Fault Resilient Encoding Schemes in Software: How Far Can We Go?

Jakub Breier[1], Xiaolu Hou[2], and Yang Liu[2]

[1]Physical Analysis and Cryptographic Engineering
Temasek Laboratories at Nanyang Technological University, Singapore
[2]School of Computer Science and Engineering
Nanyang Technological University, Singapore
jbreier@ntu.edu.sg,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 on 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. These findings show that using anticodes in such countermeasure can increase its detection capabilities. We provide a way to generate a code according to user criteria and also a method to evaluate the level of protection of assembly implementations using encoding schemes. This evaluation is based on static code analysis and provides a practical information on how good will the protection be on a real device. Finally, we verify our findings by implementing a block cipher PRESENT, protected by encoding scheme based on anticodes, and provide a detailed evaluation of such implementation.

Keywords: software encoding schemes, fault attacks, countermeasures, evaluation

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 some 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 among 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.

When it comes to fault injection countermeasures, implementers currently still rely more on hardware-based approaches, such as shielding [16], sensors [16], or hardware redundancy [19]. This is mostly because to inject a fault, physical methods are normally used, such as lasers, electromagnetic pulses, or voltage/clock glitches [2], and therefore, physical protections are effective in detecting/thwarting these.

Software countermeasures against fault attacks can be generally divided into two main groups: instruction-level and algorithm-level techniques [33]. 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 [26]. This technique was recently extended to a new approach, called *intra-instruction redundancy* [27]. 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 [30]. Temporal redundancy techniques normally execute the algorithm several times and then compare the results for inconsistencies [2, 10].

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 [8]. Breier and Hou [7] 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 [9, 32]. However, no real implementation or simulation was given in either work. Servant et al. [32] 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 [9] 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 [9], used code would have to remain a secret.

There are works that utilize encoding techniques in hardware to provide fault resiliency, e.g. [1, 31]. 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.

Another point of view is the evaluation of countermeasures. While most of the works propose new protection methods, they often only consider single operations, not in the context of the full cipher. Sometimes it is not clear whether the fully protected implementation is actually possible and also, what other vulnerabilities might arise when practical aspects are considered. For example, in case there is an integrity check after every operation that is considered secure and out of scope of the security guarantees, the attacker might simply skip such check or alter its value. Similarly, evaluation methods often focus on code snippets only [17]. In the area of general software countermeasures, there are very few works that provide evaluation results on a full cipher [23, 3].

When restricting the evaluation of countermeasures to encoding methods, to the best of our knowledge, there is no previous work that evaluates the countermeasure on a full cipher scale with both theoretical and experimental results. Fault resistance evaluation of single operations was previously simulated in [8, 7]. Calculations that provide theoretical bounds were provided in [9, 32].

Our Contribution In this work we are interested in analyzing encoding countermeasures for a full cipher implementation. To facilitate the evaluation, we formalize Differential Fault Attacks (DFAs) and encoding countermeasures in software, bringing light into understanding of what is needed and what is possible. With such information, we are able to find the optimal codes for the cipher protection.

We develop an evaluation method for encoding based countermeasures that is based on static code analysis and works directly on assembly implementation. We implemented a protected version of PRESENT-80 cipher by using an AVR assembly language and used our evaluation method to analyze such implementation from various points of view, enabling us to show trade-offs between the security level and the efficiency (speed, time). Both advantages and disadvantages of such implementation are discussed. To the best of our knowledge, this is the first work implementing and evaluating the encoding countermeasure on a full cipher.

We provide a way to automatically generate codes with required properties for protecting cryptographic implementations against DFA (if such codes exist). We adopt the notion of anticodes from coding theory and show that anticodes can offer the best fault detection capabilities.

The rest of the paper is organized as follows. Section 2 provides background on fault attacks on software targets and generalizes fault resilient encoding scheme. Theoretical evaluation of this scheme is stated in Section 3. Algorithms used for code construction and for evaluation of software implementations are detailed in Section 4. Section 5 provides a case study on block cipher PRESENT. Discussion of results is stated in Section 6 and finally, Section 7 concludes this paper and provides a motivation for future work.

2 Background and Formalization

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.

2.1 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* ith 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* $\boldsymbol{j} \in \mathbb{F}_2^N$ $(N \in \mathbb{Z}_{>0})$*, 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.*

For the evaluation, we consider a random bit fault such that all the bits have equal probability to be affected by the fault. In other words, each fault mask has the same probability to occur. We also state detailed analysis for each m-bit flip model ($1 \leq m \leq s$, where s is the register size), so that in the case of biased fault model, it is easy to see the upper bound for the fault detection.

2.2 Fault Resilient Encoding Scheme

A *binary code*, which we denote by C in this paper, 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* in C and each element $\boldsymbol{x} \in \mathbb{F}_2^n$ is called a *word* [20, 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 $\operatorname{dis}(\boldsymbol{x}, \boldsymbol{y})$, is defined to be the number of places at which $\boldsymbol{x}$ and $\boldsymbol{y}$ differ [20, p.9]. More precisely, if $\boldsymbol{x} = x_1 x_2 \dots x_n$ and $\boldsymbol{y} = y_1 y_2 \dots y_n$, then

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

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

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

Furthermore, for a word $\boldsymbol{x} \in \mathbb{F}_2^n$, the *Hamming weight* of $\boldsymbol{x}$, $\operatorname{wt}(\boldsymbol{x}) := \operatorname{dis}(\boldsymbol{x}, \boldsymbol{0})$ [20, p.46].
For a binary code C, the *(minimum) distance* of C, denoted by $\operatorname{dis}(C)$, is [20, p.11]

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

Definition 3. *[11, p.75] For a binary code C of length n, with* $\operatorname{dis}(C) = d$, *if $M = |C|$ is the number of codewords in C. Then C is called an $(n, M, d)-$binary code.*

To simplify the notation we introduce the symbol $\perp$, which indicates an error message. 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 $\boldsymbol{0} \notin C$.

Definition 4. *A* symmetric cipher *(see e.g. [18, 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 $N_1, N_2, N_3 \in \mathbb{Z}_{>0}$.

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) *$D(\kappa, \perp) = \perp$.*

And we define $\mathfrak{S}_\perp$ to be 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 $N_1, N_2, N_3 \in \mathbb{Z}_{>0}$.

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

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

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

Thus for $\texttt{Decoder}_C$ an error message $\perp$ will be returned if the input is not a codeword. 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 $\boldsymbol{x} = (x_1, x_2, \ldots, x_N) \in \mathbb{F}_2^N$, let $\boldsymbol{x}' = (x_1, x_2, \ldots, x_N, \ 0, \ldots, 0) \in \mathbb{F}_2^{N+N'}$, 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 $\boldsymbol{x} = (x_1, x_2, \ldots, x_N) \in \mathbb{F}_2^N$, let $\boldsymbol{x}_i = (x_{ik+1}, x_{ik+2}, \ldots, x_{ik+k}), 0 \leq i \leq k' - 1$ and define

$$\texttt{Encoder}_C(\boldsymbol{x}) := \Big(\texttt{Encoder}_C(\boldsymbol{x}_0), \ldots, \texttt{Encoder}_C(\boldsymbol{x}_{k'-1})\Big) \in C^{k'}.$$

It follows that $\texttt{Encoder}_C : \mathbb{F}_2^N \to C^{k'}$ is a bijective function. We define $\texttt{Decoder}_C : \mathbb{F}_2^{nk'} \cup \{\perp\} \to \mathbb{F}_2^N$ such that $\texttt{Decoder}_C\big|_{C^{k'}} \to \mathbb{F}_2^N$ is the inverse of $\texttt{Encoder}_C$ and $\texttt{Decoder}_C\big|_{(\mathbb{F}_2^{nk'} \cup \{\perp\}) \setminus C^{k'}} = \{\perp\}$.

Definition 7 (Operation). *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 a program $\mathcal{F}$ (see Definition 1) can take inputs and outputs. Furthermore, an assembly implementation of an operation is a program.

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 1. 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 $(\texttt{Encoder}_C, \texttt{Decoder}_C)$, $\texttt{Encoder}_C \in \mathcal{S}$ and $\texttt{Decoder}_C \in \mathcal{S}_\perp$.

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

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)

Remark 2. 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 $\varphi_C(E) = \varphi_C(g_1) \circ \varphi_C(g_2) \circ \cdots \circ \varphi_C(g_m)$.

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

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

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

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

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

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

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 3. Taking $k = 1$ and $C = \{01, 10\}$, we get the bit-sliced encoding, e.g. the one used in [28] ($\mathtt{Encoder}_C(0) = 01$, and $\mathtt{Encoder}_C(1) = 10$) which follows the principle of a dual-rail precharge logic. In Section 5.1, we use $k = 4$ mainly because PRESENT cipher uses 4-bit SBox (see Section 5.1).

For a better understanding of how this 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.

$$\begin{tikzcd}
\mathrm{P} \arrow[r] & \texttt{Encoder}_C(P) \arrow[dr] & & & \\
& & \boxed{\mathrm{E}} \arrow[r] & \varphi_C(E)\Big(\texttt{Encoder}_C(P), \texttt{Encoder}_C(\kappa)\Big) \arrow[r, "\texttt{Decoder}_C"] & \mathrm{Msg} \\
\kappa \arrow[r] & \texttt{Encoder}_C(\kappa) \arrow[ur] & & & \\
\mathrm{Msg} \arrow[r] & \texttt{Encoder}_C(Msg) \arrow[dr] & & & \\
& & \boxed{\mathrm{D}} \arrow[r] & \varphi_C(D)\Big(\texttt{Encoder}_C(Msg), \texttt{Encoder}_C(\kappa)\Big) \arrow[r, "\texttt{Decoder}_C"] & \mathrm{P} \\
\kappa \arrow[r] & \texttt{Encoder}_C(\kappa) \arrow[ur] & & &
\end{tikzcd}$$

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

3 Theoretical Evaluation of Fault Resilient Encoding Scheme

In this section we first define faults in encoding schemes and provide concepts of safe and exploitable faults. Then, we theoretically evaluate Fault Resilient Encoding Scheme using one operation and we show why anticodes provide better fault detection properties.

3.1 Defining Faults in Encoding Schemes

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*

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

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

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

3. *For an integer $1 \leq m \leq n$,* 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* exploitable *otherwise.*
6. *The pair $(\varphi_C(g), \mathcal{F})$ is said to be $p-$safe, if for given random input x and random fault ϱ on $\mathcal{F}$, the probability $Pr(\varrho$ is safe$) = p$*

Remark 4. A fault is closely related to a tampering function defined in [13]. 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. By Definition 2 we have the following

Definition 13 (Safe and exploitable 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* exploitable *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* exploitable *otherwise.*

Furthermore, we define:

Definition 14 ($p-$safe). *The triple $(\Phi_C\Big((\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)\Big), \mathcal{F}_1, \mathcal{F}_2)$ is said to be $p-$safe if for given random $P \in \mathcal{P}$, $\kappa \in \mathcal{K}$ and random fault ϱ on $\mathcal{F}_1$ or $\mathcal{F}_2$, $Pr(\varrho$ is safe$) = p$.*

To explain why the fault can be exploited, we have to look into preconditions of a differential fault analysis [5]. For such attack, 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 gains information about the secret key. If the fault is *exploitable*, 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.

Note that the definition of $p-$safe assumes that an attack can achieve all possible instruction skips and fault masks, furthermore, each fault happens with an equal probability.

3.2 From Encoding Schemes to Anticodes

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 which does not contain the zero codeword with associated encoding-decoding scheme $(\texttt{Encoder}_C, \texttt{Decoder}_C)$. We will use zero string to denote $\perp$, the error message. That is why we require that $\mathbf{0} \notin C$. Furthermore, we choose k such that $k = \max\{M_1, M_2\}$. Let $\mathcal{F}$ be the assembly implementation (in Figure 2) of $\varphi_C(g)$. In the code, 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 in our case. Before executing the code we precharge all the registers to zero. Note that the table has $2^n \times 2^n$ entries. The value stored at address (a, b) is zero if

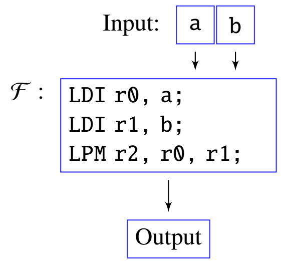

Fig. 2: Assembly implementation $\mathcal{F}$ for $\varphi_C(g)$, where $g : \mathbb{F}_2^{M_1} \times \mathbb{F}_2^{M_2} \to \mathbb{F}_2^{M_3}$ is a binary operation.

$a, b \notin C$ and the value is $\texttt{Encoder}_C(g(\texttt{Encoder}_C(x), \texttt{Encoder}_C(y)))$ if $a = \texttt{Encoder}_C(x)$ and $b = \texttt{Encoder}_C(y)$.

Then by Definition 12, the set of possible instruction skips and the set of possible fault masks for $\mathcal{F}$ are given by

$$\mathcal{G}_{(\mathcal{F},\mathrm{sk})} = \{\vartheta_1, \vartheta_2, \vartheta_3\}, \ \ \mathcal{G}_{(\mathcal{F},\mathrm{fm})} = \{\varsigma_{i,\boldsymbol{j}} : 1 \le i \le 3, \boldsymbol{j} \in \mathbb{F}_2^n\}.$$

Binary operations are very common in symmetric ciphers, e.g. `xor`, `and`, `modular addition`. We would like to point out that the analysis in this section for binary operations can be easily generalized to a whole cipher implementation. The implementation $\mathcal{F}$ is a natural way of implementing $\varphi_C(g)$ in assembly language. Analyzing the fault resistance property of $\mathcal{F}$ gives insights to the overall fault resistance of using C in our scheme and it provides a solid approximation of the fault resistance of a full cipher implementation.

We adopt the following notions defined in [7, Definitions 9 and 10]:

Definition 15. *For an (n, M, d)-binary code C such that $d \ge 2$, let $S_{m,C} := \sum_{\boldsymbol{c}\in C} |\{\boldsymbol{c}' \in C : \mathrm{dis}(\boldsymbol{c}', \boldsymbol{c}) = m\}|$. Then*

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

*is called the $m-$*bit fault resistance probability *for C. Furthermore,*

$$p_{\mathrm{rand},C} := \sum_{m=1}^{n} \frac{1}{n} p_{m,C}$$

is called the overall resistance index *for C.*

Since the registers are all precharged to zero, for any $\vartheta \in \mathcal{G}_{(\mathcal{F},\mathrm{sk})}$, ϑ is a safe fault. By Definition 12 we have

Lemma 2. *1. For given random $P \in \mathcal{P}$, $\kappa \in \mathcal{K}$ and random fault $\varsigma \in \mathcal{G}_{(\mathcal{F},\mathfrak{fm},m)}$, the probability $Pr[\varsigma$ is safe$] = p_{m,C}$.*

2. For given random $P \in \mathcal{P}$, $\kappa \in \mathcal{K}$ and random fault $\varsigma \in \mathcal{G}_{(\mathcal{F},\mathfrak{fm})}$, the probability $Pr[\varsigma$ is safe$] = p_{\mathrm{rand},C}$.

3. The pair $(\varphi_C(g), \mathcal{F})$ is $p_{\mathrm{rand},C}-$safe.

Thus we are interested in binary codes C with bigger values of overall resistance index. From Eq. (1), we can see that the smaller $\frac{S_{m,C}}{M\binom{n}{m}}$ is, the smaller overall resistance index we get. Since $\sum_{m=1}^{n} S_{m,C} = M(M-1)$ is always true, to make $p_{\text{rand},C}$ big, one strategy is to make $S_{m,C}$ small or even equal to zero for smaller values of $\binom{n}{m}$.
Let

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

It is known that (see e.g. [29, p.26])

$$\begin{cases} \binom{n}{i-1} < \binom{n}{i} & \text{if } 1 \le i < \ell \\ \binom{n}{i-1} > \binom{n}{i} & \text{if } \ell < i \le n \end{cases}, \text{ and } \begin{cases} \binom{n}{\ell-1} = \binom{n}{\ell} & \text{if } n \text{ is odd} \\ \binom{n}{\ell-1} < \binom{n}{\ell} & \text{if } n \text{ is even} \end{cases}. \tag{3}$$

Hence we would like to have $S_{i,C} = 0$ for i "close to" n or for i "close to" 0 and we would also like to make $S_{i,C} \neq 0$ when i is "close" to ℓ (see Lemma 3).

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

Definition 16. *[14]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 have repeated codewords but an anticode can have repeated rows [14]. 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 17. *Let C be an $(n, M, d)-$binary code, if furthermore*

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

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

Definition 18 (Fault Resilient Anticode Scheme). *A fault resilient encoding scheme $\Phi_C\big((\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)\big)$ is called a* fault resilient anticode scheme *if C is an anticode.*

We have the following observations

Lemma 3. *Given an $(n, M, d_1, \delta_1)-$binary anticode C_1 and an $(n, M, d_2, \delta_2)-$binary anticode C_2 such that $d_2 \le d_1 \le \ell$, $\delta_1 \le \delta_2$ and $S_{m,C_1} \ge S_{m,C_2}$ $\forall m = d_1, \ldots, \delta_1$. Denote the overall resistance index for C_1 and C_2 by p_{rand,C_1} and p_{rand,C_2} respectively, we have*

1. *For $d_1 = d_2$, $\delta_1 < \delta_2$,*
   a. *If $\ell \le \delta_1$ and $\delta_1 + d_1 + 1 > n$, then $p_{\text{rand},C_1} > p_{\text{rand},C_2}$;*
   b. *If $\delta_1 < \ell$, then $p_{\text{rand},C_1} < p_{\text{rand},C_2}$.*
2. *For $d_1 > d_2$, $\delta_1 = \delta_2$,*
   a. *If $d_1 \le \ell$ and $\delta_1 + d_1 + 1 > n$, then $p_{\text{rand},C_1} > p_{\text{rand},C_2}$;*

b. If $d_1 > \ell$, then $p_{\text{rand},C_1} < p_{\text{rand},C_2}$.

(The proof can be found in Appendix B)

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

$$N(n, d, \delta) := \max\{M : \exists (n, M, d, \delta) - \text{binary anticode}\}. \tag{4}$$

Two related well-studied coding theory concepts are [21, p.42]

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

and [15]

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

We have

Lemma 4. *i* $N(n, d, d) = B(n, d)$;
ii $N(n, d, n) \leq A(n, d)$;
iii $N(n, d, \delta) \leq N(n + 1, d, \delta)$;
iv $N(n, d, \delta) \leq N(n + 1, d, \delta + 1)$, *where* $\delta \geq d + 1$;
v $N(n, d + 1, \delta) \leq N(n, d, \delta)$, *where* $\delta > d + 1$;
vi $N(n, 2r - 1, 2\ell - 1) \leq N(n + 1, 2r, 2\ell)$ *where* $r, \ell \in \mathbb{Z}_{>0}$;
vii $N(n, 2r - 1, 2\ell) \leq N(n + 1, 2r, 2\ell)$, *where* $r, \ell \in \mathbb{Z}_{>0}$;

(The proof can be found in Appendix B)

In Section 5.1 we will study and analyze the implementation of Fault Resilient Anticode Scheme with PRESENT cipher. Because of the cipher design we will be interested in anticodes with cardinality 16 (see Section 5.1). By the above Lemma and some exhaustive search, we have in Table 1 the possible values of d and δ for $n = 8, 9, 10$ and $M = 16$.

4 Algorithms

In this section, we provide two useful algorithms for practical evaluation of encoding schemes. The first one generates binary anticodes according to user requirements and the second one evaluates software implementations that follow the Fault Resilient Encoding Scheme.

Table 1: 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

Algorithm 1: Anticode Generation Algorithm.

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

4.1 Anticode Generation Algorithm

In order to use and analyze the Fault Resilient Anticode Scheme, we first need to generate the binary anticodes. The algorithm created for this purpose is described in this section.

Pseudocode outlining the main idea of the anticode generation is stated in Algorithm 1. The inputs are: parameters n, M, d, δ for the binary anticode, and ε – probability of *exploitable* faults for binary operations. Please note that for our case, we use zero word as $\perp$ and thus in line 3 we choose sets S which do not contain $\mathbf{0}$.

The ε parameter is crucial for selecting an anticode with good detection 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 the optimal ε. However, for some n, M, d, δ parameters, there is no anticode with reasonable detection probabilities. For example, every $(8, 16, 4, 8)-$binary anticode has a property that $8-$bit flip has a probability 1 of being *exploitable*. Such anticodes are not suitable for protecting implementations, therefore we avoid them. For more details about this phenomenon, we refer to [7].

Definition 15 and Lemma 2 show that $1 - p_{m,C}$ can serve as a good condition for selecting anticodes. Line 5 uses this value to only consider anticodes that do not surpass the detection threshold for every possible bit-flip.

4.2 Static Code Analysis

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

Keep the same notations from Definition 13. $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D) \in \mathfrak{S}$ is a symmetric cipher, C is an (n, M, d, δ)−binary anticode and $(\mathcal{K}, \mathcal{P}, \mathcal{M} \cup \{\perp\}, E', D')$ is the corresponding Fault Resilient Anticode Scheme. $\mathcal{F}_1$ and $\mathcal{F}_2$ denote assembly implementations of D' and E' respectively. Algorithm 2 gives the static code analyzer for $\mathcal{F}_1$. With simple adjustments, it can be used for analyzing $\mathcal{F}_2$. In the following we describe how the algorithm works.

Inputs of the algorithm are: a plaintext P and a secret key κ, together with the corresponding ciphertext $E(P, \kappa)$, an $(n, M, d, \delta)-$ binary anticode, and a sequence of assembly instructions $\mathcal{F}$.

In the first part, we evaluate bit-flips (lines 1-10). The first loop iterates over every possible fault mask. Fault mask (see Definition 2) is a value which will be later xor-ed with the intermediate value in order to change the original value in the destination register of an instruction. Please note that 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 6). After the algorithm finishes, there is a checking of the output value (lines 7-10). If the value does not deviate from the expected ciphertext $E(P, \kappa)$, or the value is $\perp$, we consider it a *safe* fault. Otherwise, it is considered an *exploitable* 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.

The second part evaluates instruction skips (lines 11-20). It works in the same fashion as the previous part, however, in this case we save one loop because we do not need any fault mask. Output evaluation is analogous, but the outputs will be integers instead of arrays of integers.

Clearly, the time complexity of the first part 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 the second part 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 analysis algorithm.

```
Input  : $P$: plaintext, $\kappa$: secret key, $E(P,\kappa)$: ciphertext corresponding to encrypting $P$ with
         $\kappa$, $C$: $(n, M, d, \delta)-$binary anticode, $\mathcal{F}$: sequence of assembly instructions
Output: Distribution of safe and exploitable faults:
         Int[] SafeBitFlip: SafeBitFlip[$m$]= $|\{\varsigma \in \mathcal{G}_{(\mathcal{F},\mathfrak{fm},m)}$is safe$\}|$
         Int[] ExploitableBitFlip: SafeBitFlip[$m$]= $|\{\varsigma \in \mathcal{G}_{(\mathcal{F},\mathfrak{fm},m)}$is exploitable$\}|$
         Int SafeSkip: SafeBitFlip= $|\{\varsigma \in \mathcal{G}_{(\mathcal{F},\mathrm{sk})}$is safe$\}|$
         Int ExploitableSkip:SafeBitFlip= $|\{\varsigma \in \mathcal{G}_{(\mathcal{F},\mathrm{sk})}$is exploitable$\}|$
for Fault mask Int x: 1 to $2^n$ do
    for Int j: 0 to $|\mathcal{F}|$ do
        for Instruction f in $\mathcal{F}$ do
            Execute instruction $f$;
            if $f == j$ and $f$ has a destination register then
                $r_f = r_f \oplus$ x;
        if output == $\perp$ or output == $E(P,\kappa)$ then
            SafeBitFlip[HammingWeight(x)]++;
        else
            ExploitableBitFlip[HammingWeight(x)]++;
for Int j: 0 to $|\mathcal{F}|$ do
    for Instruction f in $\mathcal{F}$ do
        if $f == j$ then
            continue;
        else
            Execute instruction $f$;
    if output == $\perp$ or output == $E(P,\kappa)$ then
        SafeSkip++;
    else
        ExploitableSkip++;
return ExploitableBitFlip, SafeBitFlip, ExploitableSkip, SafeSkip.
```

5 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 A lists all the anticodes used). In Section 5.1, we provide implementation details by using a generic 8-bit microcontroller. Section 4.2 describes the algorithm we used for static code analysis of the assembly program. This algorithm is generic and can be used to analyze any algorithm implemented by Fault Resilient Anticode Scheme. Finally, Section 5.2 provides the results of the code analysis.

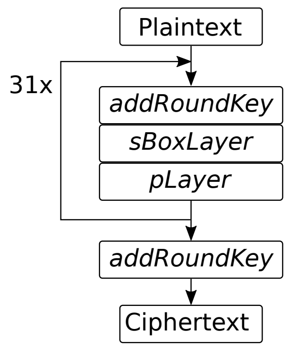

Fig. 3: Sequence of operations of PRESENT block cipher.

5.1 PRESENT Cipher

PRESENT is an ultra-lightweight block cipher, developed in 2007 [6]. 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: xor of the state with the round key, substitution by 4-bit SBox, which we refer to as PRESENT SBox, and bitwise permutation. After 31 rounds, there is one more *addRoundKey*, used for post-whitening. The whole process is depicted in Figure 3. Because of its lightweight character, it is recommended to use 80-bit key length in order to keep the computation fast and energy efficient [6]. Therefore, in our paper, we will focus on this variant, denoted by PRESENT-80.

Each round of PRESENT-80 consists of three operations: $g_1 = \text{bit permutation} : \mathbb{F}_2^{16} \to \mathbb{F}_2^{16}$, $g_2 = \text{PRESENT SBox} : \mathbb{F}_2^4 \to \mathbb{F}_2^4$and $g_3 = \text{bitwise xor with the round key} : \mathbb{F}_2^{64} \times \mathbb{F}_2^{64} \to \mathbb{F}_2^{64}$. Because g_1 and g_3 are bitwise operations, 64 and 16 are multiples of 4, we can use code with cardinality $2^4 = 16$. In particular, to apply Fault Resilient Anticode Scheme with PRESENT-80, we will use $(n, 16, d, \delta)-$binary anticodes. For our implementation, we take pre-computed round keys which are already encoded and therefore, we omit the description of the key schedule here.

Figure 4 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 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 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 2 justifies that we can split or merge multiple cipher operations while using the $C-$map (see Definition 10), preserving the correct data-flow.

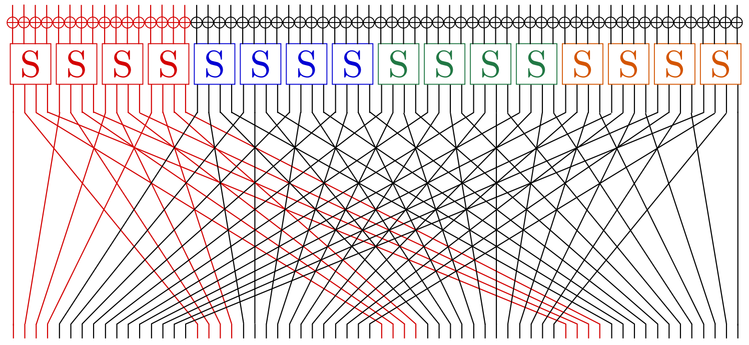

Fig. 4: One round of PRESENT.

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, as explained in Section 3.2. 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 5. The explanation of this approach is given in the following.

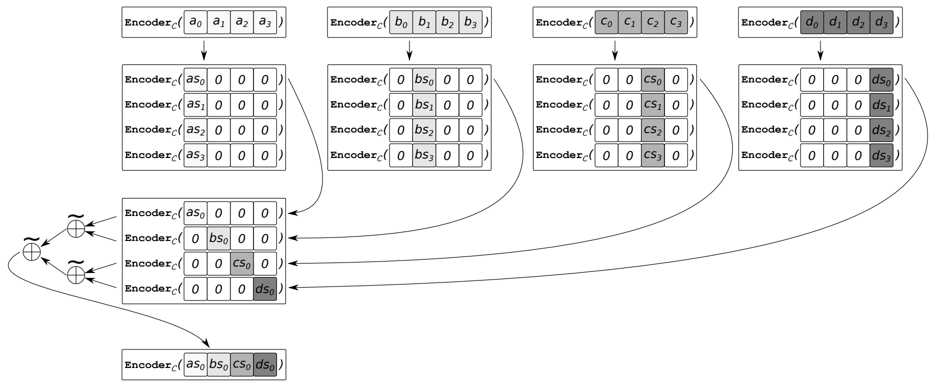

Fig. 5: Illustration of the implementation of PRESENT-80 sBoxLayer and pLayer in Fault Resilient Anticode Scheme.

Suppose we are using an $(n, 16, d, \delta)-$binary anticode C. 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 $\texttt{Encoder}_C(x_0x_1x_2x_3)$ and returns the codeword $\texttt{Encoder}_C(xs_i000)$. If the input is not a codeword, the return value will be $\perp$ (in our implementation it is zero). 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.

2. **Bit-shifting tables:** This group has four tables as well: $TB0$, $TB1$, $TB2$ and $TB3$. For a codeword of the form $\texttt{Encoder}_C(x000)$, $TB0$, $TB1$, $TB2$, $TB3$ return the codewords $\texttt{Encoder}_C(x000)$, $\texttt{Encoder}_C(0x00)$, $\texttt{Encoder}_C(00x0)$, $\texttt{Encoder}_C(000x)$, 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*.

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

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

Afterwards, we need three xor table lookups to get:

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

Here $\widetilde{\oplus}$ represents xor table lookup. We illustrate the whole process using Figure 5, which explains how the first encoded nibble is obtained.

5.2 Results

Algorithm 2 runs for one plaintext. For the evaluation purposes, we had to run it several times for random plaintexts in order to minimize the error to acceptable level. From experimental evaluation, after ≈200 random plaintexts, the error becomes negligible ($< 10^{-6}$). Therefore, we have run 200 random iterations of the algorithm for every anticode stated in Appendix A. Our PRESENT-80 algorithm implementation follows the specification stated in Section 5.1. Therefore, as inputs we used a 64-bit plaintext P, 80-bit secret key κ, and 64-bit ciphertext $E(P, \kappa)$. Please note that we did not target encoding and decoding processes. In case of faulting the encoding process, the introduced fault would change the input to the encoding algorithm, therefore the fault analysis process would just become a differential cryptanalysis. Also, this type of fault could be introduced on the data anytime before the algorithm execution which is out of scope of this work. In case of faulting the decoding process, attacker would not gain any valuable information about the secret key or the plaintext.

Fault analysis results for anticodes from Table 3 (Appendix A) with $n = 8, d = 2, 3$, and $n = 10, d = 2$ with various δ values are stated in Figures 6 and 7, 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. Percentages of *safe* random fault masks are stated in the last column of Table 3. The improvement of using a longer length for encoding the data is obvious – values of random *exploitable* faults for length 8 go up to ≈0.05, for length 9 up to ≈0.031, and for length 10 up to ≈0.016. Moreover, we can see the advantage of $(n, M, d, \delta)-$anticodes with $\delta < n$ over codes with unbounded distance, i.e. $(n, M, d, \delta)-$binary anticodes with $\delta = n$. For example, the simulation with our fault analysis algorithm shows that the probability of *safe* faults is 0.9908 for the $(10, 16, 2, 9)-$anticode and 0.9938 for the $(10, 16, 2, 6)-$anticode (the theoretical values follow the same trend). Also, for most of the cases, codes with unbounded maximum distance have the worst detection properties. In accordance with findings in Lemma 3, we can find the best performing anticodes with $n > \delta \geq \ell$ (see Equation (3)).

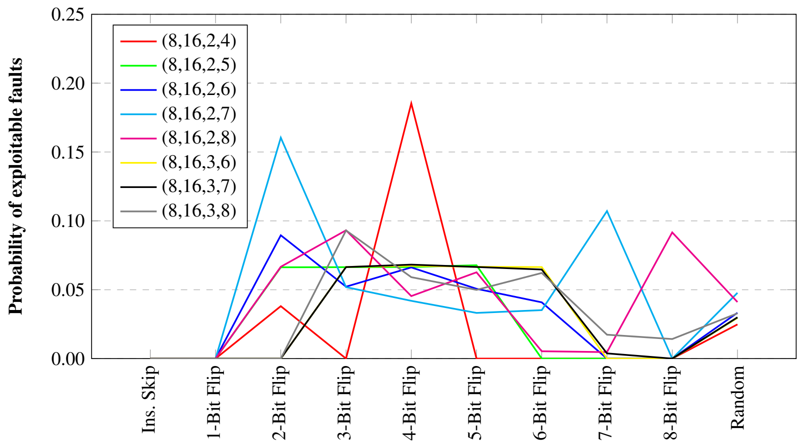

Fig. 6: Simulated results for anticodes with n = 8, d = 2, 3.

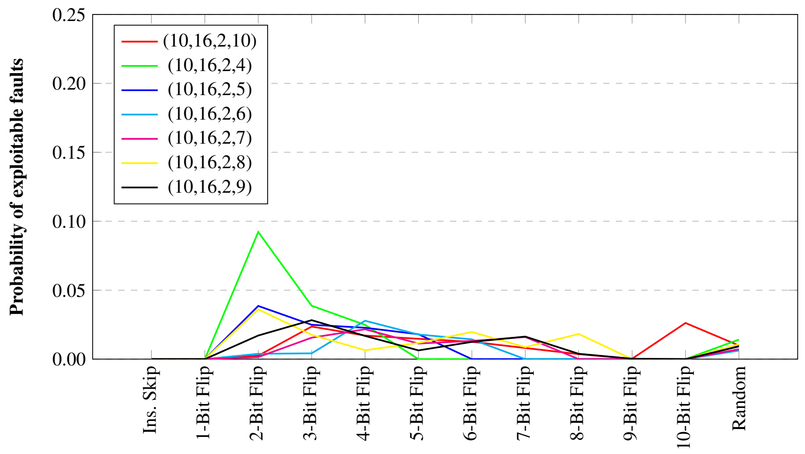

Fig. 7: Simulated results for anticodes with n = 10, d = 2.

6 Discussion

In this section, we would like to discuss the results obtained in Section 5, as well as some properties of the Fault Resilient Anticode Scheme.

Memory and Speed Trade-Offs Table 3 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 [25], out of which ≈ 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 [25]). 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 (≈ 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 (≈ 82.5% overhead). Therefore, compared to the most popular time redundancy that repeats the encryption twice and compares the results [2], 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 have a 16-bit addressing space, but only a binary anticode of length 6, the constructed table 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 lengths 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.

Table 2 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 5 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 2: 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

Comparison with Other Evaluation Methods Moro et al. [22] 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 design – 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. [33] 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. [17] 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 work by comparing the unprotected code with the protected one. However, while the duplication countermeasure was proven to be secure in this case, the work mentioned before ([33]) shows an experimental way how to break it. Therefore, it is necessary to provide simulated or experimental results when dealing with fault countermeasures since there might be hidden vulnerabilities that cannot be verified just by analyzing standalone code snippets.

In case of encoding based software countermeasures, there are no works proposing a full cipher evaluation to the best of our knowledge. The closest work to this one evaluates a single operation on encoded data [7]. Our method is universal for encoding based software countermeasures and provides details on all the possible bit flips and instruction skips. Also, the static code analysis technique that was implemented can efficiently evaluate a full cipher implementation in a short time.

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 [4]. As mentioned in [24], 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 [12]). 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 [7], along with the code properties. Similar properties could be derived for Fault Resilient Anticode Scheme in case such protection is necessary.

Open Problems Recall the notations from Definition 14. Let $(\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D) \in \mathfrak{S}$, C be an $(n, M, d, \delta)-$binary anticode associated with a decoding-encoding scheme ($\texttt{Encoder}_C$, $\texttt{Decoder}_C$) and let $(\mathcal{K}, \mathcal{P}, \mathcal{M} \cup \{\perp\}, E', D') := \Phi_C\big((\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)\big)$ be the corresponding Fault Resilient Anticode Scheme with $\mathcal{F}_1$ (resp. $\mathcal{F}_2$) being the assembly implementation of E' (resp. D'). Given $\Phi_C\big((\mathcal{K}, \mathcal{P}, \mathcal{M}, E, D)\big)$, $\mathcal{F}_1$ and $\mathcal{F}_2$, using the same theoretical calculation techniques of $p_{\text{rand},C}$ in Section 3.2, we can calculate the value p

such that $\left(\Phi_C\left((\mathcal{K},\mathcal{P},\mathcal{M},E,D)\right),\mathcal{F}_1,\mathcal{F}_2\right)$ is $p-$safe. Thus the goal is to achieve the maximum possible value of p. Note that, using a large number of "dummy" instructions in an implementation can reduce the percentage of exploitable faults. We exclude this strategy from all the discussions as this method would increase the computation time.

The primary open problem that naturally emerges can be stated as follows:
1. Given $(\mathcal{K},\mathcal{P},\mathcal{M},E,D)$, what is the best way to choose $(n,M,d,\delta)-$ binary anticodes, the associated decoding - encoding scheme $(\mathtt{Encoder}_C,\mathtt{Decoder}_C)$ and the design of the implementations $\mathcal{F}_1,\mathcal{F}_2$?

We have seen that the same implementation of Fault Resilient Anticode Scheme with PRESENT-80 behave differently under fault injection attacks for anticodes with different lengths and maximum/minimum distances. As we just mentioned, longer length gives better properties but it also requires bigger memory consumption. Another open question would be:
2. Given $(\mathcal{K},\mathcal{P},\mathcal{M},E,D)$, values of n,M, and implementations $\mathcal{F}_1,\mathcal{F}_2$, what is the best way to choose d and δ?

As mentioned in Section 3.2, it is not always possible to find a binary anticode for any values of n,M,d,δ. Considering the standard size of registers in current existing architectures is 64 bits, one would ask:
3. What is $N(n,d,\delta)$ (see Equation (4)) for $n \leq 64$?

The simulation results show that if the anticode has longer length, the fault resistance property of the Fault Resilient Anticode Scheme would be better. We conjecture that:
Conjecture: For any $\varepsilon > 0$ and any $(\mathcal{K},\mathcal{P},\mathcal{M},E,D) \in \mathfrak{S}$, there exists a binary anticode C, an encoding-decoding scheme $(\mathtt{Encoder}_C,\mathtt{Decoder}_C)$ and implementations $\mathcal{F}_1$, $\mathcal{F}_2$ such that $\left(\Phi_C\left((\mathcal{K},\mathcal{P},\mathcal{M},E,D)\right),\mathcal{F}_1,\mathcal{F}_2\right)$ is $p-$safe for some $p > 1-\varepsilon$.

7 Conclusion

In this paper, we have generalized fault resilient encoding schemes and provided a way to evaluate software implementations protected by encoding. We have practically implemented and evaluated symmetric block cipher PRESENT with encoded operations by using 8-bit microcontroller assembly code. We showed the benefits of using anticodes for cipher protection.

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

References

1. Akdemir, K.D., Wang, Z., Karpovsky, M., Sunar, B.: Design of cryptographic devices resilient to fault injection attacks using nonlinear robust codes. In: Fault Analysis in Cryptography, pp. 171–199. Springer (2012)
2. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. Proceedings of the IEEE 94(2), 370–382 (Feb 2006)
3. Barenghi, A., Breveglieri, L., Koren, I., Pelosi, G., Regazzoni, F.: Low-cost software countermeasures against fault attacks: Implementation and performances trade offs. In: Proc. of the 5th workshop on Embedded Security, WESS. pp. 7–1 (2010)

4. Bernstein, D.J.: Cache-timing attacks on AES. Tech. rep. (2005)
5. Biham, E., Shamir, A.: Differential Fault Analysis of Secret Key Cryptosystems. In: Kaliski, B.S. (ed.) Advances in Cryptology — CRYPTO '97: 17th Annual International Cryptology Conference Santa Barbara, California, USA August 17–21, 1997 Proceedings. pp. 513–525. Springer Berlin Heidelberg (1997)
6. Bogdanov, A., Knudsen, L., Leander, G., Paar, C., Poschmann, A., Robshaw, M., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In: Paillier, P., Verbauwhede, I. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2007, Lecture Notes in Computer Science, vol. 4727, pp. 450–466. Springer Berlin Heidelberg (2007)
7. Breier, J., Hou, X.: Feeding Two Cats with One Bowl: On Designing a Fault and Side-Channel Resistant Software Encoding Scheme. In: Handschuh, H. (ed.) Topics in Cryptology – CT-RSA 2017: The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14–17, 2017, Proceedings. pp. 77–94. Springer International Publishing (2017)
8. Breier, J., Jap, D., Bhasin, S.: A study on analyzing side-channel resistant encoding schemes with respect to fault attacks. Journal of Cryptographic Engineering (Jun 2017), `https://doi.org/10.1007/s13389-017-0166-5`
9. Bringer, J., Carlet, C., Chabanne, H., Guilley, S., Maghrebi, H.: Orthogonal Direct Sum Masking. In: Naccache, D., Sauveron, D. (eds.) 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. pp. 40–56. Springer Berlin Heidelberg (2014)
10. Ciet, M., Joye, M.: Practical Fault Countermeasures for Chinese Remaindering Based RSA (Extended Abstract). In: In Proceedings of Workshop on Fault Detection and Tolerance in Cryptography (FDTC'05). pp. 124–131 (2005)
11. Conway, J.H., Sloane, N.J.A.: Sphere Packings, Lattices and Groups, vol. 290. Springer Science & Business Media (2013)
12. Dobraunig, C., Eichlseder, M., Korak, T., Mangard, S., Mendel, F., Primas, R.: Exploiting ineffective fault inductions on symmetric cryptography. Cryptology ePrint Archive, Report 2018/071 (2018), `https://eprint.iacr.org/2018/071`
13. Dziembowski, S., Pietrzak, K., Wichs, D.: Non-Malleable Codes. In: ICS. pp. 434–452 (2010)
14. Farrell, P.: Linear Binary Anticodes. Electronics Letters 13(6), 419–421 (1970)
15. Fu, F.W., Kløve, T., Luo, Y., Wei, V.K.: On Equidistant Constant Weight Codes. Discrete applied mathematics 128(1), 157–164 (2003)
16. Galathy, N.F., Yuce, B., Schaumont, P.: A Systematic Approach to Fault Attack Resistant Design. In: Bhunia, S., Ray, S., Sur-Kolay, S. (eds.) Fundamentals of IP and SoC Security: Design, Verification, and Debug. Springer International Publishing (2017)
17. Goubet, L., Heydemann, K., Encrenaz, E., De Keulenaer, R.: Efficient esign and evaluation of countermeasures against fault attacks using formal verification. In: International Conference on Smart Card Research and Advanced Applications. pp. 177–192. Springer (2015)
18. Hoffstein, J., Pipher, J., Silverman, J.H., Silverman, J.H.: An Introduction to Mathematical Cryptography, vol. 1. Springer (2008)
19. Lee, P.A., Anderson, T.: Fault Tolerance: Principles and Practice. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edn. (1990)
20. Ling, S., Xing, C.: Coding Theory: A First Course. Cambridge University Press (2004)
21. MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error Correcting Codes. Elsevier (1977)
22. Moro, N., Heydemann, K., Dehbaoui, A., Robisson, B., Encrenaz, E.: Experimental evaluation of two software countermeasures against fault attacks. In: 2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST). pp. 112–117 (May 2014)

23. Moro, N., Heydemann, K., Encrenaz, E., Robisson, B.: Formal Verification of A Software Countermeasure Against Instruction Skip Attacks. Cryptology ePrint Archive, Report 2013/679 (2013), http://eprint.iacr.org/2013/679
24. Mukhopadhyay, D., Chakraborty, R.S.: Hardware Security: Design, Threats, and Safeguards. CRC Press (2014)
25. Papagiannopoulos, K., Verstegen, A.: Speed and size-optimized implementations of the present cipher for tiny avr devices. In: Hutter, M., Schmidt, J.M. (eds.) Radio Frequency Identification: Security and Privacy Issues 9th International Workshop, RFIDsec 2013, Graz, Austria, July 9-11, 2013, Revised Selected Papers. pp. 161–175. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-41332-2_11
26. Patranabis, S., Chakraborty, A., Mukhopadhyay, D.: Fault Tolerant Infective Countermeasure for AES. In: Chakraborty, R.S., Schwabe, P., Solworth, J. (eds.) Security, Privacy, and Applied Cryptography Engineering: 5th International Conference, SPACE 2015, Jaipur, India, October 3-7, 2015, Proceedings. pp. 190–209. Springer International Publishing (2015)
27. Patrick, C., Yuce, B., Ghalaty, N.F., Schaumont, P.: Lightweight Fault Attack Resistance in Software Using Intra-Instruction Redundancy. Cryptology ePrint Archive, Report 2016/850 (2016), http://eprint.iacr.org/2016/850
28. Rauzy, P., Guilley, S., Najm, Z.: Formally Proved Security of Assembly Code Against Leakage. IACR Cryptology ePrint Archive 2013, 554 (2013)
29. Riley, K.F., Hobson, M.P., Bence, S.J.: Mathematical Methods for Physics and Engineering: A Comprehensive Guide. Cambridge University Press (2006)
30. Schmidt, J.M., Medwed, M.: Countermeasures for Symmetric Key Ciphers. In: Joye, M., Tunstall, M. (eds.) Fault Analysis in Cryptography. pp. 73–87. Springer Berlin Heidelberg (2012)
31. Schneider, T., Moradi, A., Güneysu, T.: ParTI – Towards Combined Hardware Countermeasures Against Side-Channel and Fault-Injection Attacks. In: Robshaw, M., Katz, J. (eds.) Advances in Cryptology – CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II. pp. 302–332. Springer Berlin Heidelberg (2016)
32. Servant, V., Debande, N., Maghrebi, H., Bringer, J.: Study of a Novel Software Constant Weight Implementation. In: Joye, M., Moradi, A. (eds.) Smart Card Research and Advanced Applications: 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers. pp. 35–48. Springer International Publishing (2015)
33. Yuce, B., Ghalaty, N.F., Santapuri, H., Deshpande, C., Patrick, C., Schaumont, P.: Software Fault Resistance is Futile: Effective Single-Glitch Attacks. In: 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). pp. 47–58 (Aug 2016)

A Anticodes

Table 3: Table of $(n, 16, d, \delta)$−binary anticodes C generated from Algorithm 1. Each anticode C consists of codewords "**Codewords of** C", denoted by (n, M, d, δ) in Section 3.2, with overall resistance index $p_{\text{rand},C}$. The last column gives the percentage of safe faults under static code analysis for FRESH scheme using PRESENT-80 and C (Section 4.2).

Codewords of C	(n, M, d, δ)	$p_{\text{rand},C}$	Analysis
`0x1, 0x7B, 0x68, 0x22, 0xB8, 0x7, 0x46, 0x1A, 0x24, 0x29, 0x2E, 0x30, 0x33, 0x35, 0x36, 0x84`	$(8, 16, 2, 8)$	0.9421	0.9589
`0x1, 0x8, 0x2, 0xB, 0x4, 0x1D, 0x1E, 0x30, 0x7, 0x65, 0x6A, 0xAD, 0xB3, 0xCE, 0xD9, 0xF6`	$(8, 16, 2, 7)$	0.9548	0.9523
`0x1, 0x8F, 0x7D, 0x6, 0x2F, 0x3B, 0xC, 0x66, 0x1A, 0x1D, 0x20, 0x23, 0x34, 0x51, 0xDA, 0xE8`	$(8, 16, 2, 6)$	0.9605	0.9667
`0x1, 0x36, 0x50, 0xA2, 0xD2, 0x9A, 0x46, 0xC4, 0x8, 0xE, 0x17, 0x30, 0x83, 0x95, 0x9C, 0xA4`	$(8, 16, 2, 5)$	0.9643	0.9703
`0x1, 0x62, 0x64, 0x68, 0x70, 0xA2, 0xA4, 0xA8, 0xB0, 0xC2, 0xC4, 0xC8, 0xD0, 0xE3, 0xE5, 0xE9`	$(8, 16, 2, 4)$	0.9752	0.9619
`0x1, 0xAF, 0xFB, 0xA, 0x3C, 0xEC, 0xC0, 0x92, 0x17, 0x26, 0x4D, 0x54, 0x63, 0x99, 0xC7, 0xF5`	$(8, 16, 3, 8)$	0.9426	0.9671
`0x1, 0x37, 0x38, 0x42, 0x4C, 0x55, 0x5B, 0x6F, 0x8B, 0x9C, 0xA5, 0xAE, 0xB2, 0xD6, 0xE0, 0xF9`	$(8, 16, 3, 7)$	0.9599	0.9700
`0x1, 0x62, 0x6, 0x65, 0x18, 0x7B, 0x7C, 0xA8, 0x1F, 0xAF, 0xB1, 0xB6, 0xCA, 0xCD, 0xD3, 0xD4`	$(8, 16, 3, 6)$	0.9643	0.9703
$n = 9$			
`0x1, 0xD5, 0x1D6, 0x2E, 0x42, 0x158, 0x85, 0x11B, 0x106, 0x108, 0x10D, 0x110, 0x115, 0x11C, 0x120, 0x12A`	$(9, 16, 2, 9)$	0.9641	0.9734
`0x1, 0xF3, 0x167, 0xBD, 0xB0, 0x1D3, 0x25, 0xC5, 0x11C, 0x11F, 0x120, 0x123, 0x126, 0x139, 0x188, 0x1D8`	$(9, 16, 2, 8)$	0.9766	0.9839
`0x1, 0x1D9, 0x1A9, 0xA4, 0x1C2, 0x1B4, 0xD4, 0x10, 0x8D, 0x8E, 0x91, 0x92, 0x97, 0xEA, 0x10A, 0x17F`	$(9, 16, 2, 7)$	0.9799	0.9828

0x1, 0x180, 0x51, 0xD2, 0x110, 0xF8, 0x6A, 0x74, 0x16, 0x18, 0x1B, 0x26, 0x8D, 0x11C, 0x13B, 0x14C	$(9, 16, 2, 6)$	0.9819	0.9847
0x1, 0x115, 0x4C, 0x9F, 0x7D, 0x18D, 0x1D5, 0x99, 0x17, 0x25, 0x59, 0x94, 0xAD, 0xC1, 0xC7, 0xF5	$(9, 16, 2, 5)$	0.9814	0.9838
0x1, 0x2, 0x4, 0x8, 0x31, 0x32, 0x34, 0x51, 0x52, 0x58, 0x94, 0x98, 0xE0, 0x130, 0x150, 0x190	$(9, 16, 2, 4)$	0.9769	0.9796
0x1, 0x44, 0x18, 0x160, 0x9F, 0x1FA, 0xA0, 0x1A3, 0x116, 0x11B, 0x125, 0x12A, 0x13C, 0x143, 0x14D, 0x177	$(9, 16, 3, 9)$	0.9663	0.9749
0x1, 0x13C, 0x149, 0x1F6, 0x187, 0x1D3, 0x2F, 0x1E5, 0x70, 0x77, 0x82, 0x8C, 0x95, 0x9B, 0xE8, 0x132	$(9, 16, 3, 8)$	0.9791	0.9786
0x1, 0x27, 0xA, 0x1B3, 0x7E, 0x2C, 0xF0, 0xDF, 0xED, 0x104, 0x117, 0x118, 0x14B, 0x162, 0x1AA, 0x1C6	$(9, 16, 3, 7)$	0.9819	0.9849
0x1, 0x1E7, 0x8E, 0x42, 0x76, 0x11F, 0x1C4, 0x134, 0x2C, 0x55, 0x6F, 0x97, 0xA5, 0xB2, 0xDC, 0xF9	$(9, 16, 3, 6)$	0.9841	0.9866
0x1, 0x16, 0x17B, 0x2A, 0x198, 0x165, 0x18F, 0x142, 0x3D, 0x4C, 0x70, 0xA4, 0xB3, 0xD5, 0xE9, 0xFE	$(9, 16, 4, 9)$	0.9634	0.9788
0x1, 0xE4, 0x1B0, 0xBD, 0xCA, 0x179, 0x116, 0x1D5, 0x3A, 0x5C, 0x77, 0x12C, 0x14F, 0x162, 0x19B, 0x1A7	$(9, 16, 4, 8)$	0.9781	0.9869
0x1, 0xF8, 0x122, 0x1B4, 0x165, 0x76, 0x15F, 0x1EB, 0x3B, 0x4C, 0x97, 0xAD, 0xC2, 0x118, 0x18E, 0x1D1	$(9, 16, 4, 6)$	0.9841	0.9866
$n = 10$			
0x1, 0x399, 0x331, 0x2B3, 0xF6, 0x17D, 0x2C2, 0x294, 0x92, 0x95, 0x98, 0x9B, 0x9E, 0xA0, 0xA3, 0xCE	$(10, 16, 2, 10)$	0.9752	0.9901
0x1, 0x87, 0x176, 0x102, 0x1F8, 0x200, 0x38F, 0x108, 0x216, 0x218, 0x21B, 0x222, 0x225, 0x2CC, 0x2F3, 0x351	$(10, 16, 2, 9)$	0.9889	0.9908
0x1, 0x202, 0x27E, 0x45, 0x2DD, 0x38A, 0x23, 0x39B, 0x251, 0x252, 0x260, 0x267, 0x2AC, 0x314, 0x3B7, 0x3E9	$(10, 16, 2, 8)$	0.9897	0.9892
0x1, 0x46, 0x23D, 0x16E, 0x107, 0x25F, 0xE1, 0x2E7, 0x340, 0x343, 0x345, 0x349, 0x371, 0x384, 0x38A, 0x3B2	$(10, 16, 2, 7)$	0.9905	0.9928

0x1, 0x3AB, 0x14A, 0x20E, 0x1F, 0x15F, 0x23B, 0xAF, 0x8E, 0x92, 0x98, 0xCB, 0x122, 0x128, 0x26A, 0x383	$(10, 16, 2, 6)$	0.9912	0.9938
0x1, 0x24A, 0x8A, 0x298, 0x268, 0x25B, 0x109, 0x20F, 0x4C, 0x59, 0x200, 0x229, 0x28D, 0x2C1, 0x308, 0x3C9	$(10, 16, 2, 5)$	0.9898	0.9905
0x1, 0x381, 0x80, 0x140, 0x302, 0x182, 0x103, 0x304, 0x105, 0x108, 0x110, 0x121, 0x184, 0x1A0, 0x200, 0x320	$(10, 16, 2, 4)$	0.9835	0.9858
0x1, 0x6, 0x18, 0x1F, 0x2A, 0x2D, 0x33, 0x34, 0x4B, 0x4C, 0x52, 0x55, 0x60, 0x67, 0x79, 0x386	$(10, 16, 3, 10)$	0.9776	0.9914
0x1, 0x6, 0x18, 0x1F, 0x2A, 0x2D, 0x33, 0x4B, 0xD4, 0x1E0, 0x1FF, 0x2E6, 0x353, 0x37C, 0x385, 0x38A	$(10, 16, 3, 9)$	0.9896	0.9898
0x1, 0x112, 0x29A, 0x338, 0x283, 0x3C7, 0x27D, 0x389, 0x24B, 0x24C, 0x256, 0x2B5, 0x2EA, 0x33F, 0x3A4, 0x3F0	$(10, 16, 3, 8)$	0.9912	0.9927
0x2, 0x3A4, 0xD7, 0x143, 0x1FA, 0x3EB, 0x3F0, 0x283, 0xB9, 0xC0, 0xCD, 0xE3, 0x109, 0x131, 0x18E, 0x258	$(10, 16, 3, 7)$	0.9918	0.9927
0x1, 0xAC, 0x261, 0x22D, 0x59, 0x34C, 0x3C5, 0xCF, 0xC4, 0x107, 0x108, 0x1E9, 0x24A, 0x280, 0x28B, 0x29D	$(10, 16, 3, 6)$	0.9924	0.9930
0x1, 0x193, 0x277, 0xA2, 0x160, 0x3CA, 0x33E, 0xBF, 0xF8, 0x106, 0x118, 0x12B, 0x135, 0x14D, 0x1AC, 0x26C	$(10, 16, 4, 10)$	0.9779	0.9839
0x1, 0x3B1, 0x2BC, 0x156, 0x32F, 0x9B, 0x340, 0x35D, 0xE0, 0xEF, 0x138, 0x1A6, 0x20A, 0x273, 0x2C5, 0x3DA	$(10, 16, 4, 9)$	0.9904	0.9925
0x1, 0x304, 0x3DF, 0xFC, 0x86, 0xE3, 0x28B, 0x295, 0x10A, 0x11D, 0x177, 0x1D0, 0x238, 0x26E, 0x3B2, 0x3E9	$(10, 16, 4, 8)$	0.9911	0.9918
0x1, 0x2EF, 0x3A3, 0x18C, 0x395, 0x1B6, 0x370, 0x244, 0x75, 0xB8, 0x11F, 0x169, 0x1C2, 0x21A, 0x32E, 0x3DB	$(10, 16, 4, 7)$	0.9923	0.9932
0x1, 0xE, 0x32, 0x3D, 0xC4, 0xCB, 0xF7, 0xF8, 0x150, 0x15F, 0x163, 0x16C, 0x195, 0x19A, 0x1A6, 0x256	$(10, 16, 4, 6)$	0.9929	0.9939

B Additional Proofs

B.1 Proof of Lemma 1

Proof. By Definition 7, since $g_2 \circ g_1 \in \mathcal{S}$, $\exists M_1, M_2, \ldots, M_{m+2} \in \mathbb{Z}_{>0}$ s.t.

$$g_1 : \mathbb{F}_2^{M_1} \times \mathbb{F}_2^{M_2} \times \cdots \times \mathbb{F}_2^{M_m} \to \mathbb{F}_2^{M_{m+1}}, \quad g_2 : \mathbb{F}_2^{M_{m+1}} \to \mathbb{F}_2^{M_{m+2}}.$$

For $1 \le i \le m+2$, take k_i such that $\{\mathtt{Encoder}_C(\boldsymbol{x}) | \boldsymbol{x} \in \mathbb{F}_2^{M_i}\} = C^{k_i} \subseteq \mathbb{F}_2^{nk_i}$, then

$$\varphi_C(g_1) : \left(\mathbb{F}_2^{nk_1} \cup \{\perp\}\right) \times \left(\mathbb{F}_2^{nk_2} \cup \{\perp\}\right) \times \cdots \times \left(\mathbb{F}_2^{nk_m} \cup \{\perp\}\right) \to C^{k_{m+1}} \cup \{\perp\},$$

such that for $\boldsymbol{x} = \left(\mathtt{Encoder}_C(\boldsymbol{x}_1), \mathtt{Encoder}_C(\boldsymbol{x}_2), \ldots, \mathtt{Encoder}_C(\boldsymbol{x}_m)\right) \in C^{k_1} \times C^{k_2} \times \ldots C^{k_m}$, $\varphi_C(g_1)(\boldsymbol{x}) = \mathtt{Encoder}_C\left(g_1(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m)\right)$ and $\forall \boldsymbol{x} \in \left(\mathbb{F}_2^{nk_1} \cup \{\perp\}\right) \times \left(\mathbb{F}_2^{nk_2} \cup \{\perp\}\right) \times \cdots \times \left(\mathbb{F}_2^{nk_m} \cup \{\perp\}\right) \backslash C^{k_1} \times C^{k_2} \times \ldots C^{k_m}$, $\varphi_C(g_1)(\boldsymbol{x}) = \perp$. Moreover

$$\varphi_C(g_2) : \mathbb{F}_2^{nk_{m+1}} \cup \{\perp\} \to C^{k_{m+2}} \cup \{\perp\},$$

such that for $\boldsymbol{y} = \mathtt{Encoder}_C(\boldsymbol{a}) \in C^{k_{m+1}}$, $\varphi_C(g_2)(\boldsymbol{y}) = \mathtt{Encoder}_C(g_2(\boldsymbol{a}))$ 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

$$\left(\mathbb{F}_2^{nk_1} \cup \{\perp\}\right) \times \left(\mathbb{F}_2^{nk_2} \cup \{\perp\}\right) \times \cdots \times \left(\mathbb{F}_2^{nk_m} \cup \{\perp\}\right) \to C^{k_{m+2}} \cup \{\perp\}$$

such that for $\boldsymbol{x} = \left(\mathtt{Encoder}_C(x_1), \mathtt{Encoder}_C(x_2), \ldots, \mathtt{Encoder}_C(x_m)\right) \in C^{k_1} \times C^{k_2} \times \ldots C^{k_m}$,

$$\begin{aligned}
\left(\varphi_C(g_2) \circ \varphi_C(g_1)\right)(\boldsymbol{x}) &= \varphi_C(g_2)\left(\varphi_C(g_1)(\boldsymbol{x})\right) \\
&= \varphi_C(g_2)\left(\mathtt{Encoder}_C(g_1(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m))\right) \\
&= \mathtt{Encoder}_C\left(g_2(g_1(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m))\right) \\
&= \mathtt{Encoder}_C\left(g_2 \circ g_1(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m)\right),
\end{aligned}$$

and $\forall \boldsymbol{x} \in \left(\mathbb{F}_2^{nk_1} \cup \{\perp\}\right) \times \left(\mathbb{F}_2^{nk_2} \cup \{\perp\}\right) \times \cdots \times \left(\mathbb{F}_2^{nk_m} \cup \{\perp\}\right) \backslash C^{k_1} \times C^{k_2} \times \ldots C^{k_m}$,

$$\left(\varphi_C(g_2) \circ \varphi_C(g_1)\right)(\boldsymbol{x}) = \varphi_C(g_2)\left(\varphi_C(g_1)(\boldsymbol{x})\right) = \varphi_C(g_2)(\perp) = \perp .$$

On the other hand,

$$g_2 \circ g_1 : \mathbb{F}_2^{M_1} \times \mathbb{F}_2^{M_2} \times \cdots \times \mathbb{F}_2^{M_m} \to \mathbb{F}_2^{M_{m+2}},$$

and $\varphi_C(g_2 \circ g_1)$ is a map:

$$\left(\mathbb{F}_2^{nk_1} \cup \{\perp\}\right) \times \left(\mathbb{F}_2^{nk_2} \cup \{\perp\}\right) \times \cdots \times \left(\mathbb{F}_2^{nk_m} \cup \{\perp\}\right) \to C^{k_{m+2}} \cup \{\perp\}$$

such that for $\boldsymbol{x} = \left(\mathtt{Encoder}_C(\boldsymbol{x}_1), \mathtt{Encoder}_C(\boldsymbol{x}_2), \ldots, \mathtt{Encoder}_C(\boldsymbol{x}_m)\right) \in C^{k_1} \times C^{k_2} \times \ldots C^{k_m}$,

$$\left(\varphi_C(g_2 \circ g_1)\right)(\boldsymbol{x}) = \mathtt{Encoder}_C\left(g_2 \circ g_1(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m)\right),$$

and $\forall \boldsymbol{x} \in \left(\mathbb{F}_2^{nk_1} \cup \{\perp\}\right) \times \left(\mathbb{F}_2^{nk_2} \cup \{\perp\}\right) \times \cdots \times \left(\mathbb{F}_2^{nk_m} \cup \{\perp\}\right) \backslash C^{k_1} \times C^{k_2} \times \ldots C^{k_m}$, $\left(\varphi_C(g_2 \circ g_1)\right)(\boldsymbol{x}) = \perp$. □

B.2 Proof of Lemma 3

Proof. We prove 1-a and 2-a, the other cases can be proved similarly.
Let x be such that $\binom{n}{x} = \min\{\binom{n}{m} : d_1 \le m \le \delta_1\}$, then by the assumption that $\delta_1 + d_1 + 1 > n$ and Equation (3), we have $\binom{n}{x} > \binom{n}{\delta_1+1}$ and $\binom{n}{x} > \binom{n}{d_1-1}$.
Using the notations in Definition 15, we have

$$\begin{aligned} p_{\mathrm{rand},C_1} - p_{\mathrm{rand},C_2} &= \sum_{m=1}^{n} \frac{1}{n}(p_{m,C_1} - p_{m,C_2}) \\ &= \sum_{m=1}^{n} \frac{1}{nM\binom{n}{m}} (S_{m,C_2} - S_{m,C_1}). \end{aligned}$$

1-a Let $S := \sum_{m=\delta_1+1}^{\delta_2} S_{m,C_2}$, $d := d_1 = d_2$. Since

$$\sum_{m=d}^{\delta_1} S_{m,C_2} + \sum_{m=\delta_1+1}^{\delta_2} S_{m,C_2} = \sum_{m=d}^{\delta_1} S_{m,C_1} = 2M,$$

we have $\sum_{m=d}^{\delta_1}(S_{m,C_2} - S_{m,C_1}) = -S$ with $S_{m,C_2} - S_{m,C_1} \le 0$ for each $m = d, \ldots, \delta_1$. Hence $p_{\mathrm{rand},C_1} - p_{\mathrm{rand},C_2}$ is given by

$$\begin{aligned} &\sum_{m=d}^{\delta_1} \frac{1}{nM\binom{n}{m}}(S_{m,C_2} - S_{m,C_1}) + \sum_{m=\delta_1+1}^{\delta_2} \frac{1}{nM\binom{n}{m}} S_{m,C_2} \\ \ge\ & \frac{1}{nM\binom{n}{x}} \sum_{m=d}^{\delta_1}(S_{m,C_2} - S_{m,C_1}) + \frac{1}{nM\binom{n}{\delta_1+1}} \sum_{m=\delta_1+1}^{\delta_2} S_{m,C_2} \\ =\ & -\frac{S}{nM\binom{n}{x}} + \frac{S}{nM\binom{n}{\delta_1+1}} = \frac{S}{nM}\left(\frac{1}{\binom{n}{\delta_1+1}} - \frac{1}{\binom{n}{x}}\right) > 0. \end{aligned}$$

2-a Let $S := \sum_{m=d_2}^{d_1-1} S_{m,C_2}$, $\delta := \delta_1 = \delta_2$. Since

$$\sum_{m=d_2}^{d_1-1} S_{m,C_2} + \sum_{m=d_1}^{\delta} S_{m,C_2} = \sum_{m=d_1}^{\delta} S_{m,C_1} = 2M,$$

we have $\sum_{m=d_1}^{\delta}(S_{m,C_2} - S_{m,C_1}) = -S$ with $S_{m,C_2} - S_{m,C_1} \le 0$ for each $m = d_1, \ldots, \delta$. Hence $p_{\mathrm{rand},C_1} - p_{\mathrm{rand},C_2}$ is given by

$$\begin{aligned} &\sum_{m=d_2}^{d_1-1} \frac{1}{nM\binom{n}{m}} S_{m,C_2} + \sum_{m=d_1}^{\delta} \frac{1}{nM\binom{n}{m}}(S_{m,C_2} - S_{m,C_1}) \\ \ge\ & \frac{1}{nM\binom{n}{d_1-1}} \sum_{m=d_2}^{d_1-1} S_{m,C_2} + \frac{1}{nM\binom{n}{x}} \sum_{m=d_1}^{\delta}(S_{m,C_2} - S_{m,C_1}) \\ =\ & \frac{S}{nM\binom{n}{d_1-1}} - \frac{S}{nM\binom{n}{x}} = \frac{S}{nM}\left(\frac{1}{\binom{n}{d_1-1}} - \frac{1}{\binom{n}{x}}\right) > 0. \end{aligned}$$

B.3 Proof of Lemma 4

Proof. i and ii easily follow from the definitions.

iii,iv. Let C be an $(n, d, \delta)-$binary anticode. For any $\boldsymbol{c} = (c_1, c_2, \dots, c_n) \in C$, define $\tilde{\boldsymbol{c}} := (c_1, c_2, \dots, c_n, 1)$ 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 $\operatorname{dis}(\boldsymbol{c}_1, \boldsymbol{c}_2) = d$ and $\operatorname{dis}(\boldsymbol{c}_3, \boldsymbol{c}_4) = \delta$. Without loss of generality, we can assume $\boldsymbol{c}_3 \neq \boldsymbol{c}_1$ and $\boldsymbol{c}_3 \neq \boldsymbol{c}_2$. Suppose $\boldsymbol{c}_3 = (x_1, x_2, \dots, x_n)$ and take

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

Then $\operatorname{dis}((x_1, x_2, \dots, x_n, 0), \tilde{\boldsymbol{c}}_4) = \delta + 1$, $\operatorname{dis}(\tilde{\boldsymbol{c}}_1, \tilde{\boldsymbol{c}}_2) = d$ and $\forall \boldsymbol{x}, \boldsymbol{y} \in C''$, $d \leq \operatorname{dis}(\boldsymbol{x}, \boldsymbol{y}) \leq \delta + 1$. 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. $\operatorname{dis}(\boldsymbol{c}_1, \boldsymbol{c}_2) = d+1$ and $\operatorname{dis}(\boldsymbol{c}_3, \boldsymbol{c}_4) = \delta$. 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 $\boldsymbol{c} = (c_1, c_2, \dots, c_n) \in C$, define $\tilde{\boldsymbol{c}} := (1, c_2, c_3, \dots, c_n)$ and let $C' := \{\tilde{\boldsymbol{c}} : \boldsymbol{c} \in C\}$. Then $\operatorname{dis}(\tilde{\boldsymbol{c}}_1, \tilde{\boldsymbol{c}}_2) = d$ and $\forall \boldsymbol{x}, \boldsymbol{y} \in C'$, $d \leq \operatorname{dis}(\boldsymbol{x}, \boldsymbol{y}) \leq \delta$. If C' is an $(n, d, \delta)-$binary anticode, then we're done. Otherwise, $\operatorname{dis}(\tilde{\boldsymbol{c}}_3, \tilde{\boldsymbol{c}}_4) = \delta - 1$. Suppose $\boldsymbol{c}_3 = (x_1, x_2, \dots, x_n)$ and take $C'' := \Big(C' \backslash \{\tilde{\boldsymbol{c}}_3\}\Big) \cup \{(0, x_2, \dots, x_n)\}\}$, then $\operatorname{dis}((0, x_2, \dots, x_n), \tilde{\boldsymbol{c}}_4) = \delta$ and C'' is an $(n, d, \delta)-$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 $\operatorname{dis}(\boldsymbol{c}_1, \boldsymbol{c}_2) = 2r - 1$ and $\operatorname{dis}(\boldsymbol{c}_3, \boldsymbol{c}_4) = \delta$.
We add one parity check bit for each codeword in C to get a binary anticode C': For any $\boldsymbol{c} = (c_1, c_2, \dots, c_n) \in C$, define $\tilde{\boldsymbol{c}} := (c_1, c_2, \dots, c_n, c_1 + c_2 + \cdots + c_n \bmod 2)$ and let $C' := \{\tilde{\boldsymbol{c}} : \boldsymbol{c} \in C\}$. Since $2r - 1$ is odd, $\operatorname{dis}(\tilde{\boldsymbol{c}}_1, \tilde{\boldsymbol{c}}_2) = 2r$ and $\forall \boldsymbol{x}, \boldsymbol{y} \in C$, $\operatorname{dis}(\boldsymbol{x}, \boldsymbol{y}) \geq 2r$.
If $\delta = 2\ell - 1$ is odd, $\operatorname{dis}(\tilde{\boldsymbol{c}}_3, \tilde{\boldsymbol{c}}_4) = 2\ell$ and $\forall \boldsymbol{x}, \boldsymbol{y} \in C'$, $\operatorname{dis}(\boldsymbol{x}, \boldsymbol{y}) \leq 2\ell$. 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 $\operatorname{dis}(\boldsymbol{x}, \boldsymbol{y}) = \delta$, $\operatorname{dis}(\boldsymbol{x}', \boldsymbol{y}') = \delta$ and we have C' is an $(n, M, 2r, 2\ell)-$binary anticode. This proves vii.

C Further Results on Fault Analysis

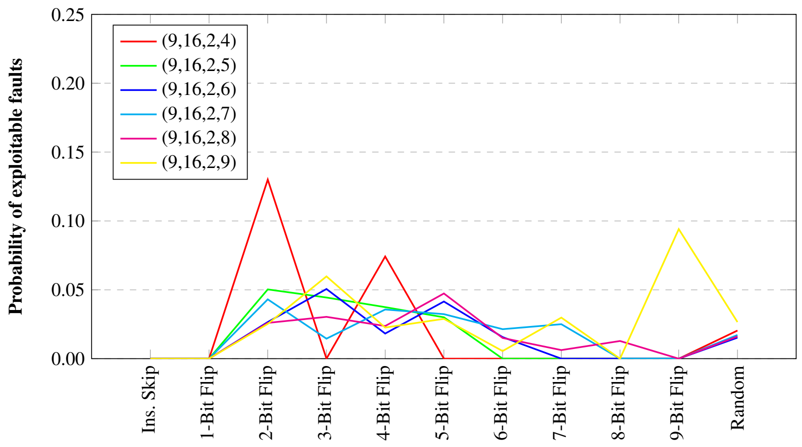

Fig. 8: Simulated results for anticodes with $n = 9, d = 2$.

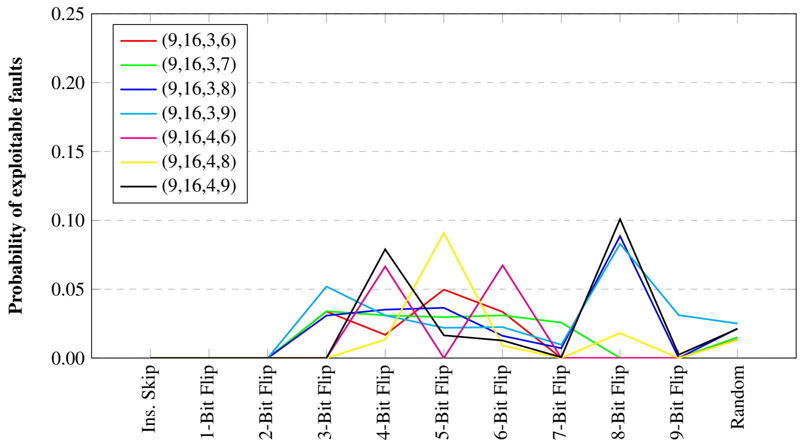

Fig. 9: Simulated results for anticodes with $n = 9, d = 3, 4$.

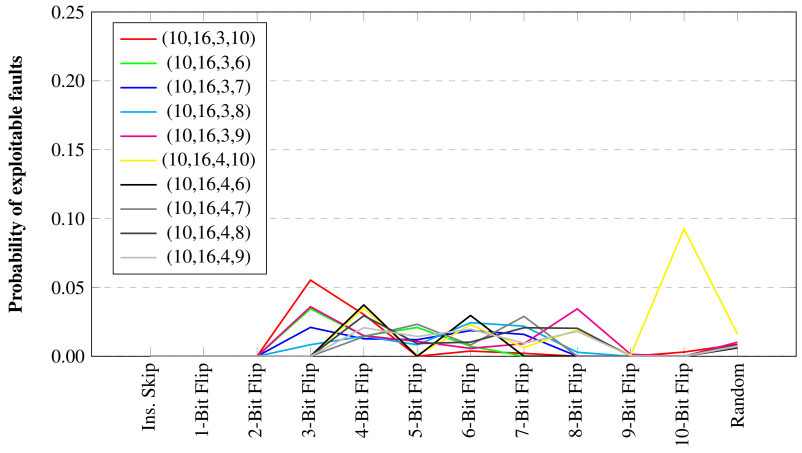

Fig. 10: Simulated results for anticodes with $n = 10, d = 3, 4$.