Comprehensive study of physical unclonable functions on FPGAs: correlation driven Implementation, deep learning modeling attacks, and countermeasures

by

Mahmoud KhalafAlla

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2020

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Basel Halak Professor, Dept. of Electronics and Computer Science, University of Southampton
Supervisor(s):	Catherine Gebotys Professor, Dept. Electrical and Computer Engineering, University of Waterloo
Internal Member:	Manoj Sachdev Professor, Dept. Electrical and Computer Engineering, University of Waterloo Gordon Agnew Professor, Dept. Electrical and Computer Engineering, University of Waterloo
Internal-External Member:	Alfred Menezes Professor, Dept. of Combinatorics & Optimization, University of Waterloo

Abstract

For more than a decade and a half, Physical Uncolnable Function (PUF) have been presented as a promising hardware security primitive. The idea of exploiting variabilities in hardware fabrication to generate a unique fingerprint for every silicon chip introduced a more secure and cheaper alternative. Other solutions using non-volatile memory to store cryptographic keys, require additional processing steps to generate keys externally, and secure environments to exchange generated keys, which introduce many points of attack that can be used to extract the secret keys.

PUFs were addressed in the literature from different perspectives. Many publications focused on proposing new PUF architectures and evaluation metrics to improve security properties like response uniqueness per chip, response reproducibility of the same PUF input, and response unpredictability using previous input/response pairs. Other research proposed attack schemes to clone the response of PUFs, using conventional machine learning (ML) algorithms, side-channel attacks using power and electromagnetic traces, and fault injection using laser beams and electromagnetic pulses. However, most attack schemes to be successful, imposed some restrictions on the targeted PUF architectures, which make it simpler and easier to attack. Furthermore, they did not propose solid and provable enhancements on these architectures to countermeasure the attacks. This leaves many open questions concerning how to implement perfect secure PUFs especially on Field Programmable Gate Arrays (FPGAs), how to extend previous modeling attack schemes to be successful against more complex PUF architectures (and understand why modeling attacks work) and how to detect and countermeasure these attacks to guarantee that secret data are safe from the attackers.

This Ph.D. dissertation contributes to the state of the art research on physical unclonable functions in several ways. First, the thesis provides a comprehensive analysis of the implementation of secure PUFs on FPGAs using manual placement and manual routing techniques guided by new performance metrics to overcome FPGAs restrictions with minimum hardware and area overhead. Then the impact of Deep Learning (DL) algorithms is studied as a promising modeling attack scheme against complex PUF architectures, which were reported immune to conventional Machine Learning (ML) techniques. Furthermore, it is shown that DL modeling attacks successfully overcome the restrictions imposed by previous research even with the lack of accurate mathematical models of these PUF architectures. Finally, this comprehensive analysis is completed by understanding why deep learning attacks are successful and how to build new PUF architectures and extra circuitry to thwart these types of attacks. This research is important for deploying cheap and efficient hardware security primitives in different fields, including Internet of Things

(IoT) applications, embedded systems, automotive and military equipment. Additionally, it puts more focus on the development of strong intrinsic PUFs which are widely proposed and deployed in many security protocols used for authentication, key establishment, and Oblivious transfer protocols.

Acknowledgements

I would like to thank God the most merciful and generous for helping me finish this thesis. Nothing was possible without his grace and blessing.

I would like to thank my supervisor Professor Catherine Gebotys for her help and guidance over the past five years. She was always available and I would not have completed this work without her continuous support and advice.

I would like to express my appreciation to my committee professors, whom their notes and insightful remarks were helpful and constructive to make this work better.

My appreciation is also extended to my lab colleagues Mustafa Faraj, Mahmoud El-Mohr, and Liao HaoHao for their technical support and fruitful discussions that helped me during my Ph.D. journey.

I would like to offer my special thanks to my friend Abdullah Rashwan for his assistance and advice on solving some machine learning technical problems that I faced during my Ph.D. studies.

I am ultimately grateful to my father Dr. Abdelrahman KhalafAlla and my mother Sohair Megahed for their continuous support and love. I would not have reached this point in my life without their help and care. I am so blessed and privileged that I have you by my side.

I would like to thank my brother Dr. Mohamed KhalafAllah and my sister Dr. Marwa KhalafAllah for their continuous support and guidance since my early childhood. It was a privilege to be their younger brother and they never hesitated to help and support me over every single stage in my life.

The final acknowledgment is for my wife, my soul mate, and my true love Kholoud Shaban and my daughters Khadija and Layan. Your ultimate love and sacrifices day by day is what kept me going on and finish my thesis. You were my shield and shelter during the hard times and you were the source of joy and light that guided me through the way until reaching the successful end.

Dedication

To my fellow free Egyptians who paid the highest price to help our beloved Egypt be free. You will always be remembered and your sacrifices will not go in vain.

Table of Contents

List of Figures

List of Tables

Abbreviations

ANN	Artificial Neural Network 21
APUF	Arbiter PUF 13
ASIC	Application-Specific Integrated Circuits 14
BR-PUF	Bistable Ring PUF 21
CLBs	configuration Logic Blocks 36
CMOS	Complementary Metal–Oxide–Semiconductor 18
CRPs	Challenge Response Pairs 2
DL	Deep Learning iv, 1
ECC	Error Correction Coding 2
ES	Evolution Strategy 56
FC	Fully Connected 75
FF-APUFs	Feed-Forward Arbiter PUFs 14
FPGAs	Field Programmable Gate Arrays iv, 1
FSM	Finite State Machine 87
HD	Hamming Distance 38, 110
IoT	Internet of Things iv, 3

IP	Intellectual Properties 1
LDA	Linear Discriminant Analysis 86
LER	Line Edge Roughness 9
LFSR	Linear-Feedback Shift Register 37
LR	Logistic regression 14
LTF	Linear Threshold Function 83
LUT	LookUp Table 29
ML	Machine Learning iv
MOSFET	Metal Oxide Semiconductor Field Effect Transistor 18
NVM	Non-Volatile Memory 2
PDL	Programmable Delay Lines 29
PGG	Poly-Gate Granularity 9
PUF	Physical Uncolnable Function iv
RDD	Random Discrete Doping 9
RFID	Radio-Frequency Identification 24
RO-PUFs	Ring Oscillator PUFs 16
SPI	Serial Peripheral Interface 37
SRAM	Static Random-Access Memory 18
SVM	Support Vector Machine 14
TBR-PUF	Twisted Bistable Ring PUFs 21
XOR-APUF	XOR Arbiter PUF 14

Chapter 1

Introduction

This chapter presents a brief introduction to PUFs, followed by the dissertation motivation, problem statement, and objectives.

The spread of technological applications that facilitate our tasks and improve interaction and communication among people is ongoing in our world. These applications have an essential role in our daily basis activities, covering a wide range of fields, from military equipment to the personal applications used in cars and smart homes. This growth of our dependence on technology has made security an important issue that needs to be handled properly to maintain the safety of personal data and Intellectual Properties (IP). Hence, security impacts the full stack of modern applications, starting from the top software layers and communication protocols down to the hardware chips.

The field of hardware security focuses on providing cryptographic functionalities on hardware and protecting hardware chips against different types of attacks. For example, attacks may be launched to get an insight into the hardware implementation of a certain intellectual property or read some protected and sensitive information to break cryptographic implementations. This dissertation focuses on analyzing the implementation of secure physical unclonable functions as a promising hardware security primitive on FPGAs. Furthermore, it studies the impact of advanced modeling attacks using DL techniques on complex PUFs architectures that showed significant resistance against conventional machine learning (ML) modeling techniques. Finally, based on the understanding of how DL attacks work, new PUF architectures are implemented to countermeasure these attacks.

1.1 Background and motivation

For more than a decade now, PUFs remains one of the promising proposed approaches to support secure and reliable solutions in hardware security. A PUF as defined in [10] is: “A physical entity that uses production variability to generate a device-specific output which usually is a binary number”. PUFs depend on the variability in the hardware production (e.g. silicon fabrication) process, which cannot be controlled. This variability produces local mismatches among various chip components. As a result, PUF circuits exploit these local mismatches and device intrinsic properties to produce binary values that are unique for every chip. Therefore, PUFs are a chip fingerprint and can be used in a wide range of security applications, such as identification, authentication, and cryptographic key generation.

PUFs have two main advantages compared to other hardware security mechanisms using on-chip and off-chip storage. Firstly, PUFs are more secure because cryptographic keys and secret information are inherently stored within the chip silicon not stored in a digital form on the chip. This property makes PUFs less prone to physical attacks that allow the attacker to read the content of registers and memories on-chip to extract the keys. Also, using off-chip memory to store keys requires a more costly secure environment otherwise, an attacker can listen to the data bus between the external memory and chip and easily extract the keys. Secondly, PUFs are a cheaper solution than storage mechanisms that use costly on-chip Non-Volatile Memory (NVM), or battery-backed RAMs [36]. Moreover, additional costs are needed to provide a secure environment to transfer the generated key back to the chip in case of using an off-chip storage mechanism. Consequently, using PUFs minimizes the number of vulnerable points, at which an attack can be launched. On the other hand, the main PUF disadvantage is that it cannot be erased, unlike key storage schemes. Once a PUF is built on a chip, its outcome cannot be destroyed because it is dependent on chip silicon. Hence, breaking PUF security causes permanent damage in current architectures. PUFs can be used in three main types of applications, identification, authentication, and key generation. PUFs are used to provide unique IDs for every chip without the need to be externally generated and stored in an internal memory as discussed before [30]. Authentication is related to identification since it is the process of verifying the authenticity of a certain ID. This can be done by using Challenge Response Pairs (CRPs), where challenges are the inputs to the PUF circuit, and responses are the designated PUF output [35]. Furthermore, there are many proposals that suggested using PUFs to generate cryptographic keys [79][32][18][57]. However, Error Correction Coding (ECC) techniques and helper data are used to overcome PUFs noisy responses, because generated keys cannot tolerate errors.

It is important to realize that PUFs are not perfect. There exist many challenges to design and deploy PUFs in different applications and these challenges are the key motivations to our research. For example, implementing ideal PUFs on FPGAs faces several challenges due to low-level placement and routing constraints, which affects PUFs architectures and force routing differences to be more dominant on PUFs response than intrinsic chip properties. As a result, the randomness and non-predictability of PUFs response is reduced, which facilitates cloning PUFs using modeling techniques. In addition, PUFs encounter errors in their output due to circuit noise, environmental changes (temperature and voltage differences), and aging effects. Hence, architectural enhancements must be implemented to counter these effects and increase the PUFs reliability performance. Finally, PUFs are not totally prone to attacks aiming at cloning its response. Many successful attack schemes were proposed in the literature using machine learning algorithms, side-channel techniques, and fault injection to break the PUF secret and clone its behavior. Ultimately, although many PUF architectures were proposed in literature last decade, these previously mentioned challenges are still present and PUFs remain active points of research in the hardware security area.

1.2 Problem statement and objectives

Applications of the IoT and embedded devices exist in every aspect of our modern life. Additionally, more personal and sensitive data are generated every minute that needs to be protected against malicious use of unwanted parties. Hence, there is a real need for PUFs as a cryptographic primitive to fulfill the increasing security demands of IoT applications that have limited power and hardware resources.

Although PUFs have been there for nearly two decades now, they are not widely deployed in security applications due to many challenges related to their reliability and vulnerability to attacks. Moreover, part of the PUF use cases requires additional post-processing of helping data to overcome some of its shortcomings, which contradicts its main advantage as a simple self-contained security primitive that can operate in a limited resource environment. In previous research, many modeling attacks were proposed using ML techniques to break PUFs architectures and new PUFs were proposed to countermeasure these attacks. however, there exists limited effort to understand the bounding limit of modeling attacks and the reasoning behind their success in breaking PUFs security. Understanding the reasons for successful attacks will clearly lead the efforts of designing new PUF architectures that are immune to such attacks.

Hence, there are three main objectives of this research. The first objective is to provide a comprehensive study on PUFs implementation on FPGAs driven by manual routing techniques and statistical analysis. The second objective is to advance modeling attacks using deep learning techniques against complex PUF architectures, which showed resistance to conventional ML techniques. Furthermore, the research aims to understand why such attacks work as an initial step to achieve the third objective of developing new PUF architectures that countermeasure modeling attacks while minimizing the overhead needed.

These research objectives face a number of challenges. Firstly, the constraints imposed by FPGAs, which affect implementing balanced and secure PUFs with minimum overhead in terms of hardware, area, and power consumption. Another challenge is extending modeling attacks on complex PUFs (e.g. XOR arbiter PUFs, feed-forward arbiter PUFs) without adding constraints to ease architecture complexity like what was done in previous research [75][76]. Furthermore, providing a modeling technique to attack PUF architectures for which we cannot derive an accurate mathematical model for its operation. Finally, finding computationally efficient techniques to hide the relationship between PUF input challenge bits and delay function in delay-based PUFs is an additional challenge in order to countermeasure attacks on PUFs.

Finally, the target PUFs in this research are delay-based and memory-based PUFs and their variations because these are the types of PUFs that can be realized on FPGAs. Additionally, they are considered among the strong performing PUFs in literature [57][42].

1.3 Dissertation overview

This Dissertation is divided into seven chapters. Chapter 2 gives an overview of PUFs. It specifies PUF types, sources of mismatches in silicon that constitute the theoretical base of electronic PUFs, PUFs properties and use of PUFs in different applications. Chapter 3 discusses the work done on implementing delay-based PUFs on FPGAs. Chapter 4 shows the modeling attacks on PUFs using deep learning techniques. Chapter 5 gives details of new PUF architectures to countermeasure modeling attacks. Chapter 6 is the conclusion and future work.

Chapter 2

Overview of Physical Unclonable Functions (PUFs)

This chapter provides a detailed overview of PUF types and applications. PUFs were categorized from different perspectives in the literature. The first perspective focused on how they function and their operation principle (e.g. electronic Vs. non-electronic PUFs, analog measurement PUFs Vs. digital measurement PUFs, etc. . .) [10][36]. Second perspective categorized PUFs based on their strength against brute force attacks and the complexity of input-output relation (e.g. Strong PUFs Vs. Weak PUFs) [76][32][77][94]. Both perspectives will be discussed in detail in the next subsection to provide reasoning of which types were chosen in our research work. Another subsection gives an overview of PUFs properties, which are used to assess PUFs performance. Finally, the last subsection is dedicated to discussing the deployment of PUFs in different security applications and modes of operation.

2.1 PUF Types

2.1.1 Non-Electronic PUFs

A non-electronic PUF is a one whose construction and/or operation is inherently non-electronic. However, very often electronic and digital techniques will be used at some point anyway to process and store these PUFs' responses in an efficient manner. Optical PUF was introduced by in [71]. As shown in Figure 2.1, its idea is based on reflecting a HeNe

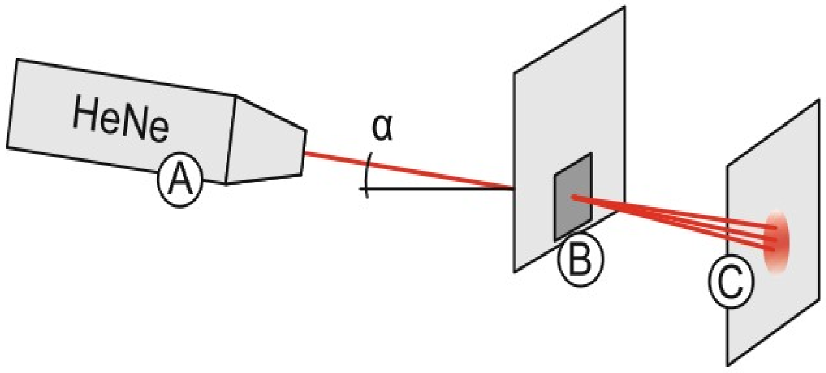


Figure 2.1: Optical PUF [10]

laser beam on a surface to generate a speckle pattern that is captured by a CCD camera for digital processing. A Gabor hash is applied to the observed speckle pattern as a feature extraction procedure. Similarly, Paper PUFs introduced in [12][13], used the reflection of a focused laser beam by the irregular fiber structure of a paper document as fingerprint of that document to prevent forgery. Furthermore, there exist many non-electronic PUFs proposed in literature like CD PUFs, which depends on deviations in measured lengths of lands and pits on CDs [34], Magnetic PUFs [73], which exploit the uniqueness of the particle patterns in magnetic media (e.g., in magnetic swipe cards where they are commercially used to prevent credit card fraud [2]), and RF-PUFs, unlike optical PUFs, it observes near-field scattering of electromagnetic waves using a matrix of RF antennas [19]. Finally, Biological PUF was introduced in [91], which uses the spatio-temporal behavior of colonized T cells as a source of randomness. Figure 2.2 shows the steps of biological PUF operation, where the first step is to create individual wells that contain cultured and suspended T cells. The T cells are allowed to form colonies for ~ 20 hours. Finally, T cell population is imaged using an imaging system with an on-stage incubator, and the processed image is used to generate the Bio-PUF.

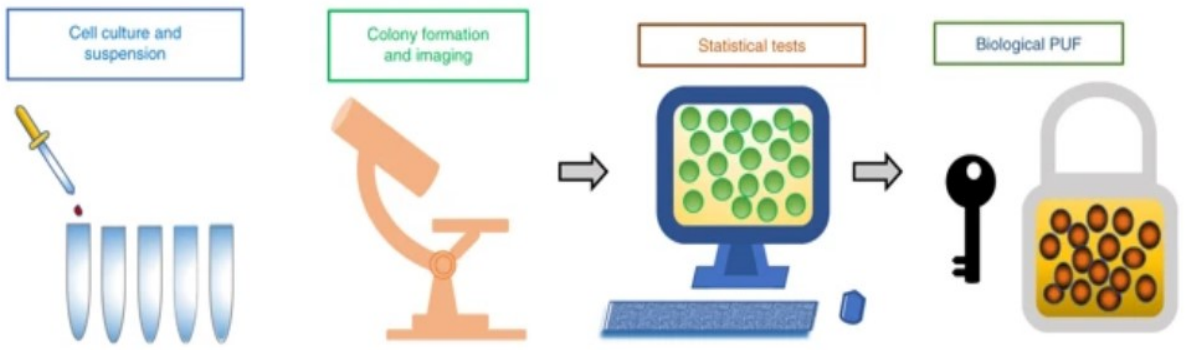


Figure 2.2: Biological PUF [91]

2.1.2 Electronic PUFs

Electronic PUFs depend on the measurement of an electronic quantity, unlike non-electronic PUFs discussed in the previous subsection. The variations of measurements depend on the mismatches between all the components within every chip to generate its unique binary value. As previously mentioned, this value will also differ from chip to chip due to the mismatches differences between chips. Although MOS transistors are the most common circuit element source for mismatches used to realize silicon PUF circuits [10], there also exist other elements used like memristors [65]. The two main types of mismatch sources are extrinsic (global) mismatch sources and (intrinsic) mismatch sources [8]. Global mismatches occur due to unintentional shifts in contemporary process conditions, while local mismatches occur due to the atomic-level differences between devices even though the devices may have identical layout geometry and environment [8].

2.1.2.1 Global mismatches

As mentioned above, global mismatches are associated with the operating dynamics of a modern fabricator. These mismatches include lot-to-lot, wafer-to-wafer, and chip-to-chip variations depending on the cause of variability [8] [10]. Figure 2.3 shows that chip-to-chip delay variation is more significant compared to wafer-to-wafer and within-chip variations, hence chip-to-chip variations have more impact on overall global variations. There are different sources that cause chip-to-chip variability like temperature gradients during thermal annealing, photoresist development, photolithographic variations, and etching [36].

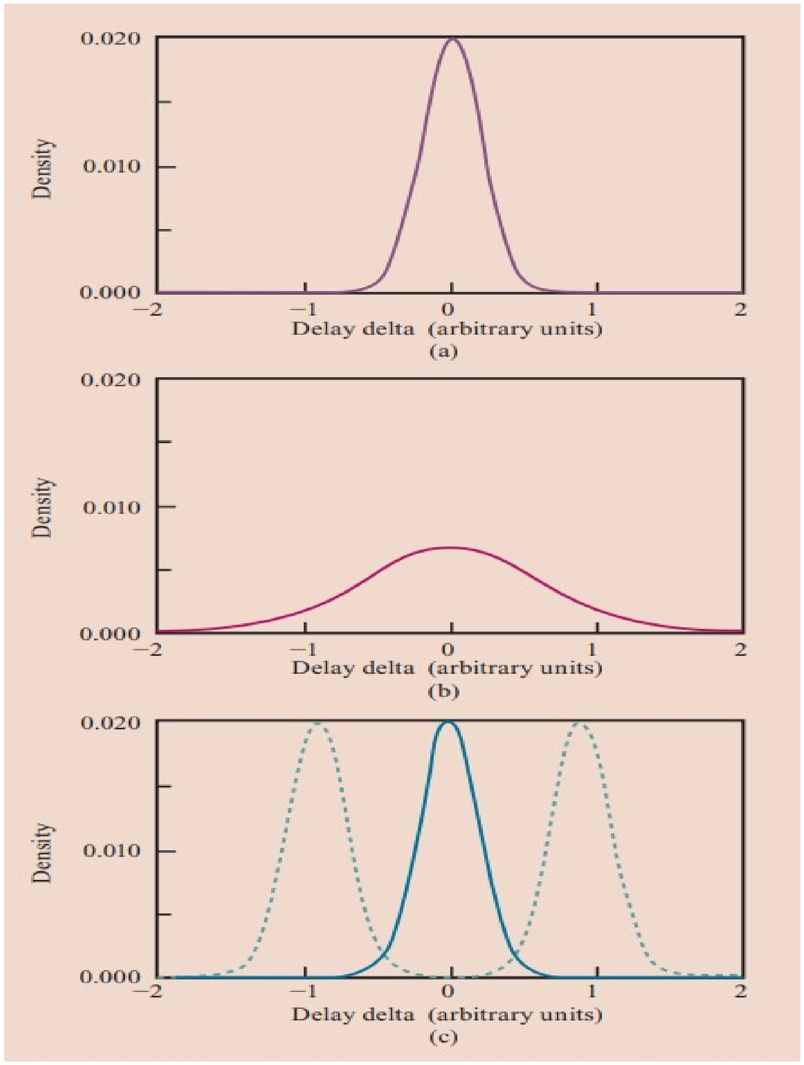


Figure 2.3: Probability distribution for various categories of delay variation on a 90-nm hardware chip: (a) wafer to wafer; (b) chip to chip; (c) within chip. The y-axis on each plot represents the relative density of gates at a given delay delta [8].

Furthermore, hardware chips with the same delay can have different characteristics as shown in Figure 2.4. This figure illustrates across-wafer variability in structures that are indicators of two different transistor attributes: source-drain resistance and gate-to-source and gate-to-drain overlap capacitance. A chip coming from the center of the wafer can exhibit the same nominal delay as a non-center chip, although source/drain resistance, overlap capacitance, and component transistor parameters are different. Such differences may, in turn, cause divergence in circuit response to across-chip voltage and temperature sensitivity. This global effect can lead to biased PUFs, which produce predicted outputs and should be avoided.

2.1.2.2 Local mismatches

Local mismatches occur due to the stochastic atom level differences resulting from inherently process variability [10]. MOSFETs are influenced by three major local mismatch sources [87]:

- Random Discrete Doping (RDD) in the channel region.
- Line Edge Roughness (LER) which describes the gate length variation along the width.
- Poly-Gate Granularity (PGG) which is the effect of poly-silicon grain boundary distribution on the threshold voltage.

Local mismatches are assumed to follow a Gaussian distribution [10], hence PUFs rely on these types of mismatches to realize a random function and produce unique values for every chip. There are other mismatches that are not a result of the variability of hardware fabrication, but they show up during the operation of the hardware chip due to aging effects, voltage drops, and temperature differences among electronic components. PUFs designers must consider these temporal mismatches and make sure it will not affect the correct operation of PUF cells.

2.1.2.3 Analog Electronic PUFs

As mentioned before, analog PUFs exploit the chip variability by constructing circuits that measure analog quantities like voltage, current, power, capacitance, and constitute digital binary output depending on measured values.

2.1.2.3.1 Vt PUFs

Vt PUFs proposed in [52] has a simple principle of operation. Many equally designed transistors are laid out in an addressable array. The addressed transistor drives a resistive load and because of the effect of manufacturing variations on the threshold voltages (Vt) of these transistors, the current through this load will be partially random. The voltage over the load is measured and converted to a bit string with an auto-zeroing comparator.

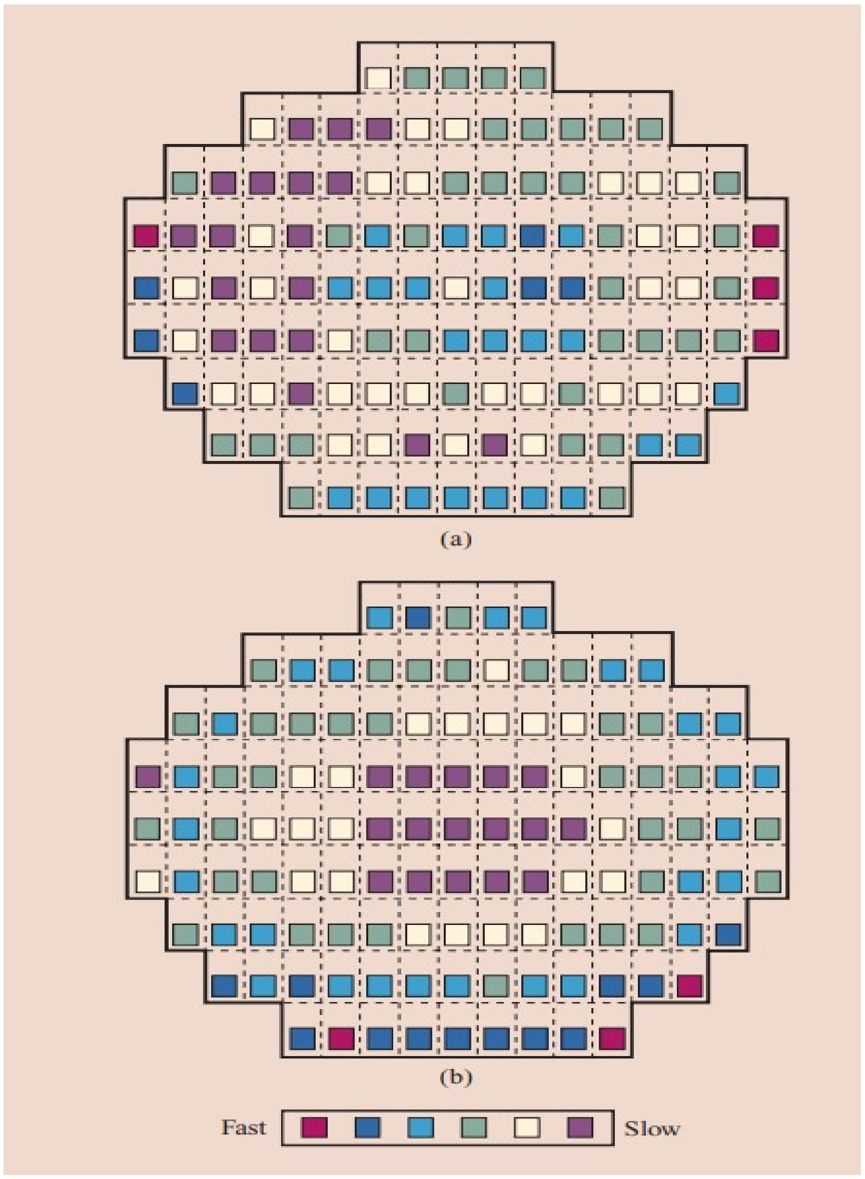


Figure 2.4: Wafer maps showing indicators of (a) source/drain resistance and (b) overlap capacitance [8].

2.1.2.3.2 Power Distribution PUFs

Power distribution PUFs introduced in [39], are based on the resistance variations in the power grid of a chip. Voltage drops and equivalent resistances in the power distribution system are measured using external instruments. It is observed that these electrical parameters are affected by random manufacturing variability, which results in measurement variations.

2.1.2.3.3 Coating PUFs

Coating PUFs were introduced in [88] by considering the randomness of capacitance measurements in comb-shaped sensors in the top metal layer of an integrated circuit. Furthermore, random elements are explicitly introduced by means of a passive dielectric coating sprayed directly on top of the sensors. Moreover, since this coating is opaque and chemically inert, it offers strong protection against physical attacks as well. Figure 2.5 shows the basic operation of coating PUF.

2.1.2.3.4 LC PUFs

LC PUFs proposed in [33] use capacitance as the analog value to measure. It is constructed as a small ($\approx 1mm^2$) glass plate with a metal plate on each side, forming a capacitor, serially chained with a metal coil on the plate acting as an inductive component. Together they form a passive LC circuit that will absorb an amount of power when placed in an RF field. A frequency sweep reveals the resonance frequencies of the circuit, which depend on the exact values of the capacitive and inductive component. Due to manufacturing variations, this resonance peak will be slightly different for equally constructed circuits.

2.1.2.4 Intrinsic PUFs

The term intrinsic PUF was introduced in [58] to describe electronic PUFs which have two main properties:

- All measurement equipment is fully integrated into the embedding device. So PUF responses can be extracted by hardware without the need for external instruments and without exchanging challenge and response messages outside the chip.

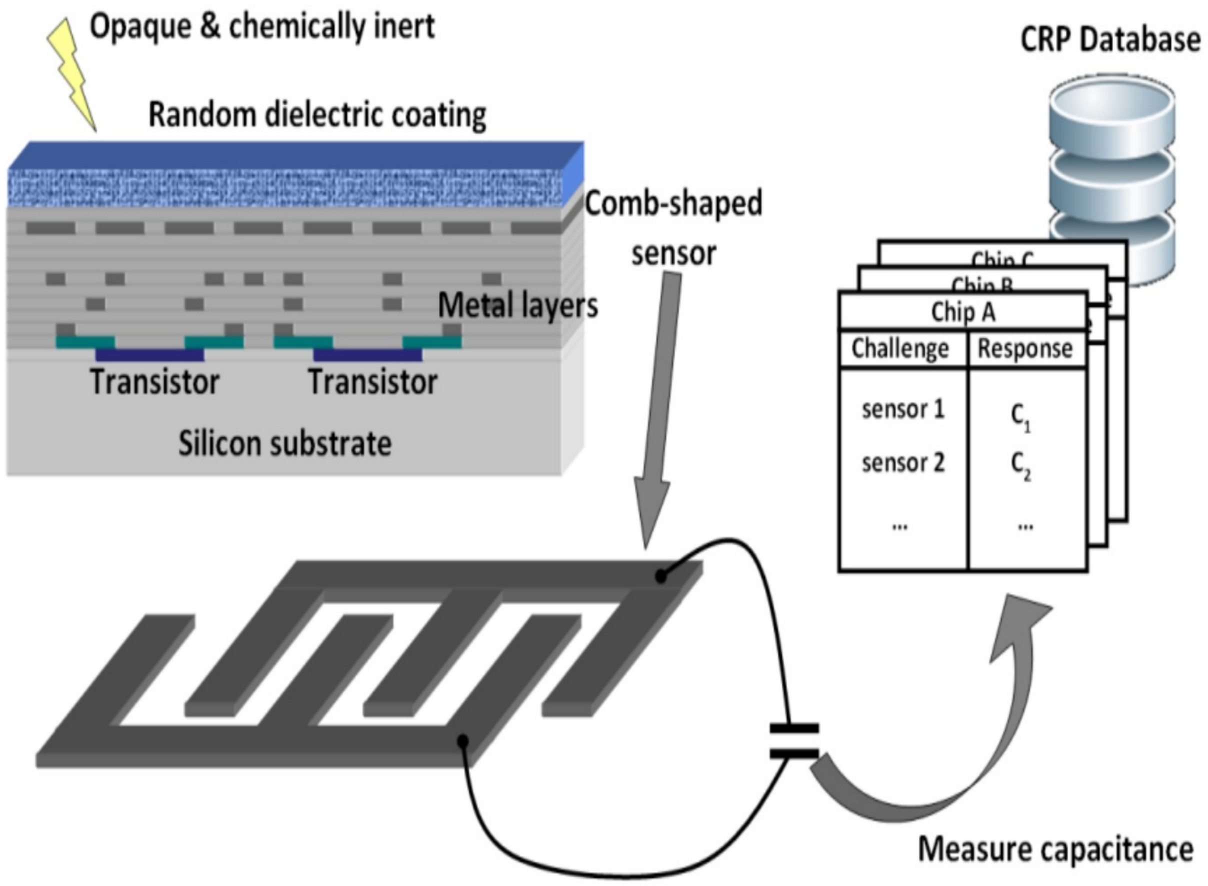


Figure 2.5: Coating PUF basic operation [58].

- PUF architecture should consist of procedures and primitives that are naturally available for the manufacturing process of the embedding device. So PUF construction does not require additional overhead like extra manufacturing steps or specialized components.

As a result, Intrinsic PUFs are the most widely proposed architectures in literature because it is more secure and easier to realize on silicon chips without equipment and processing overhead. There exist two main types of intrinsic PUFs, where they both depend on delay measurements by either using an arbitration circuit or the bi-stability property of memory cells.

2.1.2.4.1 Delay-Based Intrinsic PUFs

This subsection provides an overview of the main categories of delay-based intrinsic PUFs.

2.1.2.4.1.1 Arbiter PUF

The basic idea of Arbiter PUF (APUF) is to create two identical paths on a chip and introduce a digital race between these paths. An arbiter circuit is realized to determine the winner of this race. Since the paths are symmetrical, so the result will not be known in advance. The variations in chip fabrications will affect the physical parameters that control the exact delay of every path, hence the arbiter output will be random and unique for every chip. Figure 2.6 shows the first design proposed in [51], [17] using switch blocks

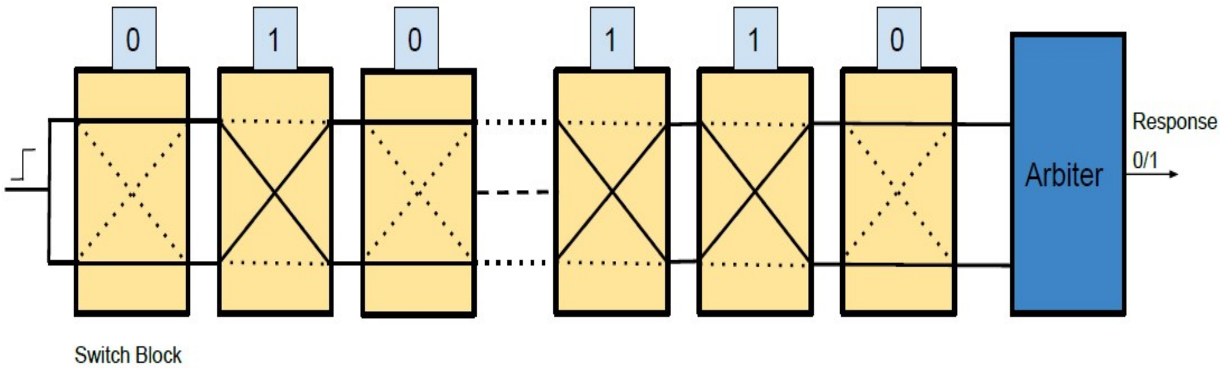


Figure 2.6: Basic operation of Arbiter PUF.

to implement the identical paths and latch or flip-flop to implement the arbiter circuit. Every switch block has two inputs and two outputs. A parameter challenge bit determines if the switch block will allow a straight or switched connection to the next block. The parameter settings of switch blocks are the PUF challenge, and the arbiter output is the PUF response. After setting the paths using challenge bits, an input high signal is propagated through the stages until it reaches the arbiter. An edge-triggered flipflop does the arbitration by connecting one path to its input data and the other path to the input clock. If the first path is faster, input data will be set to ‘1’ before the positive edge reaches the input clock and PUF response becomes ‘1’. On the contrary, if the path feeding the input clock is faster, the edge change will occur before the input signal is set to ‘1’ and PUF response will be ‘0’. The main problem of the initial version of the arbiter PUF is the assumed linear relationship between challenge bits and PUF response. The total path delay may be represented by the sum of all switch blocks delays. Hence, this design was easily broken by machine learning modeling attacks. Using conventional machine learning techniques like Logistic regression (LR), and Support Vector Machine (SVM), mathematical models were built to successfully predict the PUF output after observing a number of CRPs. These modeling attacks were successful and could achieve 99% prediction accuracy after observing 6500 CRPs on Application-Specific Integrated Circuits (ASIC) and FPGA implementations of 64 stage APUF and achieving the same accuracy using 78000 CRPs on ASIC and FPGA implementations of 128 stage APUF [76]. Research on APUFs continued to find countermeasures against modeling attacks by increasing the complexity of challenge response relationship. Adding non-linearities in the delay paths makes it harder to predict APUF responses. Consequently, many variations of APUFs with more complex architectures were proposed in the literature. For example, Feed-Forward Arbiter PUFs (FF-APUFs), proposed in [17] [62] is shown in Figure 2.7. It adds non-linearity by adding loops in the arbiter paths. This results in making some challenge inputs determined by intermediate arbiters that evaluate the delay at an intermediate point of the delay path. Hence these stages inputs are not directly dependent on input challenge bits, but they depend on the accumulated delay of the previous stages. Another variation of APUF architecture proposed in [82] is XOR Arbiter PUF (XOR-APUF). It introduces more parameters and increases the non-linearity of the PUF architecture using the XOR function. The main idea is to use many standard APUFs in parallel. The same input challenge vector is applied to all PUFs and their outcomes are XORed together to produce XOR-APUF response. Furthermore, other proposals suggested adding non-linearity by combining the use of XOR functions and complicate challenge response relationship as shown in Figure 2.8 of interleaved APUF architecture proposed by Majzoobi M. et al in [62]. Its architecture uses R parallel rows of APUFs, where R is an even integer. The challenge bits are interleaved between odd and even rows so that the last challenge bit of the PUFs located in even-numbered rows are connected to the first challenge bit of the PUFs in the odd rows. Generally, the i-th challenge bit of the PUFs in even rows are connected to the (N - i)-th challenge bit of the PUFs in odd rows where N is the total number of challenges in one row. Eventually, the outputs are combined using R-input XOR gates and leave-one-out logic so that a structure consisting of R + 1 rows, R 'R-input' XORs generate R PUF responses. These architectural modifications make it difficult to model the PUFs using conventional machine learning techniques, but they don't fully prevent it. More details on this matter are provided in chapter 4.

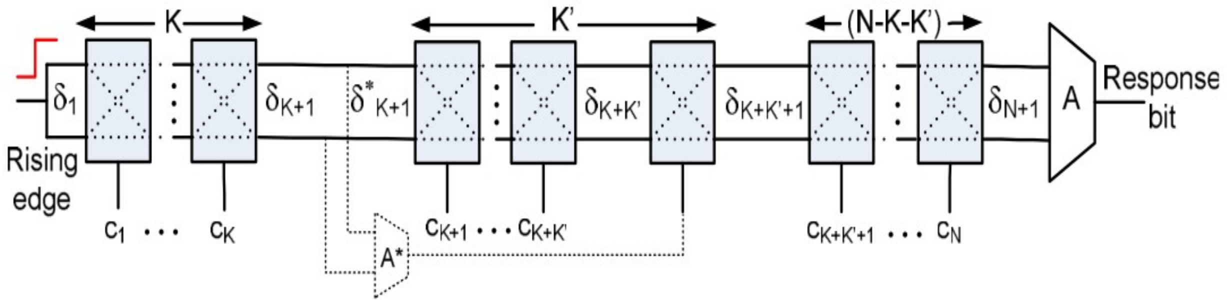


Figure 2.7: The feed-forward PUF architecture [62].

Other approaches adopted the use of extra circuitry and/or hash functions to either complicate challenge response relationship or use the output response as the input seed to the hash function. Figure 2.9 shows lightweight secure PUF introduced in [61]. It is like XOR PUFs in the sense that it includes several arbiter PUFs in parallel. The difference is that it adds more logic at the input and output of the PUF to hide the details of challenge response mapping. Hence, for every arbiter PUF, a unique challenge input is derived from the global input challenge bits, opposite to the XOR arbiter case where all arbiter PUFs have the same challenge input. It is obvious that this technique requires more resources in terms of hardware and power consumption and takes more time to generate PUF output.

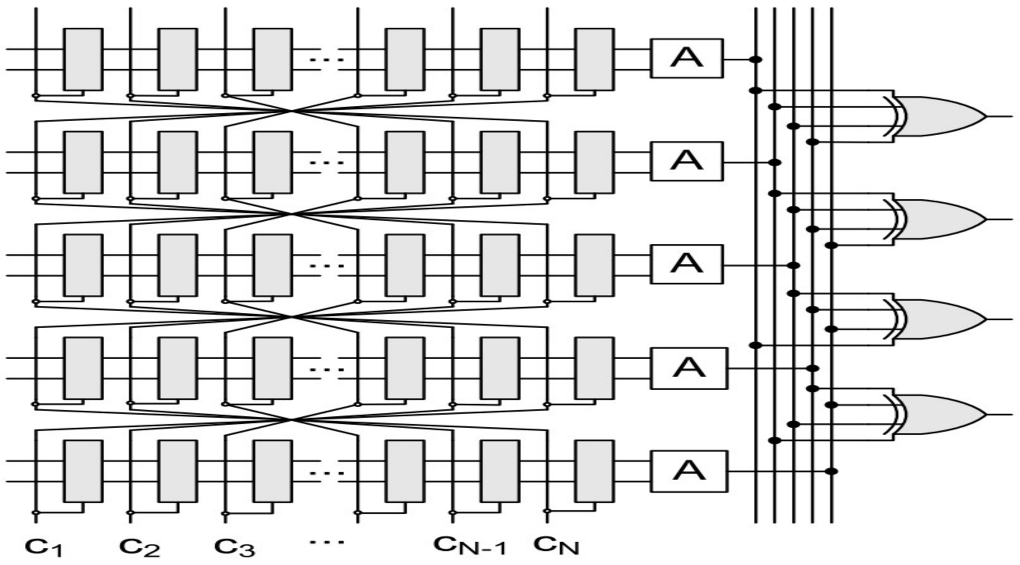


Figure 2.8: Interleaved APUF architecture [62].

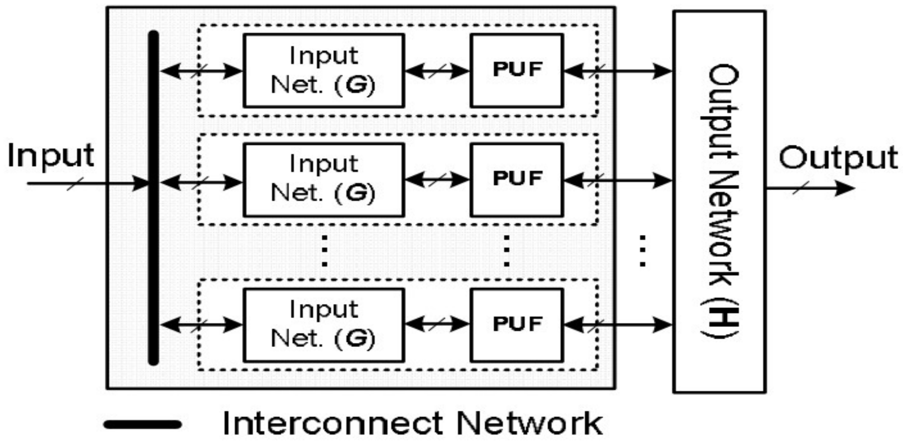


Figure 2.9: Lightweight secure PUF architecture [61].

2.1.2.4.1.2 Ring Oscillator PUF

Ring Oscillator PUFs (RO-PUFs) use another approach to measure the delay difference between symmetric paths. A ring oscillator circuit inverts the output of the digital delay line and feeds it back to the input, hence measuring the frequency generated by every path is likely the same as measuring the exact path delay. Furthermore, the exact frequency will be partially random and dependent on every chip.

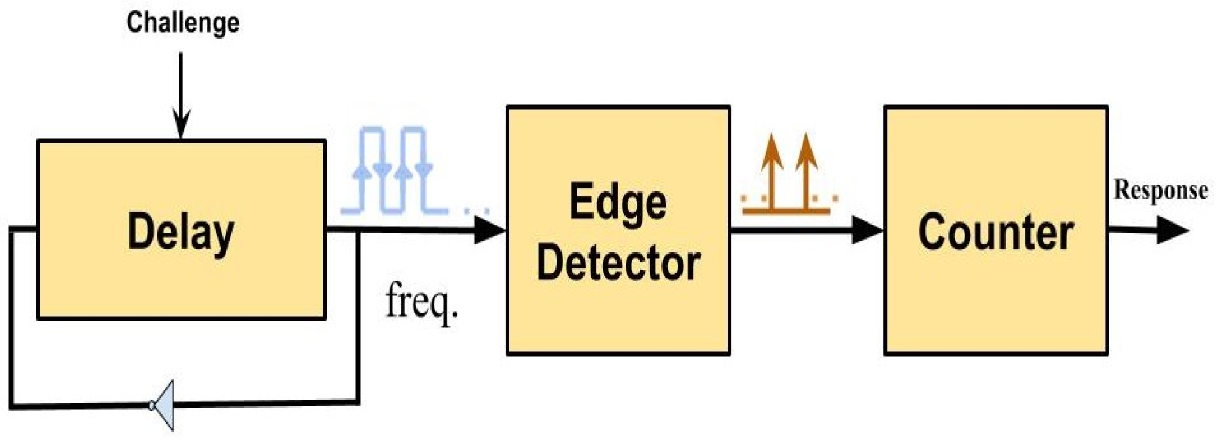


Figure 2.10: Basic operation of Ring Oscillator PUF [58].

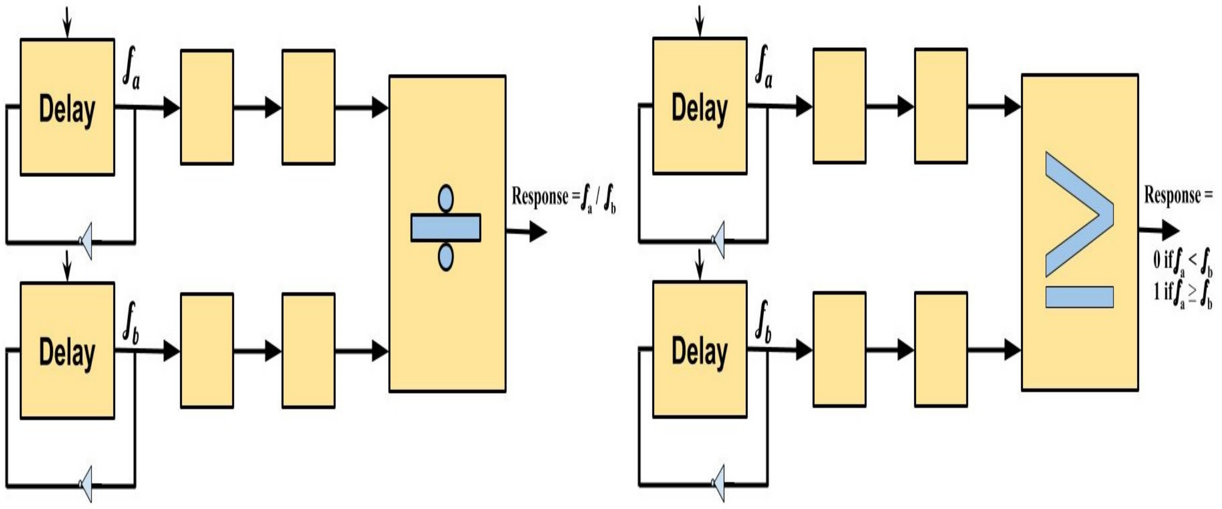


Figure 2.11: Ring oscillator with division compensation and comparator compensation respectively [58].

As shown in Figure 2.10 , the basic architecture of ring oscillator PUF includes an edge detector of the positive oscillator edges, and a counter of these edges. The counter output is considered the PUF response. The die temperature and supply voltage have a non-negligible impact on delay-based PUFs [58]. It is more apparent in ring oscillator PUFs than arbiter PUFs because, in the latter type, a differential measurement is implicitly performed by considering the two delay lines simultaneously. In ring oscillator PUFs, compensation is needed to overcome this impact. Figure 2.11 shows two compensation techniques proposed in [28], [29]. The first one is to divide the obtained counter values of two simultaneously measured oscillators. Hence, The PUF response will be more robust. The second technique proposed in [82] implements several oscillators in parallel and the challenge is to choose two of these oscillators to measure their frequency. A comparator is used at the final level to determine the final response. The main problem with ring-oscillator PUFs is their relatively small input challenges space, which is equal to log2(number of oscillators). Hence, it is considered as a weak PUF and must use extra processing (e.g. hash functions) to hide and protect the PUF output against modeling and side-channel attacks. These attacks proved to be successful against this type of PUFs as will be discussed in chapter 4. However, other variants like RO sum PUF [98] were introduced to increase the PUF CRP space.

2.1.2.4.2 Memory-Based Intrinsic PUFs

Memory-based PUFs are another class of intrinsic PUFs, where unique responses are generated using settling state of digital memory primitives. A digital memory cell is typically a digital circuit with more than one logically stable state. By residing in one of its stable states it can store information, e.g., one binary digit in case of two possible stable states. However, if the element is brought into an unstable state, it is not clear what will happen. It might start oscillating between unstable states or it might converge back to one of its stable states. In the latter case, it is observed that specific cells heavily prefer certain stable states over others. Moreover, this effect can often not be explained by the logic implementation of the cell, but it turns out that internal physical mismatch, e.g., caused by manufacturing variation, plays a role in this. For this reason, the meta-stability nature of memory cells is a good candidate for a PUF response.

2.1.2.4.2.1 SRAM PUFs

Static Random-Access Memory (SRAM) is a type of digital memory consisting of cells each capable of storing one binary digit. An SRAM cell, as shown in Figure 2.12 is logically constructed as two cross-coupled inverters. Hence, leading to two stable states. In regular Complementary Metal–Oxide–Semiconductor (CMOS) technology, this circuit is implemented with four Metal Oxide Semiconductor Field Effect Transistor (MOSFET)s, and an additional two MOSFETs are used for read/write access as shown in Figure 2.12. It is not clear from the logical description of the cell at what state it will be right after the power-up of the memory. It is observed that some cells preferably power-up storing a zero, others preferably power-up storing a one, and some cells have no real preference, but the distribution of these three types of cells over the complete memory is random.

As it turns out, the random physical mismatch in the cell, caused by manufacturing variability, determines the power-up behavior. It forces a cell to 0 or 1 during power-up depending on the sign of the mismatch. If the mismatch is very small, the power-up state is determined by stochastic noise in the circuit and will be random without a real preference. The SRAM PUFs were proposed in [32] and [40] as a promising security primitive that can be easily implemented on FPGAs, but it turns out that this type of PUFs faces two problems:

- In most common FPGAs, all SRAM cells are hard-reset to zero directly after power-up. Hence, all randomness is lost.
- It is not efficient to power-up the device every time a PUF response is required.

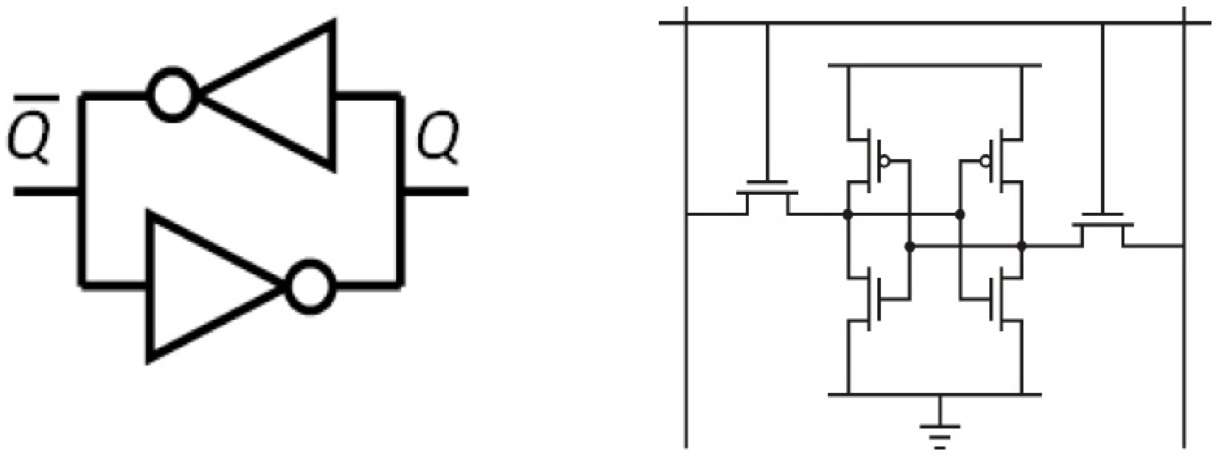


Figure 2.12: Logical and electric representation of a RAM cell [58].

However, memory-based PUFs are more resistant to modelling attacks [58], because the physical random elements contributing to different CRPs are mostly independent of each other. Consequently, other similar schemes were proposed to mimic SRAM operation and avoid its previously mentioned problems.

2.1.2.4.2.2 Butterfly PUFs

Butterfly PUFs were introduced in [50]. The behavior of an SRAM cell is mimicked in the FPGA reconfigurable logic by cross-coupling two transparent data latches. The butterfly PUF cell schematic is shown in Figure 2.13. The circuit allows two logically stable states. However, using the clear/preset functionality of the latches, an unstable state can be introduced after which the circuit converges back to one of the two stable states. This is comparable to the convergence for SRAM cells after power-up but without the need for an actual device power-up. The preferred stabilizing state of such a butterfly PUF cell is determined by the physical mismatch between the latches and the cross-coupling interconnect. It must be noted that due to the discrete routing options of FPGAs, it is not trivial to implement the cell in such a way that the mismatch by design is small. This is a necessary condition if one wants the random mismatch caused by manufacturing variability to have any effect.

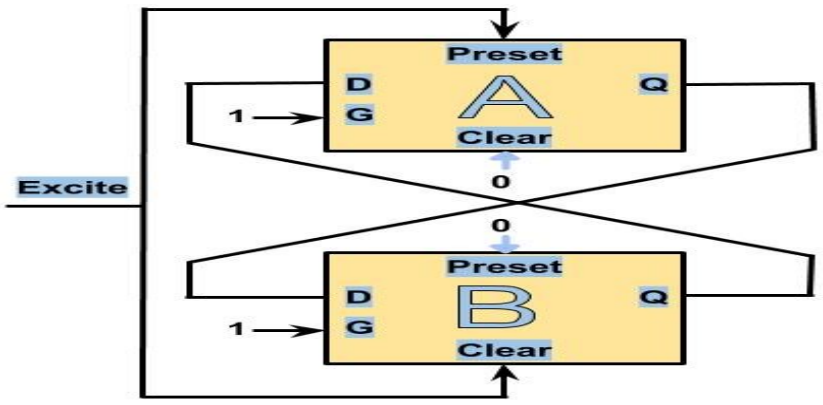


Figure 2.13: ButterFly PUF schematic circuit [58].

2.1.2.4.2.3 Latch PUFs

Latch PUF is an IC identification technique proposed in [81], which is very similar to SRAM PUFs and butterfly PUFs. Instead of cross-coupling two inverters or two latches, two NOR gates are cross-coupled as shown in Figure 2.14, constituting a simple NOR latch. By asserting a reset signal, this latch becomes unstable and then converges to a stable state depending on the internal mismatch between the electronic components. Equivalently to SRAM PUFs and butterfly PUFs, this can be used to build a PUF. Also, Flip-Flop PUFs introduced in [56] uses a similar technique.

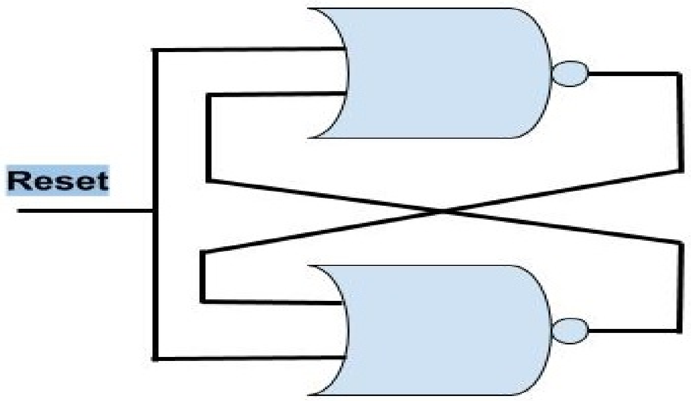


Figure 2.14: Latch PUF schematic circuit.

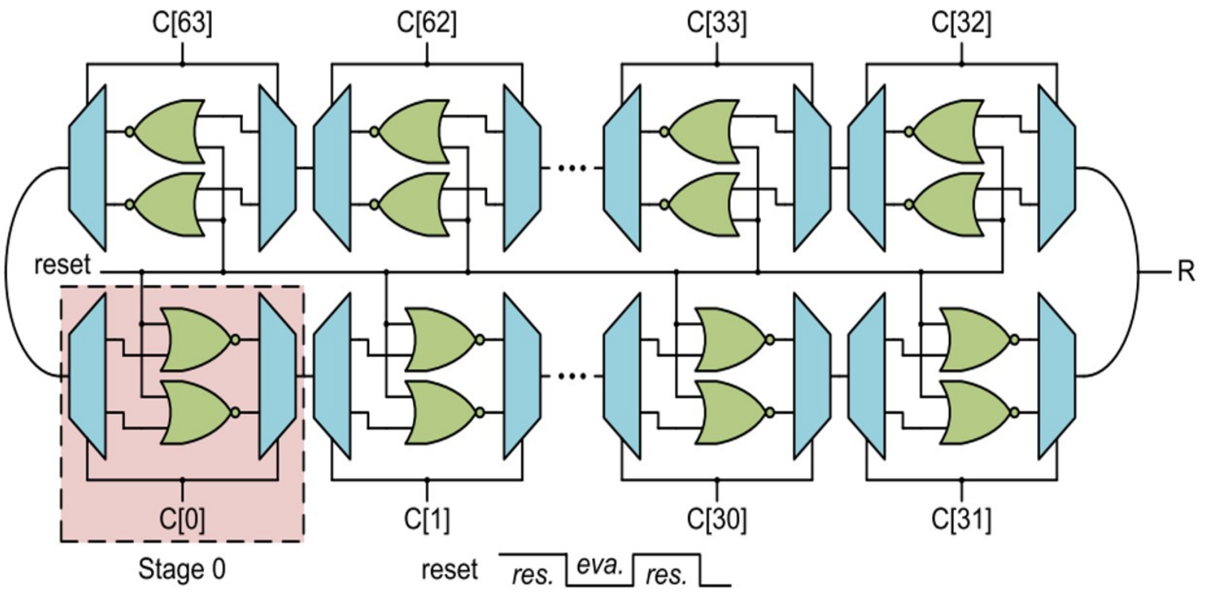


Figure 2.15: BR-PUF architecture [16].

2.1.2.4.3 Hybrid Intrinsic PUFs

Proposals of new PUF architectures tried to merge between delay-based and memory-based approaches to design a stronger PUF with more resistance against modelling attacks and large challenge space, so CRPs cannot be exhaustively read by the attackers. One of these proposals is Bistable Ring PUF (BR-PUF), introduced in [16][15]. Its basic idea is that the output of any given inverter ring with an even number of inverters has only two possible stable states. This is similar to memory-based PUFs operation except that challenge bits are inserted to select which path to be used at every stage as shown in Figure 2.14. The sequence of operations starts by setting reset signal to 1, then apply a 64-bit challenge signal and wait for the bistable ring stages to be in 0 state, then set reset signal to low and wait for the ring output to be stable before reading it out. One problem of BR-PUFs is that it takes a longer time to stabilize, which is an undesirable property of PUFs. Furthermore, BR-PUFs implementations on FPGAs showed an output bias problem as reported in [78]. As a result, other variations of BR-PUFs were proposed like Twisted Bistable Ring PUFs (TBR-PUF) [78] and XOR BR-PUFs in [94]. The latter was proposed after successful modeling attacks were reported against BR-PUFs and TBR-PUFs using SVM and single layer Artificial Neural Network (ANN) (More details on these attacks in chapter 4).

2.1.2.5 Strong PUFs Vs. Weak PUFs

As mentioned earlier, the notation of strong PUFs vs. weak PUFs has been widely used in literature to distinguish between different PUFs architectures based on their input-output space and their vulnerability against brute force attacks. Weak PUFs essentially have a limited challenge space, which allows an attacker to exhaustively read out all CRPs if possible. Memory-based PUFs and some variations of RO-PUFs are examples of weak PUFs. They are mainly used for internal key derivation in security hardware under the assumption that attackers must not be able to access the internal response of the PUF. Additional processing and circuitry might be needed to fulfill this assumption like using hash functions to hide PUF response and hardware replications to overcome noisy outputs. This adds timing, hardware, and power consumption overheads to realize the required security function. On the other hand, strong PUFs, like APUF, BR-PUF, have a very large challenge space that cannot be exhaustively read out by attackers. Their challenge-response mechanism should be complex in the sense that it is hard to derive unknown CRPs from a set of known CRPs using modeling attacks. Unlike weak PUFs, strong PUFs usually allow access to its CRP interface, i.e., anyone holding the PUF or the PUF embedding hardware can apply challenges and read out responses. Recently, strong PUFs have turned out to be a very versatile cryptographic and security primitive [94]. It can be employed for internal key derivation, like weak PUFs. They can also be deployed in different advanced cryptographic protocols, ranging from identification [71][52] to key exchange [11][90] to oblivious transfer [11]. Hence in this research, we focus our analysis on intrinsic PUFs because they can be realized on FPGAs. Furthermore, we add more highlights on strong intrinsic PUFs because they show a promising comprehensive solution in different cryptographic applications with minimum hardware overhead and power consumption.

2.2 PUFs properties

PUFs must conform to some specific properties so that they can fulfill security requirements to be deployed in cryptographic applications. These properties assess the uniqueness, randomness, and unpredictability of PUFs. Table 2.1 gives a summary of these properties and their values for an ideal secure PUF.

All performance metrics used to assess PUFs architecture compare their values to the ideal values presented in Table 2.1. This research uses the same methodology to assess implemented PUFs performance and suggests new metrics to guide the designer through implementing more secure PUFs on FPGAs as will be demonstrated in chapter 3.

Table 2.1: PUFs properties and ideal values.

Property	Description	Identifier	Ideal Value
Uniformity	The percentage of 1's and 0's in PUF response	μ	50%
Reliability	The percentage of erroneous readings of the same response bit (in normal conditions and in environmental variations of temperature and voltage).	N/A	0%
Correlation between bits	Correlation between PUFs responses on the same chip. Ideal PUFs should not have neighboring outputs that have influence on each other.	Rxx	0
Uniqueness	Correlation between PUFs responses on different chips.	HDinter	50%

2.3 PUF applications

As was mentioned in the previous chapter, PUFs have two main advantages over other hardware security mechanisms that use on-chip and off-chip storage. Firstly, providing a secure way to inherently store sensitive data within the chip silicon. Furthermore, PUFs provide a cheaper solution than expensive storage mechanisms that use on-chip NVM. Moreover, additional costs are needed to provide a secure environment to transfer the generated key back to the chip in case of using an off-chip storage mechanism. Consequently, using PUFs minimizes the number of vulnerable points, at which an attack can be launched. Hence, there is a wide spectrum of applications, in which, PUFs can be deployed. In this section, three main applications of PUFs are discussed [10][58]. Namely, Identification, authentication, and key generation. This subsection gives a brief overview of the use of PUFs in these applications

2.3.1 Identification

The identification process assigns IDs to unknown entities, so every entity must have its unique ID. When PUFs are used in identification systems, they eliminate the need to generate IDs externally and replace the internal NVM on which the ID would be stored. Hence, PUFs can reduce costs by reducing fabrication complexity and by providing the ID from the intrinsic properties of the chip. This is a huge advantage for applications

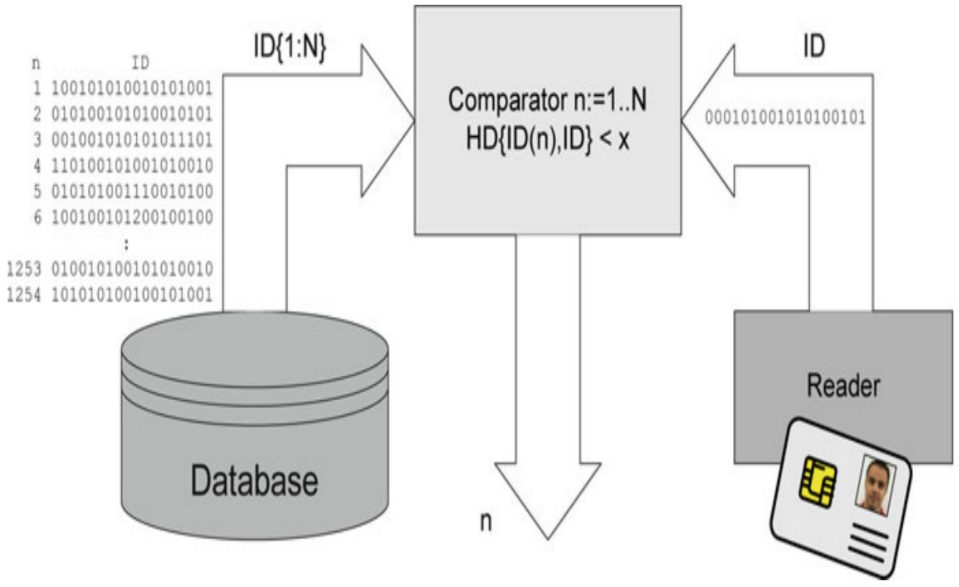


Figure 2.16: PUFs deployed in identification process [10].

requiring cheap chips and minimum processing and power consumption. For example, the use of Radio-Frequency Identification (RFID) tags instead of bar codes. Figure 2.16 shows how PUFs can be deployed in identification applications. The identification system must be tolerant because the output of PUFs is noisy. Thus, the identification starts with the enrollment phase, where the output of the PUF is stored in a database. During a regular identification process, the PUF is read out again. The received response is accepted if its hamming distance with a certain entry in the database is less than a specific tolerance threshold.

2.3.2 Authentication

Authentication is the process of verifying a claimed ID of a certain entity. The use of PUFs in the authentication process, introduce the idea of using CRPs within an authentication protocol. Figure 2.17 shows an authentication process using PUFs with hardware CRPs. Challenges are considered as questions sent to the concerned entity, and PUF response is the correct answer to this question. This requires that PUFs must have a large challenge space so an attacker cannot replay or readout all CRP pairs. Furthermore, error correction codes and helper data are used to overcome noisy PUF responses because the identification process is not error-tolerant.

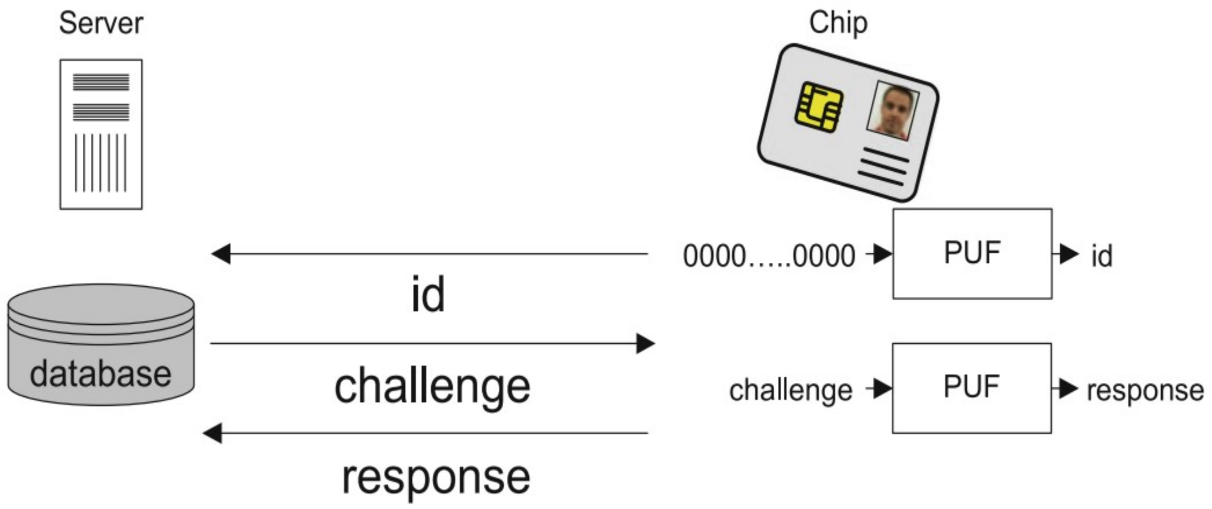


Figure 2.17: Authentication using PUFs with HW CRPs [10].

Another approach using PUFs with SW CRPs (responses are not generated by hardware PUF) suggested that PUF responses can be used as private keys to an encryption algorithm whose output is the response to the identification question as shown in Figure 2.18. The server side will decrypt the response using a public key to make sure it is the expected value.

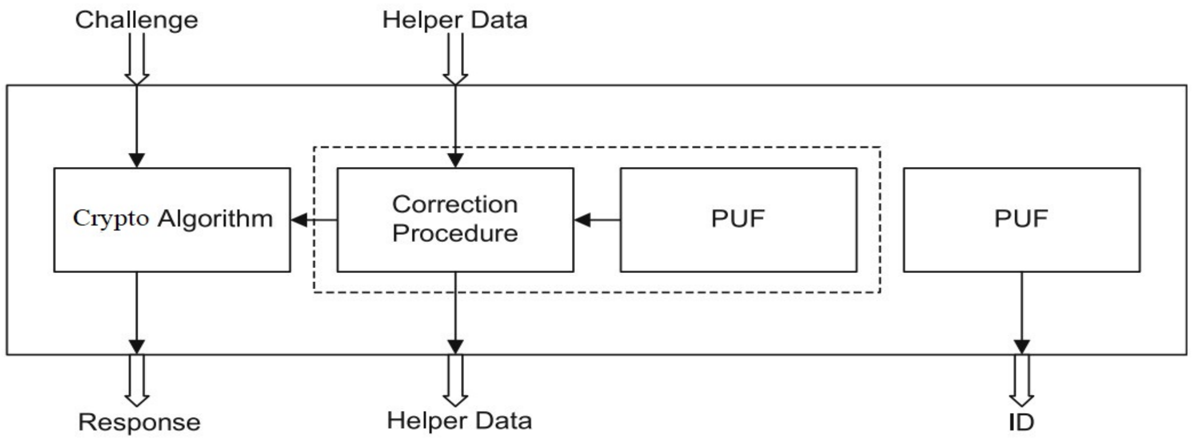


Figure 2.18: Authentication using PUFs with SW CRPs [10].

While it is harder to realize error-free PUF with hardware CRPs, the time, energy, and hardware overhead introduced by PUF with software CRP is significantly higher. Hence, it depends on the authentication application and the type of PUF architecture used. Furthermore, the use of helper data and error correction mechanisms exposes the system to attacks that target both data to reveal the secret information. The topic related to these types of attacks is beyond the scope of this dissertation.

2.3.3 Key generation

PUFs can be used to generate keys for cryptographic algorithms. While cryptographic applications have zero tolerance with errors, PUFs output are noisy and introduce some errors. Hence, error correction codes and helper data are used together to guarantee that PUFs give the exact correct response every time. Furthermore, helper data can be stored on-chip or externally on a server. Figure 2.19 shows a basic concept of using PUFs in key generation.

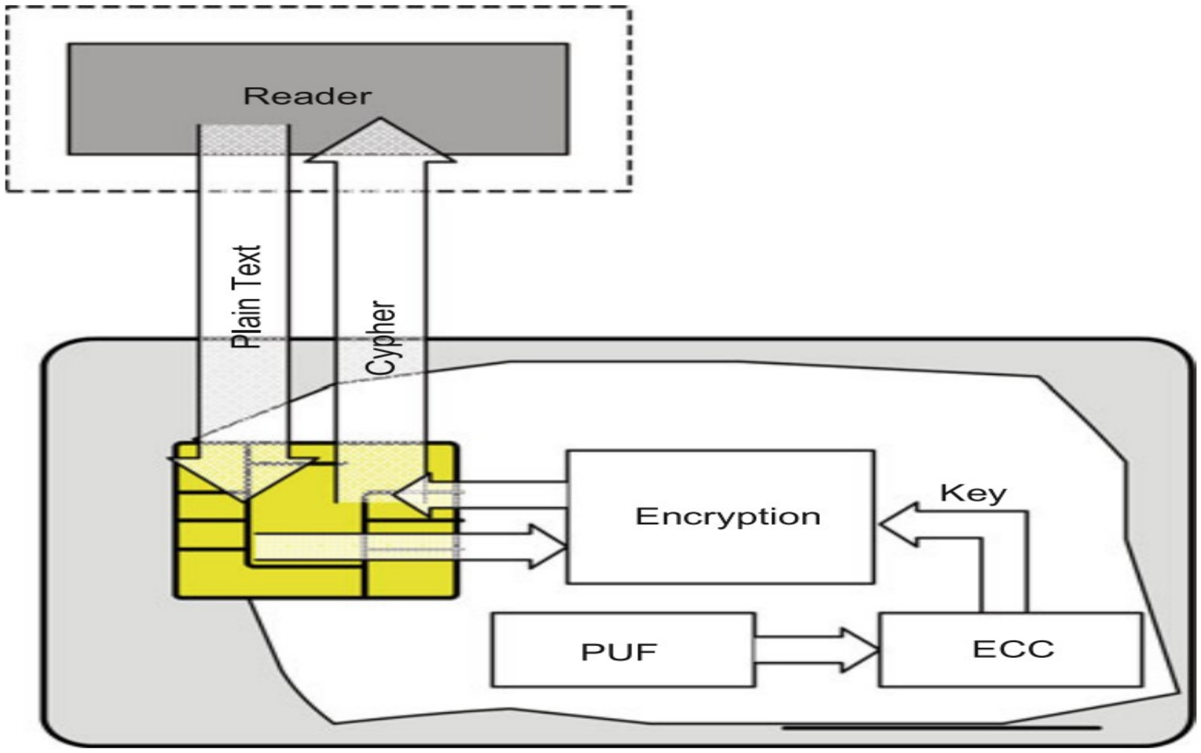


Figure 2.19: Key generation scheme using PUFs [10].

Finally, the work done in this dissertation puts more effort into the development of new strong and hybrid PUF architectures. consequently, targeting authentication and identification applications. However, strong PUFs with large CRP space can be deployed in key generation schemes as well.

Chapter 3

Correlation driven PUF implementation on FPGAs using manual routing and placement

This chapter gives a detailed overview of two main aspects that have a direct relation to this research. The first section reviews the proposed techniques to implement PUFs on FPGAs, specifically delay-based intrinsic PUFs. Although there are reported implementations of other types of PUFs on FPGAs (e.g. memory-based and RO PUFs), delay-based PUFs remain in focus because of its large CRP space and complex input-output relationship. The second and third sections discuss in detail our work of implementing delay-based arbiter PUFs guided by manual routing and placement and using the phi coefficient to help route the most influential stages on the PUF response.

3.1 Overview of PUF implementations on FPGAs

In modern hardware systems, FPGAs have been proposed as a reasonable candidate to provide co-processing functionalities like accelerating dedicated computations and/or implement cryptographic functionalities on hardware. The main justifications for using FPGAs against ASICs are the flexibility and reconfigurability features of FPGAs. Moreover, the short time to market, which enables designers to quickly implement competitive products and easily reconfigure their design to solve any detected defects. Hence, FPGAs provide a cheap and quick solution to investigate PUF different architectures in research. On

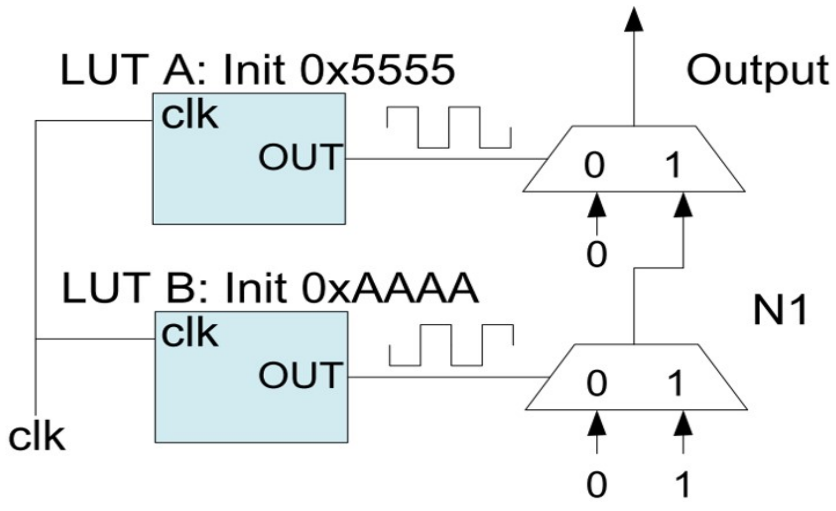


Figure 3.1: Anderson's PUF architecture [67].

the other hand, FPGAs impose many constraints on designing PUFs because of routing and placement of circuits in discrete locations, which requires more effort to handle this properly. Many proposals of PUFs implementations on FPGAs were reported in the literature. This includes diverse types of PUF architecture like memory-based PUFs [50] and delay-based PUFs [28][32]. Moreover, there were proposals for PUFs especially designed to exploit FPGA architecture like Anderson's PUF [67], which uses carry chain multiplexers present in FPGA slices to construct the PUF architecture as shown in Figure 3.1.

Delay-based PUFs are the most challenging types to be implemented on FPGAs because it requires identical path delays, which is challenging to realize under FPGA placement and routing constraints. Next subsections present different techniques for implementing such PUFs on FPGAs and discuss their defects, which is an important base for this research.

3.1.1 Delay lines technique

Programmable Delay Lines (PDL) technique was introduced in [60] and [63] by Majzoobi et al to solve the problem of imbalanced delay paths of arbiter PUFs, which is caused by the dominance of FPGA routing asymmetry. A programmable delay logic with picoseconds resolution is implemented by LookUp Table (LUT) internal structure to cancel out the path delay skews caused by asymmetric routing and systematic variations.

Figure 3.2 shows an example of programmable delay block using 3-input LUT. While

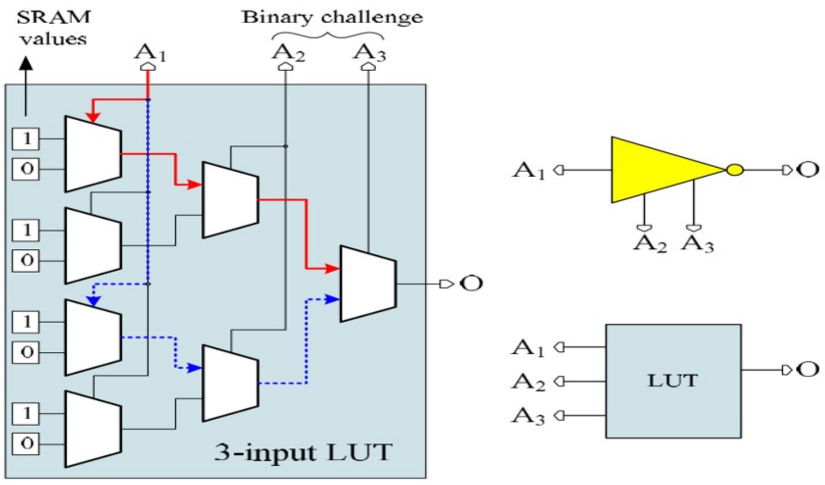


Figure 3.2: Programmable delay line block using 3-input LUT [60].

LUT input A1 is responsible for passing the logic value received from previous stages, inputs A2 and A3 are responsible for determining which path will be used to propagate this value. Hence, introducing extra delay to the path, which ranges from shortest delay marked in red line ($A_2A_3 = 00$) to longest delay marked in dashed blue line ($A_2A_3 = 11$). A delay characterization circuit is used to measure the LUT delay range as shown in Figure 3.3. Obtained results from 5-input LUTs on Xilinx Virtex 5 FPGA suggests that maximum delay difference (i.e. A = 00000 to A = 11111) is 9ps on average.

The architecture of APUF with programmable delay lines is shown in Figure 3.4. PUF is split into two phases, the normal arbiter phase using the same input challenge bits on both paths and the delay tuning phase using programmable delay lines where upper path inputs are different from lower path inputs to compensate for asymmetric routing delay.

The programmable delay technique problem is that it needs a tuning phase to determine the input values of programmable delay blocks. Hence, many CRP collection experiments are required to build a single PUF. Furthermore, adding more LUTs to the PUF design to realize the delay line logic increases the power consumption of the circuit, which is already a limiting factor of FPGAs compared to ASICs.

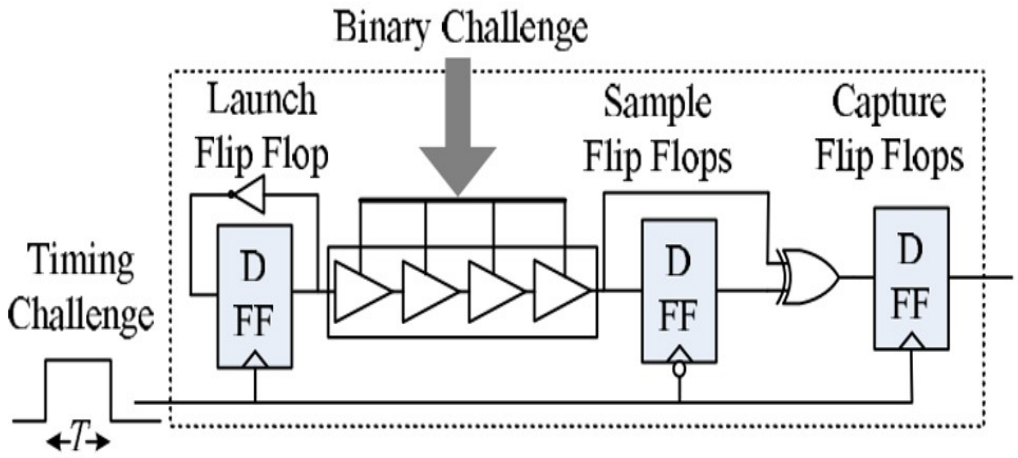


Figure 3.3: PDL Delay characterization circuit [60].

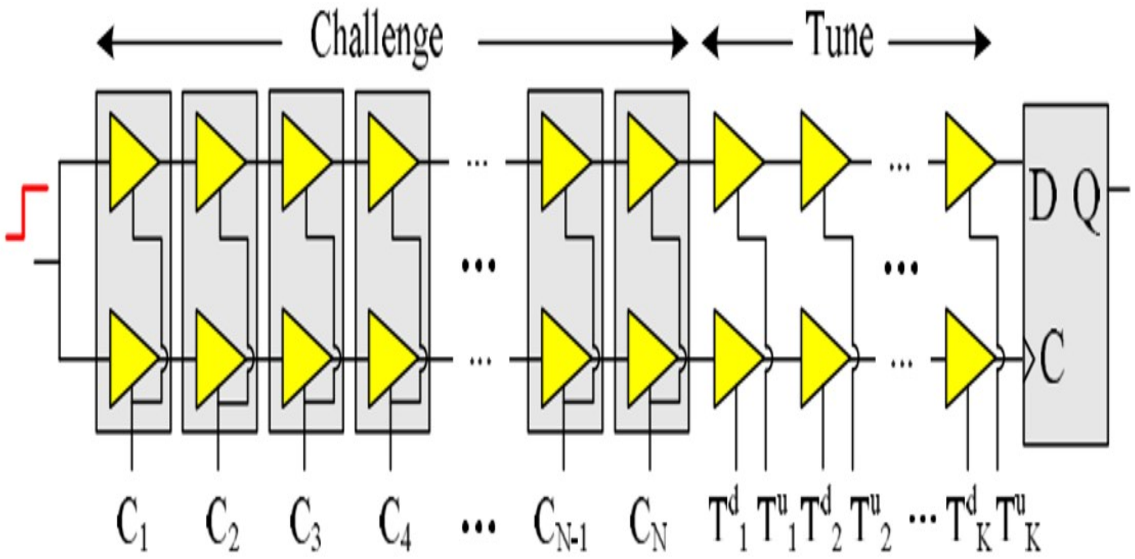

Challenge
Tune
D Q
C
C_1 C_2 C_3 C_4 ... C_{N-1} C_N T^d_1 T^u_1 T^d_2 T^u_2 ... T^d_K T^u_K

Figure 3.4: APUF with programmable delay lines [60].

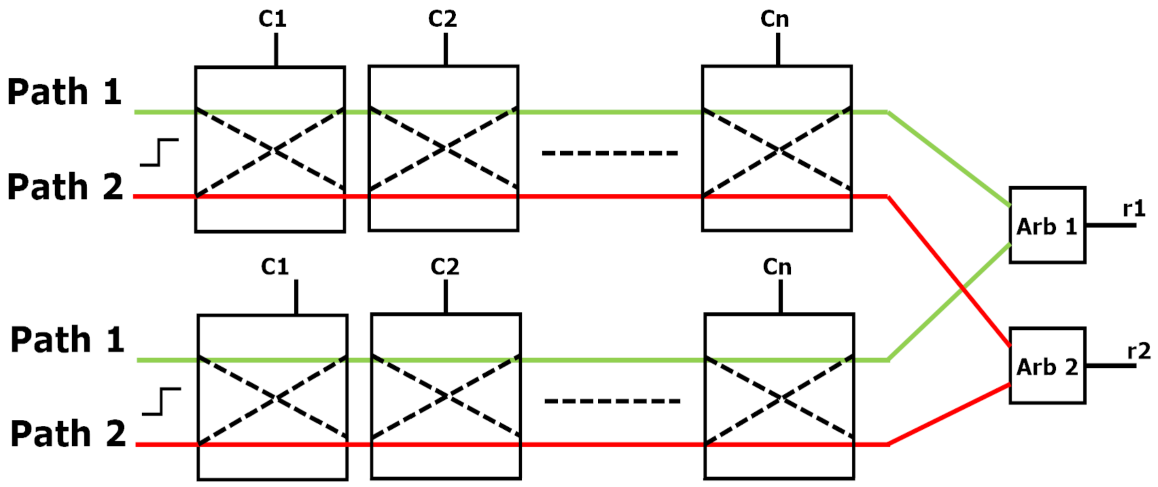


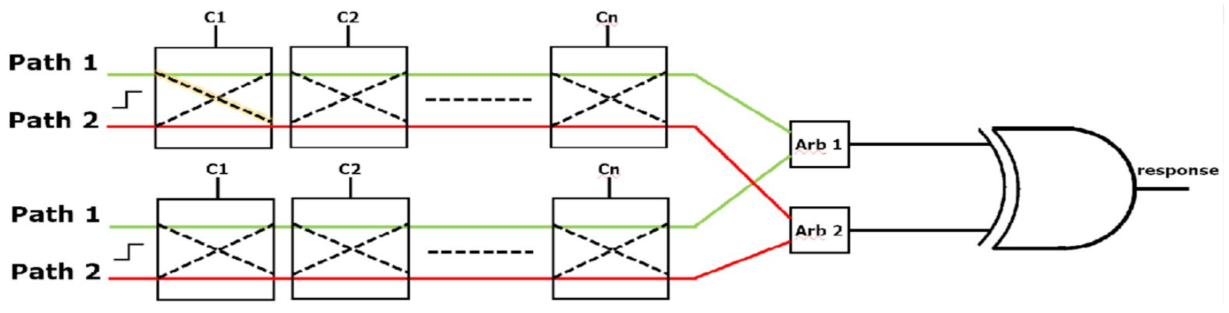
Figure 3.5: Double APUF architecture.

3.1.2 Double arbiter PUF

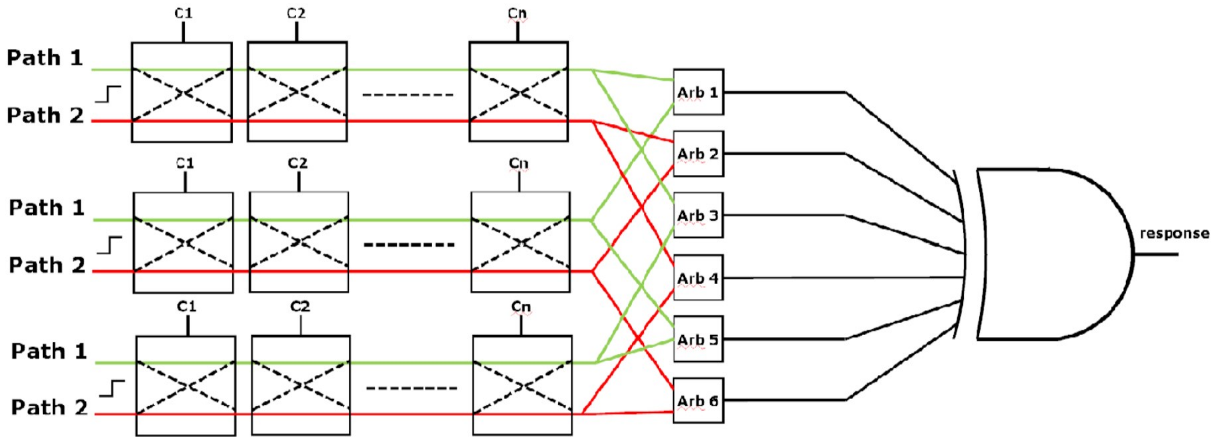
Double APUFs were introduced in [54], [55], and[85] to overcome routing constraints imposed by FPGAs when implementing delay-based PUFs. The main idea of this approach is to use two identical APUFs placed near each other to eliminate routing differences. Every delay path has identical routing delays compared to the matching path in the second PUF. Hence, PUF response is determined by the race of the matching delay paths of the double APUFs as shown in Figure 3.5.

The same authors extended the double PUF design to make it more complex and harder to model by introducing XOR between PUF responses. Figure 3.6 shows the 2-to-1, 3-to-1, and 4-to-1 double arbiter PUFs, where pairs of identical PUFs responses are XORed together to generate the final response.

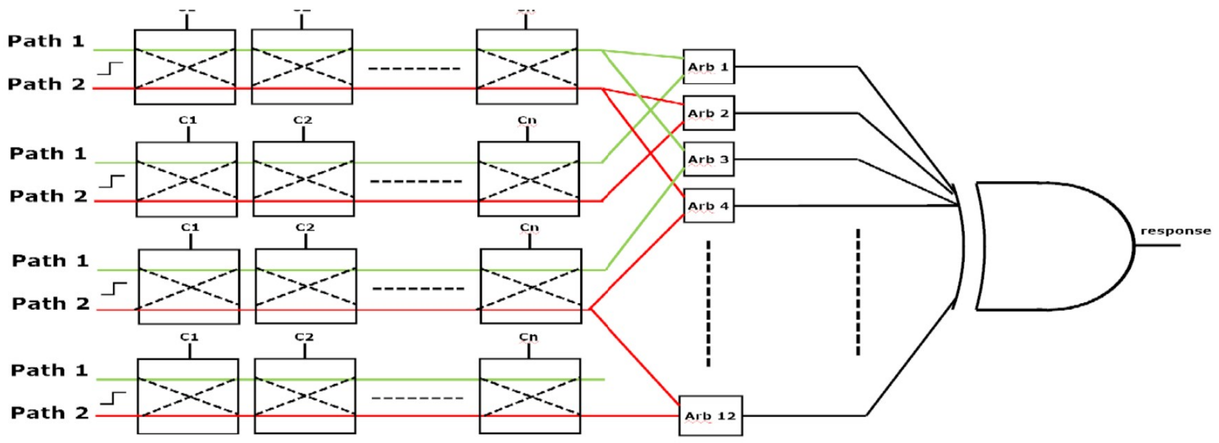
Although results show that double arbiter PUFs have better performance in terms of uniqueness, randomness, and resistance against modeling attacks, the reported results also show that PUF responses are not reliable and introduce a big fraction of error when applying the same challenge input (e.g. 35% error of one chip response using 4-to-1 double APUF). Furthermore, there exist hardware overhead which will impact overall power consumption and hardware resources utilization, especially in the cases where 128-bit PUF responses or more are required by cryptographic systems. Moreover, randomness and uniqueness results were collected using only three Xilinx Vertix-5 FPGA chips and 1000



(a)



(b)



(c)

Figure 3.6: a) 2-1 double arbiter PUF architecture, b) 3-1 double arbiter PUF architecture c) 4-1 double arbiter PUF architecture.

CRPs were used for the modeling attacks, which is not sufficient to fully characterize the new design and requires further investigation.

3.1.3 Randomly generated APUF

Another interesting idea proposed by Spenke et al in [80] was to construct arbiter PUFs on FPGAs by randomly configuring the placement of FPGA stages. In this approach, many placement configurations of APUFs are chosen randomly within a specific area of the chip. Then for every placement, a CRP set is collected given that they introduce minimal delay difference. The process of collecting these CRPs for every configuration is done using two methods:

- Running randomly chosen CRPs on a reference chip and pick the CRPs with flipping output response (i.e. response mostly changes using the same challenge)
- Using a machine learning technique to model the circuit and predict the delay difference between the two paths of APUF.

The pairs of placement configurations and 100 CRPs with minimal delay differences are securely stored at an authentication server. As shown in Figure 3.7, The authentication process is executed by first sending one of the placement configurations to the system controller to configure the FPGA chip, then challenges paired to this placement configuration are used for authentication.

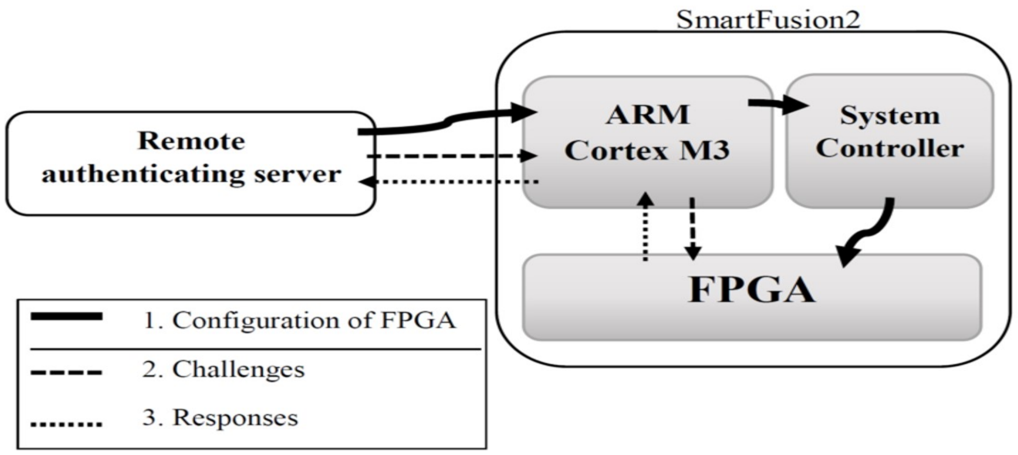


Figure 3.7: Authentication procedure of smartfusion2 chip using randomly generated APUF [80].

The main problem with this technique is that it requires storing the bitstream for every placement configuration. Furthermore, this bitstream is sent through a secure environment to a processor responsible for programming the FPGA to execute the authentication procedure. The size of the bitstream is 556Kb per placement configuration and it takes 28 seconds to program the FPGA as reported in [80]. There is also a preprocessing overhead to choose the placement layout and the metastable CRP set.

3.1.4 PUF implementation summary

From its original inception [29], it was believed that the PUF offered random output data since its design theoretically was focused on measurements of process variation. For example, arbiter PUFs will in theory output random values based upon delay path differences attributable to process variation alone (since it is assumed that the length of the two signal paths is always equal). While this is easily attainable in ASIC designs, it is not clear how to do this in FPGA technology, due to difficulties overcoming biases introduced by complex routing on FPGA implementations (such as routing through switch boxes). In particular, routing complexities along with the asymmetrical delays at the arbiter inputs were noted as possibly preventing APUFs from functioning [68].

Thus, as was shown in this section, researchers have suggested different techniques such as double arbiter PUF designs [85] and programmable delay line (PDLs), which require additional circuitry for tuning response bit output [60]. Furthermore, the random responses of PUFs on FPGAs were empirically found to be largely resulting from placement strategies [18]. Therefore, others have suggested placement in specific different areas of FPGA [64], even including spreading the PUF itself over different areas [80]. In this later research, strong correlations of delay differences in consecutive stages were also found. Moreover, recent research in [95] and [26] has determined that there are some influential challenge bits in PUFs. They proposed an estimated probability formulation and illustrated how an infinite group of challenge bits may influence the response in bistable ring PUFs and suggested this would be the same case for APUFs.

As a result, The next sections discuss our experiments to implement strong delay-based APUFs on FPGAs with metrics performance closer to ideal with minimum hardware and power consumption overhead. We show that a significant performance can be achieved by applying manual placement and guided manual routing using phi correlation coefficient analysis.

3.2 Implementation of APUFs on FPGAs using manual routing and correlation analysis

This section gives an overview of the experimental settings and equipment needed to accomplish research tasks.

3.2.1 Target FPGA and PUF setup

PUF circuits are implemented on MOJO V3 development boards [3], which contain Spartan 6 XC6SLX9 FPGA, a microcontroller (ATmega32U4) used for configuring the FPGA, USB communications, and reading the analog pins. Furthermore, an Arduino compatible bootloader to program the microcontroller and onboard flash memory to store the FPGA configuration file. Spartan FPGA family including Spartan 6 has been widely used in PUF research [76] [94]. Hence, it is reasonable to use it to facilitate comparisons with previous work. The row and column relationships between different configuration Logic Blocks (CLBs) and slices of Spartan 6 FPGA are shown in Figure 3.8.

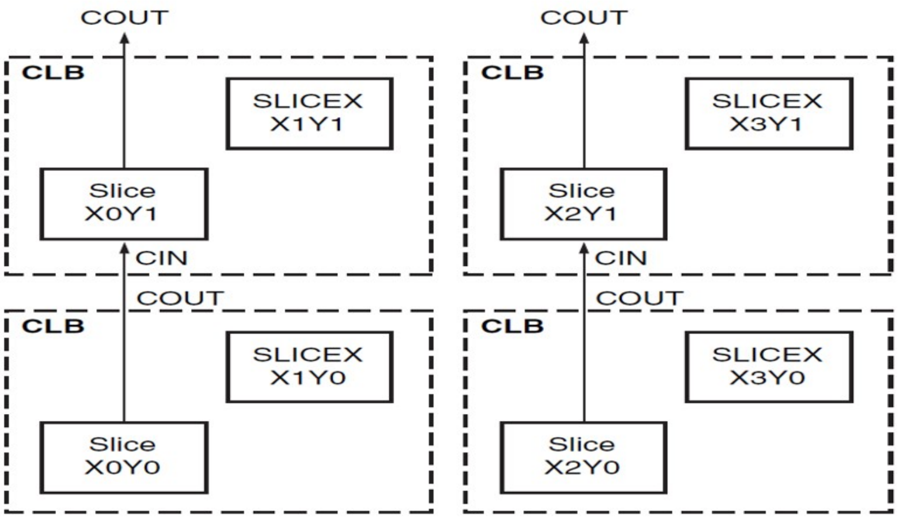


Figure 3.8: Row and Column Relationship between CLBs and Slices [5].

Every slice contains four look-up tables (LUTs) and eight storage elements. These elements are used by all slices to provide logic and ROM functions. SLICEX is the basic

Table 3.1: Spartan 6 XC6SLX9 FPGA logic resources [4].

Logic Cells	Total Slices	SLICEMs	SLICELs	SLICEXs	Number of 6-input LUTs	Maximum distributed RAM (Kb)	Shift registers (Kb)	Number of Flip-Flops
9152	1430	360	355	715	5720	90	45	11440

slice, but Some slices, called SLICELs, also contain an arithmetic carry structure that can be concatenated vertically up through the slice column and wide function multiplexers. The SLICEMs contain the carry structure and multiplexers, and add the ability to use the LUTs as 64-bit distributed RAM and as variable-length shift registers (maximum 32-bit). Table 3.1 shows the available logic resources on Spartan 6 XC6SLX9 FPGA.

Communication with the chip is done using a USB cable and Serial Peripheral Interface (SPI) communication protocol between the microcontroller and the FPGA to handle reading and writing requests. All CRPs set are generated on-chip by either using Linear-Feedback Shift Register (LFSR) or other logic in the case of deterministic CRP set (more details on that in the experimental terminology subsection 3.2.2). A python script is used from a desktop machine to communicate with the chip and collect the CRPs. Although the challenge bits used can be calculated by the software script, but the script collects them from the FPGA chip to verify that challenges are applied as expected and the obtained response belongs to this specific challenge bits. In the case of eight 64-bit APUF implementation, the script requests the read of the applied challenge (1 byte at a time using SPI), then the final byte is the 8-bit response of the 8 PUFs. That is a total of 9 bytes per one CRP (More details on the software script is presented in appendix B).

Every stage path is implemented using 3-input LUT (Challenge bit and 2 paths signals from the previous stage), therefore every stage is implemented using two 3-input LUTs for both paths and can be placed in one 6-input LUT as shown in Figure 3.9 a). There exists a delay difference between the two paths inside the LUT architecture as shown in Figure 3.9 b), where Path 1 going through LUT O6 (the upper output inside the red rectangle) has less delay than Path 2 going through LUT O5 path (The lower output). Hence, LUT placement of the two paths is flipped every stage to balance this internal delay difference. Consequently, the routing delay between stages dominates the overall paths delay difference and manual routing is used to reduce this difference as will be discussed the results section 3.3.

3.2.2 Experimental terminology

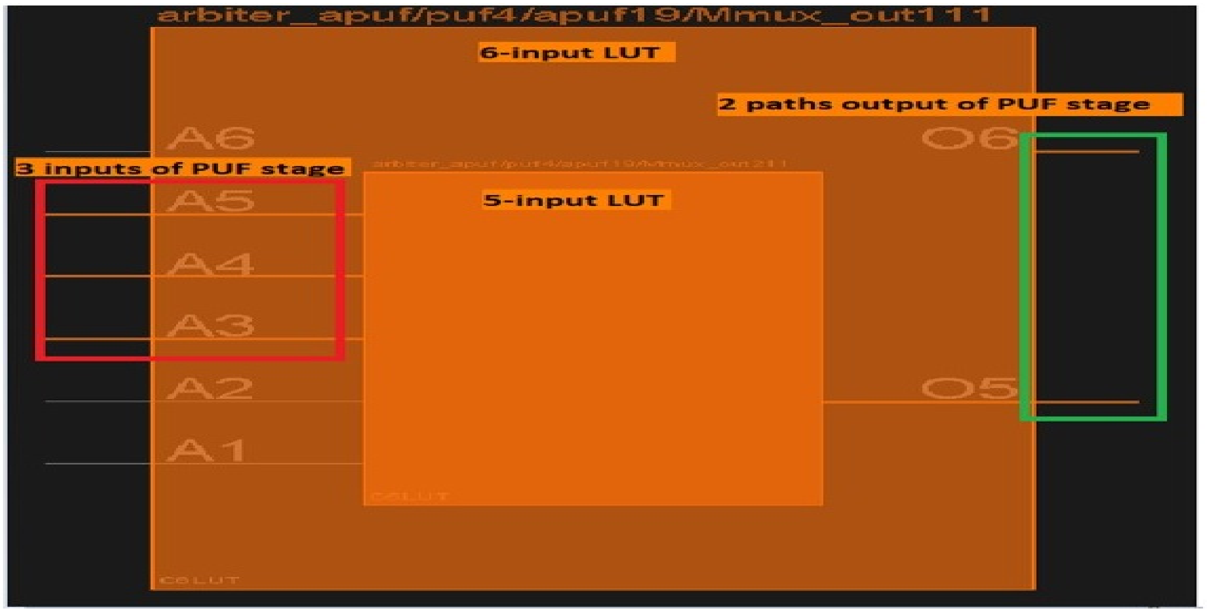
Some terminology will be defined for quantifying PUF randomness and correlations. Let the inter-chip hamming distance be defined as the Hamming Distance (HD) between two n-bit responses of two PUFs (using the same challenge), where each PUF is located on a different chip. The histogram may be used to show the inter-chip HD using PUF responses (one n-bit PUF response per chip and the same challenge) from all pairs of chips. Histograms may also be used to show the distribution of the hamming weight of all n-bit responses from an n-bit PUF on one chip.

Each PUF has a set of challenge input bits (j^{th} challenge bit j=0,1,...,(S-1), where S=64 stages), and 1-bit response output bit. These values are referred to as challenge-response pairs, specifically, the i^{th} CRP has S challenge bits and one response bit (i = 1,2,..., N, where N=1,000,000). Let $n^j_{c,r}$ represent the number of CRPs where the value of the j^{th} challenge bit is c ($c \in$ '0', '1', '-' where '-' represents don't care) and the value of the response bit is r (r $\in$ '0', '1', '-'). The estimated probabilities of challenge bit j predicting response bit have been previously used in [95] and [26] and are now defined as equation 3.1 for challenge bit values of '1' and '0' predicting a response bit of '1' as $E^j_{1,1}$ and $E^j_{0,1}$, respectively. Let the 'actual' probabilities relative to the set of CRPs be defined in equation 3.2 for challenge bit value b predicting response bit value b or $\bar{b}$ as $P^j_{b,b}$ and $P^j_{b,\bar{b}}$, respectively. Finally, the phi coefficient is a statistical measure of two binary signals association. In the PUFs context, its value ranges from +1 to -1 representing perfect agreement/disagreement and 0 value indicates both binary signals have no relationship. Hence, measuring the phi correlation of the challenge bit j with the response bit is shown in equation 3.3 as ϕ^j. Note that ϕ may be used in general (ignoring superscript j), for example, to correlate two sets of binary numbers, where for example 1-bit responses from one PUF are correlated with another set of 1-bit responses from a different PUF, both generated from the same set of challenges.

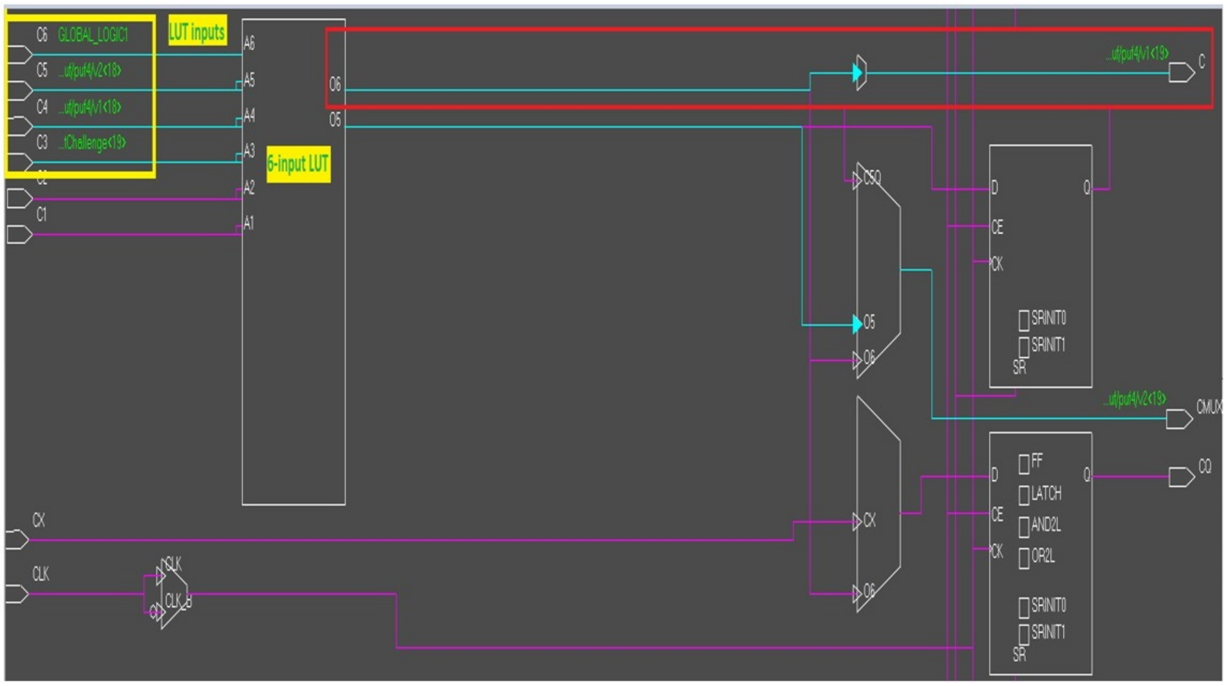
$$E^j_{1,1} = \frac{n^j_{1,1}}{n^-_{1,-}}, E^j_{0,1} = \frac{n^j_{0,1}}{n^-_{0,-}}, and E^j_{1,0} = \frac{n^j_{1,0}}{n^-_{1,-}}, E^j_{0,0} = \frac{n^j_{0,0}}{n^-_{0,-}} \tag{3.1}$$

$$P^j_{b,b} = \frac{n^j_{0,0} + n^j_{1,1}}{N}, P^j_{b,\bar{b}} = \frac{n^j_{0,1} + n^j_{1,0}}{N} \tag{3.2}$$

$$\phi^j = \frac{n^j_{0,0} n^j_{1,1} - n^j_{0,1} n^j_{1,0}}{\sqrt{n^j_{0,-} - n^j_{1,-} - n^j_{-,0} - n^j_{-,1}}} \tag{3.3}$$



(a)



(b)

Figure 3.9: a) Single PUF stage using two 3-input LUTs placed in one 6-input LUT, b) Internal architecture of the 6-input LUT in Spartan 6 FPGA.

As previously mentioned, Five Mojo V3 development boards with Spartan 6 XC6SLX9 FPGA (45nm process technology), were utilized to collect real CRPs. All readings were performed under normal voltage and temperature conditions. For every Spartan 6 chip, the process of collecting 1M CRPs was repeated 11 times. A majority vote of each 11 readings was used to create a final 1M CRPs per chip, similar to [94]. Most previous research does not disclose how CRPs are generated apart from some which use LFSRs [60] and there are not any known publically available CRPs. Hence, the experiments used two sets of CRPs, one random CRP set generated using a linear-feedback shift register (LFSR) as in [75] and another deterministic CRP set. The deterministic CRP set was generated by first randomly choosing 100 64-bit challenge starting points. The minimum step between every two starting points was set to be 65535. Next, 100 sets of consecutive 10K CRPs were created, via incrementing from every starting point. Also note that all challenges are generated within the hardware (inside the FPGA), using HW LFSR or memory to store the challenge starting points. Finally, The Xilinx ISE Design Suite 14.7, Xilinx PlanAhead 14.7, and Xilinx FPGA editor are utilized to implement, manually place and route the arbiter PUF designs.

3.3 Results

Four experiments are presented using the LFSR generated CRP set, each analyzed in terms of the PUF metrics presented in chapter 2. The implementation of one 64-stage arbiter PUFs with 1-bit response required approximately 2.5 rows of slices for manually placing the stages horizontally in the Spartan 6 XC6SLX9 (whose slice array spans X0Y2 in the lower left corner to X23Y61 in the upper right corner). Fig 38. a,b illustrates a part of the schematic and placement. Eight APUFs were implemented on each chip.

3.3.1 Experimental results of PUFs statistical metrics using LFSR generated CRPs set

Experiment 1 utilized symmetric manual placement (referred to as Fixed MP) where each PUF started at the same horizontal position in a row and successive rows were used as shown in Figure 3.10. In addition, the bitstream used to program each FPGA chip was identical. As shown in Figure 3.11 a), all APUFs show close to ideal 50% '1's responses. In contrast, previous research [42] using ASIC implementations of APUFs obtained 40% '1's responses in normal conditions. In other cases [60] on average 50% 1s responses were

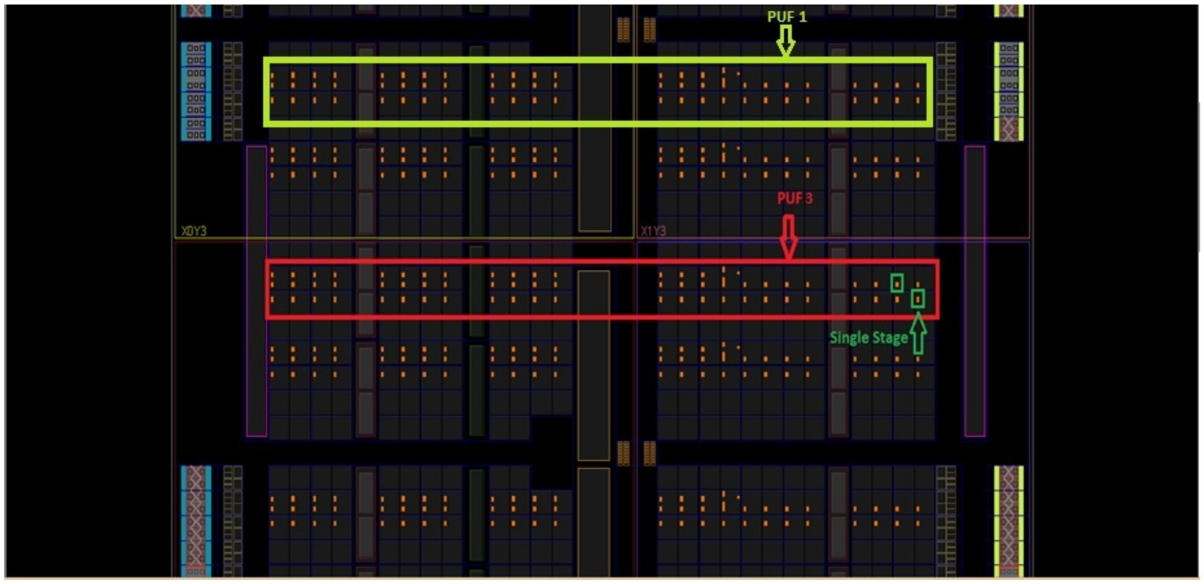
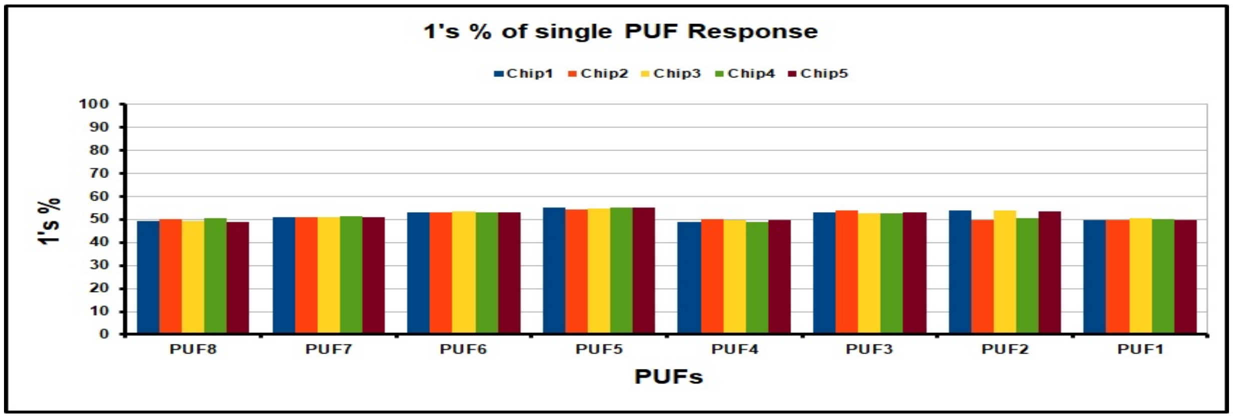
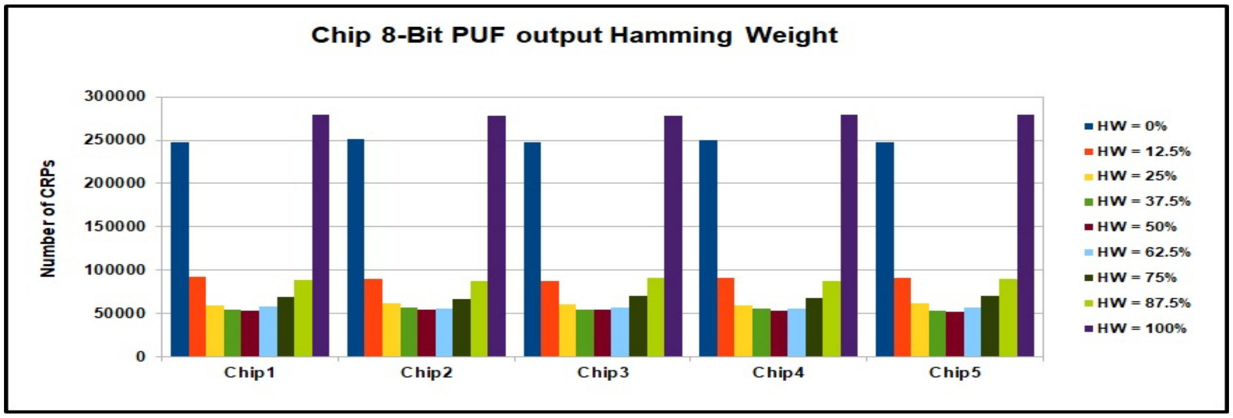


Figure 3.10: Symmetric manual placement in experiment 1

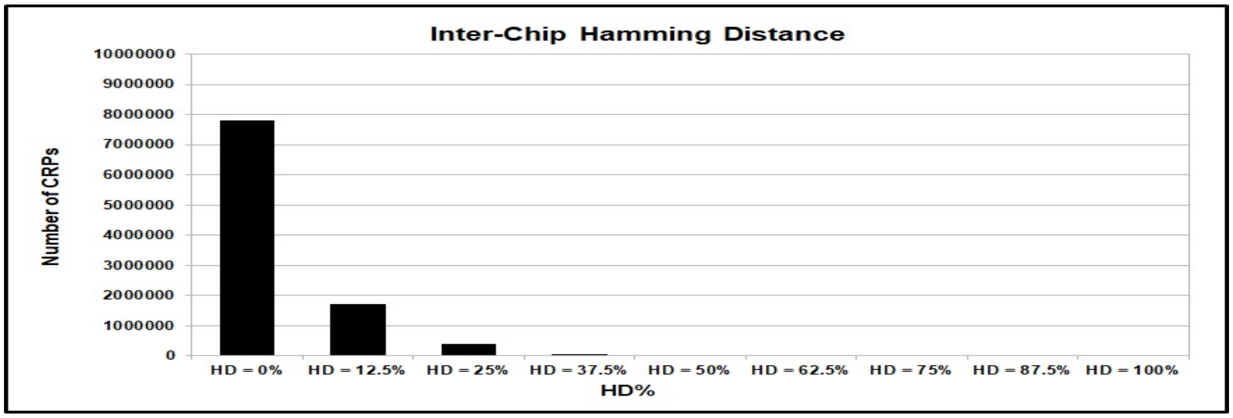
achieved using additional PDL circuitry with tuning level 14, however, results are not clear (see Fig. 10 and 11 in [60] where results show >90% 1s in 6 response bits and <10% 1s in 2 response bits out of a total of 16 response bits). In [54] and [85], similar results close to 50% '1's were reported for the 2-1 and 3-1 double arbiter PUFs, except for one deviation of one Xilinx Virtex-5 FPGA instance that had 31.4% 1s. Remote random re-configurations of arbiter PUFs on FPGA also showed similar 50% 1's results as well [80]. To assess the randomness of the 8-bit PUF response of every chip, we calculated the hamming weight representing the number of '1's and '0's in an 8-bit PUF response for every chip. Figure 3.11 b) illustrates the number of CRPs versus hamming weight of the 8-bit response per chip (e.g. all 0's response corresponds to HW = 0%, and all 1's response corresponds to HW = 100%). It is obvious that the fraction of all 0's and all 1's outputs exceeds 50% of collected CRPs in all chips. Moreover, the case of HW = 50% has the lowest fraction of CRPs, which is opposite to the ideal case. This result shows a strong correlation between the eight PUF responses, which is an undesirable property of a secure PUF. The inter-chip hamming distance between the responses of the same challenge was also calculated, and the results illustrated in Figure 3.11 c) showed approximately 78% of identical responses across the five chips (18% having HD=1, 3% with HD=2, and 1% with HD>2). This result is also consistent with the relatively flat top across chips in Figure 3.11 a). Similar hamming distance results have been reported [54] and [85] which was their motivation to propose a double arbiter PUF.



(a)



(b)



(c)

Figure 3.11: Experiment 1: a) Percentage of '1's in every single PUF response, b) Distribution of intra-chip hamming weights, c)Distribution of inter-chip hamming distance

In conclusion, the previous results indicate that APUF implementation in experiment 1 lacks the unpredictability and uniqueness properties. Hence, in experiment 2, the horizontal and vertical placement of PUFs were modified on each FPGA to improve the inter-chip and intra-chip correlation measurements (referred to as rand MP). In addition, the horizontal and vertical placements were varied per chip (so different bitstreams were utilized) as shown in Figure 3.12. In particular, the first stage of each PUF started from a different horizontal position (different X value for the first SliceXY). See Figure 3.12 as an example where the red box illustrates a 1-bit PUF whose first stage starts at the lower left and the 1-bit PUF above it (not enclosed in the red box) has its first stage shifted to the right in comparison. Also, as illustrated in Figure 3.12, the area between every two consecutive PUFs was also sometimes varied (so successive rows were not always utilized, thus creating a variation in vertical placement as well).

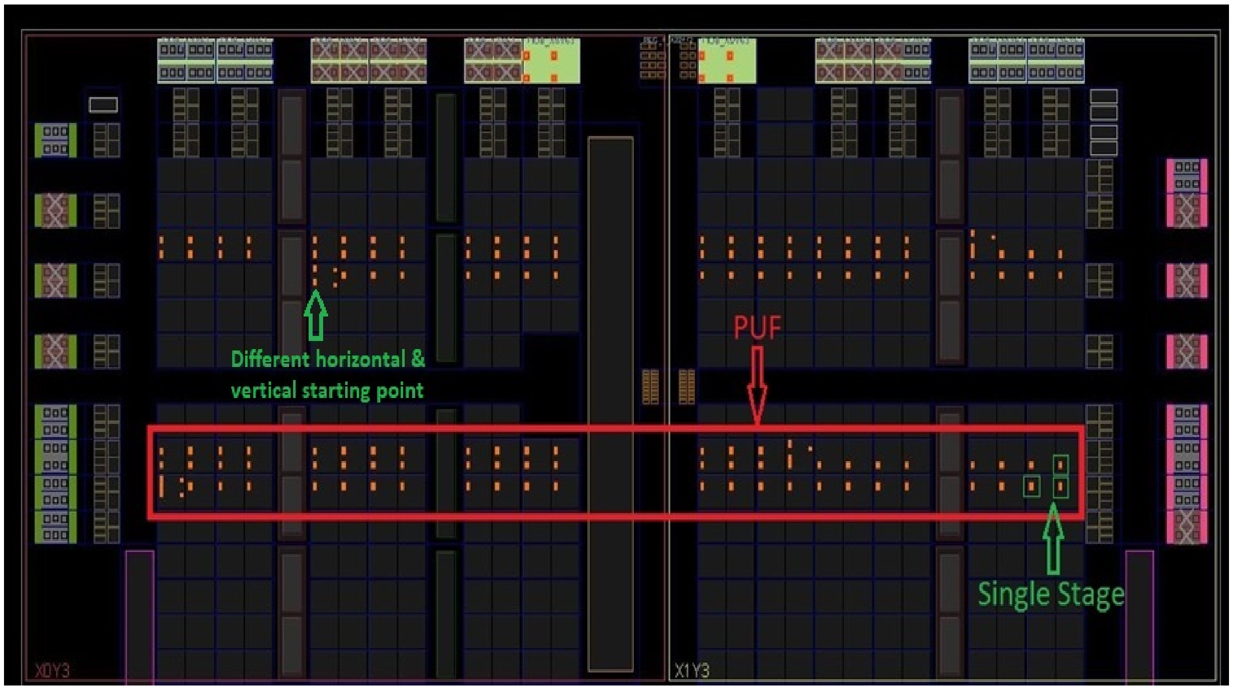
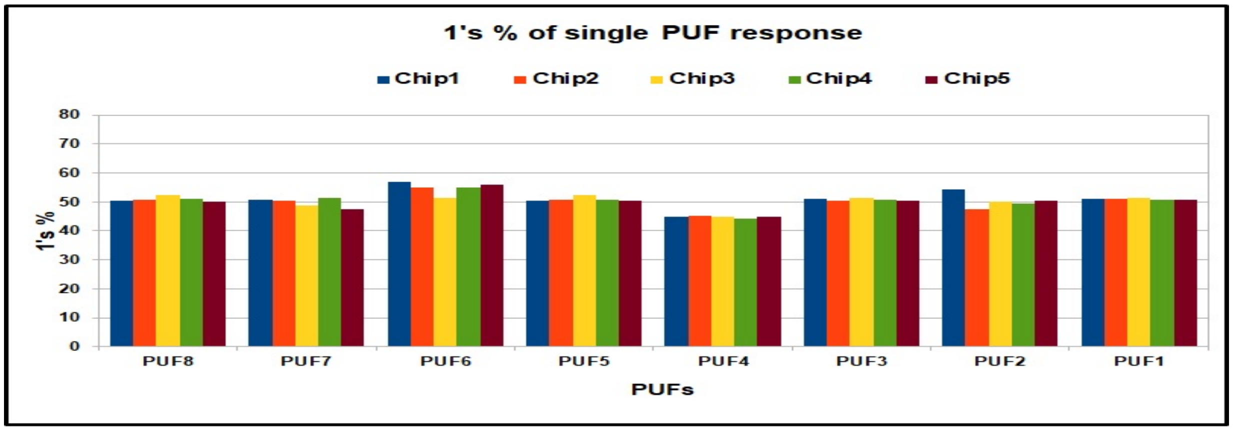
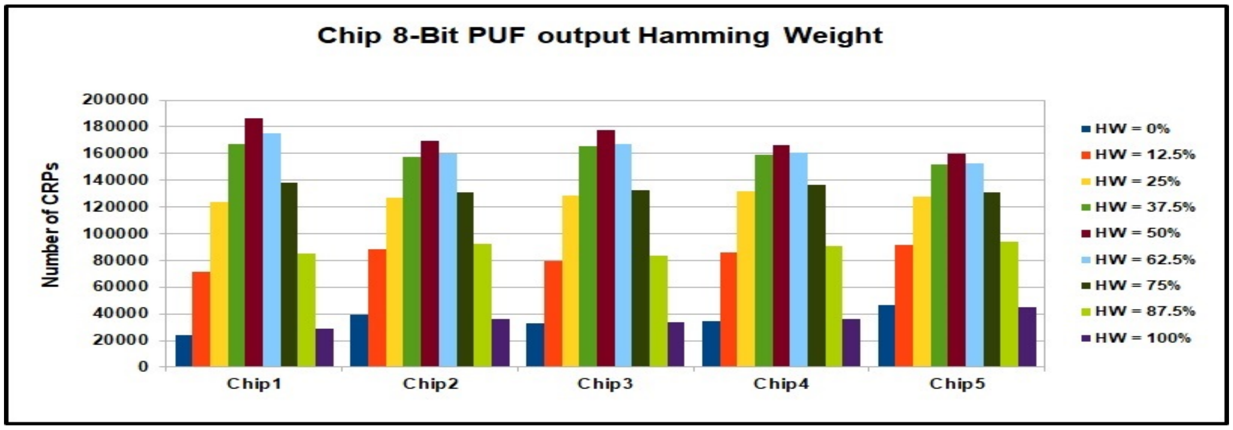


Figure 3.12: Symmetric manual placement in experiment 2

Figure 3.13 illustrates the %'1's, hamming weight distributions and inter-chip hamming distance distribution for the 5 chips. Although the results obtained in Figure 3.13 a) show slight deviation in two instances of PUF 7 and PUF 4 from the ideal 50% '1's response, it is still within the accepted range compared to the results reported in [42], which showed APUFs with a range of 60% - 40% 1's. A large improvement was achieved in the distribution of the 8-bit PUF hamming weights as shown in Figure 3.13 b), indicating an average hamming weight of 50% as needed. Also, a significant improvement in inter-chip HD measurement was achieved as shown in Figure 3.13 c) (mean HD=36.7%), although the ideal case (50%) was not reached. Other researchers [52] [76] also showed low mean HD (mean HD= 4.72%) for their APUF implementation on Xilinx Virtex-5 and Kintex-7. Researchers in [54] and [85] reported better hamming distance results using dual arbiter PUFs (mean HD = 45%). More recently nearly 50% hamming distance was achieved using random re-configurations on FPGAs [80].

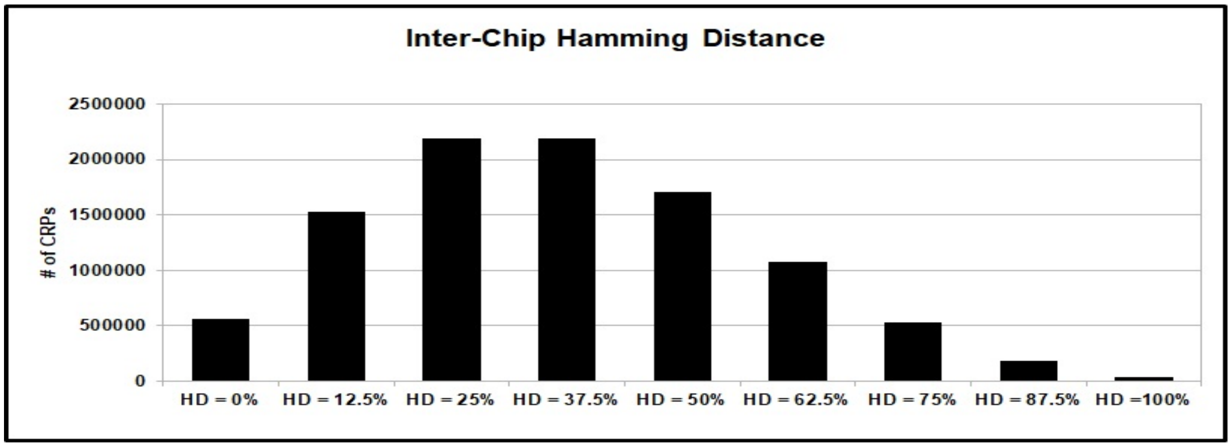


(a)



(b)

Figure 3.13: Experiment 2: a) Percentage of '1's in every single PUF response, b) Distribution of intra-chip hamming weights



(c)

Figure 3.13: Experiment 2: c)Distribution of inter-chip hamming distance

Phi correlations of 1-bit responses for the same challenge between all pairs of PUFs responses on the same chip were also performed. Equation 3.3 was utilized where the value of the jth challenge bit is respectively the PUFj response and the value of the response bit l is respectively the PUFk response (kj). In experiment 1, as shown in Table 3.2 and Figure 3.14, the maximum phi correlation was 80% (between PUF7 and PUF8) whereas in experiment 2 this maximum dropped to 45% (between PUF5 and PUF8) as shown in Table 3.3 and Figure 3.15. Experiments 1 and 2 have an average correlation between PUF pair responses of 53% and 12% respectively. The low phi correlation shows good uniqueness, indicating experiment 2 created an improved PUF. Additionally, These numbers confirm the previously mentioned hamming weight analysis results.

Table 3.2: Experiment 1: PUFs responses correlation on chip 1

	PUF 8	PUF 7	PUF 6	PUF 5	PUF 4	PUF 3	PUF 2	PUF 1
PUF 8	1.000	0.801	0.467	0.387	0.490	0.506	0.643	0.775
PUF 7	0.801	1.000	0.452	0.437	0.529	0.498	0.679	0.712
PUF 6	0.467	0.452	1.000	0.525	0.674	0.501	0.478	0.448
PUF 5	0.387	0.437	0.525	1.000	0.509	0.557	0.446	0.322
PUF 4	0.490	0.529	0.674	0.509	1.000	0.538	0.457	0.421
PUF 3	0.506	0.498	0.501	0.557	0.538	1.000	0.406	0.446
PUF 2	0.643	0.679	0.478	0.446	0.457	0.406	1.000	0.632
PUF 1	0.775	0.712	0.448	0.322	0.421	0.446	0.632	1.000

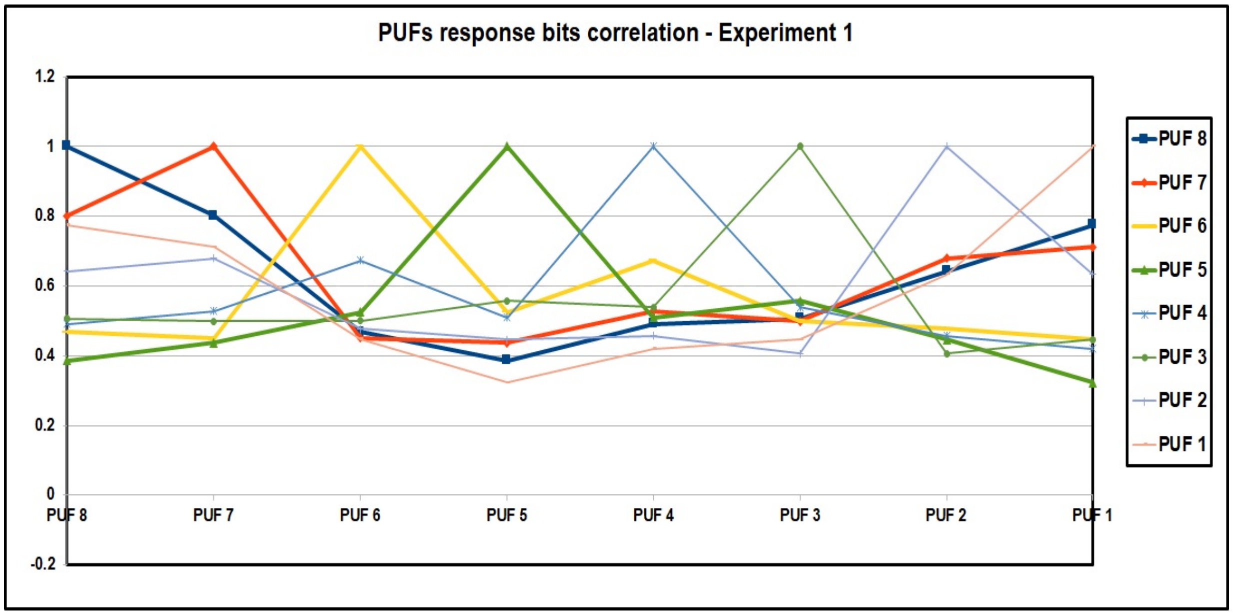


Figure 3.14: Experiment 1: 8 PUFs responses correlation analysis on chip 1

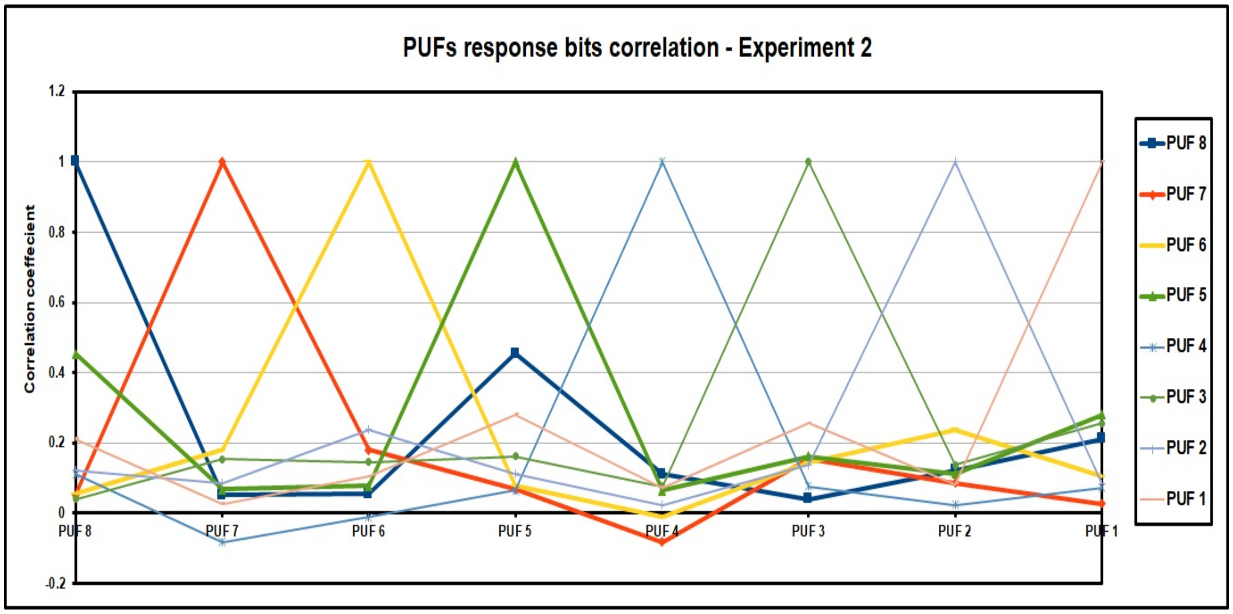


Figure 3.15: Experiment 2: 8 PUFs responses correlation analysis on chip 1

Table 3.3: Experiment 2: PUFs responses correlation on chip 1

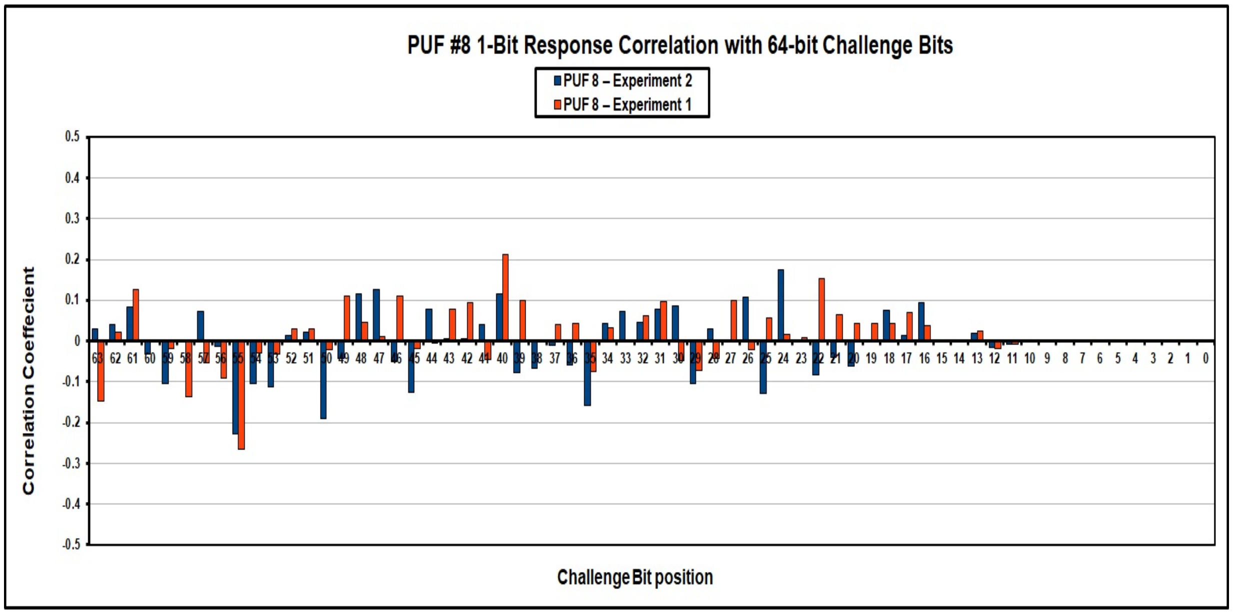
	PUF 8	PUF 7	PUF 6	PUF 5	PUF 4	PUF 3	PUF 2	PUF 1
PUF 8	1.000	0.051	0.054	0.453	0.110	0.041	0.122	0.212
PUF 7	0.051	1.000	0.180	0.069	-0.082	0.154	0.087	0.027
PUF 6	0.054	0.180	1.000	0.078	-0.011	0.146	0.238	0.104
PUF 5	0.453	0.069	0.078	1.000	0.065	0.163	0.112	0.281
PUF 4	0.110	-0.082	-0.011	0.065	1.000	0.077	0.024	0.073
PUF 3	0.041	0.154	0.146	0.163	0.077	1.000	0.138	0.256
PUF 2	0.122	0.087	0.238	0.112	0.024	0.138	1.000	0.082
PUF 1	0.212	0.027	0.104	0.281	0.073	0.256	0.082	1.000

3.3.2 Experimental results of PUFs correlation analysis using deterministically generated CRPs set

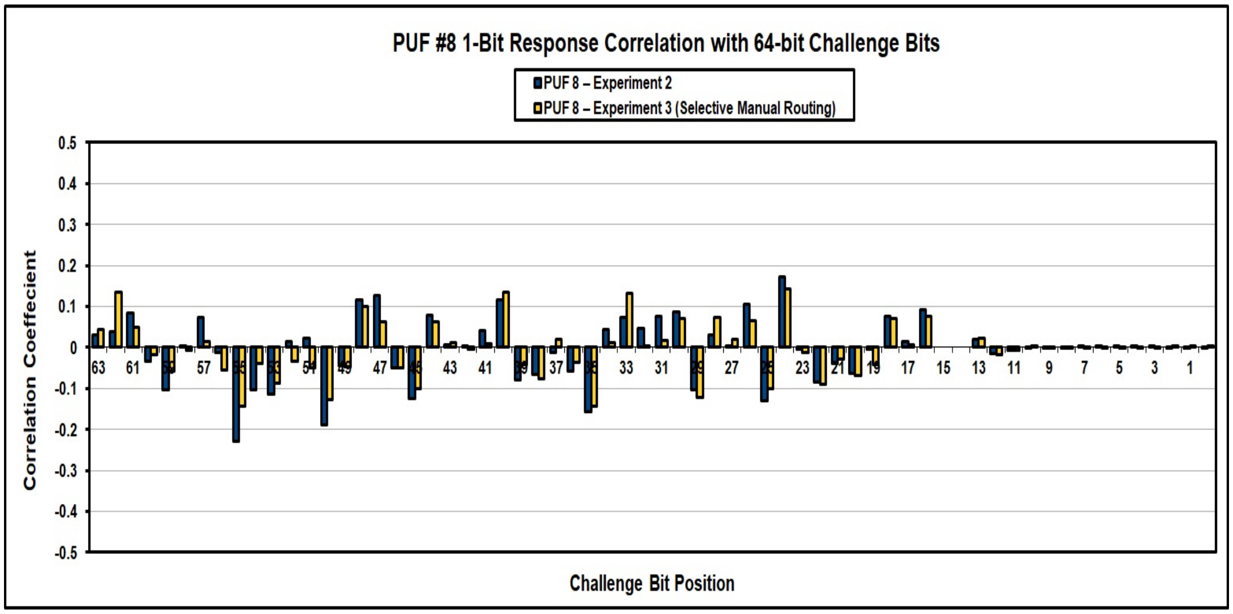
Analysis of PUFs using the deterministic CRP set was performed to further study the influence of challenge bits with large delay differences. Using this CRP set, stages with higher delay differences and PUFs with higher path delay difference (e.g., Exp-1, Exp-2 in Table 3.4) are observed and stages which require manual routing are determined.

Correlation analysis between challenge bits and PUF 8 response bit was run on the experiments 1 and 2 using phi correlation coefficient and equation 3.3. Figure 3.16 a) illustrates the correlation between the challenge bits and response bit of PUF 8 in experiments 1 and 2. The phi correlations of challenge bits 63 through 0 are plotted left to right on the x-axis, where stage 63 outputs the response bit. It is evident that the first 10 stages of the PUF (last 10 on the x-axis) have little to no correlation to the response bit, as expected since these bits are varied all the time.

Based on the correlation analysis results and instead of manually routing all stages, experiment 3 manually routes, some PUF stages of the most influential challenge bits, specifically, bits 55, 50, 45, 35, 25, to reduce the delay difference between the two arbiter PUF paths (referred to as path 1 and path 2). For example, the wire connections between stages 55 and 56 were manually routed to try to minimize the phi correlation of challenge bit 55. In many cases, significant APUF delay differences could be reduced through manual routing. For example, manual routing of stage 55 modified path 1 delay from 0.500ns to 0.664ns and path two delay from 0.825ns to 0.663ns. These changes reduced the wiring delay difference from 0.325ns to 1ps respectively. This delay difference is within the previously researched process variation of 3.5% [71] and the 5% timing analysis accuracy, as



(a)



(b)

Figure 3.16: Challenge bits correlation analysis with PUF 8 response on chip 1, a) Experiments 1 & 2: b) Experiments 2 & 3

Table 3.4: Delay difference analysis of PUF8 on Chip 1

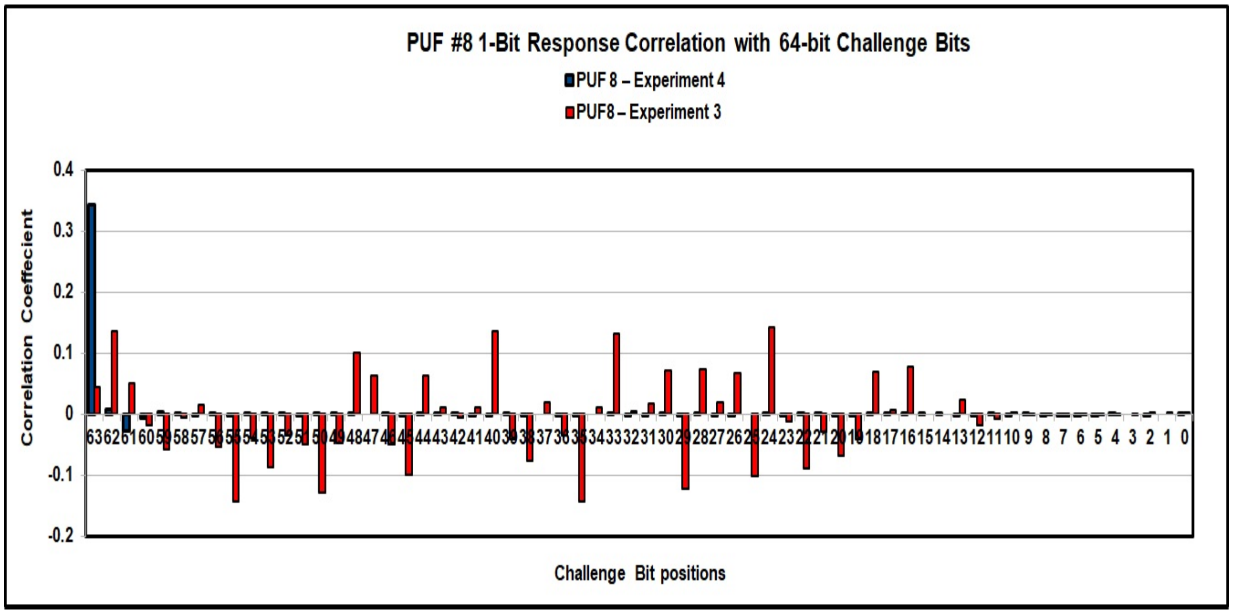
	Exp-1	Exp-2	Exp-4	Exp-4 revisited
Total 64-stage delay difference between path 1 and path 2	4.61ns	1.24ns	0.28ns	0.1ns
Average (stage delay difference / total stage delay)	33%	33%	6.4%	6.4%
Number of stages with path1 delay > path 2 delay	23	32	29	27
Number of stages with path1 delay < path 2 delay	41	32	33	35
Number of stages with path1 delay = path 2 delay	0	0	2	2

required for ideal PUFs. As another example, for one set of challenge bits, the accumulated entire PUF wiring delay was 33.62ns and 31.389ns, and thus the value of the response bit is undetermined under assumed variations (e.g. it may be positive, 33.62-31.389>0, or negative, (33.62-x%) – (31.389+x%) <0, within the variation, x=3.5% or 5%). In this example, even the asymmetrical delays to the arbiter were included. However not all accumulated and inter-stage delay paths of the PUFs had these properties.

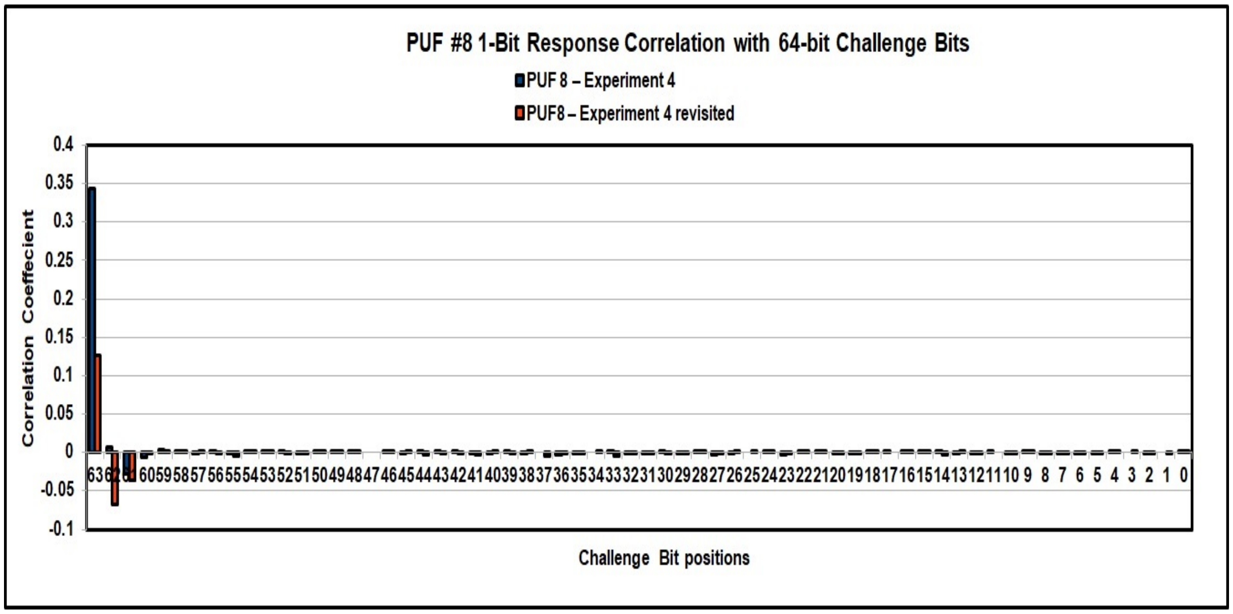
The phi correlations resulting from selective manual routing are shown in Figure 3.16 b) in comparison with experiment 2. In all cases, the maximum, range, and standard deviation of the phi correlation coefficients have decreased significantly in experiment 3 compared to experiment 2. Also note that some other bits have become more influential (e.g. bit 62, 33). However, as will be shown in subsection 3.3.3, the PUFs produced in experiment 3 did not show a significant performance against modeling attacks.

In experiment 4, repetitive manual placement was applied on PUF 8 to find better routing choices. Consequently, the PUF was manually routed to reduce the delay differences in every stage and balance the overall delay paths. Further analysis was conducted on delay difference for every stage, total delay difference between PUF paths, and the average delay difference across all PUF stages as shown in Table 3.4. Obtained results show that experiment 4 has better results in terms of the total paths delay difference and the average delay difference. It is also shown that using specific manual placements allowed us to have some stages with 0ps delay difference. results in subsection 3.3.3 will show more details of how the PUFs produced in experiment 4 is more resistant to modeling attacks.

The same correlation analysis was done on the PUF produced in experiment 4 and



(a)



(b)

Figure 3.17: Challenge bits correlation analysis with PUF 8 response on chip 1, a) Experiments 3 & 4: b) Experiments 4 & 4-revisited

Table 3.5: PUF 8 response correlation with other PUFs responses on chip 1 for experiments 2, 3, 4, and 4-revisited

	PUF 7	PUF 6	PUF 5	PUF 4	PUF 3	PUF 2	PUF 1	Avg. Corr.
Exp. 2	0.051	0.054	0.453	0.110	0.041	0.122	0.212	0.149
Exp. 3	0.103	0.178	0.418	0.196	0.033	0.058	0.407	0.199
Exp. 4	0.007	0.013	-0.027	0.027	-0.063	-0.029	0.029	-0.006
Exp.4 -revisited	0.071	0.011	-0.042	0.120	-0.019	0.023	0.097	0.037

is compared to the influential bits in experiment 3 as shown in Figure 3.17 a). Results obtained did not show significant influential bits in experiment 4 except for challenge bit 63. which showed 35% positive correlation with PUF 8 response. The high correlation of challenge bit 63 in experiment 4 is largely due to stage 62 having a relatively large delay difference compared to other stages. This stage 62 was rerouted manually to minimize its delay difference as shown in the Experiment-4 revisited column. Consequently, challenge bit 63 correlation went down from 35% to 12% as shown in Figure 3.17 b). The maximum delay difference overall stages in experiment-4 revisited was 0.072ns and two stages 22, 45 have 0ns delay difference. Correlation analysis of PUF 8 response with respect to other PUFs responses was conducted for experiments 3, 4, and 4-revisited. Results in Table 3.5 show that the correlation between PUF 8 and PUF 5 that existed in experiment 2 instance is drastically decreased. Furthermore, the average correlation between PUF 8 and other PUFs is decreased to 0.6 and 3.7% in experiment 4 and 4-revisited respectively. This confirms the role of repetitive manual placement and correlation-driven routing in improving overall PUFs correlation.

3.3.3 Modeling attacks

Modeling attacks against implemented PUFs were invoked to show their resistance to such attacks. For the sake of comparison with previous research in [75] and [76], The logistic regression (LR) technique is utilized for the model building of a single bit PUF in all 4 experiments using LFSR generated CRPs. The mathematical model derived in [75] is used to get the input features of the LR model for training (more details on the mathematical model in Chapter 4). The dataset size is 1M CRPs, and training size is varied between 600 to 100K CRPs. The test dataset is always 1M - Training set size. The results of LR attacks against PUF 8 in the four experiments are in Table 3.7, which shows the training, test accuracy scores and elapsed time to finish the training process. Results show that PUFs developed in experiments 1 to 3 could be modeled with accuracy close to the numbers reported in previous research as shown in Table 3.6, which shows the modeling accuracy of 64 stages APUFs implemented on FPGAs (using programmable delay line technique), ASIC, and software simulated PUFs. However, the PUFs developed in experiments 4 and 4-revisited showed better resistance to LR modeling attack with accuracy 7% less than what was previously reported as illustrated in Figure 3.18. Although the exploitation of manual placement and correlation-driven manual routing resulted in implementing better APUF instances on FPGAs, these types of PUFs have been shown to be architecturally vulnerable to modeling attacks. Hence, these implementation techniques can be used to realize more secure strong PUF types that are resistant to all types of modeling attacks as will be shown in the next two chapters.

Table 3.6: LR Attack results on APUFs in [75] and [76].

	Training size	Training Time	Test Accuracy
FPGA (PDL)	650	0.12s	>95%
	6500	0.83s	>99%
ASIC	640	0.01s	>95%
	6500	0.76s	>99%
Simulated CRPs	640	0.01s	>95%
	2555	0.13s	>99%
	18050	0.6s	99.9%

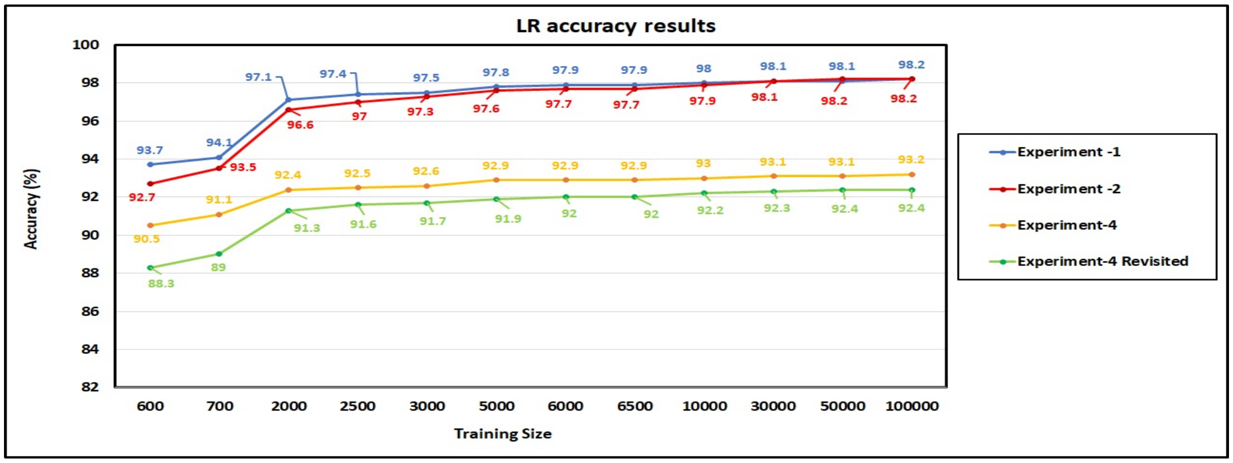


Figure 3.18: Logistic regression accuracy results on PUF 8 of experiments 1, 2, 4, and 4-revisited

Table 3.7: PUF LR attacks.

Training size	Experiment 1	Experiment 2	Experiment 3	Experiment 4	Experiment 4 revisited	
600	98.1%	97.7%	98%	95.7%	95.3%	Training accuracy
	93.7%	92.7%	95%	90.5%	88.3%	Test accuracy
	< 0.1s	< 0.1s	< 0.1s	< 0.1s	< 0.1s	Training time
700	98.6%	97.9%	98.1%	95.7%	95%	Training accuracy
	94.1%	93.5%	95.1%	91.1%	89%	Test accuracy
	< 0.1s	< 0.1s	< 0.1s	< 0.1s	< 0.1s	Training time
2000	98.5%	99.2%	98.3%	94.3%	93.4%	Training accuracy
	97.1%	96.6%	97.4%	92.4%	91.3%	Test accuracy
	< 0.1s	< 0.1s	< 0.1s	< 0.1s	< 0.1s	Training time
3000	98.6%	98.8%	98.5%	94.3%	92.7%	Training accuracy
	97.5%	97.3%	97.6%	92.6%	91.7%	Test accuracy
	< 0.1s	< 0.1s	< 0.1s	< 0.1s	< 0.1s	Training time
5000	98.6%	98.6%	98.8%	93.4%	92.2%	Training accuracy
	97.8%	97.6%	98.1%	92.9%	91.9%	Test accuracy
	< 0.1s	< 0.1s	< 0.1s	< 0.1s	< 0.1s	Training time
6000	98.5%	98.7%	98.8%	93.5%	92.2%	Training accuracy
	97.9%	97.7%	98.3%	92.9%	92%	Test accuracy
	< 0.1s	< 0.1s	< 0.1s	< 0.1s	< 0.1s	Training time
6500	98.6%	98.7%	98.7%	93.2%	92.1%	Training accuracy
	97.9%	97.7%	98.3%	92.9%	92%	Test accuracy
	0.3s	0.1s	0.1s	0.1s	< 0.1s	Training time
10K	98.4%	98.6%	99%	93.5%	92.3%	Training accuracy
	98%	97.9%	98.5%	93%	92.2%	Test accuracy
	0.3s	0.1s	0.1s	0.1s	< 0.1s	Training time
30K	98.3%	98.3%	99%	93.4%	92.5%	Training accuracy
	98.1%	98.1%	98.9%	93.1%	92.3%	Test accuracy
	0.3s	0.5s	0.5s	0.3s	0.3s	Training time
50K	98.2%	98.2%	99%	93.3%	92.4%	Training accuracy
	98.1%	98.2%	98.9%	93.1%	92.4%	Test accuracy
	0.7s	0.7s	0.8s	0.9s	0.8s	Training time
100K	98.2%	98.2%	99%	93.2%	92.4%	Training accuracy
	98.2%	98.2%	99%	93.2%	92.4%	Test accuracy
	1.3s	1.3s	1.7s	1.1s	1.5s	Training time

3.3.4 Summary

In summary, APUFs implemented in Spartan 6 FPGA have been analyzed with existing and new metrics. In experiment 1 which used one bitstream on 5 different chips, inter-chip HDs indicated that approximately 22% of pairs of 8-bit responses (from the same challenge) of the APUFs on different chips differed in one or more bits. This was likely due to either the process variation being very low or most delay differences in APUFs were large enough to be unaffected by process variation. In experiment 2, it was demonstrated that changing the horizontal and vertical placement of every single PUF on every chip, improved the randomness of every chip's 8-bit PUF response, and the inter-chip hamming distance (HD improved to 37%). This effect is likely largely due to the variation in routing performed by the tools, although the process variation is also known to vary spatially within-die. Experiment 3 used correlation driven manual routing to decrease the delay difference at the stages that showed more influence on PUF response. However, results indicated that it was not sufficient to resist machine learning modeling attacks. Hence, Experiment 4 applied repetitive manual placement to facilitate the manual routing of stages with minimum delay difference, and correlation analysis was used to further tweak the PUF 8 implementation in the revisited experiment 4 to decrease the influence of specific challenge bits. Statistical analysis showed that this hybrid technique improved the PUF properties (i.e. overall delay difference, average delay difference) and allowed to have two stages with 0s routing delay difference. Additionally, the correlation between PUF 8 response and PUF 5 is decreased from 45% to 4%. Furthermore, the PUF instances in experiment 4 and 4-revisited showed more resistance to LR modeling attacks compared to the instances developed in the other experiments, and the results reported in previous research. However, APUFs architectural design allows modeling attacks to successfully clone its response, which is a motivation for the rest of our research in chapters 4 and 5 to study the limit of modeling attacks and new architectures to counter them. More details on the placement and routing specific steps and the scripts used to calculate statistical metrics are presented in appendices A and B.

Generating PUFs without manual routing supports portability across different FPGA technologies, however, some manual routing may be crucial for good PUF performance. In general, tweaking PUF designs is complex since many metrics are sensitive to design changes (%1s, inter-chip HD, HW, Phi plots). Phi correlations are proposed as a concise metric guiding the performance tweaking of the PUF, showing promise for identifying stages that require manual rerouting. This is unlike previous research that requires higher cost overheads, specifically utilizing additional PDL circuitry [60], doubling the area[85], or spreading placement of stages across entire FPGA and storing the bitstream configurations at the authentication server [80]. This research also showed that manually routed inter-stage wiring delay differences can be within assumed tool accuracy and process variation (e.g. as low as 0ps), thus only the wiring symmetry at the arbiter stage remains to be improved in order to support an ideal fully manually-routed APUF.

Chapter 4

Deep learning modeling attacks

This chapter discusses in detail the deep learning modeling attacks of previously provable resistant PUFs against conventional machine learning techniques. The first section gives an overview of different types of attacks reported in the literature, then the next chapters will provide details and results of deep learning modeling attacks against double arbiter PUFs, bi-stable ring PUFs, and obfuscated schemes.

4.1 Overview of attacks on PUFs

4.1.1 Machine learning based attacks

Numerical Modeling attacks against PUFs were one of the earliest attacking techniques proposed in the literature. Given a set of CRPs and using machine learning techniques like SVM, LR, and Evolution Strategy (ES), it is possible to accurately predict the PUF outcome for the whole challenge-response space. Although different variants of the arbiter and ring oscillator PUFs were proposed to add non-linearity to the circuit and countermeasure the modeling attacks, it was shown that PUFs security against modeling attacks was questionable. In [75] and [76] Ruhrmair U. et al a comprehensive study was provided on machine learning attacks on PUFs. This study showed that using LR, SVM, and ES techniques, the security of different types of PUFs could be broken up to a given size and complexity. Their attacks involved APUFs, RO PUFs, XOR APUFs, feedforward APUFs, and lightweight secure PUFs. CRPs used in [75] were collected using a model generating the delay values for PUF architectures randomly. The basic idea of this type of attack is to use machine learning techniques to solve the additive linear mathematical model of delay-based PUFs. Equations 4.1 to 4.6 shows the mathematical base to model the total delay difference between two paths of a K-stage APUF [75]. The first term in the right hand side of equation 4.1 'w' is a vector of size K+1 that encodes the information of every stage delay as shown in Figure 4.1. Equation 4.3 shows the value of every element in w in terms of '$\delta_i^{0/1}$', which denotes the delay difference at stage 'i' for crossed and uncrossed signal paths respectively as shown in Figure 4.1. The second term in equation 4.1 'φ' is a vector of size K+1 that encodes the impacts of stages delay difference on the overall circuit delay as shown in equations 4.4 and 4.6. Every element in 'φ' indicates whether it contributes to the first or second path depending on how many crosses it will encounter based on the values of the subsequent input challenge bits. Hence, the product of both vectors results in the summation of the delay differences of all PUF stages. Consequently, PUF outcome is represented by the sign of this summation (i.e. PUF response = 0 if $\vec{w}^T * \vec{\varphi}$ = -value, PUF response = 1 if $\vec{w}^T * \vec{\varphi}$= +value). Conventional ML techniques like LR and SVM try to learn the effect of every stage delay difference by assigning values for $\vec{w}^T$ elements based on the CRPs training sample and calculate a separating plane to accurately predict the APUF response.

$$\Delta Delay = \vec{w}^T * \vec{\varphi} \tag{4.1}$$

$$\vec{w}^T = (w^1, w^2, ...w^{K+1})^T \tag{4.2}$$

$$w^1 = \frac{\delta_1^0 - \delta_1^1}{2}, \quad w^i = \frac{\delta_{i-1}^0 + \delta_{i-1}^1 + \delta_i^0 - \delta_i^1}{2} \quad \text{for all i=2 ,..,k,} \quad w^{K+1} = \frac{\delta_K^0 + \delta_K^1}{2} \tag{4.3}$$

$$\vec{\varphi}(\vec{C}) = (\varphi^1(\vec{C}), \varphi^2(\vec{C}), ...\varphi^K(\vec{C}), 1) \tag{4.4}$$

$$\varphi^j(\vec{C}) = (\prod_{i=j}^{K}(1 - 2C_i))^T \tag{4.5}$$

$$T_{response} = Sign(\vec{w}^T * \vec{\varphi}) \tag{4.6}$$

This parametric model can be expanded for other delay-based PUFs. For example, XOR PUF model can be easily derived from APUF model as shown in equation 4.7. The term 'J' denotes the number of XOR inputs (i.e. the number of parallel APUFs). Similarly, the machine learning algorithm tries to find the $\vec{w}_i^T$ vectors hyperplane of dimension $(K+1)^J$ to accurately predict the XOR PUF outcome.

$$T_{xor\_response} = Sign(\prod_{i=1}^{J}(\vec{w}_i^T * \vec{\varphi})) \tag{4.7}$$

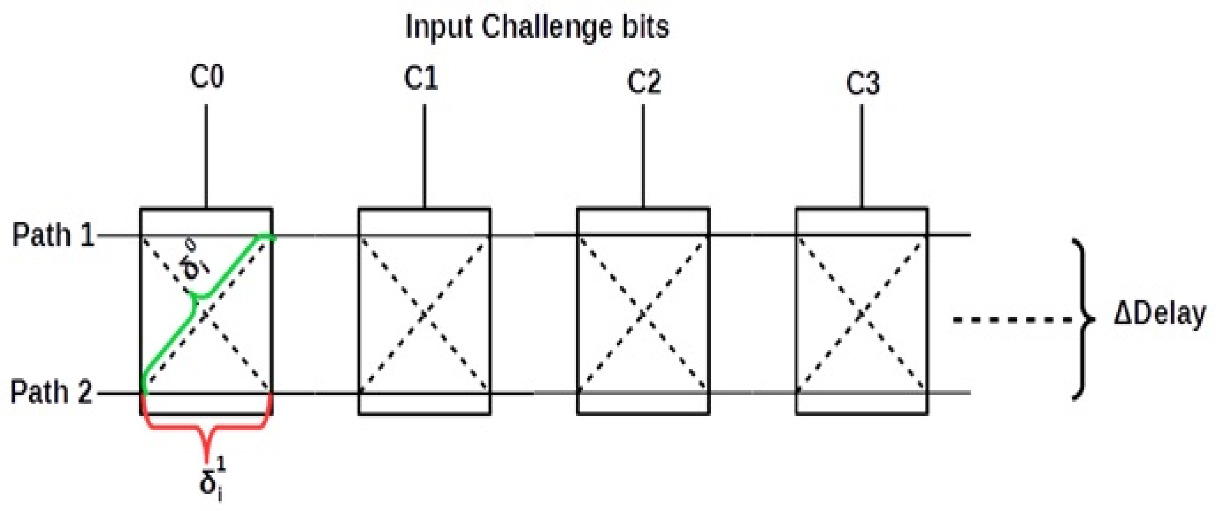


Figure 4.1: Modeling parameters of standard arbiter PUF

The work in [75] and [76] included also RO PUFs in the attack list. RO PUF under attack consists of K ring oscillators with different frequencies. The PUF inputs are two oscillators indices (i,j) to select two oscillators for the comparison. Hence it is easy to model this architecture by keeping track of all oscillators' frequencies order $(f_1, f_2, ..., f_k)$ and changing this order using a sorting algorithm by applying all possible inputs (i,j) while monitoring the output (e.g.: output = '0' if $f_i > f_j$, output = '1' otherwise). As previously mentioned, machine learning attacks were executed on model generated CRPs in [75] and it was extended in [76] to include hardware generated CRPs for APUFs and XOR PUFs using 45 nm ASICs and Spartan-6 FPGAs. Table 4.1 shows a summary of obtained results in terms of the number of CRPs and the machine learning technique used to obtain 99% accuracy.

The results show that modeling attacks against APUFs are successful using LR algorithm for up to 128 stages using model generated CRPs and 64 stages using hardware generated CRPs. Similarly, XOR APUFs could be modeled using LR for up to 5 APUFs and 128 stages. The main difference from standard APUF is that LR is not guaranteed to find a global minimum from the first trial because XOR APUF decision boundary is non-linear. Hence, LR needs to be restarted several times (N_trials) to reach a global minimum, which will be dependent on PUF parameters and the size of the training set (CRPs). Furthermore, lightweight PUF (LF PUF) showed significant resistance against machine learning attacks compared to standard and XOR APUFs, although the tested architecture

Table 4.1: Attacks results of Ruhrmair U. et al in [75] and [76]

PUF Type	No of XORs/ FF loops/ Ring Osc.	ML method	Bit Length	CRP Source	CRPs ($\times$1K)	Training Time	Prediction Rate
APUF	N/A	LR	128	Simulation	39.2	2.1 sec	99.9%
			64	ASIC	6.5	0.83 sec	99%
			64	FPGA	6.5	0.76 sec	99%
XOR APUF	5	LR	128	Simulation	39.2	17 hrs	99%
			64	ASIC	6.5	39 mins	99%
			64	FPGA	6.5	18 mins	99%
Light weight PUF	5	LR	128	Simulation	1000	267 days	99%
FF APUF	8	ES	128	Simulation	50	3:15 hrs	99%
RO PUF	1024	Quick Sort	N/A	Simulation	83.9	N/A	99%

was limited and experiments targeted the prediction of only one response bit. For the case of 128 stages and five parallel APUFs, LF PUFs needed $\sim$ 386x more training time and 2x more CRPs compared to XOR arbiter PUF. This is expected because, in LF PUFs, output bits are not directly mapped to input challenge bits nor parallel arbiter responses. Additionally, every APUF has different challenge bits from the other parallel APUFs. The FF PUF architecture under attack was restricted to ease the architecture complexity (i.e., restricting the number of forward loops in FF PUFs up to 8 non-overlapping loops with equal length and regularly distributed over the PUF path). Opposite to other PUF types, ES outperformed LR and SVM techniques and ES could break the security of FF PUFs architectures with up to eight loops and 128 stages, given the architectural restrictions mentioned earlier. Finally, the quick sort algorithm was used to order the oscillator frequencies of the RO PUF under attack depending on randomly chosen CRPs. The upper bound of needed CRPs to achieve the obtained prediction rate is given by equation 4.8, where K is the number of ring oscillators used, and ϵ is the classification error rate.

$$N_{crp} \approx \frac{K(K-1)(1-2\epsilon)}{2+\epsilon(K-1)} \tag{4.8}$$

Furthermore, Hospodar G. et al in [41], executed machine learning attacks on 64 stage

APUFs and 2-input XOR PUFs, which were physically implemented on 65 nm CMOS. Attacks were launched using artificial neural networks (ANN) and SVM and hardware collected CRPs to model the target PUFs. Obtained results showed successful modeling of a standard APUF with accuracy $> 90\%$ using 1000 CRPs, and reaching nearly 100% using 5000 CRPs. Moreover, their attack achieved 90% modeling accuracy of two-input XOR PUFs using 9000 CRPs.

Tobisch H. et al [86] introduced a more efficient implementation of the same machine learning technique used in [75] and [76]. The LR algorithm with resilient backpropagation solver (RProp) was optimized in terms of memory requirements and execution parallelization to reduce the training time needed for attacking more complex APUFs. CRPs used in experiments were generated using a model implemented by Matlab, which assumed Gaussian distribution of all PUF delay differences '${\delta_i}^{0/1}$'. The attack results are shown in Table 4.2, which shows the successful modeling of XOR APUFs of 64 stages with up to 9 parallel APUFs and 128 stages with up to 7 parallel APUFs. Furthermore, although obtained results agree with the conclusion in [75] that number of CRPs needed to model the PUFs increases exponentially with the increase of PUF complexity but it also suggested that every PUF instance has a direct impact on machine learning complexity and convergence (i.e. LR is not stuck at local minimum). A countermeasure was proposed to resist machine learning attacks using a technique called noise bifurcation [97]. The main idea of this technique is to prevent the attacker from pairing challenge bits with the corresponding response bit. More details on countermeasures will be discussed later in the countermeasure subsection.

Finally, there exist several reported machine learning attacks against BR PUFs and TBR PUFs. For example, Xu X. et al in [94] could build a model and applied SVM modeling attacks against the implementations of both PUF types on Xilinx Spartan -6 FPGA. The architectures under attack involved 32, 64, 128, 256-bit length PUFs, and collection of 1M CRPs formulated using majority votes of 11 repeated measurements. Attacks were launched using SVM with linear kernel and the target correct modeling accuracy is 95%. Table 4.3 shows the obtained results for both architectures.

As a result of the successful attacks against BR and TBR PUFs, XOR BR PUFs were proposed to resist the machine learning attacks [94]. The results of SVM attacks against XOR PUFs are shown in table 4.3 and suggests that an architecture with more than 2-input XOR is more resistant and could not be modeled using the SVM algorithm. Gangi F. et al [26] proposed a new attack against BR and TBR PUFs that does not require deriving a mathematical model of the PUF parameters. The main idea is to exploit the challenge bits with higher influence on PUF response (influential challenge bits) to construct a machine learning based boosted model that can predict the PUF outcome

Table 4.2: Attacks results of Tobisch H. et al [86].

PUF Type	Bit Length	ML method	No of XOR inputs	CRP Source	CRPs (×1K)	Training Time	Prediction Rate
XOR APUF	64	LR	4	Simulation	10	16 sec	>98%
			5		45	2:46 mins	
			6		210	30:24 mins	
			7		3K	2:43 hrs	
			8		40K	6:31 hrs	
			9		350K	37:46 hrs	
	128	LR	4	Simulation	22	2:24 min	>98%
			5		325	12:11 mins	
			6		15K	4:54 hrs	
			7		400K	66:53 hrs	

with high probability. Experiments conducted using 30K CRPs collected from 64-stage BR and TBR PUF implementations on Altera Cyclone IV FPGAs. Adaptive boost algorithm [24] was used to create the boosted classifier built over the initial weak learners, which depends on single influential bits. Obtained results showed that boosting technique could successfully model both PUFs up to 99% prediction accuracy using 50 boosting iterations. Table 4.4 gives a summary of PUFs successfully attacked by machine learning techniques and the source of CRPs used in training.

4.1.2 Hybrid side channel/Machine learning attacks

As discussed in the previous subsection, machine learning based modeling attacks provided an effective way to break PUFs security. This effectiveness led to proposing variants of delay-based PUFs architecture to increase the non-linearity in the delay path and make the PUF more resilient to modeling attacks. Modeling attack results suggest that lightweight PUFs and FF PUFs introduced more restrictions on this type of attack and limit its ability to correctly predict the PUF outcome (i.e. only successful on restricted architectures with a specific number of stages and loops). Hence, Hybrid techniques were introduced by extracting side-channel leaked information to help ease the constraints on modeling parameters. Side-channel leaked information include power side-channel [48], timing side-channel [47], electromagnetic side-channel [66] and differential fault analysis [20]. The main purpose of combining side-channel with machine learning attacks on PUFs is to

Table 4.3: Attacks results of Xu X. et al [94].

PUF Type	Bit Length	ML method	# XOR inputs	CRP Source	CRPs	Training Time	Prediction Rate
BR PUF	32	SVM (Linear kernel)	N/A	FPGA	800	Not reported	>95%
	64				1350		
	128				2400		
	256				5500		
TBR PUF	32	SVM (Linear kernel)	N/A	FPGA	420	Not reported	>95%
	64				720		
	128				1300		
	256				2700		
XOR BR PUF	32	SVM (Polynomial kernel)	2	FPGA	800	3 sec	>95%
	64				4000	10 sec	
	128				18000	6 min	
	256				—	—	50.8%
	32	SVM (Polynomial kernel)	3	FPGA	1200	5 sec	>95%
	64				7200	24 sec	>95%
	128				—	—	50.1%
	256				—	—	50.1%
	32	SVM (Polynomial kernel)	4	FPGA	—	—	50.1%
	64				—	—	50.3%
	128				—	—	49.8%
	256				—	—	50.1%

decrease the workload of building machine learning based models, the latter is still the key applicable solution on the Strong PUFs [93]. In [59], Mahmoud A. et al combined ML with power consumption side-channel information (SPA) to substantially improve the reach of modeling attacks on electrical Strong PUFs. Targeted architectures included XOR APUFs and lightweight PUFs, which showed resilience against pure machine learning based modeling attacks. Their power side-channel scheme informs the attacker of every single arbiter outcome that is input to the final XOR gate of the XOR Arbiter PUF or Lightweight PUF. The attack scheme depends on what is called “good” CRPs which result in all “ones” or all “zeros” responses. These types of responses can be identified using the power side channel and allow the attacker to easily build a model for every single APUF, then predict the XOR/lightweight PUF outcome correctly. Experiments were conducted using synthetic CRPs generated by PSPICE simulation. Table 4.5 shows the obtained results on XOR and

Table 4.4: Summary of successfully attacked PUFs including ML technique used and source of CRPs.

PUF Type	# of XORs/ FF loops/ Ring Osc.	ML method	Bit Length	CRP Source	Refs.
APUF	N/A	LR	128	Simulation	[75],[76]
		LR, SVM, ANN	64	ASIC	[75],[76],[86]
		LR	64	FPGA	[75],[76]
XOR APUF	UPTO 9	LR	128	Simulation	[75],[76],[86]
			64	ASIC	[75],[76]
			64	FPGA	[75],[76],[86]
Lightweight PUF	5	LR	128	Simulation	[75],[76]
FF APUF	8	ES	128	Simulation	[75],[76]
RO PUF	1024	Quick Sort	N/A	Simulation	[75],[76]
BR PUF	N/A	SVM (Linear kernel), Adaptive boost	Up to 256	FPGA	[94], [97]

lightweight PUFs.

Apart from using synthetic CRPs, the dependency on "good" CRPs that produce either all '1's or '0's limits the number of useful CRPs, which is inefficient when attacking more complex PUF architectures that require more CRPs for training. Note that the number of CRPs shown in Table 4.5 include both useful and non-useful CRPs. On the other hand, although the prediction rates are slightly less than the numbers obtained in [75], combining SPA with machine learning could extend the attack to successfully model XOR/lightweight PUFs with a higher number of parallel APUFs and stages in less computational time. Another work by Merli D. et al [66] proposed the use of electromagnetic side-channel leaked information to break the security of RO PUFs. Using near-field electromagnetic cartography methods, a full model of an RO PUF could be extracted from EM measurements. The measurements are taken from the backside of the chip under attack to obtain stronger EM radiation.

Figure 4.2 shows the de-capsulation steps on the Xilinx Spartan XC3S200 FPGA chip under test. 256 RO oscillators were implemented on-chip to examine the effect of the de-capsulation process by measuring their frequency before and after applying de-capsulation.

Table 4.5: Attacks results of Mahmoud A. et al [59].

PUF Type	ML method	# of stages	# of XORs	Prediction accuracy	CRPs (×1K)	Training time
Lightweight PUF	LR	64	4	97.3%	40	18 sec
			5	96.6%	80	22 sec
			6	96.6%	200	44 sec
			7	96.6%	500	91 sec
			8	96.6%	1000	98 sec
			9	96.6%	2000	105 sec
		128	4	97.4%	80	46 sec
			5	97.6%	200	122 sec
			6	97.4%	500	191 sec
			7	96.8%	1000	200 sec
			8	96.4%	2000	238 sec
			9	96.1%	4000	303 sec
		256	4	97.6%	200	200 sec
			5	97.4%	500	340 sec
			6	96.8%	1000	400 sec
			7	96.4%	2000	490 sec
			8	96.1%	4000	580 sec
			9	96.1%	8000	700 sec
XOR APUF	LR	64	4	95%	40	10 sec
			5	95%	80	37 sec
			6	95%	200	160 sec
			7	95%	500	247 sec
		128	4	95%	80	34 sec
			5	95%	200	170 sec
			6	95%	500	374 sec

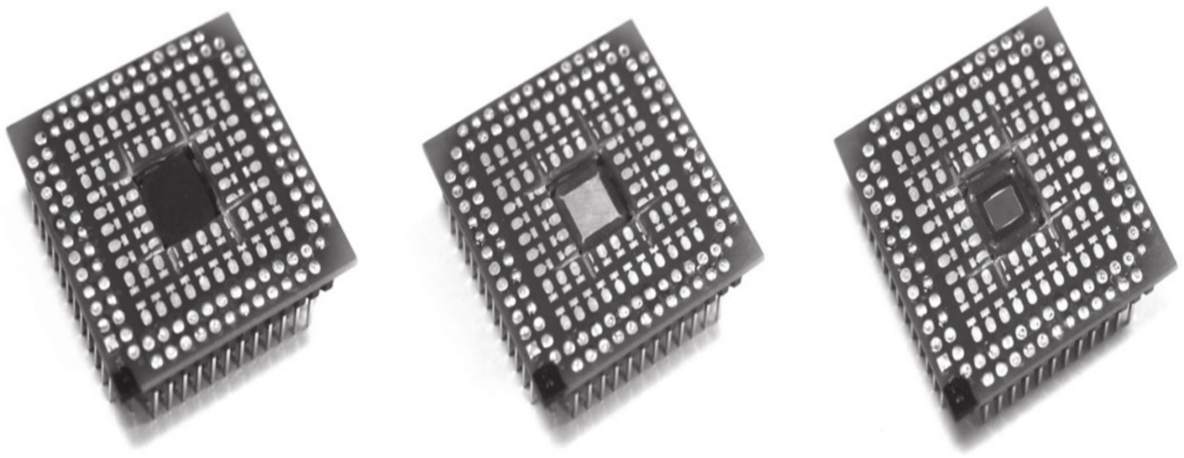


Figure 4.2: Step-by-step backside de-capsulation of an FPGA [66].

Obtained results of RO frequency maps did not show significant change, therefore the attack can be considered of semi-invasive type. The first step of the side channel EM analysis was to determine the frequency range of ROs on the chip, then identify the area of high-frequency leakage to distinguish between every two distinct RO frequencies. This is done by applying a differential analysis of frequency amplitudes measured on-chip while launching comparisons between every two RO PUFs. For example, Light areas in Figure 4.3 indicate amplitude differences between comparisons and therefore represent points of interest for an attacker.

Measurements were done using Langer ICR near-field EM probe, which has 150μm diameter and 100μm resolution as shown in Figure 4.4. The chip under attack was divided into a grid of 50×42 measurement points over the 4.8mm$\times4.0$mm die to obtain a map of location-dependent EM radiation of RO PUFs and fully model the eight ROs PUF in the experiment.

Additionally, other proposals suggested that photonic emissions by CMOS transistors during logic transitions can be used as a side-channel to gain information about PUF behavior. Helfmeier C. et al proposed the use of photonic emissions analysis (PEA) to physically characterize and clone SRAM PUFs [38]. The experiment was conducted on Atmel ATmega328P microcontroller as the chip under test after removing the packaging and the bulk silicon of the device backside. The characterization of SRAM cells was done by capturing their near-infrared photonic emission using Hamamatsu Phemos 1000 NIR-

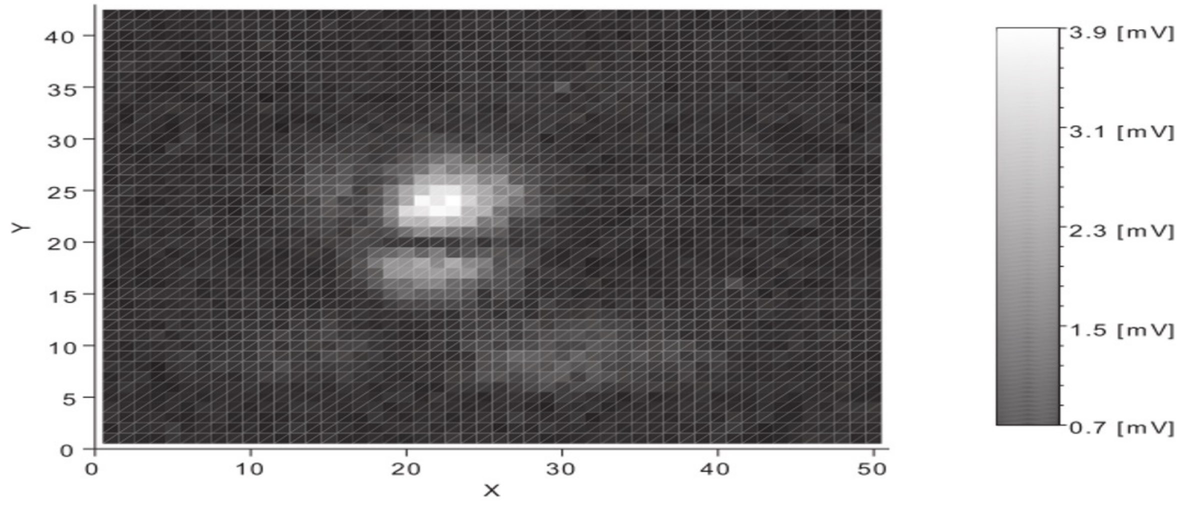


Figure 4.3: Accumulated frequency amplitude differences map [66].

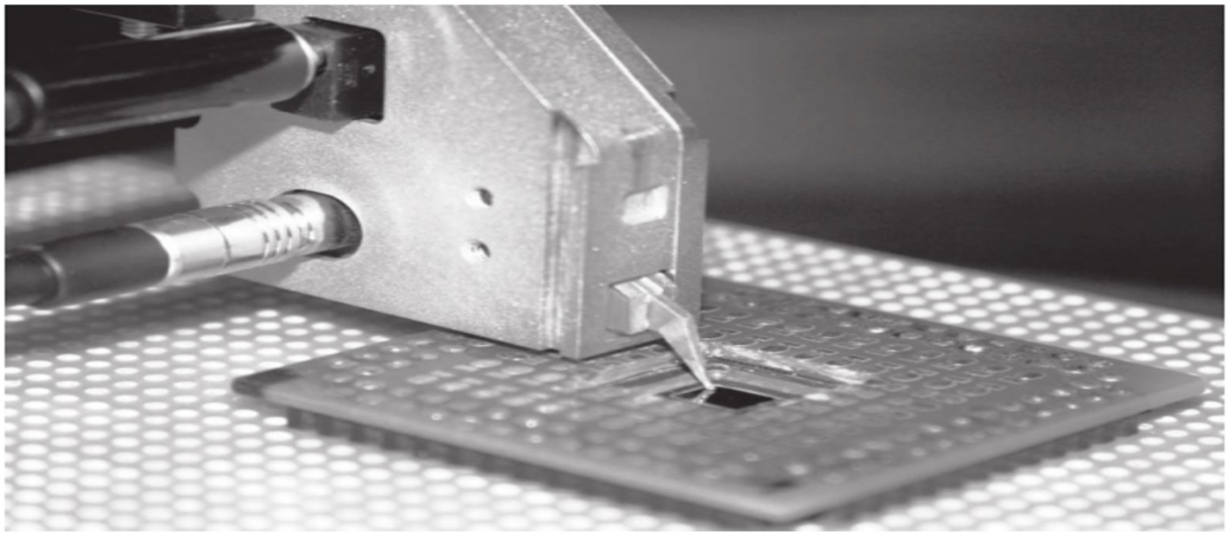


Figure 4.4: Near-field probe close to FPGA die [66].

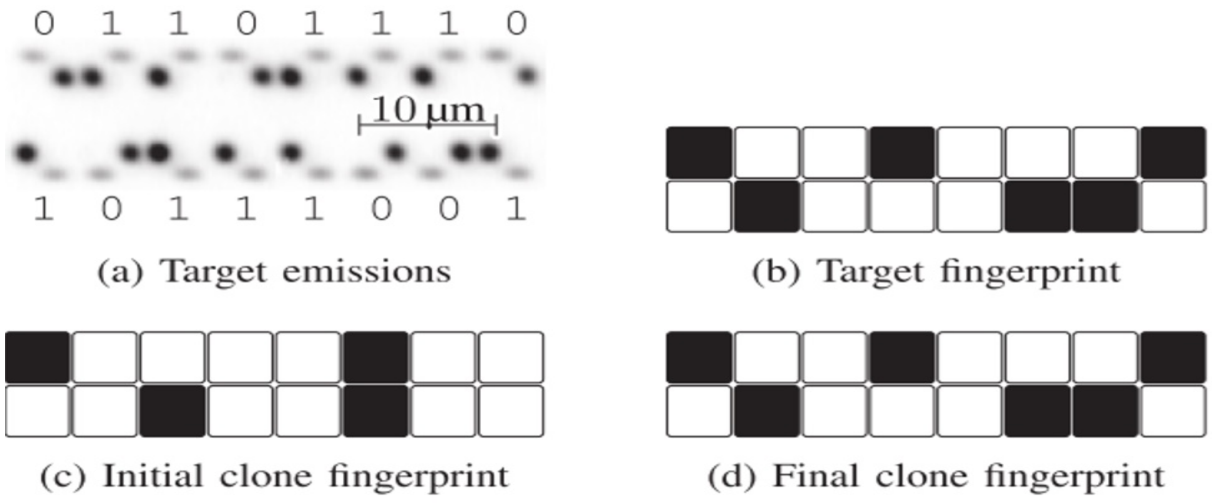


Figure 4.5: SRAM photonic emission fingerprint [38].

CCD. The program loop on the chip was executed with high frequency to increase the photonic emissions generated and facilitate the characterization of SRAM PUFs as shown in Figure 4.5 a) and b). Consequently, The SRAM PUF could be cloned using a Focused Ion Beam (FIB), which can trim individual transistors to alter their dynamic performance and leakage characteristics, hence clone the SRAM startup behavior as in Figure 4.5 d).

Another research analysis by Tajik S. et al [84] used photonic emissions with timing side-channel analysis to fully characterize arbiter PUF delays. Furthermore, this attack could measure every PUF internal delay with a 6ps resolution. Hence, it only requires a minimal number of physical measurements to solve the linear additive system of delay-based PUFs. Moreover, it does not require direct access to a large CRP set to model PUF like previously discussed techniques. The main idea is to measure the time between enabling the operation of the PUF and the photonic emission of the output at the last stage. By using CRPs with Hamming Distance (HD) = 1 with a reference CRP at every stage, the delay difference between the two paths can be identified. Figure 4.6 shows the setup of Experiments conducted on 8-bit PUFs implemented on Altera MAX V CPLD and used an optimized infrared microscope equipped with a scientific near-infrared (NIR) Si-CCD camera for spatial analysis of circuit components. A free-running InGaAs avalanche detector in Geiger Mode (SPAD) is used for the temporal analysis of the photonic emission. Its sensitivity covers a wavelength range between 1 to 1.6μm with a peak quantum efficiency of 20%. Results showed that the 8-bit PUF could be fully characterized using 9 CRPs,

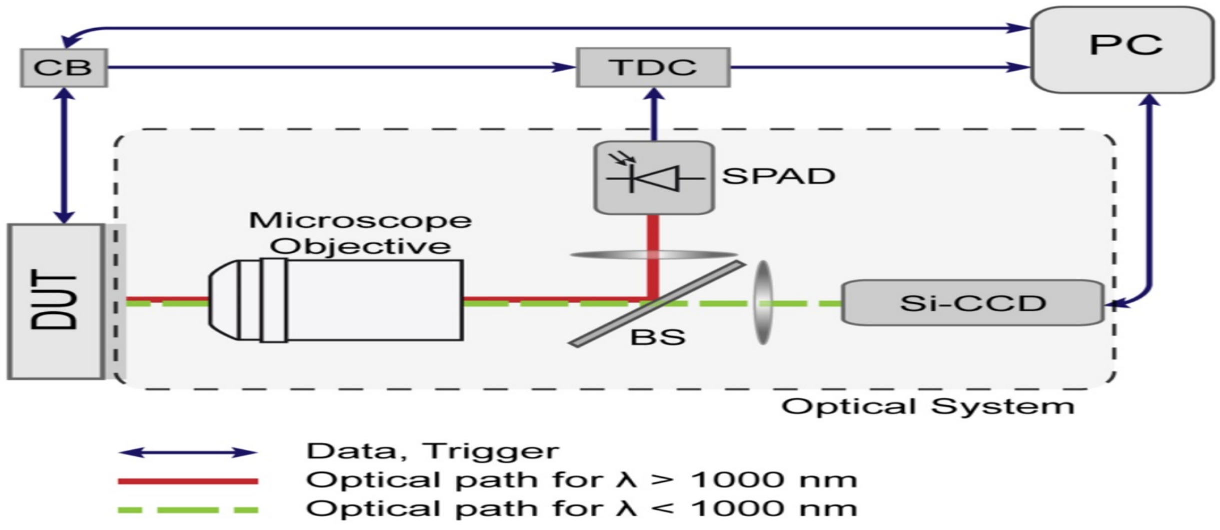


Figure 4.6: Controlling the DUT with the CB and capturing emitted photons from the DUT by SI-CCD camera and InGaAs-SPAD [84].

which is far less than required CRPS by modeling attacks. On the other hand, this attack scheme requires direct access to the device under test and applying the same challenge a million times to provide enough photonic emission. Table 4.6 provides a summary of PUF types broken by hybrid attack schemes.

4.1.3 Fault injection attacks

Fault injection is the last type of attack proposed in the literature to clone PUFs. The behavior of the system after a fault is injected can lead to a side-channel leak used by the attacker to break the system security. A fault can be injected in different ways, either by changing operating environmental conditions (e.g. temperature, operating voltage), or using the glitching of clock/power, electromagnetic pulses, or laser pulses. These injection methods cause bits values to flip, masking of clock cycles, change of response times and many other changes that affect the chip overall behavior. The first fault injection attack on delay-based PUFs has been conducted by changing the environment temperature of the chip to increase the number of unreliable CRPs [70]. The attacker can utilize unreliable CRPs as side-channel information to enhance the efficiency of modeling attacks. Their approach showed significant performance for 64-bit APUFs and RO PUFs with 40 inverters. Both PUFs were implemented on 65nm CMOS technology and CRPs were collected from

Table 4.6: Summary of successfully attacked PUFs including Side channel techniques used and source of CRPs.

PUF Type	# of XORs/ FF loops/ Ring Osc.	ML method	Bit Length	CRP Source	Refs.
LF PUF	9	PSA + LR	64	Simulation	[59]
			128		
			256		
XOR APUF	7	PSA + LR	64	Simulation	[59]
	6		128		
RO PUF	7	EMA	8	FPGA	[66]
SRAM PUF	—	PESA	—	Micro-controller	[38]
APUF	—	PESA + Timing side channel analysis	8	CPLD	[84]

the silicon chip. This idea is further extended by changing the supply voltage for specific responses and observe if they flip [7]. Using unreliable responses as new information enables the attacker to model a controlled arbiter PUF using power side-channel information. This is helpful since controlled arbiter PUFs limit the number of accepted CRPs. Experiments conducted in [7] used simulated CRPs generated by an HSpice model of 128 APUF in 45 nm technology. The result of using ES technique after applying +/-0.1V fault injection showed 97

Another fault injection attack was proposed in [83] by Tajik S. et al. The attack scheme uses laser beams to inject a fault into a point of interest to help accelerate machine learning attacks on complex non-linear PUF architectures. The points of interest in PUFs are identified and localized by applying spatial photonic emission analysis from the IC backside. As shown in Figure 4.7, Fault is injected to disable all chain outputs by disabling clock input signal except for only one APUF. Hence, the XOR PUF output is a copy of this APUF response and every single PUF can be modeled at a time. Experiments on two-input XOR PUF showed that an XOR or lightweight secure arbiter PUF can be modeled in a few seconds with far fewer numbers of CRPs, unlike pure ML attacks, which requires a lot of time and CRPs to accurately model complex PUFs.

The same technique was applied against RO-PUF as shown in Figure 4.8. The ex-

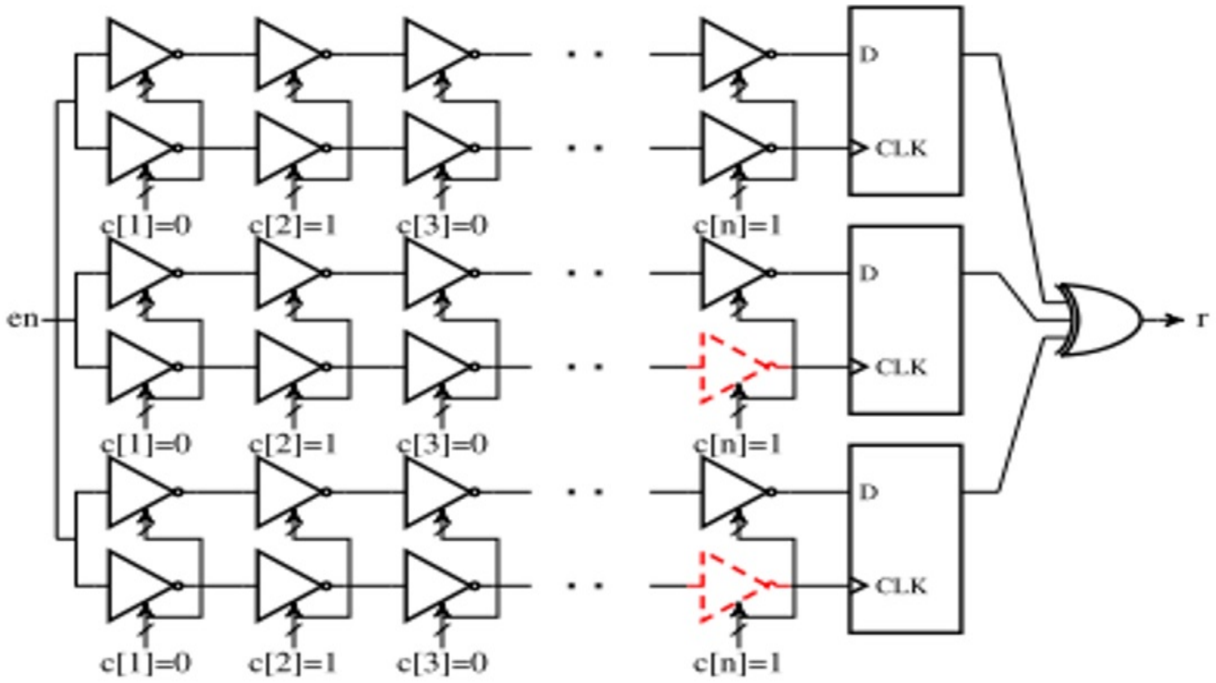


Figure 4.7: Points of laser attack for XOR-PUF [83]

periments on RO PUF showed that the entropy can drastically be decreased by disabling several ROs in the PUF. Experiments were conducted on real implementations of RO PUF with 3 inverters and 2-XOR APUF (number of stages were not reported) on Altera MAX V CPLD. Moreover, photonic emission analysis was done using a Hamamatsu PHEMOS 1000 photon emission and laser scanning microscope. Table 4.7 summarizes the fault injection attacks on PUFs and the source of CRPs used.

4.2 Modeling attacks on double arbiter PUFs

As previously mentioned in subsection 3.1.2, DAPUF was especially proposed in [85] and [54] to overcome the routing constraints on FPGA chips and complicate the relationship between input challenges and responses with minimum hardware overhead. In a typical DAPUF design, using N APUFs leads to a response that is the output of N(N-1)-input XOR. Hence, DAPUFs showed more resistance against modeling attacks using conventional machine learning (ML) techniques and its security could not be broken as reported in previous research [85],[96].

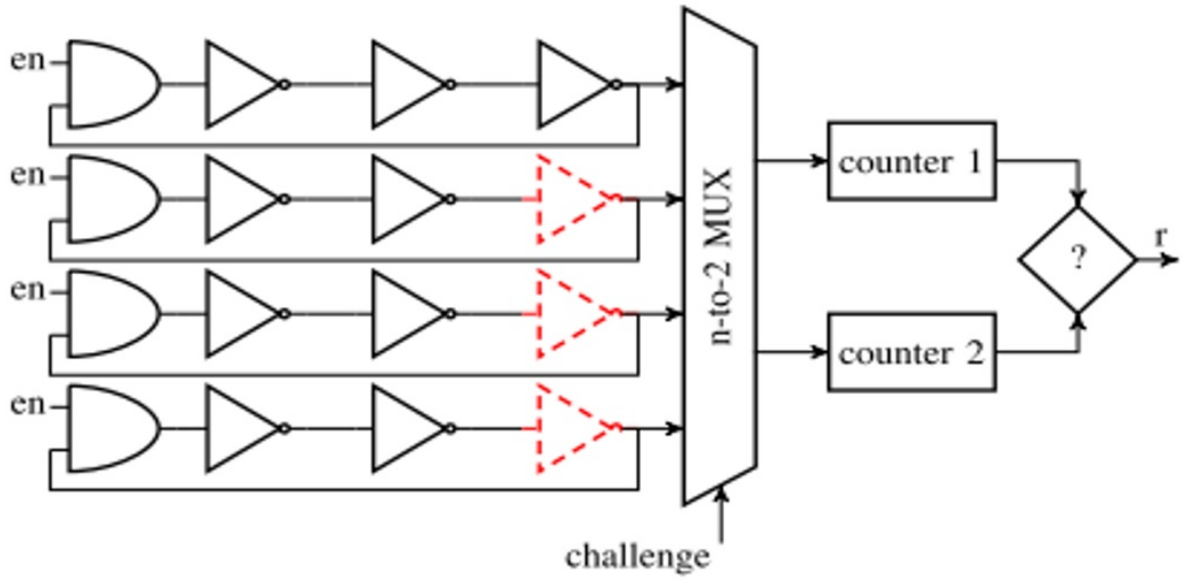


Figure 4.8: Points of laser attack for RO-PUF [83]

Table 4.7: Summary of successfully attacked PUFs using fault injection techniques and source of CRPs.

PUF Type	# of XORs/ FF loops/ Ring Osc.	ML method	Bit Length	CRP Source	Refs.
XOR APUF	2	Laser pulses	Not reported	CPLD	[83]
RO PUF	—	Voltage change	40	ASIC	[70]
RO PUF	4	Laser pulses	3	CPLD	[83]
APUF	—	Voltage	64	CPLD	[7]
APUF	—	change	128	Simulation	

The contributions of this research with respect to DAPUFs attacks are as follows:

1. Successfully modeling 2-1 DAPUFs using conventional ML techniques with CRPs collected from a real DAPUF hardware implementation.
2. The first reported successful modeling attacks against 3-1 DAPUFs using deep learning (DL) techniques which shows it performs better than conventional ML algorithms.
3. Suggest DL techniques to enhance the modeling accuracy of 4-1 DAPUF compared to previous research [96].

4.2.1 DAPUFs architectures and modeling attacks

As was shown in Figure 3.5, the main idea of this approach is to use two identical APUFs placed adjacent to each other to eliminate routing differences. Every delay path has identical routing delays compared to the matching path in the second PUF. Hence, PUF response is determined by the matching delay paths race of the DAPUFs

The DAPUF design is further extended to make it more complex and harder to model by introducing an XOR operation between PUF responses. As mentioned earlier, DAPUFs designs use minimum hardware overhead and can produce $O(N^2)$ input XOR response using $O(N)$ APUFs. For example, Figure 3.5 and Figure 3.6 show that the 2-1 DAPUF, 3-1 DAPUF, and 4-1 DAPUF are equivalent to 2-input XOR APUF, 6-input XOR APUF, 12-input XOR APUF respectively.

Modeling attacks using conventional ML techniques against DAPUFs and equivalent XOR PUFs have been reported in the literature. For example, Machida, T., et al [85] showed that LR attacks against DAPUFs were not successful. Furthermore, modeling accuracy was in range 56% - 80% for 2-1 DAPUF, $\sim$ 56% for 3-1 and 4-1 DAPUFs. However, their training sample was small (1K CRPs) and did not provide sufficient evidence of DAPUFs resistance. On the other hand, Yashiro, R., et al. [96] provided a more comprehensive analysis using 50K CRPs and their attacks were executed using SVM and DL. Their reported results showed 2-1 DAPUF could be successfully modeled using DL techniques with accuracy 90% and $\sim$ 85% using SVM. However, it was not clear how many PUF instances were under attack. Additionally, their results showed that DL and SVM could not successfully model 3-1 and 4-1 DAPUFs with modeling accuracy is 68% and 62% respectively.

Finally, as mentioned in subsection 4.1, modeling attacks against XOR PUFs were reported successfully in literature. However, these attacks were executed against maximum

6-input XOR PUFs using real PUF implementations and 200K CRPs [76] and 9-input XOR PUFs using simulated PUFs and 350K CRPs [86]. Therefore, the security performance of 3-1 DAPUF was tested by applying the same modeling technique used in [76] against 6-input XOR PUF. The poor results based on this test justifies shifting our focus to exploit DL techniques to break 3-1 DAPUF security.

4.2.2 Methodology and Experimental Setup

This subsection discusses in detail the mathematical models, network architectures, and the environment setup used to attack the DAPUFs.

4.2.2.1 The Mathematical Model of DAPUFs

As previously discussed, the DAPUF response is the output of an XOR. Therefore, the model used to attack DAPUFs is the same mathematical model used for XOR arbiter PUFs [76]. Consequently, The output response of a DAPUF is the multiplication of every single path response as shown in equation 4.9.

$$Txor = \prod_{i=1}^{l} sign(\vec{w_i^t}\vec{\phi_i}) \tag{4.9}$$

As was discussed in section 4.1 The $(\vec{w_i^t})$ is the delay difference vector of a single APUF and responsible for encoding delay difference at every stage [75], whereas $\vec{\phi_i}$ is the feature vector that represent the impact of every stage delay difference on the overall PUF response. If (δ) is the delay difference between the upper and lower path at a certain stage then its impact will be (+) or (-) depending on how many times paths will go straight or crossed after this stage. Hence, it is a function of input challenge bits and is a string of (1) and (-1) as depicted in equation 4.10 [75]. Note that 'k' represents the number of stages, 'l' is the stage position in APUF and 'C_i' is the challenge bit value at the i_{th} position.

$$\vec{\phi}(\vec{C}) = (\phi^1(\vec{C}), ...\phi^k(\vec{C}), 1), \phi^1(\vec{C}) = \prod_{i=l}^{k}(1 - 2C_i) \tag{4.10}$$

The final value of Txor is either (1) or (-1) representing (1) or (0) responses respectively. Hence, modeling the DAPUF response can be considered as a binary classification problem, which can be solved by conventional machine learning techniques (e.g. LR, SVM).

Equations 4.11 and 4.12 are derived from equation 4.9 to calculate the decision boundary needed to classify the PUF response. Equation 4.11 is doing the classification by determining a non-linear decision boundary, which requires $l \times (k+1)$ parameters (l is the number of XOR inputs and k is the number of PUF stages). Whereas equation 4.12 takes a further step and calculates the linear decision boundary by applying outer product among the wights and features of all XOR input PUFs. The latter approach requires $(k+1)^l$ parameters for every single CRP, which scales exponentially when increasing the XOR inputs. Please note that 'l' represents the number of XOR inputs in equations 4.9, 4.11, and 4.12.

$$Txor = sign(\prod_{i=1}^{l}(\vec{w_i^t}\vec{\phi_i})) \tag{4.11}$$

$$Txor = sign(\bigotimes_{i=1}^{l}\vec{w_i^t}\bigotimes_{i=1}^{l}\vec{\phi_i}) \tag{4.12}$$

The 64 stage 2-1 DAPUF is similar to 64 stage 2-input XOR PUF, thus the number of parameters for every single CRP is $(64+1)^2 = 4225$. Due to this relatively low number of parameters, building a model for the 2-1 DAPUF using logistic regression is a reasonable choice and was adopted in this work. However, deep learning techniques were used in attacking 3-1 and 4-1 DAPUFs because the number of parameters scales exponentially (65^6 in the case of 3-1 DAPUF and 65^{12} in case of 4-1 DAPUF). Furthermore, attacks against 3-1 and 4-1 DAPUFs were invoked using LR with non-linear decision boundary as a proof that conventional machine learning techniques are not successful in breaking larger DAPUFs. This is in contrast to the same technique performance against similar XOR-PUFs, which were accurately modeled using non-linear decision boundary in previous research [76][86].

4.2.2.2 The Architectures of The Deep Neural Networks

Deep neural networks (DNNs) are artificial networks with multiple hidden layers between the input and output that can model complex non-linear relationships among inputs better than shallow ANNs. They have been on the rise for some time now and widely used in many applications and complex tasks (e.g. object recognition, classification, text processing tasks). These types of networks start to build up a complete inference about the complex problem by gaining partial knowledge through the multiple hidden layers and aggregate them together at the end to provide an accurate classification/decision. For example, when a specific network tries to classify an object, its shallow layers extract features of edges and contours. Then, the deeper layers connect between features to construct shapes and classify the object at the output layer. Hence, the problem of modeling a complex PUF architecture may be solved using this approach. The DNN can learn the complex relationships among different stages by discovering the easier correlations between challenge features first and build upon that through the network to classify the final PUF response. The type of DNN used in this work is the feed-forward network, which means the flow of data goes in one direction through multiple layers from input to output as shown in Figure 4.10.

Many deep neural network architectures and configurations have been investigated in the experiments invoked to find the network that can successfully learn the complex relationships of 3-1 and 4-1 DAPUF architecture. The main idea is to build a network that emulates the DAPUF architecture, which makes the training process easier for the network to learn the non-linear relationships among different stages of DAPUF from the same/different paths.

The first architecture is illustrated in Figure 4.9 where every path handles one APUF of the six PUFs contributing to 3-1 DAPUF output. The final result equals the six PUFs multiplication and the output of the probability of the response being '1' or '0'. All layers are fully connected because convolutional layers did not produce good results on this problem and could not capture all the relationships among different stages. Furthermore, the number of neurons in every layer is 2000 after running many experiments to tune the network. Although after using $65 \times 6 = 390$ neurons results were enhanced but the best accuracy performance obtained when using 2000 neurons in every layer. Also, note that for 4-1 DAPUF the network has 12 branches instead of six.
The second architecture is shown in Figure 4.10 and it is 12 Fully Connected (FC) layers for 3-1 DAPUF and the number is increased for the case of 4-1 DAPUF to be 18 layers. At the end of the network, there is a dropout layer to avoid over-fitting problems and train a more generalized model. The second architecture introduced nearly the same accuracy results while taking nearly 40% less training time. Hence, the reported results in the next section are based on the second architecture. Furthermore, both architectures used adaptive moment estimation (Adam) optimization algorithm proposed by Kingma, D. and Ba, J. [46] because experiments showed that it converges faster and introduces better accuracy results than the normal gradient descent algorithm. More details on the training parameters and activation functions are provided in subsection 4.3.3.

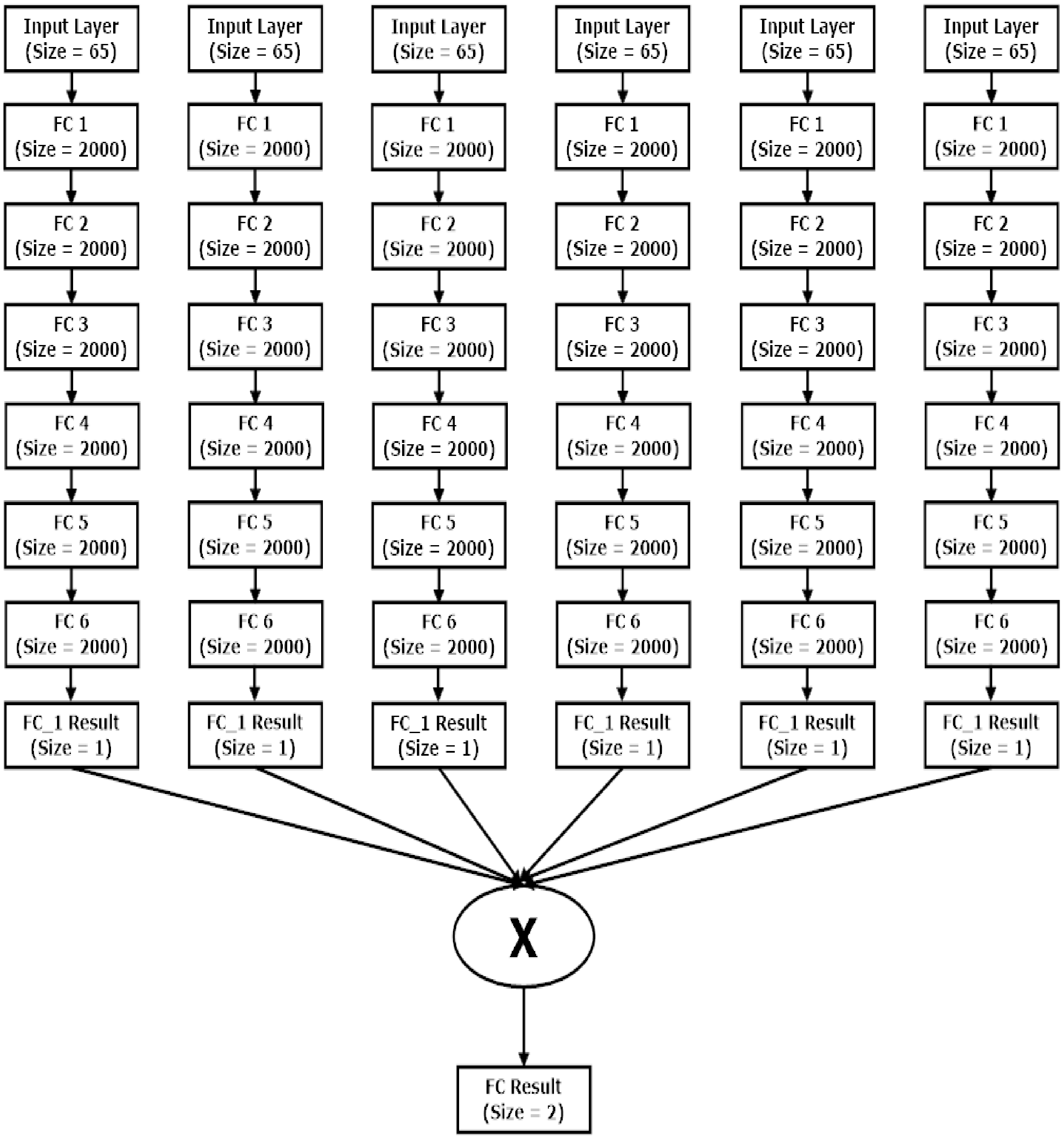


Figure 4.9: N-Branch Fully Connected with Multiplication Deep Neural Network

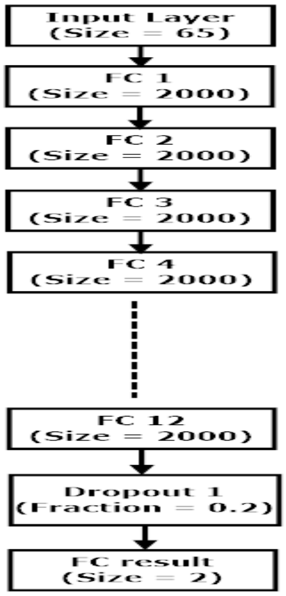


Figure 4.10: 1-Branch Fully Connected Deep Neural Network

4.2.2.3 Hardware and Software Experimental Setup

DAPUF architectures were implemented on Mojo V3 development boards, each containing a Spartan 6 XC6SLX9 FPGA [5] (45nm process technology). Real CRPs were collected from the boards and all readings were performed under normal voltage and temperature conditions. The experiments used a random CRP set generated using a linear-feedback shift register (LFSR) as in [76]. The only difference is that LFSR is implemented on chip. Finally, The Xilinx ISE Design Suite 14.7, Xilinx PlanAhead 14.7, and Xilinx FPGA editor are utilized to implement, manually place and route the DAPUF designs.

Logistic regression attacks were executed on Intel 8th Gen I7-8250 CPU with 16GB RAM of memory. The code for LR with a non-linear decision boundary is implemented by Ruhrmair, U. et al [75] and is available online [1]. Their implementation with RProp optimization algorithm was successful in attacking up to 6-input XOR PUFs. Hence, it was used to attack 3-1 DAPUF to prove The PUF resistance against conventional ML techniques and justify the need for deep learning algorithms.

Deep learning attacks were executed on a Nvidia GeForce GTX 1080 Ti GPU card with 11GB RAM and worth around 800$. The code implementation for network architectures, training, and evaluation was done using Tensorflow v1.2. The CRP set was generated using

Table 4.8: The randomness of 2-1 DAPUFs under attack.

	Chip-A	Chip-B	Chip-C
Response Randomness	43.9%	40.2%	57.3%

the same HW LFSR. However, a set of 20M CRPs were collected to have enough samples to train the massive deep neural networks.

4.2.3 Empirical Results

This section shows in detail the statistical properties of the PUFs under attack and the obtained results of the modeling attacks against 2-1, 3-1, and 4-1 DAPUFs.

4.2.3.1 LR Attacks Against 2-1 DAPUFs

Three 64 stages 2-1 DAPUFs were implemented on three Mojo V3 boards using the same bit configuration for all FPGA chips. The inter-chip hamming distance is 45% and randomness of every PUF response (percentage of '1' responses) is shown in Table 4.8. As discussed before, the CRP set size is 1M and all accuracy results shown are using a test set size = 1M - training set size.

Figure 4.11 shows the modeling attack results against the three 2-1 DAPUF instances. The maximum accuracy performance using 100K CRPs for training is 93.4%, 81.7% , and 82.4% for Chip A, B, and C respectively. The graph shows that at training size equal to 20K CRPs, accuracy could reach $\sim$ 80% - 90%. By comparing these results with what was reported in [85] by Machida, T. et al (2-1 DAPUF accuracy 56%, 69%, 80%), It is obvious that 2-1 DAPUFs are not secure against LR with a linear decision boundary modeling attacks. Furthermore, Yashiro, R. et al reported a successful modeling attack on 2-1 DAPUF using deep learning with 90% accuracy using 40K CRPs [96]. However, obtained results show that using LR with linear decision boundary can achieve better accuracy with half the number of CRPs and less cost because of no need to use deep learning techniques and GPUs.

4.2.3.2 LR and Deep Learning Attacks against 3-1 DAPUFs

Similar to the previous experiments, three 64 stage 3-1 DAPUFs were implemented on three Mojo V3 boards. The inter-chip hamming distance is $\sim$ 39% and the randomness

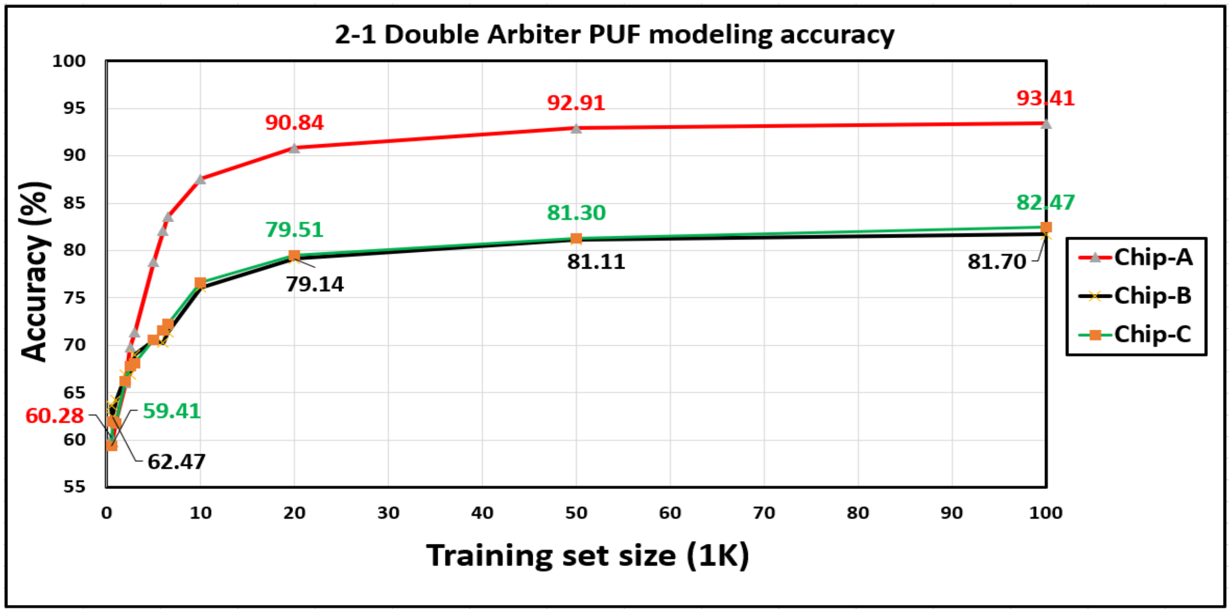


Figure 4.11: LR with Linear Decision Boundary Attack Results on 2-1 DAPUF

Table 4.9: The randomness of 3-1 DAPUFs under attack.

	Chip-A	Chip-B	Chip-C
Response Randomness	54.4%	46.7%	47.9%

of every PUF response is shown in Table 4.9. The maximum training set size is 17M and The training/test set ratios is 90/10%. LR attacks were stopped after using 4M training set because it took a long time and increasing the training data did not seem to help the model to converge.

Figure 4.12 shows the modeling attack results against 3-1 DAPUFs. The maximum accuracy performance using 17M CRPs for training is $\sim$ 86%. Moreover, The models could reach at least 83% accuracy using only 4M CRPs for training. On the other hand, the LR with non-linear decision boundary failed to converge and achieve more than 76% accuracy and took a long time because it is run on CPU. Typically, in all cases with a huge training set, it took $\sim$ 1hr-2hrs to reach 80% accuracy then within 3-5 hrs to reach the maximum accuracy. Note that this time is taken while training on one GPU. Furthermore, results show that the LR with non-linear decision boundary and the deep learning architectures used in this work are more successful than the conventional ML and DL techniques used in [55] and [96] respectively. Their reported attack results show a modeling accuracy of

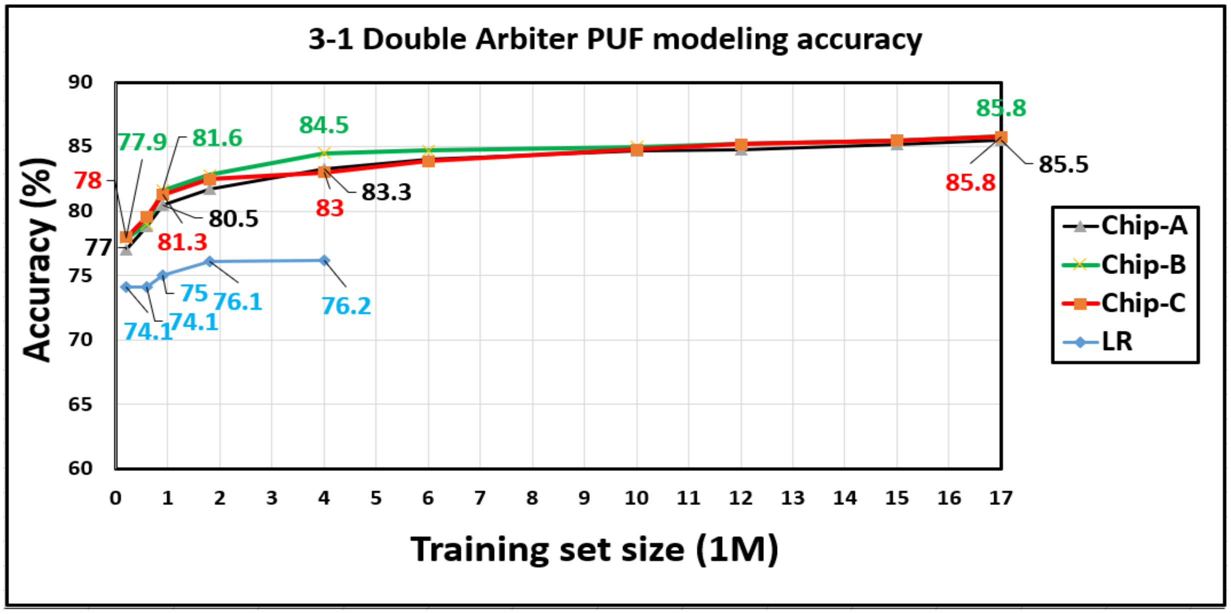


Figure 4.12: LR and Deep Learning Attacks Results on 3-1 DAPUF

Table 4.10: The Randomness of 4-1 DAPUFs under Attack.

	Chip-A	Chip-B	Chip-C
Response Randomness	48.7%	40.9%	51.8%

56% - 68% using LR, SVM, and DL techniques.

4.2.3.3 Deep Learning Attacks against 4-1 DAPUFs

The same experiments were run on three instances of 4-1 DAPUFs on three different Mojo V3 boards. The inter-chip hamming distance among the three instances is ∼ 50.9% and the randomness of every PUF response is shown in Table 4.10. Similarly, The maximum training set size is 17M and The training/test set ratios is 90/10%.

The modeling attack results against 4-1 DAPUFs are shown in Figure 4.13. The maximum accuracy performance using 17M CRPs for training is 71.3%, 81.5%, and 73.2% for chip A, B, and C respectively. The results show that models reach 70%-80% accuracy at training set size of only 6M CRPs. The increase of training set size from 6M to 17M

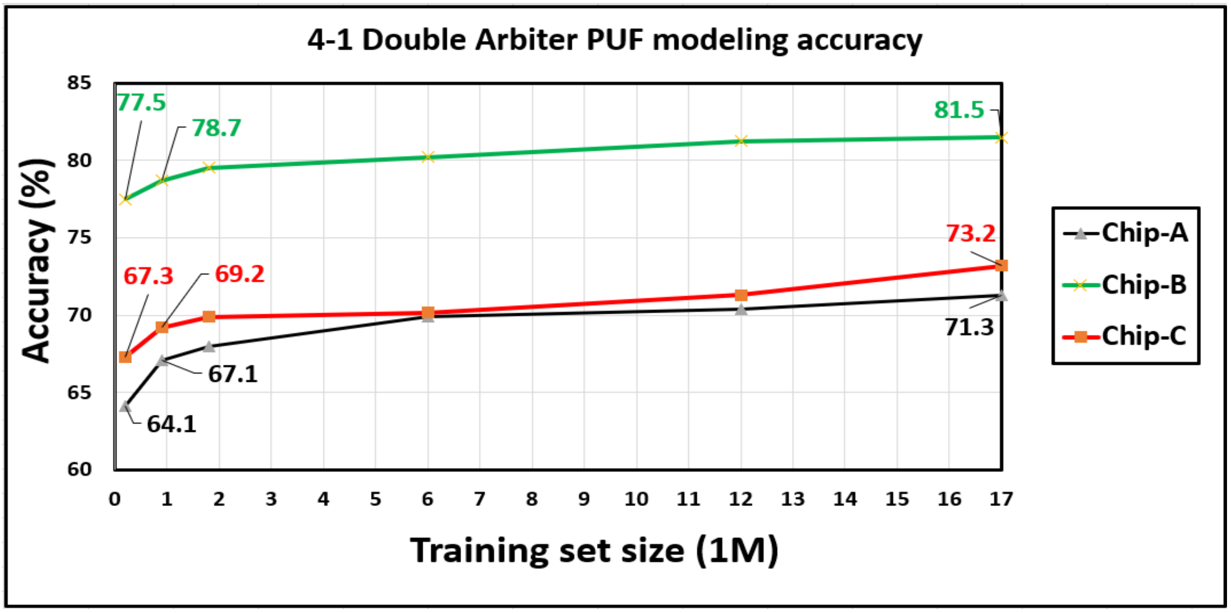


Figure 4.13: Deep Learning Attacks Results on 4-1 DAPUF

causes modeling accuracy to rise only 1%-3%. Hence, after a specific training set size, attackers should consider doing architectural modifications in their networks to strengthen the model accuracy instead of depending only on increasing the training set size. Although results show that the deep learning architectures used to attack 4-1 DAPUFs were not as successful as those used against 3-1 DAPUFs, but they still achieve better accuracy performance than the conventional ML and DL techniques used in [85] and [96] respectively. Their reported attack results show a modeling accuracy of 56% - 63% using LR, SVM and DL techniques. Hence, this work presents a step forward in fully modeling 4-1 DAPUFs and investigates more efficient techniques against PUFs with complex architectures.

4.2.4 Summary of DAPUF attacks

This section discussed the successful modeling attacks, which have been executed against different DAPUFs architectures. Obtained results show that 2-1 DAPUF can be modeled using LR with a linear decision boundary. Moreover, it achieves better accuracy (93.4%) using less CRPs and cheaper computing resources than ML & DL attacks reported in previous research [96]. Additionally, results proved that DL techniques can successfully model 3-1 DAPUF and achieved better accuracy than ML and DL attacks reported in literature (86% Vs. 68%) [85], [96]. Furthermore, experiments showed that the same LR technique that was successful against equivalent XOR PUFs failed to achieve the same performance with 3-1 DAPUF. Hence, using DL techniques is justified to achieve better modeling accuracy. Although obtained results of 4-1 DAPUF attacks did not show the same success but the DL network used achieved better accuracy than previous research (71.3%-81.5% Vs. 63%) [96]. Other DL architectures and techniques should be investigated to achieve better results on 4-1 DAPUFs.

One concern that might arise is the practicality of modeling attacks using DL techniques, which requires a huge set of CRPS for training. However, experiments were run on Mojo V3 boards running at 50MHZ frequency (clock period = 20ns), and the delay time of implemented DAPUFs ranges between 32-45ns. This means that reading one response with eased timing constraints can take less than 100ns, thus collecting 20M CRPs is a matter of several minutes. Furthermore, with the AI hype and the race to introduce more efficient hardware, it is easy and even cheaper to use many GPU cards in parallel using cloud computing services and train huge CRP sets in parallel.

Finally, these attacks draw attention to the fact that corresponding challenge and response bits should not be revealed no matter how complex the PUF architecture is, since attacks are possible using the available DL techniques and hardware can be successfully attacked and broken. This problem was addressed in the literature by proposing obfuscation schemes to hide the challenge response relationship as will be discussed in section 4.4. Furthermore, the ability to model multi-input XOR PUFs may enhance other hybrid attack schemes, which inject faults to disable part of the XOR inputs. Consequently, if the available modeling attacks can successfully model XOR PUFs with a larger number of inputs, then the fault injections needed will be minimized and equipment used will be cheaper (no need to accurately disable one input at a time). Hence, hiding the challenge response relationship should be the first priority for PUF designers before introducing new complex architectures. The next two sections will discuss more deep learning modeling attacks against PUF architectures without accurate mathematical models or using obfuscation techniques to hide the challenge-response relationship.

4.3 Modeling attacks on bi-stable ring PUFs and its variants

As mentioned in subsection 2.1.2.4.3, BR PUF was introduced in [78] and [15] as a new PUF architecture to merge between delay-based and memory-based approaches. The aim was to design a stronger PUF with more resistance against modeling attacks and large challenge space, so CRPs cannot be exhaustively read by the attackers. Its basic idea is that the output of any given inverter ring with an even number of inverters has only two possible stable states. This is similar to memory-based PUFs operation except that challenge bits are inserted to select which path to be used at every stage. One problem of BR-PUFs is that it takes a longer time to stabilize, which is an undesirable property of PUFs. Furthermore, BR-PUFs implementations on FPGAs showed an output bias problem as reported in [78]. As a result, other variations of BR-PUFs were proposed like TBR-PUF [78] and finally XOR BR-PUFs in [94]. The latter was proposed after successful modeling attacks were reported against BR-PUFs and TBR-PUFs using SVM and single-layer artificial neural network (ANN). XOR-BR PUFs showed significant resistance against modeling attacks and set an example that the approach of complicating the relationship between input challenges and responses can somehow countermeasure conventional ML modeling techniques used to break the security of previous architectures. However, it was shown in [26] that BR PUF families have a finite set of influential challenge bits and can be considered a Linear Threshold Function (LTF) similar to APUFs. Hence, an XOR version of BR PUFs can be modeled using a single-layer perceptron function as was reported in [27] against XOR APUFs. In this subsection, we show that XOR BR and XOR TBR PUFs cannot be modeled using a single layer NN and we needed to use deeper architecture along with a simplified mathematical model to fully break their security.

Our motivation to attack the XOR BR PUF family is based on the current status of PUFs design efforts, which part of it focuses on introducing new architectures solely depending on more complex challenge-response relationships or building an architecture that no accurate mathematical model can be derived for. Furthermore, measuring the architectural enhancement by how resistant these PUFs are to conventional ML modeling attacks. However, the development in deep learning (DL) techniques and the hardware running it with acceptable timing performance puts more pressure and challenges on the PUF architectures resistant to conventional modeling techniques. The contributions of BR PUF DL attacks are as follows:

1. Showing that XOR BR and XOR TBR PUFs cannot be modeled using single NNs and justifying the necessity to use deeper networks.

2. Applying deep learning modeling techniques to attack 4-input, 5-input,6-input XOR BR PUF, XOR TBR PUF, and an obfuscated version of XOR BR PUF and successfully breaking its security with different stage sizes (e.g. 64, 128, 256) and achieving a remarkable modeling accuracy $> 99\%$.

3. Taking the first step to connect between DL theory and our empirical results to understand why DL attacks work successfully and based on that we introduced a new obfuscated version of XOR BR PUF as a countermeasure (more details in chapter 5).

4. Providing detailed analysis on the scalability of DL modeling techniques in terms of layers number and hidden neurons needed with respect to the PUF architecture complexity and number of PUF stages.

4.3.1 Modeling attacks of BR PUFs family

4.3.1.1 Bistable ring PUFs

Since the BR-PUF architecture is operating like memory-based PUFs, it was expected that no mathematical model could be built for such architecture. However, there exist several reported machine learning attacks against BR PUFs by either using a simplified mathematical model [94][78] or without model [26]. Furthermore, it was found that its responses were not uniform and are biased [94]. Hence, twisted BR-PUF (TBR PUF) was introduced by Schuster D. and Hesselbarth R. after successfully breaking the security of BR-PUF using a single layer NN and a simplified mathematical model (i.e. replacing every '0' challenge bit with '-1') [78]. This new variant of BR PUF makes all inverters involved in the closed loop and input challenge bits are responsible for determining the positions of inverters inside the ring (odd or even).

Although this architectural modification showed more resistance against ANN attacks and more uniform responses, in the same paper, it was noted that NNs were learning the correlation between challenge bits and responses and there might exist more influential bits that helped in modeling BR PUFs. Another study built on this note and showed that TBR PUFs could be modeled and cloned. Gangi F. et al [26] proposed a new attack against BR and TBR PUFs that does not require deriving a mathematical model of the BR PUF family. The main idea is to exploit the challenge bits with higher influence on PUF response (influential challenge bits) to construct a machine learning based boosted model that can predict the PUF outcome with high probability. Experiments conducted using 30K CRPs collected from 64 stages BR and TBR PUF implementations on Altera Cyclone IV FPGAs. Adaptive boost algorithm [25] was used to create the boosted classifier built over the initial weak learners, which depends on single influential bits. Obtained results showed that boosting technique could successfully model both PUFs up to 99% prediction accuracy using 50 boosting iterations. Furthermore, they suggested that BR and TBR

PUFs have a small set of influential bits and their polynomial threshold function (PTF) can be approximated by an LTF.

4.3.1.2 XOR Bistable ring PUFs

In [94], Xu X. et al could build a simplified mathematical model to attack BR and TBR PUFs using SVM modeling technique. Instead of deriving an accurate non-linear model of PUF delays, A simplified additive model was adopted to represent the difference between the pull-up and pull-down strength of every inverter. Hence, the PUF response can be re-written as the summation of all stages' strength difference and therefore, SVM works by learning the weights assigned to every inverter strength difference. Equations 4.13 to 4.15 explain the model parameters, where 't_i' and 'b_i' are the pull-up and pull-down strength difference for the upper and bottom NOR gate at the i^{th} stage. Hence, an even stage will contribute to a positive PUF response with strength t_i or b_i depending on the challenge bit value and odd stages will contribute with strength $-t_i$ or $-b_i$. A generalization of these terms can be used to represent the odd and even stages contribution in this form '$-1^i t_i$' and '$-1^i b_i$'. In equation 4.14, Xu X. et al defined two terms 'α_i' and 'β_i' to facilitate the writing of PUF response summation equation with respect to input challenge bits. Hence, PUF response can be represented as a linear summation as shown in equation 4.15, where 'K' is the number of PUF stages and 'C_i' $\in -1, 1$ is the challenge bit at this stage (note that 0 value is interpreted as -1 to select the bottom NOR gate). Furthermore, the term 'α_i' can be discarded because it yields the same value for all CRPs training samples.

$$\Delta Strength^i_{upper} = -1^i t_i, \;\; \Delta Strength^i_{buttom} = -1^i b_i \tag{4.13}$$

$$\begin{gathered} \alpha_i + \beta_i = -1^i t_i \quad , \quad \alpha_i - \beta_i = -1^i b_i \text{ , hence} \\ \alpha_i = -1^i(\frac{t_i + b_i}{2}) \quad , \quad \beta_i = -1^i(\frac{t_i - b_i}{2}) \end{gathered} \tag{4.14}$$

$$T_{response} = Sign(\sum_{i=0}^{K}(\alpha_i + C_i\beta_i)) \tag{4.15}$$

The experimental results showed that BR and TBR PUFs could be successfully attacked using SVM with modeling accuracy $> 95\%$. Hence, XOR BR PUF was introduced to countermeasure this attack and the results showed that the SVM modeling technique with polynomial kernel was not successful in breaking architectures with XOR input > 3 as shown in Table 4.11.

Table 4.11: Reported results of SVM modeling attack on XOR BR PUF [94].

No. of XOR inputs	No. of Stages	No. of training CRPs	Modeling Accuracy
3	32	1200	$> 95\%$
3	64	7200	$> 95\%$
3	128, 256	N/A	50.1%
4	32,64,128,256	N/A	50.1%

4.3.1.3 Motivation for deep learning attacks

Gangi F. et al [27] showed that XORed LTFs (e.g. APUFs) with the number of XOR inputs $<$ ln(number of PUF stages) can be learned using single-layer perceptron function in polynomial time. Consequently, if BR PUFs can be approximated by an LTF because of the finite set of influential bits as discussed earlier, then it is expected that using a single layer NN can model XOR BR PUFs. However, we conducted an analysis on the implemented XOR BR and TBR PUF instances using Linear Discriminant Analysis (LDA) to confirm if both classes representing PUF response are linearly separable or not. LDA is a supervised linear transformation technique used to reduce features dimensionality by computing the linear discriminants or the directions of the axes at which, the separation between multiple classes is maximized [37]. As a result, all XOR BR and TBR PUF instances with different sizes (64, 128, 256) showed similar behavior to the example in Figure 4.14. It shows the density function of 1M data samples representing both PUF response classes '0' and '1'. It is clear that both curves are overlapping and hence, they are not linearly separable, and a single layer perceptron algorithm cannot model this type of architectures. One might attribute the different behavior of XOR BR PUFs compared to XOR APUFs to the fact that we could derive an accurate additive linear model for the latter. On the other hand, the mathematical model of XOR BR PUFs is a simplified one. It is shifting from modeling delay difference to a more abstract concept of modeling the strength difference of every stage as discussed earlier. Furthermore, this also justifies why SVM with polynomial kernel could not break XOR BR PUF security while it was reported in previous literature that similar XOR LTFs (i.e. APUFs) could be broken using logistic regression technique (LR)[75][76].

Hence, there was a motivation behind this work to explore the ability of deep learning modeling techniques to break the security of XOR BR PUFs. Furthermore, experiments were executed to attack XOR TBR PUFs because TBR PUFs showed more resistance to ANN modeling attacks compared to BR PUFs as mentioned earlier. Therefore, XOR TBR PUF was expected to be harder to break and introduces an extra challenge to confirm the

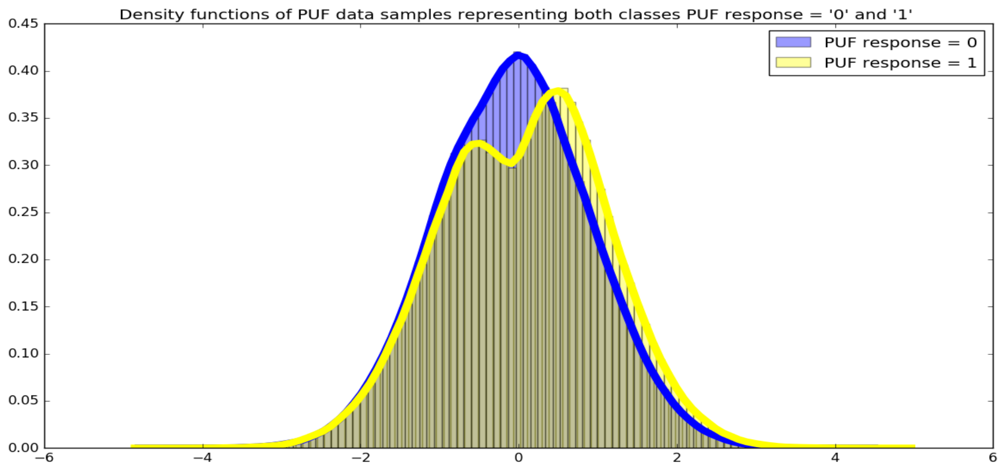


Figure 4.14: density function of LDA feature analysis for 128 stage 4-input XOR BR PUF

obtained results from attacking XOR BR PUFs.

4.3.2 BR PUF implementation on FPGA and statistical properties

In this subsection, we show how the PUFs under attack were implemented and its statistical properties. The implementation techniques used are based on the work of M. A. Elmohr in [21]

4.3.2.1 Overall system architecture

The Mojo V3 Board which features a Spartan-6 XC6SLX9 Xilinx FPGA is used alongside an ATmega32U4 AVR Microcontroller. Figure 4.15 shows the system architecture in which, PUF instances are implemented on the FPGA side and a Finite State Machine (FSM) and a UART module to control the communication between FPGA and host machine for receiving challenges and sending responses. The UART module on top of the FPGA is connected to the UART of the AVR Microcontroller, which in return transfers the data through the USB port of the Mojo board to and from an external PC.

Challenges are generated using a Galois Linear-Feedback Shift Register (Galois LFSR) pseudo-random number generator by a script on the PC side and sent to the PUF through

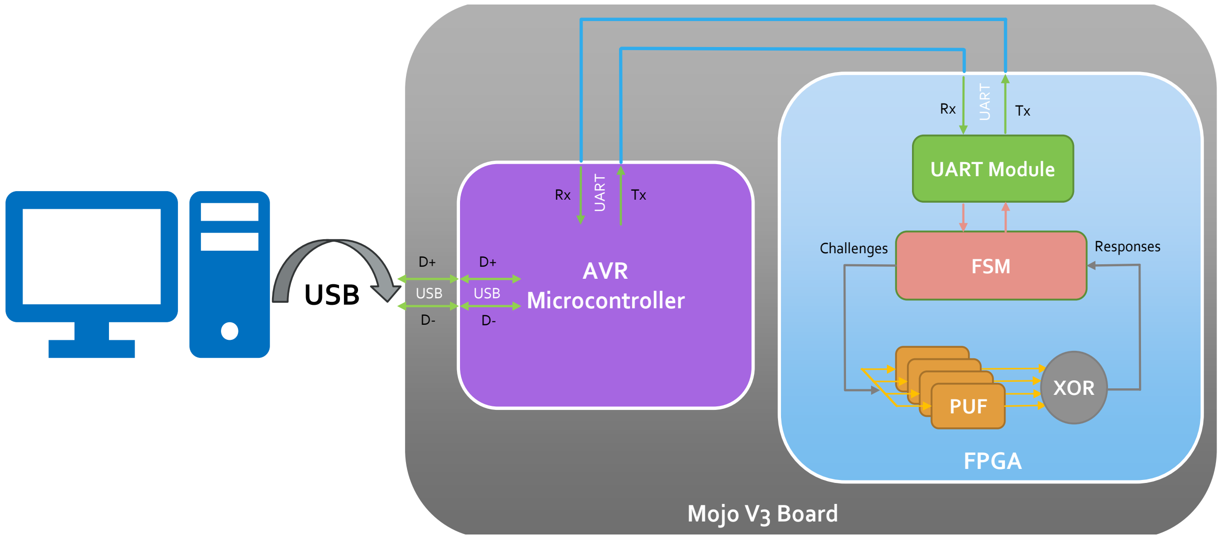
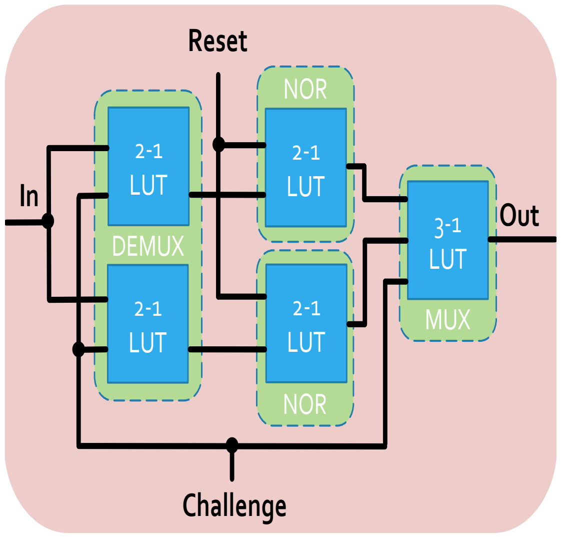


Figure 4.15: PUF Ecosystem [21]

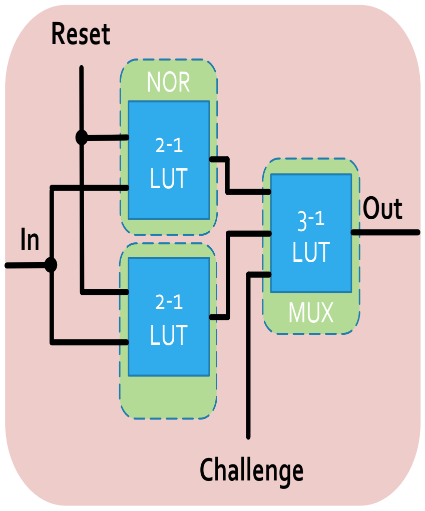
serial communication to the FPGA byte by byte. A predetermined constant time is configured before which, the PUF must converge to a stable state and its response is captured. In our implementation, all the stages' outputs are derived to confirm that all stages have the same capacitive load, otherwise one stage would have a different load than others, which might bias the PUF as was noted in [95]. Another advantage of deriving all stages' outputs is to distinguish between converged and non-converged responses. This is done by ORing response bytes together forming one byte output that is sent to the PC side. For a converged ring, that byte should be either '10101010' (0xAA) representing '1' or '01010101' (0x55) representing '0', other than that, it indicates a non-converged ring, which won't be added to the CRPs database.

4.3.2.2 BR PUF implementation technique

Every BR-PUF stage as introduced in [16] should have one MUX, one DEMUX and two inverting elements such as NOR gates. An FPGA implementation would result in five Look Up Tables (LUTs) as in Figure 4.16a similar to what was reported in [95]. however, an optimization is introduced in our implementation by removing the DEMUX. The BR-PUF functionality is not affected because the MUX chooses between the output of either NOR gates, which is the important issue for the BR-PUF functionality. Hence, it does not make a difference whether supplying the input to only one NOR gate as in the original design or to both NOR gates as in our optimized version. This optimization technique reduces the number of LUTs to be three as in Figure 4.16b.



(a)

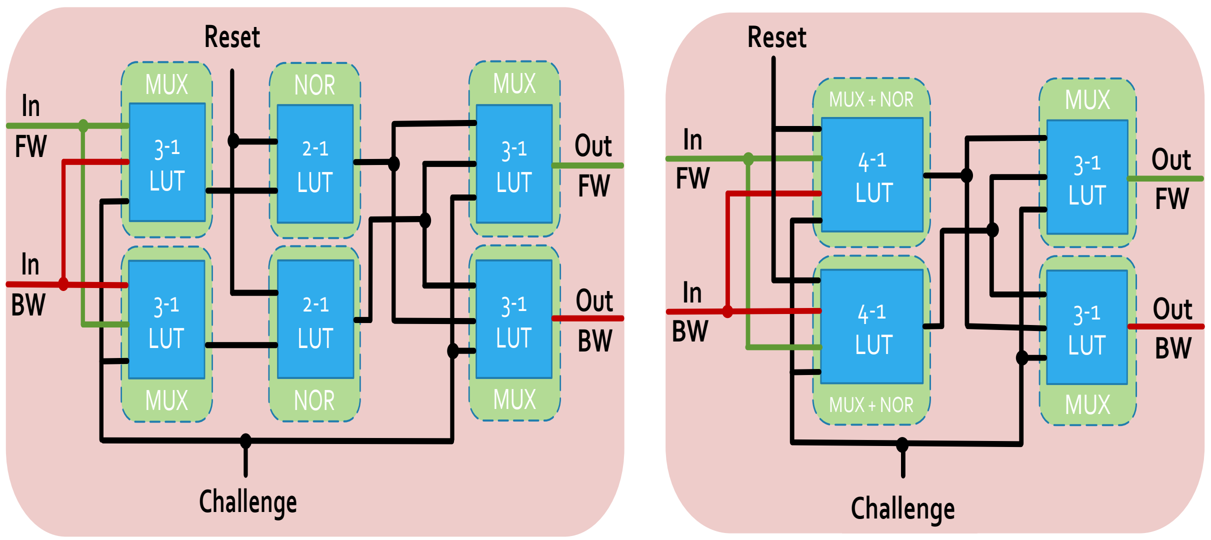


(b)

Figure 4.16: BR PUF Optmization Layout on FPGA [21]

Similarly, the TBR-PUF design of every single stage in [78] and [26], uses 2 MUXes feeding the 2 NOR gates and two other MUXes choosing between the outputs of the two NOR gates. hence, a direct implementation transforms each component of the PUF stage into a separate LUT as in Figure 4.17a resulting in a total of 6 LUTs per TBR-PUF stage. However, in our implementation, each NOR gate is merged with its preceding MUX as shown in Figure 4.17b resulting in a total of only 4 LUTs per TBR-PUF stage. This optimization does not affect the functionality of the TBR-PUF as it maintains two inverting elements and two different paths. Furthermore, the merged LUT has the same logic of the two separate LUTs combined.

The main reason behind these optimizations is that both PUFs with 128-bit and 256-bit challenge wouldn't fit on the FPGA due to the limited number of LUTs. Moreover, Preliminary experiments were conducted over the non-optimized implementations for 64-bit BR and TBR PUFs to empirically confirm that the optimizations presented for both BR and TBR PUFs do not affect their security. The obtained results showed that non-optimized architectures performed similarly against both SVM and Deep learning techniques. Hence, these optimizations do not change the PUFs' architectures and empirically do not affect their security.



(a) (b)

Figure 4.17: TBR PUF Optmization Layout on FPGA [21]

4.3.2.3 PUF characteristics

As mentioned in 2.2 reliability, unpredictability, and uniqueness are three main statistical properties that we use to measure the quality of implemented PUFs [14]. In this subsection, we give some details on how these three properties are measured and provide the actual characteristics for the implemented PUFs. These metrics are PUF noise, PUF bias, and Inter-Chip hamming distance. We also considered another important characteristic which is individual challenge bits influence on the PUF response.

1. PUF Noise: A reliable PUF would give a consistent response to a certain challenge per each chip, however, in reality, a PUF might give inconsistent responses for the same challenge. To measure noise, we apply the same challenge to the same chip for many iterations, take the majority vote to determine the supposedly right response and repeat that for all challenges. Thus we can calculate the noise as

$$N = \frac{\sum \#\ wrong\ responses}{\#\ iterations \times \#\ challenges}$$

A reliable PUF would have an ideal noise of 0.

2. PUF Bias: PUF bias represents the tendency of the PUF to respond with 0 or 1 more likely to different challenges. Bias can be calculated as

$$B = \frac{\#\ responses\ of\ '1'}{\#\ challenges}$$

An unpredictable PUF would have an ideal bias of 0.5.

3. Inter-Chip Hamming Distance: Different chips should give different responses to the same challenge. Inter-chip hamming distance represents how many responses were dissimilar for the same challenge on different chips. The normalized hamming distance between two different chips would be calculated as

$$NHD = \frac{\#\ dissimilar\ responses}{\#\ challenges}$$

A unique PUF would have an ideal normalized inter-chip hamming distance of 0.5.

4. Individual Challenge Bits Influence: Challenge bits should ideally contribute equally to the resulted response, not only some of the challenge bits. For each challenge bit, its influence is calculated as

$$Infl(i,0) = \frac{\#\ responses\ of\ '1'}{\#\ challenges\ with\ ith\ bit = 0}$$

$$Infl(i,1) = \frac{\#\ responses\ of\ '1'}{\#\ challenges\ with\ ithbit = 1}$$

A good PUF would have an ideal influence of each bit as 0.5.

To obtain the actual characteristics of the implemented PUFs detailed in Table 4.12, we used three typical Mojo boards, loaded the same PUF design on all of them and applied 1 Million different challenges each for three iterations. All experiments were conducted in room temperature.

An important note is that as we had the ability to distinguish between converged and non-converged responses in our implementation, we excluded the non-converged responses from characteristics calculations. Moreover, training and testing sets contained only the converged responses after the majority vote, thus eliminating PUFs noise for the neural networks. It is important to note that all results presented in Table 4.12 are for the XORed PUFs treated as a black box, not for individual PUFs. Also, the reported characteristics are averages over the three chips, except for the bias, since averages will not be sufficient.

Table 4.12: Implemented 4-input XOR BR & TBR PUFs characteristics

Characteristic	PUF Size & Type					
	BR PUF			TBR PUF		
	64	128	256	64	128	256
Non-converged Chip 1 (%)	19	18	14	28	22	32
Non-converged Chip 2 (%)	18	17	15	24	33	33
Non-converged Chip 3 (%)	21	18	15	27	26	27
Bias Chip 1 (%)	48	50	47	51	47	53
Bias Chip 2 (%)	49	49	54	54	52	47
Bias Chip 3 (%)	47	48	53	53	53	49
Noise Chip 1 (%)	1	2	1	3	3	3
Noise Chip 2 (%)	1	2	1	3	3	3
Noise Chip 3 (%)	1	2	1	3	3	3
Hamming Distance Chips 1&2 (%)	55	56	47	52	49	54
Hamming Distance Chips 1&3 (%)	48	51	50	55	46	55
Hamming Distance Chips 2&3 (%)	43	52	59	44	47	46
Hamming Distance Average (%)	49	53	52	50	47	52
Max Infl. Chip 1 (%)	46	51	45	55	44	53
Max Infl. Chip 2 (%)	45	51	58	59	56	45
Max Infl. Chip 3 (%)	43	47	56	59	55	46

As shown in Table 4.12, the implemented PUFs have near-ideal characteristics for bias and inter-chip hamming distance. Even noise is negligible because we eliminated the non-converged responses. The non-converged responses ranged from 14% to 31% corresponding to convergence rates between 86% and 69%, which is one of the drawbacks of BR and TBR PUFs. The reported cycles spent waiting for convergence before capturing responses (the evaluation time) are the minimum cycles needed to achieve the corresponding convergence rates, (waiting for more time would not lead to any significant improvement in the convergence rates).

Also as shown there are no individual influential bits. The maximum influence a challenge bit can get is 59% which is not a huge influence compared to the ideal influence of 50%.

Table 4.13 shows the characteristics of 5-input and 6-input 64-stage XOR BR PUFs. We calculated the influence among XOR inputs to prove that the implemented PUFs responses

Table 4.13: Implemented 5-input & 6-input XOR BR PUFs characteristics

Characteristic	PUF Size & Type	
	64-stage 5-input XOR BR PUF	64-stage 6-input XOR BR PUF
Non-converged Chip 1 (%)	12.7	16.1
Non-converged Chip 2 (%)	13.03	16.8
Non-converged Chip 3 (%)	14.4	17.65
Bias Chip 1 (%)	49.8	50.2
Bias Chip 2 (%)	49.6	49.88
Bias Chip 3 (%)	50.2	50.02
Noise Chip 1 (%)	1.9	2.25
Noise Chip 2 (%)	1.9	2.3
Noise Chip 3 (%)	2.05	2.37
Hamming Distance Chips 1&2 (%)	58.22	60.18
Hamming Distance Chips 1&3 (%)	54.78	57.81
Hamming Distance Chips 2&3 (%)	54.64	57.52
Hamming Distance Average (%)	55.88	53
Max Influence between two XOR inputs on Chip 1 (%)	55.4	58.3
Max Influence between two XOR inputs on Chip 2 (%)	58.1	62.09
Max Influence between two XOR inputs on Chip 3 (%)	58.86	62.3

are the outputs of true 5-input and 6-input XOR functions. As shown in the characteristics results, maximum influence between two XOR inputs does not exceed 62.3% compared to an ideal 50%.

4.3.3 Deep Learning Network Architecture and Experimental Setup

4.3.3.1 Deep neural network architecture

Figure 4.18 shows the deep neural network architecture used in the modeling attacks against XOR BR PUF and XOR TBR PUF. All layers of the network are fully connected, which means that every hidden neuron in layer n is connected to all neurons in layer n+1. Every connection represents a weight that reflects the neuron effect from the preceding layer

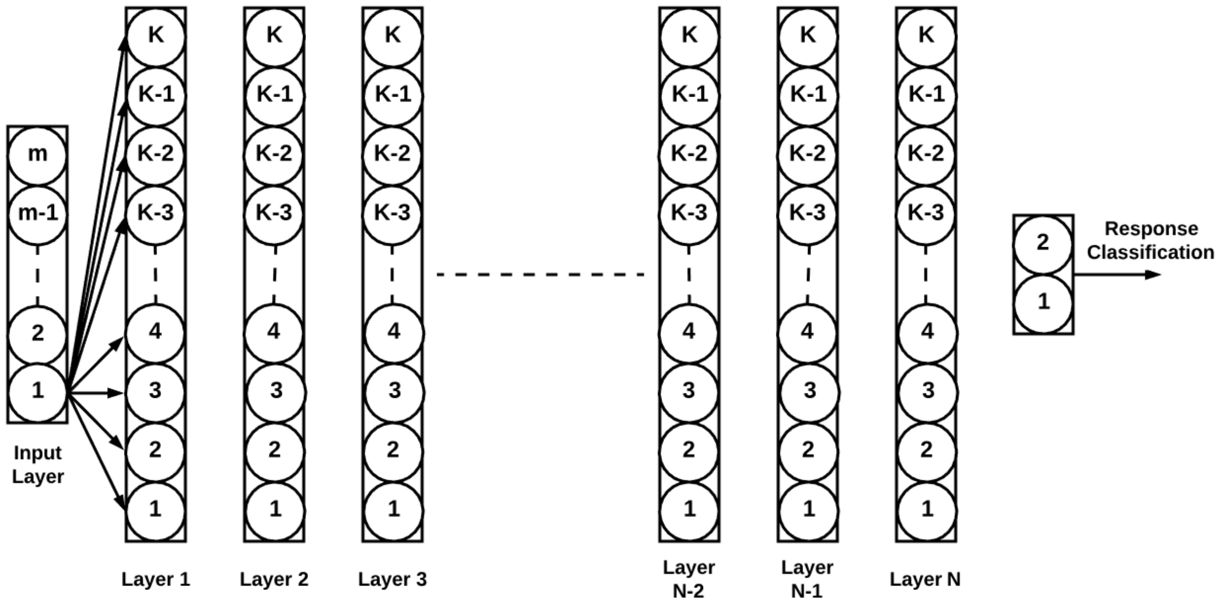


Figure 4.18: The DNN network architecture used in modeling attacks. All layers are fully connected layers.

on the output produced by the neuron in the next layer. In the graph, 'm' represents the number of input features which is one of the values 64, 128, and 256 depending on the PUF number of stages. The number of hidden neurons in every layer is represented by 'K' and the network depth is denoted by 'N', which corresponds to the number of fully connected layers in the network. Although the values of K and N were varied in the experiments to study the DNN scalability, the network used to report the modeling accuracies results is setting N = 12 and K = 2000. Finally, a dropout layer was placed between the last fully connected layer and the output to help the network generalization and avoid the problem of over-fitting.

4.3.3.2 Convolutional Layer Vs. Fully Connected Layer

In DNN, convolutional layers are used for extracting desired features from input data. It applies a convolution process using a set of filters on input data to detect important features related to the task as shown in Figure 4.19. Hence, the shallow layers are usually convolutional layers to extract primitive features and reduce input size then deeper layers

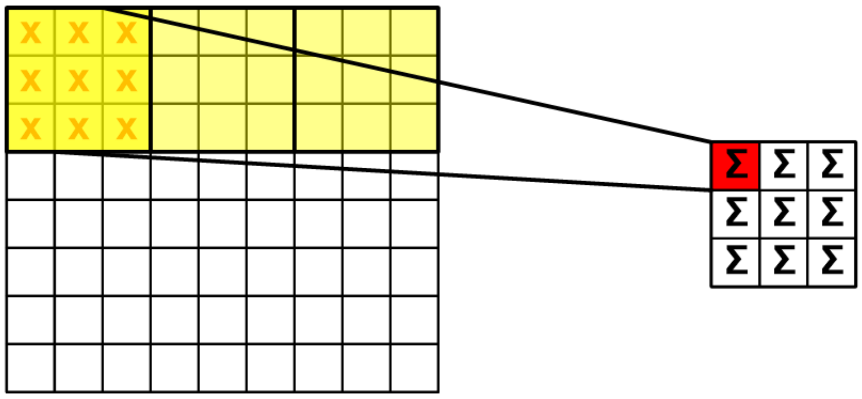


Figure 4.19: The Convolutional Layer. Small kernels are convoluted over the input image and every convolutional window produces one output

are fully connected. This approach works in object recognition and classification tasks because of two main reasons. Firstly, features in input images have a locality property therefore, it is more efficient to apply smaller feature sized windows to find pixels correlations. Secondly, the convolutional layers are less computation-intensive. However, the locality property does not exist in the context of modeling complex PUF architectures. There might be correlations between different PUF stages with distant positions or even among stages from different PUFs (In case of having XOR PUF architecture like XOR BR PUF). Furthermore, these correlations will change from one PUF instance to another, which makes using convolutional layers not practical. Hence, using fully connected layers in the network is like the brute-force technique to extract features that help in modeling PUF response.

4.3.3.3 Activation function: Tanh Vs. RELU

Activation functions are applied to every layer output to determine the effect of every neuron on the next layers. Although there are many types of functions, Tanh and RELU are the most used transfer function because of their good performance. RELU restricts the values to be between 0 and +ve, while Tanh ranges from -1 to 1 as shown in Figure 4.20. Tanh is better in the PUF modeling context because it preserves a negative value for negative inputs, which is important to differentiate between stages contributing to or against a positive response. Furthermore, the PUF output in the mathematical model outputs either 1 or -1 (as shown earlier in equation 4.15), which improved our results when using the tanh activation function in the shallower layers and the last layer of the DNN architecture used.

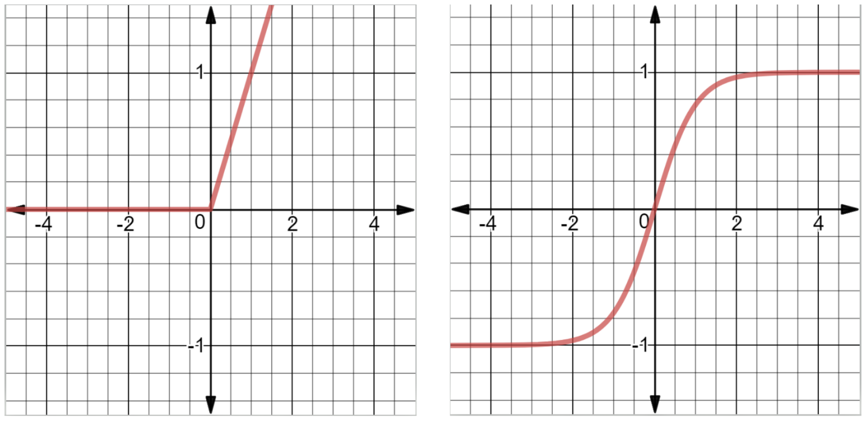


Figure 4.20: Relu and Tanh

4.3.3.4 DNN training parameters

In this subsection, we discuss the DNN parameters used in our training processes. Firstly we used the logistic function in our classification layer. it outputs a probability distribution of the problem binary classes, which in our case a two-class PUF response. The logistic function (σ) defined in equation 4.16 maps the input x to an output between 0 and 1. This value represents the probability that problem output will belong to a specific class given the input x value as shown in equation 4.17. The cross-entropy is used as the loss function, which measures the likelihood of a given set of parameters θ of the model can result in a prediction of the correct class of each input sample. The network tries to maximize this likelihood function by using an adaptive moment estimation (Adam) optimization algorithm proposed by Kingma, D. and Ba, J. [46]. As was mentioned in subsection 4.2, experiments showed that it converges faster and introduces better accuracy results than the normal gradient descent algorithm. The learning rate is usually between 0.0001 and 0.00001 for the training accuracy to be stable. For every run we do 1M iterations and a stopping condition if training accuracy is not changing to the 4th fractional digit for 5 consecutive times. Furthermore, if there is no conversion after 1M iterations, we re-run it again but for almost every training process convergence would occur within the first 100K iterations. We evaluate our models using the accuracy metrics, which measures how many correct prediction cases out of the whole test set.

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{4.16}$$

$$P(t=1|x) = \sigma(x) = \frac{1}{1+e^{-x}}, \quad P(t=0|x) = 1-\sigma(x) = \frac{1}{1+e^{-x}} \tag{4.17}$$

4.3.3.5 Modeling attacks using SVM with polynomial kernel

Experiments will include attacks using SVM to test the resistance of implemented XOR BR and XOR TBR PUFs against conventional ML attacks and to further justify the need for using DL techniques. A script is written using python and scikit-learn library to provide SVM modeling functionality. Note that a grid search was conducted first to tune the SVM model parameters. Moreover, a polynomial kernel with degree four (equivalent to the number of XOR inputs) is used as was done by Xu et al in [94].

4.3.3.6 Hardware and software experimental setup

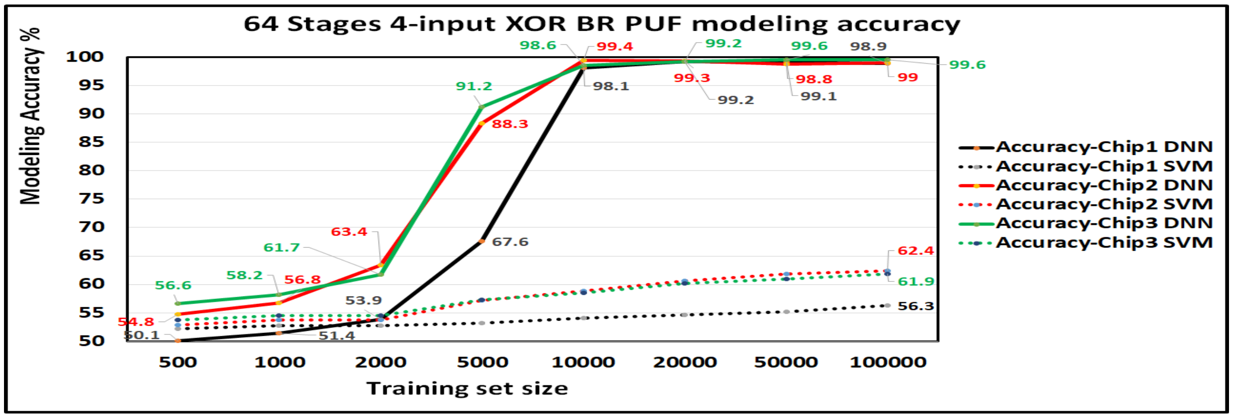
Experiments are conducted using three Mojo V3 boards, each containing a Spartan 6 XC6SLX9 FPGA [5](45nm process technology). All CRPs are generated using a linear-feedback shift register (LFSR). SVM attacks were executed on Intel 8th Gen I7-8250 CPU with 16GB RAM of memory. DL attacks were executed using Nvidia GeForce GTX 1080 Ti GPU card with 11GB RAM. PUFs were implemented using the Xilinx ISE Design Suite 14.7, Xilinx PlanAhead 14.7, and Xilinx FPGA editor. Additionally, the Tensorflow platform was used to develop the network architectures, training, and evaluation tasks. Finally, scikit-learn library was used to implement SVM.

4.3.4 DL Modeling attack results

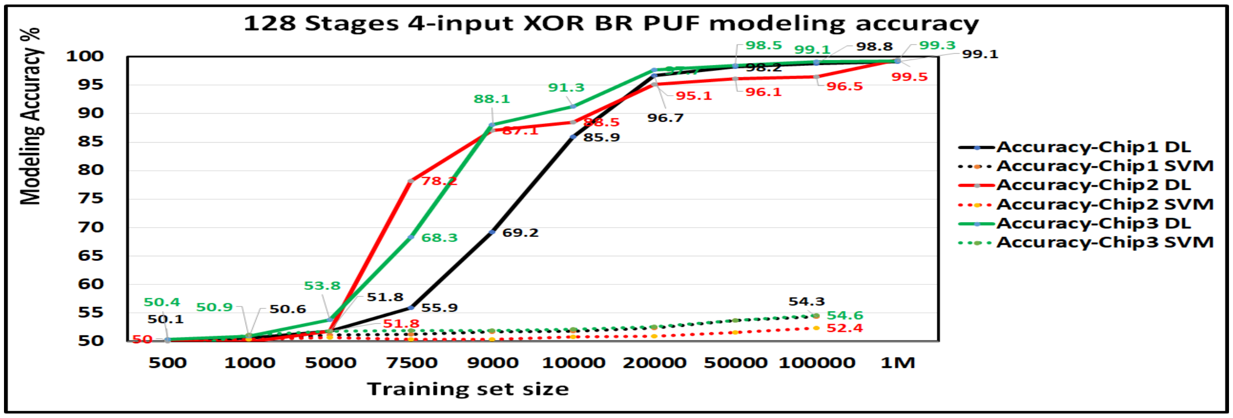
4.3.4.1 DL attack results on 4-input XOR BR and TBR PUFs

As mentioned earlier, modeling attacks were launched on 24 instances of 4, 5, and 6-input XOR BR & TBR PUF with varying stage size (64, 128, 256), which all were implemented on three mojo V3 boards. The DNN used for the attack had 12 fully connected layers and 2000 hidden neurons in every layer. The training set size was varied between 5K to 1M CRPs. Furthermore, The same PUF instances were attacked using SVM with a polynomial kernel of degree four to account for the 4-input XOR function [94]. Finally, the test set size was 100K CRPs for all experiments and CRPs were randomly generated as discussed in subsection 4.3.2. Figure 4.21 and Table 4.15 show the modeling accuracy results for all 4-input XOR BR PUF instances implemented on chip-1 using both approaches (DL, SVM). Note that similar results were obtained for the instances implemented on Chips 2 and 3.

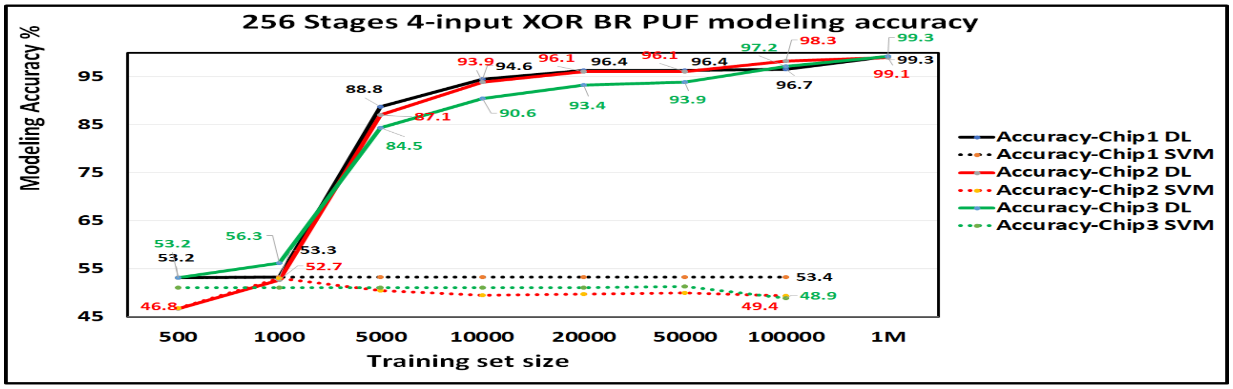
It is shown that DL modeling was successful in breaking the security of all XOR BR PUF instances. Furthermore, training using 100K CRPs was enough to reach the accuracy



(a) DNN and SVM Modeling Accuracy for 64 Stage 4-input XOR BR PUF



(b) DNN and SVM Modeling Accuracy for 128 Stage 4-input XOR BR PUF



(c) DNN and SVM Modeling Accuracy for 256 Stage 4-input XOR BR PUF

Figure 4.21: Modeling Attacks Results Against 4-input XOR BR PUF

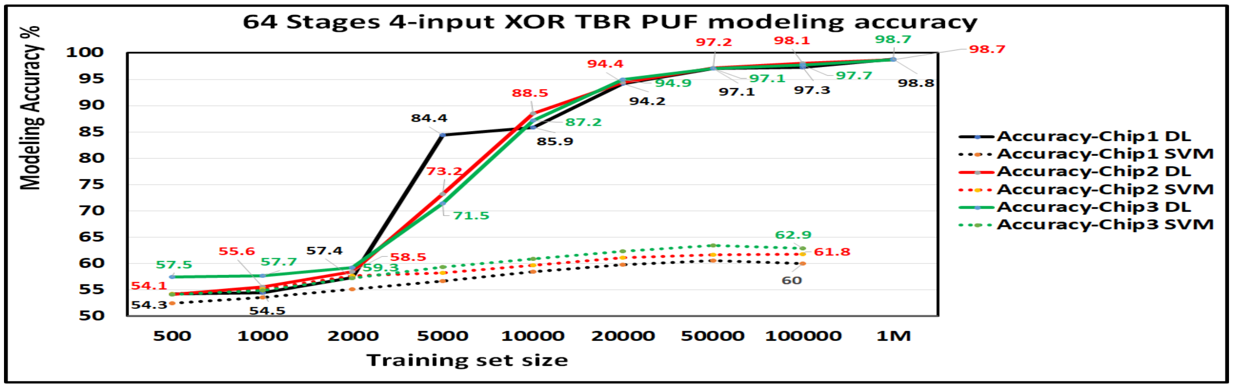
Table 4.15: 4-input XOR BR PUF modeling accuracy(%) on Chip-1.

	64-Bit		128-Bit		256-Bit	
Train-size	DL	SVM	DL	SVM	DL	SVM
5K	67.6%	53.2%	51.8%	51.1%	88.8%	53.3%
10K	98.1%	54.1%	85.9%	51.8%	94.6%	53.3%
20K	99.2%	54.6%	96.7%	52.4%	96.4%	53.3%
50K	99.1%	55.2%	98.2%	53.7%	96.4%	53.4%
100K	98.1%	56.3%	98.8%	54.3%	96.7%	53.4%
1M	99.5%	N/A	99.1%	N/A	99.3%	N/A

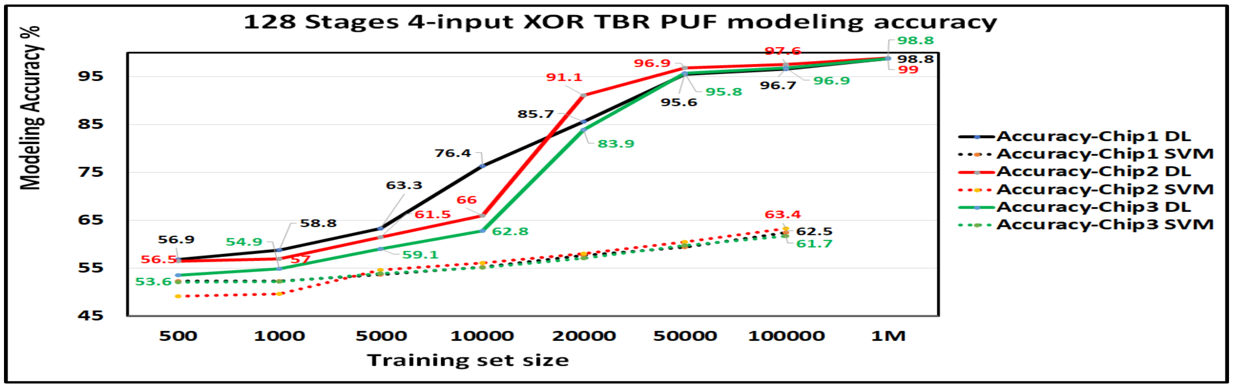
boundary of 99% in the case of 64 stages and 1M CRPs to reach this accuracy value for 128 and 256 stages. On the other hand, the SVM technique failed to successfully model the PUFs and its accuracy matched what was reported in previous literature for the same training size [94]. The maximum modeling accuracy that could be reached by SVM was 62.4% for the 64 stages PUF implemented on chip-2 using 100K CRPs for training. Moreover, SVM performance got worse when attempting to model PUFS with bigger stage sizes, while DNN did not seem to be affected by that. Instead, The DL modeling technique was powerful enough to get an accuracy $> 95\%$ using 20K CRPs for training. This is relatively a small dataset size and generally, deep networks need more samples to train on. In contrast, results show that SVM failed to get higher than the 62% accuracy using up to 100K CRPs.

Table 4.17: 4-input XOR TBR PUF modeling accuracy(%) on Chip-1.

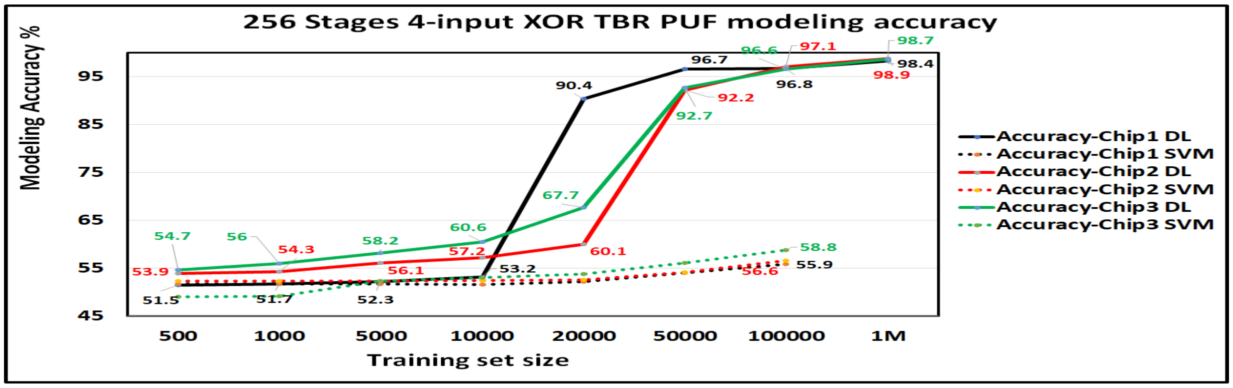
	64-Bit		128-Bit		256-Bit	
Train-size	DL	SVM	DL	SVM	DL	SVM
5K	84.4%	56.7%	63.3%	53.7%	71.5%	59.4%
10K	85.9%	58.5%	76.4%	55.3%	87.2%	60.9%
20K	94.2%	59.8%	85.7%	57.7%	94.9%	62.4%
50K	97.1%	60.6%	95.6%	59.4%	97.1%	63.5%
100K	97.3%	60%	96.7%	62.5%	97.7%	62.9%
1M	98.8%	N/A	98.8%	N/A	98.7%	N/A



(a) DNN and SVM Modeling Accuracy for 64 Stage 4-input XOR TBR PUF



(b) DNN and SVM Modeling Accuracy for 128 Stage 4-input XOR TBR PUF



(c) DNN and SVM Modeling Accuracy for 256 Stage 4-input XOR TBR PUF

Figure 4.22: Modeling Attacks Results Against 4-input XOR TBR PUF

Figure 4.22 and Table 4.17 show the modeling attacks results against 4-input XOR TBR PUF on Chip-1 with varying stage size (64, 128, 256). Similarly, DL modeling techniques could break the security of the XOR TBR PUF, while SVM showed nearly the same performance as in XOR BR PUFs. However, the models could achieve accuracy $> 95\%$ using more CRPs for training than XOR BR PUFs (50K CRPs). Furthermore, the maximum accuracy achieved was slightly less than or equal to 99%. This is expected because TBR PUFs involve all inverters in its operation, hence a slightly more complicated architecture than BR PUFs with the same number of challenge bits. In addition, all instances implemented on chips 2 and 3 showed similar behavior as the PUFs realized on chip 1.

4.3.4.2 DL attack results on 5-input and 6-input XOR BR PUFs

Figures 4.23, 4.24 and Tables 4.18, 4.19 show the modeling attacks results against 5-input and 6-input 64 stage XOR BR PUFs. As was mentioned in 4.3.2.3, the correlation among the XOR inputs were measured to confirm that no BR PUFs are correlated with each other. Hence, the PUF instances under attack represent real 5-input and 6-input XOR function. The results show that DL modeling techniques could break the security of both PUFs and reach an accuracy of $\sim 98.5\%$. However, the number of CRPs needed for the trained model to achieve at least 98% accuracy has increased to be 50K CRPS for the 5-input XOR instances and 100K - 150K CRPs for the 6-input XOR instances. This shows that increasing the number of independent XOR inputs produces similar if not a harder PUF than increasing the number of stages. Moreover, an important outcome of these results is the ability of DL techniques to break a PUF architecture that has the number of XOR inputs $>$ ln(number of stages) in a polynomial time. This contradicts with the upper bound set by Ganji F. et al in [27], which triggers open questions about the upper bounds of deep learning techniques.

4.3.4.3 DNN Scalability Vs. increasing PUF stages and XOR input

The DNN architecture is built to mimic the way PUFs operate to facilitate the learning process therefore, the hidden neurons in one layer can be considered as the stages of one PUF. Similarly, the number of layers can be used to reflect the number of XOR inputs. Consequently, a question arises about the network parameters and whether they should be the same as the number of PUF stages and XOR inputs or not. Note that increasing the network size can enhance or complicate the task for the network to learn the non-linear relationships among all stages. Accordingly, more experiments were executed to study the

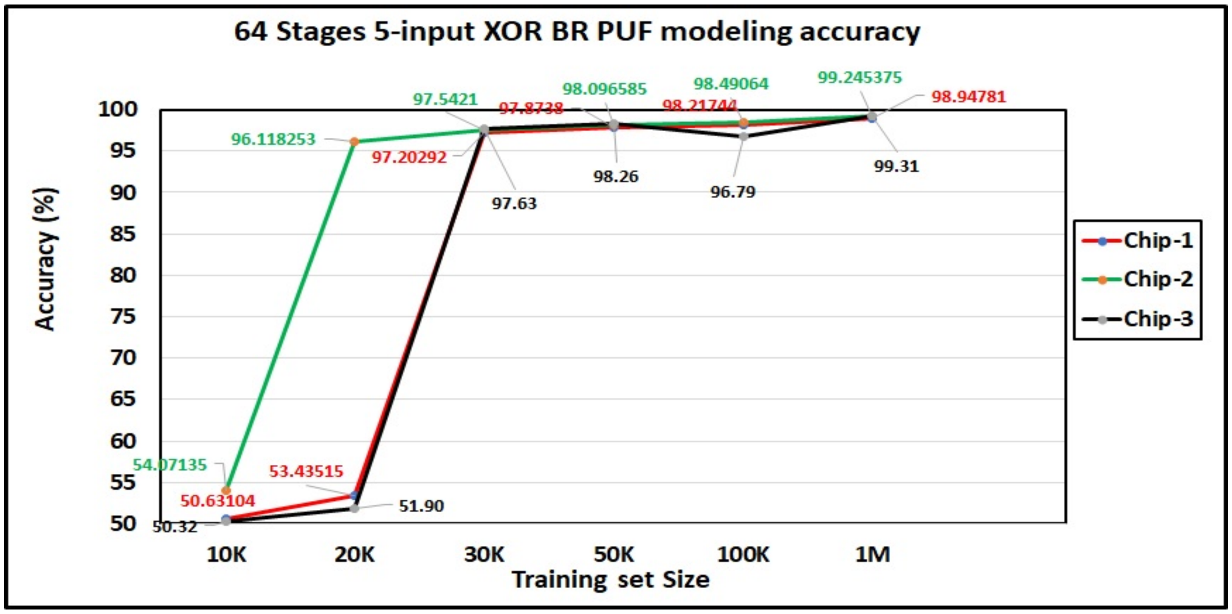


Figure 4.23: 5-input XOR BR PUF DL modeling accuracy(%)

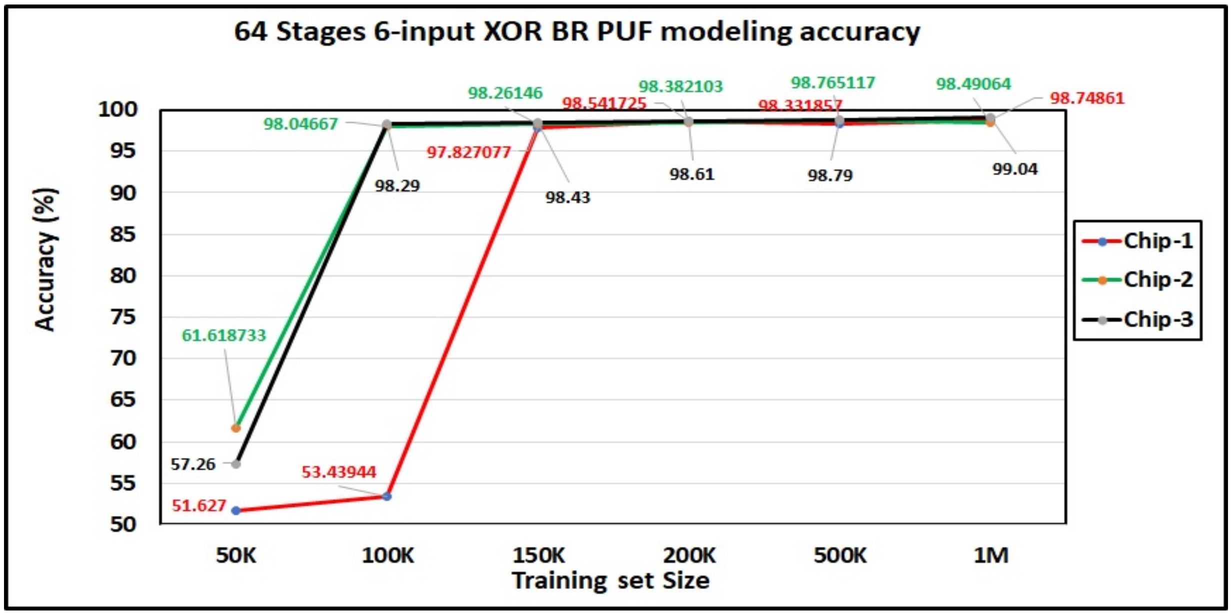


Figure 4.24: 6-input XOR BR PUF DL modeling accuracy(%)

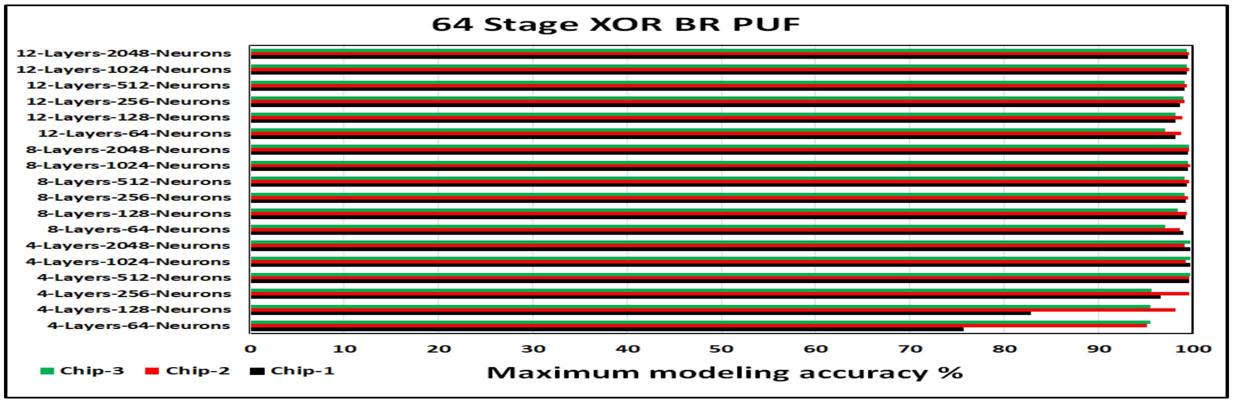
Table 4.18: 64-stage 5-input XOR BR PUF DL modeling accuracy(%).

Train-size	Chip1	Chip2	Chip3
10K	50.63%	54.07%	50.32%
20K	53.44%	96.12%	51.90%
30K	97.20%	97.54%	97.63%
50K	97.87%	98.10%	98.26%
100K	98.22%	98.49%	96.79%
1M	98.95%	99.25%	99.31%

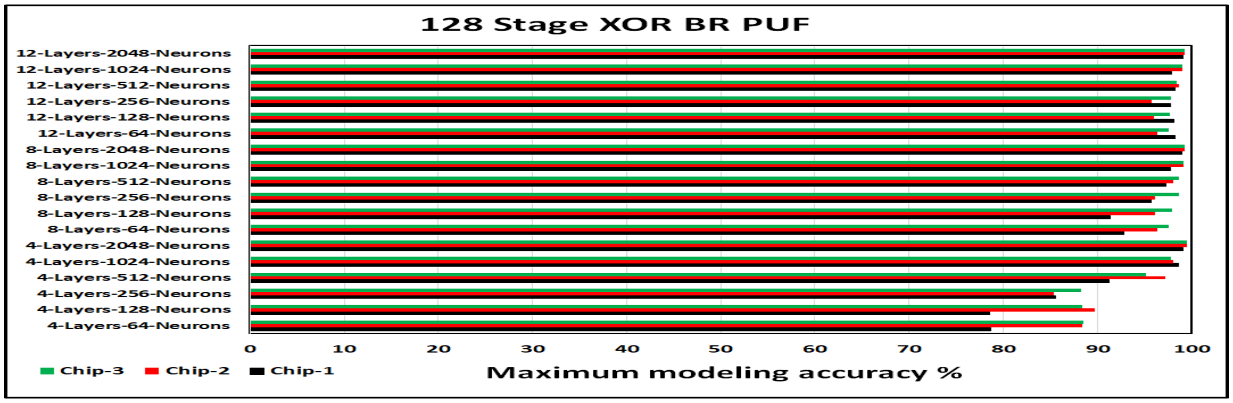
Table 4.19: 64-stage 6-input XOR BR PUF DL modeling accuracy(%).

Train-size	Chip1	Chip2	Chip3
50K	51.63%	61.62%	57.26%
100K	53.44%	98.05%	98.29%
150K	97.83%	98.26%	98.43%
200K	98.54%	98.38%	98.61%
500K	98.33%	98.77%	98.79%
1M	98.75%	98.49%	99.04%

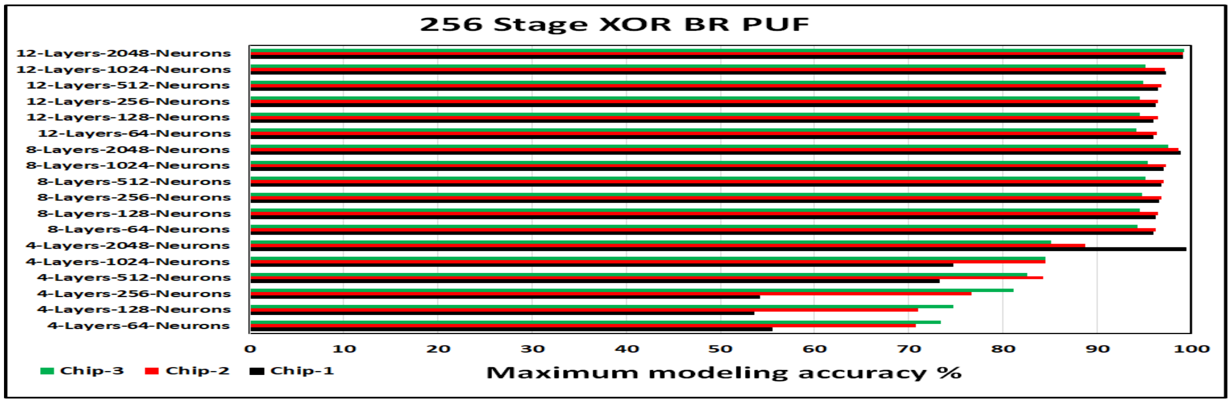
DNN parameters scalability (i.e. number of layers and number of hidden neurons per layer) with respect to the PUFs complexity and number of stages. Hence, the same modeling attack was invoked while varying the DNN number of layers (1,4,8,12) and the hidden neurons per layer (64, 128, 256, 512, 1024, 2048). Then, observe what the maximum accuracy will be and how long the network will take to converge. The training set size was chosen to be 1M CRPs in order to guarantee that obtained results depend on how the network architecture is varied not because of not enough training samples to train on. Total 540 training processes were run on all the 4-input XOR BR and XOR TBR PUF, 5-input XOR BR PUF, and 6-input XOR PUF instances. Figures 4.25, 4.26, 4.27. and 4.28 show the accuracy results of all PUF instances for all network configurations on chip 1. Additionally, Table 4.22 shows the network configurations that achieved maximum accuracy with minimum train time for every type. Obtained results for 4-input XOR BR and TBR PUFs show many interesting findings. Firstly, single layer NNs failed to model



(a) DNN Modeling Accuracy for 64 Stage 4-input XOR BR PUF with varying DNN parameters



(b) DNN Modeling Accuracy for 128 Stage 4-input XOR BR PUF with varying DNN parameters



(c) DNN Modeling Accuracy for 256 Stage 4-input XOR BR PUF with varying DNN parameters

Figure 4.25: DNN Scalability Analysis On 4-input XOR BR PUF. Showing Modeling Accuracy of Every Network Configuration

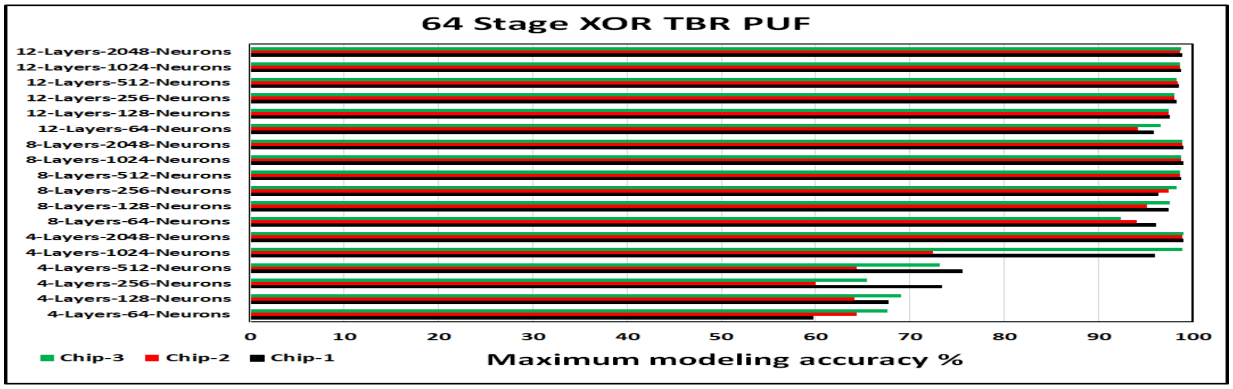
any instance successfully and modeling accuracy ranged between 55% - 62% except for one 256 XOR BR PUF instance that achieved 70% accuracy. This confirms the results obtained from LDA analysis discussed in Subsection 4.3.1.3, which showed that response classes are linearly inseparable for these PUF types. Furthermore, Table 4.22 shows that for most cases maximum accuracy could be achieved using smaller networks than the one used in our initial experiments. This means the same results could have been achieved in less training time than the bigger DNN. This is important because it affects the model size and the time needed to predict the PUF responses when operating in inference mode. Moreover, obtained results showed that network configurations with 4 layers and 2048 hidden neurons and 8 layers with any hidden size can achieve modeling accuracy $> 90\%$ for all PUF sizes. Note that there is a trade-off between convergence (number of iterations) and time, therefore for the 256 XOR TBR case in Table 4.22 a network with 4 layers converged slower than a similar one with 12 layers but it still could finish faster because it has way less number of computations. Hence, given a constant number of XOR inputs, the network scales linearly in terms of layers and hidden neurons with respect to XOR BR PUF stage size.

$$\#\text{ of Weights} = (m \times K) + ((N-1) \times K^2) + (K \times 2) \tag{4.18}$$

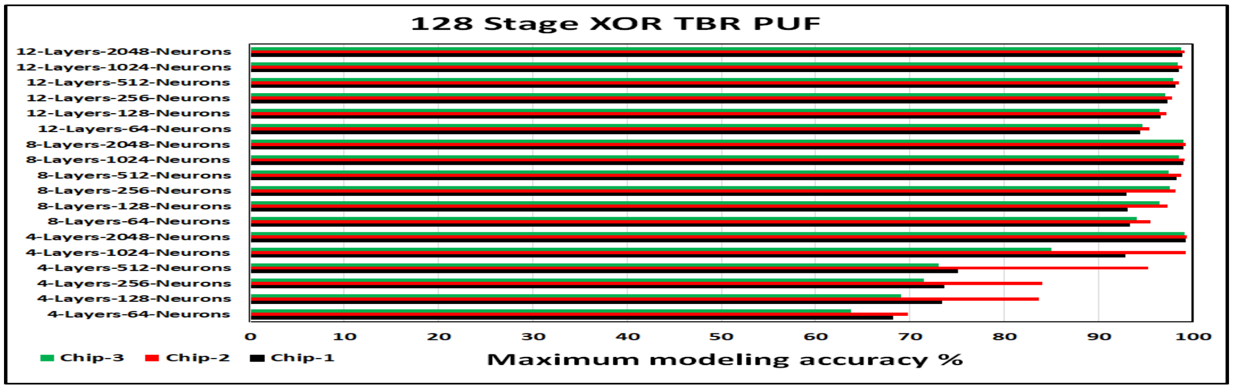
The number of weights to be updated in the network architecture shown in Figure 4.18 can be calculated using equation 4.18. Note that 'm' is the number of PUF stages, 'K' is the hidden neurons per layer, and 'N' is the number of fully connected layers. Therefore, adding a new layer has K^2 effect on the number of weights and consequently the computations. Hence, a balanced approach of constructing the DNN network for similar PUF architectures is by giving more priority to increase neurons per layer first. The following step is to slightly increase the number of layers to achieve the best accuracy/time trade-off.

The training time shown in Table 4.22 is for achieving the maximum accuracy. Using smaller network configurations could achieve a reasonable accuracy of 95% in much less training time. Hence, these timing figures are limited by the hardware used and the acceptable accuracy desired.

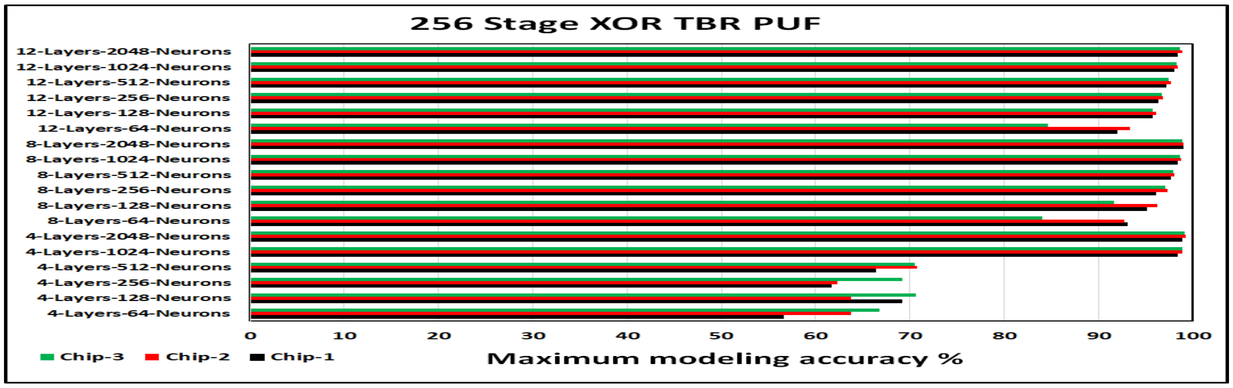
The results of 5-input and 6-input XOR BR PUFs showed similar behavior as shown in Figures 4.27 and 4.28. However, some 4-Layer networks had difficulty to achieve better accuracy as in the 4-input XOR BR PUF case. Moreover, the time needed to achieve similar accuracy increased by $\sim$ 3X for the 5-input and 6-input cases. Furthermore, these results empirically indicate that increasing the number of XOR inputs did not affect the number of layers needed to model such architectures. Hence, the depth of the network scales linearly with the increase of XOR inputs.



(a) DNN and SVM Modeling Accuracy for 64 Stage 4-input XOR TBR PUF



(b) DNN and SVM Modeling Accuracy for 128 Stage 4-input XOR TBR PUF



(c) DNN and SVM Modeling Accuracy for 256 Stage 4-input XOR TBR PUF

Figure 4.26: DNN Scalability Analysis On 4-input XOR TBR PUF. Showing Modeling Accuracy of Every Network Configuration

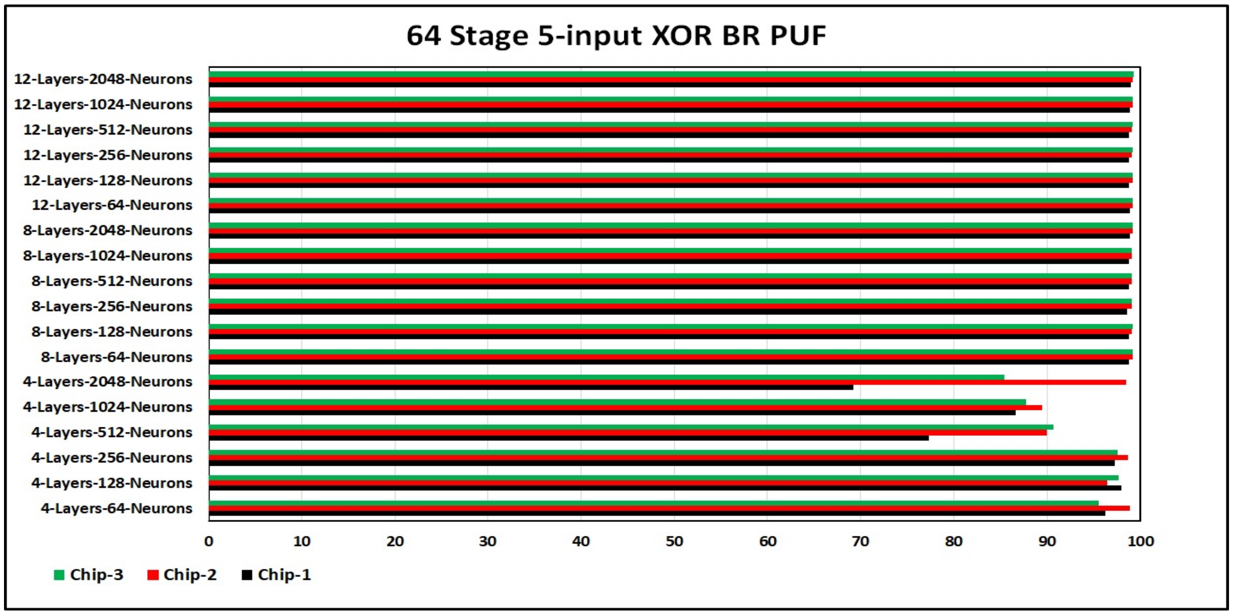


Figure 4.27: DNN Scalability Analysis On 5-input 64 stage XOR BR PUF. Showing Modeling Accuracy of Every Network Configuration

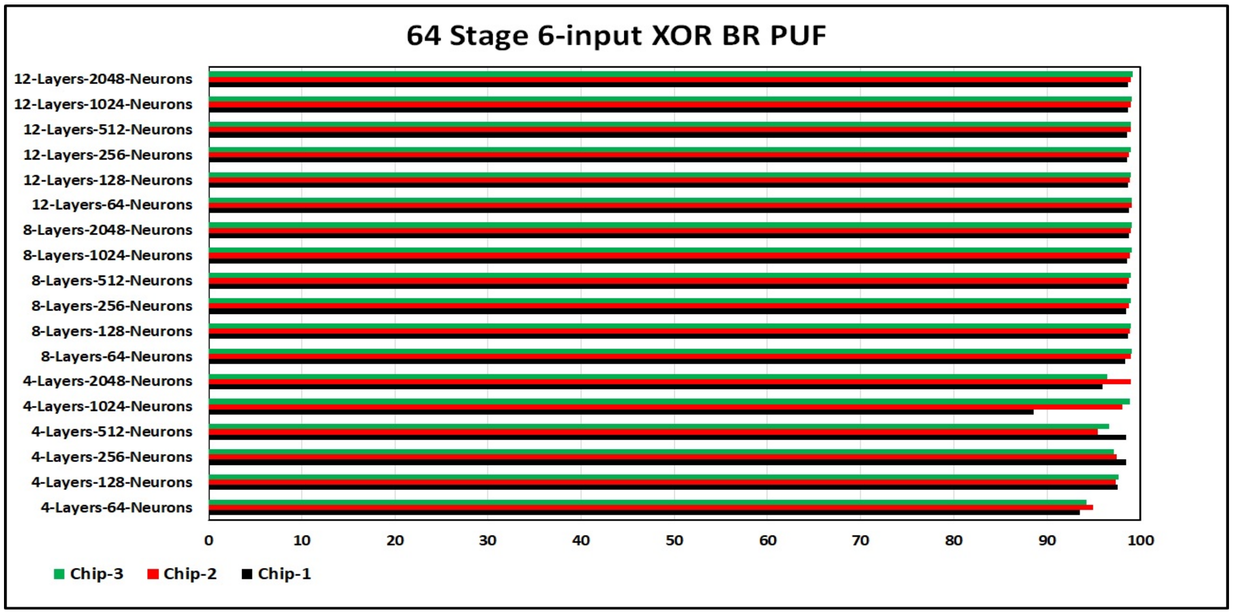


Figure 4.28: DNN Scalability Analysis On 6-input 64 stage XOR BR PUF. Showing Modeling Accuracy of Every Network Configuration

Table 4.22: DNN Scalability Analysis On 4-input XOR BR and TBR PUFs. Showing network configuration that achieved best accuracy and minimum training time

PUF Type	Accu.(%)	# of Layers	# of hidden neurons	training time (min)
64-XOR-BR	99.5%	8	1024	4.8
128-XOR-BR	99.2%	8	1024	20
256-XOR-BR	99.1%	12	2048	15
64-XOR-TBR	98.8%	8	1024	6.8
128-XOR-TBR	98.8	8	1024	7.6
256-XOR-TBR	99%	4	2048	9.6
5-input 64-XOR-BR	99.2%	12	64	15.04
6-input 64-XOR-BR	99%	8	256	15.19

4.4 Modeling attacks on obfuscated PUFs

This section provides an overview of two obfuscated PUFs architectures implemented to show how DL modeling attacks perform against challenge obfuscation techniques and how to develop countermeasures resistant to such attacks. the obfuscation logic elements including memory-based PUFs are implemented in software as a proof of concept. Hence, the original challenge is supplied as input to the obfuscation logic and the output modified challenge is sent normally to the hardware XOR BR-PUFs. For the obfuscated APUFs architecture, the memory-based PUFs responses are assumed to be generated as a binary string and the XORing with the original challenge is done on hardware. We adopted this approach because it is easier to implement and reuse the deployed XOR BR-PUF and APUF instances with no impact on hardware functionality. Furthermore, We are interested in the logical reasoning of how to design a new architecture to thwart DL modeling attacks.

4.4.1 Obfuscated PUF architecture 1 (Hierarchical XOR BR-PUF)

This architecture adopts a similar obfuscation technique to the one introduced in [53] by using a pool of memory-based PUFs responses to XOR with the original challenge. Note that memory-based PUFs should have a 50% randomness (50% responses are '1'), therefore,

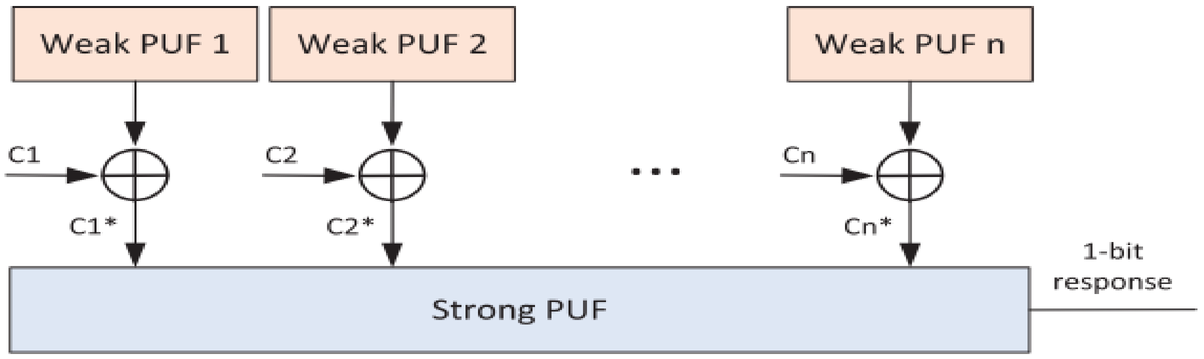


Figure 4.29: The Multi-PUF Architecture (MPUF) introduced in [53]

Table 4.23: PUF characteristics of Hierarchical XOR BR-PUF

PUF characteristic	Chip 1	Chip 2	Chip 3
Bias	53.3%	46.3%	1.5%
Noise	3.3%	1.4%	31.4%
Hamming Distance with Chip 1	—	66.05%	50.18%
Hamming Distance with Chip 2	66.05%	—	58.37%
Hamming Distance with Chip 3	50.18%	58.37%	—

nearly half of the challenge bits will be inverted and the rest will pass through without change. The XOR output result will be the new challenge to the 4-input XOR-BR PUF as shown in Fig 4.29. The PUF statistical properties are shown in Table 4.23 and it is similar to the 4-input XOR PUFs because the same PUF instances were used.

Although this technique hides the input PUF challenge, it suffers from two conceptual issues. It does not solve the problem of the limited set of influential bits for every BR PUF [26]. The relationship between the final and original challenges is not complex enough because their hamming distance will be constant all the time. Hence, only half of the challenge bits will change all the time. Therefore, It is expected that DL modeling attacks can overcome this obfuscation because it can learn the appropriate transformation of input features to correctly predict the target as we will see in the obtained results.

Table 4.24 shows the modeling attacks results against the obfuscated 4-input XOR BR PUF architectures with stage size = 64. Similar to non-obfuscated architectures, DL attacks could break the obfuscated PUF Architecture 1 (Hierarchical XOR BR-PUF). DL

Table 4.24: DL modeling accuracy(%) of obfuscated 64-Bit 4-input XOR BR PUF architectures.

Training Size	10K	20K	50K	100K
Chip 1	94.6%	97.1%	98%	99.1%
Chip 2	99.4%	99.4%	99.6%	99.7%
Chip 3	96.1%	96.2%	99.5%	99.7%

Table 4.25: PUF characteristics of Hierarchical DAPUF

	Chip 1	Chip 2	Chip 3	Chip 4	Chip 5	Chip 6
Bias	53.6%	40.67%	34.22%	50.65%	34.77%	38.7%
Average inter-chip HD	44.61%					

networks can learn the hidden relationship between the original and final challenges because the number of inverted bits and their positions is always constant. Hence, It is easy for the DNN to realize the transformation of input features.

4.4.2 Obfuscated PUF architecture 2 (Hierarchical DAPUF)

As mentioned before, The PUF architecture under attack is the same PUF introduced in [53]. Furthermore, the strong PUF used is a 64-stage arbiter PUF implemented using the Double arbiter PUF technique explained in subsection 3.1.2. Three modeling techniques were used in the attacks. LR technique was used to confirm the reported results in previous research in [53]. SVM with a polynomial kernel with degree 4 was applied to check if more sophisticated conventional ML techniques can break the PUF security. Finally, DNNs were applied to check the capability of DL techniques to overcome the obfuscation as was the case in obfuscated XOR BR PUFs. Attacks were launched against six PUFs implemented on six mojo v3 chips and Table 4.25 shows the bias and inter-chip hamming distance characteristics of the PUFs under attack. The dataset size is 1M CRPs and the testing set size is always 1M - training set size.

The results of DL modeling attacks are shown in Table 4.26 and Figure 4.30 illustrates the difference in modeling performance between SVM and LR techniques. Please note

Table 4.26: DL modeling accuracy(%) of obfuscated 64-Bit DAPUF architecture

Training Size	Chip 1	Chip 2	Chip 3	Chip 4	Chip 5	Chip 6
100K	86.91%	89.70%	92.18%	82.86%	91.8%	91.71%
200K	87.74%	90.73%	92.765%	84.67%	92.37%	91.95%
300K	88.71%	90.95%	93.23%	85.41%	92.56%	92.42%
400K	89.48%	91.27%	93.17%	85.66%	92.93%	92.60%
500K	89.22%	91.01%	93.13%	85.96%	92.88%	92.49%
600K	89.33%	91.27%	93.45%	86.03%	92.89%	92.97%
700K	89.90%	90.02%	93.39%	86.14%	93.12%	92.82%
800K	90.15%	91.58%	93.55%	86.57%	93.31%	92.91%
900K	91.27%	91.60%	93.57%	87.21%	93.25%	93.43%

that for every chip the dotted line represents LR results and SVM is represented by the continuous line and both have the same color.

Obtained results shown in Figure 4.30 confirm the same modeling performance of LR against this type of PUFs. The maximum accuracy is between 52% - 65% using training set = 100K CRPS. Furthermore, SVM with polynomial kernel outperformed the LR technique and could achieve maximum accuracy between 76% - 90% using 100K CRPs training. However, training SVM on 100K CRPs takes > 1hr for every PUF response. Hence, DL attacks were applied because it can use larger training sets and finish in a few minutes. The results in Table 4.26 show that DL attacks outperformed both LR and SVM techniques with maximum achieved accuracy between 87% - 94% using 900K CRPS for training. These results show that DL techniques could overcome the obfuscation technique suggested in [53] for both BR and APUF architectures.

4.5 Discussion on successful DL attacks and countermeasures

In order to understand why DL networks could model the previously mentioned PUF architectures while conventional ML techniques like SVM and single layer NN failed, the sources of modeling errors should be identified. Hence, let E(f) and $E_n(f)$ be the test error and the training error for any classifier f respectively. Furthermore, let F be the space of

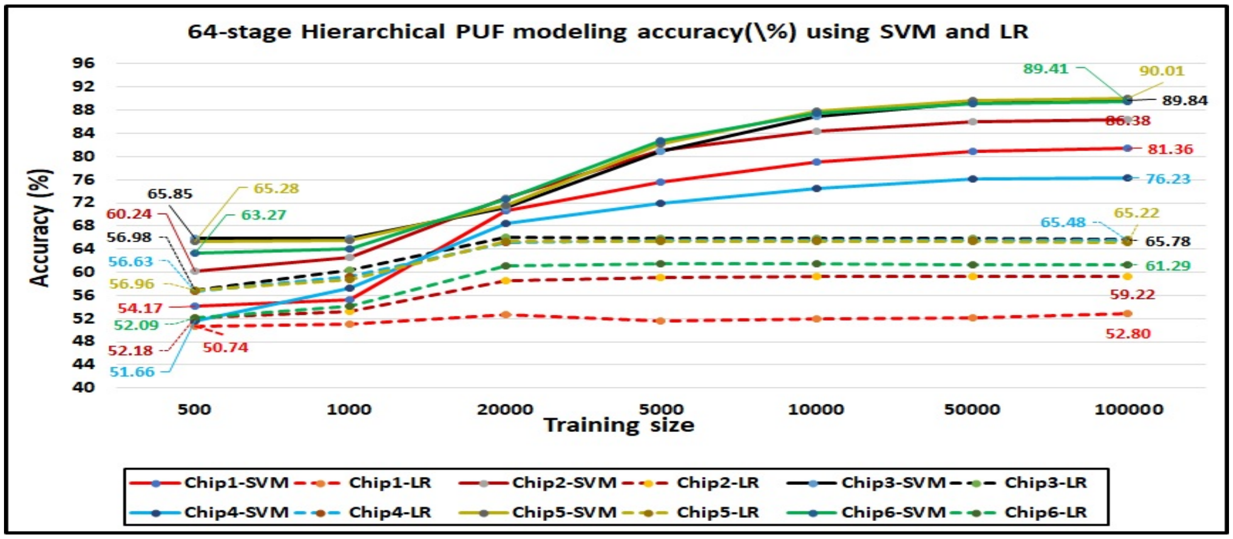


Figure 4.30: 64-stage Hierarchical PUF modeling accuracy(%) using SVM and LR

functions that can be expressed by deep neural networks, f_F^* is the best classifier in the F space and f^* is the best possible classifier. If $\hat{f}$ is the classifier function returned by the training algorithm, then its excess error from the best possible classifier $\varepsilon \triangleq E(\hat{f}) - E(f^*)$ can be attributed to two main terms as shown in equation 4.19 [23].

$$\varepsilon = [E(f_F^*) - E(f^*)] + [E(\hat{f}_n) - E(f_F^*)] \tag{4.19}$$

The first term is called the approximation error and measures how well the desired function can be approximated by a neural network using training samples. DNNs decrease the approximation error because they can express the composition of nonlinear functions effectively through their stacked layers (near zero training error. $\sim$ 0.0001 in our case). It was shown in [72] (theorem 4.1) that deep networks have a linear relationship with the input data dimension with respect to hidden neurons per layer, while shallow networks require an exponential number of neurons. This, in fact, was confirmed in our scalability analysis, where networks with a deeper number of layers could reach the 99% accuracy using neurons in the range of $\mathcal{O}(n)$ with respect to n-stage PUFs. The second term in equation 4.19 refers to the estimation error, which measures how well the trained model performs on out of sample data (generalization capability). DNNs with a large number of parameters and fully connected layers can control the generalization gap (small test error, <1% in our case), if the complexity of all functions in F space is not large (Theorems 6 and 7) in [23].

As a result, the successful DL attacks against the BR-PUF family and DAPUF family can be attributed to several reasons. Firstly the effect of influential bits that was reported in [26], which decreased the architecture complexity. Hence, introducing the XOR relationship was not sufficient to increase the complexity and counter the DL attacks. Furthermore, despite the use of a simplified mathematical model that does not represent the PUF operation accurately, DNNs could overcome that because it can learn the appropriate transformation of input features to correctly predict the target. Moreover, the lack of randomness when attempting to hide the challenge bits to counter DNNs modeling capabilities. Consequently, the obfuscated architectures could be attacked and modeled with accuracy similar to non-obfuscated ones.

4.6 The practicality of the DL attacks and applications

The discussion of DL modeling attacks involves the access to PUF, the number of CRPs required for a successful attack, and the power needed to read CRPs and perform the DL training and build a model. In the context of strong PUFs(i.e. BR PUF family), they are usually not protected against the process of sending out challenges and reading out the response to collect the CRPs necessary for the attack [76][49][6][92]. However, there have been other authentication protocols that hide the response using hash functions or other cryptographic schemes [31][99]. Therefore, for the DL modeling attacks to work successfully against these types of controlled PUF environment, The assumption is that the attacker gains physical access to the PUF. Furthermore, the response hiding technique may be overcome by probing the digital signals coming out of the PUF before being input to the cryptographic logic used to hide the response [75] and [76].

The results showed that an accuracy of 95% could be achieved using 20K and 50K, 100K CRPs for 4-input, 5-input, 6-input XOR BR-PUFs and XOR TBR-PUFs. This number is surprisingly small given the network architecture and large parameters used for training. However, it is comparable to the number of CRPs used in modeling attacks against 4-input XOR arbiter PUFs, which used 12K and 20K CRPs to break the 64 and 128-bit PUFs respectively[76]. Additionally, it is normal that electrical strong PUFs operate at frequencies of a few MHZ[75]. For example, the mojo FPGA chips operate at 50 MHZ and the maximum number of cycles needed for evaluation is 19K as mentioned in Table 4.12. Hence, with eased conditions and assuming it takes 20K cycle to read a response, it takes $\sim$ 7 mins to read 1M CRPs. Therefore, attacks against the DAPUF family, which required millions of CRPs for training could be achieved successfully.

The task of collecting CRPs is not computationally intensive, therefore, reading CRPs from chips with limited hardware resources as in [99] is applicable under the above-mentioned assumption. Additionally, as far as we know, there was not any power analysis for CRPs collection task in the previously published attacks. Although, the DNN training task is computationally intensive it is possible now to execute training tasks with minimized time and power cost using GPUs & ASIC chips (e.g. Tensor processing unit TPU and other accelerators). For cases where PUF attacks are not possible (the CRPs are hidden, silicon probing is not possible, or battery-power limits the number of CRPs), DNNs may be useful to measure post-silicon PUF security validation before the chip is employed in the field.

In our research, DL attacks were invoked against variants of BR and DAPUFs, which means that DL modeling attacks can be considered as a powerful tool in breaking the security of strong PUFs (i.e. APUF variants and XOR BR PUF family including obfuscated versions) using a simplified mathematical model. Furthermore, these attacks are practical with respect to the number of CRPs and the power needed to execute the attack. It is applicable to break wide range of security protocols that use strong PUFs for authentication [6][92][99], key establishment[89], and Oblivious transfer protocols[74].

4.6.1 Summary and Comparison with Previous Research

In this chapter, the deep learning modeling technique was introduced as a powerful tool to attack and break the security of complex strong PUF architectures implemented on real FPGAs. These architectures were reported to be resistant against conventional ML modeling techniques in previous research [94][53][85]. It was shown that DNN can be used along with a simplified mathematical model to attack 2-1, 3-1, 4-1 DAPUFs and 4-input, 5-input, 6-input XOR BR PUFs and 4-input XOR TBR PUF with varying number of stages (64, 128, 256). Furthermore, DNN attacks were invoked against 64-stage obfuscated Hierarchical XOR BR-PUF and DAPUF and could successfully model most of their responses with modeling accuracy $> 90\%$. The DL attacks on strong PUFs are practical and easy to launch because of the hardware and software support that enables training tasks in a matter of minutes as discussed in subsection 4.6. Even with the lack of an accurate mathematical model, DNNs were shown to provide better performance than conventional ML techniques like SVM with polynomial kernel and single layer NNs likely due to the ability of the nonlinear stacked layers to learn the appropriate data transformations needed (as discussed at subsection 4.5). Additionally, a detailed analysis was conducted to study the scalability of DNNs used to model XOR BR and TBR PUF architectures. This analysis included 486 modeling attacks on all PUF instances using 24 network configurations for every instance. Results showed that maximum accuracy can be achieved using smaller network architectures and the number of hidden neurons per layer scale linearly with the increase of PUFs stage size (given that the number of XOR inputs is constant), which agrees with the approximation theory of DNN as discussed in subsection 4.5. Furthermore, when increasing the number of XOR inputs, the same observation is repeated and the DNN network depth scales linearly as was shown in the 5-input and 6-input XOR BR PUFs.

Chapter 5

Shuffled challenge Obfuscation technique to countermeasure deep learning modeling attacks

This chapter will overview previous research countermeasures on PUF attacks as well as propose a new countermeasure which is empirically shown to thwart DL attacks. The proposed countermeasure is designed using the knowledge of how the DL attacks work.

5.1 Overview of Countermeasure Techniques

The previous research on PUF countermeasures focused on either proposing new complex architectures that are resilient against modeling attacks or introducing extra hardware circuits to detect and/or eliminate side-channel information leakage. The new proposed architectures to counteract modeling attacks can be summarized by two main approaches. First approach is by adding non-linearities in the mathematical relationship of PUFs challenges and response (e.g. XOR PUFs, FF PUFs) [17] [82]. This technique aimed at making it harder for conventional ML techniques to find a function that can approximate the PUF behavior. The other approach is to eliminate the accurate mathematical models of the PUFs under attack. This was done by either introducing PUF architectures for which we cannot derive an accurate mathematical model for (e.g BR PUF family), or using obfuscation techniques to hide PUF input challenge and complicate its relationship with the PUF response (e.g. controlled PUFs, lightweight PUFs, and interleaved PUFs, multiPUF

[62][61][16][53]). Additionally, some proposals from the application level tried to eliminate modeling attacks by hiding the response using hash functions or other cryptographic schemes as was done in [31][99]. Moreover, the authors in [86] proposed the use of noise bifurcation introduced in [97] as a countermeasure against modeling attacks on XOR APUFs. The main idea of this technique is to prevent the attacker from pairing challenge bits with the corresponding response bit. The authentication protocol assumes that the verifier has a software model of the PUF under test. Furthermore, the challenge C1 sent from the authentication server is used along with another challenge C2 computed using a challenge control logic to apply different m challenges on the PUF. Then, the m response bits are divided into m/d groups where d is the bit length of each group and both the response and C2 challenge are sent back to the verifier. The authentication server generates the m challenges using C1 and C2 and generates the PUF response using the already stored PUF models. The authentication server compares the responses with the response groups that have all '1's or all '0's to authenticate the chip. Hence, the attacker has no knowledge of the challenge-response relationship to build the machine learning model. Figure 5.1 shows the authentication system using noise bifurcation with response group length = 2. However, this approach has many vulnerable points like the challenge control logic and the random number generator, which may be attacked to reveal the hidden information [43][22]. Apart from strong PUFs, memory-based PUFs are resilient against modeling attacks because the physical random elements contributing to different CRPs are mostly independent of each other [58].

On the other hand, countermeasures proposed against side-channel attacks focus on adding extra noisy signals to stop the leakage. A countermeasure was proposed in [59] against PSA attacks against APUFs, which depends on CRPs with all '1's and all '0's. The main idea is to add APUF replications and use a differential approach to average the number of ones and zeros in every CRP. Another countermeasure proposed against EM side-channel attacks against RO PUFs in [66] is to force the application of parallel RO comparisons and place a restriction that a certain RO can be used once in two consecutive comparisons. Moreover, the same authors proposed the use of ripple counters instead of synchronous ones to reduce the number of clocked registers and consequently reduce the EM radiation.

As was discussed in chapter 4, it is obvious that proposed countermeasures were not successful against different attack schemes, although some of the attacks were successful given some restrictions on PUF architectures (e.g. FF PUFs, lightweight PUFs, maximum 9-input XOR PUF). Additionally, it adds more hardware overhead to store some information on the server-side, and to hide the challenge-response relation or to add noise to hide side-channel leakage. Consequently, this leads to more power consumption, which is a

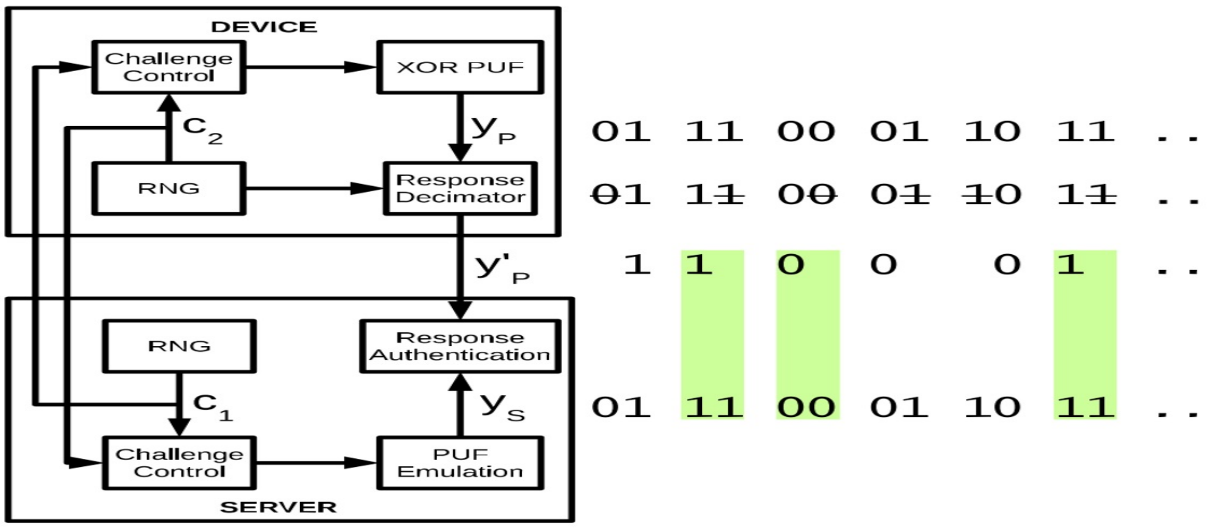


Figure 5.1: Authentication process using noise bifurcation [86].

crucial limiting factor in hardware design, especially in FPGAs. Furthermore, our research showed that using DL techniques one can extend the modeling attacks to break the security of complex PUFs that don't have an accurate mathematical model like BR PUFs or by using obfuscation techniques. Hence, in this chapter, we investigate the development of a new obfuscation technique as a countermeasure to resist DL modeling attacks with less hardware and power consumption overhead.

5.2 N-to-1 Shuffled-Challenge Hierarchical PUF

In this obfuscation technique, we try to improve the architecture in terms of the limited number of influential bits and the relationship between the original and final challenges. Firstly the N-to-1 term denotes that every N bits of the original challenge are responsible for determining the value of one final challenge bit. Fig 5.2 shows an example of a 2-to-1 shuffled-challenge hierarchical PUF, where every final challenge bit is derived by a multiplexer. Its select inputs are a pair of the original challenge bits selected randomly. The condition is no pairs are repeated and every bit is involved in determining the value of two different final challenge bits. For this architecture, there is no single bit responsible for determining the value of any specific PUF stage. Furthermore, every bit impacts two different bits of the final challenge. Hence, the number of influential bits should increase and their relationship is more complex. Furthermore, it is apparent that the number of challenge bits that change their values is not constant and their positions are dependent on the original challenge value as will be shown in the results. Additionally, in order to increase the randomness between the original and final challenges, the position of every final challenge bit at which a pair of bits determine its value is randomly assigned for every chip. Moreover, the four memory-PUF connected to every multiplexer inputs must represent 50% '1' values and their ordering is randomly selected from the six available choices that have two '1's and '0's. Finally, these design rules allowed this architecture to show a significant resistance against DL modeling attacks as will be shown in the results section.

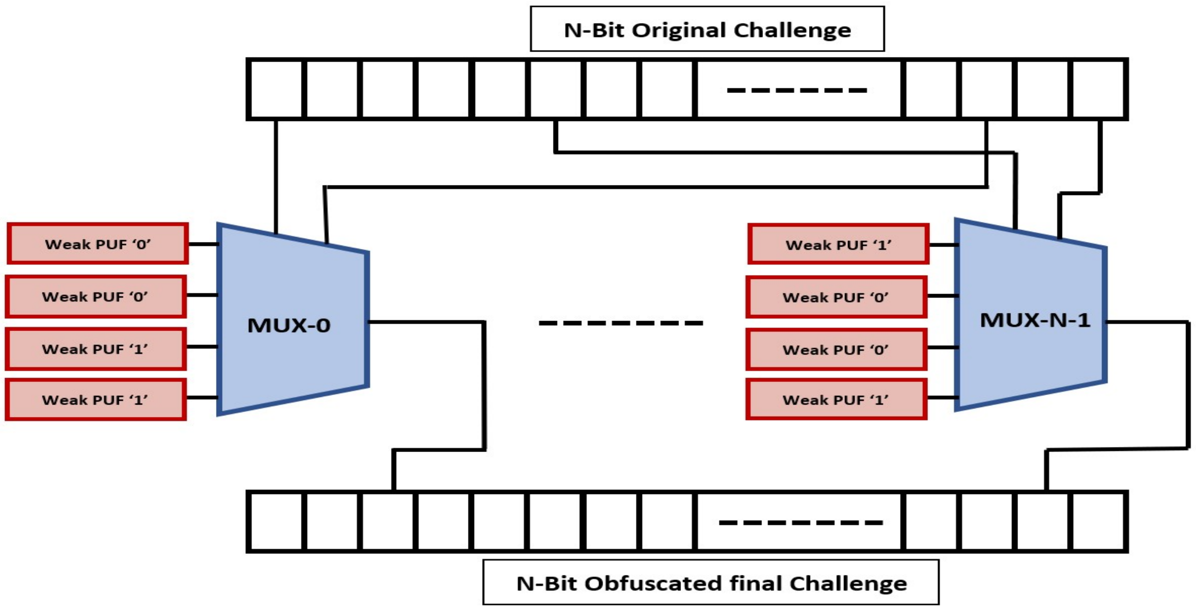


Figure 5.2: The 2-to-1 Shuffled-Challenge Hierarchical XOR BR-PUF Architecture

In this chapter, we introduce one variant of this architecture using 64-stage 4-input XOR BR PUF as the strong PUF underlying the obfuscation architecture.

5.2.1 Obtained results of 2-to-1 Shuffled-Challenge Hierarchical XOR BR-PUF

DL modeling attacks were invoked against 64-stage 4-input XOR PUFs. The three instances are implemented on the mojo v3 board similar to what was done in the previous

Table 5.1: PUF characteristics of 2-to-1 Shuffled-Challenge Hierarchical XOR BR PUF

PUF characteristic	Chip 1	Chip 2	Chip 3
Bias	47.8%	47.69%	46.5%
Noise	1.3%	1.2%	1.3%
Average inter-chip HD	49.17%		

Table 5.2: DL modeling accuracy(%) of 2-to-1 Shuffled-Challenge Hierarchical XOR BR PUF.

	Training Size = 20K	Training Size = 50K	Training Size = 100K
Chip 1	58.2%	76.5%	82.3%
Chip 2	70.4%	77.1%	83.7%
Chip 3	65.2%	76.8%	83.9%

chapter in section 4.4. Hence, the multiplexer logic is implemented in software using a python script and the original challenge is applied to the obfuscation function and the output modified challenge is sent to the PUF hardware as was discussed in section 4.3.2. Finally, Table 5.1 shows the characteristics of the PUFs under attack. The meaning of every specific characteristic is previously discussed in section 4.3.2.

Figure 5.3 shows a histogram of the hamming distance between original and modified challenge after the obfuscation over 1M CRPs. The graph shows that the average HD is between 33 and 34 bits out of 64 bits, which is close to the ideal desired value (32 bits). Furthermore, Figure 5.4 shows the number of value changes per challenge bit position out of the 1M CRPs. It is apparent that all challenge bits changed values 50% of the time except for bit 16, which changed all the time. The ideal desired case is for every challenge bit to change 50% of the time, unlike the obfuscation used in 4.4, where a constant set of the challenge bit will change all the time and the rest will retain their original values all the time.

Table 5.2 shows the modeling attacks results. The 2-to-1 Shuffled-Challenge Hierarchical XOR BR-PUF showed significant resistance against DL attacks. Reducing accuracy by nearly 30% - 40% using 20K training CRPs and maximum accuracy achieved is 84% using 100K CRPs. This is compared to the previous results in chapter 4, where 99% accuracy was achieved for the same PUF architectures even using another obfuscation technique.

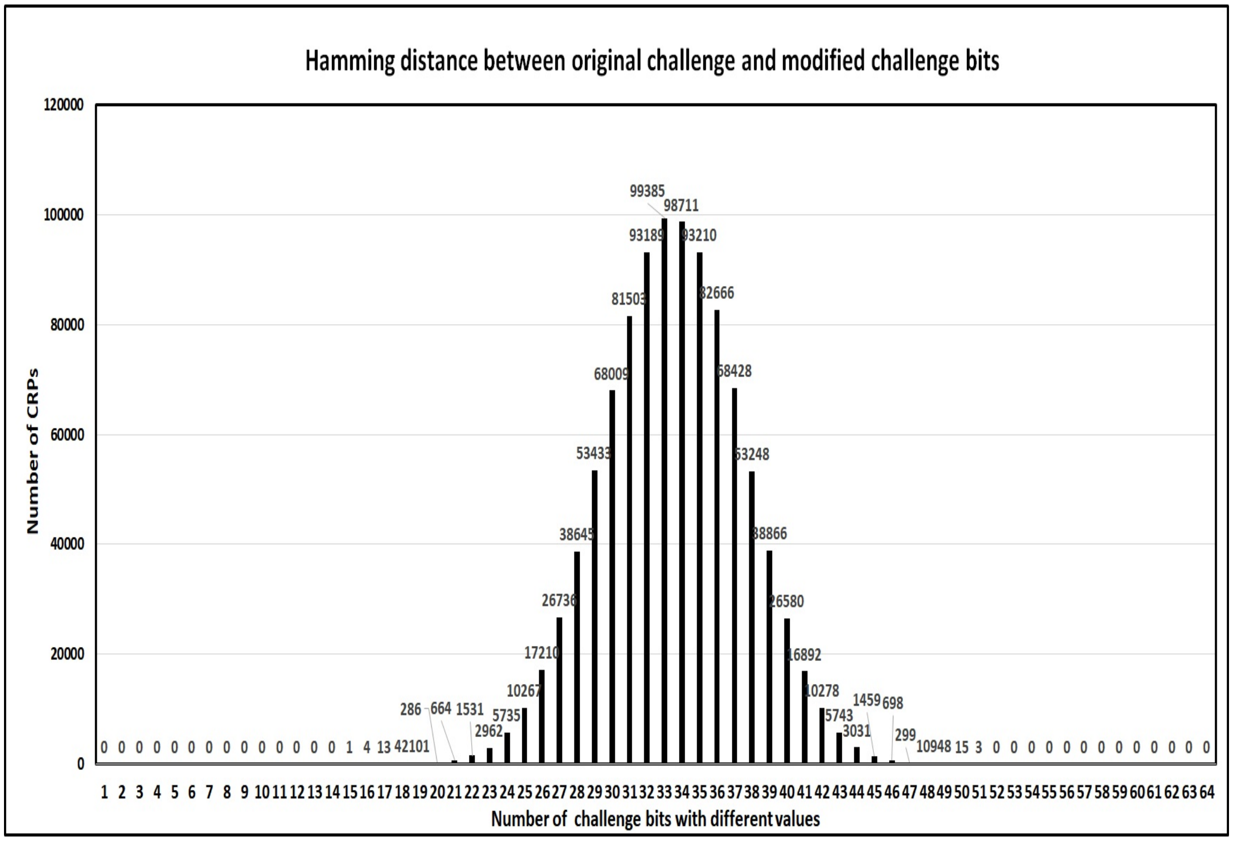


Figure 5.3: Hamming distance between original and modified challenge bits.

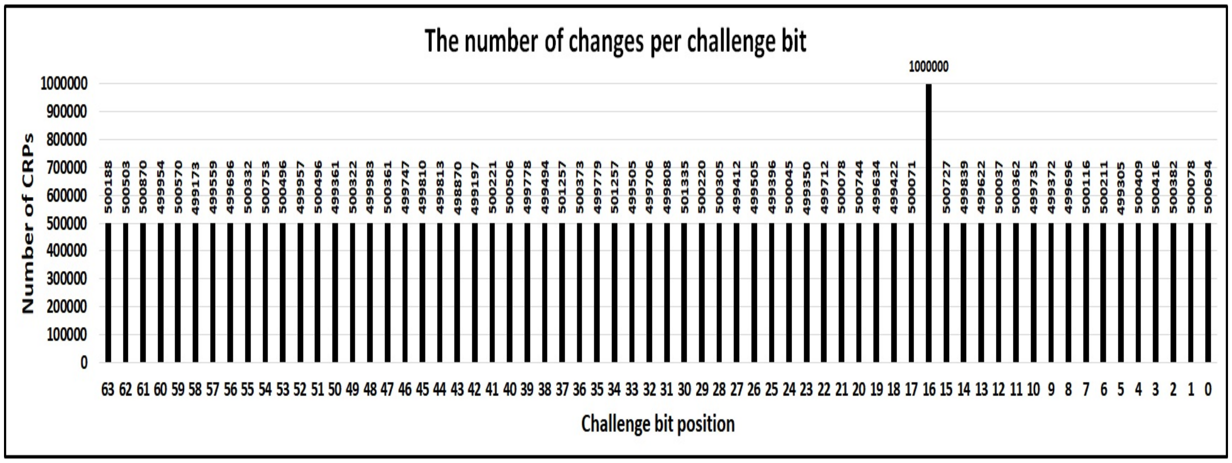


Figure 5.4: The number of changes per challenge bit.

Furthermore, it was noted that training error was small in all cases but test error of the trained model was worse, which means that the obtained model generalization gap was increased. This shows that the obfuscation technique used is relatively successful in increasing the randomness between the original and final challenges and reducing the effect of influential bits. Further analysis is required to enhance the resistance of this architecture against DL attacks and study the hardware and power overhead. However, it is worth mentioning that the multiplexers used to modify the challenge can be used for all PUF instances on the same chip, which reduces their overhead in terms of both hardware and power.

5.2.2 Summary

In this research, the design of the new countermeasure architecture was based on understanding how the DL attack works. Previous proposals of obfuscated PUF architectures like [53] did not take the DL attacks into account. Hence, The 2-to-1 Shuffled-Challenge Hierarchical XOR BR-PUF was introduced as a new architecture to countermeasure the DL attacks by overcoming the inherent problem of influential bits in BR PUFs and increasing the randomness between the original and final challenge bits. Hence, increasing the architecture complexity and the generalization gap of the DL model. Attacks on 64-stage instances showed significance resistance and a promising first step towards an ideal resilience against DL attacks. It showed far better resistance against the DL attacks because it was built to minimize the effects of the above-mentioned reasons. This could be achieved by allowing more than one bit from the original challenge to determine the value of every bit of the final input challenge supplied to the PUF. Additionally, every original challenge bit affects two different positions of the final input challenge. These modifications resulted in increasing the randomness between the original and final challenges and the architecture complexity due to spreading the influence of every challenge bit over more than one position. Hence, the deep learning generalization gap was increased and the test accuracy of DL attacks was worse despite the low training error achieved. Further analysis is needed to develop more complex versions of this obfuscated PUF and statistical metrics that measure the desired complexity of architectures to increase the generalization gap and counter these types of DL attacks. Finally, the obfuscation multiplexers used to modify the challenge can be used for all PUF instances on the same chip, which reduces their hardware and power overhead.

Chapter 6

Conclusion and future work

This dissertation has helped to address the important open questions related to how to securely implement strong PUFs on FPGAs and overcome the imposed constraints with minimum hardware, area, and power consumption overhead. Furthermore, this research investigated the ability to extend machine learning attacks to successfully break the security of more complex PUFs with less cost. Moreover, the research results have increased our state of the art understanding of how new PUF architectures and design techniques may be developed to thwart attacks. This section will provide a summary, and discussion of limitations, contributions, and future work.

6.1 PUF implementation on FPGAs

For the PUF implementation study, APUFs implemented in Spartan 6 FPGA have been analyzed with existing and new metrics using four experiments. In experiment 1 which used one bitstream (same manual placement configuration) on 5 different chips, inter-chip HDs indicated that approximately 22% of pairs of 8-bit responses (from the same challenge) of the APUFs on different chips differed in one or more bits. This was likely due to either the process variation being very low or most delay differences in APUFs were large enough to be unaffected by process variation. In experiment 2, it was demonstrated that changing the horizontal and vertical placement of every single PUF on every chip, improved the randomness of every chip's 8-bit PUF response, and the inter-chip hamming distance (HD improved to 37%). This effect is likely largely due to the variation in routing performed by the tools, although the process variation is also known to vary spatially within-die. Experiment 3 used correlation driven manual routing to decrease the delay difference at the stages that showed more influence on PUF response. However, results indicated that it was not sufficient to resist machine learning modeling attacks. Hence, Experiment 4 applied repetitive manual placement to facilitate the manual routing of stages with minimum delay difference and correlation analysis was used to further tweak the PUF implementation in the revisited experiment 4 to decrease the influence of specific challenge bits. Statistical analysis showed that this hybrid technique improved the PUF properties (i.e. overall delay difference, average delay difference) and produced two stages with a reported routing delay difference of zero. Additionally, the intra-chip PUF correlation was drastically decreased from 45% to 4%. Furthermore, the PUF instances in experiment 4 and 4-revisited showed more resistance to LR modeling attacks compared to the instances developed in the other experiments and the results reported in previous research [75]. However, APUFs architectural design allows modeling attacks to successfully clone its response, which was the motivation behind our work in chapters 4 and 5 to study the limit of modeling attacks and new architectures to counter them.

Generating PUFs without manual routing supports portability across different FPGA technologies, however, some manual routing may be crucial for good PUF performance. Although tweaking PUF designs is complex since many metrics are sensitive to design changes (%1s, inter-chip HD, HW, Phi plots), but automating the steps of finding the appropriate manual placement and routing will help this solution to be scalable and can be adopted for similar delay-based PUF architectures. Furthermore, once the perfect PUF implementation is done on one chip it can be exported to other chips with no change. Additionally, Phi correlations are proposed as a concise metric guiding the performance tweaking of the PUF, showing promise for identifying stages that require manual rerouting. This is unlike previous research that requires higher cost overheads, specifically utilizing additional PDL circuitry [60], doubling the area[85], or spreading placement of stages across entire FPGA and storing the bitstream configurations at the authentication server [80]. This research also showed that manually routed inter-stage wiring delay differences can be within assumed tool accuracy and process variation (e.g. as low as 0ps), thus only the wiring symmetry at the arbiter stage remains to be improved in order to support an ideal fully manually-routed APUF.

The task of finding the best manual placement and routing configurations is time consuming because it is done manually. Hence, this is one of our research main limitations. Therefore, future work will focus on providing more tooling support to automatically determine the best placement and routing configuration to modify the constraint configurations of the FPGA design. Hence, reduce the time needed to implement more ideal strong PUFs on FPGAs. Furthermore, we will extend our study by applying similar implementation techniques on other types of strong PUFs.

6.2 Modeling attacks against PUFs using DL techniques

For the modeling attack study, the deep learning modeling technique was introduced as a powerful tool to attack and break the security of complex strong PUF architectures implemented on real FPGAs. It was shown that DNN can be used along with a simplified mathematical model to attack 2-1, 3-1, 4-1 DAPUFs and 4-input, 5-input, 6-input XOR BR PUFs and 4-input XOR TBR PUF with varying number of stages (64, 128, 256). Furthermore, DNN attacks were invoked against 64-stage obfuscated Hierarchical XOR BR-PUF and DAPUF and could successfully model most of their responses with modeling accuracy $> 90\%$. The DL attacks on strong PUFs are practical and easy to launch because of the hardware and software support that enables training tasks in a matter of minutes as discussed in subsection 4.6.

Even with the lack of an accurate mathematical model, DNNs were shown to provide better performance than conventional ML techniques like SVM with polynomial kernel and single layer NNs likely due to the ability of the nonlinear stacked layers to learn the appropriate data transformations needed (as discussed at subsection 4.5). Additionally, a detailed analysis was conducted to study the scalability of DNNs used to model XOR BR and TBR PUF architectures. This analysis included 486 modeling attacks on all PUF instances using 24 network configurations for every instance. Results showed that maximum accuracy can be achieved using smaller network architectures and the number of hidden neurons per layer scale linearly with the increase of PUFs stage size (given that the number of XOR inputs is constant), which agrees with the approximation theory of DNN as discussed in subsection 4.5. Furthermore, when increasing the number of XOR inputs, the same observation is repeated and the DNN network depth scales linearly as was shown in the 5-input and 6-input XOR BR PUFs.

One limitation of this type of attack is that the CRPs used for training are assumed to be available to the attacker. However, recent authentication protocols have been introduced, which hide the CRPs [31][99]. Hence, future work will focus on investigating hybrid attacking techniques using side-channel leaks (e.g. photonic emissions) along with ML modeling to break the security of such protocols. Furthermore, more experiments using other DL network architectures like recurrent neural networks (RNNs) and residual networks (Resnet) should be conducted to observe their modeling capability compared to feed-forward DNNs and investigate if there exists a better DL technique that uses fewer CRPs to achieve similar modeling results. Finally, we will focus on applying DL modeling attacks on PUF architectures that use other obfuscation techniques.

6.3 Countermeasures against DL modeling attacks

For the countermeasure study, the 2-to-1 Shuffled-Challenge Hierarchical XOR BR-PUF was introduced as a new architecture to countermeasure the DL attacks. This new design was shown to overcome the inherent problem of influential bits in BR PUFs and increased the randomness between the original and final challenge bits, at the expense of increasing the architecture complexity and the generalization gap of the DL model. Attacks on 64-stage instances showed significance resistance and a promising first step towards an ideal resilience against DL attacks. It showed a far better resistance against the DL attacks (40% to 16% less modeling accuracy) because it was built to minimize the effects of the above-mentioned reasons. This could be achieved by allowing more than one bit from the original challenge to determine the value of every bit of the final input challenge supplied to the PUF. Additionally, every original challenge bit affects two different positions of the final input challenge. These modifications resulted in increasing the randomness between the original and final challenges and the architecture complexity due to spreading the influence of every challenge bit on more than one position. Hence, the deep learning generalization gap was increased and the test accuracy of DL attacks was worse despite the low training error achieved. It is assumed that the attacker has no access to the obfuscation circuit and can only read the input original challenge and its corresponding response. Moreover, the memory-based PUFs mimic the behavior of SRAM PUFs (e.g. butterfly PUFs) and are used as inputs of the multiplexers. These PUFs are characterized for every chip to guarantee that all multiplexers have 50% 0's and 1's inputs. This is not a significant overhead because all implemented PUFs should always be characterized to make sure they meet the ideal statistical properties. After the characterization step is done, the obfuscation technique is automatically implemented by rerouting within our PUF design, for example routing each memory-based PUF to an input of a multiplexer. Note the FPGA bitstream is encrypted hence it would be difficult for attackers to modify the design exposing the internal nets of the obfuscated PUF. Further analysis is needed to develop more complex versions of this obfuscated PUF and statistical metrics that measure the desired complexity of architectures to increase the generalization gap and counter these types of DL attacks. Finally, the obfuscation multiplexers used to modify the challenge can be used for all PUF instances on the same chip, which reduces their hardware and power overhead. This is an important observation because, in many IoT applications with scarce hardware resources, the use of an encryption scheme to hide the PUF input challenge may not be feasible since it may require too large a resource cost.

Future work will focus on using strong PUFs other than the BRPUF family and research the effect of the shuffled-challenge obfuscation technique on their resistance. Furthermore, new variants of this obfuscation technique will be developed using larger multiplexers to make more challenge bits involved in determining the new modified challenge bit values and observe their effect on hardening the modeling attacks against such architectures.

Another limitation of this research is that only one type of FPGAs (45nm technology) was used in the experiments. Current PUF research appears to lack investigations of the effect of technology scaling on the implemented PUF performance and randomness. Technology scale down may decrease the sources of randomness generated by hardware fabrication and affect PUF behavior. Hence, experiments on chips with smaller technology (e.g. 27nm and less) should be conducted to characterize the implemented PUFs. Moreover, it was shown in the literature that strong PUFs like APUFs suffer from a reliability issue under the voltage and temperature variation [9][42]. Hence, it is necessary to study the effect of temperature and voltage variations on the newly proposed strong PUFs.

6.4 Final conclusion

Finally, this dissertation is concluded by providing a brief summary of the main contributions as follows:

- Introducing an implementation technique of PUFs on FPGAs using repetitive manual placement, manual routing, and correlation analysis. We could introduce better APUFs compared to previous research in terms of statistical properties and resistance against ML attacks.
- Using deep learning techniques to successfully attack three families of architectures, which were resistant to ML modeling techniques in previous research:
  - 2-1, 3-1, and 4-1 DAPUFs [45]
  - 4-input & 5-input XOR BR and TBR PUFs (64, 128, 256 stages) [44]
  - Hierarchical obfuscated PUF using 4-input 64-bit XOR BR-PUF [44] and 64-bit APUF.
- Providing the first analysis of DL modeling networks scaling with respect to the PUF architecture variations (number of stages [44], number of XOR inputs).
- Empirically showing that PUF architectures with a number of XOR inputs $>$ ln(number of stages) can be successfully attacked (using 5-input and 6-input 64-bit XOR BR PUF).

- providing a new Hardware countermeasure based on understanding how DL modeling attacks work, unlike previous research that did not account for DL attacks. the N-to-1 Shuffled-challenge was introduced as an obfuscation technique, in which the 2-to-1 design proved to be (16% to 40%)more resilient to DL modeling attacks [44].

References

[1] LR with non-linear decision boundary and RProp optimization. http://www.pcp.in.tum.de/code/lr.zip. Accessed: 2018-08-30.

[2] Magnetek(r), magneprint(r). http://www.magneprint.com/. Accessed: 2020-02-15.

[3] Mojo v3 development board. https://embeddedmicro.com/products/mojo-v3.html. Accessed: 2020-02-15.

[4] Spartan-6 family overview. https://www.xilinx.com/support/documentation/data_sheets/ds160.pdf. Accessed: 2020-02-15.

[5] Spartan-6 fpga configurable logic block user guide. https://www.xilinx.com/support/documentation/user_guides/ug384.pdf. Accessed: 2020-02-15.

[6] Mete Akgün and M. Ufuk Çaglayan. Providing destructive privacy and scalability in rfid systems using pufs. *Ad Hoc Netw.*, 32(C):32–42, September 2015.

[7] Georg T. Becker and Raghavan Kumar. Active and passive side-channel attacks on delay based puf designs. *IACR Cryptology ePrint Archive*, 2014:287, 2014.

[8] K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer. High-performance cmos variability in the 65-nm regime and beyond. *IBM Journal of Research and Development*, 50(4.5):433–449, July 2006.

[9] Mudit Bhargava, Cagla Cakir, and Ken Mai. Comparison of bi-stable and delay-based physical unclonable functions from measurements in 65nm bulk CMOS. *Proceedings of the Custom Integrated Circuits Conference*, pages 0–3, 2012.

[10] Christoph Bhm and Maximilian Hofer. *Physical Unclonable Functions in Theory and Practice*. Springer Publishing Company, Incorporated, 2012.

[11] Christina Brzuska, Marc Fischlin, Heike Schröder, and Stefan Katzenbeisser. Physically uncloneable functions in the universal composition framework. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 51–70, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[12] James Buchanan, Russell Cowburn, Ana-Vanessa Jausovec, Dorothee Petit, Peter Seem, Gang Xiong, Del Atkinson, Kate Fenton, Dan Allwood, and M. Bryan. Forgery: 'fingerprinting' documents and packaging. *Nature*, 436:475, 08 2005.

[13] P. Bulens, F.-X. Standaert, and J.-J. Quisquater. How to strongly link data and its medium: the paper case. *IET Information Security*, 4:125–136(11), September 2010.

[14] Urbi Chatterjee, Rajat Subhra Chakraborty, and Debdeep Mukhopadhyay. A puf-based secure communication protocol for iot. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(3):67, 2017.

[15] Q. Chen, G. Csaba, P. Lugli, U. Schlichtmann, and U. Rührmair. Characterization of the bistable ring puf. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1459–1462, March 2012.

[16] Qingqing Chen, György Csaba, Paolo Lugli, Ulf Schlichtmann, and Ulrich Rührmair. The bistable ring puf: A new architecture for strong physical unclonable functions. In *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 134–141. IEEE, 2011.

[17] Lim D. Extracting secret keys from integrated circuits. Master's thesis, MIT, 2004.

[18] Daihyun Lim, J. W. Lee, B. Gassend, G. E. Suh, M. van Dijk, and S. Devadas. Extracting secret keys from integrated circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(10):1200–1205, Oct 2005.

[19] Gerald DeJean and Darko Kirovski. Rf-dna: Radio-frequency certificates of authenticity. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 346–363, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[20] J. Delvaux and I. Verbauwhede. Side channel modeling attacks on 65nm arbiter pufs exploiting cmos device noise. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 137–142, 2013.

[21] Mahmoud A Elmohr. Embedded systems security: On em fault injection on risc-v and br/tbr puf design on fpga. Master's thesis, University of Waterloo, 2020.

[22] S. Ergün. Attack on a microcomputer-based random number generator using auto-synchronization. In *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 1–4, 2019.

[23] Jianqing Fan, Cong Ma, and Yiqiao Zhong. A Selective Overview of Deep Learning. *arXiv e-prints*, page arXiv:1904.05526, Apr 2019.

[24] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119 – 139, 1997.

[25] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, August 1997.

[26] Fatemeh Ganji, Shahin Tajik, Fabian Fäßler, and Jean-Pierre Seifert. Strong machine learning attack against pufs with no mathematical model. Cryptology ePrint Archive, Report 2016/606, 2016. http://eprint.iacr.org/2016/606.

[27] Fatemeh Ganji, Shahin Tajik, and Jean-Pierre Seifert. Why attackers win: On the learnability of xor arbiter pufs. In Mauro Conti, Matthias Schunter, and Ioannis Askoxylakis, editors, *Trust and Trustworthy Computing*, pages 22–39, Cham, 2015. Springer International Publishing.

[28] B. Gassend. Physical random functions. Master's thesis, MIT, 2003.

[29] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 148–160, New York, NY, USA, 2002. ACM.

[30] Blaise Gassend, Daihyun Lim, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Identification and authentication of integrated circuits. *Concurrency - Practice and Experience*, 16:1077–1098, 09 2004.

[31] P. Gope, J. Lee, and T. Q. S. Quek. Lightweight and practical anonymous authentication protocol for rfid systems using physically unclonable functions. *IEEE Transactions on Information Forensics and Security*, 13(11):2831–2843, Nov 2018.

[32] Jorge Guajardo, Sandeep S. Kumar, Geert-Jan Schrijen, and Pim Tuyls. Fpga intrinsic pufs and their use for ip protection. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 63–80, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[33] Jorge Guajardo, Boris Škorić, Pim Tuyls, Sandeep S. Kumar, Thijs Bel, Antoon H. M. Blom, and Geert-Jan Schrijen. Anti-counterfeiting, key distribution, and key storage in an ambient world via physical unclonable functions. *Information Systems Frontiers*, 11(1):19–41, Mar 2009.

[34] Ghaith Hammouri, Aykutlu Dana, and Berk Sunar. Cds have fingerprints too. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 348–362, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[35] Ghaith Hammouri, Erdinç Öztürk, Berk Birand, and Berk Sunar. Unclonable lightweight authentication scheme. In Liqun Chen, Mark D. Ryan, and Guilin Wang, editors, *Information and Communications Security*, pages 33–48, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[36] Helena Handschuh, Geert-Jan Schrijen, and Pim Tuyls. *Hardware Intrinsic Security from Physically Unclonable Functions*, pages 39–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[37] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[38] C. Helfmeier, C. Boit, D. Nedospasov, and J. Seifert. Cloning physically unclonable functions. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 1–6, 2013.

[39] R. Helinski, D. Acharyya, and J. Plusquellic. A physical unclonable function defined using power distribution system equivalent resistance variations. In *2009 46th ACM/IEEE Design Automation Conference*, pages 676–681, July 2009.

[40] Daniel E. Holcomb, Wayne P. Burleson, and Kevin Fu. Initial sram state as a fingerprint and source of true random numbers for rfid tags. In *In Proceedings of the Conference on RFID Security*, 2007.

[41] G. Hospodar, R. Maes, and I. Verbauwhede. Machine learning attacks on 65nm arbiter pufs: Accurate modeling poses strict bounds on usability. In *2012 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 37–42, Dec 2012.

[42] Stefan Katzenbeisser, Ünal Kocabaş, Vladimir Rožić, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. Pufs: Myth, fact or busted? a security evaluation of physically unclonable functions (pufs) cast in silicon. In *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'12, pages 283–301, Berlin, Heidelberg, 2012. Springer-Verlag.

[43] John Kelsey, Bruce Schneier, David Wagner, and Counterpane Systems. Cryptanalytic attacks on pseudorandom number generators. *Lecture Notes in Computer Science*, 1372, 11 2000.

[44] M. Khalafalla, M. A. Elmohr, and C. Gebotys. Going deep: Using deep learning techniques with simplified mathematical models against xor br and tbr pufs (attacks and countermeasures). In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, December 2020.

[45] M. Khalafalla and C. Gebotys. Pufs deep attacks: Enhanced modeling attacks using deep learning techniques to break the security of double arbiter pufs. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 204–209, March 2019.

[46] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[47] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[48] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, page 388–397, Berlin, Heidelberg, 1999. Springer-Verlag.

[49] L. Kulseng, Z. Yu, Y. Wei, and Y. Guan. Lightweight mutual authentication and ownership transfer for rfid systems. In *2010 Proceedings IEEE INFOCOM*, pages 1–5, March 2010.

[50] S. S. Kumar, J. Guajardo, R. Maes, G. Schrijen, and P. Tuyls. Extended abstract: The butterfly puf protecting ip on every fpga. In *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 67–70, June 2008.

[51] J. W. Lee, , B. Gassend, G. E. Suh, M. van Dijk, and S. Devadas. A technique to build a secret key in integrated circuits for identification and authentication applications. In *2004 Symposium on VLSI Circuits. Digest of Technical Papers (IEEE Cat. No.04CH37525)*, pages 176–179, June 2004.

[52] K. Lofstrom, W. R. Daasch, and D. Taylor. Ic identification circuit using device mismatch. In *2000 IEEE International Solid-State Circuits Conference. Digest of Technical Papers (Cat. No.00CH37056)*, pages 372–373, Feb 2000.

[53] Qingqing Ma, Chongyan Gu, Neil Hanley, Chenghua Wang, Weiqiang Liu, and Máire O'Neill. A machine learning attack resistant multi-PUF design on FPGA. *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, 2018-Janua:97–104, 2018.

[54] T. Machida, D. Yamamoto, M. Iwamoto, and K. Sakiyama. A new mode of operation for arbiter puf to improve uniqueness on fpga. In *2014 Federated Conference on Computer Science and Information Systems*, pages 871–878, Sep. 2014.

[55] T. Machida, D. Yamamoto, M. Iwamoto, and K. Sakiyama. Implementation of double arbiter puf and its performance evaluation on fpga. In *The 20th Asia and South Pacific Design Automation Conference*, pages 6–7, Jan 2015.

[56] Roel Maes, Pim Tuyls, Ingrid Verbauwhede, and Leuven Esat-cosic. Intrinsic pufs from flip-flops on reconfigurable devices," in wissec, 2008.

[57] Roel Maes, Vincent Van Der Leest, Erik Van Der Sluis, and Frans Willems. Secure key generation from biased PUFs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9293:517–534, 2015.

[58] Roel Maes and Ingrid Verbauwhede. *Physically Unclonable Functions: A Study on the State of the Art and Future Research Directions*, pages 3–37. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[59] Ahmed Mahmoud, Ulrich Rührmair, Mehrdad Majzoobi, and Farinaz Koushanfar. Combined modeling and side channel attacks on strong pufs. *IACR Cryptology ePrint Archive*, 2013:632, 2013.

[60] M. Majzoobi, F. Koushanfar, and S. Devadas. Fpga puf using programmable delay lines. In *2010 IEEE International Workshop on Information Forensics and Security*, pages 1–6, Dec 2010.

[61] M. Majzoobi, F. Koushanfar, and M. Potkonjak. Lightweight secure pufs. In *2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 670–673, Nov 2008.

[62] M. Majzoobi, F. Koushanfar, and M. Potkonjak. Testing techniques for hardware security. In *2008 IEEE International Test Conference*, pages 1–10, Oct 2008.

[63] Mehrdad Majzoobi, Akshat Kharaya, Farinaz Koushanfar, and Srinivas Devadas. Automated design, implementation, and evaluation of arbiter-based puf on fpga using programmable delay lines. *IACR Cryptology ePrint Archive*, 2014:639, 2014.

[64] Mehrdad Majzoobi, Farinaz Koushanfar, and Miodrag Potkonjak. Techniques for design and implementation of secure reconfigurable pufs. *ACM Trans. Reconfigurable Technol. Syst.*, 2(1):5:1–5:33, March 2009.

[65] Jimson Mathew, Rajat Chakraborty, Durga Sahoo, Yuanfan Yang, and Dhiraj Pradhan. A novel memristor based physically unclonable function. *Integration, the VLSI Journal*, 51, 05 2015.

[66] Dominik Merli, Dieter Schuster, Frederic Stumpf, and Georg Sigl. Semi-invasive em attack on fpga ro pufs and countermeasures. In *Proceedings of the Workshop on Embedded Systems Security*, WESS '11, New York, NY, USA, 2011. Association for Computing Machinery.

[67] A. Mills, S. Vyas, M. Patterson, C. Sabotta, P. Jones, and J. Zambreno. Design and evaluation of a delay-based fpga physically unclonable function. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pages 143–146, Sep. 2012.

[68] Sergey Morozov, Abhranil Maiti, and Patrick Schaumont. Comparative analysis of delay based puf implementations on fpga. *IACR Cryptology ePrint Archive*, 2009:629, 01 2009.

[69] Sergey Morozov, Abhranil Maiti, and Patrick Schaumont. An analysis of delay based puf implementations on fpga. In *Proceedings of the 6th International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ARC'10, pages 382–387, Berlin, Heidelberg, 2010. Springer-Verlag.

[70] P. H. Nguyen, D. P. Sahoo, R. S. Chakraborty, and D. Mukhopadhyay. Efficient attacks on robust ring oscillator puf with enhanced challenge-response set. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 641–646, March 2015.

[71] Ravikanth Pappu, Ben Recht, Jason Taylor, and Neil Gershenfeld. Physical one-way functions. *Science*, 297(5589):2026–2030, 2002.

[72] David Rolnick and Max Tegmark. The power of deeper networks for expressing natural functions. *CoRR*, abs/1705.05502, 2017.

[73] Marcel W. Muller Ronald S. Indeck. Method and apparatus for fnger printing magnetic media, u.s. patent no. 5365586, November 1994.

[74] Ulrich Rührmair. Oblivious transfer based on physical unclonable functions. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, TRUST'10, pages 430–440, Berlin, Heidelberg, 2010. Springer-Verlag.

[75] Ulrich Rührmair, Frank Sehnke, Jan S ölter, Gideon Dror, Srinivas Devadas, and J ürgen Schmidhuber. Modeling attacks on physical unclonable functions. *Proceedings of the 17th ACM conference on Computer and communications security - CCS '10*, page 237, 2010.

[76] Ulrich Ruhrmair, Jan Solter, Frank Sehnke, Xiaolin Xu, Ahmed Mahmoud, Vera Stoyanova, Gideon Dror, Jurgen Schmidhuber, Wayne Burleson, and Srinivas Devadas. Puf modeling attacks on simulated and silicon data. *Trans. Info. For. Sec.*, 8(11):1876–1891, November 2013.

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

[78] Dieter Schuster and Robert Hesselbarth. Evaluation of bistable ring pufs using single layer neural networks. In *International Conference on Trust and Trustworthy Computing*, pages 101–109. Springer, 2014.

[79] B. Škorić, P. Tuyls, and W. Ophey. Robust key extraction from physical uncloneable functions. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 407–422, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[80] Alexander Spenke, Ralph Breithaupt, and Rainer Plaga. An arbiter puf secured by remote random reconfigurations of an fpga. In Michael Franz and Panos Papadimitratos, editors, *Trust and Trustworthy Computing*, pages 140–158, Cham, 2016. Springer International Publishing.

[81] Y. Su, J. Holleman, and B. Otis. A 1.6pj/bit 96variations. In *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pages 406–611, Feb 2007.

[82] G. E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *2007 44th ACM/IEEE Design Automation Conference*, pages 9–14, June 2007.

[83] S. Tajik, H. Lohrke, F. Ganji, J. Seifert, and C. Boit. Laser fault attack on physically unclonable functions. In *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 85–96, Sep. 2015.

[84] Shahin Tajik, Enrico Dietz, Sven Frohmann, Jean-Pierre Seifert, Dmitry Nedospasov, Clemens Helfmeier, Christian Boit, and Helmar Dittrich. Physical characterization of arbiter pufs. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 493–509, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[85] Mitsugu Iwamoto Takanori Machida, Dai Yamamoto and Kazuo Sakiyama. A new arbiter puf for enhancing unpredictability on fpga. *The Scientific World Journal*, 2015.

[86] Johannes Tobisch and Georg T. Becker. On the scaling of machine learning attacks on pufs with application to noise bifurcation. In Stefan Mangard and Patrick Schaumont, editors, *Radio Frequency Identification*, pages 17–31, Cham, 2015. Springer International Publishing.

[87] H. P. Tuinhout, A. H. Montree, J. Schmitz, and P. A. Stolk. Effects of gate depletion and boron penetration on matching of deep submicron cmos transistors. In *International Electron Devices Meeting. IEDM Technical Digest*, pages 631–634, Dec 1997.

[88] Pim Tuyls, Geert-Jan Schrijen, Boris Škorić, Jan van Geloven, Nynke Verhaegh, and Rob Wolters. Read-proof hardware from protective coatings. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, pages 369–383, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[89] P.T. Tuyls and B. Skoric. *Strong authentication with physical unclonable functions*, pages 133–148. Data-Centric Systems and Applications. Springer, Germany, 2007.

[90] M. E. Van Dijk. System and method of reliable forward secret key sharing with physical random functions, us patent 7653197, jan 2010.

[91] Akshay Wali, Akhil Dodda, Yang Wu, Andrew Pannone, Likhith Kumar Reddy Usthili, Sahin Ozdemir, Ibrahim Ozbolat, and Saptarshi Das. Biological physically unclonable function. *Communications Physics*, 2, 12 2019.

[92] He Xu, Jie Ding, Peng Li, Feng Zhu, and Ruchuan Wang. A lightweight rfid mutual authentication protocol based on physical unclonable function. *Sensors*, 18(3):760, Mar 2018.

[93] X. Xu and W. Burleson. Hybrid side-channel/machine-learning attacks on pufs: A new threat? In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014.

[94] Xiaolin Xu, Ulrich Rührmair, Daniel E. Holcomb, and Wayne Burleson. Security evaluation and enhancement of bistable ring pufs. In *Revised Selected Papers of the 11th International Workshop on Radio Frequency Identification - Volume 9440*, RFIDsec 2015, pages 3–16, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[95] Dai Yamamoto, Masahiko Takenaka, Kazuo Sakiyama, and Naoya Torii. Security evaluation of bistable ring pufs on fpgas using differential and linear analysis. In *2014 Federated Conference on Computer Science and Information Systems*, pages 911–918. IEEE, 2014.

[96] Risa Yashiro, Takanori Machida, Mitsugu Iwamoto, and Kazuo Sakiyama. Deep-learning-based security evaluation on authentication systems using arbiter puf and its variants. In Kazuto Ogawa and Katsunari Yoshioka, editors, *Advances in Information and Computer Security*, pages 267–285, Cham, 2016. Springer International Publishing.

[97] M. Yu, D. M'Raïhi, I. Verbauwhede, and S. Devadas. A noise bifurcation architecture for linear additive physical functions. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 124–129, 2014.

[98] Meng-Day Yu and Srinivas Devadas. Recombination of physical unclonable functions. In *35th Annual GOMACTech Conference*, pages 22–25, 2010.

[99] Feng Zhu, Peng Li, He Xu, and Ruchuan Wang. A lightweight rfid mutual authentication protocol with puf. *Sensors*, 19:2957, 07 2019.

APPENDICES

Appendix A

Steps for manual placement and routing of PUFs on FPGA

A.1 Manual placement of PUF stages

As was mentioned in section 3.3 of chapter 3, a repetitive placement technique was used to determine the best configuration that allows manual routing with smaller delay difference. Please note that FPGA architecture are repetitive by nature, hence, the connections among stages tend to repeat itself. Therefore, there is a constant set of placement cases that need to be accounted for. Every stage of the 64-stages PUF will use one of these limited cases. For every placement case, we manually try alter the stage position on the FPGA layout and check the routing options to determine the best configuration that provides better delay difference results. Figure A.1 shows PUF 8 after doing the repetitive manual placement compared to other PUFs. The stopping condition was to find a routing configuration that reduces delay difference to be < 50 ps. Figures A.2, A.3, and A.4 show examples of routed stages with delay differences 32ps, 5ps, and 0ps respectively.

The function of every path in a specific stage can be realized using 3-input lookup table (LUT). Hence, every stage can be implemented using two 3-input LUTs, however, we found that placing the two LUTs into one 6-input LUT as shown in Figure A.5 eases the manual routing task.

Finally, We found that paths inside every LUT will have different delays as shown in Figure A.6, which cannot be modified or controlled. Therefore, we interchanged the placement of both paths LUTs in half the number of stages. Hence, every path will go

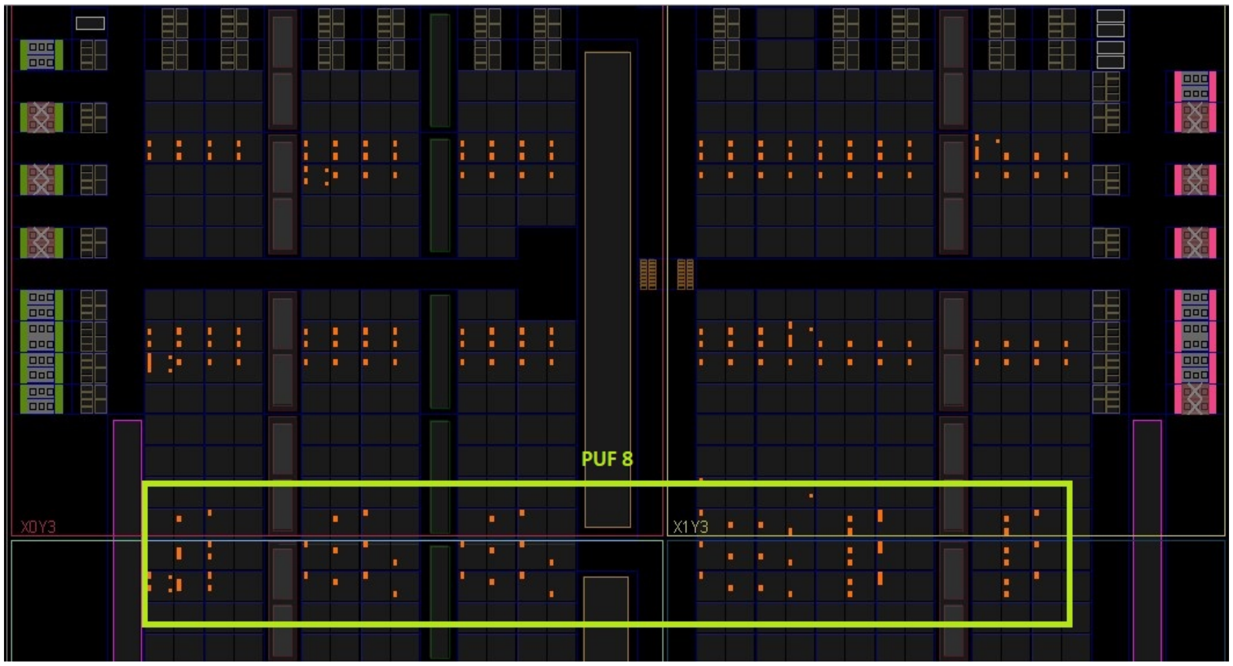
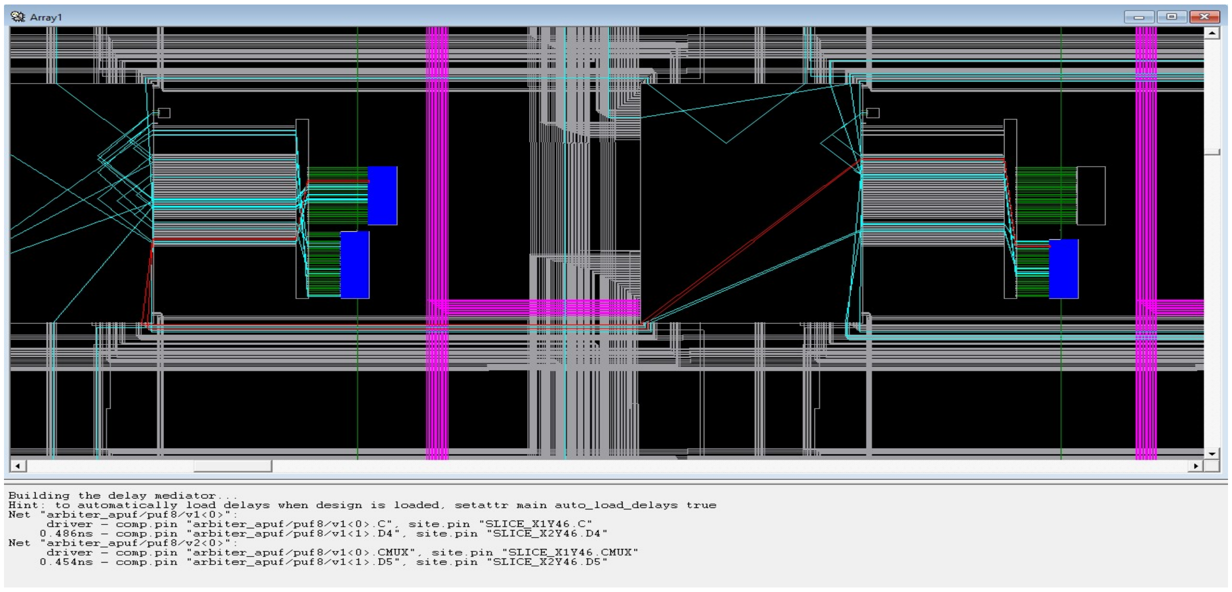
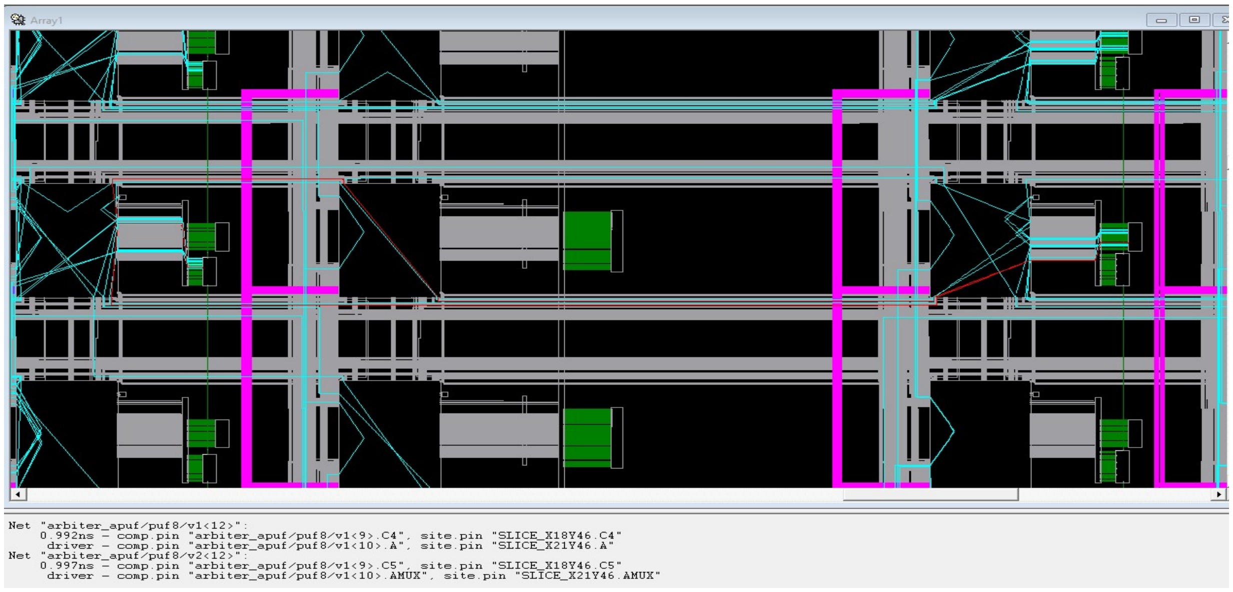


Figure A.1: Repetitive manual placement of PUF 8



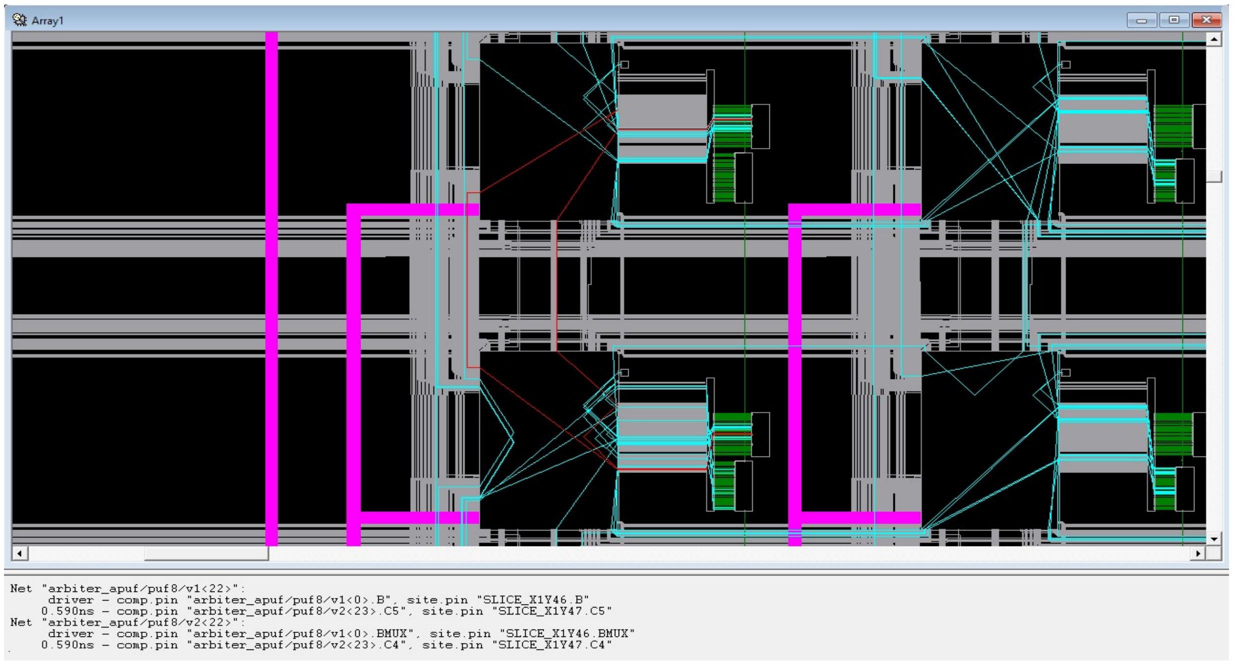
```
Building the delay mediator...
Hint: to automatically load delays when design is loaded, setattr main auto_load_delays true
Net "arbiter_apuf/puf8/v1<0>":
     driver - comp.pin "arbiter_apuf/puf8/v1<0>.C", site.pin "SLICE_X1Y46.C"
    0.486ns - comp.pin "arbiter_apuf/puf8/v1<1>.D4", site.pin "SLICE_X2Y46.D4"
Net "arbiter_apuf/puf8/v2<0>":
     driver - comp.pin "arbiter_apuf/puf8/v1<0>.CMUX", site.pin "SLICE_X1Y46.CMUX"
    0.454ns - comp.pin "arbiter_apuf/puf8/v1<1>.D5", site.pin "SLICE_X2Y46.D5"
```

Figure A.2: Delay difference example of stage 0



```
Net "arbiter_apuf/puf8/v1<12>":
    0.992ns - comp.pin "arbiter_apuf/puf8/v1<9>.C4", site.pin "SLICE_X18Y46.C4"
     driver - comp.pin "arbiter_apuf/puf8/v1<10>.A", site.pin "SLICE_X21Y46.A"
Net "arbiter_apuf/puf8/v2<12>":
    0.997ns - comp.pin "arbiter_apuf/puf8/v1<9>.C5", site.pin "SLICE_X18Y46.C5"
     driver - comp.pin "arbiter_apuf/puf8/v1<10>.AMUX", site.pin "SLICE_X21Y46.AMUX"
```

Figure A.3: Delay difference example of stage 12



```
Net "arbiter_apuf/puf8/v1<22>":
     driver - comp.pin "arbiter_apuf/puf8/v1<0>.B", site.pin "SLICE_X1Y46.B"
    0.590ns - comp.pin "arbiter_apuf/puf8/v2<23>.C5", site.pin "SLICE_X1Y47.C5"
Net "arbiter_apuf/puf8/v2<22>":
     driver - comp.pin "arbiter_apuf/puf8/v1<0>.BMUX", site.pin "SLICE_X1Y46.BMUX"
    0.590ns - comp.pin "arbiter_apuf/puf8/v2<23>.C4", site.pin "SLICE_X1Y47.C4"
```

Figure A.4: Delay difference example of stage 22

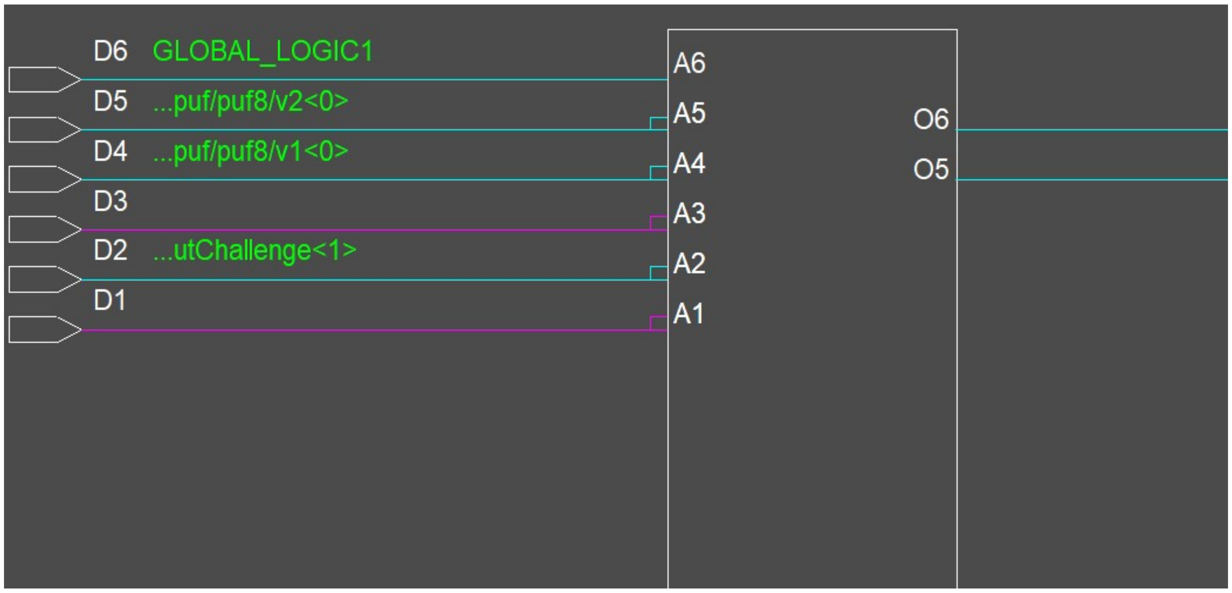


Figure A.5: Two 3-input LUTs placed into one 6-input LUT

through O5 and O6 for the same number of times to balance the delay difference inside the LUTs

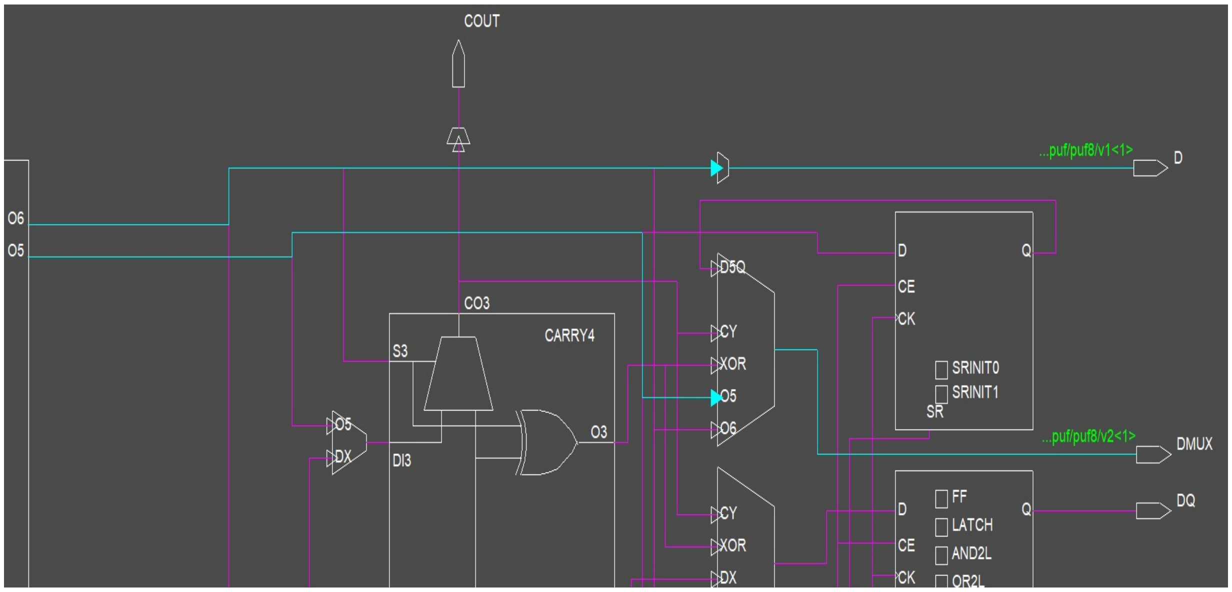


Figure A.6: The delay difference between the 6-input LUT outputs

A.2 Manual routing of PUFs inter-stage connections

The steps of manual routing are as follows:

- The manual routing step is done post mapping to have all routing resources free and give priority to the PUFs components as shown in Figure A.7.
- For every stage, both paths are automatically routed with routing option configured to be ”delay driven” as shown in Figure A.8. Hence, we get the shortest routing delay for both paths.
- We cannot optimize the delay of both paths because the delay-driven routing will get the shortest delay for every path. Hence, we will pick the path with shortest delay and try other routing options to make it longer and closer to the other path delay and reduce overall delay difference.
- Record the steps of the manual routing into a script to repeat it with other stages that have the same placement.
- After finishing the manual routing, the overall routing step is executed with option ”Reentrant Route” as shown in Figure A.9 to consider the modifications done in our previous steps.

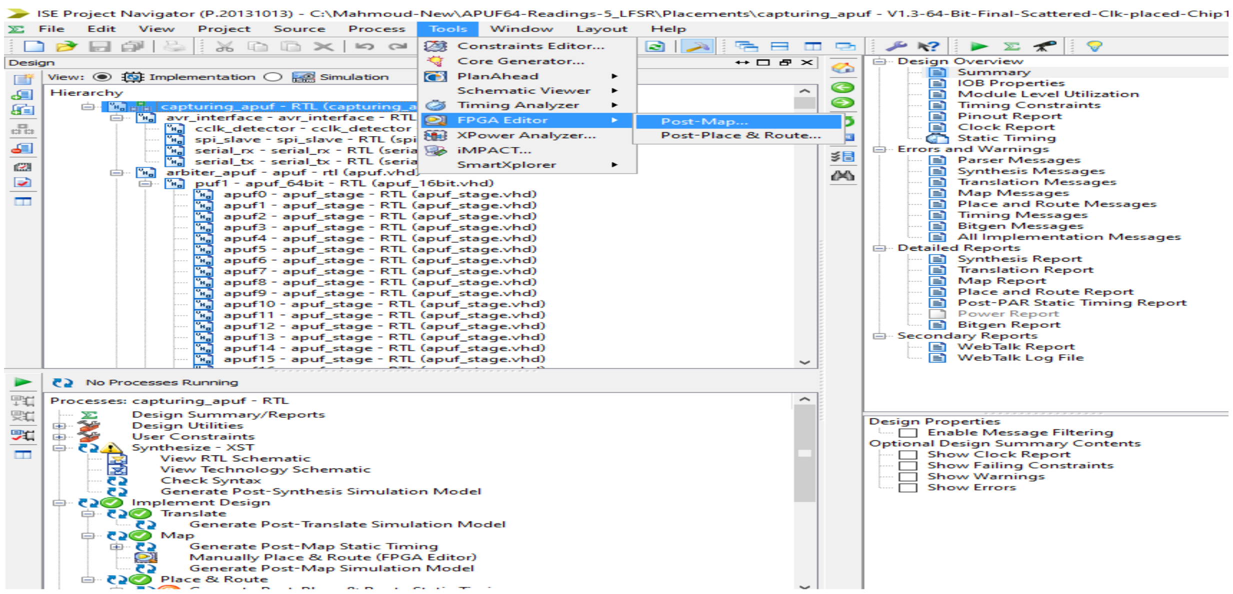
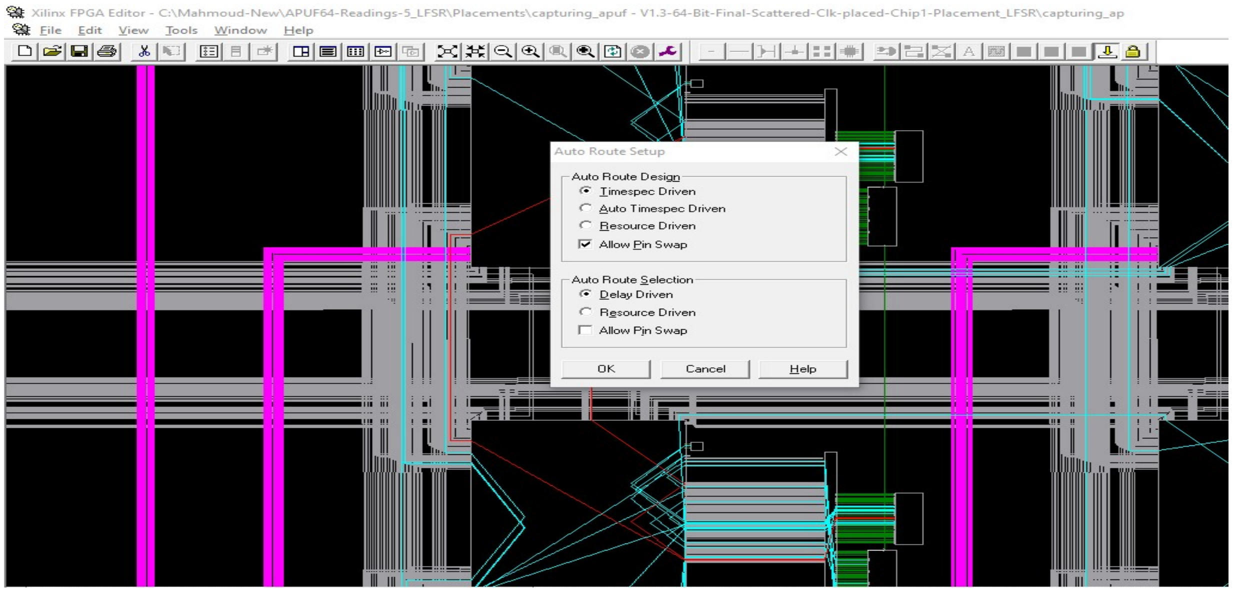

ISE Project Navigator (P.20131013) - C:\Mahmoud-New\APUF64-Readings-5_LFSR\Placements\capturing_apuf - V1.3-64-Bit-Final-Scattered-Clk-placed-Chip1
File Edit View Project Source Process Tools Window Layout Help
Constraints Editor...
Core Generator...
PlanAhead
Schematic Viewer
Timing Analyzer
FPGA Editor
XPower Analyzer...
iMPACT...
SmartXplorer
Post-Map...
Post-Place & Route...
Design
View: Implementation Simulation
Hierarchy
capturing_apuf - RTL (capturing_a
avr_interface - avr_interface - RTL
cclk_detector - cclk_detector
spi_slave - spi_slave - RTL (spi
serial_rx - serial_rx - RTL (seria
serial_tx - serial_tx - RTL (seria
arbiter_apuf - apuf - rtl (apuf.vhd)
puf1 - apuf_64bit - RTL (apuf_16bit.vhd)
apuf0 - apuf_stage - RTL (apuf_stage.vhd)
apuf1 - apuf_stage - RTL (apuf_stage.vhd)
apuf2 - apuf_stage - RTL (apuf_stage.vhd)
apuf3 - apuf_stage - RTL (apuf_stage.vhd)
apuf4 - apuf_stage - RTL (apuf_stage.vhd)
apuf5 - apuf_stage - RTL (apuf_stage.vhd)
apuf6 - apuf_stage - RTL (apuf_stage.vhd)
apuf7 - apuf_stage - RTL (apuf_stage.vhd)
apuf8 - apuf_stage - RTL (apuf_stage.vhd)
apuf9 - apuf_stage - RTL (apuf_stage.vhd)
apuf10 - apuf_stage - RTL (apuf_stage.vhd)
apuf11 - apuf_stage - RTL (apuf_stage.vhd)
apuf12 - apuf_stage - RTL (apuf_stage.vhd)
apuf13 - apuf_stage - RTL (apuf_stage.vhd)
apuf14 - apuf_stage - RTL (apuf_stage.vhd)
apuf15 - apuf_stage - RTL (apuf_stage.vhd)
No Processes Running
Processes: capturing_apuf - RTL
Design Summary/Reports
Design Utilities
User Constraints
Synthesize - XST
View RTL Schematic
View Technology Schematic
Check Syntax
Generate Post-Synthesis Simulation Model
Implement Design
Translate
Generate Post-Translate Simulation Model
Map
Generate Post-Map Static Timing
Manually Place & Route (FPGA Editor)
Generate Post-Map Simulation Model
Place & Route
Design Overview
Summary
IOB Properties
Module Level Utilization
Timing Constraints
Pinout Report
Clock Report
Static Timing
Errors and Warnings
Parser Messages
Synthesis Messages
Translation Messages
Map Messages
Place and Route Messages
Timing Messages
Bitgen Messages
All Implementation Messages
Detailed Reports
Synthesis Report
Translation Report
Map Report
Place and Route Report
Post-PAR Static Timing Report
Power Report
Bitgen Report
Secondary Reports
WebTalk Report
WebTalk Log File
Design Properties
Enable Message Filtering
Optional Design Summary Contents
Show Clock Report
Show Failing Constraints
Show Warnings
Show Errors

Figure A.7: Manual routing is done post mapping

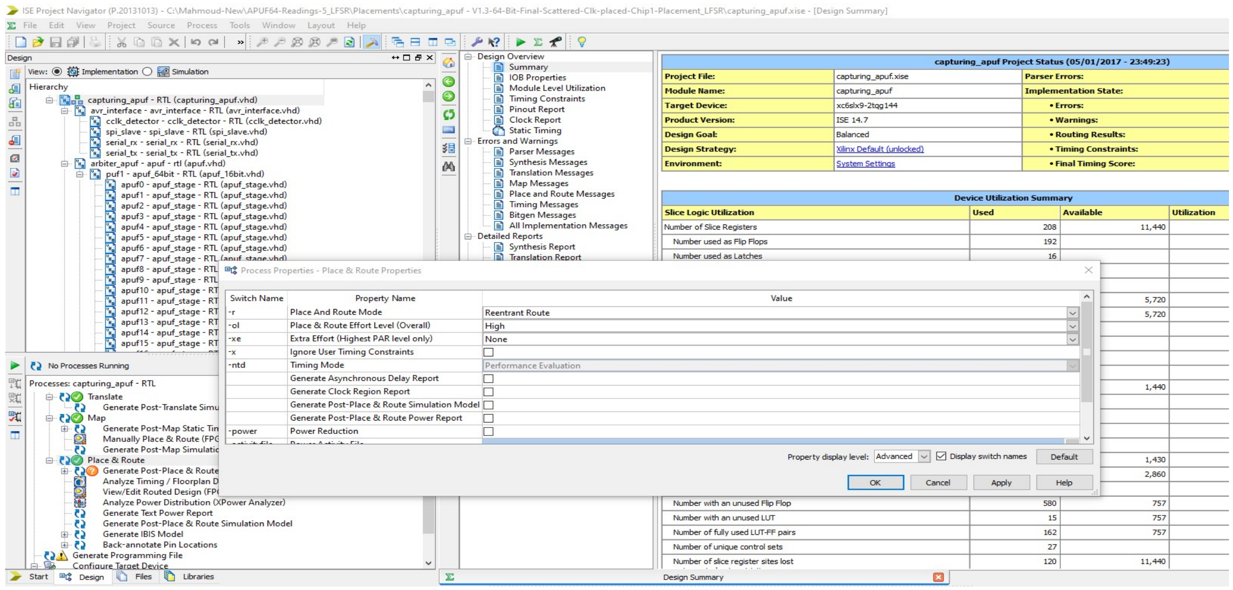

Xilinx FPGA Editor - C:\Mahmoud-New\APUF64-Readings-5_LFSR\Placements\capturing_apuf - V1.3-64-Bit-Final-Scattered-Clk-placed-Chip1-Placement_LFSR\capturing_ap
File Edit View Tools Window Help
OK
Cancel
Help

Auto Route Setup

Auto Route Design
- [✓ Timespec Driven]
- Auto Timespec Driven
- Resource Driven
- [✓ Allow Pin Swap]

Auto Route Selection
- [✓ Delay Driven]
- Resource Driven
- Allow Pin Swap

Figure A.8: Automatic routing is configured to be delay driven



ISE Project Navigator (P.20131013) - C:\Mahmoud-New\APUF64-Readings-5_LFSR\Placements\capturing_apuf - V1.3-64-Bit-Final-Scattered-Clk-placed-Chip1-Placement_LFSR\capturing_apuf.xise - [Design Summary]

capturing_apuf Project Status (05/01/2017 - 23:49:23)		
Project File:	capturing_apuf.xise	**Parser Errors:**
Module Name:	capturing_apuf	**Implementation State:**
Target Device:	xc6slx9-2tqg144	• **Errors:**
Product Version:	ISE 14.7	• **Warnings:**
Design Goal:	Balanced	• **Routing Results:**
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**
Environment:	System Settings	• **Final Timing Score:**

Process Properties - Place & Route Properties

Switch Name	Property Name	Value
-r	Place And Route Mode	Reentrant Route
-ol	Place & Route Effort Level (Overall)	High
-xe	Extra Effort (Highest PAR level only)	None
-x	Ignore User Timing Constraints	☐
-ntd	Timing Mode	Performance Evaluation
	Generate Asynchronous Delay Report	☐
	Generate Clock Region Report	☐
	Generate Post-Place & Route Simulation Model	☐
	Generate Post-Place & Route Power Report	☐
-power	Power Reduction	☐

Property display level: Advanced ☑ Display switch names

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	208	11,440	
Number used as Flip Flops	192		
Number used as Latches	16		
Number with an unused Flip Flop	580	757	
Number with an unused LUT	15	757	
Number of fully used LUT-FF pairs	162	757	
Number of unique control sets	27		
Number of slice register sites lost	120	11,440	

File Edit View Project Source Process Tools Window Layout Help
Design
View: Implementation Simulation
Hierarchy
capturing_apuf - RTL (capturing_apuf.vhd)
avr_interface - avr_interface - RTL (avr_interface.vhd)
cclk_detector - cclk_detector - RTL (cclk_detector.vhd)
spi_slave - spi_slave - RTL (spi_slave.vhd)
serial_rx - serial_rx - RTL (serial_rx.vhd)
serial_tx - serial_tx - RTL (serial_tx.vhd)
arbiter_apuf - apuf - rtl (apuf.vhd)
puf1 - apuf_64bit - RTL (apuf_16bit.vhd)
apuf0 - apuf_stage - RTL (apuf_stage.vhd)
No Processes Running
Processes: capturing_apuf - RTL
Translate
Map
Place & Route
Generate Programming File
Design Overview
Summary
IOB Properties
Module Level Utilization
Timing Constraints
Pinout Report
Clock Report
Static Timing
Errors and Warnings
Parser Messages
Synthesis Messages
Translation Messages
Map Messages
Place and Route Messages
Timing Messages
Bitgen Messages
All Implementation Messages
Detailed Reports
Synthesis Report
Translation Report
Default
OK
Cancel
Apply
Help
Start
Design
Files
Libraries
Design Summary

Figure A.9: Automatic routing is configured to be delay driven

Appendix B

Python Scripts used to calculate statistical metrics

Providing code snippets for the statistical metrics used in the dissertation.

B.1 Majority counting

```
def PrepareCRPData(startIdx , endIdx, CRPData, ChallengeLength ,
    ResponseLength, procnum):
    FormattedCrpData = list()
    for i in range(startIdx, endIdx, 1):
        temp = CRPData[i].strip().replace("\n","").split(" ")
        Challenge = temp[0].split("0b")
        Response = temp[len(temp) - 1].replace("0b","")
        FormattedResp = ""
        for j in range(1,9,1):
            FormattedResp += Challenge[j].zfill(ChallengeLength)
        FormattedResp += " " + Response.zfill(ResponseLength) + "\n"
        FormattedCrpData.append(FormattedResp)
    outFile = open("CRPs-Read-64Bit-Chip1-Modified-" + str(procnum) + ".txt"
    , "w")
    for line in FormattedCrpData:
        outFile.write(line)
    outFile.flush()
    outFile.close()


```

```
def MajorityCounting(startIdx , endIdx, CRPData1, CRPData2, CRPData3,
    CRPData4, CRPData5, CRPData6, CRPData7, CRPData8, CRPData9, CRPData10,
    CRPData11, ResponseLength, AnalysisStat, procnum):
     MajorityCRPData = list()
     for i in range(startIdx, endIdx, 1):
         temp1 = CRPData1[i].replace("\n","").split(" ")
         temp2 = CRPData2[i].replace("\n","").split(" ")
         temp3 = CRPData3[i].replace("\n","").split(" ")
         temp4 = CRPData4[i].replace("\n","").split(" ")
         temp5 = CRPData5[i].replace("\n","").split(" ")
         temp6 = CRPData6[i].replace("\n","").split(" ")
         temp7 = CRPData7[i].replace("\n","").split(" ")
         temp8 = CRPData8[i].replace("\n","").split(" ")
         temp9 = CRPData9[i].replace("\n","").split(" ")
         temp10 = CRPData10[i].replace("\n","").split(" ")
         temp11 = CRPData11[i].replace("\n","").split(" ")
         MajorityResp = temp1[0] + " "
         for j in range(0,ResponseLength,1):
             count = int(temp1[1][j]) + int(temp2[1][j]) + int(temp3[1][j]) +
      int(temp4[1][j]) + int(temp5[1][j]) + int(temp6[1][j]) + int(temp7[1][j
    ]) + int(temp8[1][j]) + int(temp9[1][j]) + int(temp10[1][j]) + int(temp11
    [1][j])
             MajorityResp += str(int(count/6))
             if(count>= 6):
                 AnalysisStat[count - 6] = AnalysisStat[count -6] + 1
             else:
                 AnalysisStat[11 - count - 6] = AnalysisStat[11 - count -6] +
      1
         MajorityResp += "\n"
         MajorityCRPData.append(MajorityResp)
     outFile = open("CRPs-Read-64Bit-Chip1-Majority-" + str(procnum) + ".txt"
      , "w")
     for line in MajorityCRPData:
         outFile.write(line)
     outFile.flush()
     outFile.close()

if __name__ == '__main__':
     for i in range(0,11,1):
         CRPmodif = open("CRPs-Read-64Bit-Chip1-"+str(i+1)+"-Modified.txt", "
    w")
         crpData = CRPorig[i].readlines()
         ctx = multiprocessing.get_context('spawn')
```

```
        p1 = ctx.Process(target=PrepareCRPData, args=(0,250000,crpData,8,8,
    1))
        p2 = ctx.Process(target=PrepareCRPData, args=(250000,500000, crpData
    ,8,8, 2))
        p3 = ctx.Process(target=PrepareCRPData, args=(500000,750000, crpData
    ,8,8, 3))
        p4 = ctx.Process(target=PrepareCRPData, args=(750000,1000000,
    crpData,8,8, 4))
        p1.start()
        p2.start()
        p3.start()
        p4.start()
        p1.join()
        p2.join()
        p3.join()
        p4.join()
        res1 = open("CRPs-Read-64Bit-Chip1-Modified-1.txt" ,"r").readlines()
        res2 = open("CRPs-Read-64Bit-Chip1-Modified-2.txt" ,"r").readlines()
        res3 = open("CRPs-Read-64Bit-Chip1-Modified-3.txt" ,"r").readlines()
        res4 = open("CRPs-Read-64Bit-Chip1-Modified-4.txt" ,"r").readlines()
        for j in range(len(res1)):
            CRPmodif.write(res1[j])
        for j in range(len(res2)):
            CRPmodif.write(res2[j])
        for j in range(len(res3)):
            CRPmodif.write(res3[j])
        for j in range(len(res4)):
            CRPmodif.write(res4[j])
        CRPmodif.flush()
        CRPmodif.close()
        CRPorig[i].close()

## Do majority vote
    CRPData1 = open("CRPs-Read-64Bit-Chip1-1-Modified.txt","r").readlines()
    CRPData2 = open("CRPs-Read-64Bit-Chip1-2-Modified.txt","r").readlines()
    CRPData3 = open("CRPs-Read-64Bit-Chip1-3-Modified.txt","r").readlines()
    CRPData4 = open("CRPs-Read-64Bit-Chip1-4-Modified.txt","r").readlines()
    CRPData5 = open("CRPs-Read-64Bit-Chip1-5-Modified.txt","r").readlines()
    CRPData6 = open("CRPs-Read-64Bit-Chip1-6-Modified.txt","r").readlines()
    CRPData7 = open("CRPs-Read-64Bit-Chip1-7-Modified.txt","r").readlines()
    CRPData8 = open("CRPs-Read-64Bit-Chip1-8-Modified.txt","r").readlines()
    CRPData9 = open("CRPs-Read-64Bit-Chip1-9-Modified.txt","r").readlines()
    CRPData10 = open("CRPs-Read-64Bit-Chip1-10-Modified.txt","r").readlines
    ()
    CRPData11 = open("CRPs-Read-64Bit-Chip1-11-Modified.txt","r").readlines
    ()
    majCounter1 = [0,0,0,0,0,0]
    MajorityCounting(0,1000000,CRPData1, CRPData2, CRPData3, CRPData4,
    CRPData5, CRPData6, CRPData7, CRPData8, CRPData9, CRPData10, CRPData11,
    8, majCounter1,1)
    res1 = open("CRPs-Read-64Bit-Chip1-Majority-1.txt" ,"r").readlines()
    Majout = open("CRPs-Read-64Bit-Chip1-Majority.txt" ,"w")
    for  j in range(len(res1)):
        Majout.write(res1[j])
    Majout.flush()
    Majout.close()
    print(majCounter1)
```

Listing B.1: Majority counting calaculation script

B.2 Inter-Chip hamming distance

```
def CalculateHD(PUFRes , startIDX , HDRes):
    for pufline in range (startIDX , len(PUFRes) , 4):
        PUF = PUFRes[pufline].replace("\n", "").split(" ")
    #print(PUF[1])
    #print(PUF[1][0])
        for pufline2 in range ( pufline + 1 , len(PUFRes) , 1):
            counter = 0
            bitPointer = 0
            TempPUF = PUFRes[pufline2].replace("\n", "").split(" ")
            if(PUF[1] == TempPUF[1]):
                HDRes[0] = HDRes[0] + 1
                continue
            for bitPointer in range (0,8,1):
                if(PUF[1][bitPointer] == TempPUF[1][bitPointer]):
                    continue
                else:
                    counter = counter + 1

            HDRes[counter] = HDRes[counter] + 1

        print("finished " + str(pufline) + "\n")

if __name__ == '__main__':

    PUFRes = open("CRPs-Read-64Bit-Chip1-Majority.txt" ,"r").readlines()
    HDCounter1 = multiprocessing.Array('i', range(9))
    HDCounter2 = multiprocessing.Array('i', range(9))
    HDCounter3 = multiprocessing.Array('i', range(9))
    HDCounter4 = multiprocessing.Array('i', range(9))
  #AutoCor = multiprocessing.Array('f', range(32768))
    ctx = multiprocessing.get_context('spawn')
        #multiprocessing.set_start_method('spawn')

    p1 = ctx.Process(target=CalculateInterHD, args=(P1File1,P1File2,P1File3,P1File4,P1File5,HDCounter1))
    p2 = ctx.Process(target=CalculateInterHD, args=(P2File1,P2File2,P2File3,P2File4,P2File5, HDCounter2))
    p3 = ctx.Process(target=CalculateInterHD, args=(P3File1,P3File2,P3File3,P3File4,P3File5, HDCounter3))
    p4 = ctx.Process(target=CalculateInterHD, args=(P4File1,P4File2,P4File3,P4File4,P4File5, HDCounter4))
    p1.start()
    p2.start()
    p3.start()
    p4.start()
    p1.join()
    p2.join()
    p3.join()
    p4.join()

    for i in range(0,len(HDCounter),1):
        HDCounter[i] = HDCounter1[i] + HDCounter2[i] + HDCounter3[i] + HDCounter4[i]
    print(HDCounter)
```

Listing B.2: Hamming distance calaculation script

B.3 Phi correlation

```
import sklearn.metrics as sk

infile = open("CRPs-Read-64Bit-Chip1-Majority.txt", "r").readlines()
outfile = open("corrOut.txt", "w")
ChallengeCorr = list()
RespCorr = list()

for i in range(8):
    for j in range(64):
        challenge = list()
        resp = list()

        for line in infile:
            temp = line.replace("\n","").split(" ")
            challenge.append(int(temp[0][j]) )
            resp.append(int(temp[1][i]) )

        corr = sk.matthews_corrcoef(challenge, resp)
        #print(corr)
        ChallengeCorr.append(corr)
        del challenge
        del resp
    print(str(i) + " is finished")

for i in range(8):
    for j in range(8):
        Resp1 = list()
        Resp2 = list()

        for line in infile:
            temp = line.replace("\n","").split(" ")
            Resp1.append(int(temp[1][i]) )
            Resp2.append(int(temp[1][j]) )

        corr = sk.matthews_corrcoef(Resp1, Resp2)
        #print(corr)
        RespCorr.append(corr)
        del Resp1
        del Resp2

for i in range(8):
    strout = "PUF Resp" + str(8 - i)+ " Challenge bits Corr:\n"
    for j in range(64):
        strout +=  str(ChallengeCorr[(64*i) + j]) + ","
    strout += "\n"
    outfile.write(strout)

outfile.write("=========================================================\n")

for i in range(8):
    strout = "PUF Resp" + str(8 - i)+ " PUF Resps Corr:\n"
    for j in range(8):
        strout += str(RespCorr[(8*i) + j]) + ", "
    strout += "\n"
    outfile.write(strout)

outfile.flush()
outfile.close()
```

Listing B.3: Phi correlation calaculation script

Appendix C

Python Scripts used for DL modeling

The script used for training a PUF model.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
import tensorflow as tf
import random

tf.logging.set_verbosity(tf.logging.INFO)


def cnn_model_fn(features, labels, mode):
  """Model function for CNN."""
  # Input Layer
  # Reshape X to 4-D tensor: [batch_size, width, height, channels]
  input_layer = tf.reshape(features["x"], [-1, 65])

  num_neurons = 2000

  FC_1 = tf.layers.dense(inputs=input_layer, units=num_neurons, activation=
    tf.nn.tanh)
  FC_2 = tf.layers.dense(inputs=FC_1, units=num_neurons,activation=tf.nn.
    tanh)
  FC_3 = tf.layers.dense(inputs=FC_2, units=num_neurons,activation=tf.nn.
    tanh)
  FC_4 = tf.layers.dense(inputs=FC_3, units=num_neurons,activation=tf.nn.
    tanh)
  FC_5 = tf.layers.dense(inputs=FC_4, units=num_neurons,activation=tf.nn.
    relu)
  FC_6 = tf.layers.dense(inputs=FC_5, units=num_neurons,activation=tf.nn.
    relu)
  FC_7 = tf.layers.dense(inputs=FC_6, units=num_neurons,activation=tf.nn.
    relu)
  FC_8 = tf.layers.dense(inputs=FC_7, units=num_neurons,activation=tf.nn.
    relu)
  FC_9 = tf.layers.dense(inputs=FC_8, units=num_neurons,activation=tf.nn.
    relu)
  FC_10 = tf.layers.dense(inputs=FC_9, units=num_neurons,activation=tf.nn.
    relu)
  FC_11 = tf.layers.dense(inputs=FC_10, units=num_neurons,activation=tf.nn.
    relu)
  FC_12 = tf.layers.dense(inputs=FC_11, units=num_neurons,activation=tf.nn.
    relu)


  # Add dropout operation; 0.8 probability that element will be kept
  dropout = tf.layers.dropout(
      inputs=FC_12, rate=0.2, training=mode == tf.estimator.ModeKeys.TRAIN)

  # Logits layer
  logits = tf.layers.dense(inputs=dropout, units=2,activation=tf.nn.tanh)

  predictions = {
      # Generate predictions (for PREDICT and EVAL mode)
      "classes": tf.argmax(input=logits, axis=1),
      # Add `softmax_tensor` to the graph. It is used for PREDICT and by the
      # `logging_hook`.
      "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
  }
  if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

  # Calculate Loss (for both TRAIN and EVAL modes)
  loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits
    )

  # Configure the Training Op (for TRAIN mode)
  if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.AdamOptimizer(learning_rate=0.0001)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    train_metric_ops = {
      "train_accuracy": tf.metrics.accuracy(
          labels=labels, predictions=predictions["classes"])}
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

  # Add evaluation metrics (for EVAL mode)
  eval_metric_ops = {
      "accuracy": tf.metrics.accuracy(
          labels=labels, predictions=predictions["classes"])}
  return tf.estimator.EstimatorSpec(
      mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)


def main(unused_argv):
  # Load training and eval data
  xor_num = 1  # number of input XOR
  num_features = 65 # number of features = number of PUF stages + 1
  SetSize = 990000
  TestSetSize = 90000 # number of test set CRPs
  TrainSetSize = 900000 # number of train set CRPs
  trainDataPos = list() # A list to hold random training indices

  input_arr = np.empty([TrainSetSize,xor_num*num_features],dtype=np.float32) # the training set challenge bit features
  target_arr = np.empty(TrainSetSize,dtype=np.int32) # The outcome of challenge cases in training set

  test_arr = np.empty([TestSetSize,xor_num*num_features],dtype=np.float32) # The testing set challenge bit features
  test_target_arr = np.empty(TestSetSize,dtype=np.int32) # The outcome of challenge cases in test set

  infile = open("CRPs-Read-64Bit-Chip6-Majority-Features.txt", "r").readlines() # file containing all the 1M Challenge features
  infile2 = open("CRPs-Read-64Bit-Chip6-Majority.txt", "r").readlines() # A file contains the responses of the PUF to the same 1M challenges



  for i in range(SetSize):
      tempstr = infile[i]
      temp = tempstr.replace('\n','').split()
      for k in range(xor_num):
```

```python
        for j in range(len(temp)):
            if( i < TrainSetSize):
                input_arr[i][k*num_features + j] = float(temp[j])
            else:
                test_arr[i - TrainSetSize][k*num_features + j] = float(temp[j])
    if( i % 100000 == 0):
        print('finished ' + str(i/10000)+ '%\n')
for i in range(SetSize):
    temp2 = infile2[i].replace('\n','').split()
    if( i < TrainSetSize):
        target_arr[i] = int(temp2[1][7])
    else:
        test_target_arr[i - TrainSetSize] = int(temp2[1][7])

print('Finished train dataset')
train_data = input_arr
train_labels = target_arr
print(train_labels.shape)
eval_data = test_arr
eval_labels = test_target_arr
my_checkpointing_config = tf.estimator.RunConfig(
  save_checkpoints_steps =6000,  # Save checkpoints every 6000 steps.
  keep_checkpoint_max = 2,       # Retain the 10 most recent checkpoints.
)
puf_classifier = tf.estimator.Estimator(
    model_fn=cnn_model_fn, model_dir="./PUF-Model", config=my_checkpointing_config)

# Set up logging for predictions
# Log the values in the "Softmax" tensor with label "probabilities"
tensors_to_log = {"probabilities": "softmax_tensor"}
logging_hook = tf.train.LoggingTensorHook(
    tensors=tensors_to_log, every_n_iter=1000000)

# Train the model
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": train_data},
    y=train_labels,
    batch_size=200,
    num_epochs=None,
    shuffle=True)
eval_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": eval_data},
    y=eval_labels,
        num_epochs=1,
        shuffle=False)

  for i in range (20):
        puf_classifier.train(
            input_fn=train_input_fn ,
            steps=6000,
            hooks=[logging_hook])
        # Evaluate the model and print results
        eval_results = puf_classifier.evaluate(input_fn=eval_input_fn)
        print(eval_results)








if __name__ == "__main__":
  tf.app.run()
```

Listing C.1: DL training script

Appendix D

Python Scripts used to generate multiplexer parameters of the N-to-1 Shuffled-Challenge Hierarchical XOR BR-PUF

```python
import random as rd
import numpy as np
import collections
import math


#These are the  two paramters for the script. Changing them will change
    everything#

#how many selector bits are required
mux_sel = 4

#The number of PUF stages
num_stage = 64
################################################

#Every element of mask_arr is the 2^mux_sel input binary string that has 50%
     '1' and 50% '0'
# for example if we are using 2-bit mux selector, then the mux input will be
     4 bits and there are 6 cases at which there are exactly two 1's and two
    0's
```

```
mask_arr = []

# The number of elements in challengeBits_select_pos = mux_sel elements.
    Each element is an array that contain the challenge indices that affect
    the output modified Chalenge at the corresponding array position
#for example challengeBits_select_pos[0] = [1,2,3,4,5],
    challengeBits_select_pos[1] = [6,7,8,9,10] then challenge bit 1 and 6 are
     the select bits for the mux that affects modified challenge bit 0, ..etc
    .
challengeBits_select_pos = []

#original challenge
challenge = np.random.randint(2, size=num_stage)


for i in range(mux_sel):
    challenge1 = np.arange(num_stage)
    np.random.shuffle(challenge1)
    challengeBits_select_pos.append(challenge1)

########################## important ####################################
## This code is commented and can be used to make sure that no challenge bit
     is used more than once at every mux select input.
#########################################################################

##print(challengeBits_select_pos)
##for i in range(64):
##    if(challengeBits_select_pos[0][i] == challengeBits_select_pos[1][i] or
     challengeBits_select_pos[0][i] == challengeBits_select_pos[2][i] or
    challengeBits_select_pos[2][i] == challengeBits_select_pos[1][i]):
##        print(i)
##        print(challengeBits_select_pos[0][i])
##        print(challengeBits_select_pos[1][i])
##        print(challengeBits_select_pos[2][i])



# determining which bit will take which mask configurations
# The number of mask configurations that has 50% 0's and 1's are determined
    by n! / ( n-r! * r!) = 2^mux_sel / (2^mux_sel-1)^2
mask_distribution = np.random.randint(int(math.factorial(2**mux_sel)/math.
    factorial(2**(mux_sel-1))**2), size=num_stage)

#loop over over all possible binary values of a length 2^mux_sel
for i in range(2**(2**mux_sel)):
```

```python
    #check if this value contains 50% 1's and 50% 0's
    if(bin(i).count('1') == (2**(mux_sel-1))):
        value = str(bin(i)).replace("0b","").zfill(2**mux_sel)
        mask = []
        for j in range(len(value)):
            mask.append(int(value[j]))
        mask_arr.append(mask)





#the target challenge after applying masking to the original LFSR challenge
challenge_modified = np.random.randint(2, size = num_stage)

#counter to know how many bits are different in the generated challenge from the original challenge
count = 0

#for loop to generate the new challenge
for i in range(len(challenge_modified)):
    index = 0
    # Every modified challenge bit depends on mux_sel challenge bits from the original challenge
    # the original challenge bits at challengeBits_select_pos[k][i] determine challenge bit i of modified challenge
    # For example of mux_sel = 2, it is like a mux with select is challenge2 challenge1 = b1b0  that's why we multiply by 2^0 and 2^1 to determine the index inside the mask
    for k in range(mux_sel):
        index += challenge[challengeBits_select_pos[k][i]] * (2**k)

    #choosing value from specific mask configuration depending on the challenge value at this poisition i
    challenge_modified[i] = mask_arr[mask_distribution[i]][index]
    if(challenge_modified[i] != challenge[i]):
        count += 1

print(collections.Counter(mask_distribution))
print(collections.Counter(challenge))
print(collections.Counter(challenge_modified))
print(mask_distribution)
print(challenge)
print(challenge_modified)
print(count)
```

Listing D.1: N-to-1 Shuffled-Challenge Hierarchical PUF parameters generation script