On evaluating fault resilient encoding schemes in software

Breier, Jakub; Hou, Xiaolu; Liu, Yang

2021

Breier, J., Hou, X. & Liu, Y. (2021). On evaluating fault resilient encoding schemes in software. IEEE Transactions On Dependable and Secure Computing, 18(3), 1065-1079. https://dx.doi.org/10.1109/TDSC.2019.2897663

https://hdl.handle.net/10356/151844

https://doi.org/10.1109/TDSC.2019.2897663

On Evaluating Fault Resilient Encoding Schemes in Software

Jakub Breier[1*], Xiaolu Hou[2*] and Yang Liu[3]
[1]Underwriters Laboratories, Singapore
[2]Acronis, Singapore
[3]School of Computer Science and Engineering
Nanyang Technological University, Singapore
Email: jbreier@jbreier.com, ho0001lu@e.ntu.edu.sg, yangliu@ntu.edu.sg

Abstract—Cryptographic implementations are often vulnerable against physical attacks, fault injection analysis being among the most popular techniques. On par with development of attacks, the area of countermeasures is advancing rapidly, utilizing both hardware- and software-based approaches. When it comes to software encoding countermeasures for fault protection and their evaluation, there are very few proposals so far, mostly focusing on single operations rather than cipher as a whole.

In this paper we propose an evaluation framework that can be used for analyzing the effectivity of software encoding countermeasures against fault attacks. We first formalize the encoding schemes in software, helping us to define what properties are required when designing a fault protection. Based on these findings, we develop an evaluation metric that can be used universally to determine the robustness of a software encoding scheme against bit flip faults and instruction skips. We provide a way to select a code according to user criteria and also a dynamic code analysis method to estimate the level of protection of assembly implementations using encoding schemes. Finally, we verify our findings by implementing a block cipher PRESENT, protected by encoding scheme based on anticodes, and provide a detailed evaluation of this implementation using different codes.

Index Terms—fault injection attacks, encoding schemes, software implementations, block ciphers, cryptography, coding theory

1 Introduction

Protection and physical attacks on cryptographic implementations are ever-evolving areas, resulting into continuous effort on each side to make advancements over the other one. Attackers utilize various techniques that can break the protection and reveal information about the data or secret key. On the other hand, data owners and custodians try to prevent these attacks by applying wide range of countermeasures.

There are various ways to analyze a device and its implementation, Fault Analysis (FA) being one of the most popular ones. Since the first reported attacks, protecting the implementations of ciphers have become a major concern. When selecting a countermeasure, one needs to decide what degree of protection to implement, taking into account the data value and protection price. There is no universal countermeasure, each method has its advantages and limitations. In general, countermeasures can be classified into hardware-based and software-based.

Implementers currently still rely more on hardware-based approaches, such as shielding [1], sensors [1], or hardware redundancy [2]. This is mostly because to inject a fault, physical methods are normally used, such as lasers, electromagnetic pulses, or voltage/clock glitches [3], and therefore, physical protections are effective in detecting/thwarting these.

There are works that utilize encoding techniques in hardware to provide fault resiliency, e.g. [4], [5], [6]. However, there is no straightforward way to implement such schemes in software and therefore, these papers do not provide any details on potential efficiency and security in case the countermeasure is ported into software.

Our Contribution

In this work we are interested in analyzing software encoding countermeasures for a full cipher implementation. To facilitate the evaluation, we formalize fault models and encoding countermeasures in software, bringing light into understanding of what is needed and what is possible.

We formalize evaluation metrics that measure the robustness of a code against bit flip faults and instruction skips on a full cipher implementation. We present the exact formula of our metric for a code used in protecting one single operation. Such an analysis gives us insights on what kind of codes to choose – we show that both the *minimum* and *maximum* distances of a code are important. This leads us to the notion of *anticode* from coding theory which is a definition of code that bounds both minimum and maximum distances of a binary code.

We provide theoretical analysis for what parameters an anticode exists, which gives a direct overview of feasibility without the need to manually search for the anticode existence. As the next step, we present an algorithm to automatically select anticodes with required properties for protecting cryptographic implementations against DFA (if such codes exist).

We develop an evaluation method for encoding countermeasures that is based on dynamic code analysis and works directly on assembly implementations. We implemented a protected version of PRESENT-80 cipher by using an AVR assembly language and

[*] *This research was done while the authors were with Nanyang Technological University, Singapore*

used our evaluation method to analyze the performance of different anticodes w.r.t. aforementioned metric. Our results reveal what trade-offs between the security level and the efficiency (speed, time) can be achieved. Both advantages and disadvantages are discussed. To the best of our knowledge, this is the first work implementing and evaluating the software encoding countermeasure on a full cipher.

The rest of the paper is organized as follows. Section 2 discusses related works. Section 3 formalizes fault attacks and encoding countermeasures in software. In Section 4 we present our metric for evaluating a code used in encoding countermeasure with respect to bit flip faults and instruction skips. The exact formula for our metric in case the code is used for protecting one operation is provided in Section 5 along with the notion of anticodes. Algorithms used for code selection and for evaluation of software implementations are detailed in Section 6. Section 7 provides a case study on block cipher PRESENT. Section 8 gives a guideline on how to choose anticode parameters. Discussion is stated in Section 9 and finally, Section 10 concludes this paper and provides a motivation for future work.

2 Related Work

2.1 Differential Fault Analysis

When it comes to analyzing symmetric block ciphers under fault conditions, the most effective and popular method is the Differential Fault Analysis (DFA) [7]. Following this method, the attacker normally disturbs the computation circuit during the last three rounds of the encryption and then she compares the faulted ciphertext with the non-faulty one. By analyzing this pair of ciphertexts, she can get the information about the secret key used in the encryption. In some cases, single pair is enough to reduce the key search space to a feasible number [8], [9]. In other cases, several fault injections are necessary [10], [11].

2.2 Countermeasures

Software countermeasures against fault attacks can be generally divided into two main groups: instruction-level and algorithm-level techniques [12]. Instruction-based countermeasures include instruction duplication or triplication, and fault-tolerant instruction sequences, where an instruction is replaced by functionally equivalent sequence of more secure instructions [13]. This technique was recently extended to a new approach, called *intra-instruction redundancy* [14]. In this technique, data is split among several instructions, by using a redundant bit-slicing.

On the other hand, algorithm-level countermeasures include temporal and information redundancy on an algorithm level [15]. Temporal redundancy techniques normally execute the algorithm several times and then compare the results for inconsistencies [3], [16].

Software encoding countermeasures fall in the second category, introducing the redundancy in the information being processed. Depending on the encoding scheme design and amount of redundancy, these countermeasures can provide a robust alternative to hardware-based approaches [17]. Breier and Hou [18] showed how to select codes with desired fault properties for protecting binary operations. Theoretical bounds of software encoding countermeasure used in a whole cipher implementation are considered in [19], [20]. However, no real implementation or simulation was given in either work. Servant et al. [20] considered a particular code when used in a full cipher, which they referred to as (3,6)-code, that is actually a $(6, 16, 2)-$binary code (see Definition 3). The probability of detecting a fault was analyzed in this case and it is 93.75%. The approach in [19] does not consider some important aspects of fault injection, such as ability of the attacker to precisely select the fault mask or his ability to inject instruction skips. Generally, to avoid a successful fault injection attack for the countermeasure in [19], used code would have to remain a secret.

2.3 Countermeasure Evaluation Methods

Moro et al. [21] developed an evaluation platform based on electromagnetic fault injection to experimentally verify temporal redundancy countermeasures at assembly instruction level. They implemented a protected version of FreeRTOS to conduct the study. Two countermeasures were tested – an instruction skip protection and a fault detection that is applicable to a subset of assembly instructions. Their experiments showed that both countermeasures work in a way they are supposed to, however with obvious limitations that come from their designs – they either protect only against instruction skips and not against other, more complex fault models, or they can only protect several chosen instructions of the code.

Yuce et al. [12] provided experimental evaluation of several instruction level countermeasures by using a single clock glitches. They showed that the most popular choices, such as instruction duplication/triplication, parity, and instruction skip countermeasure can be broken by a careful choice of fault scenario.

Goubet et al. [22] aimed at formal verification of countermeasures by using automata and SMT solver. Such approach required a decomposition of a code into pieces, while analyzing each piece separately. Also, the method works by comparing the unprotected code with the protected one. The proposed method, however, is not scalable for a full cipher evaluation – for code snippets, where 10 lines of code need 10.7 s to evaluate. Furthermore, analyzing small snippets separately might not reveal the vulnerabilities that might arise from connecting them to a full implementation (cf. Remark 10).

Breveglieri et al. [6] evaluate a subset of encoding-based countermeasures for hardware, based on parity/residue check bits. However, such methods provide only a limited amount of security – odd number of bit-flips is detected, but even number always passes the checks. Moreover, the attacker can also disturb the parity bit or the "checkpoint" which provides the integrity check.

In case of encoding based software countermeasures, there are no works proposing a full cipher evaluation to the best of our knowledge. The closest works to this one evaluate only a single operation on encoded data [18], [17].

Our method is universal for encoding based software countermeasures and provides details on all the possible bit flips and instruction skips. Also, the dynamic code analysis technique that was implemented can efficiently evaluate a full cipher implementation in a short time.

3 Software Encoding Schemes

In this section we first give the formalization of fault attacks in software. Then, we provide necessary coding theory background and present the formalization of encoding countermeasure that can be applied to all symmetric ciphers, which we refer to as *fault resilient encoding scheme.*

3.1 Fault Attacks in Software

Assembly language is a low-level programming language, specific to a particular architecture. Normally, there is a one-to-one mapping between assembly instructions and machine code that is being executed on the device. Assembly language uses a mnemonic to represent machine operations in the form of instructions. Each instruction falls into one of three categories: data movement, arithmetic/logic, and control-flow.

Operands are entities operated upon by an instruction. Addresses are the locations of specified data in the memory. Operands can be immediate (constant values), registers (values in the processor number registers), or memory (value stored in the memory). Standard instruction can have zero to three operands, the leftmost operand being usually the destination register, the second and the third are source registers.

For our purpose, registers are the most important storage units. Size of the register is typically stated in bits and depends on the device architecture (e.g. 8-bit, 32-bit, 64-bit). Normally, all the registers for a particular device have the same size. It is the fastest type of memory in a computer and it is directly accessible by the arithmetic logic unit (ALU) performing the operations.

Definition 1. *We define a* program *to be an ordered sequence of assembly instructions $\mathcal{F} = \{f_1, f_2, \ldots, f_{N_\mathcal{F}}\}$. $N_\mathcal{F}$ is called the* number of instructions *for the program. For any assembly instruction $f \in \mathcal{F}$, if f has a destination register, we denote this register by r_f. Let $\mathscr{S}$ denote the set of all programs.*

Fault attack is an intentional change of the original data value into a different value. This change can either happen in a register/memory, on the data path, or directly in ALU. In general, there are two main fault models to be considered – program flow disturbances and data flow disturbances. The first one is achieved by disturbing the instruction execution process that can result in changing or skipping the instruction currently being executed. The second one is achieved either by directly changing the data values in storage units, or by changing the data on the data paths or inside ALU. For the purpose of a fault injection attack, these three data flow changes are equivalent and can be modeled by changing the values in registers.

Definition 2 (Instruction skip and fault mask)**.** *1) For any $i \in \mathbb{Z}_{>0}$, an i*th instruction skip *is a function $\vartheta_i : \mathscr{S} \to \mathscr{S}$, such that $\vartheta_i(\mathcal{F}) = \mathcal{F}$ if $N_\mathcal{F} < i$ and $\vartheta_i(\mathcal{F}) = \mathcal{F}\backslash\{f_i\}$ otherwise.*

2) For any $\boldsymbol{j} \in \mathbb{F}_2^N\backslash\{\mathbf{0}\}$ $(N \in \mathbb{Z}_{>0})$, a fault mask $\boldsymbol{j}$ on instruction *i is a function $\varsigma_{i,\boldsymbol{j}} : \mathscr{S} \to \mathscr{S}$ such that for any $\mathcal{F} = \{f_1, f_2, \ldots, f_{N_\mathcal{F}}\} \in \mathscr{S}$,*

- *if $1 \leq i < N_\mathcal{F}$ and f_i has a destination register r_{f_i} whose length is at least N, then $\varsigma_{i,\boldsymbol{j}}(\mathcal{F}) = \{f_1, f_2, \ldots, f_i, \tilde{f}_i, f_{i+1}, f_{N_\mathcal{F}}\}$, where $\tilde{f}_i =$* `eor` *r_{f_i} $\boldsymbol{j}$, i.e. $\tilde{f}_i$ changes the value in r_{f_i}, to be the* `xor`*ed result of value in r_{f_i} and $\boldsymbol{j}$.*
- *$\varsigma_{i,\boldsymbol{j}}(\mathcal{F}) = \mathcal{F}$ otherwise.*

In our evaluation framework, we consider a *single fault adversary* – under this attacker model, at most one fault is injected during the encryption/decryption algorithm execution. The attacker can inject a random m−bit flip fault such that all the bits have equal probability to be affected by the fault. In other words, for a random m−bit flip, each fault mask value between 1 and N has the same probability to occur.

3.2 Fault Resilient Encoding Scheme

Encoding scheme in our context is a protection method that acts against fault injection attack by detecting malicious changes to secret data processed by the encryption algorithm. In this part we provide a necessary formalization which establishes the foundation for Section 4, where a generic metric for evaluating encoding scheme robustness is proposed.

A *binary code*, which we denote by C, is a subset of $\mathbb{F}_2^n$, the n−dimensional vector space over $\mathbb{F}_2$, where n is called the *length* of the code C. Each element $\boldsymbol{c} \in C$ is called a *codeword* of C and each element $\boldsymbol{x} \in \mathbb{F}_2^n$ is called a *word* [23, p.6]. Take two words $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_2^n$, the *Hamming distance* between $\boldsymbol{x}$ and $\boldsymbol{y}$, denoted by $\text{dis}(\boldsymbol{x}, \boldsymbol{y})$, is defined to be the number of places at which $\boldsymbol{x}$ and $\boldsymbol{y}$ differ [23, p.9]. More precisely, if $\boldsymbol{x} = x_1x_2\ldots x_n$ and $\boldsymbol{y} = y_1y_2\ldots y_n$, then

$$\text{dis}(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{n} \text{dis}(x_i, y_i),$$

where x_i and y_i are treated as binary words of length 1 and hence

$$\text{dis}(x_i, y_i) = \begin{cases} 1 & \text{if } x_i \neq y_i \\ 0 & \text{if } x_i = y_i \end{cases}.$$

Furthermore, for a word $\boldsymbol{x} \in \mathbb{F}_2^n$, the *Hamming weight* of $\boldsymbol{x}$, $\text{wt}(\boldsymbol{x}) := \text{dis}(\boldsymbol{x}, \mathbf{0})$ [23, p.46]. For a binary code C, the *(minimum) distance* of C, denoted by $\text{dis}(C)$, is [23, p.11]

$$\text{dis}(C) = \min\{\text{dis}(\boldsymbol{c}, \boldsymbol{c}') : \boldsymbol{c}, \boldsymbol{c}' \in C, \boldsymbol{c} \neq \boldsymbol{c}'\}.$$

Definition 3. *[24, p.75] For a binary code C of length n, with $\text{dis}(C) = d$, if $M = |C|$ is the number of codewords in C. Then C is called an (n, M, d)−binary code.*

Remark 1. *In case C is a subspace of $\mathbb{F}_2^n$, C is called a linear code. For a linear code with dimension k, a standard notion would be $[n, k, d]$, where n is its length and d is its minimum distance. For a non-linear code, there is no notion of dimension and we follow the standard notion (n, M, d) as presented in [24]. We would like to emphasize that we do not restrict the code to linear codes, allowing the analysis to more code candidates used in encoding countermeasure.*

To simplify the notation we introduce the symbol $\perp$, which indicates an error. Note that the exact implementation of $\perp$ gives certain restrictions on the code C that can be used: if zero is used to implement $\perp$, we should require that $\mathbf{0} \notin C$.

To formally define encoding countermeasure, we first adopt the definition of symmetric cipher from [25]:

Definition 4. *A* symmetric cipher *(see e.g. [25, p.37]) is a 5−tuple $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)$ such that*

$$E : \mathcal{K} \times \mathcal{P} \to \mathcal{M}, \quad D : \mathcal{K} \times \mathcal{M} \to \mathcal{P},$$

and $\forall \kappa \in \mathcal{K}$, $\forall P \in \mathcal{P}$, $D(\kappa, E(\kappa, P)) = P$. We refer to $\mathcal{K}$, $\mathcal{P}$, $\mathcal{M}$, E and D as key space, plaintext space, ciphertext space, encryption *and* decryption *of this cipher, respectively. We define $\mathfrak{S}$ to be the set of all symmetric ciphers $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)$ such that*

$$\mathcal{K} = \mathbb{F}_2^{N_1}, \quad \mathcal{P} = \mathbb{F}_2^{N_2}, \quad \mathcal{M} = \mathbb{F}_2^{N_3},$$

for some $N_1, N_2, N_3 \in \mathbb{Z}_{>0}$.

A symmetric cipher with encoding countermeasure either outputs an error message or the correct ciphertext. We give the formal definition of such a cipher as follows:

Definition 5. *An* error detection symmetric cipher *is a* $5-$*tuple* $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)$*, where*

1) $\perp \in \mathcal{M}$,
2) $E : \mathcal{K} \times \mathcal{P} \to \mathcal{M}, D : \mathcal{K} \times \mathcal{M} \to \mathcal{P} \cup \{\perp\}$ *are functions such that* $\forall \kappa \in \mathcal{K}$, $\forall P \in \mathcal{P}$
   a) *if* $D(\kappa, E(\kappa, P)) \neq \perp$ *then* $D(\kappa, E(\kappa, P)) = P$;
   b) $D(\kappa, \perp) = \perp$.

Let $\mathfrak{S}_\perp$ *denote the set of all error detection symmetric ciphers* $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)$ *such that*

$$\mathcal{K} = \mathbb{F}_2^{N_1}, \ \ \mathcal{P} = \mathbb{F}_2^{N_2}, \ \ \mathcal{M} = \mathbb{F}_2^{N_3} \cup \{\perp\},$$

for some $N_1, N_2, N_3 \in \mathbb{Z}_{>0}$.

In encoding countermeasure, the important part is the error detection, which is closely related to the encoding and decoding. Here we formalize the notion of encoder and decoder.

Definition 6. *Given an* $(n, M = 2^k, d)-$*binary code* C*, an* encoding-decoding scheme associated with C *is a pair of functions* $(\mathtt{Encoder}_C, \ \mathtt{Decoder}_C)$

$$\mathtt{Encoder}_C : \mathbb{F}_2^k \to C, \ \ \mathtt{Decoder}_C : \mathbb{F}_2^n \cup \{\perp\} \to \mathbb{F}_2^k \cup \{\perp\}$$

such that $\mathtt{Decoder}_C\big|_{(\mathbb{F}_2^n \cup \{\perp\}) \setminus C} = \{\perp\}$ *and* $\mathtt{Encoder}_C$ *is bijective with* $\mathtt{Decoder}_C\big|_C$ *being its inverse.*

Thus for $\mathtt{Decoder}_C$ an error message $\perp$ will be returned if the input is not a codeword. More details regarding encoding-decoding schemes can be found in Appendix A.

Definition 7. *An* operation *is a map* $g : \mathbb{F}_2^{M_1} \times \mathbb{F}_2^{M_2} \times \cdots \times \mathbb{F}_2^{M_m} \to \mathbb{F}_2^{M_{m+1}}$ *for some positive integers* $M_1, M_2, \ldots, M_{m+1}$*. Let* $\mathcal{S}$ *denote the set of all operations.*

Note that an assembly implementation of an operation is a program (see Definition 1).

Example 1. *The* $\mathtt{xor}$ *operation defined on* 1*-bit strings is an operation* $g : \mathbb{F}_2 \times \mathbb{F}_2 \to \mathbb{F}_2$ *such that*

$$g(0,0) = 0, \ g(0,1) = 1, \ g(1,0) = 1, \ g(1,1) = 0.$$

Definition 8. *An* operation with error detection *is a map* $h : (\mathbb{F}_2^{M_1} \cup \{\perp\}) \times (\mathbb{F}_2^{M_2} \cup \{\perp\}) \times \cdots \times (\mathbb{F}_2^{M_m} \cup \{\perp\}) \to \mathbb{F}_2^{M_{m+1}} \cup \{\perp\}$ *for some positive integers* $M_1, M_2, \ldots, M_{m+1}$ *such that if* $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m) \in (\mathbb{F}_2^{M_1} \cup \{\perp\}) \times (\mathbb{F}_2^{M_2} \cup \{\perp\}) \times \cdots \times (\mathbb{F}_2^{M_m} \cup \{\perp\})$ *satisfies* $\boldsymbol{x}_i = \perp$ *for at least one* $i \in \{1, 2, \ldots, m\}$*, then* $h(\boldsymbol{x}) = \perp$*. Let* $\mathcal{S}_\perp$ *denote the set of all operations with error detection.*

Remark 2. *By the above definition, for any symmetric cipher* $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D) \in \mathfrak{S}$*,* $E, D \in \mathcal{S}$*. For any error detection symmetric cipher* $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D) \in \mathfrak{S}_\perp$*,* $D \in \mathcal{S}_\perp$*. Furthermore, for an* $(n, M = 2^k, d)-$*binary code* C *with associated encoding-decoding scheme* $(\mathtt{Encoder}_C, \mathtt{Decoder}_C)$*,* $\mathtt{Encoder}_C \in \mathcal{S}$ *and* $\mathtt{Decoder}_C \in \mathcal{S}_\perp$.

Example 2. *Consider the following operation with error detection* $h : (\mathbb{F}_2 \cup \{\perp\}) \times (\mathbb{F}_2 \cup \{\perp\}) \to \mathbb{F}_2 \cup \{\perp\}$*.* h *outputs the* $\mathtt{xor}$ *of two bits when no error is detected:*

$$h(0,0) = 0, \ h(0,1) = 1, \ h(0,\perp) = \perp, \ h(\perp,0) = \perp, \ h(\perp,\perp) = \perp,$$
$$h(1,0) = 1, \ h(1,1) = 0, \ h(1,\perp) = \perp \ \ h(\perp,1) = \perp \ .$$

An operation $g \in \mathcal{S}$ can be changed to an operation with error detection utilizing binary codes:

Definition 9. *Given an* $(n, M = 2^k, d)-$*binary code* C*,* $\varphi_C : \mathcal{S} \to \mathcal{S}_\perp$ *is defined as follows:*
Take any $g : \mathbb{F}_2^{M_1} \times \mathbb{F}_2^{M_2} \times \cdots \times \mathbb{F}_2^{M_m} \to \mathbb{F}_2^{M_{m+1}} \in \mathcal{S}$*, for* $1 \leq i \leq m+1$*, suppose* $\{\mathtt{Encoder}_C(\boldsymbol{x}) | \boldsymbol{x} \in \mathbb{F}_2^{M_i}\} = C^{k_i} \subseteq \mathbb{F}_2^{nk_i}$*,* $\varphi_C(g)$ *is a function:*

$$\varphi_C(g) : \left(\mathbb{F}_2^{nk_1} \cup \{\perp\}\right) \times \left(\mathbb{F}_2^{nk_2} \cup \{\perp\}\right) \times \cdots \times \left(\mathbb{F}_2^{nk_m} \cup \{\perp\}\right) \to C^{k_{m+1}} \cup \{\perp\}$$

such that for $\boldsymbol{x} = \Big(\mathtt{Encoder}_C(\boldsymbol{x}_1), \mathtt{Encoder}_C(\boldsymbol{x}_2), \ldots, \mathtt{Encoder}_C(\boldsymbol{x}_m)\Big) \in C^{k_1} \times C^{k_2} \times \ldots C^{k_m}$*,* $\varphi_C(g)(\boldsymbol{x}) = \mathtt{Encoder}_C\Big(g(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m)\Big)$*, and* $\forall \boldsymbol{x} \in \left(\mathbb{F}_2^{nk_1} \cup \{\perp\}\right) \times \left(\mathbb{F}_2^{nk_2} \cup \{\perp\}\right) \times \cdots \times \left(\mathbb{F}_2^{nk_m} \cup \{\perp\}\right) \setminus C^{k_1} \times C^{k_2} \times \ldots C^{k_m}$*,* $\varphi_C(g)(\boldsymbol{x}) = \perp$.

Example 3. *Let us take the function* g *from Example 1 and take the following* $(2,2,2)-$*binary code* $C = \{00, 11\}$ *with the following encoding-decoding scheme:*

$$\begin{aligned} \mathtt{Encoder}_C : & \quad 0 \mapsto 00, \ \ 1 \mapsto 11, \\ \mathtt{Decoder}_C : & \quad 00 \mapsto 0, \ 01 \mapsto \perp, \ 10 \mapsto \perp, \ 11 \mapsto 1. \end{aligned}$$

Then $\varphi_C(g) : (\mathbb{F}_2^2 \cup \{\perp\}) \times (\mathbb{F}_2^2 \cup \{\perp\}) \to \ C$ *is an operation with error detection and*

$$\varphi_C(g)(\boldsymbol{x}_1, \boldsymbol{x}_2) = \begin{cases} 00 & \boldsymbol{x}_1 = \boldsymbol{x}_2 = 00 \textit{ or } 11 \\ 11 & \boldsymbol{x}_1 = 00, \boldsymbol{x}_2 = 11 \textit{ or } \boldsymbol{x}_1 = 11, \boldsymbol{x}_2 = 00 \\ \perp & \textit{otherwise}. \end{cases}$$

Lemma 1. *Let* $g_1, g_2 \in \mathcal{S}$ *such that* $g_2 \circ g_1 \in \mathcal{S}$*, then* $\varphi_C(g_2 \circ g_1) = \varphi_C(g_2) \circ \varphi_C(g_1)$.

(The proof can be found in Appendix B.1.)

Remark 3. *For any symmetric cipher* $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D) \in \mathfrak{S}$*, any* $(n, M = 2^k, d)-$*binary code* C *with an associated encoding-decoding scheme* $(\mathtt{Encoder}_C, \mathtt{Decoder}_C)$*, if we write* $E = g_1 \circ g_2 \circ \cdots \circ g_m$ *for* $g_1, g_2, \ldots, g_m \in \mathcal{S}$*, then* $\varphi_C(E) = \varphi_C(g_1) \circ \varphi_C(g_2) \circ \cdots \circ \varphi_C(g_m)$.

This justifies that we can split or merge multiple cipher operations while considering applying encoding countermeasure to a symmetric cipher (cf. Section 7.1).

Encoding countermeasure applied to a symmetric cipher can be considered as applying a function which is closely related to a binary code on the encryption and decryption of the cipher. Here we give the definition of such a function.

Definition 10 (Fault resilient C-map)**.** *Given an* $(n, M = 2^k, d)-$*binary code* C *with an associated encoding-decoding scheme* $(\mathtt{Encoder}_C, \mathtt{Decoder}_C)$*, we define* fault resilient C-map *to be the following function*

$$\begin{aligned} \Phi_C : \mathfrak{S} & \quad \to \quad \mathfrak{S}_\perp \\ (\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D) & \quad \mapsto \quad (\mathcal{K}, \mathcal{P}, \mathcal{M} \cup \{\perp\}, E', D'), \end{aligned}$$

such that $\forall P \in \mathcal{P}, \kappa \in \mathcal{K}, \mathtt{Msg} \in \mathcal{M} \setminus \{\perp\}$,

$$E'(\kappa, P) = \mathtt{Decoder}_C\Big(\varphi_C(E)\Big(\mathtt{Encoder}_C(\kappa), \mathtt{Encoder}_C(P)\Big)\Big),$$

$$D'(\kappa, \mathtt{Msg}) = \mathtt{Decoder}_C\Big(\varphi_C(D)\Big(\mathtt{Encoder}_C(\kappa), \mathtt{Encoder}_C(\mathtt{Msg})\Big)\Big),$$

and $D'(\kappa, \perp) = \perp$.

Now we are ready to formalize encoding countermeasure, which we refer to as *fault resilient encoding scheme*.

Definition 11 (Fault resilient encoding scheme)**.** *Given* $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D) \in \mathfrak{S}$ *a symmetric cipher and* C *an* $(n, M = 2^k, d)-$*binary code with an encoding-decoding scheme* $(\mathtt{Encoder}_C, \mathtt{Decoder}_C)$.

A cipher of the form $\Phi_C\Big((\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)\Big)$ *is called a* fault resilient encoding scheme.

Remark 4. *Taking* $k = 1$ *and* $C = \{01, 10\}$*, we get the bit-sliced encoding, e.g. the one used in [26] (*$\texttt{Encoder}_C(0) = 01$*, and* $\texttt{Encoder}_C(1) = 10$*) which follows the principle of a dual-rail precharge logic. In Section 7.1, we use* $k = 4$ *mainly because PRESENT cipher uses 4-bit SBox (see Section 7.1).*

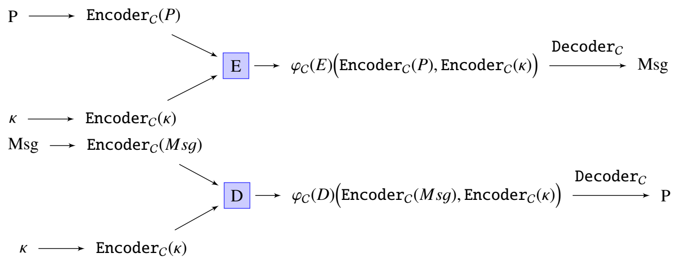


Fig. 1: Overview of the fault resilient encoding scheme.

For a better understanding of how the fault resilient encoding scheme works, the design overview is stated in Figure 1. Informally, first, an encoder is applied to both the plaintext and the key. Then, the encryption process is performed, preserving the encoding. In the end, a decoder is applied in order to get the encrypted message. The decryption process is analogous.

4 Evaluation Metric

In this section we first formalize faults in encoding schemes and provide concepts of *safe* and *missed* faults. Then, we propose two metrics for evaluating different binary codes used for fault resilient encoding scheme: one for bit flip fault model and one for instruction skip fault model.

4.1 Faults in Fault Resilient Encoding Schemes

We first give the definition of safe and missed faults for an implementation of $\varphi_C(g)$ (i.e. for a single operation), where C is a binary code and g is an operation.

Definition 12. *Given an* $(n, M = 2^k, d)-$*binary code* C *with encoding-decoding scheme* $(\texttt{Encoder}_C, \texttt{Decoder}_C)$*, an operation* $g \in \mathcal{S}$*, let* $\mathcal{F}$ *be an assembly implementation of* $\varphi_C(g)$*. Suppose* $\mathcal{F} = \{f_1, f_2, \ldots, f_{N_\mathcal{F}}\}$*,*

1) The set of possible instruction skips *for* $\mathcal{F}$ *is*

$$\mathcal{G}_{(\mathcal{F},\mathrm{sk})} := \{\vartheta_i : 1 \le i \le N_\mathcal{F}\}.$$

2) The set of possible fault masks *for* $\mathcal{F}$ *is*

$$\mathcal{G}_{(\mathcal{F},\mathfrak{fm})} := \{\varsigma_{i,\boldsymbol{j}} : 1 \le i \le N_\mathcal{F}, \boldsymbol{j} \in \mathbb{F}_2^n \backslash \{\boldsymbol{0}\}, f_i \text{ has a destination register}\}. \quad (1)$$

3) For an integer $1 \le m \le n$*,* the set of possible $m-$bit flips *for* $\mathcal{F}$ *is*

$$\mathcal{G}_{(\mathcal{F},\mathfrak{fm},m)} := \{\varsigma_{i,\boldsymbol{j}} : \varsigma_{i,\boldsymbol{j}} \in \mathcal{G}_{(\mathcal{F},\mathfrak{fm})}, wt(\boldsymbol{j}) = m\}.$$

4) A fault on $\mathcal{F}$ *is defined to be a function* ϱ *such that* $\varrho \in \mathcal{G}_{(\mathcal{F},\mathrm{sk})}$ *or* $\varrho \in \mathcal{G}_{(\mathcal{F},\mathfrak{fm})}$*.*

5) Fixing an input x, a fault ϱ *on* $\mathcal{F}$ *is said to be* safe *if* $\varrho\Big(\mathcal{F}\Big) = \perp$ *or* $g(x)$*; and it is said to be a* missed *fault otherwise.*

Remark 5. *A fault is closely related to a tampering function defined in [27]. In our notation, a fault is defined on the program code level, but in a broader sense, the effect of introducing a fault in the program execution can be considered as an application of a tampering function.*

Given an $(n, M = 2^k, d)-$binary code C associated with an encoding-decoding scheme $(\texttt{Encoder}_C, \texttt{Decoder}_C)$ and a symmetric cipher $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)$. Let $(\mathcal{K}, \mathcal{P}, \mathcal{M} \cup \{\perp\}, E', D') := \Phi_C\Big((\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)\Big)$. The assembly implementations of E' and D' are programs. If we let $\mathcal{F}_1$ and $\mathcal{F}_2$ be the assembly implementations of E' and D' respectively, then for any $\kappa \in \mathcal{K}, P \in \mathcal{P}, \texttt{Msg} \in \mathcal{M} \cup \{\perp\}$, $\mathcal{F}_1(\kappa, P) = E'(\kappa, P)$ and $\mathcal{F}_2(\kappa, \texttt{Msg}) = D'(\kappa, \texttt{Msg})$. We assume the registers involved in the implementation all have length at least n. Recall that $E, D \in \mathcal{S}$ (Remark 2), we hence give the following definition of safe and missed faults for a fault resilient encoding scheme.

Definition 13 (Safe and missed faults)**.** *For a fixed plaintext* $P \in \mathcal{P}$ *and a key* $\kappa \in \mathcal{K}$*, a fault* ϱ_1 *on* $\mathcal{F}_1$ *is* safe *if* $\varrho\Big(\mathcal{F}_1\Big)(\kappa, P) = \perp$ *or* $E(\kappa, P)$ *and it is called a* missed *fault otherwise. Similarly, a fault* ϱ_2 *on* $\mathcal{F}_2$ *is* safe *if* $\varrho\Big(\mathcal{F}_2\Big)(\kappa, P) = \perp$ *or* $D(\kappa, P)$ *and it is called a* missed *fault otherwise.*

Recall that for a differential fault analysis [7], the attacker needs to inject a fault during the execution. Based on where the fault is introduced, diffusion can spread it up to the whole cipher state by the end of encryption. Attacker then compares the faulty output with the correct one and can gain information about the secret key. If the fault is *missed*, the attacker can use similar technique. In this case, the cipher output would be equivalent to the faulty output obtained by attacking an unprotected cipher implementation. On the other hand, if the fault is *safe*, it means the output is either $\perp$ or the correct output, which will not give the attacker valuable information.

4.2 Metrics for Bit Flips and Instruction Skips

In this part we give the metrics we use to evaluate the fault resistance property of a binary code used in fault resilient encoding scheme. Since bit flips and instruction skips are quite different fault models in nature, we propose different metrics for each of them.

The metrics are defined for the implementation of encryption. Similar metrics can be defined for the implementation of decryption.

As mentioned earlier, for an $m-$bit flip fault attack model, we assume all combinations of m bits have equal probability to be flipped. Thus,

$$Pr[\varsigma \text{ was injected}] = \frac{1}{|\mathcal{G}_{(\mathcal{F}_1,\mathfrak{fm},m)}|}, \qquad \forall \varsigma \in \mathcal{G}_{(\mathcal{F}_1,\mathfrak{fm},m)}.$$

Furthermore, given a particular fault ϱ, the probability that ϱ is safe is calculated assuming that the plaintext and key are independent random variables following uniform distribution[1]. More precisely,

$$Pr[\varrho \text{ is safe}] = \frac{|\{p, \kappa : P \in \mathcal{P}, \kappa \in \mathcal{K}, \varrho \text{ is safe for plaintext } P, \text{ key } \kappa\}|}{|\mathcal{P}||\mathcal{K}|}.$$

Definition 14 ($m-$bit fault resistance probability)**.** *Following the notations from Definition 13. Let m be an integer such that*

[1] This means $Pr[\text{plaintext} = P] = \frac{1}{|\mathcal{P}|}\ \forall P \in \mathcal{P}$, similarly for κ.

$1 \leq m \leq n$, *the* $m-$bit fault resistance probability *of C w.r.t. $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)$ and $\mathcal{F}_1$, denoted by $p_{C,m}$, is defined as*

$$\begin{aligned} p_{C,m} &:= \sum_{\varsigma \in \mathcal{G}_{(\mathcal{F}_1, \mathtt{fm}, m)}} Pr[\varsigma \textit{ is safe}] Pr[\varsigma \textit{ was injected}] \\ &= \frac{1}{|\mathcal{G}_{(\mathcal{F}_1, \mathtt{fm}, m)}|} \sum_{\varsigma \in \mathcal{G}_{(\mathcal{F}_1, \mathtt{fm}, m)}} Pr[\varsigma \textit{ is safe}]. \end{aligned}$$

We are interested in the best case for the attacker, i.e. we consider she can inject a fault that has the highest probability to be missed by the encoding scheme. Therefore, we have to take the minimum of the $m-$bit fault resistance probabilities. To check the overall resistance of a code in fault resilient encoding scheme, we consider all the possible bit flips and define *bit flip resistance probability* as follows:

Definition 15 (bit flip fault resistance probability)**.** *Given an $(n, M = 2^k, d)-$binary code C, a symmetric cipher $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)$, and an implementation $\mathcal{F}_1$ of E, the* bit flip fault resistance probability *for C w.r.t. to $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)$ and $\mathcal{F}_1$, denoted by $p_{C,\mathtt{bf}}$, is defined as:*

$$p_{C,\mathtt{bf}} := \min_{1 \leq m \leq n} p_{C,m},$$

where $p_{m,C}$ is the the $m-$bit fault resistance probability of C w.r.t. to $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)$ and $\mathcal{F}_1$.

The bit flip fault resistance probability will be used as our metric for evaluating a code used in fault resilient encoding scheme w.r.t. bit flip fault attacks.

For instruction skips, we give the following metric:

Definition 16 (instruction skip resistance probability)**.** *Given an $(n, M = 2^k, d)-$binary code C, a symmetric cipher $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)$, and an implementation $\mathcal{F}_1$ of E, the* instruction skip resistance probability *for C w.r.t. to $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)$ and $\mathcal{F}_1$, denoted by $p_{C,\mathsf{sk}}$, is defined as:*

$$\begin{aligned} p_{C,\mathsf{sk}} &:= \sum_{\vartheta \in \mathcal{G}_{(\mathcal{F}_1, \mathsf{sk})}} Pr[\vartheta \textit{ is safe}] Pr[\vartheta \textit{ was injected}] \\ &= \frac{1}{|\mathcal{G}_{(\mathcal{F}_1, \mathsf{sk})}|} \sum_{\vartheta \in \mathcal{G}_{(\mathcal{F}_1, \mathsf{sk})}} Pr[\vartheta \textit{ is safe}]. \end{aligned}$$

Remark 6. *We do not assume faults on input data, such as plaintext and key. In case the attacker wants to attack these, she could do it anytime before the actual algorithm execution, even before the encoding. Similarly, we do not assume faults on ciphertext – in this case, the attacker would not get any meaningful information about the secret key.*

5 Anticodes for Fault Resilient Encoding Scheme

In this section, we first provide the exact formulas for $p_{C,\mathtt{bf}}$ and $p_{C,\mathsf{sk}}$ in case of a simple "cipher" which consists of one binary operation (Section 5.1). Binary operations are very common in symmetric ciphers, e.g. `xor`, `and`, `modular addition`.

We remark that analyzing the fault resistance property of a code C with respect to a single operation gives insights on the overall fault resistance of using C in fault resilient encoding scheme. Hence it provides a good approximation of the fault resistance of a full cipher implementation.

In Section 5.2 we introduce anticodes which give improved resistance probabilities compared to codes with unbounded distances.

Finally, Section 5.3 provides a way to check the existence of an anticode for given parameters.

```
1  LDI r0 x         // loading of plaintext
2  LDI r1 key       // loading of key
3  EOR r2 r2        // pre-charge register r2 to zero
4  LPM r2 r0 r1     // execution of g by table look up
5  ST y r2          // storing ciphertext
```

TABLE 1: Assembly implementation $\mathcal{F}$ for $\varphi_C(g)$, where $g : \mathbb{F}_2^{M_1} \times \mathbb{F}_2^{M_2} \to \mathbb{F}_2^{M_3}$ is a binary operation.

5.1 Evaluation of Single Operations

Let $g \in \mathcal{S}$ be a binary operation $g : \mathbb{F}_2^{M_1} \times \mathbb{F}_2^{M_2} \to \mathbb{F}_2^{M_3}$ and let C be an $(n, M = 2^k, d)-$binary code with associated encoding-decoding scheme $(\mathtt{Encoder}_C, \mathtt{Decoder}_C)$ and distance $d \geq 2$. We will use zero string to denote $\perp$, the error message. Hence we further require that $\mathbf{0} \notin C$. And we choose k such that $k = \max\{M_1, M_2\}$.

Remark 7. *As mentioned in Remark 1, we do not restrict our codes to be linear. Thus the method of calculating syndrome [23, p.62] of a word and check if this word is a codeword does not apply in our setting. Furthermore, using table lookup for implementation and a null word for denoting error does not require an extra computation (e.g. calculating syndrome) to detect error.*

Let $\mathcal{F}$ be the assembly implementation (in Figure 1) of $\varphi_C(g)$. In $\mathcal{F}$, two different instructions are used: `LDI` loads immediate data into the destination register, `LPM` loads data from a program memory to the destination register – serving as a table lookup for the binary operation g. Before executing each table look-up we precharge the destination register to zero by using exclusive or operation (`EOR` in line 3). Note that the table has $2^n \times 2^n$ entries. The value stored at address (a, b) is zero if $a, b \notin C$ and the value is $\mathtt{Encoder}_C(g(\mathtt{Encoder}_C(x), \mathtt{Encoder}_C(y)))$ if $a = \mathtt{Encoder}_C(x)$ and $b = \mathtt{Encoder}_C(y)$.

By Definition 12 and the assumptions stated in Remark 6, the set of possible instruction skips and the set of possible fault masks for $\mathcal{F}$ are given by

$$\mathcal{G}_{(\mathcal{F}, \mathsf{sk})} = \{\vartheta_3, \vartheta_4\}, \quad \mathcal{G}_{(\mathcal{F}, \mathtt{fm})} = \{\varsigma_{i,\boldsymbol{j}} : i = 3, 4, \boldsymbol{j} \in \mathbb{F}_2^n \backslash \{\mathbf{0}\}\}.$$

The values of $p_{C,m}$ and $p_{C,\mathsf{sk}}$ with respect to the program $\mathcal{F}$ and $\varphi_C(g)$ can be then calculated as follows:

Proposition 1. *1. For $1 \leq m \leq n$, let $S_{m,C} := \sum_{\boldsymbol{c} \in C} |\{\boldsymbol{c}' \in C : \mathrm{dis}(\boldsymbol{c}', \boldsymbol{c}) = m\}|$, then*[2]

$$p_{C,m} = 1 - \frac{S_{m,C}}{2M\binom{n}{m}}. \tag{2}$$

2. $p_{C,sk} = 1$.

(The proof can be found in Appendix B.2.)

Remark 8. *If $S_{m,C} = 0$, then $p_{C,m} = 1$. This is equivalent to saying that in case there are no two codewords in C that are at distance m from each other, $m-$bit flip fault model would not result in missed faults.*

5.2 Fault Resilient Anticode Scheme

In this part, we explain the rationale behind extending the encoding scheme with a usage of anticodes to provide better bit flip resistance probabilities.

[2] [18, Definition 9] gives a similar formula as bit flip resistance probability for binary code used in binary operations. The difference is that in [18] the authors do not assume a precharge of a register.

When selecting the code parameters, the choice of n is dependent on the architecture of the device and the memory constraint. The value of M is mostly related to the cipher design (see Section 8).

For binary codes with the same length n and cardinality M, the formula from Proposition 1 shows that the smaller the value of $\frac{S_{m,C}}{\binom{n}{m}}$, the bigger $p_{C,m}$ can be achieved. By Definition 15, to get a code with higher bit flip fault resistance probability, we want to look at codes where the value of $\frac{S_{m,C}}{\binom{n}{m}}$ is small.

Since $\sum_{m=1}^{n} S_{m,C} = M(M-1)$ is always true, to make $\frac{S_{m,C}}{\binom{n}{m}}$ small, one strategy is to make $S_{m,C}$ small or even equal to zero for smaller values of $\binom{n}{m}$. Let

$$\ell := \begin{cases} \frac{n}{2} & \text{if } n \text{ is even} \\ \frac{n+1}{2} & \text{if } n \text{ is odd} \end{cases}. \tag{3}$$

It is known that (see e.g. [28, p.26])

$$\begin{cases} \binom{n}{i-1} < \binom{n}{i} & \text{if } 1 \le i < \ell \\ \binom{n}{i-1} > \binom{n}{i} & \text{if } \ell < i \le n \end{cases}, \text{ and } \begin{cases} \binom{n}{\ell-1} = \binom{n}{\ell} & \text{if } n \text{ is odd} \\ \binom{n}{\ell-1} < \binom{n}{\ell} & \text{if } n \text{ is even} \end{cases}. \tag{4}$$

Hence, we would like to have $S_{m,C} = 0$ for m "close to" n and we do not want $S_{m,C} = 0$ for too many m (see Lemma 2).

In the view of the above, we recall the notion of *anticode*:

Definition 17. *[29] A binary anticode is an array of binary digits with n rows and M columns, constructed so that the maximum Hamming distance between any pair of rows is less than or equal to a certain value δ. This value δ is the* maximum distance *of the anticode.*

If we have a binary code, we can take its codewords as rows and then get an anticode. Note that a binary code does not have repeated codewords but an anticode can have repeated rows [29]. The above discussion shows that essentially what we want is a binary code which is also an anticode with a proper maximum distance δ. We introduce the following definition.

Definition 18. *Let C be an $(n, M, d)-$binary code and let*

$$\delta := \max_{\boldsymbol{c},\boldsymbol{c}' \in C} \operatorname{dis}(\boldsymbol{c}, \boldsymbol{c}'),$$

then C is called an $(n, M, d, \delta)-$binary anticode (we can see that $d \le \delta \le n$). Furthermore, d (resp. δ) is called the minimum distance *(resp.* maximum distance*) of C.*

From the definition, it is clear that a binary code can always be considered as a binary anticode. The difference is that the notion of anticode captures the maximum distance of the code, which is closely related to the selection of codes with better bit flip fault resistance probability. Here, we rename our fault resilient encoding scheme below to emphasize the usage of anticode.

Remark 9. *Let C be an $(n, M, d, \delta)-$binary anticode, by Proposition 1, for $m < d$ and $m > \delta$, $p_{C,m} = 0$ with respect to $\mathcal{F}$ and $\varphi_C(g)$ in Section 5.1.*

Definition 19 (Fault Resilient Anticode Scheme)**.** *Given $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D) \in \mathfrak{S}$ a symmetric cipher and C an $(n, M = 2^k, d, \delta)-$binary anticode with an encoding-decoding scheme $(\texttt{Encoder}_C, \texttt{Decoder}_C)$. A cipher of the form $\Phi_C\big((\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)\big)$ is called a* fault resilient anticode scheme*.*

To analyze the choice of C that is used in a fault resilient anticode scheme, we theoretically study the performance of C with respect to $\mathcal{F}$ and $\varphi_C(g)$ in Section 5.1. Although the following theoretical analysis only analyzes a single operation of a cipher, we will see from the simulation results in Section 7.2 that it gives good insights on what anticode to choose for a full cipher implementation.

Next, we consider n, M as fixed parameters and we assume $\ell > 2$ (Equation 3), hence we also assume $n \ge 6$. For any $(n, M, d, \delta)-$binary code C, let $p_{C,\texttt{bf}}$ (resp. $p_{C,m}$) denote its bit flip fault resistance probability (resp. $m-$bit fault resistance probability) w.r.t. $\mathcal{F}$ and $\varphi_C(g)$ in Section 5.1.

We have the following observations.

Lemma 2 (Advantage of anticodes for fault detection)**.** *1. Let C_1 be an $(n, M, d_1, n)-$binary anticode, we have $p_{C_1,\texttt{bf}} \le 1 - \frac{1}{M}$.*
2. Let C_2 be an $(n, M, d_2, \delta_2)-$binary anticode such that $\delta_2 - d_2 \le 2$, we have $p_{C_2,\texttt{bf}} \le 1 - \frac{M-1}{6\binom{n}{\ell}}$.
3. Let C_3 be an $(n, M, d_3, \delta_3)-$binary anticode,
a. if $S_{m,C_1} < 2\binom{n}{m}$ $\forall 1 \le m \le n$, then $p_{C_3,\texttt{bf}} > p_{C_1,\texttt{bf}}$;
b. if $S_{m,C_3} < \frac{M(M-1)\binom{n}{m}}{3\binom{n}{\ell}}$ $\forall 1 \le m \le n$, then $p_{C_3,\texttt{bf}} > p_{C_2,\texttt{bf}}$.

(The proof can be found in Appendix B.3.)

We remark that in 3-a, taking $m = n$ implies $S_{n,C_3} = 0$, which means in this case $\delta_3 < n$. This corresponds to our previous observation that $S_{m,C} = 0$ for m "close" to n may give anticode with better fault resilient property. Condition 3-b implies that there are at least 3 m such that $S_{m,C_3} \ne 0$, which corresponds to our observation that it is not desirable to have $S_{m,C} = 0$ for too many m.

5.3 The Possible Choices of Anticodes

The next natural question would be: for what kind of parameters n, M, d, δ, there actually exists an $(n, M, d, \delta)-$binary anticode? We introduce the following notation.

$$N(n, d, \delta) := \max\{M : \exists (n, M, d, \delta) - \text{binary anticode}\}. \tag{5}$$

Two related well-studied coding theory concepts are [30, p.42]

$$A(n, d) := \max\{M : \exists (n, M, d) - \text{binary code}\},$$

and [31]

$$B(n, d) := \max\{M : \exists (n, M, d) - \text{binary code } C, \text{ s.t. } \forall \boldsymbol{c}, \boldsymbol{c}' \in C, \operatorname{dis}(\boldsymbol{c}, \boldsymbol{c}') = 0 \text{ or } d\}.$$

We have

Lemma 3. *i $N(n, d, d) = B(n, d)$;*
ii $N(n, d, n) \le A(n, d)$;
iii $N(n, d, \delta) \le N(n + 1, d, \delta)$;
iv $N(n, d, \delta) \le N(n + 1, d, \delta + 1)$, where $\delta \ge d + 1$;
v $N(n, d + 1, \delta) \le N(n, d, \delta)$, where $\delta > d + 1$;
vi $N(n, 2r - 1, 2\ell - 1) \le N(n + 1, 2r, 2\ell)$ where $r, \ell \in \mathbb{Z}_{>0}$;
vii $N(n, 2r - 1, 2\ell) \le N(n + 1, 2r, 2\ell)$, where $r, \ell \in \mathbb{Z}_{>0}$;

(The proof can be found in Appendix B.4.)

In Section 7.1 we will study and analyze the implementation of a fault resilient anticode scheme with PRESENT cipher. Because of the cipher design we will be interested in anticodes with cardinality 16 (see Section 7.1).

By the above Lemma, we computed the possible values of d and δ for $n = 8, 9, 10$ and $M = 16$, stated in Table 2. These values are useful when considering the selection of anticodes (see Section 8). On the other hand, the existence of binary anticodes

TABLE 2: Possible values of d, δ such that there exists an $(n, 16, d, \delta)-$binary anticodes for $n = 8, 9, 10$.

n	d	δ
8	2	$4, 5, 6, 7, 8$
8	3	$6, 7, 8$
8	4	8
9	2	$4, 5, 6, 7, 8, 9$
9	3	$6, 7, 8, 9$
9	4	$6, 8, 9$
10	2	$4, 5, 6, 7, 8, 9, 10$
10	3	$6, 7, 8, 9, 10$
10	4	$6, 7, 8, 9, 10$

satisfying condition 3-a or 3-b in Lemma 2 is not guaranteed. However, by using our anticode selection algorithm (Section 6.1), we were able to find anticodes satisfying both conditions 3-a and 3-b in Lemma 2. As expected, they have high bit flip fault resistance probability when used in fault resilient anticode scheme (cf. Remark 10).

We would like to emphasize that searching for an anticode with Algorithm 1 is time-consuming, especially for codes with high n. Also, it might not be apparent whether an anticode exists until the whole code space is searched. Therefore, Lemma 3 helps in this direction – it tells us whether it makes sense to run Algorithm 1 for given parameters.

6 Algorithms

In this section, we provide two useful algorithms for practical evaluation of encoding schemes. The first one selects binary anticodes according to user requirements and the second one evaluates software implementations that follow the fault resilient anticode scheme.

6.1 Anticode Selection Algorithm

In order to use and analyze the fault resilient anticode scheme, we first need to select the binary anticodes. The algorithm created for this purpose is described in this section.

Similarly to previous section, we choose the anticodes based on their performance on a single cipher operation. This gives a good approximation of an overall resistance when it is used for a full cipher implementation.

Pseudocode outlining the main idea of the anticode selection is stated in Algorithm 1. The inputs are: parameters n, M, d, δ for the binary anticode, and ε such that we require that the selected binary anticode C satisfies $1 - p_{C,m} < \varepsilon$ for all $1 \leq m \leq n$, where $p_{C,m}$ is the $m-$bit fault resistance probability of C with respect to $\mathcal{F}$ and $\varphi_C(g)$ in Section 5.1. Thus the calculation of $p_{C,m}$ follows from Proposition 1.

We note that for our implementation we use zero word as $\perp$ and thus in line 3 we choose sets S which do not contain $\mathbf{0}$.

The algorithm takes each possible binary code S that consists of M codewords, each of length n (line 3), and test if the distance conditions are satisfied (line 4). I.e. whether the following two conditions are satisfied: 1) $\min\{\text{dis}(\boldsymbol{c}, \boldsymbol{c}') : \boldsymbol{c}, \boldsymbol{c}' \in S, \boldsymbol{c} \neq \boldsymbol{c}'\} = d$; 2) $\max\{\text{dis}(\boldsymbol{c}, \boldsymbol{c}') : \boldsymbol{c}, \boldsymbol{c}' \in S, \boldsymbol{c} \neq \boldsymbol{c}'\} = \delta$. In case the distance conditions are satisfied, we further check if the fault resistance probability of S can be fulfilled (line 5).

The ε parameter is crucial for selecting an anticode with good fault resilient capabilities. As long as at least one anticode exists for given ε, the algorithm will try to lower this value (line 9) by a pre-specified constant, to find binary anticodes which satisfy the conditions with even smaller ε.

Algorithm 1: Anticode Selection Algorithm.

```
Input  : n : length of the anticode, M : number of
         codewords, d : minimum distance of the
         anticode, δ : maximum distance of the anticode,
         and ε : probability of missed faults.
Output: An (n, M, d, δ)−binary anticode C.
1  do
2      boolean codeExists := false;
3      for Every set S of M words which does not include ⊥
         do
4          if S is an (n, M, d, δ)−binary anticode then
5              if 1 − p_{C,m} < ε∀1 ≤ m ≤ n then
6                  codeExists := true;
7                  C := S;
8                  break for;
9      ε := ε − const;
10 while codeExists;
11 return C.
```

6.2 Dynamic Code Analysis

For the purpose of fault analysis, we have designed a dynamic code analyzer that is able to simulate the code execution and fault injection with a bit precision in any instruction of the code. Along with the bit flips, it can simulate instruction skips (see Definition 2). Pseudocode implementing the evaluation is stated in Algorithm 2.

For a symmetric cipher $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)$, and an $(n, M, d, \delta)-$binary anticode C, let $(\mathcal{K}, \mathcal{P}, \mathcal{M} \cup \{\perp\}, E', D')$ denote the corresponding fault resilient anticode scheme (Definition 19). Given $\mathcal{F}$, an implementation of E', Algorithm 2 calculates approximations of the $m-$bit fault resistance probability $p_{C,m}$ (Definition 14), bit flip fault resistance probability $p_{C,\mathtt{bf}}$ (Definition 15) and instruction skip resistance probability $p_{C,\mathtt{sk}}$ (Definition 16) of C with respect to $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)$ and $\mathcal{F}$.

By definition, the values of $p_{C,m}, p_{C,\mathtt{bf}}, p_{C,\mathtt{sk}}$ should be calculated by evaluating each pair of plaintext and secret key. However, for a symmetric cipher, this would require an infeasible amount of calculations. For PRESENT-80, it would need 2^{144} evaluations of each fault model (80-bit key and 64-bit plaintext). Thus, we allow a user input `noOfIter` which specifies how many pairs of random plaintext and random secret key to consider. Hence the output will be approximations of our evaluation metrics.

We first select a random pair of plaintext and secret key, then compute the corresponding correct ciphertext (line 3).

From line 4 to 13, we evaluate bit flip faults for the selected pair of plaintext and key. The first loop iterates over every possible fault mask, which will be later xor-ed with the intermediate value in order to change the original value in the destination register of an instruction (line 9). According to Definition 2, fault mask is a binary string, however, it is more convenient and efficient to use an integer in the implementation. The second loop iterates over every instruction in $\mathcal{F}$, to select the position in the program to be faulted. The last loop is the program execution itself, it iterates over instructions in $\mathcal{F}$ and executes them one by one. In case the instruction number corresponds to the number that is currently being targeted, a bit-flip is performed (line 9). After the execution of $\mathcal{F}$ finishes, there is a checking of the output value (lines 10-13). If the value equals to the expected ciphertext $E(P,\kappa)$, or the value is $\perp$, it is a *safe* fault. Otherwise, it is a *missed* fault (see Definition 13). In each case we increment a corresponding value in the array, where the array index indicates the Hamming weight of the fault mask.

Lines 14-23 evaluate instruction skips. It works in the same fashion as the previous part, however, in this case we save one loop because we do not need a fault mask. Output evaluation is analogous, but the records of safe/missed faults will be integers instead of array of integers.

Lines 24-25 calculate the approximated values of $p_{C,m}$ for each m, which is equal to the number of safe $m-$bit flip faults divided by the total number of $m-$bit flip faults considered. Line 26 calculates the approximated value of $p_{C,\mathtt{bf}}$, which is the minimum of $p_{C,m}$ for all m. Line 27 calculates the approximated value of $p_{C,\mathtt{sk}}$, which is equal to the number of safe instruction skips divided by the total number of instruction skip faults considered.

The time complexity of lines $4-13$ is $O(N_{\mathcal{F}}(2^n-1))$, where $N_{\mathcal{F}} = |\mathcal{F}|$, since the algorithm needs to evaluate every possible fault mask on every instruction of the code. The time complexity of lines $14-23$ is $O(N_{\mathcal{F}})$ because in this case, the total time depends only on the number of instructions. To give an overview, for 8-bit microcontroller implementation of PRESENT-80, time required to analyze the assembly code is ≈ 610 seconds.

Algorithm 2: Fault simulation algorithm.

```
   Input  : noOfIter:number of random plaintexts and key
            pairs to compute, $C$: $(n, M, d, \delta)-$binary anticode,
            $\mathcal{F}$: sequence of assembly instructions
            implementing $\varphi_C(E)$, $\mathcal{P}$: plaintext space, $\mathcal{K}$: key
            space.
   Output: SafeBitFlip (SafeBitFlip[$m$]= $p_{C,m}$); $p_{C,\mathtt{bf}}$; $p_{C,\mathtt{sk}}$.
 1 for Int k: 1 to noOfIter do
 2     Take random $P \in \mathcal{P}$, random $\kappa \in \mathcal{K}$;
 3     Compute the corresponding ciphertext $E(P,\kappa)$;
 4     for Fault mask Int j: 1 to $2^n$ do
 5         for Int i: 1 to $|\mathcal{F}|$ do
 6             for Instruction f in $\mathcal{F}$ do
 7                 Execute instruction $f$;
 8                 if f is the ith instruction and f has a
                     destination register then
 9                     $r_f = r_f \oplus$ j;
10             if output == $\perp$ or output == $E(P,\kappa)$ then
11                 SafeBitFlip[HammingWeight(j)]++;
12             else
13                 MissedBitFlip[HammingWeight(j)]++;
14     for Int i: 1 to $|\mathcal{F}|$ do
15         for Instruction f in $\mathcal{F}$ do
16             if f is the ith instruction then
17                 continue;
18             else
19                 Execute instruction $f$;
20         if output == $\perp$ or output == $E(P,\kappa)$ then
21             SafeSkip++;
22         else
23             MissedSkip++;
24 for Int m: 1 to n do
25     SafeBitFlip[m] = SafeBitFlip[m] / (SafeBitFlip[m] +
         MissedBitFlip[m]);
26 $p_{C,\mathtt{bf}} = \min_m$ SafeBitFlip[m];
27 $p_{C,\mathtt{sk}}$ = SafeSkip / (SafeSkip + MissedSkip);
28 return SafeBitFlip, $p_{C,\mathtt{bf}}$, $p_{C,\mathtt{sk}}$.
```

7 Case Study

In this section, we present the case study on block cipher PRESENT, fully implemented by using fault resilient anticode scheme with $(n, 16, d, \delta)-$binary anticodes for $n = 8, 9, 10$ (Table D lists all the anticodes used). The anticodes are selected by Algorithm 1. In Section 7.1, we provide implementation details by using a generic microcontroller. Section 7.2 provides the results of the code analysis using Algorithm 2.

7.1 PRESENT Cipher Implementation

PRESENT is an ultra-lightweight block cipher, developed in 2007 [32]. It is a symmetric cipher, following an SPN structure, where the block length is 64 bits and key length can be either 128 bits or 80 bits. A round function consists of three operations: *addRoundKey* (xor of the state with the round key), *sBoxLayer* (substitution by 4-bit SBox, which we refer to as PRESENT SBox), and *pLayer* (bitwise permutation). After 31 rounds, there is one more *addRoundKey*, used for post-whitening. The whole process is depicted in Figure 2. Because of its lightweight character, it is recommended to use 80-bit key length in order to keep the computation fast and energy efficient [32]. We will focus on this variant, denoted by PRESENT-80. For our implementation, we take pre-computed round keys which are already encoded and therefore, we omit the description of the key schedule here.

By definition, evaluations of $p_{C,m}, p_{C,\mathtt{bf}}, p_{C,\mathtt{sk}}$ are done on an assembly implementation, thus it is important to specify what kind of implementation is used. The main properties of the implementation in our case study are as follows:

1) Each operation is implemented as a table look-up from memory.
2) Before the table look-up, the destination register of an operation is precharged to a zero so that single instruction skip will be protected.
3) The error message $\perp$ is denoted by the value zero **0**.

We note that for PRESENT-80, *pLayer* can be considered as four parallel bitwise operations where each is a function: $\mathbb{F}_2^{16} \to \mathbb{F}_2^{16}$. *sBoxLayer* is 16 parallel Sbox substitution operation: $\mathbb{F}_2^4 \to \mathbb{F}_2^4$. *addRoundKey*: $\mathbb{F}_2^{64} \times \mathbb{F}_2^{64} \to \mathbb{F}_2^{64}$ is a bitwise operation. Furthermore, 64 and 16 are multiples of 4. Thus we can use code with cardinality $2^4 = 16$. In particular, to apply fault resilient anticode scheme with PRESENT-80, we use $(n, 16, d, \delta)-$binary anticodes.

Figure 3 shows one round of PRESENT and gives an overview of how the *sBoxLayer* and *pLayer* work. There are 4 groups of Sboxes in the *sBoxLayer*, indicated by different colors. Output bits from each group serve as inputs to 4 distinct Sboxes in the

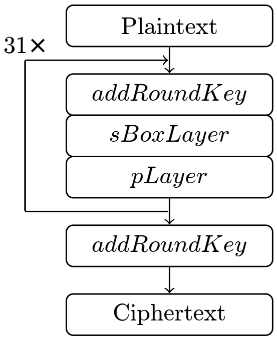


Fig. 2: Sequence of operations of PRESENT block cipher.

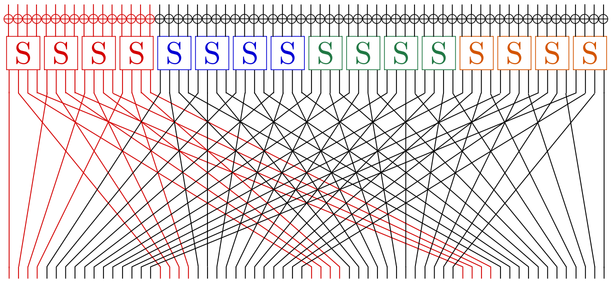


Fig. 3: One round of PRESENT.

subsequent round, thanks to the state-wise diffusion function. As illustrated in the figure, outputs of Sboxes $0, 1, 2, 3$ denoted by red color, will be inputs of Sboxes $0, 4, 8, 12$ in the next round. This property helps us to tailor the look up tables in a way that can provide more efficient space/time implementation compared to implementing the two layers separately. In the following, we will explain the design of such an implementation .

Encoded Round Function for PRESENT

In this part, we will explain the implementation of the round functions for fault resilient anticode scheme with PRESENT-80 by using $(n, 16, d, \delta)-$ binary anticodes. Remark 3 justifies that we can split or merge multiple cipher operations while using the fault resilient $C-$map (Definition 10), preserving the correct data-flow.

The *addRoundKey* is a binary operation, xor-ing the key with the current state. Therefore, it can be directly implemented by an xor lookup table, similar to the implementation $\mathcal{F}$ of $\varphi_C(g)$ in Section 5.1. The *sBoxLayer* maps an input value to an output value, therefore the standalone implementation would be even easier than the xor. However, we have decided to merge *sBoxLayer* together with the *pLayer*, because the latter cannot be implemented in a straightforward way. The overview of this merged implementation is depicted in Figure 4, which explains how the first encoded nibble is obtained. The explanation of this approach is given below.

Let C be an $(n, 16, d, \delta)-$binary anticode. The implementation of $\Phi_C\big(\text{pLayer}\circ\text{sBoxLayer}\big)$ relies on the xor lookup table and eight other tables, which can be divided into two groups:

1) **Bit-extracting Sbox tables:** This group has four tables: $T0$, $T1$, $T2$, $T3$ such that Ti takes a codeword, say $\mathtt{Encoder}_C(x_0x_1x_2x_3)$ and returns the codeword $\mathtt{Encoder}_C(xs_i000)$. If the input is not a codeword, the return value will be $\perp$. Here we assume that after PRESENT SBox, $x_0x_1x_2x_3$ becomes $xs_0xs_1xs_2xs_3$.
   In other words, this group first computes an Sbox on the encoded data, and then extracts one bit – the bit position depends on which of the four tables is used. So, the output of these tables is the codeword corresponding to either 0 or 8.
2) **Bit-shifting tables:** This group has four tables as well: $TB0$, $TB1$, $TB2$ and $TB3$. For a codeword of the form $\mathtt{Encoder}_C(x000)$, $TB0$, $TB1$, $TB2$, $TB3$ return the codewords $\mathtt{Encoder}_C(x000)$, $\mathtt{Encoder}_C(0x00)$, $\mathtt{Encoder}_C(00x0)$, $\mathtt{Encoder}_C(000x)$, in their respective order. If the input is not a codeword, the return value will be $\perp$ for all the four tables. In other words, the tables in this group provide bit shifting operations, that are necessary to finalize the *pLayer*. The outputs of tables $TB0$, $TB1$, $TB2$, $TB3$ can be codewords corresponding to 8, 4, 2, 1 or 0, depending on the value and the bit position.

After the Sbox is computed and the bit shifts on the resulting data are done, the data is combined back to 4-bit format by using an xor table – in total, three xor operations are required to combine the data. In the following, we will explain this process step-by-step.

Assume we have $\mathtt{Encoder}_C\big(a_0a_1a_2a_3b_0b_1b_2b_3c_0c_1c_2c_3d_0d_1d_2d_3\big)$, representing a cipher state, where each letter represents one nibble of information. This is what happens:

1) $\mathtt{Encoder}_C(a_0a_1a_2a_3)$ is passed to tables $T0$, $T1$, $T2$, $T3$, the four returned values are passed to $TB0$ and we get:
   $\mathtt{Encoder}_C(as_0000)$, $\mathtt{Encoder}_C(as_1000)$, $\mathtt{Encoder}_C(as_2000)$, $\mathtt{Encoder}_C(as_3000)$;
2) $\mathtt{Encoder}_C(b_0b_1b_2b_3)$ is passed to tables $T0$, $T1$, $T2$, $T3$, the four returned values are passed to $TB1$ and we get:
   $\mathtt{Encoder}_C(0bs_000)$, $\mathtt{Encoder}_C(0bs_100)$, $\mathtt{Encoder}_C(0bs_200)$, $\mathtt{Encoder}_C(0bs_300)$;
3) $\mathtt{Encoder}_C(c_0c_1c_2c_3)$ is passed to tables $T0$, $T1$, $T2$, $T3$, the four returned values are passed to $TB2$ and we get:
   $\mathtt{Encoder}_C(00cs_00)$, $\mathtt{Encoder}_C(00cs_10)$, $\mathtt{Encoder}_C(00cs_20)$, $\mathtt{Encoder}_C(00cs_30)$;
4) $\mathtt{Encoder}_C(d_0d_1d_2d_3)$ is passed to tables $T0$, $T1$, $T2$, $T3$, the four returned values are passed to $TB3$ and we get:
   $\mathtt{Encoder}_C(000ds_0)$, $\mathtt{Encoder}_C(000ds_1)$, $\mathtt{Encoder}_C(000ds_2)$, $\mathtt{Encoder}_C(000ds_3)$.

Afterwards, we need three xor table lookups:

1) The first four encoded nibbles are given by
   $\big(\mathtt{Encoder}_C(as_0000)\widetilde{\oplus}\mathtt{Encoder}_C(0bs_000)\big)\widetilde{\oplus}\big(\mathtt{Encoder}_C(00cs_00)$
   $\widetilde{\oplus}\mathtt{Encoder}_C(000ds_0)\big)$;
2) The second four encoded nibbles are given by
   $\big(\mathtt{Encoder}_C(as_1000)\widetilde{\oplus}\mathtt{Encoder}_C(0bs_100)\big)\widetilde{\oplus}\big(\mathtt{Encoder}_C(00cs_10)$
   $\widetilde{\oplus}\mathtt{Encoder}_C(000ds_1)\big)$;
3) The third four encoded nibbles are given by
   $\big(\mathtt{Encoder}_C(as_2000)\widetilde{\oplus}\mathtt{Encoder}_C(0bs_200)\big)\widetilde{\oplus}\big(\mathtt{Encoder}_C(00cs_20)$
   $\widetilde{\oplus}\mathtt{Encoder}_C(000ds_2)\big)$;
4) The fourth four encoded nibbles are given by
   $\big(\mathtt{Encoder}_C(as_3000)\widetilde{\oplus}\mathtt{Encoder}_C(0bs_300)\big)\widetilde{\oplus}\big(\mathtt{Encoder}_C(00cs_30)$
   $\widetilde{\oplus}\mathtt{Encoder}_C(000ds_3)\big)$;

Here $\widetilde{\oplus}$ represents xor table lookup.

7.2 Results

For selecting anticodes, we run Algorithm 1 for all the parameters n, M, d, δ combination in Table 2. For each (n, M, d, δ), ε was set to 1 and the anticodes selected are presented in Table 4.

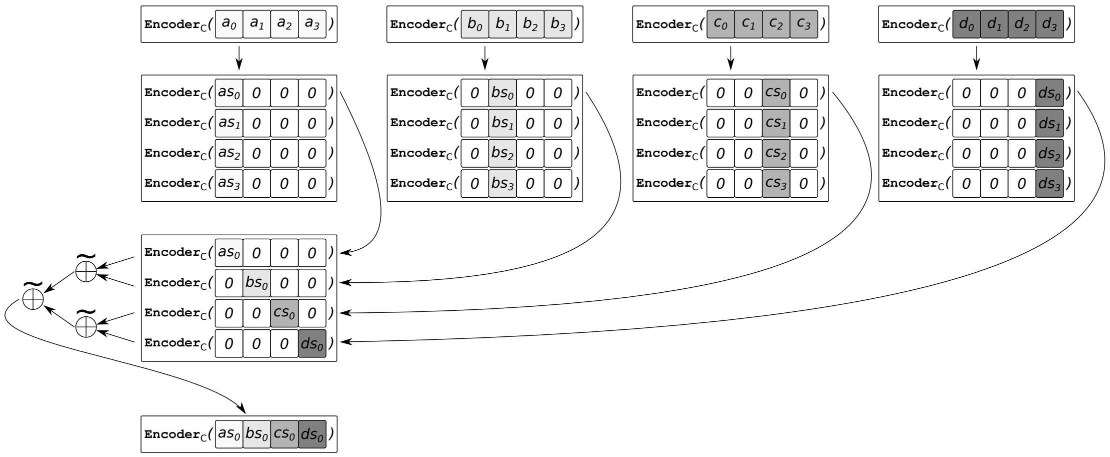

Encoder$_C$(a_0 a_1 a_2 a_3)
Encoder$_C$(b_0 b_1 b_2 b_3)
Encoder$_C$(c_0 c_1 c_2 c_3)
Encoder$_C$(d_0 d_1 d_2 d_3)
Encoder$_C$(as_0 0 0 0)
Encoder$_C$(as_1 0 0 0)
Encoder$_C$(as_2 0 0 0)
Encoder$_C$(as_3 0 0 0)
Encoder$_C$(0 bs_0 0 0)
Encoder$_C$(0 bs_1 0 0)
Encoder$_C$(0 bs_2 0 0)
Encoder$_C$(0 bs_3 0 0)
Encoder$_C$(0 0 cs_0 0)
Encoder$_C$(0 0 cs_1 0)
Encoder$_C$(0 0 cs_2 0)
Encoder$_C$(0 0 cs_3 0)
Encoder$_C$(0 0 0 ds_0)
Encoder$_C$(0 0 0 ds_1)
Encoder$_C$(0 0 0 ds_2)
Encoder$_C$(0 0 0 ds_3)
$\tilde{\oplus}$
Encoder$_C$(as_0 bs_0 cs_0 ds_0)

Fig. 4: Illustration of the implementation of PRESENT-80 sBoxLayer and pLayer in fault resilient anticode scheme.

To analyze the performance of different anticodes in fault resilient anticode scheme, for each anticode in Table 4, we run Algorithm 2 using implementation of PRESENT-80 following the specification stated in Section 7.1. To decide the input `noOfInter`, we randomly picked 10 anticodes and executed the algorithm with different values of `noOfInter`. Our results showed that for `noOfInter`$\geq$200, the change in the output probabilities for different values of `noOfInter` became negligible ($< 10^{-6}$). Therefore, we have set `noOfInter`$=$ 200 for our evaluation of each anticode.

The analysis results for anticodes (Table 4) with $n = 8, d = 2, 3$, and $n = 10, d = 2$ with various δ values are stated in Figures 5 and 6, respectively. Additional results for $n = 8, d = 2, 3$, $n = 9, d = 2, 3, 4$, and $n = 10, d = 3, 4$ are stated in Appendix C. We have the following observations.

1) The instruction skip resistance probability is 1 for all anticodes. This is due to the precharge of destination register of our implementation specification.
2) The improvement of using a longer length for encoding the data is obvious – bit flip fault resistance probabilities faults for length 8 go up to ≈0.933, for length 9 up to ≈0.966, and for length 10 up to ≈0.979.
3) For $n = 8, 9, 10$ the anticode with the best performance (i.e. the highest bit flip fault resistance probability $p_{C,\mathtt{bf}}$) are anticodes with parameters $(8, 16, 3, 6), (9, 16, 3, 7), (10, 16, 3, 8)$ respectively.
4) Every $(8, 16, 4, 8)-$binary anticode has a property that $8-$bit flip has a probability 1 of being *missed*, i.e. $p_{C,8} = 0$. We note that this finding is in accordance with the one described in [18].

Remark 10.

- *The anticodes that achieve the best bit flip fault resistance probabilities satisfy both conditions 3-a and 3-b in Lemma 2.*
- *Comparing the last two columns of Table 4, we can see the theoretical analysis of one operation does give insights on what kind of parameters to look for.*
- *However, the theoretical analysis results differ from the simulated probabilities for most of the codes, showing that the analysis of one code snippet cannot capture what happens when a full cipher implementation is considered. To be more specific, the implemented tables can provide an output which follows a non-uniform distribution over the codewords, such as bit-extracting and bit-shifting tables in the case of the PRESENT-80 implementation detailed in Section 7.1. Therefore, it shows the importance of simulating the cryptographic implementation execution for getting more precise insights on the code robustness.*

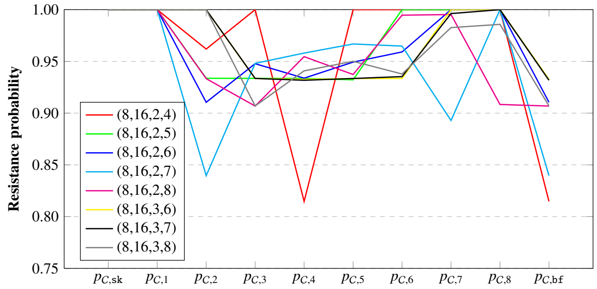


Fig. 5: Simulated results for anticodes with $n = 8$, $d = 2, 3$.

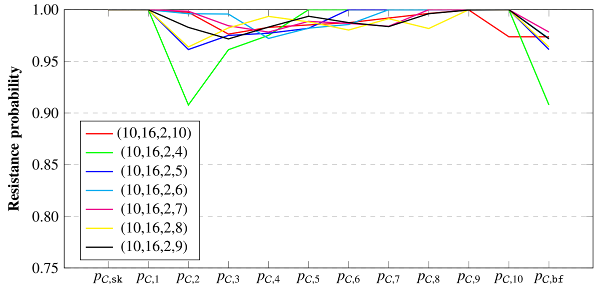


Fig. 6: Simulated results for anticodes with $n = 10$, $d = 2$.

8 Selection of Anticode Parameters

Now a natural question to ask is how to choose the parameters for anticodes in general, e.g. for different device architectures and security requirements. We propose the following guidelines:

1) **Code length** (n)**:** This parameter depends entirely on the underlying device architecture. Because of the addressing in the table look-up implementations, it is necessary to fit the whole address into one instruction. Therefore, e.g. for 8-bit device, one can use at most $n = 8$. For 16- or 32-bit architectures, greater lengths can be used. However, in that case, memory requirements need to be taken into account (these are explained more in Section 9).

2) **Number of codewords (M):** Number of codewords is, on the other hand, independent on the underlying architecture – it does not affect table size or require specific register size. The designer needs to take the cipher and the security requirements into account when deciding on number of codewords. For example, in case of PRESENT, the operations are computed on nibbles and therefore, 16 codewords is the preferred number, providing a good trade-off between security and execution speed. Lower number of codewords would mean higher security, but slower speed since the operations need to be carried on smaller chunks of data.
3) **Distance (d) and maximum distance (δ):** These parameters are not dependent on the device architecture, but can affect the resulting security significantly.

   The first, and obvious selection criterion is whether a code with certain d, δ exists for some n and M. For this purpose, Lemma 3 provides an answer, with results for $n = 8, 9, 10$ stated in Table 2.

   Another selection criterion is whether some particular fault models can be prevented by other means – suppose we have an additional error detecting module that can detect 2 or 4 bit-flips. Then, we can use (8, 16, 2, 4)-anticode from Figure 5, since only these two models are undetected using this code.

   Furthermore, the selection is also dependent on the attacker model assumption. In case we want to build an implementation resistant against specific fault model, e.g. nibble flip [33], then we would like to select an anticode with the highest $4-$bit flip fault resistance probability, $p_{C,4}$.

   For any case, while the values of n, M can be decided before the actual cipher implementation, d and δ should be decided after running the evaluation in Section 6.2.

9 Discussion

Memory and Speed Trade-Offs

Table 4 shows that if the anticode C has longer length, fault resilient anticode scheme using C has better fault resistance properties. On the other hand, it also means a bigger memory consumption that increases sub-exponentially with the code length. In the following, we will discuss the overheads.

When it comes to speed, the fastest non-bit sliced 8-bit implementation of PRESENT-80 requires 8,721 clock cycles [34], out of which $\approx$ 1,248 is a key schedule (since we consider the round keys already in the memory, we will only count 7,473 clock cycles for the implementation from [34]). In case the selected code can be fully implemented in the SRAM (and therefore, a table look-up operation `LD` takes 2 clock cycles), fault resilient anticode scheme implementation takes 9,424 clock cycles ($\approx$ 26.1% overhead). In case all the look-up tables are stored in the flash memory (`LMP` instruction taking 3 clock cycles has to be used), the approach takes 13,640 clock cycles ($\approx$ 82.5% overhead). Therefore, compared to the most popular time redundancy that repeats the encryption twice and compares the results [3], the encoding method provides reasonable timing overheads, especially if the look-up tables can be stored in the SRAM.

While the speed of the implementation might be reasonable, the memory overheads quickly grow to sizes that are not practical for real-world cryptography. It has to be noted that even if the code length is smaller than the memory address length, the table normally has to occupy the size according to this length, otherwise the unused bits in the address could be faulted and would point to another part of the memory that is used for a different purpose. Therefore, if we want to use a binary anticode of length 6 in a 16-bit addressing space device, the constructed table still has to be of size of 8×8 bits. For such architecture, codes longer than 8 bits would not be possible – in case of code length is between 9 – 16, we need a 32-bit addressing space. Also, number of codewords does not affect the memory requirements since the table size for the same code length is constant, only the number of non-zero values will change with different number of codewords. Efficient implementation of encoding schemes therefore still remains an open problem.

Table 3 provides memory requirements for some standard cryptographic operations. Since block ciphers combine several functions in order to achieve the security requirements for confusion and diffusion, several tables normally have to be stored in the memory. For example, the PRESENT implementation in Section 7 uses one xor table and eight shifting tables for the combined `pLayer` and `sBoxLayer`, resulting in total of 81,920 bytes of memory. To test the feasibility, we made an implementation for Atmel ATmega328P, an 8-bit microcontroller. However, only the eight smaller tables could fit into the device memory, while the big xor table had to be put on an external EEPROM module (256 Kbit Microchip 24LC256).

TABLE 3: Overheads for implementing fault resilient encoding scheme.

Operation Type	Code Length	Required Memory (B)
Unary (Sbox, shifts)	≤ 8	2,048
	≤ 16	524,288
Binary (XOR, AND, modular addition)	≤ 8	65,536
	≤ 16	33,554,432

Instruction Modification

Recently, a fault attack approach utilizing instruction replacement has emerged [35]. Up to date, there is no dedicated protection against such fault model. In [35], the attacker has to change the instruction opcode that specifies the operation – e.g. in case of changing `ADD` to `SUB` in AVR, as presented in [35], the instruction opcode needs to be changed from `000011` to `000110`. In case the standard instructions are used, if the attacker is able to achieve this model on a particular device, she can do that for any implementation executed on such device. However, in the case of the table look-up based fault resilient anticode scheme, the means to achieve the instruction replacement that result to executing another operation are different. Each operation is executed as fetching the result from a table and hence the address of this table specifies this operation. Instead of changing the instruction opcode, the attacker needs to know the table address she wants to change the operation to. Such address would vary from implementation to implementation and it is not trivial to predict whether the attacker would be able to achieve such precise change.

Cache Timing Attacks

Look-up tables in general are susceptible to cache timing attacks, since fetching a value from one position in the table takes a different time compared to using another position due to cache misses [36]. As mentioned in [37], there are various ways for protecting such implementations. One way to do it is to use two different round function implementations – some rounds use look-up tables, while the others do not. This method can be further investigated in order to provide the best properties w.r.t. cache-timing, power, and fault attacks. Another approach is *cache warming* that loads the whole table into the cache, resulting into constant time of execution, avoiding cache misses completely. Furthemore, one can add random delays in the execution to make the attack harder.

Other Fault Analysis Methods

Apart from the Differential Fault Analysis (DFA), there are several other methods that can be used by the attacker. There are methods that have similar requirements to DFA, such as Collision Fault Analysis or Algebraic Fault Analysis, where the knowledge of the fault propagation is necessary in order to get the secret information. Therefore, our scheme can be applied as a countermeasure for these methods as well.

On the other hand, there are approaches that utilize the behavior where the fault does not propagate in all the cases, such as Safe-Error Analysis or Ineffective Fault Analysis (recently utilized in [38]). These two methods, when used for block ciphers, require a stuck-at fault model, i.e. a model where certain value becomes either ‘0’ or ‘1’, no matter what value was in the register before. The attacker then just needs the information whether the output is faulty or not, without the knowledge of the fault value. Therefore, any error detection method that outputs $\perp$ reveals such information to the attacker. Even if it carries out the computation one more time and provides a correct output on the second run, there is already a timing difference that can be observed. However, these attacks can be thwarted by a well-designed error correction codes. Some results in this direction are stated in [18], along with the code properties. Similar properties could be derived for fault resilient anticode scheme in case such protection is necessary.

10 Conclusion

In this paper, we have formalized fault resilient anticode schemes and provided a way to evaluate software implementations protected by anticodes. We have practically implemented and evaluated symmetric block cipher PRESENT with encoded operations by using 8-bit microcontroller assembly code.

For the future work, we would like to extend our evaluation methodology to pipelined architectures.

Acknowledgement

This research was supported (in part) by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate.

References

[1] N. F. Galathy, B. Yuce, and P. Schaumont, “A Systematic Approach to Fault Attack Resistant Design,” in *Fundamentals of IP and SoC Security: Design, Verification, and Debug*, S. Bhunia, S. Ray, and S. Sur-Kolay, Eds. Springer International Publishing, 2017.

[2] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice*, 2nd ed., J. C. Laprie, A. Avizienis, and H. Kopetz, Eds. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1990.

[3] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, “The Sorcerer’s Apprentice Guide to Fault Attacks,” *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, Feb 2006.

[4] K. D. Akdemir, Z. Wang, M. Karpovsky, and B. Sunar, “Design of cryptographic devices resilient to fault injection attacks using nonlinear robust codes,” in *Fault Analysis in Cryptography*. Springer, 2012, pp. 171–199.

[5] T. Schneider, A. Moradi, and T. Güneysu, “ParTI – Towards Combined Hardware Countermeasures Against Side-Channel and Fault-Injection Attacks,” in *Advances in Cryptology – CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, M. Robshaw and J. Katz, Eds. Springer Berlin Heidelberg, 2016, pp. 302–332.

[6] L. Breveglieri, I. Koren, and P. Maistri, “An operation-centered approach to fault detection in symmetric cryptography ciphers,” *IEEE Transactions on Computers*, vol. 56, no. 5, pp. 635–649, 2007.

[7] E. Biham and A. Shamir, “Differential Fault Analysis of Secret Key Cryptosystems,” in *Advances in Cryptology — CRYPTO ’97: 17th Annual International Cryptology Conference Santa Barbara, California, USA August 17–21, 1997 Proceedings*, B. S. Kaliski, Ed. Springer Berlin Heidelberg, 1997, pp. 513–525.

[8] M. Tunstall, D. Mukhopadhyay, and S. Ali, “Differential fault analysis of the advanced encryption standard using a single fault,” in *IFIP international workshop on information security theory and practices*. Springer, 2011, pp. 224–233.

[9] K. Jeong and C. Lee, “Differential fault analysis on block cipher LED-64,” in *Future Information Technology, Application, and Service*. Springer, 2012, pp. 747–755.

[10] H. Tupsamudre, S. Bisht, and D. Mukhopadhyay, “Differential fault analysis on the families of SIMON and SPECK ciphers,” in *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2014, pp. 40–48.

[11] G. Wang and S. Wang, “Differential fault analysis on PRESENT key schedule,” in *Computational Intelligence and Security (CIS), 2010 International Conference on*. IEEE, 2010, pp. 362–366.

[12] B. Yuce, N. F. Ghalaty, H. Santapuri, C. Deshpande, C. Patrick, and P. Schaumont, “Software Fault Resistance is Futile: Effective Single-Glitch Attacks,” in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Aug 2016, pp. 47–58.

[13] S. Patranabis, A. Chakraborty, and D. Mukhopadhyay, “Fault Tolerant Infective Countermeasure for AES,” in *Security, Privacy, and Applied Cryptography Engineering: 5th International Conference, SPACE 2015, Jaipur, India, October 3-7, 2015, Proceedings*, R. S. Chakraborty, P. Schwabe, and J. Solworth, Eds. Springer International Publishing, 2015, pp. 190–209.

[14] C. Patrick, B. Yuce, N. F. Ghalaty, and P. Schaumont, “Lightweight Fault Attack Resistance in Software Using Intra-Instruction Redundancy,” Cryptology ePrint Archive, Report 2016/850, 2016, http://eprint.iacr.org/2016/850.

[15] J.-M. Schmidt and M. Medwed, “Countermeasures for Symmetric Key Ciphers,” in *Fault Analysis in Cryptography*, M. Joye and M. Tunstall, Eds. Springer Berlin Heidelberg, 2012, pp. 73–87.

[16] M. Ciet and M. Joye, “Practical Fault Countermeasures for Chinese Remaindering Based RSA (Extended Abstract),” in *In Proceedings of Workshop on Fault Detection and Tolerance in Cryptography (FDTC’05)*, 2005, pp. 124–131.

[17] J. Breier, D. Jap, and S. Bhasin, “A study on analyzing side-channel resistant encoding schemes with respect to fault attacks,” *Journal of Cryptographic Engineering*, Jun 2017.

[18] J. Breier and X. Hou, “Feeding Two Cats with One Bowl: On Designing a Fault and Side-Channel Resistant Software Encoding Scheme,” in *Topics in Cryptology – CT-RSA 2017: The Cryptographers’ Track at the RSA Conference 2017, San Francisco, CA, USA, February 14–17, 2017, Proceedings*, H. Handschuh, Ed. Springer International Publishing, 2017, pp. 77–94.

[19] J. Bringer, C. Carlet, H. Chabanne, S. Guilley, and H. Maghrebi, “Orthogonal Direct Sum Masking,” in *Information Security Theory and Practice. Securing the Internet of Things: 8th IFIP WG 11.2 International Workshop, WISTP 2014, Heraklion, Crete, Greece, June 30 – July 2, 2014. Proceedings*, D. Naccache and D. Sauveron, Eds. Springer Berlin Heidelberg, 2014, pp. 40–56.

[20] V. Servant, N. Debande, H. Maghrebi, and J. Bringer, “Study of a Novel Software Constant Weight Implementation,” in *Smart Card Research and Advanced Applications: 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, M. Joye and A. Moradi, Eds. Springer International Publishing, 2015, pp. 35–48.

[21] N. Moro, K. Heydemann, A. Dehbaoui, B. Robisson, and E. Encrenaz, “Experimental evaluation of two software countermeasures against fault attacks,” in *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, May 2014, pp. 112–117.

[22] L. Goubet, K. Heydemann, E. Encrenaz, and R. De Keulenaer, “Efficient design and evaluation of countermeasures against fault attacks using

formal verification," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2015, pp. 177–192.

[23] S. Ling and C. Xing, *Coding Theory: A First Course*. Cambridge University Press, 2004.

[24] J. H. Conway and N. J. A. Sloane, *Sphere Packings, Lattices and Groups*. Springer Science & Business Media, 2013, vol. 290.

[25] J. Hoffstein, J. Pipher, J. H. Silverman, and J. H. Silverman, *An Introduction to Mathematical Cryptography*. Springer, 2008, vol. 1.

[26] P. Rauzy, S. Guilley, and Z. Najm, "Formally Proved Security of Assembly Code Against Leakage," *IACR Cryptology ePrint Archive*, vol. 2013, p. 554, 2013.

[27] S. Dziembowski, K. Pietrzak, and D. Wichs, "Non-Malleable Codes," in *ICS*, 2010, pp. 434–452.

[28] K. F. Riley, M. P. Hobson, and S. J. Bence, *Mathematical Methods for Physics and Engineering: A Comprehensive Guide*. Cambridge University Press, 2006.

[29] P. Farrell, "Linear Binary Anticodes," *Electronics Letters*, vol. 13, no. 6, pp. 419–421, 1970.

[30] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error Correcting Codes*. Elsevier, 1977.

[31] F.-W. Fu, T. Kløve, Y. Luo, and V. K. Wei, "On Equidistant Constant Weight Codes," *Discrete applied mathematics*, vol. 128, no. 1, pp. 157–164, 2003.

[32] A. Bogdanov, L. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An Ultra-Lightweight Block Cipher," in *Cryptographic Hardware and Embedded Systems - CHES 2007*, ser. Lecture Notes in Computer Science, P. Paillier and I. Verbauwhede, Eds. Springer Berlin Heidelberg, 2007, vol. 4727, pp. 450–466.

[33] J. Breier and W. He, "Multiple fault attack on PRESENT with a hardware trojan implementation in FPGA," *arXiv preprint arXiv:1702.08208*, 2017.

[34] K. Papagiannopoulos and A. Verstegen, "Speed and Size-Optimized Implementations of the PRESENT Cipher for Tiny AVR Devices," in *Radio Frequency Identification: Security and Privacy Issues 9th International Workshop, RFIDsec 2013, Graz, Austria, July 9-11, 2013, Revised Selected Papers*, M. Hutter and J.-M. Schmidt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 161–175.

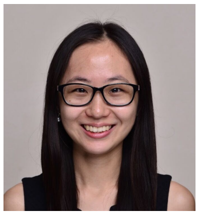
[35] S. D. Kumar, S. Patranabis, J. Breier, D. Mukhopadhyay, S. Bhasin, A. Chattopadhyay, and A. Baksi, "A practical fault attack on ARX-like ciphers with a case study on ChaCha20," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2017 Workshop on*. IEEE, 2017, pp. 33–40.

[36] D. J. Bernstein, "Cache-timing attacks on AES," Tech. Rep., 2005.

[37] D. Mukhopadhyay and R. S. Chakraborty, *Hardware Security: Design, Threats, and Safeguards*. CRC Press, 2014.

[38] C. Dobraunig, M. Eichlseder, T. Korak, S. Mangard, F. Mendel, and R. Primas, "Exploiting ineffective fault inductions on symmetric cryptography," Cryptology ePrint Archive, Report 2018/071, 2018, https://eprint.iacr.org/2018/071.

Jakub Breier is currently a Senior Cryptography Security Analyst at Underwriters Laboratories, Singapore. Before that, he was a researcher at PACE lab, Nanyang Technological University between 2013-2018. He received his PhD in Applied Informatics from Slovak University of Technology (STU), Slovakia in 2013, Master's in Information Technology Security from Masaryk University, Czech Republic in 2010, and Bachelor's in Informatics from STU, Slovakia in 2008. He was also a visiting researcher at Fraunhofer AISEC, Germany, in 2014. His research topics include fault and side-channel analysis methods and countermeasures, and advanced fault injection techniques.



Xiaolu Hou works as a Secure Computing Researcher at Acronis, Singapore since 2018, in the field of secure multi-party computation. She finished her PhD in Mathematics from Nanyang Technological University (NTU) in 2017. During her PhD studies, she was half year with Singapore University of Technology and Design, where she was doing research in location privacy. After her PhD she joined Cyber Security Laboratory, School of Computer Science and Engineering, NTU, as a Research Fellow. Her research focuses on fault injection and side-channel attacks. With a wide range of research interests, she has published her work at top venues within various fields.

Yang Liu graduated in 2005 with a Bachelor of Computing (Honours) in the National University of Singapore (NUS). In 2010, he obtained his PhD and started his post doctoral work in NUS, MIT and SUTD. In 2012 fall, he joined Nanyang Technological University (NTU) as a Nanyang Assistant Professor. He is currently an associate professor and Director of the cybersecurity lab in NTU. Dr. Liu specializes in software verification, security and software engineering. His research has bridged the gap between the theory and practical usage of formal methods and program analysis to evaluate the design and implementation of software for high assurance and security. By now, he has more than 200 publications in top tier conferences and journals. He has received a number of prestigious awards including MSRA Fellowship, TRF Fellowship, Nanyang Assistant Professor, Tan Chin Tuan Fellowship, and 8 best paper awards in top conferences like ASE, FSE and ICSE. He is leading a large research team working on the state-of-the-art software engineering and cybersecurity problems.

Appendix A
More Details on Encoding-Decoding Scheme (Definition 6)

In this section we elaborate more on encoding-decoding scheme from Definition 6.

For any $N \neq k$, we extend $\texttt{Encoder}_C$ and $\texttt{Decoder}_C$ to $\mathbb{F}_2^N$ as follows:

- If $k \nmid N$, take any $\boldsymbol{x} = (x_1, x_2, \dots, x_N) \in \mathbb{F}_2^N$, let $\boldsymbol{x}' = (x_1, x_2, \dots, x_N, \quad 0, \dots, 0) \in \mathbb{F}_2^{N+N'}$, where $N' = \min\{\ell : k|(N+\ell)\}$. i.e. we add zero bits to $\boldsymbol{x}$ to get $\boldsymbol{x}'$ so that the length of $\boldsymbol{x}'$ is divisible by k. Let $\texttt{Encoder}_C(\boldsymbol{x}) := \texttt{Encoder}_C(\boldsymbol{x}')$.

- If $k|N$, say $N = kk'$, for any $\boldsymbol{x} = (x_1, x_2, \dots, x_N) \in \mathbb{F}_2^N$, let $\boldsymbol{x}_i = (x_{ik+1}, x_{ik+2}, \dots, x_{ik+k}), 0 \leq i \leq k'-1$ and define
$$\texttt{Encoder}_C(\boldsymbol{x}) := \Big(\texttt{Encoder}_C(\boldsymbol{x}_0), \dots, \texttt{Encoder}_C(\boldsymbol{x}_{k'-1})\Big) \in C^{k'}.$$
It follows that $\texttt{Encoder}_C : \mathbb{F}_2^N \rightarrow C^{k'}$ is a bijective function. We define $\texttt{Decoder}_C : \mathbb{F}_2^{nk'} \cup \{\perp\} \rightarrow \mathbb{F}_2^N$ such that $\texttt{Decoder}_C\Big|_{C^{k'}} \rightarrow \mathbb{F}_2^N$ is the inverse of $\texttt{Encoder}_C$ and $\texttt{Decoder}_C\Big|_{(\mathbb{F}_2^{nk'} \cup \{\perp\}) \backslash C^{k'}} = \{\perp\}$.

Example. Let us consider a $(2,2,2)-$binary code $C = \{00, 11\}$ with associated encoding-decoding scheme as follows:

$$\texttt{Encoder}_C : \quad 0 \mapsto 00 \;\; 1 \mapsto 11$$
$$\texttt{Decoder}_C : \quad 00 \mapsto 0 \; 01 \mapsto \perp \;\; 10 \mapsto \perp \;\; 11 \mapsto 1.$$

Extend $\texttt{Encoder}_C$, $\texttt{Decoder}_C$ to $\mathbb{F}_2^2$, we have the following encoding-decoding scheme:

$\boldsymbol{x}$	$\texttt{Encoder}_C(\boldsymbol{x})$
00	0000
01	0011
10	1100
11	1111

$\boldsymbol{x}$	$\texttt{Decoder}_C(\boldsymbol{x})$
0000	00
0001	$\perp$
0010	$\perp$
0011	01
0100	$\perp$
0101	$\perp$
0110	$\perp$
0111	$\perp$

$\boldsymbol{x}$	$\texttt{Decoder}_C(\boldsymbol{x})$
1000	$\perp$
1001	$\perp$
1010	$\perp$
1011	$\perp$
1100	10
1101	$\perp$
1110	$\perp$
1111	11

Appendix B
Additional Proofs

B.1 Proof of Lemma 1

Proof. By Definition 7, since $g_2 \circ g_1 \in \mathcal{S}$, $\exists M_1, M_2, \dots, M_{m+2} \in \mathbb{Z}_{>0}$ s.t.
$$g_1 : \mathbb{F}_2^{M_1} \times \mathbb{F}_2^{M_2} \times \cdots \times \mathbb{F}_2^{M_m} \rightarrow \mathbb{F}_2^{M_{m+1}}, \;\; g_2 : \mathbb{F}_2^{M_{m+1}} \rightarrow \mathbb{F}_2^{M_{m+2}}.$$
For $1 \leq i \leq m+2$, take k_i such that $\{\texttt{Encoder}_C(\boldsymbol{x}) | \boldsymbol{x} \in \mathbb{F}_2^{M_i}\} = C^{k_i} \subseteq \mathbb{F}_2^{nk_i}$, then
$$\varphi_C(g_1) : \Big(\mathbb{F}_2^{nk_1} \cup \{\perp\}\Big) \times \Big(\mathbb{F}_2^{nk_2} \cup \{\perp\}\Big) \times \cdots \times \Big(\mathbb{F}_2^{nk_m} \cup \{\perp\}\Big) \rightarrow C^{k_{m+1}} \cup \{\perp\},$$
such that for $\boldsymbol{x} = \Big(\texttt{Encoder}_C(\boldsymbol{x}_1), \texttt{Encoder}_C(\boldsymbol{x}_2), \dots, \texttt{Encoder}_C(\boldsymbol{x}_m)\Big) \in C^{k_1} \times C^{k_2} \times \dots C^{k_m}$, $\varphi_C(g_1)(\boldsymbol{x}) = \texttt{Encoder}_C\Big(g_1(\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_m)\Big)$ and $\forall \boldsymbol{x} \in \Big(\mathbb{F}_2^{nk_1} \cup \{\perp\}\Big) \times \Big(\mathbb{F}_2^{nk_2} \cup \{\perp\}\Big) \times \cdots \times \Big(\mathbb{F}_2^{nk_m} \cup \{\perp\}\Big) \backslash C^{k_1} \times C^{k_2} \times \dots C^{k_m}$, $\varphi_C(g_1)(\boldsymbol{x}) = \perp$. Moreover
$$\varphi_C(g_2) : \mathbb{F}_2^{nk_{m+1}} \cup \{\perp\} \rightarrow C^{k_{m+2}} \cup \{\perp\},$$
such that for $\boldsymbol{y} = \texttt{Encoder}_C(\boldsymbol{a}) \in C^{k_{m+1}}$, $\varphi_C(g_2)(\boldsymbol{y}) = \texttt{Encoder}_C(g_2(\boldsymbol{a}))$ and $\forall \boldsymbol{y} \in \mathbb{F}_2^{nk_{m+1}} \cup \{\perp\} \backslash C^{k_{m+1}}, \varphi_C(g_2)(\boldsymbol{y}) = \perp$. We have $\varphi_C(g_2) \circ \varphi_C(g_1)$ is a map
$$\Big(\mathbb{F}_2^{nk_1} \cup \{\perp\}\Big) \times \Big(\mathbb{F}_2^{nk_2} \cup \{\perp\}\Big) \times \cdots \times \Big(\mathbb{F}_2^{nk_m} \cup \{\perp\}\Big) \rightarrow C^{k_{m+2}} \cup \{\perp\}$$
such that for $\boldsymbol{x} = \Big(\texttt{Encoder}_C(x_1), \texttt{Encoder}_C(x_2), \dots, \texttt{Encoder}_C(x_m)\Big) \in C^{k_1} \times C^{k_2} \times \dots C^{k_m}$,
$$\begin{aligned}\Big(\varphi_C(g_2) \circ \varphi_C(g_1)\Big)(\boldsymbol{x}) &= \varphi_C(g_2)\Big(\varphi_C(g_1)(\boldsymbol{x})\Big) \\ &= \varphi_C(g_2)\Big(\texttt{Encoder}_C(g_1(\boldsymbol{x}_1, \dots, \boldsymbol{x}_m))\Big) \\ &= \texttt{Encoder}_C\Big(g_2(g_1(\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_m))\Big) \\ &= \texttt{Encoder}_C\Big(g_2 \circ g_1(\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_m)\Big),\end{aligned}$$
and $\forall \boldsymbol{x} \in \Big(\mathbb{F}_2^{nk_1} \cup \{\perp\}\Big) \times \Big(\mathbb{F}_2^{nk_2} \cup \{\perp\}\Big) \times \cdots \times \Big(\mathbb{F}_2^{nk_m} \cup \{\perp\}\Big) \backslash C^{k_1} \times C^{k_2} \times \dots C^{k_m}$,
$$\Big(\varphi_C(g_2) \circ \varphi_C(g_1)\Big)(\boldsymbol{x}) = \varphi_C(g_2)\Big(\varphi_C(g_1)(\boldsymbol{x})\Big) = \varphi_C(g_2)(\perp) = \perp .$$
On the other hand,
$$g_2 \circ g_1 : \mathbb{F}_2^{M_1} \times \mathbb{F}_2^{M_2} \times \cdots \times \mathbb{F}_2^{M_m} \rightarrow \mathbb{F}_2^{M_{m+2}},$$
and $\varphi_C(g_2 \circ g_1)$ is a map:
$$\Big(\mathbb{F}_2^{nk_1} \cup \{\perp\}\Big) \times \Big(\mathbb{F}_2^{nk_2} \cup \{\perp\}\Big) \times \cdots \times \Big(\mathbb{F}_2^{nk_m} \cup \{\perp\}\Big) \rightarrow C^{k_{m+2}} \cup \{\perp\}$$
such that for $\boldsymbol{x} = \Big(\texttt{Encoder}_C(\boldsymbol{x}_1), \texttt{Encoder}_C(\boldsymbol{x}_2), \dots, \texttt{Encoder}_C(\boldsymbol{x}_m)\Big) \in C^{k_1} \times C^{k_2} \times \dots C^{k_m}$,
$$\Big(\varphi_C(g_2 \circ g_1)\Big)(\boldsymbol{x}) = \texttt{Encoder}_C\Big(g_2 \circ g_1(\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_m)\Big),$$
and $\forall \boldsymbol{x} \in \Big(\mathbb{F}_2^{nk_1} \cup \{\perp\}\Big) \times \Big(\mathbb{F}_2^{nk_2} \cup \{\perp\}\Big) \times \cdots \times \Big(\mathbb{F}_2^{nk_m} \cup \{\perp\}\Big) \backslash C^{k_1} \times C^{k_2} \times \dots C^{k_m}$, $\Big(\varphi_C(g_2 \circ g_1)\Big)(\boldsymbol{x}) = \perp$. □ □

B.2 Proof of Proposition 1

Proof. 1. For any $\boldsymbol{j} \in \mathbb{F}_2^n$, $\varsigma_{3,\boldsymbol{j}}(\mathcal{F})$ always has the same output as $\mathcal{F}$ for any plaintext $\texttt{x}$ and any key value $\texttt{key}$. Thus $\varsigma_{3,\boldsymbol{j}}$ is safe for any $\boldsymbol{j} \in \mathbb{F}_2^n$.

For a given plaintext $\texttt{x}$ and a key value $\texttt{key}$, let $\texttt{y}$ be the correct output of $\mathcal{F}$. Then for any $\boldsymbol{j} \in \mathbb{F}_2^n$, $\varsigma_{4,\boldsymbol{j}}(\mathcal{F})$ changes the output to be $\texttt{y}\oplus\boldsymbol{j}$. The final ciphertext output is $\perp$ in case $\texttt{y}\oplus\boldsymbol{j} \notin C$. Otherwise, $\varsigma_{4,\boldsymbol{j}}$ would become a missed fault. Furthermore, we know that $\texttt{y}\in C$. Since we assume plaintext and key are random variables following uniform distributions, we can also assume that $\texttt{y}$ is a random variable with values in C, following a uniform distribution. In other words,
$$Pr[\varsigma_{4,\boldsymbol{j}} \text{ is safe}] = Pr[\texttt{y}\oplus\boldsymbol{j} \notin C, \texttt{y} \in C] = 1 - \frac{|\{\boldsymbol{c} \in C : \boldsymbol{c} \oplus \boldsymbol{j} \in C\}|}{M}.$$

Now we fix an integer m, $1 \leq m \leq n$. Then we have

$$\begin{aligned}
p_{C,m} &= \frac{1}{|\mathcal{G}_{(\mathcal{F},\mathtt{fm},m)}|} \sum_{\varsigma \in \mathcal{G}_{(\mathcal{F},\mathtt{fm},m)}} Pr[\varsigma \text{ is safe}] \\
&= \frac{1}{2\binom{n}{m}} \left(\sum_{\boldsymbol{j} \in \mathbb{F}_2^n, wt(\boldsymbol{j})=m} Pr[\varsigma_{3,\boldsymbol{j}} \text{ is safe}] + Pr[\varsigma_{4,\boldsymbol{j}} \text{ is safe}] \right) \\
&= \frac{1}{2\binom{n}{m}} \left(2\binom{n}{m} - \frac{1}{M} \sum_{\boldsymbol{j} \in \mathbb{F}_2^n, wt(\boldsymbol{j})=m} |\{\boldsymbol{c} \in C : \boldsymbol{c} \oplus \boldsymbol{j} \in C\}| \right) \\
&= \frac{1}{2\binom{n}{m}} \left(2\binom{n}{m} - \frac{1}{M} \sum_{\boldsymbol{j} \in \mathbb{F}_2^n, wt(\boldsymbol{j})=m} \sum_{\boldsymbol{c} \in C, \boldsymbol{c} \oplus \boldsymbol{j} \in C} 1 \right) \\
&= \frac{1}{2\binom{n}{m}} \left(2\binom{n}{m} - \frac{1}{M} \sum_{\boldsymbol{c} \in C} \sum_{\boldsymbol{j} \in \mathbb{F}_2^n, wt(\boldsymbol{j})=m, \boldsymbol{c} \oplus \boldsymbol{j} \in C} 1 \right) \\
&= \frac{1}{2\binom{n}{m}} \left(2\binom{n}{m} - \frac{1}{M} \sum_{\boldsymbol{c} \in C} \sum_{\boldsymbol{c}' \in C, \mathrm{dis}(\boldsymbol{c},\boldsymbol{c}')=m} 1 \right) \\
&= \frac{1}{2\binom{n}{m}} \left(2\binom{n}{m} - \frac{1}{M} \sum_{\boldsymbol{c} \in C} |\{\boldsymbol{c}' \in C, \mathrm{dis}\,(\boldsymbol{c}, \boldsymbol{c}') = m\}| \right) \\
&= \frac{1}{2\binom{n}{m}} \left(2\binom{n}{m} - \frac{S_{m,C}}{M} \right) = 1 - \frac{S_{m,C}}{2M\binom{n}{m}}.
\end{aligned}$$

2. $\vartheta_3(\mathcal{F})$ is a program that consists of instructions $1, 2, 4, 5$ in Table 1. For any fixed plaintext `x` and key value `key`, the output of the program `y` is not affected by this instruction skip. By Definition 13, ϑ_3 is safe.

   $\vartheta_4(\mathcal{F})$ is a program that consists of instructions $1, 2, 3, 5$ in Table 1. For any fixed plaintext `x` and key value `key`, the output of the program `y` is always 0, which corresponds to our error message $\perp$. By Definition 13, ϑ_4 is safe.

   Thus both $\vartheta \in \mathcal{G}_{(\mathcal{F},\mathtt{sk})}$ are safe faults. We can conclude that $p_{C,\mathtt{sk}} = 1$.

□

B.3 Proof of Lemma 2

Proof. Firstly, we notice that for any C, $S_{m,C}$ is an even integer. If $S_{m,C} \neq 0$, then $S_{m,C} \geq 2$.

1. By Proposition 1,

$$p_{C_1,n} = 1 - \frac{S_{n,C_1}}{2M\binom{n}{n}} = 1 - \frac{S_{n,C_1}}{2M} \leq 1 - \frac{2}{2M} = 1 - \frac{1}{M}.$$

   By Definition 15, $p_{C_1,\mathtt{bf}} = \min_{1 \leq m \leq n} p_{C_1,m} \leq 1 - \frac{1}{M}$.

2. By definition, $S_{mC_2} = 0$ for $m < d_2$ and $m > \delta_2$. Since $\delta_2 - d_2 \leq 2$ and $\sum_{m=1}^{n} S_{m,C_2} = M(M-1)$, there exists an m_0 such that $S_{m_0,C_2} \geq \frac{M(M-1)}{3}$. We have

$$\begin{aligned}
p_{C_2,\mathtt{bf}} &= \min_{1 \leq m \leq n} p_{C_2,m} \leq p_{C_2,m_0} = 1 - \frac{S_{m_0,C_2}}{2M\binom{n}{m_0}} \\
&\leq 1 - \frac{M(M-1)}{6M\binom{n}{m}} \\
&\leq 1 - \frac{M(M-1)}{6M\binom{n}{\ell}} = 1 - \frac{M-1}{6\binom{n}{\ell}}.
\end{aligned}$$

3. We give the proof for a, the proof for b is similar.

   For any $1 \leq m \leq n$, by 1,

$$\begin{aligned}
p_{C_3,m} - p_{C_1,\mathtt{bf}} &\geq 1 - \frac{S_{m,C_3}}{2M\binom{n}{m}} - \left(1 - \frac{1}{M}\right) = -\frac{S_{m,C_3}}{2M\binom{n}{m}} + \frac{1}{M} \\
&> -\frac{2\binom{n}{m}}{2M\binom{n}{m}} + \frac{1}{M} = 0.
\end{aligned}$$

□

B.4 Proof of Lemma 3

Proof. i and ii easily follow from the definitions.

iii,iv. Let C be an (n, d, δ)−binary anticode. For any $\boldsymbol{c} = (c_1, c_2, \ldots, c_n) \in C$, define $\tilde{\boldsymbol{c}} := (c_1, c_2, \ldots, c_n, 1)$ and let $C' := \{\tilde{\boldsymbol{c}} : \boldsymbol{c} \in C\}$. Then C' is an $(n+1, d, \delta)$−binary anticode. This proves part iii.

Now assume $\delta \geq d + 1$. Take $\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3, \boldsymbol{c}_4 \in C$ such that $\mathrm{dis}\,(\boldsymbol{c}_1, \boldsymbol{c}_2) = d$ and $\mathrm{dis}\,(\boldsymbol{c}_3, \boldsymbol{c}_4) = \delta$. Without loss of generality, we can assume $\boldsymbol{c}_3 \neq \boldsymbol{c}_1$ and $\boldsymbol{c}_3 \neq \boldsymbol{c}_2$. Suppose $\boldsymbol{c}_3 = (x_1, x_2, \ldots, x_n)$ and take

$$C'' := \left(C' \backslash \{\tilde{\boldsymbol{c}}_3\}\right) \cup \{(x_1, x_2, \ldots, x_n, 0)\}.$$

Then $\mathrm{dis}\,((x_1, x_2, \ldots, x_n, 0), \tilde{\boldsymbol{c}}_4) = \delta + 1$, $\mathrm{dis}\,(\tilde{\boldsymbol{c}}_1, \tilde{\boldsymbol{c}}_2) = d$ and $\forall \boldsymbol{x}, \boldsymbol{y} \in C''$, $d \leq \mathrm{dis}\,(\boldsymbol{x}, \boldsymbol{y}) \leq \delta + 1$. Thus C'' is an $(n+1, d, \delta+1)$−binary anticode. This proves iv.

v. Let C be an $(n, d+1, \delta)$−binary anticode. Take $\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3, \boldsymbol{c}_4 \in C$ s.t. $\mathrm{dis}\,(\boldsymbol{c}_1, \boldsymbol{c}_2) = d + 1$ and $\mathrm{dis}\,(\boldsymbol{c}_3, \boldsymbol{c}_4) = \delta$. Since $\delta > d + 1$, without loss of generality, we can assume $\boldsymbol{c}_3 \neq \boldsymbol{c}_1$ and $\boldsymbol{c}_3 \neq \boldsymbol{c}_2$. Also, we can assume the first bit of $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ are different. For any $\boldsymbol{c} = (c_1, c_2, \ldots, c_n) \in C$, define $\tilde{\boldsymbol{c}} := (1, c_2, c_3, \ldots, c_n)$ and let $C' := \{\tilde{\boldsymbol{c}} : \boldsymbol{c} \in C\}$. Then $\mathrm{dis}\,(\tilde{\boldsymbol{c}}_1, \tilde{\boldsymbol{c}}_2) = d$ and $\forall \boldsymbol{x}, \boldsymbol{y} \in C'$, $d \leq \mathrm{dis}\,(\boldsymbol{x}, \boldsymbol{y}) \leq \delta$. If C' is an (n, d, δ)−binary anticode, then we're done. Otherwise, $\mathrm{dis}\,(\tilde{\boldsymbol{c}}_3, \tilde{\boldsymbol{c}}_4) = \delta - 1$. Suppose $\boldsymbol{c}_3 = (x_1, x_2, \ldots, x_n)$ and take $C'' := \left(C' \backslash \{\tilde{\boldsymbol{c}}_3\}\right) \cup \{(0, x_2, \ldots, x_n)\}\}$, then $\mathrm{dis}\,((0, x_2, \ldots, x_n), \tilde{\boldsymbol{c}}_4) = \delta$ and C'' is an (n, d, δ)−binary anticode.

vi, vii. Let C be an $(n, M, 2r-1, \delta)$ binary anticode. Take $\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3, \boldsymbol{c}_4 \in C$ such that $\mathrm{dis}\,(\boldsymbol{c}_1, \boldsymbol{c}_2) = 2r - 1$ and $\mathrm{dis}\,(\boldsymbol{c}_3, \boldsymbol{c}_4) = \delta$.

We add one parity check bit for each codeword in C to get a binary anticode C': For any $\boldsymbol{c} = (c_1, c_2, \ldots, c_n) \in C$, define $\tilde{\boldsymbol{c}} := (c_1, c_2, \ldots, c_n, c_1 + c_2 + \cdots + c_n \bmod 2)$ and let $C' := \{\tilde{\boldsymbol{c}} : \boldsymbol{c} \in C\}$. Since $2r - 1$ is odd, $\mathrm{dis}\,(\tilde{\boldsymbol{c}}_1, \tilde{\boldsymbol{c}}_2) = 2r$ and $\forall \boldsymbol{x}, \boldsymbol{y} \in C$, $\mathrm{dis}\,(\boldsymbol{x}, \boldsymbol{y}) \geq 2r$.

If $\delta = 2\ell - 1$ is odd, $\mathrm{dis}\,(\tilde{\boldsymbol{c}}_3, \tilde{\boldsymbol{c}}_4) = 2\ell$ and $\forall \boldsymbol{x}, \boldsymbol{y} \in C'$, $\mathrm{dis}\,(\boldsymbol{x}, \boldsymbol{y}) \leq 2\ell$. So C' is an $(n, M, 2r, 2\ell)$−binary anticode. This proves vi.

If $\delta = 2\ell$ is even, $\forall \boldsymbol{x}, \boldsymbol{y} \in C$ with $\mathrm{dis}\,(\boldsymbol{x}, \boldsymbol{y}) = \delta$, $\mathrm{dis}\,(\boldsymbol{x}', \boldsymbol{y}') = \delta$ and we have C' is an $(n, M, 2r, 2\ell)$−binary anticode. This proves vii.

□

Appendix C
Further Results on Fault Analysis

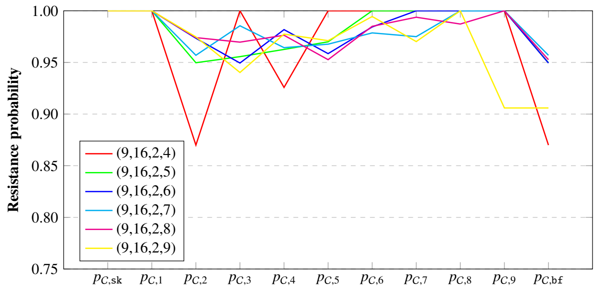


Fig. 7: Simulated results for anticodes with $n = 9, d = 2$.

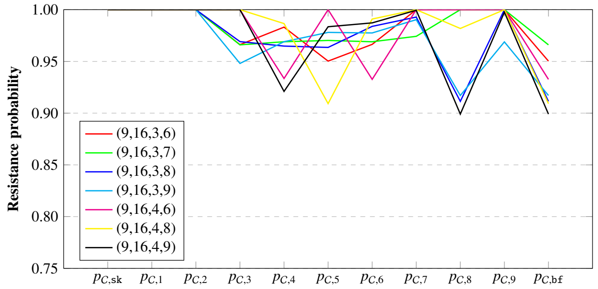


Fig. 8: Simulated results for anticodes with $n = 9, d = 3, 4$.

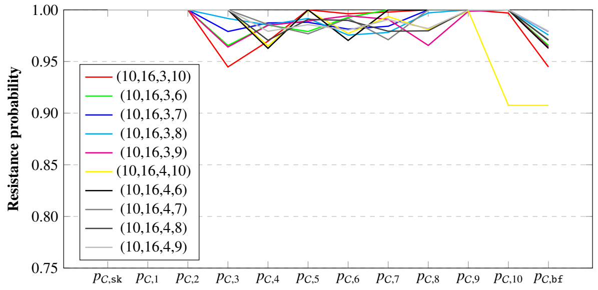


Fig. 9: Simulated results for anticodes with $n = 10, d = 3, 4$.

Appendix D
Anticodes

TABLE 4: Table of $(n, 16, d, \delta)-$binary anticodes C selected by Algorithm 1. For each anticode C with parameters (n, M, d, δ), $p_{C,\mathtt{bf}}$ in third column is calculated w.r.t. $\mathcal{F}$ and $\varphi_C(g)$ (Section 5.1). The last column gives $p_{C,\mathtt{bf}}$ computed by using Algorithm 2 for PRESENT-80 implementation in Section 7.

		$p_{C,\mathtt{bf}}$	
Codewords of C	(n, M, d, δ)	Sec. 5.1	Algo. 2
`1, 7B, 68, 22, B8, 7, 46, 1A, 24, 29, 2E, 30, 33, 35, 36, 84`	$(8, 16, 2, 8)$	0.9421	0.9068
`1, 8, 2, B, 4, 1D, 1E, 30, 7, 65, 6A, AD, B3, CE, D9, F6`	$(8, 16, 2, 7)$	0.9688	0.8396
`1, 8F, 7D, 6, 2F, 3B, C, 66, 1A, 1D, 20, 23, 34, 51, DA, E8`	$(8, 16, 2, 6)$	0.9665	0.9105
`1, 36, 50, A2, D2, 9A, 46, C4, 8, E, 17, 30, 83, 95, 9C, A4`	$(8, 16, 2, 5)$	0.9643	0.9322
`1, 62, 64, 68, 70, A2, A4, A8, B0, C2, C4, C8, D0, E3, E5, E9`	$(8, 16, 2, 4)$	0.9063	0.8147
`1, AF, FB, A, 3C, EC, C0, 92, 17, 26, 4D, 54, 63, 99, C7, F5`	$(8, 16, 3, 8)$	0.9375	0.9069
`1, 37, 38, 42, 4C, 55, 5B, 6F, 8B, 9C, A5, AE, B2, D6, E0, F9`	$(8, 16, 3, 7)$	0.9625	0.9318
`1, 62, 6, 65, 18, 7B, 7C, A8, 1F, AF, B1, B6, CA, CD, D3, D4`	$(8, 16, 3, 6)$	0.9643	0.9326
`4, D0, E6, A1, 43, 8A, 19, 32, 68, 97, CD, 75, BC, 5E, 2F, FB`	$(8, 16, 4, 8)$	0.5	0.0681
$n = 9$			
`1, D5, 1D6, 2E, 42, 158, 85, 11B, 106, 108, 10D, 110, 115, 11C, 120, 12A`	$(9, 16, 2, 9)$	0.9375	0.9060
`1, F3, 167, BD, B0, 1D3, 25, C5, 11C, 11F, 120, 123, 126, 139, 188, 1D8`	$(9, 16, 2, 8)$	0.9844	0.9527
`1, 1D9, 1A9, A4, 1C2, 1B4, D4, 10, 8D, 8E, 91, 92, 97, EA, 10A, 17F`	$(9, 16, 2, 7)$	0.9841	0.9569
`1, 180, 51, D2, 110, F8, 6A, 74, 16, 18, 1B, 26, 8D, 11C, 13B, 14C`	$(9, 16, 2, 6)$	0.9831	0.9494
`1, 115, 4C, 9F, 7D, 18D, 1D5, 99, 17, 25, 59, 94, AD, C1, C7, F5`	$(9, 16, 2, 5)$	0.9762	0.9497
`1, 2, 4, 8, 31, 32, 34, 51, 52, 58, 94, 98, E0, 130, 150, 190`	$(9, 16, 2, 4)$	0.9375	0.8700
`1, 44, 18, 160, 9F, 1FA, A0, 1A3, 116, 11B, 125, 12A, 13C, 143, 14D, 177`	$(9, 16, 3, 9)$	0.9375	0.9171

1, 13C, 149, 1F6, 187, 1D3, 2F, 1E5, 70, 77, 82, 8C, 95, 9B, E8, 132	$(9, 16, 3, 8)$	0.9836	0.9114
1, 27, A, 1B3, 7E, 2C, F0, DF, ED, 104, 117, 118, 14B, 162, 1AA, 1C6	$(9, 16, 3, 7)$	0.9831	0.9660
1, 1E7, 8E, 42, 76, 11F, 1C4, 134, 2C, 55, 6F, 97, A5, B2, DC, F9	$(9, 16, 3, 6)$	0.9821	0.9503
1, 16, 17B, 2A, 198, 165, 18F, 142, 3D, 4C, 70, A4, B3, D5, E9, FE	$(9, 16, 4, 9)$	0.9375	0.8991
1, E4, 1B0, BD, CA, 179, 116, 1D5, 3A, 5C, 77, 12C, 14F, 162, 19B, 1A7	$(9, 16, 4, 8)$	0.9643	0.9092
1, F8, 122, 1B4, 165, 76, 15F, 1EB, 3B, 4C, 97, AD, C2, 118, 18E, 1D1	$(9, 16, 4, 6)$	0.9643	0.9327
$n = 10$			
1, 399, 331, 2B3, F6, 17D, 2C2, 294, 92, 95, 98, 9B, 9E, A0, A3, CE	$(10, 16, 2, 10)$	0.9375	0.9738
1, 87, 176, 102, 1F8, 200, 38F, 108, 216, 218, 21B, 222, 225, 2CC, 2F3, 351	$(10, 16, 2, 9)$	0.9921	0.9717
1, 202, 27E, 45, 2DD, 38A, 23, 39B, 251, 252, 260, 267, 2AC, 314, 3B7, 3E9	$(10, 16, 2, 8)$	0.9921	0.9639
1, 46, 23D, 16E, 107, 25F, E1, 2E7, 340, 343, 345, 349, 371, 384, 38A, 3B2	$(10, 16, 2, 7)$	0.9917	0.9783
1, 3AB, 14A, 20E, 1F, 15F, 23B, AF, 8E, 92, 98, CB, 122, 128, 26A, 383	$(10, 16, 2, 6)$	0.9906	0.9722
1, 24A, 8A, 298, 268, 25B, 109, 20F, 4C, 59, 200, 229, 28D, 2C1, 308, 3C9	$(10, 16, 2, 5)$	0.9854	0.9614
1, 381, 80, 140, 302, 182, 103, 304, 105, 108, 110, 121, 184, 1A0, 200, 320	$(10, 16, 2, 4)$	0.9542	0.9078
1, 6, 18, 1F, 2A, 2D, 33, 34, 4B, 4C, 52, 55, 60, 67, 79, 386	$(10, 16, 3, 10)$	0.9375	0.9447
1, 6, 18, 1F, 2A, 2D, 33, 4B, D4, 1E0, 1FF, 2E6, 353, 37C, 385, 38A	$(10, 16, 3, 9)$	0.9921	0.9641
1, 112, 29A, 338, 283, 3C7, 27D, 389, 24B, 24C, 256, 2B5, 2EA, 33F, 3A4, 3F0	$(10, 16, 3, 8)$	0.9916	0.9756
2, 3A4, D7, 143, 1FA, 3EB, 3F0, 283, B9, C0, CD, E3, 109, 131, 18E, 258	$(10, 16, 3, 7)$	0.9917	0.9790
1, AC, 261, 22D, 59, 34C, 3C5, CF, C4, 107, 108, 1E9, 24A, 280, 28B, 29D	$(10, 16, 3, 6)$	0.9893	0.9655
1, 193, 277, A2, 160, 3CA, 33E, BF, F8, 106, 118, 12B, 135, 14D, 1AC, 26C	$(10, 16, 4, 10)$	0.9375	0.9075
1, 3B1, 2BC, 156, 32F, 9B, 340, 35D, E0, EF, 138, 1A6, 20A, 273, 2C5, 3DA	$(10, 16, 4, 9)$	0.9902	0.9792
1, 304, 3DF, FC, 86, E3, 28B, 295, 10A, 11D, 177, 1D0, 238, 26E, 3B2, 3E9	$(10, 16, 4, 8)$	0.9896	0.9705
1, 2EF, 3A3, 18C, 395, 1B6, 370, 244, 75, B8, 11F, 169, 1C2, 21A, 32E, 3DB	$(10, 16, 4, 7)$	0.9891	0.9711
1, E, 32, 3D, C4, CB, F7, F8, 150, 15F, 163, 16C, 195, 19A, 1A6, 256	$(10, 16, 4, 6)$	0.9794	0.9627