Worcester Polytechnic Institute
Digital WPI

Major Qualifying Projects (All Years) Major Qualifying Projects

April 2009

Authentication Schemes based on Physically Unclonable Functions

Ilan Shomorony
Worcester Polytechnic Institute

Matthew Daniel Dailey
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation
Shomorony, I., & Dailey, M. D. (2009). *Authentication Schemes based on Physically Unclonable Functions*. Retrieved from https://digitalcommons.wpi.edu/mqp-all/1078

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

Project Number: BS2 0804

Authentication Schemes based on Physically Unclonable Functions

A Major Qualifying Project Report

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

____

Matthew D. Dailey

____

Ilan Shomorony

Date: April 30, 2009

Approved:

____

Professor Berk Sunar, Major Advisor

____

Professor William J. Martin, Major Advisor

Abstract

In this project we investigate different hardware authentication schemes based on Physically Unclonable Functions. We start by analyzing the concepts of a fuzzy extractor and a secure sketch from an information-theoretic perspective. We then present a hardware implementation of a fuzzy extractor which uses the code offset construction with BCH codes. Finally, we propose a new cryptographic protocol for PUF authentication based upon polynomial interpolation using Sudan's list-decoding algorithm. We provide preliminary results into the feasibility of this protocol, by looking at the practicality of finding a polynomial that can be assigned as a cryptographic key to each device.

Contents

1 Introduction **3**

2 Physically Unclonable Functions **4**
2.1 Mathematical Model 5
2.1.1 Same neighborhood with high probability 5
2.1.2 Same output with high probability 6

3 Error-Correcting Codes **7**
3.1 Terminology 7
3.2 Linear Codes 8
3.3 BCH Codes 9
3.3.1 Encoding 9
3.3.2 Decoding 10
3.4 Reed-Solomon Codes 11
3.4.1 Code description 11
3.4.2 Alternative look at Reed-Solomon codes 11

4 Fuzzy Extractors and PUFs **13**
4.1 Basic Ideas 13
4.2 Preliminary Definitions 14
4.3 Secure Sketches 15
4.4 The Code-Offset construction 17
4.5 Fuzzy Extractors 22
4.6 Hash Functions 24
4.7 Overall design of fuzzy extractors 26

5 Hardware implementation of Fuzzy Extractors **28**
5.1 BCH codes and Fuzzy Extractors 28
5.2 Hardware architecture for a BCH decoder 29
5.2.1 Basic steps in BCH decoding 29
5.2.2 Syndrome Computation 30
5.2.3 Finding the Error-Locator polynomial 31
5.2.4 The Berlekamp-Massey Algorithm 34
5.2.5 Correcting errors 42
5.3 BCH Decoder Implementation Results 44

6 **A new approach to PUF-based hardware authentication** **48**
6.1 Polynomial as the secret 48
6.2 Polynomial Interpolation 49
6.3 Lagrange Interpolation 50
6.3.1 Restrictions on Lagrange Interpolation 51
6.4 Polynomial interpolation "with lies" 51
6.5 Sudan's algorithm 54
6.5.1 Sudan's algorithm procedure [17] 55
6.5.2 The polynomial $Q(x, y)$ 56
6.5.3 Polynomially many solutions 58
6.5.4 Factoring Polynomials 58
6.5.5 Implementing Sudan's Algorithm 59

7 **Protocol** **62**
7.1 Protocol Assumptions 62
7.2 Protocol design iterations 63
7.2.1 Initial Protocol 63
7.2.2 Hashed output 63
7.2.3 Sudan's algorithm for recovering from noise 64
7.2.4 Challenge-dependent hashed coefficients 64
7.3 Existence of low-degree polynomial 65
7.3.1 The particular case of a zero-degree polynomial 65
7.3.2 Back to the original problem 66
7.3.3 Coding-theory techniques to estimate probability 67

8 **Conclusion** **72**
8.1 Future Work 73

Bibliography **75**

Appendix **77**

A **Selected proofs** **77**

B **Hardware Complexity** **79**
B.1 Average number of nonzero coefficients for elements of $GF(2^m)$ 79
B.2 Reducing the complexity of Syndrome computation 79
B.3 Gate count for Inverter block 81
B.4 Gate count for Chien's Search block 81

C **Sudan's algorithm MAPLE source code** **82**

Chapter 1

Introduction

In today's world, the concern about piracy and counterfeit products is constantly increasing. The same advances that supply us with new technologies provide counterfeiters with a way of reverse-engineering and reproducing them. As we move into an era where we rely on small devices, such as cell-phones, credit cards and RFIDs, to store sensitive personal information and to grant us access to bank accounts, medical records and private areas, the importance of fighting counterfeiting becomes even more evident.

Research for ways of assigning digital fingerprints to devices, which allow for easy authentication and the detection of cloned devices has been receiving much attention. One of the most interesting methods of hardware fingerprinting, proposed in [14], is the idea of Physically Unclonable Functions (PUFs). These functions make use of very sensitive physical properties of a device in order to assign it a unique identifier, making the task of producing an identical device much harder.

A great deal of effort has been put into developing ways of using these PUFs to provide reliable authentication of devices. Most of the schemes proposed so far rely on collecting challenge-response pairs that are unique to each PUF device. However, the same properties that make PUFs attractive for cryptographic applications make them hard to deal with. Due to the high sensitivity to physical parameters, the responses of a PUF are naturally noisy. In this project, we examine some of the ways of handling this noise and propose a new method which presents a possible improvement in the security of the authentication protocol.

In chapter 2, we introduce the idea of a physically unclonable function which maps challenges to responses that depend highly of the physical properties of each device. Small variations between devices allow for a unique identifier to be assigned to each device, which can serve as a cryptographic key. In chapter 3, we introduce the concept of an error-correcting code which allows for messages sent over a noisy channel to be correctly received. We describe BCH codes and their special case, Reed Solomon Codes. In chapter 4, we analyze the known methods of fuzzy extractors and secure sketches as a way to recover from the noise associated with the responses of a PUF device. In chapter 5, we provide an implementation of a fuzzy extractor in hardware that aims at area efficiency. In chapter 6, we introduce a new methodology to assign a key to each PUF device based upon polynomial interpolation and present Sudan's list decoding algorithm as a way to extract the key from the noisy outputs of the device. Finally, in chapter 7, we present a protocol which is based upon this method, and analyze its feasibility.

Chapter 2

Physically Unclonable Functions

The term Physically Unclonable Function, or PUF, is used by scientists and engineers to refer to a practical device or process with a measurable output that strongly depends upon physical parameters. These "functions" are usually embodied in a hardware structure, such as a chip or a circuit board, and they obtain their "unclonability" from the fact that their outcomes depend on very sensitive physical properties, such as the length of wires or silicon doping levels, which cannot be precisely controlled in a manufacturing process.

In this report, we will be mostly concerned with a particular kind of PUFs, the *challenge-response* PUFs. These PUFs return a physically-dependent response r when given a challenge c. An important property of challenge-response PUFs is that when two devices from the same manufacturing process are given the same input, and are under the same external influences — temperature, pressure, magnetic fields, etc. — their outputs may differ. These differences are not intended, but are inherent in the nature of mass-produced devices. Therefore, if a PUF exploits a physical parameter with enough variability in such a way that the mapping between challenges and responses is unique to each device, it can be used to authenticate or uniquely identify that device.

Due to the high level of sensitivity of the physical parameters involved, it is considered infeasible for a person, when given the distribution on the responses of a specific PUF device, to fabricate another device which will produce identical responses. However, it is not considered infeasible for someone to model the responses of a specific PUF and use a different device that simply maps inputs to outputs in the same way, possibly using software.

An assumption that can be made about PUFs is that their internal workings behave as a "black box" and only their public output can be measured. For example, if a device were to receive ten challenges it would produce ten responses. If the device were to respond (output) with a hash of these ten responses, then it is assumed that the actual response to each of the ten inputs is part of the internal workings of the black box. An adversary would only have access to the hash of these responses. This assumption is justified by the fact that if an adversary opens the device to view the responses of the circuit, the physical parameters of the circuit will be altered and it will consequently produce different responses. Putting the device back together does not guarantee the device will produce the same outputs as before.

From the cryptographic perspective, PUF devices are attractive since the different outputs among these otherwise identical devices can be used to identify and authenticate

a particular device. Moreover, this can be accomplished without explicitly storing any authenticating information on the device, such as a cryptographic key. Other authentication schemes such as public-key cryptography require the device's private key to be explicitly stored somewhere in the memory on the device. An authentication scheme utilizing PUFs can use the unique physical properties of each device to generate a key every time the device is prompted with challenges. If an adversary attempted to learn this key through an active attack, the physical properties of the device would change and this key would become unusable.

Example 2.1.
We now follow [6] to present the delay-based PUF. A delay-based PUF receives a challenge $c \in \{0,1\}^k$ and responds with $r \in \{0,1\}$. Upon receiving a challenge, a pulse generator generates a signal which then splits and travels down two wires, which we will refer to as the top wire and the bottom wire. Each wire passes through k switches serially. The output of one switch is the input to the next switch. Both wires are then connected to an arbiter which outputs a 1 if the signal on the top wire arrives first, a 0 if the signal on the bottom wire arrives first, or if the difference in arrival times is below the arbiter's sensitivity level, it outputs 0 or 1 uniformly at random. When the kth challenge bit is a 0, the kth switch allows the two signals to pass through in a straight path and remain on the same wire. Otherwise, if the kth challenge bit is a 1, the kth switch switches the signal, and the signal which was on the top wire is now on the bottom wire and the signal that was on the bottom wire is now on the top wire.

These wires are designed to have equal length. However, the process variations lead to length variations and to different time delays for each signal. These delays are dependent upon the challenge received, and upon the specific circuit. Two identically produced circuits will have different length wires and will respond differently to certain challenges. It is these differences which can be used to authenticate the device.

2.1 Mathematical Model

We will use two slightly different mathematical models for the PUF function. We will first describe a PUF which will respond from a set of responses which are in some neighborhood of each other when given the same challenge. Then we consider the case where, for each challenge, the PUF produces an identical output with high probability.

2.1.1 Same neighborhood with high probability

In order to apply the concept of secure sketches to PUFs (see Section 4.3) we need to define the concept of proximity, or distance between the possible outputs of the PUF. A secure sketch can take advantage of a PUF which, when given the same challenge c, outputs responses which are a small distance away from each other (see [3] and [4]).

We formalize this as follows. Let $\mathbb{F}$ be a finite field with q elements and let $\mathcal{Y}$ be a probability space. We assume our co-domain $\mathbb{F}$ is a metric space with distance function dis. We let R_c be a set of responses to challenge a $c \in \mathbb{F}$ such that $dis(r_i, r_j) \leq t$ for all

r_i, r_j in R_c. We now define our PUF function as a map

$$\mathsf{PUF} : \mathbb{F} \times \mathcal{Y} \to \mathbb{F}$$

such that

$$\mathsf{PUF}\,(c, y)_{y \leftarrow \mathcal{Y}} \in R_c \text{ with probability at least } 1 - \epsilon \tag{2.1}$$

Property (2.1) formalizes the definition of closeness of the outputs to the same input, or the expected neighborhood.

2.1.2 Same output with high probability

In order to introduce the idea of PUF authentication through polynomial interpolation (see Section 6.1), we modify the previous definition slightly. We let $\mathbb{F}$ be a finite field with q elements and $\mathcal{Y}$ be a probability space. Then we assume that there exists a highly nonlinear function

$$f : \mathbb{F} \to \mathbb{F}$$

We now model our PUF function as

$$\mathsf{PUF} : \mathbb{F} \times \mathcal{Y} \to \mathbb{F}$$

where

$$\mathsf{PUF}\,(c, y)_{y \leftarrow \mathcal{Y}} = f(c) \text{ with probability at least } 1 - \epsilon \tag{2.2}$$

Property (2.2) states that with y ranging over all of $\mathcal{Y}$, $f(c)$ is the output with probability at least $1 - \epsilon$.

We will refer to $f(c)$ as the expected output of the PUF to challenge c. Abusing notation, we will refer to $\mathsf{PUF}\,(c, y)_{y \leftarrow \mathcal{Y}}$ as $\mathsf{PUF}\,(c)$. We therefore have that for each $c \in \mathbb{F}$, the output of the PUF function to challenge c is $\mathsf{PUF}\,(c)$ with probability at least $1 - \epsilon$.

Chapter 3

Error-Correcting Codes

Coding theory primarily addresses the problem of sending information over a channel which may distort the information. In a communication system, we assume the existence of two parties, a transmitter and a receiver. In order to assure that the information is correctly received, the transmitter and the receiver agree beforehand on a set of rules, called a *Code*, which, if followed, will make the transmitted message more resilient to noise. More specifically, the two parties agree on a subset of all transmittable messages, the *codewords*, which can be sent over the channel.

The theory of error-correcting codes deals with the detection and correction of errors in the received information and with the design of codes that will increase the chances of successful information recovery from a noisy channel.

3.1 Terminology

Working over the binary alphabet, we call a 0 or 1 a digit, and we call a sequence of digits a word. We will only consider consider block codes, in which all words have the same length. We let $\mathcal{M}$ be a metric space with dimension $n = \log_2 |\mathcal{M}|$, which consists of all binary words of length n. A code C is simply a collection of words, i.e. a subset of $\mathcal{M}$. A word that belongs to the code is called a codeword. We denote the dimension of C by $k = \log_2 |C|$, when C is a subspace of $\mathcal{M}$.

More generally, if the alphabet $\mathbb{F}$ has size q, we refer to the dimension of the code as $k = \log_q |C|$ and the dimension of the space as $n = \log_q |\mathcal{M}|$. We define the information rate of the code to be $\frac{k}{n}$.

The space $\mathcal{M}$ has a distance function $\mathsf{dis} : \mathcal{M} \times \mathcal{M} \to [0, +\infty)$. We will mostly be interested in the *Hamming distance.*

Definition 3.1. *Hamming distance:* If $w, w' \in \mathcal{M} = \mathbb{F}^n$, then $\mathsf{dis}(w, w')$ is the number of positions in which w and w' differ. Similarly, the *Hamming weight* of w is the number of nonzero positions in w, or simply $\mathsf{dis}(w, 0)$.

Definition 3.2. *Minimum distance:* The minimum distance d of a code is given by:

$$d = \min \{\mathsf{dis}(w, w') : w, w' \in C, w \neq w'\}$$

In general, a k-dimensional code which is defined over an n-dimensional metric space (and therefore has block size n), and with a minimum distance d is referred to as a $(n, M, d)_q$ code, where q is the size of the alphabet $\mathbb{F}$ and $M = |C|$.

Recovering the transmitted codeword

It is assumed that the errors on a transmission channel are somewhat unlikely (with probability less than $\frac{1}{2}$). Therefore the problem of recovering the transmitted codeword can be reduced to finding the codeword closest to the received word. In other words, an error-correcting scheme is usually concerned with finding the (preferably unique) codeword within a specified distance from the received word. The maximum distance (or number of errors) from the received word within which we are assured to have a unique codeword is the error-correction bound t of the code.

The *maximum likelihood decoding* problem, also known as the *nearest codeword* problem, is the problem of finding the nearest codeword to the received word. This is often used when a large number of errors is considered less likely. This is a generalization of the t-error correction scheme since a received element s, which is at distance most t from a codeword, will be decoded into the codeword. However, if s is at a distance greater than t from a codeword, this scheme will return the nearest codeword, or one of the nearest codewords.

The *p-reconstruction* problem also known as the *list-decoding* problem also generalizes the t-error correction. In this problem we are given an element s, and we find all codewords which are within a distance p of s. In other words, we find all codewords that are in a moderate-sized n dimensional Hamming sphere around the received element s. The decoding is considered successful if the transmitted element is a member of the list.

3.2 Linear Codes

In this report, we will mostly deal with linear codes and not all statements hold for non-linear codes. A linear code is a code which is closed under the addition of codewords,

$$u, v \in C \Rightarrow (u + v) \in C \ ,$$

and scalar multiplication, if $q > 2$. A linear code must contain the zero word. This is shown by adding a codeword c to itself $|\mathbb{F}| - 1$ times. When the codewords are from the binary field, a codeword added to itself is the zero codeword. The *distance* of a linear code C is the minimum weight of any non-zero codeword in C. This is a standard result which is shown by letting the minimum weight of a non-zero codeword be ω. Then, assume that there are two codewords, $u, v \in C : \phi = \mathsf{dis}\,(u, v) < \omega$. Since C is a linear code, $u - v$ is a codeword which has weight less than ω. Therefore we must have $u - v = 0$ which implies $u = v$.

To refer to linear codes, it is common to use the form $[n, k, d]_q$, where k is the dimension of C.

Hamming bound [7]

If C is a code of length n and minimum distance $d = 2t + 1$ or $d = 2t + 2$ then

$$|C| \left(\binom{n}{0} + \binom{n}{1} + ... + \binom{n}{t} \right) \leq 2^n$$

A perfect code is a code in which the Hamming bound is met. In other words, for each element w in our metric space $\mathcal{M}$, there exists a unique codeword $c \in C$ such that $\mathsf{dis}(w, c) \leq t$, meaning there exists a unique codeword within the error-correcting bound of the code.

We now introduce the two types of error-correcting codes used throughout this report: the *BCH* codes, and their non-binary special case, the famous *Reed-Solomon* codes.

3.3 BCH Codes

BCH codes are a large class of powerful error-correcting codes, invented in 1959 by Hocquenghem, and independently in 1960 by Bose and Ray-Chaudhuri. The great advantage of BCH codes is that given a block size $n = q^m - 1$, for any integer m, one can successfully build an $[n, k, 2t+1]_q$ code for any error-correction bound $t \leq q^{m-1} - 1$, with dimension $k \geq n - mt$ (see [15]). In general, even though BCH codes can be defined for non-binary alphabets ($q > 2$), it is in the binary case that they find their largest applicability. Among the non-binary examples, we have the ubiquitous *Reed Solomon* codes, analyzed more in depth in Section 3.4.

Formally, BCH codes are cyclic polynomial codes, constructed as follows. For a given $n = q^m - 1$ block size, we can build a linear code with minimum distance $d = 2t + 1$, by taking a primitive element α of $GF(q^m)$ and defining the code *generator polynomial* as:

$$g(x) = \text{LCM}\left(\phi_1(x), \phi_2(x), \phi_3(x), ..., \phi_{2t}(x)\right) \tag{3.1}$$

where $\phi_i(x)$ is the minimal polynomial of α^i over $GF(q)$. To show that this generates a code with minimum distance $d = 2t + 1$ we refer to Appendix A.3, where we prove this for the special case of Reed-Solomon codes. The proof for general BCH codes is similar [13].

3.3.1 Encoding

The encoding process for a BCH code takes a k-tuple with symbols from $GF(q)$ and adds $d - 1 = 2t$ parity check digits. Let us suppose that we have our k-symbol message, represented by the following polynomial:

$$a(x) = a_0 + a_1 x + ... + a_{k-1} x^{k-1}$$

Then we multiply $a(x)$ by x^{d-1} and use the division algorithm to write:

$$x^{d-1} a(x) = g(x)c(x) + b(x) \Rightarrow x^{d-1} a(x) - b(x) = g(x)c(x)$$

Since $x^{d-1} a(x) - b(x)$ is shown to be a multiple of the generator polynomial it must be a codeword.

3.3.2 Decoding

Our received codeword $r(x)$, an nth-degree polynomial, can be expressed as $r(x) = v(x) + e(x)$, where $v(x)$ was the transmitted codeword and $e(x)$ is the error vector, caused by the noise in the channel.
Since $v(x)$ is a codeword, we know that $v(\alpha^i) = 0$ for $i = 1, 2, ..., d-1$, and therefore we have

$$r(\alpha^i) = e(\alpha^i) \qquad i = 1, 2, ..., d-1$$

This allows us to compute the d-1 *syndromes* of the received message to be:

$$S_1 = r(\alpha) = e_{j_1}\alpha^{j_1} + e_{j_2}\alpha^{j_2} + ... + e_{j_\tau}\alpha^{j_\tau}$$

$$S_2 = r(\alpha^2) = e_{j_1}\alpha^{2j_1} + e_{j_2}\alpha^{2j_2} + ... + e_{2j_\tau}\alpha^{2j_\tau}$$

$$\vdots$$

$$S_{d-1} = r(\alpha^{d-1}) = e_{j_1}\alpha^{(d-1)j_1} + e_{j_2}\alpha^{(d-1)j_2} + ... + e_{j_\tau}\alpha^{(d-1)j_\tau}$$

where τ is the number of errors that actually occurred.
We then define the *error-locator polynomial* to be:

$$\begin{aligned}\sigma(x) &= (1 - \alpha^{j_1}x)(1 - \alpha^{j_2}x)...(1 - \alpha^{j_\tau}x) = \\ &= 1 + \sigma_1 x + ... + \sigma_\tau x^\tau \end{aligned} \tag{3.2}$$

Clearly, if we have the error-locator polynomial, the error locations can easily be found by finding the roots of $\sigma(x)$. Then, if α^{-i} is a root of $\sigma(x)$, then there was an error at the ith position. The coefficients in (3.2) can be found from the $d-1$ syndromes by using the Berlekamp-Massey algorithm (see Section 5.2.4).

Once we have the error-locator polynomial all we have to do is find its roots and calculate their inverses in order to find the locations in our received vector $r(x)$ where errors have occurred. At this point, if we are in the binary case, correcting the errors is trivial. We simply flip the bits at the location of the errors.

In the non-binary case, we still need to determine what the actual errors were. In order to do that we first define the *error-evaluator polynomial* to be

$$Z(x) = \sigma(x)S(x) \pmod{x^{2t+1}} \tag{3.3}$$

where $S(x)$ is the $2t^{\text{th}}$-degree polynomial formed by looking at each of the syndromes S_i as the coefficient of x^i. Then we can show that the error e_{j_i} at the j_i^{th} location is given by

$$e_{j_i} = \frac{Z(\alpha^{-j_i})}{\prod_{a=1, a\neq i}^{\tau} (1 + \alpha^a \alpha^{-j_i})} . \tag{3.4}$$

Finally, one can simply subtract each of the errors e_{j_i} from the j_i^{th} coefficient of $r(x)$ in order to obtain $v(x)$.

3.4 Reed-Solomon Codes

Invented in 1959 by Irving Reed and Gustave Solomon, the Reed-Solomon codes are probably the most ubiquitous type of error-control codes. The fact that they achieve the *Singleton Bound* (see Section 7.3.3, Equation (7.5)) allied to their robustness against burst errors made them the code of choice in several applications, ranging from compact discs to the transmission system of the *Voyager* spacecraft. However, the discovery that Reed-Solomon codes can be thought of as a special non-binary case of BCH codes is due to Gorenstein and Zierler, in 1961. Because of that, the same encoding and decoding steps described in the previous section hold for Reed-Solomon codes.

3.4.1 Code description

Reed-Solomon codes are BCH codes in which $m = 1$. Therefore, our block size is simply $n = q - 1$, where q is a prime power. Also, in order to form our generator polynomial $g(x)$, we look for the minimal polynomial over $GF(q)$ of α^i, for $i = 1, 2, ..., 2t$, where α is a primitive element of $GF(q)$ as well. Each minimal polynomial $\phi_i(x)$ from (3.1), is then given by $\phi_i(x) = (x - \alpha^i)$. Therefore we can define Reed-Solomon codes as non-binary cyclic polynomial codes, constructed as follows. For a given $n = q - 1$ block size, we can build a q-ary code with minimum distance d, by finding a primitive element α of $GF(q)$ and defining the generator polynomial to be:

$$g(x) = (x - \alpha)(x - \alpha^2)...(x - \alpha^{d-1}) \tag{3.5}$$

This generates a t-error correcting code (see Appendix A.3), where $t = \lfloor \frac{d-1}{2} \rfloor$. Since our generator polynomial has degree $d - 1$ we have that the dimension of our code is $k = n - d + 1$ (Singleton Bound).

3.4.2 Alternative look at Reed-Solomon codes

In order to understand the algorithm proposed by Sudan [17] to decode Reed-Solomon codes beyond the error-correction bound t (see Section 6.5), we need to look at Reed-Solomon codes from a different perspective. Instead of looking at a codeword $\mathbf{v} = (v_0, v_1, ..., v_{n-1})$ as the coefficients of the polynomial $v(x) = v_0 + v_1 x + ... + v_{n-1} x^{n-1}$, where $n = q - 1$, we look at $\mathbf{v}$ as the values of a certain polynomial $p(x)$ evaluated at some specific points.

In order to see this we must first define the *Discrete Fourier Transform*:

Definition 3.3. [2] Let $\mathbf{v} = (v_0, v_1, ..., v_{n-1})$ be a vector over $GF(q)$ with corresponding polynomial representation $v(x) = v_0 + v_1 x + ... + v_{n-1} x^{n-1}$, and let α be a primitive element of $GF(q)$. The Fourier Transform of the vector $\mathbf{v}$ is the vector $\mathbf{V} = (V_0, V_1, ..., V_{n-1})$, where:

$$\mathbf{V}_j = v(\alpha^j)$$

Additionally, we consider the polynomial representation of the vector $\mathbf{V}$ to be $V(x) = V_0 + V_1 x + ... + V_{n-1} x^{n-1}$.

Now, let us assume that one wants to transmit a message consisting of a polynomial

$\mathbf{v}(x)$ over $GF(q)$. All the information is encoded by the coefficients $v_0, v_1, ..., v_{k-1}$, so we assume k to be the size of the message. Then, the encoding process is as follows. For a primitive element α of $GF(q)$ we evaluate $v(\alpha^i)$ for $i = 1, 2, ..., n-1$. Then we build the vector containing the results $V = (v(\alpha^0), v(\alpha), v(\alpha^2), ..., v(\alpha^{n-1}))$. In other words, we applied the Discrete Fourier Transform to $\mathbf{v}$ and obtained $\mathbf{V}$, or its corresponding polynomial $\mathbf{V}(x)$.
Now, we look at what happens when we compute $\mathbf{V}(\alpha^i)$:

$$\begin{aligned}
\mathbf{V}(\alpha^i) &= v(1) + v(\alpha)\alpha^i + v(\alpha^2)\alpha^{2i} + ... + v(\alpha^{n-1})\alpha^{(n-1)i} \\
&= v_0 + v_1 + v_2 + ... + v_{k-1} \\
&+ \alpha^i(v_0 + v_1\alpha + v_2\alpha^2 + ... + v_{k-1}\alpha^{k-1}) \\
&+ \alpha^{2i}(v_0 + v_1\alpha^2 + v_2\alpha^4 + ... + v_{k-1}\alpha^{2(k-1)}) \\
&\vdots \\
&+ \alpha^{(n-1)i}(v_0 + v_1\alpha^{(n-1)} + v_2\alpha^{2(n-1)} + ... + v_{k-1}\alpha^{(k-1)(n-1)}) \\
&= \sum_{j=0}^{k-1} v_j \sum_{m=0}^{n-1} \alpha^{m(i+j)} \qquad (3.6)
\end{aligned}$$

Any element x of $GF(q)$ is a root of

$$1 - x^n = (1 - x)(1 + x + x^2 + ... + x^{n-1})$$

Therefore, if $\alpha^i \neq 1$, then $\sum_{m=0}^{n-1} \alpha^{mi} = 0$. Otherwise, if $\alpha^i = 1$, $\sum_{m=0}^{n-1} \alpha^{mi} = n$. Looking at the second sum in (3.6), we see that $\alpha^{i+j} = 1$ if and only if $i + j \equiv 0 \pmod{n}$ which implies $j = n - i$. Since j varies from 0 to $k - 1$ we conclude that:

$$\mathbf{V}(\alpha^i) = \begin{cases} 0 \text{ if } i < n - k + 1 = d \\ nv_{n-i} \text{ if } i \geq n - k + 1 = d \end{cases}$$

Thus, we see that $\mathbf{V}(x)$ is a codeword according to our definition of Reed-Solomon codes. Each of its components can either be seen as coefficients of the polynomial $\mathbf{V}(x)$, which is a multiple of the generator polynomial defined in (3.4.1) or as the value of the polynomial $\mathbf{v}(x)$ when evaluated at the points $\alpha^0, \alpha^1, ..., \alpha^{n-1}$. Since we only need k points to recover the coefficients of $\mathbf{v}(x)$, the other $n - k = d - 1$ points can be seen as redundancies.

An example of the application of the Discrete Fourier Transform is shown in Section 6.4.

Chapter 4

Fuzzy Extractors and PUFs

Physically Unclonable Functions provide a tamper-resilient method to assign a cryptographic key to a hardware device. However, the same dependency upon physical parameters that give PUFs their protection against active attacks causes their responses to be naturally noisy. The basic premise of PUFs is, nonetheless, that the "distance" between two responses from the same PUF is much smaller than the distance between responses from different PUFs, so that even in the presence of noise it is possible to distinguish responses originating from different PUF devices.

However, cryptography in general relies on the existence of precisely reproducible keys. This means that in order for us to take full advantage of the physical unclonability of PUFs we must find tools that allow us to cope with this noise. Moreover, it is expected that the distribution of PUF responses across multiple PUF devices will not be uniform, and this poses a second problem to the use of PUFs in cryptographic applications. In order to address these two issues, namely the small variations in the responses of the same PUF and the nonuniform distribution of responses among different PUFs, the concept of a fuzzy extractor, initially introduced in [4], is brought into play.

4.1 Basic Ideas

A fuzzy extractor is a primitive that allows the extraction of uniformly random strings from a nonuniform source in a noise-tolerant way. Since there are basically two separate issues being addressed here, it is natural to think that fuzzy extractors will consist of two independent steps. This turns out to be the case: first we have the *information reconciliation* phase, in which a noisy version of the output of a PUF is converted into a noise-free one. A *privacy amplification* phase follows, extracting randomness out of the noiseless version of the PUF output.

Regarding the information reconciliation phase, one might ask what we mean by a noiseless version of the PUF response, since all possible outputs seem noisy when compared to others. The idea here is that prior to the deployment of the device, extensive testing can be performed in order to determine the expected outcome or, even in the absence of a prevalent response, we can arbitrarily pick one of the responses as in the neighborhood of the most likely responses as "the noise-free version". This follows our first mathematical model for a PUF, given in Section 2.1.1.

A second question that one might ask, especially when an expected outcome cannot be

easily determined from multiple responses, is how a fuzzy extractor can remove the noise from a particular PUF response if it has nothing to compare it against. In other words, how can outside information about the noiseless response of the PUF be introduced into the system? This is done by means of a *Helper String*.

When a PUF device equipped with a fuzzy extractor system receives a helper string from whoever is trying to verify the identity of the device, it obtains just enough information to eliminate its response variations and output a consistent response. However, the helper data is assumed to be public, and therefore must not reveal any useful information about the PUF response to someone other than the PUF device itself. In some sense, the helper data can be thought of as a hint that only makes sense to the person to which it is given.

Once a "clean" response is obtained, traditional techniques for randomness extraction, such as hash functions, can be employed. This allows us to obtain PUF responses that are almost uniformly distributed across different PUF devices.

In the next sections, we give precise mathematical definitions and examples of the primitives mentioned here.

4.2 Preliminary Definitions

Before we describe the concepts of secure sketches and fuzzy extractors, we must first introduce a few basic definitions, by following [4]. In general, $\mathcal{M}$ will denote the metric space being considered, with a distance function $\mathsf{dis} : \mathcal{M} \times \mathcal{M} \to \mathbb{R}^+$. We will mostly be dealing with *Hamming* distances on the sets $\mathcal{M} = \mathbb{F}^n$.

In order to study the security of secure sketches and fuzzy extractors, we must first be able to quantify how secure a given random variable is. For this purpose, we use the min-entropy:

Definition 4.1. *Min-Entropy:* The min-entropy of a random variable A measures its unpredictability, or how hard it is for someone to guess its output. It is given by:

$$\mathbf{H}_\infty(A) \stackrel{\text{def}}{=} -\log\Big(\max_a P(A = a)\Big) \tag{4.1}$$

For our purposes however, we will be more interested in the cases where we want to consider how hard it is for an adversary to guess the value of a random variable A, if he/she finds out the value of another random variable B.

Definition 4.2. *Average Min-Entropy:* The average min-entropy measures the unpredictability of a random variable A, given the value of another random variable B. It is defined as:

$$\tilde{\mathbf{H}}_\infty(A\,|\,B) \stackrel{\text{def}}{=} -\log\Big(\mathbb{E}_{b\leftarrow B}\Big[\max_a P(A = a\,|\,B = b)\Big]\Big) \tag{4.2}$$

Because we are looking for secure sketches and fuzzy extractor's outputs that look uniformly random to an observer, we need a way to measure how close to uniform a random variable is. This is accomplished with the concept of *statistical distance*:

Definition 4.3. *Statistical Distance:* The statistical distance measures how statistically apart are two discrete random variables. It is given by:

$$\mathbf{SD}(A,B) = \frac{1}{2}\sum_{x} |P(A=v) - P(B=v)| \tag{4.3}$$

In all these expressions, and throughout this report, $P(A = a)$ will refer to the probability that $A = a$ and $\log x$ will refer to the logarithm base 2 of x.

4.3 Secure Sketches

Let $\mathcal{M}$ be a metric space with distance function dis.

Definition 4.4. [4] An $(\mathcal{M}, m, \tilde{m}, t)$-secure sketch is a pair of randomized procedures, "sketch" (SS) and "recover" (Rec), with the following properties:

1. The sketching procedure SS on input $w \in \mathcal{M}$ returns a bit string $s \in \{0,1\}^*$.
2. The recovery procedure Rec takes an element $w' \in M$ and a bit string $s \in \{0,1\}^*$. The correctness property of secure sketches guarantees that if $\mathsf{dis}(w, w') \leq t$, then $\mathsf{Rec}(w', \mathsf{SS}(w)) = w$. If $\mathsf{dis}(w, w') > t$, no guarantee is provided about the output of Rec.
3. The security property guarantees that for any distribution W over $\mathcal{M}$ with min-entropy m, the value of W can be recovered by the adversary who observes s with probability no greater than $2^{-\tilde{m}}$. That is, $\tilde{\mathbf{H}}_\infty(W \mid \mathsf{SS}(W)) \geq \tilde{m}$.

Example 4.1.
Let's assume $\mathcal{M} = \mathbb{Z}_7$, shown in Figure 4.1 as the vertices of a heptagon, and define the distance between two vertices as the least number of edges to go from one vertex to another one. Our element w is picked from $\mathcal{M}$ according to a distribution W. For each choice of w, $\mathsf{SS}(w)$ may have two equiprobable outcomes, which are shown inside the rectangles next to each vertex in Figure 4.1.

We notice that if two inputs w_1 and w_2 may produce the same secure sketch, then the distance between them is 3. Therefore, if $\mathsf{dis}(w, w') \leq 1$, the original w can be recovered given w' and $\mathsf{SS}(w)$.

To analyze the entropy loss, we first look at the best possible case, when the inputs are uniformly distributed and therefore $\mathbf{H}_\infty(W) = -\log\left(\frac{1}{7}\right) = \log(7)$. In this case, for any given secure sketch, an adversary that knows the sketching procedure can guess w with a $\frac{1}{2}$ probability, resulting in an entropy loss of $\lambda = \log(7) - 1$. So, we have a $(\mathbb{Z}_7, \log(7), 1, 1)$-secure sketch.

For lower values of m, $\tilde{m}$ is smaller. When $m = \log\left(\frac{7}{3}\right)$, we can have a probability distribution W on the input, such that only 3 inputs have probability greater than 0, in which case we can consider the worst case where 0, 1 and 2 have probabilities $\frac{2}{7}$, $\frac{2}{7}$ and $\frac{3}{7}$ respectively. In this scenario, if an adversary sees the secure sketch $\mathsf{SS}(w)$, he/she can guess w with probability 1, and we have $\tilde{m} = 0$.

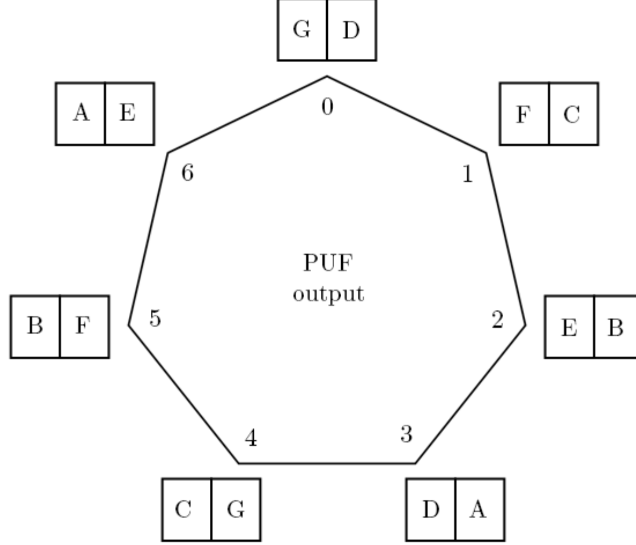


Figure 4.1: Possible outcomes of SS(w)

Claim 4.1. $\tilde{m} \leq m$
This is true because the distribution W on $\mathcal{M}$ is known to the observer. The secure sketch can only provide more information to this observer, allowing him/her to make a better guess as to which input w generated the given secure sketch. Mathematically, we have:

$$\begin{aligned}
\tilde{m} \leq \tilde{\mathbf{H}}_\infty(W \mid \mathsf{SS}\,(W)) &= -\log\left(\mathbb{E}_{s\leftarrow \mathsf{SS}\,(W)}\left[\max_w P(W = w \mid \mathsf{SS}\,(W) = s)\right]\right) \\
&= -\log\left(\mathbb{E}_{s\leftarrow \mathsf{SS}\,(W)}\left[\max_w \frac{P(W = w \cap \mathsf{SS}\,(W) = s)}{P(\mathsf{SS}\,(W) = s)}\right]\right) \\
&= -\log\left(\mathbb{E}_{s\leftarrow \mathsf{SS}\,(W)}\left[\frac{\max_w P(W = w \cap \mathsf{SS}\,(W) = s)}{P(\mathsf{SS}\,(W) = s)}\right]\right) \\
&= -\log\left(\sum_s \left[\max_w P(W = w \cap \mathsf{SS}\,(W) = s)\right]\right) \\
&\leq -\log\left(\max_w \sum_s P(W = w \cap \mathsf{SS}\,(W) = s)\right) \\
&= -\log\left(\max_w P(W = w)\right) = \mathbf{H}_\infty(W) = m
\end{aligned}$$

since (see Appendix A)

$$\sum_x \max_y f(x,y) \geq \max_y \sum_x f(x,y) \tag{4.4}$$

If the secure sketch provides no information about w to the outside observer (ideal), he/she can still guess w correctly with probability 2^{-m} (by choosing the input with the highest probability), resulting in $\tilde{m} = m$.

4.4 The Code-Offset construction

For practical applications, a good secure sketch construction must have efficient Rec and SS procedures and must minimize the entropy loss $\lambda = m - \tilde{m}$. Example 4.1 illustrates a case in which the entropy loss is very significant. That same example also suggests the existence of an underlying code, since we are essentially correcting errors whenever they are not too numerous. In the same sense, trying to find a good secure sketch is a similar problem to looking for a good code, since we try to maximize the distance between inputs with the same secure sketch while trying to have as many secure sketches as possible. This connection between secure sketches and codes becomes even clearer when we consider a Code-Offset construction, which provides an efficient and very practical method for designing secure sketches. It is defined as follows (see [4]).

We assume the existence of an $[n, k, d]$ code C with codewords defined on the same space $\mathcal{M} = \mathbb{F}^n$ from which the inputs are taken. We also define the distance to be the Hamming distance. For a given input w, the sketching procedure produces $\mathsf{SS}(w) = w + C(x)$, where x is a random element from $\mathbb{F}^k$, so that $c = C(x)$ is a random codeword. For the recovery procedure Rec, we assume an input w' such that $\mathsf{dis}(w, w') \leq \lfloor \frac{d-1}{2} \rfloor$, and we start by subtracting it from its secure sketch to obtain $c' = w + c - w' = c + (w - w')$. Since the Hamming weight of $w - w'$ is less than $\lfloor \frac{d-1}{2} \rfloor$, we know that $\mathsf{dis}(c, c') \leq \lfloor \frac{d-1}{2} \rfloor$ and we can apply a decoding algorithm to obtain the codeword c. Once we have c, we can simply subtract it from the secure sketch to obtain $(w + c) - c = w$. So we output $\mathsf{Rec}(w', \mathsf{SS}(w)) = w$.

This construction also presents the advantage that the secure sketch is just another element of $\mathbb{F}^n$ and therefore does not represent a problem in terms of storage, as opposed to a secure sketch construction using *fuzzy vaults* (see [9]) for example. In the remainder of this section, we will mostly deal with secure sketches based on linear codes defined over binary fields, in which addition and subtraction are equivalent, slightly simplifying the operations described above.

Example 4.2.
In this example, we have $\mathcal{M} = \{0, 1\}^5$. To build the secure sketch, we use the Code-Offset construction with the code being:

$$C = \mathrm{rowsp}_2 \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

The minimum distance is 3 (and $t = 1$), and this is a $[5, 2, 3]_2$-code. Now let's look at possible values for the min-entropy m of the distribution W over $\mathcal{M}$ and their corresponding $\tilde{m}$. One important observation is that we can look at $\mathcal{M}$ as an additive group, which makes C a normal subgroup. C partitions $\mathcal{M}$ into 8 cosets. The sketching procedure chooses a random codeword and adds it to the input. Since there are 4 codewords, each input may produce 4 different secure sketches, which correspond to one of the cosets. Also, since $w \in w + C$, we see that $\mathsf{SS}(w)$ and w must be in the same coset. This means that an observer who sees the secure sketch $\mathsf{SS}(w)$ has 4 possible choices when guessing the input w. This bounds the average min-entropy: $\tilde{\mathbf{H}}_\infty(W \mid \mathsf{SS}(W)) \leq 2$, and this value can only be attained if we assume a uniform distribution on the inputs.

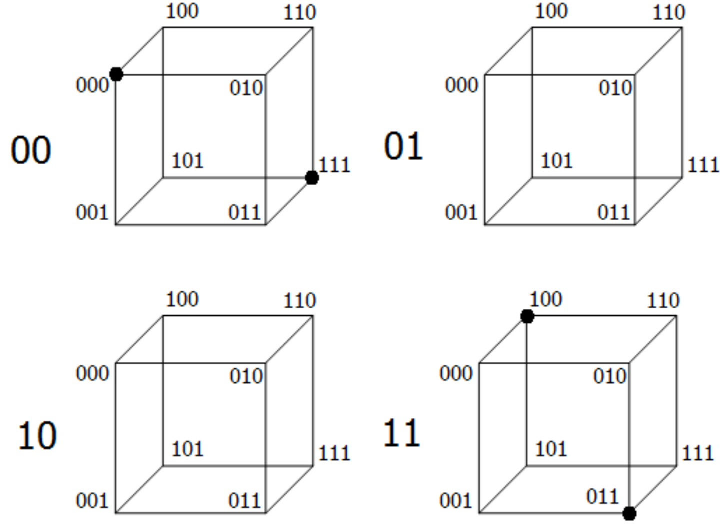


Figure 4.2: Code defined over $\{0,1\}^5$

Therefore, for inputs with min-entropies $m = 0, 1, 2, 3$, it is possible to have only 8 (or less) inputs with probability greater than 0. This means that in a worst-case scenario, one could choose each of the possible inputs to be in different cosets, which allows an observer to correctly discover the input w, given the secure sketch, with probability 1.

For $m = 4$, the problem of finding the worst possible input distribution becomes a little more difficult. For that, we state the following known Lemma:

Lemma 4.1. *Every distribution W over $\mathcal{M}$ with $\mathbf{H}_\infty(W) \geq m$ is a convex combination of distributions that are uniform on subsets of M with exactly 2^m elements.*

Using this we can write $P(W = w) = \sum_i \lambda_i P(W_i = w)$, where $\sum_i \lambda_i = 1$ and each W_i represents a distribution that is uniform over a subset of $\mathcal{M}$ with 2^m elements. Therefore we have:

$$
\begin{aligned}
\tilde{\mathbf{H}}_\infty(W \mid \mathsf{SS}(W)) &= -\log\left(\mathbb{E}_{s\leftarrow \mathsf{SS}(W)}\left[\max_w P(W = w \mid \mathsf{SS}(W) = s)\right]\right) \\
&= -\log\left(\mathbb{E}_{s\leftarrow \mathsf{SS}(W)}\left[\max_w \sum_i \lambda_i P(W_i = w \mid \mathsf{SS}(W) = s)\right]\right) \\
&\geq -\log\left(\mathbb{E}_{s\leftarrow \mathsf{SS}(W)}\left[\sum_i \lambda_i \max_w P(W_i = w \mid \mathsf{SS}(W) = s)\right]\right) \\
&= -\log\left(\sum_i \lambda_i \mathbb{E}_{s\leftarrow \mathsf{SS}(W)}\left[\max_w P(W_i = w \mid \mathsf{SS}(W) = s)\right]\right) \\
&\geq -\log\left(\max_i \mathbb{E}_{s\leftarrow \mathsf{SS}(W)}\left[\max_w P(W_i = w \mid \mathsf{SS}(W) = s)\right]\right) \\
&= -\log\left(\mathbb{E}_{s\leftarrow \mathsf{SS}(W)}\left[\max_w P(W_k = w \mid \mathsf{SS}(W) = s)\right]\right) \\
&= \tilde{\mathbf{H}}_\infty(W_k \mid \mathsf{SS}(W))
\end{aligned} \tag{4.5}
$$

where W_k is the uniform distribution over 2^4 elements that maximizes the quantity on th 5th step. This means that the minimum $\tilde{\mathbf{H}}_\infty(W \mid \mathsf{SS}(W))$, $\tilde{m}$, can be found among the average min-entropies of distributions W_i that are uniform on 2^m-subsets of M.

For our example, we would then be trying to find input distributions where 16 of the inputs have a probability of $2^{-4} = \frac{1}{16}$. Since the 16 possible inputs are equiprobable, we can simplify the average min-entropy as follows:

$$
\begin{aligned}
\tilde{\mathbf{H}}_\infty(W \mid \mathsf{SS}(W)) &= -\log\left(\mathbb{E}_{s\leftarrow \mathsf{SS}(W)}\left[\max_w P(W = w \mid \mathsf{SS}(W) = s)\right]\right) \\
&= -\log\left(\sum_s P(W = w_s \mid \mathsf{SS}(W) = s) P(\mathsf{SS}(W) = s)\right) \\
&= -\log\left(4 \sum_{i=1}^{\ell} \frac{1}{n_i} \cdot \frac{n_i}{16} \cdot \frac{1}{4}\right) \\
&= -\log\left(\frac{\ell}{16}\right) = 4 - \log(\ell)
\end{aligned} \tag{4.6}
$$

where ℓ is the number of cosets that have possible inputs in them, n_i is the number of inputs in a given coset (here we assume the cosets to be arbitrarily numbered from 1 to 8), and w_s is simply one of the inputs w such that $\mathsf{SS}(w) = s$ (they are all equiprobable).

This result basically shows that $\tilde{\mathbf{H}}_\infty(W \mid \mathsf{SS}(W))$, for a given input distribution W with min-entropy $m = 4$, is solely dependent on the number of cosets that have elements in them. It also allows us to calculate the minimum possible $\tilde{\mathbf{H}}_\infty(W \mid \mathsf{SS}(W))$, or $\tilde{m}$, to be $4 - \log(8) = 1$, since with 16 possible inputs it is possible to select at least one possible input in each of the 8 cosets.

The expression obtained for $\tilde{\mathbf{H}}_\infty(W \mid \mathsf{SS}(W))$, when $m = 4$ in the previous example, can be easily generalized for any secure sketch built with the Code-Offset construction, if the code used is binary and linear. In order to do that, we start by assuming a linear

$[n,k,d]$ code C, and again looking at C as a normal subgroup of $\{0,1\}^n$. Therefore, we have 2^{n-k} cosets, with 2^k elements in each coset, and we must have $\tilde{m} \leq k$.

Clearly, for $m = 0, 1, ..., n-k$, we can have one or zero possible inputs in each coset, therefore forcing a secure sketch via code-offset construction to reveal all the information about the input, and $\tilde{m} = 0$. For $m > n-k$ we refer to the result (4.5) to again narrow our search for $\tilde{m}$ down to the average min-entropies of distributions that are uniform on subsets of $\{0,1\}^n$ with 2^m elements and 0 otherwise.

By following the steps on (4.6) we can then find an expression for $\tilde{\mathbf{H}}_\infty(W \mid \mathsf{SS}(W))$:

$$\begin{aligned}\tilde{\mathbf{H}}_\infty(W \mid \mathsf{SS}(W)) &= -\log\left(\sum_s P(W = w_s \mid \mathsf{SS}(W) = s) P(\mathsf{SS}(W) = s)\right) \\ &= -\log\left(2^k \sum_{i=1}^{\ell} \frac{1}{n_i} \frac{n_i}{2^{m+k}}\right) \\ &= -\log\left(\frac{\ell}{2^m}\right) = m - \log(\ell)\end{aligned}$$

Since $\tilde{m}$ assumes a worst-case input distribution of entropy m, we simply have that

$$\tilde{m} = \min_{\ell}(m - \log \ell)$$

and considering we only have 2^{n-k} cosets and enough inputs to have at least one input in each of them, we can state the following theorem:

Theorem 4.1. *For a secure sketch using a code-offset construction with an (n,k,d) code, if the input follows a distribution W of min-entropy m, where m is an integer, the average min-entropy is given by*

$$\tilde{\mathbf{H}}_\infty(W \mid \mathsf{SS}(W)) = \begin{cases} m - n + k & \text{if } m > n-k \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

This result suggests that we should look for denser codes (k is closer to n) in order to obtain better secure sketching procedures. For example, we could consider the case where $M = \mathbb{Z}_{32}$, and our code is simply $C = \langle 4 \rangle$. Just like in the previous case we have 32 possible inputs, but this time we have only 4 cosets of size 8. In the best-case distribution over $\mathcal{M}$, i.e. $m = 5$, we can now construct a $(\mathcal{M}, 5, 3, 1)$-secure sketch, as compared to a $(\mathcal{M}, 5, 2, 1)$-secure sketch in the previous example.

The reason why we cannot simply reduce the number of cosets (or increase their size) arbitrarily, trying to obtain a better secure sketch, is the fact that the number of cosets bounds the minimum distance of the code. This, of course, is only true for "nice" metrics. For the remainder of this chapter, we will formalize this requirement by only considering metric spaces that can be seen as connected undirected graphs, in which the distance is simply defined as the shortest path between two vertices with the weight of each edge being one. We can then have the following claim:

Claim 4.2. For an $[n,k,d]$ linear code C, we must have $d \leq |\mathcal{M} : C|$.

Table 4.1: Entropy loss

	Ex. 4.2		$C = \langle 4 \rangle$	
m	$\tilde{m}$	λ	$\tilde{m}$	λ
0	0	0	0	0
1	0	1	0	1
2	0	2	0	2
3	0	3	1	2
4	1	3	2	2
5	2	3	3	2

Proof. If the minimum distance of the code is $d = 2t+1$, then for some pair of codewords w_a and w_b, $\mathsf{dis}(w_a, w_b) = d$. By following the path connecting w_a and w_b, we can find words $w_1, w_2, ...w_{2t} \in \mathcal{M}$ such that $\mathsf{dis}(w_a, w_i) = i$ and $\mathsf{dis}(w_b, w_i) = d - i$ for $i = 1, 2, ..., 2t$, and $\mathsf{dis}(w_j, w_k) = |j - k|$ for $1 \leq j, k \leq 2t$. We prove the claim by showing that each of these $2t$ elements must belong to a different non-identity coset, resulting in a total of at least $2t + 1 = d$ cosets.

Assume by contradiction that w_j and w_k, for $1 \leq j < k \leq 2t$, are in the same coset defined by $w_j + C = w_k + C$. By subtracting w_j, we obtain $C = (w_k - w_j) + C$, which implies that $w_k - w_j$ is a codeword. Since C is a linear code, the Hamming weight of any nonzero codeword should be at least d, which means that $\mathsf{dis}(w_j, w_k) \geq d$. But we have that $\mathsf{dis}(w_j, w_k) = k - j$, so we must have $k - j \geq d$. However, $k - j \leq 2t - j < d$ for any j, which is a contradiction. □

Since the number of cosets must also divide the size of $\mathcal{M}$, we conclude that when $\mathcal{M} = \{0,1\}^5$, by choosing $C = \langle 4 \rangle$ we obtain the best possible code-offset secure sketch. In general, we will have that the best average min-entropy provided by a secure sketch using code-offset with a linear code would be $\tilde{m} = m - n + \max_{d \leq 2^{n-k}} k$. However, in order to achieve this bound we need metric spaces that maximize the distance between words. Since we require these metric spaces to be connected graphs, we see that this bound can only be achieved if we consider cyclic graphs. However, it is usually unfeasible to work with codes defined on the vertices of a polygon. More realistically, our codes will be defined over a metric space $\mathcal{M} = \{0,1\}^i$ and the distance function will be the Hamming distance. Therefore, a much stricter bound on the best secure sketch that we can achieve can be formulated. We first notice that the **Singleton Bound** states that for a linear code using the Hamming distance we have that:

$$n - k \geq d - 1 \tag{4.8}$$

With Equation (7.5), we can have the following corollary of Theorem 4.1:

Corollary 4.1. *For an input distribution W of min-entropy m, from a metric space of dimension n with the Hamming distance, the best average min-entropy provided by a secure sketch using a code-offset construction with minimum distance d is:*

$$\tilde{m} = m - d + 1 \tag{4.9}$$

This result confirms our intuition that there is a trade-off between the error-correcting capability of a Rec procedure of a secure sketch and how much information leaks by the publishing of the secure sketch. By noticing that for a code to achieve the Singleton bound, we must have $d = 2t + 1$, where t is the error-correcting bound of the code, we find that

$$\tilde{m} = m - 2t$$

which means that to increase the noise-tolerance by one bit, we must in fact lose two bits of entropy. This creates an inherent limitation to the noise-tolerance levels of secure sketches and also to fuzzy extractors (see next section) built on top of secure sketches, especially in a setting where a large amount of noise is expected at the output of a PUF.

4.5 Fuzzy Extractors

Definition 4.5. [4] An $(\mathcal{M},m,\ell,t,\epsilon)$-*fuzzy extractor* is a pair of randomized procedures, Gen and Rep, with the following properties:

1. The generation procedure Gen on input $w \in \mathcal{M}$ outputs an extracted string $R \in \{0,1\}^\ell$ and a helper string $P \in \{0,1\}^*$.

2. The reproduction procedure Rep takes an element $w' \in M$ and a bit string $P \in \{0,1\}^*$ as inputs. The *correctness* property of fuzzy extractors guarantees that if $\mathsf{dis}(w, w') \le t$ and R, P were generated by $(R, P) \leftarrow \mathsf{Gen}(w)$, then $\mathsf{Rep}(w', P) = R$. If $\mathsf{dis}(w, w') > t$, then no guarantee is provided about the output of Rep.

3. The security property guarantees that for any distribution W on $\mathcal{M}$ of min-entropy m, the string R is nearly uniform even for those who observe P: if $(R, P) \leftarrow \mathsf{Gen}(w)$, then $\mathbf{SD}((R, P), (U_\ell, P)) \le \epsilon$.

Example 4.3.
Let $\mathcal{M} = \mathbb{Z}_{16}$, pictured as the vertices of a 16-gon, and let the distance be naturally defined as the distance between two vertices. We will define our generation procedure as follows:

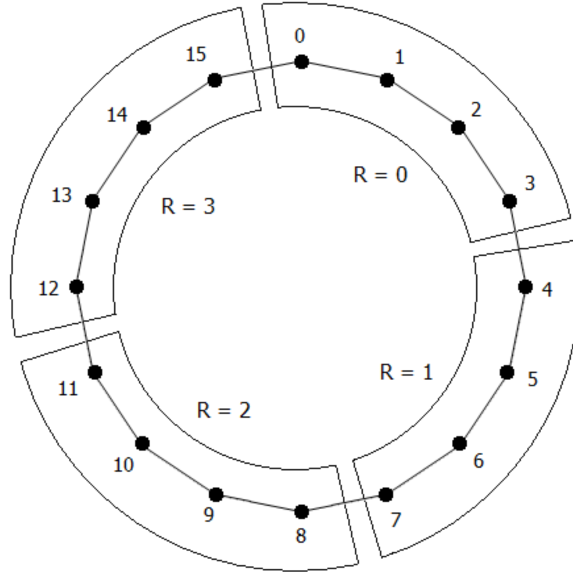
$$\begin{aligned}
&\mathsf{Gen}(w) = (R, P), \text{ where:}\\
&R = \lfloor w/4 \rfloor\\
&P = R \pmod 2
\end{aligned}$$

Our reproduction procedure is defined as follows:

$$\mathsf{Rep}(w', P) = \begin{cases} \lfloor w'/4 \rfloor & \text{if } \lfloor w'/4 \rfloor \pmod 2 = P \\ \lfloor w'/4 \rfloor - (-1)^{\lfloor w'/2 \rfloor} & \text{otherwise} \end{cases} \tag{4.10}$$

This fuzzy extractor can be understood through the following picture.

Our extracted string R tells us in which of the four sectors drawn in the Figure 4.5 w was. The helper string that is stored is simply the parity of R. Therefore, if an adversary



knows P, they still have two possible values of R to guess from. Therefore, if $\mathcal{M}$ follows a distribution W of min-entropy 4, an adversary has a $\frac{1}{2}$ chance of guessing R if given P, and we have $\tilde{m} = 1$

Given an input w' that is at a distance at most 2 of w, the parity of R is enough for us to recover the same key R that w would generate, and we have that $t = 2$. Notice that, in this case, w itself is never recovered.

With $m = 4$, we have a statistical distance $\mathbf{SD} = \frac{1}{2}(2 \times |\frac{1}{2} - \frac{1}{4}| + 2 \times |0 - \frac{1}{4}|) = \frac{1}{2}$. This is a very simple example and doesn't extract randomness from a non-uniform distribution (it assumes a uniform one).

Even though schemes like the one described in Example 4.3 can be devised, for any real application we need a more general and more flexible approach to design Fuzzy Extractors. Recall that a Fuzzy Extractor can also be understood as a two-step process. In the first step, the *information reconciliation* phase, a noisy version of an element w of our metric space is converted into a noiseless version. Then, in the *privacy amplification* phase, the fuzzy extractor fixes the non-uniformity of the distribution over $\mathcal{M}$. Clearly, secure sketches are good candidates for the information reconciliation phase. By providing this "hint" to the system, we can recover our original information. Moreover, the code-offset construction provides a straightforward method to construct secure sketches satisfying a given set of parameters, and thus we can use this method for the information reconciliation portion of fuzzy extractors.

Once we have our noiseless information w, we can then resort to known methods to extract nearly uniform random bits from a nonuniform source. Next we present a very well-studied mechanism to do this: the *hash functions*.

4.6 Hash Functions

A hash function is used when we have a non-uniform distribution on some source and we wish to extract a uniform distribution with roughly the same entropy. For example, if we have a space of binary 10-tuples with all elements having the first four digits being 0, we in fact only have six bits of information. A hash function in essence reduces the size of our space to reflect the actual amount of information in it.

Formally, a hash function is a mapping from $\{0,1\}^i$, the *keys*, to $\{0,1\}^j$, the *values*. Following [20], there are three properties of a cryptographic hash function:

1. A hash function h is a one way hash function if, for a random $x \in \{0,1\}^i$, given $h(x)$, it is hard to compute $y \in \{0,1\}^i$ such that $h(y) = h(x)$.

2. The hash of a key should be computed in polynomial time, and is sometimes performed in linear time in the length of the input.

3. A hash function is called *strongly collision free* if it is computationally infeasible to find two keys x_1, x_2 such that $h(x_1) = h(x_2)$. A hash function is *weakly collision free* if, given a key x, it is computationally infeasible to find an $x' \neq x$ such that $h(x) = h(x')$.

A hash family consists of a set K of keys, a set V of values, and a set H of hash functions which map keys to values.

Example 4.4.
Let our field be $\mathbf{H} := \{0, 1, \alpha, \beta\}$ with $\alpha^2 = \beta$ and $\alpha\beta = 1$. For addition and multiplication of this field, see Tables 4.2 and 4.3 respectively. The set of keys will be 2 tuples $(a, b) \in \mathbf{H}^2$ written as ab. The set of values V will be our field $\mathbf{H}$. Our hash family will consist of hash functions which use the keys to construct a linear polynomial which is evaluated at a particular field element depending on the specific hash function.

$+$	0	1	α	β
0	0	1	α	β
1	1	0	β	α
α	α	β	0	1
β	β	α	1	0

Table 4.2: Addition table

$\times$	0	1	α	β
0	0	0	0	0
1	0	1	α	β
α	0	α	β	1
β	0	β	1	α

Table 4.3: Multiplication table

Creating a linear polynomial over H

We now create a hash function, which will be a polynomial of degree at most 1 with coefficients from $\mathbf{H}$. This polynomial will be created by the key which is hashed. If ab is the key to be hashed, we write this polynomial as in 4.11. We have $\zeta \in \mathbf{H}$, and hence ζ determines the hash function. Our set H is $\{h_0, h_1, h_\alpha, h_\beta\}$, where the ordering of the hash functions corresponds to which element they evaluate the linear polynomial 4.11.

$$L(\zeta) := a\zeta + b \tag{4.11}$$

Our hash function is exhibited by the following mapping h_i with $i \in \mathbf{H}$. Now we create our polynomial using this function.

$$h_i : \mathbf{H}^2 \to \mathbf{H}$$
$$h_i(a,b) \mapsto a(i) + b \tag{4.12}$$

We use the key (the hashed value), (a,b) to tell us which row of the hash function to return values from. In 4.4 we show the complete table of possible return values. The first column corresponds to (a,b), the value to be hashed. The second, third, forth and fifth columns correspond to the output of the specific hash function.

ab	0	1	α	β
00	0	0	0	0
01	1	1	1	1
0α	α	α	α	α
0β	β	β	β	β
10	0	1	α	β
11	1	0	β	α
1α	α	β	0	1
1β	β	α	1	0
$\alpha 0$	0	α	β	1
$\alpha 1$	1	β	α	0
$\alpha\alpha$	α	0	1	β
$\alpha\beta$	β	1	0	α
$\beta 0$	0	β	1	α
$\beta 1$	1	α	0	β
$\beta\alpha$	α	1	β	0
$\beta\beta$	β	0	α	1

Table 4.4: Hash functions

Definition 4.6. *[4] [Strongly 2-Universal Family of Hash Functions] We say that H is a 2-universal family of hash functions if, for $x \neq y$, we have*

$$P[h(x) = h(y)]_{h\in H} \leq \frac{1}{|V|}$$

We say that H is a strongly 2-universal family of hash functions if

$$P[h(x) = x', h(y) = y']_{h\in H} = \frac{1}{|V|}$$

Claim 4.3. Our example is a 2 universal family of hash functions.

Proof. We need to show that $P[h(x) = h(y)] \leq \frac{1}{|V|}$
Since $h \in H$ is fixed, we only look at one column of 4.4 chosen uniformly at random. Since each column has the same number of occurrences of each value, we will look at just one column. For $x \in \mathbf{H}^2$ there are exactly 3 other keys which hash to the same value as x. Since there are 15 values that y can take, we conclude

$$P[h(x) = h(y)]_{h \in H} = \frac{3}{16} \leq \frac{1}{4}$$

We conclude that our example is a 2 universal family of hash functions. □

4.7 Overall design of fuzzy extractors

Hash functions are the necessary tools to implement the privacy amplification phase of fuzzy extractors. Allied with the code-offset construction for secure sketches, they allow for a straightforward construction of fuzzy extractors with flexibility in terms of the error-correcting capability. In general, during the enrollment phase, our generation procedure will use a noiseless version of the PUF output w and use it as an input to both the sketching procedure of a secure sketch and a hash function. This generates a secure sketch which can be seen as the helper data P and a hashed value R. In the authentication phase, the noisy version of the PUF output w' can be used with the helper string P in order to recover w, the same hash function can be used to generate R again. This basic scheme can be seen in Figure 4.3.

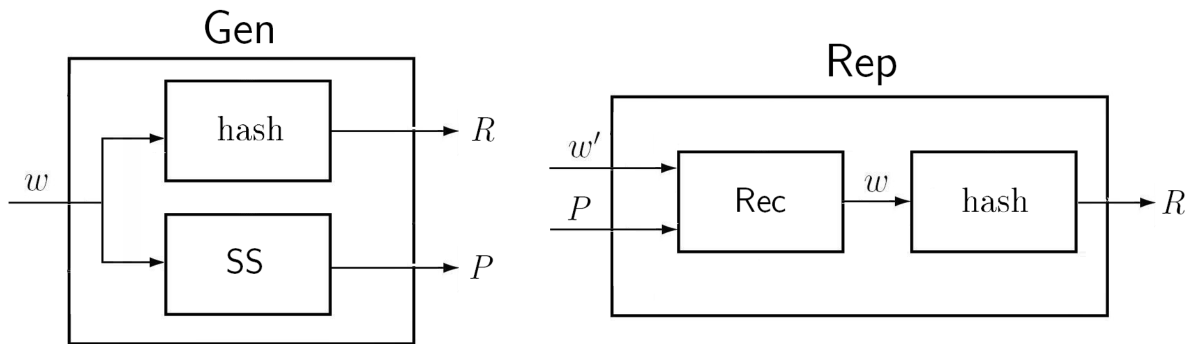


Figure 4.3: Basic scheme for generation and reproduction procedures of a fuzzy extractor

This simple scheme does not assume randomized procedures and requires the hash functions to be completely deterministic. However, this randomness is necessary for practical applications to reduce the entropy loss caused by the publication of the helper data. In order to allow randomized procedures, we need to modify the above scheme slightly. During the Gen procedure, we must be able to store the seed x which determines which hash function is chosen for a given key. This seed would then have to be stored with the secure sketch as another piece of the helper data. Later on, in the Rep procedure,

this seed can be provided to the hash function in order to assure that the right string R is generated. This new scheme can be summarized in the diagram in Figure 4.4.

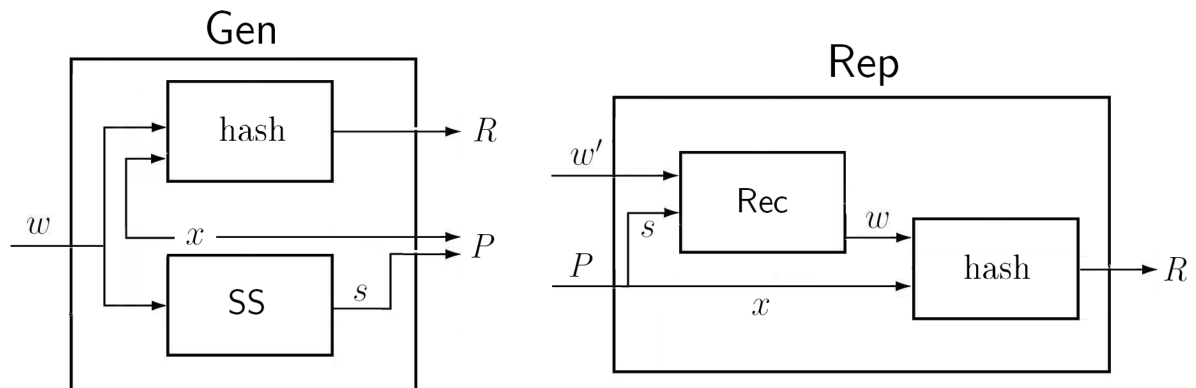


Figure 4.4: Randomized generation and reproduction procedures of a fuzzy extractor

Chapter 5

Hardware implementation of Fuzzy Extractors

One of the most appealing qualities of a PUF-based authentication system is the fact that PUFs can – and in fact have to – be implemented in hardware. This provides a security scheme for lightweight and low-cost hardware devices, such as RFIDs and smartcards [6]. However, in order to use fuzzy extractors as a way to obtain reliable cryptographic keys from PUFs, while still keeping PUFs attractive for these lightweight hardware devices, a fuzzy extractor must be implemented in hardware as well. Moreover, its implementation should be fairly area-efficient, or it will defeat the purpose of using PUFs in small hardware devices.

The code-offset/hash-function construction depicted in Figure 4.4 has the advantage of giving flexibility in terms of error-correction and security parameters. The most complex component of this scheme is the decoder that must be implemented for the recovery procedure of the secure sketch. Therefore, in this section we present a compact implementation for a decoder, which still provides good error-correcting capabilities.

5.1 BCH codes and Fuzzy Extractors

BCH codes are very powerful error-correcting codes. Even though they cannot be implemented as easily as other codes, such as Reed-Muller codes, an area-efficient hardware implementation is still feasible. This makes them very good candidates for being used in the context of fuzzy extractors. Linear codes are very important pieces in Fuzzy Extractors, in particular for the *information reconciliation* phase. We perform this task by using a secure sketch with a code-offset construction. Therefore, we need to be able to generate a random codeword, i.e. encode a random message, for the *sketching procedure* and to do decoding for the *recovery procedure*.

During the enrollment phase the device can be characterized more accurately by taking multiple samples and averaging to minimize noise. This is followed by the application of the sketching procedure which can be performed offline via software. Since this occurs prior to the deployment of the device and does not need to be done by the device itself, a hardware implementation is only needed for the recovery procedure. Therefore, we will mainly focus on the hardware implementation of a BCH decoder.

Since most PUFs output binary responses, and since there is no reason to be concerned about burst errors, binary codes satisfy our needs and we will be dealing with binary BCH codes only.

5.2 Hardware architecture for a BCH decoder

For security applications, and more specifically for hardware authentication, speed is not a major concern. In other words, it is not fundamental that a device can be correctly authenticated in microseconds. However, when we consider applications such as FPGA intellectual property (IP) protection (see [3]), it becomes clear that our main goal is area efficiency. After all, the authentication circuitry is typically part of a much larger and complex circuit, whose IP we want to protect. Therefore, our design focuses on serializing operations as much as possible, in order to reduce the overall gate count.

5.2.1 Basic steps in BCH decoding

The task of decoding an (n, k, d) BCH code in the communications setting can be summarized as follows: given a received polynomial $r(x) = v(x) + e(x)$, we want to find the original transmitted polynomial $v(x)$. Equivalently, we can also find the error polynomial $e(x)$. While designing the decoder for a binary BCH code, there are mainly four steps we have to consider.

1. **Syndrome computation**:
   We start by computing the $2t$ syndromes of the received message $r(x)$. That is done by simply plugging in the $2t$ roots of the generator polynomial, namely α^i for $i = 1, 2, ..., 2t$, to $r(x)$. Since $v(\alpha^i) = 0$ for $i = 1, 2, ..., 2t$ if $v(x)$ is a codeword, this is equivalent to computing the syndromes of the error polynomial $e(x)$.

2. **Finding the Error-Locator polynomial** $\sigma(x)$:
   The Error-Locator polynomial $\sigma(x) = 1 + \sigma_1 x + ... + \sigma_\tau x^\tau$, has τ roots of the form α^{-i_j} for $j = 1, 2, ..., \tau$. If α^{-i_k} is a root of $\sigma(x)$ for some k, then the error polynomial has a coefficient of 1 in front of its ${i_k}^{\text{th}}$ term. In other words, there was a bit error at the ${i_k}^{\text{th}}$ position.
   There are a few different known algorithms for computing the Error-Locator polynomial, given the $2t$ syndromes, such as the Peterson algorithm and the Berlekamp-Massey algorithm.

3. **Finding the roots of the Error-Locator polynomial**:
   In the binary case, finding the roots of $\sigma(x)$ is enough for us to determine the error polynomial $e(x)$, since the only possible coefficients are 0 (no error at that location) and 1 (error at that location).

4. **Finding the transmitted polynomial** $v(x)$:
   In the binary case, this simply means adding the error polynomial to the received polynomial in order to obtain $v(x)$.

5.2.2 Syndrome Computation

Syndrome computation in the case of BCH codes basically consists in plugging each of the $2t$ roots of the generator polynomial into the received polynomial. Since these roots are found in an extension field $GF(2^m)$, the calculated syndromes are also elements from $GF(2^m)$, and can, therefore, be represented by m bits.
The basic approach to computing these syndromes consists in noticing the fact that we can rewrite the expression for $r(\alpha^i)$ (the ith syndrome) as follows

$$\begin{aligned} r(\alpha^i) &= r_0 + r_1\alpha^i + r_2\alpha^{2i} + ... + r_{n-1}\alpha^{(n-1)i} = \\ & (...(((r_{n-1}\alpha^i) + r_{n-2})\alpha^i) + r_{n-3})... + r_1)\alpha^i + r_0 \ . \end{aligned} \tag{5.1}$$

Equation (5.1) tells us that we can compute the ith syndrome by taking each of the binary coefficients of $r(x)$ starting with the one with the highest degree, adding it to the previous result and then multiplying the result by α^i. This recursive recursive computation can be efficiently realized using a Linear Feedback Shift Register (LFSR).

Example 5.1.
If $n = 2^4 - 1 = 15$, then α must be chosen as a primitive element of $GF(2^4)$, say one of the roots of the irreducible polynomial $X^4 + X + 1$, and we have the following LFSR to compute the first syndrome $r(\alpha)$ ([12]).

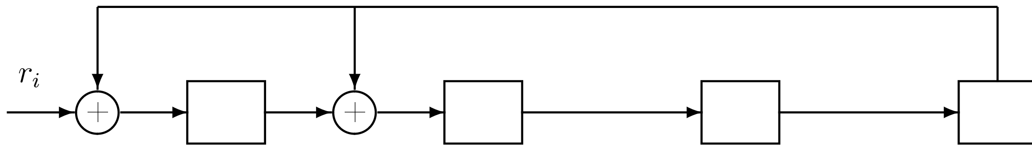


Figure 5.1: LFSR to calculate S_1

At every clock cycle, the contents of the shift-register, which can be viewed as an element of $GF(2^4)$, are multiplied by α. In addition, the introduction of the coefficient r_i into the lowest significant bit (in decreasing order of i's) adds r_i to the previous result. After exactly 15 clock cycles, the contents of the shift-register represent $S_1 = r(\alpha)$.
To compute S_i we need to modify this circuit, so that at every clock cycle, the contents of the shift-register are multiplied by α^i. For instance, to multiply an element of $GF(2^4)$, say $b(\alpha) = b_0 + b_1\alpha + b_2\alpha^2 + b_3\alpha^3$, by α^3, we evaluate

$$\begin{aligned} b(\alpha)\alpha^3 &= (b_0 + b_1\alpha + b_2\alpha^2 + b_3\alpha^3)\alpha^3 = \\ &= b_0\alpha^3 + b_1\alpha^4 + b_2\alpha^5 + b_3\alpha^6 = \\ &= b_0\alpha^3 + b_1(1+\alpha) + b_2(\alpha + \alpha^2) + b_3(\alpha^2 + \alpha^3) = \\ &= b_1 + (b_1 + b_2)\alpha + (b_2 + b_3)\alpha^2 + (b_0 + b_3)\alpha^3 \end{aligned}$$

and thus we can calculate the third syndrome S_3 with the following circuit ([12]):

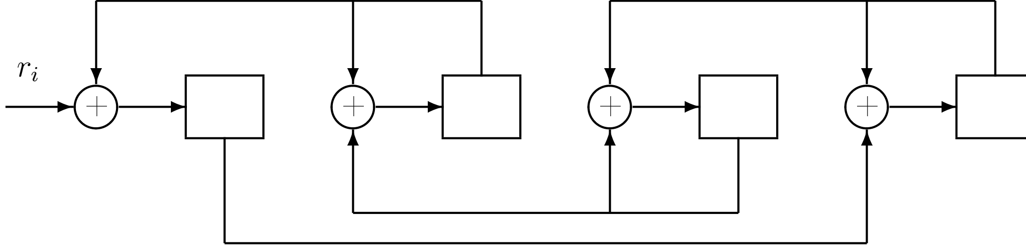


Figure 5.2: LFSR to calculate S_3

Following this procedure, we would need one circuit like the one above, with about the same level of complexity, for each syndrome. However, it is possible to reduce the number of such circuits if we notice that

$$\begin{aligned}(r(\alpha^i))^2 &= (r_0 + r_1\alpha^i + r_2\alpha^{2i} + ... + r_{n-1}\alpha^{(n-1)i}) \\ &= r_0 + r_1\alpha^{2i} + r_2\alpha^{4i} + ... + r_{n-1}\alpha^{2(n-1)i} = r(\alpha^{2i})\end{aligned}$$

Thus we conclude that $S_i^2 = S_{2i}$. Therefore, we can use this fact to reduce the number of LFSR circuits by at least half if we recombine the coefficients of S_i to form S_i^2. To do that, we just need to notice that if we square an arbitrary element of $GF(2^4)$ we obtain

$$\begin{aligned}b(\alpha)^2 &= (b_0 + b_1\alpha + b_2\alpha^2 + b_3\alpha^3)^2 \\ &= b_0 + b_1\alpha^2 + b_2\alpha^4 + b_3\alpha^6 \\ &= b_0 + b_1\alpha^2 + b_2(1+\alpha) + b_3(\alpha^2 + \alpha^3) \\ &= (b_0 + b_2) + b_2\alpha + (b_1 + b_3)\alpha^2 + b_3\alpha^3\end{aligned}$$

Similarly, we see that $b(x)^4 = (b_0 + b_1 + b_2 + b_3) + (b_1 + b_3)\alpha + (b_2 + b_3)\alpha^2 + b_3\alpha^3$. This allows us to extract S_1, S_2 and S_4 from the circuit in Figure 5.1, as shown below

For a $(15, 5, 7)$ BCH code, besides the circuit above, we also need a circuit to compute S_3 and S_6, and another one just to compute S_5. Overall, we would need 3×4 flip-flops, approximately 3×4 XORs for the multiplication circuits and 2 extra XORs for each of the 3 even syndromes. This gives us a total of 12 flip-flops and 18 XORs.

In a general case it is easy to see that we need tm flip-flops. The number of XORs is more difficult to estimate. For the LFSR circuit, we can assume that m XORs are used for each of the t circuits, and for the recombination of the LFSR outputs (forming the individual syndromes) we estimate $\frac{tm^2}{2}$ XORs for all t circuits. These assumptions can be made especially if we consider a slightly different method for computing the syndromes, explained in Appendix B.2.

5.2.3 Finding the Error-Locator polynomial

The most complicated step in decoding a BCH code is finding the Error-Locator polynomial $\sigma(x)$. We find the Error-Locator polynomial by relating the coefficients of $\sigma(x)$ with the $2t$ syndromes we computed in the last step. Since the Error-Locator polynomial

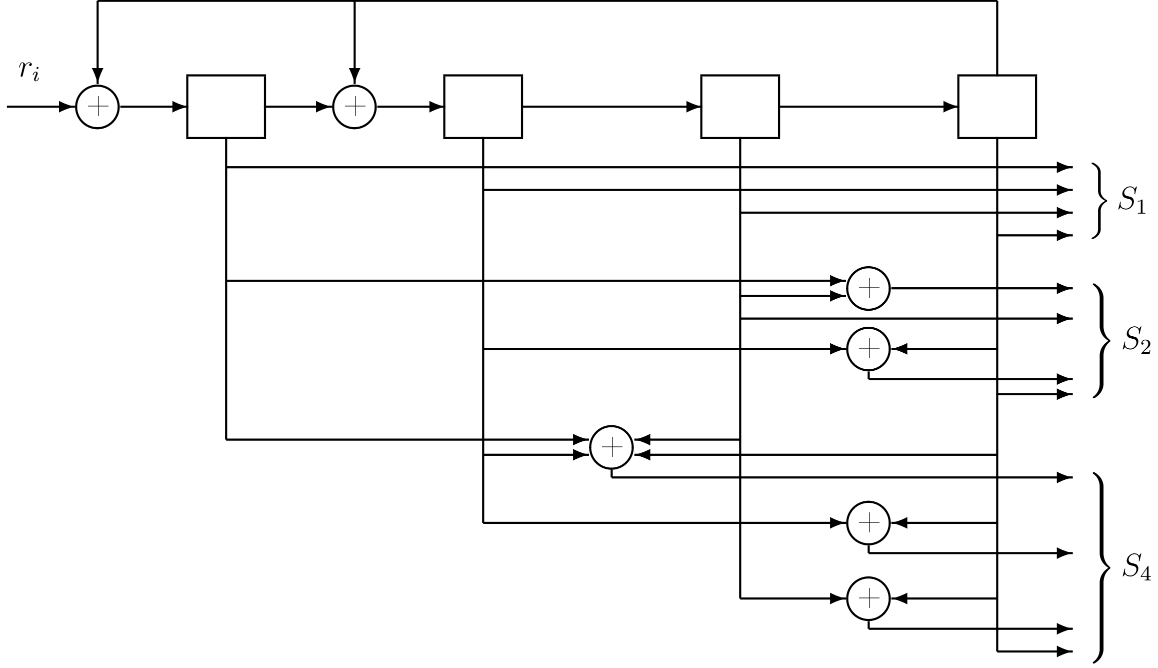


Figure 5.3: LFSR to calculate S_1, S_2 and S_4

has a root of the form α^{-j} if there was an error at the jth position of the transmitted polynomial, we can rewrite it as follows

$$\sigma(x) = 1 + \sigma_1 x + ... + \sigma_\tau x^\tau = (1 - \alpha^{j_1}x)(1 - \alpha^{j_2}x)...(1 - \alpha^{j_\tau}x) \ . \tag{5.2}$$

If we expand it out using Girard relations we obtain

$$(1 - \alpha^{j_1}x)(1 - \alpha^{j_2}x)...(1 - \alpha^{j_\tau}x)$$

$$= 1 - x\xi_1(\alpha^{j_1}, \alpha^{j_2}, ..., \alpha^{j_\tau}) + x^2\xi_2(\alpha^{j_1}, \alpha^{j_2}, ..., \alpha^{j_\tau}) - ... + (-x)^\tau \xi_\tau(\alpha^{j_1}, \alpha^{j_2}, ..., \alpha^{j_\tau}) \ .$$

where $\xi_k(x_1, x_2, ..., x_k)$ is the kth elementary symmetric polynomial on τ variables

$$\xi_k(\alpha^{j_1}, \alpha^{j_2}, ..., \alpha^{j_\tau}) = \sum_{1 \le i_1 < i_2 < ... < i_k \le \tau} \alpha^{j_{i_1} + j_{i_2} + ... + j_{i_k}} = \sigma_k \ .$$

Swapping the sides and rewriting the expressions (see [19]) we obtain

$$\sum_{k=0}^{\tau} \xi_k(\alpha^{j_1}, \alpha^{j_2}, ..., \alpha^{j_\tau})(-x)^k = \prod_{i=1}^{\tau} (1 - \alpha^{j_i}x) \ . \tag{5.3}$$

To obtain a relationship between the syndromes and coefficients of the Error-Locator polynomial we start by differentiating both sides in (5.3) and multiplying them by x

$$\sum_{k=0}^{\tau} k\xi_k(\alpha^{j_1}, \alpha^{j_2}, ..., \alpha^{j_\tau})(-x)^k = \sum_{n=1}^{\tau} \left(-\alpha^{j_n} x \frac{\prod_{i=1}^{\tau} (1 - \alpha^{j_i}x)}{(1 - \alpha^{j_n}x)} \right)$$

$$= -\left(\sum_{n=1}^{\tau} \frac{\alpha^{j_n}x}{1 - \alpha^{j_n}x} \right) \left(\prod_{i=1}^{\tau} (1 - \alpha^{j_i}x) \right) \ .$$

By re-using expression (5.3) and developing the formal power series

$$
\begin{aligned}
&= -\left(\sum_{n=1}^{\tau}\sum_{m=0}^{\infty}(\alpha^{j_n}x)^m\right)\left(\sum_{t=0}^{\tau}\xi_t(\alpha^{j_1},\alpha^{j_2},...,\alpha^{j_\tau})(-x)^t\right)\\
&= -\left(\sum_{m=0}^{\infty}\sum_{n=1}^{\tau}(\alpha^{j_n}x)^m\right)\left(\sum_{t=0}^{\tau}\xi_t(\alpha^{j_1},...,\alpha^{j_\tau})(-x)^t\right) \qquad .\\
&= \left(\sum_{m=0}^{\infty}x^m\sum_{n=1}^{\tau}\alpha^{mj_n}\right)\left(\sum_{t=0}^{\tau}(-1)^{t+1}\xi_t(\alpha^{j_1},...,\alpha^{j_\tau})x^t\right)
\end{aligned}
$$

Now we can plug in the syndromes S_m and the coefficients of the Error-Locator polynomial σ_k

$$
\sum_{k=0}^{\tau}(-1)^{k+1}k\sigma_k x^k = \left(\sum_{m=0}^{\infty}x^m S_m\right)\left(\sum_{t=0}^{\tau}(-1)^{t+1}\sigma_t x^t\right) \quad . \tag{5.4}
$$

Finally, by comparing the coefficient of x^k on both sides of (5.4), we obtain the kth Newton's identity as followss

$$
(-1)^{k+1}k\sigma_k = \sum_{t=1}^{k}(-1)^{k-t+1}S_t\sigma_{k-t} \quad . \tag{5.5}
$$

where we assume that $\sigma_i = 0$ for $i > \tau$. Since we are dealing with binary BCH codes, we can disregard the signs and re-write the expression as follows

$$
\sum_{t=1}^{k}S_t\sigma_{k-t} + k\sigma_k = 0 \ . \tag{5.6}
$$

which yields the following list of relations

$$
\begin{aligned}
S_1 + \sigma_1 &= 0\\
S_2 + \sigma_1 S_1 + 2\sigma_2 &= 0\\
&\vdots\\
S_\tau + \sigma_1 S_{\tau-1} + ... + \sigma_{\tau-1}S_1 + \tau\sigma_\tau &= 0\\
S_{\tau+1} + \sigma_1 S_\tau + ... + \sigma_{\tau-1}S_2 + \sigma_\tau S_1 &= 0\\
S_{\tau+2} + \sigma_1 S_{\tau+1} + ... + \sigma_{\tau-1}S_3 + \sigma_\tau S_2 &= 0\\
&\vdots\\
S_{2t} + \sigma_1 S_{2t-1} + ... + \sigma_{\tau-1}S_{2t+1-\tau} + \sigma_\tau S_{2t-\tau} &= 0 \quad .
\end{aligned} \tag{5.7}
$$

By looking at the last $2t - \tau$ equations we see that finding the coefficients of the error-locating polynomial $\sigma(x)$ is equivalent to looking for a linear recursion on the S_i.

$$
S_k = \sum_{i=1}^{\tau}\sigma_i S_{k-i} \quad \text{for } k > \tau \tag{5.8}
$$

For $\tau \leq t$ there should be only one such linear recursion. It can be found via the Berlekamp-Massey algorithm.

5.2.4 The Berlekamp-Massey Algorithm

The Berlekamp-Massey Algorithm provides a polynomial-time algorithm to find the shortest linear recursion that generates a given stream. In other words, it allows us to find the minimal polynomial (in our case, the Error-Locator polynomial) that generates a recurrent sequence. It is an iterative method, in which one checks if a trial polynomial $\sigma(x)$ will generate the next output in the stream and, if not, applies a correction to it. The procedure is as follows:

We start with a polynomial $\sigma^{(1)}(x)$ that satisfies the first identity from (5.7). This polynomial can also be seen as just a list of coefficients, since it will not be used as a polynomial in its proper sense during the steps of the algorithm. So we start with $\sigma^{(1)}(x) = S_1 x$ and $\sigma_i = 0$ for $i > 1$. Now we check to see if these coefficients satisfy the second identity from (5.7). If they do, we set $\sigma^{(2)}(x) := \sigma^{(1)}(x)$ and we move on to the next identity. If not, we need to add a correction term to $\sigma^{(1)}(x)$ so that it satisfies the second identity. In general, at each iteration r, a new polynomial $\sigma^{(r)}(x)$ is produced. If the coefficients $\sigma^{(r)}(x)$ do not satisfy the $r+1^{\text{th}}$ identity, we first calculate how much the rth discrepancy Δ_r is

$$\Delta_r = S_{r+1} + \sigma_1 S_r + ... + \sigma_{r-1} S_2 + \sigma_r S_1 + r\sigma_r \tag{5.9}$$

In order to eliminate this discrepancy, we look at the previous iterations and find a $\sigma^{(b)}(x)$, for $b < r$ whose discrepancy from the bth identity was also different than 0. Then we normalize the polynomial with respect to its discrepancy Δ_b, multiply it by our new discrepancy Δ_r and shift its coefficients up (or simply multiply it by a power of x) so that its coefficients, when added (or subtracted) to those of $\sigma^{(r)}(x)$, will eliminate the discrepancy. In [1], Berlekamp proved that if we choose the most recent previous iteration in $2\deg(\sigma^{(r)}(x)) < r$ (and $\Delta_r \neq 0$), this process yields the smallest-degree polynomial whose coefficients satisfy the $r+1^{\text{th}}$ identity. Therefore, after $2t$ iterations, we obtain the smallest-degree polynomial that satisfies all identities in (5.7), and thus we have our Error-Locator polynomial.

This iterative process can be systematically summarized in the following steps:

1. Set $\sigma^{(1)}(x) = 1 + S_1 x$. This satisfies the first identity.
2. Then loop through the following steps until $r = 2t$.
3. Compute the discrepancy Δ_r (from Equation (5.9))
   - If $\Delta_r = 0$ set $\sigma^{(r+1)}(x) = \sigma^{(r)}(x)$
   - If $\Delta_r \neq 0$ set $\sigma^{(r+1)}(x) := \sigma^{(r)}(x) + x^{r-b}\Delta_r\Delta_b^{-1}\sigma^{(b)}(x)$ where $\sigma^{(b)}(x)$, the solution at iteration b, is the most recent solution such that $\Delta_b \neq 0$ and $2\deg(\sigma^{(b)}(x)) < b$

Hardware implementation of Berlekamp-Massey's algorithm

The basic idea behind the hardware implementation for this iterative process is to notice that, from one identity in (5.7) to the next one, the syndromes get shifted to the right on the coefficients of $\sigma^{(r)}(x)$). Therefore, we store the coefficients of $\sigma^{(r)}(x)$) in a register

R0 and we shift the syndromes into a serial-in parallel-out shift register SR0, and the contents of the two registers are respectively multiplied at each iteration and added. This gives us the discrepancy Δ_r. The circuit is shown in Figure 5.4.

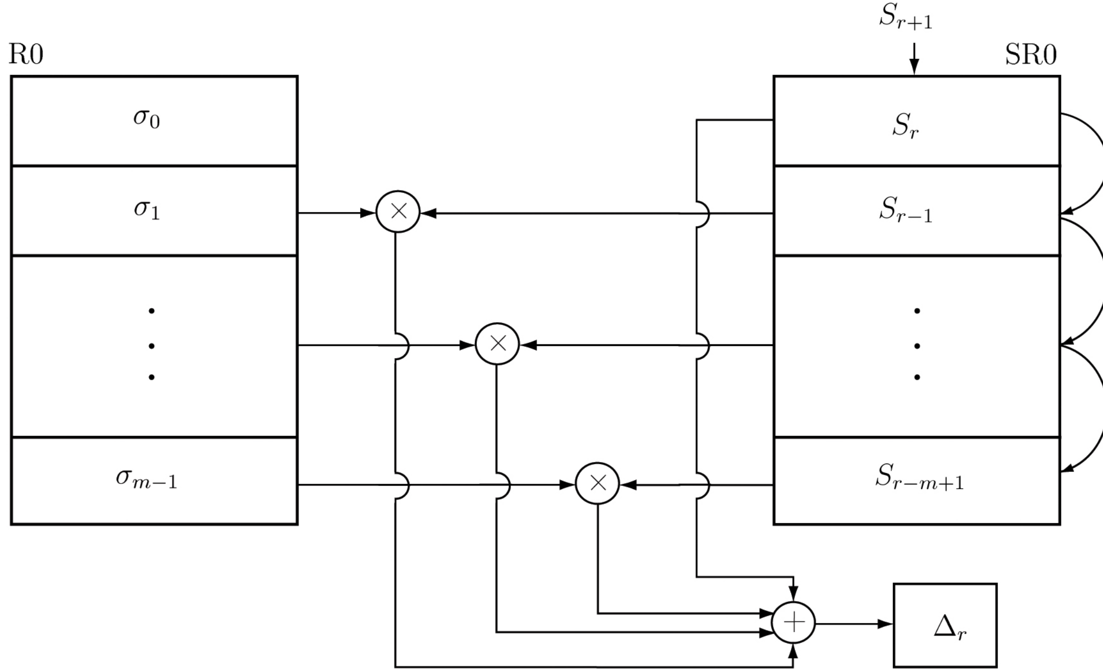


Figure 5.4: Two registers containing the polynomial $\sigma^{(r)}(x)$ and a subset of the syndromes

It is important to notice that each coefficient of $\sigma^{(r)}(x)$ and each syndrome S_i is actually an element of $GF(2^m)$ and therefore can be represented by m bits. This means that each storage unit of the registers can actually store m bits, and whenever we mention a serial operation (like in the case of the serial-in register) we are actually referring to a parallel operation of m bits.

Once we obtain our discrepancy, we can use it to decide whether a new $\sigma^{(r+1)}(x)$ should be loaded into R0 or if the previous $\sigma^{(r)}(x)$ should remain in R0, i.e. $\sigma^{(r+1)}(x) := \sigma^{(r)}(x)$.

Multiplication in $GF(2^m)$

Since we are seeking area efficiency we opt for a serial hardware implementation which takes m cycles to be completed, but uses considerably less combinational logic. We notice that in order to multiply two elements of $GF(2^m)$, say $\beta = b_0+b_1\alpha+b_2\alpha^2+...+b_{m-1}\alpha^{m-1}$ and $\gamma = c_0+c_1\alpha+c_2\alpha^2+...+c_{m-1}\alpha^{m-1}$, we can re-parenthesize the multiplication using Horner's rule as

$$\beta\gamma = \beta(c_0+c_1\alpha+c_2\alpha^2+...+c_{m-1}\alpha^{m-1}) = (...((\beta c_{m-1}\alpha+\beta c_{m-2})\alpha+\beta c_{m-3})\alpha+...)+\beta c_0$$

Therefore, at each clock cycle we multiply β by a coefficient of γ, starting from the most significant one, add it to the previous result, and multiply the expression by α. Therefore, we store the coefficients of β and use the LFSR from Figure 5.1 for the multiplication by α.

Example 5.2.
In $GF(2^4)$, we can implement the serial multiplication algorithm with the following circuit. The coefficients of $c(x)$ are shifted in one-by-one in each cycle, while the coefficients of the operand $b(x)$ are fed in parallel to the circuit and kept stable until the circuit finishes the computation after 4 cycles.

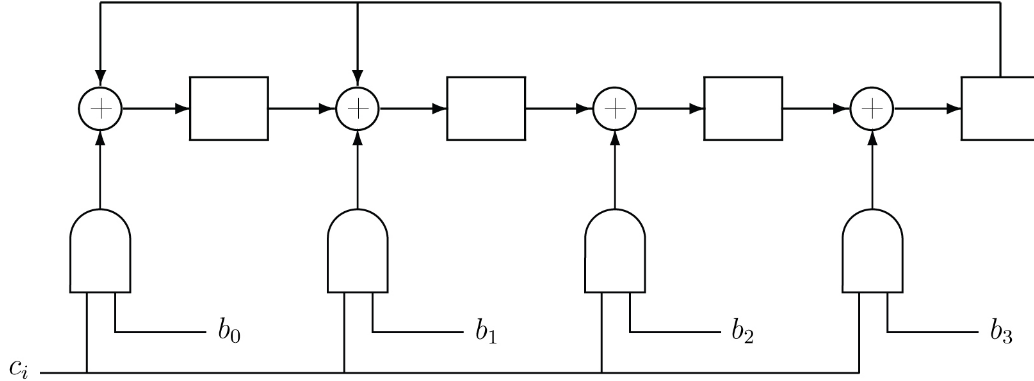


Figure 5.5: Multiplication of two elements in $GF(2^4)$

In order for us to use the above circuit to multiply the coefficients of $\sigma^{(r)}(x)$ by the syndromes, we must choose which of the two factors will represent the serial input γ. Since these factors are stored in registers, and in order to avoid the need for extra parallel-to-serial shift registers, we choose to change the design of the σ register, so that each of its coefficients $\sigma_i^{(r)}$ has each of its m bits cyclically shifted. By doing that, we can simply take the most significant bit out of each $\sigma_i^{(r)}$ and use it as the serial input to the multiplier.

Notice that this modification requires the clock that triggers the cyclic shift CLK1 of the bits to be at least m times faster than the clock CLK0 that shifts in the syndromes S_i. The multiplication will also be triggered by the faster clock (CLK1), allowing the multiplication to at least m cycles before a new syndrome is shifted in. In reality we make CLK1 $4m$ times faster to allow enough time for the first set of multiplications, for the inversion and for a second set of multiplications (see Figure 5.10).

The modified σ-register R0 is represented in Figure 5.6.

Storing a previous $\sigma^{(b)}(x)$

A big hindrance in designing a hardware architecture for the Berlekamp-Massey algorithm is that searching for the most recent previous iteration of $\sigma^{(r)}(x)$ whose discrepancy Δ_r was different than 0, and for which $2\deg(\sigma^{(r)}(x)) < r$, would require storing all previous polynomials $\sigma^{(r)}(x)$. This can be easily done in software but would require several registers in hardware and a mechanism to search for the right previous iteration. Instead, we use only one register R1 which will store a previous iteration of $\sigma^{(r)}(x)$, which we will call $\sigma^{(b)}(x)$, and which will be replaced by $\sigma^{(r)}(x)$ whenever $\Delta_r \neq 0$ and $2\deg(\sigma^{(r)}(x)) < r$.

The degree of $\sigma^{(r)}(x)$, which we will refer to as L_r for short, can be computed by using some combinational logic on the coefficients of $\sigma^{(r)}(x)$. By OR-ing together the m bits of

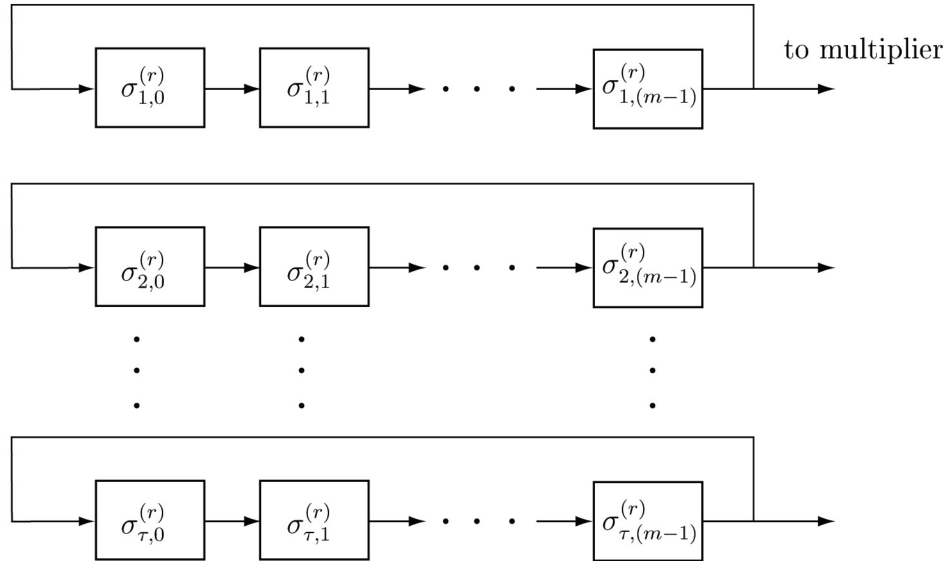


Figure 5.6: Modified register for storing $\sigma^{(r)}(x)$

each coefficient, we find which coefficients $\sigma_i^{(r)}$ are nonzero. Therefore, we can use that information to find the binary form of L_r.

Example 5.3.
For a $(15, 5, 7)$ BCH code, $\sigma(x)$ can be at most a third degree polynomial, and therefore we compute its degree or length with the following circuit.

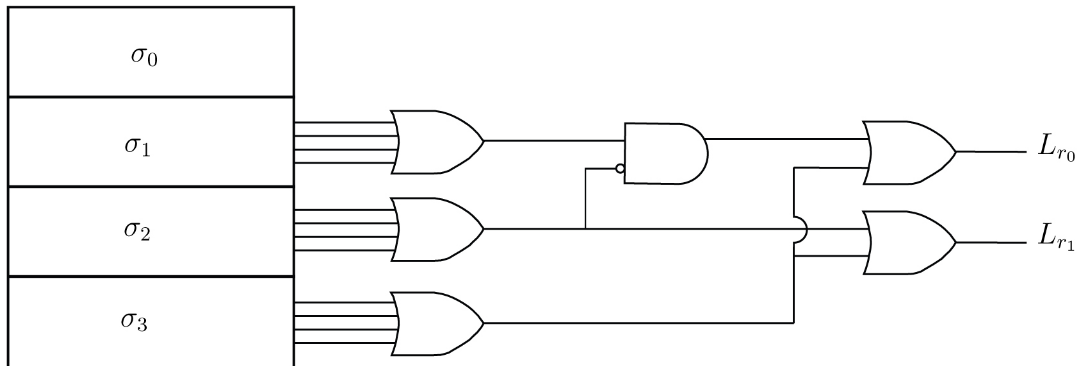


Figure 5.7: How to compute the degree of $\sigma_i^{(r)}$

Multiplying L_r by 2 is trivial and r can be obtained from a simple binary counter, triggered by CLK0. Comparing two binary numbers can again be done using some simple combinational logic, to generate a signal that indicates whether $2L_r < r$.

Example 5.4.
For a $(15, 5, 7)$ BCH code, assuming we have the value of L_r (from Example 5.3), we can perform the comparison $2L_r < r$ with the following circuit:

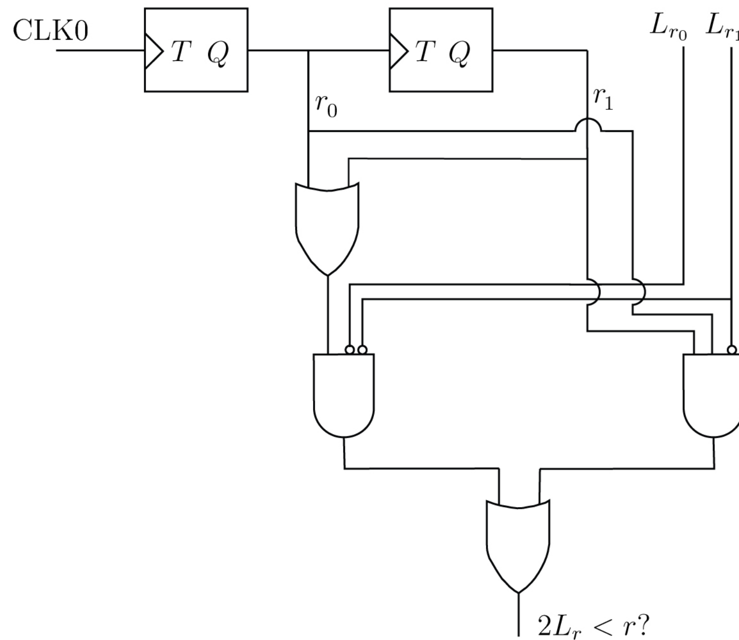


Figure 5.8: Circuit to compare $2L_r$ and r

With a circuit like the one above and the discrepancy Δ_r we can obtain the control signal that will indicate whether $\sigma_i^{(r)}$ should replace the value of $\sigma_i^{(b)}$ in R1, as shown below.

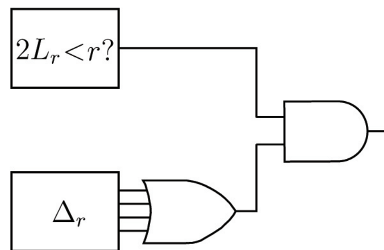


Figure 5.9: Generating the control signal for R1 (register for $\sigma^{(b)}(x)$)

By storing a previous $\sigma^{(b)}(x)$ ahead of time, we can easily use it later on to add the correction term to the current a previous $\sigma^{(r)}(x)$, whenever $\Delta_r \neq 0$.

Updating $\sigma^{(r)}(x)$

Whenever we find that $\sigma^{(r)}(x)$ does not satisfy the $(r+1)^{\text{th}}$ (or simply $\Delta_r \neq 0$) we must add the correction term as follows:

$$\sigma^{(r+1)}(x) := \sigma^{(r)}(x) + x^{r-b}\Delta_r\Delta_b^{-1}\sigma^{(b)}(x)$$

This requires us to also store Δ_b^{-1} and b with $\sigma^{(b)}(x)$. The problem of multiplying by x^{r-b} is approached in the following way. We notice that from the iteration where we store $\sigma^{(b)}(x)$ until the iteration where we use it as a correction term, there are exactly $r-b-1$ iterations. Therefore, at every iteration we multiply the contents of R1 by x, and when we update $\sigma^{(r)}(x)$ with $\sigma^{(r+1)}(x)$, we multiply the correction term once more by x. This can easily be done, since multiplication by x is simply a shift in the coefficients of $\sigma^{(b)}(x)$. This means that R1 will basically work as a shift-register, until its contents are overwritten (because $\Delta_r \neq 0$ and $2L_r < r$). We actually do not need to worry about the polynomial stored in R1 becoming larger than what R1 can store because the algorithm assures us that if $\tau \leq t$ all iterations will yield polynomials of degree at most τ. Therefore, the degree of the correction term will also be at most τ.

Also instead of storing Δ_b^{-1} (with whose computation we will deal later) what we do is we use R1 to store $\Delta_b^{-1}\sigma^{(b)}(x)$, since the multiplication by x at every iteration will not be affected by this change. Notice that the internal cyclic rotation of the bits of the coefficients in R0 will again be useful, since the $\sigma_i^{(r)}$s will also multiply Δ_r^{-1} (prior to its storage), and again we will need one of the factors to be a serial input to the multipliers.

In order to simplify the algorithm for a hardware implementation, we slightly modify the 3$^{\text{rd}}$ step by defining $\sigma^{(r+1)}(x) := \sigma^{(r)}(x) + x^{r-b}\Delta_r\Delta_b^{-1}\sigma^{(b)}(x)$ no matter what Δ_r is. This will not interfere with the algorithm since, whenever $\Delta_r = 0$, the correction term will be canceled, and we will be left with $\sigma^{(r+1)}(x) := \sigma^{(r)}(x)$ as described in the algorithm. By doing this, we can simply update $\sigma^{(r)}(x)$ at every cycle of CLK0.

Example 5.5.
For a $(15, 5, 7)$ BCH code, the overall block diagram for the Berlekamp-Massey algorithm would be like in Figure 5.10.

A few remarks must be made regarding the multipliers on Figure 5.10. Square multipliers were used to identify the "multiplication" between an element of $GF(2^4)$ and a single bit, which is equivalent to ANDing the 4 bits of the element of $GF(2^4)$ and the extra bit. Likewise, whenever three arrows point to the same multiplier, one of the signals is a single bit, and is basically ANDed with the product between the two elements from $GF(2^4)$.

At every iteration, the polynomial $\sigma^{(r)}(x)$ is multiplied by Δ_r^{-1}. However, this result is only stored in R1 if $\Delta_r \neq 0$ and $2L_r < r$. Also, notice that we do not need to store the 4$^{\text{th}}$ coefficient of $\sigma^{(r)}(x)$ in R1, since this coefficient is always 0 (unless right before the algorithm ends, in which case it is useless).

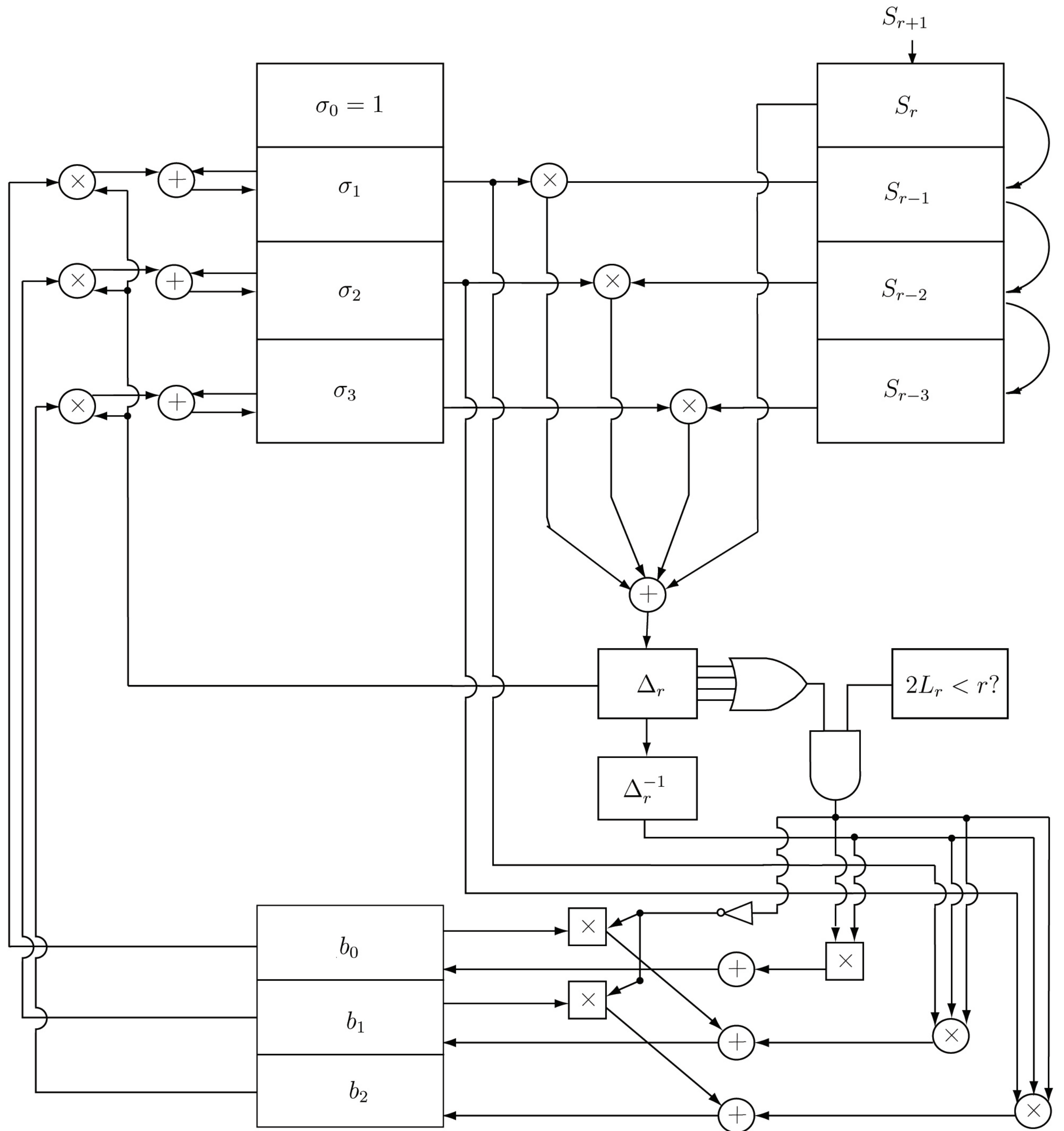


Figure 5.10: Berlekamp-Massey block diagram

Observations about timing

An interesting feature of this hardware design is that it exploits the ability to perform parallel operations to go around problems that may arise when the Berlekamp-Massey algorithm is being coded in software. The main issue is that whenever a correction term is added to $\sigma^{(r)}(x)$ to compute $\sigma^{(r+1)}(x)$, the contents of $\sigma^{(r)}(x)$ must be overwritten.

However, $\Delta_r \neq 0$ (otherwise it would not need to be overwritten), which means that $\sigma^{(r)}(x)$ was itself a candidate for being used as a correction term in future iterations and therefore, if $2L_r < r$, it should be stored in R1. So this basically means that the contents of R0 must be written to R1 while the contents of R1 must be written to R0. We solve this issue very naturally by using the set of multipliers at the output of each of these registers as buffers. They perform the multiplications during one clock cycle of CLK0 (the slowest clock) and whenever CLK0 rises again, they have already finished their computations and therefore can write onto each other with no problem.

Inversion in $GF(2^m)$

Inversion is a critical component in the hardware implementation of the Berlekamp-Massey algorithm. It requires a significant amount of area, and even when optimized representations are used, it grows with $O(m^2 \log(m)$ (see [8]). Pure look-up table based implementations, or combinational implementations are not area-efficient and do not scale at all with operand sizes. Therefore, we resort to an iterative method which uses the fact that, in $GF(2^m)$ as follows.

$$\beta^{-1} = \beta^{2^m-2} = \beta^2\beta^4...\beta^{2^{m-1}} \tag{5.10}$$

Therefore, the main idea is to have a register B containing an element of $GF(2^m)$ and whose contents are squared at every clock cycle. After each squaring operation, they multiply the contents of a register A, and the result is stored in A. So A accumulates the results of the repeated multiplications by the content of B. In Figure 5.11, we can see the schematics for a $GF(2^4)$ inverter.

Circuit complexity

The Berlekamp-Massey implementation is by far the component which takes up the most area, and therefore, for longer BCH codes, its complexity will determine the complexity of the decoder.

To estimate the number of gates, we start by noticing that each serial multiplier, implemented as in Figure 5.5, will use m flip-flops, m AND gates ($2m$ NANDs) and approximately $\frac{3}{2}m$ XOR gates. In our implementation for $GF(2^4)$, shown in Figure 5.10, we have 8 such multipliers, and in general we expect to have $3t-1$.

Our most complex block, the inverter, uses two registers ($2m$ flip-flops) and has an array of m^2 AND gates. Surprisingly, the most expensive operation ends up being the additions shown on the bottom of Figure 5.11. We estimate the number of XOR gates to be $\frac{m^3}{4} - \frac{m^2}{4}$ (see Appendix B.3).

For the registers, we need tm flip-flops and $3tm$ NANDs for each of our two registers R0 and R1, and we need $(t+1)m$ flip-flops for the shift-register SR0. Moreover, we need $(2t-1)m$ extra ANDs and $(2t-1)m+t$ extra XORs, for the bit multiplications and additions, respectively. The other gates in the circuit (such as the ones used to determine whether $2L_r < r$) are not counted here, since they do not contribute significantly to the overall complexity.

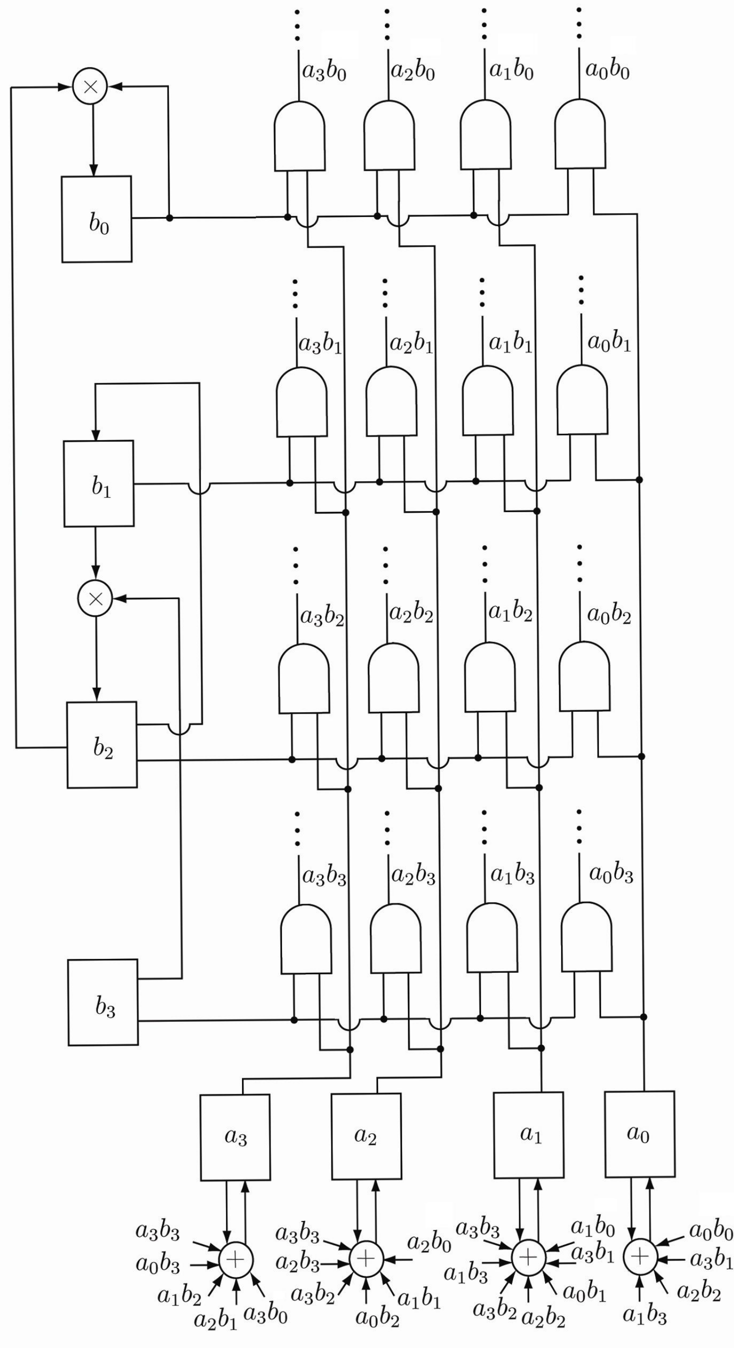


Figure 5.11: Circuit to perform inversion in $GF(2^4)$

5.2.5 Correcting errors

Now that we have the Error-Locator polynomial, it should be relatively simple to correct the errors in our received message, especially in the binary case, where all we have to do is flip the wrong bits. However, the roots of $\sigma(x)$ are the inverses of the elements of $GF(2^m)$ corresponding to the positions where errors occurred. In order to correct these errors in both a time and area efficient way, we use Chien's search algorithm.

The Chien search is based on the simple fact that, since $\alpha^n = 1$, $\alpha^{-i} = \alpha^{n-i}$, where n, the block size, is given by $n = 2^m - 1$. Therefore, if α^i is a root of our Error-Locating polynomial $\sigma(x)$, it means that there was an error at the $(n-i)^{\text{th}}$ position of $r(x)$, and therefore, in the binary case, the $(n-i)^{\text{th}}$ bit should be flipped. Therefore, this gives us a systematic way of testing roots and modifying the received polynomial at the same time: we check whether $\alpha^0, \alpha^1, \alpha^2, ..., \alpha^n$ are roots in this order, and at the same time, we modify the bits at the $n^{\text{th}}, (n-1)^{\text{th}}, (n-2)^{\text{th}}, ..., 1^{\text{st}}$ positions, in this order.

A circuit that does this operation may simply consist of LFSRs that perform multiplication by $\alpha^1, \alpha^2, ..., \alpha^\tau$, initially loaded with $\sigma_1, \sigma_2, ...\sigma_\tau$ respectively. If, after the ith multiplication, the contents of each of these LFSRs are added, we obtain the value of $\sigma(\alpha^i) - 1$. Therefore, if $\sigma(\alpha^i) = 0$ we can change the $(n-i)^{\text{th}}$ position of $r(x)$. This is actually quite convenient, since our received vector $r(x)$ is shifted into the system starting from the most significant bit (highest degree coefficient) to the least significant bit. The operation of the Chien search block can be shown in the following figure.

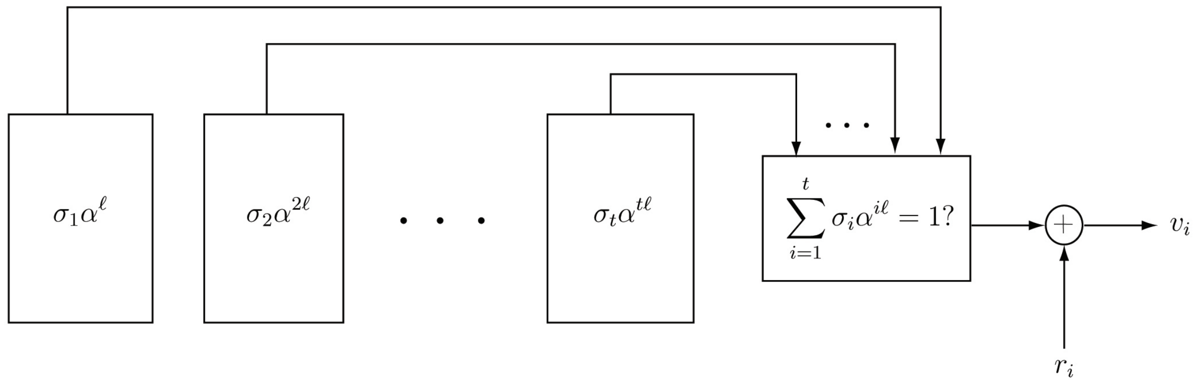


Figure 5.12: Circuit to perform Chien's search algorithm

Each of the blocks in Figure 5.12 is actually an LFSR that performs multiplication by α^i, assuming that they can be initially loaded with σ_i. For instance, the circuit in Figure 5.13 can be used as the α^3 multiplier. It is constructed based on the fact that

$$\begin{aligned} b(x)\alpha^2 &= (b_0 + b_1\alpha + b_2\alpha^2 + b_3\alpha^3)\alpha^2 = \\ &= b_0\alpha^2 + b_1\alpha^3 + b_2\alpha^4 + b_3\alpha^5 = \\ &= b_0\alpha^2 + b_1\alpha^3 + b_2(1+\alpha) + b_3(\alpha + \alpha^2) = \\ &= b_2 + (b_2 + b_3)\alpha + (b_0 + b_3)\alpha^2 + b_1\alpha^3 \end{aligned}$$

The number of flip-flops is m for each of the $2t$ multipliers. However, as we increase i, the number of additions needed to perform consecutive multiplications by α^i also increases. In Appendix B.4, we estimate the number of XOR gates needed to be $mt^2 + \frac{mt}{2}$. For the summation block we need $m(t-1)$ XORs and about $2m$ NANDs. Also, we assume that there exists a shift-register of length n for the received vector components r_i, so that it can be synchronized with the output of the Chien's Search circuit. This means that we have $n = 2^m - 1$ extra flip-flops.

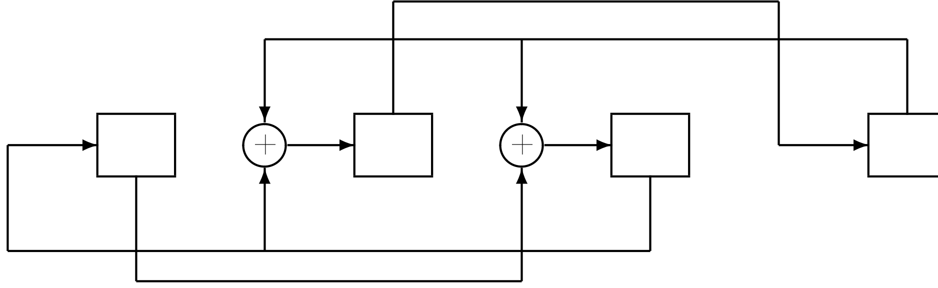


Figure 5.13: Circuit to perform consecutive multiplications by α^2

5.3 BCH Decoder Implementation Results

The main result sought by this implementation is compactness. In this sense the overall circuit complexity must be computed in order for us to be able to assess its implementability on ASIC and on an FPGA. By summing the estimates for the gate counts for the most significant blocks, we can then calculate the hardware complexity. If HC refers to the hardware complexity, we have:

HC(decoder) = HC(Syndrome Computation)+HC(Berlekamp-Massey)+HC(Chien's Search)

The individual hardware complexities per block are shown in the following table:

Table 5.1: Hardware Complexity by block

	FF	XOR	NAND
Syndrome Computation	tm	$\frac{tm^2}{2}$	-
Berlekamp-Massey			
Multiplier $(3t-1)$	m	$\frac{3m}{2}$	$2m$
Inverter (1)	$2m$	$\frac{m^3-m^2}{4}$	$2m^2$
Register (2)	tm	-	$3tm$
Shift-Register (1)	$(t+1)m$	-	-
Addition $(3t-1)$	-	m	-
Bit-Multiplication $(2t-1)$	-	-	$2m$
Chien's Search	2^m-1	$t^2m+\frac{tm}{2}$	$2m$

The 2^m-1 flip-flops attributed to the Chien's Search may seem a little out of place. However, it is necessary that we store the received vector $r(x)$ somewhere, so that after finding the error positions we can correct the errors. This is based on the general scheme for a Code-Offset construction for a secure sketch proposed in [4] (see Section 4.4).

According to this scheme, the verifier provides a helper string to the system containing a secure sketch and a seed for the strong extractor (hash function). Since we will not be dealing with the implementation of the Hash function, we will assume, for simplicity, that the helper string is only the secure sketch. Compared to the BCH decoder the

implementation of a hash function is simple. We refer to [10] for a hardware construction of *universal hash functions*.

A secure sketch constructed via a code-offset method basically consists of $w + C$, where w is the noiseless version of the output of the PUF for a given challenge x and C is a random codeword from a code defined on the same metric space as the output of the PUF. The PUF block receives the challenge x and computes $w' = \mathsf{PUF}(x)$, where w' is a noisy version of w. Both w' and the secure sketch $w+C$ are fed into the Fuzzy Extractor block, as shown in Figure 5.14.

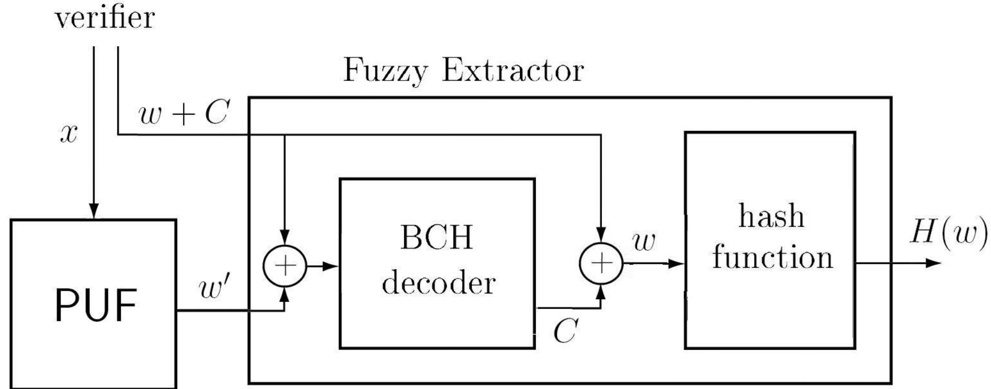


Figure 5.14: Fuzzy Extractor (Reproduction procedure) overall system

Inside the Fuzzy Extractor block, w' is added to the secure sketch $w + C$, yielding $C' = C + (w + w')$, a noisy version of C. C' can therefore be run through a decoder (in our case, a BCH decoder) which outputs C, which can, in turn, be added to the secure sketch, allowing the recovery of w. Once the noiseless w is obtained, a hash function can then extract randomness from it, assuring that the final output of the Fuzzy Extractor block, $H(w)$ looks uniformly random to an observer.

The only assumption made about the PUF is that, given a challenge vector x, it can output the PUF response $w' = \mathsf{PUF}(x)$ serially. If we assume that $\mathsf{PUF}(x)$ is parallel instead, then we need to serialize it in order to be able to use the design presented. This requires an extra $2^m - 1$-bit register, which takes up a considerable area. However, when we carefully analyze the actual implementation of the BCH decoder, we see that there is a possibility of avoiding this extra storage. This can be done if we notice that the output of the Chien's Search, before being XORed with the input to the system $C + (w + w')$, simply consists of $w + w'$. This means that an alternative way for obtaining w would be to simply XOR the Chien's Search output to w'. Therefore, we use the fact that w' was already stored, and we find w by computing $w + (w + w')$. This is helpful because, in this case, we do not need to store the secure sketch $w + C$, and the overall circuit complexity remains unchanged. This modified scheme can be seen in Figure 5.15.

Therefore, we can assume that the gate counts on Table 5.1 holds for both a serial and a parallel PUF output. Based on the values of the table, we can see that the overall

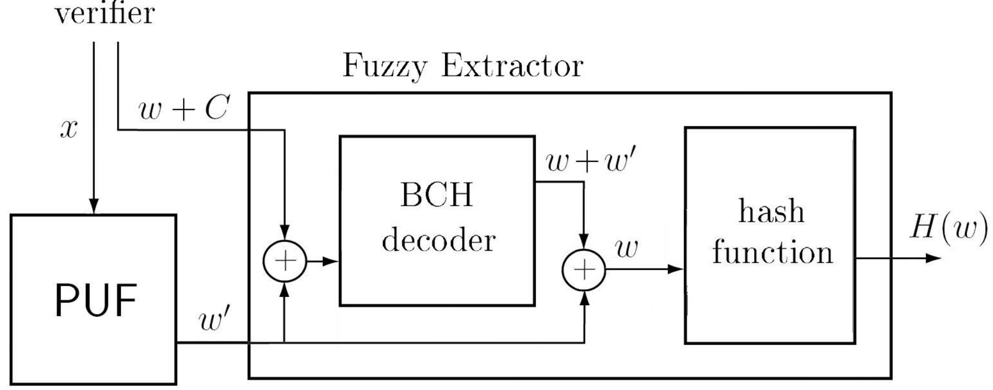


Figure 5.15: Modified Fuzzy Extractor

gate count estimate of this circuit is given by:

$$[2^m + (7t+2)m - 1]\,\text{FF} + \left[\frac{m^3}{4} + m^2\left(\frac{2t-1}{4}\right) + m\left(8t - \frac{5}{2}\right)\right]\text{XOR} + \\ + \left[2m^2 + m(3t+6)\right]\text{NAND} \qquad (5.11)$$

which allows us to estimate the hardware complexity of the decoder to be:

$$\text{HC(decoder)} = (2^m + 7tm)\text{FF} + \left(\frac{m^3}{4} + \frac{tm^2}{2}\right)\text{XOR} + (2m^2 + 3tm)\text{NAND} \qquad (5.12)$$

For the specific case of a $[15, 5, 7]$ binary BCH code ($t = 3$), subject of most of the examples here presented, with Equation (5.11), we obtain a gate count of 107 FF, 122 XOR and 80 NAND.

The time complexity of the circuit can be easily estimated if we notice that it takes us $m - 1$ cycles to compute the inverse, and since this happens simultaneously with the multiplications, which take m cycles, the multiplications will determine the time complexity of the overall circuit. Therefore, since we have $2t$ iterations of the Berlekamp-Massey algorithm, with 3 sets of multiplication in each iteration, we need $6tm$ clock cycles to find $\sigma(x)$. The syndrome computations, which happen before Berlekamp-Massey can start, contribute with $n = 2^m$ clock cycles, and the Chien's search at the end contributes with another 2^m clock cycles. Therefore the overall time complexity is given by:

$$2^{m+1} + 6tm$$

The steps of the design here presented were implemented in VHDL and simulated using Mentor Graphics ModelSim™ tool for correctness. The BCH decoder for a $[15, 5, 7]$ code was synthesized into two different FPGAs, with the following resource utilizations:

- Spartan 3: 118 out of 1920 slices (6%)
- Virtex 2Pro: 118 out of 13696 slices (0.8%)

These results show that this implementation is fairly reasonable in terms of FPGA usage. However, in order for this implementation to be useful, we must consider block sizes of at least 100 bits ($m \geq 7$). Considering the dominant term of our hardware complexity to be m^3 (which is reasonable for $m < 10$), this change would increase the FPGA usage by a factor of at most 8. This means that we should still be able to fit the design on a simple Spartan 3 board, although it would become impractical to use it in an FPGA IP protection setting.

Chapter 6

A new approach to PUF-based hardware authentication

The authentication scheme provided by the integration of fuzzy extractors and PUFs is efficient and allows for the development of secure protocols. However, a considerable amount of information entropy leaks with the publication of the helper string. For practical applications, this information leakage can be made negligible by using a code-offset construction over a large metric space with a very dense code (which nearly achieves the Singleton bound).

In this part of the report, we propose a new scheme for hardware authentication using PUFs that does not require the publication of a hint in the form of a helper string.

6.1 Polynomial as the secret

By following the model for a PUF described in Section 2.1.2, we start by pointing out the fact that the strong non-linearity of the function f makes it very difficult to characterize the input-output pairs of a function. In other words, there is no simple function whose behavior mimics the behavior of the PUF function for most values.

If we were to plot the points generated by the PUF function, due to its non-linearity, we might see something like Figure 6.1a. Our idea is to assign a low degree polynomial as an authentication key (secret) to each device. We will first assume the existence of a low degree polynomial which passes through a number of expected outputs of the PUF which is larger than its degree, as exemplified by Figure 6.1b. We then will restrict the domain of the PUF function to only those challenges whose expected output of the PUF function lies on this low degree curve (Figure 6.1c). Using (2.2) we can say that for each challenge in this restricted domain, the response from the PUF will lie on this low degree polynomial with probability at least $1 - \epsilon$. We will then be able to analyze methods for recovering this polynomial when given noisy responses from the PUF. Later, we will return to our assumptions on the existence of this polynomial, and look for the probability of finding such a polynomial.

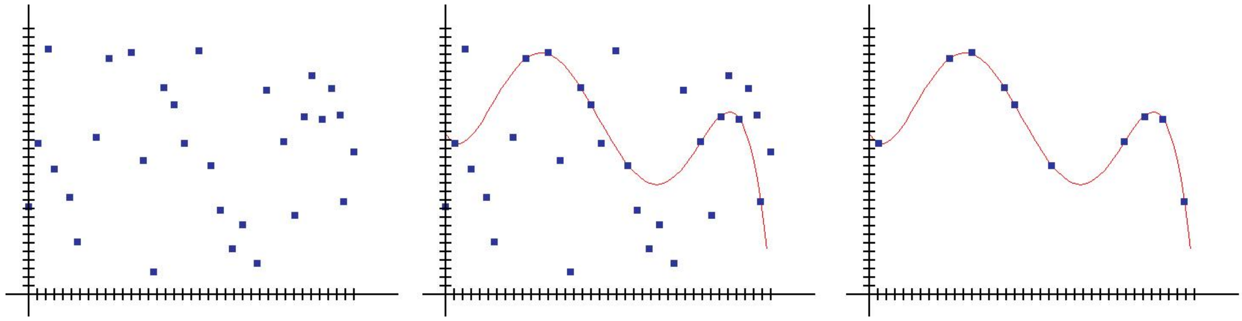


Figure 6.1: Three steps to assign a polynomial to a device

6.2 Polynomial Interpolation

Our proposed authentication scheme is as follows. A verifier provides challenges x from the restricted domain of the PUF function to the PUF device which computes each $\mathsf{PUF}(x)$. Now the device has a set of points $(x, \mathsf{PUF}(x))$. Techniques for polynomial interpolation can then be used in order for the device to find the coefficients of the interpolating polynomial, which can be used to claim its identity. This illustrates the fact that the secret, in this case the coefficients of the interpolating polynomial, does not need to be stored in the device. Instead it is generated at every authentication session.

For this protocol to make sense, the device must be able to find the interpolating polynomial. Therefore, in this section, we will be concerned with the problem of finding the polynomial of lowest degree which passes through a set of k points. We are guaranteed to find a unique polynomial of degree at most $k-1$, and if we do not bound the degree of the polynomials, an infinite number of polynomials of higher degrees which pass through these k points can be found.

We remark that this problem is different from the problem of searching for a polynomial which approximates a given function to a certain level of accuracy. In this case, the function is known and information about its derivatives can be used to find an approximating polynomial. In our application, we attempt to model the PUF function but have no information on its derivatives. We therefore use polynomial interpolation instead of polynomial approximation.

We define the degree of the polynomial to be the highest power of x with a non-zero coefficient. We may view $p(x)$ as a map, $\mathbb{F} \to \mathbb{F}$ via $x \to p(x)$ where $x \in \mathbb{F}$. We remark that the range of $p(x)$ may be less than the co-domain (all of $\mathbb{F}$), which can be seen by setting $p(x) = 0, \forall x \in \mathbb{F}$.

We call two polynomials $p(x), p'(x)$ equivalent over $\mathbb{F}$ if for all elements of $\mathbb{F}$, the two polynomials evaluate to the same field element.

Lemma 6.1. *Over a finite field of size q, a polynomial of degree greater than or equal to q is equivalent to a polynomial of degree at most $q-1$.*

Proof. Let $p(x)$ denote the polynomials with $deg(p(x)) \geq q$. We then evaluate $p(x)$ at each element $g \in \mathbb{F}$. We thus get q points of the form $(g, p(g))$. These q points define a unique polynomial of degree at most $q-1$, call it $p'(x)$. Since $p(x) = p'(x), \forall x \in \mathbb{F}$, we

say these polynomials are equivalent over $\mathbb{F}$. We can find this equivalent polynomial by taking congruency class mod $\prod_{i=0}^{q-1}(x-x_i)$ where x_i ranges over the field elements. $\square$

6.3 Lagrange Interpolation

We now examine the interpolation method due to Lagrange. This is the simplest and most common technique for polynomial interpolation and it will allow us to find the lowest-degree polynomial through a given set of points. This technique assumes unique x values, which matches our PUF application. We start by setting up a Lagrange coefficient for each point. We let

$$L_j(x) := \frac{\prod_{i\neq j}(x-x_i)}{\prod_{i\neq j}(x_j-x_i)} \tag{6.1}$$

We see that $L_j(x)$ is 0 if $x = x_i$ and $i \neq j$, which can be seen since the term $(x-x_i)$ of the numerator will be 0. If $x = x_j$ then this value equates to 1. Therefore, if we multiply L_j by y_j, when $x = x_j$ this product will be y_j. By repeating this procedure for all j, we have an interpolating polynomial

$$P_m(x) := \sum_{j=0}^{m} y_j L_j(x) \tag{6.2}$$

Equation (6.2) is a Lagrange Polynomial for the set of m points. We immediately see the difficulty in using this construct. We now provide a small example of this method.

Example 6.1.
Consider the problem of finding a polynomial of smallest possible order which passes through the three points, $\{(1,2),(3,2),(4,1)\}$ over $\mathbb{Z}_5$, the integers mod 5.

$$L_1(x) := \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} = \frac{(x-3)(x-4)}{(1-3)(1-4)} = \frac{x^2-7x+12}{6} = \frac{x^2+3x+2}{1}$$

To write $L_1(x)$ as an element of $\mathbb{Z}_5[x]$, we multiply $L_1(x)$ by the inverse of 1 which is 1 since $1*1 = 1 \equiv 1(\pmod 5)$. Therefore

$$L_1(x) = \frac{1}{1}L_1(x) = x^2+3x+2$$

We now perform a the same procedure to determine $L_2(x)$ and $L_3(x)$.

$$L_2(x) := \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} = \frac{(x-1)(x-4)}{(3-1)(3-4)} = \frac{x^2-5x+4}{-2} = \frac{x^2+4}{3}$$

Multiplying $L_2(x)$ by 2, which is the inverse of 3, we have that

$$L_2(x) = \frac{2}{2}L_2(x) = 2x^2+3$$

We now do the same for $L_3(x)$ which corresponds to the point $(4,1)$.

$$L_3(x) := \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} = \frac{(x-1)(x-3)}{(4-1)(4-3)} = \frac{x^2-4x+3}{3} = \frac{x^2+x+3}{3}$$

Therefore

$$L_3(x) = \frac{2}{2}L_3(x) = 2x^2 + 2x + 1$$

We now create our interpolating polynomial as shown in Equation (6.2)

$$P_3(x) := \sum_{j=0}^{3} y_j L_j(x) = 2L_1(x) + 2L_2(x) + 1L_3(x)$$

$$P_3(x) = (2x^2 + x + 4) + (4x^2 + 1) + (2x^2 + 2x + 1) = 3x^2 + 3x + 1$$
$$P_3(1) = (3 + 3 + 1) = 2 \pmod 5$$
$$P_3(3) = (27 + 9 + 1) = 2 \pmod 5$$
$$P_3(4) = (48 + 12 + 1) = 1 \pmod 5$$

We have that the three points match and hence this $P_3(x)$ interpolates the points. During the construction, we could make sure we were construction the $L_j(x)$ correctly. For example, we could check $L_2(x)$ and make sure $L_2(1) = 2(1)+3 = 0$, $L_2(4) = 2(16)+3 = 0$ and $L_2(3) = 2(9) + 3 = 1$. This shows the term $L_2(x)$ is the only term that contributes to $P_3(x)$ when $x = 3$, which corresponds to our second point.

6.3.1 Restrictions on Lagrange Interpolation

A major drawback to Lagrange Interpolation is its inability to handle noise. In the construction, we create polynomials L_i which correspond to a specific x value. If these x values are known, as will be the case with our protocol, the errors, or the noise, may only be present in the y values. There is no built in error-correcting mechanism that will allow for the detection of an incorrect y value. If there was additional information, such as the degree of the lowest-degree polynomial that passes through k points, then there is an obvious procedure that can be used to find such a polynomial. For example, suppose we are looking for an interpolating polynomial for all k points with degree at most $k - 2$ (extra information). We proceed by selecting all k subsets of size $k - 1$, and looking for the lowest degree polynomial that goes through them. Then for each polynomial one can check how many points it goes through. This procedure becomes tedious and expensive as the difference between the number of points and the degree of the polynomial increases. This however, motivates the search for algorithms which find such a polynomial quickly given points which may have incorrect y values. We will now present two approaches to this problem, which we will refer to as polynomial interpolation "with lies".

6.4 Polynomial interpolation "with lies"

The method provided by Lagrange Interpolation is a straightforward way of obtaining the lowest degree polynomial that goes through a set of points. However, when we consider the setting where some of those points are misplaced or, in other words, there is noise in the data, it can do very little for us.

When we consider the fact that for one to perform Lagrange Interpolation and find a polynomial of degree at most d, all we need are $d+1$ points of the actual polynomial, it is intuitive to think that the introduction of some *redundancy* in the data points could possibly help us find the original polynomial, even in the presence of noise. For example, if instead of only $d+1$ points we have $d+1+r$ points, then, even if there are up to r misplaced data points, we should still be able to find a subset of $d+1$ correct points that allow us to recover the original polynomial via Lagrange Interpolation. Nonetheless, this method requires us to check at most $\binom{d+1+r}{d+1}$ possible sets of $d+1$ points. Other methods must be introduced in order to allow for a more efficient way to find the actual polynomial.

Interpolating with the aid of the Discrete Fourier Transform

From the comparison between the problem of recovering a polynomial when the data points have some noise and the problem of transmitting information over a noisy channel comes the idea of applying solutions from the field of communications to polynomial recovery.

In a communication system, a common way to "prepare" the information for transmission and for the addition of noise is to predefine a *transmission bandwidth*. This way, the receiver "knows" where to look for the information. Similarly, for the polynomial recovery problem, we will only send nonzero information for a finite set of frequencies, or a bandwidth.

More specifically, if we have a polynomial $p(x)$ of degree $k < q-1$ over $GF(q)$, we will evaluate this polynomial at $n = q-1$ "consecutive" points of $GF(q)$, $\alpha^0, \alpha^1, ..., \alpha^{n-1}$, where α is any primitive element of $GF(q)$. Once each $V_i = \alpha^i$ is transmitted, with some noise added to it, the receiver may regard it as the coefficients of another polynomial $V(x)$, of degree $n-1$. This is equivalent to the Discrete Fourier Transform on a finite field (see Section 3.4.2).

The Discrete Fourier Transform when applied to finite fields demonstrates the duality between the points on a polynomial and its coefficients. Simply put, the coefficients of our initial polynomial $p(x)$, when multiplied by n will yield the values of $V(\alpha^i)$, except that in reverse order. Since $k < q-1$, the last few coefficients of $p(x)$, namely $v_{k+1}, v_{k+2}, ..., v_{n-1}$ can all be regarded as 0. This yields an important result: if $p(x)$ has degree at most k, then $V(\alpha^i) = 0$ if $i \leq n-k$. This can be interpreted as having a finite bandwidth containing information. If we have nonzero data at those points, we must be dealing with noise. This can be better understood with the following example.

Example 6.2.
Consider the polynomial $p(x) = 3 + 2x + x^2$ over $\mathbb{Z}_7$. By picking $\alpha = 3$ (since $|3| = 6$), we calculate the each $V_i = p(\alpha^i)$, as shown below in Table 6.4.

α^i	$3^0 = 1$	$3^1 = 3$	$3^2 = 2$	$3^3 = 6$	$3^4 = 4$	$3^5 = 5$
$V_i = p(\alpha^i)$	6	4	4	2	6	3

This allows us to form the polynomial $V(x) = 6 + 4x + 4x^2 + 2x^3 + 6x^4 + 3x^5$. We now evaluate it at α^i, for $\alpha = 1, 2, ..., n$:

α^i	$3^1 = 3$	$3^2 = 2$	$3^3 = 6$	$3^4 = 4$	$3^5 = 2$	$3^6 = 1$
$V(\alpha^i)$	0	0	0	6	5	4

Notice that $6 \cdot p(x) = 4 + 5x + 6x^2 + 0x^3 + 0x^4 + 0x^5$, which exemplifies the duality between values of the polynomial when evaluated at certain points and its coefficients.

In general, if a polynomial $p(x)$ of degree at most k is evaluated at α^i, for $i = 0, 1, 2, ..., n-1$, and this information is transmitted via a noisy channel, one can build the polynomial $V(x)$ and evaluate it at α^i, for $i = 1, 2, ..., n$, and check whether $V(\alpha^i) = 0$ for $i = 1, 2, ..., n-k$. In this case, there are two possibilities: either the values of $p(x)$ were correctly transmitted or enough points were incorrectly transmitted to make it look like another polynomials of degree k was transmitted. One can easily show that the latter case is only possible if $n-k$ or more of the points were incorrectly transmitted, and in general, we will assume the former case.

If $V(\alpha^i) \neq 0$ for some i, $1 \leq i \leq n-k$, it is guaranteed that some of the points were incorrectly transmitted. In this case, we simply assume that the received vector was:

$$V'(x) = V(x) + e(x) \tag{6.3}$$

where $e(x) = e_0 + e_1 x + ... + e_{n-1}x^{n-1}$ is the error vector. Now, if we evaluate this received polynomial at each α^i, $1 \leq i \leq n-k$, we use the fact that $V(\alpha^i) = 0$ to obtain the following equations:

$$\begin{aligned} V'(\alpha) &= e_0 + e_1\alpha + ... + e_{n-1}\alpha^{n-1} \\ V'(\alpha^2) &= e_0 + e_1\alpha^2 + ... + e_{n-1}\alpha^{2(n-1)} \\ &\vdots \\ V'(\alpha^{n-k}) &= e_0 + e_1\alpha^3 + ... + e_{n-1}\alpha^{3(n-1)} \end{aligned}$$

And we need to solve the following system:

$$\begin{pmatrix} 1 & \alpha & \alpha^2 & \alpha^3 & \cdots & \alpha^{n-1} \\ 1 & \alpha^2 & \alpha^4 & \alpha^6 & \cdots & \alpha^{2(n-1)} \\ 1 & \alpha^3 & \alpha^6 & \alpha^9 & \cdots & \alpha^{3(n-1)} \\ \vdots & & & & \ddots & \vdots \\ 1 & \alpha^{n-k} & \alpha^{2(n-k)} & \alpha^{3(n-k)} & \cdots & \alpha^{(n-k)(n-1)} \end{pmatrix} \begin{pmatrix} e_0 \\ e_1 \\ \vdots \\ e_{n-1} \end{pmatrix} = \begin{pmatrix} V(\alpha) \\ V(\alpha^2) \\ \vdots \\ V(\alpha^{n-k}) \end{pmatrix} \tag{6.4}$$

This systems has a large number of solutions, and therefore is not helpful. By assuming that we only had up to $n-k$ errors, we can then simplify the matrix in Equation (6.4) to a $(n-k)\times(n-k)$ matrix, which would yield a unique solution. However, for any choice of $n-k$ positions for the errors we would obtain a (possibly) different solution, giving us a total of $\binom{n}{n-k}=\binom{n}{k}$ solutions. If we assume that we had exactly t errors, there will be $\binom{n-t}{n-k-t}=\binom{n-t}{k}$ possible combinations of $n-k$ error positions that will give us the right solution. In [16], it is shown that a "wrong" solution to this system can be given by at most $\binom{t+k-1}{k}$ choices of the $n-k$ error positions. Therefore, in order for a "majority" argument to work here, we must have $\binom{t+k-1}{k}<\binom{n-t}{k}$ which implies that we must have

$$t+k-1<n-t\Rightarrow t<\frac{n-k+1}{2}\ .$$

Therefore, we can recover our original polynomial if we have up to $\lfloor\frac{n-k}{2}\rfloor$ wrong data points. In the next section, we consider another method for performing polynomial interpolation with lies: *Sudan's list-decoding algorithm.* This new method has the advantage that it does not require the number of errors to be at most $\lfloor\frac{n-k}{2}\rfloor$.

6.5 Sudan's algorithm

In this section we consider Sudan's randomized algorithm for constructing a list of all univariate polynomials in the variable x of degree at most δ over $\mathbb{F}$ which agree with a set of n points from $\mathbb{F}^2$ in at least τ places. Moreover, if certain constraints are met, this algorithm runs in polynomial time in n. We first define and describe the necessary input to the algorithm. We then provide the algorithm, followed by more detailed analysis. We follow the approach in [17] closely.

Weighted degree of a polynomial [17]

For weights $w_x, w_y\in\mathbb{Z}^+$, the (w_x,w_y) weighted degree of a monomial $q_{ij}x^iy^j$ is iw_x+jw_y. The (w_x,w_y) weighted degree of a polynomial $Q(x,y)=\sum_{ij}q_{ij}x^iy^j$ is the maximum, over the monomials with non, zero coefficients, of the (w_x,w_y) weighted degree of the monomial.

Example 6.3.
Consider the monomial

$$M(x,y):=3x^2y^4$$

The $(1,3)$ weighted degree of this monomial is $2\cdot 1+3\cdot 4=14$.

If we knew y corresponded to the polynomial x^2+4x+1 then we would most likely be interested in the $(1,2)$ weighted degree of $M(x,y)$ since y is a degree 2 function of x. If we knew or were searching for a degree at most ζ polynomial, then we would be interested in the $(1,\zeta)$ weighted degree. The $(1,2)$ weighted degree of $M(x,y)$ is $2\cdot 1+4\cdot 2=10$. The $(1,2)$ weighted degree of the polynomial

$$p(x,y):=xy^2+x^6+3$$

is equal to

$$\max\{\big(1\cdot 1+2\cdot 2\big),\big(6\cdot 1+0\cdot 2\big),\big(0\cdot 1+0\cdot 2\big)\}=\max\big\{5,6,0\big\}=6$$

Input points

The algorithm takes as inputs n points from $\mathbb{F}^2$. Define the set of these points as

$$A := \Big\{(x_1,y_1),(x_2,y_2),...,(x_n,y_n)\Big\}$$

The algorithm finds all degree δ or less polynomials which agree with A in at least τ places. In other words, it finds all polynomials $p(x)$ such that

$$\left|\Big\{(x_i,y_i) : f(x_i)=y_i \text{ and } (x_i,y_i)\in A\Big\}\right| \geq \tau$$

For the particular purpose of our algorithm, we do not allow repeated x_i. We remark that this restriction is also the case if this algorithm is used for Reed Solomon decoding.

Example 6.4.
Let $\mathbb{F} := \mathbb{Z}_7$ be equal to the integers mod 7. Let $n=3$ and let $A := \{(1,3),(3,2),(4,5)\}$. Since there are no repeated x values, A is a valid point set. We do not allow $A := \{(1,2),(1,3),(4,5)\}$ since there are two points $\{(1,2),(1,3)\}$ with the same x value.

6.5.1 Sudan's algorithm procedure [17]

Inputs:

$n,\delta,\tau,\Big\{(x_1,y_1),...,(x_n,y_n)\Big\}$

Parameters:

Parameters l, m to be set later

Find $Q(x,y)$:

Find any function $Q : \mathbb{F}^2 \to \mathbb{F}$ satisfying:

$$\left.\begin{array}{l} Q(x,y) \text{ has } (1,\delta) \text{ weighted degree at most } m+l\delta \\ \forall i\in[n], Q(x_i,y_i)=0 \\ Q \text{ is not identically zero.} \end{array}\right\} \tag{6.5}$$

Factor $Q(x,y)$:

Factor the polynomial $Q(x,y)$ into irreducible factors.

Output polynomials:

Output all the polynomials f such that $(y - f(x))$ is a factor of Q and $f(x_i) = y_i$ for at least τ values of i from $[n]$

We remark that Sudan's algorithm does not work on any general code, as the necessary conditions will not be met. We have the asymptotic condition $\tau \geq \sqrt{2nd}$, which generally means this algorithm works better for low rate codes. In general this means that if the algorithm is to return a low degree polynomial, it will have to agree with the initial points for a large number of x values. We now proceed to analyze the algorithm, and later we will analyze our implementation in MAPLE.

6.5.2 The polynomial $Q(x, y)$

$Q(x, y)$ is composed of terms of the form $q_{kj}x^k y^j$. More specifically

$$Q(x, y) := \sum_{j=0}^{l} \sum_{k=0}^{m+(l-j)d} q_{kj} x^k y^j \tag{6.6}$$

We remark that l is the maximum power of y, and that $m + l\delta$ is the maximum power of x.

Number of coefficients

In [17], it is claimed that the number of coefficients of $Q(x, y)$ is $(m + 1)(l + 1) + \delta\binom{l+1}{2}$

Proof.

$$\sum_{j=0}^{l} \sum_{k=0}^{m+(l-j)d} q_{kj} x^k y^j = \sum_{j=0}^{l} \sum_{k=0}^{m} q_{kj} x^k y^j + \sum_{j=0}^{l} \sum_{k=m+1}^{m+(l-j)d} q_{kj} x^k y^j \tag{6.7}$$

The first summation on the right hand side contributes $(m + 1)(\delta + 1)$ to the number of coefficients. For the second summation on the right hand side, we use a change of variables letting $r = k - m$. When $k = m + 1$ we have that $r = 1$. When $k = m + (l - j)\delta$ we have that $r = (l - j)\delta$. Therefore we can right the second summation as

$$\sum_{j=0}^{l} \sum_{k=1}^{(l-j)d} q_{(r+m)j} x^{r+m} y^j$$

We can pull out the δ from the bound on the inner summation and then multiply the resulting number of coefficients by δ. We are only interested in the number of coefficients, and not the subscripts of q. When looking at the number of coefficients we have $l + (l - 1) + (l - 2) + ... + (l - (l - 1) + 1) + (l - l + 1)$ which is the sum of the first l numbers, which is $\binom{l+1}{2}$. We multiply this number by δ to get the total contribution from the second summation of the left hand side of (6.7). By summing the contributions from the first and second summations of the right hand side of (6.7), we have the claimed number of coefficients, $(m + 1)(l + 1) + \delta\binom{l+1}{2}$.

□

Claim 6.1. If a function $Q : \mathbb{F}^2 \to \mathbb{F}$ satisfying (6.5) exists, the one can be found in time poly(n).

Proof. [17] Consider the coefficient matrix with columns corresponding to the terms of Q defined above, and the rows corresponding to the input points from $\mathbb{F}^2$. We wish to find the coefficients q_{kj} which satisfy $Q(x, y) = 0 \; \forall i \in [n]$. This is solving a linear homogeneous equation which can be done in polynomial time with row reducing the coefficient matrix. We remark that we are not interested in the trivial solution, all $q_{kj} = 0$ since then $Q(x, y)$ would be identically zero. □

Claim 6.2. If $Q(x, y)$ is a function satisfying (6.5) and $f(x)$ is a polynomial which agrees with A in at least τ places and $\tau > m + l\delta$ then $(y - f(x))$ divides $Q(x, y)$.

Proof. As given in [17], we argue $Q(x, y)$ has weighted degree at most $m + l\delta$. This is guaranteed through the definition of $Q(x, y)$. We then consider $Q(x, y)$ to be $Q(x, f(x))$. Since $t > m + \delta$, $Q(x, f(x))$ has more zeros (t) than its degree, and must be identically zero. We then consider $Q(x, y)$ to be a polynomial in y with coefficients from $F[x]$. By the polynomial remainder theorem, we have that if $Q_x(\xi) = 0$ then $(y - \xi))$ divides $Q_x(y)$. We then replace ξ with $f(x)$. □

Theorem 6.1. *[17] The algorithm given will run in time polynomial in n given that*

$$\tau \geq \delta \left\lceil \sqrt{\frac{2(n+1)}{\delta}} \right\rceil - \left\lfloor \frac{\delta}{2} \right\rfloor$$

Proof. See [17]. □

Setting m and l

To fulfill the algorithm requirements, the following parameters m and l can be set to (6.8) and (6.9) respectively (see [17]). (6.10) is used in the proof of showing $Q(x, y)$ has more zeros than its degree and must be identically zero. (6.11) is used to guarantee polynomial implementation. Recall that with the algorithm Berlekamp and Welch, polynomial implementation is guaranteed if $\tau \geq \frac{n+\delta}{2}$, which is the error correction bound of the Reed Solomon codes. This algorithm only returns at most one result. The asymptotic behavior of (6.11) is $\sqrt{2\delta n}$ but has since been improved to $\sqrt{\delta n}$ (see [18]).

$$m := \left\lceil \frac{\delta}{2} \right\rceil - 1 \tag{6.8}$$

$$l := \left\lceil \sqrt{\frac{2(n+1)}{\delta}} \right\rceil - 1 \tag{6.9}$$

$$\tau > m + l\delta \tag{6.10}$$

$$\tau \geq \delta \left\lceil \sqrt{\frac{2(n+1)}{\delta}} \right\rceil - \left\lfloor \frac{\delta}{2} \right\rfloor \tag{6.11}$$

Example 6.5.
For example, if we are looking for a degree at most 10 polynomial, we have $\delta := 10$. Let us assume that we have $30 := n$ points. Then

$$l := \left\lceil \sqrt{\frac{2(30+1)}{10}} \right\rceil - 1 = 2$$

$$m := \left\lceil \frac{10}{2} \right\rceil - 1 = 4$$

For restriction (6.10) we need $\tau > 4 + 2 \cdot 10$ which means we need at least 25 points out of 30 to match. For (6.11) we also need $\tau \geq 25$. This provides a brief example in the strict requirements upon the inputs, which limits its practicability. The algorithm is best suited for when $\frac{\delta}{n}$ is small.

We now slightly modify the parameters and compare the amount of errors which these new parameters can tolerate. We let $19 := n$ points, and we are searching for a degree at most $10 := \delta$ polynomial, we have $l = 1$ and $m = 4$ and hence we need $t \geq \max(4 + 1 \cdot 10, 10 \cdot 2 - 5) = 15$. Therefore, in order to be guaranteed a solution in polynomial time from Sudan's Algorithm, there needs to exists a degree at most 10 polynomial that agrees with the 19 points in at least 15 places.

6.5.3 Polynomially many solutions

In order for Sudan's algorithm to be implemented in polynomial time, the number of solutions which it outputs needs to be polynomially bounded. In [17], if $\frac{\tau}{n} \geq \left(\sqrt{2 + \frac{\delta}{4n}}\sqrt{\frac{\delta}{n}}\right) - \frac{\delta}{2n}$ then the number of polynomial returned by Sudan's algorithm is at most $\lfloor \frac{\tau}{\delta} + \frac{1}{2} - \sqrt{(\frac{\tau}{\delta} + \frac{1}{2})^2 - \frac{2n}{\delta}} \rfloor \leq \frac{2n}{\tau + \frac{\delta}{2}}$

We now quickly look at the more general problem of the number of polynomials which are within a certain distance of an element in the metric space. We will follow [18]. For every c, we let $e_c(n, k, d, q)$ be a function such that for every $(n, k)_q$ code C of distance $\mathsf{dis}(C) = d$, and for every received word r, there are at most $(qn)^c$ codewords in the Hamming ball of radius e around r.

The case $c = 0$ corresponds to only having one codeword in the Hamming ball of some radius e around any element. For $c = 0$ this is just the error correcting radius of the code and we let $e = \frac{d-1}{c}$.

We now examine the case when $c = 2$.

Theorem 6.2. *[18] Let n, k, d, q, e satisfy $d \leq n'$ and $e < (1 - \sqrt{1 - \frac{d}{n'}})n'$ where $n' = (1 - \frac{1}{q})n$. Then for every $(n, k)_q$ code C with $\mathsf{dis}(C) \geq d$ and for every received word r, there are at most qn^2 codewords within a Hamming distance e from r. This is a generalization of the Johnson bound.*

6.5.4 Factoring Polynomials

In this section we will present the first few steps of Lenstra's algorithm (see [11]) to factor bivariate polynomials over finite fields. We will look at $(x+2y)(x+3)$ over $\mathbb{Z}_5$. We define

$f = x^2 + 2xy + 3x + 6y \in \mathbb{F}[x, y]$, and notice that it is square free. We let $\delta_X f$ and $\delta_Y f$ denote the maximum degree of f in X and Y respectively. We have that $\delta_X f = 2$ and $\delta_Y f = 1$.

Finding $(Y - s)$

We now need to find a suitable $s \in \mathbb{F}_q$ such that $f \mod (Y - s) = f(X, s) \in \mathbb{F}_q[x]$ remains square-free. We write f as $(2x+6)y+(x^2+3x)$, which can be done by collecting the coefficients of the varying degrees of y. We now wish to find an $s \in \mathbb{F}_q$ such that when $(Y - s)$ divides f it remains square-free.

We now take $s = 1$ and divide $(Y - 1)$ into f. The result is $x^2 + x + 4$. We see that the $\gcd(x^2 + x + 4, 2x + 1) \neq 1$ and it turns out that we can write $x^2 + x + 4 = (x + 3)^2$, meaning the remainder is not square-free. We now take $s = 2$ and divide $(Y - 2)$ into f. The remainder is $x^2 + 4x + 3$, with derivative $2x + 4$. Since their gcd is 1, we have that for $s = 2$ the remainder $f \mod (Y - 2)$ remains square free.

Next we would use the Berlekamp-Hensel algorithm to determine an irreducible factor $h \mod (Y - 2)^k$ of $f \mod (Y - s)^k$ in $\frac{\mathbb{F}_q[X,Y]}{(Y-2)^k}$, for k sufficiently large.

We then let m be an integer with $\delta_X h \leq m < \delta_X f$ which in our case is $\delta_X h \leq m < 2$. The lattice $L \subset \mathbb{F}_q[Y]^m$ spanned by the linearly independent vectors $b_1, b_2, ..., b_m$ over $\mathbb{F}_q[Y]$ is defined as

$$L = \sum_{i=1}^{m} \mathbb{F}_q[Y] b_i = \{\sum_{i=1}^{3} r_i b_i : r_i \in \mathbb{F}_q[Y]\}$$

We then need to permute the rows of B in such a way that

$$|b'_i| \leq |b'_j| \quad for \quad 1 \leq i < j \leq m \tag{6.12}$$

$$|b'_{ii}| \geq |b'_{ij}| \quad for \quad 1 \leq i < j \leq m \tag{6.13}$$

$$|b'_{ii}| > |b'_{ij}| \quad for \quad 1 \leq j < i \leq m \tag{6.14}$$

(6.12) states that in the matrix B, as the rows increase, the maximum degree of Y can not decrease. (6.13) states that the diagonal elements have a higher degree of Y than all elements in their respective rows which are in columns $i + 1$ to m. (6.14) states that the diagonal elements have a strictly higher degree than any other terms in their respective rows from columns 1 up to $i - 1$.

In our case the original quotient $2x + 1$ divides f since $(2x + 1)(y + 3x) = 2xy + y + 6x^2 + 3x = x^2 + 2xy + 3x + y$.

Using this algorithm, $Q(x, y)$ can be factored over a finite field with p^m elements in $O(\delta 6 l^2 + (\delta^3 + l^3)pm)$ arithmetic operations.

6.5.5 Implementing Sudan's Algorithm

We decided to implement Sudan's Algorithm in MAPLE (see Appendix C) to take advantage of its rich libraries. We will now describe the steps used in implementing the algorithm. The Appendix contains the completed code.

Generating $Q(x, y)$ was done by creating a double summation with m, l, δ specified. We then solved for the coefficients of the terms of $Q(x, y)$ by generating a coefficient matrix whose rows corresponded to the n points, and whose columns corresponded to

the terms of $Q(x, y)$. This matrix was then reduced over $\mathbb{F}$ with the inert form of Gauss Jordan reduction. A solution was then generated using linsolve.

This solution was either the trivial solution (if $n < m + l\delta$), a non-trivial solution, or a non-trivial solution involving free variables. The implementation would generate a basis for the solution space by iteratively setting each free variable to 1 and all others to 0. The resulting $Q(x, y)$ was factored corresponding to the solution found for each free variable, and all polynomials found were reported. This process was not needed, but was originally implemented as assurance that the choice of assigning values to free variables with the restriction that at least one was non zero, was irrelevant. We note that the algorithm only requires one non zero solution.

Factoring of $Q(x, y)$ was completed using the built in command Factors(P), which returns a list of all the factors of P. Since the coefficient matrix was reduced over $\mathbb{F}$, the return of Factors was always in $F[x, y]$. For each factor, we searched for the form $(cy + g(x))$. If such an $g(x)$ was found, we would find the corresponding $f(x)$ such that $y - f(x) = cy + g(x)$. Since c is assumed to be non-zero, it has a multiplicative inverse in $\mathbb{F}$.

Then, for each such $f(x)$, we evaluate $f(x)$ at each x_i from the set of n points, and compare $f(x_i)$ to y_i. If the number of matches was at least τ, $f(x)$ would be a valid polynomial and it would be returned. In practice, we found that if such a polynomial was found, it would generally agree with the n points in a large number of values.

Generating Random Points

To generate the points, we first created a procedure to generate a list A of unique x values from the integers mod a number. We then passed these points into a second procedure with a polynomial $m(x)$ and a probability p. For each $x \in A$, the second procedure set $y_i = m(x_i)$ with probability p, and sets y_i to a random field element with probability $1 - p$. The result of this random assignment may be equivalent to $m(x_i)$. The procedure displays the number of points generated which agree with $m(x)$. This is used as a check to verify that the algorithm returns $m(x)$ if the number of agreed upon points meets the requirements of the algorithm. In other words, if setting τ to the number of points which agree with $m(x_i)$ satisfies the restriction of Sudan's algorithm, then we can easily check to see the algorithm found $m(x)$.

Example 6.6.
In this example we will be describing an execution of our MAPLE implementation of Sudan's algorithm. We will be working over $\mathbb{Z}_{11}$. The x values of the points corresponded to the eleven field elements in $\mathbb{Z}_{11}$. We set our polynomial $m(x) = x^2 + 2x + 4$ and assigned points to this polynomial with probability 0.70. There were two points generated which did not fall onto $m(x)$. They were $\{(5, 1), (8, 1)\}$ which should have been $\{(5, 6), (8, 7)\}$. We call this set of 11 points A.

We let $\delta = 6$ which means we are searching for all polynomials whose degree is at most six. We set $l = \lceil \sqrt{\frac{2(12)}{6}} \rceil - 1 = 1$ and we set $m = \lceil \frac{6}{2} \rceil - 1 = 2$. Our requirement on τ is that is must be greater than $2 + 1 \cdot 6 = 8$ by (6.9) and that it be at least $6 \cdot 2 - \frac{6}{2} = 9$ by (6.10). This means that for Sudan's algorithm to find a polynomial for any nontrivial $Q(x, y)$ there needs to exist a polynomial who agrees with our set of 11 points in at least

9 places. Since there were only two points not lying on $m(x)$, we expect to find $m(x)$.

We find one solution to $Q(x, y)$ by reducing the coefficient matrix and setting the first free variable to 1 and the rest to 0. We have

$$Q(x, y) = 10x^6 + 2x^5 + x^4y + 4x^4 + 10x^3y + 5x^3 + 6x^2y + 8xy^2 + xy \tag{6.15}$$

We now factor $Q(x, y)$ and we are returned three factors;

$$x, (x^2 + 2x + 10y + 4), (x^3 + 7x^2 + 8y)$$

The first factor (x) is not linear in y and so we only look at the other two factors. $(x^2 + 2x + 10y + 4)$ can be written as $10y - (x^2 + 2x + 4)$. We now take $-x^2 - 2x - 4$ and multiply it by the inverse of $10 \mod 11$ which is 10. We therefore have $f_1(x) = -10x^2 - 20x - 40 = x^2 + 2x + 4$. One can verify that in fact $(x^2 + 2x + 10y + 4) = y - (x^2 + 2x + 4)$. We repeat the same procedure for the third factor and multiply $-x^3 - 7x^2$ by 7 which is the inverse of $8 \mod 11$. We therefore have $f_2(x) = -7x^3 - 49x^2 = 4x^3 + 6x^2$.

We now evaluate $f_1(x)$ at the eleven elements of $\mathbb{Z}_{11}$ and see that it matches the original points, A in 9 positions. The two which did not match are the two points which did not lie on $m(x)$. We return $f_1(x)$ since we have that $\tau = 9$. We evaluate $f_2(x)$ at the eleven elements of $\mathbb{Z}_{11}$ and see that it matches A in 4 positions and so we do not return $f_2(x)$.

We note that in fact we were returned our original polynomial $m(x)$. The algorithm would have returned more than one polynomial if there was another polynomial of degree at most 6 which agreed with A in at least 9 places. To verify this claim, we ran a brute force search of all polynomials over $\mathbb{Z}_{11}$ of degree at most 6 who agreed with A in at least 9 places.

Chapter 7

Protocol

In order to analyze and improve the security of this new approach to hardware authentication, we describe here the iterative steps in the design of a hardware authentication protocol using PUFs. For each iteration, we will describe a protocol, explain a possible attack and then proceed to the next iteration trying to prevent it.

7.1 Protocol Assumptions

The purpose of our protocol is to authenticate a device. To accomplish this, we need the device to contain a secret, whether stored or generated, which can be verified by a trusted authority as being the secret belonging to that particular device. One way is to store this secret either in hardware or software, and then when requested, read the value of this secret. Another way, and the one which we will be using, is to use the physical properties of the device to create a secret which is device-specific. This secret should be protected against active attacks, meaning that if an adversary attempts to look inside of the device where the secret is generated, the physical properties of the device will change and the secret will become unrecoverable.

The entropy for our protocol is to come from the physical device, the PUF function, instead of from software or from predefined storage. The inherent problem with this source of entropy is that it is affected by noise and other factors such as temperature, pressure, and electromagnetic interference. This causes errors to propagate when compared with the expected value of $\mathsf{PUF}(x)$ for a particular challenge x. Also, the distribution of the PUF responses is not assumed to be uniform, and hence a particular outcome to a particular challenge is more likely than other outcomes. This becomes a problem as the key that is generated through the PUF responses is assumed to come from a uniform distribution when it is to be used as a key in a cryptographic algorithm or scheme. Therefore, we seek a way to take a noisy entropy source, correct the noise and map it to a uniform distribution. In [4], this is accomplished with fuzzy extractors. We notice that the use of fuzzy extractors does not increase the amount of entropy that is gained from the PUF, but allows for a more uniform distribution.

We will make the assumption concerning challenge-response schemes that the probability of a replay attack is minimal. This means that if the response is different depending upon the particular challenge, we do not consider it to be a successful attack if an outside party knows the value of the response for a specific set of challenges. We assume the set

of possible challenges to be very large, and thus, the probability that an adversary can successfully respond to several challenges chosen at random can be made exponentially small. Our protocol will not guarantee that an honest device (prover) will be able to prove their identity when given a particular set of challenge vectors. This is due to the inherent nature of the PUF which is affected by the environment. We will however be able to bound the probability that an honest prover will be able to authenticate themselves. For example, we may assume the PUF function has an error rate of p when compared with the expected value of the PUF. We can then find the probability that the device will get at least m correct responses out of n challenge vectors.

The protocol involves two parties, a user or verifier (V) and a device or prover (D). An honest user knows the secret of the device and asks the device to authenticate itself, to prove the device is the particular device it claims to be. The device does not assume an honest verifier.

7.2 Protocol design iterations

We now proceed iteratively through creating a protocol based on our mathematical assumptions of a PUF function. Each iterative step will aim to solve a possible attack in the previous protocol.

7.2.1 Initial Protocol

A first naive implementation of an authentication protocol built upon a PUF is the following. The verifier sends a set of challenges to the device. The device calculates the output of the PUF function on each of these challenges, and returns these results to the verifier. The verifier then checks if the number of incorrect responses is below some error threshold set for the particular application. If it is, then the device is authenticated. If it is not, the device is not authenticated.

One advantage to this scheme is that it is efficient. The device simply runs its PUF function on the set of challenges and returns the output of the PUF function. If the challenges are sent one at a time over the channel, the verifier can stop sending challenges once the device answers incorrectly to a threshold number of questions.

An inherent problem is that an adversary can model the behavior of the PUF. By sending each challenge multiple times, an adversary can obtain the expected output for several challenges. If we assume that the adversary knows what kind of PUF is used in the device, and how to mathematically model it, it is possible to obtain a general expression for the PUF function based on a certain number of challenge-response pairs (for an example, see [6]).

7.2.2 Hashed output

To fix the problem of an adversary modeling the behavior of the PUF, we will use a one-way hash function (see [20]). Assume the challenges are sent k at a time, and the responses are returned together. We apply the hash function to the responses to the challenges. When an adversary views the hashed returned value, it is assumed to be computationally infeasible to find the value and hence the secret which was hashed.

An advantage to this protocol is that the response from the device cannot be used to model the PUF function as this response is a hashed value.

This protocol does not tolerate noise in the PUF responses. The hashed value of the expected PUF response and the hashed value of the response with a single noisy response may be far away from one another. To account for these errors, the user would need to check the hashed value of all responses that have up to the error tolerance number of errors. In the Hamming metric, if the response is of length ν, and the error tolerance level is β, then we need to check $\sum_{i=0}^{\beta} \binom{\nu}{i}$ possible hash values.

7.2.3 Sudan's algorithm for recovering from noise

To correct the noise in the PUF responses, we will use Sudan's list decoding algorithm to find all polynomials of a certain bounded degree which pass through a suitable number of the PUF responses. Prior to the deployment of the device, the device will be assigned a low degree polynomial which passes through a large number of the device's expected outputs relative to the polynomials degree (see Section 6.1). The set of possible challenges will be restricted to those whose expected output lies on this low degree polynomial. The authenticating key of the device will be the coefficients of its polynomial. These will be found by running Sudan's algorithm on the challenge-response pairs $\{(c_i, r_i)\}$ of the PUF function. For each polynomial that is found, its coefficients will be hashed and the set of hashed coefficients will be returned. The device is authenticated if at least one of these hashed values is equal to the hash of the devices polynomial. If there is not a match, the device is not authenticated. Depending upon the specific parameters of this protocol, i.e. number of challenges, maximum degree of the polynomial, a bound can be put on the number of possible hashed responses returned by the device.

One of the advantages of this protocol is that it corrects the noise that is inherent in the responses. Sudan's algorithm allows for the polynomial to be found even if a large number of responses from the device do not lie on the polynomial. By hashing the coefficients, an adversary is not able to see the polynomial.

Although the coefficients of all the polynomials Sudan's algorithm returns are hashed, if a device is authenticated on a particular set of challenges then an adversary who views the communication between the verifier and the device knows the hash of the key polynomial is one of the returned hashed values. The adversary can simply return the same set of hashed values for any challenge and be authenticated, even though this adversary cannot generate the polynomial given only the challenges.

7.2.4 Challenge-dependent hashed coefficients

We now slightly modify the previous protocol. Each of the hashed values which are returned from the device now correspond to hashing the coefficients of the polynomial found by Sudan's algorithm appended with the value of some function f_c evaluated on the set of received challenges ξ.

We assume a large range of values for f_c. Therefore, even if an adversary knows the hash of the coefficients appended with $f_c(\xi_1)$ for some set of challenges ξ_1, the probability that another set of challenges ξ_2, is such that $f_c(\xi_1) = f_c(\xi_2)$ is considered negligible. This can be formalized using the assumption on replay attacks. We do not investigate such

a function f_c but provide evidence towards its existence. If the challenges are elements in $\{0,1\}^\ell$, then f_c can be the sum of each position $\mod 2$. For example, if $\ell = 3$, given the challenges $\{(0,1,1),(1,1,1),(0,1,0),(1,0,0)\}$, $f_c = (010)$. We see that the range of f_c has size 2^ℓ if we do not assume we are working in a restricted domain.

We now look at the case where an adversary knows the set of challenges ξ_1 and the correct hashed value of the coefficients appended with $f_c(\xi_1)$. We assume that only one output is returned for each set of challenges sent to the device. Assuming the challenges are selected uniformly at random, an adversary would be guaranteed to return the correct hashed value with probability $\frac{1}{8}$, exactly when f_c on a new set of challenges ξ_2 is the same as $f_c(\xi_1)$. If $f_c(\xi_1) \neq f_c(\xi_2)$, the adversary will have the probability of being authenticated depending upon the size of the range of the hash function.

This is our final protocol for hardware authentication based on PUFs. We will now proceed to analyze its feasibility.

7.3 Existence of low-degree polynomial

Our cryptographic protocol for PUFs, described in Section 6.1, relies on the fact that we can find a set of points of the form $(x_i, \mathsf{PUF}(x_i))$, which all fall on a low degree polynomial. This polynomial will then be used as a cryptographic key, or an ID for the device.

In this sense, the question of whether a low degree curve can be found on a set of $(x_i, \mathsf{PUF}(x_i))$ points becomes fundamental for the plausibility of our method. Clearly, given n of those points, we can find a polynomial of degree at most $n-1$ going through all of them. However, for degree d polynomials, where $d < n-1$ nothing assures us that we can find one that goes through more than $d+1$ points. Therefore, we will now analyze how likely it is that such polynomial in fact exist.

In order to make the problem approachable, we will assume a uniform distribution of the outputs of the PUF over $\mathbb{F}$. In this section, we are mainly interested in solving the following problem:

Question 1. *Given a set of n random points in $\mathbb{F}^2$, with what probability can we find a polynomial $p(x) : \mathbb{F} \to \mathbb{F}$ of degree at most d that goes through k of these points, for $k > d$?*

We approach this question by first considering a much simpler case.

7.3.1 The particular case of a zero-degree polynomial

We start by considering the case when the degree of the polynomial $p(x)$ is 0. In other words, we consider the case where we try to find a horizontal line that goes through at least k of our n points. Therefore, we look for the probability of having, among our n points, at least k with the same y-coordinate.

Question 2. *Given a set of n random points in $\mathbb{F}^2$, what is the probability that we can find at least k of those points with the same y-coordinate?*

If we consider a specific $y_i = b$, then the probability of having at least k of our randomly chosen points of the form (x_i, b) is simply given by the binomial probability below, where $q = |\mathbb{F}|$:

$$\sum_{i=k}^{n} \binom{n}{i} \left(\frac{1}{q}\right)^{i} \left(\frac{q-1}{q}\right)^{n-i} \tag{7.1}$$

Now, in order to find this probability when a single y_i is not specified, we cannot just multiply the above expression by the field size q, because we would be re-counting cases in which there is more than one set of at least k elements with the same y-coordinates. However, this only occurs when $k < \frac{n}{2}$. To eliminate the overcounting we apply an inclusion-exclusion principle, removing the cases where at least 2 sets are found, and then adding the cases where at least 3 sets are found, and so on. For example, the probability of having one subset with at least k points of the form (x_i, b) and another subset of at least k points of the form (x_i, c), where $b \neq c$, is given by:

$$\sum_{i=k}^{n-k} \sum_{j=k}^{n-i} \binom{n}{i} \binom{n-i}{j} \left(\frac{1}{q}\right)^{i+j} \left(\frac{q-1}{q}\right)^{n-i-j}$$

Since there are $\binom{q}{2}$ possible choices for b and c we still need to multiply this quantity by $\binom{q}{2}$. By generalizing this idea, we can apply the inclusion-exclusion principle and obtain the answer to question 2:

$$\begin{aligned} P = & \binom{q}{1} \sum_{i=k}^{n} \binom{n}{i} \left(\frac{1}{q}\right)^{i} \left(\frac{q-1}{q}\right)^{n-i} \\ & - \binom{q}{2} \sum_{i=k}^{n-k} \binom{n}{i} \sum_{j=k}^{n-i} \binom{n-i}{j} \left(\frac{1}{q}\right)^{i+j} \left(\frac{q-2}{q}\right)^{n-i-j} + \dots \\ & = \sum_{r=0}^{\lfloor \frac{n}{k} \rfloor} (-1)^r \binom{q}{r+1} \sum_{i_1=k}^{n-rk} \binom{n}{i} \sum_{i_2=k}^{n-i_1-(r-1)k} \binom{n-i_1}{i_2} \dots \sum_{i_{r+1}=k}^{n-\sum_{j=1}^{r-1} i_j} \binom{n-\sum_{j=1}^{r-1} i_j}{i_{r+1}} \frac{(q-(r+1))^{n-\sum_{j=1}^{r-1} i_j}}{q^n} \end{aligned} \tag{7.2}$$

7.3.2 Back to the original problem

Trying to approach the original problem, we observe the fact that a set of $d+1$ different points (with no common x coordinate) uniquely define a polynomial of degree at most d. Since we are looking for a polynomial of degree at most d going through a set of k or more of our n points, we know that, if it exists, it will be the polynomial uniquely determined by all subsets of $d+1$ points that are in the set of k or more points through which the polynomial passes.

Now let's consider a single set of $d+1$ points. The unique polynomial $p(x)$ of degree at

most d that it determines has at least $d+1$ matches with our set of n points. So we look for the probability that it has at least another $k-(d+1)$ matches with our remaining $n-(d+1)$ points. In order to calculate this probability, we observe that for each of our remaining points (x_{j_i}, y_{j_i}) for $i = 1, 2, ..., n-(d+1)$, we can evaluate $p(x_{j_i})$ and check whether $y_{j_i} = p(x_{j_i})$. Since y_{j_i} was selected at random, this happens with probability $\frac{1}{p}$. Therefore, the probability that $p(x)$ passes through at least k of our points is given by:

$$P = \sum_{i=k-(d+1)}^{n-(d+1)} \binom{n-(d+1)}{i} \left(\frac{1}{q}\right)^i \left(\frac{q-1}{q}\right)^{n-(d+1)-i}$$

The next logical step in trying to compute the actual probability of Question 1 would be to consider all possible subsets of $d+1$ points out of our n points. However, the probability when calculated for each new set of $d+1$ points is strongly dependent on previous ones, and therefore it becomes very difficult to estimate the probability this way.

7.3.3 Coding-theory techniques to estimate probability

We now consider an alternative approach to the problem of finding the probability of Question 1. In order to do that, we define a metric space $\mathcal{P}$ containing all the polynomials in $\mathbb{F}[x]$ with degree less than q. This is not trivial, since defining a metric for polynomials is not very natural. We start by noticing the following fact, stated as a lemma:

Lemma 7.1. *There is a one-to-one relationship between the polynomials in $\mathbb{F}[x]$ of degree less than q and all possible combinations of q points of the form $(x, y) \in \mathbb{F}^2$ with unique x-coordinates, where $q = |\mathbb{F}|$.*

Proof. All polynomials over $\mathbb{F}$ with degree less than q have exactly q coefficients, if we consider 0 a coefficient. Since each coefficient can assume q values, we have q^q such polynomials. By counting the number of possible sets of q points with unique x-coordinates, we also find q^q, since for each of the q possible x-coordinates there are q possible y-coordinates.

Since every set of q points with different x-coordinates uniquely determines a polynomial of degree at most $q-1$ (q coefficients) we conclude that we must have a bijection. □

This one-to-one relationship allows us to represent each element p of the metric space $\mathcal{P}$ by the q-tuple formed by evaluating $p(x)$ at each of the q possible values of x. This conversion is basically a Discrete Fourier transform, described in Section 3.4.2.

Now, we can have our metric naturally defined as the Hamming metric, where the distance between two polynomials p and s is defined as $\mathsf{dis}\,(p, s) := |\{x \in \mathbb{F} \,|\, p(x) \neq s(x)\}|$.

We now consider $\mathcal{D}$, the subspace of $\mathcal{P}$ that contains all polynomials of degree at most d. Clearly, there are q^{d+1} such polynomials, and we have the following claim:

Claim 7.1. The minimum distance of $\mathcal{D}$ is $q-d$.

Proof. We simply refer to Lemma A.2 to state the fact that two polynomials of degree d can agree on at most d points. Therefore, $\forall p, s \in \mathcal{P}$, we must have

$$\mathsf{dis}\,(p, s) = |\{x \in \mathbb{F} \,|\, p(x) \neq s(x)\}| = q - |\{x \in \mathbb{F} \,|\, p(x) = s(x)\}| \geq q - d$$

□

The probability from Question 1

To understand the probability we are looking for in question 1 in this metric space we slightly change the problem to the case where $n = q$. In the context of PUFs, this is a reasonable assumption, since prior to the deployment of the device, one can experimentally compute $\mathsf{PUF}\,(x)$, for all $x \in \mathbb{F}$. We notice that we are basically looking for the probability that a random set of q points with unique x-coordinates (an element of $\mathcal{P}$) will agree with at least k of the points of a polynomial of degree d. This is the same as looking for the probability that a random element of $\mathcal{P}$ will be within a distance of $q - k$ of an element of $\mathcal{D}$.

Therefore, if we can calculate the percentage of the space $\mathcal{P}$ that falls into balls of radius $q - k$ around each element of $\mathcal{D}$, we obtain the exact answer to Question 1.

The first case we consider is when $q - k \leq \lfloor \frac{q-d-1}{2} \rfloor$. In this situation, there is no overlap between the balls of radius $q - k$, since this is less than half of the minimum distance. We can then state the following:

Theorem 7.1. *If $k > d$ and $q - k \leq \lfloor \frac{q-d-1}{2} \rfloor$, then the probability that, for q random points with unique x-coordinates, there is a subset of these points of size k lying on a polynomial of degree at most d is given by*

$$P = q^{d-q+1} \sum_{i=0}^{q-k} \binom{q}{i} (q-1)^i \tag{7.3}$$

Proof. The "volume" of each ball is the number of q-tuples within a distance of $q-k$ from an element of $\mathcal{D}$. This can be easily calculated by adding all possible combinations of at most $q - k$ "errors". For example, there are $\binom{q}{i} (q-1)^i$ elements in $\mathcal{P}$ at a distance i of a given polynomial. Therefore, we find that the volume V of each ball is given by:

$$\begin{aligned} V &= \binom{q}{0} (q-1)^0 + \binom{q}{1} (q-1)^1 + \ldots + \binom{q}{q-k} (q-1)^{q-k} \\ &= \sum_{i=0}^{q-k} \binom{q}{i} (q-1)^i \end{aligned} \tag{7.4}$$

Since $q - k \leq \lfloor \frac{q-d-1}{2} \rfloor$, there is no overlap between these balls, so for q^{d+1} such balls, we obtain a probability of:

$$P = \frac{q^{d+1}}{q^q} \sum_{i=0}^{q-k} \binom{q}{i} (q-1)^i = q^{d-q+1} \sum_{i=0}^{q-k} \binom{q}{i} (q-1)^i$$

□

Notice that when $d = 0$, if we have that $q - k \le \lfloor \frac{q-1}{2} \rfloor$, then $k \ge q - \lfloor \frac{q-1}{2} \rfloor \ge \frac{q+1}{2}$, which means that $\lfloor \frac{q}{k} \rfloor = 1$. Therefore from Equation (7.2), we obtain:

$$\begin{aligned} P &= q \sum_{i=k}^{q} \binom{q}{i} \left(\frac{1}{q}\right)^{i} \left(\frac{q-1}{q}\right)^{q-i} = q \sum_{i=k}^{q} \binom{q}{q-i} \left(\frac{1}{q}\right)^{i} \left(\frac{q-1}{q}\right)^{q-i} \\ &= q \sum_{j=0}^{q-k} \binom{q}{j} \left(\frac{1}{q}\right)^{q-j} \left(\frac{q-1}{q}\right)^{j} = q^{-q+1} \sum_{j=0}^{q-k} \binom{q}{j} (q-1)^{j} \end{aligned}$$

which agrees with the result from Equation (7.3).

Example 7.1.
If $\mathbb{F} = \mathbb{Z}_{13}$, we can use Equation (7.3) to find the probability that there exists a polynomial of degree $d = 6$ going through $k = 10$ of $q = 13$ random points with unique x-coordinates, since $13 - 10 = 3 \le \lfloor \frac{16-6-1}{2} \rfloor = 4$. This probability turns out to be $P = 0.1047 = 10.5\%$.

The constraint $q - k \le \lfloor \frac{q-d-1}{2} \rfloor$ severely limits the cases in which we can calculate the probability precisely. For smaller values of k which do not satisfy this conditions what happens is that the radius $n - k$ becomes larger than the half of the minimum distance of $\mathcal{D}$, and some of the balls will overlap. Our subspace $\mathcal{D}$ can also be seen as a code, where all polynomials of degree at most d are codewords. It is, in fact, a *maximum distance* code, and the number of codewords achieves the Singleton bound:

$$|\mathcal{D}| = q^{d+1} = q^{n-\delta+1} \tag{7.5}$$

where $n = q$ and δ is the minimum distance of the code $n - d$. The fact that for $q - k > \lfloor \frac{q-d-1}{2} \rfloor$ we are basically recounting the overlap between the balls several times means that the expression in (7.3) can still be considered as an upper bound for the probability, but since this overlap becomes quite significant and increases very fast as we decrease k, this upper bound will often be greater than 1.

In order to analyze cases where $q - k > \lfloor \frac{q-d-1}{2} \rfloor$, we must therefore try to calculate the overlap between the balls. We start by analyzing the overlap between two balls whose centers are at distance $\delta = n - d$, the minimum distance of the code $\delta = n - d$. We are assured that there exist two codewords that are at a distance $n - d$ of each other, by simply noticing that given one polynomial of degree d, one can pick any d points out of it and a $(d+1)^{\text{th}}$ point not in the original polynomial, and form a new polynomial of degree at most d (thus another codeword) which must be different than the original. So we formulate the following question:

Question 3. *Given two codewords A and B, such that $\mathsf{dis}(A, B) = \delta$, how many elements $p \in \mathcal{P}$ are there, such that $\mathsf{dis}(A, p) \le r$ and $\mathsf{dis}(B, p) \le r$?*

The answer for $r \le \lfloor \frac{\delta-1}{2} \rfloor$ is clearly 0, since there is no overlap between the balls of radius r around A and B. For higher values of $r > \lfloor \frac{\delta-1}{2} \rfloor$, we analyze the problem in the following way. Assume, without loss of generality, that:

$$\begin{aligned} A &= (a_1, a_2, ..., a_\delta, c_{\delta+1}, c_{\delta+2}, ..., c_q) \\ B &= (b_1, b_2, ..., b_\delta, c_{\delta+1}, c_{\delta+2}, ..., c_q) \end{aligned}$$

where $a_i, b_i, c_i \in \mathbb{F}$ and $a_i \neq b_i$ for all the indexes considered. Now we attempt to count the number of polynomials p that satisfy $\mathsf{dis}\,(p, A) = \rho_A$ and $\mathsf{dis}\,(p, B) = \rho_B$. To do that, we start with the codeword polynomial B and build p by changing ρ_B entries. Assume that α of the changes are made on the b_i entries, and $\rho_B - \alpha$ changes are made on the c_i's. The changes on the c_i's bring p farther away from A and the changes on the b_i's can either bring p closer to A or farther away from it. So we assume x changes of the form $b_i \to a_i$ and y changes of the form $b_i \to \beta_i \neq a_i$, totaling $x + y = \alpha$. Therefore we have:

$$\begin{aligned} \mathsf{dis}\,(p, A) &= \delta + (\rho_B - \alpha) - x + y \\ &= \delta + (\rho_B - \alpha) - x + (\alpha - x) \\ &= \delta + \rho_B - 2x \end{aligned} \tag{7.6}$$

By setting $\mathsf{dis}\,(p, A) = \rho_A$, we conclude that:

$$x = \frac{\delta + \rho_B - \rho_A}{2} \tag{7.7}$$

Clearly, we see that $\rho_B - \rho_A$ must have the same parity as δ in order for such p to exist. Assuming it does, we can then count the number of possible ways of obtaining such p, for a fixed α, to be:

$$\binom{\delta}{x}\binom{\delta - x}{y} \cdot (q-2)^{\alpha - x} \cdot \binom{q - \delta}{\rho_B - \alpha} (q-1)^{\rho_B - \alpha}$$

and summing over α we obtain

$$S_{\rho_A, \rho_B} = \sum_{\alpha = x}^{\rho_B} \binom{\delta}{x}\binom{\delta - x}{y} \cdot (q-2)^{\alpha - x} \cdot \binom{q - \delta}{\rho_B - \alpha} (q-1)^{\rho_B - \alpha} \tag{7.8}$$

where $S_{\rho_A, \rho_B} \stackrel{\text{def}}{=} |\{p \in \mathcal{P} \,|\, \mathsf{dis}\,(p, A) = \rho_A \text{ and } \mathsf{dis}\,(p, B) = \rho_B\}|$.

For a given ρ_A, we can find all values of ρ_B that will make $S_{\rho_A, \rho_B} \neq 0$ by first invoking the triangle inequality, as follows:

$$|\delta - \rho_A| \leq \rho_B \leq \delta + \rho_A \tag{7.9}$$

So, we now restrict the problem in Question 3 to the case where $\rho_A, \rho_B < \delta$. This assumption allows us to answer the problem, provided that $r < \delta$. In this case we have that the number of elements $p \in \mathcal{P}$ such that $\mathsf{dis}\,(A, p) \leq r$ and $\mathsf{dis}\,(B, p) \leq r$ is given by:

$$\mathcal{O}_q(r, d) = \sum_{\rho_A = \delta - r}^{r} \left(\sum_{\rho_B = \delta - \rho_A}^{r} \lambda_{\rho_A, \rho_B} S_{\rho_A, \rho_B} \right) \tag{7.10}$$

where $\lambda_{\rho_A, \rho_B} = \begin{cases} 1 & \text{if } (\delta + \rho_B - \rho_A) \pmod 2 = 0 \\ 0 & \text{otherwise} \end{cases}$

Example 7.2.
If we look at polynomials of degree $d = 5$ over $\mathbb{Z}_{11}$, we can use Equations (7.10) and

(7.8) to calculate the number of polynomials of degree less than 11, which agree with two polynomials of degree 5 in at least 7 points, by setting $\delta = 11 - 5 = 6$ and $r = 11 - 7 = 4$. We find that $\mathcal{O}_{11}(4, 5) \leq 1590$. Equality is achieved when the two polynomials of degree 5 are exactly at a distance 6 away from each other (and agree in 5 points).

Although we can precisely calculate the overlap between two Hamming balls of a given radius and separated by a given distance, it is difficult to proceed from here. Technically, by summing over the elements in each Hamming ball centered at a codeword for any radius $q - k$, and subtracting the number of elements in each pairwise intersection, we find a lower bound for the probability of Question 1. However, there can be many polynomials belonging to the intersection of more than two balls. In general this approach will yield a significant underestimate to this probability, which in some cases is less than 0.

Chapter 8

Conclusion

In this project we studied hardware authentication methods which utilize physical anomalies that naturally occur in manufacturing processes. The idea of physically unclonable functions was developed as a way to use these anomalies between identically produced devices to assign a unique measurable identifier to each device. Moreover, this idea solves the problem of key distribution since each device "receives" its key from its own physical parameters. However, these PUFs rely on very sensitive physical measurements which are influenced by temperature, pressure, etc. which cause for noisy readings. In this report we analyzed the existing techniques to deal with this noise, and then proposed a new method.

A fuzzy extractor allows a reliable cryptographic key to be extracted from PUFs. They can be implemented in hardware, which makes them an attractive accessory for a PUF device. The combination of a fuzzy extractor and a PUF facilitate the design of hardware authentication protocols. In this report, we present a hardware implementation for the fuzzy extractor block, which uses a compact architecture for a BCH decoder. This is especially important for the protection of intellectual property and the prevention of counterfeiting of lightweight and low-cost hardware devices.

The only significant drawback of fuzzy extractors is the publication of a helper string to aid in the recovery of the device's key. We analyzed this scheme from an information-theoretic perspective and concluded that by trying to make it more noise-tolerant, more information is leaked by the helper string. Specifically, in a code offset construction, we linked the entropy which remains in the system after the publication of the helper string to the minimum distance of the code used. This clearly showed the trade-off between noise tolerance and security and motivated us to look for a new scheme.

The main contribution of this project is the proposal of an authentication scheme which is based upon Sudan's list decoding algorithm. This scheme allows for a larger noise-tolerance while not requiring helper information to be published. A disadvantage of this protocol is the increased complexity of the algorithm when compared with other error-correcting procedures. Furthermore, a low degree polynomial needs to be found when the device is in the enrollment phase, and it is not clear if this is always possible or if there is an efficient way to do it. This has severe practical implications to this scheme, since the probability of the existence of this curve determines the expected number of devices from a manufacturing line that can actually be deployed.

8.1 Future Work

The advantages of the scheme proposed in this paper are clear, although its feasibility has to be throughly analyzed. Clearly, the complexity of the implementation (especially in hardware) would be compromised, and it is important to analyze how much is gained in terms of security.

We now informally compare our protocol to one which utilizes a fuzzy extractor with an underlying secure sketch. The analysis provides motivation towards the possible improvements which can be made to our protocol in both security and noise tolerance. Our protocol allows the recovery of the cryptographic key of a device when the device is given challenges whose expected outputs lie on a low degree polynomial specific to the device. We assume an adversary does not increase their probability of guessing this polynomial when seeing which challenges are sent to the device. We chose to use a list decoding procedure in our protocol to allow for a large amount of noise in the responses of the PUF device to be corrected. Asymptotically, we can use Sudan's algorithm to allow up to $n - \sqrt{n\delta}$ responses to not fall on the device's low degree polynomial [17].

From Corollary 4.1, we have that the average min entropy of a secure sketch using the code offset construction is bounded by $m - d + 1$. Assuming that $d = 2t + 1$, we can write this expression as $m - 2t$. Furthermore, if we assume a uniform input distribution on our metric space, we have that $m = n$. For this comparison to make sense we use log base q in our entropy measurements.

The key of each device in our protocol is a degree at most δ polynomial, which is assigned to each device depending upon the responses of that device. To allow for a comparison to a fuzzy extractor, we view Sudan's algorithm as a decoder for Reed-Solomon codes, and in more familiar notation, we have that $\delta = k - 1$. Asymptotically, this algorithm allows for up to $t = n - \sqrt{nk}$ noisy measurements, and solving for k, we obtain $k = \frac{(n-t)^2}{n}$. We now wish to compare the informational entropy associated to the cryptographic keys generated in each scheme.

In the code-offset construction, an adversary's probability of correctly guessing the element which corresponds to the helper string is

$$\left(\frac{1}{q}\right)^{n-2t}.$$

In our protocol, the probability of correctly guessing the polynomial is

$$\left(\frac{1}{q}\right)^{k} = \left(\frac{1}{q}\right)^{\frac{(n-t)^2}{n}}.$$

Comparing these exponents, we want to know the relationship between $n - 2t$ and $\frac{(n-t)^2}{n}$. We see however that we can write $n - 2t$ as $\frac{(n-t)^2}{n} - \frac{t^2}{n}$. This means that the entropy that remains in the system with fuzzy extractors is always less than the entropy in our proposed system. Therefore, our informal analysis leads to the probability of an adversary guessing the polynomial of the device in our protocol to be $\frac{1}{q}^{\frac{t^2}{n}}$ times the probability an adversary guessing on a secure sketch. This provides motivation to analyze whether our algorithm provides more security with the same number of challenges.

Since our protocol allows for multiple responses to be returned from the device, as Sudan's algorithm is a list-decoding algorithm, we must look at the number of polynomials which can be returned in conjunction with the decreased probability. Future analysis can be done on this trade-off between a larger error tolerance, which is highly desirable for PUF devices which are particularly noisy, and the increased probability that an adversary can guess the correct value by being allowed to return multiple polynomials. Again, we reiterate the need for future work on the existence and practicality of finding the initial low-degree polynomial from the expected outputs of the PUF device, which is critical for the implementation of this protocol.

Bibliography

[1] E. Berlekamp: Algebraic Coding Theory - Revised 1984 Edition: Aegean Park Press, Walnut Creek, 1984

[2] R. Blahut: Algebraic Codes for Data Transmission. Cambridge University Press, Cambridge, 2003

[3] C. Bosch, J. Guarjardo, A. Sadeghi, J. Shokrollahi and P. Tuyls: Efficient Helper Data Key Extractor on FPGAs. *Lecture Notes in Computer Science*, 5154:181-197, 2008.

[4] Y. Dodis, R. Ostrovsky, L. Reyzin and A. Smith: Fuzzy Extractors: How to Generate Strong Kets from Biometrics and Other Noisy Data. *Siam J. Computing* , **38**, pages 97-139, 2008

[5] B. Gassend, D. Clarke, M. Dijk and S. Devadas: Delay-based circuit authentication and applications. *Proceedings of the 2003 ACM Symposium on Applied Computing*, pages 294-301 (2003)

[6] G. Hammouri and B. Sunar: PUF-HB: A Tamper-Resilient HB based Authentication Protocol. *Lecture Notes in Computer Science*, **5037** (2008)

[7] D.G. Hoffman, D.A. Leonard, C.C. Lindner, K.T. Phelps, C.A. Rodger and J.R. Wall: Coding Theory the Essentials. Marcel Dekker, New York, 1991

[8] T. Itoh and S. Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases. *Information and Computation*, **78**, pages 171-177, 1988.

[9] A. Juels and M. Sudan: A Fuzzy Vault Scheme. *Designs, Codes and Cryptography* **38**: pages 237-257, 2006

[10] J. Kaps, K. Yuksel and B. Sunar: Energy Scalable Universal Hashing. *IEEE Transactions on Computers*, Vol. X, No. X, 2004

[11] A. K. Lenstra: Factoring Multivariate Polynomials Over Finite Fields. *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 189-192, New York, 1983.

[12] S. Lin and D. Costello, Jr.: Error Control Coding - Fundamentals and Applications. Prentice-Hall, New Jersey, 1983

[13] J. H. van Lint: Introduction To Coding Theory: Graduate Texts In Mathematics - Third Edition. Springer, 2004

[14] R. Pappu, B. Recht, J. Taylor and N. Gershenfeld: Physical one-way functions. *Science*, **6**, 2002

[15] V. Pless: Introduction to the Theory of Error-Correcting Codes - Second Edition. Wiley-Interscience, 1990

[16] I.S. Reed and G. Solomon: Polynomial Codes over Certain Finite Fields. *SIAM Journal of Applied Mathematics*, Vol. 8, pages 300-304, 1960

[17] M. Sudan: Decoding of Reed Solomon codes beyond the error-correction bound. *IBM Thomas J. Watson Research Center* (1997)

[18] M. Sudan: List Decoding: Algorithms and Applications. *SIGACT News Complexity Theory* **25** (2000)

[19] J. Tignol: Galois' theory of algebraic equations World Scientific, 2001

[20] W. Trappe and L. Washington: Introduction to Cryptography with Coding Theory - Second Edition. Pearson, Upper Saddle River, New Jersey, 2006

Appendix A

Selected proofs

Lemma A.1. $\max_y \sum_x f(x,y) \leq \sum_x \max_y f(x,y)$

Proof.

$$\begin{aligned}\max_y \sum_x f(x,y) &= f(x_1, y_k) + f(x_2, y_k) + ... + f(x_n, y_k) \\ &\leq \max_y f(x_1, y) + \max_y f(x_2, y) + ... + \max_y f(x_n, y) \\ &= \sum_x \max_y f(x,y)\end{aligned}$$

□

Lemma A.2. *Two different polynomials of degree at most d over $\mathbb{F}$ can agree on at most d points.*

Proof. Let $p(x)$ and $q(x)$ be two polynomials of degree at most d over $\mathbb{F}$. We prove the lemma by showing that if they agree on more than d points, than $p(x) \equiv q(x)$.

Let us assume that $p(x_i) = q(x_i)$ for $i = 1, 2, ..., m$, where $m > d$. Then, if we form the polynomial $r(x) = p(x) - q(x) = r_0 + r_1 x + r_2 x^2 + ... + r_d x^d$, we have that $r(x_i) = 0$ for $i = 1, 2, ..., m$. Since this polynomial has more roots than its degree, we conclude that we must have $r(x) \equiv 0$, which in turn means that $p(x) \equiv q(x)$. □

Lemma A.3. *An $[n, k, d]$ Reed-Solomon code has minimum distance $d = n - k + 1$, and thus achieves the Singleton Bound.*

Proof. For a $[n = q - 1, k = n - d + 1, d]$-RS code, every codeword ω, represented by the polynomial

$$\omega(x) = a_0 + a_1 x^2 + ... + a_{n-1} x^{n-1},$$

has α_i, for $i = 1, 2, ..., d - 1$, as roots. This happens since those are all the roots of the generator polynomial $g(x)$, which divides any codeword $\omega(x)$. This is equivalent to saying that the vector $\mathbf{a} = (a_0, a_1, ..., a_{n-1})$ formed by the coefficients of $\omega(x)$ satisfies $H\mathbf{a}^T = 0$ where:

$$H = \begin{pmatrix} 1 & \alpha & \alpha^2 & \alpha^3 & \cdots & \alpha^{n-1} \\ 1 & \alpha^2 & \alpha^4 & \alpha^6 & \cdots & \alpha^{2(n-1)} \\ 1 & \alpha^3 & \alpha^6 & \alpha^9 & \cdots & \alpha^{3(n-1)} \\ \vdots & & & & \ddots & \vdots \\ 1 & \alpha^{d-1} & \alpha^{2(d-1)} & \alpha^{3(d-1)} & \cdots & \alpha^{(d-1)(n-1)} \end{pmatrix}$$

To prove that the minimum distance of the code is d it suffices to show that the minimum weight of any codeword is d. So let us assume there is a codeword with vector representation $\mathbf{c} = (c_0, c_1, ..., c_{n-1})$ whose weight is $d_c = m < d$. So we must have $c_{j_i} \neq 0$ for $i = 1, ..., m$ and $c_{j_i} = 0$ otherwise. Therefore, by starting with the following expression,

$$H\mathbf{c}^T = \begin{pmatrix} 1 & \alpha & \alpha^2 & \alpha^3 & \cdots & \alpha^{n-1} \\ 1 & \alpha^2 & \alpha^4 & \alpha^6 & \cdots & \alpha^{2(n-1)} \\ 1 & \alpha^3 & \alpha^6 & \alpha^9 & \cdots & \alpha^{3(n-1)} \\ \vdots & & & & \ddots & \vdots \\ 1 & \alpha^{d-1} & \alpha^{2(d-1)} & \alpha^{3(d-1)} & \cdots & \alpha^{(d-1)(n-1)} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{pmatrix} = 0$$

and considering the 0 components of the vector $\mathbf{c}$ we can obtain:

$$H\mathbf{c}^T = \begin{pmatrix} \alpha^{j_1} & \alpha^{j_2} & \alpha^{j_3} & \cdots & \alpha^{j_m} \\ \alpha^{2j_1} & \alpha^{2j_2} & \alpha^{2j_3} & \cdots & \alpha^{2j_m} \\ \alpha^{3j_1} & \alpha^{3j_2} & \alpha^{3j_3} & \cdots & \alpha^{3j_m} \\ \vdots & & & \ddots & \vdots \\ \alpha^{(d-1)j_1} & \alpha^{(d-1)j_2} & \alpha^{(d-1)j_3} & \cdots & \alpha^{(d-1)j_m} \end{pmatrix} \begin{pmatrix} c_{j_1} \\ c_{j_2} \\ \vdots \\ c_{j_m} \end{pmatrix} = 0$$

Since $m < d$ the number of equations is greater than or equal to the number of unknowns. Therefore, we can reduce the number of equations, in order to obtain a square matrix and the following homogeneous system:

$$\begin{pmatrix} \alpha^{j_1} & \alpha^{j_2} & \alpha^{j_3} & \cdots & \alpha^{j_m} \\ \alpha^{2j_1} & \alpha^{2j_2} & \alpha^{2j_3} & \cdots & \alpha^{2j_m} \\ \alpha^{3j_1} & \alpha^{3j_2} & \alpha^{3j_3} & \cdots & \alpha^{3j_m} \\ \vdots & & & \ddots & \vdots \\ \alpha^{mj_1} & \alpha^{mj_2} & \alpha^{mj_3} & \cdots & \alpha^{mj_m} \end{pmatrix} \begin{pmatrix} c_{j_1} \\ c_{j_2} \\ \vdots \\ c_{j_m} \end{pmatrix} = 0$$

Since $\mathbf{c}' = (c_{j_1}, c_{j_2}, ..., c_{j_m})$ is a nontrivial solution the matrix determinant must be 0. However, we have:

$$\begin{vmatrix} \alpha^{j_1} & \alpha^{j_2} & \cdots & \alpha^{j_m} \\ \alpha^{2j_1} & \alpha^{2j_2} & \cdots & \alpha^{2j_m} \\ \alpha^{3j_1} & \alpha^{3j_2} & \cdots & \alpha^{3j_m} \\ \vdots & & \ddots & \vdots \\ \alpha^{mj_1} & \alpha^{mj_2} & \cdots & \alpha^{mj_m} \end{vmatrix} = \alpha^{j_1}\alpha^{j_2}\alpha^{j_3}...\alpha^{j_m} \begin{vmatrix} 1 & 1 & 1 & 1 \\ \alpha^{j_1} & \alpha^{j_2} & \cdots & \alpha^{j_m} \\ \alpha^{2j_1} & \alpha^{2j_2} & \cdots & \alpha^{2j_m} \\ \vdots & & \ddots & \vdots \\ \alpha^{mj_1} & \alpha^{mj_2} & \cdots & \alpha^{mj_m} \end{vmatrix} =$$

$$= \alpha^{j_1}\alpha^{j_2}\alpha^{j_3}...\alpha^{j_m}(\alpha^{j_m} - \alpha^{j_1})(\alpha^{j_m} - \alpha^{j_2})...(\alpha^{j_m} - \alpha^{j_{m-1}})(\alpha^{j_{m-1}} - \alpha^{j_1})...(\alpha^{j_2} - \alpha^{j_1}) \neq 0$$

(since all α^{j_i} are different), which is a contradiction. □

Appendix B

Hardware Complexity

B.1 Average number of nonzero coefficients for elements of $GF(2^m)$

For our complexity estimation, especially for the number of LFSR taps (which translates into XOR gates) it is important to know how many nonzero bits there are on the binary representation of an arbitrary element of $GF(2^m)$, say α^i, when $i \geq m$. In order to do that, we estimate the average of nonzero bits on α^i, by summing over all combinations of at least 2 nonzero bits among m bits:

$$\begin{aligned}\frac{\sum_{i=2}^{m}\binom{m}{i}i}{2^m - m} &= \frac{\sum_{i=2}^{m} m\binom{m-1}{i}}{2^m - m} = \frac{m\sum_{i=1}^{m-1}\binom{m-1}{i}}{2^m - m} \\ &= \frac{m}{2}\left(\frac{\sum_{i=1}^{m-1}\binom{m-1}{i}}{2^{m-1} - \frac{m}{2}}\right) = \frac{m}{2}\left(\frac{2^{m-1} - 1}{2^{m-1} - \frac{m}{2}}\right) \approx \frac{m}{2}\end{aligned} \tag{B.1}$$

B.2 Reducing the complexity of Syndrome computation

The scheme for computing the syndromes S_i presented in Section 5.2.2 is intuitive, and it makes sense to introduce it first. However, especially for larger block sizes n, this scheme becomes much larger. Therefore, we introduce a modification to the circuits for syndrome computation, which manages to reduce the complexity.

This other way of computing $S_i = r(\alpha^i)$ is based on the fact that the division algorithm for polynomials allows us to write:

$$r(x) = \phi_i(x) \cdot a(x) + b(x)$$

where $\phi_i(x)$ is the minimal polynomial of α^i over $GF(2)$ and $b(x)$ is the remainder of the division of $r(x)$ by $\phi_i(x)$. Since $\phi_i(\alpha_i) = 0$, we have that $r(\alpha_i) = b(\alpha_i)$. Building a circuit that computes $b(\alpha^i)$ can again be done with an LFSR.

Example B.1.
Since α^3 has $\phi_3(x) = 1 + x + x^2 + x^3 + x^4$ as minimal polynomial over $GF(2)$, we use the fact that by applying the division algorithm to x^k, where $k \geq 4$, we obtain $x^k = \phi_3(x) \cdot x^{k-4} + x^3 + x^2 + x + 1$, to devise the following circuit:

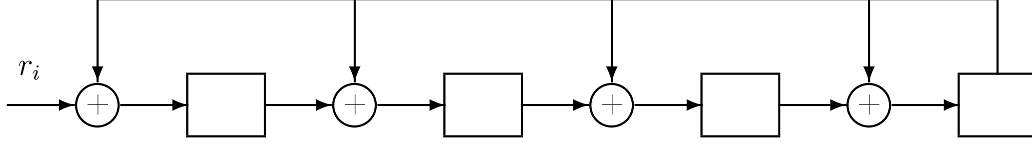


Figure B.1: LFSR to calculate $b(x)$

After all coefficients of $r(x)$ are shifted in, the LFSR contains the coefficients of $b(x)$. Now, in order to find $b(\alpha^3)$ we simply notice that:

$$b(\alpha^3) = b_0 + b_1\alpha^3 + b_2\alpha^6 + b_3\alpha^9 = b_0 + b_3\alpha + b_2\alpha^2 + (b_1 + b_2 + b_3)\alpha^3$$

and we only need to recombine the outputs of the LFSR (with corresponding additions) to obtain $b(\alpha^3) = r(\alpha^3)$.

This new method for computing syndromes may seem to achieve the same level of complexity than the one presented in Section 5.2.2. The main difference comes from the fact that, in general, division by $\phi_i(x)$ requires fewer gates than multiplication by $alpha^i$ (required in the previous method). To compute the $2t$ syndromes, the complexity of each method is described as follows:

1. When we multiply $b(\alpha) \in GF(2^m)$ by α^i, we obtain $b(\alpha)\alpha^i = b_0\alpha^i + b_1\alpha^{i+1} + b_2\alpha^{i+2} + \ldots + b_{m-1}\alpha^{i+m-1}$. In order to reduce this expression to its m-bit representation, we must use the generator polynomial of the Galois Field to find the equivalent representation of each of the last m α^{i+k}'s. In general, we assume that each of those α^{i+k} will have a representation with $\frac{m}{2}$ nonzero bits (from Equation (B.1)), resulting in $i\frac{m}{2}$ feedback loops (and XOR gates) in the LFSR. Since we need one LFSR for each odd syndrome, we obtain a total of $\frac{m}{2}\sum_{\ell=1}^{t}(2\ell - 1) = \frac{mt^2}{2}$ XOR gates for the LFSRs, and an extra $\frac{m}{2}$ XORs per even syndrome.

2. When we use the remainder approach, we only need at most m feedback loops (and XOR gates) for each minimal polynomial being considered (since their degree is at most m). However, the outputs must still be recombined to form the remainders. We do that by following Example B.1 and computing $b(\alpha^i) = b_0 + b_1\alpha^i + b_2\alpha^{2i} + \ldots + b_{m-1}\alpha^{(m-1)i}$. For each of the $m - m/i$ α^{i+k}'s, we again must find an m-bit equivalent representation, requiring $(m - \frac{m}{i})\frac{m}{2}$ feedback loops in each LFSR. Overall we have $m + \frac{m}{2}\sum_{\ell=1}^{t}\frac{2\ell}{2\ell-1} \leq m + \frac{tm^2}{2}$ XOR gates.

   Since, for larger codes $t > m$, we conclude that for our complexity analysis we can assume the second implementation.

B.3 Gate count for Inverter block

We can basically divide the inverter into three parts. For the "squarer" LFSR, we realize that we will need $m \cdot \frac{m}{2}$ XORs. The array of AND gates essentially computes all m^2 combinations between each bit from a and each bit from b. In order to estimate the number of XORs needed for the combinational logic responsible for the parallel multiplication, we observe that when we multiply to generic elements of $GF(2^m)$, we obtain:

$$\begin{aligned}\left(a_0 + a_1\alpha + a_2\alpha^2 + ... + a_{m-1}\alpha^{m-1}\right)\left(b_0 + b_1\alpha + b_2\alpha^2 + ... + b_{m-1}\alpha^{m-1}\right) = \\ = a_0b_0 + (a_0b_1 + a_1b_0)\alpha + ... + \sum_{i+j=k} a_ib_j\ \alpha^k + ... + a_{m-1}b_{m-1}\alpha^{2m}\end{aligned} \tag{B.2}$$

Since all the pairs a_ib_j are already available to us we just need to count the number of sums, which, from Equation (B.2), turn out to be:

$$0 + 1 + 2 + ... + (m-2) + (m-1) + (m-2) + ... + 2 + 1 + 0$$

However, the coefficients of α^i for $i \geq m$ actually become feedback loops (since we simplify them to their m-bit representation). Since, according to Equation (B.1), we obtain in average $\frac{m}{2}$ feedback loops for each such α^i, we calculate the number of XORs to be:

$$\begin{aligned}(0 + 1 + 2 + ... + (m-1)) + \frac{m}{2}\left((m-2) + (m-3) + ... + 1 + 0\right) = \\ = \frac{m(m-1)}{2} + \frac{m}{2} \cdot \frac{(m-1)(m-2)}{2} = \frac{m^3}{4} - \frac{m^2}{4}\end{aligned} \tag{B.3}$$

B.4 Gate count for Chien's Search block

In the Chien's Search block, for each of the t blocks containing an α^i multiplier, we have m flip-flops. Additionally, we obtain $i\frac{m}{2}$ LFSR taps, and we therefore have our total XOR gates count to be:

$$\frac{m}{2}\sum_{i=1}^{2t} i = \frac{m}{2}(2t+1)\,t = \frac{m}{2}(2t+1)\,t = mt^2 + \frac{mt}{2} \tag{B.4}$$

In the summation block, we need to take each of the m-bits from each of the t multiplier blocks, which requires a total of $m(t-1)$ XORs. To test whether the m bits of the result are all 0 except for the least significant one we need $2m$ NAND gates.

Appendix C

Sudan's algorithm MAPLE source code

```
# Matthew Dailey and Ilan Shomorony
# advised by Prof. Martin
# March 3, 2009
# Sudan list decoding algorithm

with(linalg):
with(PolynomialTools):
with(difforms):

# input n,d,t set of points  m,l
# need Q(x,y) to have weighted degree at most m+Ld

# This procedure implements Sudan's list decoding algorithm
# n is the number of points
# d is the maximum degree of a polynomial that one is searching for
# t is the number of points in which a polynomial found must match
    the n points in
# m,L are used as parameters and are used for creating Qxy
# allPts is a list of the points stored in a [n x 2] array
# theModulus in a prime number representing which field we are
    working over

sudanCoeff:=proc(n,d,t::integer,m,L,allPts, theModulus)
  local numPoints, numCoeff,
    r,c,allTerms,coeffMat,term,pointX,pointY,coeff, j ,k, theRank,
  i,b,B,Qxy, allPoly, solNum, copyMat, allFactors, listFacts,
    numFact, kk, currPoly, currInverse,
  Thm5, claim4B,boundN,copyCoeff:

  # tell if the algorithm gurantees to return a polynomial if one
    exists
  Thm5 := d* ceil(sqrt(2*(n+1))/d)-floor(d/2);
  claim4B := m+l*d:
```

```
boundN := (m+1)*(l+1)+d*(l+1)*l/2:
printf("For claim 3, we need : %d  > : %d :=n\n", boundN, n);
printf("For claim 4, we need t := : %d    > : %d\n" , t,
  claim4B);
printf("For Theorem 5, we need t:= %d  >  %d\n", t, Thm5);

allPoly:= []:    # a list with all the possible curves
numPoints := n:
# get all terms which have weighted degree less than or equal to
  m+Ld
allTerms := simplify(sum(sum( (y^k)*(x^j) ,
  j=0..m+(L-k)*d),k=0..L));
#print(allTerms);
numCoeff := nops(allTerms);
printf("numPoints = %d      number of coefficients =
  %d\n",numPoints, numCoeff);

# generate coefficient matrix
coeffMat := matrix(numPoints, numCoeff):

# Fill the coefficient matrix with the corrent values for each
  point
# meaning for each term (i.e. x^2*y) fill in the x value and y
  value from the current point
for c from 1 by 1 to numCoeff do
  term := op( c, allTerms):
  for r from 1 by 1 to numPoints do
    pointX := allPts[r,1]:
    pointY := allPts[r,2]:
    coeff := subs(x=pointX, y=pointY, term):
    coeffMat[r,c] := coeff:
  end do:
end do:

# To reduce the matrix to find our coefficients
# we need to create the solution vector which is the zero vector
b:=vector(numPoints ):
for i from 1 by 1 to numPoints do
  b[i]:=0:
end do:

# We need to use the inert form to row reduce our matrix to stay
  in
# our finite field
coeffMat := Gaussjord(coeffMat, 'thRank', 'theDet') mod
  theModulus ;
```

```
# Now we wish to solve for the coefficients of the terms, such
  as x^2 and xy
# The number of free variables is equal to
  (numCoeff-theRank):=numSolutions
B:=linsolve(coeffMat,b,'theRank',mm );
printf("The rank of the coefficient matrix is %d\n", theRank);

# For each free variable, we will set it equal to one and get
  numSolutions different possible solution
# and for each one we will look for factors that are linear in y.
for solNum from 1 by 1 to 1 do     #(numCoeff-theRank)  -- is
  number of different free variables
  copyMat:= evalm(B):

  # Set the specific variable equal to one and all others zero
  for i from 1 by 1 to (numCoeff - theRank) do
    for j from 1 by 1 to numCoeff do
      if i = solNum then
        copyMat[j] := subs( mm[i]=1, copyMat[j]):
      else
        copyMat[j] := subs( mm[i]=0, copyMat[j]):
      end if:
    end do:
  end do:

  # Now we have a specific solution and we print it out
  # we take the coefficients and dot product them with the
  actual terms to construct the polynomial
  Qxy:=sum( copyMat[ii]*op(ii, allTerms),ii=1..numCoeff) :
  printf("Solution number %d, and polynomial\n ",solNum);

  # now we want to loop through all the factors and find all
  those which are linear in y
  # and output their inverses (of f(x)) such that we write a
  factor linear in y as (y-f(x))
  allFactors := Factors(Qxy) mod theModulus:
  print(allFactors);

  # if we have non-trivial factors
  if nops(allFactors) = 2 then
    listFacts := allFactors[2]:
    numFact := nops(listFacts):
    for kk from 1 by 1 to numFact do
      currPoly := (listFacts[kk])[1]:    # get the actual factors
      # printf("Current polynomial :: %a\n", currPoly);
```

```
        # find the inverse, if it exits and add it to the list
        currInverse := findInverse( currPoly, y, theModulus):
        if currInverse <> -1 then
          print(currInverse);
        end if:

    #------------------------------------------------------------------
        # make sure that for each of the polynomials that we will
    return
        # that it agrees in at least t places with the points we
    inputted
        if currInverse <> -1 and isMatcht( t, allPts, n,
    currInverse, theModulus) then
          allPoly := [op(allPoly), currInverse]:
        end if:
      end do:
    end if:

  end do:

  # return all possible found polynomials in a list
  return allPoly:
end proc:


# This procedure takes in a value t, a list of points, and an
    expression in x, a modulus, the number of points
# it returns true if f(points_x_i_value) = (y_i mod theMod) for at
    least t x_i's
# The points can have repeated x coordinates
# however, for our usage, the points should not have repeated
    coordinates

isMatcht := proc( t::integer, allPts, numPts::integer,
    anExpression, theMod::integer)
  local numMatches, i:
  numMatches := 0:

  for i from 1 by 1 to numPts do
    if allPts[i,2] = ( subs(x=allPts[i,1], anExpression) mod
    theMod ) then
      numMatches := numMatches + 1:
    end if:
  end do:

  printf ("numMatches is %d and t is %d and degree is %d\n",
```

```
    numMatches, t, degree(anExpression,x));
  return numMatches >= t:
end proc:



# Procedure to build the terms for Sudan's algorithm
# this creates Qxy

generateAllTerm := proc(l::integer, m::integer, d::integer)
  local allTheTerms:
  allTheTerms := simplify(sum(sum(y^k*x^j, j = 0 .. m+(l-k)*d), k
    = 0 .. l)):
  return simplify(allTheTerms):
end proc:



# a procedure to make aPoly = (aVar - f(x)) and then find the
    inverse of
# f(x) mod primeField
# Requires the coefficient of aVar is a constant

findInverse:=proc(aPoly, aVar, primeField)
  local theCoeff, newPoly:
  # Look through aPoly to see if it is linear in aVar
  # if it is, then find aVar-newPoly := aVar (mod primeField)
  # Then find the inverse of newPoly (mod primeField)
  # return this value, or -1 if not found
  if type(aPoly, linear(aVar)) = false then
    return -1:
  end if:
  # extract the coefficient of aVar
  theCoeff := coeff(aPoly,y,1):

  # make sure the coefficient is a constant
  if type(theCoeff, const) = false then
    return -1:
  end if:

  newPoly := (-1*(aPoly - theCoeff*aVar)) mod primeField:
  # if the factor is just c*y, then 0 does not have an inverse
    (i.e. newPoly = 0 now)
  if newPoly = 0 then
    return -1:
  end if:
```

```
  newPoly := (newPoly / theCoeff) mod primeField:
  newPoly := simplify(newPoly):    # standard way to write

  return newPoly;
end proc:



# this procedure will generate n random points with success
    probability p
# of the function thePoly, and will generate a random field
    element otherwise
# the field elements are from 0 to (theMod - 1)
# This function returns a list n by 2 matrix of the (x,thePoly(x))
    points
# if a fifth argument is included, it is a list of points with the
    x values entered.

genListPoints := proc(n::posint, thePolyB, theMod, p)
  local numCorrPoints, thePoints, THEMAXRAND, theNoise,
    noiseLevel, roll, i, aVal:
  numCorrPoints := 0:    # for our records, how many successful
    points

  # for randomness of the x values and for probability
  THEMAXRAND := 1000000:
  theNoise := rand(1..THEMAXRAND):
  noiseLevel := ceil(p*THEMAXRAND):  # if below this value, success
  roll := rand(0 .. theMod-1):

  # See if the user sent in the points
  if nargs(args) = 4 then
    thePoints := matrix(n, 2):
    for i from 1 to n do
      thePoints [i, 1] := roll();
    end do:
  else  # user sent in the points
    thePoints := args[5]:
  end if:

  # generate the (x,y) points
  for i from 1 to n do
    aVal := theNoise():
    if p = 0 then
      thePoints [i, 2] := roll():
```

```
    elif aVal < noiseLevel then
      # generate a real point
      thePoints [i, 2] := thePolyB(thePoints[i, 1]) mod theMod;
      numCorrPoints := numCorrPoints + 1:
    else
      # generate a random point
      thePoints [i, 2] := roll():
      if thePoints[i,2] = ( thePolyB(thePoints[i,1]) mod theMod)
    then
        numCorrPoints := numCorrPoints + 1:
      end if:
    end if:
  end do:

  printf("Number of actual points generated by %a is: %d\n",
    thePoly, numCorrPoints);

  return thePoints:
end proc:



# this procedure will generate n random unique x coordinates
# the field elements are from 0 to (theMod - 1)
# This function returns a list n by 2 matrix of the (x, - ) points

genListUniqueX:= proc(n::posint, theMod)
  local THEMAXRAND,roll, i, aVal, allPossPts, thePts, tempPoint, j;
  thePts := matrix(n, 2);

  if n > theMod then
    printf("Not enough field elements\n");
    return:
  end if:

  # This will tell us if the point has been chosen already
  allPossPts := matrix(theMod, 2):

  for i from 1 to theMod do
    allPossPts[i,1] := i mod theMod:
    allPossPts[i,2] := 0:  # not yet selected
  end do:

  roll := rand(1 .. theMod):
```

```
  for i from 1 to n do
    tempPoint := roll();
    for j from 1 by 1 while allPossPts[tempPoint, 2] = 1 do
      tempPoint := roll();
    end do:

    # now have a unique x coordinate, add it
    allPossPts[tempPoint,2] := 1;
    thePts[i,1] := tempPoint mod theMod:
  end do:

  return thePts:
end proc:





#**************************************************************
#**************************************************************
#**************************************************************
# NOW the new way to call the procedure
myN:= 16:  # number of points to generate
theMod := 17:  # We are working in Z mod theMod

# This is the probability of selecting a random point
p := .7:  # the probability that the random point will be from the
    function

f := x -> 4*x^3 + 9*x^2+3*x+2 mod theMod:  # the polynomial we are
    trying to find
d := 3;    # the maximum degree of a polynomial we are looking for
n:= 16:
m:=ceil(d/2) - 1;
m:= 1:
l := ceil ( sqrt(2*(n+1)/d) ) - 1;
l:= 3:
# The number of places in which a factor must match the list of
    points
t := max(m+l*d+1, d*(l+1)-m + 1):

# generate all the terms needed
allTerms := generateAllTerm(l, m, d):

# Create unique x points
theMat := genListUniqueX(myN, theMod):
```

```
# assign y values to these points
theMat := genListPoints(myN, f, theMod, p, theMat):
theMat := evalm(theMat):

# Call sudan's list decoding algorithm
A := sudanCoeff(myN, d, t, m, l, theMat, theMod):
```