Improvement and integration of software tools for the evaluation and realization of Physical Unclonable Functions (PUFs) into an open-source library of cryptographic components (CogniCrypt)

Weiterentwicklung und Integration von Werkzeugen zur Evaluation sowie Realisierung von Physical Unclonable Functions(PUFs) in eine Open Source Bibliothek für Kryptografische Komponenten (CogniCrypt)

Master-Thesis von Prankur Chauhan

Tag der Einreichung: 27. Oktober 2017

1. Gutachten: Prof. Dr. Stefan Katzenbeisser
2. Gutachten: Nikolaos Athanasios Anagnostopoulos
3. Gutachten: Prof. Dr. Mira Mezini
4. Gutachten: Michael Reif

TECHNISCHE UNIVERSITÄT DARMSTADT

Collaborative Research Center
CROSSING
Security Engineering Group
Software Technology Group

Improvement and integration of software tools for the evaluation and realization of Physical Unclonable Functions (PUFs) into an open-source library of cryptographic components (CogniCrypt)

Weiterentwicklung und Integration von Werkzeugen zur Evaluation sowie Realisierung von Physical Unclonable Functions(PUFs) in eine Open Source Bibliothek für Kryptografische Komponenten (CogniCrypt)

Vorgelegte Master-Thesis von Prankur Chauhan

1. Gutachten: Prof. Dr. Stefan Katzenbeisser
2. Gutachten: Nikolaos Athanasios Anagnostopoulos
3. Gutachten: Prof. Dr. Mira Mezini
4. Gutachten: Michael Reif

Tag der Einreichung: 27. Oktober 2017

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 27. Oktober 2017

(Prankur Chauhan)

Contents

1 Introduction

As the internet grows so does the threat to its infrastructure. In recent wake of the Heartbleed bug, the threat to cloud storage and the NSA debacle tracking private data, security becomes more important than ever.

Well-known protection mechanisms like symmetric shared key-based or asymmetric public/private key-based schemes require algorithms to generate secure keys and/or pass certificates between two communicating parties. Generation and storage of keys and certificates on embedded devices with limited hardware resources is cost ineffective. Securing the data in the memory requires specialized hardware security modules which are expensive. For example, a specialized tamper-proof hardware may cost more than 3000 USD [1].

Physical(ly) Unclonable Functions (PUFs) [2] provide a low-cost solution to the economic problem of key-generation on limited resource hardware embedded devices. PUFs are a result of the manufacturing variations while printing the Integrated Circuits (ICs); these variations are unclonable which means even the manufacturer cannot generate two identical ICs. A PUF depends on its physical microstructure and exhibits challenge/response behavior to evaluate its structure. It is a hardware analogy to a one-way function that is easy to evaluate but difficult to predict. When a physical stimulus (challenge) is applied to the structure it outputs an unpredictable (but repeatable) response which depends on the physical factors introduced during manufacturing of the PUF. So the PUF is unique and can be seen as a fingerprint of the hardware [3].

A fuzzy extractor [4] can be applied to the PUF to derive an exclusive and strong cryptographic key from the underlying physical microstructure. Since, the output from the evaluation is reproducible and unique, every time the same key is generated. Some PUFs like SRAM PUFs may not need additional costs and are generated implicitly by variations in the manufacturing of SRAM modules itself.

There are a lot of applications of PUFs like realization of a secure key storage [5–7], Intellectual Property (IP) protection and remote service activation [8, 9], anti-counterfeiting [10, 11], device authentication and secret key generation [12–14], remote attestation protocols [15, 16] and the integration in cryptographic algorithms [17]. Based on the use case, PUF instances must satisfy different criteria and implement special properties to be integrated into the security mechanism.

The behavior of a PUF instance is sensitive to differing operational conditions such as supply voltage, ambient temperature or aging effects that can result in slightly different PUF responses each time the PUF is challenged. Consequently an adequate pre-assessment of the PUF-instance is required to ensure that the response is consistent, required criteria are accomplished and essential properties are provided for the desired security mechanism. This pre-evaluation also verifies the stability of the PUF reaction under fluctuating operating conditions. For security mechanism, correct and stable working behavior plays a major role; specifically if the mechanism was built on top of a noisy PUF-instance or if the PUF-instance is the central building block.

CogniCrypt is an opensource library that is implemented as an Eclipse plugin that supports Java developers in using Java Cryptographic APIs [18]. Cryptographical algorithms are highly configurable and a configuration which may be secure in one scenario might have flaws and loopholes in another. CogniCrypt aims to provide non-security expert Java developers an opportunity to securely and correctly use the underlying cryptographical APIs.
It classifies the different configurations based on developer's answers to high-level questions in non-expert terminology [19]. Apart from generating secure implementations for cryptography tasks it also analyzes developer code and generates alerts for misuses of cryptographic APIs [18].

1.1 Goal of the Thesis

The goal is to extend the user interface that helps designers and researchers to evaluate PUF responses and then integrate the toolkit as a JAR archive in CogniCrypt wrapped in Java Native Interface.
This Thesis is an extension of the Master Thesis "Development of a user interface and implementation of specific software tools for the evaluation and realization of PUFs with respect to security applications" [20] where the original UI was designed and implemented. The design did not incorporate BCH encoding and decoding in the toolkit and was presented as separate UI. The secure key storage approach in [14] was realized using a fuzzy extractor, which was based on the syndrome generation of a Golay(23,12,7) error-correction code in combination with a binary linear repetition code and a binary majority voting for the reconstruction [20]. Golay (23, 12, 7) error-correction code can correct three errors in each syndrome. Therefore an implementation of this error-correction code was used to realize the fuzzy extractor, which was included in the toolkit.

1.2 Contribution

This Thesis aims to improve the UI, add new functionalities to the PUF response evaluation, integrate a BCH and a Golay encoder and decoder for fuzzy extractor scheme to the toolkit. Then wrap the toolkit in Jar to be accessible by CogniCrypt via Java Native Interface.
In particular, the contribution of this work is characterized by the following points:

- Improvement of a console user interface for the PUF Toolkit in C++.
- Implementation of specific software tools with respect to PUF relevant metrics in C++.
  - Hamming Distance Menu
  - Golay encoder
  - Golay Decoder
  - Intra Jaccard Index

    - Inter Jaccard Index

- BCH encoder and decoder integration to the PUF Toolkit.
- Augmenting the offset functionality.
- Enhancing the toolkit to output fractional distance.
- Wrapping the toolkit via JNI to be made accessible in CogniCrypt.
- UI implementation for CogniCrypt module to generate boilerplate code.

1.3 Outline

The present work is divided into the following chapters: Chapter 2 covers the state of the art research with a focus on Physical(ly) Unclonable Functions, their definition, classification, working principles, and categorization.In addition, some information and mathematical concepts behind linear codes and cyclic codes like BCH error-correction code are given. In Chapter 3, the original version of the PUF Toolkit is summarized and extended metrics implementation and improvement to the Toolkit is explained. The integration of the fuzzy extractor to the toolkit and extension of the fuzzy extractor is present thereafter. The integration of the PUF Toolkit into CogniCrypt, an Opensource Java-based library is presented towards the end of Chapter 3. The testing and evaluation of the Toolkit are presented in Chapter 4. A final conclusion can be found in Chapter 5 and future work is proposed in Chapter 6.

2 Related Work

This chapters explains the related work that is helpful in understanding the concepts, mathematical constructs, terms, and definitions that are necessary to understand the working details of the thesis. The first part discusses Physical(ly) Unclonable Functions, second part introduces the concept of the BCH code, third part talks about Golay code constructs and finally a synopsis of Jaccard Index is presented.

2.1 Physical(ly) Unclonable Functions - Background

In order for the reader to have a better understanding of the fundamentals of Physical(ly) Unclonable Functions, this section discusses some basics of PUFs. Initially the historical emergence of PUFs is presented, followed by a general definition of the PUF. Then the PUFs are classified and a selection of different types of PUFs is presented. Lastly, the basics of linear codes like BCH and Golay codes are discussed that form the basis for the fuzzy extractor. Work in this section is mostly derived from [21, 22].

2.1.1 History and origins

Physical(ly) Unclonable Functions (PUFs) are based on unique and non-reproducible artifacts, which were caused by production variances during manufacturing processes. PUFs were first presented in the early eighties [23]. Fingerprint identification of humans goes back to at least nineteenth century [24] and from that emerged the field of biometrics. In the twentieth century, random patterns in paper and optical tokens were used for exclusive identification of currency notes and strategic arms [5, 12, 25]. A formalization of this concept was introduced in the beginning of the twenty-first century. In 2001, Pappu et al. [26, 27] presented *physical one-way functions*. The next year 2002, Gassend et al. [28] proposed a silicon-based PUF approach as a *physical random function*. This led to coining of the acronym PUF (*Physical(ly) Unclonable Function*) to avoid confusion with the concept of pseudo-random functions (PRF), which was an already established concept in cryptography.

The promising properties of PUFs like physical unclonability and tamper evidence which are favorable for security mechanisms, led to the increase in popularity of PUFs and in coming years new types of PUFs were proposed. Due to their practical usages and encouraging properties the interest in PUFs has risen significantly; they are still a hot topic in the field of hardware security and can contribute to the evolution of security mechanism and applications.

2.1.2 Definition

The definition of PUFs is taken from Gassend et al. [28]:
A Physical(ly) Unclonable Function (PUF) is a function that maps challenges to responses, that is embodied by a physical device, and that verifies the following properties:

1. Easy to evaluate: The physical device is easily capable of evaluating the function in a short amount of time.
2. Hard to characterize: From a polynomial number of plausible physical measurements (in particular, determination of chosen challenge-response pairs), an attacker who no longer has the device, and who can only use a polynomial amount of resources (time, matter, etc.) can only extract a negligible amount of information about the response to a randomly chosen challenge.

To simplify, PUFs are functions that use hardware manufacturing process variations to generate a random output. They are easy to evaluate which means that for a given input, the result is extracted without much effort. Unclonable keyword implies the output function cannot be duplicated to make another PUF. Random response means it contains an equal number of ones and zeros.

2.1.3 PUF terminologies

This section introduces commonly used terms used for describing PUFs and their characteristics. Work in this subsection is inspired from [29]

2.1.3.1 Challenge and Responses

As discussed in the definition, PUF produces output on being queried with an input. Since the input may have more than one possible output so PUFs are not functions in a mathematical sense rather they are considered function in engineering sense i.e. a procedure performed by or acting upon a specific (physical) system. [29]. The input to PUF is called *Challenge* and output is termed as *Response*, together they are known as *Challenge-Response Pair* or *CRP* and the relation imposed between challenges and responses by a particular PUF is called as its *CRP behavior*. PUF is applied in two phases, the first phase is *enrollment*, where a certain number of CRPs are collected from a PUF and stored in a *CRP database*. In the second phase called *verification*, a challenge from CRP database is applied to the PUF and the produced response is compared with the corresponding response from the database (section of 2.1 [29]).

2.1.3.2 Intra- and Inter-Distance metrics

The concept of inter-versus intra-(class) distance is inherited from the theory of classification and identification:

- for a specific challenge, inter-distance between two PUFs instances is the distance between two responses produced by applying the same challenge to both PUFs.
- for a specific challenge, the intra-distance between two evaluations of the single PUF is the distance between two responses produced by applying the same challenge twice to the same PUF. (section of 2.2 [29])

In our case, we deal with challenges and responses that are output in bit strings (after decoding and quantization of analog physical stimuli and measured effect), so Hamming Distance is a good metric to measure the difference in the responses i.e. degree by which the responses from the same challenge differ. To amplify the Hamming distance it is often expressed as a fraction of the length of the considered strings, in that case it is known as *fractional Hamming distance*

2.1.4 Reliability Issues

As pointed out earlier in the introduction, PUF is sensitive to external factors and therefore the output of PUF is not consistent. These factors can be inevitable random noise or measurement uncertainties which give rise to the intra-distance between PUF responses (section 2.3 of [29]). Apart from these undesirable effects PUFs are susceptible to other sources of noise e.g. varying temperature or supply voltage on PUFs integrated circuit, these factors contribute to a *systematic* effect on response measurement. Another cause for unreliable responses is the *aging effect*, which causes gradual degradation of the device resulting in varying PUF responses. Consequently, PUF responses need post-processing techniques to be applicable in practical security scenarios. One such case is extracting a reliable key from PUF that requires a scheme called Fuzzy Extractor (covered in more detailed in later section 2.2.3).

2.1.5 Classifications

The popularity of PUFs and immense research on the topic has resulted in various concepts and constructions, we talk about the ones that are concerned with this thesis and for completeness briefly describe other classifications.
The following sections talk about the classification of PUFs that are subdivided into three different groups based on [21]:

1. Implementation technology and material of the proposed construction
2. A more general set of physical construction properties
3. Algorithmic properties of the challenge-response behavior

The first criterion for classification is the implementation technology of the construction. PUFs can be realized with different technologies and materials like glass, plastic, paper, electronic components and integrated circuits (ICs). Essentially there are two classes in this group based on the *electronic* nature of the identifying features.

Non-electronic PUFs are PUFs where the nature of the components in the system that contributes to the random physical microstructure of unique PUF is of non-electronic origin. It should be noted that keyword 'non-electronic' only reflects the origin of the PUF behavior, post-processing (like fuzzy extraction) or intermediate steps can be electronic. Examples can be Optical PUFs where the core element is an optical microstructure constructed by mixing microscopic refractive glass spheres in a transparent epoxy plate (section 3.1.1 of [29]). A unique response is obtained upon irradiating the transparent material with a laser, this can be interpreted as a PUF response. Another example are Paper-based PUFs where the random fiber structure of the paper is scanned using a laser. The reflection from the erratic fiber serves as a unique identifier and can be considered as a PUF response [29].

Electronic PUFs proposed construction contains random variations derived from electronic properties of the underlying material e.g. resistance, capacitance etc. *Silicon PUFs* are a major subclass of electronic PUFs. They were the first practical realization of PUFs introduced by Gassend et al. [28]. Since Silicon PUFs can be connected to the integrated circuit on the same chip, they can be instantly deployed as hardware building block in cryptographic implementations. [21]. Another example is a construction called RF-DNA where copper wires are arranged randomly on a silicone medium to generate PUF responses.

The second classification which is based on the construction properties subdivides the PUFs in **Intrinisic PUFs** and **Non-Intrinsic PUFs**. Initially proposed by Guajardo et al. [9], *Intrinsic PUFs* must meet following two conditions:

- the evaluation of the PUF is performed internally (the measurement instrument is integrated into the device).
- the random instance specific features are implicitly imported during its manufacturing process.

We look with some detail in the above two conditions. First one states that the evaluation, which is a physical measurement, can be internal or external. An external evaluation is the measurement of features that are externally observable and is done with the help of equipment external to the physical entity. Internal evaluations, on the other hand, are measurements of features that are internal to the instance and is performed by equipment absolutely embedded in the instance itself. There are some advantages for internal evaluations, one there is a less security risk since the measurement entity is embedded in the instance and the PUF response is protected from the outside world. Also, there is a practical leverage because every PUF instance can evaluate itself without any external restriction. Hindrance from outside influences and measurement errors are also minimized in the internal evaluation. With *Non-intrinsic PUFs*, the external measurement can pose a security hazard if an adversary observes the PUF response.

The second one distinguishes between the source of the random variations. The manufacturer can explicitly add randomization procedure to introduce random features that are later measured during PUF evaluation or measured random features can be an implicit side-effect introduced during the production of the integrated circuit and PUF instance. This means they arise naturally and cannot be controlled by the manufacturer, hence even he/she cannot reproduce and clone the same random features. These implicit random variations incorporated during manufacturing are called *process variations* come at no extra cost, so they incur no additional cost and are attractive from economic viewpoint.

The final classification is based on the Challenge-Response behavior security properties. Guajardo et al. [9] introduced the concept of **Strong** and **Weak PUFs** which was further expanded by Rührmair et al. [30]. They state that a Strong PUF is a PUF that an adversary can have access to for a long period of time and still he/she cannot uncover the working of the PUF function or how the PUF is evaluated to map to a specific response. In other words, we can still come up with a new challenge that the adversary does not know what the response to it will be. Consequently, this implies that:

- the considered PUF has a very large challenge set, or else the adversary can apply brute force to know responses to all challenges.

- it is not possible to know the mapping function between CRPs based on observed CRPs, that means PUF is unpredictable.

PUFs that are not *Strong PUFs* and those which have a small challenge set are called *Weak PUFs*. Naturally Strong PUFs are favored in a good securely designed application as compared to Weak PUFs. However, construction of such a Strong and practical (intrinsic) PUFs is rather difficult and still an open problem [30].

2.1.6 PUF types

This section presents a selection of different PUF constructions according to [21]. The general working principles of different PUF realizations are explained accompanied with security-related information.

2.1.6.1 Arbiter PUFs

Lee et al. [31] proposed arbiter PUF as a delay-based silicon PUF. The idea is to have two digital paths with random process variations and explicitly introduce a race condition between signals that flow via these digital paths. The paths end in an arbiter circuit that referees the race i.e it resolves which of the paths was faster and correspondingly outputs a binary bit. It is made sure that both paths are designed to have (nearly) same nominal delays, therefore the result of the race and hence response of the arbiter cannot be precisely determined based on the design. As we discussed even after having almost identical nominal delays, the delay experienced by the signal is a random delay that a side effect (in our case the desired effect) of the random silicon process variations on the delay parameters. This silicon process variation is random per device but static for a particular PUF device. So we can have the arbiter circuit device specific random response as the basis for the PUF behavior.

In case both paths happen to have same nominal delays (due to complex nature of the electronic circuitry and used gates), both signals arrive almost synchronously at the arbiter in which case arbiter circuit goes in a *metastable state* and logic output of the arbiter circuit is temporarily unknown (electronically, voltage of arbiter output is between levels for a logic high and low). After a short random backoff time arbiter switches from its metastable state and outputs a random binary value not dependent on the conclusion of the race, but in this case, the output of the arbiter is not device-specific and not static. This phenomenon is the origin of unreliability (noise) of the responses in an arbiter PUF.

The implementation of arbiter PUF consists of two delay paths which contain a chain of switch blocks that connects two input signals to its two outputs either in a cross or straight configuration. This configuration is modifiable via a configurable selection bit. The Figure 2.1 shows the inner cross-section of the switches, which are logically implemented as 2-to-1 multiplexers (muxes). A total n number of switch blocks are linked with each block having a selection bit (n configuration bit-vector) to configure a total of 2^n possible delay path pairs. This n-bit vector is considered as a challenge, and the random process variations in the delay paths make sure that the input-output delays are different and each of the 2^n paths leads to a race

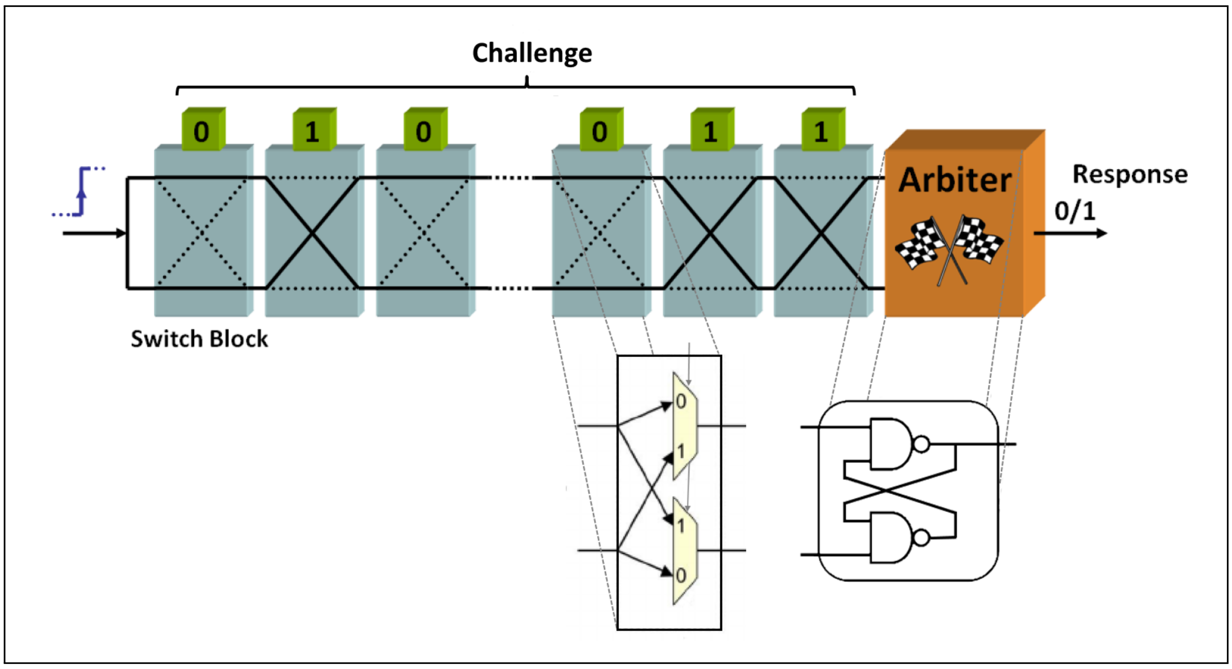


Figure 2.1: Construction of a basic arbiter PUF, *switching blocks* realized with muxes and an *SR latch* realized with NAND gates, modified graphic based on [21, 22, 32]

to be resolved by the arbiter circuit. The final output bit from the arbiter is considered as a response to the PUF challenge. Arbiter circuit can be implemented as a simple SR NAND latch or some other digital latch/flip-flop. Following two conditions are important to determine if the PUF response depends individually on process variations or on the delay parameters:

- the delay paths are designed to be symmetrical, the cause of any difference is process variation.
- arbiter circuit must not be biased for one input over other.

Security Issues: Though the challenge set is large still the number of delay parameters which effect the PUF response are linear in n, hence the 2^n challenges cannot generate independent responses. A certain model or mathematical clone can be built for a given arbiter PUF if the adversary learns the underlying delay parameters. This clone can accurately forecast the PUF responses rendering the arbiter PUF hackable in security applications scenarios. Machine learning techniques like artificial neural networks (ANNs) and support-vector machines (SVMs) can be employed to implicitly learn the underlying delay parameters by observing the challenge-response pairs.

2.1.6.2 SRAM PUFs

The concept of an SRAM PUF was introduced by Guajardo et al. [9] and Holcomb et al. [33] in 2007. Static Random-Access Memory (SRAM) is a digital memory technology based on bistable circuits. (section 2.4.4 of [21]). It consists of six MOSFETs transistors, out of which four are contained in two inverters. Each inverter comprises of one p-MOS and one n-MOS MOSFET. As you can see from Figure 2.2 the inverters are cross-coupled at the SRAM cell's core. In the terms of mathematical logic, the circuit has two stable states (bistable), each state represented by a binary digit (0 or 1) that is stored in the cell.

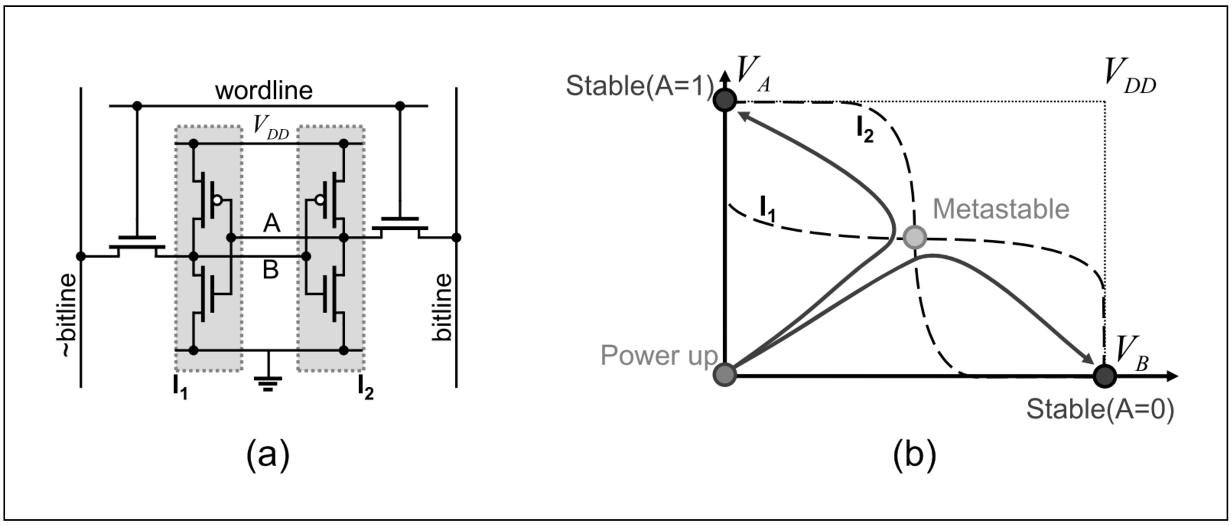


Figure 2.2: (a) Construction of a standard CMOS SRAM cell circuit, (b) initial voltage transfer curves [21].

Two MOSFETs are used to read and write the cell contents. An SRAM cell is volatile and does not store its state on power-off.

There are three possible operating points, two are stable and one is metastable as shown in Figure 2.2. The deviation from stable points is automatically restored to its original point due to the feedback from the circuit. Adversely any deviation from a metastable point is amplified by the positive feedback from the circuit and cell moves to either of the stable points. After the supply voltage V_{DD} comes up, the cell shifts to one of the preferred stable points, which is determined by the difference in strength (device mismatch) of the MOSFETs in the cross-coupled inverter circuit. Due to performance and efficiency reasons, the inverters are designed to be perfectly in sync, the device mismatch is a result of the random process variations in the silicon production process. This random preferred initial operating point is unique to a specific cell. For a small mismatch, the preferred initial stable point is determined by the sign of the mismatch, though voltage noise can render the cell to power-up in a non-preferred state. Finally the cells with almost zero mismatch power-up in the metastable state and shift to one of the stable points randomly.

Most SRAM cells have strongly preferred cell-specific initial state due to the magnitude of the process variations on the device mismatch; only a small number of the cells have weakly preferred or no preferred state, hence a typical SRAM cell exhibits strong PUF behavior. These cells are arranged in a large array giving millions of response bits, challenge for this cell assembly is the address of the cell.

2.1.6.3 Ring Oscillator PUF

Proposed by Gassend et al. [28] it is another type of delay-based intrinsic PUF. There are two parts to ring oscillator PUF, first is the ring oscillator and other is the frequency counter which are arranged in a configuration as shown in the Figure 2.3 and connected to a response generating algorithm. Gassend et al. uses a variant of the switch block base delay line similar to arbiter PUF 2.1.6.1 which is transformed

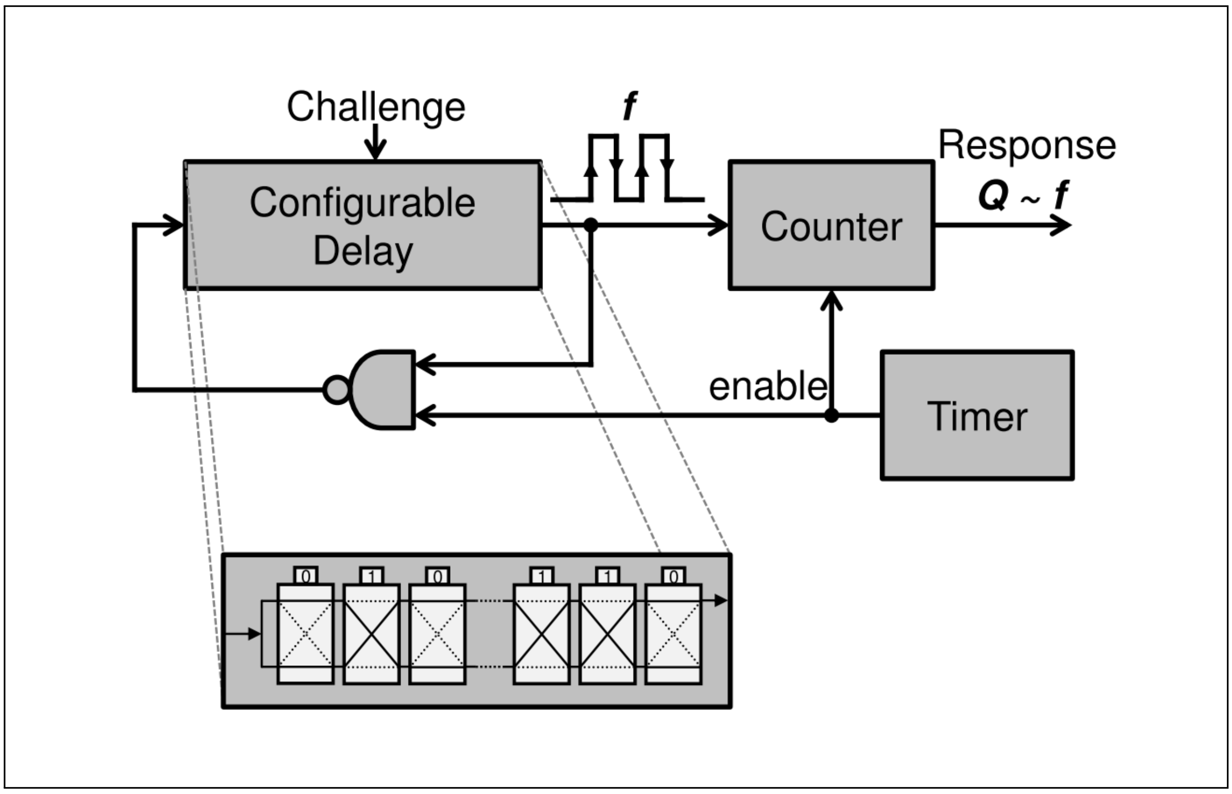


Figure 2.3: Construction of a basic ring oscillator PUF as proposed by Gassend et al. [21, 28].

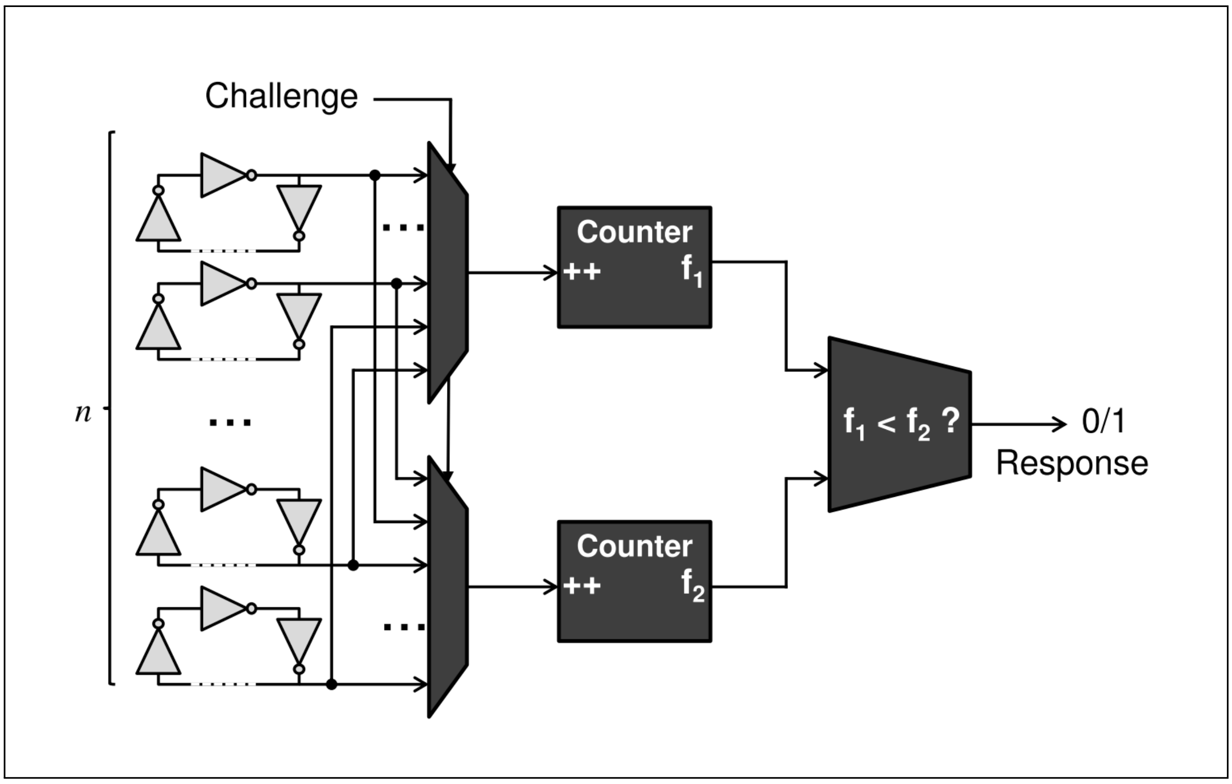


Figure 2.4: Construction of a ring oscillator PUF as proposed by Suh and Devadas [12, 21].

in an oscillator with the help of a negative feedback. An AND gate is added to turn off/on the oscillation, which is fed to a frequency counter that counts the number of oscillating cycles in a fixed time interval.

The same setup for ring oscillator on another device with the same implementation has different counter value because of the silicon process variations which forms the basis of the PUF response.

There are some other side effects associated with ring oscillator PUFs, one of them is significant influence from the environmental conditions like temperature changes and voltage fluctuations. The frequency changes introduced by these factors outweigh the deviations caused by the process variations, so to counteract these changes we need a post-processing technique known as *compensated measuring*. The main idea is to calculate the ratio of the frequency of two ring oscillators on the same device since they both are affected by identical environmental factors, hence the ratio would be more uniform. According to observations average intra-distance between evaluations is approximately 10% and the average inter-distance is approximately 50% (section 2.4.2 [21])

2.1.6.4 Optical PUFs

Even before the introduction of PUFs an unclonable identification system based on random optical reflection patterns was proposed in [34]. Proposed by Pappu et al. as Physical One-Way Functions (POWFs), optical PUFs contain a microstructure constructed by blending microscopic (500 μm) refractive glass spheres in a miniature (10 x 10 x 2.54 mm) transparent epoxy plate. This token is illuminated with a helium-neon laser which produces an irregular wavefront; the cause for this irregularity is the multiple scattering of the laser by the refractive particles. Finally, a CCD camera captures the speckle pattern and digitally processes it by applying Gabor hash to it as a feature extraction procedure (section 3.1.1 of [29]). The final output is a string of bits that deviates significantly if the orientation of the laser beam is even minutely changed.

The PUF challenge is made up from the exact positioning of the laser and the resulting Gabor hash of the emerged speckle pattern is considered as a response. The operation and implementation for optical PUFs are shown in Figure 2.5. A number of experiments were performed in [26, 27] for testing the characteristics of the optical PUFs and the following text summarizes their results. The values are taken from [26, 27]. In total, four optical tokens were used with 576 individual challenges, to calculate the PUF responses. Intra- and inter-Hamming distances were evaluated with an average inter-distance of μ_{inter} = 49.79% and average intra-distance of μ_{intra} = 25.25%. One of the main limitations of such an optical PUF is the burdensome large setup consisting of a laser and mechanical positioning system. They are classified as non-electronic, non-intrinsic and strong PUFs.

2.2 Linear Codes, BCH and Golay Codes

In order to understand the working of the toolkit and fuzzy extractors (discussed later in section 2.2.3), we need to look at two specific types of linear codes. The following sections discuss the basics of linear codes and their mathematical constructs. After the initial explanation of what linear codes are, we go on to explain BCH codes and Golay codes that are used in the PUF toolkit to extract private and public keys in conjunction with PUFs. The second part gives the details about two important concepts related to

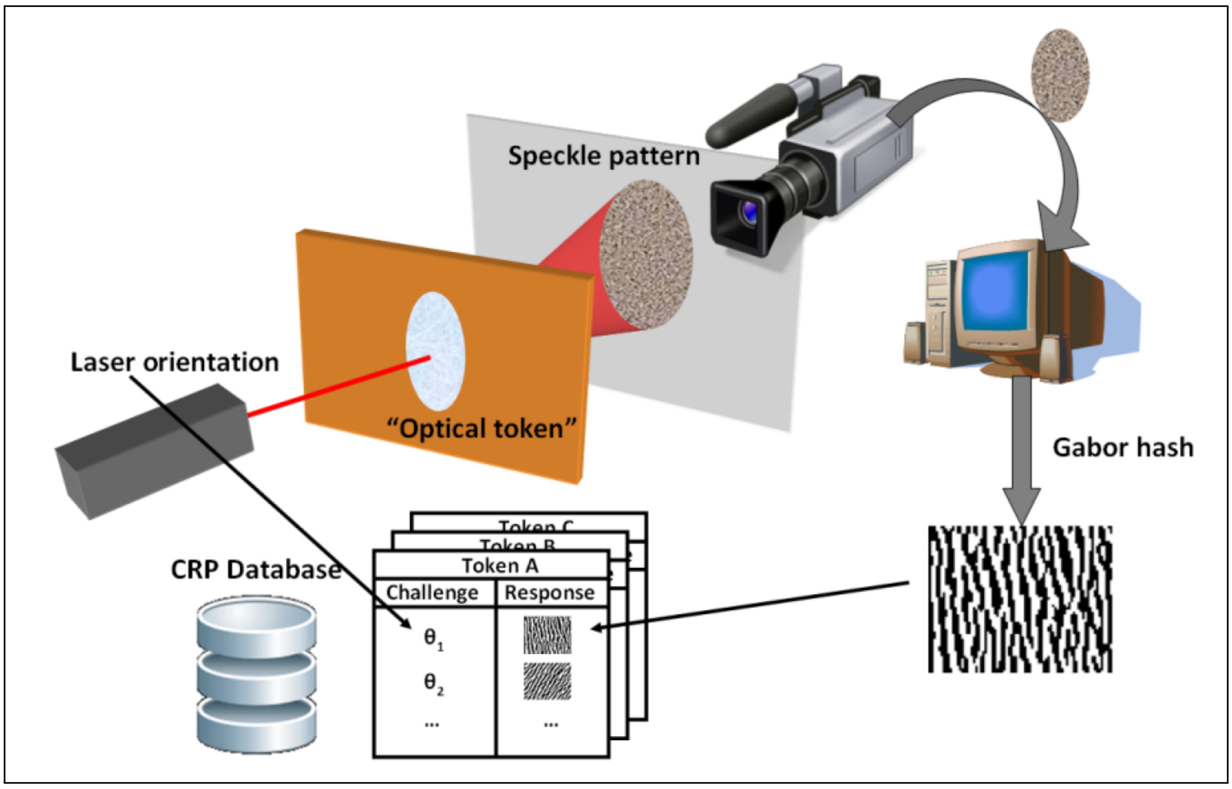


Figure 2.5: Construction of a basic optical PUF as proposed by Pappu et al. [22, 26].

coding known as Majority Voting and a simple Linear Repetition Code.The last part of this section deals with a technique called fuzzy extractor that takes the concepts of the linear codes, LR codes, Majority Voting and combines them to reevaluate and generate the helper data for a specific PUF response used for secure key storage on the PUF instance. The major content for these sections is inspired from [35].

Before diving into linear codes let us first look at the definition of a field in mathematics from the book [35]:
Definition 1: A field F is given by a triplet (S, +, ·), where S is the set of elements containing special elements 0 and 1 and +, · are functions $F \times F \rightarrow F$ with the following properties:

- Closure: For every a, b ∈ S, we have both a + b ∈ S and a · b ∈ S.
- Associativity: + and · are associative, that is, for every a, b, c ∈ S, a +(b +c) = (a +b)+c and a · (b · c) = (a · b) · c.
- Commutativity: + and · are commutative, that is, for every a, b ∈ S, a + b = b + a and a · b = b · a.
- Distributivity: · distributes over +, that is for every a, b, c ∈ S, a · (b + c) = a · b + a · c.
- Identity: For every a ∈ S, a + 0 = a and a · 1 = a.
- Inverse: For every a ∈ S, there exists a unique additive inverse -a such that a + (-a) = 0. Also for every a $\in S \setminus \{0\}$ there exists a unique multiplicative inverse a^{-1} such that $a \cdot a^{-1} = 1$.

Lemma: Let p be a prime. Then $F_p = (\{0,1,...,p-1\}, +_p, \cdot_p)$ is a field, where $+_p$ and $\cdot_p$ are addition and multiplication mod p. [35]

For example, $F_2 = \{0, 1\}$ is a binary finite field with two symbols 0 and 1 where $+$ and $\cdot$ are operations modulo 2.

Linear Subspaces. Before defining linear codes we need to know what is a linear subspace.
Definition 2: $S \subseteq F_q^n$ is a linear subspace if the following properties are satisfied [35]:

1. For every $x, y \in S, x + y \in S$, where addition is vector addition over F_q
2. For every $a \in F_q$ and $x \in S, a \cdot x \in S$, where multiplication is over F_q

A simple example to understand above-mentioned Definition 2 and Lemma is F_5^3
S = {(0, 0, 0), (1, 1, 1), (2, 2, 2), (3, 3, 3), (4, 4, 4)}. As you can see (1, 1, 1) + (3, 3, 3) = (4, 4, 4) $\in S$ and 2·(4, 4, 4) mod 5 = (3, 3, 3) $\in S$ as stated in the Definition 2.
Now, we can define a **linear code** as a code of length n over the field F which is a subspace of F^n . The words of the codespace F^n are vectors, and we often refer to codewords as codevectors. If C is a linear code that, as a vector space over the field F , has dimension k, then we say that C is an [n, k] linear code over F , or an [n, k] code [36]. The *distance* of the code is the number of symbol changes required in a codeword to assume another codeword. E.g., Let C(3,3) be a subspace in F_2^3, with two codewords 000 and 111; then the distance between them is 3 since we need to change three symbols, from 0 to 1. The *rate* of an [n, k] linear code is **k/n**. If C has a minimum distance d, then C is an [n, k, d] linear code over F . The number n - k is called the *redundancy* of C [36].

A **cyclic code** is a linear block code where, if c is a codeword, then all the cyclic shifts of c are also codewords. For example a code with codewords 000, 110, 101, 011 is a cyclic code. These codes can be completely produced by a generator string G, where all codevectors are multiples of G.

This section tries to cover the basics for understanding BCH codes and Golay Codes, in no way we claim the completeness of the mathematical constructs and other proofs that are skipped. Please see the references for more details.

2.2.1 BCH Codes

The *BCH codes* belong to the class of powerful random error-correction cyclic codes and named after Bose, Chaudhuri and Hocquenghem. These codes exercise great control over code's parameters like code length, information bits etc. The major parameters to the BCH code are the length of the input message (input bits) which produces output codewords based on the *BCH mode* selected and a number of bit errors that can be corrected with decoding. This section tries to summarize the BCH codes and their basic error correcting capability.

A BCH code consists of three code parameters or has three dimensions; the code length n of the codevectors, that are generated from the input bits of length k and lastly a very important parameter d that is called minimum distance, this d is responsible for deriving the error correcting capability of BCH code

$t = \frac{d}{2} + 1$, i.e. the maximum number of bit errors the code can correct. These parameters are inter-dependent on each other and cannot be set directly; alternatively to set the legitimate BCH modes the user needs to set m and t. The parameter m determines the degree of the primitive polynomial $p(x)$ that is used in computing the Galois Field $GF(2^m)$ and also to calculate the coefficients of $p(x)$ which are crucial for code generation. The length of the codewords must fall within the range: $2^{m-1} - 1 < n \leq 2^m - 1$. The other parameter t ascertains the maximal error correcting capability for the code and has explicit influence over redundancy created via the encoding process. The minimum distance parameter d is calculated via the formula $d = 2 * t + 1$, the redundancy bits that are mandatory for a successful bit correction in the codeword are calculated w.r.t. d. So we can infer that the error correction is restricted by code length n and the number of input bits k are defined by $k = n - redundancy$, where the value of t must be such that $k \geq 2$. Now we can formally define BCH code, this definition is taken from [37]:

For a positive integer m (m ≥ 4 and m ≤ 14) and t ($t < 2^{m-1}$), there exists a binary BCH (n, k, d) code with the following parameters:

- **Block length:** $n = 2^m - 1$
- **Number of redundancy bits:** $n - k \leq mt$
- **Minimum distance:** $d_{min} \geq 2t + 1$

“For the construction of BCH codes, the roots of their generator polynomials specified by their zeros are used. A BCH code of $d_{min} \geq 2t_d + 1$ is a cyclic code with a generator polynomial $g(x)$ that has $2t_d$ consecutive roots α^b, α^{b+1},α^{b+2t_d-1} and is able to correct t or fewer bit errors for a block of $n = 2^m - 1$. The generator polynomial of that code is specified by its roots from the Galois field $GF(2^m)$. A binary BCH (n, k, d_{min}) code has a generator polynomial $g(x) = LCM\{\phi_b(x), \phi_{b+1}(x), ..., \phi_{b+2t_d-1}(x)\}$, length $n = LCM\{n_b, n_{b+1}, ..., n_{b+2t_d-1}\}$ and dimension $k = n - deg[g(x)]$. The *BCH bound* is the lower bound of a BCH code’s minimum distance and can be used to estimate the error-correcting capabilities of cyclic codes. The elements α^b, α^{b+1},α^{b+2t_d-1} are roots of the generator polynomial $g(x)$ and every codeword v in the BCH code is associated with a polynomial $v(x)$ which is a multiple of $g(x)$. If the generator polynomial $g(x)$ of a cyclic (n, k, d) code has l consecutive roots: α^b, α^{b+1},α^{b+l-1}, then $d \geq 2l + 1$. In the encoding phase, the input to the BCH code will be transformed block-wise into the corresponding codewords. In the decoding phase, these codewords are decoded and at max t bit errors in each codeword can be corrected. The $GF(2^m)$ arithmetic is used to find the error positions in the codewords during the decoding. This is done by solving a set of equations, which are obtained from the error polynomial $e(x)$ and the zeros of the code α^j, for $b \geq j \geq b + 2t_d - 1$.” [37]. The BCH decoding uses a syndrome systematic decoding technique, which allows for an efficient hardware implementation. Following steps summarize the decoding:

- The syndromes are computed through the evaluation of the received polynomial at the zeros of the code.
- The coefficients of the error-locator polynomial $\sigma(x)$ must be solved for.
- The inverses of the roots of $\sigma(x)$ have to be calculated, which inform about the error locations α^{j1},...,α^{jv}.

- The decoded error locations can then be used to inverse the bits at the corresponding position, and thereby correcting the errors.

This concludes our discussion of the BCH code, for more details and the inner workings of the BCH encoder and decoder please refer to [37].

2.2.2 Golay Codes

Golay code is named after its discoverer swiss mathematician Marcel J.E Golay, it falls in the class of linear error-correcting code used in digital transmissions. It is the only known code that can correct three or less random errors in a block of 23 elements. A successor of Golay code is an extended Golay code [24, 12, 8] which encodes 12 bits of data in 24-bit length codewords with error-correcting capability of 3-bits and can detect errors up to 7 bits. In our toolkit, we have implemented the Golay code G23 i.e. a [23, 12, 7] code, so we will be only concerned with the definition and constructs of the latter. A lot of work in this subsection is influenced and taken from [38].

Binary Golay codes are constructed using the vectors u = 1 1 0 1 1 1 0 0 0 1 0 and J = 1 1 1 1 1 1 1 1 1 1 and P = (p_{ij}), which is a 11x11 full-cycle permutation matrix, i.e. all the elements of the matrix are zero except:

$$p_{2,1} = p_{3,2} = p_{i+1,i} = ... = p_{n,n-1} = p_{1,n} = 1$$

We define matrix B as follows:

$$\text{B} = \begin{bmatrix} u & 1 \\ uP & 1 \\ uP^2 & 1 \\ uP^3 & 1 \\ uP^4 & 1 \\ uP^5 & 1 \\ uP^6 & 1 \\ uP^7 & 1 \\ uP^8 & 1 \\ uP^9 & 1 \\ uP^{10} & 1 \\ & \\ J & 0 \end{bmatrix} = \begin{bmatrix} 1\,1\,0\,1\,1\,1\,0\,0\,0\,1\,0 & 1 \\ 1\,0\,1\,1\,1\,0\,0\,0\,1\,0\,1 & 1 \\ 0\,1\,1\,1\,0\,0\,0\,1\,0\,1\,1 & 1 \\ 1\,1\,1\,0\,0\,0\,1\,0\,1\,1\,0 & 1 \\ 1\,1\,0\,0\,0\,1\,0\,1\,1\,0\,1 & 1 \\ 1\,0\,0\,0\,1\,0\,1\,1\,0\,1\,1 & 1 \\ 0\,0\,0\,1\,0\,1\,1\,0\,1\,1\,1 & 1 \\ 0\,0\,1\,0\,1\,1\,0\,1\,1\,1\,0 & 1 \\ 0\,1\,0\,1\,1\,0\,1\,1\,1\,0\,0 & 1 \\ 1\,0\,1\,1\,0\,1\,1\,1\,0\,0\,0 & 1 \\ 0\,1\,1\,0\,1\,1\,1\,0\,0\,0\,1 & 1 \\ & \\ J & 0 \end{bmatrix}$$

The generator matrix for the binary Golay code is defined as $\begin{bmatrix} I_{12} & | & \hat{B} \end{bmatrix}$ where I_{12} is a 12x12 identity matrix i.e. a matrix whose principal diagonal elements are ones and all other elements are zero. This generator matrix will be denoted as G_{23} in the remaining text.

Decoding: We summarize the steps for decoding the Golay code. We use parity check matrix for "extended Golay code" denoted by $H = G_{24}^t$ where G_{24} is generator matrix for extended Golay code [38], the error pattern is assigned a variable $u = v + w$, where w denotes the code word received and v, nearest codeword to w. In this context we denote $wt(x)$ as the weight of vector x, (or Hamming weight or the number of "ones" in x). Once we have found the value of u we assume the corrected received word is $v = w + u$ and we denote the received word w as $w = [w_1, w_2, \cdots, w_{12}, w_{13}, w_{14}, \cdots, w_{23}]$. The algorithm for decoding the binary Golay code is taken from [38] and described below:

1. If weight of w, i.e. $wt(x)$ is even then construct
   $\hat{w} = [w_1, w_2, \cdots, w_{12}, w_{13}, w_{14}, \cdots, w_{23}, 0]$
   else
   $\hat{w} = [w_1, w_2, \cdots, w_{12}, w_{13}, w_{14}, \cdots, w_{23}, 1]$

2. Compute the syndrome using extended Golay code generator matrix:
   $$s = mod(wG^t, 2)$$

3. If the Hamming weight of syndrome is less than or equal to *three*, then the error pattern is:
   $$u = [s,\ 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0]$$
   else, if the weight of s is greater than *three*, but for some i = 1, 2, ..., 12, the weight of $s + A_i$ is less than or equal to *two*, then the error pattern is $u = [s + A_i, e_i]$, where e_i is the i^{th} row of the 12x12 identity matrix and A is the matrix from which the extended Golay code generator matrix is generated as $G_{24} = [I_{12} | A]$ (refer to [38] for details about A and G_{24})

4. Compute the second syndrome $s_2 = sA$

5. If the weight of s_2 is less than or equal to *three*, then the error pattern is:
   $$u = [0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0,\ s_2]$$

6. If the weight of s_2 is greater than *three*, but for some i = 1, 2, ..., 12, the weight of $s_2 + A_i$ less than or equal to *two*, then the error pattern is considered as $u = s_2 + A_i$

7. If u is still not determined then a retransmission is requested.

8. After correcting the received codeword u, remove the last digit.

2.2.2.1 Linear repetition code

The following two subsections talk about Linear repetition code and Majority Voting which are crucial to the enrollment and reconstruction phase in the fuzzy extractor (discussed in section 2.2.3). The text is inspired from [20].

The notion behind linear repetition code is to produce redundancy in a binary bit string. The idea is rather simple; take a binary string and duplicate each bit of the input bit string by a factor r. The factor r can only assume odd values; this restriction is important for further processing that establishes a distinct evaluation of the original bit value in the recovery phase and prohibits the evaluation entering an undefined state.

The linear repetition code is used to augment the error-correction capability of the fuzzy extractor by producing redundant bits. To realize a PUF-based secure key storage, the linear repetition code can be integrated in the enrollment phase of the fuzzy extractor. Table 2.1 shows the working principle of the linear repetition code with factor $r = 7$.

Working principle of the linear repetition code	
Factor r	7
Input	10
Output	11111110000000

Table 2.1: Exemplary illustration of the working principle of the linear repetition code with factor 7.

2.2.2.2 Majority Voting

The majority vote is an inversely related concept to the above discussed linear repetition code. Similar to the linear repetition code it also uses a factor r whose value must be the same value as selected in the linear repetition step to ensure correct processing. The input to the majority vote is an input binary bit string of 1 and 0s with length r. The result is ascertained by the distribution of the bits in such a string; the majority of the bit values in a string (with odd numbers of elements) can either be 1 or 0. The string is divided into groups of factor r and then if more than 50% of the bits in the string are 1, the result of the majority vote is '1', else the result is '0'

The purpose of the majority vote is to enhance the error-correction capability of the fuzzy extractor (refer to section 2.2.3) by filtering bit errors from a redundant bit string. To accomplish this, the majority vote concept can be added into the reconstruction phase of the fuzzy extractor for the realization of a PUF-based secure key storage. Table 2.2 depicts the working principle behind the majority vote with factor $r = 7$.

Working principle of the majority vote	
Factor r	7
Input	11101110000001
Output	10

Table 2.2: Exemplary illustration of the working principle of the majority vote with factor 7.

2.2.3 Fuzzy extractor

We have already established that external factors like noise, voltage fluctuation, aging etc. instill errors in the PUF responses with some of the bits flipped between two responses from the same device, giving rise to intra-distance between them. *Fuzzy Extraction* is a well-known technique which aims to rectify these errors with the help of the error-correction codes. In this thesis, we have used binary BCH codes, binary linear repetition code, binary majority vote and binary Golay codes to realize the PUF-based secure key storage.

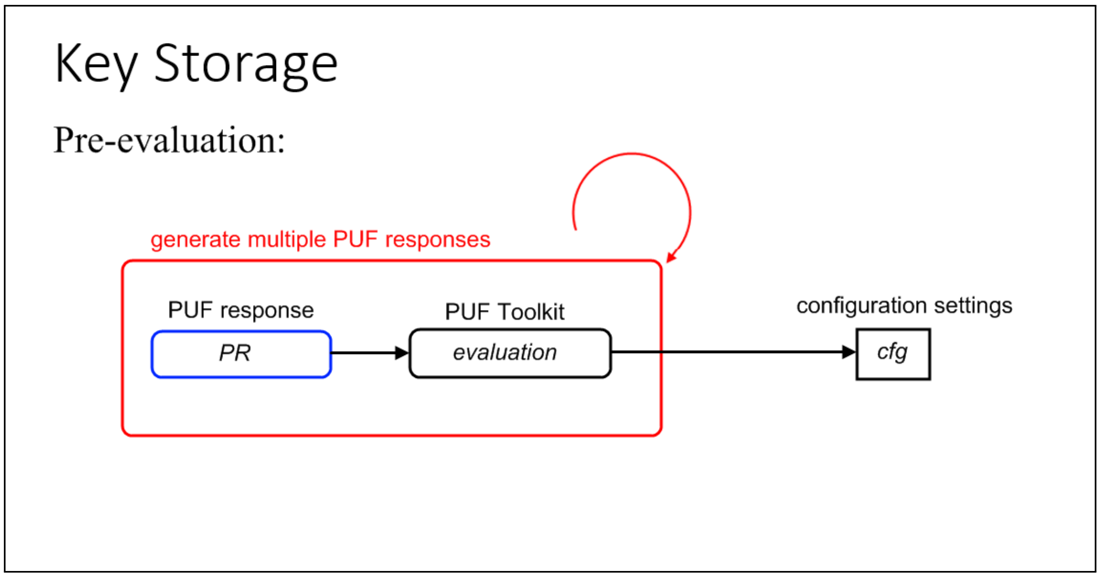


Figure 2.6: Conceptual design of the pre-evaluation phase for a PUF-based secure key storage.

The influence of noise and other environmental conditions can be pre-evaluated, so a selection of competent error-correction algorithms can be optimized and an efficient implementation of a fuzzy extractor to regulate the variations can be established. Most implementations of fuzzy extractors utilize a static approach which implies that the enlisted error-correction code has a static maximum error-correction capability. The Golay (23, 12, 7) code can correct up to three errors per 12-bits but has a limitation; if the filesize of the message to be encoded is larger than particular number of bytes then the consequent helper data generated may exceed the PUF response length and Golay decoding cannot proceed further. For this reason the BCH code is also used as a building block for the fuzzy extractor. There are three major steps to implement a secure key recovery using a Fuzzy extractor. In the first phase, known as Pre-evaluation we discard the unreliable PUF bits to reduce the burden on the error-correction scheme. The second phase is called generation/enrollment followed by the reproduction phase.

Pre-evaluation: The PUF response properties are evaluated in this phase. Some of the parameters that affect the PUF instance are static in nature and remain constant with different PUF response generations, so an approximation of the disturbance by these parameters can be done and compensated for. Pre-evaluating the response also gives insight into the instensity of noise disturbance that can occur, consequently a BCH mode can be selected to adapt and correct the errors introduced in the PUF response by that noise. The conceptual design for this phase is illustrated in Figure 2.6.

Generation: The second phase also called enrollment phase. It takes the secret key s, to be stored in the PUF-based secure key storage, and generates the matching helper data hd_s. The design for the generator is depicted in Figure 2.7 The secret s is provided as input to the BCH (n, k, d) encoder that is

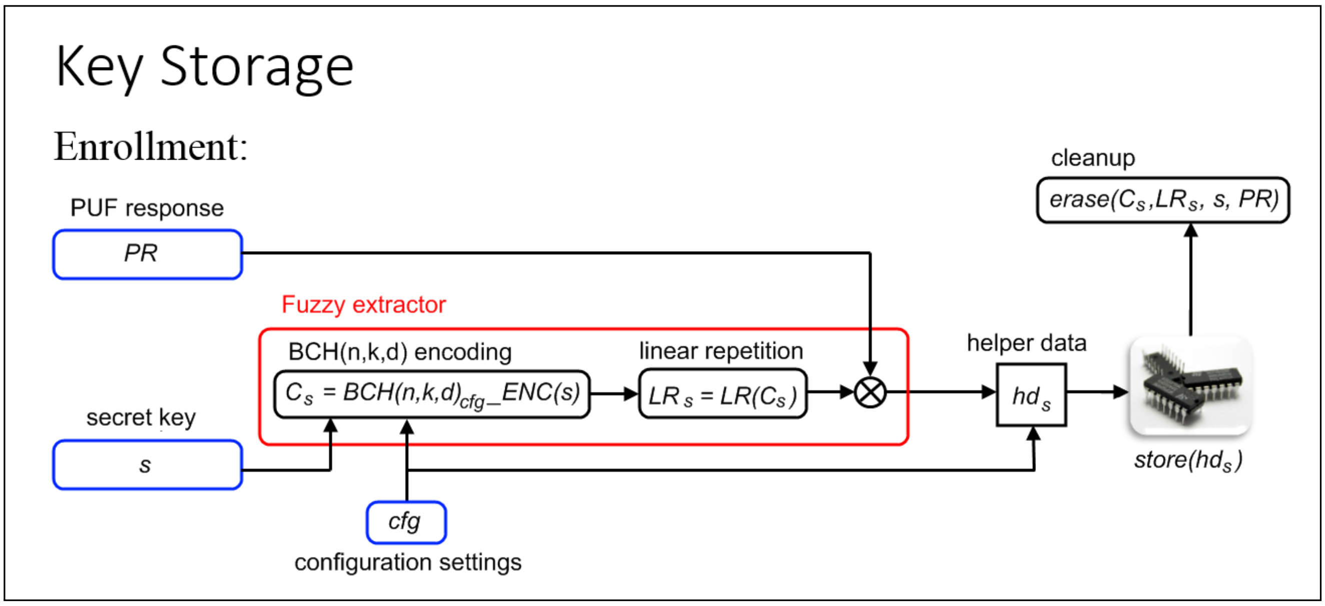


Figure 2.7: Conceptual design of the enrollment phase for a PUF-based secure key storage.

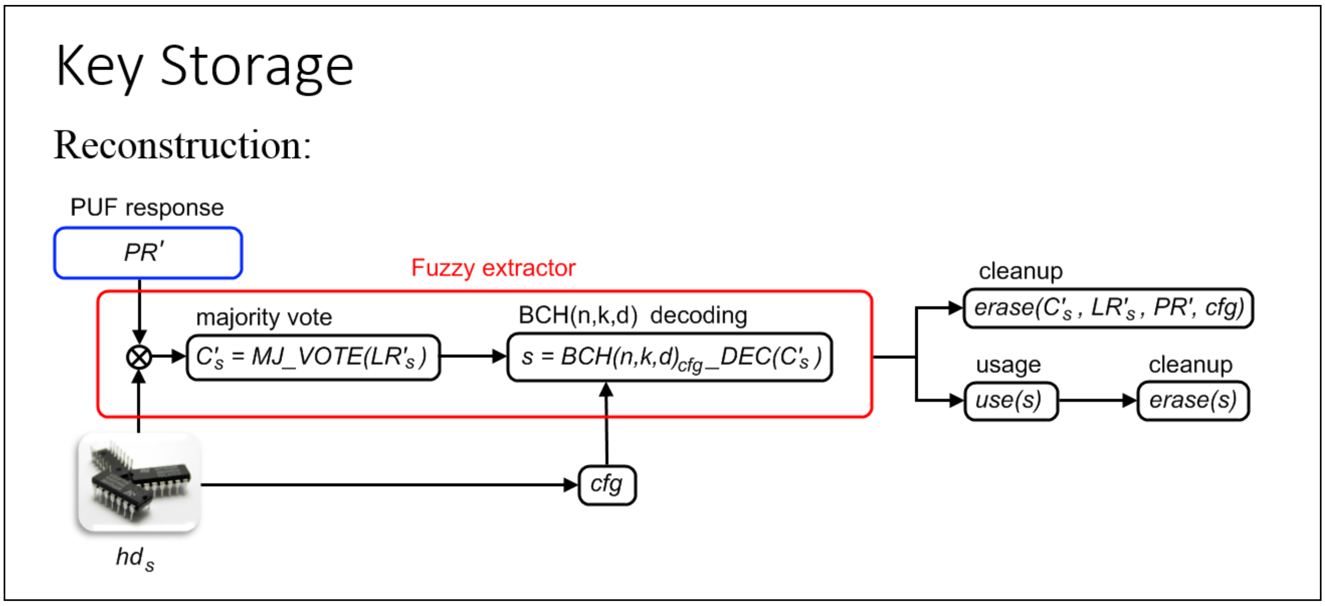


Figure 2.8: Conceptual design of the reconstruction phase for a PUF-based secure key storage.

configurable via the input line *cfg*, this results in the output of BCH encoded codewords C_s, since we are using systematic encoding in the BCH algorithm this means that the encoded secret s is contained in the codevectors. We denote this encoding for a certain configuration '*cfg*' by $BCH(n,k,d)_{cfg}\_ENC(s) = C_s$. The configuration is decided in the *pre-evaluation* phase, by setting the BCH code parameters m, t and n. Only the non-secret part (part of the output from BCH encoder that does not contains secret s) is taken from the codeword C_s and a linear repetition code LR is applied to it to generate LR_s, denoted in the text as $LR_s = LR(C_s)$. In the end, helper data hd_s for the secret key s is derived by XORing the PUF response PR and LR_s along with the configuration settings concatenated to the XORed result. This process is expressed in an equation as $hd_s = (PR \oplus LR_s) \cup cfg$. The helper data is stored in the non-volatile memory of the hardware device: $store(hd_s)$. The last step in this phase is “cleanup” where all the generated values $C_s, LR_s and PR$ are erased.

Reconstruction: After storing the key as helper data the same secret key s must be restored. First, the 'cfg' setting that was appended to the end of the helper data is stripped. A new PUF response from the same device PR' is XORed with stripped helper data hd_s to get LR_s': $LR_s' = PR' \oplus (hd_s \backslash cfg)$. Next, a Majority vote algorithm is applied to LR' from which the encoded codeword C_s' is recovered: $C_s' = MJ\_VOTE(LR_s')$. After this the reconstruction of the secret key s is done by feeding the obtained C_s' in the BCH (n,k,d) decoder: $C_s' = BCH(n,k,d)_{cfg}\_DEC(C_s')$. The last step is similar to the generation phase i.e. the cleanup of PR', cfg and LR_s'. The reproduced secret key s is also erased after usage. An illustration for the reconstruction design is depicted in Figure 2.8.

3 Implementation

This chapter explains the major tasks implemented in the thesis and new contributions made to the PUF Toolkit. The first part of the chapter briefly explains the current PUF toolkit implementation, without delving much into the details. This is followed by an explanation of the Hamming Distance Menu which was implemented as a new menu item combining the already implemented Intra- and Inter-Hamming Distance under one menu. The third part talks about the new modifications that were done to the toolkit in the form of BCH fuzzy extractor encoder and decoder integration, which were previously not a part of the main toolkit and were developed as separate executables with separate menu items. We then go on to explain the Golay code implementation for both the encoder and decoder and explain their integration in the toolkit as a distinct menu item. Then other modifications like the addition of *'offsets from beginning and end', error codes* and other intricate code development changes are presented together as one section. After this, a new metric called *Jaccard Index* is introduced and its implementation along with *Intra- and Inter-Jaccard Index* is explained in detail.

The final two sections deal with the Java Native Interface (JNI) and CogniCrypt opensource library, they first touch upon the basics of JNI wrapper and the functioning of JNI with shared libraries that are packaged as a JAR file and are made accessible to Java compiler as an external library. Finally, we describe the Clafer model of the CogniCrypt and how it assists users and Java developers without any previous knowledge about cryptography, to select a strong PUF-based secure key evaluation algorithm derived from the answers to the questions asked by the CogniCrypt. In the end of the last section, we also talk about the XSL model of the CogniCrypt that is built on the Clafer model to generate a boilerplate Java code to help the Java developer by presenting him/her with a sample use case of the Java code implementation showing a secure and flawless usage of PUF evaluation algorithms.

For the implementation, the ISO standard programming language C++ was chosen. This selection was made based on the efficient and general purpose features provided by C++. Apart from the object-oriented and generic programming features, C++ has a high abstraction level and the compilation and code development can be done on diverse systems. The implementations are dissociated from a specific hardware to support a wide range of systems and applications. This makes the toolkit easier to extend and conform to a particular hardware for next iterations. The JNI framework supports native calls and the wrapper is written in Java, for Clafer model we use Javascript and Json files along with the Clafer extension modeling language [39]. The code to generate a sample Java boilerplate code is written in XSL.

3.1 PUF Toolkit

The original version of the PUF toolkit was implemented and presented in the thesis [20]. The main aim of the original toolkit was to evaluate various PUF responses based on well-established metrics and thereby helping researchers and designers to gain useful insights into the properties and behavior of PUF responses. The toolkit implemented the following list of metrics:

- (Shannon) Entropy
- Hamming Weight
- Intra-Hamming Distance
- Inter-Hamming Distance
- Min-entropy
- Median and average

These metrics are thoroughly explained in [20], so we take the liberty of not explaining the details of each metric here again. It must also be noted that there are other metrics and definitions that use identical concepts and/or apply the metrics in a different way to generalize the correlation between different PUF instances and their responses. More exhaustive and comprehensive information related to these metrics and definitions for PUFs can be found in [40–43].

Apart from the above-mentioned metrics, the PUF-based secure key storage fuzzy extractor is already implemented, using BCH encoder and decoder as two separate executables. The design of the user interface and the data structures used in these two executables are similar to the ones used in PUF toolkit.

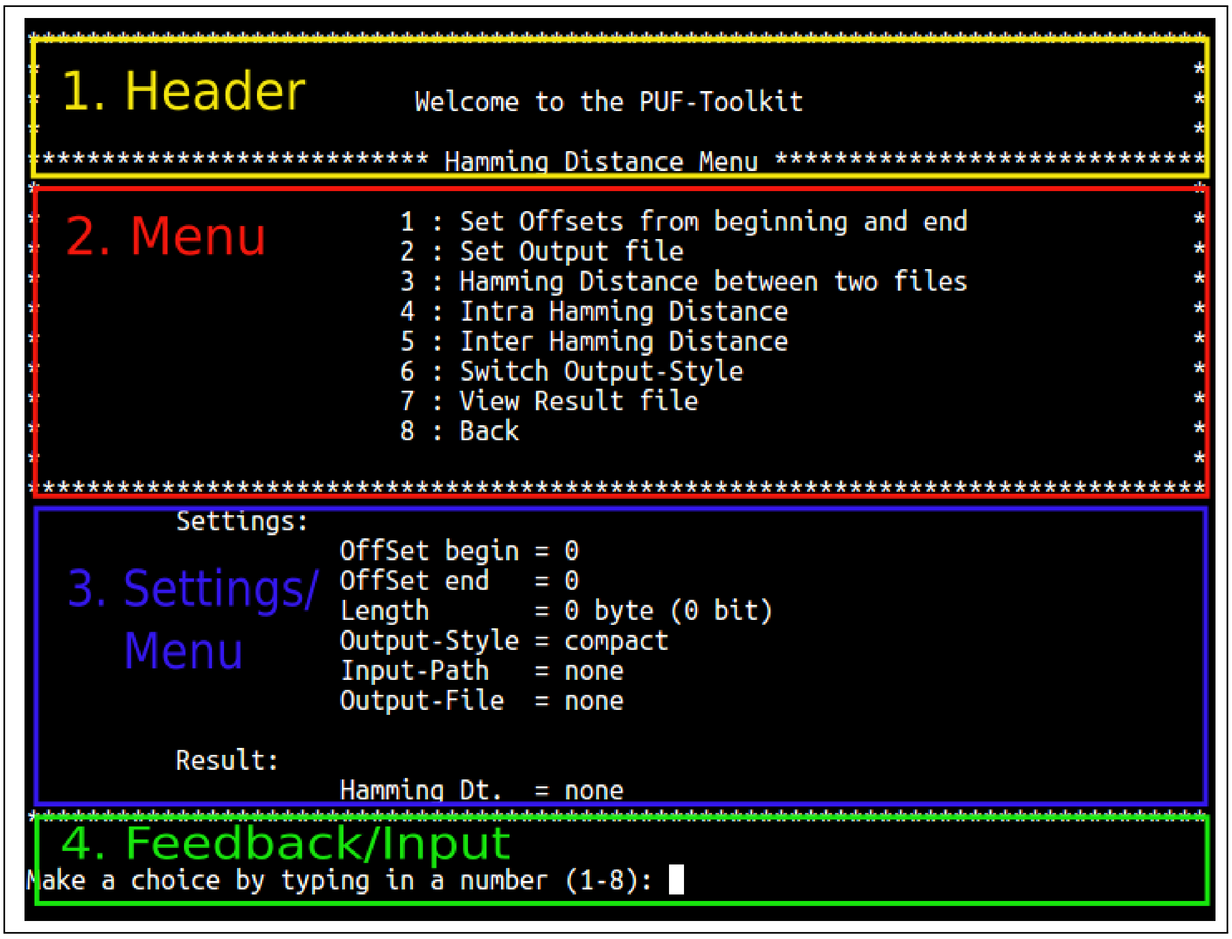
3.1.1 User Interface Design

The notion of the console user interface was inspired from Nielsen and Molich's nine user interface design guidelines [44]. These guidelines and their resulting effects are shown Table 3.1 below:

No.	Guideline	Effect
1	Simple and natural dialogue	Clear and logical dialogue structure
2	Speak the user's language	Clear instructions
3	Minimize the user's memory load	Clean design, brief help and guide texts
4	Be consistent	Consistent design in all menus
5	Provide feedback	Provide feedback and status
6	Provide clearly marked exits	Provide "back" and "exit" in each menu
7	Provide shortcuts	Inputs by abbreviations
8	Good error messages	Provide useful error messages
9	Prevent errors	Handle wrong inputs

Table 3.1: The nine design guidelines according to [44] and their effect on the UI design.

The design of the menu and the sub-menus is consistent and recurses itself to make the user interface more intuitive and friendly. Each menu option is written in clear and easy to understand instructions



```
                       Welcome to the PUF-Toolkit
************************* Hamming Distance Menu *****************************
                       1 : Set Offsets from beginning and end
                       2 : Set Output file
                       3 : Hamming Distance between two files
                       4 : Intra Hamming Distance
                       5 : Inter Hamming Distance
                       6 : Switch Output-Style
                       7 : View Result file
                       8 : Back
*****************************************************************************
         Settings:
                    OffSet begin = 0
                    OffSet end   = 0
                    Length       = 0 byte (0 bit)
                    Output-Style = compact
                    Input-Path   = none
                    Output-File  = none

         Result:
                    Hamming Dt.  = none
*****************************************************************************
Make a choice by typing in a number (1-8):
```

Figure 3.1: Conceptual design of the PUF Toolkit user interface.

and the menu items are precisely structured to help the user to navigate the toolkit with ease. The organization of the menu follows a hierarchical design and a "back" function is provided for the user to go to the parent menu. A conceptual depiction of the User Interface is shown in Figure 3.1. The Graphical User Interface (GUI) is subdivided into four parts, the color markers are related to the illustration in 3.1.

- The first part, marked in yellow, shows the *header* part of the menu. It gives the user general information, like the title of the current menu/sub-menu or brief instructions.

- The second part of the GUI shows the options and functions that the user can choose from the menu/sub-menu; the number in front of the option must be supplied by the user. The conceptual illustration of the part is marked in red and is named *Menu*.

- Below the Menu part there is *settings and results* that are marked in blue. It shows the current configuration and provides the user with essential information that is vital for the actual computation of the main function. It also shows the output after the computation is performed in the *results* part of this section.

- The fourth part of the GUI is for *feedback and inputs* and is marked in green. It presents the user with the actual processing status, or if an error occurred, the type of the error. Additionally, it

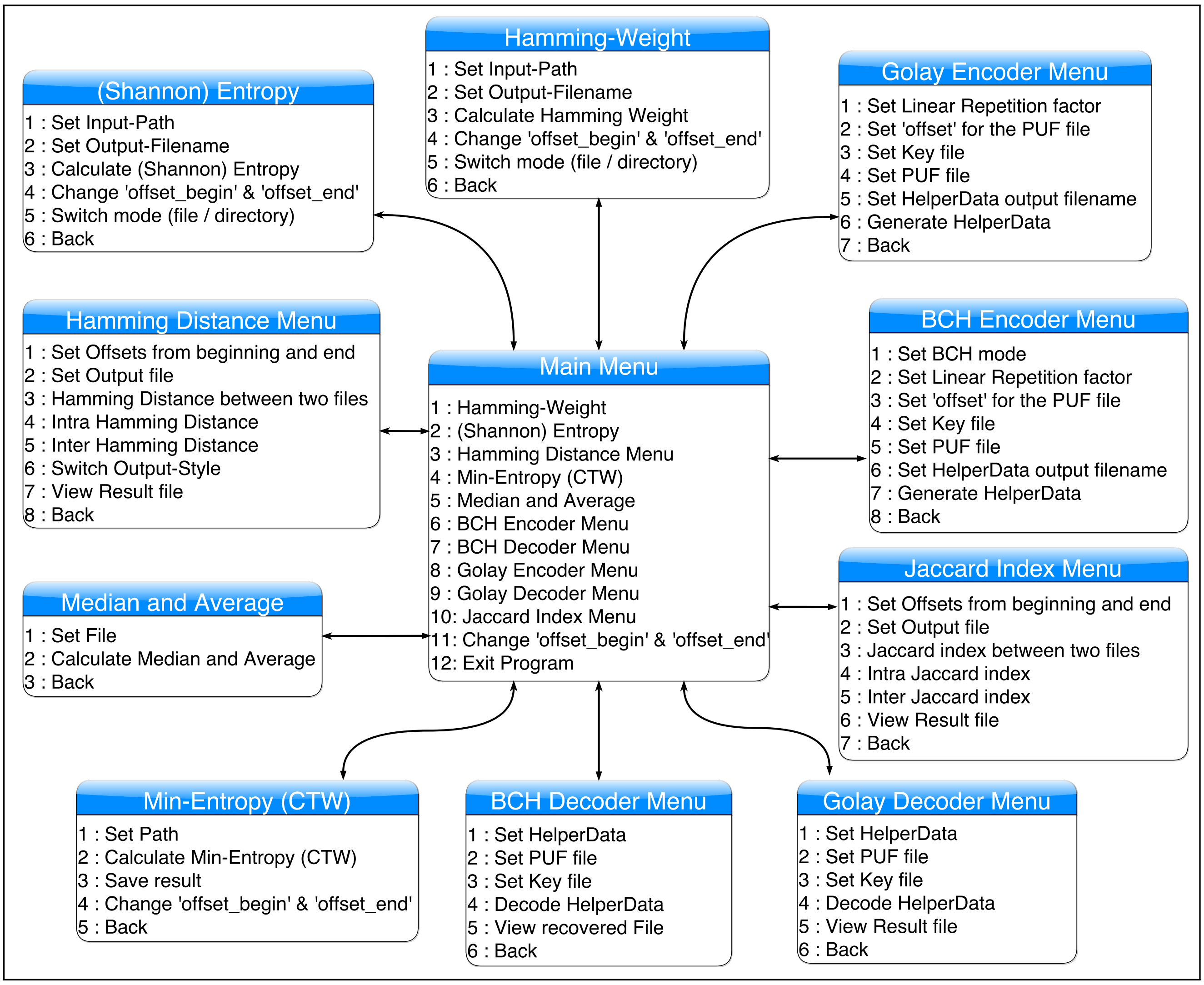


Figure 3.2: Hierarchical Structure of the PUF Toolkit menu and available options

contains a user interactive input cursor; where a number must be typed in by the user in order to select the options/functions of the toolkit.

The design of the GUI is kept simple without the use of extended graphic components to keep the toolkit compatible with other operating systems like Linux. The UI shows only relevant information depending on the current state of the program and portrays a simple but an aesthetically clean design. Error correction and incorrect user input is efficiently handled, all possible inputs are rigorously checked and depending on the false input, meaningful error information is displayed to the user, that can be used to recover from the erroneous state. Also for all mandatory inputs, a brief guide/help text with examples is shown, the current settings are saved and kept in each sub-menu (wherever applicable) to avoid unnecessary redundant input by the user.

By rigorously complying to the standardized design guidelines for the toolkit, we have achieved to implement an effective and intuitive console user interface. This in turn decisively supports developers and researchers in the evaluation of PUFs responses. [20]

3.2 Hamming Distance Menu

The Hamming distance functionality was added to and integrated as part of this Master thesis. Hamming Distance between two “bit strings” of the same length is defined as the number of the bits that differ amongst the two binary bit strings. E.g., if we have two strings $A = 1100$ and $B = 1010$ then the Hamming Distance between them would be “*two*”, since the bits at second and third position (starting from the left) differ between these two strings. The implementation of this functionality is straightforward; by using the in-built *bitwise XOR* operation of the ISO C++ standard and then counting the number of ones in the result string, which is achieved by calling the PUF Toolkit’s Hamming weight function on the resultant string.

As depicted in Figure 3.1 there are other options like *Intra-Hamming Distance* and *Inter-Hamming Distance* in the Hamming Distance Menu, that are organized later as subsections where we explain about the different modes and other options and possible input values the user can give.

The first menu item is related to the ***offset functionality*** that is a mandatory option which must be set by the user before proceeding on to the other options of the toolkit. The PUF toolkit is designed for the evaluation of PUF responses that are in binary format where not all the data in a PUF response binary file is useful for evaluation. So a functionality is provided to the user through which he/she can manually select how much data(in bytes) he/she desires to skip from the beginning and from the end of the file. For example, the first 400 bytes of the PUF response of a Texas Instruments Stellaris LM4F120XL Launchpad Evaluation Kit are used during the booting process and are not random, so they are irrelevant in evaluating the PUF responses and must be skipped. This kind of device specific behavior requires suitable handling to ensure valid results.

struct *Item*	Purpose
offset_begin	Defines the starting point in a binary file (bytes to skip from beginning)
offset_end	Defines the ending point in a binary file (bytes to skip from end)
input_length	Defines the number of bytes to use
input_file_name	Defines the name of the first input (key) file (name and path)
input_PUF_name	Defines the name of the second input (PUF) file (name and path)
output_file_name	Defines the name of the output result file (name and path)
zeros	Stores the occurrences of 0s in a defined file
ones	Stores the occurrences of 1s in a defined file
frd	Stores the fractional distance in a defined file
result	Stores the result and feedback regarding the calculations
HD_error_pos	Additional error information

Table 3.2: Definition of the elements of the data structure *Item* and their purpose.

Consequently, the *offset* determines the selection range of the bytes from the PUF response for calculation and also establishes the actual *length* of the response. E.g., if a PUF response is 32784 bytes long and we skip 400 from the beginning and 200 from the end, the actual length becomes $32784 - 600 = 32184$ bytes and the processing starts from the 401^{st} byte till 32184^{th} byte.

The result of the Hamming Distance between two PUF responses can be optionally stored in an output file that can be set using the option *two* (Set Output file). This is not mandatory if we are evaluating only two files but in cases of Inter/Intra-Hamming Distance this option becomes mandatory. Another functionality added to the Hamming Distance and other Functions in the toolkit as well is that of the Fractional Distance.

§**Definition**: We define *Fractional Hamming Distance* as the Hamming Distance between two PUF responses divided by their length. For example, if Hamming Distance between two PUF responses each of length 31784 bytes is 10, then their Fractional Hamming Distance would be $10/31784 = 0.000314624$ (NOTE: the toolkit displays the fractional Hamming distance rounded off to 9 digits after decimal).

The Hamming Distance can be calculated using option *three* of the Hamming Distance Menu, where the user must input the names and paths of two PUF responses; if the path and filenames are correct, then the result will be displayed in the "feedback" part of the GUI. If the output filename has been set before, then the result is also stored in that file and can be viewed using the option *seven* (View Result file).

In order to understand the code flow and working of the Hamming distance function, it is vital to look at the structure *Item* and its data members. This structure is used by other menus in the toolkit to share the current settings and global configurations like Offsets, filenames, paths etc. and thereby reducing the user's effort and memory load. Table 3.2 only highlights the data members and their purpose that are relevant to the Hamming Distance Menu.

```
1      //bitwise XOR file1data and file2data
2      //to get the positions where bits differ
3      for (i = 0; i < dsize; i++) {
4          data[i] = f1data[i] ^ f2data[i];
5      }
6      //calculate hamming distance
7      item->ones = hammingwt(data, dsize);
8      item->frd = (float) item->ones / dsize;
```

Listing 3.1: Hamming Distance calculation using Hamming weight and bitwise XOR operator

The four lines of code presented in Listing 3.1 form the basis for calculation of the Hamming distance. First the data is read from the two input PUF files into two arrays of size *dsize* where dsize is equal to the length of the files after applying the Offsets, then as depicted in lines 3-5 of Listing 3.1 each byte of the data from the two PUF responses is bitwise XORed and the result is stored in a third array of same length. The resultant data array contains *ones* in the position where the bits differ between the two PUF responses. We then just need to count the number of ones in the XORed result. This is done by calling the function *hammingwt* of the toolkit that computes the Hamming Weight of the XORed result.

```
1   int hammingwt(char *data, int size)
2   {
3       int i;
4       int wt = 0;
5       for (i = 0; i < size; i++) {
6           (data[i] & 0x80 ? wt++ : wt); //8th bit position
7           (data[i] & 0x40 ? wt++ : wt); //7th bit position
8           (data[i] & 0x20 ? wt++ : wt); //6th bit position
9           (data[i] & 0x10 ? wt++ : wt); //5th bit position
10          (data[i] & 0x08 ? wt++ : wt); //4th bit position
11          (data[i] & 0x04 ? wt++ : wt); //3th bit position
12          (data[i] & 0x02 ? wt++ : wt); //2nd bit position
13          (data[i] & 0x01 ? wt++ : wt); //1st bit position
14      }
15      return wt;
16  }
```

Listing 3.2: Hamming Weight calculation using bitwise AND operator

Hamming Weight is one of the essential functions of the toolkit and it is also referred to by other functions, so it's important that we list its code here. Listing 3.2 shows the Hamming Weight function as implemented in the toolkit, it takes the data array as a character pointer and its size as arguments. The for loop in lines 5-14 takes each byte of the data and then bitwise ANDs this byte with each bit position (total 8 bit-positions) to count the total occurrences of "ones" in the byte. This process is iterated for the entire length "size" of the array, each time incrementing the counter "wt" whenever a "one" is encountered. After the computation is done, the Hamming weight is returned to the caller function.

This section explains the Intra-Hamming Distance feature which is a comparison of the PUF responses from the same device. As discussed in section 2.1.3.2 we already observed that the PUF responses from the same device are not identical due to various factors like voltage fluctuation, temperature variation, and other aging effects, which give rise to distance between responses of the same PUF. We refer to this distance as “intra” because it concerns responses of the same device. Since the responses from a single PUF instance are kept in a single directory, we need only to give the “input path” for processing Intra-Hamming Distance; the output file option is mandatory to process Intra-Hamming Distance because the result must be stored for all the responses so that it can be viewed later. Along with the Hamming Distance between PUF responses, the fractional distance is also saved for better assessment of the PUF instance.

```
******************************************************************************

                             Intra-Hamming Distance

                          Used Settings:

                          Device folder: 'data/check'
                          offset_begin:         400
                          offset_end  :         0
                          Length:         32385 byte (259080 bit)

******************************************************************************


------------------------------------------------------------------------------
                                                                      Filename
                                                               data/check/PUF4
------------------------------------------------------------------------------
data/check/PUF2                                 13403       0.41386
data/check/PUF1                                 13559       0.41868
data/check/PUF3                                 0           0.00000
------------------------------------------------------------------------------



------------------------------------------------------------------------------
                                                                      Filename
                                                               data/check/PUF2
------------------------------------------------------------------------------
data/check/PUF1                                 13950       0.43075
data/check/PUF3                                 13403       0.41386
------------------------------------------------------------------------------



------------------------------------------------------------------------------
                                                                      Filename
                                                               data/check/PUF1
------------------------------------------------------------------------------
data/check/PUF3                                 13559       0.41868
------------------------------------------------------------------------------
```

Figure 3.3: Visualization of the *compact* output style format for Intra-Hamming Distance.

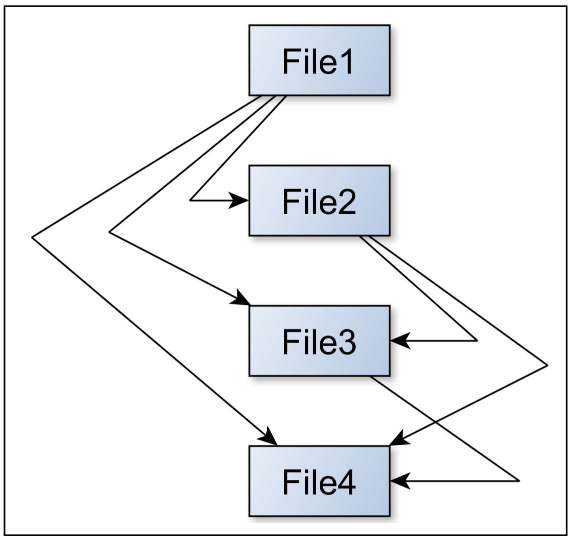


Figure 3.4: Exemplary illustration of the working principle of the *compact* mode, for the Intra-Hamming Distance.

A crucial attribute for the Intra-Hamming Distance result is the “Switch output-style” which can be used to change the save format of the output file, there are two possibilities for Intra-Hamming Distance: *compact* and *minimal*. The compact mode takes each file in the directory and recursively compares it with all the other files, then the second file is chosen and compared with the all the others except the first one and so on. Figure 3.4 illustrates this comparison. The output style “*compact*” is the default format style and should be used as an output format for a regular text file. Due to the symmetrical behavior of the comparison (A to B = B to A), we need not exhaust all 1-to-1 possible combinations. So in compact mode first entry from the directory is compared to all the other entries till the last entry, then the second entry is compared to the third entry till the last entry, then the third entry compared with fourth entry and so on till we are at the last file which need not be compared to any of the above entries since we have already exhausted the symmetric comparisons.

A sample output file with compact style format is shown in Figure 3.3 that evaluates the PUF instance from a single directory using Intra-Hamming Distance metric, it also appends the fractional distance information to the file. It consists of a header with general information that shows the configuration (like offsets, device folder etc.) that was used for evaluation, and three tables. In the upper right corner of each table the utilized input file is shown and on the left side of the table the files, that are compared to the selected input file, are displayed. The Hamming Distance and Fractional Hamming Distance, separated by a tab length are printed in the center (right) of each table. Another output style known as the *minimal* output style strips off all the path and filename information and only saves the Hamming Distance values separated by spaces, this type of file format can be used by another metric of the toolkit called Median and Average calculation (section 4.1 of [20]) which accepts only files containing plain numbers separated with space. The *View* option can be used to look at the output/result file.

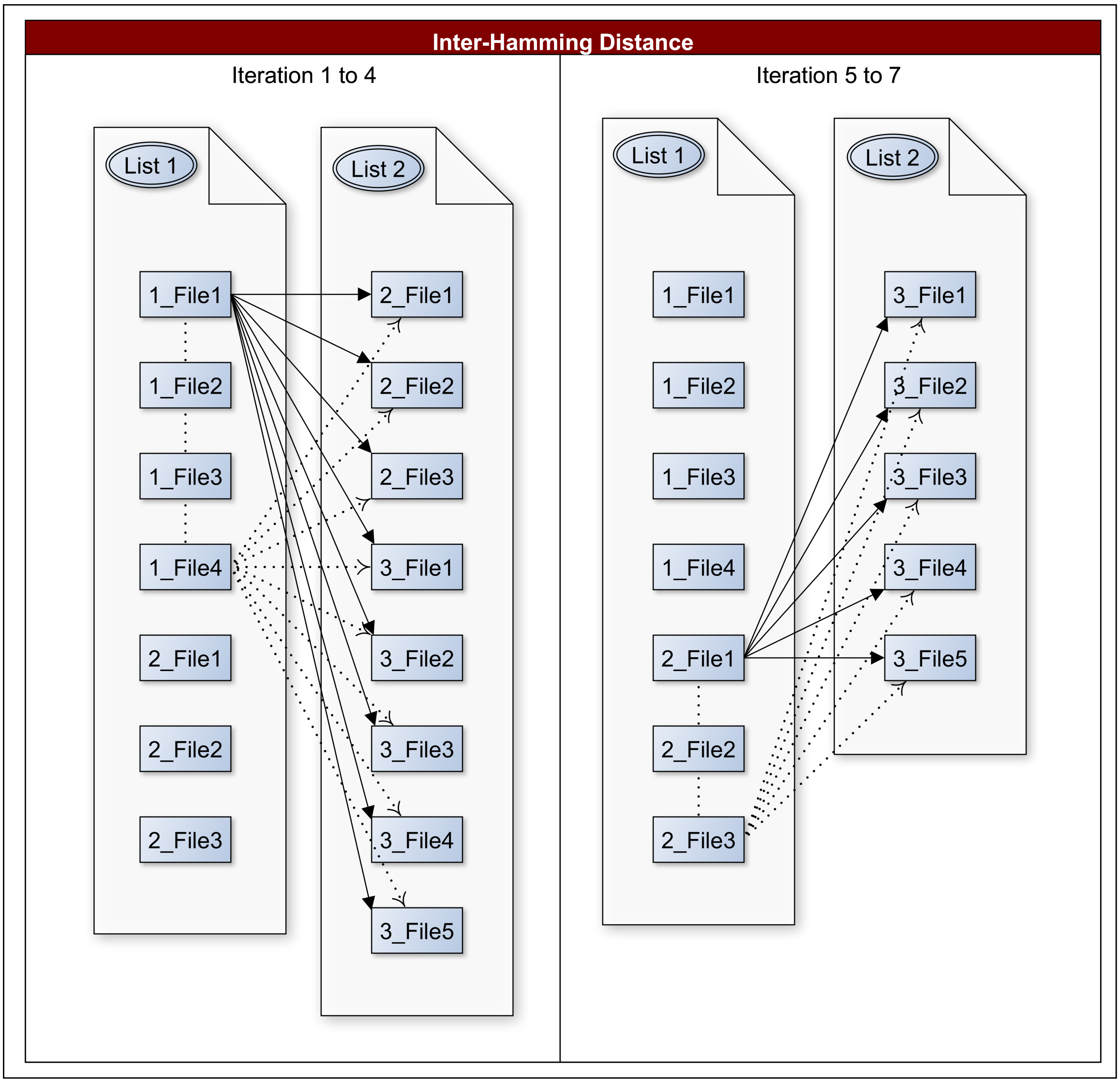


Figure 3.5: Exemplary illustration of the working principle of the *compact* mode, for the Inter-Hamming Distance.

Contrary to the comparisons between PUF responses from the same device, Inter-Hamming distance evaluates responses from different PUF instances. The co-relation is again based on the same metric Hamming Distance. In this case, the toolkit can simultaneously compare between 2 to 99 PUF instances. All the PUF responses from a specific instance are arranged in a single directory so that means the toolkit can take a variable number of input paths between 2 and 99 to evaluate different PUF devices. The output style for Inter-Hamming Distance in addition to the discussed compact and minimal stye has a *detailed* style as an alternative. The compact and minimal style formats are same as Intra-Hamming Distance, but the detailed style differs from the compact style such that it does not implicitly ignores the duplicate symmetric comparisons and therefore maps out all the permutations and combinations between files of different folders. It should be noted that Inter-Hamming Distance metric does not compare PUF responses in the same directory. So in detailed mode, all the responses from the first directory are compared to all the PUF responses from the remaining 3 to 99 directories, then all files in the second folder are compared to all the files in all the other folders except its own, and so on. The output of the Inter-Hamming Distance in compact mode is similar to that of Intra-Hamming Distance as previously shown in Figure 3.3. For the compact mode we need to ignore the duplicate comparisons and therefore the following approach is taken:

- store all the filenames and their corresponding paths from all but last folder in list1.
- store all the filenames and their corresponding paths from all but the first folder in list2.
- iterate over list1 (iteration 1-4) and compare the list1 objects with all entries of list2.
- remove the second folder objects from list2.
- iterate over list1 (iteration 5-7) and compare the list1 objects with all files of list2.

This process is summarized in Figure 3.5 for three folder input and can be generalized for more input folders. We have emphasized on the inner workings of the output style modes because the exact same techniques are used for Jaccard's index metric that will be discussed later in this chapter.

Menu Modifications: In contrast to the original toolkit where Intra- and Inter-Hamming Distance were separated in their own sub-menus, the new changes merged both of them under Hamming Distance Menu. Seeing as it is relevant to combine them if they use the same underlying metric. On the other hand, the Hamming Weight menu item was untouched since it takes only one PUF response file as an input and gives out its Hamming weight and there is no evaluation between two or more files.

3.3 Fuzzy extractor

As discussed in section 2.2.3 the PUF-based secure key storage approach uses Fuzzy Extractor technique that is divided into three phases *(i)Pre-evaluation*, *(ii) Enrollment* and *(iii) Reconstruction*. The second and third phase follow a separate implementation of the respectively dedicated functionalities, according to [14]. The enrollment phase employs PUF BCH encoder which will be discussed first followed by the Reconstruction phase, that is realized by the PUF BCH Decoder implementation. We will not go into the details of the actual implementation of the encoder and decoder since they are already well documented in [20], but rather explain the changes made to integrate these functionalities in the PUF toolkit.

The concept of fuzzy extractor in combination with BCH coding and linear repetition code is depicted in Figures 3.6 and 3.7.

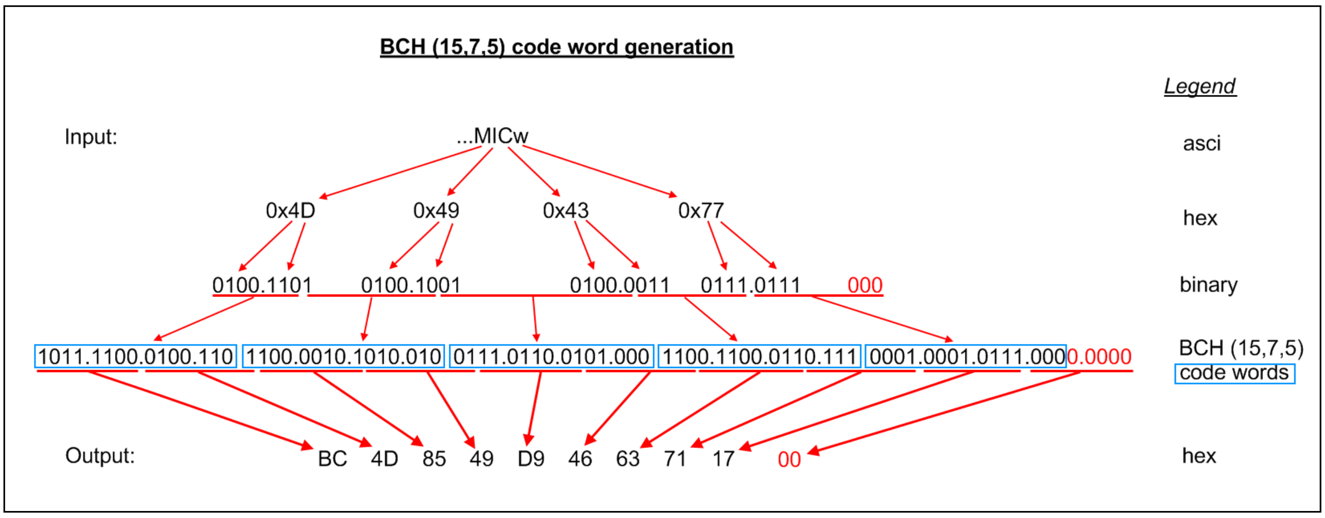


Figure 3.6: Exemplary code word generation with a BCH (15,7,5) code, based on [14].

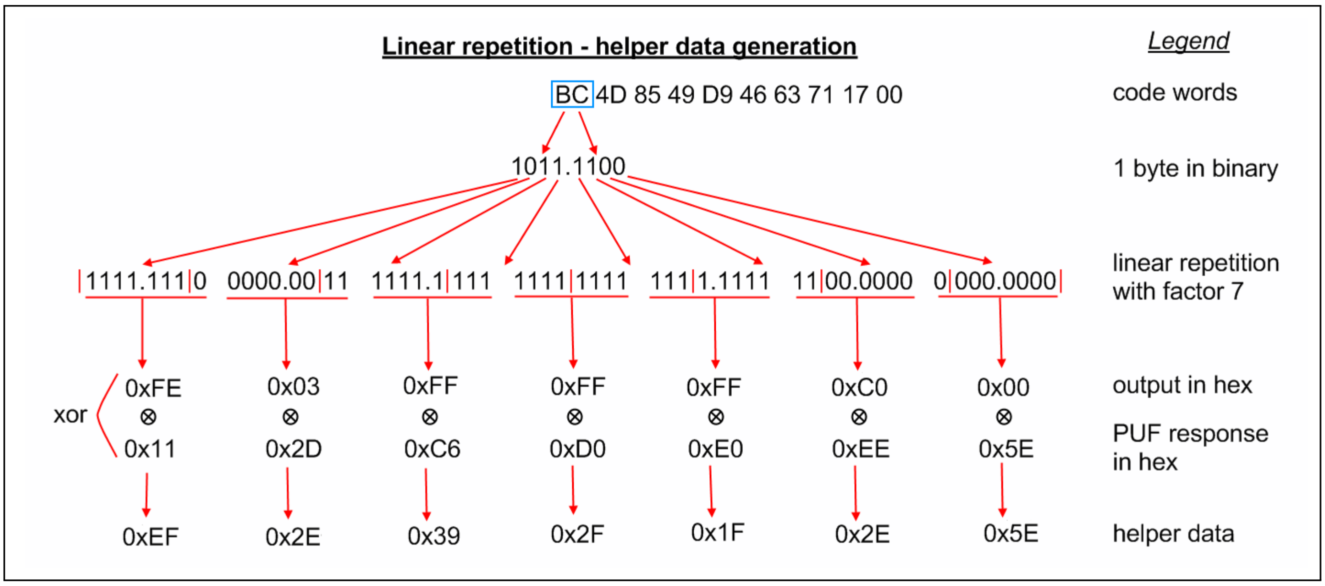
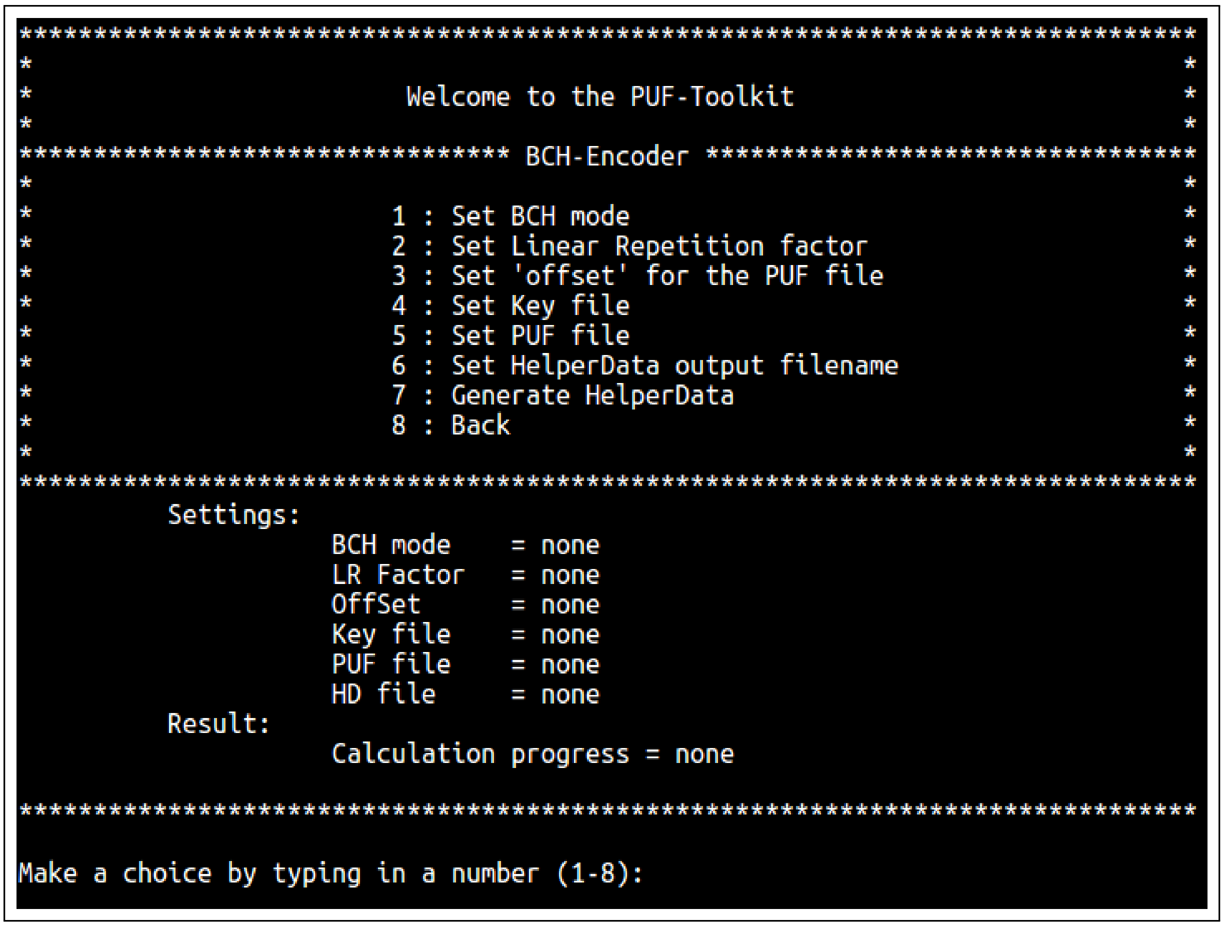


Figure 3.7: Exemplary helper data generation with a linear repetition code and factor 7, based on [14].

3.3.1 PUF BCH Encoder

This section first briefly mentions the implementation algorithm used in the BCH encoding of the data, after that the user interface of the BCH decoder is illustrated, followed by an Entity Relation Diagram that shows the major functions of the encoder. The changes are presented as a subsection named integration changes and the section is concluded by showing the steps taken to verify the integrated BCH encoder.

The BCH encoding implementation is taken from *Morelos-Zaragoza's Encoder/Decoder* for binary BCH codes in C [45]. The algorithm uses systematic encoding to encode the message, hence the encoded codewords contain the initial message in it, which is stripped from the codeword to get the helper data. The message is encoded using the BCH(n,k,d) code (refer section 2.2.1) to generate the codewords which are then combined in groups of *length k* and extra 0s are added to complete the last set as shown in Figure 3.6. In the second part of the enrollment of the fuzzy extractor, the LR code is applied to the generated codeword, based on the user input the bits are repeated either 7 or 15 times, Figure 3.7 shows this LR process with factor 7. It was noted that there were some limitations to the original algorithm and the encoding worked only for small values of m, after code review and research the error states were removed and/or documented.



```
******************************************************************************
*                                                                            *
*                         Welcome to the PUF-Toolkit                         *
*                                                                            *
******************************** BCH-Encoder *********************************
*                                                                            *
*                        1 : Set BCH mode                                    *
*                        2 : Set Linear Repetition factor                    *
*                        3 : Set 'offset' for the PUF file                   *
*                        4 : Set Key file                                    *
*                        5 : Set PUF file                                    *
*                        6 : Set HelperData output filename                  *
*                        7 : Generate HelperData                             *
*                        8 : Back                                            *
*                                                                            *
******************************************************************************
          Settings:
                     BCH mode     = none
                     LR Factor    = none
                     OffSet       = none
                     Key file     = none
                     PUF file     = none
                     HD file      = none
          Result:
                     Calculation progress = none

******************************************************************************

Make a choice by typing in a number (1-8):
```

Figure 3.8: Console user interface of the *PUF BCH Encoder*.

The relation between the length of the secret key, number of BCH iterations, linear repetition factor and BCH parameters n and k is stated in the following two Equations:

- For an input data of length N (bits) and a BCH (n,k,d) code, a '*len*' number of BCH encoding iterations have to be performed to process the complete input file [20]:

$$len = \left\lceil \frac{N}{k} \right\rceil \tag{3.1}$$

- For a BCH (n,k,d) code that requires '*len*' BCH encoding iterations to process the complete input file, the result of the linear repetition code has a length of '*LengthAfterLR*' with respect to the chosen linear repetition factor '*LRfactor*' (7 or 15) [20]:

$$LengthAfterLR = \left\lceil \left(\frac{\left(\frac{len}{k} * n \right)}{8} \right) \right\rceil * LRfactor \tag{3.2}$$

The User Interface for the BCH Encoder sub-menu is shown in Figure 3.8. The first option is to set the BCH mode in which the user is required to input the code length n i.e. derived from m such that $(2^{m-1} - 1) < n <= (2^m - 1)$, after that it is also mandatory to enter the desired correction capability t of the BCH (n,k,d) code that in turn will determine the parameter d of the code from the formula $t = \lfloor (d-1)/2 \rfloor$. These values must be consistent with the equations 3.1 and 3.2 or else the decoding of the secret key will not be successful. The second option chooses the LR factor as 7 or 15, the third option sets the offsets for the PUF response that will also determine the actual length of the PUF response that is used to generate the helper data. After this the user must supply the key filename he/she wants to securely store and the name of the PUF response file, finally the output filename for helper data must be provided to be saved on permanent storage and to be used in the later reconstruction phase of the fuzzy extraction. These discussed inputs are all mandatory in the BCH menu and once they are initialized, only then we can use option seven of the BCH encoder menu to process and generate the helper data or else the toolkit throws appropriate error messages to the user about missing parameters.

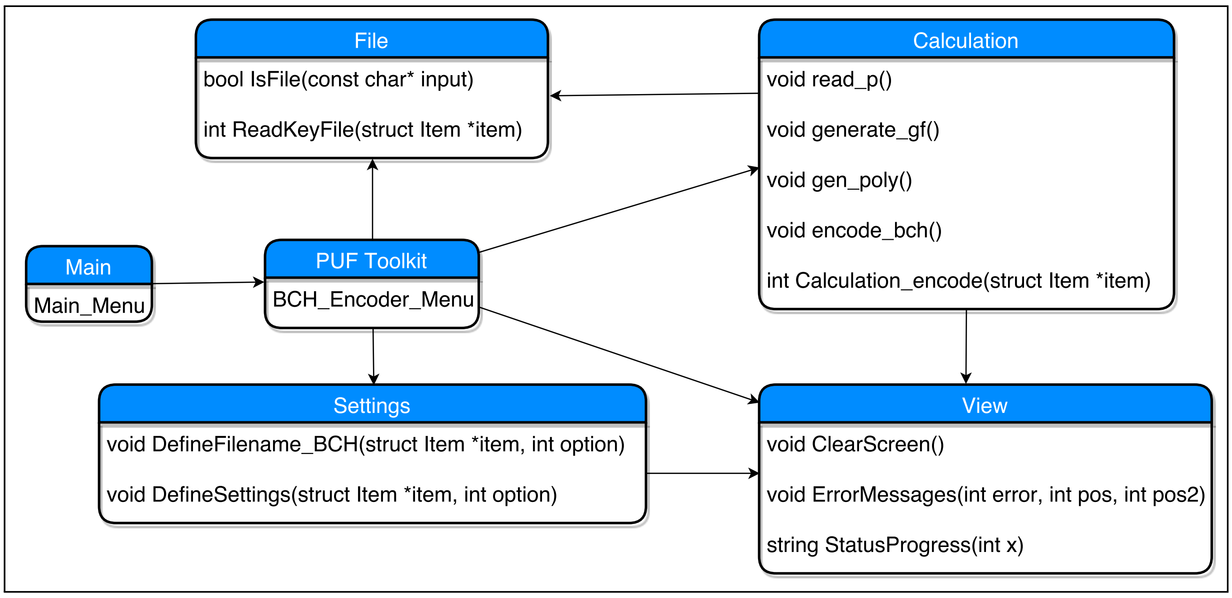

File
bool IsFile(const char* input)
int ReadKeyFile(struct Item *item)
Calculation
void read_p()
void generate_gf()
void gen_poly()
void encode_bch()
int Calculation_encode(struct Item *item)
Main
Main_Menu
PUF Toolkit
BCH_Encoder_Menu
Settings
void DefineFilename_BCH(struct Item *item, int option)
void DefineSettings(struct Item *item, int option)
View
void ClearScreen()
void ErrorMessages(int error, int pos, int pos2)
string StatusProgress(int x)

Figure 3.9: Dependencies of the BCH Encoder Modules

The functions of the BCH encoder arranged in modules that are related to each other are depicted in Figure 3.9. The data structure *Item* that was discussed in the section Hamming Distance Menu contains relevant data members related to BCH encoder, which is summarized in Table 3.3.

Name	Type	Description
offset_begin	long	Defines the starting point in a binary file (bytes to skip from beginning)
offset_end	long	Defines the ending point in a binary file (bytes to skip from end)
input_Key_length	long	Defines the length of the input key file
input_Key_name	char [102]	Char array for 102 symbols, to define the input key file
input_PUF_name	char [102]	Char array for 102 symbols, to define the input PUF file
output_HD_name	char [102]	Char array for 102 symbols, to define the output helper data file
BCHmode	char [25]	Char array for 25 symbols, to define the BCH mode
LR	int	Definition of the utilized linear repetition factor
result	char [52]	Char array for 52 symbols, to provide feedback

Table 3.3: Names and types of each element in the data structure *Item* for the *PUF BCH Encoder* and a description regarding their purpose.

3.3.1.1 Integration changes

Initially, the BCH Menu item was defined in the toolkit module to be called by the main menu, after that support for offset from beginning and end, was made available similar to Hamming Distance menu. Since the "Calculation module" functions of the BCH encoder are all called internally without any dependence on the user, so they were unified with the original PUF Toolkit Calculation module without much hindrance. The "Settings module" was merged to the toolkit where the original DefineFilename function was renamed to *DefineFilename_BCH* to avoid a name conflict with *DefineFilename* function of the original toolkit. The DefineFilename_BCH is dedicated to processing the input PUF filename, output helper data file and input secret key of the BCH encoder. The "File and View" modules were easy to unify with the toolkit, special care was taken to add the BCH global variables and global vectors to the toolkit. Changes in the "View module" consisted of majorly the changes in the *ErrorMessages* routine, which informs the user about the incorrect input and error states and the ways to recover from them. Major challenges faced during the integration were the same name functions as in the original toolkit that resulted in linking errors, they were resolved by renaming them appropriately for BCH encoder and all the declarations of the resulting functions were unified in the header files of the respective modules.

3.3.1.2 Verification

The correctness after merging the BCH encoder was authenticated via exhaustive run through all the test cases. In addition to that, a final verification of the entire implementation was done. While doing the verification, the following main points were given priority:

- The correctness of each module verified through utilization of test cases.
- All the possible menu options were checked and reviewed for their expected functionality
- Wrong inputs were tested to make sure that the toolkit accepts only legitimate inputs and displays appropriate error messages to rectify the error.

- Unusual workflows were inspected to assess their impact on Encoder functionality.

3.3.2 PUF BCH Decoder

We now go on to discuss the second phase of the fuzzy extractor i.e."regeneration" that uses PUF BCH Decoder as an essential element to recover the secret key that was encoded in the first phase, with the help of the generated helper data and a PUF response from the same device as used in the enrollment phase. Again we do not indulge in the exact implementation details since they are presented in great detail in [20].

This section looks at the user interface of the BCH decoder where the possible inputs and their meaning is discussed, then a summary of the main modules involved in the BCH decoder is presented with the help of an Entity Relation Diagram. Similar to BCH encoder section we examine the changes required during merging with the toolkit in the subsection "Integration Changes" and finally talk about the verification steps taken to assert the correctness of the merged decoder functionality in the toolkit.

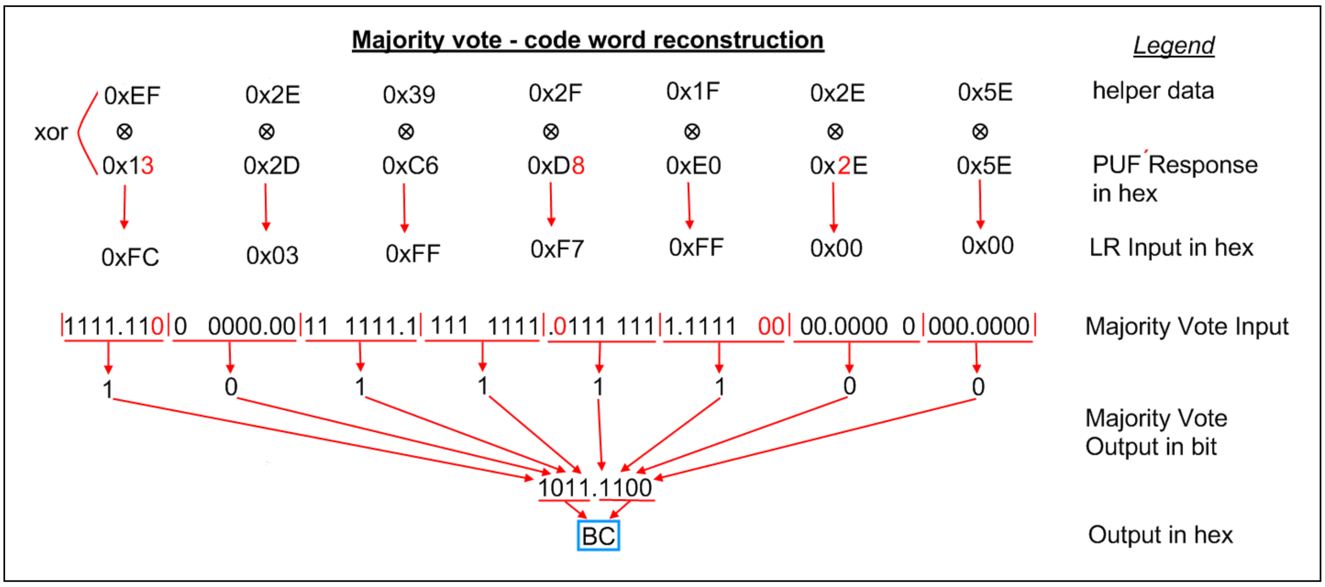
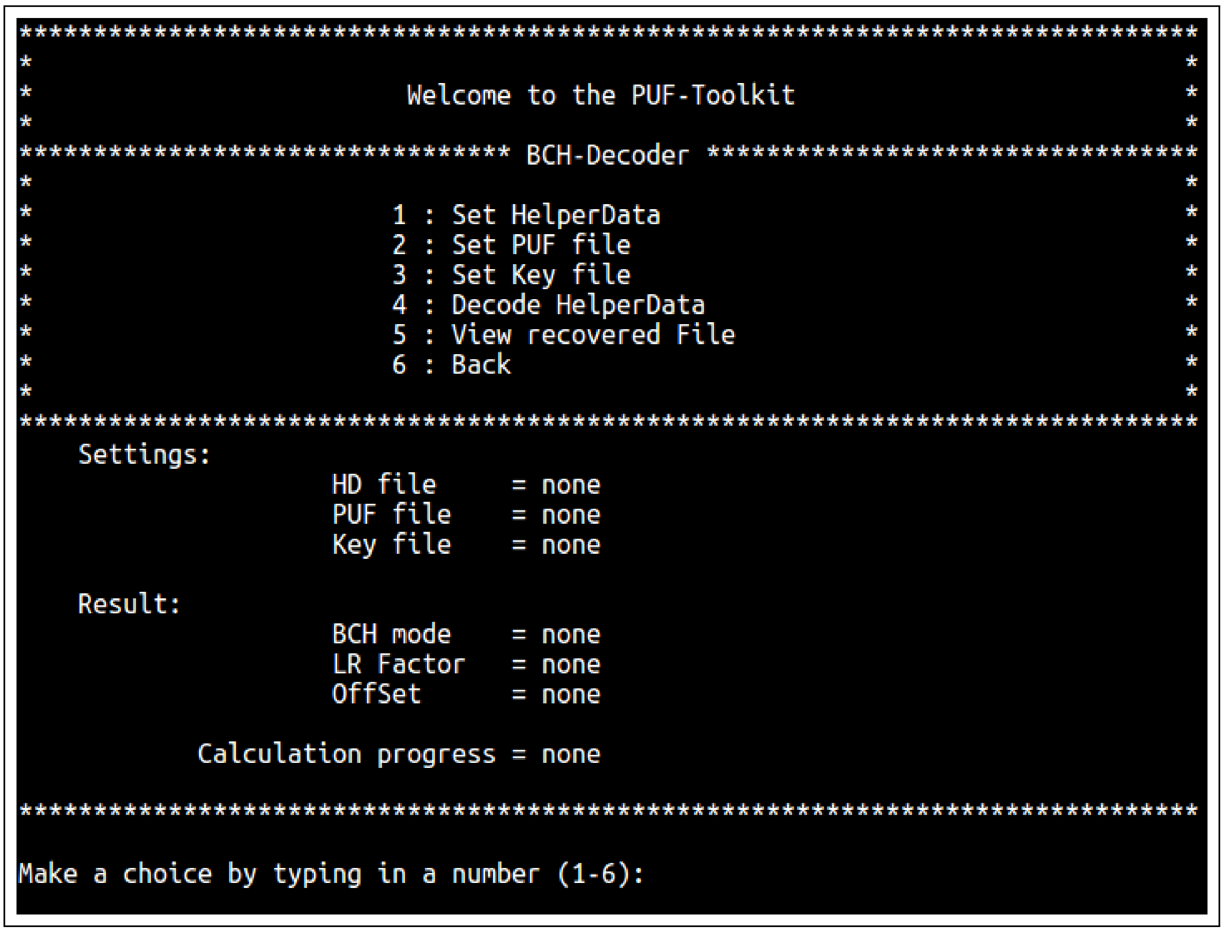


Figure 3.10: Exemplary code word reconstruction with the majority vote and factor 7, based on [14].

The decoder implementation is also the extension of the Morelos-Zaragoza's code [45] but before applying the decoding functionality, the LR encoded helper data received from the first phase is processed in the majority vote module of the fuzzy extractor as shown in Figure 3.10. This counteracts the linear repetition introduced in the LR coding module of the "enrollment phase". Figure 3.10 shows this operation on the helper data received from Figure 3.7, notice that the PUF response has some bits changed due to the various environmental factors (like temperature change and voltage fluctuation) that introduce intra-distance between the PUF responses from the same device. After the Majority Vote operation the BCH decoder recovers the configuration settings *cfg* that are appended to the end of the helper data to satisfy correct boundary conditions as seen in the BCH encoding phase. Usually, the BCH(n,k,d) can



```
*******************************************************************************
*                                                                             *
*                         Welcome to the PUF-Toolkit                          *
*                                                                             *
********************************* BCH-Decoder *********************************
*                                                                             *
*                        1 : Set HelperData                                   *
*                        2 : Set PUF file                                     *
*                        3 : Set Key file                                     *
*                        4 : Decode HelperData                                *
*                        5 : View recovered File                              *
*                        6 : Back                                             *
*                                                                             *
*******************************************************************************
    Settings:
                     HD file     = none
                     PUF file    = none
                     Key file    = none

    Result:
                     BCH mode    = none
                     LR Factor   = none
                     OffSet      = none

            Calculation progress = none

*******************************************************************************

Make a choice by typing in a number (1-6):
```

Figure 3.11: Console user interface of the *PUF BCH Decoder*.

correct up to t bit flip errors though along with Majority Voting and BCH decoding the fuzzy extractor can restore the original input secret key.

The User interface is a simple GUI similar to the BCH encoder shown in Figure 3.11, but in this case, the user need not enter the BCH mode details and offsets since they are predetermined in the encoding part and are saved as *cfg* settings. We only need to give the helper data filename and the PUF response filename to decode and restore the input secret key. The output key filename is required to store the recovered input secret key and using option "5' of the BCH Decoder menu, the retrieved key can be viewed without exiting the toolkit.

The five modules of the BCH decoder and their functions are depicted in Figure 3.12.
The members of the data structure *Item* that are relevant to the BCH decoder are summarized in Table 3.4. The only difference with the encoder is that we need input helper data filename instead of output helper data filename and output key filename instead of the input key filename.

3.3.2.1 Integration Changes

To integrate the BCH decoder on top of the already merged BCH encoder in the toolkit, required careful alterations to the code. The *Main_Menu* routine of the toolkit was modified to insert a BCH decoder

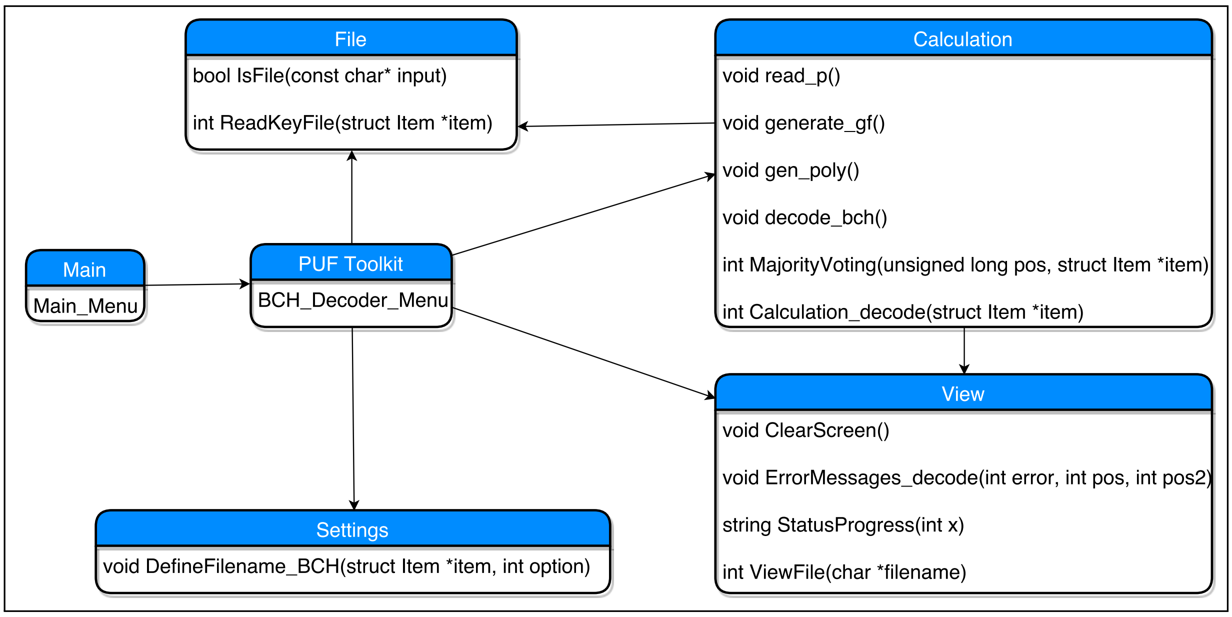

File
bool IsFile(const char* input)
int ReadKeyFile(struct Item *item)
Calculation
void read_p()
void generate_gf()
void gen_poly()
void decode_bch()
int MajorityVoting(unsigned long pos, struct Item *item)
int Calculation_decode(struct Item *item)
Main
Main_Menu
PUF Toolkit
BCH_Decoder_Menu
View
void ClearScreen()
void ErrorMessages_decode(int error, int pos, int pos2)
string StatusProgress(int x)
int ViewFile(char *filename)
Settings
void DefineFilename_BCH(struct Item *item, int option)

Figure 3.12: Dependencies of the BCH Decoder Modules

Name	Type	Description
offset_begin	long	Defines the starting point in a binary file (bytes to skip from beginning)
offset_end	long	Defines the ending point in a binary file (bytes to skip from end)
input_HD_name	char [102]	Char array for 102 symbols, to define the input helper data file
input_PUF_name	char [102]	Char array for 102 symbols, to define the input PUF file
output_Key_name	char [102]	Char array for 102 symbols, to define the output key file
BCHmode	char [25]	Char array for 25 symbols, to define the BCH mode
LR	int	Definition of the utilized linear repetition factor
result	char [52]	Char array for 52 symbols, to provide feedback

Table 3.4: Names and types of each element in the data structure *Item* for the *PUF BCH Decoder* and a description regarding their purpose.

menu that can be invoked by the user. The “Calculation” module functions were merged without much conflict with the existing functions, the function *generate_gf*, responsible for Galois Field computation was unchanged. The other two functions *read_p_decode and gen_poly_decode* that are used for the polynomial generation were renamed because of the overlapping function names with the encoder. The error messages displaying routine of the “View module” was redeclared as *ErrorMessages_decode* because the error states and the possible inputs for the decoder are different from the BCH encoder.

MajorityVoting and *ViewFile* were added to the family of BCH functions and correspondingly their declarations written in respective header files. The routine *DefineFilename_BCH* was specifically modified to take two more arguments i.e. input helper data filename and output recovered key name since these two arguments were not implemented in the encoder. Accordingly, the body of the same routine was changed to display appropriate messages for the input of helper data and output key name that would be restored after decoding. The other challenge was to find and modify all the invoking points to the newly renamed functions, this was resolved by careful code investigation and with the help of insight gained via compiler and linker errors.

3.3.2.2 Verification

For verifying the integrated BCH decoder a similar approach to BCH encoder was taken, the correctness was authenticated by running all the test cases that cover all the code flows including exceptions and error states. A final verification of the entire implementation was done by cross-checking the recovered key with the original secret key.

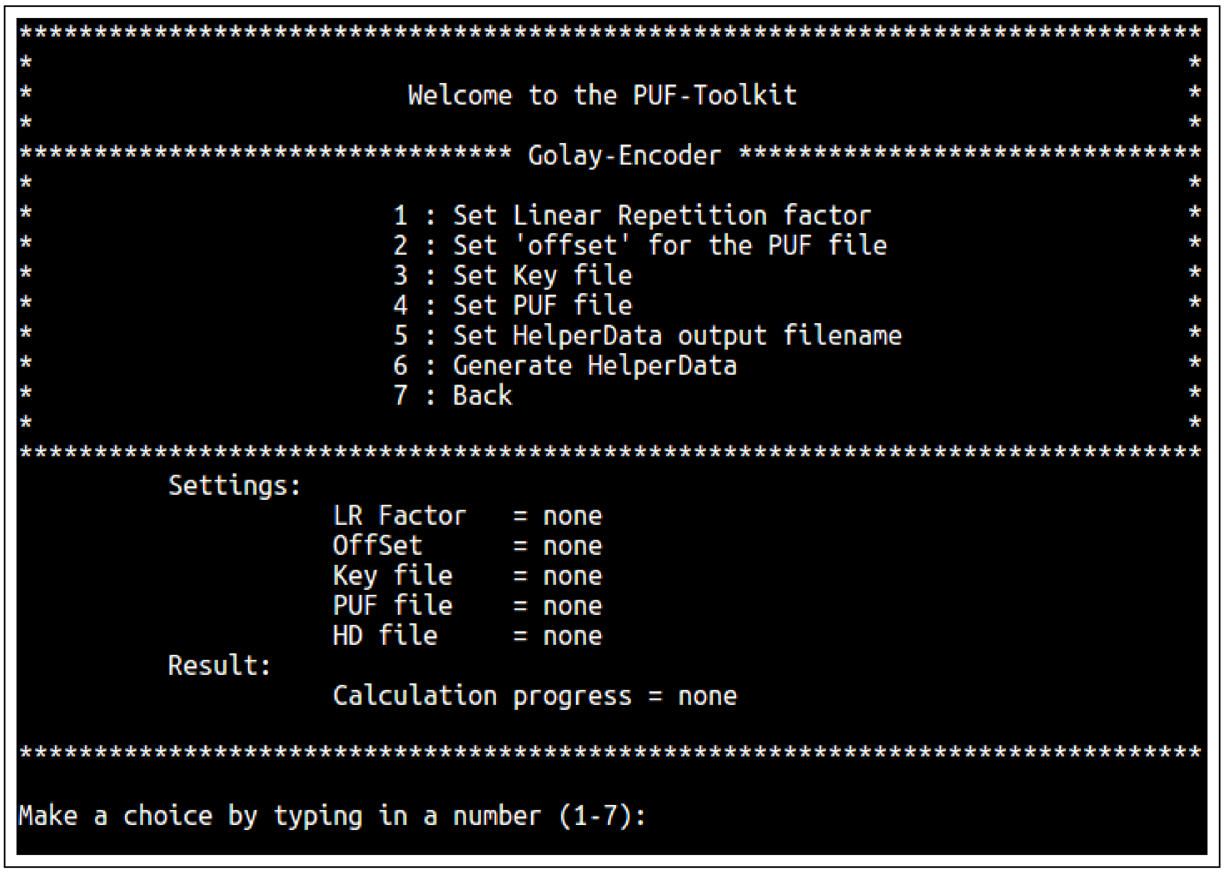
- The correctness of each module was verified through utilization of test cases.
- All the possible menu options were checked and reviewed for their expected functionality
- Erroneous inputs to the toolkit were checked to make sure that the toolkit accepts only legitimate inputs and displays appropriate error messages to rectify the error.
- Unusual workflows were inspected to assess their impact on Decoder functionality.
- The restored key was matched with the original secret key using the Linux command line utility “diff”.

3.3.3 Golay Encoder

This section discusses the Golay Encoder that was integrated into the toolkit. We first discuss the user interface then look at the functions and modules of the Golay encoder followed by a basic description of the important routines in the code and then the steps taken to verify the implementation.

The UI is similar to BCH encoder but in this case we do not need to set the dimensions of the code “n, k and d” because Golay is a static code with the universal dimensions of (23, 12, 7) as seen in section 2.2.2. The user interface is portrayed in Figure 3.13. Using the first option the user can set the Linear repetition factor as ’7’ or ’15’, the offsets of the PUF file can be adjusted using option two of the Golay Encoder menu. After this, it is mandatory to input the secret key filename, to be stored securely, and input the PUF filename to generate the helper data using the fuzzy extractor algorithm, using option three and four of the Golay Encoder Menu respectively. The helper data filename is provided by the user by selecting option five. If all the inputs are satisfied and conditions are met then the helper data will be generated by choosing the last option six of the Golay Encoder Menu.

The functions of the Golay Encoder are divided into four modules similar to the BCH encoder depicted in Figure 3.14. The *Setting* module contains the familiar functions “DefineFilename” and “DefineSettings” along with “DefineOffsetlength” which sets the value of the offsets. The *View* module contains the “ErrorMessages” routine which is implemented in the toolkit already to check for incorrect inputs by the user and other routines to check the progress status and clear the screen. The *Calculation* module forms the backbone of the Golay Encoder. Its details are listed below:



```
****************************************************************************
*                                                                          *
*                          Welcome to the PUF-Toolkit                      *
*                                                                          *
******************************** Golay-Encoder *****************************
*                                                                          *
*                         1 : Set Linear Repetition factor                 *
*                         2 : Set 'offset' for the PUF file                *
*                         3 : Set Key file                                 *
*                         4 : Set PUF file                                 *
*                         5 : Set HelperData output filename               *
*                         6 : Generate HelperData                          *
*                         7 : Back                                         *
*                                                                          *
****************************************************************************
         Settings:
                    LR Factor   = none
                    OffSet      = none
                    Key file    = none
                    PUF file    = none
                    HD file     = none
         Result:
                    Calculation progress = none

****************************************************************************

Make a choice by typing in a number (1-7):
```

Figure 3.13: Console user interface of the *PUF Golay Encoder*.

- *Golay_encode* routine's main task is to read and allocate memory for the PUF and Key file using their filenames that are given by the user. This routine then makes a call to the "generateHelperData" function with the initialized data members of the Item structure and memory pointers to the read files.

- *generateHelperData* generates the encoding table using another internal function *get_syndrome* (this table is analogous to the generator matrix discussed in section 2.2.2 of the Golay Code). Then the secret key is encoded using the Golay Coding technique. The generated Codewords are stored in an array that undergoes the linear repetition process and finally, the LR_Encoded codewords are XORed with the PUF response to obtain the helper data. The generateHelperData function stores the final output in the helper data filename specified by the user along with the "cfg" settings i.e. offsets, LR factor and secret key filesize. This information will be read and used in the decoder module. Table 3.5 summarizes the data structure "Pattern" that presents the cfg settings.

- *get_syndrome* function computes the generator polynomial for the corresponding (23, 12, 7) Golay code.

A helper function *SetInputLen* was implemented to aid the encoder functions in calculating the length of the file based on the offsets. This function is called internally before reading and allocating the memory for key and PUF response. It takes two arguments, first the Item structure with initialized data

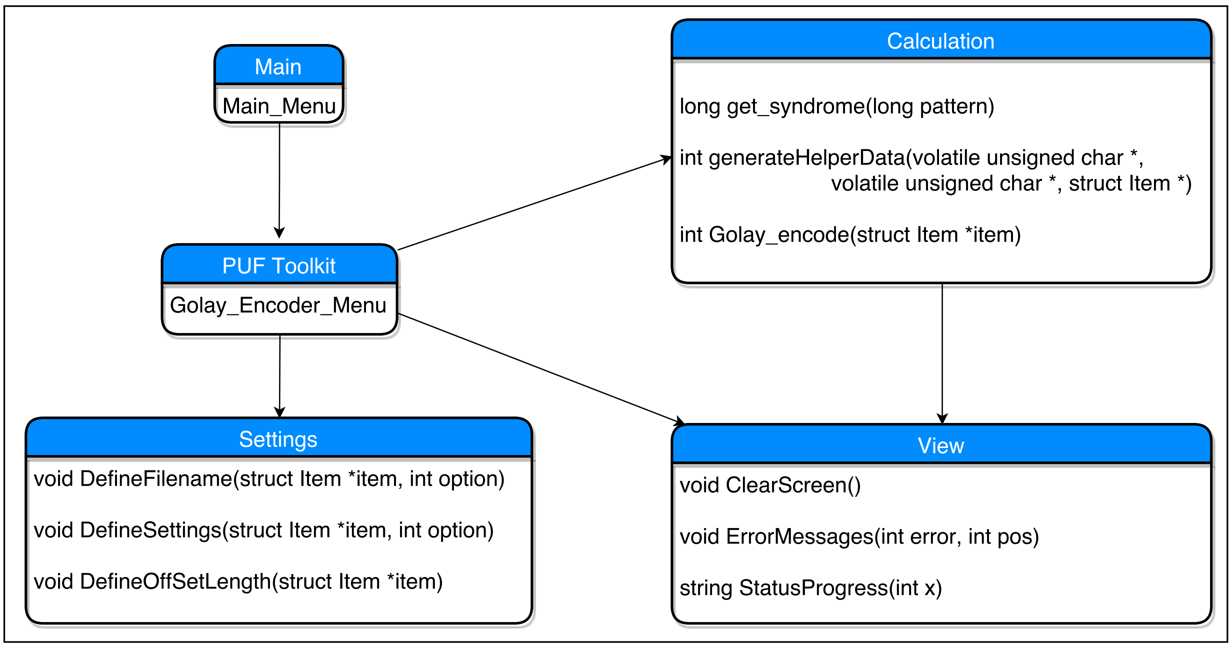

Main
Main_Menu
PUF Toolkit
Golay_Encoder_Menu
Calculation
long get_syndrome(long pattern)
int generateHelperData(volatile unsigned char *, volatile unsigned char *, struct Item *)
int Golay_encode(struct Item *item)
Settings
void DefineFilename(struct Item *item, int option)
void DefineSettings(struct Item *item, int option)
void DefineOffSetLength(struct Item *item)
View
void ClearScreen()
void ErrorMessages(int error, int pos)
string StatusProgress(int x)

Figure 3.14: Illustration of the Golay Encoder modules and their inter dependency.

Name	Type	Description
errorCode	int	Defines the utilized error-correction code (0 = Golay, 1 = BCH)
Golay_BCH_length	int	Defines the *length* of one code word, for Golay and BCH
Golay_d_BCH_t	int	Defines for Golay the distance *d* and for BCH the correction capability '*t*'
Golay_k_BCH_m	int	Defines the *message length* for Golay and the parameter *m* for BCH
linearRep	int	Defines the utilized linear repetition factor
puf_offSet_begin	long	Defines the starting point in a binary file (bytes to skip) from beginning
puf_offSet_end	long	Defines the ending point in a binary file (bytes to skip) from the end
original_filesize	long	Defines the length of the original input key file

Table 3.5: Names and types of each element in the data structure *Pattern* for the *PUF Golay Encoder* and a description regarding their purpose.

members as filenames and a second integer argument that points to PUF response or secret key, based on these values it calculates the input length as $input\_length = original\_filesize - (Offset\_end + Offset\_beginning)$.

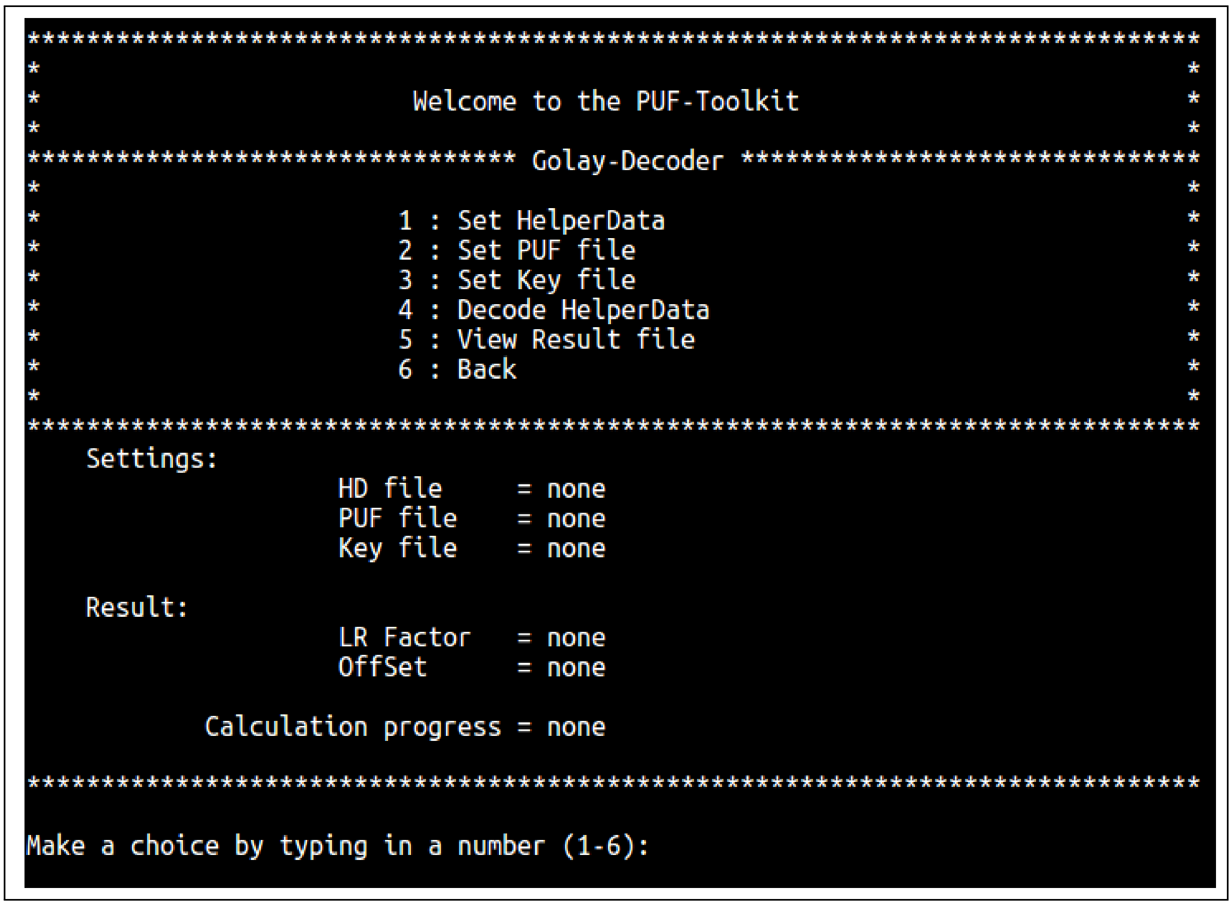
3.3.3.1 Verification

The user interface was designed to follow the same pattern as BCH encoder to simplify the user experience. The "Main" module was modified to add one more menu item *Golay_Encoder_Menu*. The merging and reuse of the existing functions was done with great care and detail to modularize the code that is suitable for further extension in the future. The verification of the consolidated code was done keeping in mind the same steps as BCH encoder:

1. The correctness of each module verified through utilization of test cases.

2. All the possible menu options were checked and reviewed for their expected functionality
3. Wrong inputs were tested to make sure that the toolkit accepts only legitimate inputs and displays appropriate error messages to rectify the error.
4. Unusual workflows were inspected to assess their impact on Golay Encoder functionality.

Due to the static nature of the Golay Code (23, 12, 7) it can only correct up to “three errors” per “twelve bits” of the message. Consequently, this code is not suitable for encoding large secret files. Extensive testing showed that when the resultant helper data is greater in size compared to the PUF response, the secret file is not recovered completely and via extensive testing the filesize for complete recovery of the secret key was approximated to 2340 bytes for a 32 kilobyte PUF response. So the recovery of ssh private keys which are typically greater than 3000 bytes is not possible using Golay and SRAM PUF responses. Therefore, the user must choose the BCH fuzzy extractor instead.



```
********************************************************************************
*                                                                              *
*                         Welcome to the PUF-Toolkit                           *
*                                                                              *
********************************* Golay-Decoder ********************************
*                                                                              *
*                         1 : Set HelperData                                   *
*                         2 : Set PUF file                                     *
*                         3 : Set Key file                                     *
*                         4 : Decode HelperData                                *
*                         5 : View Result file                                 *
*                         6 : Back                                             *
*                                                                              *
********************************************************************************
    Settings:
                    HD file     = none
                    PUF file    = none
                    Key file    = none

    Result:
                    LR Factor   = none
                    OffSet      = none

            Calculation progress = none

********************************************************************************

Make a choice by typing in a number (1-6):
```

Figure 3.15: Conceptual design of the Golay Deocder user interface.

3.3.4 Golay Decoder

The next step after encoding the secret key and obtaining the helper data is to recover the key via Golay decoding. This subsection presents the user interface and explains the procedure to restore the securely stored secret key, then the modules and functions of the decoder are illustrated. The discussion then follows the code improvement done to enhance the menu of the toolkit and finally the verification strategy to check the integrated changes is shown.

The user interface as shown in Figure 3.15 has five main options where the user must input the helper data filename that was generated in the enrollment phase using option one of the Golay Decoder menu. The other two mandatory options are to set the PUF response filename that is generated from the same PUF instance as used in encoding phase and to specify the output key filename where the secret key will be restored. If all the inputs are correctly set then the "regeneration" can be performed via option four. The output and status is displayed under the *Result* section of the UI whereas *Settings* subsection of the interface shows the filenames. If the input is incorrect or if the given filename is not present on the disk then a suitable error with details to recover from a faulty state is shown to the user.

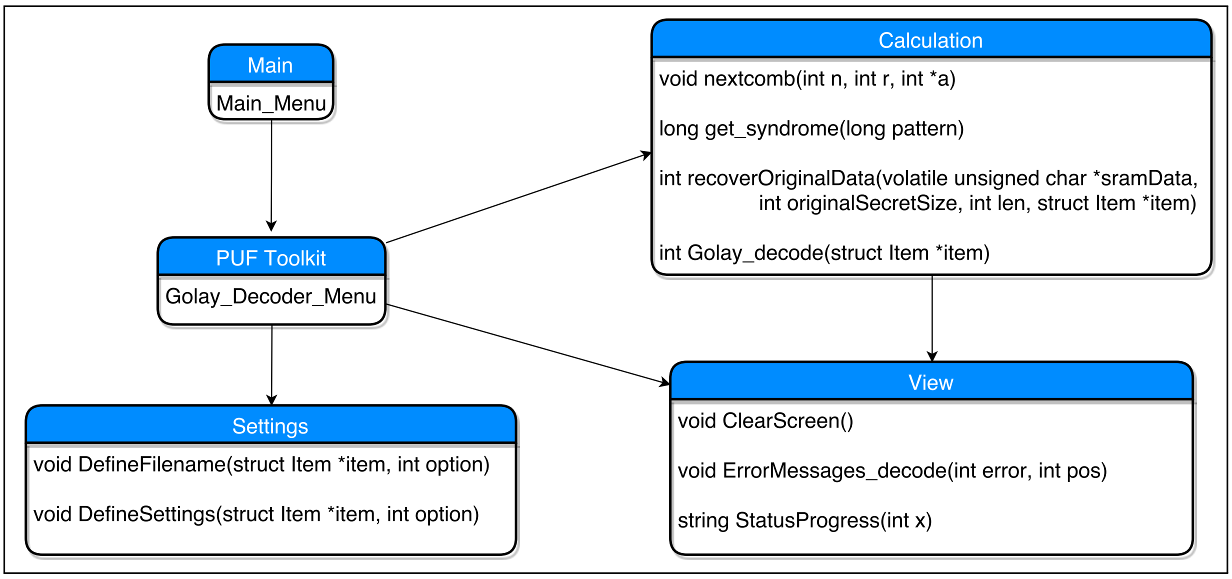

Main
Main_Menu
PUF Toolkit
Golay_Decoder_Menu
Calculation
void nextcomb(int n, int r, int *a)
long get_syndrome(long pattern)
int recoverOriginalData(volatile unsigned char *sramData, int originalSecretSize, int len, struct Item *item)
int Golay_decode(struct Item *item)
Settings
void DefineFilename(struct Item *item, int option)
void DefineSettings(struct Item *item, int option)
View
void ClearScreen()
void ErrorMessages_decode(int error, int pos)
string StatusProgress(int x)

Figure 3.16: Illustration of the Golay Decoder modules and their inter dependency.

Golay decoder code is partitioned across five modules similar to the ones discussed in Golay encoder as illustrated in Figure 3.16. The first module is the Main_Menu function that calls the Golay decoder menu. This menu was implemented and added to the PUF toolkit, and when it is selected it points to the user interface with menu items of the Golay Decoder. The Golay Decoder user interface is implemented in the routine *Golay_Decoder_Menu*, where the decode option is linked to the *Golay_decode* function that performs the decoding of the helper data. The usual routines in the "View" module are used to set the helper data, PUF response and output recovered key filenames. There is no need to set the offsets and original key filesize in the decoder menu since during the "enrollment" phase these details are appended at the end of helper data as part of the 'cfg' settings. The "Calculation" module is crucial to the decoding operation and follows the below-mentioned steps:

1. The Golay decode function reads the 'cfg' settings using a subroutine *read_infos* that reads the settings appended at the end of the helper data and initializes the corresponding data members of

the Item structure. The details of the data members used in Golay decoding are listed in Table 3.5. Then based on the settings the input length of the PUF response is calculated as $input\_length = original\_filesize - (Offset\_end + Offset\_beginning)$. The files helper data and PUF response are read and stored in the memory and the control is given to the *recoverOriginalData*.

2. The next function *recoverOriginalData* performs the decoding of the helper data. It first applies "Majority Voting" on the encoded helper data to reverse the effects of the "Linear Repetition" done during enrollment phase. With the help of an API *next_comb* the decoding table for Golay code (23, 12, 7) is calculated and is used to recover the original key.

3. After the restoration of the key is complete it is saved in the output key filename by calling a subroutine *SaveFile* that takes the recovered key (represented in code as a character array) and output filename as arguments and uses standard C file operations to write it to the file.

The *ErrorMessages* function was used to detect incorrect input and display the errors to the user since it is redundantly used in the other menu functionalities of the PUF toolkit like BCH encoder, we summarize the Error Codes in Table 3.6

3.3.4.1 Verification

After the merging is complete and the Golay decoder menu is functional it was tested rigorously to make sure all the options work as expected. The same approach for verification was taken as mentioned in previous subsections and a final check was done to compare the recovered key with the original secret to ensure the overall functionality of both the Golay Encoder and Decoder is correct.

- Every menu item of the Golay Decoder was checked and reviewed to make sure it behaves as expected.
- Incorrect inputs were inserted deliberately to verify the ErrorMessages routine displays corresponding error messages and ways to recover from the incorrect state of the toolkit.
- Unusual workflows were inspected to assess their impact on Decoder functionality.
- The recovered key was compared with the original secret key via the Linux "diff" command line utility and they were found to be identical.

3.4 Jaccard Index

This section explains the Jaccard Index implementation that was developed as a new feature for the toolkit. Then the two sub-menu options Intra-Jaccard and Inter-Jaccard Index are presented as subsections. The modules and code structure was inspired from the Hamming Distance Menu since both of the metrics (Hamming Distance and Jaccard Index) are closely related to each other and operate on either files or folders. We first discuss the definition of Jaccard Index and its relation to the Hamming weight. We then look at the user interface for the Jaccard Index Menu which contains two more sub-menu options *Intra-* and *Inter-Jaccard index* and then go on to explain the modules of the Jaccard index. In the end, the section presents the verification steps after the development of this new feature was complete.

Error code	Error message
1	ERROR! Invalid input - Only digits are allowed! No digit at *pos + 1*.
2	ERROR! Invalid input - Input too long.
3	ERROR! Invalid input - Try again...
4	ERROR! Opening output file.
5	ERROR! Invalid input - 'Filename' too long.
6	ERROR! Invalid input - No input.
7	ERROR! Reading PUF file.
8	ERROR! Opening PUF file.
9	ERROR! Invalid input - PUF filename is not valid.
10	ERROR! Invalid input - Key filename is not valid.
11	ERROR! Opening HelperData file.
12	ERROR! Reading HelperData file.
13	ERROR! Selected an invalid choice.
14	ERROR! HelperData: The recovered 'decoding information' is invalid! -> Parameter '*m*' is out of range.
15	ERROR! HelperData: The recovered 'decoding information' is invalid! -> Parameter '*t*' is out of range.
16	ERROR! HelperData: The recovered 'decoding information' is invalid! -> Parameter '*length*' is out of range.
17	ERROR! HelperData: The recovered 'decoding information' is invalid! -> Parameter '*LR factor*' is not 7 or 15.
18	ERROR! HelperData: The recovered 'decoding information' is invalid! -> Parameter '*offSet*' is out of range.
19	ERROR! HelperData: The recovered 'decoding information' is invalid! -> Parameter '*filesize*' is out of range.
20	ERROR! Opening file to view.
21	WARNING! Looks like no decoding was performed.
22	Looks like the calculation was not started.
23	ERROR! Looks like the file was empty.
24	ERROR! Looks like the file was not in the supported format.
25	WARNING! Looks like the 'offset' is not set.
26	ERROR! Looks like the 'Output file or Result' is not set/calculated.

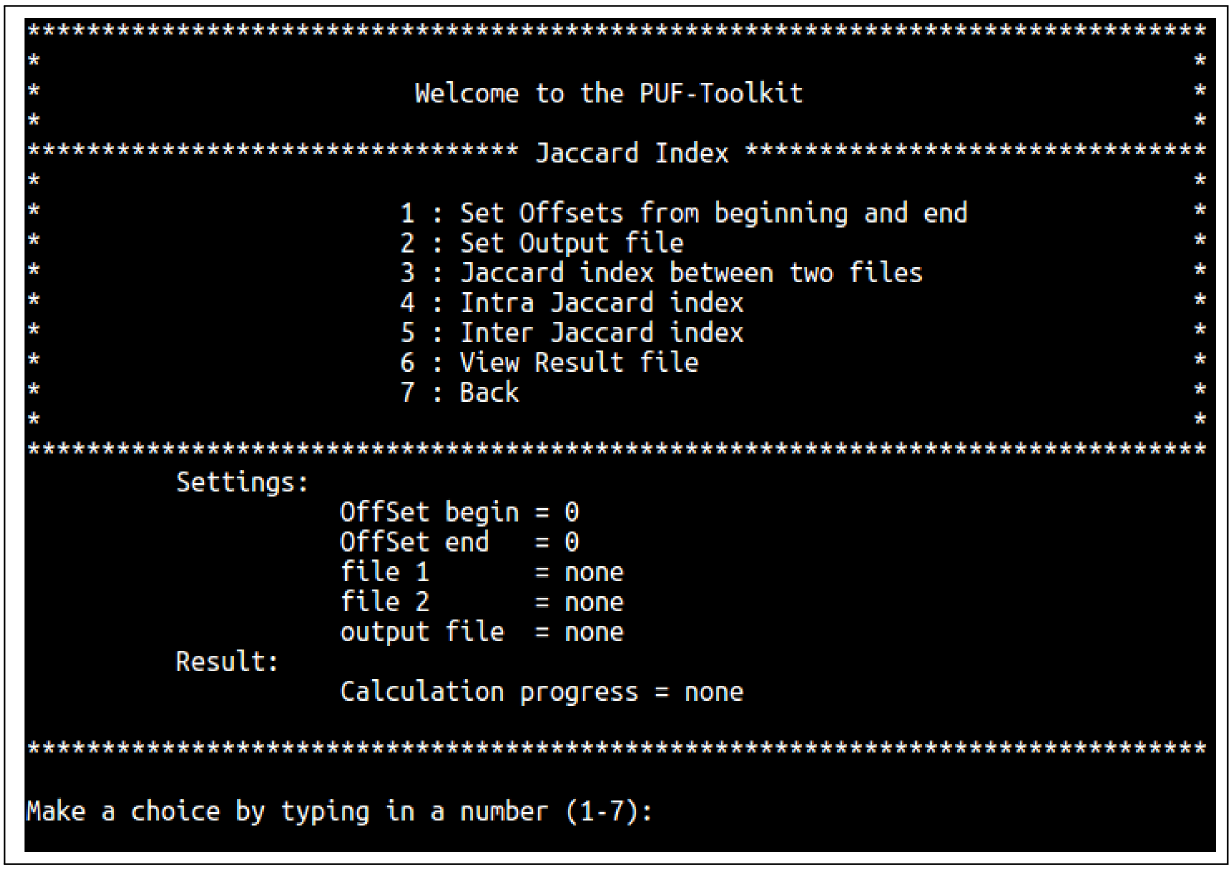
Table 3.6: Error codes and corresponding error messages for the *PUF Golay Decoder*.

- **Definition**: Jaccard index between two bytes A and B is defined by the Equation 3.3

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{3.3}$$

  where |A| is the Hamming weight of A, |B| is the Hamming weight of B and $|A \cap B|$ is the number of times both A and B have one in the same bit position.

For example, if we have two bytes A = 1100 1111 and B = 1111 0000 then |A| = 6, |B| = 4 and $|A \cap B|$ = 2 (since only first two common positions have ones in both the bytes). Substituting these values in the above equation we get $J(A,B) = (2)/(6+4-2) = 0.25$.



```
*****************************************************************************
*                                                                           *
*                         Welcome to the PUF-Toolkit                        *
*                                                                           *
******************************** Jaccard Index ******************************
*                                                                           *
*                        1 : Set Offsets from beginning and end             *
*                        2 : Set Output file                                *
*                        3 : Jaccard index between two files                *
*                        4 : Intra Jaccard index                            *
*                        5 : Inter Jaccard index                            *
*                        6 : View Result file                               *
*                        7 : Back                                           *
*                                                                           *
*****************************************************************************
          Settings:
                     OffSet begin = 0
                     OffSet end   = 0
                     file 1       = none
                     file 2       = none
                     output file  = none
          Result:
                     Calculation progress = none

*****************************************************************************

Make a choice by typing in a number (1-7):
```

Figure 3.17: Conceptual design of the Jaccard Index user interface.

The user interface for the Jaccard Index menu is similar to the Hamming Distance Menu as shown in Figure 3.17. Using the first option of this menu, the user can set the offsets from the beginning and the end of the file and as a result, those bytes will be skipped during Jaccard Index calculation. The second option lets the user set the output filename which is mandatory since the results will be stored in this file. The third option performs the computation of the Jaccard index between the two given files. It first asks for the two filenames to be prompted by the user and then the result along with the *Fractional Index* is stored in the output file. Option four and five of the Jaccard Index menu call the Intra- and Inter-Jaccard Index routines respectively and are discussed in following subsections. The six and the last option can be used to view the output file without exiting the toolkit.

The modules of the Jaccard's index are similar to those of the Hamming Distance and therefore this section takes the liberty of not showing it here again. Rather than looking at each and every module, we look at the main routines and algorithm to calculate the Jaccard Index.

- The main Jaccard index between two files is calculated by the *Jaccard_Index* that takes data structure Item pointer as an argument with initialized filenames. It first sets the input length of both the files based on the offsets from beginning and end by calling the *SetInputLen* API which was designed specifically for this purpose.

- Then the files are read using the sub-routine *readfile* that takes the name of the file, offset from beginning and filesize as an argument, based on that it reads and *mallocs* the file in memory and passes the pointer to the allocated memory back to the caller function.
- To get the common number of “1s” at same bit positions in both the files, a bitwise AND operation is performed on the read files and the result is stored in a character array. The resultant character array contains the data with “1s” in the same bit position since bitwise AND operation only yields result “one” if both the input bits are “one”. This operation can be seen in Table 3.7.
- Once the bitwise AND result is obtained then the calculation of Jaccard Index is relatively simple. The number of ones in each input file and in the bitwise ANDed output file are computed by calling the *hamming_wt* function explained in section 3.2 under Listing 3.2 and then the values are substituted in Equation 3.3.
- An additional metric called Fractional Jaccard’s Index is also calculated using the definition $F(A,B) = J(A,B)/size(AorB)$ and written to the result file along with the Jaccard Index.

3.4.1 Intra-Jaccard Index

Instead of computing Jaccard index between two files, the entire folder containing PUF responses from a single device can be evaluated by selecting the Intra-Jaccard Index sub-menu item. The filename where the results will be written is essential before processing the Intra-Jaccard Index. The menu item asks for an input directory with a relative or absolute path on the machine and if the input path is correct and an output file is set then the result is generated and stored in the output file.

Contrary to the intra-Hamming distance the “Switch output-style” (refer section 3.2.1) option was not made available to the user to simplify the functionality. The default output-style was set as *Compact* which is explained in detail in the Intra-Hamming Distance section 3.2.1 and illustrated in Figure 3.4. A sample result output file is presented in Figure 3.18 which shows the compact mode comparison between the files in the directory where the Jaccard Index in conjunction with fractional Jaccard Index is written for each comparison.

To realize the Intra-Jaccard Index the function *Jaccard_Intra* was implemented that takes the directory as input and first checks if the path is valid and there are more than one files present in the folder.

Bitwise AND of two bytes A and B	
Byte A	1100 0000
Operation	&
Byte B	1111 0011
Output C	1100 0000

Table 3.7: Bitwise ANDing the two bytes to get the common ones in the same bit position

```
*******************************************************************************

                             Intra-Jaccard Index

                          Used Settings:

                          Device folder: 'data/check'
                          offset_begin:         400
                          offset_end  :         0
                          Length:         32385 byte (259080 bit)

*******************************************************************************


-------------------------------------------------------------------------------
                                                                       Filename
                                                                 data/check/PUF4
-------------------------------------------------------------------------------
data/check/PUF2                                  0.88839          0.00003
data/check/PUF1                                  0.88704          0.00003
data/check/PUF3                                  1.00000          0.00003
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
                                                                       Filename
                                                                 data/check/PUF2
-------------------------------------------------------------------------------
data/check/PUF1                                  0.88414          0.00003
data/check/PUF3                                  0.88839          0.00003
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
                                                                       Filename
                                                                 data/check/PUF1
-------------------------------------------------------------------------------
data/check/PUF3                                  0.88704          0.00003
-------------------------------------------------------------------------------
```

Figure 3.18: A sample output result showing the *compact* output style format for Intra-Jaccard Index

Then all filenames with their absolute paths are stored in a standard C++ vector data structure. The *Jaccard_Intra* then iterates through this vector and computes the Jaccard Index for each comparison by calling the *Jaccard_Index* procedure that was explained in section 3.4 above. Finally, the result is written in each iteration of the loop and appended to the output file.

3.4.2 Inter-Jaccard Index

Unlike its Intra- counterpart, the Inter-Jaccard Index can be used to evaluate the PUF responses across different devices. It compares the PUF responses in a directory (analogous to a device since all PUF responses from a single device are stored in the same folder) to PUF responses in other directories. The output-style format for the Inter-Jaccard Index by default is designed in the code to be "compact" since it simplifies the metric and also avoids any duplicate comparisons. The details of the compact mode file comparison for Inter-Jaccard index are the same as for Inter-Hamming Distance which is explained in detail in section 3.2.2 and the compact comparison operation between three folders is shown in Figure 3.5.

The code implementation for Inter-Jaccard Index is achieved by the routine *Jaccard_Inter* which takes the data structure Item as an argument. The user needs to input the number of paths or folders he/she wants to evaluate, this number is bounded between *two* and *ninety-nine* i.e. $2 <= n < 99$, then the folder names with their absolute or relative path are required by the toolkit. All these paths are verified by the code as valid and present in the memory, the routine *Jaccard_Inter* verifies if each given folder has at least one file. These folder names and paths are then stored in a standard C++ vector which is navigated in compact comparison fashion to compute the Jaccard Index. The nested for loop iterations are abstracted in this text to avoid complexity. In each iteration of the nested loop the comparison between PUF responses of different devices is achieved by redundant calls to the *Jaccard_Index* function explained in section 3.4 and the result is written to the output file as specified by the user.

3.4.3 Verification

The merging of a new feature to the toolkit required careful testing to ensure correctness and exceptions to be handled gracefully. Some of the approaches were inspired from the fuzzy extractor testing and are summarized below:

- All the menu item were checked and reviewed for their expected functionality.
- Testing was done for faulty and incorrect inputs to the Jaccard Index menu options to ensure relevant error messages are displayed to the user. For example, incorrect paths or filenames of the directories will be prompted to the user suggesting to enter the paths/filenames again.
- The Jaccard output file was reviewed to check for proper header information and tables.
- For Intra- and Inter-Jaccard index, the mandatory options like setting of the offsets and the output file was checked before computing the index. In case these settings are not configured then relevant error information was displayed to the user.

This marks the completion of the improvement and development phase of the toolkit. Next part of the report discusses the integration of the toolkit to the CogniCrypt opensource library by interfacing the C++ APIs to the Java runtime via Java Native Interface.

3.5 CogniCrypt Integration

To make the C++ source code functions available to the Java runtime they must be interfaced using the Java Native Interface Technology. This section looks at the details of the Java Native interface, most of which is inspired from [46], then the code implementation for the JNI is explained. Next, the wrapping procedure of the toolkit as a shared library and packaged as a Java Archive (JAR) to be externally linked by the Java source code is clarified. The final subsections describe the CogniCrypt and its Clafer model along with the XSL coding techniques to generate the Java boilerplate code to assists the Java developers.

3.5.1 Java Native Interface

The Java Native Interface standard and programming techniques are defined by Oracle^{TM} as a native programming interface which allows Java code that runs inside a Java Virtual Machine (JVM) to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly. The JNI does not impose any restriction on the implementation of the underlying Java Virtual Machine so the native code works irrespective of the Virtual machine.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class jni_toolkit */

#ifndef _Included_jni_toolkit
#define _Included_jni_toolkit
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     jni_toolkit
 * Method:    hammingwt
 * Signature: (Ljava/lang/String;Ljava/lang/String;Z)V
 */
JNIEXPORT void JNICALL Java_jni_toolkit_hammingwt
  (JNIEnv *, jclass, jstring, jstring, jboolean);
```

Listing 3.3: Auto generated header file showing the syntax for Java Native Interface function declaration

To implement the JNI code a lot of help was taken from the tutorial [47], for each metric of the toolkit an API was designed in accordance with the syntax of the JNI. The text explains the development process for a single metric, Hamming weight, these steps are applied to all the other metrics too. The toolkit.java source file was written containing the native interfaces to the functions in the PUF toolkit. For example, Hamming weight functionality of the toolkit was available via a public *native* Java function *hammingwt* that takes “file/folder name”, “output filename” and “mode”(file mode or folder mode) as arguments. These native APIs are the class member functions of the Java *toolkit* class.

The compilation of the toolkit Java source was performed resulted in the generation of the toolkit.class file. Next, the header file to be included in the JNI C++ source code was generated using the “javah” utility. The resultant header file contains the syntactical prototypes of the C++ interface functions that must be implemented. Such a file with Hamming weight metric is shown in Listing 3.3.

The *Java_jni_toolkit_hammingwt* interface first parses the arguments that it obtained from the Java source code. It takes the jstring arguments file/folder name and output filename and converts them into standard C character array using the standard JNI conversion function *GetStringUTFChars*. The jboolean argument is implicitly an integer so it does not require any transformation. The code then initializes the "Item" data structure with these values and calls the *HammingWeight(Item *, int)* function of the toolkit to compute the Hamming weight of the file or the entire folder based on the value of $mode$.

The *ErrorMessages* routine of the toolkit is used to inform the user about invalid inputs and faulty toolkit states. Since these inputs are passed from the Java source as strings, the Java developers can check at their end for the authenticity of the file/folder names and their path on the machine. Other metrics of the PUF toolkit like "entropy", "median and average" etc. are implemented similarly in the Java source and C++ source code.

3.5.1.1 Packaging in Java Archive

The C++ JNI source code along with the entire PUF toolkit must be compiled and linked as a shared library since the Java source code calls the native functions from the shared library or shared object. Before packaging the shared library and other Java-related class files in a JAR file the toolkit must be compiled as a shared object file. This is achieved by compiling and linking the PUF toolkit with the "-fPIC" and "-shared" flag of the GNU g++ compiler. The resultant "libtoolkit.so" can then be packaged in a JAR file.

The JNI standard provides the API *System.loadLibrary* to load the shared objects or Dynamic-Link Libraries (DLL) from the Java source code, but the CogniCrypt project requires the shared library and the Java toolkit.class to be packaged in a JAR , so that they can be included in the project as an External Library. For this purpose, another class is needed that can extract the shared object from the JAR , load it using the standard JNI function and then delete before the Java runtime exits. Such a wrapper class was already implemented and after a few changes, it was tailored to meet the needs of this project. To know more details about the implementation of this Java source please refer to [48].

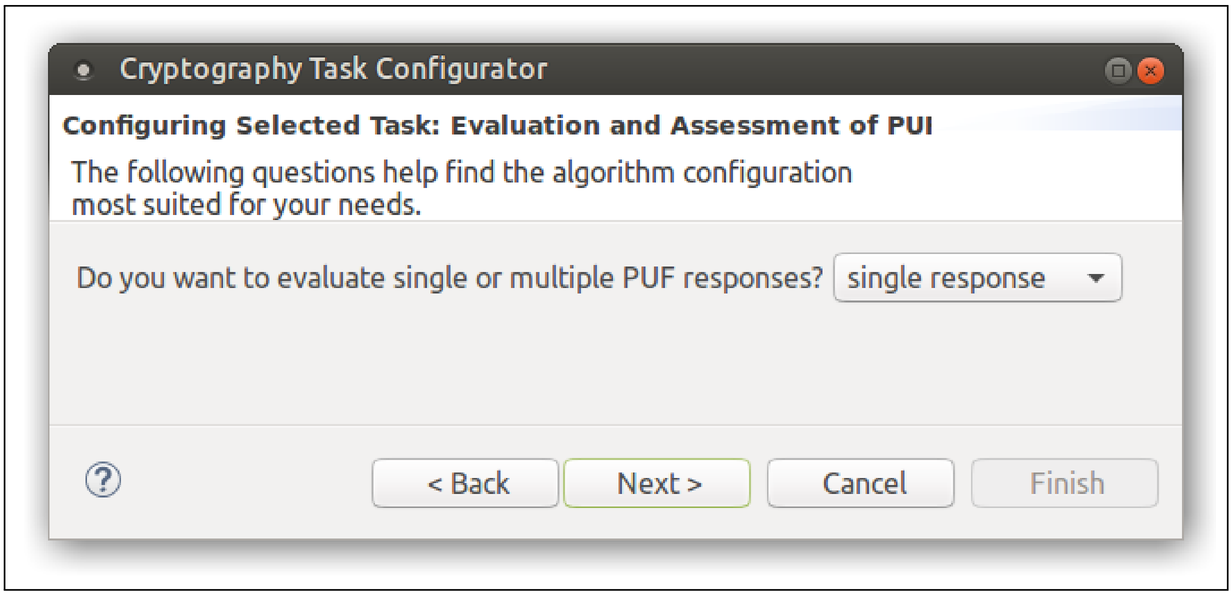
Once the toolkit.class, NativeUtils.class and libtoolkit.so are compiled they are packaged as a JAR file which is exported in the CogniCrypt as a Java Archive.

3.5.2 CogniCrypt Clafer Model

CogniCrypt was implemented as an Eclipse Plugin to support the Java developers to enable easier use of the Cryptographic APIs [18]. It helps the developers to:

- Generate secure implementations for cryptographic tasks like Data Encryption.
- Analyze the Java code and alerts for insecurely used cryptographic APIs.

CogniCrypt supports various crypto tasks and for the PUF toolkit to be made available on the list, a new task “Evaluation and Assessment of PUF responses” was implemented. After selecting the task a few high-level questions like if the developer wants to evaluate a single PUF response or multiple PUF responses, as shown in Figure 3.19, must be answered.


Cryptography Task Configurator

Configuring Selected Task: Evaluation and Assessment of PUI

The following questions help find the algorithm configuration
most suited for your needs.

Do you want to evaluate single or multiple PUF responses? single response

Figure 3.19: An example of high-level questions asked by the CogniCrypt for the Evaluating PUF responses

Once all the questions are acknowledged, based on their answers the CogniCrypt presents a list of the algorithms that are eligible for selection for example, if the developer answered he wants to evaluate PUF response from “multiple devices” then Inter-Hamming Distance is only shown in the drop-down since it is the only metric in the toolkit that takes more than one directory as input. In this case, the CogniCrypt generates a Java boilerplate code that shows the proper usage of the Inter-Hamming Distance function of the toolkit with directories (bounded between 2 and 99) as parameters to the API.

```
abstract Metric
        description -> string
        fileMode -> FileMode
        devices -> Devices

Hamming_Distance : Metric
        [description = "hamming␣distance"]
        [fileMode = File || fileMode = Folder]
        [devices = Single]
```

Listing 3.4: Code snippet of a sample metric “Hamming Distance” in the PUF Clafer model

To achieve the above-mentioned functionality, CogniCrypt employs a Clafer-based model which is a lightweight modeling language that facilitates a mix of class and feature modeling [18, 39]. It supports constraint solvers that can generate instances of a model and satisfy its constraints [18]. The answers to the high-level questions determine the constraints on the model which is composed of different elements known as clafers. For the PUF toolkit each evaluation metric was implemented as clafer element and named as “Metric” in the model, one of the sample Metric is presented in listing 3.4.

The Hamming distance extends the abstract Metric class that has two enums “filemode” and “devices”, and a string element “description” as data members. These elements are initialized in the *Hamming_distance* class, lines 7-9 ensure the Hamming distance can only be evaluated for a file or an entire folder, such conditions are applied to other metrics too, for example, Inter-Hamming distance must always be evaluated for multiple devices.

After the Clafer model is finalized and implemented, auto-generation of the Java code, that shows the developers the correct usage of the PUF toolkit functions, is achieved by the XSL stylesheet. For the PUF Toolkit the already implemented stylesheet in the CogniCrypt was enhanced to support the new functionality. Based on the answers by the developer to the high-level questions the algorithm type or the metric is set in the Clafer model which can be used by the <xsl: if> statements of the relevant XSL stylesheet to match and generate appropriate usage sample code.

4 Evaluation

The evaluation chapter covers the analysis of the integrated code and the improvements done to the toolkit and the evaluation of CogniCrypt interface to the toolkit. This analysis includes appraisal of the Design Guidelines and interpretation of the results of the metrics of the PUF Toolkit for PUF response evaluation. The results of the measurements related to CPU usage and time requirements are presented as histograms for better assessment of the number of CPU cycles used by the different metrics of the PUF Toolkit. For fuzzy extractor techniques, the appraisal of the time requirement of the preprocessing with respect to the parameter m and a PUF response length analysis of the available BCH modes is given.

4.1 Design Guidelines

The user interface should be intuitive and easy-to-use which ensures user-satisfaction and effective and efficient use of the tool. While extending the toolkit and implementing new metrics, special consideration was given to the user interface design. The existing toolkit followed the standard guidelines as listed in [44] and are summarized below:

- **Simple and natural dialogue:** User must be provided with information related to the task such that the comprehension of the task is natural. The mapping of the user's mental model to the workflow of the toolkit has to be accurate. To satisfy this property the sub-menus are in a top-down structure i.e. important options are always in the upper section of each menu. These options are also mandatory e.g. the settings like offsets and file names must be set first before progressing to the calculation. The sub-menu items after calculation are optional, the toolkit menu follows a natural top-down workflow that is intuitive to the user and thus represents a simple and natural dialogue.

- **Speak the user's language:** The intended user group of the toolkit is assumed to be familiar with some technical vocabulary. Translating each and every instruction into layman's language distorts the meaning and is not always helpful. So all the dialogues are written as clear instructions with additional examples given to ensure a successful interaction between the user and the program.

- **Minimize the user's memory load:** To reduce retyping and user load, all the chosen settings are displayed in each menu. The long inputs like the paths to the different folders can be avoided via explained shortcuts. Common settings like offset need to be set only once and are applied to all the sub-menus. Additionally, brief texts and guidelines to relax the short-term memory of the user are provided that promote a clear design and visual clarity.

- **Be consistent:** The layout and positioning of the menu items and design concept is consistent throughout and wherever possible exact same words, menu entries, and settings options are used for the user to recognize a pattern and prediction of the actions. The control behavior is also consistent, meaning the upper area always shows the important options as mentioned in the text above.

- **Provide Feedback:** The information header in the toolkit informs the user where he/she is in the menu hierarchy. The feedback and the input area provide with essential information about the result. Not only for errors, but feedback for each performed action and its status is displayed.

- **Provide clearly marked exits:** The "exit" or "back" is the lowest menu option in each sub-menu level. This facilitates user to navigate through the hierarchical structure of the menu or to exit the toolkit.

- **Provide shortcuts:** The selection of each menu entry is implemented as a shortcut represented by the corresponding number in front of the option. This helps in fast and spontaneous navigation through the menus.

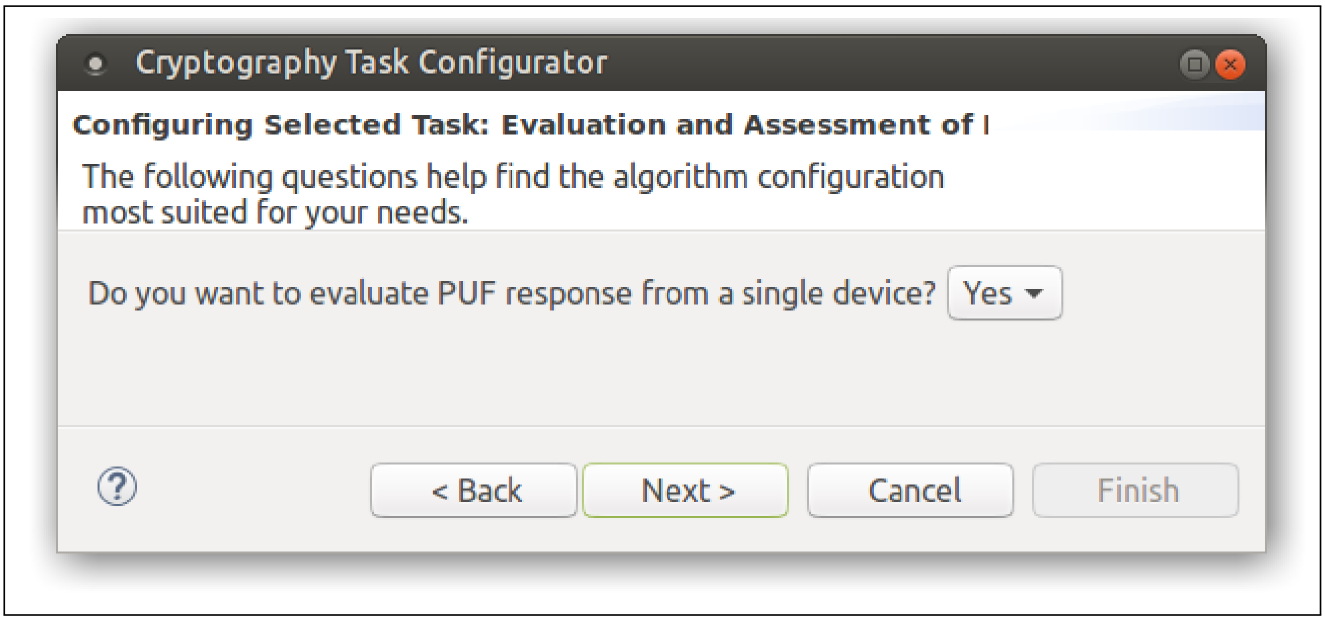
- **Good error messages:** Whenever user input is incorrect or toolkit is in a faulty state, the reason for the error and information to recover from the erroneous state is given in the feedback area. The instructions are simplified and only relevant information to the error with as much detail as possible is presented thereby assisting user for a fast recovery.

- **Prevent errors:** The implementation has a two-level check for handling the user inputs to ensure that only legitimate inputs will be processed. Further information related to the input format and representative examples of such sample inputs with brief helper text are displayed to the user for him/her to avoid errors. Exhaustive error checking is performed before calculating the metric to assure that all relevant data and settings are set and valid, so as to prevent the program from running into an undesired state and crashes.

4.1.1 CogniCrypt User Interface

The CogniCrypt user interface consists of dialog boxes that ask simple questions to help the Java developers call the functions of the toolkit. The above-mentioned design guidelines were considered for the development of the UI of CogniCrypt. They not only help to implement a secure Java code for the invocation of the toolkit functions but also assist the developers in getting familiar with the pattern of the toolkit and structure of the PUF responses. For example, one major question asked by the CogniCrypt to the developer is if he/she wants to evaluate a particular PUF type using a single PUF instance or multiple PUF instances (folders), for multiple folders the CogniCrypt selects Inter-Hamming Distance. Such a question dialog box is represented in Figure 4.1 which shows the simple and intuitive design of the User Interface.

During the testing of the CogniCrypt interface, a critical bug was discovered and resolved. If the absolute path of the file name exceeded a certain number of characters then the Toolkit crashed with a segmentation fault. The root cause was the generation of white space by calling the C++ API "string" that takes *length* as one of the argument. But when the absolute path is very big the length becomes negative and


Cryptography Task Configurator

Configuring Selected Task: Evaluation and Assessment of I

The following questions help find the algorithm configuration most suited for your needs.

Do you want to evaluate PUF response from a single device? Yes

Figure 4.1: User Interface of the CogniCrypt, represented as a dialog box that asks simple questions for the evaluation of the PUF metrics.

the toolkit throws a runtime error. This bug was fixed by increasing the limit for the acceptable path length.

To further enhance the developer experience and improve the UI a survey was taken and the suggestions of the Java developers were integrated into the CogniCrypt. For details of the survey please refer to the research paper [49].

4.2 Interpretation of the results for the extended metrics

The interpretation of the core metrics that were implemented already in the toolkit is explained in detail in [20]. This section presents the interpretation of the extended metrics and also touches upon the Hamming weight interpretation since it is a base metric upon which other PUF evaluation metrics are built.

- **Hamming Weight:** This metric was introduced in section 3.2 and is used to ensure that a PUF response is not biased towards 0 or 1. For the result to be unbiased, the PUF response evaluation for the Fractional Hamming weight metric between two files should be as close to 50% as possible. Slight deviations are acceptable.

- **Jaccard Index:** This metric is also sometimes referred to as Jaccard similarity coefficient. As its alternate name suggests the Jaccard index measures similarity between two files so the Jaccard Index should be as close to 1 or the $JaccardIndex * 100$ should be as close to 100% as possible for two files to be similar.

- **Intra-Jaccard Index:** The general use case for evaluation of the Intra-Jaccard Index is when evaluating PUF responses from a single PUF instance with the same particular challenge. Ideally,

Metric	Instance(s)	Challenge(s)	Ideal result	Deviations	Interpretation
Hamming weight	-	-	(fractional) 50%	minimal acceptable	PUF response unbiased
(Shannon) entropy	-	-	1	minimal acceptable	Randomness of PUF response
Intra-Hamming distance	same	same	(fractional) 0%	minimal acceptable	Reliability
	same	different	(fractional) 50%	minimal acceptable	Uniqueness
Inter-Hamming distance	different	same	(fractional) 50%	minimal acceptable	Uniqueness
Min-entropy	same	same	0	minimal acceptable	Robustness / Reliability
	different	same	(fractional) high	minimal acceptable	Uniqueness
	same	same	(fractional) high	minimal acceptable	Noise extraction possible
Jaccard Index	-	-	100%	minimal acceptable	PUF response similarity
Intra-Jaccard Index	same	same	100%	minimal acceptable	Reliability
	same	different	0%	minimal acceptable	Uniqueness
Inter-Jaccard Index	different	same	0%	minimal acceptable	Uniqueness

Table 4.1: Extended version of the table showing Interpretation of the results regarding the metrics in dependency to the utilized use case.

the PUF responses from the same PUF instance and same challenge must be identical, but errors are introduced due to external factors like operating temperature change, voltage fluctuation etc. The PUF responses from the same PUF instance are stored in a particular directory and evaluation for intra-Jaccard Index is done for this entire folder. The result for each comparison should be close to 1 which establishes similarity between files. Note that slight deviations are tolerable. Another use case for this metric is to gain insight in the uniqueness of the PUF responses from a single PUF instance when it is stimulated with different PUF challenges. In such a case the resultant PUF responses should be different from each other depending on the PUF challenge. Note this evaluation only holds ground if the PUF instance offers a challenge space that is greater than one. In this case the Jaccard Similarity Index should be close to 0 which establishes exclusiveness amongst PUF responses.

- **Inter-Jaccard Index:** Contrary to the Intra-Jaccard Index where evaluation of the PUF responses from a single device is done, Inter-Jaccard Index evaluates PUF responses from multiple devices. As stated above each PUF instance stores the responses in a specific directory, so multiple directories represent multiple PUF instances containing PUF responses which ideally must be similar within the same directory but must be unique and exclusive amongst other directories. Therefore the inter-Jaccard Index for the same PUF challenge to different PUF responses from separate PUF instances should be as close to 0 as possible to ascertain uniqueness. Slight deviations from this ideal number are always acceptable.

For completeness table 4.1 summarizes the result interpretation of the metrics with respect to the utilized use case. The table also contains metrics from the original toolkit that are not explained in detail to avoid redundancy. To know more about the metrics in the previous version of the toolkit refer to [20].

Time requirements: To test the new metrics of the toolkit an analysis of the time taken by the calculation APIs for the PUF response evaluation of multiple instances was performed. The time taken by the user to input the settings was skipped, the measurement of the time was done using the standard “clock()” API of the GNU C library which is used to determine the process time. For testing, SRAM PUF responses from Texas Instrument (TI) Stellaris LM4F120XL LaunchPad Evaluation Kits with a size of 32 Kilobytes and an offset of 400 bytes from the beginning was used. The analysis was performed on an Intel i3 quad-core CPU with 2.00 GHz and 3 GB RAM. The results of the time requirements for Intra-Jaccard Index and Inter-Jaccard Index are visualized in Figures 4.2 and 4.3 respectively. Figures 4.4 and 4.5 show the average percentage of CPU usage for Intra and Inter-Jaccard Index with the same parameters. The RAM usage for the Intra- and Inter-Jaccardi Index was constant to be around 0.1% and thus is not represented. Finally, the time analysis for the Jaccard Index is not represented since it is constant around 0.7 milliseconds.

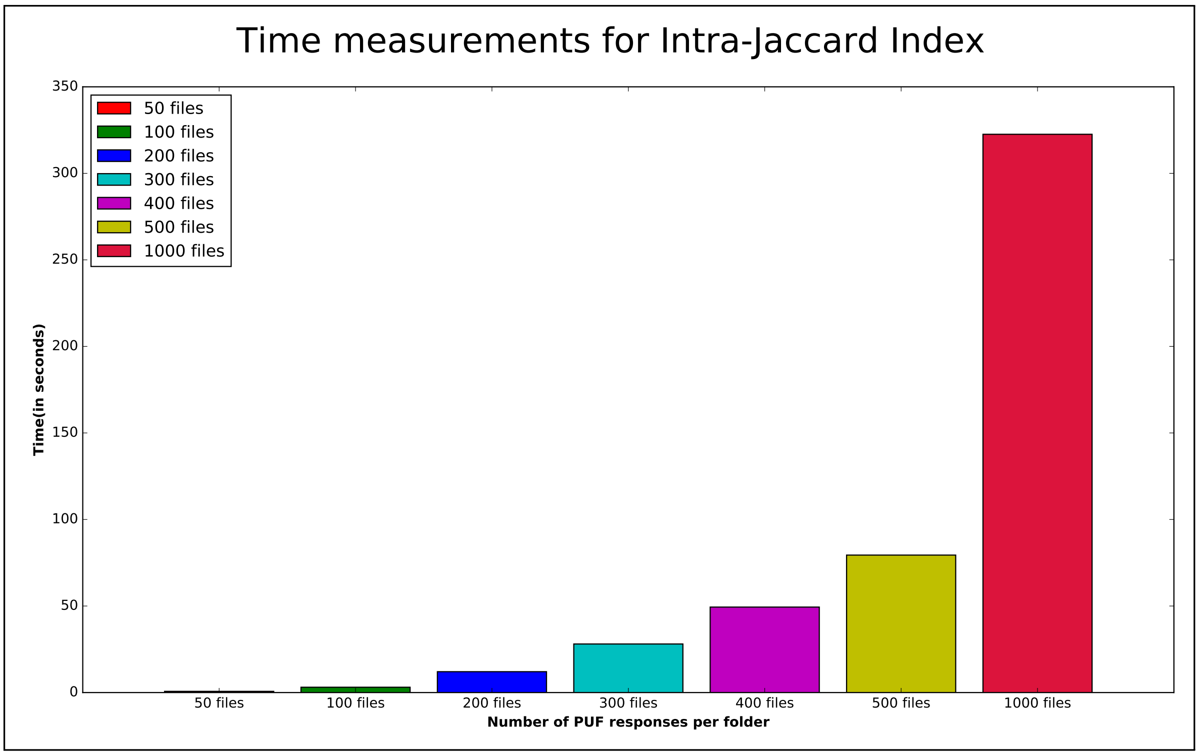


Figure 4.2: Time requirement measurements of the *Intra-Jaccard Index* with PUF response of 32 kilobytes.

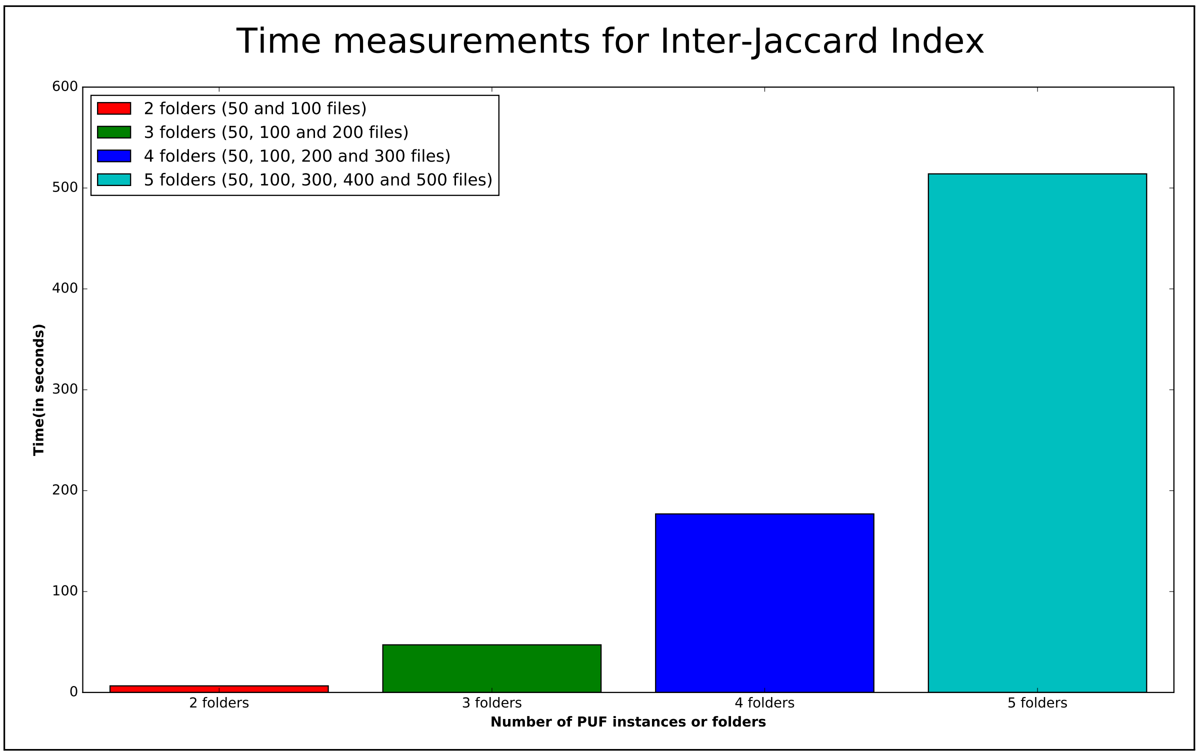


Figure 4.3: Time requirement measurements of the *Inter-Jaccard Index* with each PUF response of size 32 kilobytes.

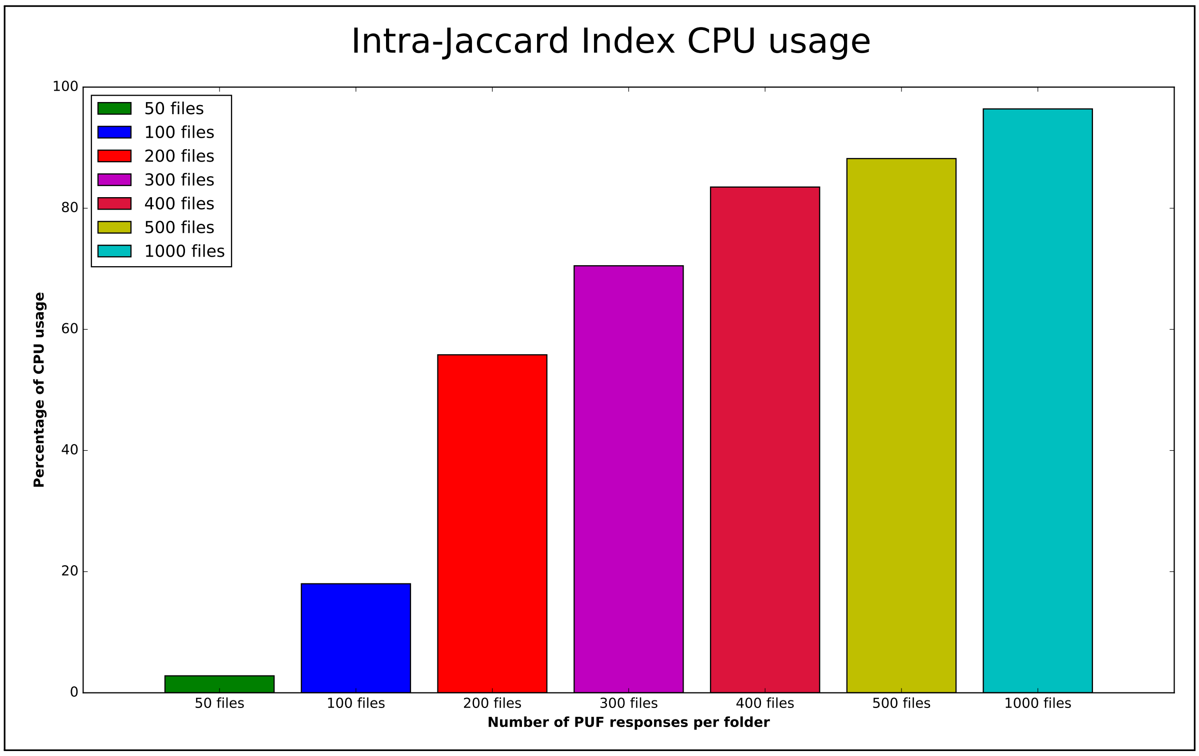


Figure 4.4: Visualization of CPU usage by the *Intra-Jaccard Index* for folders containing multiple PUF responses of size 32 kilobytes.

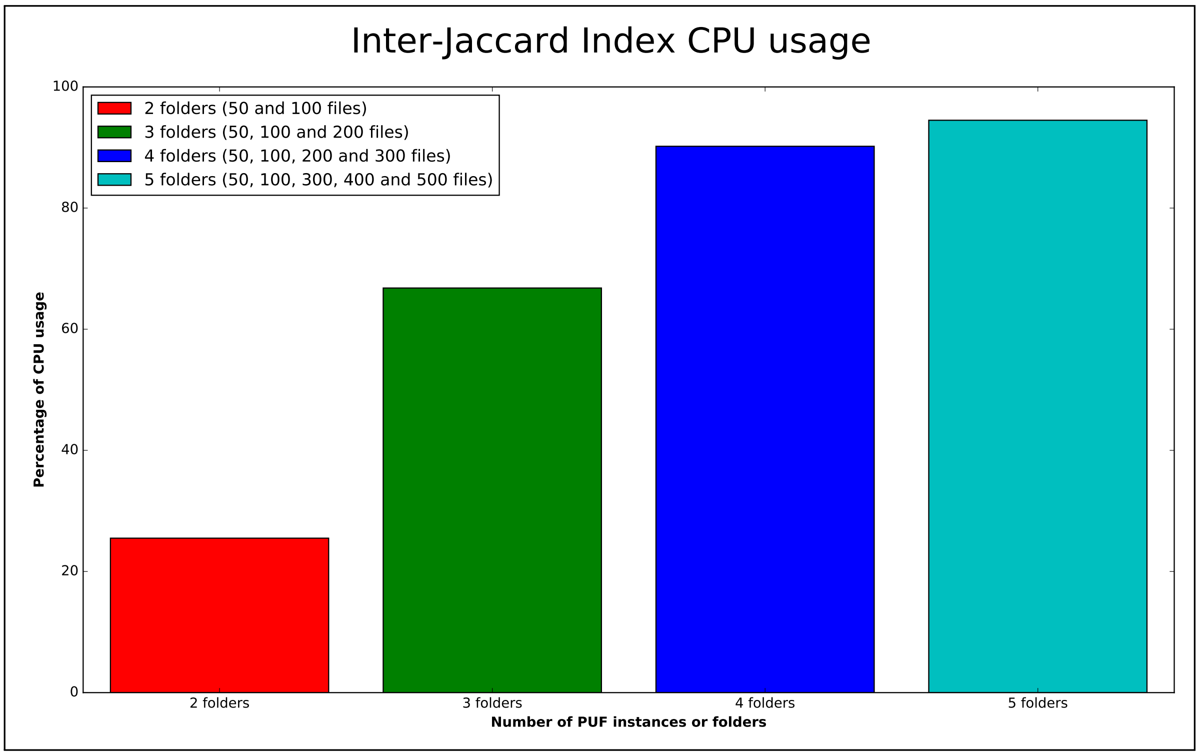


Figure 4.5: Visualization of the CPU usage by the *Inter-Jaccard Index* between mutiple folders containing multiple PUF responses of size 32 kilobytes.

4.2.1 Evaluation of the Fuzzy extractor

To evaluate the fuzzy extractor we compare both the coding techniques, Golay code and BCH code used in the fuzzy implementation. Both of them are assisted with linear repetition algorithm and majority voting procedure. The error-correction capability t of Golay code is limited to a maximum of three-bit errors for a message length of *12 bits*. Both codes are linear codes so they take the same parameters (n, k, d), where n is the length (in bits) of the encoded codeword, k is the length (message length) of the input secret/message that is fed to the encoder and d is the minimal Hamming distance from which the error correcting capability is derived using the formula $t = \lfloor (d-1)/2 \rfloor$. The Golay code uses a lookup table to encode the *12 bits* of message to *23 bits* of codeword. In case there are more than three bits of errors introduced while processing a PUF response block (due to external noise and environmental factors), then the decoding lookup table for Golay code is not capable to correct errors which leads to an undesired state. To avoid this behavior, the user is presented with an error message that it is out of Golay code's scope to correct so many errors and to use BCH coding instead.

The BCH coding implementation, on the contrary, corrects as many bit errors as possible in each codeword. If errors exceed the correction capability t of the BCH code then only t errors are corrected and remaining errors are ignored. This avoids the undesired behavior unlike the Golay code and the information about the detected but not corrected errors is displayed to the user which enhances the robustness of the fuzzy extractor. In addition, the parameters of the BCH code are highly configurable and can be adjusted to the noise level that introduces the errors. The pre-evaluation of the PUF responses from the PUF instances can be utilized to predict the needed error-correction capability and the parameters of the BCH code can be adjusted accordingly to ensure a reliable working procedure. Table 4.2 lists the possible parameters that are valid for the BCH(n, k, d) code.

4.2.2 Pre-evaluation Relevance

PUF response length analysis: To select the optimal BCH mode parameters it is important to determine the length of the PUF response which also forms a part of the pre-evaluation phase. The PUF response length is based on the size of the input secret and parameters k and d of the BCH code along with the Linear repetition factor selected for encoding. The PUF response length that satisfies the BCH code selection parameters can be verified using Equation 4.1.

- The size of the input secret is N bits, the BCH parameters correspond to the chosen BCH(n, k, d) mode and the linear repetition factor is denoted with LR (7 or 15). The required length of the PUF response is represented with the symbol *RequiredPUFLength* [20].

$$RequiredPUFLength \geq \left\lceil \left(\frac{\left(\left\lceil \frac{N}{k} \right\rceil * n \right)}{8} \right) \right\rceil * LR \tag{4.1}$$

Common BCH codes of order less than 2^{10}

n	k	d	t	n	k	d	t	n	k	d	t	n	k	d	t
7	4	3	1	127	8	63	31	511	484	7	3	511	166	95	47
15	11	3	1	255	247	3	1		475	9	4		157	103	51
	7	5	2		239	5	2		466	11	5		148	107	53
	5	7	3		231	7	3		457	13	6		139	109	54
31	26	3	1		223	9	4		448	15	7		130	111	55
	21	5	2		215	11	5		439	17	8		121	117	58
	16	7	3		207	13	6		430	19	9		112	119	59
	11	11	5		199	15	7		421	21	10		103	123	61
	6	15	7		191	17	8		412	23	11		94	125	62
63	57	3	1		187	19	9		403	25	12		85	127	63
	51	5	2		179	21	10		394	27	13		76	171	85
	45	7	3		171	23	11		385	29	14		67	175	87
	39	9	4		163	25	12		376	31	15		58	183	91
	36	11	5		155	27	13		367	33	16		49	187	93
	30	13	6		147	29	14		358	37	18		40	191	95
	24	15	7		139	31	15		349	39	19		31	219	109
	18	21	10		131	37	18		340	41	20		28	223	111
	16	23	11		123	39	19		331	43	21		19	239	119
	10	27	13		115	43	21		322	45	22		10	255	127
	7	31	15		107	45	22		313	47	23	1023	1013	3	1
127	120	3	1		99	47	23		304	51	25		1003	5	2
	113	5	2		91	51	25		295	53	26		993	7	3
	106	7	3		87	53	26		286	55	27		983	9	4
	99	9	4		79	55	27		277	57	28		973	11	5
	92	11	5		71	59	29		268	59	29		963	13	6
	85	13	6		63	61	30		259	61	30		953	15	7
	78	15	7		55	63	31		250	63	31		943	17	8
	71	19	9		47	85	42		241	73	36		933	19	9
	64	21	10		45	87	43		238	75	37		923	21	10
	57	23	11		37	91	45		229	77	38		913	23	11
	50	27	13		29	95	47		220	79	39		903	25	12
	43	29	14		21	111	55		211	83	41		893	27	13
	36	31	15		13	119	59		202	85	42		883	29	14
	29	43	21		9	127	63		193	87	43		873	31	15
	22	47	23	511	502	3	1		184	91	45		863	33	16
	15	55	27		493	5	2		175	93	46		858	35	17

Table 4.2: BCH codes generated by primitive elements of order less than 2^{10} and the corresponding parameters *n*, *k* *d* and *t*.

The pre-evaluation is inevitable for an effective and efficient development of PUF-based security approaches and it should be included in the development cycle of the PUF-based security algorithms. The pre-evaluation of the TI Stellaris LM4F120XL Launchpad Kit showed that the complete SRAM from this hardware cannot be considered as a PUF, because of the boot process which affects the randomness of the first 400 bytes and renders these bytes not to be considered a part of the PUF response. Without pre-evaluation such exceptional behavior might not be discovered which can result in an insecure implementation of the PUF-based security approaches. For determining the required error-correction capability of the BCH code, the pre-evaluation of the PUF instance is critical for the selection of the optimal parameters. These two scenarios illustrate that the pre-evaluation must be considered as a decisive factor in the development of the PUF-based security approaches.

4.3 Time requirement analysis

The BCH code is highly configurable that means the parameters m, n, t can be constructed in various possible ways. For time requirement analysis the preprocessing time for the generation of a BCH mode was taken into account. Depending on the selected parameters, specifically m, the preprocessing can create some timing overhead. Since this process is inevitable for both encoder and decoder, the timing overhead is considered with respect to the selection of the parameters, especially since the adaptation of the fuzzy extractor is intended for hardware machines with limited resources. For testing, all possible m (in the range [4,14]), the corresponding maximum n ($2^{n-1}-1 < n <= 2^n - 1$) and the highest possible error-correction capability t were chosen. The time requirement analysis was performed on an Intel i3 quad-core CPU with 2.0 GHz and 3 GB RAM. The results of the time measurements for the BCH mode pre-processing are visualized in Figure 4.6.

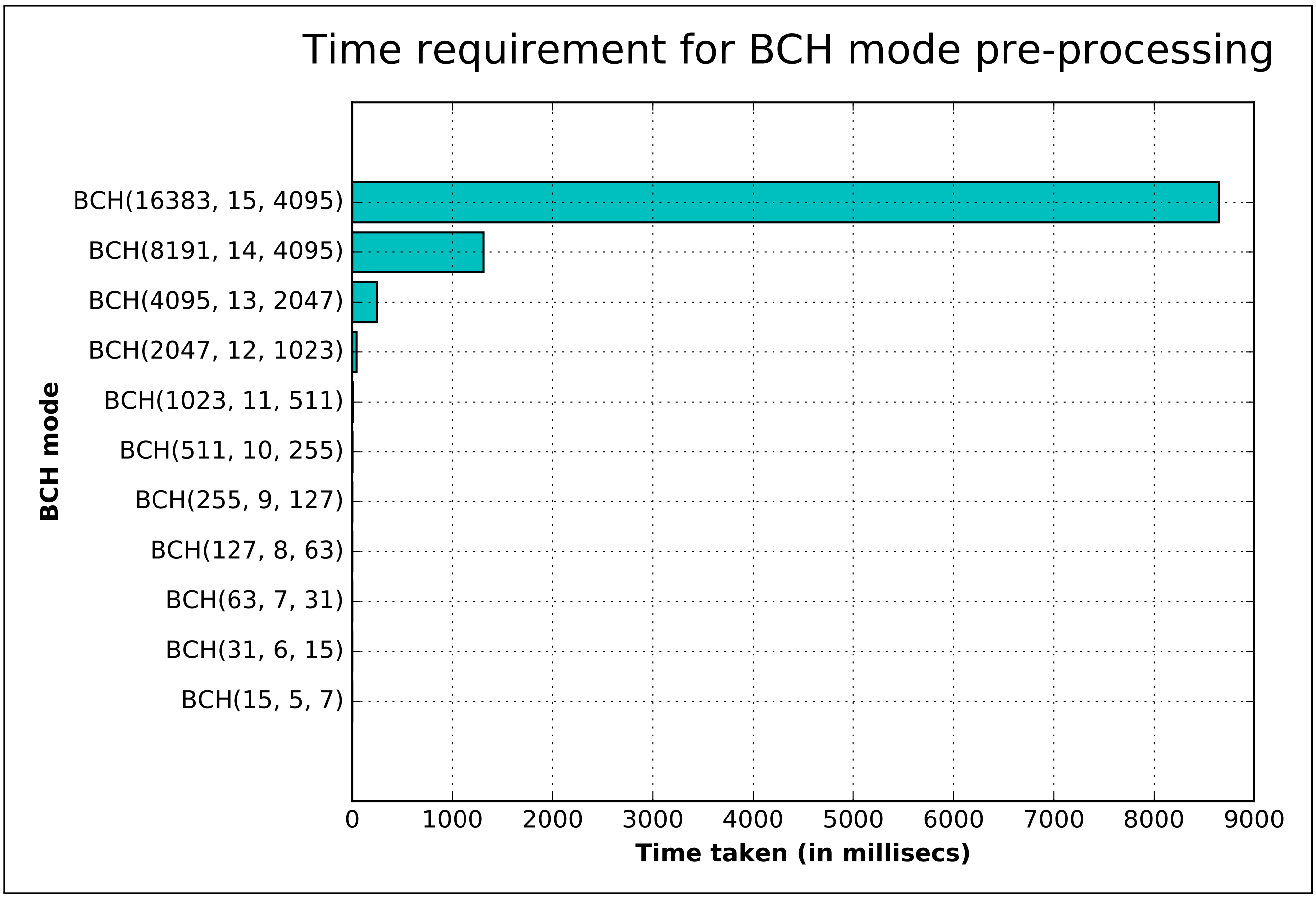


Figure 4.6: Time requirement measurements for the BCH mode pre-processing in relation to parameter m.

The Golay code is a static (23, 12, 7) code where the user is not required to input any parameters. The pre-processing time of the Golay code remains constant and is not a timing overhead, so it can be ignored. The time required for encoding and decoding steps depends on the size of the file that is to be

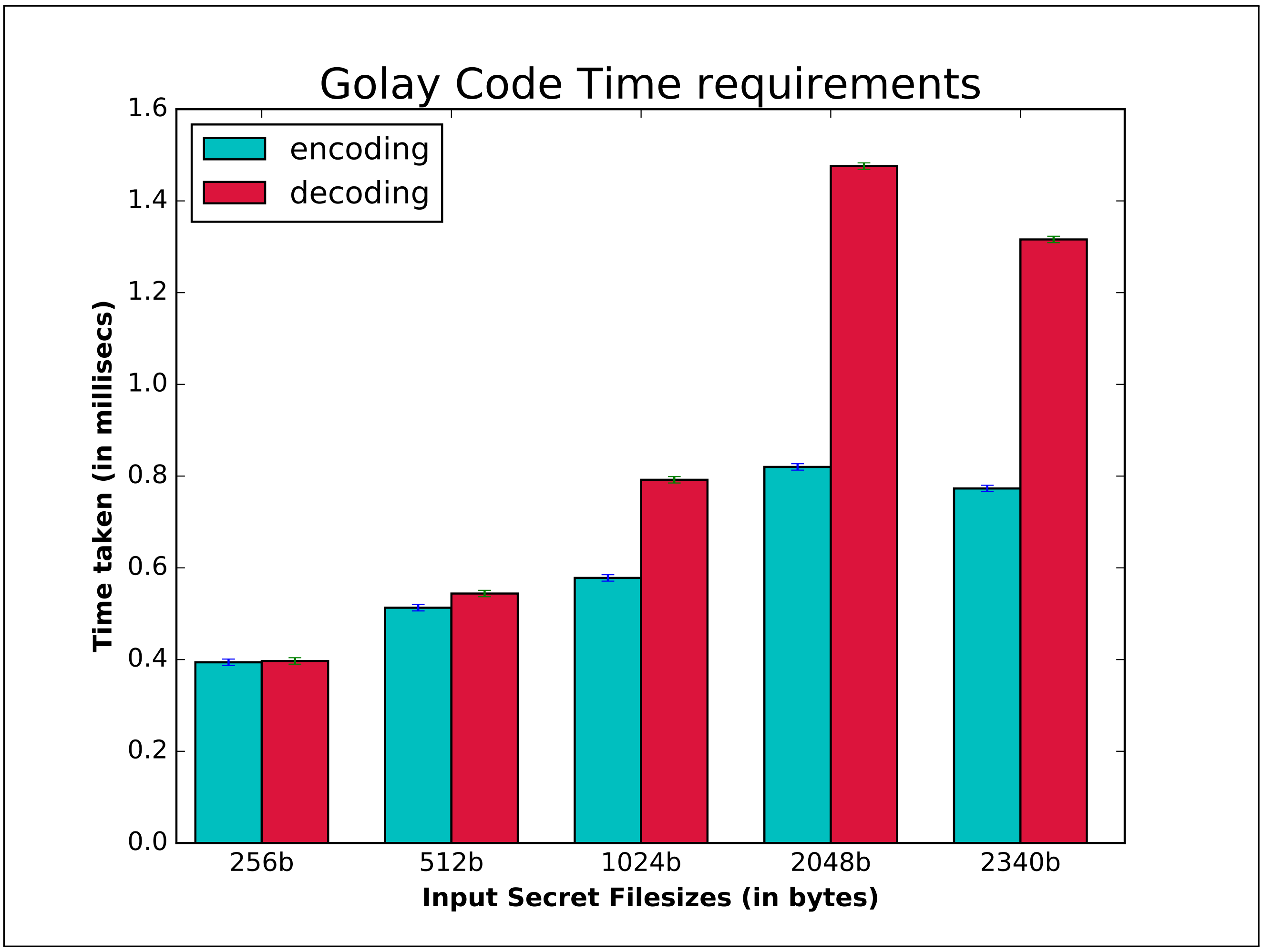


Figure 4.7: Time requirement measurements for the Golay encoding and decoding for varying filesizes

encoded/decoded. For this analysis, we used a PUF response from TI Stellaris LM4F120XL LaunchPad Evaluation Kits and the size was varied with the help of offsets, the original size of the PUF response was 32 Kilobytes. Figure 4.7 illustrates the encoding and decoding analysis of Golay code with different file sizes as a bar graph.

5 Conclusion

5.1 PUF Toolkit

In this work, an already developed software-based toolkit named *PUF Toolkit* was improved upon and extended. The implementation boosts the easy-to-use console user interface and provides the implementation of new common and proven metrics for the evaluation of the PUF responses. (To be specific the following metrics were added to the toolkit: *Jaccard Index (Intra-Jaccard Index, Inter-Jaccard Index)* and *Hamming Distance*. The implementation of the metrics favored an object-oriented approach and hence the toolkit was designed in C++ and to ensure an effective and intuitive working with the toolkit the user interface was derived from the nine main design guidelines by Nielsen and Molich [44]. To support the designers and researchers in the evaluation of the PUF responses, the visualization of the metrics results was optimized with respect to PUF responses. The toolkit's functionality was enhanced to also skip bytes from the end of the PUF response (using an offset) and menu items were rearranged for better navigation. Apart from these optimizations, the visual layout of the result file was improved to contain new parameters like offset from the end of the file, fractional Jaccard Index and Fractional Hamming Distance. A consistent development cycle and verification phase was performed for each metric implementation. The evaluation explained the interpretation of the results for the extended metrics and summarized already implemented metrics in a table. The user interface design guidelines were outlined and a resource-friendly working of the toolkit was presented and finally the testing of the toolkit was done with a large set of samples and the data was recorded and visualized as bar graphs.

5.2 Extension and Integration of Fuzzy Extractor

The already implemented BCH code-based fuzzy extractor was added and integrated to the PUF Toolkit. The implementation of the fuzzy extractor was realized in C++ and it achieves a PUF-based secure key storage on hardware devices with limited resources. The integration part covered the combining of the data structures of the already implemented BCH code (based on Morelos-Zaragoza Encoder/Decoder for binary BCH codes in C [45]) to the structure of the PUF Toolkit. Intensive code review was done to correct some errors in the original Morelos-Zaragoza's implementation to ensure that the encoder and decoder functionalities work even with large values of parameter m. The erroneous state recovery, error correction capability and other functionalities of the PUF Toolkit were merged to the fuzzy extractor. The PUF BCH Encoder represents the enrollment phase of the fuzzy extractor for the realization of PUF-based secure storage. The BCH code is highly configurable and the code parameters (n, k, d) along with error correcting capability t and parameter m can be adjusted according to user needs and PUF properties of the desired PUF instance. The BCH codes create code words which are processed by a linear repetition code before the result is "one time encrypted" with an XOR operation with the PUF response. The Encoder menu adheres to the above-mentioned design guidelines and provides an easy-to-use and well-structured user interface. The PUF BCH decoder implementation is the counterpart of the PUF BCH Encoder and represents the second (reconstruction) phase of the fuzzy extractor. The reconstruction phase of the fuzzy extractor combines XOR operation with the PUF response and subsequently executes Majority Voting algorithm followed by the BCH decoder functionality to retrieve the original secret data. The integration procedure of the BCH decoder to the toolkit was similar to the integration procedure of the BCH encoder to the toolkit. The sub-menu of the decoder is intuitive and a well-structured user interface that is inspired by the BCH encoder menu. The factors of the linear repetition code and the majority vote can take the value 7 or 15. For evaluation, the BCH fuzzy extractor was tested with different code parameters and the time requirements for the pre-processing step that creates a timing overhead were recorded and visualized as a time chart. The recovered file was cross-checked with the original secret making sure they were identical.

The toolkit was extended with the Golay-based Fuzzy extractor. The original implementation of the Golay code was in a very basic format which hardcoded the input file, PUF response and other data structures, and had a very restrained functionality with no support for offsets, error recovery and user files input. All these functionalities were first added to the base version and then both the encoder and decoder parts of the Golay-based Fuzzy extractor were integrated to the PUF Toolkit. The Golay encoder forms the generation phase of the fuzzy extractor. Similar to BCH encoder, it performs an XOR operation of the secret key with the PUF response and then applies linear repetition algorithm followed by Golay encoding to generate the helper data that is used in the decoding phase to recover the encoded secret key. But unlike BCH code, the parameters of the Golay code are fixed to (23, 12, 7); due to this limitation the recovery of large files that resulted in a helper data bigger than the PUF response is not possible and the user is suggested to use BCH-based fuzzy extractor instead. The encoder also appends the configuration settings like offsets, input filesize and linear repetition factor to the helper data. The Golay decoder constitutes the reconstruction phase of the fuzzy extractor. It reads the saved configuration settings from the end of the helper file and then proceeds to an XOR operation of the helper data with the PUF response followed by the Majority Voting procedure with the same factor as the one used in by the linear repetition algorithm in the encoder part. Lastly based on the settings and code parameters (n, k, d) the decoder creates a decoding table and restores the original secret from the helper data, thus completing the PUF-based secure storage recovery. The menus for both encoder and decoder are derived from the BCH-based fuzzy extractor and adhere to the same design guidelines as the PUF Toolkit, thereby making its user interface easy-to-understand and well-structured. The evaluation of the Golay code was done by testing the encoder and decoder for their time requirements and CPU usage with different PUF response sizes by adjusting the offsets and represented as a bar graph for analysis. A final check was performed using the “diff” utility to ensure that the recovered file does match the original input secret key.

5.3 Integration to the CogniCrypt

The metrics of the toolkit were made available to the Java developers for evaluation of PUF responses via the opensource library CogniCrypt. Using this library the developers with no prior knowledge about the PUF Toolkit can develop programs in Java by calling the functions of the toolkit. This was made possible by developing a separate module to interface the C++ code with the Java runtime via the Oracle Standard technology Java Native Interface. JNI is a programming framework that enables the Java virtual machine to call the functions of the native applications and libraries written in other languages like C, C++, and assembly. The JNI enabled toolkit is compiled as a shared library and packaged as a JAR (Java Archive) to be included in the Java project as an external library for development. The CogniCrypt is an opensource library [18] based on the Clafer lightweight modeling language [39] and was developed to assist Java developers for secure and correct usage of the cryptographic APIs. It asks simple questions from the developers to select the appropriate algorithm based on the answers and generate a boilerplate code that shows the sample usage of the cryptographic API. For this thesis the specific metrics of the PUF Toolkit namely: *Hamming Weight, Intra-Hamming Distance, Inter-Hamming Distance, (Shannon) Entropy, min-Entropy* and *median and average* were implemented as clafer algorithms and added to the CogniCrypt model. The graphical user interface of the CogniCrypt is based on the advice taken from a survey, done with Java developers as the audience, on how to improve the usability of cryptographical APIs. Thus making it simple, intuitive and easy to use.

The extended and improved *PUF Toolkit* integrated with the *Fuzzy Extractor* functionality and interfaced to the *CogniCrypt* opensource library, contributes to and accelerates the development of new security approaches. On one hand, it is helpful in the evaluation of PUFs in the context of security applications and on the other hand, it provides its functionalities to be used in a Java project and can be further used to realize new or enhanced security techniques that are based on PUFs. Although there might be additional aspects that could be optimized in the future to further increase the PUF Toolkit application area, this thesis enhances the scope of the initial *PUF Toolkit* to assist the researchers and developers in the evaluation of PUF instances.

For future projects, a survey of the *PUF Toolkit* by the intended user group can be carried out for getting useful optimization information and user feedback. This information can be about the workflow procedure, the result visualization and other data to improve the interaction of the user with the Toolkit.

The development of a graphical front-end for the PUF Toolkit could be a great step to increase the user-friendliness and contribute more to the efficient use of the PUF Toolkit. Specifically, with regard to file selection mechanism, a graphical UI can present an easy alternative to the user rather than typing.

There can be new metrics added to the Toolkit that will further enhance the Toolkit capability to evaluate PUF responses and instances, giving one more dimension to the functionality of the PUF Toolkit. A possible selection of the metrics can be as follows:

- The *Context Tree Weighting (CTW)* is a fascinating metric that can be used to evaluate the randomness of the PUF response. The compression ratio of the CTW can be exploited to gain information about the entropy of the PUF response.

- The *identification of extremely stable PUF cells* of a PUF instance, especially for SRAM or memory based PUFs, can assist the approaches that require very stable PUF properties. One example can be deriving keys from the start-up values of a PUF response and generating a unique RSA key pair.

- The *identification of extremely unstable PUF cells* of a PUF instance, especially for SRAM or memory-based PUFs, can support the approaches that need an almost perfectly random behavior from a PUF. This means the probability of occurrence of a 0 or 1 in a PUF cell is identical and equal to 0.5 ($p = 0.5$). This identification can be used in the realization of a True Random Number Generator (TRNG).

6.1 Fuzzy Extractor

The fuzzy extractor of the PUF Toolkit can be extended to support new Error Coding Techniques which will allow for a comprehensive evaluation and comparison of the properties amongst the different fuzzy extractor implementations. A possible selection of the error correcting mechanisms that can be added to the fuzzy extractor is:

- *Reed-Solomon Code*, which is derived from the BCH code but belongs to the class of non-binary cyclic error codes.

- *Reed-Muller Code*, which belongs to a class of linear block codes rich in algebraic and geometric structure and also includes Extended Hamming code [50].

6.2 CogniCrypt

The current version of the CogniCrypt-integrated PUF Toolkit does not contain the new metrics eg. Jaccard Index and the fuzzy extractor functionality. These metrics can be interfaced to the Java-based OpenSource library as part of a future endeavor. The UI of the CogniCrypt library can be modified to show an interpretation of the results for different metrics in a color format. For example, red means PUF response is not random, therefore it is insecure to be applied to a PUF-based security mechanism, green means the PUF response is random/unique and can be used in further security applications etc.

List of Figures

List of Tables

Bibliography

[1] IBM Systems. *IBM 4765 PCIe Cryptographic Coprocessor.* Available at `http://www-03.ibm.com/security/cryptocards/pciecc/overview.shtml`.

[2] Blaise Gassend, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 148–160. ACM, 2002.

[3] Vincent Van der Leest, Geert-Jan Schrijen, Helena Handschuh, and Pim Tuyls. Hardware intrinsic security from d flip-flops. In *Proceedings of the fifth ACM workshop on Scalable trusted computing*, pages 53–62. ACM, 2010.

[4] Hyunho Kang, Yohei Hori, Toshihiro Katashita, Manabu Hagiwara, and Keiichi Iwamura. Cryptographie key generation from puf data using efficient fuzzy extractors. In *Advanced Communication Technology (ICACT), 2014 16th International Conference on*, pages 23–26. IEEE, 2014.

[5] Daihyun Lim, Jae W Lee, Blaise Gassend, G Edward Suh, Marten Van Dijk, and Srinivas Devadas. Extracting secret keys from integrated circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(10):1200–1205, 2005.

[6] Boris Škorić, Pim Tuyls, and Wil Ophey. Robust key extraction from physical uncloneable functions. In *International Conference on Applied Cryptography and Network Security*, pages 407–422. Springer, 2005.

[7] Pim Tuyls, Tom Kevenaar, et al. *Security with noisy data: on private biometrics, secure key storage and anti-counterfeiting*. Springer Science & Business Media, 2007.

[8] Ryan Helinski, Dhruva Acharyya, and Jim Plusquellic. A physical unclonable function defined using power distribution system equivalent resistance variations. In *Proceedings of the 46th Annual Design Automation Conference*, pages 676–681. ACM, 2009.

[9] Jorge Guajardo, Sandeep S Kumar, Geert-Jan Schrijen, and Pim Tuyls. Fpga intrinsic pufs and their use for ip protection. In *International workshop on Cryptographic Hardware and Embedded Systems*, pages 63–80. Springer, 2007.

[10] Srinivas Devadas, Edward Suh, Sid Paral, Richard Sowell, Tom Ziola, and Vivek Khandelwal. Design and implementation of puf-based" unclonable" rfid ics for anti-counterfeiting and security applications. In *RFID, 2008 IEEE International conference on*, pages 58–64. IEEE, 2008.

[11] Leonid Bolotnyy and Gabriel Robins. Physically unclonable function-based security and privacy in rfid systems. In *Pervasive Computing and Communications, 2007. PerCom'07. Fifth Annual IEEE International Conference on*, pages 211–220. IEEE, 2007.

[12] G Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th annual Design Automation Conference*, pages 9–14. ACM, 2007.

[13] Ahmad-Reza Sadeghi, Ivan Visconti, and Christian Wachsmann. Enhancing rfid security and privacy by physically unclonable functions. In *Towards Hardware-Intrinsic Security*, pages 281–305. Springer, 2010.

[14] Sebastian Schurig. Public-key kryptographie mit hilfe von sram pufs, 2014.

[15] Steffen Schulz, Ahmad-Reza Sadeghi, and Christian Wachsmann. Short paper: Lightweight remote attestation using physical functions. In *Proceedings of the fourth ACM conference on Wireless network security*, pages 109–114. ACM, 2011.

[16] Joonho Kong, Farinaz Koushanfar, Praveen K Pendyala, Ahmad-Reza Sadeghi, and Christian Wachsmann. Pufatt: Embedded platform attestation based on novel processor-based pufs. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6. ACM, 2014.

[17] Frederik Armknecht, Roel Maes, Ahmad-Reza Sadeghi, Berk Sunar, and Pim Tuyls. Memory leakage-resilient encryption based on physically unclonable functions. In *Towards Hardware-Intrinsic Security*, pages 135–164. Springer, 2010.

[18] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. Cognicrypt: Supporting developers in using cryptography. In *Automated Software Engineering (ASE'17)*. ACM, November 2017.

[19] Steven Arzt, Sarah Nadi, Karim Ali, Eric Bodden, Sebastian Erdweg, and Mira Mezini. Towards secure integration of cryptographic software. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 1–13. ACM, 2015.

[20] Sebastian Schurig. Development of a user interface and implementation of specific software tools for the evaluation and realization of pufs with respect to security applications, 2017.

[21] Roel Maes. Physically unclonable functions: Concept and constructions. In *Physically Unclonable Functions*, pages 11–48. Springer, 2013.

[22] Roel Maes and Ingrid Verbauwhede. Physically unclonable functions: A study on the state of the art and future research directions. In *Towards Hardware-Intrinsic Security*, pages 3–37. Springer, 2010.

[23] DW Bauder. An anti-counterfeiting concept for currency systems. *Sandia National Labs, Albuquerque, NM, Tech. Rep. PTK-11990*, 1983.

[24] John Berry and David A Stoney. The history and development of fingerprinting. *Advances in fingerprint Technology*, 2:13–52, 2001.

[25] Jorge Guajardo, Sandeep S Kumar, Geert-Jan Schrijen, and Pim Tuyls. Brand and ip protection with physical unclonable functions. In *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, pages 3186–3189. IEEE, 2008.

[26] Ravikanth Pappu, Ben Recht, Jason Taylor, and Neil Gershenfeld. Physical one-way functions. *Science*, 297(5589):2026–2030, 2002.

[27] Ravikanth Srinivasa Pappu. *Physical one-way functions*. PhD thesis, Massachusetts Institute of Technology, 2001.

[28] Blaise Gassend, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 148–160. ACM, 2002.

[29] Ahmad-Reza Sadeghi and David Naccache. Towards hardware-intrinsic security. In *Information Security and Cryptography*. Springer, 2010.

[30] Ulrich Rührmair, Heike Busch, and Stefan Katzenbeisser. *Strong PUFs: Models, Constructions, and Security Proofs*, pages 79–96. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[31] Jae W Lee, Daihyun Lim, Blaise Gassend, G Edward Suh, Marten Van Dijk, and Srinivas Devadas. A technique to build a secret key in integrated circuits for identification and authentication applications. In *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*, pages 176–179. IEEE, 2004.

[32] Shrikant S Vyas. Variation aware placement for efficient key generation using physically unclonable functions in reconfigurable systems. 2016.

[33] Daniel E Holcomb, Wayne P Burleson, Kevin Fu, et al. Initial sram state as a fingerprint and source of true random numbers for rfid tags. In *Proceedings of the Conference on RFID Security*, volume 7, page 2, 2007.

[34] Keith M Tolk. Reflective particle technology for identification of critical components. Technical report, Sandia National Labs., Albuquerque, NM (United States), 1992.

[35] Atri Rudra Venkatesan Guruswami and Madhu Sudan. *Essential Coding Theory*. 2017. Available at `https://www.cse.buffalo.edu/faculty/atri/courses/coding-theory/book/`.

[36] *Chapter 3 Linear Codes*. Available at `http://users.math.msu.edu/users/jhall/classes/codenotes/Linear.pdf`.

[37] Robert H Morelos-Zaragoza. *The art of error correcting coding*. John Wiley & Sons, 2006.

[38] Golay Codes. Coding theory massoud malek.

[39] Kacper Bąk, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. Clafer: unifying class and feature modeling. *Software & Systems Modeling*, 15(3):811–845, 2016.

[40] Abhranil Maiti, Vikash Gunreddy, and Patrick Schaumont. A systematic method to evaluate and compare the performance of physical unclonable functions. In *Embedded systems design with FPGAs*, pages 245–267. Springer, 2013.

[41] André Schaller, Anthony van Herrewege, Vincent van der Leest, Tolga Arul, and Stefan Katzenbeisser. Empirical analysis of intrinsic physically unclonable functions found in commodity hardware. 2015.

[42] Stefan Katzenbeisser, Ünal Kocabaş, Vladimir Rožić, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. Pufs: Myth, fact or busted? a security evaluation of physically unclonable functions (pufs) cast in silicon. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 283–301. Springer, 2012.

[43] Frederik Armknecht, Daisuke Moriyama, Ahmad-Reza Sadeghi, and Moti Yung. Towards a unified security model for physically unclonable functions. In *Cryptographers' Track at the RSA Conference*, pages 271–287. Springer, 2016.

[44] Jakob Nielsen and Rolf Molich. Teaching user interface design based on usability engineering. *ACM SIGCHI Bulletin*, 21(1):45–48, 1989.

[45] Morelos-zaragoza, encoder/decoder for binary bch codes in c (version 3.1), 1994 (revised 1997). Available at `www.eccpage.com/bch3.c`.

[46] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.

[47] Java Native Interface. *Java Programming Tutorial*. Avaialable at `https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html`.

[48] Adam Heinrich. *Git repo: adamheinrich/native-utils*. Available at `https://github.com/adamheinrich/native-utils/blob/master/src/main/java/cz/adamh/utils/NativeUtils.java`.

[49] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. " jumping through hoops": Why do java developers struggle with cryptography apis? In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 935–946. IEEE, 2016.

[50] Sebastian Raaphorst. Reed-muller codes. *Carleton University, May*, 9, 2003.