November 2016

Variation Aware Placement for Efficient Key Generation using Physically Unclonable Functions in Reconfigurable Systems

Shrikant S. Vyas
University of Massachusetts Amherst

VARIATION AWARE PLACEMENT FOR EFFICIENT KEY GENERATION USING PHYSICALLY UNCLONABLE FUNCTIONS IN RECONFIGURABLE SYSTEMS

A Thesis Presented

by

SHRIKANT VYAS

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2016

Electrical and Computer Engineering

VARIATION AWARE PLACEMENT FOR EFFICIENT KEY GENERATION USING PHYSICALLY UNCLONABLE FUNCTIONS IN RECONFIGURABLE SYSTEMS

A Thesis Presented

by

SHRIKANT VYAS

Approved as to style and content by:

Russell Tessier, Co-chair

Daniel Holcomb, Co-chair

Wayne Burleson, Member

Christopher V. Hollot, Department Chair
Electrical and Computer Engineering

ACKNOWLEDGMENTS

Thanks to Professor Tessier and Professor Daniel Holcomb for their guidance on this thesis document. I would also like to thank Naveen Dumpala for his expertise in the system design and Aftab Usmani for his design in the key generation section.

ABSTRACT

VARIATION AWARE PLACEMENT FOR EFFICIENT KEY GENERATION USING PHYSICALLY UNCLONABLE FUNCTIONS IN RECONFIGURABLE SYSTEMS

SEPTEMBER 2016

SHRIKANT VYAS

B.Tech., NMIMS UNIVERSITY

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Russell Tessier and Professor Daniel Holcomb

With the importance of data security at its peak today, many reconfigurable systems are used to provide security. This protection is often provided by FPGA-based encrypt/decrypt cores secured with secret keys. Physical unclonable functions (PUFs) use random manufacturing variations to generate outputs that can be used in keys. These outputs are specific to a chip and can be used to create device-tied secret keys. Due to reliability issues with PUFs, key generation with PUFs typically requires error correction techniques. This can result in substantial hardware costs. Thus, the total cost of a n-bit key far exceeds just the cost of producing n bits of PUF output.

To tackle this problem, we propose the use of variation aware intra-FPGA PUF placement to reduce the area cost of PUF-based keys on FPGAs. We show that placing PUF instances according to the random variations of each chip instance reduces

the bit error rate of the PUFs and the overall resources required to generate the key. Our approach has been demonstrated on a Xilinx Zynq-7000 programmable SoC using FPGA specific PUFs with code-offset error correction based on BCH codes. The approach is applicable to any PUF-based system implemented in reconfigurable logic.

To evaluate our approach, we first analyze the key metrics of a PUF - reliability and uniqueness. Reliability is related to bit error rate, an important parameter with respect to error correction. In order to generate reliable results from the PUFs, a total of four ZedBoards containing FPGAs are used in our approach. We quantify the effectiveness of our approach by implementing the same key generation scheme using variation-aware and default placement, and show the resources saved by our approach.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Trends

A novel approach for device authentication and identification of electronic devices has emerged over the past few years. Physical unclonable functions or PUFs can extract unique secret information from the physical characteristics of a device using a challenge and response procedure. This method for device authentication which is based on physical characteristics is extremely hard or impossible to reproduce [3].

Field-programmable gate arrays (FPGAs) are used for an increasingly large number of applications which require security. Bitstream encryption and secure encrypt/ decrypt cores which are implemented with the user's design are often used to protect FPGAs. These cores require secret keys that are often customized on a per device basis. PUF-based keys are uniquely tied to each device and are generally safe from side-channel attacks. Although the logic needed to create PUFs is modest, the amount of circuitry needed to create consistent keys repeatedly can be significant. We tackle this problem in our work to show a reduction in the hardware costs. The work described in this thesis is applicable to any PUF-based FPGA key implementation. The specific contributions of this thesis are as follows:

- We analyze the spatial randomness of unreliable PUF instances across multiple FPGAs and propose a novel system of per-device configuration to generate cryptographic keys. This spatially-aware key generation helps reduce the implementation costs of the entire system.

- We implement an FPGA-based PUF which was previously targeted to a Virtex 5 architecture to a more contemporary Xilinx Virtex 7 architecture [2]. We quantify the PUF's uniqueness and reliability in the new device architecture.
- We target our approach to a data-processing applications which require security keys. This analysis strengthens our claim of cost reduction in terms of area.

1.2 Thesis Overview

In this thesis we propose a novel approach for device specific placement of PUFs based on device-level reliability. PUFs are needed in applications requiring high security. However, the security of PUF-based keys comes at a high hardware cost. This hardware cost is primarily due to the unreliability of PUF outputs which require error correcting codes to detect and correct errors in the PUF outputs. Highly unreliable PUFs produce an output which requires more error correction thereby increasing the area for error correction compared to more reliable PUFs. In this work, we analyze all possible PUF locations on a chip in terms of their reliability. The idea behind this approach is to use the most reliable PUFs on a chip to reduce the size of error correction hardware and the overall required area of the key generation circuitry.

1.3 Thesis Outline

In Chapter 2, we review different types of PUFs that have been developed and implemented on FPGAs. Chapter 3 covers the design of the PUF used in this work. Detailed analysis about the uniqueness and reliability of PUFs is covered. Chapter 4 focuses on the core idea behind the research. The advantages of performing device specific placement of the PUFs is discussed. Also, a detailed description of error correcting codes is covered and our system implementation with encryption cores is presented. Insights into the keys generated by well-positioned PUFs are also explored. Chapter 5 discusses a new model for BCH code generation. The codes are used

for the PUF data set. Chapter 6 discusses an approach to obtain the benefits of variation-aware placement without performing device-specific placement. Chapter 7 summarizes the thesis work and offers directions for future work.

CHAPTER 2

BACKGROUND

Physical unclonable functions (PUFs) produce chip specific signatures at runtime. Different types of PUFs have been designed and implemented in FPGAs. In this chapter, background is provided about various types of PUF implementations, primarily the Arbiter PUF [23], Butterfly PUF [14] and the Ring Oscillator PUF [3].

2.1 Arbiter PUF

An Arbiter PUF, like other PUFs, produces its output due to the process variations on a chip which lead to different delays on two identical paths. Figure 2.1 illustrates an Arbiter PUF. The circuit consists of a pair of symmetric interconnects and measures any delay mismatch that may occur on the paths. The delay difference between the two paths is not fixed beforehand. This difference forms the crux behind Arbiter PUF operation. Figure 2.1 illustrates a PUF delay circuit based on MUXes and an arbiter which is primarily an edge-triggered D-Flip flop. The key generated by the Arbiter PUF is based on a challenge which produces a specific response. The circuit has a multiple-bit input X and a 1-bit output Y based on the relative delay difference between two paths with the same layout length [18]. The output of the design is 1 if the input to the D port of the flip flop have a smaller delay and 0 otherwise.

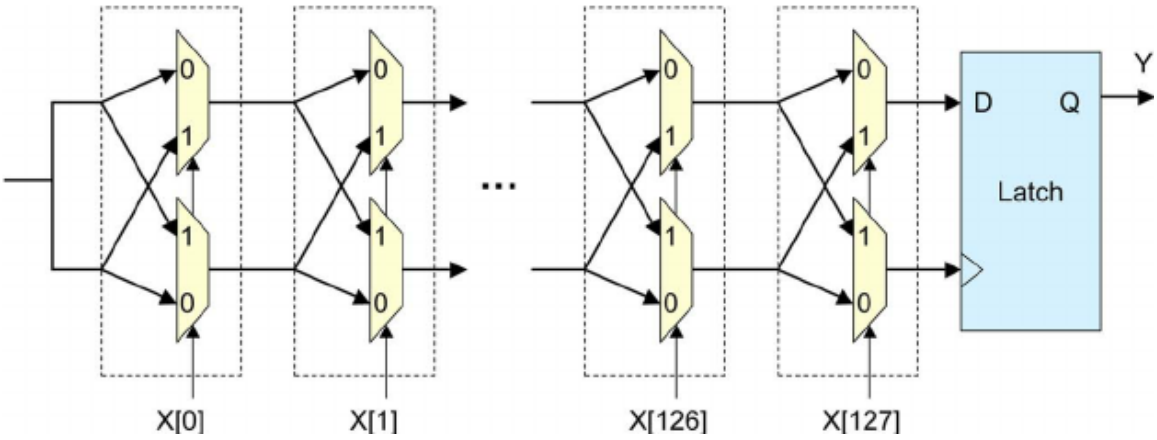


Figure 2.1: Arbiter PUF [23]

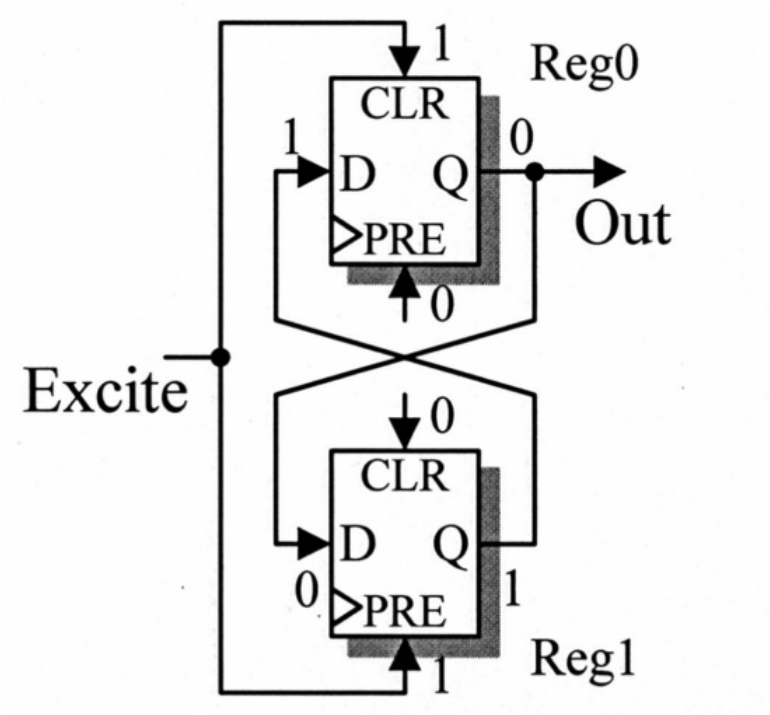


Figure 2.2: Butterfly PUF [14]

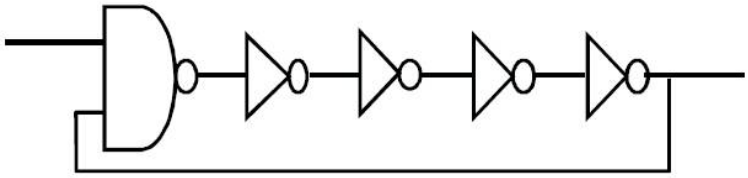


Figure 2.3: Ring Oscillator

2.2 Butterfly PUF

The Butterfly PUF tries to match the startup behavior of an SRAM cell [14]. The structure of the Butterfly PUF (BPUF) is shown in Figure 2.2. The PUF consists of a cross coupled combinational loop using latches created in the FPGA logic. The latches contain preset and clear signals. An excite signal triggers the preset signal of one latch and the clear signal of the other. The BPUF works by bringing the design to an unstable state using the excite signal and allowing the circuit to settle to one of the two stable states that are possible.

The BPUF reaches an unstable state due to the cross coupling of the outputs. When this signal is made low after a few clock cycles, the BPUF starts to attain a stable state. This state depends on the differences in the delays of the symmetric paths which are imparted during manufacturing.[14].

2.3 Ring Oscillator PUF

The Ring Oscillator PUF (ROPUF) is based on delay loops (ring oscillators) and counters rather than arbiters or cross coupled latches. A ROPUF, shown in Figure 2.3, consists of N similar ring oscillators, two counters, two N-bit multiplexers and a comparator [19]. Every ring oscillator oscillates at a different frequency due to process variations. By counting the number of oscillations of a pair of PUFs selected using the challenge as inputs to the select line of the multiplexers, the ROPUF produces a 0 or a 1 based on the output of the comparator.

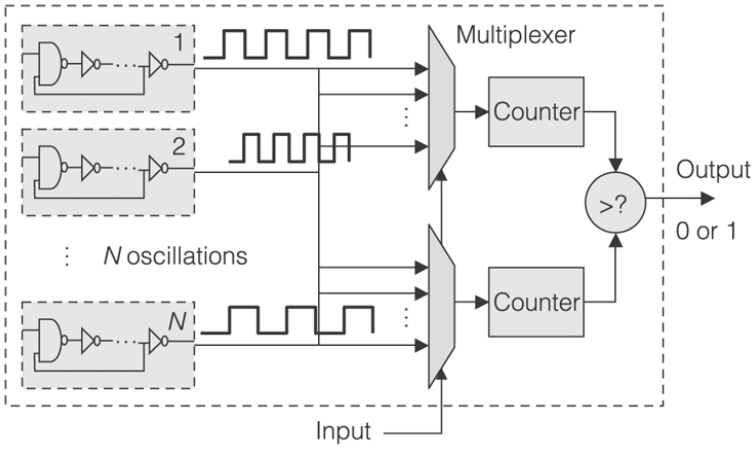


Figure 2.4: Ring Oscillator PUF using multiple ring oscillators and a counter [24]

2.4 Analysis of Delay based PUFs

According to [18], every path consists of two delay components: a static delay component and a random delay component which is present due to process variations. Ideally, the PUF output should only be dependent on its process variation. Hence, out of the two components, for a PUF, the random delay component should be the significant factor. Arbiter, Butterfly and Ring Oscillator PUFs produce results on the assumption that the static delay involved in the symmetrical paths cancel out [14].

$$d_N = d_S + d_R \tag{2.1}$$

In Equation 2.1 from [18], d_S and d_R refer to the static delay component and the random delay components of a path, respectively. The delay differences between two paths can be expressed by Equation 2.2 [18]. This equation suggests that the delay difference between two paths is primarily the sum of the difference of the individual components.

$$\Delta d = d_{S1} - d_{S2} + d_{R1} - d_{R2} = \Delta d_S + \Delta d_R \tag{2.2}$$

In an ideal case, the static delay differences Δd_S would tend to 0 and the delay comparison between the two net delays would be a function of the random delay component. Even a slight contribution by the static delay component can result in a biased PUF output. If $d_S > d_R$, then the effect of random variation on the output will be insignificant and the bits generated by the PUF will be biased.

For an Arbiter PUF, timing analysis results performed by [18] indicate that the difference in the static delays between the two paths is much higher than expected and overshadows the difference in delays due to random variations. The primary reason for such an observation is that the net routing to a clock input of a flip flop requires sending the signal through multiple additional components to reach the clock port whereas the route to the D input of the flip flop is comparatively simple. The results produced by [18] show that there is a difference in factor of almost 12 between the two delay components due to such routing.

For a Butterfly PUF, it can be safely assumed that the two latches used in the design are identical. However, the major issue observed is the symmetry of the interconnection nets. Similar to the arbiter PUF, the timing analysis performed on the Butterfly PUF by [18] showed that Δd_S is an order of magnitude higher than Δd_R.

Ring Oscillator PUFs have the same requirement of symmetric routing as the prior two PUFs in order to keep the various ring oscillators in the design to be identical. However, they do not suffer the same drawbacks as the Arbiter and Butterfly PUFs. According to [23], the RO PUFs are easier to implement in FPGAs. However, they are slower, larger and consume more power than Arbiter PUFs. This Ring Oscillator PUF disadvantage, which results in a higher number of elements needed to generate an output bit, proves to be of significance in our research. The goal of our work is to reduce the area consumed by the PUFs and error correcting codes. Hence, a Ring Oscillator PUF, in spite of being highly reliable, does not suit our requirements.

CHAPTER 3

A PHYSICAL UNCLONABLE FUNCTION NATIVE TO THE XILINX ARCHITECTURE

In this chapter, a detailed operational description of the *Anderson* PUF [2] used in this work is provided. This PUF has been shown to work effectively in Xilinx FPGAs. The PUF overcomes the drawbacks of the delay-based PUFs described in Chapter 2 by avoiding the need for careful symmetric routing. The PUF uses internal component connections with fixed delays inside logic clusters. The primary advantage of using Anderson PUFs for our work is the lack of design dependence on delay variations due to *programmable* interconnect.

This chapter explains our analysis of the PUF in terms of reliability and uniqueness on a Virtex 7 chip. Detailed analysis of the circuit output shows a between class (across chip) Hamming distance of approximately 62 for a 128-bit output generation and an average within class (same chip) distance of 5.59. To understand the correlation of the PUFs on separate chips, the Pearson coefficient [25] for all chip pairings is computed. The coefficient for all pairings falls between -0.035 and 0.040, indicating highly uncorrelated output bits.

3.1 Xilinx Virtex 7 Architecture

A Configuration Logic Block (CLB) in the Xilinx Virtex 7 architecture is comprised of two logic slices and each slice consists of a combination of lookup tables (LUTs) and registers (flip flops). CLBs are arranged in a two-dimensional array on the FPGA chip and are connected to each other through a programmable interconnection matrix. The LUTs in a slice can be configured to implement any logic function

or, in some cases, to serve as a small memory. As we will describe in the next section, our design uses shift registers configured from LUTs which implement the memories. About 25% of the LUTs in the Virtex 7 architecture can be implemented as memories. These LUTs are present in the slices termed SLICEM where 'M' indicates memory [2].

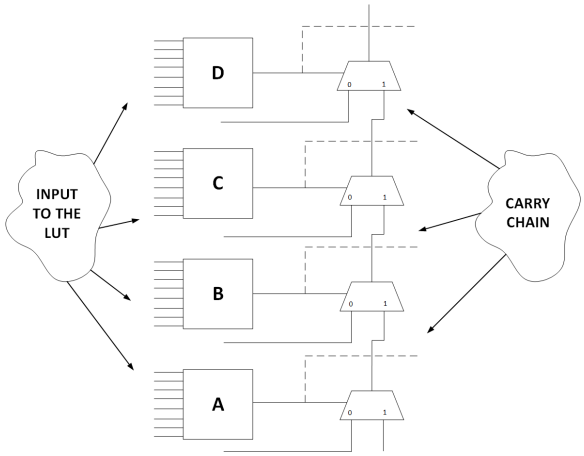


Figure 3.1: SLICE Details

Figure 3.1 shows the Virtex 7 CLB structure of a SLICEM which is of interest. It consists of MUXs connected as a carry chain and 4 LUTs whose outputs act as the select lines of the multiplexers. Each multiplexer receives one of its inputs from the output of the multiplexer directly below it. The output of the top multiplexer can either be used as an input to the multiplexer in the next slice above the current one or can be directed to a flip flop. In our design, both possibilities have been explored.

3.2 Anderson PUF Design

As mentioned in Chapter 2, an Arbiter PUF or a Butterfly PUF can be difficult to implement in FPGAs. The PUF design proposed by Anderson produces an output which is more clearly based on random interconnect delay variations and does not suffer from the problems faced by the other PUFs which require the use of programmable interconnect in FPGAs.

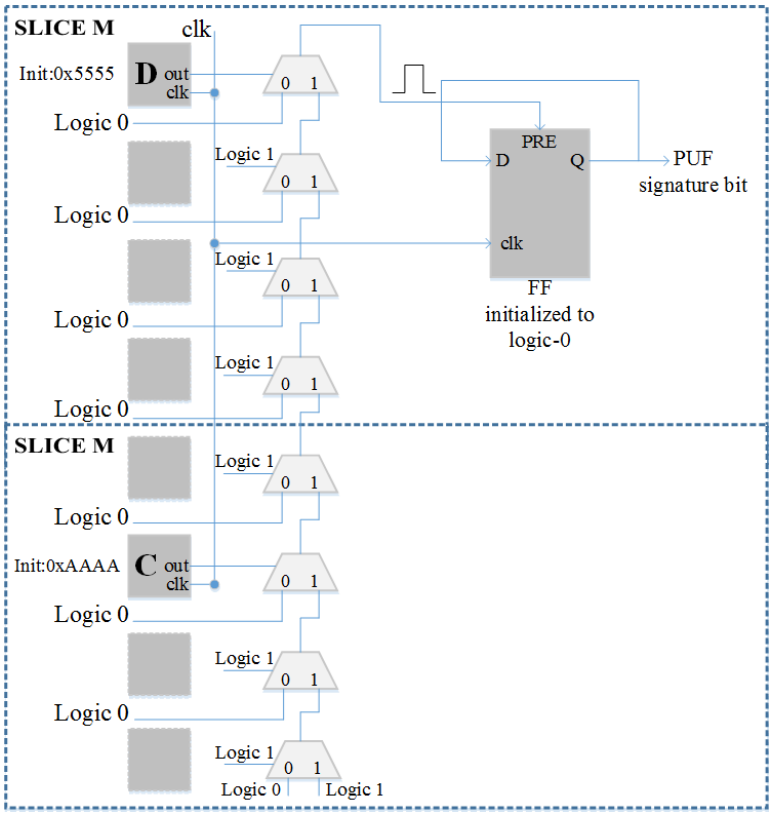


Figure 3.2: Anderson PUF design for Xilinx architectures [2]

Figure 3.2 shows the core of the PUF design. Two LUTs (D and C) are used to implement shift registers with alternating outputs of 0 and 1. The bits generated from the shift registers are applied to the select lines of the adjacent carry chain multiplexers. This connection between the LUTs and the carry chain is fixed and not programmable. The carry chain output is connected to a SLICEM flip flop. Figure 3.3 shows four multiplexers between the two shift registers. This gap is necessary to produce a signal delay wide enough to trigger the preset port of a flip flop if a glitch is generated during the shift register transitions. This flip flop captures the glitch produced by the carry chain and the shift registers.

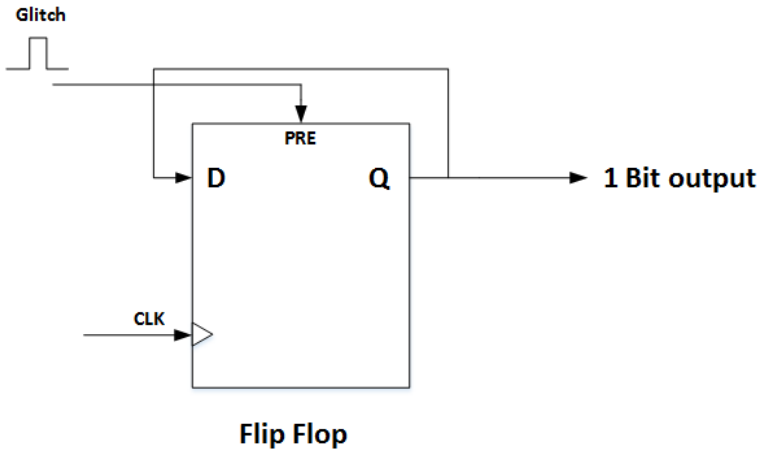


Figure 3.3: Flip flop used to capture carry chain glitch

3.3 Anderson PUF Operation

As mentioned in the previous subsection, two LUTs, D and C, are used in a 16-bit shift register mode. The shift registers need to be pre-initialized as follows:

LUT D: 0101010101010101

LUT C: 1010101010101010

We need to make sure that the initialization bitstrings of the two shift registers are complementary to each other. The shift register output drives the select line of the multiplexers in the carry chain. The "0" data inputs of all design multiplexers are tied to logic 0 while the bottom carry chain multiplexer has its "1" input tied to logic 1 [2].

Initially, the output of LUT D is logic 0 while that of LUT C is logic 1. The output of the top multiplexer is at logic 0 while the output of the bottom multiplexer is set to logic 1. At the next rising edge of clock signal, the output of the LUT D shift register transitions from 0 to 1 while the output of the LUT C shift register transitions from 1 to 0. Due to random process variations, the two transitions occur with different delays. This property is exploited for generating the PUF output. The case in which

the LUT D transition is slower than the LUT C does not alter the output from the previous state and hence the output of the top multiplexer will remain at logic 0.

If the output of LUT D transitions from 0 to 1 with a smaller delay when compared to the transition of LUT C from 1 to 0, a short positive glitch (spike) will appear on the top multiplexer until the LUT C transition from 1 to 0 reaches the mux. This glitch is used to determine the PUF bit which acts as the preset signal to a flip flop show in Figure 3.3. The flip flop is initialized to 0 and its output is fed back to its input. The width of the glitch signal needs to be sufficiently large to trigger the flip flop to change its output from 0 to 1. Care needs to be taken that the glitch is not always too wide or too narrow such that it causes the output of the flip flop to always transition to 1 or remain at 0, respectively. Hence, to create a meaningful PUF, the position of the bottom shift register should be considered.

In our experiments, we observed that the position of the top shift register should be in LUT D while the bottom shift register should be in LUT C. This gap provides a total of four multiplexers in between the multiplexers of the corresponding shift registers. Due to an increase in the number of multiplexers between the two registers, the width of the glitch can be increased to produce an unbiased output. The select lines of all the multiplexers in between are tied to 1 thereby propagating the signal generated by the bottom shift register.

3.4 Experimental Validation

A research goal is to determine if there are locations on the FPGA chip which are more favorable to PUF performance than others. PUF performance is defined by two primary factors, uniqueness and reliability.

For our analysis, we instantiated our PUF design at all possible locations in a target FPGA. We evaluated the design using four ZedBoards which include a Xilinx XC7Z020-1CLG484C Zynq-7000 AP SoC. Each board has approximately 4,200

SLICEM's and each PUF circuit uses two slices to generate a bit. Hence, a total of around 2100 bits can be generated. In this section, we describe the experiments used to analyze these parameters. We quantify the two properties by dividing all the PUF instances into blocks of 128 PUFs to generate 128 bits across the chip for all four chip instances. In total, we implement 16 disjoint 128-bit PUFs on each chip.

3.4.1 Uniqueness

PUFs are primarily used for secret key generation and device identification. Hence, the outputs of a PUF-based key must be able to identify a device uniquely. It is important that no two devices give similar responses. The difference between the responses of two devices can be formalized by their Hamming distance (HD) using Equation 3.1 from [6].

$$HD(R_i, R_j) = \sum_{t=1}^{n} r_{i,t} \bigoplus r_{j,t} \tag{3.1}$$

Here, $R_i = r_{i,1}, r_{i,2}...r_{i,n}$ and $R_j = r_{j,1}, r_{j,2}...r_{j,n}$ are the two responses from device i and j respectively for all bits n.

For a comparison between m devices, the average Hamming distance is called as *inter distance* or *between class* Hamming distance given by Equation 3.2 from [6]. Ideally, the between class Hamming distance is half the output size. In this case, half the bits between the two responses are different. Since we consider 128-bit outputs, the ideal inter Hamming distance in our case is 64.

$$HD_{inter} = \frac{1}{\binom{m}{2}} \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} HD(R_i, R_j) \tag{3.2}$$

To study uniqueness, we consider two different variants of between class Hamming distance in our analysis. In the first case, we analyze the Hamming distance between the outputs from two different randomly selected 128 bit PUFs. Comparisons of two different PUFs from the same chip and on different chips were performed at

random. Over 10,000 comparisons, the mean distance obtained was 63.60. This value is close to the expected ideal value of 64. The second case of between class Hamming distances were confined to only compare PUF pairings that occupy the same locations on different chips. This case could show a reduced Hamming distance if PUF output values were significantly influenced by deterministic bias instead of device-specific process variations. A mean Hamming distance of 61.22 was obtained, indicating that the implemented PUFs produce highly unique outputs even when positioned at the same location on different chips.

3.4.2 Reliability

For any PUF circuit, it is of importance that the responses in each chip are within an acceptable error limit. Reliability refers to the repeatability of PUF outputs over time. This property is measured as the Hamming distance between several responses for the same device and location. The intra Hamming distance or within class Hamming distance over k trials and j responses is defined in Equation 3.3 from [6]. Our results are based on a total of 10000 trials and 1000 responses.

$$HD_{intra}(i) = \frac{1}{k-1} \sum_{l=2}^{j} HD(R_{i1}, R_{il}) \tag{3.3}$$

The ideal value of the within class Hamming distance is 0, however, slightly larger values are acceptable and error values can be rectified with the help of error correcting schemes. In our experiments, within class Hamming distance values were generated using two randomly selected output trials from the same 128-bit PUF. Over 10,000 within class comparisons with randomly selected PUFs trials and chips, an average distance of 5.59 was observed.

3.4.3 Constant Switching

In the previous experiments to validate the strength of the PUF design, no switching activity was considered in the evaluation. However, in a real application, the area

around the PUF might undergo constant switching. This can adversely affect the properties mentioned above. In order to verify the uniqueness and reliability of our design, we try to mimic this switching of real scenarios by placing toggle flip flops in the same CLB as the PUF. Specifically, a total of 5 toggle flip flops have been placed around each PUF in order to study the behavior of the PUF.

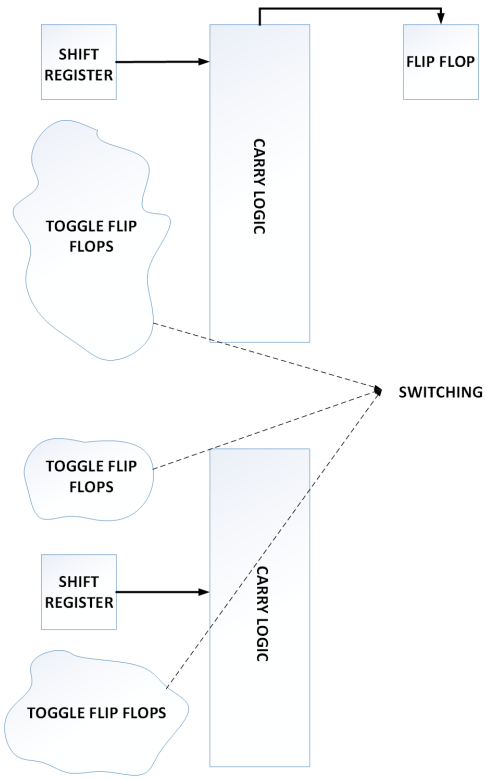


Figure 3.4: PUF design surrounded by Toggle Flip Flops

Figure 3.4 depicts our visualization of the switching around the design. We can see that our approach implements toggle flip flops in the LUTs above and below the shift registers of the PUF. Based on our analysis, the within class Hamming distance remained similar and we obtained an average distance of 5.29 over 10,000 comparisons. Similar experiments to compute the between class Hamming distance were performed on the same boards used in our previous analysis. The average between class hamming distance spanning over all locations across all chips over 10,000 comparisons was seen to be 62. This again is close to the ideal value of 64. On the other hand, the average

between class Hamming distance across the same locations between different chips over the same number of comparisons was around 58.63. By this analysis, we can conclude that the PUF produces a very unique and reliable output even in a practical scenario where switching takes place. It shows that the PUF design is not affected by other parts of the design which can be implemented alongside the PUF.

3.5 Results and Analysis

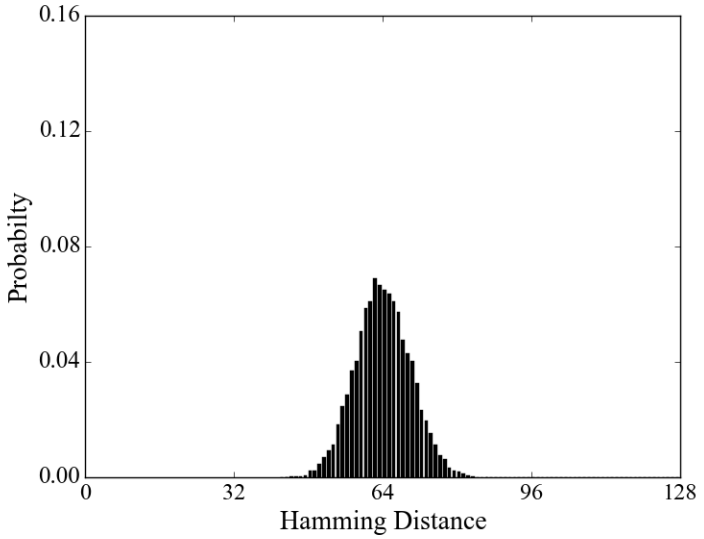


Figure 3.5: Between class Hamming distances for two different 128 bit PUFs

The between class Hamming distance obtained by comparing two randomly selected 128 bit PUFs from random chips and randomly selected output trials over 10,000 iterations is shown in Figure 3.5. The same analysis with the presence of toggle flip flops is shown in Figure 3.6.

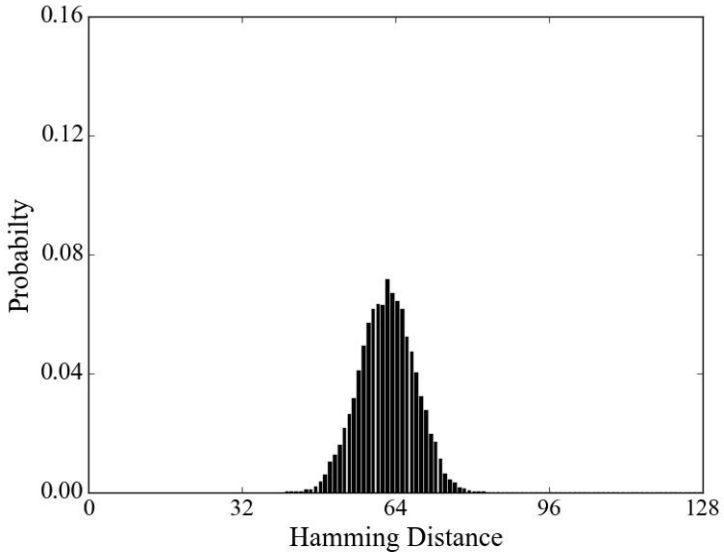


Figure 3.6: Between class Hamming distances for two different 128 bit PUFs with surrounding flip flops

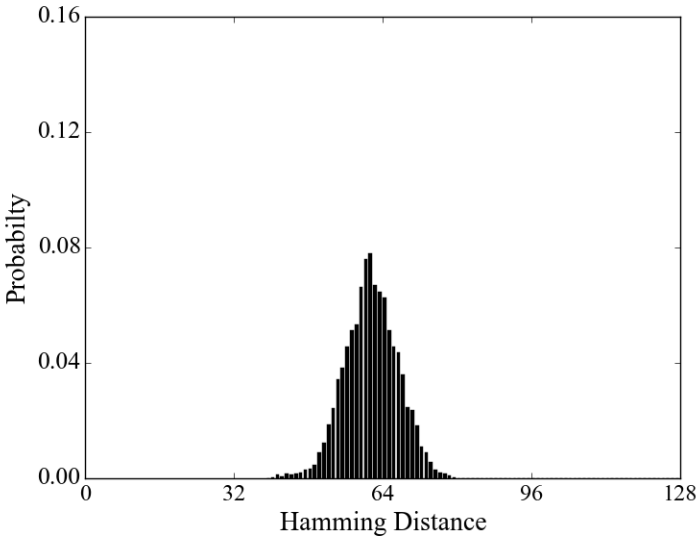


Figure 3.7: Between class Hamming distances for 128 bit PUFs that occupy the same locations

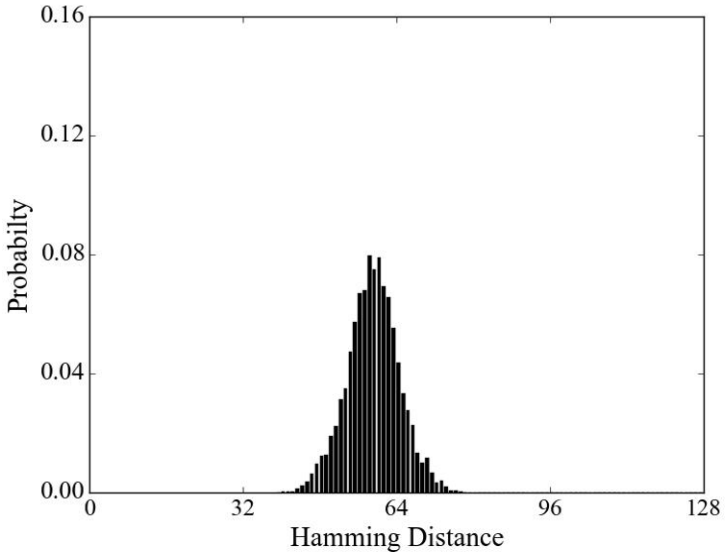


Figure 3.8: Between class Hamming distances for 128 bit PUFs that occupy the same locations with surrounding toggle flip flops

The between class Hamming distance obtained by comparing two 128 bit PUFs in the same locations from random chips and randomly selected output trials over 10,000 iterations is shown in Figure 3.7 while the result with constant switching is shown in Figure 3.8.

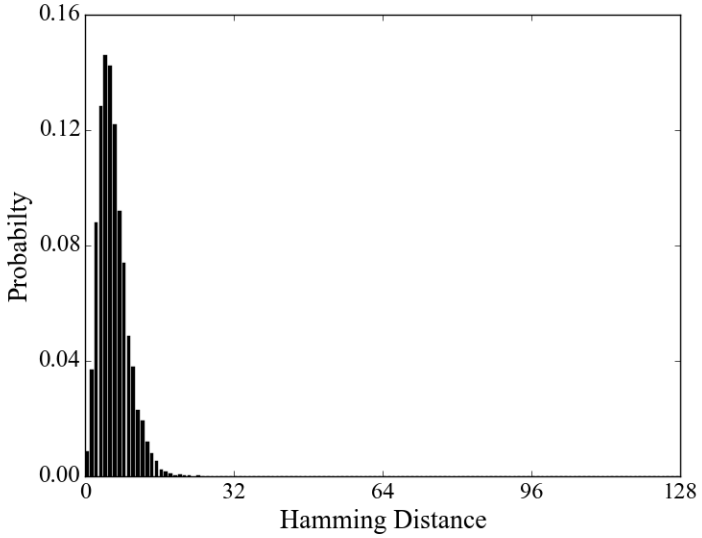


Figure 3.9: Within class Hamming distances between the same 128 bit PUF instances

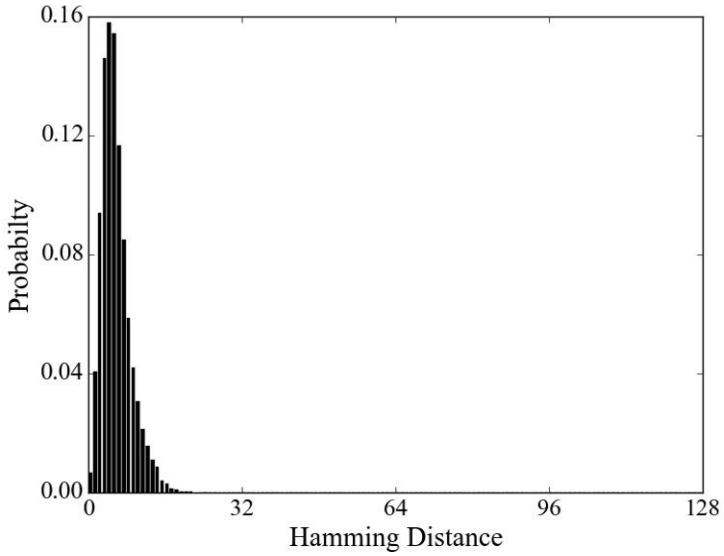


Figure 3.10: Within class Hamming distances between the same 128 bit PUF instances with surrounding toggle flops

The within class Hamming distance which compares the measurements from the same 128 bit PUF instance over time is shown in Figure 3.9 and Figure 3.10 which shows the results based on the design including switching around the PUFs. From

the experimental results, we can conclude that the PUF produces a unique output with almost 50 percent of the PUF bits being different. A low within class Hamming distance supports the reliability of the PUFs. Table 3.1 tabulates the results that we have obtained for the Hamming distances with the PUFs standalone as well as the PUF design with the toggle flip flops.

Table 3.1: Mean Within Class Hamming Distance & Between Class Hamming Distance obtained with standalone PUFs vs PUFs with Toggle Flip Flops

Design	Within Class Hamming Distance	Between Class Hamming Distance (all locations)	Between Class Hamming Distance (same locations)
PUF	5.59	63.60	61.22
PUF with Toggle Flip Flops	5.29	62	58.63

CHAPTER 4

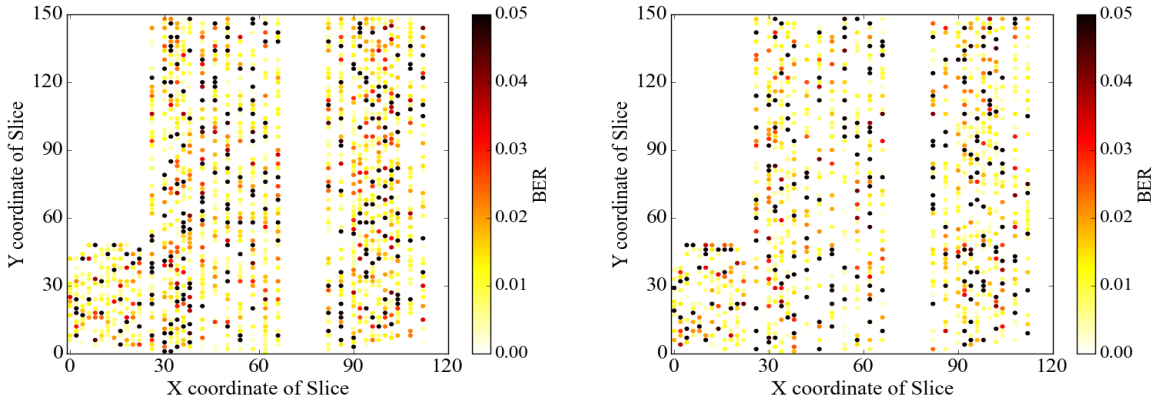
SYSTEM DESIGN

In this research, we propose the idea of per-device placement of PUFs. In this chapter, we provide evidence to support our approach by calculating the reliability of the PUFs on all the locations of a chip and observing the spatial correlation of the unreliable PUFs with respect to other chips. This chapter gives a detailed description about the error correcting codes used in our work along with a full system design to generate a key by utilizing encryption cores along with process variation dependent bits generated by the PUF corrected by error correcting codes.

4.1 Device Specific Location of Unreliable PUF Instances

The work in Chapter 3 quantified the uniqueness and reliability of our implementation of the Anderson PUF. In this section, we quantify key generation using the bit error rate (BER) metric which signifies the probability that a PUF will produce an incorrect output. We observe the BER of every PUF instantiated on the chip to select the most reliable PUFs or the PUFs with the lowest BER values. PUF selection based on BER directly relates to the size of the required error correcting code. This metric is related to hardware cost.

A PUF bit placed in a specific location produces an error when its output differs from what is expected. The BER of a location is the percentage of computation trials where a location produced an error. Figure 4.1 for a single chip shows the BER of 2,080 PUF instances according to their locations. These values were obtained using the bits produced by each location for 1,000 trials. Darker areas on the heat map



(a) Heatmap of Chip 1

(b) Heatmap of Chip 2

(c) Heatmap of Chip 3

(d) Heatmap of Chip 4

Figure 4.1: Figure shows the BER of PUF instances placed at different locations on each chip

correspond to higher BER (a more unreliable location) while lighter areas correspond to higher reliability. The lack of a clear pattern in the figure gives an indication that the unreliable PUFs are likely to be random.

Following the approach used to obtain the within class Hamming distance in the previous chapter, the BER can be obtained by dividing the number of incorrect output trials with the total number of trials. In Figure 4.2, the Y-axis denotes the percentage of the BERs of the PUFs while the X-axis denotes the number of most reliable PUFs out of 2,080 total PUF instances. From the figure we can observe that by using the

most reliable PUFs, we are faced with a smaller BER and hence fewer errors need to be corrected. This decreases the size of the error correcting code needed and results in substantial area savings.

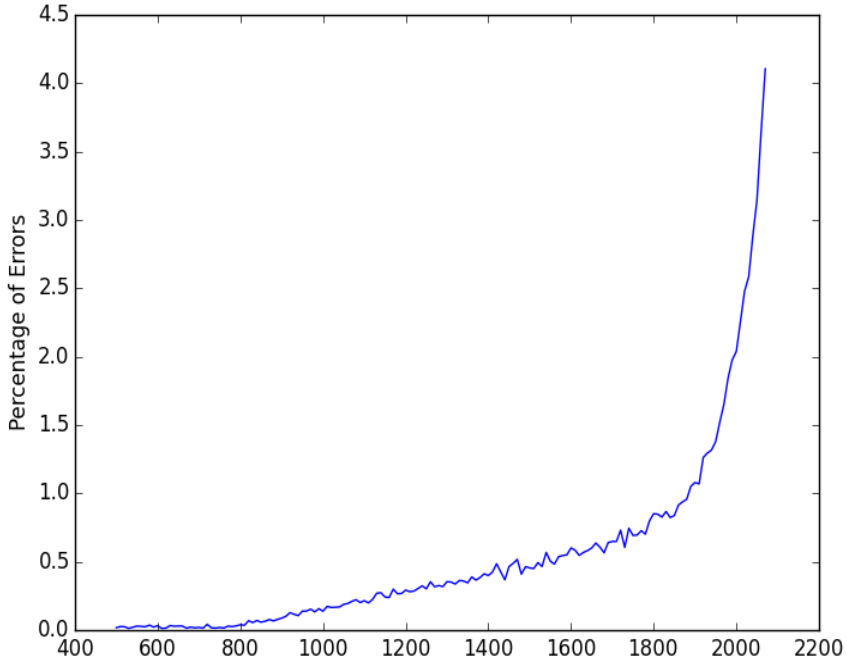


Figure 4.2: Percentage of BERs achieved by selecting the best PUF locations

Our approach of a per-device placement of PUFs considers the lack of spatial correlation of PUFs placed in the same location across devices. Figure 4.3 shows the BER correlation for a single pairing of chips. Each point on the plot represents one of the 2,080 possible PUF instances and its X and Y coordinates indicate its BER when instantiated on chips on two different boards. Correlated BERs would produce a majority of points along the diagonal which is not the situation in our case. More formally, the correlation of per-location BERs among all the pairs of chips is analyzed using the Pearson coefficient.

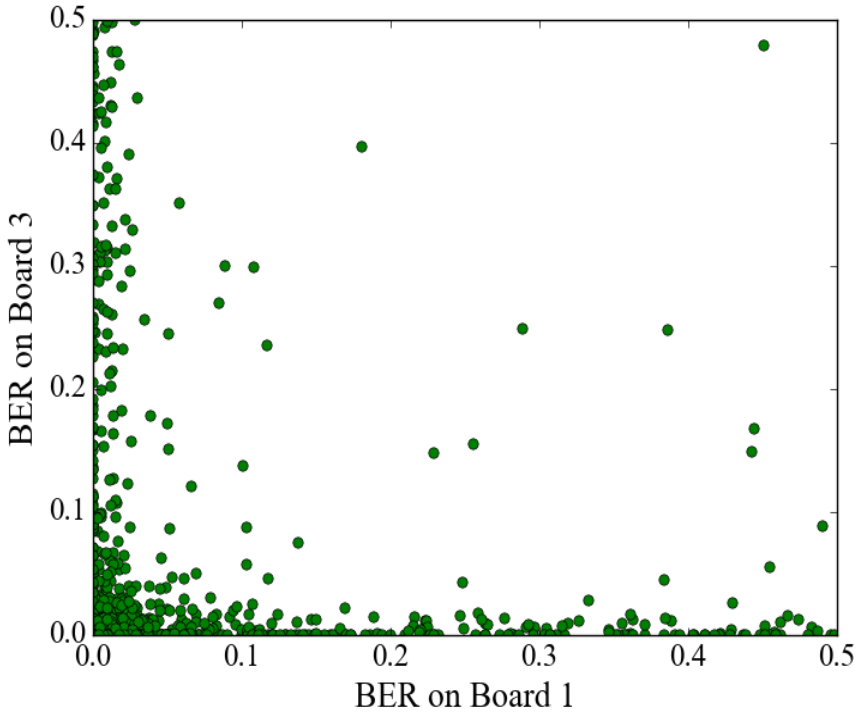


Figure 4.3: Respective BERs of same location PUFs on two different chips

$$r_{x,y} = \frac{\sum_{i=1}^{2080}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{2080}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{2080}(y_i - \overline{y})^2}} \tag{4.1}$$

4.1.1 Pearson Coefficient

For two chips x and y, the Pearson Coefficient $r_{x,y}$[25] is computed using Equation 4.1. Here, x_i represents the BER of PUF location i on chip x and $\overline{x}$ represents the mean BER of chip x. A value close to 0 indicates that the BERs across the chips are uncorrelated. We observed that the Pearson coefficients for all six pairings of the four chips fall between -0.035 and 0.040. This result indicates that the locations of unreliable PUF instances are largely unique to each chip.

4.1.2 Spatial Autocorrelation of PUF Location BERs

While the previous subsection has showed that unreliable PUF locations are uncorrelated across chips, it is important to also consider whether they are correlated

spatially within each chip, as spatial correlation could imply a common cause for unreliability, instead of random per-device variations. The heatmap of Figure 4.1 shows, for a single chip, the reliability of 2,080 PUF instances according to their locations. Informally, the lack of a clear pattern in this figure gives some visual indication that the unreliable PUFs are likely to be random and chip-specific. To formalize the apparent lack of spatial correlation in Figure 4.1, we use Moran's I as a metric to quantify the spatial autocorrelation in the BER of PUF instances. For any single chip instance, Moran's I is computed using Equation 4.3, where B_i and $\bar{B}$ are the BER of PUF instance i and the mean BER of the chip respectively. Computing Moran's I requires a spatial weight w_{ij} to indicate which PUF locations should be considered local to each other. For PUF locations i and j, we compute the weight w_{ij} as shown in Equation 4.2, where r_i and c_i are row and column indices of the i^{th} PUF location. Restating this, the weight is set to 1 if the Euclidean distance between the row and column indices of two PUF locations is less than 10. Moran's I can take values between -1 and 1, where 1 indicates high spatial autocorrelation, and 0 indicates no spatial autocorrelation. The range of values of I obtained on any of the 4 chips is in between 0.013 and 0.017, indicating that the unreliable PUFs do not tend to be highly clustered.

$$w_{ij} = \begin{cases} 1 \text{ if } \sqrt{(r_i - r_j)^2 + (c_i - c_j)^2} < 10 \\ 0 \quad \text{otherwise} \end{cases} \tag{4.2}$$

$$I = \frac{N}{\sum_i \sum_j w_{ij}} \frac{\sum_i \sum_j w_{ij}(B_i - \bar{B})(B_j - \bar{B})}{\sum_i (B_i - \bar{B})^2} \tag{4.3}$$

4.2 Error Correction

In this section, we explain the importance of error correction codes and their use with PUF-based keys. The process of key enrollment and generation with the help

of error correction codes is described along with a detailed analysis of system area is affected by unreliable PUFs.

4.2.1 PUF Based Keys

The generation of cryptographic keys from PUFs should be repeatable over time. The bits generated by a PUF are generally noisy and cannot be used directly as keys without error correction. Fuzzy extractors [5][13] derive reliable key values from noisy data. When a key is first derived from a PUF, the fuzzy extractor generates *helper data* to facilitate generation of the same key at a later time. When the key is later generated in the field, the helper data and the PUF are used to derive the key. The generated key matches the enrolled key as long as the PUF values used at enrollment are within a configurable Hamming distance of each other. The reliability of the key stems from the fuzzy extractor's use of error correcting codes. The security of the key relies on an adversary's inability to guess the PUF output. We use a code-offset fuzzy extractor [5] construction with BCH codes for error correction in this work. In BCH codes, each code is described by a tuple (n,k,t); parameter n is the block size, parameter k is the number of information bits, and parameter t is the number of correctable errors.

In Figure 4.4, k bits are enrolled using n PUF instances. A larger key is generated by splitting up the key into k-bit blocks and using a series of n PUF instances to enroll and generate each key block. The key enrollment is a one time process. During key enrollment, the i^{th} key segment is chosen as a k-bit string X_i and encoded into a n-bit BCH codeword $C(X_i)$: X_i can be decoded from any n-bit string that is within Hamming distance t of coded word $C(X^i)$. The codeword is offset using XOR with an n-bit PUF output W_i and the result is stored as helper data H_i.

During key generation, from Figure 4.5, the helper data H_i is offset by the PUF output observation that may slightly differ from the W_i used during enrollment. This

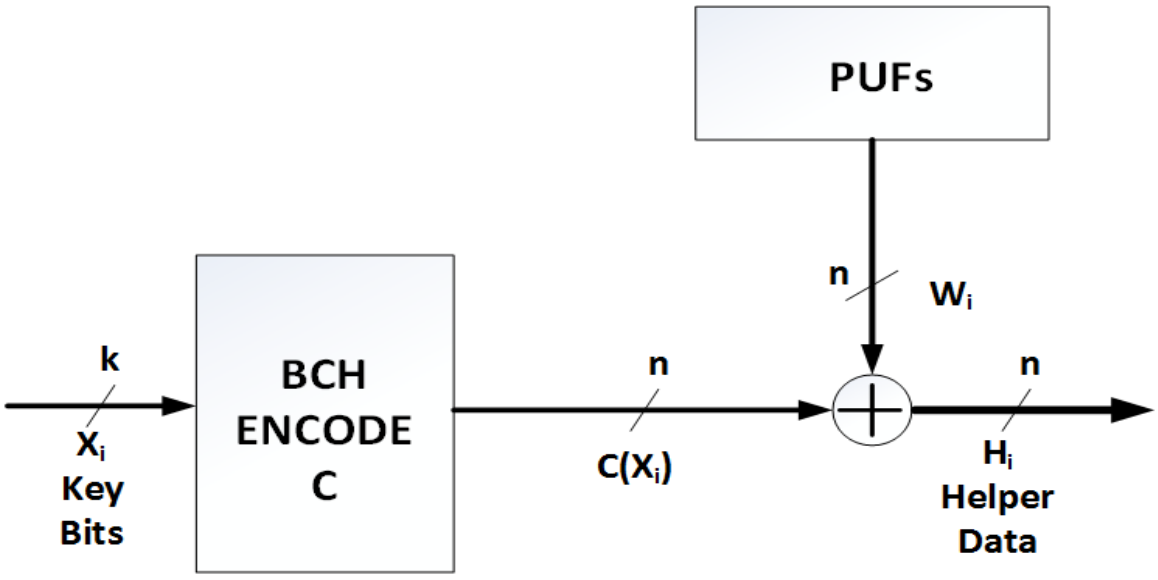


Figure 4.4: One time key enrollment

produces a corrupted codeword that is the original codeword $C(X_i)$ offset by a value. The corrupted codeword can be decoded to generate the enrolled value X_i as long as the corrupted value is within Hamming distance t. In other words, the key bits X_i are generated correctly if the difference between the PUF values used at enrollment and generation does not exceed the maximum number of errors that can be corrected by the BCH code used.

4.2.2 Cost versus Bit Error Rate

The costs associated with error correction are the number of PUF instances and the complexity of the BCH decoder used to correct the errors. These costs increase sharply with the bit error rate of the PUFs. The former cost translates to area while the latter cost is incurred in power and either area or latency. For a given block size (n), there is a tradeoff between the number of information bits encoded (k), and the number of correctable errors (t). For example, for a 127-bit block size, there can be two different codes used. One could correct five errors and carry 92 information bits while another could carry only 36 information bits but correct up to 15 errors.

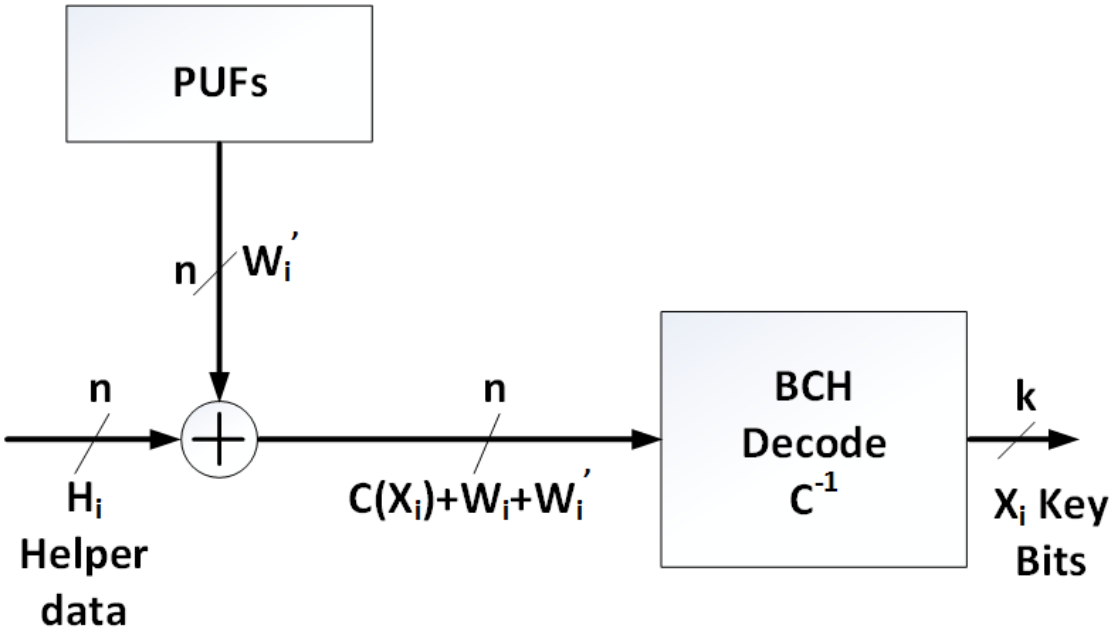


Figure 4.5: Key generation

So, a 256-bit key generated from the above two codes would require 3 and 8 blocks respectively.

In our analysis, we denote the bit error rate as p_{bit} and the probability of incorrectly decoding a block as p_{block}. The latter is computed using Equation 4.4 which shows the probability of finding more than t erroneous bits among n codeword bits when the bit error rate is p_{bit}. The probability of incorrectly generating the 256-bit key is denoted by p_{key} and denoted by Equation 4.5.

$$p_{block} = \sum_{i=t+1}^{n} \begin{pmatrix} n \\ i \end{pmatrix} p_{bit}^{i}(1 - p_{bit})^{n-i} \tag{4.4}$$

$$p_{key} = 1 - (1 - p_{block})^{[256/k]} \tag{4.5}$$

Table 4.1 shows the number of PUF instances and code blocks required to generate a 256-bit key using various 127-bit BCH codes. We can see that a code capable of correcting more errors requires a larger number of code blocks and associated PUF

Table 4.1: Number of PUF instances and code blocks to produce a 256 bit key from 127-bit block size BCH codes

BCH Code			PUF instances and	Code	p_{bit}
n	k	t	helper data size	Blocks	
127	113	2	381	3	0.0001
127	106	3	381	3	0.0004
127	99	4	381	3	0.0010
127	92	5	381	3	0.0020
127	85	6	508	4	0.0032
127	78	7	508	4	0.0048
127	71	9	508	4	0.0088
127	64	10	508	4	0.0112
127	57	11	635	5	0.0135
127	50	13	762	6	0.0190
127	43	14	762	6	0.0221
127	36	15	1016	8	0.0248
127	29	21	1143	9	0.0466
127	22	23	1524	12	0.0541
127	15	27	2286	18	0.0703
127	8	31	4064	32	0.0870

instances, but tolerates a higher BER. For example, a reduction of BER from 0.06 to 0.03 can reduce the number of code blocks from 18 to 9, and reduce the number of PUFs required from 2,286 to 1,143.

4.3 Implementation

Our complete system implements several encryption schemes using PUF-based keys[1]. We implemented authenticated encryption using AES in Galois Counter Mode for 256-bit and 128-bit keys and DES in Electronic Code Book mode. The three primary components of the system are the PUF instances, the BCH decoder for error correction and the encryption blocks. Publicly available Verilog code is used to implement the BCH decoder [4], and the encryption blocks [1][11].

[1]Some of the results in this subsection were generated by Mr. Naveen Dumpala.

4.3.1 Per Device Placement

To determine the benefits of placing PUFs in low-cost positions, we consider two cases. For *variation-aware* PUF placement, per-chip PUF behavior is characterized and low-cost PUFs are used. In *variation-agnostic* PUF placement, PUF locations are randomly selected. In this section, we compare the costs of variation-aware per-device placement approach to variation-agnostic placement. In variation-agnostic placement, the PUFs are placed arbitrarily by the tool and the same placement is used for all chips. This approach is the typical use case for designers today.

To determine PUF locations where maximum reliability can be achieved, the BER of each PUF for every chip location was computed. Assuming the BER to be 0.010 for half the PUFs for a 256-bit key requires an n = 127, k = 64, t = 10 BCH code and four code blocks. With four code blocks, a total of 508 PUF instances are required. We constrain the tool to use the 508 lowest BER PUF instances on the chip.

In variation agnostic placement, we assume the mean BER of the PUFs to be 0.042 based on an analysis of our results. The generation of a 256-bit key requires a more robust BCH code and a larger number of PUFs. For this BER, an n = 127, k = 29, t = 21 BCH code is needed, with 9 code blocks used. Relative to variation-aware placement, this represents a 125% increase in the number of PUFs and helper data bits, and a 78% increase in the size of the BCH decoder to implement the more complex code.

The system architecture is shown in Figure 4.6. The helper data is stored in the block RAM and transferred into a 127-bit register sequentially in 4-bit words. In variation aware placement, the BCH decoder operates on a 127-bit block and corrects the errors in the lower 64 bits to produce 64 bits of key. Operating on four blocks in sequence produces the entire 256-bit key from 508 bits of helper data and 508 PUF outputs.

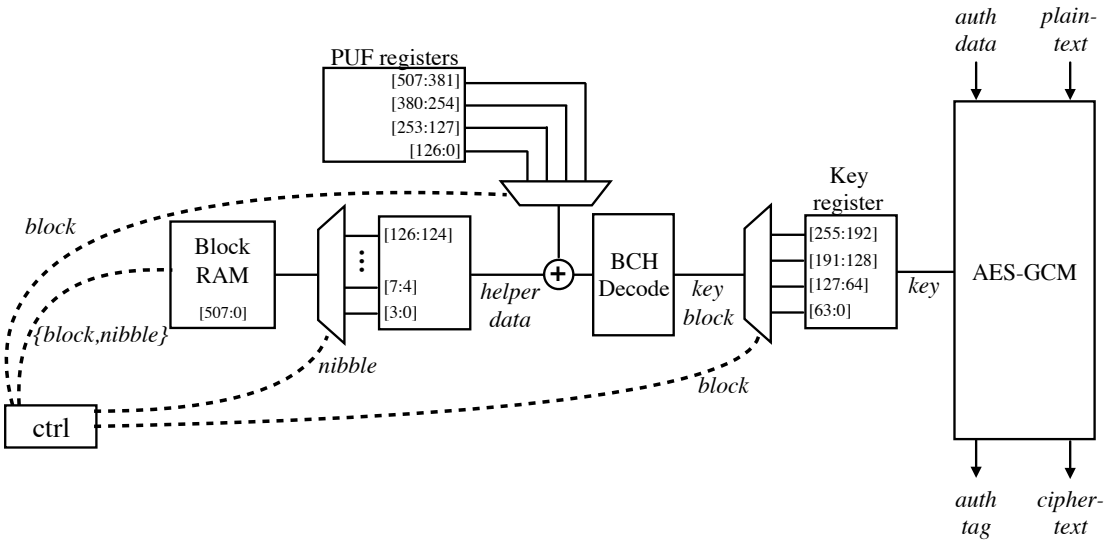


Figure 4.6: Implemented system of AES-GCM authenticated encryption using PUF based key generation. The specific configuration shown uses a BCH code with $n = 127$, $k = 64$ and $t = 10$. Four code blocks are used to generate an overall 256-bit key. Each code block generates 64 key bits from 127 bits of helper data and the outputs of 127 PUF instances; a total of 508 PUF instances and 508 bits of helper data are used to generate the 256-bit AES key. The 127-bit blocks of helper data are loaded from block RAM in 4-bit words.

Variation agnostic placement extracts only 29 bits of key from each block requiring a total of nine blocks. From Table 4.2 we can see that the variation aware placement scheme uses fewer LUTs for the PUFs and the error correcting code hardware.

Table 4.3 provides information about the number of blocks needed for variation-aware placement when compared to the variation agnostic scheme for the three types of encryption cores.

4.4 Key Generation

Based on the results of Section 4.3, we can confidently claim the benefits of our approach. By analyzing the chip for reliable locations and thereby constraining the PUFs to specific locations to achieve optimum reliability, substantial area savings

Table 4.2: Breakdown of LUT counts by function

	PUF	BCH	Core	Total
Variation Agnostic AES 256	2569	2219	6312	11100
Variation Aware AES 256	1143	1249	6308	8700
Variation Agnostic AES 128	1397	2160	4771	8328
Variation Aware AES 128	508	1292	4770	6570
Variation Agnostic DES 56	889	2055	250	3194
Variation Aware DES 56	254	1082	292	1628

Table 4.3: Comparison of the number of blocks required to generate respective sized keys

	Variation-agnostic	Variation-aware
PUF BER used in analysis	0.034	0.010
BCH code parameters (n,k,t)	(127,29,21)	(127,64,10)
Code blocks for AES 256-bit key	9	4
Code blocks for AES 128-bit key	5	2
Code blocks for DES 56-bit key	2	1

can be gained as per Table 4.2. In this section, we analyze the key obtained by using these most reliable locations for PUFs.

The results obtained for the variation aware and variation agnostic schemes depend on the BCH code that has been used. The size of the BCH code is determined by the BERs of the PUFs that are being used in the design. For our analysis, an average BER of all the PUFs has been taken into consideration to determine the size of the BCH code needed. Chapter 5 provides a detailed explanation on the drawbacks of this single parameter based BER and introduces a new model to decide on the number of errors that need to be corrected in a block. Our underlying assumption of having an equal number of $1s$ and $0s$ in a block of PUFs holds firm until the point of selection of the most reliable PUFs. On selecting the most reliable PUFs on a chip i.e the locations which flip the least from their most frequently produced outputs, we observed that a large number of these locations produce a 1.

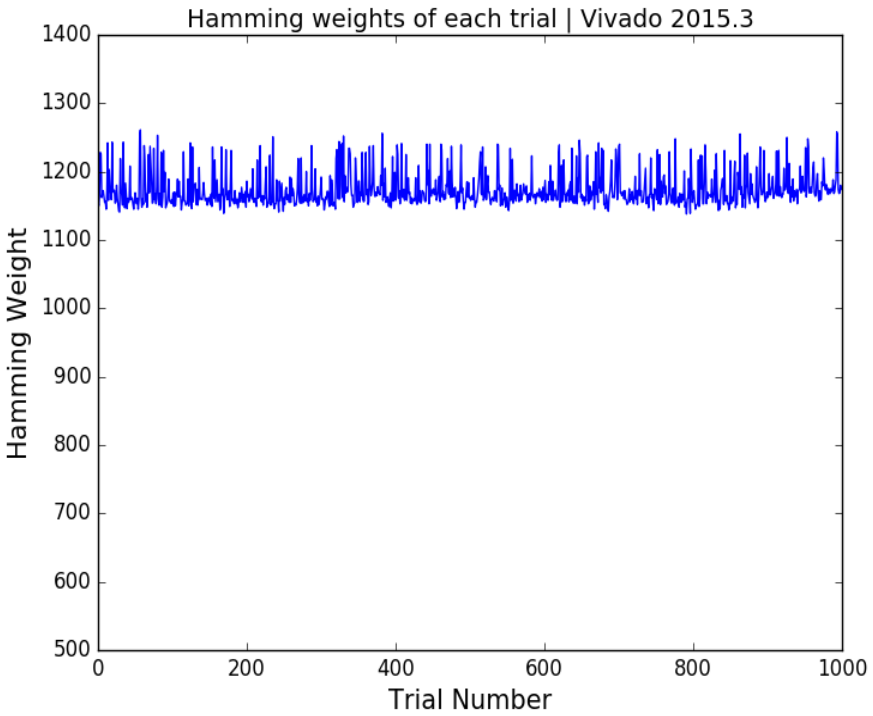


Figure 4.7: Number of ones in a trial of 2,048 PUFs

To understand this PUF behavior, an understanding of the nature of the output bits of each PUF on every trial is required. Figure 4.7 illustrates the bits produced by the PUFs. Every point in the figure relates to the total Hamming weight of all the PUFs in a trial on the y-axis and the trial number on the x-axis. By observation, we can see a high number of PUFs flipping on every trial, leading to a large number of spikes. The direction of the spikes provides evidence that a number of PUFs flip from a 0 to a 1.

Due to the 0 producing PUFs flipping to a 1, the BER of these PUFs increases while that of the 1 producing PUFs remains low. Hence our approach of generating a key by selecting the most reliable PUFs based on the BER of the PUFs leads to the selection of a large number of 1s.

To observe if the locations that exhibit this behavior are localized or similar on multiple trials, the most frequently flipping locations must be identified. Figure 4.8

represents the locations of the PUFs which typically produce a 0 but flip to a 1 on occasion. The four plots in the figure are from the four trials that exhibit the highest number of flips. A point on each plot indicates if a location has flipped or not. Colored points represent the locations that have flipped in the particular trial. The lack of a clear pattern or common locations on the four plots indicates that there are no specific locations or areas on the chip that indicate a concentration of PUFs that flip to a 1.

To understand this PUF behavior, we varyied the clock frequency and time interval parameters of our design to calculate the Hamming weights of every trial during experimentation. By varying the clock frequency of our design and changing the time interval between successive trials, variations in PUF behavior can be observed. Sections 4.4.1 and 4.4.2 provides details about these experiments and the results observed.

4.4.1 Analysis of PUF outputs at different frequencies

Clock frequency can be considered in evaluating our PUFs. Our PUFs contain two shift registers which output their data based on a clock signal. Clock skew may affect PUF behavior. To understand the clock behavior of our PUFs, the input clock frequency was varied. Ideally, PUF outcomes should remain consistent across input clock frequency. To achieve a comprehensive data set, we ran our design at three different clock frequencies: 10 MHz, 50 MHz and 100 MHz. The results for each of the frequencies are shown in Figure 4.9. By observation we can state that the outputs of the PUFs over the 1000 trials remain approximately constant. Hence, PUF behavior is not affected by the input clock frequency.

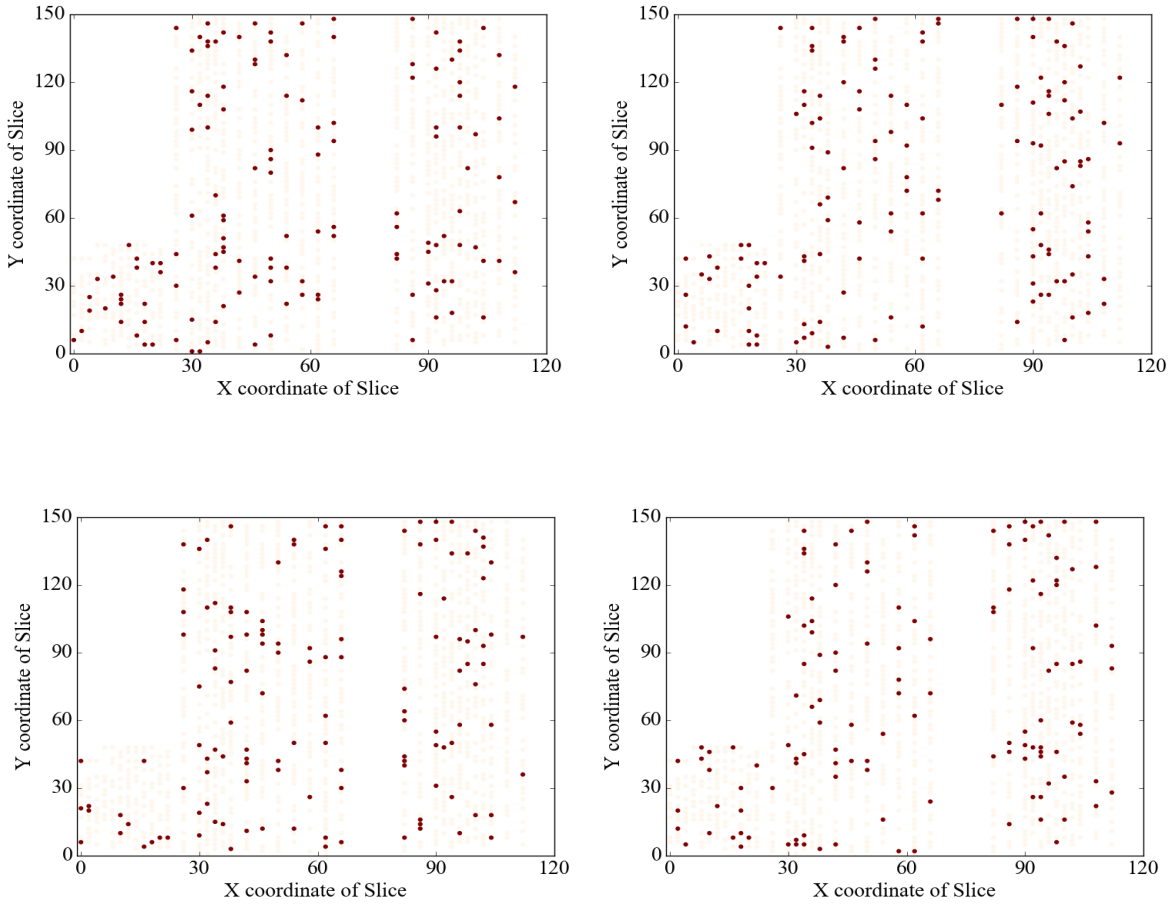
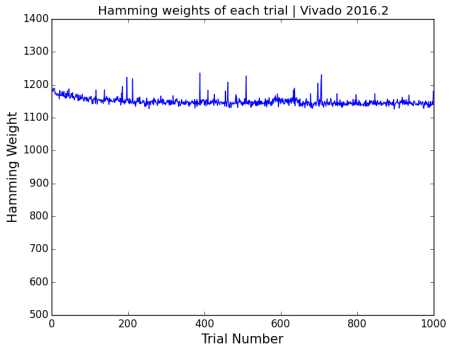


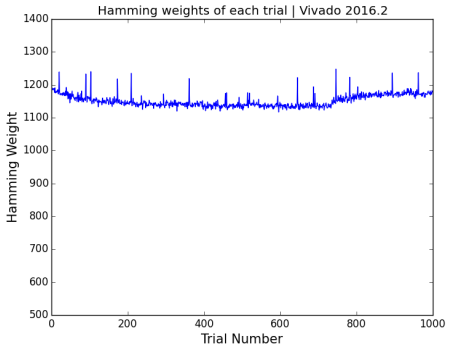
Figure 4.8: Locations of the PUFs flipping to a 1

4.4.2 Analysis of PUF outputs over increasing time intervals between successive trials

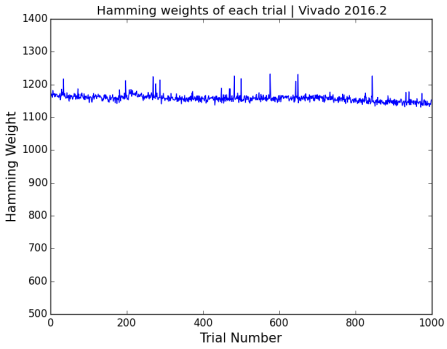
Our analysis thus far has involved computing the outputs of 2,080 PUFs over 1,000 trials. Every trial involves a reprogramming of the FPGA with a time interval of 200 milliseconds between successive trials. To eliminate the possibility of this time interval as the cause of errors, we increased the interval to 600 milliseconds. Figure 4.10 depicts the Hamming weight of the PUFs in a trial. Similar to the results observed in Section 4.4.1, the change in time interval between trials does not affect the PUF outcomes.



(a) 100 MHz



(b) 50 MHz



(c) 10 MHz

Figure 4.9: Hamming weights at varying clock frequencies

Based on the results in Section 4.4.1 and 4.4.2, we can observe that the PUFs produce approximately the same number of spikes. Additionally, Figures 4.10 and 4.9 indicate a reduced number of spikes when compared to those obtained in Figure 4.7. On inspection, it was found that the results from the two sections were produced using PUF statistics generated by Xilinx Vivado version 2016.2 while the results in Figure 4.7 were produced using Xilinx Vivado version 2015.3. At this time, no specific reason has been identified for the differences across Vivado versions. The routing differences generated by the two versions likely are the cause, although this remains to be confirmed.

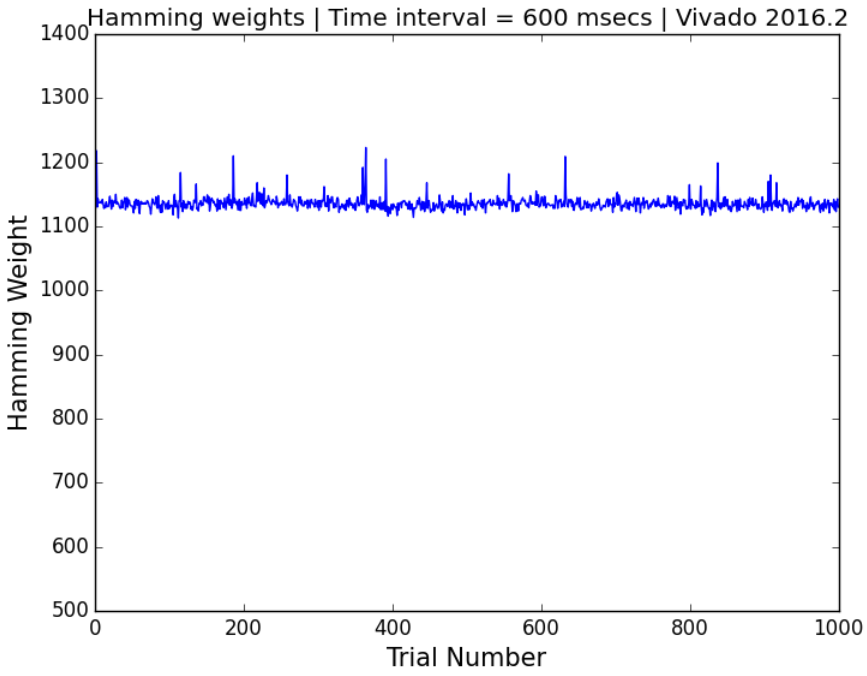


Figure 4.10: Number of ones in a trial with an extended time delay between trials of 600 ms

CHAPTER 5

TWO PARAMETER MODEL FOR ERROR CORRECTION

The security benefits of PUF usage are accompanied by post processing. In addition to reliability and uniqueness, efficiency also plays are important part in PUF based system. To understand the PUF behavior in terms of reliability, an accurate model that closely fits the statistics of a PUF is required to better predict the number of errors to be corrected [15]. This chapter primarily focuses on predicting an accurate BCH code depending on our PUF behavior using a more robust model as in [15].

5.1 Fixed Error Rate

The commonly used PUF reliability model used in previous sections considers a fixed error rate. In this approach, the response bit of each PUF is assumed to be equally prone to errors. Hence the approach suggests computing the average BER of all PUFs used in the application to determine the BCH code size. A comment on such an approach in [15] is: 'Many details are lost by reducing the reliability behavior to a single average-case parameter'. Specifically, there is a possibility that the average PUF BER increases due to a small number of highly unreliable PUFs or decreases due to a small number of highly reliable PUFs. The limitations of such an approach is that it does not take into account the randomness of the enrollment value (i.e. the value of the PUF taken for comparison) to determine if the PUF output is correct or not. Additionally, by considering all the PUFs to be equally prone to errors, it does

not take into account the exact behavior of each PUF. Hence, such an approach does not accurately predict the number of errors that need correction. This observation motivates the use of another model to predict the strength of the required BCH code required by considering actual PUF behavior.

5.1.1 Two Parameter Model

To obtain a better estimate of the number of errors to be corrected in a block of PUFs and hence utilize a more realistic BCH code for our PUFs, we refer to the model implemented by [15]. This model considers the error variation of each PUF and assumes that every PUF has its own error probability.

The behavior of an evaluation j of a PUF i as described in [15] is defined by the following variables.

1. The one probability (p_i): The probability of a cell i returning a '1' during a random evaluation.

2. The error probability ($p_{e,i}$): The probability of a PUF to produce an output different from an earlier recorded output during enrollment.

PUF behavior is determined by the physical processes involved in making the circuit. The two factors that dominate the output of a PUF are as follows:

1. The process variable (m_i): This factor determines the effect of process variations on a PUF outcome which is imparted during manufacturing. It is sampled once upon device creation [15].

2. The noise variable (n_i^j): This factor determines the effect of random noise on the outcome of a PUF during evaluation. The noise variable is sampled at every evaluation.

5.1.2 Fitting the Distribution

Equation 5.1 [16] can be used to represent PUF statistics. Here, $\lambda_1 = \sigma_N$ and $\lambda_2 = (t - \mu_M)/\sigma_M$ where μ_M & σ_M represent the mean and standard deviation of the process variable, μ_N & σ_N represent the mean and standard deviation of the noise variable, while t represents a threshold value. If the combined effect of the process and noise variables is above this threshold, it cause a PUF to produce an incorrect output. Non-linear optimization is performed over (λ_1, λ_2) to minimize the mean square error of Equation 5.2 using the Levenberg-Marquardt algorithm. Here $F(x)$ represents the empirical probability of a PUF producing a 1 on x fraction of trials. Keeping in mind that our PUF produces unknown spikes in random evaluations, the algorithm produces an accurate fit at $\lambda_1 = 0.0801$ and $\lambda_2 = -0.0346$. Figure 5.1 shows that the model yields a close fit to the PUF statistics.

$$cdf_p(x) = \phi(\lambda_1 \phi^{-1}(x) + \lambda_2) \tag{5.1}$$

$$cdf_p(x) - F(x) \tag{5.2}$$

By using the values of λ_1 and λ_2 in Equation 5.1, a one probability distribution can be obtained.

$$pdfp_e(x) = \lambda_1(1 - x)\frac{\varphi(\lambda_1\phi^{-1} + \lambda_2) + \varphi(\lambda_1\phi^{-1}(x) - \lambda_2}{\varphi(\phi^{-1}(x))} \tag{5.3}$$

Equation 5.3 from [15] gives the probability distribution function of the error probabilities where $\varphi(x)$ and $\phi^{-1}(x)$ refer to the probability density function of a normal distribution and the inverse of the cumulative distribution function of a normal distribution. For our analysis, we sampled 1000 points of this distribution by varying x from 0 to 0.999 in steps of 0.001.

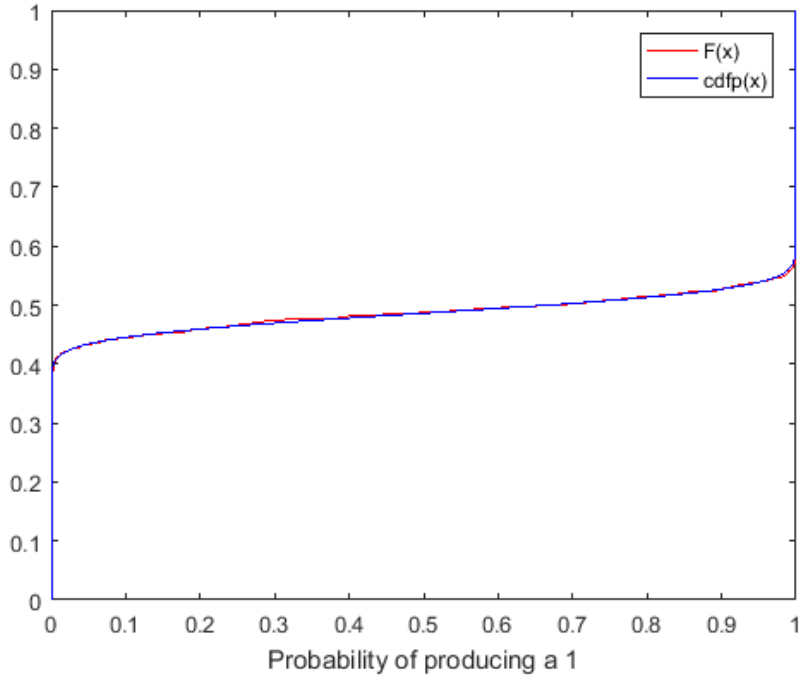


Figure 5.1: Good fit of the PUF statistics obtained using $\lambda_1 = 0.0801$ and $\lambda_2 = -0.0346$

5.1.3 Key Failure Rate

Key generation fails if the number of errors observed in a block of PUF bits exceeds the error correcting capability t of the BCH code. The block failure rate p_{fail} is the failure probability. With the fixed error model, each PUF is equally prone to error and hence the number of errors in an n-bit response is binomially distributed. In the new model with random error-probabilities, the number of errors is Poisson-binomially distributed [15]. The Poisson binomial distribution, $F_{PB}(t; p_e^n)$, can be calculated using Equation 5.5 [15] where t refers to the number of errors that can be corrected by the BCH code, n refers to the block size and p_e^n refers to the error probabilities of n PUFs.

$$p_{fail}(p_e^n) = 1 - F_{PB}(t; p_e^n) \tag{5.4}$$

$$F_{PB}(t; p_e^n) = \frac{t+1}{n+1} + \frac{1}{n+1} \sum_{i=1}^{n} \frac{1 - C^{-i(t+1)}}{1 - C^{-i}} \prod_{k=1}^{n} (p_{e,k} C^i + (1 - p_{e,k})) \tag{5.5}$$

$$C = e^{\frac{j2\pi}{n+1}} \tag{5.6}$$

For our analysis, a block size of 127 bits is chosen. To compute the block failure rate, a random sample of 127 points of p_e from the distribution described by Equation 5.3 is chosen. By using this computed value, the failure rate of the block is generated using Equation 5.4. This procedure is performed 1,000 times to obtain the distribution of the block failure rate.

We performed the above steps for the variation agnostic, multiplexer selection (described in Chapter 6) and variation aware placement schemes using the following selection approaches:

1. Variation Agnostic: Random sampling of blocks of size 127

2. Multiplexer selection: Pairwise selection of 127 points from blocks of size 254: In this approach, 254 points of p_e are sampled instead of 127. The 127 points from these points are obtained by selecting, from every pair, the point with a lower value of p_e.

3. Variation Aware: Global optimum selection of 127 points from blocks of size 254: Similar to the previous approach, a total of 254 points are sampled from p_e. By sorting the points based on their values, the lower 127 points are using in this approach.

Our goal is to obtain a key failure rate of lower than $1e^{-6}$ on at least 99% of the keys. Since keys are made up of multiple blocks, it is essential to look at individual blocks. Figure 5.2 gives information about the block failure distribution for the variation agnostic approach. In this plot, the block failure distribution for BCH codes capable of correcting 23, 27 and 31 errors can be observed.

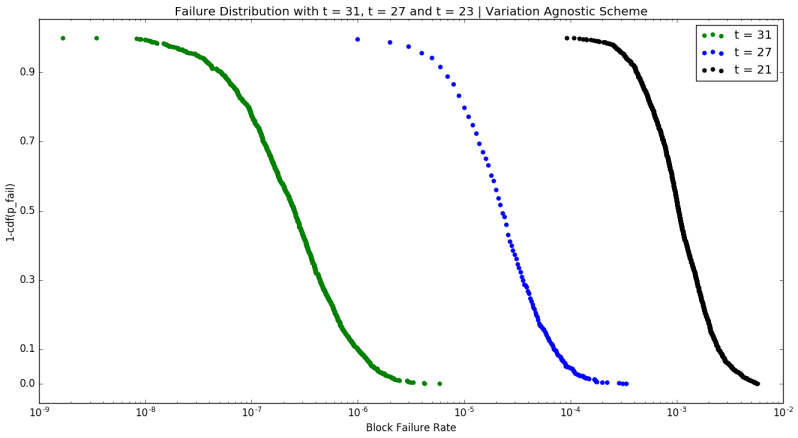


Figure 5.2: Block failure distribution for variation agnostic selection

Figure 5.3 provides information about the block failure distribution for the multiplexer-selection scheme for BCH codes capable of correcting 23, 27 and 31 errors.

Figure 5.4 shows the failure rate distribution of a block for the optimum approach. In this approach, the optimum 127 from a random sample of 254 points are selected. The figure indicates a BCH code able to correct just 23 errors is needed for the optimum approach compared to 31 for the agnostic and 27 for the multiplexer scheme. This reduction would save area for the BCH error correction hardware.

As previously stated, the key failure rate is dependent on the blocks that make up a key. For example, to generate a total of 128 information bits, the BCH code requirements of each scheme would require a total of 16, 9 and 6 blocks, respectively, based on the BCH codes obtained above. Since keys are made of multiple blocks, the failure rate of each block needs to be considered to generate a key failure rate that meets our requirements. Equation 5.7 shows $p_{keyfail}$, the probability of a key failure. Here p_{fail} refers to the probability a block can observe more errors than it can correct which can be computed using Equation 5.4. This probability needs to be taken into account to predict the key failure rate and the strength of the BCH code.

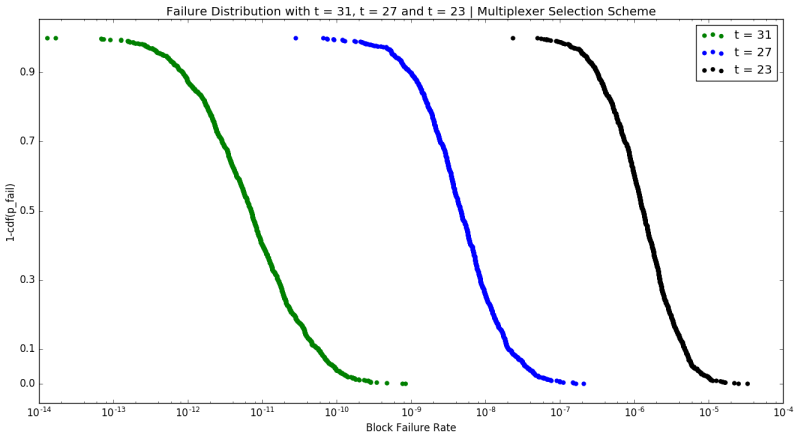


Figure 5.3: Block failure distribution for multiplexer selection approach using 2:1 selection

$$p_{keyfail} = 1 - \prod_{i=1}^{n} 1 - p_{i,fail} \tag{5.7}$$

Figures 5.5, 5.6 and 5.7 illustrate the key failure distribution of each of the three schemes. The variation aware and multiplexer selection schemes need BCH codes of (127,22,23) and (127,15,27), respectively, to meet the requirements. The variation agnostic approach would require a stronger code than the one considered to compute the block failure distribution. A stronger code can be used in the future by allowing a larger block size to be considered. By using a BCH code capable of correcting 31 errors for the variation agnostic scheme, only 1% of the PUFs have a failure rate lower than $1e^{-6}$.

The process flow of generating the key failure rate distribution using the 2 parameter model is shown in Figure 5.8. It provides a high level view of the steps involved in obtaining the key failure for a particular BCH code.

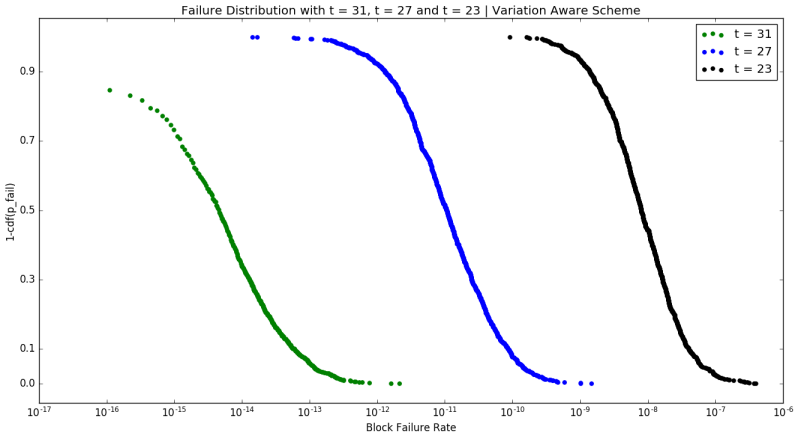


Figure 5.4: Block failure distribution for variation aware selection

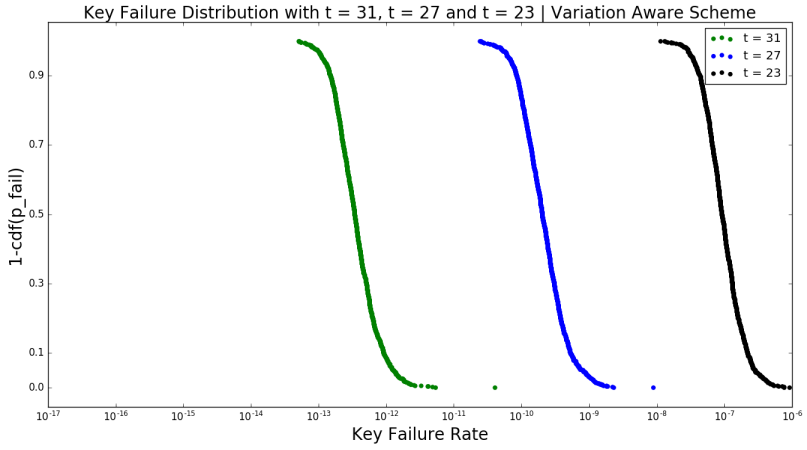


Figure 5.5: Key failure distribution for variation aware selection

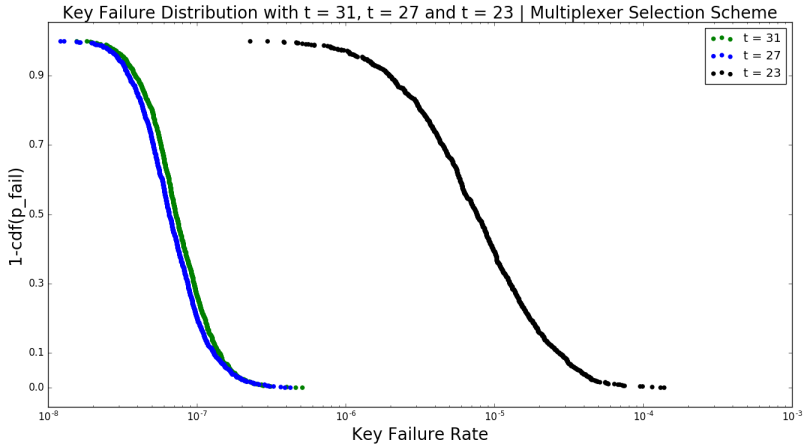


Figure 5.6: Key failure distribution for multiplexer selection

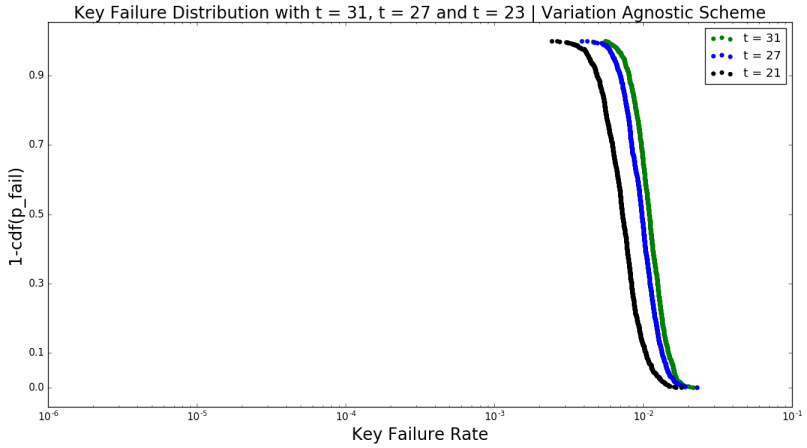


Figure 5.7: Key failure distribution for variation agnostic selection

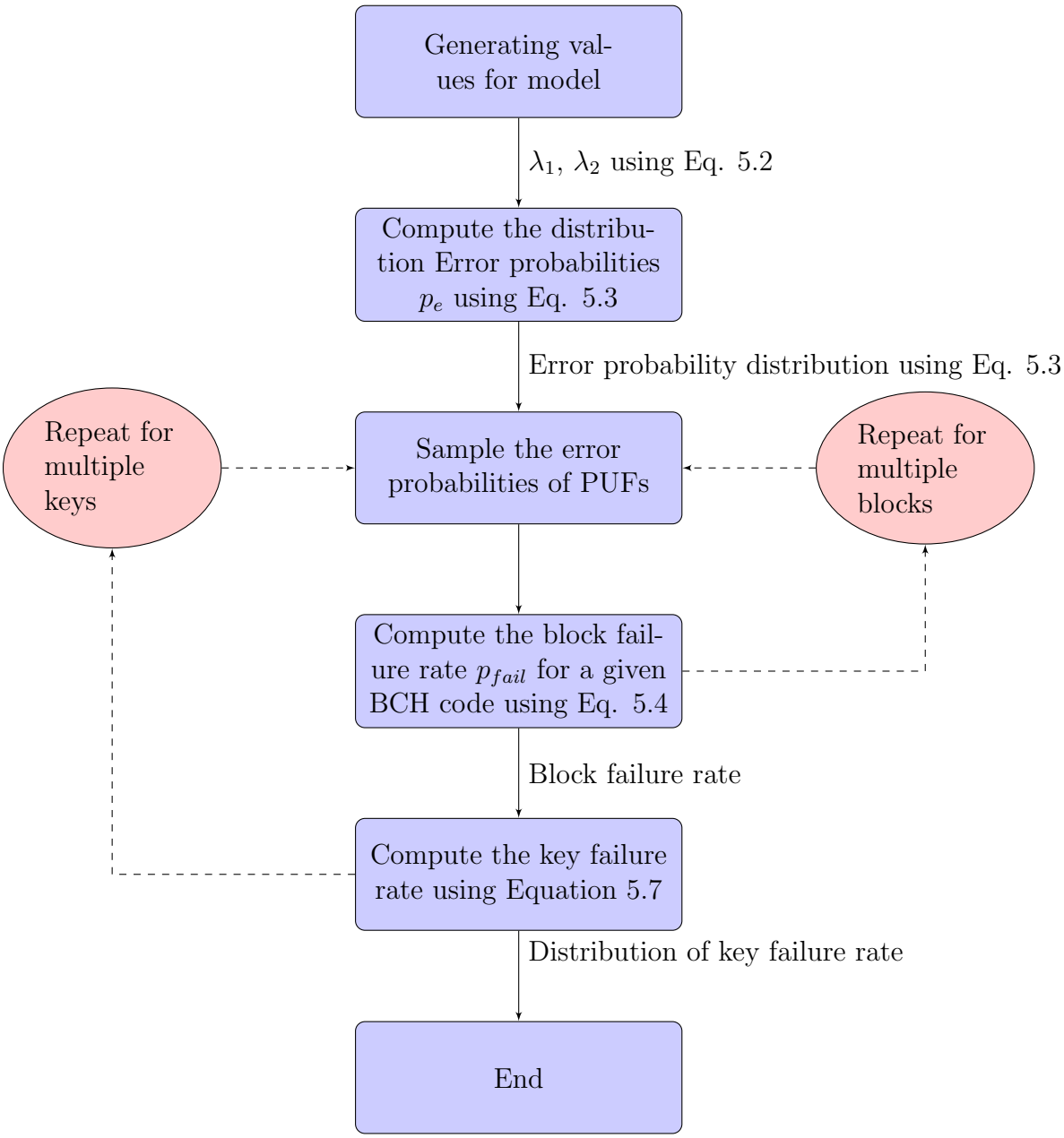


Figure 5.8: Process flow to generate the key failure distribution using a chosen BCH code and fitted parameters λ_1 and λ_2

CHAPTER 6

PUF SELECTION USING MULTIPLEXERS

In Chapter 4, it was shown that variation aware placement reduces area consumption by 21% for AES cores and about 50% for DES cores compared to variation agnostic placement. In the former case, PUFs are constrained to locations which are highly reliable. In the latter case PUF locations are randomly selected without considering their reliability. The reliability difference impacts BCH code hardware size. Although variation aware placement is effective, it requires an FPGA to be recompiled for every chip.

In this chapter, we propose a new placement method which strikes a balance between the two schemes discussed so far, while eliminating the need for per-device recompilation.

6.1 Multiplexer Selection

Variation-aware placement requires the following steps

1. An FPGA design is created such that PUFs are placed at all locations on the chip.

2. The outputs generated by the PUFs are evaluated to get an estimate of reliability.

3. The FPGA design is recompiled to only include PUFs in locations which demonstrate high reliability along with other design logic.

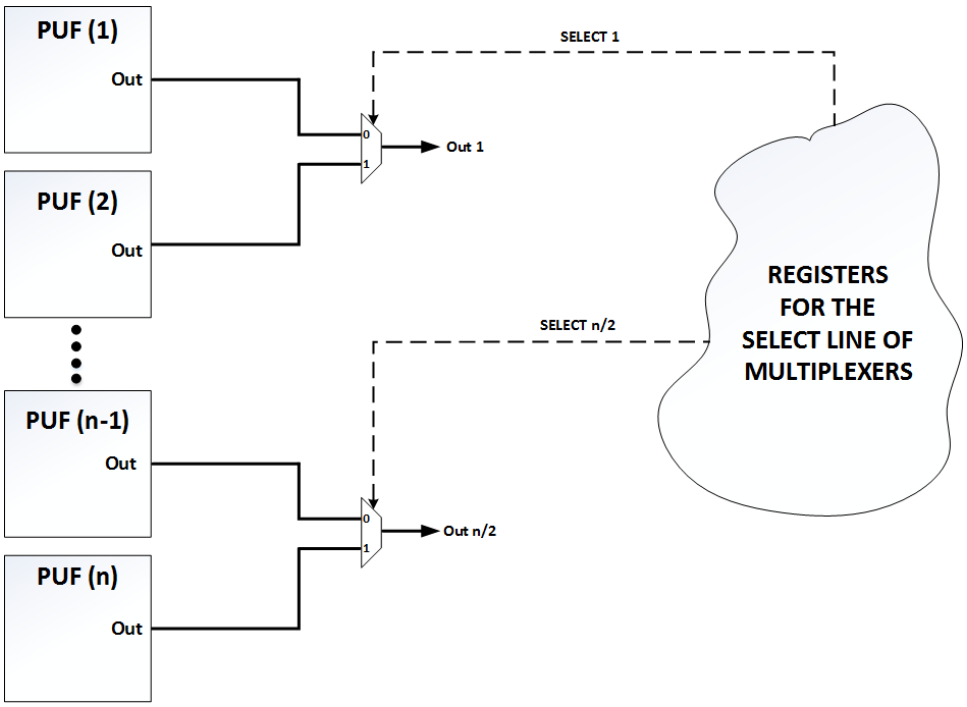


Figure 6.1: Multiplexer selection of the PUFs

The third step above requires FPGA design resynthesis for every chip. Since companies often ship thousands of copies of the same product, each holding the FPGA design, such recompilation may not be practical since our designs required about 20 minutes to fully compile. Note that the variation agnostic approach does not have this issue. The PUFs can be located at the same locations on every chip.

To overcome the drawbacks of the variation aware scheme but allow for rapid per-device PUF selection after place-and-route, we propose an alternative scheme. Figure 6.1 depicts the multiplexer selection approach. Each pair of PUFs has an associated multiplexer. This multiplexer selects the PUF with the lower BER among the pair. The selection bits on the multiplexers can be selected on a per-FPGA basis without the need to recompile the device. In our experimentation, the Zynq processor on a Xilinx ZedBoard is used to configure the multiplexers after the FPGA configuration is loaded. The select lines of the multiplexers are driven from writable registers inside the FPGA that are accessed by the Zynq. Consider the generation of 128 information

bits from a BCH code of (127,64,10) just for illustrative purpose. To generate 128 bits, the number of standalone PUFs with this BCH code would be 254 (two blocks). Since in this approach one PUF is selected from a pair of PUFs, a total of 508 PUFs are needed to generate the required number of information bits. The steps of this scheme are as follows:

1. 508 PUFs are placed on the chip. The locations of the PUFs are determined by the FPGA place and route tool.

2. The outputs of these 508 PUFs are analyzed to determine which PUF among the pair has lower BER.

3. The FPGA is now programmed with the user design which includes design logic, the 508 PUFs, and the select logic.

4. The values of the multiplexer select lines in the loaded design are written to on-chip registers to select 254 PUFs. A serial port and Xilinx SDK is used.

The selected 254 PUFs provide reliability which is between the variation agnostic and variation aware cases.

6.2 Results

6.2.1 Fixed Error Rate

The results obtained in this section assume a fixed error rate model, i.e single parameter of average BER. The BER values of the PUF with different sizes of multiplexers are shown in Figure 6.2. This figure considers all the PUFs on the chip. The figure also indicates the average BER of variation agnostic selection, i.e using all the PUFs on the chip. For the variation agnostic approach where no selection is done, the BER is 0.042. By using multiplexers, the BER associated with the PUFs is reduced. A 2-to-1 multiplexer selects one PUF out of two and the associated BER of

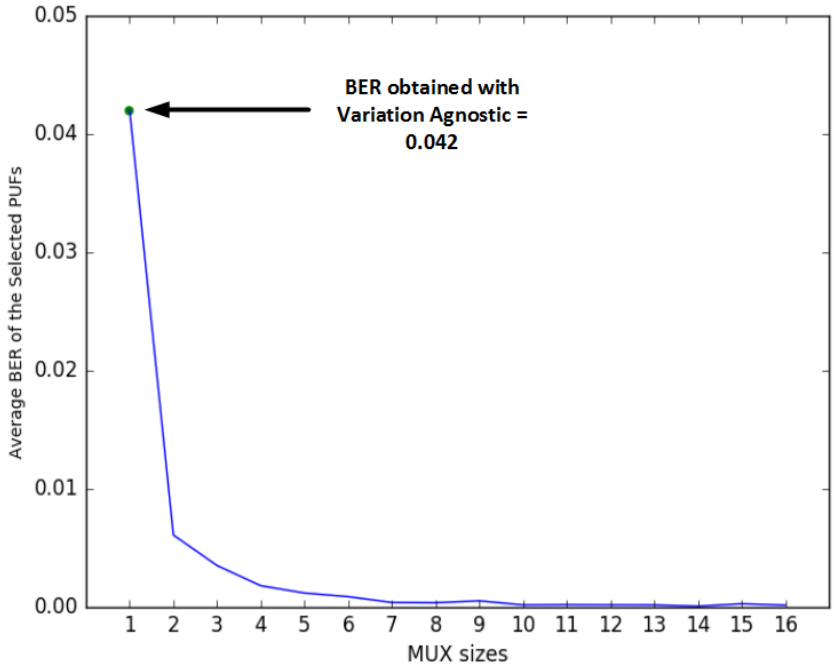


Figure 6.2: Effect of MUX size on the BER. Selection is taking place across 2,080 total available PUFs

the selected PUFs is 0.006. Lower BERs are observed by increasing the multiplexer size to 4-to-1 where one out of four PUFs is selected. A better BER value compared to the agnostic placement can be achieved which leads to a smaller BCH code and lower area utilization.

There is a drawback of the selection scheme. Figure 6.3 indicates the total number of usable PUFs obtained with the multiplexer selection scheme. It provides information on the number of PUFs available using different sized multiplexers. With a total of 2080 PUFs on a chip, a 2-to-1 multiplexer scheme would provide a total of 1040 usable PUFs. Similarly, a 4-to-1 will provide a total of 520 usable PUFs out of 2080 PUFs. Hence, by using the approach, the total number of usable PUFs decreases. By increasing the size of the multiplexer, a BER reduction of the selected PUFs is also observed.

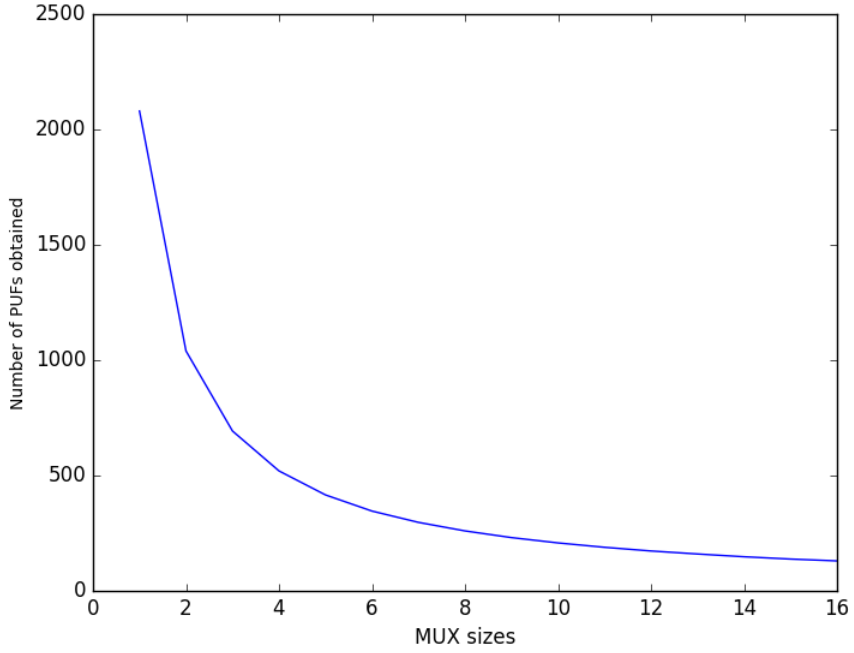


Figure 6.3: Effect of MUX size on number of PUFs. Selection is taking place across 2,080 total available PUFs

6.2.2 Two parameter model

In Chapter 5, a more accurate model to predict the number of errors that need to be corrected based on our PUF statistics than averaging was developed. Section 5.1.3 gives us the BCH code to be used with a 2-to-1 multiplexer scheme to meet our criterion of $1e^{-6}$. Using the BCH codes obtained from Section 5.1.3, Table 6.1 lists the area utilization of our system in terms of the PUFs, error correcting codes (BCH) and a DES 56 core for variation agnostic, variation aware, and 2:1 multiplexer selection. The table shows that the size of the BCH code is reduced from a variation agnostic approach.

Table 6.1: Area utilization (in LUTs) of variation aware, variation agnostic, and multiplexer select approaches

Configuration	PUF	BCH	DES 56	Total	Failure Rate
Variation Agnostic (127,8,31)	2,286	2,902	251	5,439	99% $<1e^{-2}$
2:1 Configuration (127,15,27)	2,520 + 690 (MUXes)	2,555	249	5,994	99%$<1e^{-6}$
Variation Aware (127,22,23)	889	2,235	249	3,373	99% $<1e^{-6}$

CHAPTER 7

CONCLUSION

Our work examines PUF placement in FPGAs based on PUF reliability (variation-aware placement). Our results are compared with random PUF placement (variation agnostic placement). We also examine a PUF selection approach which allows for post-configuration selection of PUFs. Although this latter approach reduces BCH error correcting hardware, the overhead of the PUFs and selection logic is significant. We have implemented three cryptosystems using different PUF-based key sizes and have demonstrated that our variation-aware approach can save between 21% and 49% of area in these systems compared to a variation-agnostic approach.

Future work could consider improving the multiplexer scheme by using bit masking of PUFs. New approaches to PUF implementation in FPGAs could also be considered.

BIBLIOGRAPHY

[1] Basic DES crypto core. `http://opencores.org/project,basicdes`, 2010.

[2] Anderson, Jason H. A PUF design for secure fpga-based embedded systems. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference* (2010), IEEE Press, pp. 1–6.

[3] Cherkaoui, Abdelkarim, Bossuet, Lilian, and Marchand, Cédric. Design, evaluation and optimization of physical unclonable functions based on transient effect ring oscillators. Tech. rep., Cryptology ePrint Archive, Report 2015/623, 2015.

[4] Dill, Russ. bch_verilog. `https://github.com/russdill/bch_verilog`, 2014.

[5] Dodis, Yevgeniy, Reyzin, Leonid, and Smith, Adam. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In *Advances in cryptology-Eurocrypt 2004* (2004), Springer, pp. 523–540.

[6] Feiten, Linus, Spilla, Andreas, Sauer, Matthias, Schubert, Tobias, and Becker, Bernd. Analysis of ring oscillator PUFs on 60nm FPGAs. *European cooperation in science and technology*.

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

[8] Guajardo, J, Kumar, S, Schrijen, GJ, and Tuyls, P. FPGA intrinsic PUFs and their use for IP protection. *Cryptographic Hardware and Embedded Systems* (2007).

[9] Holcomb, Daniel E, and Fu, Kevin. Bitline puf: building native challenge-response PUF capability into any SRAM. In *International Workshop on Cryptographic Hardware and Embedded Systems* (2014), Springer, pp. 510–526.

[10] Hori, Y., Yoshida, T., Katashita, T., and Satoh, A. Quantitative and statistical performance evaluation of arbiter physical unclonable functions on fpgas. In *2010 International Conference on Reconfigurable Computing and FPGAs* (Dec 2010), pp. 298–303.

[11] Hsing, Homer. tiny_aes AES core. `http://opencores.org/project,tiny_aes`, 2015.

[12] Hu, Kekai, Wolf, Tilman, Teixeira, Thiago, and Tessier, Russell. System-level security for network processors with hardware monitors. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)* (2014), IEEE, pp. 1–6.

[13] Juels, Ari, and Wattenberg, Martin. A fuzzy commitment scheme. In *Proceedings of the 6th ACM conference on Computer and communications security* (1999), ACM, pp. 28–36.

[14] Kumar, Sandeep S, Guajardo, Jorge, Maes, Roel, Schrijen, Geert-Jan, and Tuyls, Pim. The butterfly PUF protecting IP on every FPGA. In *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on* (2008), IEEE, pp. 67–70.

[15] Maes, Roel. An accurate probabilistic reliability model for silicon PUFs. In *International Workshop on Cryptographic Hardware and Embedded Systems* (2013), Springer, pp. 73–89.

[16] Maes, Roel, Tuyls, Pim, and Verbauwhede, Ingrid. A soft decision helper data algorithm for SRAM PUFs. In *2009 IEEE International Symposium on Information Theory* (2009), IEEE, pp. 2101–2105.

[17] Majzoobi, Mehrdad, Koushanfar, Farinaz, and Devadas, Srinivas. FPGA PUF using programmable delay lines. In *2010 IEEE International Workshop on Information Forensics and Security* (2010), IEEE, pp. 1–6.

[18] Morozov, Sergey, Maiti, Abhranil, and Schaumont, Patrick. An analysis of delay based PUF implementations on FPGA. In *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2010, pp. 382–387.

[19] Rahman, Tauhidur, Forte, Domenic, Fahrny, Jim, and Tehranipoor, Mohammad. Aro-puf: An aging-resistant ring oscillator PUF design. In *Proceedings of the Conference on Design, Automation & Test in Europe* (3001 Leuven, Belgium, Belgium, 2014), DATE '14, European Design and Automation Association, pp. 69:1–69:6.

[20] Rührmair, U., and Holcomb, D. E. PUFs at a glance. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)* (March 2014), pp. 1–6.

[21] Shrikant Vyas, Naveen Kumar Dumpala, Russell Tessier Daniel E. Holcomb. Improving the Efficiency of PUF-Based Key Generation in FPGAs using Variation-Aware Placement. *FPL* (2016).

[22] Simpson, Eric, and Schaumont, Patrick. Offline hardware/software authentication for reconfigurable platforms. In *CHES* (2006), vol. 4249, Springer, pp. 311–323.

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

[24] Suh, G. Edward, O'Donnell, Charles W., and Devadas, Srinivas. Aegis: A single-chip secure processor. *IEEE Design & Test of Computers 24*, 6 (2007), 570–580.

[25] Wikipedia. Pearson product-moment correlation coefficient — wikipedia, the free encyclopedia, 2016. [Online; accessed 20-April-2016].

[26] Xu, Xiaolin, and Holcomb, Daniel. A clockless sequential PUF with autonomous majority voting. In *Proceedings of the 26th Edition on Great Lakes Symposium on VLSI* (New York, NY, USA, 2016), GLSVLSI '16, ACM, pp. 27–32.