at https://github.com/sifa-aux/countermeasures

Table 2: Comparison of computing 18 rounds of Keccak-f[200] using different implementations. Numbers do not include generation of randomness: The DOM approach requires 5 random bits per S-box ($200 + 200 \times 18 = 3800$ bits in total), which can be reduced with techniques such as Changing of the Guards. The Toffoli approach requires just $200 + 40$ in total when implemented as proposed for Circuit 7.b.

Implementations	Runtime w/o PRNG (Clock Cyles)	Binary Size (Bytes)
Empty main.c `-Os`	0	5 385
Compact C with ASM S-box `-Os`	49 371	6 939
Masked with ASM DOM S-box `-Os`	107 617	9 648
Masked with ASM Toffoli S-box `-Os`	109 753	9 632
Compact C with ASM S-box `-O3`	38 455	9 811
Masked with ASM DOM S-box `-O3`	88 332	12 434
Masked with ASM Toffoli S-box `-O3`	87 426	12 385

uniform distribution or not. This procedure is then repeated for every instruction within the S-box.

According to our practical evaluation, where clock glitches cause effects like memory corruption or instruction skips, no instruction within our S-box implementation is susceptible to SIFA. This result is backed up by our simulated fault injection experiments where we simulate the effect of stuck-at faults and bitflips for which we do not own a set-up to reproduce them in practice.

6.3 From an Abstract Model to Software Implementations

In Section 2, we have introduced an abstraction model and explicitly defined what faults are in this model. When considering the implementation of our circuits in software, it would seem that even in the most trivial implementations, it is ensured that basic circuits are nicely separated and hence, fault attacks and also side-channel attacks cannot be a threat. However, the reality is more subtly nuanced. Hence, we want to discuss what has to be considered when implementing our circuits in real software implementations and which faults on software implementations are covered by considering faults on basic circuits.

As mentioned in Section 2, we consider circuit faults in a single basic circuit instance. This directly corresponds to faults in software that directly manipulate values of variables stored in registers of a CPU [SZK+18], change a variable in memory before it is loaded, or even target the load of a variable from memory [CMD+18]. 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 can lead to cases where the boundaries between basic circuits are violated. This is especially a threat if an implementation of a basic circuit uses registers in addition to the registers storing the shares in order to store results of intermediate computations. However, potential negative effects of a clock glitch can be mitigated by always initializing an additional register to 0 before use, or by performing instructions on the shares in place (e.g., $a_0 = a_0 \oplus b_0$) whenever possible.

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.

Furthermore, our model considers a single permanent fault, e.g, permanently faulting a lookup table, as multiple faults. However, we advise not to use implementations with lookup tables.

A notable case that is not considered in our abstraction 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 Circuit 5.b. 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 anyway. 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, registers have to be properly initialized before usage. The problem with uninitialized registers 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 registers can be easily solved by always writing 0 to them before the result of a computation is stored.

Finally, we want to note that modern ciphers can usually be implemented in a bit-sliced manner, meaning that for 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. For ciphers that consist of layers applying many small bijective S-boxes in parallel to the state, we can define basic circuits to work on bit-vectors instead of single bits. This implies that injecting a single fault in several of these parallel S-boxes in a single layer causes no issues with respect to our strategy of Section 3, since these faults will correspond to a single circuit fault of an incomplete circuit.

6.4 From an Abstract Model to Hardware Implementations

In general, our abstraction as circuit lends itself quite naturally to dedicated hardware implementations, but requires additional considerations. In particular, one needs to take into account the effects of glitches. Glitches are the result of the behavior of the physical layout and are thus unavoidable. Since signals do not propagate evenly through a hardware-circuit (due to 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. In the context of a fault injection, also faults can "glitch". As a result, in each clock cycle there is sequence of transitional states in the physical circuit that depend on combinations of variables that differ from the ones that the circuit should finally compute.

These effects imply that the behavior of a hardware-circuit cannot be controlled by just using combinatorial logic gates. Using registers limits these transitional effects in the sense that it puts barriers between combinatorial blocks. 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 not only the increased gate count, but also the evaluation of the hardware-circuit requires more clock cycles and thus, the latency increases.

Hence, when instantiating our abstract circuits in hardware, registers are required at several places to separate the basic circuits and ensure resistance to glitching effects. This is no different for other masking methods. For instance, TI implementations [NRR06,NRS11] use registers after each uniformly shared function and the DOM scheme [GMK16] uses

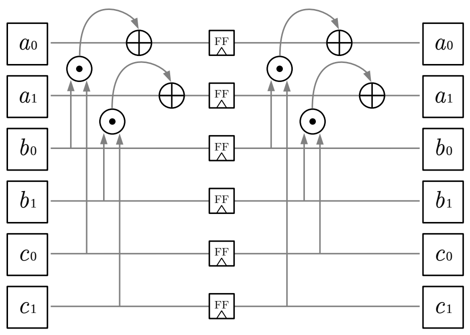


Figure 6: Masked and single-fault SIFA-protected Toffoli gate in hardware.

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 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. The upper two registers are required to hinder the propagation of glitches that could violate the side-channel resistance of the implementation. The lower four registers are the relevant ones for protection against SIFA. Again, no share can be used twice in two different basic circuits within the same clock cycle. 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.

7 Conclusion

In this paper, we proposed two different approaches to counteract SIFA on an algorithmic level. First, we showed that by relying on the Toffoli gate for the non-linear operations in the implementation of masked ciphers, 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 show how this approach could be extended to the AES S-box using the Toffoli gate for bigger fields and fine-grained detection on S-box level that can be implemented efficiently for implementations using duplication and is also quite efficient for implementations using time-redundancy when using a linear checksum. We verified the correctness of the masking of our Toffoli-based circuits for S-boxes from AES and KECCAK using `maskVerif`. Finally, we show how this approach based on fine-grained detection can be generalized to protect arbitrary masked circuits, and how it can be 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, and via the K-project DeSSnet, which is funded in the context of COMET - Competence Centers for Excellent Technologies by BMVIT, BMWFW, 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

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 430–454. Springer, 2015.

[BBD+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 457–485, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[BBP+17] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Private multiplication over finite fields. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, volume 10403 of *LNCS*, pages 397–426. Springer, 2017.

[BDF+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, volume 10210 of *LNCS*, pages 535–566, 2017.

[BDH+] Guido Bertoni, Joan Daemen, Seth Hoffert, Michael Peeters, Gilles Van Assche, and Ronny Van Keer. eXtended Keccak Code Package. `https://github.com/XKCP/XKCP`. Accessed: 2019-01-11.

[BDH+17] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Farfalle: parallel permutation-based cryptography. *IACR Transactions on Symmetric Cryptology*, 2017(4):1–38, 2017.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT '97*, volume 1233 of *LNCS*, pages 37–51. Springer, 1997.

[BDP+16a] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Ketje v2. Submission to the CAESAR competition, 2016. `https://keccak.team/files/Ketjev2-doc2.0.pdf`.

[BDP+16b] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keyak v2. Submission to the CAESAR competition, 2016. `https://keccak.team/files/Keyakv2-doc2.2.pdf`.

[BDPV11] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. The Keccak SHA-3 submission (Version 3.0). `http://keccak.noekeon.org/Keccak-submission-3.pdf`, 2011.

[BECN+06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[Ben73] Charles H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[BKHL19] Jakub Breier, Mustafa Khairallah, Xiaolu Hou, and Yang Liu. A countermeasure against Statistical Ineffective Fault Analysis. Cryptology ePrint Archive, Report 2019/515, 2019. https://eprint.iacr.org/2019/515.

[BKL$^+$07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007*, volume 4727 of *LNCS*, pages 450–466. Springer, 2007.

[BNN$^+$15] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, Natalia N. Tokareva, and Valeriya Vitkup. Threshold implementations of small S-boxes. *Cryptography and Communications*, 7(1):3–33, 2015.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology – CRYPTO '97*, volume 1294 of *LNCS*, pages 513–525. Springer, 1997.

[Can05] David Canright. A very compact S-box for AES. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *LNCS*, pages 441–455. Springer, 2005.

[Cla07] Christophe Clavier. Secret external encodings do not prevent transient fault analysis. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007*, volume 4727 of *LNCS*, pages 181–194. Springer, 2007.

[CMD$^+$18] Brice Colombier, Alexandre Menu, Jean-Max Dutertre, Pierre-Alain Moëllic, Jean-Baptiste Rigaud, and Jean-Luc Danger. 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.

[Dae95] Joan Daemen. *Cipher and hash function design, strategies based on linear and differential cryptanalysis*. PhD thesis, KU Leuven, 1995. http://jda.noekeon.org/.

[Dae17] Joan Daemen. Changing of the guards: A simple and efficient method for achieving uniformity in threshold sharing. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, volume 10529 of *LNCS*, pages 137–153. Springer, 2017.

[DAN$^+$19] Lauren De Meyer, Victor Arribas, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. M&M: Masks and Macs against physical attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(1):25–50, 2019.

[De 07] Christophe De Cannière. *Analysis and design of symmetric encryption algorithms*. PhD thesis, KU Leuven, 2007.

[DEG$^+$18] Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Stefan Mangard, Florian Mendel, and Robert Primas. Statistical Ineffective Fault Attacks on masked AES with fault countermeasures. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, volume 11273 of *LNCS*, pages 315–342. Springer, 2018.

[DEK$^+$18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: Exploiting ineffective fault inductions on symmetric cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):547–572, 2018.

[DEMS16] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. Submission to the CAESAR Competition, 2016. https://ascon.iaik.tugraz.at/files/asconv12.pdf.

[DHVV18] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. The design of Xoodoo and Xoofff. *IACR Transactions on Symmetric Cryptology*, 2018(4):1–38, 2018.

[DMMP18] Christoph Dobraunig, Stefan Mangard, Florian Mendel, and Robert Primas. Fault attacks on nonce-based authenticated encryption: Application to Keyak and Ketje. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography – SAC 2018*, volume 11349 of *LNCS*, pages 257–277. Springer, 2018.

[DN16] Thomas De Cnudde and Svetla Nikova. More efficient Private Circuits II through Threshold Implementations. In *Fault Diagnosis and Tolerance in Cryptography – FDTC 2016*, pages 114–124. IEEE Computer Society, 2016.

[DPVR00] Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. Nessie proposal: the block cipher Noekeon. Nessie submission, 2000. http://gro.noekeon.org/.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES – The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[DRB$^+$16] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Masking AES with d+1 shares in hardware. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, volume 9813 of *LNCS*, pages 194–212. Springer, 2016.

[FJLT13] Thomas Fuhr, Éliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault attacks on AES with faulty ciphertexts only. In Wieland Fischer and Jörn-Marc Schmidt, editors, *Fault Diagnosis and Tolerance in Cryptography – FDTC 2013*, pages 108–118. IEEE Computer Society, 2013.

[GBB] Benjamin Grégoire, Gilles Barthe, and Sonia Belaïd. maskVerif – automatic tool for the verification of side-channel countermeasures. https://cryptoexperts.com/maskverif/. Accessed: 2019-10-10.

[GIB18] Hannes Groß, Rinat Iusupov, and Roderick Bloem. Generic low-latency masking in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):1–21, 2018.

[GM12] Oleg Golubitsky and Dmitri Maslov. A study of optimal 4-bit reversible Toffoli circuits and their synthesis. *IEEE Transactions on Computers*, 61(9):1341–1353, 2012.

[GM17] Hannes Groß and Stefan Mangard. Reconciling d+1 masking in hardware and software. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, volume 10529 of *LNCS*, pages 115–136. Springer, 2017.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. 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.

[GSM17] Hannes Groß, David Schaffenrath, and Stefan Mangard. Higher-order side-channel protected implementations of KECCAK. In Hana Kubátová, Martin Novotný, and Amund Skavhaug, editors, *Digital System Design – DSD 2017*, pages 205–212. IEEE Computer Society, 2017.

[IPSW06] Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and David A. Wagner. Private Circuits II: Keeping secrets in tamperable circuits. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 308–327. Springer, 2006.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private Circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, 2003.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.

[KLPR10] Lars R. Knudsen, Gregor Leander, Axel Poschmann, and Matthew J. B. Robshaw. PRINTcipher: A block cipher for IC-printing. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems – CHES 2010*, volume 6225 of *LNCS*, pages 16–32. Springer, 2010.

[Lan61] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[LP07] Gregor Leander and Axel Poschmann. On the classification of 4 bit S-boxes. In Claude Carlet and Berk Sunar, editors, *Arithmetic of Finite Fields – WAIFI 2007*, volume 4547 of *LNCS*, pages 159–176. Springer, 2007.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security – ICICS 2006*, volume 4307 of *LNCS*, pages 529–545. Springer, 2006.

[NRS11] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *Journal of Cryptology*, 24(2):292–321, 2011.

[QS01] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In Isabelle Attali and Thomas P. Jensen, editors, *Smart Card Programming and Security – E-smart 2001*, volume 2140 of *LNCS*, pages 200–210. Springer, 2001.

[RAD19] Keyvan Ramezanpour, Paul Ampadu, and William Diehl. RS-Mask: Random space masking as an integrated countermeasure against power and fault analysis. arXiv Report abs/1911.11278, 2019.

[RBN$^+$15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, volume 9215 of *LNCS*, pages 764–783. Springer, 2015.

[RDB+18] Oscar Reparaz, Lauren De Meyer, Begül Bilgin, Victor Arribas, Svetla Nikova, Ventzislav Nikov, and Nigel P. Smart. CAPA: The spirit of beaver against physical attacks. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, volume 10991 of *LNCS*, pages 121–151. Springer, 2018.

[SJR+19] Sayandeep Saha, Dirmanto Jap, Debapriya Basu Roy, Avik Chakraborti, Shivam Bhasin, and Debdeep Mukhopadhyay. Transform-and-Encode: A countermeasure framework for Statistical Ineffective Fault Attacks on block ciphers. Cryptology ePrint Archive, Report 2019/545, 2019. https://eprint.iacr.org/2019/545.

[SMG16] Tobias Schneider, Amir Moradi, and Tim Güneysu. ParTI – towards combined hardware countermeasures against side-channel and fault-injection attacks. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, volume 9815 of *LNCS*, pages 302–332. Springer, 2016.

[SPMH03] Vivek V. Shende, Aditya K. Prasad, Igor L. Markov, and John P. Hayes. Synthesis of reversible logic circuits. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 22(6):710–722, 2003.

[SRM19] Aein Rezaei Shahmirzadi, Shahram Rasoolzadeh, and Amir Moradi. Impeccable Circuits II. Cryptology ePrint Archive, Report 2019/1369, 2019. https://eprint.iacr.org/2019/1369.

[Sug19] Takeshi Sugawara. 3-share threshold implementation of AES S-box without fresh randomness. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(1):123–145, 2019.

[SZK+18] Bodo Selmke, Kilian Zinnecker, Philipp Koppermann, Katja Miller, Johann Heyszl, and Georg Sigl. 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*, pages 7–14. IEEE Computer Society, 2018.

[TBM14] Harshal Tupsamudre, Shikha Bisht, and Debdeep Mukhopadhyay. Destroying fault invariant with randomization – A countermeasure for AES against Differential Fault Attacks. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, volume 8731 of *LNCS*, pages 93–111. Springer, 2014.

[Tof80] Tommaso Toffoli. Reversible computing. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming, 1980*, volume 85 of *LNCS*, pages 632–644. Springer, 1980.

A Threshold Implementations of p_T and p_χ

Figure 7 shows the algorithmic representation of a securely masked (using three shares) and single-fault SIFA-protected Toffoli gate p_{TT} 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 threshold implementation of $p_{\chi T}$ follows analogously. A secure hardware variant of the Toffoli gate is shown in Figure 8. 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.

```
Name: $p_{TT}$
State: $\{a_0, a_1, a_2, b_0, b_1, b_2, c_0, c_1, c_2\}$
$p_T(a_0, b_0, c_0)$
$p_T(a_0, b_0, c_2)$
$p_T(a_0, b_2, c_0)$

$p_T(a_1, b_1, c_1)$
$p_T(a_1, b_1, c_0)$
$p_T(a_1, b_0, c_1)$

$p_T(a_2, b_2, c_2)$
$p_T(a_2, b_2, c_1)$
$p_T(a_2, b_1, c_2)$
```

Figure 7: Algorithmic representation of masked Toffoli gate (p_{TT}) using 3 shares.

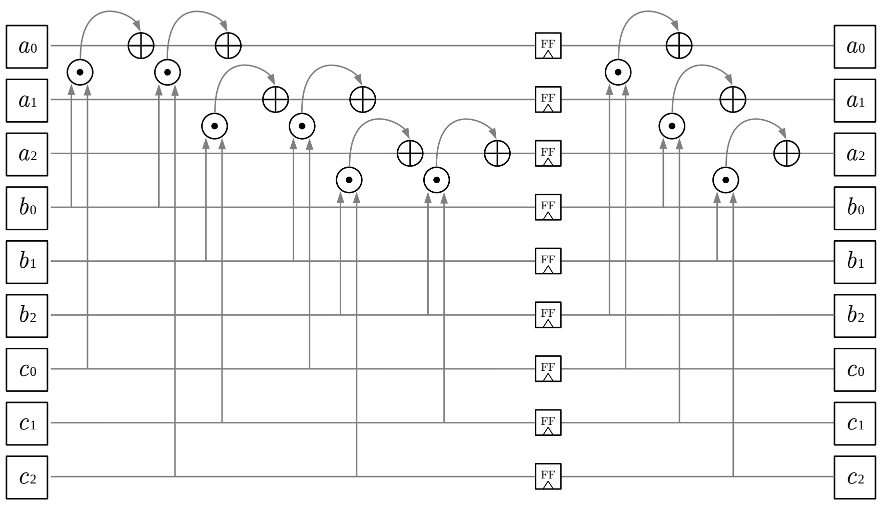


Figure 8: 3-share TI and single-fault SIFA-protected Toffoli gate in hardware.

B Subcircuits for AES S-box

In this section, we give a description of the required basic circuits that we use in the AES S-box. This description is fully based on the work of Canright [Can05]. Again, we denote the higher half of a variable with the superscript H and the lower half with L, e.g, for a four bit field element $x \in \mathbb{F}_{2^4}$, the higher two bits (coefficients) are denoted with x^H and the lower with x^L. Alternatively, we denote each single bit with x^3, x^2, x^1, and x^0. Note that we just give the algorithmic description of the building blocks and do not consider their implementation.

We start with the description of the linear permutations $\Gamma(x)$ and $\Xi(x)$. Both are binary matrix multiplications. For $\Gamma(x)$, we have according to Canright [Can05]:

Name: Γ

State: $\{x\}$

$$\begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix}$$

and for $\Xi(x)$:

Name: Ξ

State: $\{x\}$

$$\begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix}$$

For the square scaling $s^4_{sc}(a, b, c)$, we denote with $\|$ a concatenation and get:

Name: s^4_{sc}

State: $\{a, b, c\}$

$\mathrm{T}_0 \leftarrow b \oplus c$

$a \leftarrow a \oplus ((\mathrm{T}_0^0 \oplus \mathrm{T}_0^2) \| (\mathrm{T}_0^3 \oplus \mathrm{T}_0^1) \| (\mathrm{T}_0^0 \oplus \mathrm{T}_0^1) \| \mathrm{T}_0^0)$

For the square scaling $s^2_{sc}(a, b, c)$, we get:

Name: s^2_{sc}

State: $\{a, b, c\}$

$\mathrm{T}_0 \leftarrow b \oplus c$

$a \leftarrow a \oplus (\mathrm{T}_0^1 \| (\mathrm{T}_0^0 \oplus \mathrm{T}_0^1))$

The inversion a^{-1} and squaring a^2 in $GF(2^2)$ can be done as bit-swap, where $\|$ denotes concatenation:

Name: Inv
State: $\{a\}$
$a \leftarrow a^0 \| a^1$

The constant addition with the hexadecimal value `0x63` can be done as:

Name: AddConstant
State: $\{a\}$
$a \leftarrow a \oplus \texttt{0x63}$

The multiplication $c \leftarrow a \cdot b$ in $GF(2^2)$ can be done as:

$$c^1 \leftarrow ((a^1 \oplus a^0) \odot (b^1 \oplus b^0)) \oplus (a^1 \odot b^1)$$
$$c^0 \leftarrow ((a^1 \oplus a^0) \odot (b^1 \oplus b^0)) \oplus (a^0 \odot b^0)$$

The multiplication $c \leftarrow a \cdot b$ in $GF(2^4)$ is a bit more complex. In the following algorithm, $\cdot$ denotes above multiplication in $GF(2^2)$ and $\|$ a concatenation:

$$\mathrm{T}_0 \leftarrow (a^H \oplus a^L) \cdot (b^H \oplus b^L)$$
$$\mathrm{T}_1 \leftarrow \mathrm{T}_0^0 \| (\mathrm{T}_0^0 \oplus \mathrm{T}_0^1)$$
$$c^H \leftarrow \mathrm{T}_1 \oplus (a^H \cdot b^H)$$
$$c^L \leftarrow \mathrm{T}_1 \oplus (a^L \cdot b^L)$$

C Inverse of Masked AES S-box

The S-box in Circuit 8.b is still a permutation on its inputs. This can be seen by showing that the implementation is invertible. Given a choice for $e_0, f_0, g_0, h_0, y_0, y_1$, we can do the computation of Figure 9.

$$
\begin{aligned}
&x_0 = \Gamma^{-1}(e_0)\\
&x_1 = x_0 + S^{-1}(y_0 + y_1)\\
&e_1 = \Gamma(x_1)\\
&\textsc{t}_0 = 0\\
&s_{sc}^4(\textsc{t}_0, e_0^H, e_0^L)\\
&a_0 = f_0 + \textsc{t}_0 + e_0^H \cdot (e_0^L + e_1^L)\\
&a_1 = a_0\\
&\textsc{t}_1 = 0\\
&s_{sc}^4(\textsc{t}_1, e_1^H, e_1^L)\\
&f_1 = a_1 + \textsc{t}_1 + e_1^H \cdot (e_0^L + e_1^L)\\
&\textsc{t}_0 = 0\\
&s_{sc}^2(\textsc{t}_0, f_0^H, f_0^L)\\
&b_0 = g_0^{-1} + \textsc{t}_0 + f_0^H \cdot (f_0^L + f_1^L)\\
&b_1 = b_0\\
&\textsc{t}_1 = 0\\
&s_{sc}^2(\textsc{t}_1, f_1^H, f_1^L)\\
&g_1 = (b_1 + \textsc{t}_1 + f_1^H \cdot (f_0^L + f_1^L))^{-1}\\
&c_0^H = h_0^H + f_0^L \cdot (g_0 + g_1)\\
&c_1^H = c_0^H\\
&h_1^H = c_1^H + f_1^L \cdot (g_0 + g_1)\\
&c_0^L = h_0^L + f_0^H \cdot (g_0 + g_1)\\
&c_1^L = c_0^L\\
&h_1^L = c_1^L + f_1^H \cdot (g_0 + g_1)\\
&d_0^H = \Xi^{-1}(y_0^H + \texttt{0x6}) + e_0^L \cdot (c_0 + c_1)\\
&d_1^H = \Xi^{-1}(y_1^H) + e_1^L \cdot (c_0 + c_1)\\
&d_0^L = \Xi^{-1}(y_0^L + \texttt{0x3}) + e_0^H \cdot (c_0 + c_1)\\
&d_1^L = \Xi^{-1}(y_1^L) + e_1^H \cdot (c_0 + c_1)
\end{aligned}
$$

Figure 9: Inverse computation to show that Circuit 8.b is a permutation.