CHOICE OF OPTIMAL ERROR-CORRECTING CODE FOR PHYSICAL UNCLONABLE FUNCTIONS

by

Brian Jarvis

A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Engineering

Committee:

_____	Dr. Kris Gaj, Thesis Director
_____	Dr. Jens-Peter Kaps, Committee Member
_____	Dr. Houman Homayoun, Committee Member
_____	Dr. Monson H. Hayes, Department Chair
_____	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering
Date:_____	Fall Semester 2015 George Mason University Fairfax, VA

Choice of Optimal Error-Correcting Code for Physical Unclonable Functions

A Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

by

Brian Jarvis
Bachelor of Engineering
Vanderbilt University, 2006

Director: Kris Gaj, Associate Professor
Department of Electrical and Computer Engineering

Fall Semester 2015
George Mason University
Fairfax, VA

DEDICATION

This is dedicated to my wife, the lovely and talented Amanda Jarvis. Thank you for all of your help and support as I pursued my degree and for helping me stay motivated to finish.

ACKNOWLEDGEMENTS

I would like to thank the following people who were instrumental in helping me complete this work:

My advisor, Dr. Kris Gaj, for sharing with me his passion for cryptographic research through course projects and research for this thesis.

The GMU CERG PUF Group: Dr. Kris Gaj, Bilal Habib, Ahmed Ferozpuri, Aaron Hunter, and Cédric Marchand, for the fruitful roundtable discussions of our research pursuits.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

ABSTRACT

CHOICE OF OPTIMAL ERROR-CORRECTING CODE FOR PHYSICAL UNCLONABLE FUNCTIONS

Brian Jarvis, M.S.

George Mason University, 2015

Thesis Director: Dr. Kris Gaj

This thesis explores error-correcting codes which can be used in physical unclonable function (PUF) applications. We investigate linear block codes and concatenated codes, which are traditionally used with PUFs, and compare them to convolutional codes using the criteria of error correction capability, decoder hardware requirements, flexibility of code parameters, and code rate. The application of the selected code to various fuzzy extractor schemes is analyzed. Further, the selected convolutional codes are implemented in hardware using a Xilinx Artix 7 FPGA and its resource utilization estimated. Extensive experiments based on software implementations in C++ and Python are performed in order to determine the code resistance to various error patterns seen at the outputs of practical implementations of PUFs, such as Ring-Oscillator PUF and SR latch PUF. We conclude that convolutional codes, implemented independently or in a concatenated construction, are capable of matching the error correction performance of the often used

BCH code for the majority of realistic error patterns. At the same time, many of the selected convolutional codes occupy far fewer hardware resources when implemented in an FPGA.

CHAPTER ONE: INTRODUCTION

There is a growing need for trust at the hardware level. An increasingly popular method for addressing this need is the use of Physical Unclonable Functions (PUFs). PUFs allow a hardware component to exploit manufacturing variations or any other part-specific data in order to generate a unique ID or key for that individual part. This is analogous to biometric information such as a human fingerprint. The appeal of PUFs is that the same circuit design can be implemented on any number of hardware devices, yet due to the uniqueness of each device, each will produce a unique ID. In [1] a usage of PUFs is described in which a bank customer uses a PUF device to authenticate at an ATM. The advantage brought by PUFs in this application is that an adversary would be unable to duplicate the unique sequence of PUF IDs sent in response to the ATM challenges. Other usages of the PUF cryptographic primitive allow for secure key generation directly from hardware without a need to store the key externally.

A problem arises with PUFs whereby on successive PUF ID generation, the outputs may not be identical. In [2] it is claimed that the error rate in PUF ID regeneration can be as high as 10%. In particular, environmental and system factors such as ambient temperature and voltage fluctuations can influence the PUF ID reproducibility. In order to use a PUF system for establishing trust each device must be

able to reliably regenerate the same ID. It is for this reason that an error-correcting code (ECC) is often used to augment the PUF ID generation.

The field of coding theory is vast and has applications which reach far beyond PUFs, which was the primary application explored during this research. For that reason, the field of candidate codes was pared down to those which use minimal resources when implemented in an FPGA. Due to the typical PUF concept of operations, speed is not as important as implementation area. The ECC implementation must fit in the FPGA alongside the pure PUF and main application using the PUF response. The ECC portion ideally would be just a small subset of the overall circuit, thus must be implemented as efficiently as possible to reduce the resources needed. Additionally, the selected ECC must have maximal error correcting capability while minimizing the encoded output rate. The output rate is the ratio of the size of an encoder's output to the size of its input. The output rate must be small because larger encoded data size necessitates a larger PUF ID, which in turn requires more PUF circuitry. Ideally, the selected code allows for variable parameters for more flexible usage in a PUF-based application.

Traditionally, BCH codes are used in PUF designs. This is due to the strong error correction performance of these codes. The BCH code suffers from two critical flaws, though. First, due to the complexity of the decoder it results in a large area requirement in an FPGA implementation. Second, the BCH code has a finite set of code parameters. When generating private data, a minimum level of entropy, or measure of randomness, is desired. This measures the ability of an attacker to guess the private data as well as the ability to guess any intermediate data which could lead to regeneration of the private

data. In many PUF-based application constructions, this includes the input to an error correcting code. If a particular security level is desired, for example generation of a 256-bit private key, then the size of the input to the ECC must allow at least 256 bits of entropy to be carried by the output of the system. When using BCH codes, only certain combinations of message and codeword sizes allow for the desired level of error correction performance. For this reason, depending on the desired security level, very large BCH codes may be required. This results in a very large amount of public helper data relative to the size of key being generated. Depending on the PUF-based application being constructed, this could also result in requiring a very large number of pure PUF components.

This paper researches alternatives to the BCH error-correcting code for PUF applications. Prior work is evaluated on the bases of error correction capability, decoder hardware requirement, code parameter flexibility, and code rate. Several fuzzy extractor schemes are analyzed with respect to the effects of ECC choice. Key sizes of 128, 192, and 256 bits, the sizes typically used in the AES encryption algorithm, are used in these analyses. An optimal convolutional code is selected which best fits these schemes and maximizes the selection criteria. FPGA and C++ implementations of the selected code are compared against the BCH code with results analyzed.

CHAPTER TWO: BACKGROUND

Three key concepts are relevant to this research: physical unclonable functions, error-correcting codes, and fuzzy extractors.

Physical Unclonable Functions

The field of PUFs is becoming very vast and encapsulates many different constructions. An extensive study of PUFs is presented in [21]. In it, various PUF and PUF-like designs are analyzed, including analog, delay-based, and memory-based PUFs. While the individual designs vary greatly and continue to be an area of active research, at their core they share a common fundamental idea. A circuit can be designed which is identical in theory, yet produces output which is specific to each device when implemented in hardware. This property makes PUFs a very powerful construct when used in cryptographic applications; a PUF circuit is capable of generating device-specific secret information which can be used in key generation or challenge-response schemes.

PUFs have an inherent problem which has not yet been completely eliminated. On subsequent regenerations of the PUF ID, the response will vary slightly. Active research continues to push the reliability higher in new PUF designs, however no PUF is able to reproduce an identical response 100% of the time. Also, even the most reliable PUFs fall victim to environmental fluctuations such as temperature and voltage, which cause a

decrease in a PUFs reproducibility. This is why PUFs must be combined with an error-correcting code and implemented in a fuzzy extractor to regain 100% reproducibility.

Another inherent problem that PUFs have is non-uniformity of their responses. Ideally, a PUF ID would have 50% zeros and 50% ones, randomly distributed. In reality, PUFs will have varying uniformity with certain bit positions biased slightly towards one value or the other. These effects reduce the minimum entropy, or measure of randomness, of the PUF ID. When used in a fuzzy extractor, a hash function is included in the construction. This component increases the randomness and uniformity at the output, at the expense of decreased entropy in the generated data.

Error-Correcting Codes

Error-correcting codes are a system which allows for errors in a data set to be detected and corrected. A typical ECC scheme involves an encoder and a decoder. The encoder adds redundancy to an input message to create a codeword. The decoder is able to recover the original input message from the codeword, even in the presence of bit errors. A codeword incurs errors due to storage in or transmission through a noisy medium, for example magnetic disk storage or wireless transmission. In the case of a PUF, the unreliable bits are the source of error. This general data flow is illustrated in Figure 1.

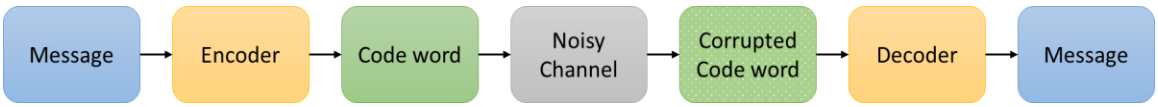

Figure 1 Error correcting code data flow

It is often the case that an ECC's code parameters require an input message size which is smaller than the total amount of data to be encoded. In this situation, the larger word is broken into smaller message blocks which are able to be passed to the encoder. After decoding, each of these message blocks reconstructs the original word.

There are two broad classes of ECC into which most codes fall: block codes and convolutional codes. Regardless of the specific code or class of codes, a common vocabulary will be used to describe each:

Word The source data to be encoded.

Message The input data to the encoder. May be less than or equal to the size of the word.

Codeword The output data from the encoder.

Message length (k) The size, in bits, of the input data to the encoder.

Codeword length (n) The size, in bits, of the output data from the encoder

Correctable errors (t) The number of errors that an ECC is guaranteed to correct.

Code Rate (R) The ratio of message bits to codeword bits; k/n.

Output Rate (R^{-1}) The measure of expansion through the encoder. This can be thought of as the number of output bits per bit of input; n/k.

Block Codes

Block codes, as the name implies, are a class of codes that operate on a fixed size of bits at a time. The codeword output from a block code is a function of the input

message only. Examples of block codes include BCH codes, Reed-Muller codes, and Golay codes. The following class-specific vocabulary is used to describe block codes.

Minimum distance (d_{min}): The smallest number of bits which differ between any two codewords that an encoder can output. This property has an important relationship to *t*, the number of correctable errors:

Equation 1 Minimum distance

$$d_{\min} = 2t + 1$$

Equation 2 Correctable errors

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor$$

This property indicates that *t* errors can be corrected over a block of *n* bits. Block codes are typically denoted using the following triplet of parameters: (n, k, d_{min}).

Convolutional codes

Convolutional codes are a class of ECC in which the codeword output is a function of the input message and internal state. This differs from block codes which produce codewords only as a function of input message. The input and output sizes of a convolutional code are typically much smaller than that of a block code. For example, a common code rate for convolutional codes is 1/2, with just one bit of input and two bits of output. Convolutional codes have the following class-specific vocabulary.

Constraint length, K: The depth of a convolutional code. Each of the *n* bits generated by a convolutional encoder is a function of the *k* input message bits and the last *K*-1 inputs. In general, the larger the value of *K*, the more complex the decoder will be. This is because there are *K*-1 memory elements and 2^{K-1} decoder states for a code with constraint length *K*.

Free distance (d_{free}): The smallest Hamming distance between any two finite-length codewords in which the internal state of the encoder starts and ends in the all-zero state. This property is analogous to the minimum distance (d_{min}) in block codes. Similarly, it has the same relationship to the number of correctable errors:

Equation 3 Free distance

$$d_{free} = 2t + 1$$

Equation 4 Correctable errors

$$t = \left\lfloor \frac{d_{free} - 1}{2} \right\rfloor$$

The number of correctable errors in the context of a convolutional code is not, by itself, a useful metric. This is because convolutional codes do not have discrete blocks which are encoded and decoded. In theory, an infinite stream of input bits could be constantly encoded. In this hypothetical scenario, it does not make sense that there would be a finite number of correctable errors on the decoding side. In reality, the correction of *t* errors occurs inside of an *error correction window*.

Error correction window: The span of bits over which *t* errors can be corrected in convolutional decoding. The window can be thought of as sliding across the codeword, correcting *t* errors at a time. For example, if a code's error correction window is 8 bits and the code has a value of *t*=3, then the decoder will be guaranteed to recover the original message as long as no more than 3 errors are present over a span of any 8 bits.

Convolutional codes are denoted using the triplet of parameters, (*n, k, K*).

Concatenated codes

Independent of the class of ECC, there is a code construction technique known as concatenation in which multiple codes can be combined to form a larger ECC. This construction was first described in [7] and is depicted in Figure 2.

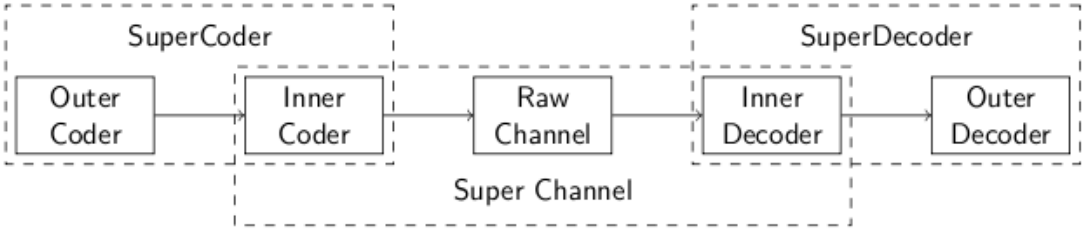

Figure 2 Concatenated code construction

Two stages of encoders and decoders are used in this construction. The outer encoder first encodes the message. The inner encoder then encodes the output from the outer encoder. The output from the inner encoder is the codeword of the concatenated ECC. On the decoding side, the codeword is first decoded with the inner decoder. The outer decoder then decodes the output of the inner decoder. The output from the outer

decoder is the original message. The concatenated coding approach leads to a very powerful property. The parameters *n*, *k*, and d_{min} (or d_{free}, in the case of a convolutional code) of the component codes are multiplied together to derive the concatenated code's parameters.

Equation 5 Concatenated minimum distance

$$d_{\min_{concat}} = d_{\min_{outer}} * d_{\min_{inner}}$$

Equation 6 Concatenated message size

$$k_{concat} = k_{outer} * k_{inner}$$

Equation 7 Concatenated codeword size

$$n_{concat} = n_{outer} * n_{inner}$$

When considering Equation 4 and the relationship between d_{min} (or d_{free}) and *t*, it becomes clear how an additive cost in implementation complexity results in a multiplicative increase in error correction capability.

Fuzzy Extractors

Fuzzy Extractors are comprised of several individual components and address two problems which arise from using PUFs as cryptographic primitives: non-reproducibility and non-uniformity. First described in [6], a fuzzy extractor extracts nearly uniform randomness from its input while at the same time is error-tolerant such that if the input changes up to a prescribed amount the output remains the same. To accomplish these goals, the following cryptographic primitives are included in most PUF-based fuzzy

extractor designs: PUF, random number generator (RNG), and hash function. Additionally, an ECC is used in the construction to address the non-reproducibility problem.

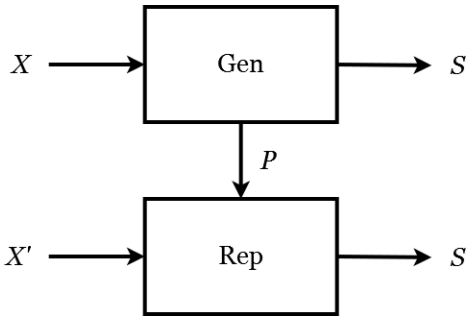

Figure 3 High level operation of a fuzzy extractor

Figure 3 illustrates an example of how a fuzzy extractor operates. Input data, *X*, is used to generate private data, *S*, and public helper data, *P*. The input source is noisy and cannot reliably reproduce the same value of *X* that was used during the initial generation of *S*. Using the public helper data, the reproduction circuit can use a corrupted version of *X*, denoted *X',* to add reliability to the system and reproduce the expected value of *S*.

CHAPTER THREE: COMPARISON OF EXISTING APPROACHES

There are two high level approaches which existing research has taken: using a single ECC to correct a large number of errors, and using multiple smaller ECCs concatenated to achieve a stronger error correction capability.

Single code approaches

There are four block codes which are prevalent in existing single-code designs. BCH codes in particular are widely used, for example in [2]. Repetition codes [3], Reed-Muller codes, and the Golay code [5] are also used. Each of these has advantages and disadvantages. These are summarized in Table 1. In this table, the four prominent ECCs are mapped to the features that an optimal ECC for PUFs should have: large error correction capability, low hardware area requirement, variable code parameters, and low output rate.

Table 1 Features of commonly used linear block codes

Code Feature	BCH	Repetition	Reed-Muller	Golay
Large error correction capability	X	X	X	X
Low area requirement in hardware		X	X	X
Variable code parameters	X		X	
Low output rate, n/k	X			X

Some codes, such as the Golay code, have a fixed value of n. This means that if an input word is larger than n it must be broken into smaller blocks and each block then

encoded independently. For other codes, it may be tempting to intentionally use multiple smaller codes in the same manner to save on area utilization. For example, if a total of *n* bits of codeword is needed, one could use the aggregate of four smaller codes each with codeword size *n*/4 to produce the same amount of output. This leads to these codes being highly susceptible to burst errors. Even though each individual code may be able to correct a large percentage of its codeword length in errors, if more than *t* errors occur over a window of *n* bits the code breaks and cannot reliably correct the error pattern. Because an ECC for a PUF application requires the entire PUF ID to be reproduced, using many small codes in this manner adds a lot of risk since any one of them failing means the entire PUF ID will not be usable. For block codes, the better approach is to use a single code which is capable of correcting a large number of errors over *n* bits, where *n* is as close as possible to the word size.

BCH codes are often selected for PUF applications due to their strong error correction performance with low output rate. For example, the often used *(255, 131, 37)* code is capable of correcting 18 errors in blocks of 255 bits, or 7.06% of the codeword length. This is achieved with an output rate, *n*/*k*, of just 1.95. Additionally, these codes allow for variable code parameters *n* and *k*. However, these parameters are not truly flexible because there is a finite set of valid code parameters that produce functional BCH codes. The disadvantage of this inflexibility becomes apparent in the application of this code to various fuzzy extractor schemes in Chapter Four: Analysis of Fuzzy Extractor Schemes. The major disadvantage to the use of BCH codes lies in the complexity of its

decoder, which scales linearly with the size of the codeword. This makes using the BCH code for larger PUF ID lengths infeasible.

Repetition codes offer extremely strong error correction performance. For example, a *(7, 1, 7)* code is capable of correcting 3 errors in blocks of 7 bits, or 42.86% of the codeword length. A *(13, 1, 13)* code can correct 46.15% of the codeword length in error. Because the values of n and d_{min} are always equal for repetition codes, the error correction performance percentage will asymptotically approach 50% as the size of the code increases. These codes also have the lowest hardware area requirements. Sufficiently small codes can easily be implemented using just combinational logic. This performance comes at the expense of two criteria, though. The code parameter k, by definition, must be 1. This leads to an inflexibility of the code's parameters. Also this means that the output rate for every repetition code will be very high. The smallest practical repetition code, the *(3, 1, 3)* code, has an output rate of 3. In the case of the *(7, 1, 7)* code, the output rate is 7. This means that for a 128 bit input, 896 bits of output are generated. These levels of output rate are too high to be practically implemented when 128, 192, or 256 bit inputs are considered.

Reed-Muller codes provide strong error correction performance with relatively low hardware area requirements. These codes have the disadvantage that their output rates become very high as the error correction capability increases. For example, the *(32, 6, 16)* Reed-Muller code is able to correct 7 errors in a block of 32 bits, or 21.86% of codeword length. Its output rate is 5.33, though. Conversely, Reed-Muller codes with low

output rates also have low error correction performance. For example, the *(32, 26, 4)* code can correct only one error over 32 bits, or 3.13% of codeword length.

Golay codes are very attractive in several categories. These codes have strong error correction performance; they are able to correct 3 errors in blocks of 24 bits, or 12.5% of the codeword length. Additionally, the decoder can be efficiently implemented in hardware with extremely low slice utilization in an FPGA [19]. These codes also have an output rate of 2 which is among the lowest of all highly performant codes. The disadvantage to these codes is that the code parameters are fixed at *(24, 12, 7).* This means that large inputs must be broken into blocks of 12 bits and encoded independently. This leads to the high susceptibility to burst errors which was discussed earlier. Even though the code can correct 12.5% of its codeword length in errors, if more than three errors occur over a window of 24 bits in any of the component block decode stages then the entire PUF ID will not be usable. This level of risk makes these codes an unideal choice.

Concatenated code approaches

Concatenated coding schemes allow for multiple error correcting codes to be cascaded to produce a new code. The concatenated code's parameters are the product of the parameters of the component codes. This allows for multiple small, relatively weak, codes to be combined to form a larger code which has much stronger performance. For example, consider a code capable of correcting 3 errors concatenated with another code capable of correcting 4 errors. Using Equation 1, the minimum distances of each are computed as 7 and 9, respectively. Using Equation 5, the concatenated minimum distance

is found to be 63. Equation 2 is then used to determine the number of correctable errors. For this example, $t = 31$, meaning the concatenated code is capable of correcting 31 errors. Concatenated codes for PUF applications were first explored in [3]. Entropy loss of these constructions was analyzed in [4].

Due to their simple implementations, many existing publications have used repetition codes as the inner or outer code in a concatenated scheme. This has the benefit of a very large increase in error correction performance for a relatively small increase in hardware resources. An unfortunate property of many concatenated constructions lies in the effect that concatenation has on the codeword size, *n*, and message size, *k*, parameters. From Equation 6 and Equation 7, the output rate of a concatenated code can be derived. Recall that output rate, R^{-1}, is equal to n/k. When concatenated, these parameters of each component code are multiplied. Consider, for example, the *(24, 12, 7)* Golay code concatenated with a *(3, 1, 3)* repetition code. Using Equations 5, 6, and 7, the parameters of the concatenated code are calculated to be *(72, 12, 21)*. This new code has very good error correction performance; it can correct 10 errors in a block of 72 bits, or 13.89% of codeword length. Its output rate, however, is 6. This is common amongst any concatenated scheme which uses repetition codes. In practice, the implementations of these concatenated schemes are very efficient in terms of hardware utilization. Unfortunately the high output rates incurred make using them impractical.

CHAPTER FOUR: ANALYSIS OF FUZZY EXTRACTOR SCHEMES

Fuzzy extractors address the problems of non-reproducibility and non-uniformity inherent to PUFs. To achieve reproducibility, a strong fuzzy extractor construction must include a strong ECC. Additionally, in order to guarantee minimum leftover entropy on a generated key, the entropy at various points in the fuzzy extractor circuit must be considered. Four fuzzy extractor schemes first described by Dodis et al. [6] are evaluated below.

Key from PUF

Figure 4 shows the Key from PUF construction.

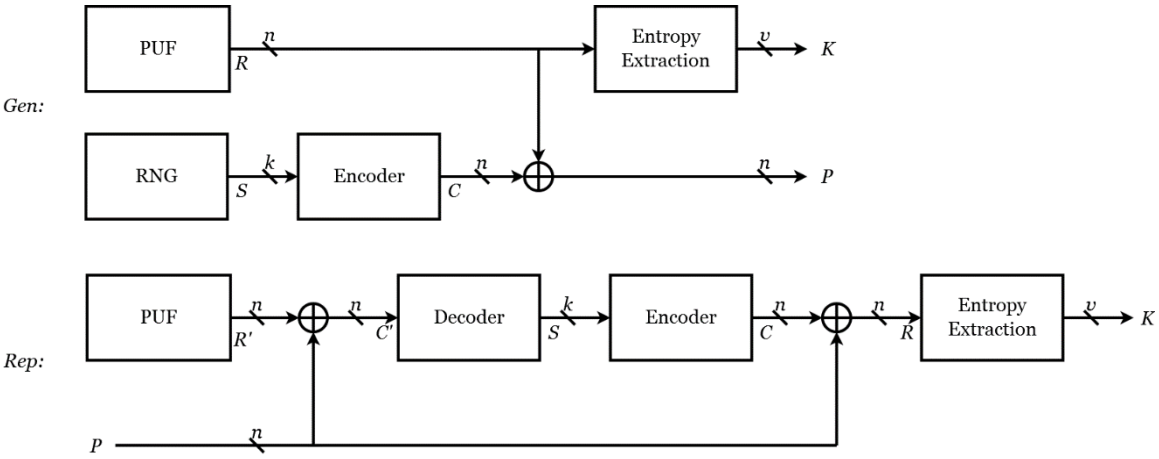

Figure 4 Key from PUF fuzzy extractor construction

This design can also be described by the following equations.

Equation 8 Key from PUF Generation

$$K = h(R)$$
$$P = R \oplus C(S)$$

Equation 9 Key from PUF Reproduction

$$K = h(P \oplus C(D(P \oplus R')))$$

Equation 8 describes the generation of a key, *K*, and public helper data, *P*, from this fuzzy extractor. The key is derived from the PUF output, *R*, passed through an entropy extraction block; typically a cryptographically strong hash function, denoted as *h(R)*. The public helper data is derived from the XOR of the PUF output and an encoded random secret, denoted *C(S)*.

Equation 9 describes how a reproduced PUF output, *R'*, combined with the public helper data is able to reproduce the key. The reproduced, and possibly error-laden, PUF output is XORed with the public helper data to produce *C'*, the encoded random secret which may contain errors. This value is decoded to correct any errors and reproduce the error-free random secret, *S*. The random secret is then encoded to reproduce an error-free *C*. This value XORed with the public helper data reproduces the error-free PUF response, *R*. This can then be used to regenerate the original key, *K*, by using the hash function from the generation phase.

Entropy considerations

The maximum entropy of the key, $H_{max}(K)$, is dictated by the points in the circuit where correctly guessing the value leads to the ability to recreate the key. The PUF

response, *R*, and the random secret, *S*, are two such points. Additionally, the entropy of the key cannot be greater than the size of the key, as an attacker can simply guess the value of the key.

Equation 10 Key from PUF Maximum Entropy

$$H_{\max}(K) = \min[H_{\infty}(R), H(S), v]$$
$$H_{\max}(K) = \min(\rho * n, k, v)$$

Equation 10 describes the maximum entropy of the key generated by this fuzzy extractor. This value is derived from the minimum of the min-entropy of the PUF response, $H_{\infty}(R)$, the entropy of the random secret, $H(S)$, and the entropy of the key itself, v. The min-entropy of *R* is equal to the product of the entropy density, ρ, and the PUF ID length, *n*. This describes the ability to directly guess the PUF ID combined with biases inherent to the PUF itself. The entropy of the random secret is equal to its size in bits, *k*, indicating the ability to directly guess the value. The entropy of the key is also equal to its size in bits, *v*, indicating the ability to guess the value.

Key from Secret

Figure 5 shows the Key from Secret construction.

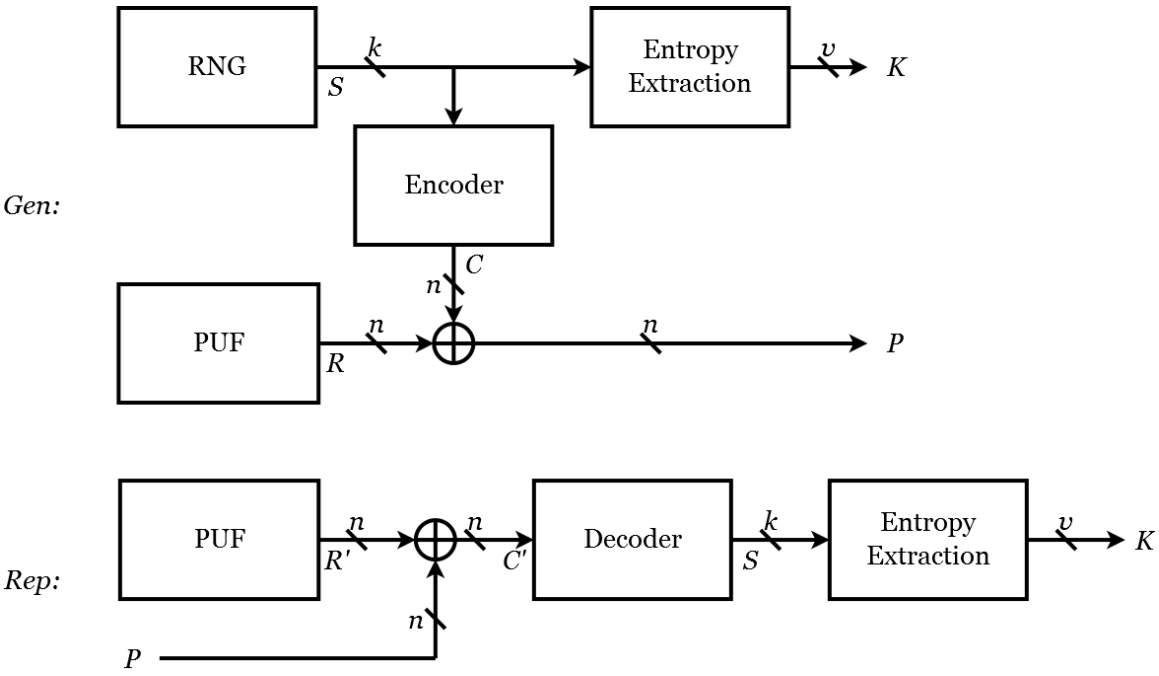

Figure 5 Key from Secret fuzzy extractor construction

This design can also be described by the following equations.

Equation 11 Key from Secret Generation

$$K = h(S)$$

$$P = R \oplus C(S)$$

Equation 12 Key from Secret Reproduction

$$K = h(D(P \oplus R'))$$

Equation 11 describes the generation of a key, *K*, and public helper data, *P*, from this fuzzy extractor. The key is derived from the random secret, *S*, passed through a hash function. The public helper data is derived from the XOR of the PUF output and an encoded random secret. This fuzzy extractor differs from the previous design in that the PUF block and encoded secret blocks are swapped. This results in the public helper data being identically generated and the key being generated as a function of *S* rather than *R*.

Equation 12 describes how a reproduced PUF output combined with the public helper data is able to reproduce the key. The reproduced PUF output is XORed with the public helper data to produce *C'*, the encoded random secret which may contain errors. This value is decoded to correct any errors and reproduce the error-free random secret. The *S* value can then be used to regenerate the key through the hash function.

The reproduction construction for this design is simpler than that of the Key from PUF fuzzy extractor. The public key equations for both constructions are identical, so the order of reproduction for both involves first reproducing the random secret, *S*. The difference is that in the Key from Secret fuzzy extractor, *S* can be used to generate the key directly. This results in a simpler reproduction circuit since the error-free value of *R* never needs to be reproduced.

Entropy considerations

Similar to the Key from PUF construction, the maximum entropy of the key generated by the Key from Secret fuzzy extractor is dictated by the points in the circuit where correctly guessing the value leads to the ability to recreate the key: *R* and *S*. Again,

the entropy of the key cannot be greater than the size of the key, as an attacker can simply guess the value of the key, so the value v is included in the expression.

Equation 13 Key from Secret Maximum Entropy

$$H_{\max}(K) = \min[H_{\infty}(R), H(S), v]$$
$$H_{\max}(K) = \min(\rho * n, k, v)$$

Equation 13 describes the maximum entropy of the key generated by this fuzzy extractor. This value is derived from the minimum of the min-entropy of the PUF response, the entropy of the random secret, and the entropy of the key itself. The values of $H_{\infty}(R)$, $H(S)$, and v are derived in the same way as in the Key from PUF case.

Key from PUF, 2-Secret

Figure 6 shows the Key from PUF 2-Secret construction.

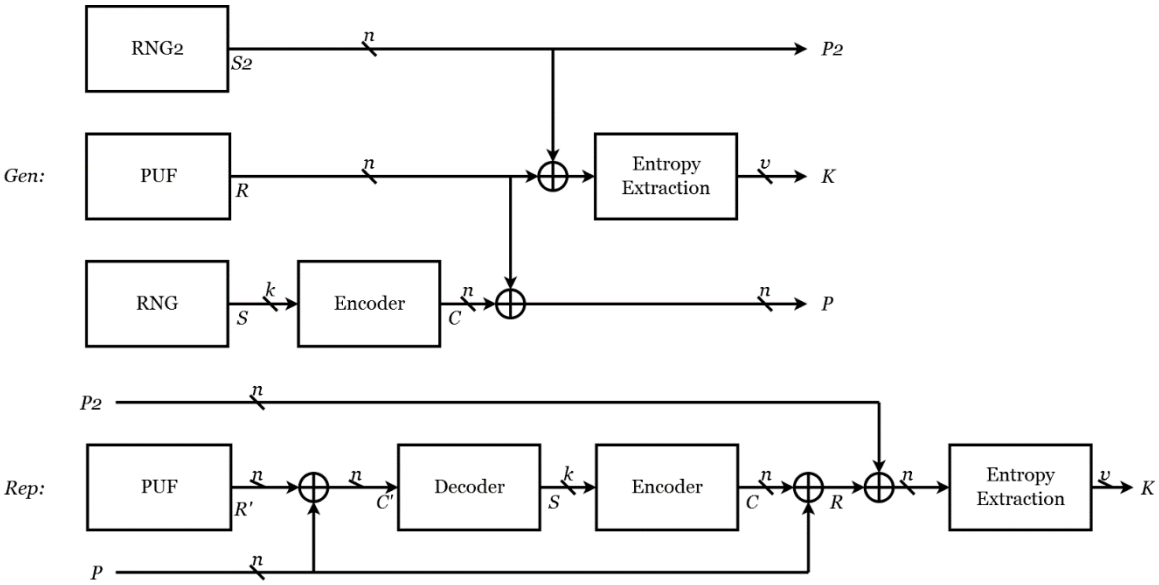

Figure 6 Key from PUF, 2-Secret fuzzy extractor construction

This design can also be described by the following equations.

Equation 14 Key from PUF, 2-Secret Generation

$$K = h(R \oplus S2)$$

$$P = R \oplus C(S)$$

$$P2 = S2$$

Equation 15 Key from PUF, 2-Secret Reproduction

$$K = h\left(P \oplus P2 \oplus C\left(D\left(P \oplus R'\right)\right)\right)$$

This fuzzy extractor builds upon the Key from PUF design by adding a second random secret. Equation 14 describes the generation of a key and two public helper data components, *P* and *P2*, from this fuzzy extractor. The key is derived from the PUF response, *R*, XORed with the second random secret, *S2*, and passed through a hash function. The first component of public helper data, *P*, is derived from the XOR of the

PUF output and the encoded first random secret, *C(S)*. The second component of public helper data, *P2*, is equal to the second random secret. The construction of this generator is identical to the Key from PUF generator with the addition of the *S2* value which gets inserted just before the entropy extraction block and used as part of the public helper data.

The 2-secret construction has the advantage that multiple keys can be generated from a single PUF. A single generation produces a triplet of values (*K, P, P2). K* depends on the values of *R* and *S2*. If a new key is needed then the value of *S2* can be altered to *S2'* produce a new triplet of values (*K', P', P2'*) with no correlation between *K'* and *K*. As a result, the knowledge of *K* and all public parameters (*P*, *P2*, *P'*, *P2'*) does not reveal any information about *K'*.

Equation 15 describes how a reproduced PUF output combined with the two public helper data components can reproduce the key. The reproduced, and possibly error-laden, PUF output is XORed with the first component of public helper data, *P*, to produce *C'*, the encoded first random secret which may contain errors. This value is decoded to correct any errors and reproduce the error-free first random secret, *S*. The first random secret is then encoded to reproduce an error-free *C* value. This value XORed with the first public helper data component reproduces the error-free PUF response, *R*. This, XORed with the second component of public helper data, can then be used to regenerate the original key by using the hash function from the generation phase. The reproduction construction for this design is identical to that of the Key from PUF fuzzy

extractor with the addition of the second public helper data component just before the entropy extraction block.

Entropy considerations

As in the designs discussed previously, the maximum entropy of the key generated by this fuzzy extractor is dictated by the points in the circuit where correctly guessing the value leads to the ability to recreate the key. Even though this construction adds a second random secret, and the generated key is a function of *S2*, this value does not contribute to maximum entropy. This is because *S2* is public data and does not need to be guessed by an attacker. The values which contribute to the maximum entropy are the PUF response, *R*, the first random secret, *S*, and the size of the generated key *K*.

Equation 16 Key from PUF, 2-Secret Maximum Entropy

$$H_{\max}(K) = \min[H_{\infty}(R), H(S), v]$$

$$H_{\max}(K) = \min(\rho * n, k, v)$$

Equation 16 describes the maximum entropy of the key generated by this fuzzy extractor. Similar to the previous designs, the value is derived from the minimum of the min-entropy of the PUF response, $H_{\infty}(R)$, the entropy of the first random secret, $H(S)$, and the entropy of the key itself, v.

Key from Secret, 2-Secret

Figure 7 shows the Key from Secret 2-Secret construction.

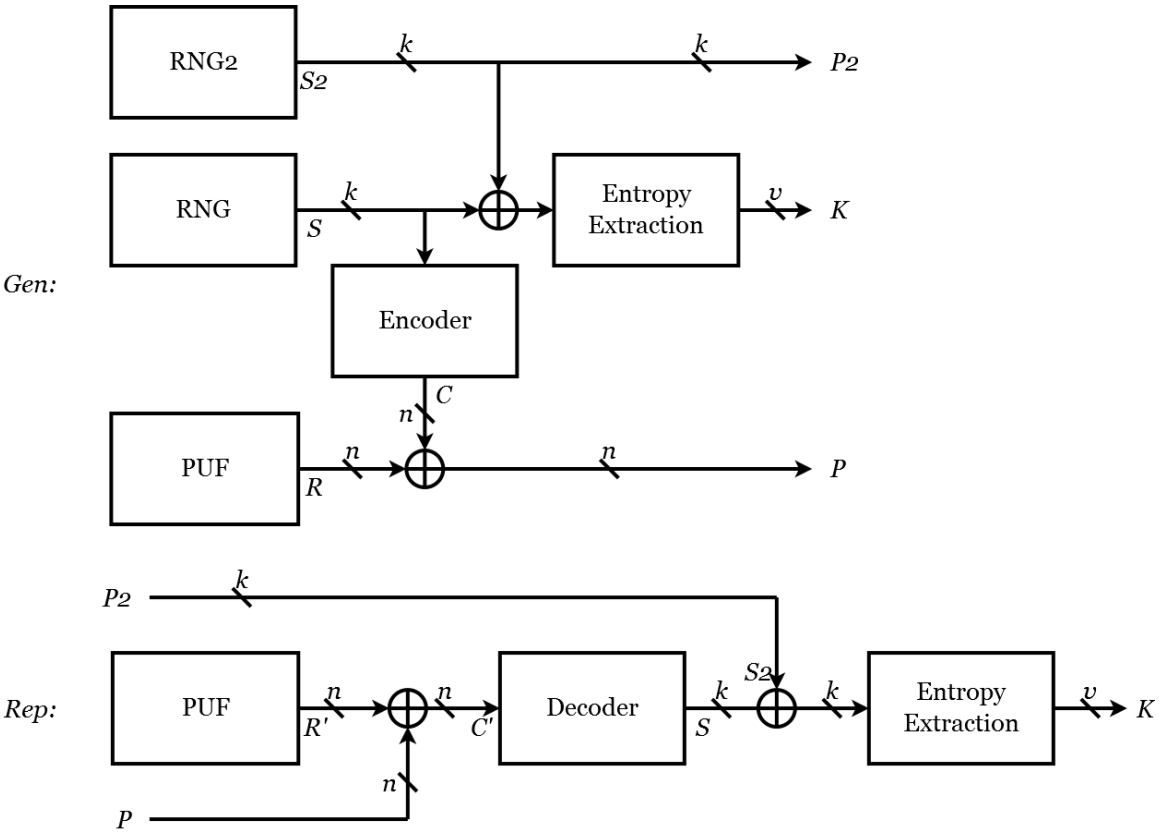

Figure 7 Key from Secret, 2-Secret fuzzy extractor construction

This design can also be described by the following equations.

Equation 17 Key from Secret, 2-Secret Generation

$$K = h(S \oplus S2)$$

$$P = R \oplus C(S)$$

$$P2 = S2$$

Equation 18 Key from Secret, 2-Secret Reproduction

$$K = h\left(P2 \oplus D\left(P \oplus R'\right)\right)$$

This fuzzy extractor builds upon the Key from Secret design by adding a second random secret. Equation 17 describes the generation of a key and two public helper data components, *P* and *P2*, from this fuzzy extractor. The key is derived from the XOR of both random secrets, *S* and *S2*, passed through a hash function. The first component of public helper data, *P*, is derived from the XOR of the PUF output and the encoded first random secret. The second component of public helper data is equal to the second random secret. The construction of this generator is identical to the Key from Secret generator with the addition of the *S2* value which gets inserted just before the entropy extraction block and used as part of the public helper data. Just as in the single random secret designs, this generator differs from that of the Key from PUF, 2-secret construction only in that the PUF block and encoded first random secret blocks are swapped. This fuzzy extractor has an advantage over the Key from PUF, 2-secret design in terms of total public helper data size. This design requires ($k + n$) bits versus ($n + n$) bits in the previous design. The values k and n correspond to the sizes of message and codeword, respectively, for the ECC used in the construction. As a result, n will always be larger than k due to it containing redundancy bits needed for error correction. Thus, the total public helper data size for this fuzzy extractor is preferable to that of the previous design.

This fuzzy extractor also has the advantage over its single random secret counterpart in that multiple keys can be generated from a single PUF. Each generation produces a triplet of values (*K, P, P2). K* depends on the values of *S* and *S2*. Just as with the Key from PUF, 2-secret construction, *S2* can be altered to *S2'* to produce a new triplet of values *(K', P', P2')* which have no relation to the previous triplet of values.

Equation 18 describes how a reproduced PUF output combined with the two public helper data components can reproduce the key. The reproduced PUF output is XORed with the first public helper data component to produce *C'*, the encoded first random secret which may contain errors. This value is decoded to correct any errors and reproduce the error-free first random secret. The *S* value can then be XORed with the second public helper data component and used to regenerate the key through the hash function. The reproduction construction for this design is identical to that of the Key from Secret fuzzy extractor with the addition of the second public helper data component just before the entropy extraction block. This fuzzy extract's reproduction circuit is simpler than that of the Key from PUF, 2-secret design. Similar to the Key from Secret fuzzy extractor, the reproduction phase is simpler by virtue of generating its key from the secret instead of from the PUF. An error-free secret is needed in order to generate an error-free PUF response, so this design eliminates the portion of the circuit dedicated to reproducing the PUF response from the secret.

Entropy considerations

As in the designs discussed previously, the maximum entropy of the key generated by this fuzzy extractor is dictated by the points in the circuit where correctly guessing the value leads to the ability to recreate the key. Like the Key from PUF, 2-secret fuzzy extractor, the *S2* value of this construction does not contribute to the maximum entropy since it is public data. The values which contribute to the maximum entropy are the PUF response, *R*, the first random secret, *S*, and the size of the generated key *K*.

Equation 19 Key from Secret, 2-Secret Maximum Entropy

$$H_{\max}(K) = \min[\, H_{\infty}(R), H(S), v]$$

$$H_{\max}(K) = \min(\, \rho * n, k, v)$$

Equation 19 describes the maximum entropy of the key generated by this fuzzy extractor. Similar to the previous designs, the value is derived from the minimum of the min-entropy of the PUF response, $H_{\infty}(R)$, the entropy of the first random secret, $H(S)$, and the entropy of the key itself, v.

Fuzzy Extractor ECC requirements

The BCH codes are a family of linear block codes which are most often used in existing research. These codes, like all linear block codes, can be described by the triplet of parameters (n, k, d_{min}). These parameters are dictated by fixed relationships between the minimum number of errors the code can correct, t, and the sizes of the message and codeword, k and n. Using BCH code tables from [13] and the maximum entropy equations from the sections above, we determine the parameters required in order to satisfy a minimum error correction performance at various desired entropy levels. Specifically, we desire codes which can be used to generate 128, 192, and 256 bit keys and have an error correction performance of at least 6.25%. This metric is interpreted as the percentage of codeword length, n, that can be in error and still be correctable. The value 6.25% comes from [10], where the worst case number of bit flips observed over 256 bits was 16. So, to meet or exceed a performance of 6.25% for the aforementioned

key sizes, a code must be able to correct 8 errors if n = 128, 12 errors if n = 192, and 16 errors if n = 256.

Table 2 Fuzzy extractor BCH code requirements with public helper data sizes

Key size (bits)	Code Parameters (n, k, d_{min})	(t/n) * 100	Fuzzy extractor scheme public helper data size			
			Key from PUF	Key from Secret	Key from PUF, 2-Secret	Key from Secret, 2-Secret
128	(255, 131, 37)	7.06	255	255	510	386
192	(511, 193, 87)	8.41	511	511	1022	704
256	(1023, 443, 147)	7.14	1023	1023	2046	1466

Table 2 describes the BCH parameters required to implement each of the four fuzzy extractor schemes, with error correction performance of at least 6.25%, at key sizes of 128, 192, and 256 bits. The size of the public helper data generated by each scheme and code combination are also listed. The value (t/n)*100 describes the error correction capability as a percentage of codeword length.

The process for selecting a BCH code parameter triplet which satisfies the error correction and entropy requirements is to first start with the entropy. From the maximum entropy equations discussed previously we know that the three components which drive maximum entropy of the generated key are $H_\infty(R)$, $H(S)$, and v. These expressions evaluate to the sizes ρ*n, k, and v bits, respectively. The value of k will always be greater than or equal to the value of v because the hash function which produces K should have an input size greater than or equal to its output size. Additionally, the value of n will be larger than the value of k due to the fact that n is equal to the size of the ECC encoder output and k is the size of the encoder input. An ECC encoder always adds redundancy

bits so n will be larger than k. Thus the only way that v can be larger than $H_\infty(R)$ is if the entropy density, ρ, of the PUF response is extremely low. The entropy density takes into account all biases inherent to the PUF which reduce its entropy beyond just its size in bits. For the purposes of this evaluation, we assume that a PUF with high entropy density is used. This means that the size of the key will be the minimum value in the maximum entropy equations for each fuzzy extractor scheme.

With knowledge that the key size dictates the maximum entropy of the key for these scenarios, the BCH code tables can be examined for parameters which match the desired security and error correction levels. Specifically, the value of k should be the lowest value which is greater than or equal to the desired key size and capable of correcting the desired number of errors. To satisfy both of these requirements, larger values of n may need to be considered. For example, beginning with the 128 bit key requirement, the BCH code tables are inspected for parameters where the value of k is greater than or equal to 128. The first value of n which supports values of k greater than or equal to 128 is n=255. The first code with n=255 with a sufficiently large value of k is the *(255, 131, 37)* code. Now the code must be verified for error correction performance. For this code, its performance is 7.06%, which is sufficient. The public helper data sizes for this code are equal to n for the Key from PUF and Key from Secret constructions, 2*n for the Key from PUF, 2-Secret construction, and n+k for the Key from Secret, 2-Secret construction.

For the 192 bit key requirement, the process is repeated. The first BCH code which has a value of k greater than or equal to 192 is the *(255, 199, 15)* code. This code

only has an error correction performance of 2.75% so the process continues until a code with sufficient error correction is found. This occurs with the *(511, 193, 87)* code. This code has an error correction performance of 8.41% which satisfies the requirement.

For the 256 bit key requirement, the process is again repeated. The first code satisfying the entropy requirement is the *(511, 259, 61)* code. This code only has an error correction performance of 5.87% so the search continues. The first code with sufficient error correction performance is the *(1023, 443, 147)* code, which can correct 73 errors in a 1023 bit code word, or 7.14%.

With specific BCH code parameters assigned to each of the desired security levels it becomes obvious how the inflexibility of the BCH code affects the complexity of the system. Only certain combinations of *n, k,* and d_{min} values yield sufficient error correction performance with BCH codes. Combined with the requirement on the minimum value of *k*, this leads to code selections which sometimes have large output rates and large codeword sizes. This, in turn, leads to large public helper data requirements. These values will be the baseline against which our candidate ECC will be compared.

CHAPTER FIVE: CONVOLUTIONAL CODE ANALYSIS

The block codes described previously represent some of the strongest performing ECC which are practical for PUF applications. As was discussed, however, each has its own critical flaws. An additional disadvantage to block codes is that most encoders operate in systematic form. In a systematic encoder the message is first passed through unmodified followed by the parity bits. Even though most fuzzy extractor implementations mask the PUF response with random data, the most ideal case would be that the ECC encoder disguises its input rather than passing it through unmodified. Alternate generator matrices can be used which will make the output appear more random. However, when analyzing the security of these applications, it should be assumed that the encoder design is public knowledge. Therefore due to the fact that block codes are only based on the current input, an attacker with knowledge of the encoder design can still determine which output positions represent the locations of original input data.

Convolutional codes present a much more attractive solution. In terms of code parameter flexibility, the values of n, k, and K (constraint length, in the context of convolutional codes) can all be modified to find codes which provide sufficient error correction performance. In general, the larger the value of n and K, the higher the error correction capability. This also increases the hardware area requirements of the decoder,

though, so care must be taken to only increase these values to the point where error correction capability is satisfied. Also, larger values of *n* without a corresponding increase in the value of *k* will result in a very large output rate, which is to be avoided. The value for constraint length can be as small as 2, for an encoder that uses a single memory element, to as large as can be supported by the hardware. Experimentally, it was observed that codes with a constraint length less than 4 lack sufficient error correction capability to be useful for the PUF application. Codes with a constraint length larger than 10 were found to be too large to be practically implemented in hardware. Convolutional codes with higher values of *n* were avoided to keep the output rate as low as possible. Additionally, codes with a value of *k* larger than 1 were avoided because they do not allow for easy self-concatenation. A very attractive property of convolutional codes is that the same decoder can easily be used multiple times to achieve concatenation without an increase in area utilization as long as the value of *n* is divisible by *k*. The codes which allows this with the least amount of implementation complexity are those with rate 1/*n*. Any convolutional code can be realized in a systematic or non-systematic form. As mentioned previously, ideally the ECC should disguise its input, so the non-systematic form should be used. Additionally, any convolutional code can be realized with a feedback or feed-forward design. According to [13] the free distances of feed-forward designs exceed those of the feedback equivalents, so only feed-forward designs should be used. With these parameter constraints in mind, a field of 23 rate 1/2 convolutional codes with constraint lengths between 4 and 10 were selected from [13], [14], and [15] for performance evaluation.

Convolutional code tables typically describe a code using an octal representation of its generator polynomials. These values indicate which memory elements in the convolutional encoder contribute to each output bit. For example, the generator polynomial octal representations for the encoder depicted in Figure 8 below are 31 and 13. The top path, which generates output 1, has a binary representation of 11001, or 31 in octal. The bottom path, which generates output 2, has a binary representation of 10011, or 13 in octal. The constraint length, *K*, of this example encoder is 5.

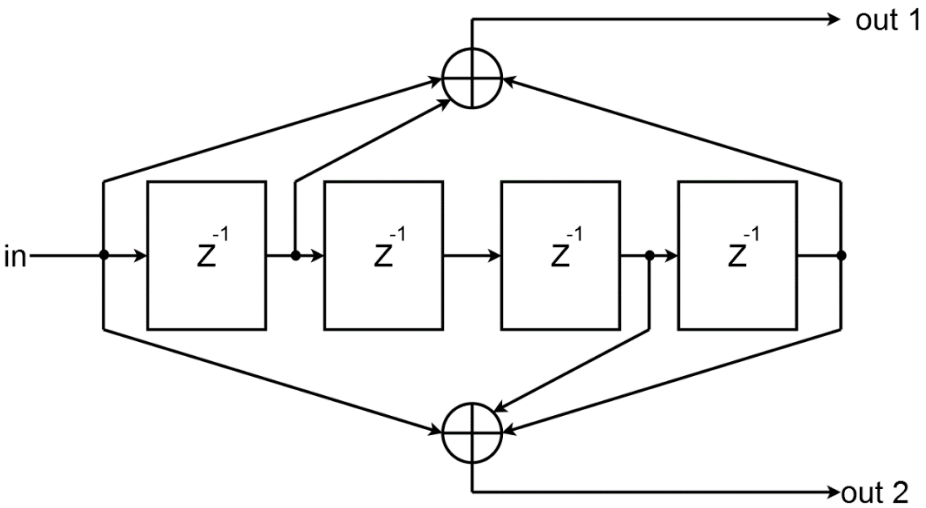

Figure 8 Example convolutional encoder with generator polynomials 31 and 13

The generator polynomials, G_1 and G_2, and constraint lengths of the 23 rate 1/2 convolutional codes under consideration are as follows.

Table 3 Rate 1/2 candidate convolutional codes

Code	*K*	G_1	G_2
C1	4	13	15

C2	4	13	17
C3	5	27	31
C4	5	23	35
C5	5	31	33
C6	6	53	75
C7	6	77	51
C8	6	77	45
C9	7	117	155
C10	7	163	135
C11	7	133	175
C12	7	634	564
C13	7	135	171
C14	8	247	371
C15	8	323	275
C16	8	626	572
C17	9	561	753
C18	9	457	755
C19	9	657	435
C20	9	751	557
C21	10	1131	1537
C22	10	1337	1475
C23	10	7664	5714

Software simulation

In order to evaluate the performance of each code, four software applications were used. The first was a C++ library originally developed by Lyppens [9]. This software allows for any convolutional code to be modeled by specifying the generator polynomials as well as the code parameters, *n*, *k*, and *K*. In addition to performing encoding and decoding, this library allows for inspection of any configured code's free distance property.

For this research, the Lyppens convolutional code library was augmented to also compute the error correction window. That is, the number of bits output during the free distance path. This metric indicates the span of bits over which the code can correct its

maximum number of errors, *t*. The application was also modified to be built for Linux, which involved removing references to Windows-specific system calls that ultimately were not necessary for proper operation. The C++ software was compiled using GCC on Ubuntu Linux.

The second software set is a collection of Python modules developed specifically for this research. This software tests each code for correctability of an error pattern based on the free distance and error correction window size derived using the first software application. The purpose of this software is to estimate the performance of a particular code based on its parameters. The reason for estimating rather than actually performing the encoding and decoding operations is because of the time needed to perform actual decoding in software. The estimation program runs in a fraction of the time needed to run actual decoding.

The logic of the code performance estimation core routine is as follows. An error vector evaluated which consists of a series of 1's and 0's. A 1 is treated as an error value and a 0 as a correct value. The error vector is iterated over one bit at a time. If an error value is encountered the code looks forward in the vector to see if, over the span of bits dictated by the error correction window parameter, the total number of errors exceeds the error correction capability. If it does, then the simulated decoding is considered a failure. If it does not, then the code jumps forward to the end of the error correction window and continues evaluating the remainder of the error vector. If no error correction windows contain more than the number of errors correctable as dictated by the code parameters, then the decode simulation is considered successful. This core capability was further built

upon to allow for experimentation and plotting of results. In addition to the suite of built-in libraries, NumPy, and Matplotlib were also used in the development of these utilities.

The third software set was another set of Python modules developed for this research. This software used the CommPy toolkit [20] to test actual encoding and decoding. This software was used instead of the Lyppens C++ library due to the author's preference of the Python programming language for rapid code iteration. Additionally, operations such as parsing comma-separated value (CSV) files are much more natural using Python, due to a rich set of built-in support libraries.

The final software set used in this analysis was a modification of the previous Python application for testing concatenated codes. The CommPy toolkit is again used for this software. In order to adapt the test for concatenated coding, the same trellis and generator polynomials are used twice consecutively on both encoding and decoding. Intermediary results are fed into the second stage of the concatenated scheme to produce the concatenated codeword. For all Python applications developed for this research, Python 2.7 in Ubuntu Linux was used as the interpreter.

Data Sources

A combination of simulated random data as well as real-world PUF responses were used in the evaluation of the candidate codes. The Python application to estimate code performance was run against both random input data as well as the real PUF responses. The Python application to perform true encoding and decoding was run against just the real PUF responses.

Random Data Generation

The random data patterns were produced by using the random number generator in the NumPy library. Using this library, a 1 was placed at random locations throughout the error pattern until a desired error density was achieved. Error density is defined as a percentage of the total codeword length. For placing errors, the error density value is cast as an integer to get a value that is in whole bits. For example, to generate an error density of 10% for a 256 bit codeword, 25 bits are set in the error vector. The random placement relies on the NumPy random function to produce sufficiently uniform distribution of values. If not, then the resulting error patterns will be biased towards certain positions in the vector and create a more burst-like error distribution. The documentation for the NumPy "random.randint" function indicates that the values returned are from a discrete uniform distribution. The random error pattern experiment does not necessarily reflect realistic error conditions. In reality, each PUF design will be biased slightly to produce a less-than-ideal level of uniformity. Also, as environmental factors fluctuate, different biases may appear in the generated PUF responses. This data set is intended to be used to provide a relative ranking of the candidate codes since all are evaluated using the same random data.

PUF Data

The PUF IDs used for real-world tests were derived from several sources, across multiple FPGA families. The first data set was collected by [16] using a ring-oscillator PUF design on Xilinx Spartan 3 XC3S500E FPGAs. The second dataset was collected by [10] using an SR-Latch PUF design on Xilinx Spartan 6 SC6SLX16 FPGAs. The final dataset was collected by [10] using an SR-Latch PUF on Xilinx Zynq XC7Z010 FPGAs.

Table 4 PUF Data Environmental Conditions

Device	PUF	Voltage (V)	Temperature (° C)	Number of devices	Number of measurements
XC3S500E	Ring-Oscillator	0.96	25	5	512
XC3S500E	Ring-Oscillator	1.08	25	5	512
XC3S500E	Ring-Oscillator	1.20*	25*	5	512
XC3S500E	Ring-Oscillator	1.20	35	5	512
XC3S500E	Ring-Oscillator	1.20	45	5	512
XC3S500E	Ring-Oscillator	1.20	55	5	512
XC3S500E	Ring-Oscillator	1.20	65	5	512
XC3S500E	Ring-Oscillator	1.32	25	5	512
XC3S500E	Ring-Oscillator	1.44	25	5	512
SC6SLX16	SR-Latch	1.14	25	5	256
SC6SLX16	SR-Latch	1.20	0	10	256
SC6SLX16	SR-Latch	1.20*	25*	15	256
SC6SLX16	SR-Latch	1.20	85	10	256
SC6SLX16	SR-Latch	1.26	25	5	256
XC7Z010	SR-Latch	0.95	25	5	256
XC7Z010	SR-Latch	1.00	0	6	256
XC7Z010	SR-Latch	1.00*	25*	9	256
XC7Z010	SR-Latch	1.00	85	6	256
XC7Z010	SR-Latch	1.05	25	5	256

Table 4 summarizes the environmental conditions under which each PUF measurement was taken. The nominal conditions for each device are indicated with an asterisk (*). For the XC3S500E and SC6SLX16 devices, this is 1.2 V at 25° C. For the XC7Z010 device, this is 1.0 V at 25° C. For the XC3S500E device, 512 Ring-Oscillator PUF instances were used to create the measurements. For the SC6SLX16 and XC7Z010 devices, 256 SR-Latch PUF instances were used to create the measurements.

In order to evaluate the PUF data using the two Python applications, the measurements first had to be converted into error vectors. This is a two-step process.

First, the raw measurements must be converted to PUF IDs. This was done by comparing two neighboring raw PUF measurements. If the second value is greater than the first, this is considered a 1 bit. Otherwise it is considered a 0 bit. For example, if the first measurement is a 1234 and the second is 2345, this evaluates to a 1 bit. If the first measurement is a 1234 and the second is 1200, this evaluates to a 0 bit. Using this technique, *N* measurements generate a PUF response of *N*-1 bits. There are many different approaches that can be taken to generate PUF IDs from raw measurements. This is a topic of ongoing research, for example in [10]. For the purposes of this thesis, only the neighboring-value method was used.

The second step is to compare the PUF IDs generated at non-nominal conditions to the PUF IDs generated at nominal conditions. If, at a given bit position, the non-nominal bit differs from the nominal bit, a 1 is placed in the error vector for that position. Otherwise a 0 is placed in the error vector for that position. Using this technique, 36 error vectors were derived from the XC3S500E data, 30 error vectors from the SC6SLX16 data, and 22 error vectors from the XC7Z010 data.

Experimentation

Code Performance Estimation

The first experiment focused on ranking the field of candidate convolutional codes based on their expected performance. To do this, the Python application which evaluates a code's ability to correct an error vector purely based on the code parameters was used. Each of the 23 candidate codes was evaluated at 15 error densities between 0% and 14% in 1% steps. At each error density level, each code was tested against 1024

random error vectors. If the random error vector did not contain more errors over the error correction window than the code allows for, then the test was considered a success. The success rate for a code at a given error density level is defined as the number of successes divided by the total number of trials, 1024. The success rates for each code at each error density level are plotted in Figure 9 below.

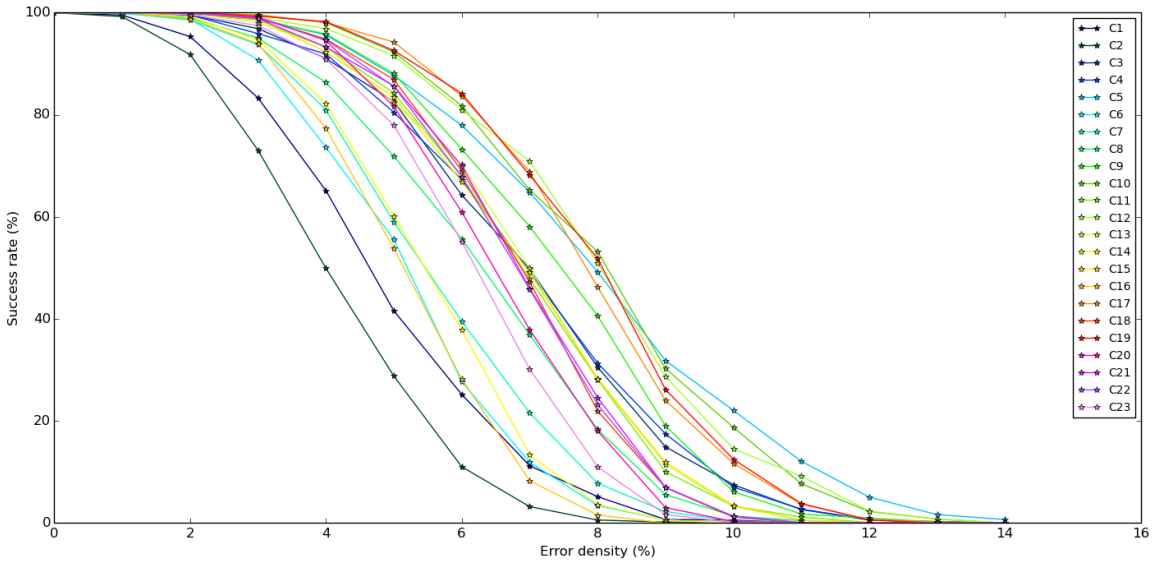

Figure 9 Candidate codes expected performance against 1024 random error patterns

With so many candidate codes in the plot above it can be difficult to discern the highest performing codes at the higher error densities. Figure 10 below illustrates the tail portion of this plot to more clearly see the performance of the codes at these high error densities. The number of errors present in this range is such that a burst error scenario is more often encountered. Since this is the most likely way that convolutional codes fail to

decode, it is worthwhile to consider how the codes perform in these situations relative to the others.

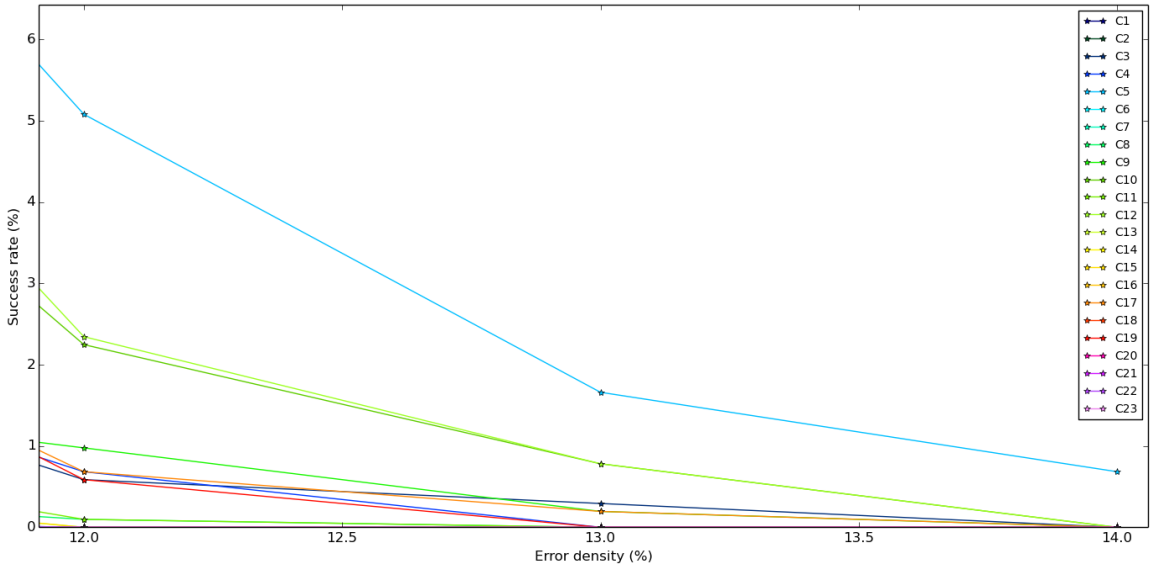

Figure 10 Candidate codes expected performance against 1024 random error patterns, zoomed

Based on estimated code performance, the top three performing codes in this range are C5, C12, and C10. This test merely ranks the codes relative to each other based on expected performance against random errors, though. To better characterize the expected behavior the experiment was repeated with real-world PUF data instead of random data.

Table 5 Candidate codes expected performance against PUF Data error vectors

Code	XC3S500E	SC6SLX16	XC7Z010	Overall
C1	42.5%	83.33%	54.55%	59%
C2	35%	70%	50%	51%
C3	52.5%	93.33%	86.36%	75%
C4	52.5%	93.33%	86.36%	75%
C5	65%	100%	90.91%	83%

C6	47.5%	86.67%	77.27%	68%
C7	50%	90%	81.82%	72%
C8	52.5%	93.33%	81.82%	74%
C9	70%	96.67%	95.45%	85%
C10	75%	96.67%	95.45%	88%
C11	62.5%	96.67%	95.45%	82%
C12	75%	96.67%	95.45%	88%
C13	62.5%	96.67%	95.45%	82%
C14	50%	93.33%	81.82%	73%
C15	62.5%	96.67%	95.45%	82%
C16	40%	93.33%	81.82%	69%
C17	70%	100%	95.45%	87%
C18	62.5%	100%	90.91%	82%
C19	70%	100%	95.45%	87%
C20	62.5%	100%	90.91%	82%
C21	62.5%	100%	90.91%	82%
C22	62.5%	100%	90.91%	82%
C23	62.5%	96.67%	86.36%	80%
BCH	80.0%	100%	100%	92%

Table 5 summarizes the estimated success rates of the 23 codes against real-world PUF data generated by the XC3S500E, SC6SLX16, and XC7Z010 devices. An overall estimated success rate for each code is included as well, which is the aggregate performance across each of the three devices weighted by the number of responses from each device. From this data, the four most performant codes are expected to be codes C10, C12, C17, and C19. The BCH code is also included for comparison. With the number of errors present in the Spartan 3 data, even the BCH code is unable to correct 100% of IDs.

Code Performance Using PUF Data

The estimated performance tests are useful because when analyzing a large number of codes in software, the time needed to run actual decoding becomes quite large

as the constraint length increases. The estimated performance can be used to pare down the field of candidate codes to only those which are expected to perform the best. There is no substitute for actually running the decoder, though, so that was the next test performed. Each of the candidate codes were run against the error vectors generated from real-world PUF data.

Table 6 Candidate codes performance against PUF Data error vectors

Code	XC3S500E	SC6SLX16	XC7Z010	Overall
C1	45%	83%	68%	64%
C2	50%	97%	73%	72%
C3	58%	93%	91%	78%
C4	60%	97%	91%	80%
C5	55%	97%	86%	77%
C6	65%	93%	91%	81%
C7	60%	97%	91%	80%
C8	63%	100%	95%	84%
C9	63%	100%	95%	84%
C10	70%	100%	91%	85%
C11	68%	100%	95%	86%
C12	58%	87%	77%	72%
C13	43%	90%	73%	66%
C14	73%	100%	95%	88%
C15	70%	97%	91%	84%
C16	75%	97%	95%	87%
C17	73%	100%	95%	88%
C18	75%	100%	95%	89%
C19	73%	100%	95%	88%
C20	73%	100%	100%	89%
C21	75%	100%	100%	90%
C22	75%	100%	100%	90%
C23	73%	100%	95%	88%

Table 6 summarizes the actual success rates of the 23 codes against real-world PUF data generated by the XC3S500E, SC6SLX16, and XC7Z010 devices. An overall

success rate for each code is included as well, which is the aggregate performance across each of the three devices weighted by the number of responses from each device. From this data, the four most performant codes are C18, C20, C21, and C22. With the exception of C18, each of these matches the performance of the BCH code against the Spartan 6 and Zynq PUF IDs. For the Spartan 3 PUF IDs, the BCH code outperforms each of the candidate codes.

The overall success rates of the estimated performance and the actual performance are compared in Figure 11.

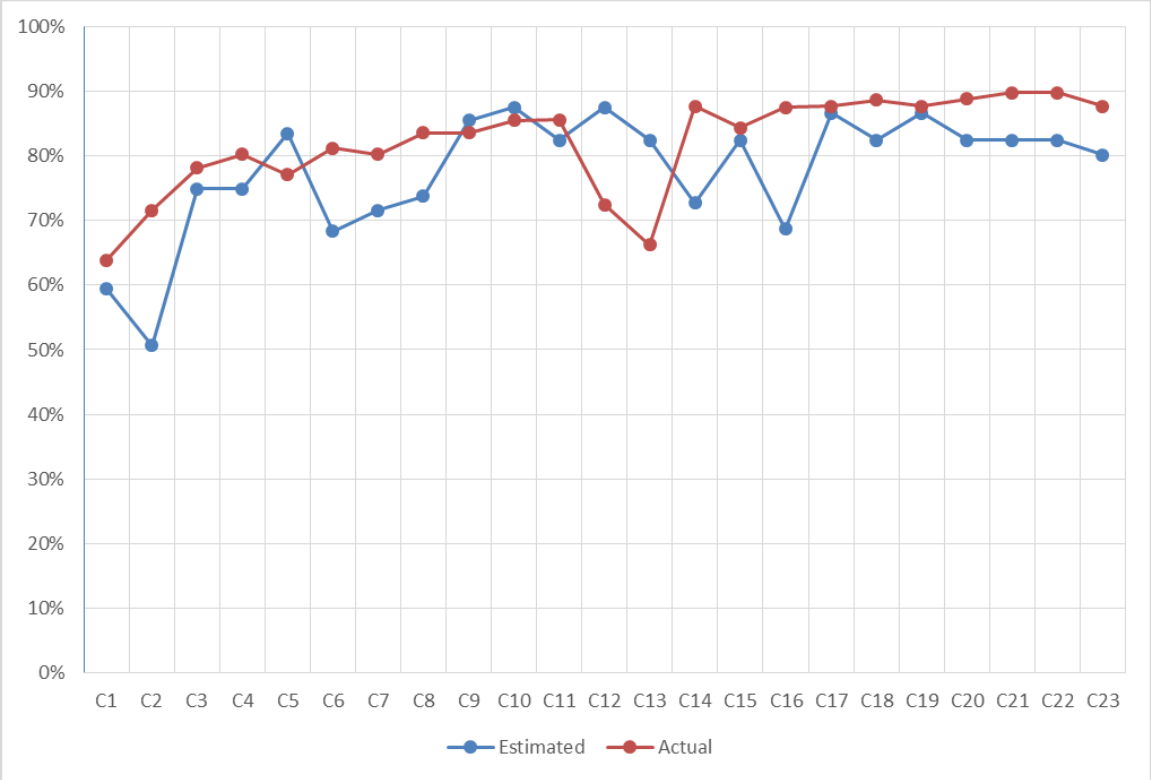

Figure 11 Estimated versus actual success rates for each candidate code

This comparison shows a bias in the estimation algorithm. The median difference between estimate and average is -5%. This indicates that the Python program to estimate performance based on code parameters tends to mark more trials as failures than an actual decoder would. This is likely due to two factors. First, the estimation program uses the *t* parameter as a hard limit on the number of errors which can be corrected. In reality, *t* is the number of errors which are guaranteed to be correctable. More than *t* errors may be corrected when the actual decoding takes place. Second, the algorithm used to determine failure may not be as reliable a model for the decoder's behavior as originally thought. Ultimately the measured success rates lack statistical significance. Each code was tested against just 88 PUF error vectors. Future work in this area could attempt decoding of orders of magnitude more real-world PUF error vectors to draw a more substantiated trend of the estimate and actual performance. Additionally, further research into a truer model of decoder performance is worthwhile. With a reliable model of code performance, one can save a large amount of time in software evaluation compared to actual decoding, especially at higher constraint lengths. This application can then be used to pare down a large field of candidate codes to only those which perform the best in simulations.

Concatenated Convolutional Codes

The next test performed was to concatenate each of the codes with itself in order to measure the performance improvement in decoding the real-world PUF data with a concatenated convolutional code. By concatenating each rate 1/2 code with itself, the resulting super-code will have an output rate of 4. The hardware area requirement for this construction is the same as for the non-concatenated case. This is due to using the same

code for both phases of the concatenation, which allows the same physical decoder to be used in implementation.

Table 7 Candidate codes self-concatenated performance against PUF Data error vectors

Code	XC3S500E	SC6SLX16	XC7Z010	Overall
C1	63%	97%	91%	81%
C2	63%	97%	95%	82%
C3	78%	97%	95%	89%
C4	75%	100%	100%	90%
C5	75%	100%	95%	89%
C6	73%	97%	91%	85%
C7	73%	100%	100%	89%
C8	75%	100%	100%	90%
C9	75%	100%	95%	89%
C10	70%	100%	95%	87%
C11	80%	100%	95%	91%
C12	80%	100%	100%	92%
C13	43%	90%	73%	66%
C14	78%	100%	100%	91%
C15	80%	100%	100%	92%
C16	83%	100%	100%	93%
C17	78%	100%	95%	90%
C18	88%	100%	100%	95%
C19	80%	100%	100%	92%
C20	83%	100%	100%	93%
C21	83%	100%	100%	93%
C22	83%	100%	100%	93%
C23	75%	100%	100%	90%

Table 7 summarizes the success rates of using each code in a self-concatenated construction. Although this approach doubles the output rate through the encoders, the number of codes able to correct 100% of error vectors from a given device more than doubles. Many of the self-concatenated codes meet or exceed the performance of the BCH code. Additionally, using such a construction allows a much smaller code to be

used, in terms of hardware area resources. The smallest code capable of correcting 100% of any device's error vectors in the non-concatenated case is C8, which has a constraint length of 6. Using a concatenated design, the smallest code capable of the same feat is C4, which has a constraint length of 5. The smallest code capable of matching the BCH code's performance when self-concatenated is C12. The smallest code capable of exceeding the BCH code's performance when self-concatenated is C16. The highest performing self-concatenated code is C18.

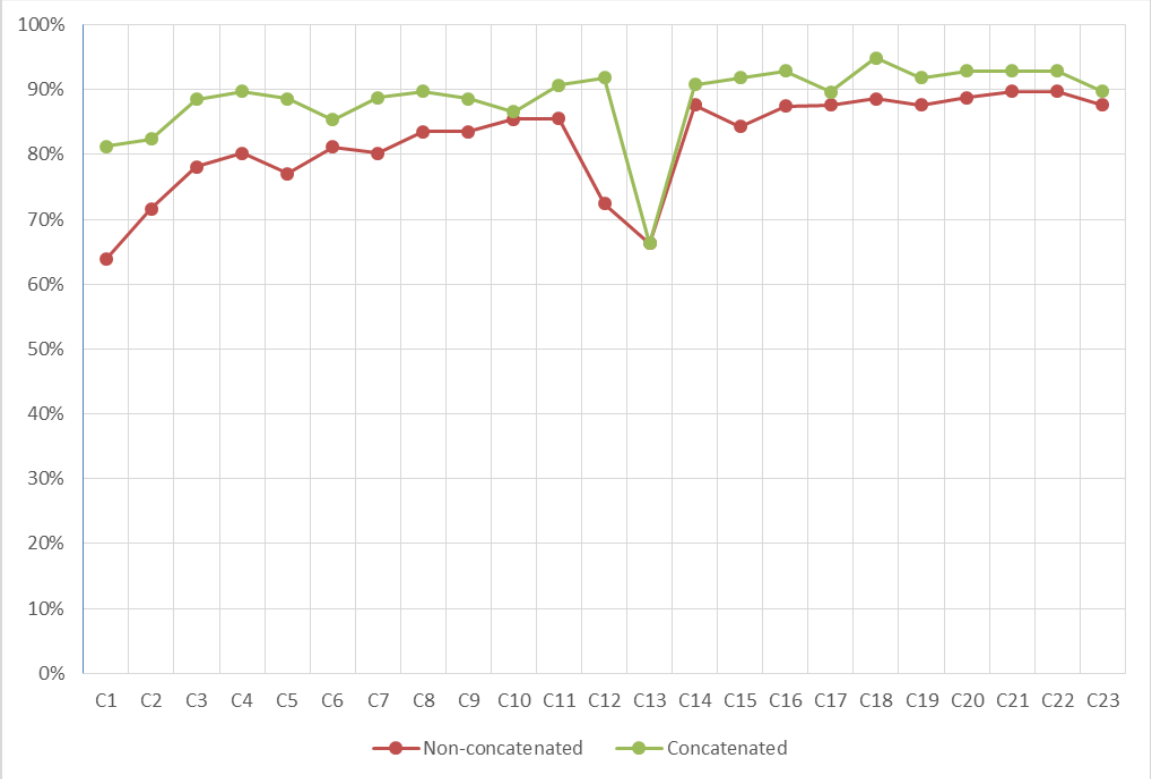

Figure 12 Non-concatenated versus concatenated success rates for each candidate code

Figure 12 illustrates the overall success rate improvement of the concatenated construction over the non-concatenated case. Only one code, C13, did not see any

improvement by using a concatenated design. The median success rate increase is 5%. Note that in this figure, none of the codes reach 100% overall due to the high error density of the Spartan 3 data.

CHAPTER SIX: IMPLEMENTATION OF ERROR CORRECTING CODES

In order to evaluate the hardware area requirements of the 23 candidate convolutional codes, the Creonic Viterbi Decoder was used [8]. The decoder was written in VHDL by Fehrenz and Alles. This decoder allows for simple configuration of any convolutional code. Additional parameters allow specification of whether a systematic encoder should be used as well as whether block RAM resources should be used. For these tests, the parameters were set to not use a systematic design and not use any block RAM resources. Each candidate code was configured by specifying the generator polynomial coefficients in the supporting VHDL files included with the package. The constraint length is inferred by the VHDL from the size of the generator polynomials specified. Synthesis and implementation of each candidate code was achieved using Xilinx ISE Design Suite 14.7.

Table 8 Candidate convolutional codes implemented for a Xilinx XC7A100T Artix 7 FPGA

Code	*K*	Slice Registers	Slice LUTs	Occupied Slices
C1	4	409	726	249
C2	4	409	726	249
C3	5	547	1095	401
C4	5	547	1095	401
C5	5	547	1095	401
C6	6	805	1747	578
C7	6	805	1747	578
C8	6	805	1747	578
C9	7	1320	3075	1012
C10	7	1320	3075	1012

C11	7	1320	3075	1012
C12	7	1320	3075	1012
C13	7	1320	3075	1012
C14	8	2345	5745	1808
C15	8	2345	5745	1808
C16	8	2345	5745	1808
C17	9	4651	12075	3601
C18	9	4651	12075	3601
C19	9	4651	12075	3601
C20	9	4651	12075	3601
C21	10	9108	23794	6465
C22	10	9108	23794	6465
C23	10	9108	23794	6465

Table 8 summarizes the area requirements when implemented for a Xilinx Artix 7 FPGA (XC7A100T). For a particular code implementation, the generator polynomials dictate the state transitions which must be evaluated by the decoder, however the constraint length dictates the area required by the decoder. This is why implementations of codes with the same constraint length have the same area utilization.

For comparison to BCH codes, a configurable C application presented by Jamro [17] was used. This software uses user-specified parameters to construct a BCH encoder and decoder. The application generates synthesizable VHDL for the encoder and decoder as well as a simulation circuit to test with artificially introduced errors. A combination of C algorithms and template files are used to accomplish this. The application was configured using an input file where the code parameters were specified. This C application and associated support files were used to generate VHDL for the BCH codes

needed by the fuzzy extractor schemes in Table 2. Using the Xilinx ISE Design Suite each of these codes was synthesized and implemented for the same Artix 7 part as before.

Table 9 BCH codes implemented for Artix 7 FPGA

BCH Code	Slice registers	Slice LUTs	Occupied Slices
(1023, 443, 147)	4284	13711	3578
(511, 241, 73)	1932	5546	1787
(255, 131, 37)	885	2628	1420

The resulting BCH area utilizations are presented in Table 9. Recall from Table 2 that in order to retain at least 256 bits of entropy through each of the fuzzy extractor constructions, the *(1023, 443, 147)* BCH code is needed. This code occupies 3578 total slices when implemented for the XC7A100T FPGA. To achieve this same level of entropy using a convolutional code, the C20 code could also be used. This code was able to correct 100% of SC6SLX16 and XC7Z010 PUF error vectors and 73% of XC3S500E PUF error vectors. In terms of area requirements, this code requires 3601 slices. If targeting the SC6SLX16 device specifically, code C8 is a more area-efficient choice. This code was able to correct 100% of error vectors from this device. When implemented for the XC7A100T FPGA, the code occupies just 578 slices.

Each of the candidate convolutional codes also have an advantage over the BCH codes in terms of output rate. Each of the codes tested have an output rate of 2. The *(1023, 443, 147)* BCH code has an output rate of 2.31. If larger output rates are acceptable, then the convolutional code approach can do even better with concatenation. Code C4 when self-concatenated is able to correct 100% of SC6SLX16 and XC7Z010

PUF error vectors and 75% of XC3S500E PUF error vectors. This code requires 401 slices when implemented for the XC7A00T FPGA, representing a reduction of 88.79% from the slice utilization of the BCH code.

CHAPTER SEVEN: CONCLUSION

In previous research on PUFs where an ECC is needed, the BCH code is often used due to its strong error correction properties. This code has a serious disadvantage due to the area required by its decoder at larger key sizes. This effect is especially apparent when used with fuzzy extractors where minimum entropy levels must be maintained. Additionally, the public helper data sizes for such fuzzy extractors are much larger when using BCH codes due to the need to use larger codes to guarantee certain entropy levels. This is largely due to inflexibility in the BCH code's n and k parameters when minimum levels of error correction performance must be met.

We propose using a rate 1/2 convolutional code in either an independent or concatenated construction for larger key sizes instead of the widely used BCH codes. These convolutional codes are capable of matching BCH code error correction performance for the majority of realistic PUF errors tested. Additionally, the use of certain convolutional codes results in a drastic reduction in required FPGA resources. For example, when using the concatenated *(2, 1, 5)* convolutional code, C4, only 11.21% of the resources required by the BCH code are needed.

The choice of which of these convolutional codes to select is a trade-off between output rate and hardware area utilization. The *(2, 1, 9)* convolutional code, C20, uses more area, however it only has an output rate of 2 versus the output rate of 2.31 incurred

by using the BCH code. The area requirements can also be much lower than that of the BCH, depending on the code and construction used. The concatenated *(2, 1, 5)* convolutional code, C4, uses significantly less hardware area resources at the expense of a doubling of its output rate to 4. The gain in hardware resource availability using this construction is remarkable, so if the output rate of 4 is acceptable then this design is the most marked area utilization improvement over the BCH code. Future work in this area would include analysis of rate 1/3 and 1/4 convolutional codes since their output rates are equal or less than that of the self-concatenated rate 1/2 codes.

The final advantage that these convolutional codes have over the BCH codes is that the error correction performance scales to larger key sizes without the need for additional hardware area resources. This research has focused on key sizes of 128, 192, and 256 bits, which are the standard key sizes for the AES encryption algorithm. There is nothing preventing larger key sizes from being used, such as the 2048 or 4096 bit keys used in the RSA cryptosystem. Such key sizes become prohibitively large for BCH codes to be used, since the area requirements increase along with key size. The only technique which would allow BCH codes to be used for such large codewords would be to use the aggregate of multiple smaller codes. As discussed earlier, this approach saves area but adds risk of failure since the overall ECC scheme becomes much more susceptible to burst errors. Future work in this area would include a comparison of an aggregate BCH code approach to the independent and concatenated convolutional code performance against codewords as large as 4096 bits.

REFERENCES

[1] K. Frikken, M. Blanton, and M. Atallah, "Robust authentication using physically unclonable functions," in ISC, Pisa, Italy, 2009, vol. 5735, pp.262-277.

[2] R. Maes, A. Van Herrewege, and I. Verbauwhede, "PUFKY: A fully functional PUF-based cryptographic key generator," in Proc. CHES, Leuven, Belgium, 2012, vol. 7428, pp. 302-319.

[3] C. Bosch, J. Guajardo, A.-R. Sadeghi, J. Shokrollahi, and P. Tuyls, "Efficient helper data key extractor on FPGAs," in Proc. CHES, Washington, D.C., 2008, vol. 5154, pp. 181-197.

[4] P. Koeberl, J. Li, A. Rajan and W. Wu, "Entropy loss in PUF-based key generation schemes: The repetition code pitfall," in Intl. Symp. HOST, Arlington, VA, 2014, pp. 44-49.

[5] V. van der Leest, B. Preneel, and E. van der Sluis, "Soft decision error correction for compact memory-based PUFs using a single enrollment," in Proc. CHES, Leuven, Belgium, 2012, vol. 7428, pp. 268-282.

[6] Y. Dodis, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," in Proc. EUROCRYPT, Interlaken, Switzerland, 2004, vol. 3027, pp. 523-540.

[7] G. D. Forney, Jr, "Concatenated codes," Ph.D. dissertation, Dept. Elect. Eng., MIT, Cambridge, MA. 1965.

[8] M. Fehrenz and M. Alles. (2012, Jan. 16). Viterbi decoder (AXI4-stream compliant). [Online]. Available: http://www.opencores.org/project,viterbi_decoder_axi4s

[9] H. Lyppens. (1997, Nov. 1). Convolutional error-control codes. [Online]. Available: http://www.drdobbs.com/cpp/convolutional-error-control-codes/184410317

[10] B. Habib, J. Kaps and K. Gaj, "Efficient SR-Latch PUF," in Proc. ARC, Bochum, Germany, 2015, vol. 9040, pp. 205-216.

[11] G.E. Suh, S. Devadas, “Physical unclonable functions for device authentication and secret key generation,” in Proc. DAC, San Diego, CA, 2007, pp. 9-14.

[12] J. Guajardo, S. S. Kumar, G. Schrijen, and P. Tuyls, “FPGA intrinsic PUFs and their use for IP protection,” in Proc. CHES, Vienna, Austria, 2007, vol. 4727, pp. 63–80.

[13] S. Lin, and D. Costello, Error Control Coding: Fundamentals and Applications. 2nd ed. Upper Saddle River, NJ: Pearson-Prentice Hall, 2004, pp. 66-95, 453-510, 540, 1231-1248.

[14] J. Cioffi, “System Design with Codes,” in Digital Communication: Signal Processing. [Online]. Available: http://web.stanford.edu/group/cioffi/doc/book/chap10.pdf

[15] R. Johannesson and P. Stahl, “New rate 1/2, 1/3, and 1/4 binary convolutional encoders with an optimum distance profile,” IEEE Trans. Inform. Theory, vol. 45, no. 5, pp. 1653-1658, 1999.

[16] PUF Artifacts. (2011). [Online]. Available: http://rijndael.ece.vt.edu/puf/artifacts.html

[17] E. Jamro. “The Design of a VHDL Based Synthesis Tool for BCH Codecs,” M. Phil. Thesis, Univ. of Huddersfield, Huddersfield, England, 1997. [Online]. Available: http://home.agh.edu.pl/~jamro/bch_thesis/bch_thesis.html

[18] Viterbi Decoding of Convolutional Codes. (2010, Oct. 6). MIT. [Online]. Available: http://web.mit.edu/6.02/www/f2010/handouts/lectures/L9.pdf

[19] S. Sarangi and S. Banarjee, “Efficient Hardware Implementation of Encoder and Decoder for Golay Code,” IEEE Trans. VLSI Systems, vol. 23, no. 9, pp. 1965-1968, 2015.

[20] V. Taranalli. (2015). CommPy: Digital Communication with Python, version 0.3.0. [Online]. Available: http://veeresht.github.io/CommPy

[21] R. Maes and I. Verbauwhede, “Physically Unclonable Functions: A study on the state of the art and future research directions,” Towards Hardware-Intrinsic Security, pp. 3-37, 2010, Springer Berlin Heidelberg.

BIOGRAPHY

Brian Jarvis graduated from Cape Henry Collegiate School, Virginia Beach, Virginia, in 2002. He received his Bachelor of Engineering in Computer Engineering from Vanderbilt University, graduating *magna cum laude* in 2006. For six years he was employed as an embedded software engineer by Argon ST, a defense contractor in Fairfax, Virginia, which is now a subsidiary of The Boeing Company. He was subsequently employed as a software engineer for Azure Summit Technology, a defense contractor in Oakton, Virginia. During his tenure at Azure Summit Technology he began his graduate studies at George Mason University. He is currently employed as a software development engineer at Amazon Web Services in Herndon, Virginia.