March 2010

SRAM PUF Analysis and Fuzzy Extractors

Christopher J. Trufan
Worcester Polytechnic Institute

Isaac Clark Edwards
Worcester Polytechnic Institute

Patrick Joseph Newell
Worcester Polytechnic Institute

Project Number: WJM5000

SRAM PUF Analysis and Fuzzy Extractors

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:

Isaac Edwards

Patrick Newell

Chris Trufan

Date: March 1, 2010

Approved:

Professor William J. Martin, Major Advisor

Professor Berk Sunar, Co-Advisor

Abstract

In this project, we investigate authentication systems that utilize fuzzy extractors and Physically Unclonable Functions (PUFs) to uniquely identify hardware components. More specifically, we seek to verify authenticity using PUFs based on Static Random Access Memory (SRAM). We propose an implementation of a fuzzy extractor in software which allows for the extraction of uniquely identifying information from this type of PUF. Included is a working prototype and framework for this authentication system to facilitate future research on this topic.

Table of Contents

1 Introduction

In today's world of globalization in the electronics industry, the continuing search for the lowest bidder often results in components that underperform, perform incorrectly or simply don't perform at all. In the consumer electronics market, this is usually only a minor annoyance as a legitimate supplier will generally refund or replace the components in question. However, in a secure or other high-risk situation, not only must the components perform reliably and correctly, but they must also only perform the operations the original manufacturer specifies. Many companies manufacture components outside of their supervision where defects or malicious functionality can be easily introduced. For example, it is often cheaper for a company to design a component in the U.S. and have it manufactured in another country with cheaper labor rates. To put this into perspective, the Pentagon produces in secure facilities only about two percent of the components used in military systems [1]. Recently, in 2008, the F.B.I. and the Pentagon discovered that a large amount of counterfeit networking components were being used by U.S. military agencies [2]. While these components in particular were not constructed with malicious intent, the possibility of a hostile entity hiding the relatively small amount of functionality needed to compromise secure systems within the highly complex hardware used today is very real and dangerous.

To solve this problem, we need an inexpensive form of hardware authentication for both components within secure systems and components that interface with those systems. This authentication is beneficial for all parties involved with the secure application. For the designers of the secure application, it ensures that they are using the correct components within their system and that those components will withstand given tolerances and handle given situations. For the manufacturers of the components, it ensures that the components being used in a system are produced following their specifications and are not counterfeit or otherwise defective. For the users of the system, it ensures that both the system and the user are legitimately engaging in use of the system and activities on that system are done so over a defined set of rules and security levels. For example, the components of a life support system must be of the highest quality so that those relying on the system are not put in danger. At any given point during the life cycle of the system, we should know that every component is working as specified.

To achieve these goals in a hardware-based environment, we call upon the services of PUFs or Physically Unclonable Functions. The term PUF is used to describe a broad class of algorithms and measuring protocols that produce uniquely random data thanks to miniscule differences in construction brought about by the fact that we live in an analog world. We can use these PUFs as a sort of fingerprint for hardware devices, a way to identify and differentiate them from the time they are born, that is when

they are manufactured. In addition, these PUFs are impossible to reproduce or tamper with without destroying the hardware that generates the unique fingerprint.

One particular source of PUFs is Static Random Access Memory. SRAM produces large quantities of unique static data. However, a portion of this data is corrupted by random noise. Nonetheless, we can still produce valuable fingerprints with this data through the use of a fuzzy extractor. In general a fuzzy extractor eliminates noise to a certain degree while retaining the uniqueness of the static data. With careful analysis of the data provided we can produce a fuzzy extractor that performs an integral role in a hardware authentication system.

A company interested in secure hardware authentication is defense contractor General Dynamics C4 Systems (GDC4S). GDC4S develops and integrates communication systems in a wide variety of secure scenarios. GDC4S thoughtfully provided our project group with equipment and resources that allowed us to complete this project and we thank them for their efforts. In addition, a previous MQP performed at GDC4S had the goal of producing and analyzing PUF data. The PUF source they used was SRAM startup values from two Field Programmable Gate Arrays (FPGAs). This data was provided to us for use in our project and we thank them for their work as well.

To approach this problem we first analyzed the data provided to us. We extracted properties from this data such as correlation between data bits, entropy, and noise levels. Using this analysis, we were able to determine a suitable bit-extraction procedure and optimal parameters for a fuzzy extractor tailored to this PUF. Simultaneously, we took a closer look at the underlying mechanics of a fuzzy extractor authentication system and produced a framework for rapidly building and testing specific system configurations. Finally, we implemented a fuzzy extractor system using this framework based on our analysis of the data provided. Additionally, we developed some of the more complex components required to deploy this system in a real-world setting such as encoding and decoding implementations for BCH codes.

How to Read this Paper

This report covers the process of designing and implementing this fuzzy extractor authentication system from a top-down perspective. While the entire process is detailed in this report, those with a more focused interest should refer to specific sections. For those interested in simply using the framework developed in this project, refer to Section 4.2, Building a Simple Authentication System. For those interested in extending or modifying the existing system, refer to the entirety of Section 4. For those

interested in using this system as a foundation for a hardware or system-level software implementation, refer to Sections 2 and 5.

2 Background

In this project, we seek to construct an authentication system that uses a Physically Unclonable Function, or PUF, as a means of verification. Based on previous research, we know that the unique properties of a specific kind of PUF, known as a challenge-response PUF, can be harnessed for authentication purposes through the use of a fuzzy extractor. However, before delving into the design and construction of the fuzzy extractor authentication system that utilizes PUFs based on Static Random Access Memory, or SRAM, it is essential that we first obtain an understanding of its theoretical background. Many of the design considerations of this system are based on these underlying mathematical foundations. Therefore, we will briefly look into the core aspects of these foundations as they apply to this project.

In this section, we examine the basic structure of a fuzzy extractor and how it is used with Physically Unclonable Functions. Next, we will obtain a more in-depth understanding of a fuzzy extractor by examining the concept of BCH coding theory and that of universal families of hash functions. Finally, we assemble this information into a concise set of processes that summarize the basic functionality of a fuzzy extractor authentication system. The individual subsections that are of particular relevance to the scope of this project are sections 2.1 and 2.4, which cover only the essential elements of information required to understand the basic, higher-level functionality of a fuzzy extractor authentication system. While the remaining sections, 2.2 and 2.3, cover useful theory behind the lower-level encoding and hashing mechanisms of a fuzzy extractor, only a minimal understanding of these concepts is needed to design an effective authentication system from an engineering perspective.

2.1 Fuzzy Extractors and Physically Unclonable Functions

As discussed previously, we seek to build a fuzzy extractor authentication system that utilizes Physically Unclonable Functions. More specifically, we seek to utilize the properties of challenge-response PUFs based on Static Random Access Memory, or SRAM. Before discussing SRAM and how it presents a suitable PUF, let us first establish a definition for a general PUF. In Authentication Schemes based on Physically Unclonable Functions, a PUF is defined as “a practical device or process with a measurable output that strongly depends upon physical parameters” [3]. In other words, a PUF can provide us with a quantifiable output from a function that is greatly affected by one or more physical properties. Regarding the scope of this project, we are primarily concerned with the family of challenge-response PUFs, which returns a physically-dependent response when given a particular challenge. As a

trivial example, we can imagine a circuit that simply consists of a pulse generator connected to an arbiter by two wires. When given a challenge, the pulse generator produces a signal that travels through both wires and the arbiter outputs either a 0 or 1 depending on which wire the signal arrived from first. In other words, the output of this circuit is dependent on a race condition. In this case, slight variations in the properties of the physical circuit, such as differing wire lengths, can easily affect the output. Therefore, the response of one circuit may be different than the output of another circuit when given the same challenge.

Let us now discuss how SRAM can be used as a challenge-response PUF. SRAM is a type of memory that consists of multiple memory cells made up of several transistors. The manufacturing processes used introduce miniscule variations in these cells that bias their start-up values. As in typical computer memory, these are binary values. These variations form the basis for our SRAM PUF.

Since we know that the contents of SRAM are heavily dependent on the characteristics of the physical implementation, then we can take advantage of this to produce a unique fingerprint of the device. However, the process of generating unique information based on this PUF is greatly impeded by noise, which can be introduced by numerous sources. If two individual readings of SRAM from the same device are not exactly identical as a result of noise, we cannot expect to use this data for identification without some form of error correction.

An efficient means by which to remove this inherent noise and extract a unique key from the contents of SRAM is to use a fuzzy extractor. A fuzzy extractor is a process that accomplishes this by first averaging multiple readings of SRAM data in order to estimate a single noiseless reading. This reading is then used to create two elements of information: a hashed value and a “secure sketch”. The hashed value is generated by first hashing the noiseless reading using a universal family of hash functions. The resulting digest is then passed through a second, cryptographic hash function to produce the final hashed value. The secure sketch is generated by using a Code-Offset construction, which in this phase, adds the reading to a randomly selected codeword from a set of error correcting codes using a bitwise XOR operation. In this project, we use BCH codes for this construction.

This secure sketch and hashed value can then be used to efficiently authenticate noisy SRAM readings by first subtracting a reading from the secure sketch to produce a codeword with added noise. An error-correcting scheme is then used to decode the original codeword, which is subsequently subtracted from the secure-sketch to produce the original noiseless reading. Finally, this noiseless reading is hashed and compared to the original hashed value. This result can be used to verify the authenticity of a device.

In addition to providing an efficient means of error-correction, the fuzzy extractor also allows for privacy to be maintained if only the hash and secure sketch are stored in the authentication system. The universal family of hash functions provides what is known as "privacy amplification" by producing a uniformly distributed set of output values from a set of input values that are not uniformly distributed. In other words, for every hashed digest produced by this universal hash function, there are always an equal number of possible input values from which it could have been derived. Take for example that the input to a particular universal hashing function is of size 8 and the output is of size 4. For any given output, there are $2^{(8-4)}$, or 16, possible input values. The secure sketch also provides minimal information regarding the original noiseless reading due to the fact that it is masked by a randomly-selected codeword of equal length.

2.2 BCH Codes

As we discussed earlier, an important portion of the fuzzy extractor system is noise removal. To achieve this noise reduction, we employ BCH error correcting codes. BCH codes are a family of error correcting codes which were created by Hocquenghem and independently by Bose and Ray-Chaudhuri at around the same time. Error-correcting codes are a class of schemes for encoding messages so that they can be recovered correctly when there is noise introduced in the sending or receiving of the message. For example, a simple coding scheme over a binary message would replace every 0 with 000 and every 1 with 111. To decode this on the receiver's end, we simply look at each triplet and reduce it to either a 0 or 1 based on which value appeared more often in the triplet. This allows us to correct one binary error per binary triplet but increases the length of the code to three times the message length. BCH codes are a very flexible set of codes in that within certain bounds there is a great amount of choice in code parameters and are relatively efficient in message length and error correction. The code parameters are as follows:

- q – A prime power which will determine the number of symbols used (e.g., $q = 2$ is binary)
- m – The power to which to raise q to generate a Galois Field for the construction of the code.
- d – The minimum Hamming distance between distinct codewords.

These parameters lead to several derived parameters which are standard parameters of linear codes:

- n – The block length of the code; for our special case, $n = q^m - 1$
- t – The number of errors that can be corrected; this is the largest integer satisfying $d \geq 2t + 1$
- k – The number of message bits in a codeword; in our case, this satisfies $k \geq n - mt$

Any choice of parameters following these guidelines will result in a valid code.

Much of the math in this section comes from Elwyn R. Berlekamp's book Algebraic Coding Theory[4].

2.2.1 Galois Fields

Galois fields or finite fields are fields which contain only finitely many elements. Informally, a field is a number system in which we may add, subtract, multiply and divide. For example, we can consider the integers modulo 7. Addition, subtraction, multiplication and division (by nonzero elements) on the integers modulo 7 receive as parameters integers between 0 and 6 and each produce a result in the same integer range. Galois fields are classified by size and there are Galois field of size 4, 8, 9, 16, 25, 27, etc. In fact there is exactly one Galois field of size p^m for every prime p and every positive integer m. Each field contains p^m elements, two of which are called 0 and 1, and has defined on it two binary operations, addition and multiplication. These two operations, given any field elements x and y, yield field elements $x + y$ and $x * y$. The axioms require that we may freely move parentheses, reorder a sum or product, distribute a product over sums and that there are special elements 0 and 1 that obey the following rules:

- 0 + x = x; for any x
- 0 * x = 0; for any x
- 1 * x = x; for any x

The rest of the outcomes of the two operations must be defined such that they take in only elements from the field and output only elements from the field. For example, here we have a field with $p = 2$ and $m = 2$, called GF(2^2) or GF(4): the addition (Cayley) table and multiplication table are

+	0	1	A	B
0	0	1	A	B
1	1	0	B	A
A	A	B	0	1
B	B	A	1	0

*	0	1	A	B
0	0	0	0	0
1	0	1	A	B
A	0	A	B	1
B	0	B	1	A

Each element appears exactly once in each row and column of either table, with the exception that the 0 * x = 0 for any element x.

2.2.2 Primitive Polynomials

A primitive polynomial for a Galois field GF(p^m) is a polynomial *f(x)* with coefficients in GF(p) that has a root α in GF(p^m) which is very special: α, when raised to each power up to $p^m - 2$, generates $p^m - 2$ different elements in the field.

In fact, tools from abstract algebra allow us to identify the elements of GF(p^m) with exactly the set of polynomials of degree less than m over GF(p), but the multiplication of polynomials requires a modular reduction (as in the Division Algorithm) based on the irreducible (i.e. unfactorable) polynomial *f(x)* of degree m. From this viewpoint, α is identified with the polynomial x and raising the α to every power from 0 to $p^m - 2$ generates the entire field except 0. For example, a primitive polynomial for GF(8) is $f(x) = x^3+x+1$.

Power	Binary	Polynomial
0	000	0
1	001	1
α	010	α
α^2	100	α^2
α^3	011	α+1
α^4	110	$\alpha^2+\alpha$
α^5	111	$\alpha^2+\alpha+1$
α^6	101	α^2+1

While generating the field, the Division Algorithm can be implemented by a simple substitution: when some power of α, in polynomial form, reaches degree m (e.g. $m = 3$ in our example), the primitive polynomial is used to substitute $x^3 = x + 1$ to reduce the polynomial to one of degree less than m. In addition, because we are working in GF(2^3), i.e. $p = 2$, all of the coefficients are binary. This means that if we raise α to the sixth power, we get $\alpha^5 * \alpha = (\alpha^2 + \alpha + 1)* \alpha = \alpha^3 + \alpha^2 + \alpha$. Substituting the primitive polynomial $\alpha^3 = \alpha + 1$, this becomes $\alpha^2 + \alpha + \alpha + 1 = \alpha^2 + 1$. Likewise, taking α to the seventh power brings the value back to 1: $(\alpha^2 + 1)*\alpha = \alpha^3 + \alpha = \alpha + \alpha + 1 = 1$. Alternatively, we can use the curious fact that, over the binary field, $(y + z)^2 = y^2 + z^2$ and find $\alpha^6 = (\alpha^3)^2 = (\alpha + 1)^2 = \alpha^2 + 1$.

2.2.3 Minimal Polynomials

The next step toward BCH encoding is generating the minimal polynomials for elements in GF(p^m). To create the BCH generator polynomial, we must multiply together the minimal polynomials of α^k for k = 1 to k = d-1: g(x) = $m_1(x) * m_2(x)*\ldots*m_{d-1}(x)$. To generate each minimal polynomial we must solve a linear system containing powers of α according to the above rules for polynomial multiplication. First, to get the *i*th minimal polynomial, we take powers of α following α^{ik} where *k* is a positive integer (or zero) and find the first linear combination of powers which when added together produce 0. These powers then become the exponents of the minimum polynomial. For example, continuing with the primitive polynomial $f(x) = x^3 + x + 1$ and using the table above, to find the minimal polynomial of α we generate consecutive powers of α^1:

Power	Binary	Polynomial
α^0	001	1
α^1	010	α
α^2	100	α^2
α^3	011	α + 1

Clearly $\alpha^3 + \alpha^1 + 1 = 0$ and no lower degree combination will give us zero. So our minimal polynomial for α^1 is $m_1(x) = x^3 + x^1 + x^0 = x^3 + x + 1$. Notice that this minimal polynomial is always equal to the primitive polynomial *f(x)* used to generate the Galois field. To generate the minimal polynomial of α^2, we generate powers of α^2 until we find a linearly dependent set:

Power (of x)	Power (of α^2)	Binary	Polynomial
0	α^0	001	1
1	α^2	100	α^2
2	α^4	110	$\alpha^2 + \alpha$
3	α^6	101	$\alpha^2 + 1$

In this case adding together α^6, α^2 and α^0 equals zero. So the polynomial $x^3 + x + 1$ has α^2 as a root and no polynomial of smaller degree achieves this. So we have our minimal polynomial: $m_2(x) = x^3 + x^1 + x^0 = x^3 + x + 1$. In this case, we have gotten the same result as the first example. This is because $m_1(x) =$

$m_2(x)$ in the binary case. In fact, the equation $(y + z)^2 = y^2 + z^2$, that whenever β is a root of $m(x)$, so also are β^2, β^4, …

Finally, we will take a look at an example which will result in a minimal polynomial that is not equal to the primitive polynomial we use. For this example, we use the primitive polynomial $f(x) = x^4 + x + 1$ and attempt to find the third minimal polynomial.

Power (of x)	Power (of α^3)	Binary	Polynomial
0	α^0	0001	1
1	α^3	1000	α^3
2	α^6	1100	$\alpha^3 + \alpha^2$
3	α^9	1010	$\alpha^3 + \alpha$

In this case, computing only four powers is not enough to find a linear combination. Notice that there is a 1 to be cancelled but no other power has a 1. We must continue generating powers until we find a linear combination:

Power (of x)	Power (of α^3)	Binary	Polynomial
0	α^0	0001	1
1	α^3	1000	α^3
2	α^6	1100	$\alpha^3 + \alpha^2$
3	α^9	1010	$\alpha^3 + \alpha$
4	α^{12}	1111	$\alpha^3 + \alpha^2 + \alpha + 1$

We see that the fourth power of x finally gives us a linear combination between all of the powers. Since we use all of the powers of x, the third minimal polynomial for the primitive polynomial $f(x) = x^4 + x + 1$ is $m_3(x) = x^4 + x^3 + x^2 + x + 1$.

2.2.4 BCH Generator Polynomial

Now that we have minimal polynomials, creating the BCH generator polynomial is simply a matter of multiplying the odd numbered minimal polynomials together from $m_1(x)$ to $m_{d-1}(x)$. We must remember that we are still working with q-order coefficients at this point, and in the binary case this means even coefficients are cancelled out and odd coefficients are reduced to 1. For example, if we want to find the BCH generator polynomial for a code with length 15 and distance of anywhere from 2 to 3, which are all the same code, we find the case where the generator polynomial is simply $m_1(x)$. If

we want to find the polynomial for a code with length 15 and a distance of 4 or 5, we must multiply $m_1(x)$ by $m_3(x)$:

$m_1(x) = x^4 + x + 1$

$m_3(x) = x^4 + x^3 + x^2 + x + 1$

$(x^4 + x + 1)(x^4 + x^3 + x^2 + x + 1)$

$= x^8 + x^7 + x^6 + x^5 + x^4 + x^5 + x^4 + x^3 + x^2 + x + x^4 + x^3 + x^2 + x + 1$

$= x^8 + x^7 + x^6 + \cancel{x^5} + \cancel{x^4} + \cancel{x^5} + \cancel{x^4} + \cancel{x^3} + \cancel{x^2} + \cancel{x} + x^4 + \cancel{x^3} + \cancel{x^2} + \cancel{x} + 1$

$= x^8 + x^7 + x^6 + x^4 + 1$

This polynomial is the BCH generator polynomial for a code with parameters $q = 2$, $m = 4$ and $d = 5$.

2.2.5 BCH Encoding

The final step in encoding a message with BCH is fairly brief. First, we convert the message to be encoded into a polynomial with q-order coefficients and pad the right with a number of zeros equal to the order of the BCH generator polynomial. Second, divide the message by the BCH generator polynomial. The remainder of this division is padded with zeros on the left as necessary to have it be the same order as the generator polynomial and then appended to the message to form the codeword. For example, if we want to encode the number 5 with the code of length 15 and distance 5 from the previous section:

BCH Generator Polynomial = $x^8 + x^7 + x^6 + x^4 + 1$

Message = $x^2 + 1$

Since we are using a binary code, the polynomials we are dividing have binary coefficients, this allows us to do the division as a series of shifted XORs between bit strings.

BCH Generator Polynomial = 111010001

Message = 0000101

Padded Message = 10100000000

```
111010001 | 10100000000
          111010001
           100100010
           111010001
            111100110
```

```
          111010001
             110111
```

Remainder = 110111

Padded Remainder = 00110111

Codeword = 000010100110111

2.2.6 Syndrome Computation

Now that we can encode messages, we will move on to decoding messages. The first step in decoding messages is to compute the syndromes of the received codeword. If our code can correct *t* errors, we must compute *2t* syndromes. We compute the *i*th syndrome $S_i(x)$ by taking a modulo reduction of the codeword by the *i*th minimal polynomial $m_i(x)$. We perform this modulo reduction by dividing and taking the remainder as we did for the final step of encoding. If all of the syndromes in terms of *x* are zero, the codeword was received without error and we can stop the decoding here. However, if they are nonzero we must move on to the next step, which involves substituting increasing powers of α for x and reducing the resulting equation to a single term power of α. For example, let us first assume we have received the codeword calculated in the previous section without error:

Codeword = 000010100110111

The code has a distance of 5 so it can correct 2 errors. For 2 errors, we must construct 4 syndromes.

$m_1(x) = x^4 + x + 1$

$m_2(x) = x^4 + x + 1$

$m_3(x) = x^4 + x^3 + x^2 + x + 1$

$m_4(x) = x^4 + x + 1$

Since 3 of the 4 syndromes are the same, to compute syndromes of x we need only compute two modulos.

```
10011 | 10100110111
        10011
          11111
          10011
           11000
           10011
            10111
```

```
    10011
      10011
      10011
          0
```

We could also remember the fact that we can use $x^4 + x + 1$ *as* $x^4 = x + 1$ *and reduce as follows:*

Codeword = $x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$

= $x^4x^4x^2 + x^4x^4 + x^4x + x^4 + x^2 + x + 1$

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

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

= $x^4 + \cancel{x^2} + \cancel{x^2} + x + 1$

= $\cancel{x} + \cancel{1} + \cancel{x} + \cancel{1}$

= 0

Computing the other syndrome:

```
11111 | 10100110111
        11111
         10111
         11111
          10001
          11111
           11100
           11111
              11111
              11111
                  0
```

Since all of the syndromes are zero, we know the codeword was received without error and to retrieve the message, we simply remove the 8 parity check bits added by the encoding. Next we will compute the syndromes of a more interesting example:

Codeword = 10100110111

Codeword with errors = 10**0**0**1**110111

First, second and fourth syndromes:

```
10011 | 10001110111
        10011
           10110
           10011
             10111
             10011
               1001
```

Third syndrome:

```
11111 | 10001110111
        11111
         11101
         11111
            10101
            11111
             10101
             11111
              10101
              11111
               1010
```

Now we have the syndromes in terms of x*:*

$S_1(x) = x^3 + 1$

$S_2(x) = x^3 + 1$

$S_3(x) = x^3 + x$

$S_4(x) = x^3 + 1$

We now compute the syndromes in terms of increasing powers of α*:*

$S_1(\alpha) = \alpha^3 + 1$

$S_2(\alpha^2) = (\alpha^2)^3 + 1$

$S_3(\alpha^3) = (\alpha^3)^3 + \alpha^3$

$S_4(\alpha^4) = (\alpha^4)^3 + 1$

To reduce these equations to single term powers of α*, remember that we can multiply powers of* α *as normal, but when adding we must XOR the values because we have binary coefficients. Finally once we have a single value, we must look at the GF table generated by our primitive polynomial and find the power of* α *that the value corresponds to.*

$S_1(\alpha) = \alpha^3 + 1 = \alpha^{14}$

$S_2(\alpha) = \alpha^6 + 1 = \alpha^4\ \alpha^2 + 1 = \alpha^3 + \alpha^2 + 1 = \alpha^{13}$ (remember that we are working modulo $x^4 + x + 1$)

$S_3(\alpha) = \alpha^9 + \alpha^3 = \alpha^4\ \alpha^4\ \alpha + \alpha^3 = (\alpha+1)(\alpha+1)\ \alpha + \alpha^3 = \cancel{\alpha^3} + \alpha + \cancel{\alpha^3} = \alpha$

$S_4(\alpha) = \alpha^{12} + 1 = \alpha^4\ \alpha^4\ \alpha^4 + 1 = \alpha^3 + \alpha^2 + \alpha + \cancel{1} + \cancel{1} = \alpha^3 + \alpha^2 + \alpha = \alpha^{11}$

2.2.7 Error Locator Polynomial

Now that we have our syndromes, we can move on to the next step, which is computing the error locator polynomial. For this we use the Peterson-Gorenstein-Zierler algorithm, which uses matrices to compute the coefficients of the polynomial. We use the formula $S_{txt}\lambda_{tx1} = C_{tx1}$, where S_{txt} is a t by t matrix of the syndrome values as follows:

$$S_{txt} = \begin{bmatrix} S_1 & S_2 & S_3 & \dots & S_t \\ S_2 & S_3 & S_4 & \dots & S_{t+1} \\ S_3 & S_4 & S_5 & \dots & S_{t+2} \\ \dots & \dots & \dots & \dots & \dots \\ S_t & S_{t+1} & S_{t+2} & \dots & S_{2t+1} \end{bmatrix}$$

and C_{tx1} is a t by 1 matrix of syndrome components as follows:

$$C_{tx1} = \begin{bmatrix} S_{t+1} \\ S_{t+2} \\ \dots \\ S_{2t} \end{bmatrix}$$

The error locator polynomial is the values in λ, which we must compute. To do this, we must find the inverse of S_{txt} and then multiply both sides by the inverse. However, if the determinant of S_{txt} is zero, we cannot find the inverse. If this is the case, we reduce t by 1 and start the Peterson-Gorenstein-Zierler algorithm over. If t reaches 0 and we have still not found an inverse, there are too many errors in the received codeword to be decoded properly. For example, using the syndromes we computed above:

$S_{2x2} = \begin{bmatrix} \alpha^{14} & \alpha^{13} \\ \alpha^{13} & \alpha^{1} \end{bmatrix}$

$C_{2x1} = \begin{bmatrix} \alpha^{1} \\ \alpha^{11} \end{bmatrix}$

We next compute the determinant of S_{2x2}:

$\det(S_{2x2}) = \alpha^{14}\,\alpha^{1} - \alpha^{13}\,\alpha^{13} = \alpha^{15} - \alpha^{26} = 0001 \oplus 1110 = 1111 = \alpha^{12}$

Now we can calculate the inverse of S_{2x2}:

Matrix of minors: $\begin{bmatrix} \alpha^{1} & \alpha^{13} \\ \alpha^{13} & \alpha^{14} \end{bmatrix}$

Inverse: $\begin{bmatrix} \alpha^{4} & \alpha^{1} \\ \alpha^{1} & \alpha^{2} \end{bmatrix}$

Now that we have the inverse of S_{2x2}, we multiply by C_{2x1} to get the coefficients of the error locator polynomial:

$$\begin{bmatrix} \alpha^{4} & \alpha^{1} \\ \alpha^{1} & \alpha^{2} \end{bmatrix} \times \begin{bmatrix} \alpha^{1} \\ \alpha^{11} \end{bmatrix} = \begin{bmatrix} \alpha^{5} + \alpha^{12} \\ \alpha^{2} + \alpha^{13} \end{bmatrix} = \begin{bmatrix} (x^{2} + x) + (x^{3} + x^{2} + x + 1) \\ (x) + (x^{3} + x^{2} + 1) \end{bmatrix} = \begin{bmatrix} x^{3} + 1 \\ x^{3} + 1 \end{bmatrix} = \begin{bmatrix} \alpha^{14} \\ \alpha^{14} \end{bmatrix}$$

The coefficients given by the array from the Peterson-Gorenstein-Zierler are in decreasing order, with the bottom coefficient having order 1 in the polynomial, so the final error locator polynomial is $\alpha^{14}x^{2} + \alpha^{14}x + 1$.

2.2.8 Chien's Search

Now that we have an error locator polynomial, we can use a method known as Chien's Search [5]to find the actual locations of the errors in the received codeword. To perform the search, we take the coefficients of the error locator polynomial and multiply them by α^{i} where i is the power of x of the coefficient in the polynomial. We multiply by these powers of α until adding all the coefficients together yields 1 and count the number of times we multiplied. We then subtract the count from the length of the code n to get the location of the error in the received codeword. We repeat this procedure of multiplying until the coefficients add to 1 until we have done it t times. For example, using the error locator polynomial computed previously:

$\alpha^{14}x^{2} + \alpha^{14}x + 1$

Both coefficients are α^{14}, however we raise them to different powers of α.

	x	x^2	Result
Initial Value	$\alpha^{14} = \alpha^{3} + 1$	$\alpha^{14} = \alpha^{3} + 1$	0
1 Shift	$\alpha^{15} = \alpha^{0} = 1$	$\alpha^{16} = \alpha^{1} = \alpha$	α + 1

2 Shifts	$\alpha^{16} = \alpha^{1} = \alpha$	$\alpha^{18} = \alpha^{3}$	$\alpha^{3} + \alpha$
3 Shifts	$\alpha^{17} = \alpha^{2}$	$\alpha^{20} = \alpha^{5} = \alpha^{2} + \alpha$	α
4 Shifts	$\alpha^{18} = \alpha^{3}$	$\alpha^{22} = \alpha^{7} = \alpha^{3} + \alpha + 1$	$\alpha + 1$
5 Shifts	$\alpha^{19} = \alpha^{4} = \alpha + 1$	$\alpha^{24} = \alpha^{9} = \alpha^{3} + \alpha$	$\alpha^{3} + 1$
6 Shifts	$\alpha^{20} = \alpha^{5} = \alpha^{2} + \alpha$	$\alpha^{26} = \alpha^{11} = \alpha^{3} + \alpha^{2} + \alpha$	α^{3}
7 Shifts	$\alpha^{21} = \alpha^{6} = \alpha^{3} + \alpha^{2}$	$\alpha^{28} = \alpha^{13} = \alpha^{3} + \alpha^{2} + 1$	1
8 Shifts	$\alpha^{22} = \alpha^{7} = \alpha^{3} + \alpha + 1$	$\alpha^{30} = \alpha^{0} = 1$	$\alpha^{3} + \alpha$
9 Shifts	$\alpha^{23} = \alpha^{8} = \alpha^{2} + 1$	$\alpha^{32} = \alpha^{2}$	1

Notice we have a 1 in the result column at 7 and 9 shifts. We subtract 7 and 9 from 15 to get 8 and 6, which are the locations of the errors in the codeword (with the right most bit being bit 0). Since we are using a binary code, we can simply flip the erroneous bits in the received codeword to get the correct codeword.

Codeword = 10100110111

Codeword with errors = 10**0**0**1**110111

Flip bits 8 and 6: 10**1**0**0**110111

2.3 Hashing

Another integral component to the fuzzy extractor is the hashing mechanism, which is employed through a set of hash functions. In general, a hash function is used to produce a uniformly distributed output from an input with an insecure or non-uniform distribution without sacrificing entropy. This allows for us to reduce the input in size while still reflecting the actual information in the output [3]. The hashing mechanism in a fuzzy extractor is essential to enabling the enrollment and authentication phases, while at the same time amplifying privacy.

The underlying process in the hashing mechanism of a fuzzy extractor, as illustrated in Figure 2-1, essentially involves two steps. The first is to use a universal hash function with a randomly selected seed to hash the input values, of w and w'. This universal hash function is guaranteed to generate a uniform distribution of hash values and also amplify privacy. The second step in this process is to hash the result of the universal hash function using a one-way cryptographic hash function. The cryptographic hash function adds an additional layer of security and privacy by ensuring that the original input, w or w', cannot be retrieved from the resulting hash, h. In this section, we are primarily

concerned with the universal hashing component of this system. Therefore, we will discuss the theoretical background of this topic and one specific implementation.

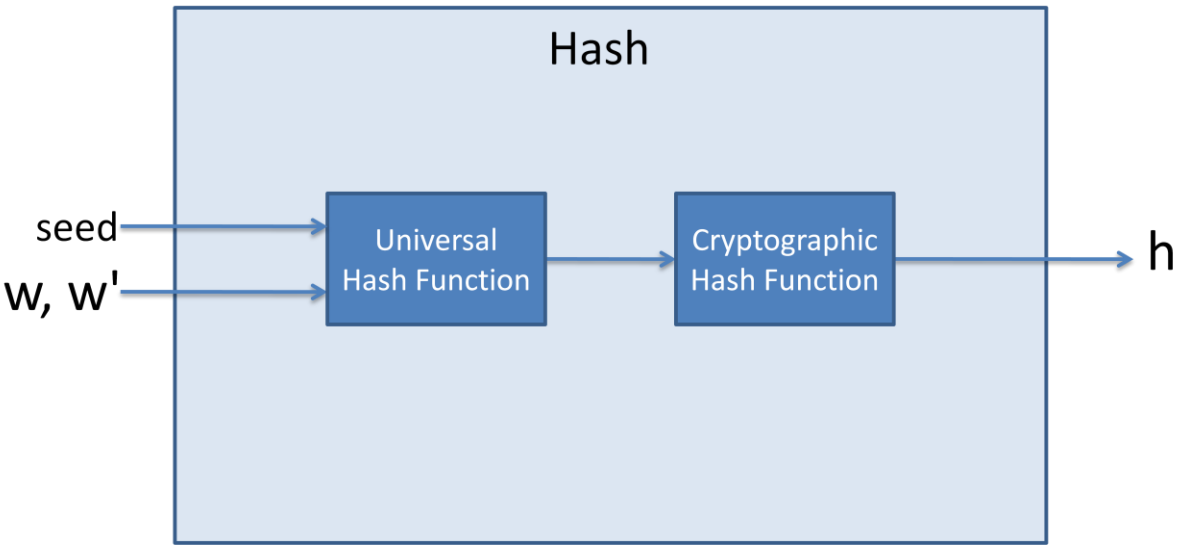

Figure 2-1. Fuzzy Extractor Hash Component

2.3.1 Universal Hashing

Families of universal hash functions are one of the computationally simplest classes of hash functions [6], which create a near evenly distributed, many-to-one relationship between a set of inputs and a set of outputs. An implementation of one of these families of hash functions is essential in the fuzzy extractor, as we need to be able to make strong statements regarding the probability of collisions that come about from two inputs having the same output. In other words, we need an implementation that gives absolutely no additional information to an attacker regarding an input value, given an output value. This will also ensure some degree of privacy, as the outputs of these functions are passed to cryptographic hash functions which require a certain distribution of inputs for maximum proven security.

2.3.2 Theoretical Background

In terms of universal hashing, a family of hash functions can be described as *k-universal or strongly k-universal*, which is defined as follows [6]:

> Let U be a universe with $|\mathrm{U}| \geq \mathrm{n}$ and let $\mathrm{V} = \{0,1, \dots, \mathrm{n}-1\}$. A family of hash functions $\mathcal{H}$ from U to V is said to be k-universal if, for any elements $\mathrm{x}_1, \mathrm{x}_2, \dots, \mathrm{x}_\mathrm{k}$ and for a hash function h chosen uniformly at random from $\mathcal{H}$, we have:

$$\Pr\big(h(x_1) = h(x_2) = \cdots = h(x_k)\big) \leq \frac{1}{n^{k-1}}$$

A family of hash functions $\mathcal{H}$ from U to V is said to be strongly k-universal if, for any elements $x_1, x_2, \ldots, x_k$, any values $y_1, y_2, \ldots, y_k \in \{0,1, \ldots, n-1\}$, and a hash function h chosen uniformly at random from $\mathcal{H}$, we have:

$$\Pr\big((h(x_1) = y_1) \cap (h(x_2) = y_2) \cap \cdots \cap = (h(x_k) = y_k)\big) = \frac{1}{n^k}$$

For the purpose of this project, we are primarily concerned with the family of hash functions that are *2-universal*. When a hash function is chosen randomly from this family, the probability of any two elements, x_1 and x_2, having the same hash is exactly $1/n$.

2.3.3 Privacy Amplification

Following the definition above, we know that a randomly selected hash function will produce the same hash for a given number of inputs within a known probability. That is, given a hash function, h from $\mathcal{H}$, and a hashed value, y, we know that the possible values of x, where $h(x) = y$, are distributed evenly over the domain. From a privacy perspective, this tells us that given any single hashed value, y, an attacker obtains no information whatsoever.

2.3.4 Universal Class of Hash Functions Used in this System

In their paper, Universal Classes of Hash Functions, J. Lawrence Carter and Mark N. Wegman propose several universal families of hash functions that can be evaluated quickly and efficiently. One such family of hash functions, $\mathcal{H}$, is defined as follows [7]:

Let A be the set of i-digit numbers written in base α, and B as the set of binary numbers of length j. Then $|A| = \alpha^i$ and $|B| = 2^j$. Let M be the class of arrays of length iα, each of whose elements are bit strings of length j. For $m \in M$, let $m(k)$ be the bit string which is the k^{th} element of m, and for $x \in A$, let x_k be the k^{th} digit of x written in base α. We define the hash function, $f_m(x)$, as follows:

$$f_m(x) = m(x_1 + 1) \oplus m(x_2 + x_1 + 2) \oplus \ldots \oplus m\left(\left(\sum_{k=1}^{i} x_k\right) + k\right)$$

Due to their simplicity, this family of hash functions is attractive for a software or hardware implementation of a fuzzy extractor. However, we must first prove that this family of hash functions is 2-universal [7]:

Claim: The family of hash functions, $\mathcal{H}$, defined above is 2-universal.

Proof: For x and y in A, suppose $f_m(x)$ is the exclusive-or of the rows $r_1, r_2, \ldots, r_s$ of m, and $f_m(y)$ is $r_{s+1} \oplus \ldots \oplus r_t$. Notice that $f_m(x) = f_m(y)$ if and only if $r_{s+1} \oplus \ldots \oplus r_t = 0$. Assuming $x \neq y$, there will be some k, such that r_k is involved in the calculation of only one of $f_m(x)$ or $f_m(y)$. Then $f_m(x)$ will equal $f_m(y)$ if and only if r_k is the exclusive-or of the other r_i's. Since there are $2^j = B$ possibilities for that row, x and y will collide for one B^{th} of the possible functions f_m. Thus, the class of all f_m's is 2-universal.

Since this family of universal hash functions is proven to be 2-universal, we can use its implementation in the hashing mechanism of our fuzzy extractor in unison with a cryptographic hash function.

To better understand this universal family of hash functions, let us now follow a simple example in the context of our fuzzy extractor system. For the sake of simplicity, let us assume that we are extracting 4 bits of data from an SRAM reading in order to obtain a 2-bit key. That is, we need our universal family of hash functions to compress these 4 bits of raw data into 2 bits of key data. In this example, let the data bits, x, be equivalent to the following:

$$x = 1011$$

We must now construct and populate our two-dimensional array, $m \in M$, with random bits. According to the definition by Carter and Wegman above, m has a size of $i\alpha \times j$, where $i = 4$ (length of the input data), $\alpha = 2$ (as our input data is represented in base 2), and $j = 2$ (the length of the key data). In this example, let this array be equivalent to the following:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \\ 1 & 1 \\ 0 & 0 \\ 1 & 0 \\ 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Now that we have constructed and populated m, we can begin the process of generating our output hash. We begin by first initializing our array index, which we will call $index$, to 0 (we will assume that our array is 1-indexed), and our hashed value, h, to the following:

$$h = 0000$$

We now follow the algorithm as shown below in the following pseudocode:

```
for x = 1 to i
    index = index + input_data[x] + 1;
    h = h ⊕ m[index];
end
```

The values of $index$ and h for every iteration of this algorithm are listed in the table below:

x	$index$	h
1	2	10 (00 ⊕ 10)
2	3	10 (10 ⊕ 00)
3	5	10 (10 ⊕ 00)
4	7	01 (10 ⊕ 11)

This leaves us with a final hashed value of 10 as our key bits.

2.4 Enrollment and Authentication

With our working knowledge of a fuzzy extractor, we can now apply this theory to construct an authentication system. This authentication system is composed of two high-level processes, known as enrollment and authentication. For the purposes of this project, we define *enrollment* as the process by which an entity is registered into the authentication system. In this project, these *entities* are referred to as devices, since the primary scope is that of authenticating hardware devices based on the physical properties of SRAM. We define *authentication* as the process by which the system determines whether or not a given entity, or device, is enrolled in the system. Using the functionality provided by fuzzy extractors, we may implement these two processes to create an authentication system tailored to SRAM PUFs.

2.4.1 The Enrollment Process

The enrollment process is essentially the means by which new devices are registered into the authentication system. Ideally, enrollment would only be performed once per every new device. This process is illustrated in the UML sequence diagram in Figure 2-2.

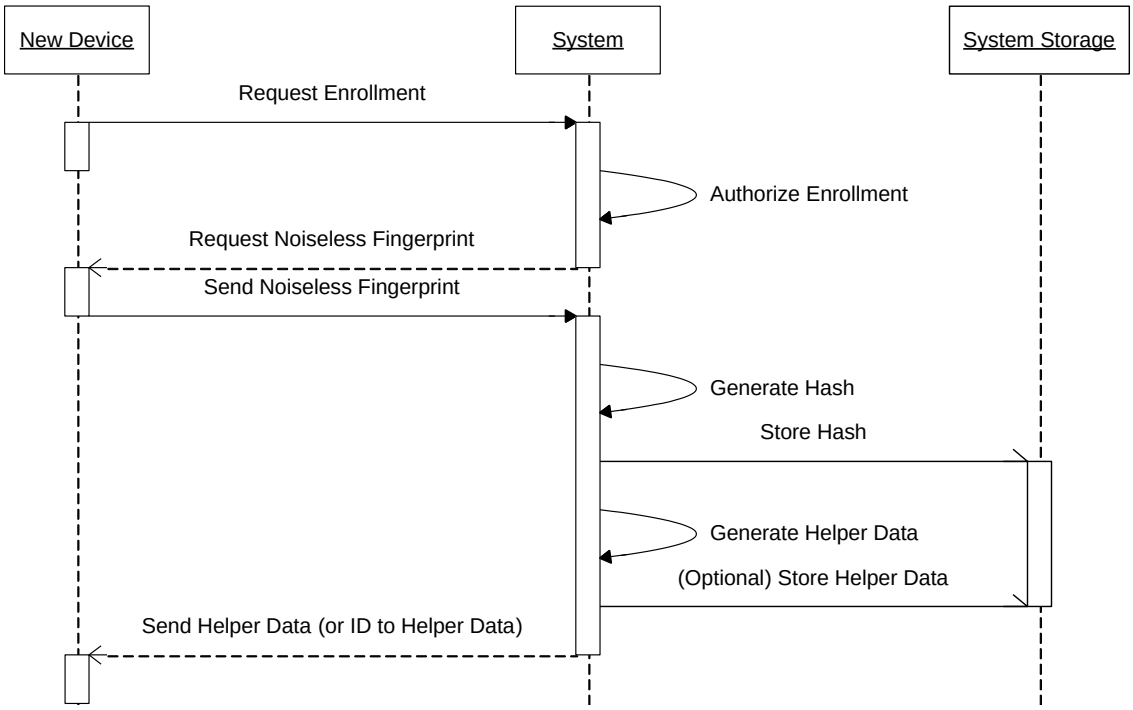

Figure 2-2. UML Sequence Diagram of the Enrollment Process

The general sequence of events as shown by this sequence diagram is as follows:

1. **Request Enrollment**

   A new device sends a request to the authentication system to seeking to enroll.

2. **Authorize Enrollment**

   The authentication system verifies that the device is "OK to enroll" by the administrators of the system. This sub-process would typically be handled by a higher-level protocol.

3. **Request Baseline Fingerprint**

   The authentication system sends a request back to the new device for a reliable baseline reading of its SRAM contents, w.

4. **Send Baseline Fingerprint**

   The new device sends some baseline fingerprint, w, to the authentication system. In practice, this may be achieved via repeated sampling of the device with additional (e.g. statistical) processes. This sub-process can be performed on either the device or the authentication system, or the tasks can be distributed among the two.

5. **Generate Hash**

   The hashing mechanism in the system, as described in Section 2.3, will produce a hash value, h, of w.

6. **Store Hash**

   The hash value generated, h, is stored on some medium in the authentication system for future verification of the device.

7. **Generate Helper Data**

   A helper string is generated by the system that is used to reconstruct the noiseless fingerprint from the noisier samples that will be obtained during the authentication process. The helper string consists of the secure sketch generated by the fuzzy extractor and any other information needed in the authentication process, such as the seed used for the randomly selected universal hash function.

8. **(Optional) Store Helper Data**

   In some implementations, the helper data produced by the fuzzy extractor might be too large in size to feasibly send and retain in storage on the new device. If this is the case, the helper data can also be stored in the authentication system; at the cost of a bit more communication, the device may later retrieve this data by sending an identifier that is much smaller in size.

9. **Send Helper Data (or ID that Maps to Helper Data)**

   The authentication system sends the helper data produced by the fuzzy extractor back to the new device. At this point, the new device is successfully enrolled in the system. Alternatively, if

Step 8 above is invoked, the system by instead send a short identifier which enables quick retrieval of the helper data from system storage.

2.4.2 The Authentication Process

The purpose of the authentication process is to verify that a requesting device has been registered in the authentication system through the enrollment process. This process is summarized in the UML sequence diagram in Figure 2-3.

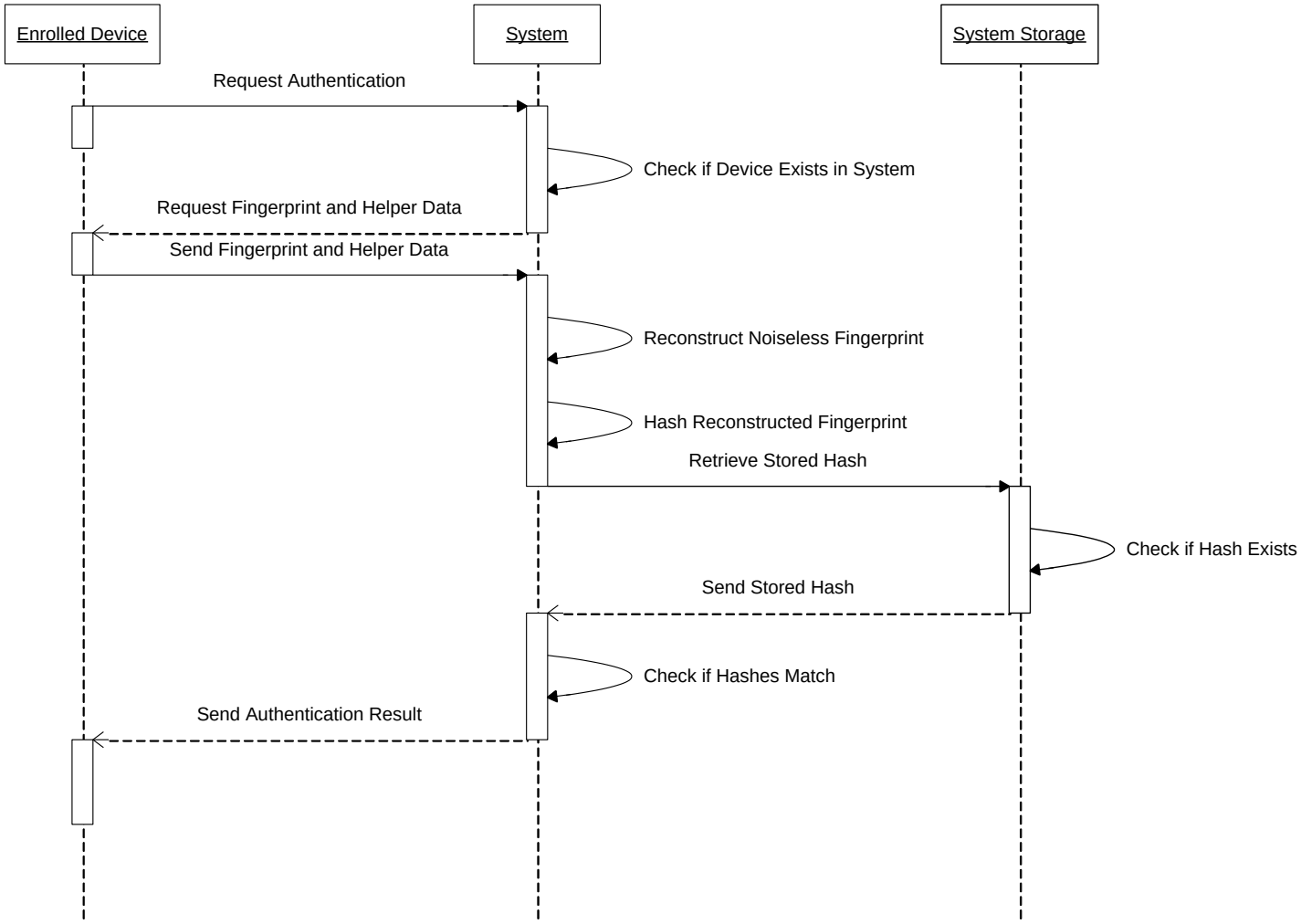

Figure 2-3. UML Sequence Diagram of the Authentication Process

The general sequence of events as shown by this sequence diagram is detailed in the items below. In this particular example, we assume that the device being authenticated has previously been enrolled in the system.

1. **Request Authentication**

   A device that has been previously enrolled in the authentication system makes a request to the system to verify its identity. The details of this event are dependent on the specific implementation. For example, the device might send a hardware address or serial number for higher level verification procedures.

2. **Check if Device Exists in System**

   The system verifies that the requesting device has enrolled in the system based on any initial data provided in the previous step. As with the previous step, the details of this step are dependent on the implementation.

3. **Request Fingerprint and Helper Data**

   The system responds to the requesting device by requesting its helper data (or an index to the helper data) and a single reading of SRAM, w'.

4. **Send Fingerprint and Helper Data**

   The device send w' and its helper data (or an index to the helper data), which was provided to it through the enrollment process, to the authentication system.

5. **Reconstruct Noiseless Fingerprint**

   Upon receiving w' and the helper data from the device, the system attempts to reconstruct w by using the Code-Offset construction process discussed in Section 2.3.1.

6. **Hash Reconstructed Fingerprint**

   The system hashes w using the same hashing mechanism that was used to enroll the device.

7. **Retrieve Stored Hash**

   The system requests a copy of the hashed fingerprint in storage for comparing.

8. **Check if Hash Exists**

   The system checks if a hash of the fingerprint exists in storage.

9. **Send Stored Hash**

   A copy of the stored hash is retrieved from storage.

10. **Check if Hashes Match**

    The system checks if the reconstructed hash and the hash in storage match. A match indicates a successful verification of the device.

11. **Send Authentication Result**

    The system sends the result of the authentication process to the requesting device. Depending on the specific implementation, this would include higher-level processes, such as allowing the device to join a network.

3 System Design

In the previous section, we surveyed the theoretical background necessary to attack the problem at hand. In this section, we perform an information-theoretic analysis of the SRAM device and use this together with the theory to devise an appropriate fuzzy extractor for this device. The system must have a degree of error tolerance that is sufficient to mitigate the inherent noise in SRAM. However, it must also retain privacy and efficiency without sacrificing any of this error tolerance. Therefore, we must analyze the SRAM contents to identify any factors that might impact our bit selection, our choice of BCH code, or how many key bits we can extract. In this section, we will use the contents of SRAM extracted by a previous group of researchers as the sample space for our analysis. We will then convert this data into other formats, analyze any patterns in the data, and calculate optimal parameters for our authentication scheme.

3.1 Data Analysis

In order to utilize the SRAM device as a source for our fuzzy extractor, several steps need to be taken. We will analyze the format of the data we are working with and consider converting it into formats better suited for any calculations we will perform. Then we shall examine whether any bits are better suited than others to be used as input for our fuzzy extractor. We will also undertake an examination to ensure that the bits we choose to use as input do not have any correlations that are unacceptable from a security perspective. This will allow us to choose an optimal pattern to select the bits that we will pass as input to our fuzzy extractor.

3.1.1 Data Format

The initial format for our data is a collection of readings each of which is represented by a 5x262144 double array in MATLAB. Each number is 8 bits, except the 1st row, which has only 4 bits of data representing the 4 parity bits from the board. The remaining 36 bits of a column make up a word. To simplify our data analysis we discard the double containing 4 bits of parity data. This leaves us with 32 bits of data per word, and a 4x262144 double array. In order to work with the data in C as well as in MATLAB, we store our data in binary format as a stream of bytes prefixed by a header indicating how many bytes are in the file. To make analyzing the data easier, we also convert the data in MATLAB into a 1x8388608 array of doubles with values of 0 or 1, each double representing a single bit.

Figure 3-1. Original Data Format as Extracted from SRAM. Each word consists of 5 rows of 8 bits each. The first row of each word contains the parity bits from the SRAM as the first 4 bits of the byte. The remaining 4 bits are the first 4 bits of the 32-bit word.

In addition to simply being able to see which bits are noisy, it will be helpful if we can see how biased bits are as well. The easiest way to do this is simply to sum up the values a bit has across every reading from a single board. If a bit from board 1 has a sum of 1432, we know it is a stable 1, because we have 1432 readings of board 1. Likewise, a sum of 0 means it is a stable 0, while all numbers between 0 and 1432 indicate the bit has some level of noise.

3.1.2 Selecting Bit Extraction Pattern

The first step to constructing our fuzzy extractor is identifying which bits we are going to use to generate our key. The SRAM reading is over 8 million bits long, but we will only need a fraction of these bits to generate our strong key. When choosing which of these bits we will use to generate our key, it is important to determine if any bits are particularly biased. Running a histogram over the 32 bit positions in a word allows us to make sure that within a word no particular bits are more or less likely to be biased one way or another.

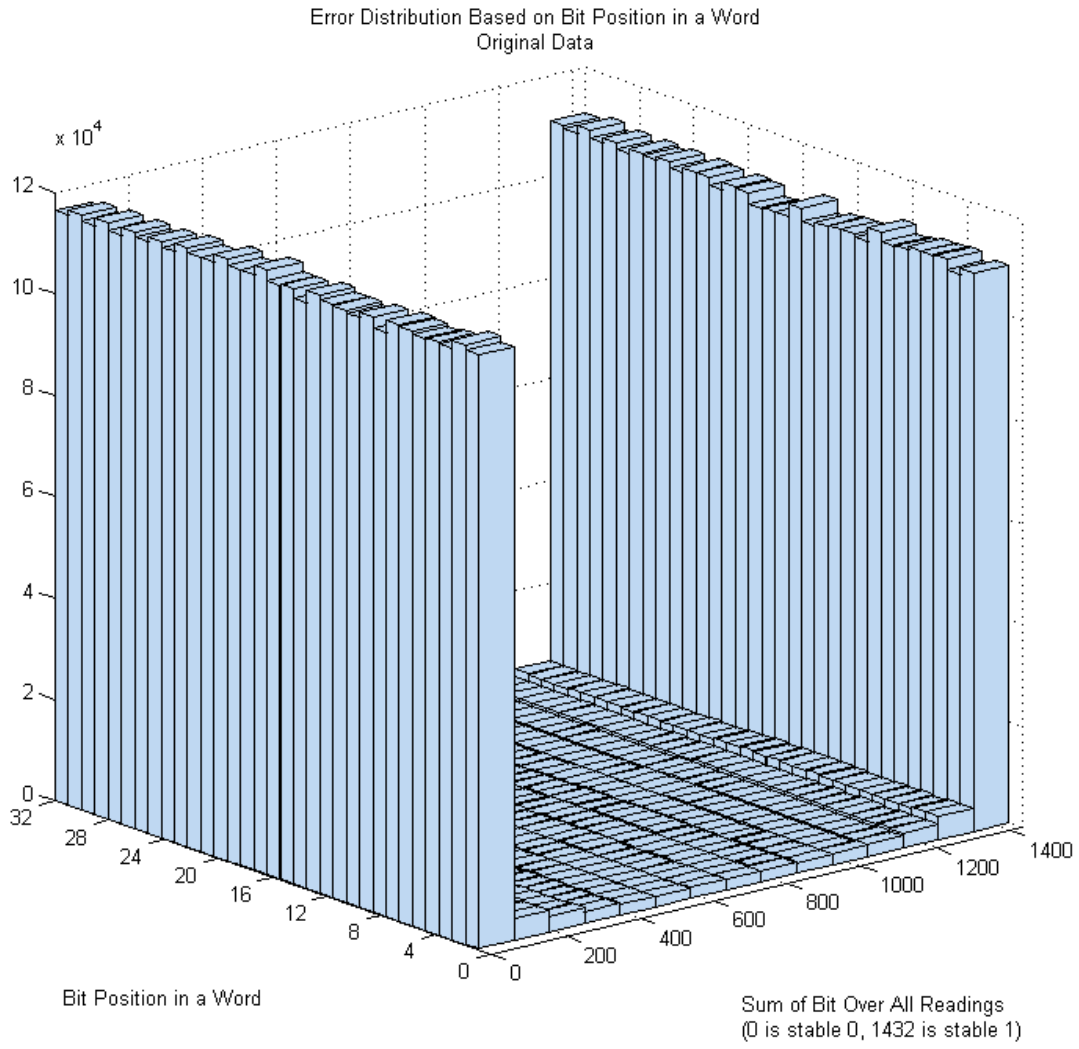

Figure 3-2. Estimated bit values for SRAM locations based on the relative position of the bit in a word. The horizontal axis on the left indicates which relative bit position is being examined, from the 1st bit from every word to the 32nd bit in every word. Locations were each read 1432 times. Horizontal axis on right is number of readings returning 1 (0-1432). A result of 0 would mean the bit was a stable 0, while a result of 1432 would indicate the bit was a stable 1. Vertical axis is the number of occurrences of that particular noise level.

As we can see, the probability that a bit is a stable 0, a stable 1, or noisy is approximately the same regardless of the where the bit is located within a word. Whether we examine the 1st bits of each word or the 20th bits, they have approximately the same distribution of noise. With that knowledge, the remaining factor which might impact our bit selection is whether any of the bits are dependent on each other. For instance, if a bit being 1 means the bit next to it is more likely to be 1, we might not want to take consecutive bits.

To determine whether there are any correlations between bits, we run a MATLAB script that looks at whether bits are a stable 1, stable 0, or noisy, then looks at adjacent bits and sees how often those bits are a stable 1, stable 0, or noisy. If we for instance saw that 60% of the bits following a 1 are 1, but only 30% of the bits following a 0 are 1, that would indicate an unacceptable correlation, as our knowledge of the value an unknown bit might have would chance if we knew a previous bit. Even correlations much smaller than that extreme example might be significant enough from a security position to impact how we select our bits. Our script repeats this analysis for bits farther and farther away from the bit we treat as known. Running this over the entire SRAM reading allows us to see whether knowledge about 1 bit allows us to determine approximately what other bits are going to be.

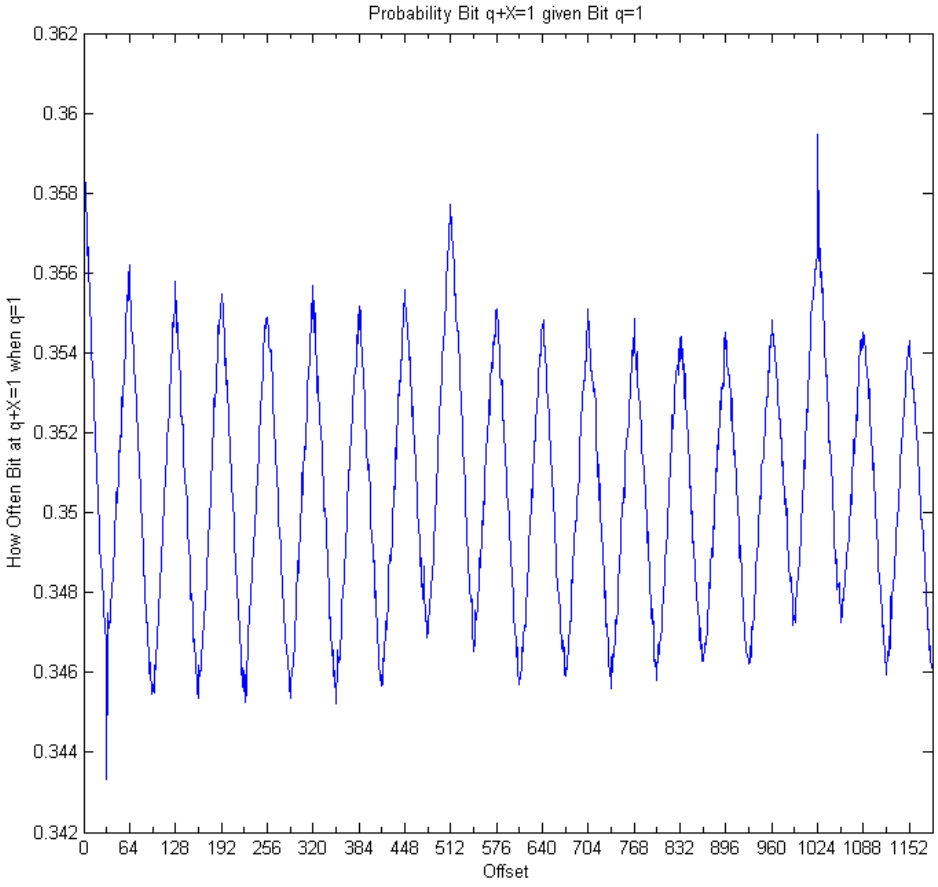

Figure 3-3. Estimating the bit values for SRAM locations compared to known values in nearby locations. The horizontal axis indicates the offset being considered – a value of 1 indicates how often the bit immediately adjacent bit to a stable 1 is going to be a stable 1. The vertical axis indicates the fraction of bits that are 1 that follow a 1 with the given offset. In this graph, the bit immediately adjacent to a stable 1 has a 35.6% chance of being a stable 1, while the bit 32 bits away in the SRAM has a 34.3% chance of being a stable 1.

A very interesting trend seems to appear. If a bit is 1, bits farther and farther away are slightly less and less likely to be 1, until the matching bit in the next word (which is 32 bits away). The matching bit in the next word then seems to be slightly less likely to be 1 than any other bits (and inversely, slightly more likely to be 0). The trend then continues in the other direction, until the matching bit two words away (a 64 bit offset) is slightly more likely to be 1 again – with about the same probability as the immediately adjacent bit! This cycle seemingly repeats for the entire dataset. While seemingly small, these deviations are unacceptable from a security perspective.

We need to figure out what this trend means. If there is a correlation between bits, why is it the same strength or slightly stronger on bits that are farther away? The graph has peaks and valleys at offsets that are multiples of 32. When we think about this more, an offset of 32 is when we are comparing every single bit to the matching bit in the next word. If we look at an offset of 31 where it first approaches that valley, almost every bit is being compared to an earlier bit in the next word. Every bit except the first bit in a word, which is being compared to the last bit in the same word. This leads us to an idea – perhaps the reason we get values that progress evenly back and forth from peaks to valleys is because the primary factor that is impacting the relationship between bits is which word the bit is in. If this was the case, bit 1 would have approximately the same correlation to bits 33 through bit 64, changing only at bit 65 when it was being compared to another word. We get a midpoint at 16 because at an offset of 16 exactly half of bits are being compared to the same word, and half to the next word, so it averages to be about in the middle. This explains the overall shape we are getting with our figure, though not where the correlation is from in the first place.

To test the theory that odd numbered words trend towards one value while even numbered trend towards the other, we run a histogram of noise values over sets of even and odd words. We divide our memory up into 64 bit blocks and perform a histogram on each individual bit from the set. The first 32 bits on our histogram then show us our odd numbered words while the 33rd through 64th bits show us the noise distribution of our even words. This helps us to see if the correlation we observed earlier is tied primarily to whether a bit is in an odd or even word.

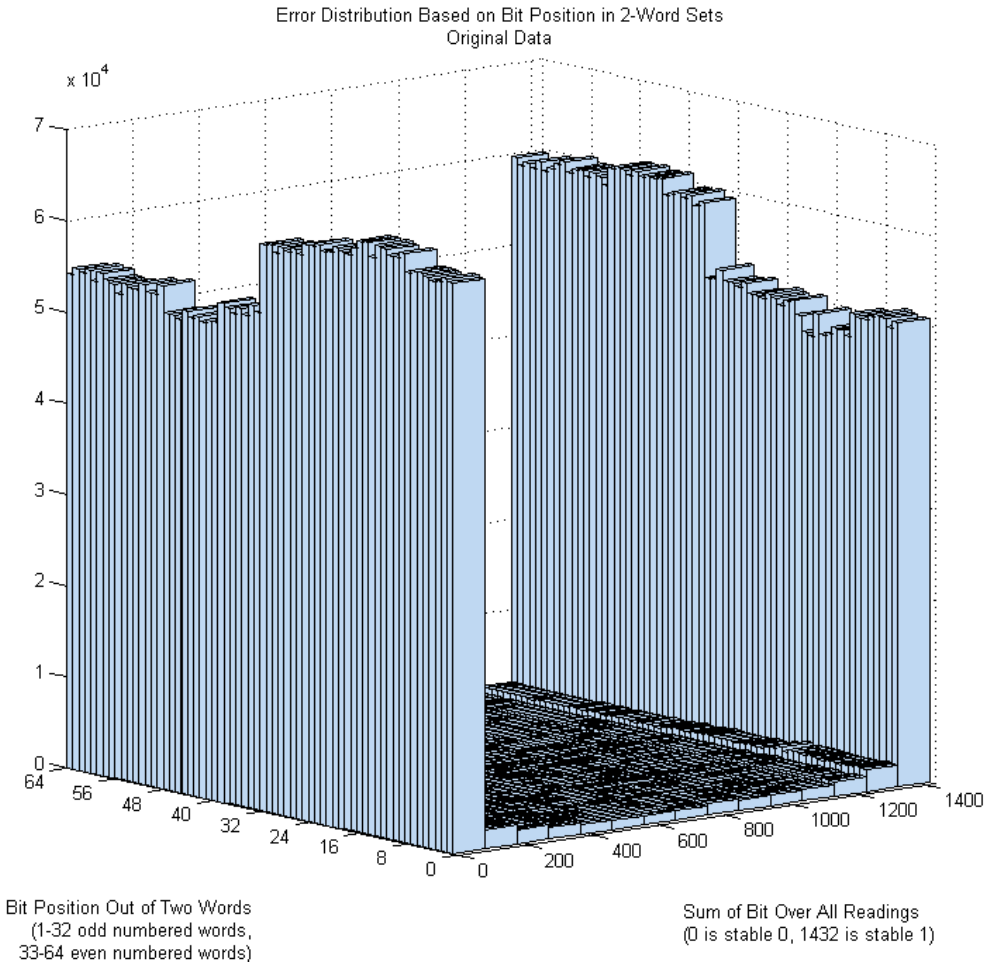

Figure 3-4. Estimated bit values for SRAM locations based on the relative position of a bit in odd or even words. Horizontal axis on left indicates which relative bit position is being examined. Bits 1 through 32 indicate the 1st through 32nd bits of all odd words, while bits 33 through 64 indicate the 1st through 32nd bits of all even words. Locations were each read 1432 times. Horizontal axis on right is number of readings returning 1 (0-1432). A result of 0 would mean the bit was a stable 0, while a result of 1432 would indicate the bit was a stable 1. Vertical axis is the number of occurrences of that particular noise level in a given bit position.

An analysis of this histogram shows what is happening. The odd numbered words represented by bits 1 through 32 are more likely to be a stable 0 than they are a stable 1. For example, if you examine the first bit in an odd numbered word there will be approximately 60,000 occurrences where the bit is a stable 0, but only approximately 50,000 occurrences where the bit is a stable 1. On the other hand, the first bit in an even numbered word (bit 33, in this figure) is a stable 1 on approximately 50,0000 occasions and a stable 0 on approximately 60,000 occasions. While it would be interesting to examine why this might be happening, for the purposes of our fuzzy extractor all we care about is keeping it from impacting our extracted bits. One potential solution to solve this discrepancy would simply be to extract alternating words. For instance, we could extract only odd words or only even

words. The problem with this idea is that it leaves us with an uneven distribution of zeros and ones, since either of those sets is not balanced. This is easy to picture just by looking at the previous histogram and imagining we were working with just one half or the other. Fortunately, there is a different solution that will balance out the words – we perform an exclusive or operation on alternating words.

$$\begin{aligned} New\ Word_0 &= Word_0 \oplus Word_1 \\ New\ Word_1 &= Word_2 \oplus Word_3 \\ &\dots \\ New\ Word_x &= Word_{2x} \oplus Word_{2x+1}. \end{aligned}$$

We will call this new data the transformed data, and the original unmodified data the original data. We repeat our 32 and 64 bit histograms on the transformed data to verify that we have avoided introducing a bit level bias towards zero or one, while eliminating the above-mentioned dependency between consecutive words.

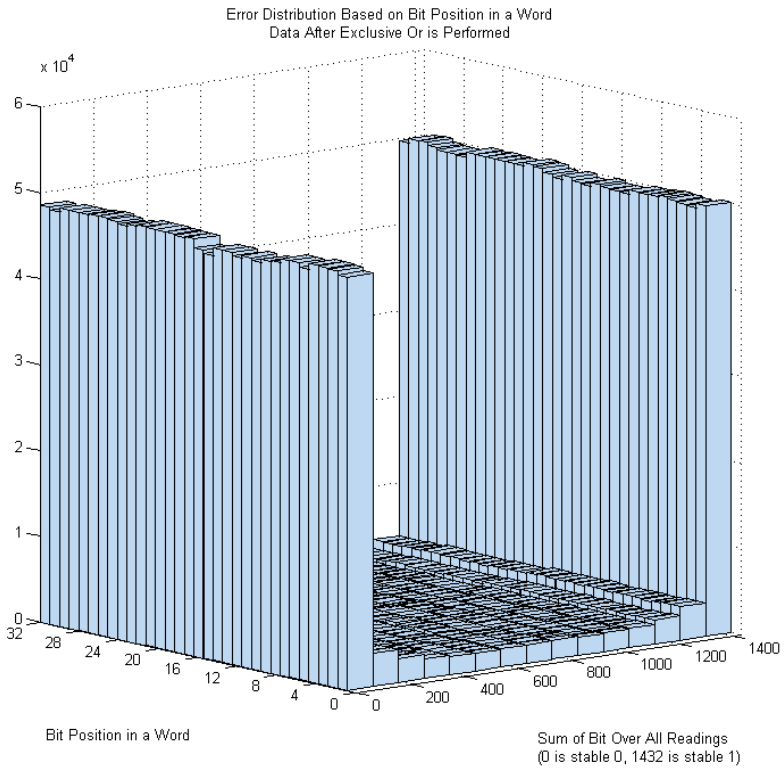

Figure 3-5. Using the transformed data, the estimated bit values for SRAM locations based on the relative position of the bit in a word. The horizontal axis on the left indicates which relative bit position is being examined, from the 1st bit from every word to the 32nd bit in every word. Locations were each read 1432 times. Horizontal axis on right is number of readings returning 1 (0-1432). A result of 0 would mean the bit was a stable 0, while a result of 1432 would indicate the bit was a stable 1. Vertical axis is the number of occurrences of that particular noise level.

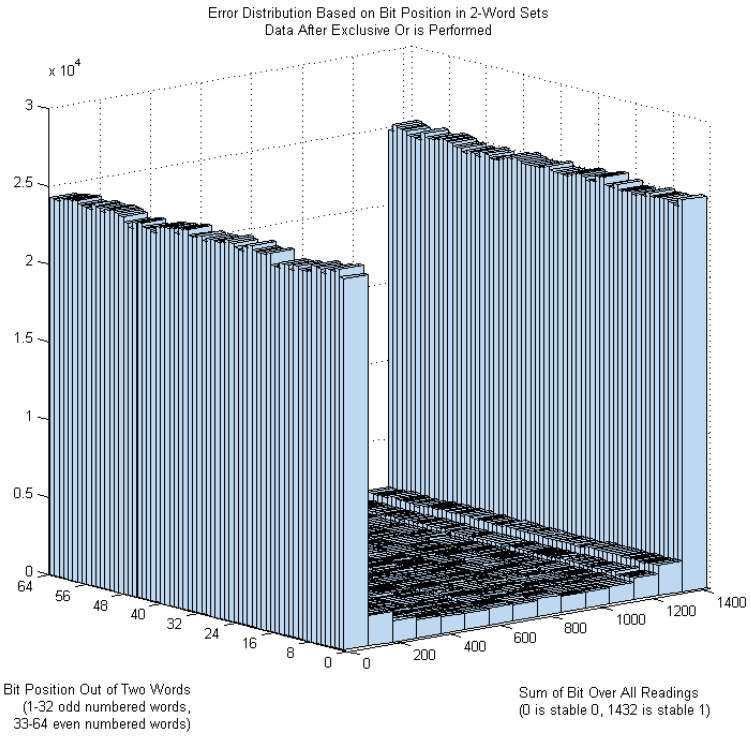

Figure 3-6. Using the transformed data, the estimated bit values for SRAM locations based on the relative position of a bit in odd or even words. Horizontal axis on left indicates which relative bit position is being examined. Bits 1 through 32 indicate the 1st through 32nd bits of all odd words, while bits 33 through 64 indicate the 1st through 32nd bits of all even words. Locations were each read 1432 times. Horizontal axis on right is number of readings returning 1 (0-1432). A result of 0 would mean the bit was a stable 0, while a result of 1432 would indicate the bit was a stable 1. Vertical axis is the number of occurrences of that particular noise level in a given bit position.

As we can see, the exclusive or operation does not introduce a bias in the level of zeros and ones, while correcting the balance issues we had with the original data. Now we need to make sure that this operation removed the correlation between bits. To simplify the comparison between the correlation we had in the original data and any correlation we might find in the transformed data, we plot the knowledge we are able to obtain about other bits from a given bit for both data sets. For example, one data set might indicate that if we know that a given bit is 1, the bit immediately following it is a 1 35% of the time, while if we know a given bit is 0 the bit immediately following it is a 1 30% of the time. If that was the case, the value located at 1 on the x axis (an offset of 1, to represent the immediately adjacent bit) would be .05 for that data set, or 5%. The more the y values of the graph deviate from 0, the more information an attacker will have about other bits if they know the value of one bit. The closer this value is to 0, the less knowledge we are able to obtain about other bits based on any other given bit. Since we are only interested in the areas in which the correlation is particularly strong in either direction, we will plot every 32nd offset.

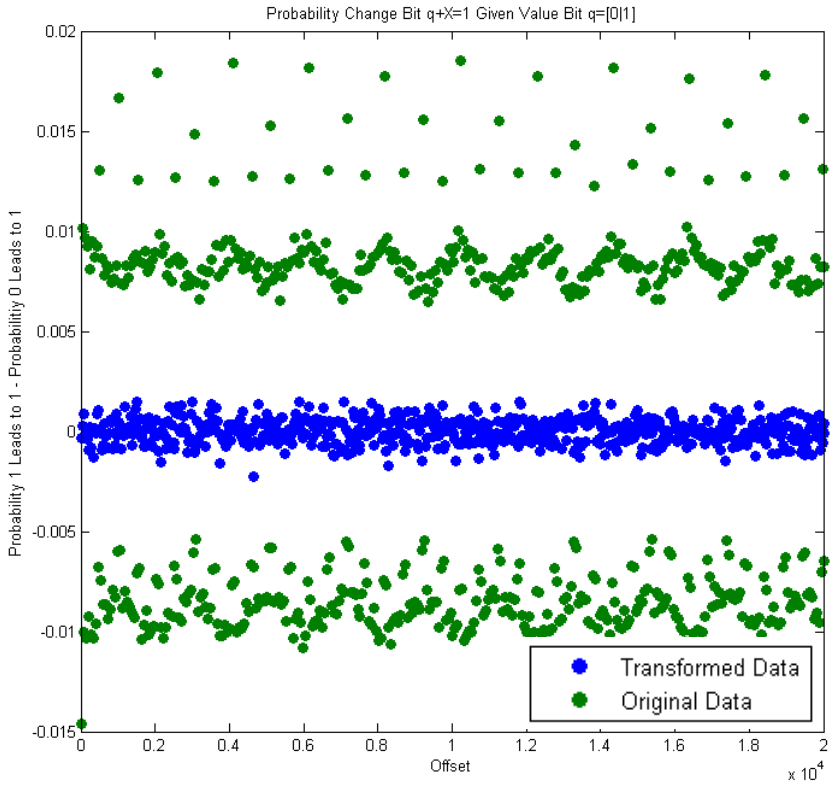

Figure 3-7. When given the value of a bit, the probability that a later bit will be a stable 1. The horizontal axis indicates the offset that is being considered – 1024 being the probability change of the bit 1024 bits away from your given bit. The vertical axis indicates the actual probability change – a value of .01 would indicate that if you are given the value of a bit as 1, the offset bit is 1% more likely to be a stable 1 than if you were told your given bit was a stable 0.

Compared to our previous correlation, the transformed data has negligible correlation between bits. We can now use the transformed data as the input for our fuzzy extractor. It is worth noting that this latest graph displays other interesting correlations that are no longer present in the transformed data. In addition to peaking at offsets which are multiples of 32, the correlation peaked especially high at offsets which are powers of 2. The correlation is even higher than the other peaks at offsets which are multiples of 512, then higher still at offsets of 1024 and offsets of 2048. Since the correlation is no longer present in our transformed data it does not impact our work, but it could prove interesting to explore in the future.

We conclude this section with a summary of the processing that we will perform on the SRAM data to obtain the input to our fuzzy extractor

- Discard all parity bits.

- Replace each consecutive pair of 32-bit words by the 32-bit exclusive or of this pair.
- This bitstream is then fed directly into the fuzzy extractor.

3.2 Calculating Optimal Parameters

In the previous section we talked about how we select the bits we will use as the input to our fuzzy extractor. This is only one consideration when it comes to designing our system. In order to deal with the inherent noise in the data readings, we need to select a BCH code that will properly achieve balance between conflicting needs. The fuzzy extractor system needs to meet the following conditions:

- Tolerate errors in measurement of the same device.
- Minimize an opponent's ability to achieve authentication of an unenrolled device.
- Utilize a BCH code of a length feasible for efficient computation and implementation.
- Provide sufficient entropy in the final hash value of the fuzzy extractor.

The BCH code that we select must provide enough error tolerance that our noisy readings are all identified as belonging to the same device. On the other hand, we must ensure that there is not excessive error tolerance such that it would be possible for an opponent to authenticate an unenrolled device. In this section we work on calculating optimal parameters by taking into account the needs of our system. We discuss the equations we will use to calculate valid length and dimension values for the BCH code we use in our fuzzy extractor, as well as the entropy and the bit error rate of the SRAM model.

3.2.1 Calculating Range of N and K Values

In the paper, "CDs Have Fingerprints Too" [8], Hammouri and Sunar give the following equation which characterizes an optimal, average-case $(M, m, \ell, t, \varepsilon)$-fuzzy extractor:

$$\ell = m + kf - nf - 2\log\left(\frac{1}{\varepsilon}\right) + 2\,.$$

The variables appearing here are:

ℓ = key length

m = min-entropy (entropy-per-bit times the length of our code)

k = number of message bits

n = length of codeword

f = number of bits required to store a field element (in our case, $f = 1$)

ε = a security parameter that measures deviation from a uniform distribution (we consider both $\varepsilon = 2^{-64}$ and $\varepsilon = 2^{-80}$).

This equation helps us to determine what key lengths ℓ we will be able to obtain from various combinations of n and k (once we estimate the min-entropy and the bit-error rate (BER)). The steps we will follow to obtain our ℓ values are as follows:

1. Estimate the bit-error rate of our source data.
2. Estimate the minimum entropy per bit of our source data.
3. Determine whether the Shannon entropy of the bit-error rate is less than our minimum entropy per bit.
4. Simplify equation to calculate ℓ such that the only unknowns are n, k, and ℓ.
5. Find BCH codes with a probability of failure less than one in a billion.
6. Calculate ℓ values based on those BCH codes.
7. If necessary, explore concatenating multiple codes together.

Let us first consider the bit-error rate, which is in our case the noise level between a perfect noiseless board and a normal noisy reading. Of course, we cannot actually get a reading from the board that has no noise, so we instead simulate a noiseless reading by averaging many noisy readings together. We would get a reading closest to ideal if we simply used every reading that we had of the board. However, if we do that we will be unable to properly compare a given noisy reading to our ideal reading. The comparison would be biased, since that very same noisy reading went into creating our noiseless reading. Instead we divide our sample space of 1432 readings into two randomly selected sets and combine 715 readings together for our ideal reading. Note that each set must have an odd size in order to ensure that no bits are computed as being 0 and 1 equally often. We then compare our ideal reading to the other 717 readings. The actual comparison is easiest if we simply use a bitwise exclusive or - we can then compute the Hamming weight to find out how many bits did not match between the two readings. Once we average the number of noisy bits we obtained among all our comparisons, we find the fraction of bits that are noisy by dividing by the total number of bits per reading, 8388608. We get a value of .0361 noisy bits for our regular data and .0693 noisy bits for the data on which we performed the word-wise exclusive or.

Now that we have our bit error rate, we move on to estimate the minimum entropy per bit of our source data. In "The Context Tree Weighting Method: Basic Properties" [9] a method was proposed to estimate entropy called Context Tree Weighting (CTW). Context Tree Weighting is a sequential universal data compression procedure that achieves the Rissanen lower bound [9]. To remove the

entropy of the noise from the equation, we take our data and discard all values that were not identical over all readings from a single board. Excluding the noise in this fashion allows us to calculate the entropy of the actual data bits. We then run our data through the CTW program and it produces a value of 7.44 as the entropy per byte. So, we get a per-bit entropy of δ = 0.93.

Before determining values of n and k, we must also ensure that the Shannon entropy, H_2, of the bit-error rate is less than the min-entropy per bit, $\frac{\delta}{f} = \frac{0.93}{1} = 0.93$. We calculate that

$$H_2(p) = -p * log_2(p) - (1-p) * \log_2(1-p)$$
$$H_2(BER) = H_2(0.0693) = 0.3633 < 0.93.$$

Since the entropy of the noise is significantly less than the entropy per bit, we are able to proceed. If this condition was not met, we would not be able to use the SRAM readings as a source for the input of a fuzzy extractor and meet all of our design goals.

Now that we have calculated the min-entropy per bit and the bit-error rate of our data, we can derive an equation that will produce the maximum key-length ℓ from a given n and k, and a chosen ε. The difference between setting $\varepsilon = 2^{-64}$ and setting $\varepsilon = 2^{-80}$ is that the smaller value will provide more security at the cost of having less useable key bits. For $\varepsilon = 2^{-64}$ we get the following equation:

$$\ell = 0.93n + k - n - 2\log\left(\frac{1}{2^{-64}}\right) + 2$$
$$\ell = 0.93n + k - n - 126$$
$$\ell = k - 0.07n - 126.$$

This gives us how many usable key bits we obtain using $\varepsilon = 2^{-64}$ for a code of length n and dimension k. For $n \leq 128$ this does not look promising. However, for sufficiently large n we may even have enough slack to use $\varepsilon = 2^{-80}$ for enhanced security; the corresponding equation is:

$$\ell = k - 0.07n - 158.$$

So we see that in general that key length decreases by 32 bits when we pass from $\varepsilon = 2^{-64}$ to $\varepsilon = 2^{-80}$. For example if we use $n = 2^{10} - 1 = 1023$ we might find a code with $k = 808$ and get $l = 610$ for $\varepsilon = 2^{-64}$ and $l = 578$ for $\varepsilon = 2^{-80}$.

3.2.2 Calculating Number of Errors to Correct

We must now construct a second bound for our BCH code that will determine the minimum error-correcting capabilities needed to satisfy a pre-determined probability of failure, P_{Fail}. The parameter P_{Fail} is the probability that our BCH code will fail to successfully reconstruct the transmitted codeword from the received bitstring. For our purposes, we require $P_{Fail} \leq 10^{-9}$, or at most a one in a billion chance of failure.

Assuming that bit errors within a given transmitted codeword are independent events, the probability of success is the probability that we get a number of errors less than or equal to the maximum number of correctable errors. In other words, the sum of the probability that we have 0 errors, 1 error, 2 errors, and so on up to t errors, and subtract the overall probability of these good scenarios from 1.0 to find:

$$P_{Fail} = 1 - \sum_{i=0}^{t} \binom{n}{i} p^i (1-p)^{n-i}$$

The variables appearing here are:

t = number of errors the code can correct

n = length of the code

p = bit-error probability

Using this equation, we can determine the minimum value of t that satisfies our value of P_{Fail}, given a value of n. Since we know that n must be one less than a power of 2, ($n = 63, 127, 255$, etc.), we choose multiple valid values of n and calculate the maximum value of k which will still give us at least t correctable errors. This gives us the parameters we need for our BCH code. We then calculate the value of ℓ using these BCH codes to determine the number of key bits we can extract using each scheme.

3.2.3 Optimal Parameters for Single Codeword

We run the n and k combinations that our P_{Fail} calculation indicated provided the necessary error tolerance through our formula to obtain ℓ, or the maximum number of key bits we can obtain from the code. Negative values can be treated as producing a maximum key length of 0, since they do not produce any legitimate key bits.

Optimal BCH Codes for 1/billion failure Rate			
Key Length ℓ		n Value	BCH Code Used $[n, k, t]$
$\varepsilon = 2^{-64}$	$\varepsilon = 2^{-80}$		
0	0	63	No Valid Codes
0	0	127	No Valid Codes
0	0	255	[255, 29, 47]
0	0	511	[511, 76, 85]
0	0	1023	[1023, 153, 125]
138	106	2047	[2047, 408, 218]

Table 3-1. The n Value column indicates the n value we were considering on that row. The BCH Code Used column indicates the dimensions of the BCH code that fit our requirements. The ℓ columns indicate the maximum number of key bits we can obtain with $\varepsilon = 2^{-64}$ and $\varepsilon = 2^{-80}$, respectively.

Disappointingly, we can't produce any key bits until we use a codeword of size 2047, and even then we do not produce enough for a 256 bit key.

3.2.4 Concatenation

Given the size of the BCH codes that are required to generate sufficient size keys from our data, and the inefficiencies that come with larger size BCH codes [10], it is worthwhile to explore concatenating multiple codes together. By concatenating multiple codes it is possible to increase the value of ℓ while still using smaller, more efficient BCH codes. It is necessary to update our P_{Fail} calculation to take into account that we are using multiple codes. The probability of success changes from being the probability that a single received codeword will have less than the maximum correctable number of errors, to the probability that all of the received bitstrings will individually have less than the maximum number of errors. In other words, it is the probability that the first bitstring will be successfully recovered multiplied by the probability that the second bitstring will be successfully recovered, continued in that fashion for all received bitstrings. The new equation for the probability of failure changes as we would expect:

$$P_{Fail} = 1 - \left(\sum_{i=0}^{t} \binom{n}{i} p^i (1-p)^{n-i} \right)^c$$

The variables appearing here are:

t = number of errors the codeword can correct

n = length of the codeword

p = bit-error probability

c = number of times the code is concatenated.

It would be possible to concatenate different code sizes together (for instance, concatenate a 511-bit codeword with a 255-bit codeword). However, it would be complex to compute minimum t values for such concatenation, since increasing the correctable errors for one codeword might allow you to decrease them for another. It would also make decoding more complicated, as our system would need to be able to decode different codes. Instead, we want to ideally find a single codeword size such that we can concatenate multiple codewords of the same dimensions together to produce our key.

Since we will not be mixing different codes together, the only dimension we add here is the number of concatenations we are using for each code. There are two primary ways this can impact the functionality of our system – it changes the overall amount of bits we need, and it affects the time decoding might take. We generate a list consisting of valid codes and a value indicating how many times we would repeat that code. For this list we exclude values that do not produce an ℓ of at least 128. To constrict the size of our table further, we restrict the both the maximum number of extracted bits and the maximum size of the BCH code. This results in removing from our table all values which do not fit the following criteria:

$$\ell \geq 128$$
$$nc \leq 2^{13}$$
$$n \leq 2^{11} - 1.$$

The variables appearing here are:

ℓ = maximum obtainable key length

n = length of the codeword

c = number of times the code is concatenated.

An addition to our script of some timing calculations will allow us to incorporate decoding time into our BCH code selection. Our timing script decodes 100 strings with a number of errors ranging from 0% to 99% of the maximum correctable errors. The script performs this timing calculation for each BCH code that fits our P_{Fail} equation, and uses the average decode time of each code as the baseline time for that particular code. If we have concatenated c copies of the same code and we estimate t seconds for decoding a message in one copy, we model overall decoding time t' for the concatenated codes as

$t' = ct$. An alternative timing model is based on the assumption that the generator polynomial must be calculated at the beginning of each authentication; in this scenario where the same code is used throughout, subsequent decodings require less time and our model, $t' = ct$, would be invalid. For our particular implementation we plan on preselecting a BCH code, so we will not need to regenerate the generator polynomial whenever we attempt to authenticate a device and our model remains valid. In addition, we are performing these timings in the MATLAB implementation of our fuzzy extractor – different decoding algorithms may behave differently in other implementations.

Optimal BCH Codes for 10^{-9} Failure Rate					
Key Length ℓ		Number of Bits Harvested	Number of Concatenations	BCH Code Used $[n, k, t]$	Decode Time (seconds)
$\varepsilon = 2^{-64}$	$\varepsilon = 2^{-80}$				
138	106	2047	1	[2047,408,218]	0.053089
155	123	3577	7	[511, 76, 85]	0.036267
159	127	4092	4	[1023,143,126]	0.058397
180	148	5115	5	[1023,133,127]	0.07384
195	163	4088	8	[511, 76, 85]	0.041448
236	204	4599	9	[511, 76, 85]	0.046629
242	210	6138	6	[1023,133,127]	0.088608
276	244	5110	10	[511, 76, 85]	0.051811
303	271	7161	7	[1023,133,127]	0.103376
316	284	5621	11	[511, 76, 85]	0.056992
356	324	6132	12	[511, 76, 85]	0.062173
365	333	8184	8	[1023,133,127]	0.118144
396	364	6643	13	[511, 76, 85]	0.067354
403	371	4094	2	[2047,408,218]	0.106178
437	405	7154	14	[511, 76, 85]	0.072535
477	445	7665	15	[511, 76, 85]	0.077716
517	485	8176	16	[511, 76, 85]	0.082897
668	636	6141	3	[2047,408,218]	0.159267
932	900	8188	4	[2047,408,218]	0.212356

Table 3-2. Table illustrating what ℓ values we obtain from particular BCH codes concatenated a given number of times. Highlighted are the fastest decoding times that produce $\ell \geq 128$ and $\ell \geq 256$ for each ε value we consider. Highlighted in blue are the codes chosen for $\varepsilon = 2^{-64}$. Highlighted in green are the codes chosen for $\varepsilon = 2^{-80}$.

As we expected, concatenation greatly increases our supply of codes; we now have several options to generate sufficient length keys. The SRAM has over 8 million bits that we already determined were adequate to use in our key generation, so for our purposes decoding time is a more important metric than the overall bits a particular code is going to require. Examining the table we notice an exciting correlation. The fastest codes at $\varepsilon = 2^{-64}$ are 7 concatenations of code [511, 76, 85] for $\ell \geq 128$, and 10 concatenations of code [511, 76, 85] for $\ell \geq 256$. Then the fastest codes at $\varepsilon = 2^{-80}$ are 8 and 11 concatenations of that same code. The BCH code with 511 total bits, 76 data bits, and 85 correctable errors is a clear winner in this implementation. Whether we aim for $\ell \geq 128$ or for $\ell \geq 256$,, this particular code is faster than the alternatives.

4 MATLAB Framework

One of the requirements for this project was to develop an implementation of an authentication system based on fuzzy extractors in MATLAB as a proof of concept. This particular implementation serves as a framework that makes use of the object-oriented programming paradigm to provide modularity and robustness. The state and behavior of each software component is encapsulated in a MATLAB class and most of the coupling between classes is accomplished through abstract classes and interfaces. The behavior in this system as a whole is also organized in a small hierarchy of class composition and aggregation. The end goal of this architecture is that users of this framework should be able to rapidly build a fuzzy extractor authentication system with only a limited knowledge of the underlying details of the system. In addition to this, end-users should also have a sufficient amount of freedom to extend the functionality of the system, without having to modify existing code. However, end-users of this system should first have at least a basic understanding of the architecture of this framework. In this section, we will examine the overall design and process flow of this software system. We will also build a simple prototype that will test the enrollment and authentication procedures for two sets of SRAM readings from different devices using this framework.

4.1 Architecture

Each class in this authentication system framework is compartmentalized into higher-level groupings, known as packages. A package structure tends to be useful in large software applications, where the natural grouping that occurs at the class-level is too specific to perform high-level analysis on the overall system. While this authentication system is by no means a large software application, the package structure still provides additional organization, which mitigates the task of understanding and eventually extending the software. Higher-level analysis on the system is also useful to a degree, as it provides insight into how it might evolve or how it can be improved in future iterations.

The package structure of this system consists of four packages, which is illustrated in Figure 4-1. The “Core” package consists of the classes that make up the high-level enrollment and authentication protocols. This package is dependent on three other packages, which are the “Encoder”, “Hash”, and “Device” packages. The “Encoder” and “Hash” packages contain the classes relevant to the encoding and hashing functionality of the fuzzy extractor, respectively. The “Device” package contains the classes that encapsulate the behavior for reading and extracting data from varying sources.

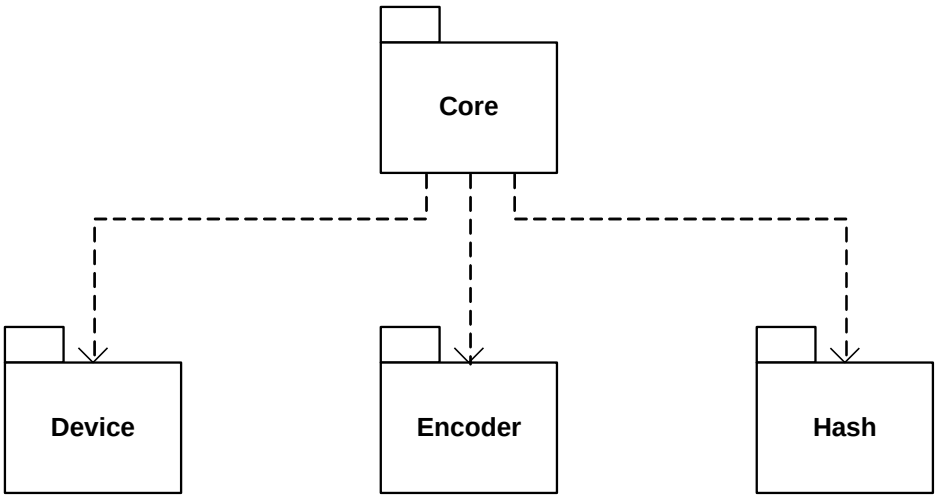

Figure 4-1. UML Package Diagram of Authentication System

The aspect of this system that best represents the high-level behavior of a fuzzy extractor authentication system is the relationship between the Core package and the Device package. The Core package consists of two classes, one of which is the `DeviceAuthenticator` class. As the name might suggest, the `DeviceAuthenticator` class handles the high-level behavior of authentication in this system through the `enroll` and `authenticate` methods. These methods are used to enroll and authenticate objects that extend the `ADeviceReader` abstract class, which is a member of the Device package. The `enroll` and `authenticate` methods create an association between the Core and Device packages, as shown in Figure 4-2, where `DeviceAuthenticator` class must interface directly with the `ADeviceReader`. Since the `DeviceAuthenticator` class depends on the `ADeviceReader` class, the Core package is dependent on the Device package. In other words, changes that occur to the contents of the Device package will directly affect the Core package. However, changes in the Core package will not affect the contents of the Device package. Regardless of dependencies, this relationship models the interaction between the fuzzy extractor authentication system and the devices interacting with the system.

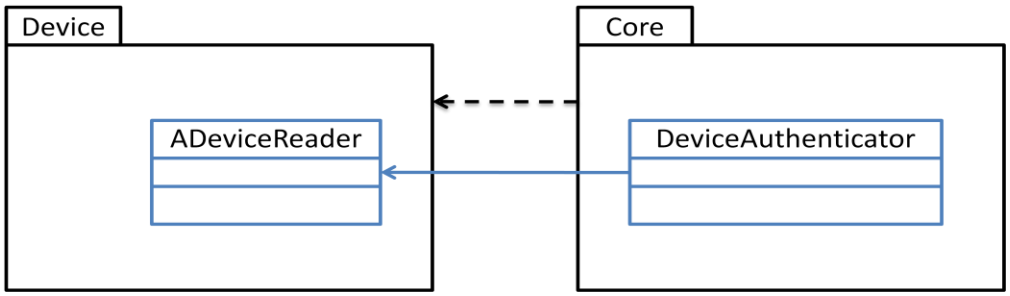

Figure 4-2. Relation Between Core and Device Package

This relationship is further illustrated in the following pseudocode:

```
% Setup the DeviceAuthenticator
devAuth = DeviceAuthenticator(...);

% Setup a SimpleDeviceReader, where SimpleDeviceReader < ADeviceReader
devReader = SimpleDeviceReader(...);

% Enroll the device
devAuth.enroll(devReader);

% Authenticate the device
isEnrolled = devAuth.authenticate(devReader);
```

In this pseudocode, a single instance of a `DeviceAuthenticator` and a `SimpleDeviceReader`, which extends `ADeviceReader`, is instantiated. Next, the `SimpleDeviceReader` is enrolled into the system through the `enroll` method. Finally, a Boolean value indicating whether or not the `SimpleDeviceReader` is enrolled is returned by calling the `authenticate` method. The details regarding setting up both the `ADeviceReader` and the `DeviceAuthenticator` will be covered in the sections to follow.

Now that we have a basic understanding of the overall structure of this framework, let us take a more in-depth look into the individual system packages. In the remainder of this section we will first examine the Device package in order to understand how device entities are represented in this system. Next we will look at the Core package to see how these devices interact and are incorporated into the fuzzy extractor authentication system. Finally, we will inspect the Encoder and Hash packages to understand how the lower-level encoding and hashing functionality is modeled and implemented.

4.1.1 The Device Package

One of the more difficult tasks in the design of this system is determining how to model the devices that interact with the fuzzy extractor system. Since the scope of this project involves authenticating devices based on the contents of SRAM, the system should be able to interact with devices regardless how the SRAM data is accessed and provided. In addition to this, the specific data that the system needs from the device might be different or vary from the entirety of the data that can be accessed from the device. These two design considerations are implemented in this system in the Device package through the `ADevice` and `ADeviceReader` abstract classes. These two abstract classes provide a common interface for varying classes representing physical devices to interact with the system.

Let us first look at the `ADevice` abstract class, which is shown in Figure 4-3. This class provides an interface for varying types of devices to provide their SRAM contents or other unique data in a common format. Each class extending the `ADevice` abstract class must implement the `getData` abstract method. Although MATLAB is inherently loosely typed, the `getData` method is intended to return a vector of doubles ranging in value from 0 to 1. The format of this data-type is referred to as `Bits` as it is intended to represent individual bits of data. The `ADevice` abstract class also includes a `getUID` method, which is used to uniquely identify a device through non-cryptographic means. From the `ADevice` abstract class, we can derive concrete classes that represent both physical and virtual devices that will be able to interface with the fuzzy extractor authentication system. In this system, there are currently two classes that extend this abstract class, which are `VirtualDevice` and `VirtexBoardDevice`.

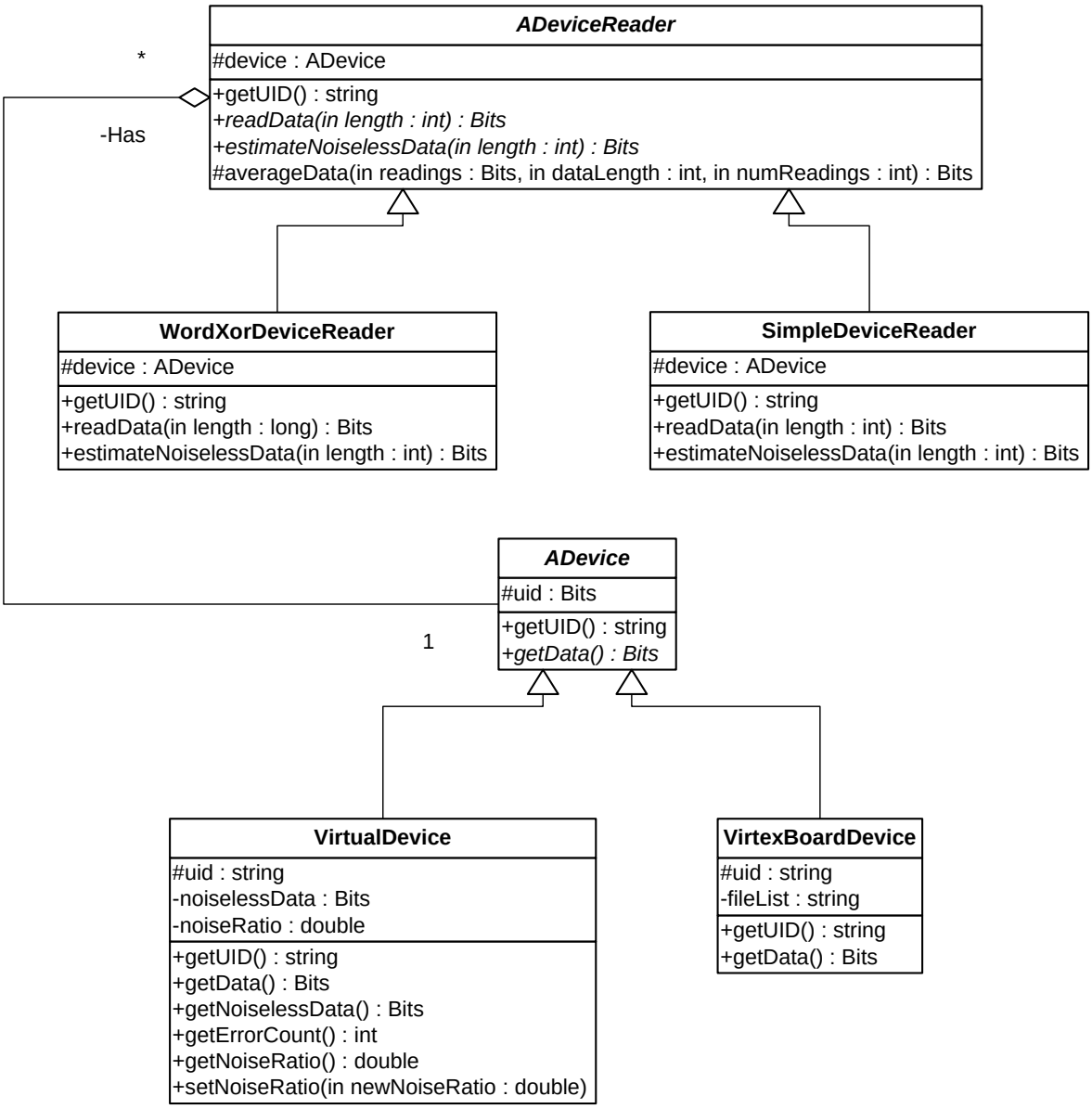

Figure 4-3. UML Class Diagram of the Device Package

The `VirtualDevice` class models a device by providing data with a set amount of noise added at runtime. It is intended to act as a "mock object", that can be used to verify correct functionality of other components in the system that depend on this class, without having to provide actual SRAM readings. The `VirtualDevice` class contains an instance of `Bits`, `noiselessData`, and a double value representing its noise level, `noiseRatio`, where $0 \leq$ `noiseRatio` ≤ 1. Upon instantiation of a `VirtualDevice` object, `noiselessData` is randomly populated with 1's and 0's and the value of `noiseRatio` is provided by the constructor. The `getData` method is implemented by returning the contents of `noiselessData`, but with a number of bits flipped determined by the value of `noiseRatio`. The number of bits flipped, n, is determined by the following simple equation:

$$n = \lceil \text{noiseRatio} * \ |\text{noiselessData}| \rceil$$

The bit position of each flipped bit is randomly selected at runtime for each call of `getData`. The behavior of this implementation is intended to simulate random noise as would be expected from readings of SRAM data based on the results of the data analysis in Section 3.1.

While the `VirtualDevice` class is intended to test only correct system behavior and functionality, the `VirtexBoardDevice` class is additionally intended to show system behavior similar to that which it would show when interfacing with actual physical devices. Although the `VirtexBoardDevice` does not interface directly with a physical device, it does provide data from a collection of over 1000 readings of SRAM contents. Similar to how the implementation in the `VirtualDevice` class adds noise, this class implements the `getData` method by returning a randomly selecting an SRAM reading from a list of readings at runtime. Using the `VirtexBoardDevice` class allows for accurate results to be collected when using differing parameters for a particular system configuration, as it mimics the data that would be returned from an actual physical device.

Now that we have a common interface by which to provide the fuzzy extractor authentication system with unique data, we need a means by which to select a subset of this data to be used for enrollment and authentication. Otherwise, the authentication system would have to process millions of bits for each device. This behavior is provided by another abstract class, `ADeviceReader`. Each class extending `ADeviceReader` has an instance of one object that extends `ADevice` as a protected class variable. This allows for classes extending `ADeviceReader` to have access to a single source of data, from which to select data that is to be provided to the authentication system. This subset of data from

the objects extending `ADevice` is returned by the `ADeviceReader` through the `readData` abstract method. Classes extending `ADeviceReader` must also implement the `estimateNoiselessData` abstract method, which is intended to be the means by which it returns the same subset of data as returned by `readData`, but with noise removed.

In the current state of this system, there are two classes available that extend the `ADeviceReader` class, which are `SimpleDeviceReader` and `WordXorDeviceReader`. The `SimpleDeviceReader` is relatively simple, as it implements the `readData` abstract method by returning the first n bits from the device's `Bits` vector with a given bit position offset. The `WordXorDeviceReader` class is slightly more complicated, as it implements the word-wise XOR method as described in Section 3.1.2. Both of these classes can be used with any of the two classes currently extending `ADevice`, as should be expected given the inherent loose coupling between `ADeviceReader` and `ADevice`.

4.1.2 The Core Package

Now that we have a basic understanding Device package, we can examine the interface between the authentication system and the devices in more detail by reviewing the contents of the Core package. As discussed in the beginning of Section 4.1, the Core package is responsible for enrolling and authenticating these devices through the coupling between the `ADeviceReader` abstract class and the `DeviceAuthenticator` class. This package consists of two classes, shown in Figure 4-4, which are the `DeviceAuthenticator` class and the `FuzzyExtractor` class. Similar to how `ADeviceReader` and `ADevice` are connected through aggregation, these two classes are related through composition, where the `DeviceAuthenticator` contains a single instance of a `FuzzyExtractor` object.

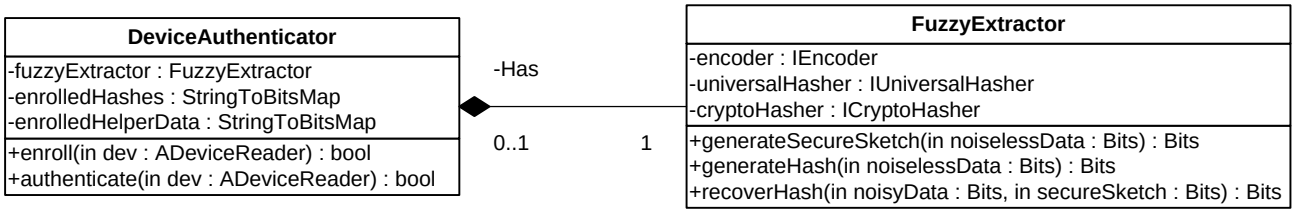

Figure 4-4. UML Class Diagram of the Core Package

In addition to the `FuzzyExtractor` object, the `DeviceAuthenticator` class contains a mapping from device UID's to the hashed value of the data extracted from the respective device, which is stored in the `enrolledHashes` class variable. It also contains a mapping of device UID's to the respective helper data of the device, which is stored in the `enrolledHelperData` class variable. Both of these mappings are implemented through the `containers.Map` object provided by the standard library in MATLAB. Keys and values are added to these two variables on successful device enrollment through the methods provided by the `FuzzyExtractor` class. During the authentication procedure, the data in these two variables is used to reconstruct and verify the hash of the device being authenticated. As discussed previously, the enrollment and authentication procedures are carried out by the `enroll` and `authenticate` methods, respectively.

While the `DeviceAuthenticator` class handles the high-level enrollment and authentication procedures, the `FuzzyExtractor` class is responsible for the functionality of the fuzzy extractor in the authentication system. This class contains three methods, which are `generateSecureSketch`, `generateHash`, and `recoverHash`. The `generateSecureSketch` method is used to generate the helper data for a device during the enrollment phase, while the `generateHash` method is used to construct the hash for the device in both the enrollment and authentication phases. The `recoverHash` method is used only during the authentication phase, which verifies that the hash constructed for an authenticating device matches the hash constructed during the enrollment phase.

Similar to how the `DeviceAuthenticator` class is responsible for only the high-level authentication procedures, the `FuzzyExtractor` class is only responsible for the higher-level functionality of a fuzzy extractor. The details of the encoding and hashing procedures are delegated to classes implementing the `IEncoder` and hashing interfaces respectively, as shown in Figure 4-5. A single instance of each of these classes is contained in the `DeviceAuthenticator` class through composition. This essentially allows for different encoding and hashing procedures and schemes to be "swapped" in and out of a `FuzzyExtractor` object without having to modify the `FuzzyExtractor` class. However, before we discuss how to these `FuzzyExtractor` objects are constructed, we must first understand the encoding and hashing components, which are covered in the following two sections.

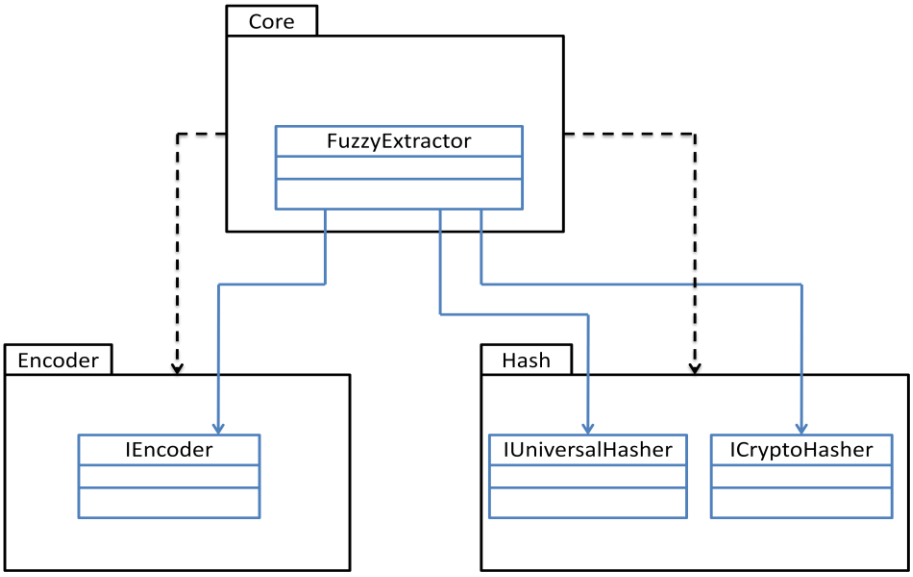

Figure 4-5. Dependency of Core Package on Encoder and Hash Packages

4.1.3 The Encoder Package

The Encoder package is responsible for implementing the encoding and decoding procedures needed to perform the lower-level operations in the `FuzzyExtractor` class, such as generating codeword and recovering device data from a noisy source. As shown in the UML class diagram in Figure 4-6, this package consists of only a single interface, the `IEncoder` interface, which contains three methods: `generateCodeword`, `decode`, and `getCodeLength`. The `generateCodeword` method is used to generate a randomly selected codeword from a linear BCH code. In the `FuzzyExtractor` class, this method is used to generate a codeword to be used in the code-offset construction. The `decode` method is used to decode codewords from the same set of codewords provided by the `generateCodeword` method. Finally, the `getCodeLength` method returns the length of the codewords that are returned from the `generateCodeword` method. This allows for the `FuzzyExtractor` class to know how many bits are needed from the device data to be used in the code-offset construction.

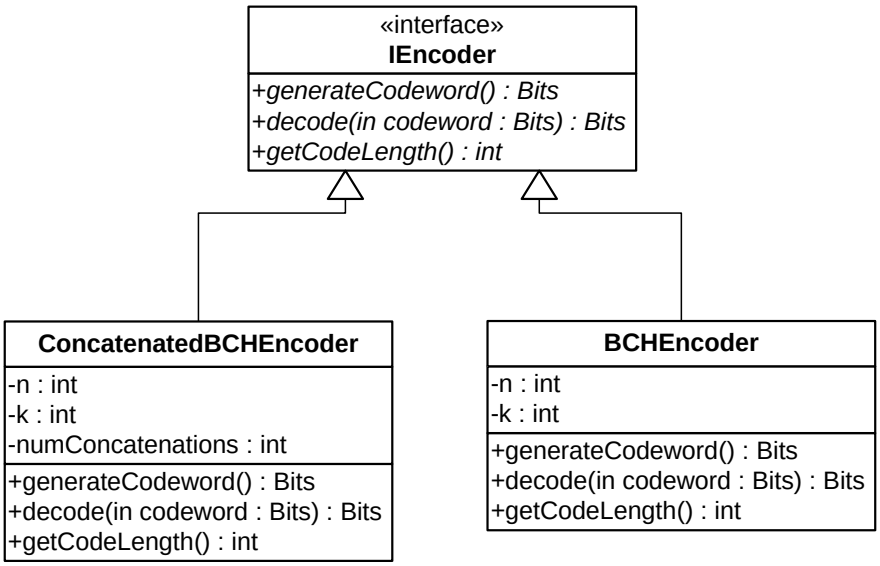

Figure 4-6. UML Class Diagram of the Encoder Package

Now that we know how the `IEncoder` interface is defined, let us look into the two classes that implement this interface, `BCHEncoder` and `ConcatenatedBCHEncoder`. The `BCHEncoder` class uses the BCH encoding scheme discussed in Section 3.2.4 through the functionality provided by MATLAB in the Communications Toolbox. This class contains two class variables, `n` and `k`, which correspond to the length of the codeword and the number of message bits in the codeword, respectively. The codewords that are generated and decoded are expected to be BCH codewords having these parameters. The `BCHEncoder` class implements the `getCodeLength` method by simply returning the value of `n`.

The `ConcatenatedBCHEncoder` class is essentially identical to the `BCHEncoder` class, except that it constructs and decodes BCH codewords that are concatenated together. See Section 3.2.4 for the mathematics of this construction. In addition to `n` and `k`, this class has another class variable, `numConcatenations`, which contains the value of the number of concatenations to use when generating and decoding a codeword. For example, if the value of `n` is 15 and the value of `numConcatenations` is 4, the length of a generated codeword is 60. The `decode` method will also expect codewords of length 60 and will split the concatenations automatically when decoding the individual BCH codewords.

4.1.4 The Hash Package

The Hash package is responsible for implementing the hashing procedures used by the `FuzzyExtractor` class. As discussed in Section 2.3, the fuzzy extractor first uses a universal hash function to compress the input data into an approximately uniformly distributed output of a fixed length. It then uses a cryptographic hash function to perform a one-way hash on this output for added security. These two types of hash functions are modeled through the `IUniversalHasher` and `ICryptoHasher` interfaces, which are members of this Hash package as shown in Figure 2-1. Both of the interfaces have a single abstract method that take in a vector of doubles ranging in value from 0 to 1 as a parameter. The `IUniversalHasher` takes in an additional parameter, `seed`, which determines the seed of the pseudorandom number generator used to generate the random binary vector in the hash function. The `FuzzyExtractor` class contains an instance of a class that implements `IUniversalHasher` and an instance of a class that implements `ICryptoHasher`, which allows it to implement the hashing task of the fuzzy extractor.

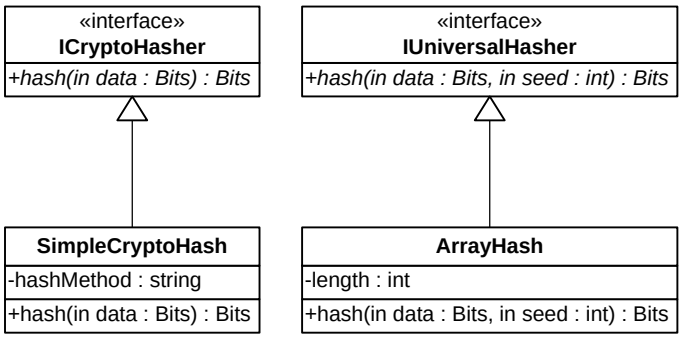

Figure 4-7. UML Class Diagram of the Hash Package

In the Hash package, there is only one class that implements `IUniversalHasher` and one class that implements `ICryptoHasher`, which are the `ArrayHash` and `SimpleCryptoHash` classes, respectively. The `ArrayHash` class implements the family of universal hash functions as discussed in Section 2.3.4. This class contains a single class variable, which is the length of the output values of this hash function. The `SimpleCryptoHash` uses a third-party implementation that makes use of the cryptographic hashing functionality provided by Java. This class contains a single class variable, which contains the string representation of the desired hash function (valid values of which are "MD2", "MD5", "SHA-1", "SHA-256", "SHA-384", and "SHA-512").

4.2 Building a Simple Authentication System

Now that we have an understanding of the architecture of this system, let us walk through the construction of a basic fuzzy extractor authentication system using this framework in MATLAB. In this example, we will use a `VirtexBoardDevice` object to make use of real SRAM readings for our device data and a `WordXorDeviceReader` object to extract data from this device. For the fuzzy extractor component of this system, we will use a `ConcatenatedBCHEncoder` object for encoding, while using `ArrayHash` and `SimpleCryptoHash` objects for the universal and cryptographic hashing respectively. The parameters for this system are listed below in the following table:

System Parameter	Value
BCH Code $[n,k,t]$	[511,76,85]
Number of Copies Concatenated	8
Key-Length	128
Cryptographic Hashing Function	SHA-512

Let us begin by first setting up the device component of this system. We start by instantiating an instance of a `VirtexBoardDevice` object. This is performed through the following MATLAB code:

```
fileList = dir('C:\mat_data\*.mat');
dev = VirtexBoardDevice('device_1',fileList);
```

The first line of this MATLAB code creates a file list containing all ".mat" files in the "C:\mat_data" directory. These files are expected to contain the SRAM reading data in the form of binary vectors, as described in Section 3.1.1. The second line calls the constructor for the `VirtexBoardDevice` class. The constructor for this class requires two parameters, which are the device UID and the list of files returned by the `dir` function from the first line of code.

Before moving on, we note that the device UID has two purposes. The first is for developers to be able to easily distinguish between device objects in a testing environment or to be used in higher-level protocols. The second is that it allows the DeviceAuthenticator to quickly determine whether or not a given device is enrolled in the system before authenticating. This allows the developer or tester to

simulate adversarial cloning of a device by incorporating into their model a UID identical to that of a device that is already enrolled.

Now that our `VirtexBoardDevice` object is initialized, we can instantiate our `WordXorDeviceReader` object, which will allow the `VirtexBoardDevice` object to indirectly interface with the authentication system. This is performed through the following MATLAB code:

```
devReader = WordXorDeviceReader(0,dev,9);
```

This line of MATLAB code calls the constructor for the `WordXorDeviceReader` class, which has three arguments. The first parameter is the bit position offset used to begin extracting identification bits from the device data. In this case, we are using an offset of 0, which tells the device reader to begin extracting bits at the first bit position. The second parameter is an object extending the `ADevice` abstract class. This object is the device that data is read from which, in this case, is the `VirtexBoardDevice` object that we created earlier. The last parameter is the number of readings taken from the device to be used in the averaging function. This averaging function is used to estimate and produce a noiseless set of device data to the authentication system.

At this point, we have completed the process of setting up a single device to be used with the fuzzy extractor authentication system. Let us now complete the system construction process by focusing on the authentication system component. We begin by instantiating our encoding and hashing objects through the following MATLAB code:

```
encoder = ConcatenatedBCHEncoder(511,76,8);
universalHasher = ArrayHash(128);
cryptoHasher = SimpleCryptoHash('SHA-512');
```

The first line of this MATLAB code calls the constructor for the `ConcatenatedBCHEncoder` object, which has three parameters. The first two are the length n and dimension k of the BCH code. The last parameter is the number of codewords that are concatenated together. The second line of code calls the constructor for the `ArrayHash` class, which has one parameter. This parameter is simply the size of the output value produced by the hash function. The last line of code calls the constructor for the `SimpleCryptoHash` class, which simply takes in the string value of the hash function to be used. In this case, we are using the SHA-512 algorithm, which is represented by the string value, “SHA-512”.

Now that we have instantiated the lower-level components of this system, we can initialize the final components of the authentication system by instantiating the `FuzzyExtractor` and `DeviceAuthenticator` classes. This is achieved through by the following two lines of MATLAB code:

```
fuzzyExtractor = FuzzyExtractor(encoder,universalHasher,cryptoHasher);
devAuth = DeviceAuthenticator(fuzzyExtractor);
```

The first line in this code calls the constructor for the `FuzzyExtractor` class. This constructor has three parameters, which are the individual objects extending `IEncoder`, `IUniversalHasher`, and `ICryptoHasher`, in that order. In this case, we are simply using the encoder and hashing objects we created earlier. Finally, the last line of code calls the constructor for the `DeviceAuthenticator` class, which simply takes in the `FuzzyExtractor` object as its only parameter.

At this point, we have completed the setup process for this example of a fuzzy extractor authentication system. The following MATLAB code sums up the individual steps that we have completed in this process:

```
% Retrieve the list of files containing the SRAM readings
fileList = dir('C:\mat_data\*.mat');

% Setup a single device
dev = VirtexBoardDevice('device_1',fileList);
devReader = WordXorDeviceReader(0,dev,9);

% Setup the authentication system
encoder = ConcatenatedBCHEncoder(511,76,8);
universalHasher = ArrayHash(128);
cryptoHasher = SimpleCryptoHash('SHA-512');

fuzzyExtractor = FuzzyExtractor(encoder,universalHasher,cryptoHasher);
devAuth = DeviceAuthenticator(fuzzyExtractor);
```

Now that the authentication system is fully constructed, we can perform the enrollment and authentication procedures on the device that we created earlier. These procedures are executed by the following MATLAB code:

```
devAuth.enroll(devReader);
isEnrolled = devAuth.authenticate(devReader);
```

As the name suggests, the `enroll` method called from the `DeviceAuthenticator` object enrolls the `WordXorDeviceReader` that we created earlier into the system. Next, the `authenticate` method performs the authentication procedures on the device reader to determine if the device is enrolled in the system. The result of this operation is returned from this method, which is stored in the `isEnrolled` variable.

By now, we have covered the entire procedure by which we set up a fuzzy extractor authentication system. We have also covered how the two basic operations of enrollment and authentication are executed using this framework. Using these simple procedures, we can construct varying types of fuzzy extractor authentication systems. In turn, these systems can be used to observe the behavior of authenticating different types of devices or other media that present a suitable PUF.

4.3 Extending the Framework

At some point during the lifecycle of this prototype system, it may be necessary to extend its functionality. Fortunately, this framework was developed with extendibility in mind and was designed to make the extension process as simple as possible. New components can be added to the framework by simply extending the appropriate abstract components. For example, if we discovered a new, more efficient universal family of hash functions, we could add an implementation of this to the framework by creating a class that extends the `IUniversalHash` interface. To better illustrate how this framework can be extended, we will follow the process of adding a new class that extends the `ADevice` abstract class using MATLAB 7.9.0 (R2009b). We will also briefly review some of the object-oriented MATLAB syntax and semantics as they are needed.

4.3.1 Creating a New Class File

Let us begin by first navigating to the Device package in the source folder of this framework in MATLAB, as shown in Figure 4-8. It is assumed that the source code for the framework is readily accessible on the host machine and that the necessary working folders have been configured in MATLAB.

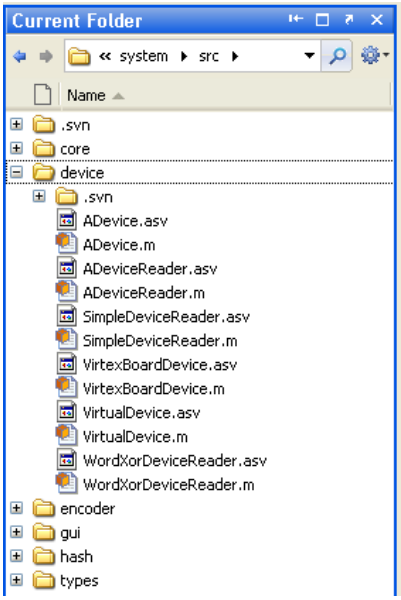

Figure 4-8. Device Package in MATLAB

Next, we create a new MATLAB class by right-clicking the device folder and creating a new "Class M-File", as shown in Figure 4-9.

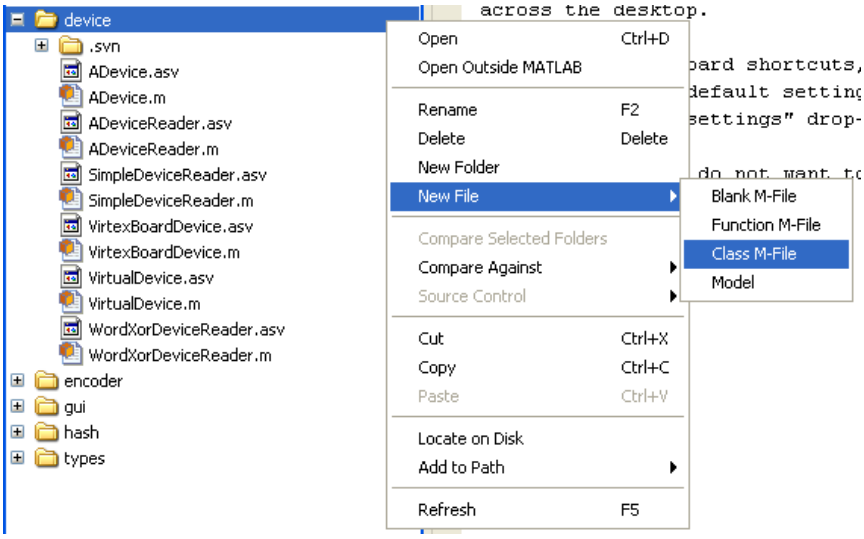

Figure 4-9. Creating a New MATLAB Class File

For the purposes of this example, we will name this file "SimpleDevice.m", which will also effectively require the class name to be `SimpleDevice` as well.

4.3.2 Implementing the New Class

With our new class file created, we can now begin implementing this class. We start this process by first defining our class in the following code:

```
classdef SimpleDevice < ADevice
```

```
        %SIMPLEDEVICE A simple implementation of ADevice
        %
        %   The SimpleDevice class extends the ADevice abstract class by
        %   providing user-defined data.
        %
        %   Worcester Polytechnic Institute
        %   General Dynamics C4 Systems
        %   Fuzzy Extractors MQP
        %
        %   Author(s):  Isaac Edwards
        %               Patrick Newell
        %               Chris Trufan
        %
        %   Copyright 2010

    end
```

In the above code, we first define our class, `SimpleDevice`, by using the `classdef` keyword. We then indicate that this class extends the ADevice class by adding "`< ADevice`". In Java, this would be synonymous with "`class SimpleDevice extends ADevice`", or in C# with "`class SimpleDevice : ADevice`". This class definition is terminated with the `end` keyword.

It is also important to note the formatting of the comments (denoted by the '%' character) directly underneath the class definition. This is more or less the standard MATLAB documentation format for MATLAB classes. In addition to providing other developers with useful information regarding the class, these comments also allow for documentation to be auto-generated by MATLAB when the `doc` command is invoked. For example, when the command, `doc SimpleDevice`, is invoked in the MATLAB interpreter, a window similar to that shown in Figure 4-10 would be displayed. This help window displays the details of the `SimpleDevice` class, such as its properties, methods, and superclasses. This command can also be invoked on any of the existing classes in the framework.

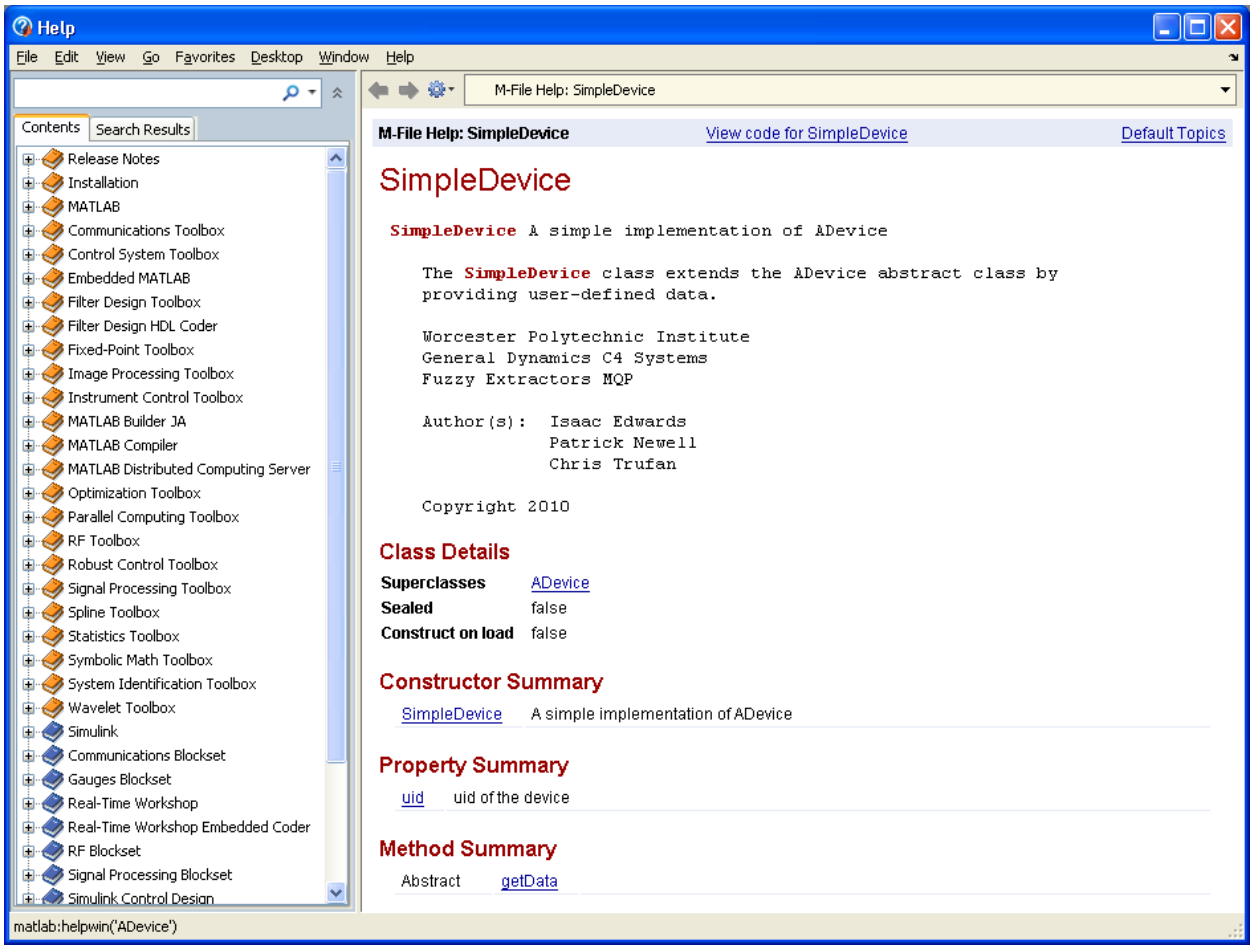

Figure 4-10. Auto-generated MATLAB Documentation for the SimpleDevice Class

We can now begin to fill in the details of this class to finish our implementation of `SimpleDevice`. Let us first define the class properties for this class. Class properties are variables that are members of their parent class. For the purposes of this example, these properties are essentially synonymous with class variables in Java and C#. Below, we first define a "class properties" section by using the `properties` keyword and then list our single class property as shown below:

```
    % ================================================
    % Private Properties
    % ================================================
    properties(SetAccess='private',GetAccess='private')

        data    % Authentication data

    end
```

In the above code, we define a single class variable, `data`, which will contain the data for this device. We also set the variable access to private through the `SetAccess` and `GetAccess` meta-properties so that the variable is only visible inside the class and cannot be accessed outside the class. It should be noted that the `SetAccess` and `GetAccess` default to public when their values are not specified.

Now that we have our class property, we can define the methods for this class. We start by writing the method stubs for the constructor and the implementation of the abstract method, `getData`. This is shown in the code below:

```
    % ================================================
    % Public Methods
    % ================================================
    methods

        function self = SimpleDevice(uid,data)
            % SimpleDevice(data)
            %
            %   Constructor for the SimpleDevice class.
            %
            %   Parameters:
            %   - data      data to be used for authentication

        end

        function retData = getData(self)
            % retData = getData(self)
            %
            %   Retrieves the data for this device.
            %
            %   Parameters:
            %   - self      calling SimpleDevice object
            %
            %   Returns:
            %   - data      the device data

        end

    end
```

Note that the methods for this class must also be defined within a method block using the `method` keyword, similar to how we defined the class properties. The individual methods in this block are defined using the `function` keyword. The first method in this block is the constructor for the class. For this constructor, we require two parameters, which are the UID and the data for this device. The second method in this block is the `getData` method, which is the implementation of the abstract

method in `ADevice`. The `getData` method has a single parameter, `self`, which is required for all non-static methods in MATLAB classes. This is similar to how class methods are defined in Python. However, this parameter is not required for the constructor.

Let us now implement both the constructor and the `getData` method. The constructor is relatively simple in this case, as all we need to do is set the value of the class variable, `data`, and the inherited class variable, `uid`. This is accomplished through the following code:

```
function self = SimpleDevice(uid,data)
    % SimpleDevice(data)
    %
    %   Constructor for the SimpleDevice class.
    %
    %   Parameters:
    %   - data      data to be used for authentication

    % Set the UID
    self.uid = uid;

    % Set the value of data
    self.data = data;
end
```

The implementation of the `getData` method is also simple in this case, as shown by the following code:

```
function retData = getData(self)
    % retData = getData(self)
    %
    %   Retrieves the data for this device.
    %
    %   Parameters:
    %   - self      calling SimpleDevice object
    %
    %   Returns:
    %   - data      the device data

    % Return the data
    retData = self.data;

end
```

In the above code, the variable, `retData`, is the return value for the function. In this particular case, we simply set the value of `retData` to the value of the class variable, data. Also note that in both the constructor and in the `getData` method, the class variable is referred to by adding the prefix, "self.", to the variable. This indicates that the variable is a member of the containing class.

At this point, our implementation of the `SimpleDevice` class is complete. The entirety of this implementation is shown in the MATLAB code below:

```
classdef SimpleDevice < ADevice
    %SIMPLEDEVICE A simple implementation of ADevice
    %
    %   The SimpleDevice class extends the ADevice abstract class by
    %   providing user-defined data.
    %
    %   Worcester Polytechnic Institute
    %   General Dynamics C4 Systems
    %   Fuzzy Extractors MQP
    %
    %   Author(s):  Isaac Edwards
    %               Patrick Newell
    %               Chris Trufan
    %
    %   Copyright 2010

    % ================================================
    % Private Properties
    % ================================================
    properties(SetAccess='private',GetAccess='private')
        data    % Authentication data
    end

    % ================================================
    % Public Methods
    % ================================================
    methods
        function self = SimpleDevice(uid,data)
            % SimpleDevice(data)
            %
            %   Constructor for the SimpleDevice class.
            %
            %   Parameters:
            %   - data      data to be used for authentication

            % Set the UID
            self.uid = uid;

            % Set the value of data
            self.data = data;
        end

        function retData = getData(self)
            % retData = getData(self)
            %
            %   Retrieves the data for this device.
            %
            %   Parameters:
            %   - self      calling SimpleDevice object
            %
            %   Returns:
            %   - data      the device data
```

```
            % Return the data
            retData = self.data;
        end
    end
end
```

4.3.3 Testing the New Class

Now that we have implemented the `SimpleDevice` class, we can test its functionality with the rest of the framework. Just as we did in Section 4.2, let us start by constructing and initializing the device components. This time, however, we will use our newly created `SimpleDevice` class as our object that extends `ADevice`. We start by instantiating an instance of the `SimpleDevice` class through the following code:

```
data = randi([0 1],10000,1);
dev = SimpleDevice('device_1',data);
```

The first line of code simply populates the contents of the vector, `data`, with 10,000 random 0's and 1's using the `randi` function provided by MATLAB. The second line of code calls the constructor for our `SimpleDevice` class. As we know from its implementation earlier, the constructor for this class requires a device UID and a vector of data. Here, we simply provide the value, "device_1", for the device UID and the contents of the vector, `data`, for the device data.

After instantiating the `SimpleDevice` object, we can follow the steps provided in Section 4.2 to create a simple fuzzy extractor authentication system and then test the enrollment and authentication operations on this new device object. If the system is correctly implemented, a `SimpleDevice` object should never fail to authenticate if it is enrolled, simply because it always produces the same, noise-free data. While this may not seem very useful, it does provide a baseline for varying constructions of a fuzzy extractor authentication system. Since we know that any `SimpleDevice` object should authenticate correctly every time when it is enrolled in the system, we can quickly detect if errors are present in the system if a `SimpleDevice` fails to authenticate.

5 BCH C Implementation

In practice, a fuzzy extractor authentication system would most likely be implemented as an embedded system. Although an object-oriented framework is ideal for prototyping and modularity, a more concise system-level software or even hardware implementation would be more appropriate in terms of efficiency. In this section, we discuss an implementation of the BCH encoder and decoder component of the fuzzy extractor in the C programming language.

5.1 Mathematical Representation

There are two types of values that we store for our BCH computations: polynomials with binary coefficients and powers of alpha. To store the polynomials with binary coefficients, any data type we can perform binary operations on is acceptable. Unsigned integers were initially chosen for ease of use and debugging, however a single variable will not handle BCH codes of any significant length. To solve this problem, we use a C structure which contains an array of unsigned integers and another unsigned integer to provide a length of the data contained within the structure.

```
typedef struct Polynomial_s {
      unsigned int* poly;
      unsigned int length;
} Polynomial;
```

To work with the structures we use several functions that recreate operators that work with single unsigned integers. There are several bitwise functions, including XOR, AND, OR and set of logical shifts that all work on Polynomial structures of any length. A few other miscellaneous functions include functions to check equality or a bit position and functions to initialize and destroy the data structure.

The code that uses single unsigned integers remains in the code listing as a way to familiarize oneself to the code without worrying about the specific handling of the data structures that must be done, such as memory allocations.

The other values that need to be represented are powers of alpha which are used when working with the Galois fields. However we can simply use a single unsigned integer which can handle sufficiently large codes for most purposes.

5.2 Finding Primitive Polynomials

We take a naïve approach when searching for a primitive polynomial to use for our BCH code. The function `generateMinimumPrimitivePolynomial` takes in the order of the polynomial to find,

computes the minimum and maximum values of the possible polynomials of that order and searches through all the possible values by passing the value to the `isPrimitivePoly` function. The minimum value is calculated as $2^{order} + 1$ and the maximum value is $2^{order+1}$. The range is reduced by half if we take into consideration the fact that for a polynomial to be primitive it must be irreducbile so it must contain a 1 coefficient for x^0, or else it would be divisible by some power of x. With this in mind, we count upward from the minimum to the maximum by two. The `isPrimitivePoly` takes in a polynomial to test and an order. The function first tests that the polynomial does not have an even number of terms, as this would cause it to be divisible by x + 1. It then continues by checking to see if raising the polynomial to higher and higher powers of alpha causes a 1 to appear before it should. In this way, the function checks if the polynomial generates the entire field and thus is primitive.

5.3 Generating Minimal Polynomials

To generate the minimal polynomials, we use the function `generateMinPoly` which takes in a primitive polynomial, the order of the unsigned polynomial and a number indicating which minimal polynomial to calculate. The function builds a list of polynomials which are the values of α^{ik}, and at every new polynomial calls `findMinimalPolynomialZeros` which attempts to find a linear combination of the polynomials which when added together equals zero. The `findMinimalPolynomialZeros` function simply looks through all possible combinations of polynomials until it finds a combination that XOR's to 0. Along the way, it keeps track of a resulting minimal polynomial by XOR'ing a result variable by the index of the polynomial in the array it receives from `generateMinPoly`.

5.4 Generating BCH Generator Polynomials and BCH Encoding

To generate the BCH generator polynomial, we build off of the functions we already have to find a primitive polynomial and the minimal polynomials we need. We then multiply the minimal polynomials together using the `polynomialMultiply` function which shifts and XORs because we are working with binary coefficients.

To encode a message with BCH we simply use a BCH generator polynomial and some bitwise operations followed with a call to `polynomialDivideRemainder` which finds the remainder of dividing the generator polynomial and the shifted message that we passed in. We then OR the result to the shifted message to make the completed codeword.

5.5 Computing Syndrome Components

To compute the syndrome components of a received codeword, we use the `computeSyndromeComponents` function. The function takes in the received codeword and the parameters of the BCH code. It generates a primitive polynomial and some minimal polynomials and then uses the `polynomialDivideRemainder` function to take the modulo reduction of the codeword by the *i*th minimal polynomial. We then pass the computed syndrome components to `computeSyndromePowerComponents` which plugs in the consecutive powers of α to compute the power componets. It does this by looping through each component and checking if the current bit is set, if it is the function XORs the remainder variable by the row value of the power of α equal to the bit position multiplied by the index of the current component. Since we know that the power components will always be a single power of α, we can get the α power that corresponds to the value computed and return that for each component.

5.6 Computing the Error Locator Polynomial

To compute the error locator polynomial, we use the `computeErrorLocatorPolynomial` function. This function takes in the power components computed in the previous step and fills in the arrays following the Peterson-Gorenstein-Zierler alrgorithm. We then use the `matrixInverseGF` function to find the inverse of the *S* matrix which contains powers of α. If the inverse is not successful and our *t* is still greater than zero, we recursively call the error locator function with a decremented *t*. If the inverse is not successful and our *t* is zero, the algorithm fails and we return null. However, if the inverse is successful, we call another function `matrixMultiplyGF` which multiplies the S^{-1} and the *C* matrices containing powers of α. This gives us the coefficients of the error locator polynomial and we cleanup the output a bit before returning the values.

5.7 Chien's Search and BCH Correction and Decoding

To perform Chien's Search, we call the `chienSearch` function, passing in the coefficients of the error locator polynomial and a few other parameters. The function loops through all of the coefficients, which are powers of α and increases their power by their position in the array. It then checks if XOR'ing all of their values together equals 1. If the values do not equal 1 then it continues increasing the powers of α. If they do equal 1, it adds the current error location to the array by subtracting the power increase from the length of the code and continues. Once it has found error locations equal to the number of coefficients it received, it returns the array of error locations.

To correct the received codeword, we build upon the previous sections by using the functions to compute the syndromes, compute the power components, compute the error locator polynomial and find the error locations. If the syndrome components are all zero, we know that the codeword was received without error so we can return the received codeword without correction and skip the other steps. If not, we pass the values down through the functions until we receive error locations from Chien's Search. Once we have the error locations, we simply flip the bit at each error location and return the codeword as corrected. To decode, we simply call the correction function and return the corrected codeword with the error check bits shifted out.

6 Future Work

There are a myriad of ways in which this project could be extended. It would be useful to gather SRAM readings from devices of the same model to ensure that the characteristics such as noise and entropy remain similar from board to board. Similarly, it could be useful to try and test boards that are not only the same model but were also produced consecutively out of the same production process. This would help us to determine the source of uniqueness between devices. It would also be useful to test a single board under a wide variety of conditions. For example, we could observe how noise changes with respect to changing temperatures or exposure to electromagnetic radiation. In order for our fuzzy extractor to be functionally useful, it is necessary to know whether we can consistently expect a board to provide the same readings within a given error tolerance even after a modest amount of wear and tear.

Timing analysis could also prove to be a useful area for future work. While we examined the timing of BCH decoding in MATLAB, different physical systems may have different timing results or memory requirements, particularly depending on how they perform the decoding process. Other non-MATLAB implementations of a fuzzy extractor would benefit from a more thorough examination of what the time and space requirements of each BCH code are. Coding theory also provides yet another avenue of advancement for the project. The use of codes other than BCH codes could be examined. The use of BCH codes themselves could also be extended – for example, there are more powerful ways to concatenate codes than the ones which we used for this project.

Perhaps the most interesting way in which to extend this project is to automate the process of calculating optimal parameters. Our system could fairly easily be extended to automatically calculate the bit-error rate and entropy, and perform the calculations necessary to determine what BCH codes are optimal. Even the process of taking readings from a device could conceivably be incorporated into our fuzzy extractor, such that you someone could simply connect new devices and the system would pick a BCH code and bit extraction pattern.

Finally, one could perform more work to examine the bit correlation that we found in the SRAM. One could also look for more patterns and correlations between bits. More boards of the same model could be used to try and gain more information about the SRAM behavior, or different model boards could be examined for patterns. It could prove fascinating to try and determine whether the bias towards more zeros or ones in a given word had a relationship to the physical layout of the SRAM.

As was already stated, there are a myriad of ways in which this project could be extended. As a result of that, there are far too many to enumerate here. Our fuzzy extractor is modular enough that it

would be relatively simple to extend any given section of it, and there are still many ways in which SRAM data itself could be analyzed.

7 Works Cited

1. **Markoff, John.** Cyberwar - Secure Sources Lacking for Weapons Electronics. *The New York Times.* [Online] October 26, 2009. [Cited: March 6, 2010.] http://www.nytimes.com/2009/10/27/science/27trojan.html.

2. —. F.B.I. Says the Military Had Bogus Computer Gear. *The New York Times.* [Online] May 9, 2008. [Cited: March 6, 2010.] http://www.nytimes.com/2008/05/09/technology/09cisco.html?_r=2&ref=technology.

3. **Dailey, Matthew D. and Shomorony, Ilan.** *Authentication Schemes based on Physically Unclonable Functions.* s.l. : Worcester Polytechnic Institute, 2009.

4. **Berlekamp, Elwyn R.** *Algebraic Coding Theory.* New York : McGraw-Hill, 1968. 978-0894120633.

5. **Chien, R. T.** Cyclic Decoding Procedures for Bose-Chaudhuri-Hocquenghem Codes. *IEEE Transactions on Information Theory.* 1964, Vols. IT-10, October 1964.

6. **Mitzenmacher, Michael and Upfal, Eli.** *Probability and Computing, Randomized Algorithms and Probabilistic Analysis.* New York : Cambridge University Press, 2005.

7. **Carter, J. Lawrence and Wegman, Mark N.** Universal Classes of Hash Functions (Extended Abstract). *Proceedings of the ninth annual ACM symposium on Theory of computing.* 1977, pp. 106 - 112.

8. *CDs Have Fingerprints Too.* **Hammouri, Ghaith, Dana, Aykutlu and Sunar, Berk.** Heidelberg, Germany : Springer-Verlag, 2009, In: Clavier, C., Gaj, K., (eds.) Proceedings of the 11th Workshop on Cryptographic Hardware and Embedded Systems (CHES 2009), LNCS, Vol. 5747, pp. 348-362.

9. *The Context Tree Weighting Method: Basic Properties.* **Willems, Frans M. J., Shtarkov, Yuri M. and Tjalkens, Tjalling J.** 1995, IEEE Transactions on Information Theory, Vol. 41, pp. 653-664.

10. *Near-Optimum Decoding of Product Codes: Block Turbo Codes.* **Pyndiah, R. M.** 8, August 1998, Communcations, IEEE Transactions on, Vol. 46, pp. 1003-1010. 0090-6778.