StaTI: Protecting against Fault Attacks Using Stable Threshold Implementations

Siemen Dhooghe[0000−0003−0591−7355], Artemii Ovchinnikov[0000−0002−9035−523X], and Dilara Toprakhisar[0000−0003−4551−6775]

COSIC, KU Leuven, Leuven, Belgium
firstname.lastname@esat.kuleuven.be

Abstract. Fault attacks impose a serious threat against the practical implementations of cryptographic algorithms. *Statistical Ineffective Fault Attacks* (SIFA), exploiting the dependency between the secret data and the fault propagation overcame many of the known countermeasures. Later, several countermeasures have been proposed to tackle this attack using error detection methods. However, the efficiency of the countermeasures, in part governed by the number of error checks, still remains a challenge.

In this work, we propose a fault countermeasure, *StaTI*, based on threshold implementations and linear encoding techniques. The proposed countermeasure protects the implementations of cryptographic algorithms against both side-channel and fault adversaries in a non-combined attack setting. We present a new composable notion, *stability*, to protect a threshold implementation against a formal *gate/register-faulting adversary*. Stability ensures fault propagation, making a single error check of the output suffice. To illustrate the stability notion, first, we provide stable encodings of the XOR and AND gates. Then, we present techniques to encode threshold implementations of S-boxes, and provide stable encodings of some quadratic S-boxes together with their security and performance evaluation. Additionally, we propose general encoding techniques to transform a threshold implementation of any function (*e.g.*, non-injective functions) to a stable one. We then provide an encoding technique to use in symmetric primitives which encodes state elements together significantly reducing the encoded state size. Finally, we used StaTI to implement a secure Keccak on FPGA and report on its efficiency.

1 Introduction

Cryptographic algorithms that are designed to resist cryptanalytic attacks are still prone to physical attacks. These attacks target the implementations of the algorithms when deployed in an embedded device. Such attacks can be grouped into two categories: (1) passive observation of the device's behavior (*e.g.*, power consumption [KJJ99], timing [Koc96], and electromagnetic emanation [GMO01]), and (2) inducing errors in the computation through a physical means (*e.g.* clock/voltage glitching [AK97], electromagnetic waves [DDRT12],

and laser injections [Hab65]) and observing the device's response to the induced errors.

Side-Channel Analysis (SCA) is a passive attack technique that exploits information leakage from physical effects, such as timing and power consumption. Over the years, numerous attacks have been proposed, and countermeasures protecting against these attacks have been developed. Among the most prominent countermeasures employed to protect against SCA is masking [CJRR99, ISW03, RBN+15, GMK16]. The idea of masking is to split the secret input into a number of shares such that each share is statistically independent of the secret. Consequently, observing all but one shares does not compromise the privacy of the secret input. Threshold Implementations (TI) by Nikova *et al.* [NRR06] is such a technique that is based on secret sharing and threshold cryptography. The computation is performed by coordinate functions that operate on non-complete sets of shares of the secret input. TI is designed for hardware implementations, making it effective even when the circuit is affected by glitches.

In contrast to the passive nature of SCA, fault attacks actively disrupt the computation. Since the seminal work of Boneh *et al.* [BDL97] introducing fault attacks on RSA, numerous attack techniques targeting the physical properties of the implementations of cryptographic primitives have been developed. To mitigate these attacks, various countermeasures have been designed, with redundancy emerging as an extensively employed technique. Redundancy (in time, area, or information) is utilized to detect whether a fault is injected into a circuit. Upon fault detection, these countermeasures either suppress or infect the output, such that it is no longer exploitable by the adversary. Additionally, redundancy can also be leveraged for error correction purposes, such as majority voting. Another promising approach that has gained significant adoption is the combination of redundancy and masking. These countermeasures, combining masking and redundancy, were typically considered as secure against SCA and fault attacks. However, a novel attack, Statistical Ineffective Fault Attacks (SIFA) [DEK+18] that is performed on non-faulty (*i.e.,* correct) ciphertexts has been proposed. The attack exploits the dependency between the fault propagation to the output and the secret values, thereby circumventing simple redundancy. Building upon this, Dobraunig *et al.* [DEG+18] used SIFA to overcome most of the redundancy combined with masking based countermeasures proposed prior to the introduction of SIFA. Examples of such combined countermeasures include, among others, ParTI [SMG16], CAPA [RMB+18], Private Circuits II [IPSW06], M&M [MAN+19], Impeccable Circuits [AMR+20, SRM20, RSM21], and Transform-and-Encode (TaE) [SJR+19], which were designed both before and after the introduction of SIFA. Taking a different approach, Daemen *et al.* [DDE+20] proposed the use of reversible operations to ensure the propagation of the fault to the output, and error detection methods to detect the faults at the output. However, as noted in [DDE+20], the implementation of such countermeasures becomes inefficient as the complexity of the protected functions increase.

Contributions. In this paper, we propose a fault attack countermeasure, *StaTI*, that extends threshold implementations using linear encoding. In *StaTI*, extending the notions of threshold implementations, we propose a new composable notion, *stability*, that ensures the propagation of the injected faults to the output. As a result, the injected faults can be detected using a single error detection circuit at the end. *StaTI* can be applied to any threshold implementation following the respective technique proposed in this paper.

We formally define the gate/register-faulting adversary and the respective security model. Based on these models, we present encodings for the basic building blocks XOR and AND gates which propagate any injected fault to the output. Consecutively, we propose general methodologies to encode the threshold implementation of any function that does not propagate the faults, such that they become secure under the adversary and security models we define. These methodologies ensure the fault propagation to the output in two different ways:

- Explicitly detecting faults at the input of each encoded function.
- Mapping the faulty inputs of an encoded function to the faulty outputs.

These methodologies provide generic protection against the proposed adversary, meaning we can defend a function against effective and ineffective faults at the same time, and the methodology can be applied to any function.

On the basis of the *stability* notion, we present efficient encoding techniques for $(t+1)$ share and two share threshold implementations for any degree t function. Consecutively, to assess the security and the efficiency of our methodology, we present the stable encodings for the quadratic class $\mathcal{Q}_{12}^4$ and the Keccak S-box using duplication and the parity code. The duplication of the Keccak S-box is then used for an FPGA implementation of Keccak-f[1600]. To assess their security, we make use of `VerMFI` [ANR18, AWMN20]. Our assessment shows that *StaTI* achieves first-order gate/register-faulting security in addition to the probing security of threshold implementations. Moreover, it brings no additional latency, and roughly a factor of two overhead in area cost compared to regular threshold implementations. Lastly, following the encoding techniques for S-boxes, we provide state-wide encodings to be employed in a symmetric primitive. This technique groups state elements and encode them together reducing the encoded state size compared to encoding state elements separately.

Outline. In Section 2, we discuss the adversary and security models that *StaTI* assumes, and the masking and linear code concepts that *StaTI* makes use of. Additionally, we introduce statistical ineffective faults and go over some countermeasures against it. In Section 3, we describe a new property, *stability*, that allows efficient encodings while protecting against statistical ineffective faults. Then, we present stable encodings of the XOR and AND gates in Section 4 based on the security assumptions made in Section 2. Next, we present general methodologies to implement stability for unstable functions in Section 5. We provide a methodology to obtain stable encodings of $t+1$-shared threshold implementations of degree t functions in Section 6, and examples of two-shared threshold permutations in Section 7. Then, in Section 8 we provide state-wide encodings

that would be employed in a symmetric primitive. Finally, in Section 9 we evaluate the security and the performance of the $t+1$ and two share encodings of 4-bit quadratic functions and the 5-bit Keccak S-box presented in the appendix. Moreover, we present a security and performance evaluation of Keccak$-f[1600]$ protected with *StaTI*.

2 Preliminaries

In this section, we introduce the probing and the gate/register-faulting adversaries together with their respective security models assumed in this work. We then introduce Boolean masking and linear codes as a countermeasure against the two adversaries, respectively. Finally, we introduce threshold implementations protecting the computation against side-channel attacks by masking, and the fault attack countermeasure proposed by Daemen *et al.* [DDE+20] protecting the computation on masked and encoded data using permutations.

2.1 Adversary and Security Models

We consider an adversary with probing and faulting capabilities. The attack surface is considered as a Boolean circuit that is a directed acyclic graph whose vertices are Boolean gates, and whose edges are wires. The Boolean circuit takes an input, has an internal state, and produces an output where the state corresponds to the sharing of the secret data stored in the registers.

Wire Probing. To capture an attacker performing side-channel analysis, we consider the d-probing model as introduced by Ishai *et al.* [ISW03]. This adversary can observe at most d wires of the Boolean circuit and these wires are specified before the circuit is computed. Specifically in this work, we consider the first-order probing security.

In order to capture the physical effect of glitches on a hardware platform, we extend the probing model to the glitch-extended robust probing model from Faust *et al.* [FGP+18]. In this extended model, a glitch-extended probe allows obtaining all the registered inputs leading to the gate/wire which is probed.

Considering the security against the probing adversary, we follow a simulation model as first proposed by Ishai *et al.* [ISW03]. In order to prove security, a simulator is created which needs to simulate the probed wire values of the circuit without having the secret input of the circuit (*e.g.* the key and the plaintext of a block cipher). Such a simulation can be successful in case the circuit uses randomness as a countermeasure. This randomness is not given to the adversary and, as a result, the simulator can use it to fool the adversary. In practice, a proof of simulation security comes down to showing the distribution of the probed wires is independent of the input of the circuit.

Gate/Register Faulting. To capture an attacker performing a fault attack, we consider a k-gate/register-faulting adversary which can replace the outputs of a total of k gates or registers in the circuit. When considering Boolean circuits, this means that the adversary can replace a total of k gates or registers to either: output zero (reset faults); output one (set faults); or flip the output (bitflip faults). While there is currently no standard adversary model in the literature, the gate/register-faulting adversary is used in many different works [AMR+20, SRM20, RSM21]. These works assume an adversary that can fault a number of wires in a clock cycle. As each wire is an output of a gate or register, the faults are modeled as modifications on gates or registers. Nevertheless, in this work we consider the first-order gate/register-faulting security. That is, we consider an adversary which is capable of replacing one gate or register in the circuit during the whole computation.

Considering the security models tied to the gate/register-faulting model, we require that the circuit is *correct* and *private*. A circuit is correct against a k-gate/register-faulting adversary when, over all possible faults, the circuit always gives back a correct output or an abort signal. A circuit is called private when, for every fixed injected fault, the probability of the abort signal does not depend on the secret inputs of the circuit. In fact, similar to the probing model, we consider a simulation game where the simulator is given the faults and needs to simulate the abort signal.

2.2 Masking and Encoding

In order to thwart probing and faulting adversaries, we make use of masking and encoding techniques. Boolean masking is a technique based on splitting each secret variable $x \in \mathbb{F}_2$ in the circuit into s_x shares $\bar{x} = (x_0, x_1, \ldots, x_{s_x-1})$ such that $x = \sum_{i=0}^{s_x-1} x_i$ over $\mathbb{F}_2$. A random Boolean masking of a fixed secret is uniform if all sharings of that secret are equally likely. Before moving on to encoding a circuit to protect against faults, we describe some linear code notions that we use in this work.

Definition 1 (Binary linear code). *A binary linear $[n, k, d]$-code $\mathcal{C}$ is a vector subspace over $\mathbb{F}_2^n$, where n denotes the length, k denotes the rank, and d denotes the minimum distance of the code which is the minimum Hamming distance between two distinct codewords of $\mathcal{C}$.*

An $[n, k, d]$-code $\mathcal{C}$ maps messages $x \in \mathbb{F}_2^k$ to codewords $c \in \mathcal{C} \subset \mathbb{F}_2^n$ and its error detection capability (*i.e.,* the number of faults which can be detected) is determined by d.

Definition 2 (Generator matrix and encoding). *A matrix $G \in \mathbb{F}_2^{kxn}$ is said to be a generator matrix for the binary linear $[n, k, d]$-code $\mathcal{C}$ if it consists of k basis vectors of $\mathcal{C}$ with length n. Then, $\mathcal{C}$ has an encoding map $C : \mathbb{F}_2^k \mapsto \mathbb{F}_2^n$ which is a linear transformation of the form $x \mapsto xG$.*

More specifically, in this work, we consider systematic codes.

Definition 3 (Systematic code). *A linear code $\mathcal{C}$ is a systematic code if its generator matrix G is of the form $[I_k|P]$ where I_k is the $k \times k$ identity matrix, and P is some $k \times (n-k)$ matrix.*

A systematic $[n, k, d]$-code, denoted by $\mathcal{C}$, is used to encode $x \in \mathbb{F}_2^k$ to $\langle x, x' \rangle \in \mathbb{F}_2^n$ where x is the message, and x' is the check bits such that $x' = xP$. Then, we call $\langle x, x' \rangle$ a codeword of $\mathcal{C}$. In order to transform $\langle x, x' \rangle$ to another correct codeword, at least d bits need to be flipped. Consider an element $y \in \mathbb{F}_2^n$, in case $y \notin \mathcal{C}$, we call y "faulty". We make use of the parity check matrix to detect if a received codeword is faulty or not.

Definition 4 (Parity check matrix). *A matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ is said to be a parity check matrix of the $[n, k, d]$-code $\mathcal{C}$ such that $Hc^T = \mathbf{0}$ if and only if $c \in \mathcal{C}$.*

When encoding and masking are combined to thwart probing and faulting adversaries, the order of encoding and masking is important. That is, first encoding the state of the cipher and then sharing it, versus first sharing the state of the cipher and then encoding it are interpreted differently. In this work, we first share the state of the cipher (denoted by $\bar{x}$) and then encode the shares (where the parity bits are denoted by $\bar{x}'$). For example, a bit $x \in \mathbb{F}_2$ is first shared into $x_0, x_1 \in \mathbb{F}_2$ such that $x_0 + x_1 = x$, and then encoded to $\langle x_0, x_1, x_0, x_1 \rangle$ meaning that each share is duplicated. In the rest of the work, $\tilde{x}$ denotes the shared and encoded variables, and $\tilde{F}$ denotes the shared and encoded functions.

2.3 Threshold Implementations

In this paper, we use the notions of threshold implementations as introduced by Nikova *et al.* [NRR06]. As such, we introduce the notions of correctness, non-completeness and uniformity.

Correctness ensures that the sum of the output shares yields the desired output. A shared function is non-complete if each of its coordinate functions f_i operate on data independent from the input secret. This notion has been extended by Bilgin *et al.* [BGN+14] to capture each set of d coordinate functions being jointly non-complete.

Definition 5 (d^{th}-order non-complete [NRR06, BGN+14]). *A shared function is d^{th}-*order non-complete *if any d coordinate functions f_i jointly work on data independent from the input secret.*

The above notion was created as a necessary property to secure maskings against higher-order univariate attacks including the effect of glitches. The notion still holds for when the input is linear encoded and masked.

We call, for a fixed secret x, a shared variable $\bar{X}$ that is assigned to a sharing $\bar{x}$ depending on the randomness, is uniform if it is uniformly distributed over the set of all sharings of x denoted $Sh(x)$.

Definition 6 (Uniform sharing). *A shared variable $\bar{X}$ is* uniform *when*

$$\Pr[\bar{X} = \bar{x} \,|\, \bar{x} \in Sh(x)] = 1/|Sh(x)|\,.$$

A shared function is called uniform if, when given a uniform input sharing, it outputs a sharing which is uniform.

Definition 7 (Uniformity [NRR06]). *A shared function* $\bar{F}(\bar{x}) = \bar{y}$ *is uniform if* $\forall x \in \mathbb{F}$, $\forall \bar{y} \in Sh(F(x))$:

$$\left| \left\{ \bar{x} \in Sh(x) \,\middle|\, \bar{F}(\bar{x}) = \bar{y} \right\} \right| = \frac{|Sh(x)|}{|Sh(F(x))|} .$$

Note that in order to verify whether an encoded function is uniform, we perform the verification only over the correct codewords. In other words, for encoded and shared variables, $Sh(x)$ is the set of all correct codewords of share vectors of x.

2.4 Statistical Ineffective Faults and Countermeasures

In this section we summarize Statistical Ineffective Fault Attacks (SIFA) by Dobraunig *et al.* [DEK+18, DEG+18] and introduce the strategy proposed by Daemen *et al.* [DDE+20] to protect circuits against it on which we base our countermeasure.

SIFA. SIFA [DEK+18] exploits the dependency between the propagation of an injected fault and the secret value. Following the initial attack, Dobraunig *et al.* exploited SIFA on implementations protected using masking together with simple error detection [DEG+18], and showed that the combined countermeasures based on simple redundancy and masking are not sufficient to protect against SIFA. We distinguish these two attacks as done by Saha *et al.* [SJR+19]: SIFA-1 and SIFA-2. SIFA-1 assumes an attacker injecting a statistically biased fault (*i.e.,* set or reset) to the state variable or linear operations. Whereas, SIFA-2 assumes an attacker injecting an unbiased fault (*i.e.,* bit flip) in the nonlinear operations like an S-box. With respect to the gate/register fault model, we can model SIFA-1 and SIFA-2 adversaries as faulting a register that stores the cipher state, or a (gate/register in a) linear operation using set or reset faults, and faulting a (gate/register in a) nonlinear operation using a bit flip, respectively. Dobraunig *et al.* apply SIFA-2 to a TI of AND gate with four shares to show the impact of SIFA-2 on TI:

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

A bitflip to the input x_0 will be ineffective if only if the shares y_0, y_1, y_2, and y_3 are zero which indicates that the secret input y is zero. That is, the fault causes a fault-free ciphertext depending on a secret value. Therefore, the dependence of the fault propagation to a secret value harms the privacy of the circuit.

Combined Countermeasures. Masking and redundancy (linear codes) have been the main building blocks in countermeasures defeating both fault attacks and side channel analysis both prior to and after the introduction of SIFA-1 and SIFA-2. Impeccable Circuits I [AMR+20] and ParTI [SMG16] employ masking and error detection. Impeccable Circuits II [SRM20] and III [RSM21], DOM-REP [GPK+21], the countermeasure from Breier *et al.* [BKHL20], and Combined Private Circuits [FGM+23] employ masking and error correction codes to output correct ciphertexts despite the fault injection. Impeccable Circuits III combines error detection and correction to relax the high cost of error correction codes. TaE [SJR+19] introduces a different approach which consists of two phases: domain transformation that randomizes the state in each cipher execution, and using a repetition code to correct errors. Masking and majority voting are the techniques used in the countermeasure for both phases, respectively. CAPA [RMB+18] employs a multiparty computation protocol using MAC tags that performs intermediate checks to ensure the circuit is not disturbed by a fault injection. Despite its effectiveness against SIFA, CAPA has a high implementation cost in practice. On the other hand, M&M [MAN+19] being inspired by CAPA in terms of employing MAC tags, does not protect against ineffective faults despite the efficiency gain compared to CAPA. We mention other countermeasures that protect against both fault attacks and side channel analysis such as RS-mask [RAD20] which randomizes the computation, Bringer *et al.* [BCC+14] utilizes orthogonal direct sum masking, Patranabis *et al.* [PRC+19] uses fault space transformation, Bhasin *et al.* [BDF+09] uses dual rail with precharge logic styles, and Breier *et al.* [BH17] uses error correction codes. In the work by Daemen *et al.* [DDE+20] a different approach is followed which makes use of Toffoli gates as basic primitives to implement nonlinear functions. We base our work on [DDE+20] and explain it further.

Protecting against Statistical Ineffective Faults [DDE+20]. The countermeasure by Daemen *et al.* protects against circuit faults which modify the circuit such that it returns a faulty output for at least one input using Toffoli gates and masking. The authors propose two countermeasures to achieve protection against SIFA:

- A circuit is recursively split into interconnected subcircuits which consist of simple operations such as addition and multiplication. Then, each subcircuit is required to be a permutation such that the injected fault to a subcircuit needs to propagate to the output of that subcircuit. As a result, due to the interconnected nature of subcircuits, a single error check at the end of the composite circuit is enough to detect the fault. In addition, each subcircuit is required to be non-complete such that the injected fault causing a wrong output does not depend on any secret value.
- A fault detection circuit for arbitrary circuits is added. Namely, the inputs of nonlinear nodes are error checked.

The first method bears resemblance to the properties of threshold implementations as a Boolean function property, namely the requirement for the encoding to be a permutation.

In order to secure a circuit using their methodology, the circuit is separated in linear functions and Toffoli gates (the gate mapping $(a, b, c) \in \mathbb{F}_2^3$ to $(a, b, c+ab) \in \mathbb{F}_2^3$). Each linear function and Toffoli gate is then masked and encoded using a duplication code.

3 Stability

In this section, we tweak the notions of security from [DDE+20] to allow more efficient encodings and thereby, introduce the *stability* notion that ensures the propagation of the injected faults to the output.

As a first step, we consider a different adversary model compared to the model by Daemen *et al.* In their original model, a fault is modeled at the circuit level as a deviation of the circuit instance from the fact that its output is *fully* determined by only its input. A circuit being described as a composition of basic circuits (*i.e.*, circuits consisting of simple operations such as addition and multiplication), an adversary can target the entire basic circuit. Then, the fault modifies the circuit such that it returns a faulty output for at least one input. On the other hand, in this work as discussed in Section 2.1, we consider faults which are injected to the output of a gate or a register. We emphasize that the model we use has been used in many other works such as [AMR+20, SRM20, RSM21].

Since we are discussing threshold implementations, we consider their circuit representation such that each shared function (register to register) is non-complete and the functions are jointly uniform. Each input share is given to several non-complete coordinate functions as depicted in Figure 1.

However, we require an additional property from the circuit representation of a threshold implementation when considering fault attacks. Namely, we require that no gate is shared between two coordinate functions, *i.e.,* the coordinate functions are a partition of the gates used in that stage of the computation. We call this property *fault non-completeness*.

Definition 8 (Fault non-completeness). *A circuit is fault non-complete if, per register stage, each gate of the circuit drives only one output share.*

Fault non-completeness ensures that a fault in a gate only targets one output share. Thus, the propagation of a fault to the output does not depend on a secret value. This property can be relaxed such that a fault in a gate only affects a non-complete set of output shares and can not change the correctness of the output. However, this relaxation is left out for simplicity. Fault non-completeness alone, as shown in Section 2.4, is not sufficient to secure the composition of encoded functions.

For a second step, we look at the secure composition of encoded functions. Recall from the work by Daemen *et al.* that a circuit can be secured against SIFA by adding error checks to the circuit or having each subcircuit being a permutation. We adopt these two methods for Boolean functions similar to the properties of threshold implementations. We propose a new property, *stability*, that provides security against a gate/register-faulting adversary when combined

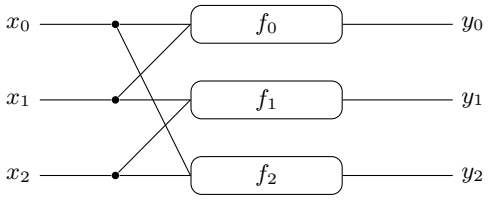

Fig. 1. Example circuit model in hardware for a three shared non-complete function. Each combinational gate is part of exactly one coordinate function f_i.

with correctness, non-completeness, fault non-completeness, and uniformity, (see Theorem 1) and generalizes the two methods by Daemen *et al.* into a single notion.

Definition 9 (Stability). *Consider a shared and encoded register-to-register function $\tilde{F} : \mathbb{F}_2^n \to \mathbb{F}_2^n$ and a code $\mathcal{C} \subset \mathbb{F}_2^n$. We call $\tilde{F}$* stable *if for any $\tilde{x} \in \mathcal{C}$ and $e \notin \mathcal{C}$, $\tilde{F}(\tilde{x} + e) \notin \mathcal{C}$.*

Stability states that any incorrect input codeword is mapped to an incorrect output codeword. In particular, a correct and injective shared function (*e.g.*, permutations) is stable as it does not map any incorrect codeword to the correct ones. It is important to note that a stable implementation of a correct and injective shared function can be considered as a specific instance of TaE [SJR+19]. This is due to the stability naturally exhibited by such functions, and therefore, they do not require additional measures other than simple masking and encoding to ensure stability. In this instance, the "Transform" phase is implemented with TI, and the "Encode" phase is implemented with linear codes.

The above definition is naturally extended when the number of outputs is different from the number of inputs. In this case, the output linear code differs from the input one. We cover this case in Section 7 which requires additional measures besides masking and encoding making it differ from an instance of TaE.

This notion together with the TI notions additionally provides security over the serial composition of shared and encoded functions.

Theorem 1. *The serial composition of correct, stable, non-complete, fault non-complete, and uniform shared functions is secure against a probing adversary and a single gate/register-faulting adversary in a non-combined attack scenario.*

Proof. The security against a probe placed in the design follows from the non-completeness and uniformity properties of threshold implementations and was proven by Dhooghe *et al.* [DNR19].

We then consider a single gate/register-faulting adversary. We analyze gate and register faults separately. From the circuit representation, we know that each gate belongs to exactly one coordinate function of $\tilde{F}$ (fault non-completeness). As a result, we allow the adversary to arbitrarily change (fault) this coordinate

function using a gate fault. Since the coordinate function is non-complete, the output $\tilde{F}(\tilde{x})$ is faulted by an additive difference e which does not depend on the secret value x. Due to the interconnected nature of the circuit, this faulty output is either the output of the whole circuit, or an input to the next function. Being the output of the whole circuit does not pose any security problems as we already proved that the fault does not depend on the secret. If the output is an input to the next function, then we can model it as a single register fault on this function. As only one share is affected by the fault, we safely assume that $e \notin \mathcal{C}$ as a single share fault cannot craft a valid encoding. Additionally, a single register fault does not depend on any secret. Since the circuit is the serial composition of stable functions, and $e \notin \mathcal{C}$, the fault is propagated to the circuit output. As a result, the fault is independent of any secret and is detected at the end of the composite circuit making the circuit abort. □

Stability is an independent notion that ensures the propagation of the faults present in the input to the output in a shared and encoded register-to-register function. However, Theorem 1 establishes the necessity of additional notions (*e.g.,* TI notions) for probing and gate/register-faulting security. For instance, non-completeness ensures that the injected gate faults do not harm the privacy of the circuit. Consequently, stability cannot be readily extended to other masking schemes to provide the desired security. That is, additional implementation constraints should be considered to ensure that any injected gate fault yields an incorrect output codeword, and its propagation is independent of the secret data. For example, this can be ensured by utilizing forced independence [AMR+20], where no gate is shared by two distinct output bits.

Considering the security model presented in Section 2.1, Theorem 1 states that a stable threshold implementation of a function is correct and private against a single gate/register-faulting adversary. This implies that the implemented circuit always gives back a correct output or an abort signal over all possible single gate/register faults. Additionally, for every fixed injected single fault, the probability of the abort signal does not depend on the secret inputs of the circuit. Consequently, a stable threshold implementation of a function is secure against the attacks exploiting the wrong outputs, such as DFA [BS97] or SFA [FJLT13], as the circuit is correct and does not give wrong outputs. Similarly, a stable threshold implementation of a function is secure against the attacks exploiting the data dependency of fault activation/propagation to the output, such as SIFA [DEK+18,HPB22], FTA [SBR+20] or SEFA/SHFA [VZB+22]. This is attributed to the fact that the implementation is private and the abort signal is independent of the secret data.

4 Stable Encodings of XOR and AND Gates

The XOR and AND gate comprise the two basic operations over the field $\mathbb{F}_2$. As a result, they are the corner stones in cryptographic circuits (such as in block ciphers). Nevertheless, these gates (similar to the other basic gates such

as OR or NAND gates) have a fan-in of two and a single output. Thus, it would not be possible to secure such gates in the framework by Daemen *et al.* without increasing the output size or performing intermediate error checks as the masking and encoding of it can never be injective.

In this section, we show that with the notion of stability (Definition 9) it is possible to secure basic gates without resorting to error checks or adding additional output wires. More specifically, we provide masked functions for the XOR and the AND gates encoded with a duplication code.

Consider a naive XOR gate encoded with a duplication code. Take the encoded input (a, b, a', b') where $a = a'$ and $b = b'$ if the input is correctly encoded (*i.e.,* the input is a valid codeword). Then, the naive XOR gate computes $(a+b, a'+b')$. This encoding of the XOR is not stable as an injected fault vector $e = (1, 1, 0, 0)$, which is not a valid codeword as $(1, 1) \neq (0, 0)$, to the input yields an output which is still a valid codeword as $a + 1 + b + 1 = a + b = a' + b'$. In fact, an XOR gate will always propagate the error to the output as long as $e \neq (1, 1, 0, 0), (0, 0, 1, 1)$. Therefore, we propose the following encoded XOR gate with a duplication code which modifies the output if and only if $e = (1, 1, 0, 0)$ or $(0, 0, 1, 1)$:

$$
\begin{aligned}
z &= a + b + (a + a')(b + b') \\
z' &= a' + b'
\end{aligned}
$$

For the other injected fault vectors that are invalid codewords, no modification is done as the naive XOR gate already propagates them to the output. Therefore, this encoding maps every invalid input codeword to an invalid output codeword while leaving the correctness of the function unaltered, which makes it stable.

Similarly, consider a naive AND gate encoded with a duplication code. Given the input $(a, 0, a', 0)$, a valid codeword, the gate computes $(a \cdot 0, a' \cdot 0) = (0, 0)$. Considering an injected fault vector $e = (1, 0, 0, 0)$, which is not a valid codeword as $(1, 0) \neq (0, 0)$, to the input, the output of the gate is still a valid codeword as $(a + 1) \cdot 0 = 0 = a' \cdot 0$. In fact, if one of the inputs is zero, a fault injected to the other input will not propagate to the output. Therefore, we propose the following encoded AND gate with a duplication code which propagates the fault only if one or two of the inputs are zero:

$$
\begin{aligned}
z &= ab + (a + a')(b' + 1) + (b + b')(a' + 1) \\
z' &= a'b'
\end{aligned}
$$

For the other injected fault vectors that are invalid codewords, no modification is done as the naive AND gate already propagates them to the output. Therefore, this encoding maps every invalid input codeword to an invalid output codeword while leaving the correctness of the function unaltered, which makes it stable.

The stable encoding of the XOR gate is trivially extended to work over shared inputs and keep its stability. However, the case of a stable shared and encoded AND gate is more complex and is provided for two-shared threshold implementation in Section 7.

5 General Methodologies for Stability

In this section, we provide methodologies that transform unstable (Definition 9) encodings into stable encodings without affecting the non-completeness, uniformity, and correctness of the sharings.

5.1 Error Detection of Unstable Functions

Finding a stable encoding for an injective function is trivial using a simple linear code. However, finding stable encoding for a non-injective function is relatively challenging. The simplest examples are the XOR and AND gates. In Section 4, stable encodings for these gates were presented. This is a notable advancement, but it does not provide us guidance on securing an arbitrary non-injective encoding. In this section, we present a general solution to transform an unstable sharing $\tilde{F}$ to a stable one.

Consider a correct, non-complete, and uniform encoded sharing $\tilde{F}$ of F. We add a single value to the design denoted by R which holds the current state of the circuit. Meaning, if $R = 0$, the state of the circuit is correct and no faults are present. If $R \neq 0$ then a fault was detected. In other words, R expands our encoded state with additional parity bits. In case these bits are non-zero, then the state is not a valid codeword. At the end of the circuit's computation, the decoder verifies the value of R and aborts in case $R \neq 0$. Given this extra value, making a stable sharing $\tilde{F}$ from F is straightforward. Namely, one can update the error status of the circuit R, which is initially set to zero, by error checking the input of $\tilde{F}$ using the parity check matrix H and ORing the result to R. Consider an encoded value $\tilde{x}$ with its parity check matrix H and the output error state R_{next}, then the following sharing is stable:

$$\tilde{F}(\tilde{x}) = \tilde{y}$$
$$R_{next} = R \vee H\tilde{x}^T$$

It is clear that the above sharing is stable as any input error or $R \neq 0$ causes $R_{next} \neq 0$ which means that the output is not a valid codeword of the expanded linear code.

We note that it is also possible to combine this method with the changing of the guard's construction by Daemen [Dae17]. In this case, the construction ensures both uniformity and stability of a sharing.

5.2 Interpolation

Considering a shared and encoded value $\tilde{x}$ and a function F, we show that via interpolation it is possible to find a map $\tilde{F}$ which is correct and stable for F.

Consider the list of F's correct encoded inputs and outputs, namely $(\tilde{x}, \tilde{y}) \in \mathcal{C}^2$ such that $\tilde{y} = \tilde{F}(\tilde{x})$. Then, consider all invalid input codewords $\tilde{x} \notin \mathcal{C}$ and map them to a fixed invalid output codeword $\alpha \notin \mathcal{C}$. Thus, for every $\tilde{x} \notin \mathcal{C}$, we add $(\tilde{x}, \alpha)$ to the previous list of correct input and outputs. The resulting list goes over all possible encoded inputs. Thus, via interpolation, there exists a function $\tilde{F}$ which maps every valid codeword $\tilde{x} \in \mathcal{C}$ to a valid codeword of $\tilde{F}(\tilde{x})$. In addition, $\tilde{F}$ maps every "faulty" input to a fixed faulty output α. As a result, $\tilde{F}$ is correct and stable.

There are multiple ways to generate $\tilde{F}$, namely by mapping incorrect inputs to different incorrect outputs. The encoded XOR and AND gates from Section 4 are examples of this approach.

Moreover, given a correct and uniform sharing $\bar{F}$ of F, we can make a stable encoding of F which remains uniform by mapping the correct shared inputs to the shared outputs of the uniform function and by mapping the incorrect input sharings to an incorrect output sharing. However, this method is not guaranteed to keep non-completeness. The interpolation technique might create a high-degree function that is not non-complete. Nevertheless, in Section 6, we provide a methodology that does ensure the non-completeness when working with $t + 1$ shares for a degree t function.

6 Stable $t + 1$ Threshold Implementations

In this section, we present a methodology to provide stable encodings of the traditional $t + 1$ shared threshold implementations of degree t functions. As opposed to the methodologies from Section 5, we are now also concerned with ensuring the threshold implementation properties such as uniformity and non-completeness.

Linear Functions We consider the case of an invertible linear function L. Consider a systematic linear code $\mathcal{C}$, with $G = (I|P)$ its generator matrix and $H = (-P^T|I)$ its parity check matrix. Using these notations, we transform L to its stable encoded version $\tilde{L}$. Let $P(x) = xP = x'$ be the matrix calculating the parity bits of a message. Then,

$$\tilde{L} = \begin{pmatrix} L & 0 \\ P + (P \circ L) & I \end{pmatrix}.$$

First, the map $\tilde{L}$ is correct since if a message x is encoded as $\langle x, P(x)\rangle$ then $\tilde{L}(\langle x, xP\rangle) = \langle L(x), P(x) + (P \circ L)(x) + x'\rangle$ which is equivalent to the correct encoding of the output $\langle L(x), (P \circ L)(x)\rangle$ if there is no fault injected. Second, the map $\tilde{L}$ is stable since it is a permutation as L is invertible. If a fault is injected to

a gate, then due to the non-completeness, $\tilde{L}$ outputs an invalid encoding which will be propagated by the next stable function.

The above map is trivially extended to a shared and encoded map by applying it share by share. As a result, the encoding of linear layers is easy which allows us to use affine equivalences to classify nonlinear functions where it suffices to provide non-complete, uniform, and stable sharings of each class.

Invertible Nonlinear Functions We extend the above method for linear functions to nonlinear ones. Given a permutation F and its uniform and non-complete sharing $\bar{F}$. Because F is a permutation and $\bar{F}$ is uniform, $\bar{F}$ is a permutation itself. We encode $\bar{F}$ as

$$\tilde{F} = \begin{pmatrix} \bar{F} & 0 \\ \bar{P} + (\bar{P} \circ \bar{F}) & I \end{pmatrix},$$

where $\bar{P}$ is the parity check function which works share by share.

The above function is still correct since $\tilde{F}(\langle \bar{x}, \bar{P}(\bar{x}) \rangle) = \langle \bar{F}(\bar{x}), \bar{P} \circ \bar{F}(\bar{x}) \rangle$. Similarly, the function is stable since it is invertible, namely

$$\tilde{F}^{-1} = \begin{pmatrix} \bar{F}^{-1} & 0 \\ (\bar{P} \circ \bar{F} + \bar{P}) \circ \bar{F}^{-1} & I \end{pmatrix}.$$

Moreover, since $\tilde{F}$ maps correct encoded inputs to correct encodings of $\bar{F}$'s outputs, the map is still uniform. Finally, $\tilde{F}$ is non-complete since $\bar{F}$ is non-complete and, since $\bar{P}$ works share-wise, $\bar{P} + (\bar{P} \circ \bar{F}) + I$ is also non-complete.

The above method allows us to find stable encodings of threshold implementations with $t+1$ shares of a function of degree t for any systematic linear code. We provide non-complete, uniform, and stable sharings for $\mathcal{Q}_{12}^4$ encoded with a duplication code and commonly used extended Hamming code in Appendix A. However, when using fewer shares, a sharing which is both non-complete and uniform can not be found and thus more specific encodings and sharings are required. We additionally note that due to the S-box usually being a permutation, simple duplication does not necessarily require the above method to be implemented, unlike the case with other codes.

7 Stable Two-Share Threshold Implementations

Unlike the $t+1$ shared encodings of threshold implementations, we can not achieve non-completeness and uniformity at the same time with an equal number of outputs and inputs while operating on two shares. Therefore, we start with the work of Shahmirzadi and Moradi [SM21a] proposing uniform sharings of the four-bit quadratic classes which are first-order probing secure, but operate over two cycles.

We first elaborate on the security requirements of an encoded function over multiple cycles. Similar to the work of Shahmirzadi and Moradi, a masked function is required to be first-order probing secure independent of the number of

cycles. Additionally, a masked function is required to be secure against a first-order register/gate faulting adversary. To guarantee the composability of the design against probing adversaries, we require that the masked function (over two cycles) is uniform. Similarly, for the composability against fault attacks, we require that the encoded function is stable considering both its intermediate and final outputs.

To encode masked functions with stability, we start by providing an example of a stable masked AND gate. First, we recall the two shared AND gate from Shahmirzadi and Moradi that is performed in two cycles where a_0, a_1, b_0, b_1 are the input shares, and z_0, z_1 are the output shares:

$$\begin{array}{ll} \text{Stage 1} & \text{Stage 2} \\ x_0 = a_0b_0 + b_0 & \\ x_1 = a_0b_1 & z_0 = x_0 + x_1 \\ \hline x_2 = a_1b_1 & z_1 = x_2 + x_3 \\ x_3 = a_1b_0 + b_0 & \end{array} \tag{1}$$

We encode this function with a duplication code by replacing the AND and XOR gates in Eq. (1) with the stable encodings of them given in Section 4. Then, the stable two shared AND gate is given as follows.

$$\begin{array}{ll} \text{Stage 1} & \text{Stage 2} \\ x_0 = a'_0b'_0 + b_0 + (a_0 + a'_0)(b_0 + b'_0 + 1) & \\ x_1 = a'_0b_1 + (a_0 + a'_0) + (b_1 + b'_1)(a_0 + 1) & z_0 = x_0 + x_1 + (x_0 + x'_0)(x_1 + x'_1) \\ \hline x_2 = a'_1b_1 + (a_1 + a'_1) + (b_1 + b'_1)(a_1 + 1) & z_1 = x_2 + x_3 + (x_2 + x'_2)(x_3 + x'_3) \\ x_3 = a'_1b'_0 + b_0 + (a_1 + a'_1)(b_0 + b'_0 + 1) & \end{array}$$

$$\begin{array}{ll} x'_0 = a'_0b'_0 + b'_0 & \\ x'_1 = a'_0b'_1 & z'_0 = x'_0 + x'_1 \\ \hline x'_2 = a'_1b'_1 & z'_1 = x'_2 + x'_3 \\ x'_3 = a'_1b'_0 + b'_0 & \end{array}$$

We encoded the stable two shared AND gate using the stable encodings of the regular XOR and AND gates. As a result, for any injected fault vector e that is not a valid codeword, the above function will propagate the fault to the output which is not a valid codeword[1].

We provide non-complete, uniform, and stable two shared masked encodings for $\mathcal{Q}_{12}^4$ using a duplication code and the $[8, 4, 4]$ extended Hamming code in Appendix B.

8 State-wide Encodings

Until this section, the focus point of this work was to encode small functions such as S-boxes. In this section, we investigate encoding the entire state of the

[1] The stability of the encoded two shared AND gate is verified for all input and fault combinations.

primitive to reduce the overall state size compared to encoding all state elements separately. We note the experimental work by Bartkewitz *et al.* [BBM+22] shows that there is a direct relation between the number of parity bits and the practical security of a design against a laser fault injection. We emphasize that in this work, we focus on protection against the adversary introduced in Section 2.1 and find techniques to improve the efficiency of the encodings without losing this protection.

Consider the linear layer (or a part thereof) of a symmetric primitive represented as a matrix M. For example, the AES MixColumns matrix is

$$M = \begin{pmatrix} 2\,3\,1\,1 \\ 1\,2\,3\,1 \\ 1\,1\,2\,3 \\ 3\,1\,1\,2 \end{pmatrix}.$$

We encode the state using a systematic linear code $\mathcal{C}$ with generator matrix $G = (I|P)$. In case $(xM^T)(I|P) = x(M^T|P)$ or when $M^T P = P$, then the encoded state still consists of valid parity bits after the linear operation. That means using a code where the P matrix consists of eigenvectors of the transposed diffusion layer M^T with eigenvalue one. Equivalently, P consists of vectors in the nullspace of $(M^T - I)$. If such a code is found, we can encode the state where the cost of the linear layer comes for free (over the regular shared cost).

As an example, we consider symmetric primitives with a four-by-four state which shift the rows and applies a linear layer M per column. This structure is used in a lot of ciphers such as AES [AES01], SKINNY [BJK+16], Midori [BBI+15], PRINCE [BCG+12], and LED [GPPR11]. We start by considering a four-by-four state over some vector space $\mathbb{F}_2^n$:

$$\begin{pmatrix} x[0,0] & \dots & x[0,3] \\ \vdots & \ddots & \vdots \\ x[3,0] & \dots & x[3,3] \end{pmatrix}.$$

Let the state be encoded by applying a linear code $\mathcal{C}$ over $\mathbb{F}_2^n$ (such as duplication or a parity code) with generator matrix G and consider the encoded state

$$\begin{pmatrix} x[0,0] & \dots & x[0,3] & p[0] \\ \vdots & \ddots & \vdots & \vdots \\ x[3,0] & \dots & x[3,3] & p[3] \end{pmatrix},$$

where $p[i] = \sum_{j=0}^{3} x[i,j]G$. That is, the state is encoded using the sum of the encodings of the elements in a single row. Using this encoding, an encoded ShiftRows operation has no additional overhead as the parity bits do not need to change. The MixColumns operation also has a reduced cost as it can be directly applied to the column $p[i]$. Moreover, the number of parity bits in the state is reduced by 75% over encoding each cell in the state by $\mathcal{C}$.

It is possible to reduce the above encoding further by finding codes which commute with the entire linear layer of primitives. Specifically the MixColumns operation from PRINCE, SKINNY, MIDORI, the Keccak permutation[2], and AES has the property that

$$\sum_{(i,j)=(0,0)}^{(3,3)} x[i,j] = \sum_{(i,j)=(0,0)}^{(3,3)} y[i,j]\,,$$

with $y = xM^T$, the state after the application of the MixColumns diffusion layer. More specifically, their MixColumns matrix has the eigenvector $(1,1,1,1)$. As a result, for a linear code $\mathcal{C}$ with generator matrix G,

$$G \sum_{(i,j)=(0,0)}^{(3,3)} x[i,j] = G \sum_{(i,j)=(0,0)}^{(3,3)} y[i,j]\,.$$

This allows us to encode the entire state using only one extra cell of parity bits which ensures the encoded ShiftRows and MixColumns operations come at no additional cost over their shared version.

Unlike the linear layer, getting a linear encoding through a nonlinear bricklayer function is trickier and needs a different approach. We consider the case when working with $t+1$ shares for a degree t function F. Given a non-complete and uniform sharing $\bar{F}$, the following function is a non-complete, uniform, and stable masking of a row of the state (where the state is encoded row-by-row via a parity check code)

$$\tilde{F} = \begin{pmatrix} \bar{F} & 0 & 0 & 0 & 0 \\ 0 & \bar{F} & 0 & 0 & 0 \\ 0 & 0 & \bar{F} & 0 & 0 \\ 0 & 0 & 0 & \bar{F} & 0 \\ \bar{P} + (\bar{P} \circ \bar{F}) & \bar{P} + (\bar{P} \circ \bar{F}) & \bar{P} + (\bar{P} \circ \bar{F}) & \bar{P} + (\bar{P} \circ \bar{F}) & I \end{pmatrix},$$

with $\bar{P}(\cdot)$ the parity check function of $\mathcal{C}$, the code over one element of the state. The above structure is secure, but it does not significantly reduce the cost of the encoded S-box. Instead, the encoding reduces the state size and eases the cost of the symmetric primitive's linear layer. For the case of nonlinear functions over two shares, we did not find a general construction and thus leave the computation of the nonlinear layers over two shares as interesting future work.

9 Practical Application to Keccak

In this section, we discuss the stability of the proposed encodings (Appendix A-D), and their performance.

[2] For the Keccak permutation, we abuse the name and consider θ as a "MixColumns" operation.

9.1 Architecture

Keccak [BDPA13] is a cryptographic primitive based on the sponge construction that allows to absorb and process an input of arbitrary length producing an arbitrary length output. The core of the algorithm is a permutation function usually referred as Keccak-$f[b]$, where b is a variable state size. The permutation consists of a number of rounds including four linear layers and one nonlinear layer (known as χ). We focus on two versions of Keccak-f with state lengths of 1600 and 200 bits for the area benchmarks and floorplanning, respectively. The larger design consists of 24 rounds, while the lightweight version requires 18 rounds to complete.

To implement Keccak-f, we applied the duplication code to the entire state. Since all the functions except for χ are linear permutations, a simple duplication of those layers satisfies the stability notion regardless of the number of shares used for the threshold implementations, approximately doubling the area. Combining the linear layers with the encodings from Appendices C and D, we obtained stable threshold implementations of Keccak-$f[1600]$ and Keccak-$f[200]$ with four and two shares. Additionally, for the following experiments, we implemented a duplicated two-shared version without stability resulting in a single change in the χ compression step. The constructions with two shares require three register stages to ensure non-completeness, whereas it is sufficient to have only one for the four-shared design. As a result, we have triple the latency overhead for the two-shared design. The architectures of all the designs are described in Verilog.

Finally, we note that the designs were implemented using the `KEEP_HIERACHY` option to avoid further circuit optimizations performed by the synthesizer to achieve fault non-completeness.

9.2 Formally Verifying Fault Security

We perform a tool-assisted verification to ensure our encodings are stable (Definition 9). Considering the other tools such as Danira [HPB22], we decide to make use of `VerMFI`, an open-source tool [3] by Arribas *et al.* [ANR18,AWMN20] for the formal verification against the register/gate-faulting adversary from Section 2.1. The tool receives an HDL design or the netlist of a design. If an HDL design is given, then the tool obtains its netlist. Then, the tool either verifies the TI properties (*i.e.* non-completeness, uniformity), or performs a fault evaluation of the design using a user specified fault configuration file. After the evaluation, the tool reports all non-detected faults per input test vector, as well as the ineffective faults. Note that the tool has a straightforward definition of ineffective faults, *i.e.,* any fault that causes no change in the ciphertext, whereas we additionally require a dependency on a secret value.

To verify the stability (*i.e.,* security against register faults), we go over all incorrect codewords for the input and verify that the faults are propagated to the

[3] https://github.com/vmarribas/VerMFi

output which then can be detected. Using `VerMFI`, we go over all gate faults in the design and verify that each fault's output state (ineffective or not) does not depend on the input secrets. To confirm this, we simply compare the number of ineffective faults for each input test vector (*i.e.* all possible sharings of an input). If we obtain the same number of ineffective faults for each unshared input, then we verify the independence of the faults on the secret values.

We verified the security of the HDL designs of the encodings presented in Appendix A B, C, and D which includes the nonlinear layer[4] of the protected Keccak implementation described in `Verilog` using `VerMFI`. All designs were verified to be secure against a single register/gate fault.

Verification via Attack Simulation. In addition to the formal verification, we performed an attack simulation on both the stable and fault non-complete Keccak S-box, and a simple (unstable) duplication of the two-share threshold implementation by Shahmirzadi and Moradi [SM21a] to demonstrate the security advantages of the stability notion. Specifically, we performed a SIFA-2 simulation on both implementations. Using `VerMFI`, we went through all the gates/registers in the obtained netlist covering all possible inputs and injected bit flip faults with effectiveness probability equal to one. Only one gate was found to be susceptible to SIFA-2 in the unstable two-shared Keccak implementation in which the propagation of the fault to the S-box output is dependent on one of the unmasked input bits. Meaning, given that a bit flip fault is injected to the "insecure" gate, one can perform a successful SIFA-2 attack that reveals the value of one of the unmasked input bits with a success probability one. In other words, if the "insecure" gate out of 169 gates is hit by the fault injection, then one can perform a successful SIFA-2 attack. Assuming the fault location is randomly chosen, the success probability of a SIFA-2 attack is 1/169, independent of the inputs. Conversely, as mentioned earlier in this section, we did not detect any single susceptible gate in the netlist of the stable fault non-complete two-shared Keccak implementation. Interestingly enough, we found that enforcing the fault non-completeness notion on the simple duplication (enforcing each combinatorial gate drives a non-complete set of output shares) was enough to prevent the SIFA attack. We note that this is due to the permutation structure of the Keccak S-box and its masking. More precisely, since the composition of the two cycles of the duplicated masked S-box is a permutation, it is also stable. The transition from the first to the second stage of the masked S-box is unstable, however, since this is a simple compression layer (consisting only of XORs), it remains secure against a single gate-fault. As a result, similar to the observations by Shahmirzadi and Moradi [SM21a] that the notion of uniformity does not need to hold each cycle, but that it suffices to re-gain uniformity after a number of cycles, we believe it is possible to relax stability to hold only every couple of cycles and still ensure the security against a single fault adversary.

[4] We note that only the nonlinear layer of the Keccak was verified since its linear layer (or larger parts of the state) would require too much computational complexity from `VerMFI`.

9.3 Floorplan Results

We deploy the stable two-shared threshold implementation of Keccak based on the encoding from Appendix D on FPGA and report on its floorplan. Since the complexity of compiling the floorplan is expensive, we only provide the floorplan of the encoded and masked Keccak-f[200]. We then compare this floorplan with that of a simple two-shared threshold implementation of Keccak, where the masking is simply duplicated.

We used a Xilinx Sakura-X board equipped with a Kintex-7 XC7K160T-1FBGC FPGA. The floorplan was generated using the Xilinx PlanAhead tool with no placement constraints applied during the "Placement and Routing" step. We note that the implementations were synthesized using only the `KEEP_HIERACHY` option. The results are shown in Figure 2.

The floorplan results demonstrate that the simply duplicated two-shared design neatly separates the state from the duplicated state, whereas the stable two-shared design mixes both components, because they are interconnected. Following the work from Bartkewitz *et al.* [BBM+22], this could mean that a single laser fault can potentially break the stable design when a multi-bit fault is injected, whereas it would be more difficult to do so with the simple duplicated design.

To demonstrate that stability does not necessarily affect the placement of the duplicated state, we show that the four-shared stable Keccak design (Figure 2, bottom) has a floorplan similar to the unstable duplicated design (Figure 2, top-right). Namely, the four-shared design would be secure against SIFA and provide more protection against laser faults. A further study into SIFA resistant designs which properly split the placement of its code-coordinates is left as interesting future work.

9.4 Benchmarks

In this section, we elaborate on the costs of the protected Keccak which is detailed in Tables 1, 2, 3. With respect to the overhead in time, our countermeasure does not increase the latency of the unencoded TI. With respect to the overhead in area, our countermeasure brings roughly a factor of two to three overhead compared to the unencoded TI depending on the used linear code, plus a single detection circuit at the end.The costs of the unencoded TI of the Keccak S-box (*i.e.,* where no linear codes are applied), and the encoded TIs of the Keccak S-box using StaTI are shown in Table 1. Section 6 refers to the encoded TI of the Keccak S-box with four shares, and Section 7 refers to the encoded TI of the Keccak S-box with two shares. We note that the cost calculation does not take an error detection circuit into account.

We compare our countermeasure with the one by Daemen *et al.* [DDE+20] applied on hardware implementations of the Keccak S-box. In the work by Daemen *et al.* it is reported that five masked Toffoli gates ($(a, b, c) \mapsto (a, b, c + ab) \in \mathbb{F}_2^3$) and two XORs are required to implement the two share Keccak S-box which consists of 20 AND/AND-NOTs and 22 XORs (Section 3.5 in [DDE+20]). We

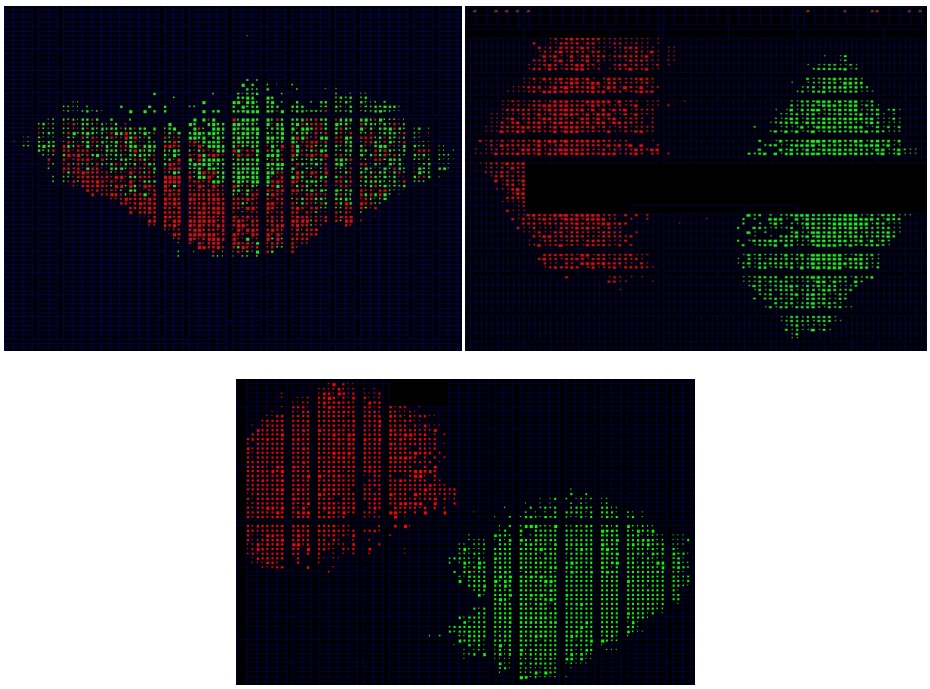

Fig. 2. The floorplan results of the two-shared stable (top-left), the two-shared unstable (top-right) and the four-shared stable (at the bottom) Keccak-$f[200]$. In red we report the used logic elements working on the regular state and in green we report the elements working on the duplicated state.

adopt this implementation to hardware considering the authors reported that one shared Toffoli gate requires two cycles in order not to be affected by glitches (Section 6.4 in [DDE+20]). Therefore, without any parallelism, this pipelined implementation takes 10 cycles with the area cost of one shared Toffoli gate and one XOR. However, with careful parallelism, we predict the implementation to take four cycles in the best case scenario. These Toffoli gates are then duplicated to implement the redundancy. Then, the implementation takes 10 cycles and is expected to require 10 XORs and 8 AND/AND-NOTs.

We then consider the implementation of the Keccak S-box protected using our countermeasure for both the duplication and the $[6, 5, 2]$-code from Appendix C for four shares, and Appendix D for two shares. The four share implementation of the Keccak S-box takes one cycle, and requires 78 XORs and 30 ANDs. With simple duplication these numbers increase to 156 XORs and 60 ANDs which satisfies the stability. The four share implementation using the $[6, 5, 2]$-code takes again one cycle, and in total requires 145 XORs and 44 ANDs. This implementation also refers to the technique proposed in Section 8 as all five state elements are encoded in one element in the encoded state. The two share implementation

Table 1. Benchmarks for different protected Keccak S-box implementations using Nangate 45nm Open Cell Library

Method	Cycles	# Shares	Area (GE)	# XORs	# ANDs
Daemen *et al.*	10	2	96	10	8
Unencoded TI	1	4	194.7	78	30
Section 6 + duplication	1	4	389.3	156	60
Section 6 + parity code	1	4	338	145	44
Unencoded TI	2	2	212	36	20
Section 7 + duplication	2	2	497.3	102	50
Section 7 + parity code	2	2	470	96	30

of the Keccak S-box takes two cycles, and requires 20 ANDs and 36 XORs. Using the duplication code, these numbers increase to 50 ANDs and 102 XORs in total. Likewise, the two share implementation using the $[6, 5, 2]$-code takes two cycles, and requires 30 ANDs and 96 XORs in total. This implementation again refers to the technique proposed in Section 8 due to the encoding. We compare all these results in Table 1. Our countermeasure comes at a lower latency cost (tenfold smaller for four shares, fivefold smaller for two shares), whereas there is an increase in the area cost (fourfold larger for four shares, sevenfold larger for two shares) when compared to the countermeasure from [DDE+20]. Moreover, there is a trade-off when different TIs of Keccak S-box are encoded. Two share implementations come with higher latency and lower combinational area cost when compared to four share implementations. However, as the two share TI implementations take two cycles, in addition to the combinational area, they bring additional non-combinational area which makes the total area larger than the four share TI implementations.

Additionally, we consider the implementation of the state-wide encodings (from Section 8) and compare it with a simple duplication in terms of the overhead in state size and area in Table 2. We limit our implementation to the χ function of the Keccak-$f[1600]$ state. When compared to the countermeasure from Daemen *et al.*, state-wide encoding comes at a fourfold larger area cost and tenfold smaller latency cost. As already discussed in Section 8, state-wide encodings do not provide great improvement with respect to the area costs for the nonlinear layer. However, the state-wide encodings allow a reduction of the encoded state size and a reduction in the cost of the linear layer.

We also consider the complete Keccak-$f[1600]$ implementation from Section 9.1 to elaborate on the advantages of StaTI over simple duplication. To this end, we compare the performance numbers (latency and area) for simple duplicated threshold implementations and threshold implementations protected using StaTI for four and two shares in Table 3. While simple duplication offers protection against attacks exploiting wrong outputs (*e.g.*, DFA), the incorporation of StaTI further offers protection against attacks exploiting the data dependency of fault activation/propagation. Simple duplicated threshold implementations

Table 2. Benchmarks for different encodings of the χ function applied to the Keccak-$f[1600]$ state using Nangate 45nm Open Cell Library

Method	Cycles	# Shares	State Size (bits)	Area (kGE)
Daemen *et al.*	10	2	3200	30.7
Unencoded TI	1	4	6400	62.3
Section 6 + duplication	1	4	12800	124.6
Section 8	1	4	6500	110

can also be perceived as instances of TaE [SJR+19]. As previously discussed, a duplicated threshold permutation (also as a TaE instance) is inherently stable, and secure against single gate/register-faulting adversary. In case of a two-share threshold implementation, the compression layer of the S-box necessitates the utilization of stable XOR gates. Nevertheless, Table 3 shows that in the stable threshold implementation of the two-share Keccak, there is only a 7.8% increase in the area when compared to simple duplication. That is, StaTI offers protection against ineffective faults (*i.e.,* data dependency of the fault) in addition to effective faults with limited additional costs. For the two-shared threshold implementation, we can consider an implementation with three repetitions on bit level as an instance of TaE that would provide security against SIFA-2. Excluding error detection/correction circuits, StaTI would offer more efficient security against gate/register-faulting adversary.

Table 3. Benchmarks for complete masked Keccak-$f[1600]$ using the Nangate 45nm Open Cell Library.

Fault Countermeasure	# Shares	Stable	Cycles	Area (kGE)
Unprotected [BDN+13]	4	✗	24	139.4
Unprotected [SM21b]	2	✗	72	125.4
Simple duplication	4	✓	24	246.9
Simple duplication	2	✗	72	249.5
Section 7 + duplication	2	✓	72	271.9

10 Conclusion

In this paper, we extended threshold implementations with the *stability* notion that addresses the serial composable security against formal gate/register-faulting adversaries. We proved that an encoding of a threshold implementation is secure against a gate/register fault if it is serially composed of functions such that for every wrong codeword addition to the input of a function, the function outputs a wrong codeword. That is, an encoding of a threshold implementation

is secure if it propagates any fault to the output. Based on *stability*, we provided stable encodings for XOR and AND gates that would allow us to build secure encodings for more complex functions. Consecutively, we presented general methodologies to transform any unstable encoding to a stable one. We provided efficient specific methodologies for $t+1$ and two-share threshold implementations of S-boxes. Next, we provided an efficient encoding technique by encoding state elements together depending on the state structure. Following these methodologies, we provided encodings for the quadratic class $\mathcal{Q}_{12}^4$ and the Keccak S-box. We then provided a StaTI protected Keccak-f[1600] implementation on FPGA and reported on their security and performance evaluation. Comparing the Keccak implementation with the one from [DDE+20], we improved their encodings in terms of latency with the trade-off in area cost. Concerning formal security, our designs were verified to be secure against a single gate/register fault by the `VerMFI` tool.

To defeat fault attacks exploiting the data dependency of the fault propagation, we designed StaTI such that an injected fault is always propagated to the output regardless of the secret data. Following a different approach, most of the state-of-the-art countermeasures employ masking and redundancy, albeit without ensuring fault propagation, which subsequently necessitates intermediate error checks/corrections. In contrast, we show that by ensuring the fault propagation, we can omit the need for intermediate checks. Furthermore, compared to threshold implementations providing provable security against SCA encoded with duplication to protect against wrong output attacks, StaTI offers protection against ineffective faults with a low overhead in area cost.

While claiming no security against combined attacks, we note that the stability notion can prove advantageous in defeating FTA-SCA [SBJ+21] when combined with the TI notions (*i.e.,* StaTI) for some cases. For example, the stable two-shared masked encoding for the AND gate (Section 7) exhibits security against FTA-SCA. This is attributed to the stable encodings of the XOR and AND gates that are utilized in both cycles, which ensures that any injected fault is propagated to the masked output. On the other hand, the stable two-shared masked encoding with duplication for $\mathcal{Q}_{12}^4$ (Appendix B) is not secure against FTA-SCA. This is because the first cycle of the two-shared TI of $\mathcal{Q}_{12}^4$ does not require the stable encodings of XOR and AND gates to be utilized, as it already exhibits a stable behavior, *i.e.,* any injected fault is propagated to at least one of the output bits. Therefore, when the shares are recombined in the second cycle, the fault is still ineffective in some output bits which makes it susceptible to FTA-SCA. However, we believe that replacing the XOR and AND gates with their stable encodings would yield implementations secure against FTA-SCA.

In this work, we have focused on first-order fault attacks where the adversary injects a fault in a single gate or register. Regarding higher-order fault attacks, where multiple gates are attacked, StaTI's share-wise error detection would be effective as long as the the faults are injected in a single cycle and do not generate valid codewords (*i.e.* by using codes with a larger minimal distance). However, to secure the case where faults are injected in different cycles, we think that

intermediate error checks (*e.g.*, after each S-box) would be essential to detect these faults before they become undetectable. Nevertheless, we consider higher-order fault attacks as future work. Additionally, we have explored the theoretical security of StaTI and verified this on implementations using tools. However, we leave the practical verification of our countermeasure against various fault attacks such as laser faults or glitch injections as future work.

Acknowledgements. We thank Svetla Nikova for the helpful discussions. This work was supported by CyberSecurity Research Flanders with reference number VR20192203.

References

AES01. Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, November 2001.

AK97. Ross J. Anderson and Markus G. Kuhn. Low cost attacks on tamper resistant devices. In Bruce Christianson, Bruno Crispo, T. Mark A. Lomas, and Michael Roe, editors, *Security Protocols, 5th International Workshop, Paris, France, April 7-9, 1997, Proceedings*, volume 1361 of *Lecture Notes in Computer Science*, pages 125–136. Springer, 1997.

AMR+20. Anita Aghaie, Amir Moradi, Shahram Rasoolzadeh, Aein Rezaei Shahmirzadi, Falk Schellenberg, and Tobias Schneider. Impeccable circuits. *IEEE Trans. Computers*, 69(3):361–376, 2020.

ANR18. Victor Arribas, Svetla Nikova, and Vincent Rijmen. Vermi: Verification tool for masked implementations. In *25th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2018, Bordeaux, France, December 9-12, 2018*, pages 381–384. IEEE, 2018.

AWMN20. Victor Arribas, Felix Wegener, Amir Moradi, and Svetla Nikova. Cryptographic fault diagnosis using verfi. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020*, pages 229–240. IEEE, 2020.

BBI+15. Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A block cipher for low energy. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, volume 9453 of *Lecture Notes in Computer Science*, pages 411–436. Springer, 2015.

BBM+22. Timo Bartkewitz, Sven Bettendorf, Thorben Moos, Amir Moradi, and Falk Schellenberg. Beware of insufficient redundancy an experimental evaluation of code-based FI countermeasures. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(3):438–462, 2022.

BCC+14. Julien Bringer, Claude Carlet, Hervé Chabanne, Sylvain Guilley, and Houssem Maghrebi. Orthogonal direct sum masking - A smartcard friendly computation paradigm in a code, with builtin protection against side-channel and fault attacks. In David Naccache and Damien Sauveron, editors, *Information Security Theory and Practice. Securing the Internet of*

Things - 8th IFIP WG 11.2 International Workshop, WISTP 2014, Heraklion, Crete, Greece, June 30 - July 2, 2014. Proceedings, volume 8501 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2014.

BCG+12. Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - A low-latency block cipher for pervasive computing applications (full version). *IACR Cryptol. ePrint Arch.*, page 529, 2012.

BDF+09. Shivam Bhasin, Jean-Luc Danger, Florent Flament, Tarik Graba, Sylvain Guilley, Yves Mathieu, Maxime Nassar, Laurent Sauvage, and Nidhal Selmane. Combined SCA and DFA countermeasures integrable in a FPGA design flow. In Viktor K. Prasanna, Lionel Torres, and René Cumplido, editors, *ReConFig'09: 2009 International Conference on Reconfigurable Computing and FPGAs, Cancun, Quintana Roo, Mexico, 9-11 December 2009, Proceedings*, pages 213–218. IEEE Computer Society, 2009.

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, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.

BDN+13. Begül Bilgin, Joan Daemen, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Gilles Van Assche. Efficient and first-order DPA resistant implementations of keccak. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, volume 8419 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 2013.

BDPA13. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 313–314. Springer, 2013.

BGN+14. Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Higher-order threshold implementations. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 326–343, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Heidelberg, Germany.

BH17. Jakub Breier and Xiaolu Hou. Feeding two cats with one bowl: On designing a fault and side-channel resistant software encoding scheme. In Helena Handschuh, editor, *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings*, volume 10159 of *Lecture Notes in Computer Science*, pages 77–94. Springer, 2017.

BJK+16. Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology -*

CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II, volume 9815 of *Lecture Notes in Computer Science*, pages 123–153. Springer, 2016.

BKHL20. Jakub Breier, Mustafa Khairallah, Xiaolu Hou, and Yang Liu. A countermeasure against statistical ineffective fault analysis. *IEEE Trans. Circuits Syst.*, 67-II(12):3322–3326, 2020.

BS97. Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997.

CJRR99. Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

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 *Lecture Notes in Computer Science*, pages 137–153, Taipei, Taiwan, September 25–28, 2017. Springer, Heidelberg, Germany.

DDE+20. Joan Daemen, Christoph Dobraunig, Maria Eichlseder, Hannes Gross, Florian Mendel, and Robert Primas. Protecting against statistical ineffective fault attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):508–543, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8599.

DDRT12. Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic transient faults injection on a hardware and a software implementations of AES. In Guido Bertoni and Benedikt Gierlichs, editors, *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012*, pages 7–15. IEEE Computer Society, 2012.

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 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*, volume 11273 of *Lecture Notes in Computer Science*, 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 Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):547–572, 2018.

DNR19. Siemen Dhooghe, Svetla Nikova, and Vincent Rijmen. Threshold implementations in the robust probing model. In Begül Bilgin, Svetla Petkova-Nikova, and Vincent Rijmen, editors, *Proceedings of ACM Workshop on Theory of Implementation Security, TIS@CCS 2019, London, UK, November 11, 2019*, pages 30–37. ACM, 2019.

FGM+23. Jakob Feldtkeller, Tim Güneysu, Thorben Moos, Jan Richter-Brockmann, Sayandeep Saha, Pascal Sasdrich, and François-Xavier Standaert. Combined private circuits - combined security refurbished. Cryptology ePrint Archive, Paper 2023/1341, 2023. https://eprint.iacr.org/2023/1341.

FGP+18. Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):89–120, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7270.

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, *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013*, pages 108–118. IEEE Computer Society, 2013.

GMK16. Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In Begül Bilgin, Svetla Nikova, and Vincent Rijmen, editors, *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*, page 3. ACM, 2016.

GMO01. Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2001.

GPK+21. Michael Gruber, Matthias Probst, Patrick Karl, Thomas Schamberger, Lars Tebelmann, Michael Tempelmeier, and Georg Sigl. Domrep-an orthogonal countermeasure for arbitrary order side-channel and fault attack protection. *IEEE Trans. Inf. Forensics Secur.*, 16:4321–4335, 2021.

GPPR11. Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The LED block cipher. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2011.

Hab65. D. H. Habing. The use of lasers to simulate radiation-induced transients in semiconductor devices and circuits. *IEEE Transactions on Nuclear Science*, 12(5):91–100, 1965.

HPB22. Vedad Hadzic, Robert Primas, and Roderick Bloem. Proving SIFA protection of masked redundant circuits. *Innov. Syst. Softw. Eng.*, 18(3):471–481, 2022.

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, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, 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, 23rd Annual International Cryptology Conference,*

Santa Barbara, California, USA, August 17-21, 2003, Proceedings, volume 2729 of *Lecture Notes in Computer Science*, 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, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

Koc96. Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

MAN+19. Lauren De Meyer, Victor Arribas, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. M&m: Masks and macs against physical attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(1):25–50, 2019.

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, *ICICS 06: 8th International Conference on Information and Communication Security*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545, Raleigh, NC, USA, December 4–7, 2006. Springer, Heidelberg, Germany.

PRC+19. Sikhar Patranabis, Debapriya Basu Roy, Anirban Chakraborty, Naveen Nagar, Astikey Singh, Debdeep Mukhopadhyay, and Santosh Ghosh. Lightweight design-for-security strategies for combined countermeasures against side channel and fault analysis in iot applications. *J. Hardw. Syst. Secur.*, 3(2):103–131, 2019.

RAD20. Keyvan Ramezanpour, Paul Ampadu, and William Diehl. Rs-mask: Random space masking as an integrated countermeasure against power and fault analysis. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, December 7-11, 2020*, pages 176–187. IEEE, 2020.

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 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 764–783. Springer, 2015.

RMB+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 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 121–151. Springer, 2018.

RSM21. Shahram Rasoolzadeh, Aein Rezaei Shahmirzadi, and Amir Moradi. Impeccable circuits III. In *IEEE International Test Conference, ITC 2021, Anaheim, CA, USA, October 10-15, 2021*, pages 163–169. IEEE, 2021.

SBJ+21. Sayandeep Saha, Arnab Bag, Dirmanto Jap, Debdeep Mukhopadhyay, and Shivam Bhasin. Divided we stand, united we fall: Security analysis of some SCA+SIFA countermeasures against sca-enhanced fault template attacks.

In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part II*, volume 13091 of *Lecture Notes in Computer Science*, pages 62–94. Springer, 2021.

SBR+20. Sayandeep Saha, Arnab Bag, Debapriya Basu Roy, Sikhar Patranabis, and Debdeep Mukhopadhyay. Fault template attacks on block ciphers exploiting fault propagation. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 612–643. Springer, 2020.

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. *IACR Cryptol. ePrint Arch.*, page 545, 2019.

SM21a. Aein Rezaei Shahmirzadi and Amir Moradi. Re-consolidating first-order masking schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):305–342, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8736.

SM21b. Aein Rezaei Shahmirzadi and Amir Moradi. Second-order SCA security with almost no fresh randomness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):708–755, 2021.

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 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 302–332. Springer, 2016.

SRM20. Aein Rezaei Shahmirzadi, Shahram Rasoolzadeh, and Amir Moradi. Impeccable circuits II. In *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*, pages 1–6. IEEE, 2020.

VZB+22. Navid Vafaei, Sara Zarei, Nasour Bagheri, Maria Eichlseder, Robert Primas, and Hadi Soleimany. Statistical effective fault attacks: The other side of the coin. *IEEE Trans. Inf. Forensics Secur.*, 17:1855–1867, 2022.

A Three Sharing of $\mathcal{Q}_{12}^4$

The quadratic class $\mathcal{Q}_{12}^4 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15, 10, 11]$ is given as follows in its algebraic normal form.

$$
\begin{aligned}
&\mathcal{Q}_{12}^4(a, b, c, d) = x, y, z, w \\
&x = f_0(a, b, c, d) = a \\
&y = f_1(a, b, c, d) = ac + b \\
&z = f_2(a, b, c, d) = ab + ac + c \\
&w = f_3(a, b, c, d) = d
\end{aligned}
$$

Duplication Code For the duplication code, we denote the encoding (x, x') with x the original value and x' its duplicate. Similarly, we denote $(\bar{x}, \bar{x}')$ the shared message bits with $\bar{x}'$ their corresponding parity bits (encoded share-by-share). Since $\mathcal{Q}_{12}^4$ is a quadratic function, we have $t = 2$ and thus use three shares to find a non-complete and uniform sharing. The correct, non-complete, uniform, and stable sharing $\bar{\mathcal{Q}}_{12}^4(\bar{a}, \bar{b}, \bar{c}, \bar{d}, \bar{a}', \bar{b}', \bar{c}', \bar{d}') = (\bar{x}, \bar{y}, \bar{z}, \bar{w}, \bar{x}', \bar{y}', \bar{z}', \bar{w}')$ is given below.

$$
\begin{aligned}
x_0 &= a_0 \\
x_1 &= a_1 \\
x_2 &= a_2 \\
y_0 &= a_0c_0 + a_0c_1 + a_1c_0 + a_0 + a_1 + c_0 + c_1 + b_0 \\
y_1 &= a_1c_1 + a_1c_2 + a_2c_1 + a_1 + a_2 + c_1 + c_2 + b_1 \\
y_2 &= a_2c_2 + a_2c_0 + a_0c_2 + a_0 + a_2 + c_0 + c_2 + b_2 \\
z_0 &= a_0b_0 + a_0b_1 + a_1b_0 + a_0c_0 + a_0c_1 + a_1c_0 + c_0 \\
z_1 &= a_1b_1 + a_1b_2 + a_2b_1 + a_1c_1 + a_1c_2 + a_2c_1 + c_1 \\
z_2 &= a_2b_2 + a_2b_0 + a_0b_2 + a_2c_2 + a_2c_0 + a_0c_2 + c_2 \\
w_0 &= d_0 \\
w_1 &= d_1 \\
w_2 &= d_2
\end{aligned}
$$

$$
\begin{aligned}
x'_0 &= a'_0 \\
x'_1 &= a'_1 \\
x'_2 &= a'_2 \\
y'_0 &= a'_0c'_0 + a'_0c'_1 + a'_1c'_0 + a'_0 + a'_1 + c'_0 + c'_1 + b'_0 \\
y'_1 &= a'_1c'_1 + a'_1c'_2 + a'_2c'_1 + a'_1 + a'_2 + c'_1 + c'_2 + b'_1 \\
y'_2 &= a'_2c'_2 + a'_2c'_0 + a'_0c'_2 + a'_0 + a'_2 + c'_0 + c'_2 + b'_2 \\
z'_0 &= a'_0b'_0 + a'_0b'_1 + a'_1b'_0 + a'_0c'_0 + a'_0c'_1 + a'_1c'_0 + c'_0 \\
z'_1 &= a'_1b'_1 + a'_1b'_2 + a'_2b'_1 + a'_1c'_1 + a'_1c'_2 + a'_2c'_1 + c'_1 \\
z'_2 &= a'_2b'_2 + a'_2b'_0 + a'_0b'_2 + a'_2c'_2 + a'_2c'_0 + a'_0c'_2 + c'_2 \\
w'_0 &= d'_0 \\
w'_1 &= d'_1 \\
w'_2 &= d'_2
\end{aligned}
$$

Extended Hamming Code We consider the systematic $[8, 4, 4]$ extended Hamming code with the following generator matrix

$$
G_{[8,4,4]} = \begin{pmatrix}
1\,0\,0\,0 & 1\,1\,1\,0 \\
0\,1\,0\,0 & 1\,1\,0\,1 \\
0\,0\,1\,0 & 1\,0\,1\,1 \\
0\,0\,0\,1 & 0\,1\,1\,1
\end{pmatrix}.
$$

For the Hamming code, we again denote (x, x') with x the message bits and x' the corresponding parity bits. Similarly, we denote $(\bar{x}, \bar{x}')$ the shared message bits with $\bar{x}'$ their corresponding parity bits (encoded share-by-share). The correct, non-complete, uniform and stable sharing of $\bar{\mathcal{Q}}^4_{12}(\bar{a}, \bar{b}, \bar{c}, \bar{d}, \bar{a}', \bar{b}', \bar{c}', \bar{d}') = (\bar{x}, \bar{y}, \bar{z}, \bar{w}, \bar{x}', \bar{y}', \bar{z}', \bar{w}')$ is given below.

$$
\begin{aligned}
x_0 &= a_0 \\
x_1 &= a_1 \\
x_2 &= a_2 \\
y_0 &= b_0 + a_0c_0 + a_0c_1 + a_1c_0 + a_0 + a_1 + c_0 + c_1 \\
y_1 &= b_1 + a_1c_1 + a_1c_2 + a_2c_1 + a_1 + a_2 + c_1 + c_2 \\
y_2 &= b_2 + a_2c_2 + a_2c_0 + a_0c_2 + a_0 + a_2 + c_0 + c_2 \\
z_0 &= c_0 + a_0b_0 + a_0b_1 + a_1b_0 + a_0c_0 + a_0c_1 + a_1c_0 \\
z_1 &= c_1 + a_1b_1 + a_1b_2 + a_2b_1 + a_1c_1 + a_1c_2 + a_2c_1 \\
z_2 &= c_2 + a_2b_2 + a_2b_0 + a_0b_2 + a_2c_2 + a_2c_0 + a_0c_2 \\
w_0 &= d_0 \\
w_1 &= d_1 \\
w_2 &= d_2
\end{aligned}
$$

$$
\begin{aligned}
x'_0 &= a_0 + a_1 + c_0 + c_1 + a_0b_0 + a_0b_1 + a_1b_0 + a'_0 \\
x'_1 &= a_1 + a_2 + c_1 + c_2 + a_1b_1 + a_1b_2 + a_2b_1 + a'_1 \\
x'_2 &= a_0 + a_2 + c_0 + c_2 + a_2b_2 + a_2b_0 + a_0b_2 + a'_2 \\
y'_0 &= a_0 + a_1 + c_0 + c_1 + a_0c_0 + a_0c_1 + a_1c_0 + b'_0 \\
y'_1 &= a_1 + a_2 + c_1 + c_2 + a_1c_1 + a_1c_2 + a_2c_1 + b'_1 \\
y'_2 &= a_0 + a_2 + c_0 + c_2 + a_2c_2 + a_2c_0 + a_0c_2 + b'_2 \\
z'_0 &= a_0b_0 + a_0b_1 + a_1b_0 + a_0c_0 + a_0c_1 + a_1c_0 + c'_0 \\
z'_1 &= a_1b_1 + a_1b_2 + a_2b_1 + a_1c_1 + a_1c_2 + a_2c_1 + c'_1 \\
z'_2 &= a_2b_2 + a_2b_0 + a_0b_2 + a_2c_2 + a_2c_0 + a_0c_2 + c'_2 \\
w'_0 &= a_0 + a_1 + c_0 + c_1 + a_0b_0 + a_0b_1 + a_1b_0 + d'_0 \\
w'_1 &= a_1 + a_2 + c_1 + c_2 + a_1b_1 + a_1b_2 + a_2b_1 + d'_1 \\
w'_2 &= a_0 + a_2 + c_0 + c_2 + a_2b_2 + a_2b_0 + a_0b_2 + d'_2
\end{aligned}
$$

B Two Sharing of $\mathcal{Q}^4_{12}$

We again consider the $\mathcal{Q}^4_{12}$ quadratic class as defined in Appendix A.

Duplication Code The correct, non-complete, uniform, and stable sharing of the Q^4_{12} function $\bar{\mathcal{Q}}^4_{12}(\bar{a}, \bar{b}, \bar{c}, \bar{d}, \bar{a}', \bar{b}', \bar{c}', \bar{d}') = (\bar{x}, \bar{y}, \bar{z}, \bar{w}, \bar{x}', \bar{y}', \bar{z}', \bar{w}')$ is given over two cycles below.

Stage 1	Stage 2
$k_0 = a_0$	$x_0 = k_0$
$k_1 = a_1$	$x_1 = k_1$
$l_0 = b_0 + a_0 c_0$	
$l_1 = a_0 c_1$	$y_0 = l_0 + l_1 + (l_0 + l'_0)(l_1 + l'_1)$
$l_2 = a_1 c_1$	$y_1 = l_2 + l_3 + (l_2 + l'_2)(l_3 + l'_3)$
$l_3 = b_1 + a_1 c_0$	
$m_0 = c_0 + a_0 b_0 + a_0 c_0$	
$m_1 = a_0 b_1 + a_0 c_1$	$z_0 = m_0 + m_1 + (m_0 + m'_0)(m_1 + m'_1)$
$m_2 = a_1 b_0 + a_1 c_0$	$z_1 = m_2 + m_3 + (m_2 + m'_2)(m_3 + m'_3)$
$m_3 = c_1 + a_1 b_1 + a_1 c_1$	
$n_0 = d_0$	$w_0 = n_0$
$n_1 = d_1$	$w_1 = n_1$
$k'_0 = a'_0$	$x'_0 = k'_0$
$k'_1 = a'_1$	$x'_1 = k'_1$
$l'_0 = b'_0 + a'_0 c'_0$	
$l'_1 = a'_0 c'_1$	$y'_0 = l'_0 + l'_1$
$l'_2 = a'_1 c'_1$	$y'_1 = l'_2 + l'_3$
$l'_3 = b'_1 + a'_1 c'_0$	
$m'_0 = c'_0 + a'_0 b'_0 + a'_0 c'_0$	
$m'_1 = a'_0 b'_1 + a'_0 c'_1$	$z'_0 = m'_0 + m'_1$
$m'_2 = a'_1 b'_0 + a'_1 c'_0$	$z'_1 = m'_2 + m'_3$
$m'_3 = c'_1 + a'_1 b'_1 + a'_1 c'_1$	
$n'_0 = d'_0$	$w'_0 = n'_0$
$n'_1 = d'_1$	$w'_0 = n'_1$

Extended Hamming Code The correct, non-complete, uniform, and stable sharing $\bar{\mathcal{Q}}^4_{12}(\bar{a},\bar{b},\bar{c},\bar{d},\bar{a}',\bar{b}',\bar{c}',\bar{d}') = (\bar{x},\bar{y},\bar{z},\bar{w},\bar{x}',\bar{y}',\bar{z}',\bar{w}')$ is given over two cycles below.

Stage 1

$$
\begin{aligned}
k_0 &= a_0\\
k_1 &= a_1\\
l_0 &= b_0 + a_0c_0\\
l_1 &= a_0c_1\\
l_2 &= a_1c_1\\
l_3 &= b_1 + a_1c_0\\
m_0 &= c_0 + a_0b_0 + a_0c_0\\
m_1 &= a_0b_1 + a_0c_1\\
m_2 &= a_1b_0 + a_1c_0\\
m_3 &= c_1 + a_1b_1 + a_1c_1\\
n_0 &= d_0\\
n_1 &= d_1
\end{aligned}
$$

$$
\begin{aligned}
k'_0 &= k_0 + l_0 + m_0 + a_0 + b_0 + c_0 + a'_0\\
k'_1 &= k_1 + l_1 + m_1 + a_1 + b_1 + c_1 + a'_1\\
l'_0 &= k_0 + l_0 + n_0 + a_0 + b_0 + d_0 + b'_0\\
l'_1 &= k_1 + l_1 + n_1 + a_1 + b_1 + d_1 + b'_1\\
l'_{21} &= l_2 + a_0 + a'_0 + b'_0 + c'_0\\
l'_{22} &= c_1 + a'_1 + c'_1 + d'_1\\
l'_{31} &= l_3 + a_1 + a'_1 + b'_1 + c'_1\\
l'_{32} &= c_0 + a'_0 + c'_0 + d'_0\\
m'_0 &= k_0 + m_0 + n_0 + a_0 + c_0 + d_0 + c'_0\\
m'_1 &= k_1 + m_1 + n_1 + a_1 + c_1 + d_1 + c'_1\\
m'_{21} &= m_2 + b_1 + b'_1 + c_1 + c'_1\\
m'_{22} &= a_0 + a'_0 + b'_0 + c'_0\\
m'_{31} &= m_3 + b_0 + b'_0 + c_0 + c'_0\\
m'_{32} &= a_1 + a'_1 + b'_1 + c'_1\\
n'_0 &= l_0 + m_0 + n_0 + b_0 + c_0 + d_0 + d'_0\\
n'_1 &= l_1 + m_1 + n_1 + b_1 + c_1 + d_1 + d'_1
\end{aligned}
$$

Stage 2

$x_0 = k_0$
$x_1 = k_1$
$y_0 = l_0 + l_2 + (l_0 + k'_0 + l'_0 + n'_0)(l_2 + l'_{21} + l'_{22})$
$y_1 = l_1 + l_3 + (l_1 + k'_1 + l'_1 + n'_1)(l_3 + l'_{31} + l'_{32})$
$z_0 = m_0 + m_2 + (m_0 + k'_0 + m'_0 + n'_0)(m_2 + m'_{21} + m'_{22})$
$z_1 = m_1 + m_3 + (m_1 + k'_1 + m'_1 + n'_1)(m_3 + m'_{31} + m'_{32})$
$w_0 = n_0$
$w_1 = n_1$

$x'_0 = x_0 + k_0 + k'_0 + l'_{21} + l'_{22} + m'_{21} + m'_{22}$
$x'_1 = x_1 + k_1 + k'_1 + l'_{31} + l'_{32} + m'_{31} + m'_{32}$
$y'_0 = x_0 + w_0 + k_0 + n_0 + l'_0 + l'_{21} + l'_{22}$
$y'_1 = x_1 + w_1 + k_1 + n_1 + l'_1 + l'_{31} + l'_{32}$
$z'_0 = x_0 + w_0 + k_0 + n_0 + m'_0 + m'_{21} + m'_{22}$
$z'_1 = x_1 + w_1 + k_1 + n_1 + m'_1 + m'_{31} + m'_{32}$
$w'_0 = w_0 + l_0 + n'_0 + l'_{21} + l'_{22} + m'_{21} + m'_{22}$
$w'_1 = w_1 + n_1 + n'_1 + l'_{31} + l'_{32} + m'_{31} + m'_{32}$

C Four Sharing of χ

The χ function which is the nonlinear operation in Keccak is given as follows in its algebraic normal form

$$\begin{aligned}
&\chi(a,b,c,d.e) = x,y,z,w,q\\
&x = f_0(a,b,c,d,e) = de + a + d\\
&y = f_1(a,b,c,d,e) = ae + b + e\\
&z = f_2(a,b,c,d,e) = ab + a + c\\
&w = f_3(a,b,c,d,e) = bc + b + d\\
&q = f_4(a,b,c,d,e) = cd + c + e\,.
\end{aligned}$$

Duplication Code The correct, non-complete, uniform, and stable sharing of the χ function $\bar{\chi}(\bar{a},\bar{b},\bar{c},\bar{d},\bar{e},\bar{a}',\bar{b}',\bar{c}',\bar{d}',\bar{e}') = (\bar{x},\bar{y},\bar{z},\bar{w},\bar{q},\bar{x}',\bar{y}',\bar{z}',\bar{w}',\bar{q}')$ is given below.

$$\begin{aligned}
x_0 &= a_0 + c_0\\
x_1 &= a_1 + c_1 + (b_1 + b_2 + b_3)(c_1 + c_2 + c_3)\\
x_2 &= a_2 + c_2 + b_0(c_2 + c_3) + c_0(b_2 + b_3) + b_0c_0\\
x_3 &= a_3 + c_3 + b_0c_1 + c_0b_1\\
y_0 &= b_0 + d_0\\
y_1 &= b_1 + d_1 + (c_1 + c_2 + c_3)(d_1 + d_2 + d_3)\\
y_2 &= b_2 + d_2 + c_0(d_2 + d_3) + d_0(c_2 + c_3) + c_0d_0\\
y_3 &= b_3 + d_3 + c_0d_1 + d_0c_1\\
z_0 &= c_0 + e_0\\
z_1 &= c_1 + e_1 + (d_1 + d_2 + d_3)(e_1 + e_2 + e_3)\\
z_2 &= c_2 + e_2 + d_0(e_2 + e_3) + e_0(d_2 + d_3) + d_0e_0\\
z_3 &= c_3 + e_3 + d_0e_1 + e_0d_1\\
w_0 &= d_0\\
w_1 &= d_1 + (1 + e_1 + e_2 + e_3)(a_1 + a_2 + a_3)\\
w_2 &= d_2 + a_0 + e_0(a_2 + a_3) + a_0(e_2 + e_3) + e_0a_0\\
w_3 &= d_3 + e_0a_1 + a_0e_1\\
q_0 &= e_0 + b_0\\
q_1 &= e_1 + b_1 + (a_1 + a_2 + a_3)(b_1 + b_2 + b_3)\\
q_2 &= e_2 + b_2 + a_0(b_2 + b_3) + b_0(a_2 + a_3) + a_0b_0\\
q_3 &= e_3 + b_3 + a_0b_1 + b_0a_1
\end{aligned}$$

$$
\begin{aligned}
x'_0 &= a'_0 + c'_0 \\
x'_1 &= a'_1 + c'_1 + (b'_1 + b'_2 + b'_3)(c'_1 + c'_2 + c'_3) \\
x'_2 &= a'_2 + c'_2 + b'_0(c'_2 + c'_3) + c'_0(b'_2 + b'_3) + b'_0c'_0 \\
x'_3 &= a'_3 + c'_3 + b'_0c'_1 + c'_0b'_1 \\
y'_0 &= b'_0 + d'_0 \\
y'_1 &= b'_1 + d'_1 + (c'_1 + c'_2 + c'_3)(d'_1 + d'_2 + d'_3) \\
y'_2 &= b'_2 + d'_2 + c'_0(d'_2 + d'_3) + d'_0(c'_2 + c'_3) + c'_0d'_0 \\
y'_3 &= b'_3 + d'_3 + c'_0d'_1 + d'_0c'_1 \\
z'_0 &= c'_0 + e'_0 \\
z'_1 &= c'_1 + e'_1 + (d'_1 + d'_2 + d'_3)(e'_1 + e'_2 + e'_3) \\
z'_2 &= c'_2 + e'_2 + d'_0(e'_2 + e'_3) + e'_0(d'_2 + d'_3) + d'_0e'_0 \\
z'_3 &= c'_3 + e'_3 + d'_0e'_1 + e'_0d'_1 \\
w'_0 &= d'_0 \\
w'_1 &= d'_1 + (1 + e'_1 + e'_2 + e'_3)(a'_1 + a'_2 + a'_3) \\
w'_2 &= d'_2 + a'_0 + e'_0(a'_2 + a'_3) + a'_0(e'_2 + e'_3) + e'_0a'_0 \\
w'_3 &= d'_3 + e'_0a'_1 + a'_0e'_1 \\
q'_0 &= e'_0 + b'_0 \\
q'_1 &= e'_1 + b'_1 + (a'_1 + a'_2 + a'_3)(b'_1 + b'_2 + b'_3) \\
q'_2 &= e'_2 + b'_2 + a'_0(b'_2 + b'_3) + b'_0(a'_2 + a'_3) + a'_0b'_0 \\
q'_3 &= e'_3 + b'_3 + a'_0b'_1 + b'_0a'_1
\end{aligned}
$$

Parity Code The systematic $[6, 5, 2]$ parity code computes the one bit parity for each 5-bit chunk with the following generator matrix

$$
G_{[6,5,2]} = \begin{pmatrix} 1\,0\,0\,0\,0\,\;1 \\ 0\,0\,1\,0\,0\,\;1 \\ 0\,0\,0\,1\,0\,\;1 \\ 0\,0\,0\,0\,1\,\;1 \end{pmatrix}.
$$

We denote the parity bit of the input p which is defined as $p = a+b+c+d+e$. Similarly, the parity bit of the output is denoted o such that $o = x + y + z + w + q$. The correct, non-complete, uniform, and stable sharing $\bar{\chi}(\bar{a}, \bar{b}, \bar{c}, \bar{d}, \bar{e}, \bar{p}) = (\bar{x}, \bar{y}, \bar{z}, \bar{w}, \bar{q}, \bar{o})$ encoded with the $[6, 5, 2]$ code is given below.

$$
\begin{aligned}
x_0 &= a_0 + c_0 \\
x_1 &= a_1 + c_1 + (b_1 + b_2 + b_3)(c_1 + c_2 + c_3) \\
x_2 &= a_2 + c_2 + b_0(c_2 + c_3) + c_0(b_2 + b_3) + b_0 c_0 \\
x_3 &= a_3 + c_3 + b_0 c_1 + c_0 b_1 \\
y_0 &= b_0 + d_0 \\
y_1 &= b_1 + d_1 + (c_1 + c_2 + c_3)(d_1 + d_2 + d_3) \\
y_2 &= b_2 + d_2 + c_0(d_2 + d_3) + d_0(c_2 + c_3) + c_0 d_0 \\
y_3 &= b_3 + d_3 + c_0 d_1 + d_0 c_1 \\
z_0 &= c_0 + e_0 \\
z_1 &= c_1 + e_1 + (d_1 + d_2 + d_3)(e_1 + e_2 + e_3) \\
z_2 &= c_2 + e_2 + d_0(e_2 + e_3) + e_0(d_2 + d_3) + d_0 e_0 \\
z_3 &= c_3 + e_3 + d_0 e_1 + e_0 d_1 \\
w_0 &= d_0 \\
w_1 &= d_1 + (1 + e_1 + e_2 + e_3)(a_1 + a_2 + a_3) \\
w_2 &= d_2 + a_0 + e_0(a_2 + a_3) + a_0(e_2 + e_3) + e_0 a_0 \\
w_3 &= d_3 + e_0 a_1 + a_0 e_1 \\
q_0 &= e_0 + b_0 \\
q_1 &= e_1 + b_1 + (a_1 + a_2 + a_3)(b_1 + b_2 + b_3) \\
q_2 &= e_2 + b_2 + a_0(b_2 + b_3) + b_0(a_2 + a_3) + a_0 b_0 \\
q_3 &= e_3 + b_3 + a_0 b_1 + b_0 a_1
\end{aligned}
$$

$$
\begin{aligned}
o_0 &= p_0 + b_0 + c_0 + d_0 + e_0 \\
o_1 &= p_1 + b_1 + c_1 + d_1 + e_1 + (a_1 + a_2 + a_3 + d_1 + d_2 + d_3)(e_1 + e_2 + e_3) + \\
&\quad (b_1 + b_2 + b_3 + d_1 + d_2 + d_3)(c_1 + c_2 + c_3) + (a_1 + a_2 + a_3)(b_1 + b_2 + b_3 + 1) \\
o_2 &= p_2 + a_0 + b_2 + c_2 + d_2 + e_2 + (a_0 + d_0)(e_2 + e_3) + (c_0 + e_0)(d_2 + d_3) + \\
&\quad (b_0 + e_0)(a_0 + a_2 + a_3) + (b_0 + d_0)(c_0 + c_2 + c_3) + (a_0 + c_0)(b_2 + b_3) + d_0 e_0 \\
o_3 &= p_3 + b_3 + c_3 + d_3 + e_3 + a_0(b_1 + e_1) + b_0(a_1 + c_1) + c_0(b_1 + d_1) + \\
&\quad d_0(c_1 + e_1) + e_0(a_1 + d_1)
\end{aligned}
$$

D Two Sharing of χ

Duplication Code The correct, non-complete, uniform, and stable sharing of the χ function $\bar{\chi}(\bar{a}, \bar{b}, \bar{c}, \bar{d}, \bar{e}, \bar{a}', \bar{b}', \bar{c}', \bar{d}', \bar{e}') = (\bar{x}, \bar{y}, \bar{z}, \bar{w}, \bar{q}, \bar{x}', \bar{y}', \bar{z}', \bar{w}', \bar{q}')$ is given.

Stage 1	Stage 2
$k_0 = d_0e_0 + a_0 + d_0$	
$k_1 = d_0e_1$	$x_0 = k_0 + k_1 + (k_0 + k_0')(k_1 + k_1')$
$k_2 = d_1e_0 + a_1 + d_1$	$x_1 = k_2 + k_3 + (k_2 + k_2')(k_3 + k_3')$
$k_3 = d_1e_1$	
$l_0 = a_0e_0$	
$l_1 = a_0e_1 + b_0$	$y_0 = l_0 + l_1 + (l_0 + l_0')(l_1 + l_1')$
$l_2 = a_1e_0 + e_0$	$y_1 = l_2 + l_3 + (l_2 + l_2')(l_3 + l_3')$
$l_3 = a_1e_1 + b_1 + e_1$	
$m_0 = a_0b_0$	
$m_1 = a_0b_1 + a_0 + c_0$	$z_0 = m_0 + m_1 + (m_0 + m_0')(m_1 + m_1')$
$m_2 = a_1b_0 + a_1 + c_1$	$z_1 = m_2 + m_3 + (m_2 + m_2')(m_3 + m_3')$
$m_3 = a_1b_1$	
$n_0 = b_0c_0 + a_0 + d_0$	
$n_1 = b_0c_1 + a_0 + b_0$	$w_0 = n_0 + n_1 + (n_0 + n_0')(n_1 + n_1')$
$n_2 = b_1c_0$	$w_1 = n_2 + n_3 + (n_2 + n_2')(n_3 + n_3')$
$n_3 = b_1c_1 + b_1 + d_1$	
$p_0 = c_0d_0 + e_0$	
$p_1 = c_0d_1 + c_0$	$q_0 = p_0 + p_1 + (p_0 + p_0')(p_1 + p_1')$
$p_2 = c_1d_0 + a_1 + b_1 + e_1$	$q_1 = p_2 + p_3 + (p_2 + p_2')(p_3 + p_3')$
$p_3 = c_1d_1 + a_1 + b_1 + c_1$	
$k_0' = d_0'e_0' + a_0' + d_0'$	
$k_1' = d_0'e_1'$	$x_0' = k_0' + k_1'$
$k_2' = d_1'e_0' + a_1' + d_1'$	$x_1' = k_2' + k_3'$
$k_3' = d_1'e_1'$	
$l_0' = a_0'e_0'$	
$l_1' = a_0'e_1' + b_0'$	$y_0' = l_0' + l_1'$
$l_2' = a_1'e_0' + e_0'$	$y_1' = l_2' + l_3'$
$l_3' = a_1'e_1' + b_1' + e_1'$	
$m_0' = a_0'b_0'$	
$m_1' = a_0'b_1' + a_0' + c_0'$	$z_0' = m_0' + m_1'$
$m_2' = a_1'b_0' + a_1' + c_1'$	$z_1' = m_2' + m_3'$
$m_3' = a_1'b_1'$	
$n_0' = b_0'c_0' + a_0' + d_0'$	
$n_1' = b_0'c_1' + a_0' + b_0'$	$w_0' = n_0' + n_1'$
$n_2' = b_1'c_0'$	$w_1' = n_2' + n_3'$
$n_3' = b_1'c_1' + b_1' + d_1'$	
$p_0' = c_0'd_0' + e_0'$	
$p_1' = c_0'd_1' + c_0'$	$q_0' = p_0' + p_1'$
$p_2' = c_1'd_0' + a_1' + b_1' + e_1'$	$q_1' = p_2' + p_3'$
$p_3' = c_1'd_1' + a_1' + b_1' + c_1'$	

Parity Code The $[6, 5, 2]$ encoded correct, non-complete, uniform, and stable sharing $\bar{\chi}(\bar{a}, \bar{b}, \bar{c}, \bar{d}, \bar{e}, \bar{t}) = (\bar{x}, \bar{y}, \bar{z}, \bar{w}, \bar{q}, \bar{o})$ is given below.

Stage 1	Stage 2
$k_0, k'_0 = d_0e_0 + a_0 + d_0$	
$k_1, k'_1 = d_0e_1$	$x_0 = k_0 + k_1 + (k_0 + k'_0)(k_1 + k'_1)$
$k_2, k'_2 = d_1e_0 + a_1 + d_1$	$x_1 = k_2 + k_3 + (k_2 + k'_2)(k_3 + k'_3)$
$k_3, k'_3 = d_1e_1$	
$l_0, l'_0 = a_0e_0$	
$l_1, l'_1 = a_0e_1 + b_0$	$y_0 = l_0 + l_1 + (l_0 + l'_0)(l_1 + l'_1)$
$l_2, l'_2 = a_1e_0 + e_0$	$y_1 = l_1 + l_3 + (l_2 + l'_2)(l_3 + l'_3)$
$l_3, l'_3 = a_1e_1 + b_1 + e_1$	
$m_0, m'_0 = a_0b_0$	
$m_1, m'_1 = a_0b_1 + a_0 + c_0$	$z_0 = m_0 + m_1 + (m_0 + m'_0)(m_1 + m'_1)$
$m_2, m'_2 = a_1b_0 + a_1 + c_1$	$z_1 = m_2 + m_3 + (m_2 + m'_2)(m_3 + m'_3)$
$m_3, m'_3 = a_1b_1$	
$n_0, n'_0 = b_0c_0 + a_0 + d_0$	
$n_1, n'_1 = b_0c_1 + a_0 + b_0$	$w_0 = n_0 + n_1 + (n_0 + n'_0)(n_1 + n'_1)$
$n_2, n'_2 = b_1c_0$	$w_1 = n_2 + n_3 + (n_2 + n'_2)(n_3 + n'_3)$
$n_3, n'_3 = b_1c_1 + b_1 + d_1$	
$p_0, p'_0 = c_0d_0 + e_0$	
$p_1, p'_1 = c_0d_1 + c_0$	$q_0 = p_0 + p_1 + (p_0 + p'_0)(p_1 + p'_1)$
$p_2, p'_2 = c_1d_0 + a_1 + b_1 + e_1$	$q_1 = p_2 + p_3 + (p_2 + p'_2)(p_3 + p'_3)$
$p_3, p'_3 = c_1d_1 + a_1 + b_1 + c_1$	
$t'_0 = k_0 + l_0 + m_0 + n_0 + p_0 +$ $a_0 + b_0 + c_0 + d_0 + e_0 + t_0$	$o_0 = k_1 + l_1 + m_1 + n_1 + p_1 + t'_0$
$t'_1 = k_3 + l_3 + m_3 + n_3 + p_3 +$ $a_1 + b_1 + c_1 + d_1 + e_1 + t_1$	$o_1 = k_2 + l_2 + m_2 + n_2 + p_2 + t'_1$