Chapter 7
Automated Deployment of Software Encoding Countermeasure

Jakub Breier and Xiaolu Hou

7.1 Introduction

As it was shown before [15], fault countermeasures often lower the implementation resistance against side-channel attacks. Therefore, it is necessary to consider these two classes of physical attacks together when protecting the algorithm. In this chapter, we show how to automatically construct a code-based countermeasure that significantly reduces the success of a fault injection attack while keeping low information leakage via side-channels.

There are two main countermeasure classes to protect implementations against side-channel attacks. *Masking* [9] is a software-level countermeasure which tries to "mask" the relationship between the intermediate values and power leakage. *Hiding* [20] tries to reduce the signal and increase noise by utilizing various techniques—it "hides" the operations performed by the device. While masking can make fault attacks more challenging, it does not help to prevent them. On the other hand, some hiding techniques, such as dual-rail precharge logic (DPL), help in preventing fault attacks by detecting faults [18].

In 2011, DPL was extended to software by Hoogvorst et al. [10], by using balanced encoding schemes. Since then, there were several other proposals

This research was conducted when author "J. Breier" was with Temasek Laboratories, NTU.
This research was conducted when author "X. Hou" was with Nanyang Technological University.

J. Breier
Underwriters Laboratories, Singapore, Singapore
e-mail: jbreier@jbreier.com

X. Hou (✉)
Acronis, Singapore, Singapore
e-mail: ho0001lu@e.ntu.edu.sg

J. Breier et al. (eds.), *Automated Methods in Cryptographic Fault Analysis*,
https://doi.org/10.1007/978-3-030-11333-9_7

[6, 13, 14, 19], all of them using various coding techniques to prevent side-channel leakage. However, it was shown that unlike hardware DPL representation, its software counterpart is not fault resistant by default [4]. Therefore, to prevent both attack techniques, it is necessary to design the coding scheme from the beginning with this goal in mind.

In this chapter, we focus on approach presented in [2]. We first explain the theoretical background necessary for designing software hiding countermeasures that are resistant to both side-channel and fault attacks. We provide an algorithm to automatically find optimal codes for various code distances and number of codewords with given code length. We also provide evaluation of the codes—by using detection and correction probabilities and by an automated fault simulator. This simulation is done by using a general-purpose microcontroller implementation and an instruction set simulator that is capable of injecting different fault models into any instruction of the code. Our evaluation shows that the codes generated by our algorithm provide a high security level with respect to both side-channel and fault attacks.

The rest of the chapter is organized as follows: Section 7.2 provides an overview of the related work in this field, together with necessary background on coding theory. Section 7.3 defines the properties of codes with respect to fault attacks. Section 7.4 details our algorithm and provides estimated and simulated results on chosen codes. These results are further discussed in Sect. 7.5. Finally, Sect. 7.6 summarizes this chapter.

7.2 General Background

In this section we provide a necessary background on software encoding-based side-channel countermeasures and on coding theory necessary for developing a combined countermeasure. Section 7.2.1 overviews the related work in the field. Section 7.2.2 provides basic definitions that are used later in this chapter.

7.2.1 Related Work

After the paper by Hoogvorst et al. [10], who presented a method to extend the DPL to software implementations, several works were published in the area of software hiding schemes.

Rauzy et al. [14] developed a scheme that encodes the data by using bit-slicing, where only one bit of information is processed at a time. They claim that this kind of protection is 250 times more resistant to power analysis attacks compared to the unprotected implementation, while being 3 times slower. For testing, they used PRESENT cipher, running on an 8-bit microcontroller.

Chen et al. [6] proposed an encoding scheme that adds a complementary bit to each bit of the processed data, resulting in a constant Hamming weight code. Their countermeasure was implemented on a Prince cipher, using an 8-bit microcontroller.

Servant et al. [19] introduced a constant weight implementation for AES, by using a (3,6) code. To improve the performance, they split 8-bit variables into two 4 bit words and encode them separately. This implementation was also capable of detecting faults with 93.75% probability. Their implementation used a 16-bit microcontroller.

Maghrebi et al. [13] proposed an encoding scheme that differs from the previous proposals. For their case, they did not assume the Hamming weight leakage model for register bits; therefore, they concluded that balanced codes might not be the optimal ones to use generally. In their method, they first obtain the profile of a device to get a vector of register bit leakages. Then they estimate leakage values for each codeword and build a code by using codewords with the lowest leakage. Their algorithm selects the optimal code by ranking the codes based on the difference in power consumption between the codewords and on the power consumption variance. Our algorithm extends this idea by adding the variance of register bits in order to achieve better leakage characteristics and by adding conditions for error detection and correction.

In general, none of the previous schemes has been designed for fault resistance. Schemes proposed in [6, 14] have been analyzed with respect to fault attacks by Breier et al. [4], concluding that without additional modifications to assembly code, the probability of a successful fault attack is non-negligible. Therefore, in this chapter we focus on design and automated generation of fault tolerant and side-channel resistant coding schemes.

When it comes to combined countermeasures, in [17], Schneider et al. proposed a hardware countermeasure based on combining threshold implementation with linear codes. As stated in the paper, their proposal is not considered for software targets. In the execution process, there are multiple checking steps that protect the implementation against faults. However, in software, it would be easy to overcome such checks by multiple fault injections [21]. Also, it would be possible to inject faults that are impossible with hardware implementations, such as instruction skips [3].

This chapter provides the reader with the following information:

- We specify theoretical bounds for encoding schemes with respect to fault attacks that are necessary to be taken into account when designing a fault resistant scheme.
- We show how to automatically design a code that is capable of protecting the implementation against side-channel and fault attacks and we show trade-offs between these two resistances.
- We adopt the ranking algorithm proposed in [13] and show how to improve it for constructing side-channel resistant codes with better properties—by ranking the codes according to the codeword with the highest leakage, and by calculating

the register bit variance. We add the conditions for selecting the codes with the desired error-detection/correction capabilities in an automated way.
- We analyze the codes constructed by the code generation algorithm—we calculate leakages, fault detection, and correction probabilities, and we simulate the assembly code implementing the codes on a general-purpose microcontroller.

7.2.2 Coding Theory Background

A *binary code*, denoted by C, is a subset of the n-dimensional vector space over $\mathbb{F}_2$-$\mathbb{F}_2^n$, 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* [11, p. 6]. Take two codewords $\boldsymbol{c}, \boldsymbol{c}' \in C$, the *Hamming distance* between $\boldsymbol{c}$ and $\boldsymbol{c}'$, denoted by $\text{dis}\left(\boldsymbol{c}, \boldsymbol{c}'\right)$, is defined to be the number of places at which $\boldsymbol{c}$ and $\boldsymbol{c}'$ differ [11, p. 9]. More precisely, if $\boldsymbol{c} = c_1 c_2 \ldots c_n$ and $\boldsymbol{c}' = c_1' c_2' \ldots c_n'$, then

$$\text{dis}\left(\boldsymbol{c}, \boldsymbol{c}'\right) = \sum_{i=1}^{n} \text{dis}\left(c_i, c_i'\right),$$

where c_i and c_i' are treated as binary words of length 1 and hence

$$\text{dis}\left(c_i, c_i'\right) = \begin{cases} 1 & \text{if } c_i \neq c_i' \\ 0 & \text{if } c_i = c_i' \end{cases}.$$

Furthermore, for a binary code C, the *(minimum) distance* of C, denoted by $\text{dis}\,(C)$, is [11, p. 11]

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

Definition 7.1 ([7, p. 75]) For a binary code C of length n, $\text{dis}\,(C) = d$, let $M = |C|$ denote the number of codewords in C. Then C is called an (n, M, d)-binary code.

This minimum distance of a binary code is closely related to the error-detection and error-correction capabilities of C.

Definition 7.2 ([11, p. 12]) Let u be a positive integer. C is said to be *u-error-detecting* if, whenever there is at least one but at most u errors that occur in a codeword in C, the resulting word is not in C.

From the definition, it is easy to prove that C is u-error-detecting if and only if $\text{dis}\,(C) \geq u + 1$ [11, p. 12]. A common decoding method that is used is *nearest neighbor decoding*, which decodes a word $\boldsymbol{x} \in \mathbb{F}_2^n$ to the codeword $\boldsymbol{c}_{\boldsymbol{x}}$ such that

$$\text{dis}\,(\boldsymbol{x}, \boldsymbol{c}_{\boldsymbol{x}}) = \min_{\boldsymbol{c} \in C} \text{dis}\,(\boldsymbol{x}, \boldsymbol{c}). \tag{7.1}$$

When there are more codewords $\boldsymbol{c_x}$ satisfies (7.1), the *incomplete decoding rule* requires a retransmission [11, p. 10].

Definition 7.3 ([11, p. 13]) Let v be a positive integer. C is *v-error correcting* if minimum distance decoding with incomplete decoding rule is applied, v or fewer errors can be corrected.

Remark 7.1 C is v-error correcting if and only if $\text{dis}\,(C) \geq 2v + 1$ [11, p. 13].

Definition 7.4 ([8]) An (n, M, d)-binary code C is called an *equidistant code* if $\forall \boldsymbol{c}, \boldsymbol{c}' \in C,\ \text{dis}\left(\boldsymbol{c}, \boldsymbol{c}'\right) = \text{dis}\,(C)$.

For our purpose, we will use binary code for protecting the underlying implementation.

We propose two choices of look-up tables:

1. Correction Table: This table will treat a word $\boldsymbol{x} \in \mathbb{F}_2^n$ the same as the codeword $\boldsymbol{c_x} \in C$ which satisfies $\text{dis}\,(\boldsymbol{c_x}, \boldsymbol{x}) \leq \lfloor \frac{d-1}{2} \rfloor$, where d is the distance of C. Note that this is equivalent to using *bounded distance decoding* [12, p. 36] and taking the bounded distance to be $\lfloor \frac{d-1}{2} \rfloor$. To use this table we require that $\text{dis}\,(C) \geq 3$.
2. Detection Table: This is a normal look-up table that returns a null value when $\boldsymbol{x} \notin C$ is accessed.

We will give a theoretical criterion to measure the bit flip fault resistant capability of a binary code when it is used as an encoding countermeasure against fault injection attacks in Sect. 7.3. Afterwards we propose three coding schemes. The encoding scheme will be simulated (and implemented) and evaluated in Sect. 7.4.

Let m be a positive integer such that $1 \leq m \leq n$, where n is the code length.

Definition 7.5 An *m-bit fault* is a fault injected in the codeword that flips exactly m bits. We assume each bit has equal probability to be flipped.

Definition 7.6 When the fault is analyzed, we adopt the following terminologies:

- *Corrected*: Fault is detected and corrected.
- *Null*: Fault is detected and results into zero output.
- *Invalid*: Fault is detected and results into an output that is not a codeword.
- *Valid*: Fault is not detected and fault injection is successful, i.e., it results in the output of a valid but incorrect codeword.

7.3 Theoretical Analysis

In this section we will first give the theoretical analysis for the fault resistant capabilities of binary code in general. Then we propose two different coding schemes and analyze their fault resistant probabilities.

7.3.1 Correction Table

Definition 7.7 For an (n, M, d)-binary code C such that $d \geq 3$, let

$$F_{\boldsymbol{c},m} := \left\{ \boldsymbol{x} \in \mathbb{F}_2^n : \text{dis}\,(\boldsymbol{c}, \boldsymbol{x}) = m \text{ and } \exists \boldsymbol{c}' \in C \text{ such that dis}\left(\boldsymbol{x}, \boldsymbol{c}'\right) \leq \left\lfloor \frac{d-1}{2} \right\rfloor \right\}.$$

Then

$$p_{m,(e)} := \begin{cases} 1 & m \leq \lfloor \frac{d-1}{2} \rfloor \\ 1 - \frac{1}{M\binom{n}{m}} \sum_{\boldsymbol{c} \in C} |F_{\boldsymbol{c},m}| & m > \lfloor \frac{d-1}{2} \rfloor \end{cases} \tag{7.2}$$

is called the *m-bit fault resistance probability with error correction* for C.

As mentioned earlier, when a Correction Table is used, it is equivalent to using bounded distance decoding. When $m \leq \lfloor \frac{d-1}{2} \rfloor$ bits are flipped, by Remark 7.1, the error will be corrected and hence $p_{m,(e)} = 1$. When $m > \lfloor \frac{d-1}{2} \rfloor$ bits are flipped, the fault will be valid if the resulting word is at distance at most $\lfloor \frac{d-1}{2} \rfloor$ from any codeword. Thus by Definition 7.6, $1 - p_{m,(e)}$ gives the theoretical probability of a *Valid* fault and the bigger the $p_{m,(e)}$ is, the more resistant the binary code to m-bit fault. Furthermore, when $m = 1$, the fault will be corrected and most of the cases are expected to return *Corrected*.

Another interesting fault model is random fault, i.e., assuming there is an equal probability for m-bits fault to occur $\forall 1 \leq m \leq n$. Taking this into account, we define the following.

Definition 7.8 For an (n, M, d)-binary code C such that $d \geq 3$, let $p_{m,(e)}$ be its m-bit fault resistance probability with error for $1 \leq m \leq n$, then

$$p_{\text{rand},(e)} := \sum_{m=1}^{n} \frac{1}{n} p_{m,(e)}$$

is called the *overall resistance index with error correction* for C.

As suggested by the name, the bigger the $p_{\text{rand},(e)}$ is, the more resistant the code C is to random faults.

7.3.2 Detection Table

Now we consider Detection Table.

Definition 7.9 For an (n, M, d)-binary code C such that $d \geq 2$, let

$$S_m := \sum_{\boldsymbol{c} \in C} |\{\boldsymbol{c}' \in C : \text{dis}\left(\boldsymbol{c}', \boldsymbol{c}\right) = m\}|.$$

Then

$$p_m := 1 - \frac{S_m}{M\binom{n}{m}} \tag{7.3}$$

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

When an m-bit fault is injected in the codeword, if the resulting word is not a codeword then the value will be set to *Null*. The only case when the fault is valid is when after m bits are flipped, the resulting word is still a codeword. Thus by Definition 7.6, $1 - p_m$ gives the theoretical probability of a *Valid* fault. Hence, the bigger the p_m, the better the m-fault resistance of the binary code.

Remark 7.2 When $m \leq d$, no codeword is at distance m from each other and hence $p_m = 1$.

Note that if $S_n = M$, i.e., for each codeword $\boldsymbol{c} \in C$, there exists a $\boldsymbol{c}' \in C$ such that $\text{dis}(\boldsymbol{c}, \boldsymbol{c}') = n$, then we have

$$p_n = 1 - \frac{M}{M\binom{n}{n}} = 1 - 1 = 0.$$

That means, for this code, n-bit fault will always be injected successfully. In view of this, we exclude these kind of codes from our selection (see Algorithm 1). In practice, n and M are the fixed known values, from Eq. (7.3), to get bigger p_m the goal of choosing the code C is to make S_m small. There are several ways of achieving this depending on the preference of the user:

1. For small values of m, make $p_m = 0$: Choose code with a bigger minimum distance d, then p_m will be 1 for more values of m. Of course, there is a limit for the minimum distance that can be achieved (see Table 7.1). This particular scheme will be discussed in Sect. 7.3.3, where it is called Detection Scheme.
2. A certain m_0-bit fault resistance is desired: Choose code such that $S_{m_0} = 0$.
3. Sacrificing one m_0-bit fault resistance to achieve m-bit fault resistance for all other values of $m \neq m_0$: This is possible by using equidistant codes. That is, take code such that $|S_{m_0}| = M$. This particular scheme will be discussed in Sect. 7.3.3, where it is called Equidistant Detection Scheme.
4. Making all p_m almost equally large: Choose C such that S_m are similar for all $m > d$. Note that

$$\sum_{m=d+1}^{n} S_m = 2M$$

is always true.

Similar to last subsection, considering random fault, we define the following.

Algorithm 1: Ranking algorithm that chooses the code with the optimal leakage properties

Input : n: the codeword bit-length, M: number of codewords, d: minimum distance of the code, α_i: the leakage bit weights of the register, where i in $[\![1, n]\!]$

Output: An (n, M, d) binary code

for *Every set S of M words* **do**
  for $x == 0$; $x < |S|$; x++ **do**
    for $y == x + 1$; $y < |S|$; y++ **do**
      Calculate the distance dis $(S[x], S[y])$;
      if dis $(S[x], S[y]) < d$ *(or* dis $(S[x], S[y]) ! = d$*, depends on equidistance condition)* **then**
        continue with a different set S;
      if dis $(S[x], S[y]) == n$ **then**
        $n_{distance}$++
    if $n_{distance} == n$ **then**
      continue with a different set S;
    Compute the estimated power consumption for codeword $S[x]$ and store the result in table A: $A[S[x]] = \Sigma_{i=1}^{n} \alpha_i S[x][i]$;
    Compute the estimated variance for bit leakages in $S[x]$ and store the result in table B: $B[S[x]] = \Sigma_{i=1}^{n}((\alpha_i S[x][i]) - \mu_{S[x]})^2$;
    Compute the bit with the highest bit leakage in $S[x]$ and store the result in table C: $C[S[x]] = max(\alpha_i S[x][i])$;
  Compute the register leakage variance for codewords in S and store the result in table D: $D[S] = \Sigma_{S[x]=1}^{|S|}(A[S[x]] - \mu_S)^2$;
  Choose the highest variance for register bit leakages for codewords in S and store the result in table E: $E[S] = max(B)$;
  Choose the value of the highest register bit leakage among the codewords in S and store the result in table F: $F[S] = max(C)$;

Get the optimal candidate using the following criteria:
1. Choose the candidates with the lowest register variances from $D[S]$;
2. From this set, choose the candidates with the lowest value of the highest leakage according to $F[S]$;
3. Finally, choose from the previous set, take the candidate with the lowest bit leakage variance according to $E[S]$;

return M codewords in case all the conditions are met, or an empty set otherwise

Table 7.1 Possible (n, M, d)-binary codes for $n = 8, 9, 10$, $M = 16$ and $n = 8$, $M = 4$

n	M	d
8	4	2, 3, 4, 5
8	16	2, 3, 4
9	16	2, 3, 4
10	16	2, 3, 4

Definition 7.10 For an (n, M, d)-binary code C such that $d \geq 2$, let p_m be its m-bit fault resistance probability for $1 \leq m \leq n$, then

$$p_{\text{rand}} := \sum_{m=1}^{n} \frac{1}{n} p_m$$

is called the *overall resistance index* for C.

Note that the bigger the p_{rand} is, the more resistant the code C is to random faults.

Lemma 7.1 *For an (n, M, d)-binary code C, if it is equidistant, then*

$$p_m = \begin{cases} 1 & m \neq d \\ 1 - \frac{M-1}{\binom{n}{d}} & m = d \end{cases}, \quad \text{and} \quad p_{\text{rand}} = 1 - \frac{M-1}{\binom{n}{d} n}.$$

7.3.3 Coding Schemes

Here we propose two different coding schemes:

1. Detection Scheme: Using binary code which has minimum distance at least 2.
2. Correction Scheme: Using binary code which has minimum distance at least 3 with error correction enabled look-up table.

Furthermore, as will be seen from the rest of this chapter, equidistant codes have different behaviors than codes that are not equidistant. Hence when equidistant codes are used, we emphasize the usage by referring to the schemes as "Equidistant detection scheme" and "Equidistant correction scheme," respectively.

We will analyze the m-bit fault resistant probability (with error) as well as overall resistance index (with error) for each of them using (n, M, d) binary codes for $n = 8, 9, 10$ and $M = 4, 16$. We chose $M = 4$ because it is easy to analyze and explain, and $M = 16$ because it can encode one nibble of the data; therefore, it is usable in a practical scenario. To illustrate the usage of the schemes we refer the reader to Appendix 2 for calculations of the probabilities for some specific codes as examples.

First, we discuss the possible values of the minimum distance d. As is well known in coding theory, fixing the length of the code n and minimum distance d, M is upper bounded by certain value. This upper bound is tight for small values n and d and still open for a lot of other values [7, p. 247]. In particular, for $n = 8, 9, 10$ and different values of d we know the exact possible values of M. In return, the possible values of d are known when n, M are fixed. In Table 7.1 we list the possible minimum distances that can be achieved for $n = 8, 9, 10$ and $M = 4$ or 16. Note that the values are taken from [7, p. 247, 248] and [5].

For equidistant binary code, we have the following constraint on d.

Lemma 7.2 *Let C be an (n, M, d) equidistant binary code such that $M \geq 3$, then d is even.*

Proof Recall the *Hamming weight* of a word $\boldsymbol{x} \in \mathbb{F}_2^n$ denoted by $\text{wt}(\boldsymbol{x})$ is defined to be the number of nonzero coordinates in $\boldsymbol{x}$ [11, p. 46]. And we have the following relation (see [11, Corollary 4.3.4 and Lemma 4.3.5]):

$$wt(\boldsymbol{x}) + wt(\boldsymbol{y}) \equiv \text{dis}(\boldsymbol{x}, \boldsymbol{y}) \mod 2.$$

Take an (n, M, d) equidistant binary code C and any three distinct codewords $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in C$, we have

$$\text{dis}(\boldsymbol{x}, \boldsymbol{y}) + \text{dis}(\boldsymbol{y}, \boldsymbol{z}) + \text{dis}(\boldsymbol{z}, \boldsymbol{x}) \equiv 2wt(\boldsymbol{x}) + 2wt(\boldsymbol{y}) + 2wt(\boldsymbol{z}) \equiv 0 \mod 2.$$

Hence, d cannot be odd.

Furthermore we have $M \leq n + 1$[8]. Thus we will only consider $(8, 4, 2)$ and $(8, 4, 4)$ equidistant binary codes. The fact that such codes exist can be derived from [8].

7.4 Automated Generation and Evaluation of Codes

In this section, we will utilize the findings stated in Sect. 7.3 to design the algorithm that automatically generates codes with the optimal side-channel and fault detection properties for a given code length. First, we present the algorithm that finds the codes based on searching criteria in Sect. 7.4.1. Then we show properties of the codes that were produced by the algorithm in Sect. 7.4.2. To verify our theoretical results, we simulate fault injections into these codes, by using an automated fault simulator which will be explained in Sect. 7.4.3. Finally, we present and discuss the simulation results in Sect. 7.4.4.

7.4.1 Code Generation and Ranking Algorithm

When it comes to device leakage, it normally depends on the processed intermediate values. In [13], they proposed the first encoding scheme that assumed a stochastic leakage model over the Hamming weight model. In such model, leakage is formulated as follows:

$$T(x) = L(x) + \epsilon, \tag{7.4}$$

where L is the leakage function mapping the deterministic intermediate value (x) processed in the register to its side-channel leakage, and ϵ is the (assumed) mean-

free Gaussian noise. For 8-bit microcontroller case, we can specify this function as $L(x) = \alpha_0 + \alpha_1 x_1 + \cdots \alpha_8 x_8$, where x_i is the ith bit of the intermediate value, and α_i is the ith bit weight leakage for specific register [16]. The α_i values can be obtained by using the following equation:

$$\alpha = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{T}, \tag{7.5}$$

where $\mathbf{A}$ is a matrix of intermediate values and $\mathbf{T}$ is a set of traces. After the device profiling which obtains the α values, we can use our ranking algorithm to select the optimal code with given inputs (Algorithm 1). Note that one can still use the Hamming weight model—for that case, α has to be defined as unity. In the following, we will explain how the algorithm works.

First, the inputs have to be specified—length (n), number of the codewords (M), minimum distance (d), and leakages of the register bits (α_i). Depending on these values, the algorithm analyzes every possible set of M codewords that can be a potential code candidate. Lines 2–3 iterate over every combination of two codewords. Lines 4–6 test if the minimum distance condition is fulfilled. Then, lines 7–10 check, whether for each codeword there exists another codeword which is at distance n from it—if yes, we skip this set. This condition is necessary in order to get a code resistant against n-bit flip (we will detail such case in Sect. 7.5). Lines 11–13 compute the 3 values that are used in order to calculate the values for the whole code in the later phase: estimated power consumption for the codeword, stored in table A, estimated variance for bit leakages in the codeword, stored in table B, and the highest bit leakage value, stored in table C. Next, the codeword value is stored in the index table I.

Lines 14–16 use the values from tables A, B, C to compute the register leakage variance ($\mu_{S[x]}$ denotes the mean leakage for a word $S[x]$), highest variance for bit leakages within registers, and value of the highest bit leakage within registers for the set S. These values are stored in tables D, E, F, respectively, and are used in the final evaluation.

The final evaluation is the last phase of the algorithm. First, it takes a subset of D with the best register leakage variance (μ_S denotes the mean leakage for codewords in S). It narrows this subset to candidate codes with the lowest value of the highest bit leakage according to set E. From these, it chooses the code with the lowest bit leakage variance using table F.

7.4.2 Properties of Generated Codes

Codes with the best side-channel and fault resistance properties according to Algorithm 1 with 4 codewords and length 8 can be found in Table 7.2. Their detailed properties are stated in Table 7.3. More codes with cardinality 16 and various distances can be found in Appendix 1.

Table 7.2 Codes used in evaluation

Code	Distance	Denoted by
0x3D, 0x9D, 0xAD, 0xBC	$= 2$	$C_{8,4,eq2}$
0x0B, 0x19, 0x35, 0xA6	$>= 2$	$C_{8,4,min2}$
0x19, 0x35, 0x8A, 0xA6	$>= 3$	$C_{8,4,min3}$
0x55, 0x93, 0xA5, 0xC6	$= 4$	$C_{8,4,eq4}$
0x19, 0x27, 0x8A, 0xB4	$>= 4$	$C_{8,4,min4}$
0x19, 0x6A, 0x87, 0xF4	$>= 5$	$C_{8,4,min5}$

For calculating the register variance, we follow the similar methodology as used in [13], together with their generated α values, but we improved their ranking algorithm by calculating the bit variances inside registers and by selecting the code which has the lowest leakage value for the highest leaking codeword. First part of Table 7.3 shows these three values, with the order of preference according to our ranking algorithm. Second part of the table shows bit fault resistance probabilities, denoted by p_m for m-bit flips in the codeword, as well as overall resistance index, denoted by p_{rand} for the code. The last part of the table shows the fault resistance probabilities with error correction, denoted by $p_{m,(e)}$, as well as overall resistance index with error correction, which is denoted by $p_{\text{rand},(e)}$. We do not consider codes with distance 1 because such codes do not provide protection against 1-bit flips and therefore the fault protection would be very low. However, such codes can still be used for minimizing the side-channel leakage.

In general, if we aim for higher distance values, we get better detection and correction capabilities, but the side-channel leakage is higher as well. That is because if the distance is higher, it is more likely that the variance of leakage among the codewords is bigger. Also, we can see that equidistant codes have a constant detection probability of 1 except the case when number of bit flips is the same as the code distance. Moreover, if we sum up the probabilities of all the bit flip faults for non-equidistant codes, the overall detection probability is lower. However, the side-channel leakage of equidistant codes is more than 10 times higher compared to non-equidistant codes.

7.4.3 Automated Fault Simulation

The fault simulator we used was customized for the purpose of evaluating a microcontroller assembly table look-up implementation of the encoding schemes presented in this chapter. More details on this simulator are provided in [1]. This simulator helps us to extend the theoretical results to real-world results, where one has to use capabilities of microprocessors for computing the results.

Table 7.3 Side-channel and fault properties of the codes

α = [0.613331, 0.644584, 0.602531, 0.190986, 0.586268,0.890951, 1.838814, 1.257943, 0.899922, 0.614699]						
Code	$C_{8,4,eq2}$	$C_{8,4,min2}$	$C_{8,4,min3}$	$C_{8,4,eq4}$	$C_{8,4,min4}$	$C_{8,4,min5}$
Codeword variance	0.0158	1.150×10^{-5}	9.800×10^{-6}	0.0021	6.440×10^{-5}	6.743×10^{-3}
Highest leakage	4.9003	3.3445	3.3413	2.9514	3.3377	3.3445
Bit variance	0.2492	0.2748	0.2776	0.1535	0.2776	0.3702
p_1	1	1	1	1	1	1
p_2	0.8929	0.9821	1	1	1	1
p_3	1	0.9911	0.9821	1	1	1
p_4	1	0.9929	0.9857	0.9571	0.9857	1
p_5	1	0.9821	1	1	0.9643	0.9643
p_6	1	1	1	1	1	0.9643
p_7	1	0.9375	0.8750	1	1	1
p_8	1	1	1	1	1	1
p_{rand}	0.9866	0.9857	0.9804	0.9946	0.9938	0.9911
$p_{1,(e)}$	–	–	1	1	1	1
$p_{2,(e)}$	–	–	0.8929	1	1	1
$p_{3,(e)}$	–	–	0.9107	0.7857	0.9286	1
$p_{4,(e)}$	–	–	0.9143	0.9571	0.8429	0.8571
$p_{5,(e)}$	–	–	0.9286	0.7857	0.8929	0.8571
$p_{6,(e)}$	–	–	0.7500	1	0.7857	0.75
$p_{7,(e)}$	–	–	0.8750	1	1	0.75
$p_{8,(e)}$	–	–	0	1	1	1
$p_{\text{rand},(e)}$	–	–	0.7839	0.9411	0.9313	0.9018

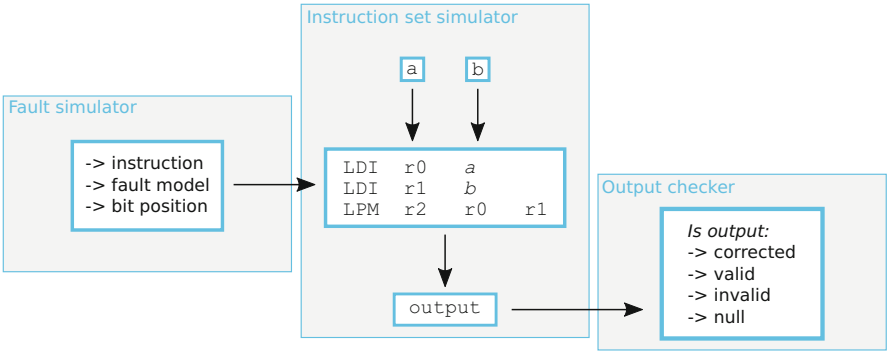

Fig. 7.1 Fault simulator operation overview

A high-level overview is given in Fig. 7.1. There are three instructions in total—the first two `LDI` load the two operands into registers `r0` and `r1`. Both of the operands are already encoded according to one of the coding schemes. The `LPM` instruction loads the data from the look-up table stored in the memory by using the values in `r0` and `r1`, and the result is stored to register `r2`. This part works as a standard instruction set simulator. During each execution, a fault is injected into the code. For each type of fault, we test all the possible combinations of codewords, and we disturbed all the instructions in our code. We have tested the following fault models:

- **Bit faults:** In this fault model, one to n bits in the destination register change its value to a complementary one.
- **Random byte faults:** The *random byte fault* model changes random number of bits in the destination register.
- **Instruction skip:** Instruction skip is a very powerful model that is capable of removing some countermeasures completely. We have tested a single instruction skip on all three instructions in the code.
- **Stuck-at fault:** In this fault model, the value of the destination register changes to a certain value, usually to all zeroes. Therefore, we have tested this value in our simulator.

After the output is produced under a faulty condition, it is analyzed by the output checker, which decides on its classification. Outputs can be of four types (*Corrected, Valid, Invalid, and Null*), and these types are described in detail in Sect. 7.2.2.

7.4.4 Simulated Results

Figure 7.2 shows plots for $C_{8,4,min4}$ and $C_{8,4,eq4}$, with and without the error correction. Instruction skip faults and stuck-at faults show zero success when

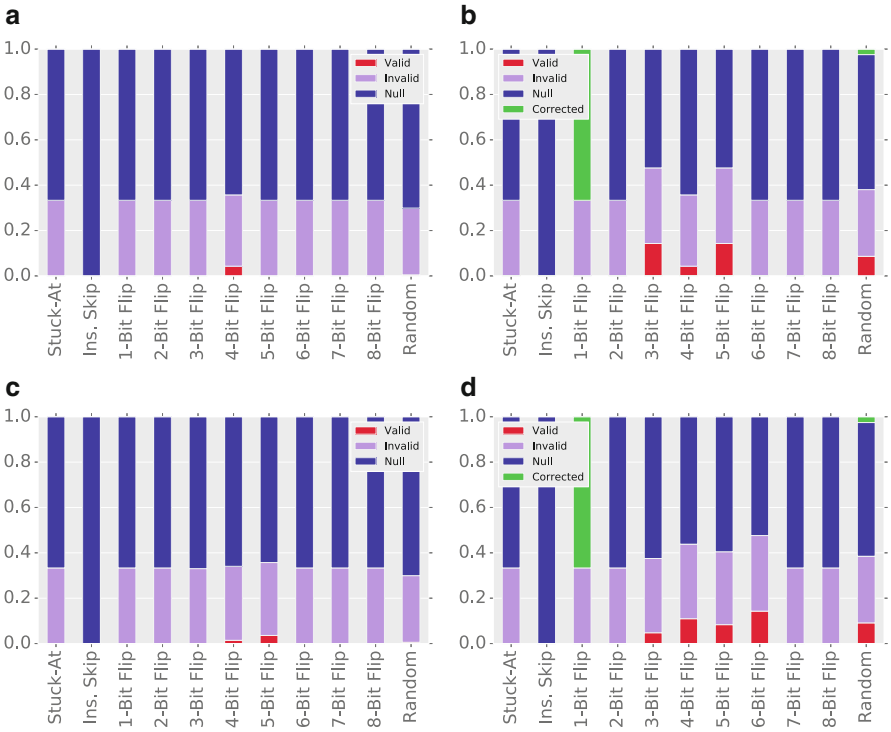

Fig. 7.2 Simulation results for $C_{8,4,eq4}$ with equidistant detection scheme in (**a**) and with equidistant correction scheme in (**b**); $C_{8,4,min4}$ with detection scheme in (**c**) and with correction scheme in (**d**)

attacking any of the generated codes. When it comes to bit flips, we can see that for better fault tolerance, one should not use the error-correction capabilities, since the properties of such codes allow changing the faulty codeword into another codeword, depending on the number of bit flips and minimum distance of the code. When deciding whether to choose an equidistant code or not, situation is the same as in Table 7.3—equidistant codes have slightly better fault detection properties, but worse side-channel leakage protection. Therefore, it depends on the implementer to choose a compromise between those two.

7.5 Discussion

First, we would like to explain the difference between the calculated results in Table 7.3 and the simulated results in Fig. 7.2 in equidistant code $C_{8,4,min4}$. Table 7.3 shows theoretical results assuming that error happens before using the look-up table.

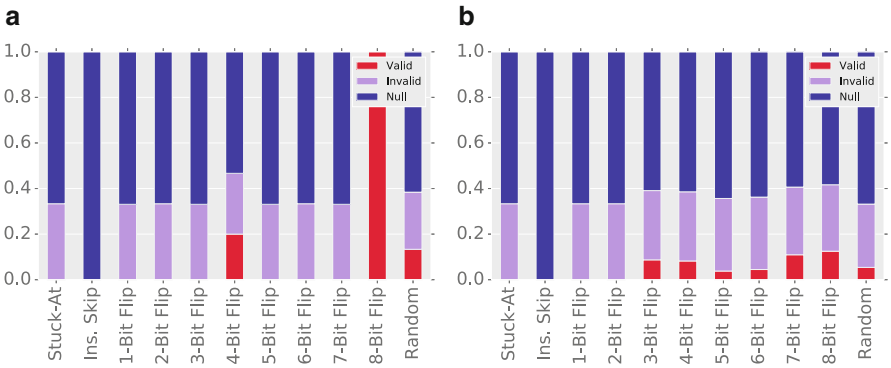

Fig. 7.3 Simulation results for the codes: (**a**) $C_{8,16,min4}$ and (**b**) $C_{8,16,min3}$

However, in a real-world setting, fault can be injected at any point of the execution, including the table look-up, or even after obtaining the result from the table. That is also why there are *Invalid* faults, despite the table always outputs *Null* in case of being addressed by a word that does not correspond to any codeword. Because there are three instructions in the assembly code, faulting the destination register of the last one after returning the value from the table results into 1/3 of *Invalid* faults in all the cases except instruction skips.

To explain the condition on lines 7–8 of Algorithm 1, we can take the code with $n = 8$, $M = 16$, and $d = 4$ as an example. The simulation result for this code is stated in Fig. 7.3a. Full results for this code are then in Table 7.5 in the appendix. There are no codes with these parameters that could satisfy the abovementioned condition—all 480 codes that can be constructed have the property that if any codeword is faulted by n bit flip, it will change to other codeword. Therefore, such codes are not suitable for protecting implementations against fault attacks. For this reason, it is more suitable to use the $C_{8,16,min3}$ code, stated in Fig. 7.3b, that does not suffer from such property.

To summarize the evaluation results, we point out the following findings:

- Correction scheme is not suitable for fault tolerant implementations—while it can be helpful in non-adversary environments, where it can be statistically verified, how many bits are usually faulted, and therefore, a proper error-correction function can be specified, in adversary-based settings, one cannot estimate the attacker capabilities. In case of correcting 1-bit error, for example, attacker who can flip multiple bits will have a higher probability of producing *Valid* faults, compared to using detection scheme with the same code.

- We can find an optimal code either from the fault tolerance perspective or from side-channel tolerance perspective—if we consider both, a compromise has to be made, depending on which attack is more likely to happen or how powerful an attacker can be in either setting. If we sacrifice the fault tolerance, we will normally get a code with distance 2 (e.g., side-channel resistant codes in [13] all have distance 2 and they are not equidistant codes); therefore, such codes will be vulnerable to 2-bit faults. On the other hand, by relaxing the power consumption variance condition, we will be able to choose codes with bigger distance, being able to resist higher number of bit faults.
- Both types of resistances can be improved if we sacrifice the memory and choose codes with greater lengths.
- Equidistant detection schemes is a good option in case the implementation can be protected against certain number of bit flips—because all the *Valid* faults are achieved only if the attacker flips the same number of bits as is the distance. However, this condition does not hold in case of equidistant correction schemes.

7.6 Chapter Summary

In this chapter, we provided a necessary background for constructing side-channel and fault attack resistant software encoding schemes. Current encoding schemes only cover side-channel resistance, and either do not discuss fault resistance or only state it as a side product of the construction, such as [19]. Our work defines theoretical bounds for fault detection and correction and provides an automated way to construct efficient codes that are capable of protecting the underlying computation against both physical attack classes.

To support our result with a practical case study, we designed an automated simulator to evaluate the table look-up operation under faulty conditions, by using a microcontroller assembly code. As expected, the codes constructed using the stated algorithm provide robust fault resistance, while keeping the side-channel leakage at the minimum.

Appendix 1: Generated Codes

In this section, we state the remaining codes generated by Algorithm 1, for $M = 16$ and $n = 8, 9, 10$ (Tables 7.4 and 7.5).

Table 7.4 Codes generated by Algorithm 1

Code	Length	Distance	Denoted by
0x0E, 0x4D, 0xF1, 0xEC, 0x2D, 0x26, 0x86, 0x8D, 0xA5, 0x46, 0xD9, 0x13, 0xD2, 0x79, 0x72, 0x5A	8	>= 2	$C_{8,16,min2}$
0x4D, 0x8B, 0x96, 0x43, 0xE9, 0xE2, 0xBA, 0xD5, 0x33, 0x2E, 0x3D, 0xFC, 0xA5, 0x5A, 0x76, 0xCE	8	>= 3	$C_{8,16,min3}$
0xBA, 0xD9, 0xEF, 0x73, 0x1F, 0xD6, 0x83, 0xB5, 0x26, 0x4A, 0x7C, 0x45, 0x29, 0x8C, 0xE0, 0x10	8	>= 4	$C_{8,16,min4}$
0x145, 0x15A, 0x1CA, 0x95, 0xCC, 0xDA, 0xC5, 0x18C, 0x0E, 0xD3, 0x19A, 0x185, 0x07, 0x193, 0x9C, 0x153	9	>= 2	$C_{9,16,min2}$
0x07, 0xF3, 0x146, 0xB5, 0xEC, 0x2E, 0x1BA, 0x165, 0x13C, 0x1D, 0x1D9, 0x5B, 0x1D4, 0x18B, 0x96, 0x185	9	>= 3	$C_{9,16,min3}$
0x3B, 0x75, 0x9D, 0x14B, 0x1D4, 0x1A5, 0xEC, 0x13C, 0x1F9, 0x193, 0x07, 0xDA, 0x166, 0xB6, 0x1AA, 0xE3	9	>= 4	$C_{9,16,min4}$
0x5D, 0xDC, 0x34B, 0x25C, 0x1CB, 0x359, 0xCE, 0x3CA, 0x3E6, 0x1F5, 0x1E7, 0x3F4, 0x375, 0x24E, 0x4F, 0x1D9	10	>= 2	$C_{10,16,min2}$
0xA7, 0x235, 0x3C8, 0x22A, 0x14C, 0x39, 0x298, 0x3C5, 0x3B1, 0x8B, 0x1B4, 0x1C, 0x326, 0x156, 0x169, 0x353	10	>= 3	$C_{10,16,min3}$
0x2D, 0x16A, 0x18C, 0x97, 0x136, 0x21A, 0x347, 0x3D4, 0x3A5, 0x159, 0x275, 0x2E6, 0xCB, 0xF8, 0x1F3, 0x24C	10	>= 4	$C_{10,16,min4}$

Appendix 2: Fault Resistance Probabilities

In this section, we show the detailed theoretical calculations of fault resistance probabilities and the overall resistance index (with error) for some specific examples.

Equidistant Detection Scheme
Using Lemma 7.1, we list the values of p_ms and p_{rand} in Table 7.6 for (8, 4, 2) and (8, 4, 4) equidistant binary codes.

Table 7.5 Side-channel and fault properties of the codes from Table 7.4

α	0.613331, 0.644584, 0.602531, 0.190986, 0.586268, 0.890951, 1.838814, 1.257943, 0.899922, 0.614699								
Code	$C_{8,16,min2}$	$C_{8,16,min3}$	$C_{8,16,min4}$	$C_{9,16,min2}$	$C_{9,16,min3}$	$C_{9,16,min4}$	$C_{10,16,min2}$	$C_{10,16,min3}$	$C_{10,16,min4}$
Codeword variance	0.0013	0.1190	1.8231	2.935×10^{-4}	0.0091	0.1043	3.920×10^{-5}	0.0017	0.0134
Highest leakage	3.2607	3.2607	0.1910	3.9510	3.9967	3.5960	4.7812	3.8545	3.9011
Bit variance	0.4657	0.3949	0.3367	0.2552	0.3571	0.3366	0.3170	0.3875	0.3929
p_1	1	1	1	1	1	1	1	1	1
p_2	0.9241	1	1	0.9167	1	1	0.9417	1	1
p_3	0.9286	0.9129	1	0.9732	0.9628	1	0.9854	0.9844	1
p_4	0.9714	0.9179	0.8	0.9752	0.9692	0.9147	0.9857	0.9857	0.9780
p_5	0.9308	0.9621	1	0.9683	0.9692	1	0.9866	0.9861	0.9767
p_6	0.9241	0.9554	1	0.9881	0.9643	0.9613	0.9911	0.9839	0.9911
p_7	1	0.8906	1	0.9549	0.9722	1	0.9854	0.9721	0.9865
p_8	0.1250	0.8750	0	1	0.9861	0.8889	0.9861	0.9861	0.9861
p_9	–	–	–	1	1	1	1	1	0.9625
p_{10}	–	–	–	–	–	–	1	1	1
p_{rand}	0.8505	0.9392	0.85	0.9771	0.9804	0.9739	0.9862	0.9904	0.9881
$p_{1,(e)}$	–	1	1	–	1	1	–	1	1
$p_{2,(e)}$	–	0.4778	1	–	0.7400	1	–	0.8750	1
$p_{3,(e)}$	–	0.5022	0	–	0.7782	0.4881	–	0.8844	0.8458
$p_{4,(e)}$	–	0.4179	0.8	–	0.6667	0.9147	–	0.8399	0.8381
$p_{5,(e)}$	–	0.4174	0	–	0.6726	0.4187	–	0.8343	0.8219
$p_{6,(e)}$	–	0.5089	1	–	0.6964	0.9613	–	0.8131	0.7970
$p_{7,(e)}$	–	0.4531	0	–	0.6944	0.5069	–	0.8240	0.8823
$p_{8,(e)}$	–	0	0	–	0.7639	0.8889	–	0.8111	0.8028
$p_{9,(e)}$	–	–	–	–	0.8750	0	–	0.8750	0.8375
$p_{10,(e)}$	–	–	–	–	–	–	–	1	0.6250
$p_{\text{rand},(e)}$	–	0.4722	0.4750	–	0.7652	0.6865	–	0.8757	0.8450

Table 7.6 Theoretical values of p_m for (n, M, d)-equidistant binary code

(n, M, d)	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_{rand}
(8, 4, 2)	1	0.8929	1	1	1	1	1	1	0.9866
(8, 4, 4)	1	1	1	0.9571	1	1	1	1	0.9946

Table 7.7 Distance between each pair of codewords in the (8, 4, 4)-binary code $C_{8,4,min4}$

dis $(\cdot, \cdot)$	00011001	00100111	10001010	10110100
00011001	0	5	4	5
00100111	5	0	5	4
10001010	4	5	0	5
10110100	5	4	5	0

Detection Scheme

Since we require that dis $(C) \geq 2$ for Detection Scheme, for 1-bit fault, we expect the results to be *Null*, which means $p_1 = 1$. Now we give a theoretical calculation for the (8, 4, 4)-binary code $C_{8,4,min4} = \{00011001, 00100111, 10001010, 10110100\}$. We first list the distance between every pair of codewords in Table 7.7.

By Eq. (7.3), we can then calculate the m-bit fault resistance probabilities and the overall resistance index for C:

$$p_2 = p_3 = 1 - \frac{1}{4}(0 + 0 + 0 + 0) = 1,$$

$$p_4 = 1 - \frac{1}{4\binom{8}{4}}(2 + 0 + 1 + 1) = \frac{69}{70} \approx 0.9857,$$

$$p_5 = 1 - \frac{1}{4\binom{8}{5}}(2 + 2 + 2 + 2) = \frac{27}{28} \approx 0.9643,$$

$$p_6 = p_7 = p_8 = 1 - \frac{1}{4}(0 + 0 + 0 + 0) = 1, \quad p_{\text{rand}} = \sum_{m=1}^{8} \frac{1}{8} p_m = 0.9938.$$

Correction Scheme

m-bit fault resistance probabilities with error correction for the same (8, 4, 4)-binary code $C_{8,4,min4} = \{00011001, 00100111, 10001010, 10110100\}$. As dis $(C) = 4$, by Remark 7.1 it is an 1-error-correcting code. By Eq. (7.2), $p_{m,(e)} = 1$ for $m = 1$. To calculate $p_{m,(e)}$ for $m \geq 2$, we first list the table of cardinalities of $F_{\boldsymbol{c},m}$ for $\boldsymbol{c} \in C$ and $m = 2, 3, \ldots, 8$ in Table 7.8.

By Eq. (7.2), we can then calculate the m-bit fault resistance probabilities with error correction as well as the overall resistance index with error correction for C.

Table 7.8 Cardinality of $F_{c,m}$ for $m = 2, 3, \ldots, 8$ and $\boldsymbol{c} \in C_{8,4,min4}$

	$\lvert F_{\boldsymbol{c},2}\rvert$	$\lvert F_{\boldsymbol{c},3}\rvert$	$\lvert F_{\boldsymbol{c},4}\rvert$	$\lvert F_{\boldsymbol{c},5}\rvert$	$\lvert F_{\boldsymbol{c},6}\rvert$	$\lvert F_{\boldsymbol{c},7}\rvert$	$\lvert F_{\boldsymbol{c},8}\rvert$
00011001	0	4	11	6	6	0	0
00100111	0	4	11	6	6	0	0
10001010	0	4	11	6	6	0	0
10110100	0	4	11	6	6	0	0

$$p_{2,(e)} = 1 - \frac{1}{4\binom{8}{2}}(0+0+0+0) = 1,$$

$$p_{3,(e)} = 1 - \frac{1}{4\binom{8}{3}}(4+4+4+4) = \frac{13}{14} \approx 0.9286,$$

$$p_{4,(e)} = 1 - \frac{1}{4\binom{8}{4}}(11+11+11+11) = \frac{59}{70} \approx 0.8429,$$

$$p_{5,(e)} = 1 - \frac{1}{4\binom{8}{5}}(6+6+6+6) = \frac{25}{28} \approx 0.8929,$$

$$p_{6,(e)} = 1 - \frac{1}{4\binom{8}{6}}(6+6+6+6) = \frac{11}{14} \approx 0.7857,$$

$$p_{7,(e)} = p_{8,(e)} = 1 - \frac{1}{4}(0+0+0+0) = 1,$$

$$p_{\text{rand},(e)} = \sum_{m-1}^{8} \frac{1}{8} p_{m,(e)} = 0.9313.$$

References

1. J. Breier, On analyzing program behavior under fault injection attacks, in *2016 Eleventh International Conference on Availability, Reliability and Security (ARES)* (IEEE, Piscataway, 2016), pp. 1–5
2. J. Breier, X. Hou, Feeding two cats with one bowl: on designing a fault and side-channel resistant software encoding scheme, in *Cryptographers' Track at the RSA Conference* (Springer, Berlin, 2017), pp. 77–94
3. J. Breier, D. Jap, C.-N. Chen, Laser profiling for the back-side fault attacks: with a practical laser skip instruction attack on AES, in *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security (CPSS '15)* (ACM, New York, 2015), pp. 99–103
4. J. Breier, D. Jap, S. Bhasin, The other side of the coin: analyzing software encoding schemes against fault injection attacks, in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (IEEE, Piscataway, 2016), pp. 209–216
5. A.E. Brouwer, J.B. Shearer, N.J.A. Sloane, W.D. Smith, A new table of constant weight codes. IEEE Trans. Inf. Theory **36**(6), 1334–1380 (1990)

6. C. Chen, T. Eisenbarth, A. Shahverdi, X. Ye, Balanced encoding to mitigate power analysis: a case study, in *International Conference on Smart Card Research and Advanced Applications*. Lecture Notes in Computer Science (Springer, Berlin, 2014), pp. 49–63
7. J.H. Conway, N.J.A. Sloane, *Sphere Packings, Lattices and Groups*, vol. 290 (Springer, Berlin, 2013)
8. F.-W. Fu, T. Kløve, Y. Luo, V.K. Wei, On equidistant constant weight codes. Discret. Appl. Math. **128**(1), 157–164 (2003)
9. L. Goubin, J. Patarin, DES and differential power analysis. The "duplication" method, in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Lecture Notes in Computer Science (Springer, Berlin, 1999), pp. 158–172
10. P. Hoogvorst, J.-L. Danger, G. Duc, Software implementation of dual-rail representation, in *Second International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*, Darmstadt (2011)
11. S. Ling, C. Xing, *Coding Theory: A First Course* (Cambridge University Press, Cambridge, 2004)
12. F.J. MacWilliams, N.J.A. Sloane, *The Theory of Error Correcting Codes* (Elsevier, Amsterdam, 1977)
13. H. Maghrebi, V. Servant, J. Bringer, There is wisdom in harnessing the strengths of your enemy: customized encoding to thwart side-channel attacks – extended version–. Cryptology ePrint Archive, Report 2016/183, 2016. http://eprint.iacr.org/
14. P. Rauzy, S. Guilley, Z. Najm, Formally proved security of assembly code against leakage. IACR Cryptology ePrint Arch. **2013**, 554 (2013)
15. F. Regazzoni, L. Breveglieri, P. Ienne, I. Koren, Interaction between fault attack countermeasures and the resistance against power analysis attacks, in *Fault Analysis in Cryptography* (Springer, Berlin, 2012), pp. 257–272
16. W. Schindler, K. Lemke, C. Paar, A stochastic model for differential side-channel cryptanalysis, in *International Workshop on Cryptographic Hardware and Embedded Systems* (Springer, Berlin, 2005), pp. 30–46
17. T. Schneider, A. Moradi, T. Güneysu, ParTI – towards combined hardware countermeasures against side-channel and fault-injection attacks, in *Annual Cryptology Conference* (Springer, Berlin, 2016), pp. 302–332
18. N. Selmane, S. Bhasin, S. Guilley, T. Graba, J.-L. Danger, WDDL is protected against setup time violation attacks, in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)* (IEEE, Piscataway, 2009), pp. 73–83
19. V. Servant, N. Debande, H. Maghrebi, J. Bringer, Study of a novel software constant weight implementation, in *International Conference on Smart Card Research and Advanced Applications* (Springer, Berlin, 2014), pp. 35–48
20. K. Tiri, I. Verbauwhede, A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation, in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 1 (IEEE, Piscataway, 2004), pp. 246–251
21. E. Trichina, R. Korkikyan, Multi fault laser attacks on protected CRT-RSA, in *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)* (IEEE, Piscataway, 2010), pp. 75–86