Feeding Two Cats with One Bowl: On Designing a Fault and Side-Channel Resistant Software Encoding Scheme

Jakub Breier[1] and Xiaolu Hou[2]

[1]Physical Analysis and Cryptographic Engineering
Temasek Laboratories at Nanyang Technological University, Singapore
[2]Divison of Mathematical Sciences
Nanyang Technological University, Singapore
jbreier@ntu.edu.sg,ho0001lu@e.ntu.edu.sg

Abstract. When it comes to side-channel countermeasures, software encoding schemes are becoming more popular and provide a good level of security for general-purpose microcontrollers. However, these schemes are not designed to be fault resistant, and this property is discussed very rarely. Therefore, implementers have to pile up two different countermeasures in order to protect the algorithm against these two popular classes of attacks.
In our paper, we discuss the fault resistance properties of encoding schemes in general. We define theoretical bounds that clearly show the possibilities and limitations of encoding-based countermeasures, together with trade-offs between side-channel and fault resistance. Moreover, we simulate several codes with respect to most popular fault models, using a general-purpose microcontroller assembly implementation. Our algorithm shows how to implement fault resistance to an encoding scheme that currently has the best side-channel resistant capabilities. As a result, we are able to design a code by using automated methods, that can provide the optimal trade-off between side-channel and fault resistance.

Keywords: software encoding schemes, side-channel attacks, fault attacks, countermeasures

1 Introduction

When it comes to small, constrained devices, such as the ones designed for Internet of Things applications, they are usually easy to access and do not contain comprehensive security measures to protect them. Therefore, even though a strong cryptography is used to protect the communication, hardware attacks pose a serious threat. Side-channel and fault attacks are among the most popular means to breach the device security. When designing a cryptographic implementation, it is necessary to consider countermeasures against these attacks.

There are two main countermeasure classes to protect implementations against side channel attacks. *Masking* [8] is a software-level countermeasure which tries to "mask" the relationship between the intermediate values and power leakage. *Hiding* [18] 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 [16].

In 2011, DPL was extended to software by Hoogvorst et al. [9], by using balanced encoding schemes. Since then, there were several other proposals [13, 5, 17, 12], 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 [2]. Therefore, to prevent both attack techniques, it is necessary to design the coding scheme from the beginning with this goal in mind.

In this paper, we introduce a theoretical background necessary for designing software hiding countermeasures that are resistant to both side-channel and fault attacks. We provide an algorithm for constructing such codes and ranking them according to required properties. We select optimal codes for various code distances and number of codewords, and evaluate them – by using detection and correction probabilities and by simulating them in a faulty environment. 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 results show 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 paper is organized as follows. Section 2 provides an overview of the related work in this field, together with necessary background on coding theory. Section 3 defines the properties of codes with respect to fault attacks. Section 4 details our algorithm, and provides estimated and simulated results on chosen codes. These results are further discussed in Section 5. Finally, Section 6 concludes this paper and provides a motivation for further work.

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. Subsection 2.1 overviews the related work in the field. Subsection 2.2 provides basic definitions that are used later in this paper.

2.1 Related Work

After the paper by Hoogvorst et al. [9] presented a method to extend the DPL to software implementations, several works were published in the area of software hiding schemes.

Rauzy et al. [13] developed a scheme that encodes the data by using bit-slicing, where only one bit of information is processed at a time. They claim 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. [5] 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. [17] 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. [12] 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. 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 have been designed for fault resistance. Schemes proposed in [13, 5] have been analyzed with respect to fault attacks by Breier et al. [2], concluding that without additional modifications to assembly code, the probability of a successful fault attack is non-negligible. Therefore, to improve the current state-of-the-art, we focus on designing fault tolerant and side-channel resistant coding schemes.

When it comes to combined countermeasures, in [15], 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 [19]. Also, it would be possible to inject faults that are impossible with hardware implementations, such as instruction skips [3].

Our contributions in this work are:

- We define theoretical bounds for encoding schemes with respect to fault attacks that are necessary to take into account when designing a fault resistant scheme.
- We show how to 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 improve the ranking algorithm proposed in [12] (current state-of-the-art) 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. Furthermore, 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 our algorithm – we calculate leakages, fault detection and correction probabilities, and we simulate the assembly code implementing the codes on a general-purpose microcontroller.

2.2 Coding Theory Background

A *binary code*, denoted by C, is a subset of the $n-$dimensional vector space over $\mathbb{F}_2$-$\mathbb{F}_2^n$. And 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* [10, p.6]. Take two codewords $\boldsymbol{c}, \boldsymbol{c'} \in C$, the *Hamming distance* between $\boldsymbol{c}$ and $\boldsymbol{c'}$, denoted by $\mathrm{dis}(\boldsymbol{c}, \boldsymbol{c'})$, is defined to be the number of places at which $\boldsymbol{c}$ and $\boldsymbol{c'}$ differ [10, p.9]. More precisely, if $\boldsymbol{c} = c_1 c_2 \dots c_n$ and $\boldsymbol{c'} = c'_1 c'_2 \dots c'_n$, then

$$\mathrm{dis}(\boldsymbol{c}, \boldsymbol{c'}) = \sum_{i=1}^{n} \mathrm{dis}(c_i, c'_i),$$

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

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

Furthermore, for a binary code C, the *minimum distance* of C, denoted by $\mathrm{dis}(C)$, is [10, p.11]

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

Definition 1. *[6, p.75] For a binary code C of length n, $\mathrm{dis}(C) = d$, 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 2. *[10, 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 $\mathrm{dis}(C) \geq u+1$ [10, 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 $\boldsymbol{c_x}$ such that

$$\mathrm{dis}(\boldsymbol{x}, \boldsymbol{c_x}) = \min_{\boldsymbol{c} \in C} \mathrm{dis}(\boldsymbol{x}, \boldsymbol{c}). \tag{1}$$

When there are more codewords $\boldsymbol{c_x}$ satisfies (1), the *incomplete decoding rule* requires a retransmission [10, p.10].

Definition 3. *[10, 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 1. C is $v-$error correcting if and only if $\mathrm{dis}(C) \geq 2v + 1$ [10, p.13].

Definition 4. *[7] An (n, M, d) binary code C is called an* equidistant code *if $\forall \boldsymbol{c}, \boldsymbol{c'} \in C$, $\mathrm{dis}(\boldsymbol{c}, \boldsymbol{c'}) = \mathrm{dis}(C)$.*

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

We propose two choices of lookup 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 $\mathrm{dis}(\boldsymbol{c_x}, \boldsymbol{x}) = 1$. Note that this is equivalent to using *bounded*

distance decoding [11, p.36] and taking the bounded distance to be 1. To use this table we require that dis $(C) \geq 3$ so that there exists at most one such $\boldsymbol{c_x}$.

2. Detection Table: This is a normal lookup 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 Section 3. Afterwards we propose three coding schemes. Then detailed calculations will be carried out for some specific codes using the three coding schemes. The encoding scheme will be simulated (and implemented) and evaluated in Section 4.

Let m be a positive integer such that $1 \leq m \leq n$.

Definition 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 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.*

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 three different coidng schemes and analyze their fault resistant probabilities.

3.1 Correction Table

Definition 7. *For an* $(n, M, d)-$*binary code* C *such that* $d \geq 3$*, let*

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

Then

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

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 1, the error will be corrected and hence $p_m = 1$. 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 6, $1 - p_m$ gives the theoretical probability of a *Valid* fault and the bigger p_m 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*.

3.2 Detection Table

Now we consider Detection Table.

Definition 8. *For an (n, M, d)-binary code C such that $d \geq 2$, let*

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

Then

$$p_m := 1 - \frac{S_m}{M\binom{n}{m}}, \tag{3}$$

*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 6, $1 - p_m$ gives the theoretical probability of a *Valid* fault. Hence the bigger is p_m, the better the binary code is $m-$fault resistant.

Remark 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 $\mathrm{dis}\,(\boldsymbol{c}, \boldsymbol{c}') = n$, 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 fixed known values, from equation (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 1). This particular scheme will be discussed in Section 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 is possible by using equidistant codes. That is, take code such that $|S_{m_0}| = M$. This particular scheme will be discussed in Section 3.3, where it is called Equidistant Scheme.

4. Making all p_m almost equally small: choose C such that S_m are similar for all $m > d$. Note that

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

is always true.

Lemma 1. *For an* $(n, M, d)-$*binary code* C*, if it is equidistant, then*

$$p_m = \begin{cases} 1 & m \neq d \\ 1 - \frac{M-1}{\binom{n}{d}} & m = d \end{cases}.$$

3.3 Coding Schemes

Here we propose the three different coding schemes:

1. Equidistant Scheme: using binary code which is an equidistant code with minimum distance at least 2.

2. Detection Scheme: using binary code which has minimum distance at least 2.

3. Correction Scheme: using binary code which has minimum distance at least 3 with error correction enabled lookup table.

We will analyze the $m-$*bit fault resistant probability* 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 will calculate the exact values for some specific codes as examples.

Firstly, 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 [6, 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 know when n, M are fixed. In Table 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 [6, p.247,248] and [4].

Table 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

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

Lemma 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 $\mathrm{wt}(\boldsymbol{x})$ is defined to be the number of nonzero coordinates in $\boldsymbol{x}$ [10, p.46]. And we have the following relation (see [10, Corollary 4.3.4 and Lemma 4.3.5])

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

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

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

Hence, d cannot be odd.

Furthermore we have $M \leq n + 1$[7]. 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 [7]. Next we give theoretical calculations for some specific examples, for more values, we list in Table 6.

Equidistant Scheme. Using Lemma 1, we list the value of P_m in Table 2 for $(8, 4, 2)$ and $(8, 4, 4)$ equidistant binary codes.

Table 2: 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
$(8, 4, 2)$	1	0.8929	1	1	1	1	1	1
$(8, 4, 4)$	1	1	1	0.9571	1	1	1	1

Detection Scheme. Since we require that $\mathrm{dis}(C) \geq 2$ 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 $C_{4,min4} = \{00110111, 01001011, 01110000, 10011110\}$. We first list the distance between every pair of codewords in Table 3.

Table 3: Distance between each pair of codewords in the $(8, 4, 4)-$binary code $C_{4,min4}$.

$\mathrm{dis}(\cdot, \cdot)$	00110111	01001011	01110000	10011110
00110111	0	5	4	4
01001011	5	0	5	5
01110000	4	5	0	6
10011110	4	5	6	0

By equation (3), we can then calculate the $m-$bit fault resistance probabilities for C:

$$p_2 = p_3 = 1 - \frac{1}{4}(0 + 0 + 0 + 0) = 1, \qquad p_4 = 1 - \frac{1}{4\binom{8}{4}}(2 + 0 + 1 + 1) = \frac{69}{70} \approx 0.9857,$$

$$p_5 = 1 - \frac{1}{4\binom{8}{5}}(1 + 3 + 1 + 1) = \frac{109}{112} \approx 0.9732, \quad p_6 = 1 - \frac{1}{4\binom{8}{6}}(0 + 0 + 1 + 1) = \frac{55}{56} \approx 0.9821,$$

$$p_7 = p_8 = 1 - \frac{1}{4}(0 + 0 + 0 + 0) = 1.$$

Correction Scheme. We give a theoretical calculation for the $m-$bit fault resistance probabilities with error correction for the same $(8,4,4)-$binary code $C_{4,min4} = \{00110111, 01001011, 01110000, 10011110\}$. As $\mathrm{dis}(C) = 4$, by Remark 1 it is an $1-$error correcting code. By equation (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 4.

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

	$\lvert F_{\boldsymbol{c},2}\rvert$	$\lvert F_{\boldsymbol{c},3}\rvert$	$\lvert F_{\boldsymbol{c},4}\rvert$	$\lvert F_{\boldsymbol{c},5}\rvert$	$\lvert F_{\boldsymbol{c},6}\rvert$	$\lvert F_{\boldsymbol{c},7}\rvert$	$\lvert F_{\boldsymbol{c},8}\rvert$
00110111	0	8	7	9	3	0	0
01001011	0	0	15	3	9	0	0
01110000	0	4	6	11	4	2	0
10011110	0	4	6	11	4	2	0

By equation (2), we can then calculate the $m-$bit fault resistance probabilities with error correction for C.

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

$$p_{3,(e)} = 1 - \frac{1}{4\binom{8}{3}}(8+0+4+4) = \frac{13}{14} \approx 0.9286,$$

$$p_{4,(e)} = 1 - \frac{1}{4\binom{8}{4}}(7+15+6+6) = \frac{123}{140} \approx 0.8786,$$

$$p_{5,(e)} = 1 - \frac{1}{4\binom{8}{5}}(9+3+11+11) = \frac{95}{112} \approx 0.8482,$$

$$p_{6,(e)} = 1 - \frac{1}{4\binom{8}{6}}(3+9+4+4) = \frac{23}{28} \approx 0.8214,$$

$$p_{7,(e)} = 1 - \frac{1}{4\binom{8}{7}}(0+0+2+2) = \frac{7}{8} = 0.875,$$

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

4 Evaluation Methodology and Results

In this section, we will utilize the findings stated in Section 3 to build the codes with the optimal side-channel and fault detection properties. First, we construct an algorithm that finds the codes based on searching criteria in Section 4.1. Then we show properties of the codes that were produced by the agorithm in Section 4.2. To verify our theoretical results, we simulate fault injections into these codes, by using the fault simulator explained in Section 4.3. Finally, we present and discuss the simulation results in Section 4.4.

4.1 Code Generation and Ranking Algorithm

When it comes to device leakage, it normally depends on the processed intermediate values. In [12], they proposed the first encoding scheme that assumed a stochastic leakage model over the Hamming weight model. In such model, leakge is formulated as follows:

$$T(x) = L(x) + \epsilon, \tag{4}$$

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 $L(x) = \alpha_0 + \alpha_1 x_1 + ... \alpha_8 x_8$, where x_i is the i-th bit of the intermediate value, and α_i is the i-th bit weight leakage for specific register [14] The α_i values can be obtained by using the following equation:

$$\alpha = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{T}, \tag{5}$$

where $\mathbf{A}$ is a matrix of intermediate values and $\mathbf{T}$ is a set of traces. After the device profiling in order to obtain the α, we can use our ranking algorithm for selecting the optimal code (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) and number (M) of the codewords, minimum distance (d) between them, 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 between all the codewords from the tested set. Then, lines 7-10 check, wheter 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 Section 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, 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. 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.

4.2 Estimated Values for Chosen Codes

Codes with the best side-channel and fault resistance properties according to Algorithm 1 with 4 codewords and code length 8 can be found in Table 5. Their detailed

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 $\mathrm{dis}(S[x], S[y])$;
      if $\mathrm{dis}(S[x], S[y]) < d$ *(or* $\mathrm{dis}(S[x], S[y])! = d$*, depends on equidistance condition)* **then**
        continue with a different set S;
      if $\mathrm{dis}(S[x], S[y]) == n$ **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

properties are stated in Table 6. Rest of the codes with 16 codewords and various code distances can be found in Appendix A.

For calculating the register variance, we follow the similar methodology as used in [12], 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 6 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. The last part of the table shows the fault resistance probabilities with error correction, denoted by $p_{m,(e)}$. We do not consider codes with $n = 1$ 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.

Table 5: Codes used in evaluation.

Code	**Distance**	**Denoted by**
`0x21, 0x22, 0x24, 0x30`	= 2	$C_{4,eq2}$
`0x17, 0x41, 0x44, 0x94`	>= 2	$C_{4,min2}$
`0x35, 0x4A, 0x8D, 0x9A`	>= 3	$C_{4,min3}$
`0x3B, 0x52, 0x68, 0xA2`	= 4	$C_{4,eq4}$
`0x37, 0x4B, 0x70, 0x9E`	>= 4	$C_{4,min4}$
`0x4E, 0x61, 0x9B, 0xB4`	>= 5	$C_{4,min5}$

Table 6: 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	$C_{4,eq2}$	$C_{4,min2}$	$C_{4,min3}$	$C_{4,eq4}$	$C_{4,min4}$	$C_{4,min5}$
Codeword Variance	4.537×10^{-4}	1.460×10^{-5}	1.045×10^{-4}	9.555×10^{-3}	4.997×10^{-4}	8.032×10^{-4}
Highest Leakage	1.4772	2.4413	2.6648	2.7935	3.2823	3.2769
Bit Variance	0.0232	0.3821	0.4830	0.4560	0.3768	0.3779
p_1	1	1	1	1	1	1
p_2	0.8929	0.9821	1	1	1	1
p_3	1	0.9821	0.9911	1	1	1
p_4	1	0.9857	0.9857	0.9571	0.9857	1
p_5	1	0.9911	0.9911	1	0.9732	0.9643
p_6	1	1	0.9821	1	0.9821	0.9643
p_7	1	1	0.9375	1	1	1
p_8	1	1	1	1	1	1
$p_{2,(e)}$	-	-	0.9464	1	1	1
$p_{3,(e)}$	-	-	0.9196	0.7857	0.9286	1
$p_{4,(e)}$	-	-	0.9143	0.9571	0.8786	0.8571
$p_{5,(e)}$	-	-	0.8661	0.7857	0.8482	0.8571
$p_{6,(e)}$	-	-	0.8036	1	0.8214	0.75
$p_{7,(e)}$	-	-	0.8125	1	0.875	0.75
$p_{8,(e)}$	-	-	0.5	1	1	1

4.3 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 paper. 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.

A high-level overview is given in Figure 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 acoording 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

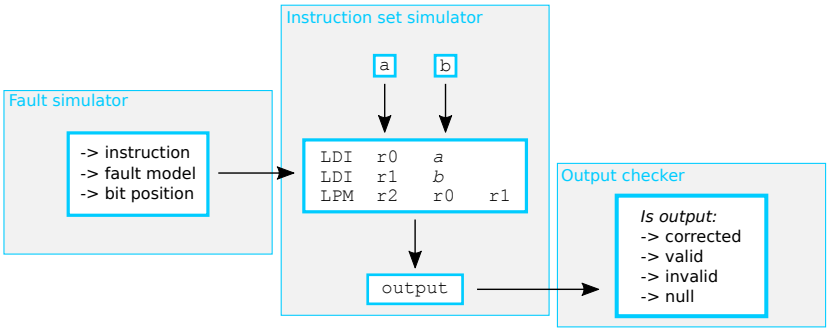


Fig. 1: Fault simulator operation overview.

all the possible combinations of codewords, and we disturbed all the instructions in our code. We have tested following fault models:

- **Bit faults:** in this fault model, one to n bits in the destination register change its value to a complementary one. This model also includes a *random byte fault* model, therefore we do not need to simulate it separately.
- **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 conditions, it is analyzed by the output checker, which decides on its classification. Outputs can be of four types (*Corrected, Valid, Invalid, Null*), and these types are described in detail in Section 2.2.

4.4 Simulated Results

Figure 2 shows plots for $C_{4,min4}$ and $C_{4,eq4}$, with and without the error correction. Instruction skip faults and stuck-at faults show zero success when 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

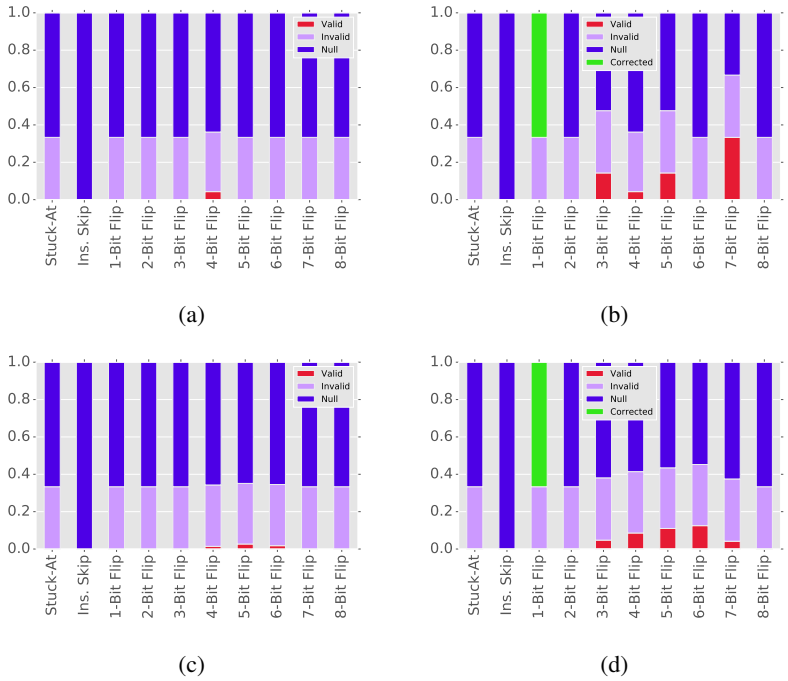


Fig. 2: Simulation results for (a) $C_{4,eq4}$, (b) $C_{4,eq4}$ with error correction, (c) $C_{4,min4}$, and (d) $C_{4,min4}$ with error correction.

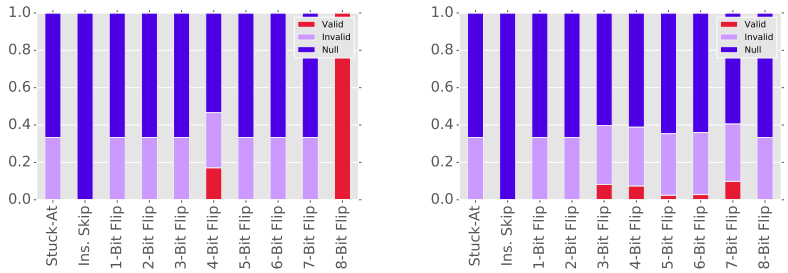
allow changing the faulty codeword into another codeword, depending on the number of bit flips and code distance. When deciding whether to choose an equidistant code or not, situation is the same as in Table 6 – equidistant codes have slightly better fault detection properties, but worse side-channel leakage protection. Therefore, it depends on implementer to choose a compromise between those two.

5 Discussion

First, we would like to explain the difference between the calculated results in Table 6 and simulated results in Figure 2 in equidistant code $C_{4,min4}$. Table shows theoretical results assuming that error happens before using the error correction table. 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 address 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 the 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 Figure 3 (a). Full results for this code are then in Table 8 in the appendix. There are

no codes with these parameters that could satisfy the abovementioned condition – all 450 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 $C_{8,min3}$ code, stated in Figure 3 (b), that does not suffer from such property.



(a) (b)

Fig. 3: Simulation results for the codes: (a) $C_{8,min4}$ and (b) $C_{8,min3}$.

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

- Error correction 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 usualy 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 errors for example, attacker who can flip multiple bits will have a higher probability of producing *Valid* faults, compared to error detecting code of the same properties.
- We can design 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 can attacker be in either setting.
- Equidistant codes are 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.

6 Conclusions

In this paper, 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 [17]. Our work defines theoretical bounds for fault detection and correction and provides a 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 simulated the table look-up under faulty conditions, by using a microcontroller assembly code. As expected, the codes constructed by using our algorithm provide noticeably better fault resistance properties compared to state-of-the-art, while keeping the side-channel leakage at the minimum.

References

1. Supressed for to double-blind review
2. Breier, J., Jap, D., Bhasin, S.: 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). pp. 209–216. IEEE (2016)
3. Breier, J., Jap, D., Chen, C.N.: 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. pp. 99–103. CPSS '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2732198.2732206
4. Brouwer, A.E., Shearer, L.B., Sloane, N., et al.: A new table of constant weight codes. In: IEEE Trans Inform Theory. Citeseer (1990)
5. Chen, C., Eisenbarth, T., Shahverdi, A., Ye, X.: Balanced Encoding to Mitigate Power Analysis: A Case Study. In: CARDIS. Lecture Notes in Computer Science, Springer (November 2014), paris, France
6. Conway, J.H., Sloane, N.J.A.: Sphere packings, lattices and groups, vol. 290. Springer Science & Business Media (2013)
7. Fu, F.W., Kløve, T., Luo, Y., Wei, V.K.: On equidistant constant weight codes. Discrete applied mathematics 128(1), 157–164 (2003)
8. Goubin, L., Patarin, J.: DES and Differential Power Analysis. The "Duplication" Method. In: CHES. pp. 158–172. LNCS, Springer (1999), Worcester, MA, USA
9. Hoogvorst, P., Danger, J.L., Duc, G.: Software Implementation of Dual-Rail Representation. In: COSADE (2011), Darmstadt, Germany
10. Ling, S., Xing, C.: Coding theory: a first course. Cambridge University Press (2004)
11. MacWilliams, F.J., Sloane, N.J.A.: The theory of error correcting codes. Elsevier (1977)
12. Maghrebi, H., Servant, V., Bringer, J.: 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/
13. Rauzy, P., Guilley, S., Najm, Z.: Formally Proved Security of Assembly Code Against Leakage. IACR Cryptology ePrint Archive 2013, 554 (2013)
14. Schindler, W., Lemke, K., Paar, C.: A stochastic model for differential side channel cryptanalysis. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 30–46. Springer Berlin Heidelberg (2005)
15. Schneider, T., Moradi, A., Güneysu, T.: Parti – towards combined hardware countermeasures against side-channel and fault-injection attacks. Cryptology ePrint Archive, Report 2016/648 (2016), http://eprint.iacr.org/2016/648
16. Selmane, N., Bhasin, S., Guilley, S., Graba, T., Danger, J.L.: WDDL is Protected Against Setup Time Violation Attacks. In: FDTC. pp. 73–83 (2009)
17. Servant, V., Debande, N., Maghrebi, H., Bringer, J.: Study of a Novel Software Constant Weight Implementation, pp. 35–48. Springer International Publishing, Cham (2015), http://dx.doi.org/10.1007/978-3-319-16763-3_3
18. Tiri, K., Verbauwhede, I.: A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In: DATE'04. pp. 246–251 (2004), Paris, France.
19. Trichina, E., Korkikyan, R.: Multi fault laser attacks on protected crt-rsa. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on. pp. 75–86 (Aug 2010)

A Generated Codes

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

Table 7: Codes generated by the Algorithm 1.

Code	Length	Distance	Denoted by
0x37, 0xFD, 0xFE, 0xDB, 0xCF, 0xF4, 0xF1, 0xE5, 0xF2, 0xE6, 0xE3, 0x7F, 0xD6, 0xD5, 0xEA, 0xE9	8	>= 2	$C_{8,min2}$
0x2E, 0xFB, 0xFC, 0x76, 0xB7, 0xE1, 0xCE, 0x5F, 0xD2, 0xD5, 0x6D, 0x43, 0xA2, 0x8B, 0x58, 0x44	8	>= 3	$C_{8,min3}$
0x08, 0x16, 0x23, 0x3D, 0x45, 0x5B, 0x6E, 0x70, 0x8F, 0x91, 0xA4, 0xBA, 0xC2, 0xDC, 0xE9, 0xF7	8	>= 4	$C_{8,min4}$
0x62, 0x1FD, 0x1FE, 0x1EA, 0xDF, 0x1E9, 0x1BF, 0x1EC, 0x16F, 0x1F8, 0x17B, 0xF5, 0x1D5, 0xF6, 0x1D6, 0xF3	9	>= 2	$C_{9,min2}$
0xBC, 0x1FA, 0x1FD, 0xD7, 0x1E1, 0x1B7, 0x167, 0x1CB, 0xEE, 0x15F, 0x1C6, 0x174, 0x7B, 0x1D0, 0xCD, 0x19E	9	>= 3	$C_{9,min3}$
0x5A, 0x1F7, 0x1F8, 0xEE, 0x1CD, 0x16B, 0x7D, 0x1C2, 0xE1, 0xD4, 0x164, 0x19B, 0x151, 0x13E, 0x47, 0xB2	9	>= 4	$C_{9,min4}$
0x1C1, 0x3FD, 0x3FE, 0x3C7, 0x2E7, 0x3D3, 0x2F3, 0x3D6, 0x1D7, 0x2F6, 0xF7, 0x3D5, 0x2F5, 0x2DF, 0x3EA, 0x1EB	10	>= 2	$C_{10,min2}$
0x6B, 0x3FB, 0x3FC, 0x2D7, 0x3E2, 0x3B7, 0x367, 0x1E5, 0x3CE, 0x17F, 0xFE, 0x2ED, 0x1DD, 0x1D3, 0x2F1, 0x1F0	10	>= 3	$C_{10,min3}$
0x12F, 0x3F7, 0x3F8, 0xFB, 0x1DE, 0x3CD, 0x2EE, 0x1E2, 0x35B, 0x27D, 0x2E1, 0x1D1, 0xF4, 0xC7, 0x2D2, 0x364	10	>= 4	$C_{10,min4}$

Table 8: Side-channel and fault properties of the codes from Table 7.

α	0.613331, 0.644584, 0.602531, 0.190986, 0.586268, 0.890951, 1.838814, 1.257943, 0.899922, 0.614699								
Code	$C_{8,min2}$	$C_{8,min3}$	$C_{8,min4}$	$C_{9,min2}$	$C_{9,min3}$	$C_{9,min4}$	$C_{10,min2}$	$C_{10,min3}$	$C_{10,min4}$
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	2.0515	0.1910	3.3250	3.1977	2.6648	4.5902	4.1370	3.7101
Bit Variance	0.4657	0.4560	0.3367	0.2832	0.4657	0.3797	0.2231	0.3565	0.3430
p_1	1	1	1	1	1	1	1	1	1
p_2	0.8169	1	1	0.9097	1	1	0.9111	1	1
p_3	0.9554	0.9040	1	0.9643	0.9464	1	0.9823	0.9687	1
p_4	0.9161	0.9161	0.8	0.9603	0.9563	0.9067	0.9696	0.9738	0.9512
p_5	0.9821	0.9687	1	0.9792	0.9772	1	0.9955	0.9861	0.9950
p_6	0.9821	0.9598	1	0.9911	0.9821	0.9717	0.9946	0.9923	0.9899
p_7	1	0.8906	1	0.9896	0.9826	1	0.9979	0.9948	0.9958
p_8	1	1	0	1	1	0.9027	1	1	0.9833
p_9	-	-	-	1	1	1	1	1	0.9875
p_{10}	-	-	-	-	-	-	1	1	1
$p_{2,(e)}$	-	0.4241	1	-	0.6250	1	-	0.7450	1
$p_{3,(e)}$	-	0.4844	0	-	0.6845	0.4405	-	0.7854	0.6583
$p_{4,(e)}$	-	0.4071	0.8	-	0.6280	0.9067	-	0.7655	0.9214
$p_{5,(e)}$	-	0.4286	0	-	0.6875	0.4206	-	0.8165	0.7004
$p_{6,(e)}$	-	0.5536	1	-	0.7932	0.9717	-	0.8881	0.9435
$p_{7,(e)}$	-	0.6094	0	-	0.8576	0.6076	-	0.9406	0.8750
$p_{8,(e)}$	-	0.1250	0	-	0.86111	0.9028	-	0.9583	0.9250
$p_{9,(e)}$	-	-	-	-	1	0.1250	-	1	0.8375
$p_{10,(e)}$	-	-	-	-	-	-	-	1	0.8750