Implementing Arbiter Physical Unclonable Function

1st Avinash Ayalasomayajula
Electrical and Computer Engineering
University of Florida
Gainesville, USA
a.ayalasomayajul@ufl.edu
UFID : 0699-6946

2nd Samrat Kautilya Chitimalla
Electrical and Computer Engineering
University of Florida
Gainesville, USA
s.chitimalla@ufl.edu
UFID : 7419-2986

3rd Yashwanth Katta
Electrical and Computer Engineering
University of Florida
Gainesville, USA
yashwantkatta@ufl.edu
UFID : 3451-7972

***Abstract*— Device Authentication has become one of the most explored concepts in recent times. Authentication refers to identifying and authenticating chips, boards etc. Security has become a pressing issue and is taken very seriously. Although various techniques and implementation have come to surface to address this issue, one idea stood out the most. This is a PUF. It stands for Physical Unclonable Function. PUF implementations exploit the uncontrollable variations of integrated circuit manufacturing. The said physical variations can be described as unique and unclonable which are utilized by the PUFs to generate a bit stream unique to each component. A challenge is given to a PUF as an input and we expect to get a one-bit response at the output. Following these concepts, challenge and response pairs are formed for every PUF and are evaluated based on reliability, uniqueness and uniformity. This project is about implementing a 64-bit Arbiter PUF, Feed Forward PUF and XOR PUF. These three PUFs are implemented on a Xilinx Spartan 6 XC6SLX45 FPGA. Xilinx's ISE Integrated Development (IDE) is used to design, synthesize, place and route all the three PUFs. Verilog was the employed language for coding.**

Keywords—Device Authentication, PUF, Process Variation, Arbiter PUF, XOR PUF, Feed-Forward PUF and FPGA.

I. INTRODUCTION

For years security of systems was solely confined to software cryptography. Introduction of software encryption techniques such as AES and DES have paved way to the protection of user data via encryption of data. But was software encryption enough to protect a system? There was a new threat to systems in the form of hardware as well. Introduction of "Hardware Trojans", Recycled ICs and Cloned ICs have risen, and these made systems vulnerable through hardware. This gave rise to the introduction of security measures embedded into the hardware of a system. Security measures were more vital for the protection of secret keys exchanged between systems for data transfers. Non-volatile memories are used to store keys which are then sent to the processor for encryption. But these keys can easily be extracted from these non-volatile memories. Additionally, unsecure environments where there is physical access to the memory are problematic Physical Unclonable Functions (PUF) provides us with extra security as the keys are generated within the device and from a vulnerable non-volatile memory.

A. Physical Unclonable Function (PUF)

A PUF generates keys from within a device by using the process variations that we obtain post fabrication and packing of device. It is inevitable that there are variations in the ICs produced from fabrication (i.e length, width, oxide thickness etc.). This makes properties such as delay along a path different for every IC produced making each IC unique. These variations of each IC are unique and cannot be cloned. We can use these process variations to generate keys.

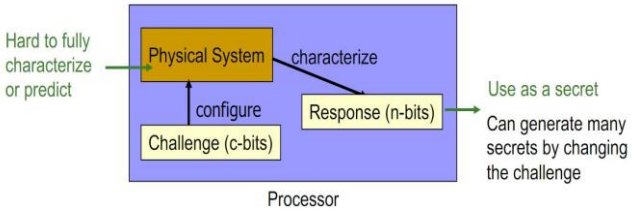

Fig. 1. Block Diagram of a PUF

A PUF can be designed within a device to generate keys and used for authentication. One of the majors that the PUF provides is that is volatile i.e. turns off when power is off but can produce the same keys for a given input upon startup.

By providing an input called challenge(n-bit), the PUF generates an output(n-bit) which is solely determined by the internal configuration of the PUF. These pairs of challenges and responses are known as CRPs

$$f(\text{challenge}) = \text{response}$$

where *f* denotes the internal configuration of PUF.

PUF are used for authenticating devices in two phases. In the 1st phase challenges are given to the PUF and the responses generated are collected. These CRPs are then used to authenticate the devices by giving the challenge bits and then comparing the responses.

This is how the following paper is divided. Section II describes our motivation behind this implementation. Section III details the problem statement. Section IV explains about the Architecture for the PUFs, Section V speaks about the Design Methodology and Principle employed. Section VI is about the Implementation on the FPGA board and Section VII is about the Results obtained. The Section VIII is about the Challenges we faced throughout the course of the project and the final section is about the Conclusion.

II. MOTIVATION

Memories have long been used to store keys for data encryption in devices. These memories being non-volatile store the key information even when the device is switched off. The extraction of keys from these memories has become possible only through cyber attacks but also hardware attacks. Micro-probing and reverse engineering techniques can be used to extract key information by monitoring the interconnects that carry these keys. This major problem has led to the use of Physical Unclonable Functions (PUFs) in devices for key generation and device authentication. PUFs are based on process variation and thus cannot be easily cloned. Due to the increasing costs in the design and manufacture of ASICs, FPGAs are become wildly used due to their lower cost and faster time to market. FPGAs can

implement variety of hardware designs via LUTs present in the FPGA. But the bitstream used to configure the FPGA resides in the non-volatile memory of the FPGA and thus can be prone to an attack. These vulnerabilities have motivated us to try and implement a PUF within an FPGA to increase security.

III. Problem Statement

The main objective of this project is to design, implement and test a set of 64-bit PUFs, which are Arbiter PUF, XOR based arbiter PUF and Feed forward arbiter PUF on a FPGA board. The FPGA board we chose is Xilinx Spartan 6 XC6SLX45 with -CSG324 packaging. To program the behavior of the PUFs we used Xilinx ISE Design Suite 14.7 version. Using this tool proved to be much comfortable in implementing our project. The codes for the project are written in Verilog HDL. 64-bit challenges were given as input and the response bits have been obtained. This comprised the first half of the project which is implementation.

The second phase was about the performance analysis of the PUFs. For this we used MATLAB tool. The performance of the PUF was evaluated based on reliability, uniqueness and uniformity/randomness. The response bits obtained for each PUF are stored in an Excel files. These files are imported in MATLAB and are used for computation.

IV. Architecture

This section describes the three PUFs which we implemented. We discuss about their functionality and design briefly so as to make following sections understandable.

A. *Arbiter PUF*

An Arbiter PUF is composed of two identically configured delay paths that are stimulated by an activating signal. These delay paths are connected to multiple stages and after the last stage, they finally end up at an edge triggered flip-flop known as arbiter[1]. The arbiter measures the difference in the propagation delay of the signal in the said two identically configured delay paths and then determines which signal arrived first[2]. Several PUF response bits can be generated by configuring the delay paths in multiple ways using the challenge inputs. The response which is generated from the arbiter is unique to each to every device implementing the same PUF. A typical multiplexer-based Arbiter PUF is shown in Fig. 2.

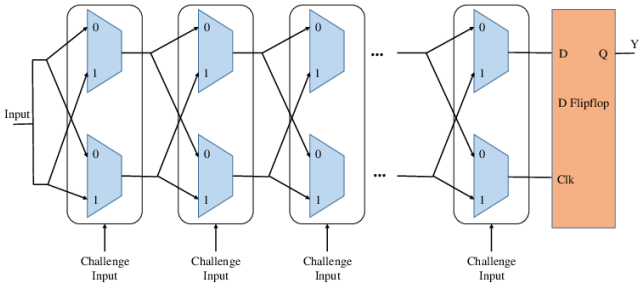

Fig. 2. Block Diagram of Multiplexer-based Arbiter PUF

B. *XOR Arbiter PUF*

The main intent behind designing the XOR-Arbiter PUF is to increase the resistance of Arbiter PUFs against machine learning attacks by adding a non-linear element[3]. The design of an XOR-Arbiter PUF is the same as the standard Arbiter PUF except there is an addition of an XOR- gate at the output of the standard Arbiter PUF. Each of the Arbiter PUFs receive the same challenges and the responses of the n PUFs are XORed to build the final response bit. This adds another level of security as it keeps the internal responses hidden and is easy to implement. XOR-Arbiter PUFs depend on the parallel PUF circuit outputs to be stable in order to get a stable response. The XOR-Arbiter PUF is modeled in Fig. 3.

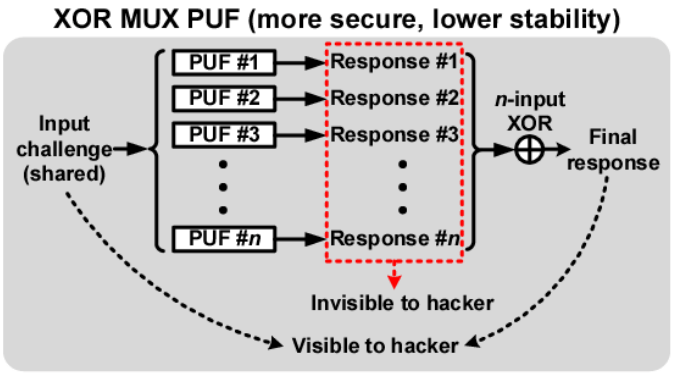

Fig. 3. Block Diagram of XOR-Arbiter PUF

C. *Feed-Forward Arbiter PUF*

Another technique to add non-linear element to the regular arbiter PUF is with the use of Feed-Forward arbiter. Feed-forward arbiter PUFs are among the groups of PUFs which have showed their strength against machine learning modeling unless large computation time is used for machine learning process. Feed-Forward Arbiter PUFs also have the lowest circuit complexity in terms of number of transistors needed for implementation[4] A Feed-Forward Arbiter PUF is implemented by adding feed-forward circuitry to the same standard PUF circuit in order to introduce non-linearity to the original PUF design. This is useful against machine learning attacks because unlike a regular PUF, a feed-forward arbiter PUF has some of the challenge input bits receiving from the outputs of feed-forward arbiters, and each of the feed-forward arbiters takes two output bits of an earlier stage of the arbiter PUF as shown in Fig. 4.

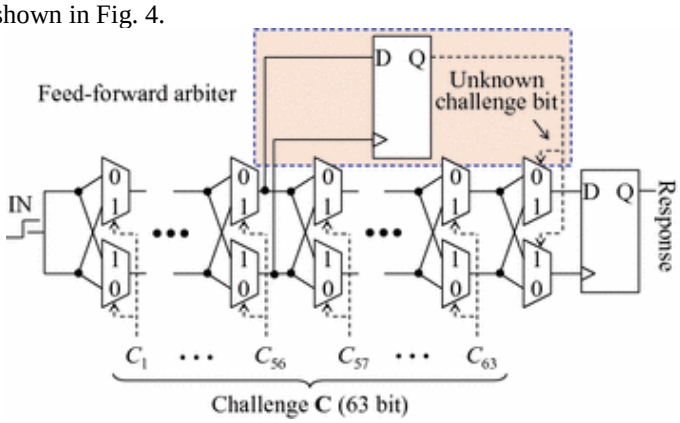

Fig. 4. Block diagram of Feed-forward PUF

V. DESIGN METHODOLOGY

This section is about describing the principle behind the design of the PUFs and explaining the design of the Spartan 6 FPGA.

A. *Principle of Design*

In this paper, we focus on the design and implementation of the delay-based arbiter PUFs. We use delay as a parameter to calculate the response bit. The basic principle is that if the upper path in the PUF design is faster, we get a response bit a logic HIGH and if the lower path reaches first then we get a LOW response bit. So, this way we can get random responses for a stream of challenge bits.

There is a design practice that should be followed to ensure the proper functioning of the design. The stages are connected in such a way that the two paths between each of them are identical and symmetric, establishing a race condition. This ensures the randomness in the output. A *n*-bit challenge is given to an arbiter PUF as a sequence of *n* binary bits where each bit serves as input to each of the *n* stages. The sequence of challenge bits is used as the selection bit for each multiplexer pair in a stage. After the last stage, the arbiter outputs a 1-bit response that depends on the paths taken by the input signal propagated through the multiplexer pairs.

In essence, this circuit of multiplexers creates two delay paths, of equal path length, for a given challenge and produces a response based on which path is faster [1]. The circuit can easily be replicated *x* times and operated in parallel to produce a *x*-bit response. Fig. 2 shows Arbiter PUF. We can see that there is a pulse that is sent through the PUF which is the input and challenge bits are the select lines.

In theory, the signals that travel through each path should reach the arbiter at the same time. However, delays provide the source of randomness for the response since they cannot be easily predicted. For this reason, a single challenge produces a unique response.

B. *Xilinx Spartan 6*

We have used the Atlys circuit board which is based on Xilinx Spartan 6-LX45 FPGA, speed grade 3. The large FPGA and on-board collection of high-end peripherals including Gbit Ethernet, HDMI Video, 128MByte 16-bit DDR2 memory, and USB and audio ports make the Atlys board an ideal host for a wide range of digital systems, including embedded processor designs based on Xilinx's MicroBlaze. Before designing the PUFs we had to do quite some research about the slices and LUTs that the FPGA is providing. Based on this information we were able to design our PUFs. Fig. 5 shows the slice patterns in Spartan 6 FPGA.

This Spartan 6 FPGA offers 6,822 slices[7], each containing four 6- input LUTs and eight flip-flops. The slices are SLICE X, SLICE M and SLICE L. We have used SLICE X for our design because of their dominating number.

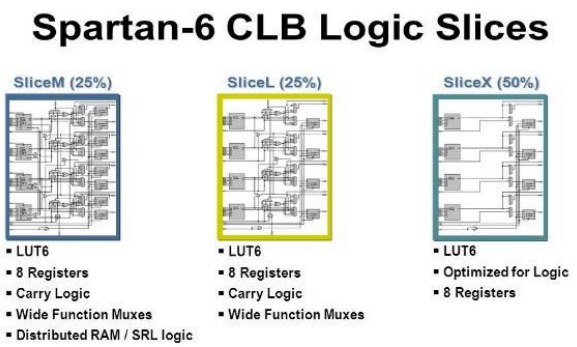

Fig. 5. Spartan 6 CLB Logic Slices

Now we have designed the PUFs using Verilog HDL. Attention to detail ahs been given in designing the PUF. All the multiplexers are manually routed because when the routing is done by the tool, an optimized design is formed which defeats the whole idea of equal path delays. So, to make sure the delay constraint is maintained we used a multiplexer as a structure instead of defining its behavior in the design directly. This ensures the multiplexers to be easily routed manually. Once the slices have been researched, the design was almost done and only implementation was left to be done.

VI. IMPLEMENTATION

The PUFs have been designed using Verilog HDL. Once the delay constraint is taken care of, where we designed the multiplexer as a block which was used repeatably, design is synthesized using Xilinx ISE Design Suite 14.7. We had to make sure that the slices we chose to use for manual routing were correct. So, we used the PlanAhead tool provided by Xilinx to look at the floorplan of the design. This way we could make sense of the design by looking it being displayed on the FPGA design. Fig. 6 shows the schematic of the Arbiter PUF obtained from the tool after synthesizing the design.

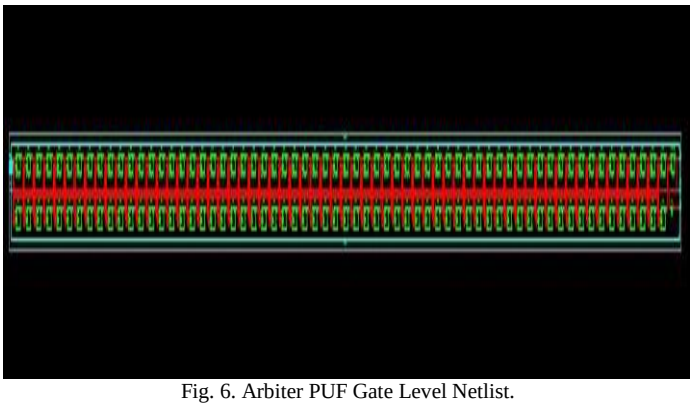
Fig. 6. Arbiter PUF Gate Level Netlist.

Fig. 7 shows the schematic for a Feed Forward PUF and Fig. 8 shows the schematic for a XOR based Arbiter PUF. These are obtained after synthesizing the RTL codes in ISE Design Suite. As we can see from the images, the design resembles as of the architecture we expected to design for all the PUFs which was explained in the previous sections of this paper. The Feed Forward schematic is deviated a bit because of the usage of D Flip Flops between the regular Arbiter PUF.

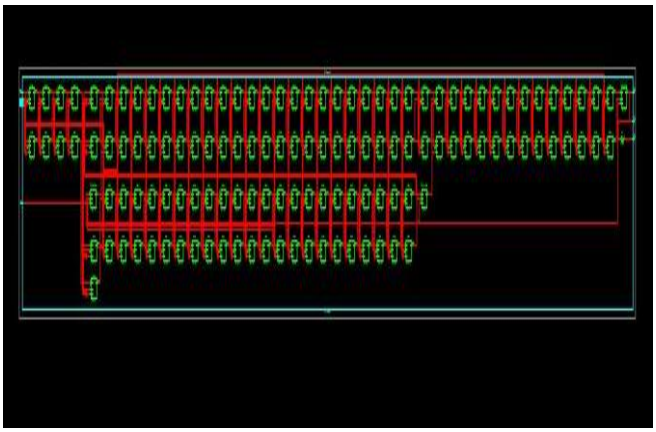

Fig. 7. Feed Forward PUF Gate Level Netlist.

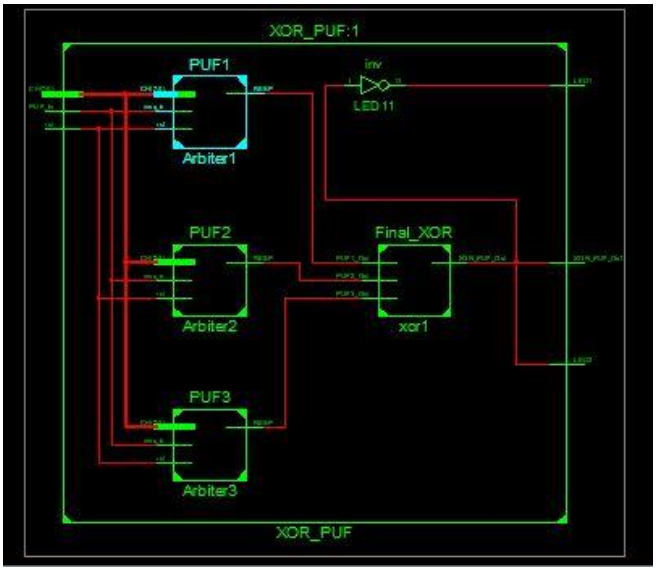

Fig. 8. XOR based Arbiter PUF Gate Level Netlist.

Now that the design has been synthesized, we shall look at the implementation part. For the PUF to work we should give challenge bits to the PUF. We tried using a counter to increment on every clock cycle to give as challenge bits to the PUF. But we faced some difficulties in doing so (This will be clearly explained in the Challenges section). So, we used on board DIP pins to give challenge bits manually to the PUFs. There were only eight DIP pins and we had to use 64-bit challenges. So, we designed the PUFs such a way that the same sequence of bits repeats for every 8 bits in a 64-bit challenge. For example, if we set the input as "11110111" and this is given through the DIP pins. Now the Challenge bits would be "11110111…….11110111". The pattern '11110111' repeats eight times thus forming a 64-bit challenge.

So, this way we could give **256 challenges** to the PUF manually and obtain **256 responses**. Since we have to give a pulse at the beginning of a PUF, we have connected a 100MHz clock to the pulse. All the I/O connections, clock and various other connections are connected using a "**Constraints.ucf**" file. After careful analysis and research, we could get a constraints file for the Spartan 6 LX45 board we were using. Using this constraint file, we connected the input pulse to the clock on board, the DIP bits to the challenge input, the reset pin and the output pins which are connected to the on-board LEDs. We tried using block RAMs, but we faced a few difficulties there as well. So, we read the responses manually and recorded them accordingly.

This was done a few times for each PUF until there was enough data upon which we could do performance analysis. This shall be discussed in the results section.

VII. RESULTS

The PUFs have been synthesized and implemented using Xilinx ISE Design Suite. Then a bit file has been generated. The bit file was uploaded onto the Spartan 6 FPGA. The inputs to the PUF are 64-bit challenges and a pulse at the input to the multiplexers which is given for every challenge. A total of 256 challenges are given to the circuit through the DIP pins on the FPGA board as discussed in the Implementation section . The results are seen through the LEDs on the FPGA. When the response is '1' the right most LED, pin N12, glows and when '0', the left most LED, which is pin 'U18', glows thus giving us the visual proof of the response bits. All the challenge bits have been given manually and the responses are recorded manually. Figure 9(a) and 9(b) shows the real time implementation of the PUFs for a specific challenge.

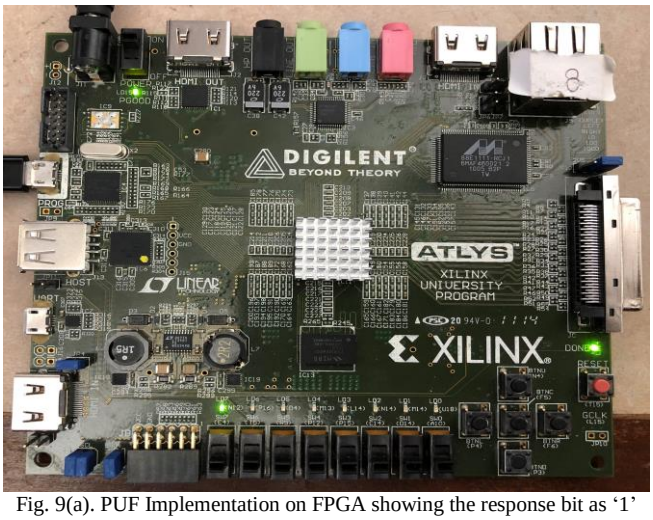

Fig. 9(a). PUF Implementation on FPGA showing the response bit as '1'

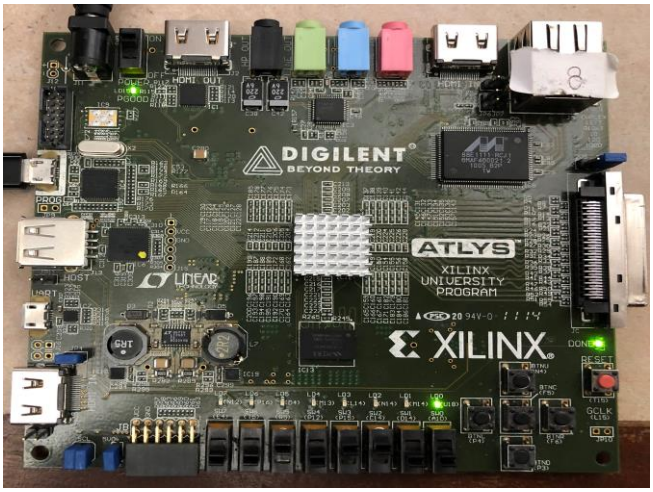

Fig. 9(b). PUF Implementation on FPGA showing the response bit as '0'

As we can see, in Fig. 9(a), the left most LED is ON which indicates that the response is '1' and Fig. 9(b)

shows for response '0'. Once all the responses are recorded and stored in an Excel sheet, we started performance analysis. The performance of the PUFs are evaluated based on reliability, uniqueness and uniformity. MATLAB script has been used for calculating these evaluation metrics.

A. Uniformity

PUF	Uniformity
Arbiter PUF	57.42%
Feed Forward PUF	47.27%
XOR PUF	55.08%

Table 1 : Uniformity for the PUFs

The uniformity for a PUF is computed by computing the total number of responses which are '1' and is divided by the total number of challenges for which all the responses are obtained. This gave us the randomness of which we desired. The results show that Arbiter PUF has the most randomness in its design followed by XOR and Feed Forward PUFs. Thus, we could compute the uniformity based on the obtained data.

B. Reliability

PUF	Reliability
Arbiter PUF	98.83%
Feed Forward PUF	95.31%
XOR PUF	98.05%

Table 2 : Reliability for the PUFs

Reliability is generally measured by varying the temperature and voltage of the device, i.e., to check if the PUF works well under different environment conditions. This can also be done by measuring the responses at different times and repeating this process a few number of times. For each PUF, a total of four tests have been performed. Here each test would mean giving the challenge bits and checking the response. If the response was different then we put an X In that place. Finally, when we had to compute the reliability, we wrote a script to count the number of non-'X's in the response file and divided it by the total number of responses. This was how reliability was computed.

As we can see from Table 2, Arbiter PUF is highly reliable followed by XOR PUF and Feed Forward PUF.

C. Uniqueness

PUF	Uniqueness (Inter-HD)
Arbiter PUF	0.3984
Feed Forward PUF	0.5430
XOR PUF	0.4 to 0.5

Table 3 : Uniqueness for the PUFs

To compute uniqueness of a PUF, we have to calculate the number of bit flips in the responses obtained as a result of applying the same challenge to a PUF which is designed using different slices. We initially used slices X0 and X1 for designing PUFs and thus obtained corresponding responses. To check if the sequence of response is unique, we would have to give the same challenge to a different PUF, i.e., design the PUFs with a different set of slices. So, now we used X2, X3 and lastly X4, X5 slices to obtain two more CRPs. Now we wrote a script in MATLAB to compute the number of bit flips which is the inter-hamming distance.

Table-3, shows that the Inter-HD for the PUFs. For a device to be used for authentication, the Inter-HD should be close to 0.5. This is true in case of Feed Forward and slightly in case of Arbiter. We couldn't do it for XOR as we were using Slice X already and including Slice M was causing some kind of delay issues as we couldn't place all three PUFs without causing some convolution in the design. But we estimate depending on the results we obtained from the PUFs we used, it to be around 0.43.

VIII. Challenges

There were a lot of challenges faced during the course of our project. Initially we had trouble deciding about the FPGA and finalized on Spartan 6 as it is compatible with Xilinx ISE Design Suite. We then got used to the frequent errors and fixed them slowly. A major setback was when we couldn't figure out how to use a counter to give Challenge bits to the PUF. We developed a Counter, but we had trouble with fixing the clock for the counter and synchronizing it with input pulse. Once we could find a work around that, we couldn't produce results on the FPGA using LEDs as the output. So, we tried using a clock divider.

The second roadblock we encountered was how to instantiate a block RAM and how to read the stored responses. We had trouble figuring out the clock to the address generator for the RAM, to the Counter and the input pulse. We were reaching the deadline, so decided to go for the manual inputs. We chose to use DIP pins and obtained some promising responses. But we could only do this for 256 Challenges. But that being said, the design worked with expected results.

IX. Conclusion

Three kinds of PUFs; Arbiter PUF, Feed Forward PUF and XOR based Arbiter PUF, have been implemented on a Spartan 6 LX45 FPGA. 256 CRPs are used for each test and the obtained responses are recorded and used for performance analysis. The three PUFs are evaluated based on Reliability, Uniqueness and Uniformity/Randomness. The obtained results have been studied and have been explained.

For now, we have used manual way of giving Challenges and recording responses. A future extension could be fixing the code where we used a counter and a block RAM to avoid manual implementation.

Acknowledgment

We would like to thank Dr. Mark M. Tehranipoor for his continuous support and valuable guidance throughout the course of the project. We would also like to thank the Teaching Assistant's; Andrew Stern, Jason Vosatka and Mohammad Rahman for their constant support and guidance in making this project take off. We would like to thank them for giving access to the SCAN lab and providing valuable feedback and insight in solving various challenges.

REFERENCES

[1] S. Morozov, A. Maiti, and P. Schaumont, “An Analysis of Delay Based PUF Implementations on FPGA”, 2010.

[2] S. Tajik, E. Dietz, S. Frohmann, J.P. Seifert, D. Nedospasov, C. Helfmeier, C. Boit, H. Dittrich, “Physical Characterization of Arbiter PUFs”, 2014.

[3] Georg T. Becker, “The Gap Between Promise and Reality: On the Insecurity of XOR Arbiter PUFs”, 2015.

[4] M.S. Alkatheiri, Y. Zhuang, “Towards Fast and Accurate Machine Learning Attacks of Feed-Forward Arbiter PUFs”, 2017.

[5] M. Majzoobi, A. Kharaya, F. Koushanfar and S. Devadas , "Automated design, implementation, and evaluation of arbiter-based PUF on FPGA using programmable delay lines" , 2014.

[6] A. Maiti et al., “A systematic method to evaluate and compare the performance of physical unclonable functions,” In Embedded Systems Design with FPGAs, P. Athanas, D. Pnevmatikatos, and N. Sklavos (Eds.). Springer, New York, 245267.

[7] https://reference.digilentinc.com/_media/atlys:atlys:atlys_rm.pdf