ParTI – Towards Combined Hardware Countermeasures against Side-Channel and Fault-Injection Attacks

Tobias Schneider[1], Amir Moradi[1], and Tim Güneysu[2]

[1] Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
{tobias.schneider-a7a, amir.moradi}@rub.de
[2] University of Bremen and DFKI, Germany
tim.gueneysu@uni-bremen.de

Abstract. Side-channel analysis and fault-injection attacks are known as major threats to any cryptographic implementation. Hardening cryptographic implementations with appropriate countermeasures is thus essential before they are deployed in the wild. However, countermeasures for both threats are of completely different nature: Side-channel analysis is mitigated by techniques that hide or mask key-dependent information while resistance against fault-injection attacks can be achieved by redundancy in the computation for immediate error detection. Since already the integration of any single countermeasure in cryptographic hardware comes with significant costs in terms of performance and area, a combination of multiple countermeasures is expensive and often associated with undesired side effects.
In this work, we introduce a countermeasure for cryptographic hardware implementations that combines the concept of a provably-secure masking scheme (i.e., threshold implementation) with an error detecting approach against fault injection. As a case study, we apply our generic construction to the lightweight LED cipher. Our LED instance achieves first-order resistance against side-channel attacks combined with a fault detection capability that is superior to that of simple duplication for most error distributions at an increased area demand of 12%.

1 Introduction

Over the last years, implementation attacks have seen a rise in popularity due to their ability to break cryptographic implementations which were believed to be cryptanalytically secure. Their power is based on vulnerabilities in the physical implementation instead of flaws in the cryptographic algorithm. The two most popular types of implementation attacks are side-channel analysis (SCA) and fault injection (FI) attacks.

SCA are passive attacks which exploit the information leakage related to cryptographic device internals through side channels, e.g., power consumption of a device [24]. Usually they involve a considerable number of measurements and statistical tools to extract the sensitive information from the device. Over

the years, various different types of attacks have been proposed with a large variety in capabilities and complexity. As a consequence, a wide range of countermeasures has been developed to thwart these attacks. Compared to other types of countermeasures, masking (as a form of secret sharing) has attracted the most interest inside the side-channel community. With its sound theoretical foundation, masking can be applied at different levels of abstraction to secure designs. Still the secure implementation of a masking scheme remains a major challenge since effects such as glitches in hardware circuits can completely invalidate the security assumptions of the schemes [26, 27]. In response, Threshold Implementation (TI) [32], as a concept between Boolean masking and multi-party computation, has been specifically developed for hardware platforms to maintain security properties even in the presence of glitches. The TI concept has been applied to many algorithms including PRESENT [36], AES [5, 16, 29], KATAN [6, 31], Keccak [4], arithmetic addition [43], Simon [44], PRINCE and Midori [30], and all 4-bit Sboxes [8].

Active FI attacks pose a further serious threat to instantiated cryptographic algorithms [3] by injecting a fault during its execution. The adversary then derives sensitive information from the erroneous output of the device. For more sophisticated attacks on symmetric schemes to work, multiple of these erroneous outputs need to be combined. Like for SCA, there are a wealth of attacks and possibilities to generate faults during the computation, e.g., by clock or power glitches or positioned photon injection using lasers. In terms of countermeasures, the majority of published concepts are based on the principle of concurrent error detection (CED). The main idea is to utilize redundancy in time or area to enable quasi-immediate detection of faults. Some CED schemes integrate the use of error detecting codes to enhance their level of protection. Over the years, various different codes have been studied to harden cryptographic implementations against FI attacks. Due to its simplicity, parity check codes are commonly used in this context [2, 23]. Other schemes based on non-linear codes (e.g., [21, 22]) were brought up to their beneficial fault coverage. Recently, the class of infective countermeasures have been put forward which do not require an explicit final check before returning the result. [17].

As previously discussed, it is mandatory for cryptographic devices to integrate dedicated SCA and FI countermeasures if they are operated in untrusted environments. Still the majority of proposed SCA and FI countermeasures have been solely evaluated separately, though both classes need to be integrated in a single device. For simple countermeasures (e.g., applying plain redundancy in area and time), a separate evaluation is justified since multiple executions of the same SCA-protected operation are admissible (with few exceptions). However, more sophisticated FI countermeasures are likely to affect the SCA countermeasure to a higher degree which can have a severe impact on the security and efficiency of the combined scheme. For example, if parity bits used by FI countermeasures are computed over *unmasked* intermediate values, it leads to a side-channel leakage even if the rest of the design in perfectly masked. Thus,

a careless integration may easily lead to contradicting the assumptions of the underlying masking scheme, and hence failure of the masked design.

Related Work. In response, a few countermeasures have been proposed providing resistance against both kind of attacks. At the gate level, we refer to dual-rail logic styles (e.g., WDDL [46]) which – due to the additional presence of dual counterparts of the circuit – inherently offer a fault detection feature. However, the error detection rate is limited to the concept of simple duplication.

Furthermore, coding schemes have been used for combined countermeasures as well. Wiretap codes that have been applied as an SCA countermeasure [12,28] at algorithm level, can also provide a certain level of fault detection. Additionally, there are further examples [11] that use coding techniques for enhanced resistance against both types of attacks. However, most of the schemes are either designed for software implementations or provide only limited security at the expense of high overheads.

Besides combined countermeasures, there are also combined attacks which use a combination of fault injection and side-channel analysis to extract a secret. Several different attacks have been proposed against protected AES implementations where masking together with various fault countermeasures are integrated [13, 15, 39]. Our analyses consider this powerful threat as well.

Our Contribution. We propose a new combined physical protection scheme targeting hardware platforms. As mentioned before, the integration of CED schemes by simple *time* or *area redundancy* into masked designs is straightforward (see [48] for definitions). However, such constructions are not able to detect certain types of faults (e.g., identical faults which are injected in both instances of the design) and rather costly. Therefore, our target in this work is to merge more sophisticated *information redundancy* approaches (namely *error detecting codes*) with provably secure masked hardware designs. More precisely, we demonstrate how to integrate an error detecting code into first- (or higher-) order TI designs, while preserving all security requirements and features of the underlying TI concept. We formalize our methodology to allow various types of codes which provide the most flexibility in terms of protection and area requirement. We include a thorough analysis on the resistance of the combined countermeasure regarding the chosen order of TI and the parameters of the code. Note that the straightforward hardware duplication can be regarded as subtype of our combined countermeasure, but our generic concept enables to tweak the protection of the resulting design by the choice of code.

For practical evaluation we present a case-study on the cipher LED [19] that simplifies the explanation of the underlying concept due to its simple structure. We provide practical evaluations of our design implemented on an FPGA with respect to any detectable first-order leakage. Moreover, we evaluate the performance, area overhead as well as the fault coverage of the integrated *information redundancy* scheme. Note that the representations included in this work primarily discuss the case of a first-order TI design of LED with fault detection facility

based on Hamming codes. But we like to emphasize that our generic construction can be similarly applied to any-order TI designs of other ciphers that are using different error detecting codes.

2 Background

For self-containedness of this work we briefly discuss the relevant background in this section.

2.1 Threshold Implementation

We briefly review the concept of Threshold Implementations (TI). For detailed information we refer the interested reader to the original articles [6, 33].

For simplicity but without loss of generality, let us assume a 4-bit intermediate value of an arbitrary cipher with 4-bit S-Box, e.g., PRESENT [10] or LED [19]. We denote this corresponding 4-bit value as $\boldsymbol{x} = \langle x_1, \ldots, x_4 \rangle$. In TI, $\boldsymbol{x}$ is represented in $n-1$ order Boolean masked form $(\boldsymbol{x}^1, \ldots, \boldsymbol{x}^n)$, where $\boldsymbol{x} = \bigoplus_{i=1}^{n} \boldsymbol{x}^i$ and each $\boldsymbol{x}^i$ similarly denotes a 4-bit vector $\langle x_1^i, \ldots, x_4^i \rangle$.

The linear functions, such as MixColumns of AES or LED, can be simply applied to the shares of $\boldsymbol{x}$ by $\mathsf{L}(\boldsymbol{x}) = \bigoplus_{i=1}^{n} \mathsf{L}(\boldsymbol{x}^i)$. However, the realization of the S-Box over Boolean masking is not trivial. If the algebraic degree of the S-Box is denoted by t, the minimum number of shares to realize an S-Box protected against first-order attacks is $n = t+1$. To ensure *correctness* for the computation, this S-Box needs to provide the output $\boldsymbol{y} = \mathsf{S}(\boldsymbol{x})$ in shared form $(\boldsymbol{y}^1, \ldots, \boldsymbol{y}^m)$ with $\boldsymbol{y} = \bigoplus_{i=1}^{m} \boldsymbol{y}^i$ and $m \geq n$ for bijective S-Boxes. In case of bijective S-Boxes (e.g., as for AES, PRESENT, and LED) the bit length of $\boldsymbol{x}$ and $\boldsymbol{y}$ (respectively of their shared forms) are identical.

Each output share $\boldsymbol{y}^{j \in \{1,\ldots,m\}}$ is given by a component function $\mathsf{f}^j(\cdot)$ over a subset of input shares. For first-order security, each component function $\mathsf{f}^{j \in \{1,\ldots,m\}}(\cdot)$ must be independent of at least one input share. This requirement on the independence from at least one share is defined as *non-completeness* property.

Since the security of masking schemes is based on the uniform distribution of the masks, the output of a TI S-Box must be also uniform since it is used as input in further parts of the implementation (e.g., the S-Box output of one cipher round which is given to the next S-Box after being processed by the linear diffusion layers). This property of *uniformity* requires for a bijective S-Box ($n = m$) each $(\boldsymbol{x}^1, \ldots, \boldsymbol{x}^n)$ to be mapped to a unique $(\boldsymbol{y}^1, \ldots, \boldsymbol{y}^n)$. In other words, it is sufficient in this case to check whether the TI S-Box forms a bijection with a $4 \cdot n$ input (and output) bit length.

As an example, take the PRESENT (LED) S-Box which has algebraic degree of $t = 3$. Hence, the number of input and output shares $n = m > 3$ what directly affects the complexity of the circuit and its associated area overhead.

Therefore, it is preferable to decompose the S-Box $\mathsf{S}(\cdot)$ into smaller bijections, e.g., $\mathsf{g} \circ \mathsf{f}(.)$, each with maximum algebraic degree of 2. The authors of [36] presented a decomposition of the PRESENT S-Box into two bijections g and f, each of which with an algebraic degree of 2. These parameters keep the number of shares for input and output at a minimum, i.e., $n = m = 3$.

2.2 Error Detecting Codes

Error detecting codes (EDC) are primarily used to transmit data over an unreliable communication channel. Those properties and notation of EDC that are also relevant for remainder of this work will be highlighted in the following [25].

Definition 1. *A linear code* $\mathbf{C}$ *of length* n *over* $\mathbb{F}_q$ *is a vector subspace over* $\mathbb{F}_q^n$.

We only consider binary codes (i.e., $q = 2$) in this work since they provide the best performance for our projected use-case in symmetric cryptography. A linear code $\mathbf{C}$ that maps messages of length k to codewords of length n is commonly denoted as an $[n, k]$-code.

Definition 2. *A generator matrix* G *of an* $[n, k]$*-code* $\mathbf{C}$ *comprises* n *basis vectors of* $\mathbf{C}$ *with length* k.

A generator matrix can be used to transform a given message $m \in \mathbb{F}_q^k$ to the corresponding code word $c \in \mathbf{C}$ as $c = m \cdot G$.

Definition 3. *A matrix* $H \in \mathbb{F}_q^{(n-k)\times n}$ *with the property*

$$\mathbf{0} = H \cdot c^T, \ \forall c \in \mathbf{C} \tag{1}$$

is denoted as parity check matrix of the code $\mathbf{C}$.

Such matrix can be used to easily check if a given c is a valid codeword of $\mathbf{C}$.

Definition 4. *The minimum distance* d *of a linear code* $\mathbf{C}$ *is defined as*

$$d = min(\{wt\,(c_1 \oplus c_2)\,|c_1, c_2 \in \mathbf{C}, c_1 \neq c_2\}), \tag{2}$$

where $wt(x)$ returns the number of 1's in the vector x (known as Hamming weight). We denote a linear code $\mathbf{C}$ of length n, rank k and minimum distance d as an $[n, k, d]$-code. The minimum distance of a code determines its error detection and correction property.

Definition 5. *A code* $\mathbf{C}$ *with minimum distance* d *can be used to either detect* $u = d - 1$ *or correct* $v = \left\lfloor \frac{d-1}{2} \right\rfloor$ *errors. If* d *is even,* $\mathbf{C}$ *can simultaneously detect* $u = \frac{d}{2}$ *and correct* $v = \frac{d-2}{2}$ *errors.*

Given an erroneous codeword $c' = c \oplus e$, where e is known as the error vector, a u-error detecting code is able to detect that c' is faulty as long as $wt(e) \leq u$.

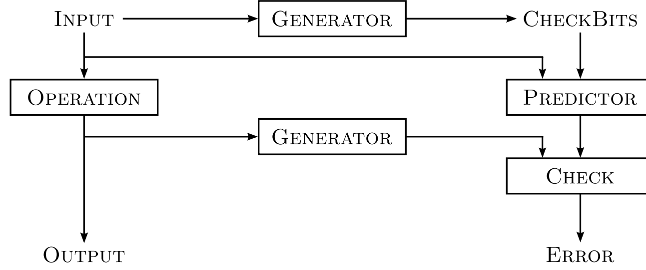


Fig. 1. A common structure of CED schemes using EDC.

Definition 6. *The generator matrix G of a systematic code $\mathbf{C}$ is of the form $G = [I_k|P]$ where I_k denotes the identity matrix of size k.*

Each codeword c of a systematic code consist of the message itself which is padded by *check bits*, i.e, $c = [m|p]$. The check bits p[1] are generated by the rearward part of the generator matrix G represented by P. Note that all linear non-systematic codes can be transformed into a systematic code with the same minimum distance [9].

2.3 Concurrent Error Detection

Concurrent Error Detection (CED) systems are commonly used to detect arbitrary faults during the execution of an operation what makes them also an appropriate countermeasure against FI attacks [20]. Typically CED techniques rely on different types of redundancy to detect faulty computations. The most straightforward approach implements redundancy by multiple executions which results either in an increased area or in an increased time complexity. Certain intermediate values of different runs are compared with each other to detect errors.

As already indicated in the introduction, some CED schemes use error detecting codes following a structure similar to Figure 1 to achieve a better fault coverage. In this basic example, CED is used to protect an Operation which is applied to a given Input. Initially, the CheckBits of the Input are generated by means of the Generator matrix of the code. A Predictor takes Input and CheckBits and returns the predicted check bits of the output of Operation. These are compared with the actual CheckBits of the output. If a detectable error (depending on the type of the code) occurred during the execution, these two types of CheckBits will not be identical. Thus, a possible attack can be detected and averted. It should be noted that, depending on the target algorithm and the integrated code, the prediction functions can have an exalted level of complexity. Thus, the overhead of some CED schemes using EDC can be similar to a complete duplication of the operation.

[1] Note that p can be also considered as a form of parity bits.

Traditionally, the effectiveness of these fault detection countermeasures was examined in a uniform fault model. However, recent publications [20] have shown that this model does not closely resemble real-world attacks and that some of the presented countermeasures are in fact vulnerable to *biased fault attacks* [35].

3 Methodology

In this section, we introduce our methodology to develop a combined countermeasure against side-channel and fault injection attacks that is specifically tailored for hardware platforms. We first discuss the necessary considerations and restrictions of a combined scheme. This is followed by a detailed description of the attacker model and how to design a scheme to support arbitrary applications.

3.1 Design Considerations

Firstly, our countermeasure is designed for hardware platforms. Thus, efficiency in software is not a concern in the design process. As hardware circuits are often used to achieve high performance, a primary design goal is to minimize the impact of the countermeasure on the performance.

Secondly, in terms of SCA countermeasure we aim at providing provable security (at least to a certain order). Therefore, hiding techniques are not applicable and we have to rely on masking. Given the first design goal, this leaves us with TI as it comes with a reasonable performance overhead compared to other masking schemes in hardware circuits [37].

Thirdly, our scheme aims to be more secure against (realistic) FI attacks than simple duplication. Doubling the masked hardware circuit is a straightforward way to combine masking with some form of redundancy. However as mentioned before, simple duplication can be highly vulnerable to fault attacks if the fault model follows a different distribution than uniform. Therefore, we aim at building a scheme that is more robust against adversaries exploiting the effect of (reasonably) biased distributions. In this context, we choose EDC due to their sound theoretical foundation providing solid bounds on the number of detectable errors. Nevertheless, the balance between the error detection capability and runtime performance is essential to not severely impact our first design goal.

3.2 Attacker Model

Since our scheme aims to provide resistance against both SCA and FI attacks, we evaluate our methodology in a model that incorporates both types of threats. In the following, we assume an adversary that can observe the physical characteristics of the design during execution and further is able to inject faults in the circuit.

We assume a computationally bounded adversary that can observe the power consumption of our design during a finite number of executions. Note that security guarantees of TI also hold with respect to other side channels, e.g., electromagnetic emanations. Due to the computational restriction of the adversary, we can bound the number of possible observations. Given that the complexity of an attack increases with its order, we bound the adversary by the highest order of an attack he is able to mount. In other words, the adversary is able to observe a limited number of executions that is just enough to perform attacks of order d but not of order $d+1$. The actual order depends on the platform and the desired level of security.

Furthermore, the adversary is able to inject faults in the hardware circuit. In our model, we assume that injected faults only target the data path of the implementation and exclude the control flow. This is a different aspect of fault attacks which is not specific to our scheme. Given that the control flow usually does not need to be protected by masking, it is not considered in our combined countermeasure. Nevertheless, our combined countermeasure needs to be implemented together with a protected control flow to ensure complete security. There are various solutions to this problem. Even the EDC aspect from our combined countermeasure can be used to harden the control flow as described in [45]. Therefore, we model the injected faults as an error random variable E following a specific distribution $\mathcal{E}$. In our model, an error vector $e \in \mathbb{F}_q^n$ with probability $Pr[E = e]$ is sampled for each injected fault from the distribution and added (XORed) to the current *state* of the execution as $state' = state \oplus e$. The execution continues the computation with the altered state $state'$. In the following, we consider two different types of the error distribution.

Since most existing works assume a uniform fault model, we also first examine our combined countermeasures against an adversary with a uniform distribution $\mathcal{E}_U$ so that $Pr[E = e_1] = Pr[E = e_2], \forall e_1, e_2 \in E$.

Furthermore, we consider a biased distribution $\mathcal{E}_B$, where one specific set of error vectors $E_1 \subset E$ is significantly more probable than the set of remaining error vectors $E_2 \subset E$ with $Pr[E = e_1] \gg Pr[E = e_2], \forall e_1 \in E_1, e_2 \in E_2$. The sets are determined by the type of faults that are considered in the model, e.g., $E_1 : \forall e, wt(e) \leq u$. In an extreme case, E_1 only contains one specific error vector e with $wt(e) = 1$. This scenario is akin to laser-based fault injections in which single bits can be targeted.

3.3 Code Selection

Obviously, the choice of the code strongly affects the efficiency and fault coverage of the resulting combined scheme. In this context, it is not possible to provide one specific code that exhaustively fits to all possible application scenarios. Instead, the code needs to be specifically chosen according to the target algorithm to yield optimal results. A poorly chosen code can cause a significant overhead while offering only little benefit in terms of fault coverage. In this subsection we discuss about necessary considerations made in the code selection process and give guidelines on the criteria how to pick a code.

Linear Codes. One important aspect in the design of a TI is the algebraic degree of the targeted functions. As explained in Section 2.1, the algebraic degree determines the minimum number of necessary shares. Given that the prediction functions are also part of the intended TI, it is crucial that they possess the same algebraic degree as the original function. Otherwise, the requirement of an additional share negatively affects the area complexity of the resulting design. This property is trivially fulfilled by linear codes. The encoding and decoding functions of linear codes are linear. Therefore, adding a decoding function before and an encoding function after the target function (cf. Figure 3) to obtain the predictor guarantees that the emerging function has the same algebraic degree as the target function. For non-linear codes this property is not always satisfied. In addition, the encoding/decoding functions of linear codes can be implemented extremely efficiently which makes the necessary error check also very efficient. In the remainder of this work, we therefore only consider linear codes.

Systematic Codes. Systematic codes are advantageous to improve the efficiency. Due to their specially structured generator matrix (cf. Definition 6), the output of the targeted function does not need to be decoded to recover the correct result since the message is part of the codeword. This helps to eliminate one otherwise necessary step at the end of the design. Furthermore, the distinction between target function and predictor – as depicted in Figure 1 – is otherwise not easily possible. Since one half of the design is nearly completely unaltered by the inclusion of fault countermeasure, it also allows the reuse of existing TI designs. This design decision does not limit the choice of codes since (as already mentioned in Section 2.2) every linear non-systematic code can be transformed into a systematic code with the same minimum distance.

Code Parameters. The choice of the three parameters of a linear code n, k, and d depends on the target algorithm. A good practice is to derive the code dimension k from the size of a single element that is used in most functions of the targeted algorithm, e.g., for an algorithm that performs most of its operations in $GF(2^8)$ it is advisable to set $k = 8$. This way unnecessary overhead due to the split or merge of check bits is avoided. Furthermore, the code length n also affects both the efficiency and fault coverage of the design. To achieve a desired error detecting level of $u = d - 1$, a certain minimal size of n is required. However, if n is chosen too large, the number of check bits increases resulting in a high area complexity. Therefore, it is important to find a good tradeoff between the length of the code n and its minimum distance d. In the following, we aim at a design in which the predictors work solely on the check bits. To achieve this, it is necessary that the message m can be fully recovered using the check bits p. Assuming that the message has full entropy (which is usually the case in symmetric cryptographic applications), it is advisable to set the rank to at least $n \geq 2k$.

3.4 Threshold Implementations with Error Detecting Codes

To achieve the desired level of security against SCA adversaries, it is necessary to implement all required functions according to the principles of TI. In particular, this includes the prediction functions as well. As it was already thoroughly discussed in [7, 8] we omit the detailed explanation how to construct TI-compliant shared representations of arbitrary functions. Instead, we describe the specifics of including EDC in a TI design and how to easily find the TI of the predictors.

Notation. In the following, we assume a systematic linear code, which allows message recovery from the check bits. Further, we denote the input to the target algorithm by $\boldsymbol{m}_i$ with $n_m = |\boldsymbol{m}_i|$ as its bit length and the corresponding check bits as $\boldsymbol{p}_i$ with $n_p = |\boldsymbol{p}_i|$. The output of the target algorithm and its corresponding check bits are indicated by $\boldsymbol{m}_o$ and $\boldsymbol{p}_o$ respectively. Since the code does not change during the execution, the outputs have the same size as their corresponding inputs. Further, we assume that the TI of the target algorithm requires a minimum of s shares to be secure. To this end, the messages and their corresponding check bits need to be masked accordingly as

$$\boldsymbol{m}_i = \bigoplus_{j=1}^{s} \boldsymbol{m}_i^j, \qquad \boldsymbol{p}_i = \bigoplus_{j=1}^{s} \boldsymbol{p}_i^j, \qquad \boldsymbol{m}_o = \bigoplus_{j=1}^{s} \boldsymbol{m}_o^j, \qquad \boldsymbol{p}_o = \bigoplus_{j=1}^{s} \boldsymbol{p}_o^j.$$

Basic Structure. Due to the special characteristics of the chosen code, it is possible to split up the computations of the underlying target algorithm and the predictors. The two output values $\boldsymbol{m}_o$ and $\boldsymbol{p}_o$ are calculated completely independent of each other. This leads to the basic structure as depicted in Figure 2. There is an additional element (Error Check) which receives intermediate states of both circuits as input and checks if an error has occurred. The frequency for these checks is a variable in the specific design process, but it affects both the area and the fault coverage of the complete circuits. The higher the check frequency, the higher is the fault coverage but also the area requirements. In the most basic approach, only $\boldsymbol{m}_o$ and $\boldsymbol{p}_o$ are checked after a cipher run is complete.

For some TI designs a mask refresh during the execution is necessary to retain uniformity, e.g., for the AES S-Box [5]. Given that our proposed predictors are identical to the target function with an initial and final affine transformation, it is likely that they require a mask refresh depending on the shared function. Obviously, this can lead to a non-negligible overhead depending on the target algorithm. However, the separate computation paths (of the original and predictors) allow reusing the fresh randomness to some degree. Since both parts are completely independent and their respective intermediate values are never given to a joint function (except for the error check), it is possible to use the same random bits to refresh both sides. In the other case, where the predictors get inputs from both sides, this is not feasible without harming the uniformity property which would violate the security proofs of TI. For the error check, it is necessary to compute a function which takes inputs from both sides. However,

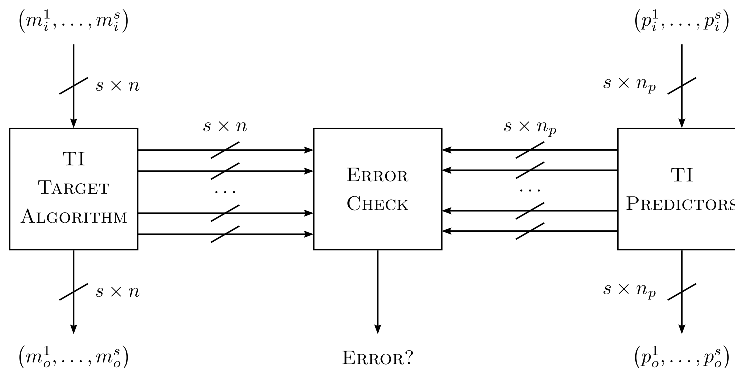


Fig. 2. The basic structure of our combined scheme.

in our scheme (and many others) this check can be implemented in a way that it only leaks the occurrence of a fault. For this to work, it is necessary that both sides use the same random masks which enables a separate error check on every share as also proposed as a countermeasure against combined attacks in [15,39]. This has obviously an impact on the efficiency of the design since the total amount of randomness is reduced due to the mask reuse.

We illustrate this problem with an example. Let us assume a function F with two input bits a, b with $F(a,b) = ab$. The corresponding check bit is defined as $c = a+b$ with the predictor $F_p(a,c) = a+ac$. As noted in [8], there is no uniform sharing of F. Instead, a virtual share is added to achieve uniformity. The shared functions using one virtual share are

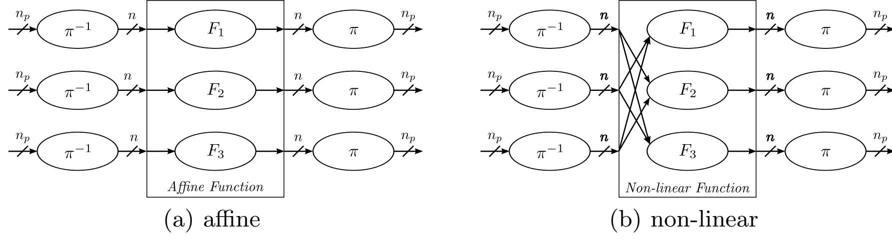
$$F_1 = a_2b_2 + a_2b_3 + a_3b_2 + r \quad (3)$$
$$F_2 = a_3b_3 + a_1b_3 + a_3b_1 + a_1r + b_1r \quad (4)$$
$$F_3 = a_1b_1 + a_1b_2 + a_2b_1 + a_1r + b_1r + r, \quad (5)$$

where r is randomly drawn from a uniform distribution. Analogously, the predictor can be shared as

$$F_{p1} = a_2 + a_2c_2 + a_2c_3 + a_3c_2 + r \quad (6)$$
$$F_{p2} = a_3 + a_3c_3 + a_1c_3 + a_3c_1 + c_1r \quad (7)$$
$$F_{p3} = a_1 + a_1c_1 + a_1c_2 + a_2c_1 + c_1r + r. \quad (8)$$

If both (F_1, F_2, F_3) and (F_{p1}, F_{p2}, F_{p3}) share the same r, the resulting six output bits would not be jointly uniform. Meaning that, they cannot be used as input to another joint function (i.e., another predictor) without violating the uniform input property of TI. To fix this, double amount of fresh randomness (i.e., one r bit for each part) is required.

Shared Predictors. Contrary to ordinary CED schemes, our predictors need to comply with the requirements of TI. In other words, the prediction functions



(a) affine (b) non-linear

Fig. 3. Derivation of shared predictors for three shares.

work on masked check bits and fulfill the non-completeness, correctness, and uniformity properties. Finding functions with all these characteristics can be difficult for certain codes. However, in our presented scenario (i.e., a systematic linear code with a *sufficiently large* rank) it can be significantly simplified.

The general approach is shown in Figure 3 with the example of affine and non-linear functions with three shares. $\pi : \mathbb{F}_2^{n_m} \rightarrow \mathbb{F}_2^{n_p}$ denotes the generation of the check bits using P, the right part the generator matrix. Respectively, $\pi^{-1} : \mathbb{F}_2^{n_p} \rightarrow \mathbb{F}_2^{n_m}$ is defined as its inverse, i.e., recovery of the message from the check bits. To derive the shared representation of the predictor from the target functions, each input share is first transformed using π^{-1}. Then the target function is applied and each resulting share is run through π to generate the corresponding check bits again. Of course, the steps do not have to be performed segregated. Instead, they are merged and subsequently optimized to achieve a better performance. The resulting functions trivially comply with the correctness property. Given that π and π^{-1} operate on single shares, non-completeness is also maintained. Regarding the uniformity of the output shares, we need to differentiate between two cases. For $n_p = n_m$, the uniformity property is preserved from the target functions as noted in [41]. The encoding and decoding operations are only affine transformations which do not influence the uniformity in this setting. However, for $n_p > n_m$ this observation does not generally hold. If the steps are performed in an isolated manner, the reduction of the input shares to size n_m will come with a reduction in entropy. In result, the enlarged output shares are no longer uniform. A trivial solution would be the inclusion of a fresh random value to restore uniformity. However, this reduces the performance of the design and is therefore undesirable. A more efficient solution is to merge the three steps (i.e., π^{-1}, F, π) and eliminate the reduction of the input shares.

Depending on the operation, this optimization can be very effective. Especially if π and π^{-1} are linear over the F they can be partially canceled out. As mentioned before, functions with a high degree are often decomposed to reduce the number of shares. Usually there are multiple possibilities for decomposition with different efficiencies. Depending on the scenario, these decompositions do not need to be the same for the predictors. In these cases, the final result is still the same but not necessarily the intermediate values. This enables more efficient

designs while leading to some limitations in the error detection, as discussed later on.

Error Detection. As noted before, the rate of error detection inside the algorithm affects the performance, area consumption and fault coverage of the design. Frequent error checking thwarts potential optimizations of the predictors what finally leads to larger circuits.

The error checking is performed similar to Figure 1. However, in our basic scenario (without reusing randomness) the intermediate values are split up into multiple shares via Boolean masking. To still detect if an error has occurred, a two-step approach denoted as CHECK-AND-COMBINE is required. In the first step CHECK, the parity check matrix is multiplied with each share of the codeword. Thus, the resulting error check vectors $\boldsymbol{v}_{int}^{j}$ are computed as

$$\boldsymbol{v}_{int}^{j} = H \cdot \left(\boldsymbol{c}_{int}^{j}\right)^{T} = \pi(\boldsymbol{m}_{int}^{j}) \oplus \boldsymbol{p}_{int}^{j},\ 1 \leq j \leq s. \quad (9)$$

If no error has occurred, these error vectors are a random sharing of the null vector. To check this, the error vectors are combined via XOR in the second step COMBINE. However, without any registers this procedure is equivalent to a function which has all shares of both parts of the circuit as input. This certainly violates the non-completeness property of TI. To this end, it is necessary to split up the second step COMBINE into multiple parts and include registers in between. In case of a first-order secure design, all but one of the shares are first combined. The result and the last share are then stored in a register and combined as

$$\boldsymbol{v}_{int} = \left(\bigoplus_{j=1}^{s-1} \boldsymbol{v}_{int}^{j}\right) \oplus \boldsymbol{v}_{int}^{s}. \quad (10)$$

If $\boldsymbol{v}_{int}$ is not the null vector, an error has been detected. The last XOR technically violates the non-completeness property as it unmasks $\boldsymbol{v}_{int}$ by merging all shares. However, $\boldsymbol{v}_{int}$ holds no information about the sensitive intermediate values of the circuit. Therefore, the SCA resistance of the design is not jeopardized by this. An exemplary error check procedure with three shares is depicted in Figure 4.

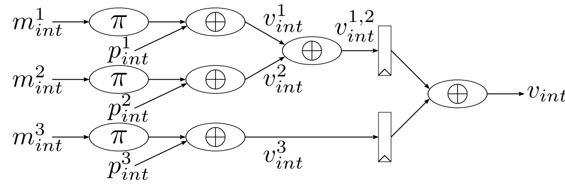


Fig. 4. Computation and unmasking of the error check vector for three shares in a first-order secure design.

It should be noted that the initial input values are indeed in compliance with

the uniformity property of TI. The input values $\boldsymbol{v}_{int}^{1,2}$ and $\boldsymbol{v}_{int}^{3}$ to the second part (right of the registers) are not jointly uniform given that if no error has occurred they are identical. Yet this does not affect the security of the resulting design since (as argued before) $\boldsymbol{v}_{int}$ does not hold any information related to sensitive intermediate values. This security guarantee still holds if the same randomness is used for masking both $\boldsymbol{m}_{int}$ and $\boldsymbol{p}_{int}$. Even though the input to the multiplication with H is not uniform, it does not pose a problem as it is applied to each share separately and the resulting $\boldsymbol{v}_{int}$ does not hold any information related to sensitive intermediate values.

The Check-And-Combine procedure can be further simplified. To this end, it is necessary that all randomness is reused and the check bits are carefully generated and predicted during the cipher run. One possibility to generate the check bits assuming $n_p = n_m$ is

$$\boldsymbol{p}_i^j = \begin{cases} \pi\left(m_i\right) \oplus r^j, & \text{for } 1 \leq j < s \\ \pi\left(m_i\right) \oplus \left(\bigoplus_{j=1}^{s-1} r^j\right), & \text{for } j = s \end{cases} \tag{11}$$

where r^j denotes uniformly distributed fresh random masks with $r^j \in_R \mathbb{F}_2^k,\ 1 \leq j < s$. Hence, the same masks are used for $\boldsymbol{m}_i$ and $\boldsymbol{p}_i$. However, each share of the codeword $\boldsymbol{c}_i^j = [\boldsymbol{m}_i^j | \boldsymbol{p}_i^j]$ is for itself not a valid codeword. The Combine-step is still necessary for error detection. To avoid this, the generation of the check bits need to be adjusted to

$$\boldsymbol{p}_i^j = \pi(\boldsymbol{m}_i^j),\ 1 \leq j \leq s. \tag{12}$$

Now each share of the codeword is valid and can be checked separately. In other words, each $\boldsymbol{p}_i^j$ can now be used to check its related $\boldsymbol{m}_i^j$ which makes the Combine-step unnecessary. Instead, if no error has occurred every $\boldsymbol{v}_i^j$ will be the null vector. To maintain this property, it is necessary that the predictors match exactly the main functions with additional encoding. Therefore, the aforementioned optimization technique regarding the decompositions of functions cannot be applied. Otherwise the $\boldsymbol{p}_i^j$ would lose this characteristic and an additional Combine-step becomes necessary. It should be noted that this only works given that $n_p = n_m$. Otherwise additional fresh randomness is required to achieve a uniform sharing of $\boldsymbol{p}_i$ making it impossible to check each share separately.

Overhead. The overhead of our scheme obviously depends on the chosen code and the underlying algorithm. Simple duplication, for example, is just an extreme case of our combined countermeasure in which P of the generator matrix is set to the identity matrix. However, if randomness is reused and $n_p \leq n$, the amount of fresh randomness is independent of the chosen code. In this case, our combined countermeasure uses the same amount of randomness as simple duplication. For other metrics it is not possible to give such a definite rule. Both area and performance can be worse or better than simple duplication, depending on how good the predictors can be optimized.

Combined Attacks. As mentioned in the introduction, there are combined attacks which can break AES implementations with certain combinations of countermeasures. Our proposed countermeasure can be also vulnerable to these kind of attacks depending on the underlying cipher and chosen code. However, most of these attacks focus on the error check and exploit that usually a combination of multiple shares is required. As described before our scheme can be instantiated without the necessity of a COMBINE-step which helps to prevent these attacks that rely on this as a point of attack. In this case, the leakage only contains information if an error has occurred but not more.

3.5 Security Analysis

We now discuss about the security properties of our combined countermeasure under the previously defined attacker model. Here, we distinguish between resistance against SCA attacks and FI attacks. In the latter case, our combined countermeasure is generally compared with a simple duplication of the TI.

SCA Resistance. As mentioned before, the security of a TI is derived from its order. A first-order TI is provable secure against first-order attacks [32]. Given that the adversary in our model can perform attacks up to order d, a TI of order d is accordingly required to protect our design. By following our proposed approach, the shared predictors are in compliance with the principle of a d-order TI. Therefore, they provide the same level of security as the d-order TI of the main circuit. Therefore, our proposed combined countermeasure has the exact level of SCA-security as a plain d-order TI without FI countermeasures. Furthermore, this level is independent of the chosen code meaning that simple duplication does not provide better or worse SCA-protection than a more complex EDC.

FI Resistance. The level of security against FI attacks depends on the parameters of the chosen code. In particular, the code distance d is important for the detection of certain types of errors. In this context, we can model the simple duplication countermeasure as a linear $[2k, k, 2]$-code $\mathbf{D}$ with $d = 2$. This is a comparably low distance given that such a distance can be achieved by a (in most cases) much shorter parity $[k + 1, k, 2]$-code.

The efficiency of a fault countermeasure can be assessed by its fault coverage rate which measures the proportion of undetectable faults. To simplify the analysis, we first assume that the fault is injected into an intermediate state of the execution which is used for error detection. That is, $\boldsymbol{p}_{int}$ are valid check bits for $\boldsymbol{m}_{int}$. As defined before, a fault is modeled as an error vector $e \neq \mathbf{0}$ that is added to the state $\boldsymbol{c}_{int} = [\boldsymbol{m}_{int}|\boldsymbol{p}_{int}]$. For a fault to be undetectable, e needs to be a valid codeword of the deployed code $\mathbf{C}$. This is rooted in the characteristic of linear codes in which every valid codeword can be written as the sum of two valid codewords as

$$c_3 = c_1 + c_2 = m_1 \cdot G + m_2 \cdot G = (m_1 + m_2) \cdot G,$$

with $c_1, c_2, c_3 \in \mathbf{C}$. Therefore, if e is not a valid codeword of $\mathbf{C}$ the erroneous result would also not be a valid codeword. Note that the aforementioned addition property of linear codes still holds for shared codewords. Meaning that if a valid codeword is added to one of the shares, it would result in a new shared codeword. With this, we can formally define the fault coverage of a code $\mathbf{C}$ as

$$Coverage_{\mathbf{C}}[E \sim \mathcal{E}] = 1 - Pr[e \in \mathbf{C} \wedge e \neq \mathbf{0}], \tag{13}$$

where the error variable E follows an error distribution $\mathcal{E}$.

Usually the rank of the code k is not chosen to be equal to the size of the whole input of the algorithm for efficiency reasons. Therefore, the intermediate state of the execution consists of multiple valid codewords. To further simplify the analysis we first assume that the adversary only injects one fault in one share of one codeword of the intermediate state. With $|C| = 2^k$ and $|E| = 2^n$ we can derive $Pr[e \in \mathbf{C} \wedge e \neq \mathbf{0}] = (2^k - 1)/2^n$ and define the fault coverage of the code $\mathbf{C}$ as

$$Coverage_{\mathbf{C}}[E \sim \mathcal{E}_U] = 1 - \frac{2^k - 1}{2^n},$$

in the uniform fault model. Notably, the fault coverage in this model is independent of the code distance d. It means that it depends only on the rank k and the length n. Consequently, simple duplication provides the same fault coverage as any other code with the same k and n against this type of faults. For $\mathbf{D}$ the length is derived from the rank as $n = 2k$. The coverage can then be simplified to

$$Coverage_{\mathbf{D}}[E \sim \mathcal{E}_U] = 1 - \frac{1}{2^k} + \frac{1}{2^{2k}}.$$

As noted before, the uniform fault model is not a realistic assumption for all scenarios. Therefore, it is closer to reality to assume that the error distribution is biased to a certain degree [20]. For example, a clock glitch might cause similar errors in identical circuits which are close together (i.e., simple duplication). In this scenario, the fault coverage is severely reduced given that simple duplication cannot detect identical errors in both circuits. In the following, we assume that only a limited number of bits is affected by the fault. In the most extreme case, only one bit is affected which is related to laser fault injection[2]. We consider a biased distribution $\mathcal{E}_{B_b}$ with the corresponding subsets

$$E_1 = \{e \mid e \in E \wedge wt(e) \leq b\} \text{ with } Pr[e \in E_1] = 1, \tag{14}$$

$$E_2 = \{e \mid e \in E \wedge wt(e) > b\} \text{ with } Pr[e \in E_2] = 0. \tag{15}$$

[2] Note that bit flips which we assume in our attacker model might not be realistic for laser fault injection in certain scenarios [40]. However, we still use it in our model. The ability to set and reset bits instead of flipping enables trivial attacks in which the adversary tests each bit of the key to be zero or one. This attack cannot be directly prevented by our method without additional logic (e.g., allow only a certain number of faults). However, this is true for a majority of countermeasures and therefore not an issue unique to our methodology. The designer needs to include further countermeasures against this attack vector, e.g., splitting the key into multiple shares can increase the complexity of the attack.

We assume further that the error vectors in E_1 are all equally probable. Depending on the method of fault injection, certain values of b are easier to achieve than others. Following this definition, $\mathcal{E}_{B_n}$ is equivalent to $\mathcal{E}_U$. In this fault model, it is possible to give specific bounds in which a complete fault coverage is achieved by our proposed countermeasure. It is trivial to see that an $[n, k, d]$-code which can detect $u = d - 1$ errors still achieves a complete fault coverage in the model following $\mathcal{E}_{B_u}$. However, it depends on the specific code how the fault coverage evolves for higher values of $b > u$. For simple duplication it can be easily calculated as

$$Coverage_{\mathbf{D}}[E \sim \mathcal{E}_{B_b}] = 1 - \frac{\sum_{j=1}^{\lfloor \frac{b}{2} \rfloor} \binom{k}{j}}{\sum_{i=1}^{b} \binom{n}{i}}.$$

It is notable that a simple duplication scheme achieves full fault coverage only for $b = 1$.

Depending on the scenario, there are other possible biased distributions. For example, if the attacker is only able to inject faults in one part of the design (target algorithm or check bits) the full fault coverage is achieved for all codes with $d > 1$. Furthermore, the ability to inject symmetric errors in both parts strongly reduces the security of simple duplication. In the most extreme case, the adversary can pick bits to fault, e.g., by laser injection. In this case the error detecting capability is directly proportional to the attack complexity assuming that targeting more single bits by laser at different places increases the costs of the attack.

In reality, it might not be possible to only target one specific codeword, e.g., with round-based architectures. This affects the fault coverage since the error vector e needs to be valid codeword for every element of the state. Therefore, the estimation of the coverage can be adapted to include the number of state elements n_s

$$Coverage_{\mathbf{C}}[E \sim \mathcal{E}] = 1 - (Pr[e \in \mathbf{C} \wedge e \neq \mathbf{0}])^{n_s}\ . \tag{16}$$

We assume an error check in which each share is not checked separately. Therefore, the number of shares does not play any role in this estimation since it is enough to check whether the sum of all shares is a valid codeword as

$$(c^1 \oplus e^1) \oplus (c^2 \oplus e^2) \oplus (c^3 \oplus e^3) = c \oplus e. \tag{17}$$

If each share is checked separately, the fault coverage needs to include the number of shares in the calculation similar to n_s.

Up to now, we only considered faults that are injected at one point in time into an encoded state which is checked for errors. Depending on the power of the adversary, this scenario may be realistic. However, there are also other cases in which an attacker is more powerful and can inject more sophisticated types of errors.

One of these types are faults which are injected into a state between layers that is not directly checked. Instead, multiple operations are first performed on the erroneous state before it is checked. In this case, the fault coverage rate stays the same based on the fact that none of the operations change the validity of a codeword. In other words, if the error is detectable in one state, it should be also detectable in every following state.

Another important aspect for fault coverage is multiple faults at different points in time. Assume for example a linear transformation F of a codeword c in which an error e is injected before the transformation is applied. This results in

$$F(c \oplus e) = F(c) \oplus F(e) \tag{18}$$

meaning that the output of F is combined with a transformed error $F(e)$. Given the structure of the functions and predictors, $F(.)$ cannot make a valid codeword $F(e)$ if e is not a valid codeword. Therefore, the fault coverage is not impaired for faults at a single point in time. However, $F(.)$ can increase the Hamming weight of e making it easier for an attacker to inject an additional error after the transformation. In the most extreme case, an attacker injects only two errors e_1, e_2 with $wt(e_1) = wt(e_2) = 1$ and can create an undetectable fault as

$$F(c \oplus e_1) \oplus e_2 = F(c) \oplus F(e_1) \oplus e_2, \tag{19}$$

with $wt(F(e_1)) = d - 1$. This approach works similarly for non-linear layers. However, in this case the output of the function is not the sum of the two transformed values. This attack vector can be prevented by introducing more error checks in the design. If every encoded state before a transformation is checked, this attack can be thwarted since the injection of the first error $F(c \oplus e_1)$ would be detected. Introducing more checks can obviously result in an increased area complexity.

As of now, all of the errors are added to an encoded state between layers. However, depending on the scenario it might be also possible to inject faults inside the combinatorial logic between these states. Since the logic usually consists of a cascade of multiple gates modeling, the fault as an addition of an error vectors is not trivial. However, depending on the abilities of the attacker this type of fault can be powerful. For example, an attacker can target one gate which derives multiple output bits. In this case, we have the same scenario as in the previous example that the injected fault e has $wt(e) = 1$ but it cannot be detected when the check is performed on the output of the combinatorial circuit where such e leads to $e' \in \mathbf{C}$. To completely avoid this type of attack it is necessary to isolate the logic for all output lines from each other. This way a faulty gate can only affect one of the output bits which prevents the aforementioned attack.

As illustrated by the previous example, it is important to realistically estimate the power of potential FI attackers. Choosing a code with a large distance and implementing the previously proposed countermeasures might lead to a highly secure system. However, each of these aspects can negatively influence the size of the design. As for many other systems, the balance between area and the level of security is an important aspect in the design process.

4 Case Study: LED

Up to now, our combined countermeasure has been only discussed from the theoretic perspective without targeting a specific algorithm. To better illustrate the rationales and parameters of the design process, we implement a block cipher according to our methodology. For the sake of comprehensibility, a relatively straightforward example is picked to explain the design choices in detail.

The most obvious target for this would be AES as it is the most widely deployed cipher. However, while the predictors for the linear layers of AES are comparably easy to implement, the TI of its non-linear layer poses still a challenge even without FI resistance [5]. In particular, it requires a significant amount of fresh randomness to achieve all the necessary TI properties. Another standardized cipher, for which an efficient TI exists, is PRESENT [10]. Its 4-bit S-Box can be efficiently implemented in various ways [7, 41]. Contrary to AES, its permutation layer is very efficient in hardware, but its predictors are comparably inefficient. A brief assessment of the permutation layer of PRESENT is given in Appendix B.

A better example to demonstrate our combined countermeasure is LED. It combined the best aspects of AES and PRESENT by incorporating the PRESENT S-Box and AES-like linear layers. Thus, an efficient TI and predictors can easily be achieved. In our case study, we present one way to implement LED with our methodology. Note that depending on the targeted attacker model, different choices are possible, e.g., higher-order TI or another code with a large distance. The SCA security of the final design is practically evaluated using an FPGA prototype, while the FI resistance is examined using the previously introduced attacker models.

4.1 Cipher Description

LED is a lightweight block cipher introduced in 2011 [19]. It has a 64-bit state and can be instantiated with different key sizes (primarily 64 or 128 bits). The basic structure of the cipher as depicted in Figure 5 consists of addition of the round keys (addRoundKey) and so-called steps (step). In each step, four rounds of encryption are applied to the state. One round is made up of four layers AddConstants, SubCells, ShiftRows and MixColumnsSerial. During AddConstants constants which are derived from an LFSR are added to half of the state. The following three layers are similar to the layers of AES [34] and consist of a nibble-wise substitution and row/column-wise affine transformations. For 128-bit key (resp. 64-bit keys) LED-128 (resp. LED-64) performs 12 steps in total (resp. 8 steps) with key additions between them.

One important characteristic of LED is its very simple key schedule. Instead of using different round keys derived by a schedule function applied on a main key, the cipher directly uses 64 bits from the user-defined key for each round. This means that for the 64-bit version all round keys are the same, while in the 128-bit instantiation the key halves are used alternately.

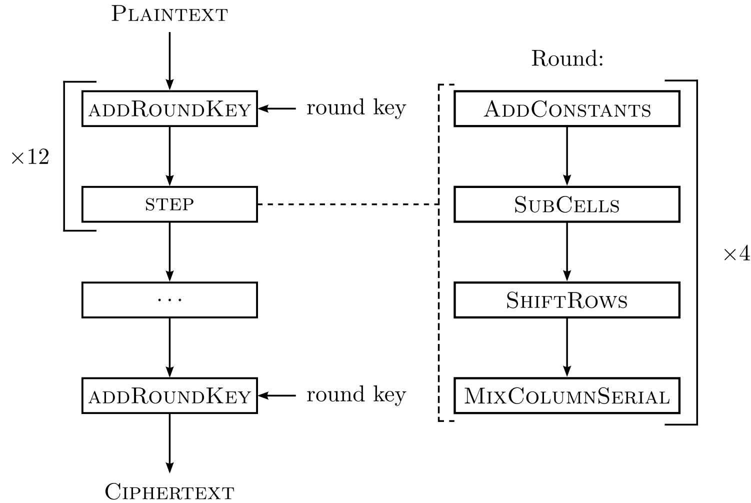


Fig. 5. The basic structure of LED-128.

4.2 Design and Implementation

We implement a design that is secure against first-order attacks. We decompose the S-Box that allows us to implement TI using three shares. In the following, we explain the selection of the code and the predictors for each layer of LED in detail.

Code Selection. Given that LED is a nibble-oriented cipher in which all operations work on either one or multiple nibbles of the state, we consider only codes with a rank of $k = 4$. This way, expensive merge or split of codewords can be minimized. Furthermore, we decided to set the length of the code to $n = 8 = 2 \cdot k$ to avoid additional fresh randomness. It would be beneficial to select a code over $GF(2^4)$, since most of the LED operations are in this field[3]. However, none of the 16 possible $[8, 4]$-codes has a distance larger than $d = 3$. Therefore, to achieve a higher level of protection against FI attacks, we choose a different code outside of $GF(2^4)$ but with a better error detection property.

The extended Hamming code is a basic extension of the $[7, 4, 3]$-Hamming code. By adding an extra parity bit the code is transformed to a $[8, 4, 4]$-code, i.e., with $d = 4$. In our implementation we use the following generator and parity check matrices:

$$G = \left(\begin{array}{cccc|cccc} 1&0&0&0&1&1&1&0 \\ 0&1&0&0&1&1&0&1 \\ 0&0&1&0&1&0&1&1 \\ 0&0&0&1&0&1&1&1 \end{array}\right), \qquad H = \left(\begin{array}{cccc|cccc} 1&1&1&0&1&0&0&0 \\ 1&1&0&1&0&1&0&0 \\ 1&0&1&1&0&0&1&0 \\ 0&1&1&1&0&0&0&1 \end{array}\right). \tag{20}$$

[3] In this case, P is chosen in such a way that $\boldsymbol{p} = \pi(\boldsymbol{m}) = \boldsymbol{m} \cdot x$ with $x \in GF(2^4)$.

Due to its simplicity, the code enables the use of efficient predictors while still achieving a high error detection capability with respect to its length.

Linear Layers. As described before, LED consists of four different linear layers. We discuss the application of the extended Hamming code to each layer without specifically considering TI, since every linear layer and corresponding predictor can be applied to each share separately as explained in Section 3.4. Note that the key and constants are not shared, following the same design strategy as in [5,29,36,41]. Therefore, in the two layers (AddRoundKey, AddConstants) where a value is added to the state, it is applied only to one share (of three).

AddRoundKey. Since this layer only consists of a basic addition in $GF(2^4)$ of the round key to the state of the cipher, its predictor can be implemented very efficiently. It can be optimized to

$$\begin{aligned} \boldsymbol{p}_{int_2} &= \pi\left(\pi^{-1}\left(\boldsymbol{p}_{int_1}\right) \oplus key\right) \\ &= \boldsymbol{p}_{int_1} \oplus \pi\left(key\right), \end{aligned} \tag{21}$$

where $\boldsymbol{p}_{int_1}$ (resp. $\boldsymbol{p}_{int_2}$) denotes the input (resp. output) check bits to AddRoundKey, and key a round key. Furthermore, LED does not include a key schedule. Thus, by computing $\pi\left(key\right)$ (of both key halves for LED-128) once at the start of the cipher, the predictor for the key addition can be easily realized without additional overhead.

AddConstants. Two types of round constants are added to the state. One is derived from the key length and does not change over the course of the cipher. The bit size of the key length is stored in eight bits ($ks_7ks_6ks_5ks_4\ ks_3ks_2ks_1ks_0$). The lower and upper four bits of the bitstring are each considered as one encoded element. Since the key size does not change during the execution, this type of constant does not need to be updated. For LED-128 the specific bits are

$$\begin{aligned} (ks_7ks_6ks_5ks_4\ ks_3ks_2ks_1ks_0) &= (1000\ 0000), \\ (ksp_7ksp_6ksp_5ksp_4\ ksp_3ksp_2ksp_1ksp_0) &= (1110\ 0000) \end{aligned} \tag{22}$$

where ksp_i denotes the corresponding check bits for this constant.

The other constant consists of six bits ($rc_5rc_4\ rc_3rc_2rc_1rc_0$) which are updated for every round by an LFSR. The update function can be represented by a matrix multiplication in $GF(2)$ as

$$\begin{pmatrix} rc'_0 \\ rc'_1 \\ rc'_2 \\ rc'_3 \\ rc'_4 \\ rc'_5 \end{pmatrix} = \underbrace{\begin{pmatrix} 0\,0\,0\,0\,1\,1 \\ 1\,0\,0\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0 \\ 0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,1\,0\,0 \\ 0\,0\,0\,0\,1\,0 \end{pmatrix}}_{U} \cdot \begin{pmatrix} rc_0 \\ rc_1 \\ rc_2 \\ rc_3 \\ rc_4 \\ rc_5 \end{pmatrix} + \underbrace{\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}}_{c}, \tag{23}$$

where U denotes the update matrix. The related check bits defined as

$$(rcp_3 rcp_2 rcp_1 rcp_0) = \pi(rc_3 rc_2 rc_1 rc_0) \quad (24)$$
$$(rcp_3 rcp_2 rcp_1 rcp_0) = \pi(00|rc_5 rc_4) \quad (25)$$

need to be updated accordingly. To this end, the update matrix is first enlarged to incorporate the two padded zeros to

$$U_L = \begin{pmatrix} 0&0&0&0&1&1&0&0 \\ 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{pmatrix}. \quad (26)$$

The update matrix for the check bits ($U_{L_{check}}$) can be derived from by $\pi\left(U_L\left(\pi^{-1}(\cdot)\right)\right)$. Therefore, we can write (note that P is self-inverse):

$$U_{L_{check}} = \begin{pmatrix} 1&1&1&0&0&0&0&0 \\ 1&1&0&1&0&0&0&0 \\ 1&0&1&1&0&0&0&0 \\ 0&1&1&1&0&0&0&0 \\ 0&0&0&0&1&1&1&0 \\ 0&0&0&0&1&1&0&1 \\ 0&0&0&0&1&0&1&1 \\ 0&0&0&0&0&1&1&1 \end{pmatrix} \cdot U_L \cdot \begin{pmatrix} 1&1&1&0&0&0&0&0 \\ 1&1&0&1&0&0&0&0 \\ 1&0&1&1&0&0&0&0 \\ 0&1&1&1&0&0&0&0 \\ 0&0&0&0&1&1&1&0 \\ 0&0&0&0&1&1&0&1 \\ 0&0&0&0&1&0&1&1 \\ 0&0&0&0&0&1&1&1 \end{pmatrix}. \quad (27)$$

The same procedure is applied to the constant factor of the update function (denoted as c in Eq. (23)). Overall, the check bits of the round constant can be updated as

$$\begin{pmatrix} rcp'_0 \\ rcp'_1 \\ rcp'_2 \\ rcp'_3 \\ rcp'_4 \\ rcp'_5 \\ rcp'_6 \\ rcp'_7 \end{pmatrix} = \underbrace{\begin{pmatrix} 0&0&1&1&0&0&1&1 \\ 0&1&0&1&0&0&1&1 \\ 0&1&1&0&0&0&1&1 \\ 1&0&0&0&0&0&0&0 \\ 0&1&1&1&1&1&1&0 \\ 0&1&1&1&1&1&1&0 \\ 0&1&1&1&0&0&0&1 \\ 0&0&0&0&1&1&1&0 \end{pmatrix}}_{U_{L_{check}}} \cdot \begin{pmatrix} rcp_0 \\ rcp_1 \\ rcp_2 \\ rcp_3 \\ rcp_4 \\ rcp_5 \\ rcp_5 \\ rcp_5 \end{pmatrix} + \underbrace{\begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}}_{cp}. \quad (28)$$

It is obvious that the update of the check bits requires additional resources. Still this overhead is negligible since the round constant update is only a small part of the cipher and not split up into multiple shares.

ShiftRows. This layer manipulates the state in a nibble-wise fashion. Since the codewords are not modified in any way, it is sufficient to apply the same permutation on the check bits.

MixColumnSerial. Four nibbles of the state are combined using a matrix A four consecutive times. The matrix multiplication is performed in $GF(2^4)$. Since addition is linear over $GF(2)$, we do not need to change the values of A for the check bits. Only the field multiplications with 2 and 4 need to be adapted to the predictor. The two multiplications with the reduction polynomial $X^4 + X + 1$ can be represented as a matrix multiplications in GF(2) as

$$2 \cdot \begin{pmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \end{pmatrix} = \begin{pmatrix} 0\,0\,0\,1 \\ 1\,0\,0\,1 \\ 0\,1\,0\,0 \\ 0\,0\,1\,0 \end{pmatrix} \cdot \begin{pmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \end{pmatrix}, \quad 4 \cdot \begin{pmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \end{pmatrix} = \begin{pmatrix} 0\,0\,1\,0 \\ 0\,0\,1\,1 \\ 1\,0\,0\,1 \\ 0\,1\,0\,0 \end{pmatrix} \cdot \begin{pmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \end{pmatrix}. \tag{29}$$

For the check bits, these matrices need to be adapted similar to Equation (27) but with 4×4 matrices. The resulting matrices for the check bits are

$$\pi\left(2 \cdot \pi^{-1}\begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix}\right) = \begin{pmatrix} 0\,0\,1\,1 \\ 0\,1\,0\,1 \\ 0\,0\,0\,1 \\ 1\,1\,1\,1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix}, \quad \pi\left(4 \cdot \pi^{-1}\begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix}\right) = \begin{pmatrix} 1\,1\,1\,0 \\ 1\,0\,1\,0 \\ 1\,1\,1\,1 \\ 1\,0\,0\,0 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix}. \tag{30}$$

This layer is also slightly more costly for the check bits. However, the overhead is not as significant as for PRESENT (cf. Appendix B).

Non-Linear Layer. Similar to [36], we decomposed the S-Box into two steps to reduce the number of required shares to three. The functions for the state and check bits are optimized independently of each other. As mentioned before, this procedure results in a more efficient implementation in terms of area with the penalty of not being able to check the correctness of each share individually. To find an area-efficient representation, we applied the same idea as in [7]. In particular, different affine transformations with different combinations of quadratic bijective classes (as defined in [8]) are tested and compared by their number of XOR and AND operations [36]. For the non-encoded TI we tested combinations of the form

$$S = A_3 \circ T_2 \circ A_2 \circ T_1 \circ A_1, \tag{31}$$

where A_1, A_2, A_3 are affine transformations and T_1, T_2 are quadratic bijections. We tested all possible valid combinations of Table 1 from [41] and decomposed the S-Box as $S(m) = F\left(G\left(x\right)\right), \forall m$ with

$$F = A_3 \circ T_2, \qquad G = A_2 \circ T_1 \circ A_1.$$

For the check bits Equation (31) is slightly adjusted to

$$S_p = \pi \circ S \circ \pi^{-1} = \pi \circ A_3 \circ T_2 \circ \pi^{-1} \circ \pi \circ A_2 \circ T_1 \circ A_1 \circ \pi^{-1}, \tag{32}$$

and the S-Box for the check bits is split as $S_p(p) = Q(R(p)), \forall p$ with

$$Q = \pi \circ A_3 \circ T_2 \circ \pi^{-1}, \qquad R = \pi \circ A_2 \circ T_1 \circ A_1 \circ \pi^{-1}.$$

We found the most efficient decomposition for the classical TI using the quadratic class $\mathcal{Q}_{12}$ for both T_1, T_2 (see [7]). For the check bits the most efficient decomposition was obtained by the quadratic classes $\mathcal{Q}_{294}$ and $\mathcal{Q}_{299}$ for T_1 and T_2, respectively.

As a side note, since $R \neq \pi \circ G \circ \pi^{-1}$ (and likewise for Q and F), the error-checking procedure cannot be performed in-between the S-Box computation. Below we list the algebraic normal form (ANF) of the derived (and applied) functions (a and e as least significant bits).

$$G(d,c,b,a) = (h,g,f,e): \quad \begin{array}{ll} e = a + c + d + cb & f = a \\ g = 1 + a + d + b + cb & h = 1 + a + bc + bd + cd \end{array} \tag{33}$$

$$F(d,c,b,a) = (h,g,f,e): \quad \begin{array}{ll} e = a & f = c + d + bd \\ g = 1 + a + b + c + cd & h = c + bd \end{array} \tag{34}$$

$$R(d,c,b,a) = (h,g,f,e): \quad \begin{array}{ll} e = a + b + db + dc & f = b + c \\ g = c + ba + ca & h = d + b + cb \end{array} \tag{35}$$

$$Q(d,c,b,a) = (h,g,f,e): \quad \begin{array}{ll} e = 1 + a + b + c + db + dc & f = 1 + a + b \\ g = a + d + db + dc & h = c + ab + ac + ad + bc + bd \end{array} \tag{36}$$

The uniform shared representations of the component functions (G_1, G_2, G_3), (F_1, F_2, F_3), (R_1, R_2, R_3), (Q_1, Q_2, Q_3) can be derived by direct sharing [8]. All required formulas are given in Appendix A.

Basic Structure. We implemented the LED encryption with our countermeasure following a round-based architecture. The basic structure of our design is depicted in Figure 6 with the predictors in the left half. As stated above, the S-Box and its corresponding function on check bits do not follow the same decomposition. Therefore, we perform the error check only at the first registered state $State_1^{i\in\{1,2,3\}}$. The Error Check module has been implemented following the concept of Check-And-Combine, illustrated in Section 3.4. Both AddRoundKey and AddConstants are only applied to the first share since the key and the constants are not shared. An additional register stage is necessary inside SubCells (between G and F as well as between R and Q) to avoid the propagation of glitches. The initial randomness is shared between both parts of the circuit and none of the layers requires additional fresh randomness to achieve uniformity. It should be noted that except for the initial loading (right half with

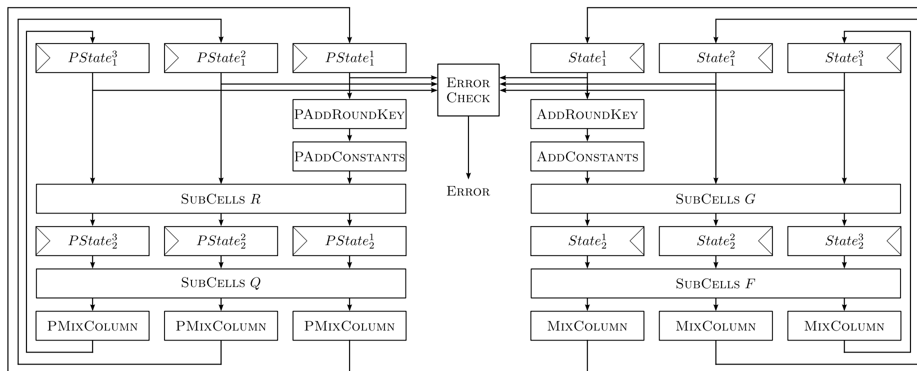


Fig. 6. The basic structure of our proposed LED design. Multiplexers for the plaintext and AddRoundKey are omitted.

shared plaintext, and left half with shared corresponding check bits) the two halves of the design do not interact with each other, and each one operates independently. At every clock cycle, the Error Check module examines the consistency of the state and its corresponding check bits.

The proposed design can be easily extended to provide security against higher-order attacks by increasing the number of shares. As the linear functions are applied on each share separately, their basic structure does not change, while non-linear functions require further adjustment. A second-order TI of the PRESENT S-Box is given in [31]. However, mask refreshing might be necessary to ensure resistance against multivariate higher-order attacks as indicated in [38]. Note, however, that for higher-order TI the error check needs to be also adjusted accordingly to comply with the TI properties. In other words, extra registers should be integrated into Combine step of Check-And-Combine module (see Figure 4) and the Combine should be performed in several clock cycles to ensure that the desired higher-order resistance is not violated.

4.3 Area Comparison

We synthesized our implementations with the Synopsys Design Compiler using the UMCL18G212T3 [47] ASIC standard cell library (UMC 0.18μm). The results are presented in Table 1.

As expected, the state registers constitute a significant portion of each circuit part (in the following referred to as *Original* and *Predictors*). Furthermore, the decomposed S-Box is in both cases the largest layer of the design. Since we make use of Check-And-Combine, the error detection circuitry is relatively large due to the required additional registers of the Combine step. Overall, the predictors require around 27% more area than the original TI. With the same error detection module, our design with the extended Hamming code is around 12% bigger than simple duplication.

Table 1. Size of our design for an ASIC platform.

Module	**Area [GE]**			
	Original	Predictors	Error Detection	Control
AddRoundKey	171	171	-	-
AddConstants	32	32	-	-
SubCells 1	1750	1584	-	-
SubCells 2	1051	2795	-	-
ShiftRows	0	0	-	-
MixColumnSerial	1532	2048	-	-
Total	7891	10028	2023	270
LED-ParTI	**20212**			

The synthesized circuit can operate at the maximum frequency of 148 MHz and requires 96 clock cycles for one encryption. The design forms a pipeline, where two plaintexts can be consecutively fed. This results in a maximum throughput of 197.3 Mbit/s. In comparison, the unprotected round-based implementation requires 46 clock cycles for one encryption and can operate at a maximum frequency of 131 MHz. This results in a throughput of 174.7 Mbit/s since the design does not allow a pipeline.

4.4 Resistance against SCA

Given that all functions are compliant to the principles of TI, our design is provably secure against first-order attacks. Nevertheless, we also evaluated the security of our design experimentally using an FPGA and ported our design to the FPGA-based side-channel evaluation platform SAKURA-G [1] populated with a Xilinx Spartan-6 FPGA. The power traces obtained for our the design have been collected by means of a digital oscilloscope at sampling rate of 500 MS/s while the design was operating at a frequency of 3 MHz.

As an evaluation metric we used the non-specific t-test as proposed in [14,18] which has become a popular generic evaluation method in recent years [42]. In such a test, the leakages related to two sets of measurements are compared, one with a fixed input (plaintext) and the other one with randomly selected input. During the measurements, for both sets (which are also randomly interleaved) the 128-bit masks (used for initial sharing of the plaintexts as well as the check bits) are randomly selected with a uniform distribution.

While the test can examine the existence of detectable leakage at certain orders, we omit the details here. For further information, the interested reader is referred to the original articles [14, 18, 42]. Figure 7 depicts the results for univariate tests at first, second and third orders using 100 million measurements. The diagram show that our design is indeed first-order secure while – as expected – leakages for higher orders can be observed.

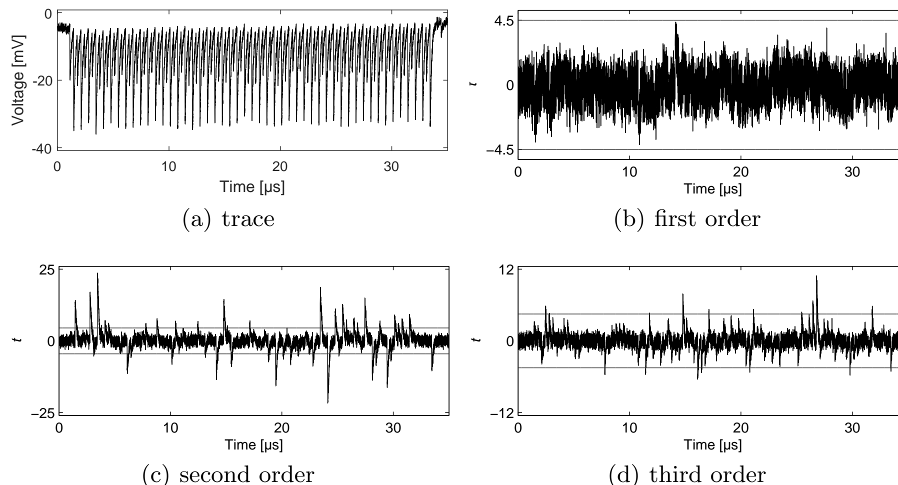


Fig. 7. A sample trace and the result of non-specific t-tests at orders one to three.

4.5 Resistance against FI

We further examined the fault coverage of our scheme considering the previously introduced attacker model. Given that the extended Hamming $[8, 4, 4]$-code has a distance of $d = 4$, it can detect errors up to $wt(e) \leq u = 3$. The coverage of this code is compared to the coverage that can be achieved with a simple $[8, 4, 2]$-duplication code with $u = 1$. We compute the fault coverage of both codes for the uniform distribution as well as for biased distributions $\mathcal{E}_{B_1}$ to $\mathcal{E}_{B_8}$. We consider both the best case (BC) and worst case (WC) for an attacker. In the best case, the attacker is able to inject a fault into one share of a single codeword. In the worst case, he can inject faults into all shares of all codewords simultaneously. Given that the TI of LED operates on a 16-element state, the fault coverage is significantly increased in this case. Since we do not check each share separately, the number of shares does not influence the fault coverage rate of the worst case.

Table 2 represents the fault coverage rates for the examined cases for both codes. We already discussed how to compute the fault coverage for a duplication code in Section 3.5. In order to derive the corresponding fault coverage for the $[8, 4, 4]$-code, we look at the distribution of the Hamming weight of the codewords. Since $k = 4$, there exist 16 different codewords. 14 of them have Hamming weight $wt(c) = 4$, while there are two codewords with $wt(c) = 0$ and $wt(c) = 8$ respectively. Therefore, only some error vectors with Hamming weight of 4 or 8 (excluding the zero error vector) have the possibility to be undetectable by our scheme[4].

[4] Since $d = 4$, errors which flip 4 or 8 bits can turn a valid codeword into another valid codeword, and are hence undetectable.

Table 2. Fault coverage for different distributions and codes.

		$\mathcal{E}_U$	$\mathcal{E}_{B_1}$	$\mathcal{E}_{B_2}$	$\mathcal{E}_{B_3}$	$\mathcal{E}_{B_4}$	$\mathcal{E}_{B_5}$	$\mathcal{E}_{B_6}$	$\mathcal{E}_{B_7}$
$[\mathbf{8},\mathbf{4},\mathbf{4}]$	BC	0.94	1.00	1.00	1.00	0.91	0.93	0.94	0.94
	WC	$1-2^{-65}$	1.00	1.00	1.00	$1-2^{-56}$	$1-2^{-63}$	$1-2^{-66}$	$1-2^{-66}$
$[\mathbf{8},\mathbf{4},\mathbf{2}]$	BC	0.94	1.00	0.89	0.95	0.93	0.95	0.94	0.94
	WC	$1-2^{-65}$	1.00	$1-2^{-51}$	$1-2^{-72}$	$1-2^{-64}$	$1-2^{-71}$	$1-2^{-66}$	$1-2^{-66}$

This observation is confirmed by the results in Table 2. The $[8,4,4]$-code provides full fault coverage in the biased model up to $\mathcal{E}_{B_3}$. Given that most of the valid codewords have a Hamming weight of 4, and $d=4$, the biased distribution $\mathcal{E}_{B_4}$ leads to the lowest fault coverage. In short, $\mathcal{E}_{B_4}$ and $\mathcal{E}_{B_5}$ are the only distributions with which the simple duplication scheme is better than the extended Hamming code. For all other cases, the extended Hamming code outperforms (or is equal to) the simple duplication scheme. As expected, the worst case leads to very high fault coverage given that the probability to inject an error which results in a valid codeword for every element of the state is very low.

5 Conclusions

We presented an advanced hardware countermeasure which offers resistance both against SCA and FI attacks. In short, we proposed a construction to combine error detecting codes with the concept of threshold implementations. We have identified and discussed generic strategies to that additions for information redundancy do not contradict to the assumptions and requirements of the underlying masking scheme.

From an general point of view, our combined countermeasure can be applied to arbitrary ciphers and supports different level of protections, i.e., first- or higher-order SCA resistance as well as various fault coverage settings. As an example, we have illustrated how to apply our methodology on the LED block cipher with the aim of maintaining first-order SCA protection while integrating an extended Hamming code to detect faults. Supported by our experimental validation, we have demonstrated how to realize an efficient design that satisfies the requirement to provide protection against SCA and FI.

Acknowledgment

The authors want to thank Falk Schellenberg for his helpful discussions and comments. The research in this work was supported in part by the DFG Research Training Group GRK 1817/1.

References

1. Side-channel AttacK User Reference Architecture. `http://satoh.cs.uec.ac.jp/SAKURA/index.html`.
2. G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri. Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard. *IEEE Trans. Computers*, 52(4):492–505, 2003.
3. E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *CRYPTO*, volume 1294 of *LNCS*, pages 513–525. Springer, 1997.
4. B. Bilgin, J. Daemen, V. Nikov, S. Nikova, V. Rijmen, and G. V. Assche. Efficient and First-Order DPA Resistant Implementations of Keccak. In *CARDIS 2013*, volume 8419 of *LNCS*, pages 187–199. Springer, 2013.
5. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. A More Efficient AES Threshold Implementation. In *Progress in Cryptology - AFRICACRYPT 2014*, volume 8469 of *LNCS*, pages 267–284. Springer, 2014.
6. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. Higher-Order Threshold Implementations. In *Advances in Cryptology - ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 326–343. Springer, 2014.
7. B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, and G. Stütz. Threshold Implementations of All 3 ×3 and 4 ×4 S-Boxes. In *CHES 2012*, volume 7428 of *LNCS*, pages 76–91. Springer, 2012.
8. B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, N. Tokareva, and V. Vitkup. Threshold implementations of small S-boxes. *Cryptography and Communications*, 7(1):3–33, 2015.
9. R. E. Blahut. *Algebraic codes for data transmission.* Cambridge Univ. Press, Cambridge, 2003.
10. A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *CHES*, volume 4727 of *LNCS*, pages 450–466. Springer, 2007.
11. J. Bringer, C. Carlet, H. Chabanne, S. Guilley, and H. Maghrebi. Orthogonal Direct Sum Masking - A Smartcard Friendly Computation Paradigm in a Code, with Builtin Protection against Side-Channel and Fault Attacks. In *WISTP*, volume 8501 of *LNCS*, pages 40–56. Springer, 2014.
12. J. Bringer, H. Chabanne, and T. Le. Protecting AES against side-channel analysis using wire-tap codes. *J. Cryptographic Engineering*, 2(2):129–141, 2012.
13. C. Clavier, B. Feix, G. Gagnerot, and M. Roussellet. Passive and active combined attacks on aes???combining fault attacks and side channel analysis. In *FDTC*, pages 10–19. IEEE Computer Society, 2010.
14. J. Cooper, E. Demulder, G. Goodwill, J. Jaffe, G. Kenworthy, and P. Rohatgi. Test Vector Leakage Assessment (TVLA) Methodology in Practice. International Cryptographic Module Conference, 2013.
15. F. Dassance and A. Venelli. Combined fault and side-channel attacks on the AES key schedule. In *FDTC*, pages 63–71. IEEE Computer Society, 2012.
16. T. De Cnudde, B. Bilgin, O. Reparaz, V. Nikov, and S. Nikova. Higher-Order Threshold Implementation of the AES S-box. In *CARDIS 2015*, 2015.
17. B. Gierlichs, J. Schmidt, and M. Tunstall. Infective Computation and Dummy Rounds: Fault Protection for Block Ciphers without Check-before-Output. In *LATINCRYPT*, volume 7533 of *LNCS*, pages 305–321. Springer, 2012.
18. G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi. A testing methodology for side channel resistance validation. In *NIST non-invasive attack testing workshop*, 2011.

19. J. Guo, T. Peyrin, A. Poschmann, and M. J. B. Robshaw. The LED Block Cipher. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011*, volume 6917 of *LNCS*, pages 326–341. Springer, 2011.
20. X. Guo, D. Mukhopadhyay, C. Jin, and R. Karri. Security analysis of concurrent error detection against differential fault analysis. *J. Cryptographic Engineering*, 5(3):153–169, 2015.
21. M. G. Karpovsky, K. J. Kulikowski, and A. Taubin. Differential fault analysis attack resistant architectures for the advanced encryption standard. In *CARDIS*, volume 153 of *IFIP*, pages 177–192. Kluwer/Springer, 2004.
22. M. G. Karpovsky, K. J. Kulikowski, and A. Taubin. Robust Protection against Fault-Injection Attacks on Smart Cards Implementing the Advanced Encryption Standard. In *DSN*, pages 93–101. IEEE Computer Society, 2004.
23. R. Karri, G. Kuznetsov, and M. Gössel. Parity-Based Concurrent Error Detection of Substitution-Permutation Network Block Ciphers. In *CHES 2003*, volume 2779 of *LNCS*, pages 113–124. Springer, 2003.
24. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Advances in Cryptology - CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
25. F. J. MacWilliams and N. J. A. N. J. A. Sloane. *The theory of error correcting codes*. North-Holland mathematical library. North-Holland Pub. Co. New York, Amsterdam, New York, 1977. Includes index.
26. S. Mangard, T. Popp, and B. M. Gammel. Side-Channel Leakage of Masked CMOS Gates. In *CT-RSA 2005*, volume 3376 of *LNCS*, pages 351–365. Springer, 2005.
27. S. Mangard, N. Pramstaller, and E. Oswald. Successfully Attacking Masked AES Hardware Implementations. In *CHES 2005*, volume 3659 of *LNCS*, pages 157–171. Springer, 2005.
28. A. Moradi. Wire-Tap Codes as Side-Channel Countermeasure - An FPGA-Based Experiment -. In *INDOCRYPT*, volume 8885 of *LNCS*, pages 341–359. Springer, 2014.
29. A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In *Advances in Cryptology - EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 69–88. Springer, 2011.
30. A. Moradi and T. Schneider. Side-Channel Analysis Protection and Low-Latency in Action - case study of PRINCE and Midori. Cryptology ePrint Archive, Report 2016/481, 2016. `http://eprint.iacr.org/`.
31. A. Moradi and A. Wild. Assessment of Hiding the Higher-Order Leakages in Hardware - What Are the Achievements Versus Overheads? In *CHES 2015*, volume 9293 of *LNCS*, pages 453–474. Springer, 2015.
32. S. Nikova, C. Rechberger, and V. Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In *ICICS 2006*, volume 4307 of *LNCS*, pages 529–545. Springer, 2006.
33. S. Nikova, V. Rijmen, and M. Schläffer. Secure Hardware Implementation of Non-linear Functions in the Presence of Glitches. *J. Cryptology*, 24(2):292–321, 2011.
34. NIST. FIPS PUB 197: Advanced Encryption Standard. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`, as of June 22, 2016.
35. S. Patranabis, A. Chakraborty, P. H. Nguyen, and D. Mukhopadhyay. A Biased Fault Attack on the Time Redundancy Countermeasure for AES. In *COSADE*, volume 9064 of *LNCS*, pages 189–203. Springer, 2015.
36. A. Poschmann, A. Moradi, K. Khoo, C. Lim, H. Wang, and S. Ling. Side-Channel Resistant Crypto for Less than 2,300 GE. *J. Cryptology*, 24(2):322–345, 2011.

37. E. Prouff and T. Roche. Higher-Order Glitches Free Implementation of the AES Using Secure Multi-party Computation Protocols. In *CHES*, volume 6917 of *LNCS*, pages 63–78. Springer, 2011.
38. O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs, and I. Verbauwhede. Consolidating Masking Schemes. In *CRYPTO 2015*, volume 9215 of *LNCS*, pages 764–783. Springer, 2015.
39. T. Roche, V. Lomné, and K. Khalfallah. Combined fault and side-channel attack on protected implementations of AES. In *CARDIS*, volume 7079 of *Lecture Notes in Computer Science*, pages 65–83. Springer, 2011.
40. C. Roscian, A. Sarafianos, J. Dutertre, and A. Tria. Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells. In *FDTC*, pages 89–98. IEEE Computer Society, 2013.
41. P. Sasdrich, A. Moradi, and T. Güneysu. Affine Equivalence and its Application to Tightening Threshold Implementations. In *SAC 2015*, volume ?, page ? Springer, 2015. `http://eprint.iacr.org/2015/749`.
42. T. Schneider and A. Moradi. Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations. In *CHES*, volume 9293 of *LNCS*, pages 495–513. Springer, 2015.
43. T. Schneider, A. Moradi, and T. Güneysu. Arithmetic Addition over Boolean Masking - Towards First- and Second-Order Resistance in Hardware. In *ACNS 2015*, volume 9092 of *LNCS*, pages 559–578. Springer, 2015.
44. A. Shahverdi, M. Taha, and T. Eisenbarth. Silent Simon: A threshold implementation under 100 slices. In *HOST 2015*, pages 1–6. IEEE, 2015.
45. B. Sunar, G. Gaubatz, and E. Savas. Sequential circuit design for embedded cryptographic applications resilient to adversarial faults. *IEEE Trans. Computers*, 57(1):126–138, 2008.
46. K. Tiri and I. Verbauwhede. A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In *DATE*, pages 246–251. IEEE Computer Society, 2004.
47. Virtual Silicon Inc. 0.18 μm VIP Standard Cell Library Tape Out Ready, Part Number: UMCL18G212T3, Process: UMC Logic 0.18 μm Generic II Technology: 0.18μm, July 2004.
48. D. M. Xiaofei Guo and R. Karri. Provably Secure Concurrent Error Detection Against Differential Fault Analysis. Cryptology ePrint Archive, Report 2012/552, 2012. `http://eprint.iacr.org/`.

A Shared Functions S-Box

In this section we give the shared representation of the functions used for the S-Box with three shares.

$$\begin{aligned} & G_1(d_2, c_2, b_2, a_2, d_3, c_3, b_3, a_3) = (h_1, g_1, f_1, e_1) \\ e_1 = {} & a_2 + b_2 + c_2 + d_2 + b_3 + c_2b_3 + b_2c_3 + b_3c_3 \\ f_1 = {} & a_2 \\ g_1 = {} & 1 + a_2 + d_2 + b_3 + c_2b_3 + b_2c_3 + b_3c_3 \\ h_1 = {} & 1 + a_2 + b_2 + b_3 + c_2b_3 + d_2b_3 + b_2c_3 + d_2c_3 + b_3c_3 \\ & + b_2d_3 + c_2d_3 + b_3d_3 + c_3d_3 \end{aligned} \tag{37}$$

$$\begin{aligned} & G_2(d_3, c_3, b_3, a_3, d_1, c_1, b_1, a_1) = (h_2, g_2, f_2, e_2) \\ e_2 = {} & a_3 + b_3 + c_3 + d_3 + b_1 + c_3b_1 + b_3c_1 + b_1c_1 \\ f_2 = {} & a_3 \\ g_2 = {} & 1 + a_3 + d_3 + b_1 + c_3b_1 + b_3c_1 + b_1c_1 \\ h_2 = {} & 1 + a_3 + b_3 + b_1 + c_3b_1 + d_3b_1 + b_3c_1 + d_3c_1 + b_1c_1 \\ & + b_3d_1 + c_3d_1 + b_1d_1 + c_1d_1 \end{aligned} \tag{38}$$

$$\begin{aligned} & G_3(d_1, c_1, b_1, a_1, d_2, c_2, b_2, a_2) = (h_3, g_3, f_3, e_3) \\ e_3 = {} & a_1 + b_1 + c_1 + d_1 + b_2 + c_1b_2 + b_1c_2 + b_2c_2 \\ f_3 = {} & a_1 \\ g_3 = {} & 1 + a_1 + d_1 + b_2 + c_1b_2 + b_1c_2 + b_2c_2 \\ h_3 = {} & 1 + a_1 + b_1 + b_2 + c_1b_2 + d_1b_2 + b_1c_2 + d_1c_2 + b_2c_2 \\ & + b_1d_2 + c_1d_2 + b_2d_2 + c_2d_2 \end{aligned} \tag{39}$$

$$\begin{aligned} & F_1(d_2, c_2, b_2, a_2, d_3, c_3, b_3, a_3) = (h_1, g_1, f_1, e_1) \\ e_1 = {} & a_2 \\ f_1 = {} & c_2 + d_2 + d_2b_3 + b_2d_3 + b_3d_3 \\ g_1 = {} & 1 + a_2 + b_2 + c_2 + d_2c_3 + c_2d_3 + c_3d_3 \\ h_1 = {} & c_2 + d_2b_3 + b_2d_3 + b_3d_3 \end{aligned} \tag{40}$$

$$
\begin{aligned}
&F_2(d_3, c_3, b_3, a_3, d_1, c_1, b_1, a_1) = (h_2, g_2, f_2, e_2) \\
e_2 =\ & a_3 \\
f_2 =\ & c_3 + d_3 + d_3 b_1 + b_3 d_1 + b_1 d_1 \\
g_2 =\ & 1 + a_3 + b_3 + c_3 + d_3 c_1 + c_3 d_1 + c_1 d_1 \\
h_2 =\ & c_3 + d_3 b_1 + b_3 d_1 + b_1 d_1
\end{aligned} \tag{41}
$$

$$
\begin{aligned}
&F_3(d_1, c_1, b_1, a_1, d_2, c_2, b_2, a_2) = (h_3, g_3, f_3, e_3) \\
e_3 =\ & a_1 \\
f_3 =\ & c_1 + d_1 + d_1 b_2 + b_1 d_2 + b_2 d_2 \\
g_3 =\ & 1 + a_1 + b_1 + c_1 + d_1 c_2 + c_1 d_2 + c_2 d_2 \\
h_3 =\ & c_1 + d_1 b_2 + b_1 d_2 + b_2 d_2
\end{aligned} \tag{42}
$$

$$
\begin{aligned}
&R_1(d_2, c_2, b_2, a_2, d_3, c_3, b_3, a_3) = (h_1, g_1, f_1, e_1) \\
e_1 =\ & a_2 + b_2 + d_2 b_3 + d_2 c_3 + b_2 d_3 + c_2 d_3 + b_3 d_3 + c_3 d_3 \\
f_1 =\ & b_2 + c_2 \\
g_1 =\ & c_2 + b_2 a_3 + c_2 a_3 + a_2 b_3 + a_3 b_3 + a_2 c_3 + a_3 c_3 \\
h_1 =\ & d_2 + b_3 + c_2 b_3 + b_2 c_3 + b_3 c_3
\end{aligned} \tag{43}
$$

$$
\begin{aligned}
&R_2(d_3, c_3, b_3, a_3, d_1, c_1, b_1, a_1) = (h_2, g_2, f_2, e_2) \\
e_2 =\ & a_3 + b_3 + d_3 b_1 + d_3 c_1 + b_3 d_1 + c_3 d_1 + b_1 d_1 + c_1 d_1 \\
f_2 =\ & b_3 + c_3 \\
g_2 =\ & c_3 + b_3 a_1 + c_3 a_1 + a_3 b_1 + a_1 b_1 + a_3 c_1 + a_1 c_1 \\
h_2 =\ & d_3 + b_1 + c_3 b_1 + b_3 c_1 + b_1 c_1
\end{aligned} \tag{44}
$$

$$
\begin{aligned}
&R_3(d_1, c_1, b_1, a_1, d_2, c_2, b_2, a_2) = (h_3, g_3, f_3, e_3) \\
e_3 =\ & a_1 + b_1 + d_1 b_2 + d_1 c_2 + b_1 d_2 + c_1 d_2 + b_2 d_2 + c_2 d_2 \\
f_3 =\ & b_1 + c_1 \\
g_3 =\ & c_1 + b_1 a_2 + c_1 a_2 + a_1 b_2 + a_2 b_2 + a_1 c_2 + a_2 c_2 \\
h_3 =\ & d_1 + b_2 + c_1 b_2 + b_1 c_2 + b_2 c_2
\end{aligned} \tag{45}
$$

$$
\begin{aligned}
& Q_1(d_2, c_2, b_2, a_2, d_3, c_3, b_3, a_3) = (h_1, g_1, f_1, e_1) \qquad (46) \\
e_1 =\ & 1 + a_2 + b_3 + c_3 + d_2b_3 + d_2c_3 + b_2d_3 + c_2d_3 + b_3d_3 + c_3d_3 \\
f_1 =\ & 1 + a_2 + b_2 \\
g_1 =\ & a_2 + b_2 + c_2 + d_2 + b_3 + c_3 + d_2b_3 + d_2c_3 + b_2d_3 + c_2d_3 + b_3d_3 + c_3d_3 \\
h_1 =\ & d_2 + c_3 + d_3 + b_2a_3 + c_2a_3 + d_2a_3 + a_2b_3 + c_2b_3 + d_2b_3 + a_3b_3 \\
& + a_2c_3 + b_2c_3 + a_3c_3 + b_3c_3 + a_2d_3 + b_2d_3 + a_3d_3 + b_3d_3
\end{aligned}
$$

$$
\begin{aligned}
& Q_2(d_3, c_3, b_3, a_3, d_1, c_1, b_1, a_1) = (h_2, g_2, f_2, e_2) \qquad (47) \\
e_2 =\ & 1 + a_3 + b_1 + c_1 + d_3b_1 + d_3c_1 + b_3d_1 + c_3d_1 + b_1d_1 + c_1d_1 \\
f_2 =\ & 1 + a_3 + b_3 \\
g_2 =\ & a_3 + b_3 + c_3 + d_3 + b_1 + c_1 + d_3b_1 + d_3c_1 + b_3d_1 + c_3d_1 + b_1d_1 + c_1d_1 \\
h_2 =\ & d_3 + c_1 + d_1 + b_3a_1 + c_3a_1 + d_3a_1 + a_3b_1 + c_3b_1 + d_3b_1 + a_1b_1 \\
& + a_3c_1 + b_3c_1 + a_1c_1 + b_1c_1 + a_3d_1 + b_3d_1 + a_1d_1 + b_1d_1
\end{aligned}
$$

$$
\begin{aligned}
& Q_3(d_1, c_1, b_1, a_1, d_2, c_2, b_2, a_2) = (h_3, g_3, f_3, e_3) \qquad (48) \\
e_3 =\ & 1 + a_1 + b_2 + c_2 + d_1b_2 + d_1c_2 + b_1d_2 + c_1d_2 + b_2d_2 + c_2d_2 \\
f_3 =\ & 1 + a_1 + b_1 \\
g_3 =\ & a_1 + b_1 + c_1 + d_1 + b_2 + c_2 + d_1b_2 + d_1c_2 + b_1d_2 + c_1d_2 + b_2d_2 + c_2d_2 \\
h_3 =\ & d_1 + c_2 + d_2 + b_1a_2 + c_1a_2 + d_1a_2 + a_1b_2 + c_1b_2 + d_1b_2 + a_2b_2 \\
& + a_1c_2 + b_1c_2 + a_2c_2 + b_2c_2 + a_1d_2 + b_1d_2 + a_2d_2 + b_2d_2
\end{aligned}
$$

B Encoded PRESENT Permutation Layer

The permutation layer of PRESENT is very efficient in hardware. It consists of simple bit permutation over the whole 64-bit state and, therefore, does not require any gates. The predictor for this layer using the extended Hamming code can be found similar to Equation (27) by transforming the 64×64 permutation

matrix. To this end the permutation matrix need to be twice multiplied with

$$\begin{pmatrix}
P & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & P & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & P & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & P & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & P & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & P & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & P & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & P & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & P & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & P & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & P & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & P & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & P & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & P & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & P & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & P
\end{pmatrix}. \tag{49}$$

In our scenario the two matrices P and $\mathbf{0}$ are defined as

$$P = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \text{ and } \mathbf{0} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{50}$$

While the original permutation matrix has only a single '1' per row, this is not the case for the predictors. Compared to the original permutation, the corresponding predictor requires several XOR gates to transform, split and merge the separate codewords. Therefore, it requires a non-negligible overhead to be implemented.