FPGA Implementation and Evaluation of an Arbitrary Physical Unclonable Function

Μπίνας Βασίλειος

Διπλωματική Εργασία

Επιβλέπων: Βασίλειος Τενέντες

Ιωάννινα, <Μήνας>, <2025>

ΤΜΗΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Acknowledgements

With the completion of my thesis, I would like to give my thanks to everyone that stood by my side and guided me in the span of this thesis making.

First I want to thank my professor Vasileios Tenentes, for his invaluable guidance and the advice. His support was instrumental in the completion of this work.

Lastly I would like to extend my heartfelt appreciation to my family and friends for their unwavering encouragement and support during this process.

Best regards,

Vasileios Binas

Περίληψη

Αυτή η διπλωματική εργασία εξετάζει τον σχεδιασμό, την υλοποίηση και την αξιολόγηση ενός Arbiter Physical Unclonable Function (PUF) σε FPGA ως μια πρωτογενή μέθοδο ασφάλειας υλικού. Τα PUF αξιοποιούν τις εγγενείς μεταβολές της κατασκευαστικής διαδικασίας για τη δημιουργία μοναδικών και μη προβλέψιμων ζευγών πρόκλησης-απόκρισης (CRP), καθιστώντας τις χρήσιμες για ασφαλή δημιουργία κλειδιών και αυθεντικοποίηση. Η μελέτη ξεκινά με την εισαγωγή στα αξιόπιστα υπολογιστικά συστήματα και στις βασικές αρχές των PUF, ακολουθούμενη από μια επισκόπηση διαφόρων αρχιτεκτονικών PUF, συμπεριλαμβανομένων εκείνων που έχουν ήδη υλοποιηθεί σε FPGA.

Το Arbiter PUF σχεδιάζεται σε Verilog και ενσωματώνεται σε ένα σύστημα βασισμένο σε FPGA, με firmware γραμμένο σε Embedded C για την εξαγωγή αποκρίσεων και τον υπολογισμό μετρικών. Για την ενίσχυση της ασφάλειας, εισάγεται η πύλη Balance XOR, η οποία βελτιώνει την ομοιομορφία της PUF χωρίς να επηρεάζει σημαντικά τη μοναδικότητα. Ωστόσο, αυτή η βελτίωση συνεπάγεται αυξημένη χρήση πόρων του FPGA, ιδιαίτερα στις Look-Up Tables (LUT), ενώ η χρήση των Flip-Flop (FF) παραμένει σχεδόν αμετάβλητη.

Τα πειραματικά αποτελέσματα καταδεικνύουν την αποτελεσματικότητα της Balance XOR στη μείωση της προκατάληψης, αν και απαιτείται περαιτέρω δοκιμή για την αξιολόγηση του αντίκτυπού της σε διαφορετικά μεγέθη PUF. Τα ευρήματα αναδεικνύουν τον συμβιβασμό μεταξύ ασφάλειας και αποδοτικότητας υλικού, παρέχοντας χρήσιμες πληροφορίες για τη βελτιστοποίηση μηχανισμών ασφαλείας βασισμένων σε PUF σε FPGA.

Λέξεις Κλειδιά: Physical Unclonable Functions, Trusted Computing System, Arbiter PUF, FPGA implementation

Abstract

This thesis explores the design, implementation, and evaluation of an Arbiter Physical Unclonable Function (PUF) on an FPGA as a hardware security primitive. PUFs leverage inherent manufacturing variations to generate unique and unpredictable challenge-response pairs (CRPs), making them useful for secure key generation and authentication. The study begins by introducing trusted computing systems and PUF fundamentals, followed by an overview of various PUF architectures, including those previously implemented on FPGAs.

The Arbiter PUF is designed in Verilog and integrated into an FPGA-based system, with firmware written in Embedded C for response extraction and metric computation. To enhance security, the Balance XOR gate is introduced, improving the PUF's uniformity without significantly affecting uniqueness. However, this enhancement comes at the cost of increased FPGA resource utilization, particularly in Look-Up Tables (LUTs), while Flip-Flop (FF) usage remains largely unchanged.

Experimental results demonstrate the effectiveness of the Balance XOR in reducing bias, but further testing is necessary to assess its impact across different PUF sizes configurations. The findings highlight the trade-off between security and hardware efficiency, providing insights for optimizing PUF-based security mechanisms in FPGA implementations.

Keywords: Physical Unclonable Functions, Trusted Computing System, Arbiter PUF, FPGA implementation,

Table of Contents

Table of Figures

Table of Tables

Chapter 1. Introduction

The integration of Physical Unclonable Functions (PUFs) within Trusted Computing Systems is essential for enhancing security, particularly in authentication, secret key generation, and resistance against physical and machine-learning-based attacks. Traditional security mechanisms rely on stored cryptographic keys, which are susceptible to extraction through side-channel or invasive attacks. In contrast, PUFs leverage inherent manufacturing variations to generate unique and unclonable responses, thereby eliminating the need for externally stored secrets [5].

A fundamental component of Trusted Computing Systems is the **Hardware Root of Trust (HRoT)**, which establishes a secure foundation for the entire system. The HRoT consists of dedicated hardware mechanisms that ensure system integrity, secure boot, cryptographic processing, and attestation. Unlike software-based security measures, HRoT implementations provide a tamper-resistant environment that prevents unauthorized modifications and ensures that trust is established from the hardware level up [1,4,12]. PUFs play a crucial role in HRoT by generating cryptographic keys on demand, eliminating the risk of stored key extraction while ensuring that security operations, such as authentication and encryption, rely on a device's intrinsic and unclonable physical properties [5].

One of the primary applications of PUFs in Trusted Computing Systems is device authentication. Since each PUF instance produces a unique response to a given challenge, it becomes an effective tool for ensuring the identity of hardware components in security-sensitive applications. Research has demonstrated the effectiveness of Arbiter PUFs in such applications, particularly in FPGA-based implementations where lightweight security solutions are crucial [13]. Additionally, PUFs provide strong resistance against physical tampering, as any attempt to clone the hardware structure alters the device's response characteristics, rendering unauthorized replication infeasible [5].

Moreover, PUFs contribute significantly to **key generation and management**. They enable cryptographic keys to be generated on demand rather than being stored permanently, thereby reducing exposure to attacks. Secure key storage using PUF-based architectures has been explored in various studies, with notable advancements in Arbiter PUF designs improving their reliability and resilience against machine learning attacks [19]. These features make PUFs an ideal choice for augmenting Trusted Computing Systems, ensuring not only authentication but also enhancing overall security and integrity in hardware-based secure environments.

Given the increasing reliance on FPGAs in secure computing applications, integrating a PUF within a Trusted Computing System offers a promising security enhancement. By embedding an Arbiter PUF within an FPGA-based Trusted Computing System, the design ensures a balance between security and performance while mitigating vulnerabilities associated with traditional cryptographic methods [14].

1.1 Trusted Computing System

Trusted computing systems are designed to enhance security and reliability in computing environments by ensuring that hardware and software behave as expected, even in the presence of potential threats. These systems employ a combination of cryptographic techniques, hardware-based security mechanisms, and attestation protocols to establish trust between components and users. The primary objective is to create a computing environment where critical operations and sensitive data remain protected from unauthorized access and tampering [1].

The concept of trusted computing emerged in the early 2000s with the introduction of the Trusted Computing Group (TCG), an industry consortium that aimed to define security standards and frameworks for hardware-based trust [4]. Early implementations included the Trusted Platform Module (TPM), a secure cryptographic processor that enabled features such as secure boot and remote attestation. Over time, trusted computing has evolved to encompass various architectures, including FPGA-based solutions that offer reconfigurability and enhanced security [2].

A fundamental principle of trusted computing is the ability to measure and verify the integrity of a system's state. This is achieved through attestation, a process where a trusted hardware component reports its current state to an external verifier. Secure boot mechanisms ensure that only authorized and untampered software is executed during system startup, preventing malware infections at an early stage [1]. Additionally,

isolation techniques, such as hardware-enforced separation of trusted and untrusted components, further enhance security [3].

1.2 Physical Unclonable Functions (PUF)

Physical Unclonable Functions (PUFs) are hardware security primitives that exploit the inherent manufacturing variations in semiconductor devices to generate unique, device-specific responses to challenges. These variations are unpredictable, even by the manufacturer, making PUFs an effective tool for secure authentication and cryptographic key generation [5].

The concept of PUFs emerged in the early 2000s as researchers explored the potential of intrinsic physical randomness for security applications [25]. Since then, PUFs have gained significant attention due to their ability to provide lightweight security without requiring non-volatile memory storage of secret keys. Various PUF architectures have been developed, including Arbiter PUFs, Ring Oscillator PUFs, and SRAM PUFs, each offering different trade-offs in terms of security, reliability, and implementation complexity [4].

PUFs are widely used in applications such as device authentication, secure key storage, and hardware-based cryptographic protocols. Their ability to generate responses dynamically makes them resistant to invasive attacks and cloning attempts, reinforcing their role in trusted computing systems [6]. However, PUFs also face challenges, including environmental sensitivity and machine learning-based modeling attacks, which have driven ongoing research to enhance their robustness and security [6].

By leveraging unique physical properties for security, PUFs serve as a foundation for enhancing trust in embedded systems and cryptographic applications, particularly in resource-constrained environments such as FPGA-based designs.

1.3 Types of Trusted Computing Systems

Trusted Computing Systems (TCS) encompass a range of hardware and software-based security mechanisms designed to establish a secure execution environment. These systems ensure data integrity, confidentiality, and platform authentication by implementing a **hardware root of trust**. Different approaches have been developed to

achieve these security guarantees, each tailored to specific computing environments. The three primary types of trusted computing systems are the following:

1. **TPM-Based Trusted Computing:** Trusted Platform Modules (TPMs) provide a hardware root of trust by securely storing cryptographic keys, ensuring integrity measurement, and enabling secure boot mechanisms. TPMs have been widely adopted in enterprise computing and personal devices, offering attestation capabilities that verify system integrity before execution [3]. However, their reliance on discrete hardware modules limits flexibility in modern computing environments [4].
2. **TEE-Based Trusted Computing:** Trusted Execution Environments (TEEs), such as Intel SGX and ARM TrustZone, create isolated secure regions within a processor to execute sensitive operations. TEEs allow applications to run in an environment protected from the main operating system, reducing the risk of software attacks [1]. They have gained traction in cloud security and mobile devices but face challenges related to side-channel attacks and limited memory availability [2].
3. **FPGA-Based Trusted Computing:** Field-Programmable Gate Arrays (FPGAs) offer a reconfigurable hardware approach to trusted computing, allowing for customizable security architectures. By leveraging dynamic reconfiguration, FPGA-based systems can provide efficient isolation, secure boot processes, and real-time attestation mechanisms [4]. Their adaptability makes them a strong candidate for hardware-assisted security, particularly in embedded systems and IoT applications [12].

1.4 Objective

Trusted computing necessitates Hardware Root of Trust (HRoT) capabilities, which are provided by circuits such as Physical Unclonable Functions (PUFs). Among various PUF architectures, Arbiter PUFs are widely used due to their lightweight nature and suitability for FPGA implementations.

The objective of this work is the design, implementation, and evaluation of an Arbiter PUF on Xilinx FPGAs. This involves developing a robust PUF architecture, implementing

it on FPGA hardware, and assessing its performance through an extensive evaluation framework. The implementation includes the development of embedded firmware to interface with the PUF, collect Challenge-Response Pairs (CRP), and facilitate its integration within a trusted computing environment. Additionally, an evaluation flow is presented to analyze key PUF metrics, ensuring robustness and reliability. The experimental evaluation is conducted on the target FPGA platform to assess its performance and security properties. To enhance the efficiency of the implemented PUF, a Balance XOR gate is introduced to improve response uniformity and unpredictability.

This thesis is structured as follows:

Chapter 2 provides detailed information on the different types of PUFs, explaining their principles and security characteristics.

Chapter 3 describes the design and implementation of the proposed Arbiter PUF on FPGA.

Chapter 4 presents the experimental results, including the FPGA-based evaluation of the PUF using embedded C.

Chapter 5 concludes the thesis, summarizing key findings and potential future directions.

Chapter 2. Background

Physical Unclonable Functions (PUFs) are hardware security primitives that leverage inherent variations in semiconductor manufacturing to generate unique, device-specific responses. PUFs are broadly categorized into Physical and Digital PUFs, while they can also be classified as Strong or Weak PUFs based on their security characteristics.

Strong PUFs utilize challenge-response mechanisms, offering a large challenge-response space, making them suitable for authentication and cryptographic protocols. These PUFs are resistant to physical cloning but can be vulnerable to machine-learning-based modeling attacks due to their mathematical structure.

In contrast, Weak PUFs provide a stable, device-unique response, which can be used for cryptographic key generation, secure storage (with helper data), and device identification. Since they do not take a challenge as input, their response space is limited, making them unsuitable for authentication-based applications.

Several studies provide a comprehensive overview of strong and weak PUFs. [6] categorizes strong PUFs as those with a vast challenge-response space, making them resistant to direct cloning but potentially vulnerable to predictive modeling attacks. [15] further analyzes strong PUFs in terms of their theoretical security models and proposes methods to enhance their security. Meanwhile, weak PUFs, as described in [16], typically rely on intrinsic chip variations to generate a stable cryptographic key. The review by Kusum Lata et al. [13] provides a comparative analysis of FPGA-based PUF implementations, discussing how different designs fit into the strong and weak PUF paradigms.

The following subsections will detail specific PUF architectures found in the literature, illustrating their characteristics, security implications, and practical implementations.

2.1 Arbiter PUF Variants

The Arbiter PUF is a widely studied delay-based Physical Unclonable Function (PUF) that leverages the inherent manufacturing variations in circuit delays to generate unique responses. It falls under the category of strong PUFs (physical PUFs) due to its challenge-response behavior and high entropy source. The fundamental design of an Arbiter PUF consists of two parallel delay paths, with a challenge input determining how signals propagate through the paths. The final output is resolved by an arbiter, typically a D flip-flop, which decides the faster signal path, thereby producing a unique response [5][13][17].

Several variants of the Arbiter PUF exist, each addressing different security and reliability concerns. The **XOR Arbiter PUF** enhances security by combining multiple Arbiter PUF instances with an XOR function to mitigate machine learning attacks [18]. The **Feed-Forward Arbiter PUF** introduces internal loops, making the delay paths dependent on intermediate signals, thereby increasing complexity but at the cost of reliability [19]. The **Priority Arbiter PUF (PA-PUF)** modifies the arbitration mechanism to improve resistance against modeling attacks while maintaining lower hardware overhead [20].

A visual representation of the standard Arbiter PUF will be presented later in Chapter 3, alongside its final implemented form.

In designing a trusted computing system, selecting an appropriate Physical Unclonable Function (PUF) is crucial to ensuring security, efficiency and feasibility of implementation. Among the various PUF designs discussed in the previous subsections, the Arbiter PUF was chosen for this implementation due to its well-established architecture, compatibility with FPGA-based designs, and efficient hardware utilization. The Arbiter PUF offers several advantages over alternative PUF architectures:

Efficient FPGA Implementation – The Arbiter PUF is relatively simple to implement using FPGA-based delay elements, making it a suitable choice for embedded applications. [21] highlight its feasibility in FPGA environments, especially when resource constraints are a concern.

1. **Lower Hardware Overhead** – Compared to Ring Oscillator PUFs, which require multiple oscillators to generate stable frequency differences, the Arbiter PUF has a more compact circuit design, leading to reduced hardware footprint [14].
2. **Scalability with XOR Variants** – While the standard Arbiter PUF is vulnerable to machine learning (ML) attacks, XOR-based variants significantly increase

security and resilience against such attacks [19]. This allows for a configurable security level based on the number of XOR layers added.

3. **High Challenge-Response Space** – Unlike SRAM PUFs, which have a fixed set of responses derived from the power-up state of memory cells, Arbiter PUFs offer a much larger challenge-response space**,** making them suitable for applications requiring a large number of unique responses [18].
4. **Better Aging and Environmental Robustness** – Compared to Ring Oscillator PUFs**,** which are sensitive to environmental variations such as temperature and supply voltage fluctuation**s**, the Arbiter PUF demonstrates better stability when properly designed [20].

While the Arbiter PUF presents multiple advantages, it also has some limitations that need to be considered:

1. **Higher Sensitivity to Process Variations** – Compared to SRAM PUFs, which rely on intrinsic manufacturing variations in memory cells, the Arbiter PUF depends on delay path differences, which can be affected by fabrication inconsistencies. As a result, careful design calibration is required to ensure reliability [18]
2. **Limited Reliability in Harsh Environments** – Although more robust than Ring Oscillator PUFs, Arbiter PUFs may still exhibit bit-flip errors under extreme conditions [20]. However, post-processing techniques such as error correction and majority voting can address this issue.

The Arbiter PUF was selected due to its hardware efficiency, scalability, and large challenge-response space, making it a practical choice for trusted computing applications. Its lightweight design allows for efficient FPGA implementation while maintaining a significant challenge-response capability. Given these considerations, the Arbiter PUF offers a balance between implementability and resource efficiency, making it a suitable candidate for FPGA-based trusted computing systems.

2.2 Ring-Oscillator PUF

The Ring Oscillator PUF (RO-PUF) is another well-known delay-based PUF that exploits variations in ring oscillator frequencies to generate unique responses. Unlike the Arbiter PUF, which relies on signal propagation delays, the RO-PUF utilizes multiple ring oscillators instantiated on an FPGA, each experiencing slight frequency variations due to process variations. These differences are then compared to produce a stable and unique response [13][21].

RO-PUFs are normally classified as weak PUFs because they typically generate a limited number of challenge-response pairs (CRPs), making them more suitable for key generation and device identification rather than large-scale authentication. However, some advanced RO-PUF designs introduce configurable oscillators, differential frequency measurements, or additional challenge mechanisms, increasing the challenge space and exhibiting strong PUF-like behavior [15]. In comparison to Arbiter PUFs, RO-PUFs are often more resilient to noise and environmental variations due to their frequency-based mechanism [22].

A common implementation consists of multiple identical ring oscillators, each composed of an odd number of inverters connected in a feedback loop. A frequency counter measures their oscillation rates, and a comparator determines which oscillator is faster, producing a binary response. Various enhancements, such as RO-PUFs with controlled oscillators and differential measurement techniques, have been proposed to improve reliability and security against modeling attacks [15].

The following image illustrates the standard structure of a Ring Oscillator PUF, showing the placement of ring oscillators, frequency counters, and comparators in the design.

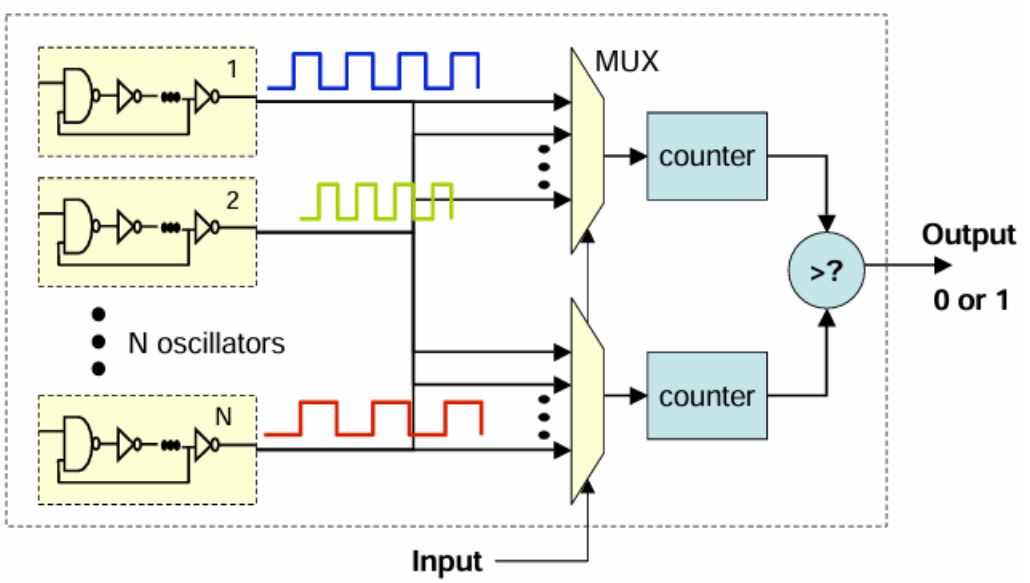


Figure 1. Ring Oscillator PUF representation, reprinted from [5]

‘

2.3 SRAM PUF

SRAM-based Physical Unclonable Functions (SRAM PUFs) are one of the most commonly used types of PUFs, particularly in embedded security applications. They are classified as physical PUFs but are typically considered weak PUFs because they generate a fixed, device-unique response rather than supporting a large challenge-response space [6][13]. Unlike physical, weak PUFs, which store responses for later retrieval, SRAM PUFs dynamically generate their responses at power-up, eliminating the need for non-volatile storage [16].

SRAM PUFs exploit the startup behavior of uninitialized SRAM cells. When an SRAM cell is first powered on, it settles into either a '0' or '1' state based on minute variations in transistor properties such as threshold voltage and doping concentration. These variations are introduced during manufacturing and are unique to each chip, creating a distinct and reproducible binary pattern for every device [13][21]. Since these responses are stable across multiple power cycles under normal conditions, SRAM PUFs are widely used for secure device authentication and cryptographic key generation [6][15].

A key advantage of SRAM PUFs is that they require no additional hardware beyond existing SRAM memory, making them highly efficient for integration into microcontrollers and secure processors [22]. However, external factors such as temperature fluctuations, aging effects, and voltage variations can introduce noise in the response. To address this, techniques such as fuzzy extractors and error correction codes are commonly employed to ensure reliability and consistency [6][16].

Several studies provide insights into SRAM PUF architectures, security challenges, and real-world applications. [13] presents a comparative analysis of FPGA-based PUF designs, including SRAM PUFs. [6] discusses strong PUFs and their suitability for entity authentication, covering SRAM-based approaches. [15, 16] explore the cryptographic security and resilience of SRAM PUFs against modeling attacks, while [21] examines their reliability in FPGA implementations.

An illustration of a typical SRAM PUF structure is provided below to visually depict its operation.

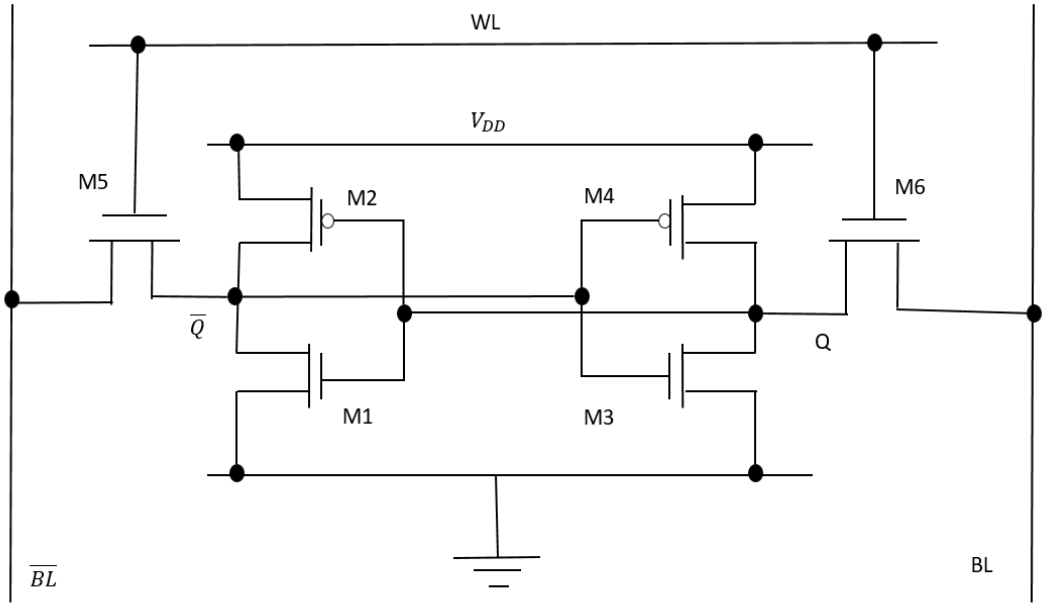


Figure 2. SRAM PUF representation

2.4 Other PUF Designs

Beyond Arbiter, Ring Oscillator, and SRAM PUFs, various other PUF designs have been proposed in the literature, leveraging different physical properties of hardware to

generate unique device signatures. These PUFs can be classified based on two independent axes: strong vs. weak (which depends on challenge-response behavior and security properties) and physical vs. digital (which depends on whether they rely on intrinsic physical variations or algorithmic techniques).

The **XOR Arbiter PUF** is an enhancement of the standard Arbiter PUF, designed to improve resilience against ML attacks. It consists of multiple Arbiter PUF instances running in parallel, with their outputs combined using an XOR operation. This increases the non-linearity of the response function, making it more resistant to modeling attacks. However, increasing the number of XOR stages also introduces stability concerns due to accumulated circuit delays. This PUF falls under the strong, physical PUF category, as it significantly expands the challenge-response space and offers improved resistance against ML attacks. Studies such as [18] and [19] analyze the security and trade-offs of XOR-based Arbiter PUFs.

The **Priority Arbiter PUF (PA-PUF)** is a variant designed to enhance security and reliability by introducing a hierarchical arbitration mechanism instead of a single arbiter. By prioritizing specific delay paths, it improves security against ML attacks while maintaining robustness. Like the standard Arbiter PUF, it is still fundamentally based on deterministic delay paths. However, due to its increased complexity and improved resilience against ML attacks, it is categorized as a strong, physical PUF. This approach is discussed in [20].

Several **delay-based PUFs** extend the concepts of Arbiter and Ring Oscillator PUFs by introducing different structures to measure delay variations more effectively. These variants aim to balance hardware efficiency and security. For example, [14] explores optimizations that minimize FPGA resource utilization while maintaining security properties. Since they rely on deterministic delay measurements and are susceptible to ML attacks, they are categorized as weak, physical PUFs.

The **SiCBit-PUF** is a strong, physical PUF, specifically designed for trusted System-on-Chip (SoC) applications. It utilizes in-cache bitflips to generate challenge-response pairs, leveraging intrinsic memory effects rather than delay-based variations. This approach improves resilience against modeling attacks and enhances the unpredictability of responses. As a result, the SiCBit-PUF is categorized as a strong, physical PUF due to its reliance on hardware-specific physical properties for secure authentication [26].

The **TI-PUF** is a strong PUF, specifically designed to resist side-channel attacks, such as power analysis attacks, by incorporating threshold implementation techniques. This PUF leverages physical properties of hardware rather than purely digital delay

variations, making it harder to model. Due to its improved resistance to invasive and side-channel attacks, it is categorized as a strong, physical PUF. [23] discusses its security advantages and implementation details.

Some PUF designs are tailored for specific applications, such as IoT security and secure key storage, and may fall into different categories depending on their implementation.

[24] proposes a **PUF-based authentication mechanism for medical IoT devices**, improving device identification and security. The classification of this PUF depends on its underlying implementation, which is not explicitly stated in the study.

[21] discusses **FPGA-specific optimizations** for PUF implementations, focusing on reducing overhead and improving efficiency. These are typically digital, weak PUFs, as FPGA-specific constraints often lead to reproducible responses rather than fully unpredictable physical characteristics.

The classification of PUFs as strong vs. weak and physical vs. digital depends on their response behavior, security properties, and scalability. Strong PUFs, such as traditional Arbiter PUFs, the TI-PUF, XOR Arbiter PUF, and SiCBit-PUF, provide high resistance against ML attacks and side-channel attacks, whereas weak PUFs, like traditional Ring Oscillator PUFs, and SRAM PUFs, offer efficient authentication mechanisms but are more vulnerable to modeling attacks. Additionally, physical PUFs rely on intrinsic variations in hardware, while digital PUFs may be constructed using algorithmic or reconfigurable logic approaches rather than relying on unique manufacturing variations. The choice of a PUF depends on the specific application requirements, including security, hardware constraints, and environmental conditions.

2.5 PUFs implemented on FPGA

Physical Unclonable Functions (PUFs) have been widely implemented on FPGAs due to their inherent reconfigurability and ability to generate unique, device-specific responses. Various PUF architectures have been explored to enhance security, randomness, and efficiency in FPGA implementations.

A comprehensive review of different FPGA-based PUF designs is provided by Lata and Cenkeramaddi [13], where they analyze the strengths and weaknesses of various implementations, including Arbiter, Ring Oscillator, and SRAM PUFs. Their study

highlights the trade-offs in terms of area, power consumption, and security properties [13].

Arbiter PUFs, a well-known PUF type, have been successfully implemented on FPGAs while aiming to reduce hardware overhead. Ivaniuk and Zalivaka [14] proposed an optimized Arbiter PUF design that minimizes resource usage without compromising security [14]. Similarly, Zhang et al. [21] introduced techniques for FPGA-specific PUFs, focusing on improving robustness and resistance against environmental variations [21].

To address resource constraints in FPGA-based security applications, Vega [22] explored efficient PUF implementations through FPGA resource re-utilization. His work demonstrates how PUF designs can be optimized to minimize overhead while maintaining reliability [22]. Additionally, Rosero-Montalvo, István, and Hernandez [4] provided an extensive survey on trusted computing solutions that leverage FPGAs, discussing the role of PUFs in securing hardware platforms [4].

These studies collectively emphasize the effectiveness of FPGA-based PUFs in securing cryptographic keys, authentication systems, and trusted computing environments. As research progresses, further optimizations in design and implementation continue to enhance their practicality for real-world applications.

Chapter 3. Arbiter PUF Design & Firmware Implementation on FPGA

In this chapter, the following aspects of the project are presented: the design and architecture of the Multi-instance Arbiter PUF, the Verilog code for the PUF modules, the steps taken after developing the Verilog code, and the development of the embedded C program that was executed on the FPGA for testing. Additionally, the integration of the Balance XOR and its impact on the design are discussed.

Below is a diagram that illustrates the workflow followed during the implementation process.

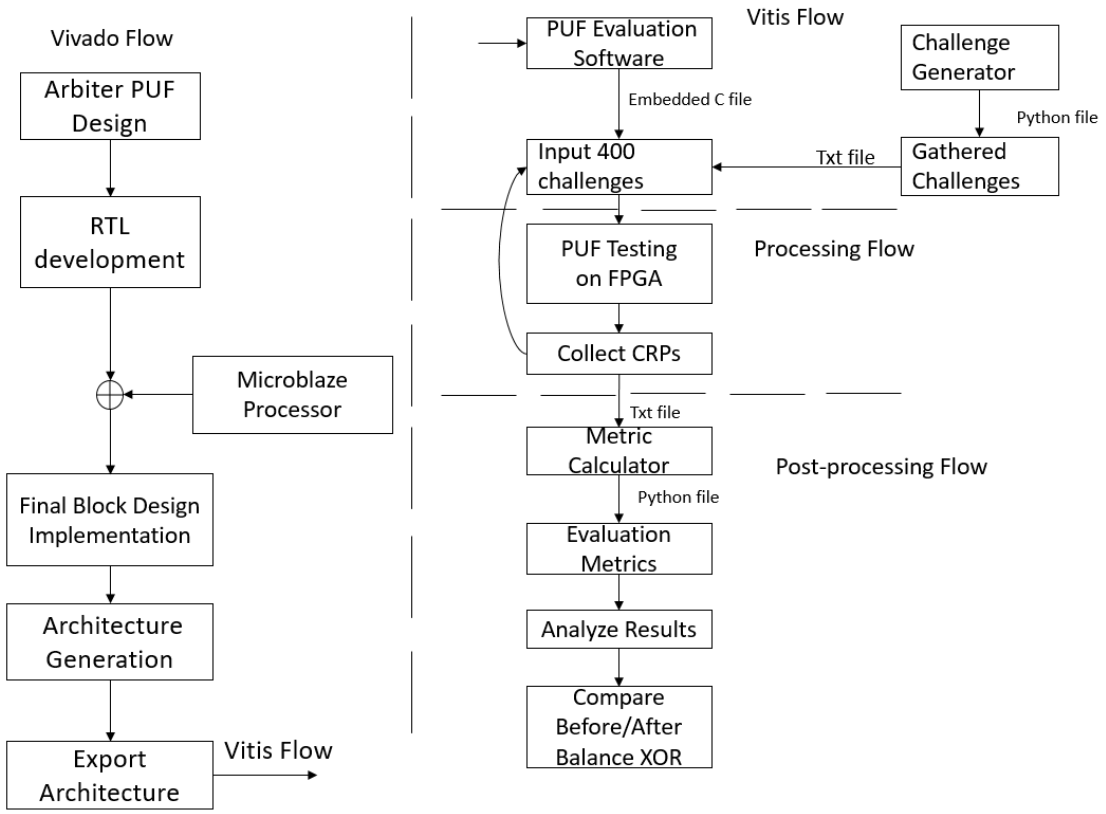


Figure 3. Arbiter PUF design flow and firmware on FPGA

The flow diagram illustrating the Vivado and Vitis processes for the design is presented in Figure 3. The diagram is split into four flows: the Vivado flow, which outlines the steps taken to create the final block design, the Vitis flow, which describes the process after the block design is completed and the architecture has been exported, leading to the development of PUF evaluation software and challenge input, the Processing Flow, that covers the execution of embedded C on the FPGA and the collection of CRPs and the Post-processing flow that focuses on calculating and analyzing PUF metrics, as well as comparing the design before and after the addition of Balance XOR..

For the Vivado flow, the process begins with the design of the Arbiter PUF. Next, Verilog modules are developed to describe the Multi-Instance Arbiter PUF. After converting this module into an IP block, the PUF block is added and connected to the MicroBlaze microprocessor. Once the final block design is completed, the synthesis, implementation, and bitstream generation steps are performed. After these steps, the design is ready to be exported for use in the Vitis project.

The Vitis flow will be discussed in Chapter 4.

3.1 Presentation of Multi-instance Arbitrary PUF

In this subsection, the design of the Multi-instance Arbiter PUF is discussed. This design is composed of two main components: the Delay Line and the Arbiter. Below, these components are introduced:

- **Delay Line**: The Delay Line consists of multiple switch blocks, with each switch block containing two 2-to-1 multiplexers. These multiplexers determine the path that the pulse takes based on the challenge bits. If a challenge bit is '1', the pulse follows one path; if it is '0', the pulse follows another. The input to the first switch block is a pulse, which is fed into both inputs of the switch block. The length of the delay line can be adjusted as needed; increasing the number of challenge bits enhances the complexity of the design.

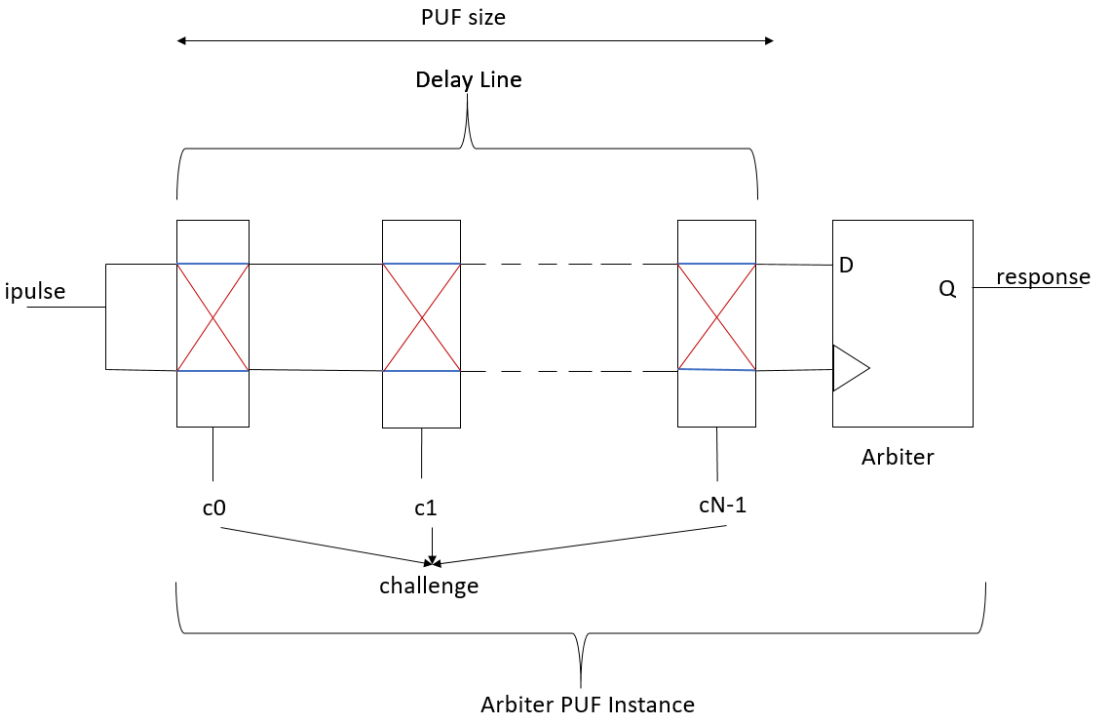


Figure 4. Arbiter PUF instance architectonic

In an Arbiter PUF, the challenge is a sequence of bits (c0, c1, ..., cN-1) that determines how the input pulse propagates through the Delay Line. Each challenge bit controls the multiplexers in the switch blocks, altering the pulse's path. If a challenge bit is '1', the pulse takes one route; if it is '0', it takes another. This variation in paths introduces slight differences in delay between the two signals.

Figure 4 visually represents this process. In the figure an Arbiter PUF instance is presented. The red lines inside the switch blocks indicate that a challenge bit of '1' was selected, while the blue lines indicate a challenge bit of '0'. The pulse enters the first switch block and propagates through the network of multiplexers, with each switch directing the signal based on the challenge bits. The longer the delay line (i.e., the more challenge bits used), the more complex and unpredictable the signal path becomes. In the figure the PUF size is also shown.

The figure also illustrates the PUF size, which is defined as the number of stages in the delay line, or in other words, the length of the challenge in bits.The response is determined by the Arbiter, which detects which of the two signals arrives first at its inputs. If the signal at the D input arrives before the one at the clock input, the response is '1', otherwise, it is '0'. Since each challenge results in a unique propagation delay, the Arbiter's output acts as a challenge-response pair.

This structure allows the PUF to generate device-specific, unpredictable responses, forming the basis for its security.

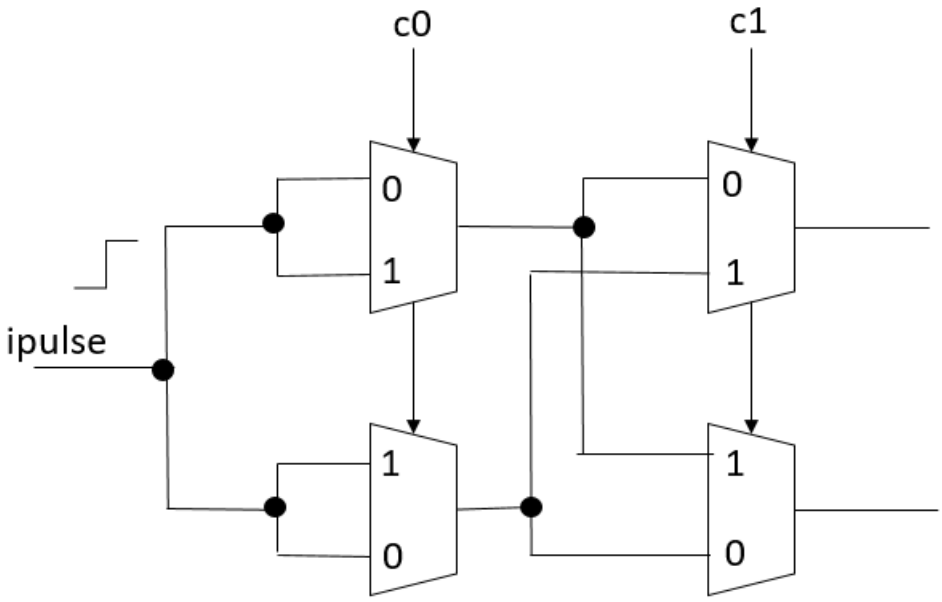


Figure 5. Switch Block 1 and 2 connection

Figure 5 illustrates the connection between the first and second switch blocks, demonstrating how these blocks are linked together. The input to the first switch block is a pulse, which is fed into both inputs of the switch block. The length of the delay line

can be adjusted as needed; increasing the number of challenge bits enhances the complexity of the design.

- **Arbiter**: The Arbiter is responsible for determining the final output of the PUF. It can be implemented using either a Latch or a D Flip-Flop, as depicted in Figure 2.The Arbiter works by detecting which of the two signal paths completes first**.** If the signal arrives first at the D input, the output will be '1'. If the signal arrives first at the clock input, the output will be '0'.

Security Considerations:

While the Arbiter PUF is known to be one of the easiest PUF designs to attack, its resilience against machine learning attacks improves as the number of switch blocks increases. In other words, a longer Delay Line increases the difficulty of modeling the PUF and extracting its secret key.

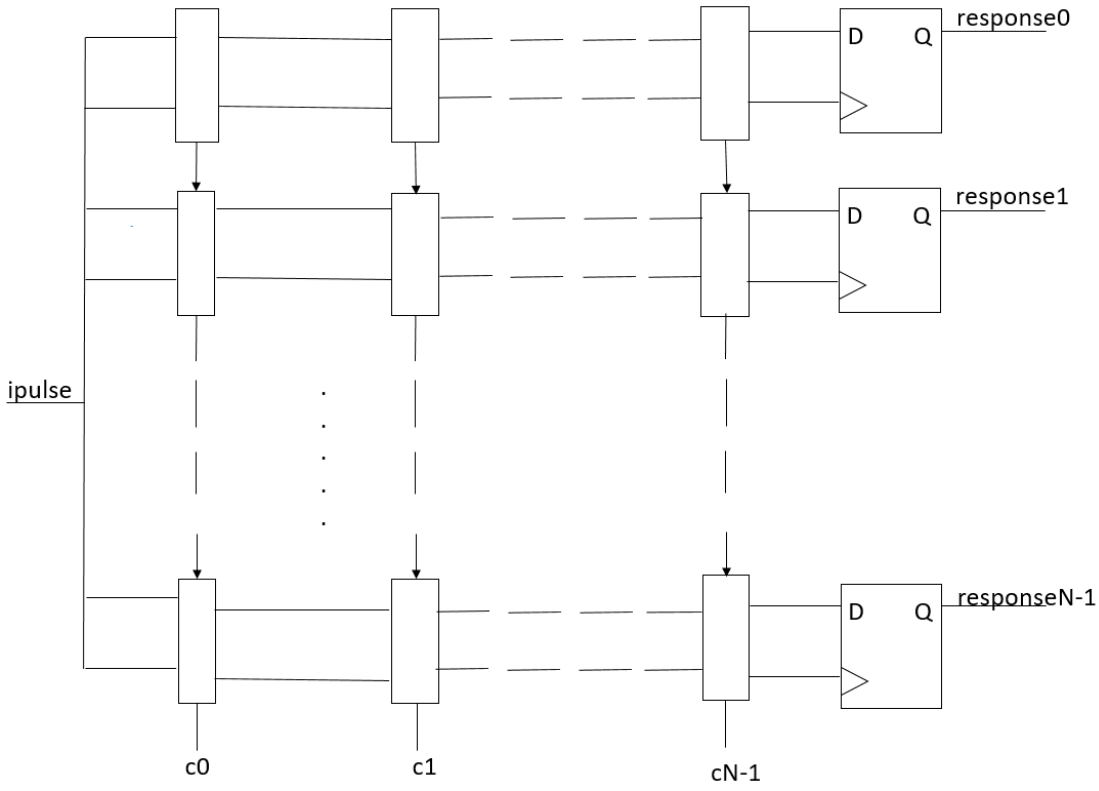


Figure 6. Multi-instance Arbitrary PUF representation

Figure 6 presents the design of the Multi-instance Arbiter PUF. As the name suggests, it consists of multiple Arbiter PUF instances. Each instance outputs a response, and together, they form a final response of the desired length. This multi-instance approach helps in enhancing security and reliability by increasing the entropy of the responses.

3.2 Verilog Code Implementation

In this subsection, the Verilog code used for the creation of the Multi-instance Arbiter PUF is discussed. The code is structured hierarchically, with each module serving a specific function in the overall design.

- **AXI Wrapper (Top Level) –** This module acts as an interface between the Multi-instance Arbiter PUF and the MicroBlaze microprocessor, enabling communication via the AXI protocol.
- **Multi-instance Arbiter PUF Module** – This module instantiates multiple Arbiter PUF instances, allowing the design to generate multiple responses in parallel.
- **Arbiter PUF Module** – This module integrates the Delay Line and the Arbiter, forming a single Arbiter PUF instance.
- **Delay Line & Arbiter Modules** – These modules define the fundamental behavior of the PUF structure, where the challenge bits influence signal propagation through the Delay Line, and the Arbiter determines the final response.
- **2-to-1 Multiplexers (Lowest Level)** – These are the basic building blocks of the Delay Line, responsible for controlling signal paths based on challenge bits.

This hierarchical structure ensures modularity, making the design easier to test, modify, and integrate within FPGA-based implementations. Below is a Figure that illustrates the module hierarchy.

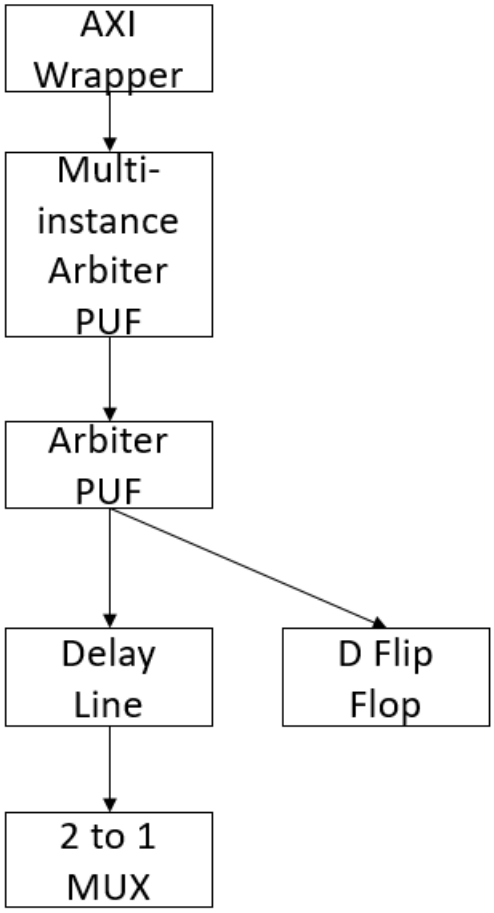


Figure 7. Module hierarchy

3.3 Embedded C and the rest of the process

This subsection finalizes the design creation and outlines the process leading up to the development of the embedded C program. After implementing the Verilog code for the Multi-instance Arbiter PUF, the next step is to integrate it as an IP Core. Vivado's "Create and Package New IP" tool facilitates this integration.

Once the IP Core is generated, it is added to a new project where a MicroBlaze processor is instantiated. Vivado automatically configures essential connections for the processor. Additional components, such as the PUF module and AXI UART, are then incorporated. The AXI UART enables response collection via the Vitis terminal when the embedded C code is executed on the FPGA. To enhance the system, the XADC Wizard is included in the design, allowing real-time monitoring of temperature and power consumption.

After completing synthesis, implementation, and bitstream generation, the hardware design is exported to Vitis. A new Vitis project is then created, including both the platform and application, for developing the embedded C code. The challenge values are inserted into the challenge array within the program, making it ready to run on the FPGA receive the responses on the terminal.

3.4 Balance XOR

This subsection discusses the addition of an XOR operation to the delay_line module. The term "Balance XOR" is used because, although an XOR operation is performed and stored in a wire, its result is never utilized elsewhere in the code. As shown in Chapter 4, the Uniformity metric moves closer to 50% with its addition. Therefore, the name “Balance XOR” reflects its balancing effect

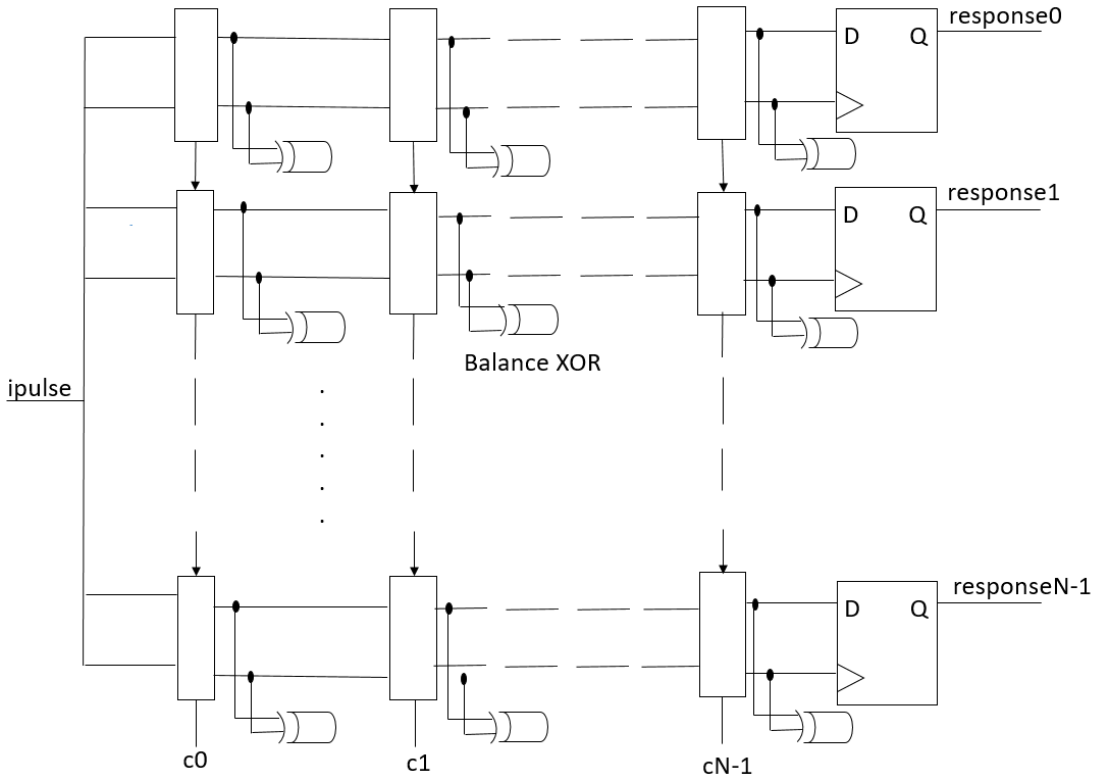


Figure 8. Multi-instance Arbitrary PUF representation with the Balance XOR

Figure 8 illustrates the design of the Multi-Instance Arbiter PUF with the Balance XOR integrated. Although the Balance XOR is present in the design, its output is not utilized, as previously discussed. The figure also shows how the Balance XOR is connected throughout the entire Delay Line. To ensure it remains in the design, the dont_touch Verilog attribute is applied.

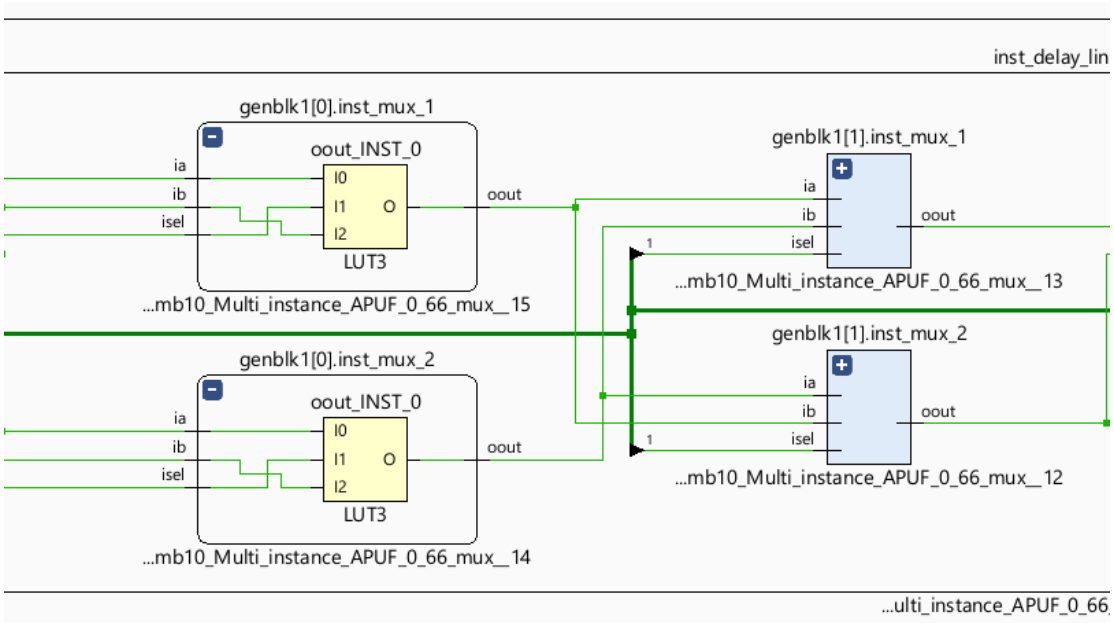


Figure 9. LUT of a switch block in post implementation schematic, taken from Vivado

Figure 9 presents the Look-Up Tables (LUTs) from a random switch block of the PUF. This schematic is taken from the post-implementation stage of the final block design.

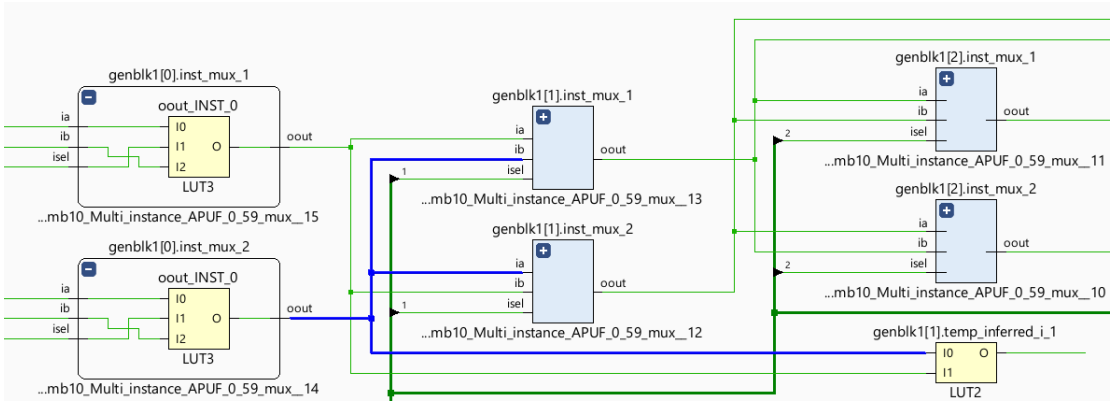


Figure 10. LUT of a switch block with the Balance XOR in post implementation schematic, taken from Vivado

Figure 10 illustrates the Look-Up Tables (LUTs) of a random switch block and the LUT of the Balance XOR. Similar to the previous schematic, this one is taken from the post-implementation stage of the final block design, including the addition of the Balance XOR. In the figure, the Balance XOR is shown connected to the outputs of the switch block via a blue wire.

As we will show in chapter 4, the additional XOR gate between the switch blocks slightly improves the efficacy of the Arbiter PUF. We attribute this effect to the additional physical constraints that this gate imposes which may lead the mapping tool to place the

Look-Up Tables (LUT) that implement these gates as close as possible to the switch block

Chapter 4. Experiments

In this chapter, the resource utilization of the circuit along with the Metrics of Uniformity and Uniqueness are presented. The following tables and diagrams illustrate how resource usage varies and how the Uniformity and Uniqueness metrics change across different PUF sizes. Additionally, we compare the circuit before and after the inclusion of the Balance XOR to evaluate its impact.

4.1 Experimental Evaluation Flow

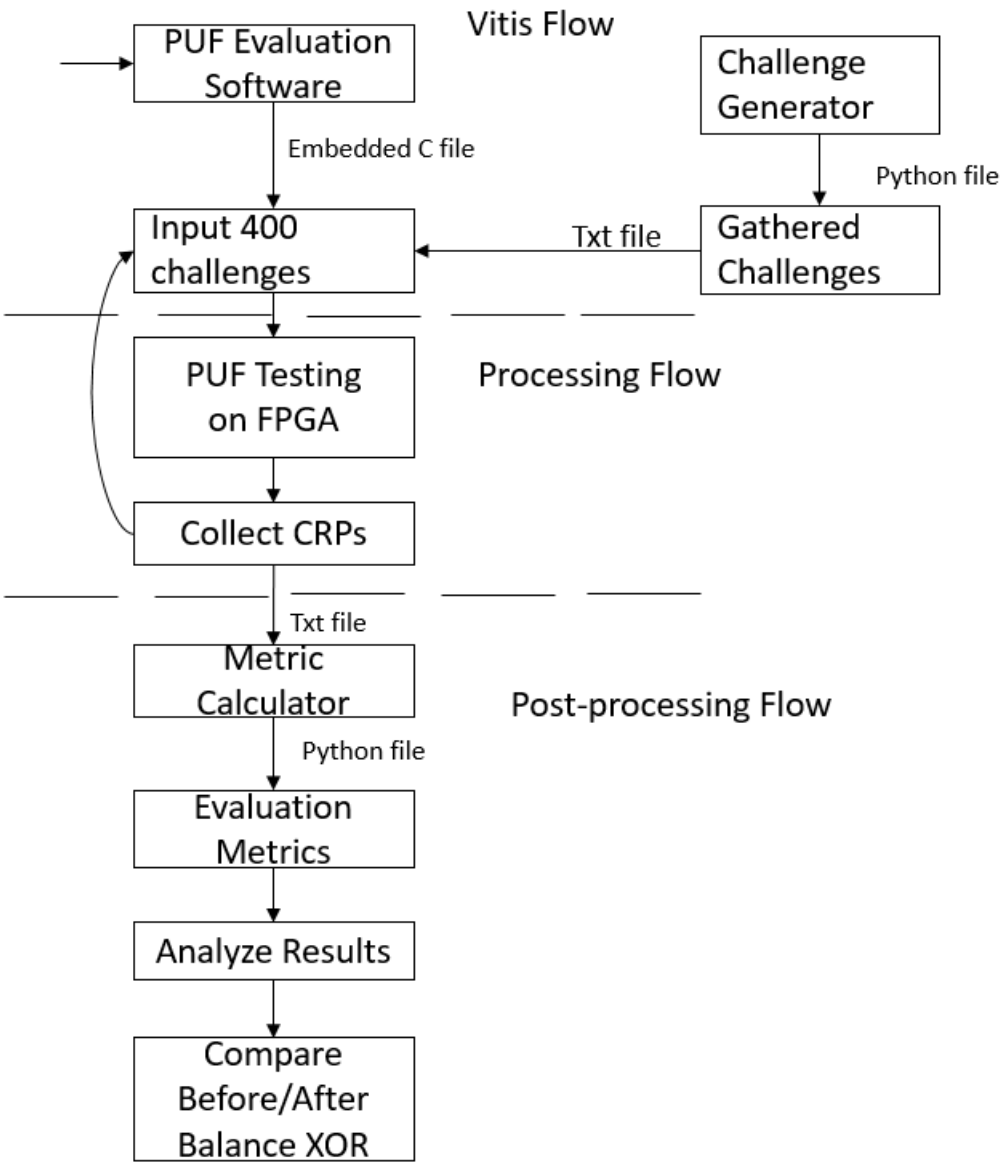


Figure 11. Experimental Workflow for PUF Response Analysis

The evaluation and metric collection followed the workflow presented in Figure 11. Python programs were developed for both challenge generation and metric calculation. The responses collected from the FPGA were stored in a text file, which was later used for metric evaluation.

After the development of the embedded C program and the input of challenges in the program file, the embedded C code is executed on the FPGA. First the challenges are sent to the PUG, followed by the pulse, allowing the responses to be collected via UART and stored in a text file for later analysis. Due to Vitis limitations, only 400 challenges could be processed at a time. To reach a total of 10,000 responses for each challenge-response

size, this process was repeated multiple times: selecting a set of challenges, running the program on the FPGA, collecting the responses, and storing them.

Once all 10,000 responses were gathered, the Uniformity and Uniqueness metrics were computed. The final step involved analyzing how close these metrics were to the ideal 50% and comparing the results across different PUF sizes and between the designs before and after the addition of the Balance XOR.

4.2 Evaluation Metrics

In this subsection, the methodology used to calculate the Uniformity and Uniqueness metrics for the Multi-instance Arbiter PUF is described. These metrics are essential for evaluating the quality and security of Physical Unclonable Functions (PUFs). A structured workflow outlining the process from response collection to metric calculation is also provided.

4.2.1 Uniformity

Definition of Uniformity

Uniformity measures the balance between '0's and '1's in a PUF's responses. Ideally, a PUF should generate an equal number of '1's and '0's, meaning the Uniformity should be as close to 50% as possible. A significant deviation from 50% indicates bias, reducing randomness and making the responses more predictable, thus increasing vulnerability to attacks such as machine learning-based modeling attacks [19]. Ensuring uniformity is critical for PUF reliability in key generation and authentication systems [5].

To address bias and improve uniformity, techniques such as bitflip-based response computation have been proposed [26] These methods enhance randomness by introducing controlled perturbations, making responses more resilient against machine learning attacks while maintaining the uniqueness and stability of the PUF.

Uniformity Evaluation

Uniformity is evaluated as the average **Hamming Weight (HW)** across k challenge-response pairs:

$$Uniformity = \frac{1}{k}\sum_{i=1}^{k} HW(R_i) * 100\% \qquad (1)$$

where:

- R_i is the response for the i-th challenge C_i.
- HW (R_i) represents the number of '1's in the response.

A Uniformity value close to 50% ensures that PUF responses are random and unbiased, enhancing security. A highly biased PUF (e.g., 70% '1's) would be easier to predict, weakening its cryptographic strength [13]. Additionally, strong uniformity helps maintain PUF resilience against environmental variations and aging effects, which are crucial for long-term reliability in FPGA implementations [5].

4.2.2 Uniqueness

Definition of Uniqueness

Uniqueness measures how different the responses of two different PUF instances are when given the same challenge. Ideally, responses from different instances should differ by 50% of their bits, ensuring that each PUF-based device has a unique and unpredictable signature. This uniqueness prevents attackers from replicating responses and impersonating devices, making it a fundamental property of secure PUFs [26]. High uniqueness is particularly important in hardware security applications, including device authentication and cryptographic key generation [5].

.**Uniqueness Evaluation**

Uniqueness is evaluated using the **Hamming Distance (HD)** between responses of different instances, often referred to as inter-die HD. If two PUF instances, i and j ($i \neq j$), generate responses R_i and R_j for the same challenge CCC, the average inter-die Hamming Distance across k instances is calculated as:

$$Uniqueness = \frac{2}{k(k-1)} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{HD(R_i, R_j)}{n} * 100\% \qquad (2)$$

where:

- HD (R_i, R_j) is the Hamming Distance between the response R_i from PUF instance i and R_j from instance j.

- n is the length of the response (e.g., 64 bits).
- k is the number of different PUF instances.

A uniqueness value close to 50% ensures that different PUF instances produce sufficiently distinct responses, making cloning or impersonation nearly impossible [26]. However, if uniqueness is too low (e.g., 40%), PUF responses may be too similar, increasing the risk of machine learning attacks and reducing entropy [19]. Conversely, extremely high uniqueness (e.g., above 55%) may indicate excessive randomness, which could reduce reliability [13]. The balance between uniqueness and reliability is a key factor in designing practical PUF-based security systems.

4.3 Experimental Results

Subsection 4.3 presents and discusses the Uniformity and Uniqueness metrics for the Multi-instance Arbiter PUF before and after the addition of the Balance XOR. It also includes comparisons between different PUF sizes. And between the circuit before and after adding the Balance XOR. Uniformity and Uniqueness are evaluated for PUF sizes to observe their impact on the PUF's behavior.

Following is the table for the circuit before the addition of Balance XOR.

PUF size	Uniformity (%)	Uniqueness (%)
4	87.48	20.72
8	74.93	37.31
16	67.70	43.41
32	62.77	46.57
64	59.00	48.33
128	56.76	49.07

Table 1. Metrics calculated after the tests across different PUF sizes

Table 2 presents uniformity and uniqueness metrics for different PUF sizes, showing how challenge and response sizes affect randomness and PUF instance differentiation.

1. **Uniformity** (ideal: 50%) decreases as PUF size increases. It starts at 87.48% (4-bit) and drops to 56.76% (128-bit). The sharpest drop is from 4-bit (87.48%) to 8-bit (74.93%), followed by a more gradual decline. Higher uniformity in smaller pairs indicates bias, while larger pairs improve randomness and security.

2. **Uniqueness** increases with PUF size, from 20.72% (4-bit) to 49.07% (128-bit), making PUF instances more distinguishable. Shorter PUF size exhibit higher bias, while longer pairs enhance uniqueness, though still below the ideal 50%.

The diagram illustrates these trends, with uniformity declining as PUF size grows (from 87.48% at 4-bit to 56.76% at 128-bit), aligning with expected randomness improvements. Uniqueness follows an upward trend (20.72% → 49.07%), enhancing security by differentiating PUF instances.

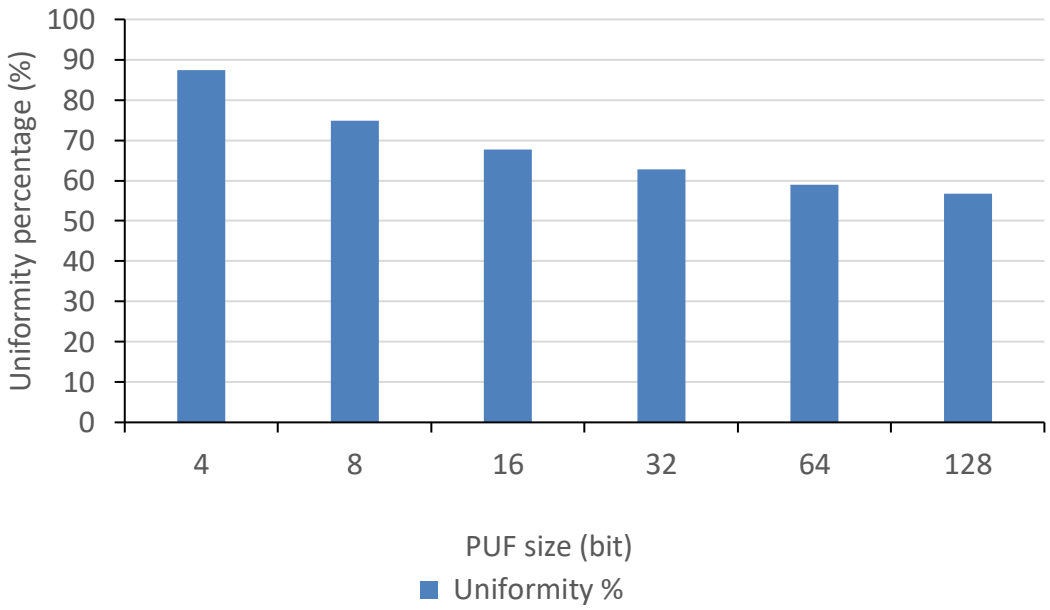


Figure 12. Diagram of Uniformity percentage across different PUF sizes

Figure 12 visually represents the uniformity trend described in Table 1. As PUF size increases, uniformity decreases from 87.48% to 56.76%. The diagram highlights the sharpest decline between 4-bit and 8-bit PUF size, while subsequent reductions become more gradual. This confirms that higher challenge lengths improve randomness, making responses more balanced for security applications.

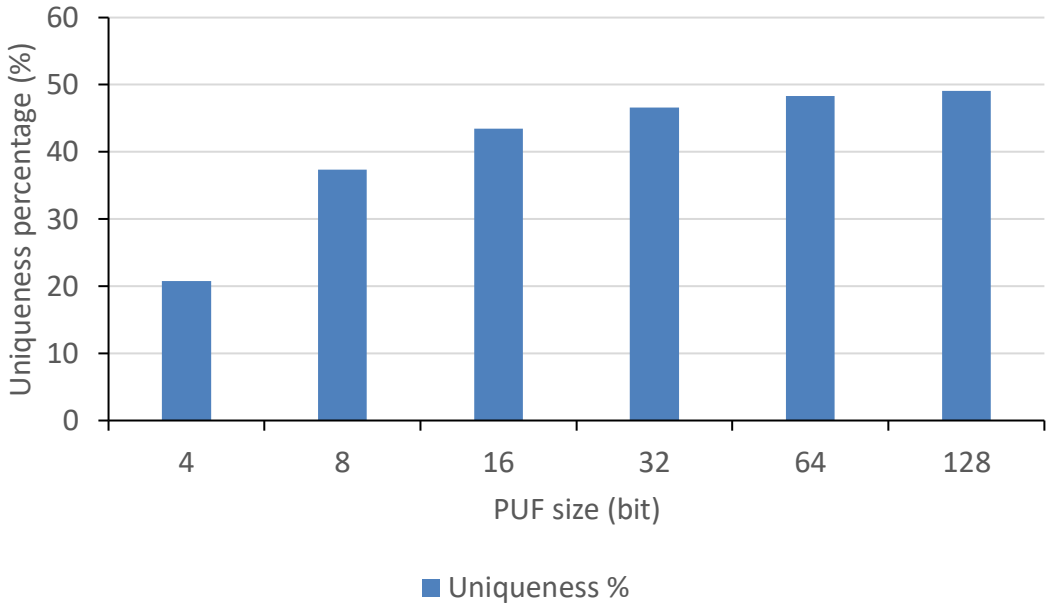


Figure 13. Diagram of Uniqueness percentage across different PUF sizes

Figure 13 illustrates the uniqueness trend across varying PUF sizes. As PUF size increases, uniqueness improves from 20.72% to 49.07%. The diagram reinforces that larger challenges enhance PUF instance differentiation, with uniqueness approaching the ideal 50%. The trend aligns with Table 1's observations, supporting the claim that PUF size plays a crucial role in improving PUF security.

Due to FPGA resource constraints, 128-bit PUF size is not included in the Balance XOR metrics in the next subsection.

Below are the table and diagrams for the circuit after the addition of the Balance XOR.

PUF size	Uniformity (%)	Uniqueness (%)
4	83.25	23.95
8	69.47	41.28
16	61.19	47.08
32	58.26	48.05
64	55.12	49.38

Table 2. . Metrics calculated after the tests across different PUF sizes with the Balance XOR

Table 2 presents the uniformity and uniqueness results for the Balance XOR variant of the PUF. Compared to the original design, key trends emerge:

Uniformity Reduction: The Balance XOR design consistently shows lower uniformity. The drop is most pronounced at smaller PUF sizes (e.g., 4-bit, 87.48% -> 83.25%), but as the PUF size increases, the gap narrows (e.g., 64-bit values are similar).

Uniqueness Improvement: Uniqueness is consistently higher, enhancing device-specific responses. At 64-bit PUF size, it reaches 49.38%, slightly exceeding the original's 48.33% - 49.07% range, improving resistance to cloning.

Trade-off: The Balance XOR favors uniqueness at the expense of uniformity. While a perfect 50% uniformity is ideal, the observed decrease suggests a slight bias toward '1's or '0's.

Trend Consistency: Both designs show decreasing uniformity and increasing uniqueness with larger challenges, but Balance XOR exhibits more extreme shifts.

The diagram visualizes these changes across different PUF size. The x-axis represents PUF size in bits, while the y-axis shows Uniformity and Uniqueness.

With Balance XOR, uniformity still declines as PUF size grows, but the drop is more gradual, preventing extreme bias. At 4-bit, uniformity starts at 83.25%, stabilizing closer to 55% for larger PUF sizes (e.g., 64-64: 55.12%). Uniqueness, on the other hand, improves significantly, rising from 23.95% (4-4) to 49.38% (64-64), making PUF instances harder to correlate. Unlike the unmodified design, where uniqueness remained well below 50%, Balance XOR pushes the system toward more distinct and secure responses.

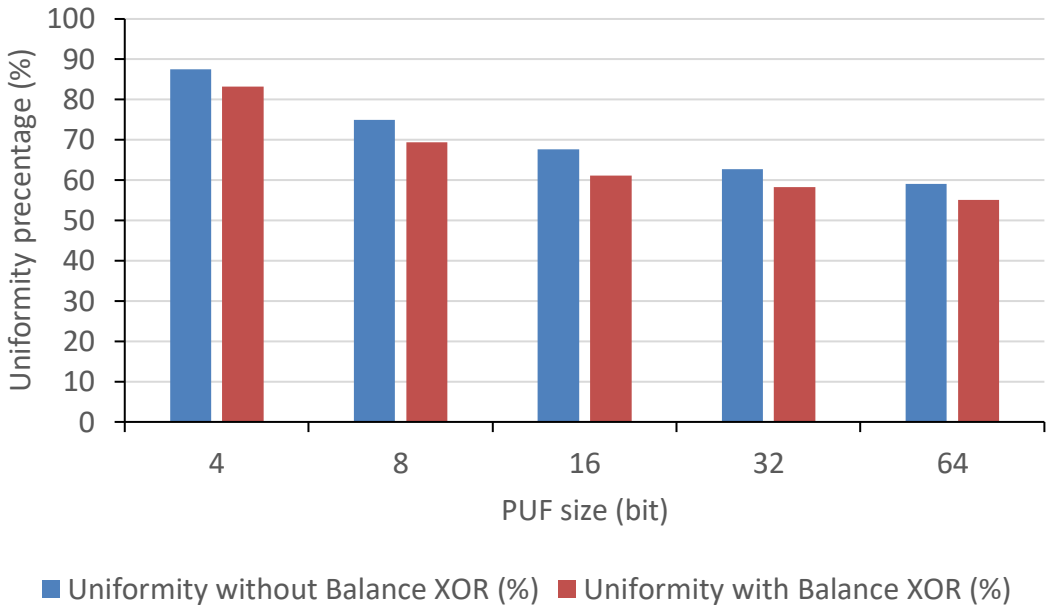


Figure 14. Diagram of Uniformity percentage across different PUF sizes for the design before and after the addition of Balance XOR

Figure 14 showcases the uniformity trend described in Table 2 in comparison to the one described in Table 1. As PUF size increases, uniformity decreases from 83.25% to 55.12%. The diagram highlights the sharpest decline between 4-bit and 8-bit PUF size, while subsequent reductions become more gradual. This confirms that higher challenge lengths improve randomness, making responses more balanced for security applications. Additionally, in the diagram the difference of the Uniformity percentage between the circuit before and after the addition is also shown visually to depict how the Balance XOR produces better Uniformity percentage across the different PUF sizes.

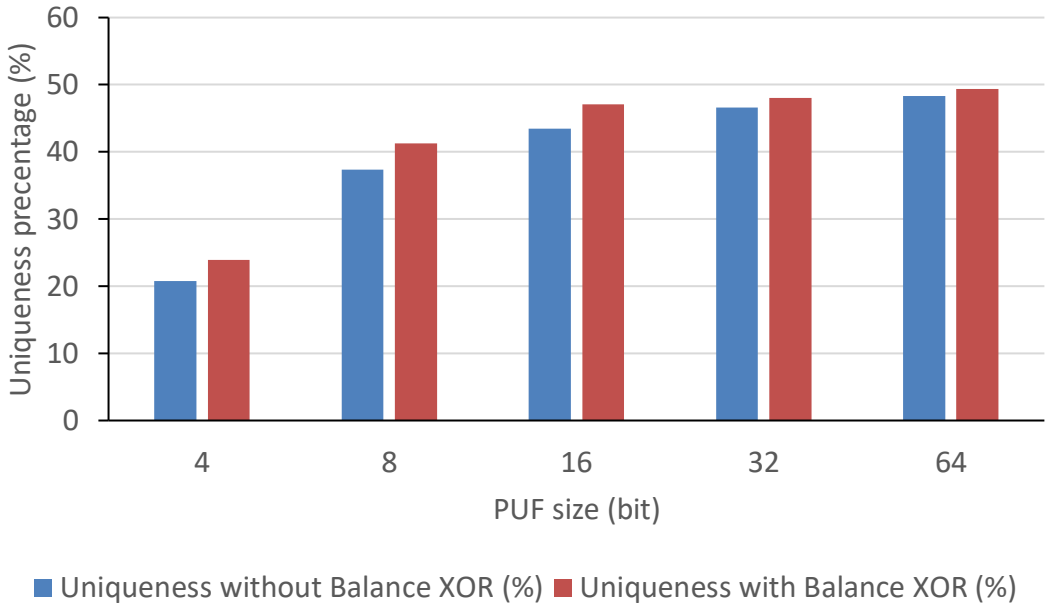


Figure 15. Diagram of Uniqueness percentage across different PUF size for the design before and after the addition of Balance XOR

In contrast to Figure 14 where Uniformity gets closer to the ideal 50%, in Figure 15 where the diagram of Uniqueness percentage across the different PUF sizes for the circuit before and after the design, After the 32 bit PUF sizes Uniqueness between the two circuits have little to no notable differences. For example for the 64-bit PUF size before the addition of Balance XOR, Uniqueness is at 48.33 %. However after the addition it is at 49.38 %, meaning that even for bigger PUF sizes there wouldn't be a point for Balance XOR to be added based on Uniqueness alone.

4.4 Resource Utilization

After the implementation step in Vivado, we can analyze the resource utilization of the circuit on the Artix-7 35T FPGA. The following tables and graphs present both the absolute resource usage and the corresponding utilization percentages.

	LUT	FF	LUTRAM	BRAM	IO	BUFG	MMCM
FPGA	20800	41600	9600	41600	50	32	5

Table 3. Total resources of the FPGA

Table 3 presents the total available resources in the FPGA Artix-7 35T, highlighting its logic, memory, I/O, and clocking capabilities. These resources define the computational and connectivity potential of the FPGA, making them essential for implementing various digital circuits and embedded systems.

1. Look-Up Tables (LUTs) – 20,800: LUTs are the fundamental building blocks of FPGA logic. They implement combinational logic, replacing traditional logic gates by storing predefined output values for specific input combinations.
2. Flip-Flops (FFs) – 41,600: Flip-flops are sequential logic elements used for data storage, state retention, and clock synchronization. They store binary values and are crucial for implementing registers and state machines.
3. LUTRAM – 9,600: Some LUTs can be configured as LUTRAM, providing small, distributed memory within the FPGA fabric. This memory is useful for high-speed, low-latency data storage, such as FIFOs and caches.
4. Block RAM (BRAM) – 41,600 bits: BRAM offers larger, dedicated memory blocks within the FPGA for storing data or implementing buffers. Unlike LUTRAM, BRAM supports structured memory architectures with efficient read/write operations.
5. I/O Pins (IO) – 50: These pins enable communication between the FPGA and external devices such as sensors, microcontrollers, and other peripherals. They support various signaling standards for flexible interfacing.
6. Global Clock Buffers (BUFG) – 32: BUFGs are dedicated clock distribution networks that ensure efficient low-skew clocking throughout the FPGA. They help maintain timing integrity across large designs.
7. Mixed-Mode Clock Managers (MMCM) – 5: MMCMs allow precise clock frequency synthesis, phase shifting, and jitter reduction. They enable designers to generate multiple clock domains, ensuring accurate timing for different sections of a circuit.

These resources collectively define the computational, memory, and interfacing capabilities of the FPGA, enabling the implementation of complex digital designs with efficient resource utilization and optimized performance.

	LUT	FF	LUTRAM	BRAM	IO	BUFG	MMCM
Without PUF	1514	1518	138	32	5	4	1

Table 4. Resource Utilization of the FPGA from the circuit without the PUF

Table 2 presents the resource utilization of the FPGA Artix-7 35T for the circuit without the PUF. The design occupies 1,514 LUTs and 1,518 Flip-Flops (FFs), representing the base logic and sequential elements required for the system before integrating the PUF. Additionally, 138 LUTs are configured as LUTRAM, while 32 bits of BRAM, 5 I/O pins, 4 global clock buffers (BUFGs), and 1 mixed-mode clock manager (MMCM) are utilized.

Since the values for LUTRAM, BRAM, IO, BUFG, and MMCM remain constant across different challenge and response configurations, they will not be discussed further below. The focus will instead be on LUT and FF utilization, as these resources change depending on the PUF configuration.

From this point onward, the tables and diagrams represent the resource utilization of the complete circuit, including the processor and the Multi-Instance Arbiter PUF without the Balance XOR that was discussed before. These results include key FPGA resource metrics such as Look-Up Tables (LUTs), Flip-Flops (FFs), and utilization percentages. The comparison between different challenge-response pair configurations highlights how the PUF affects the overall resource consumption of the system.

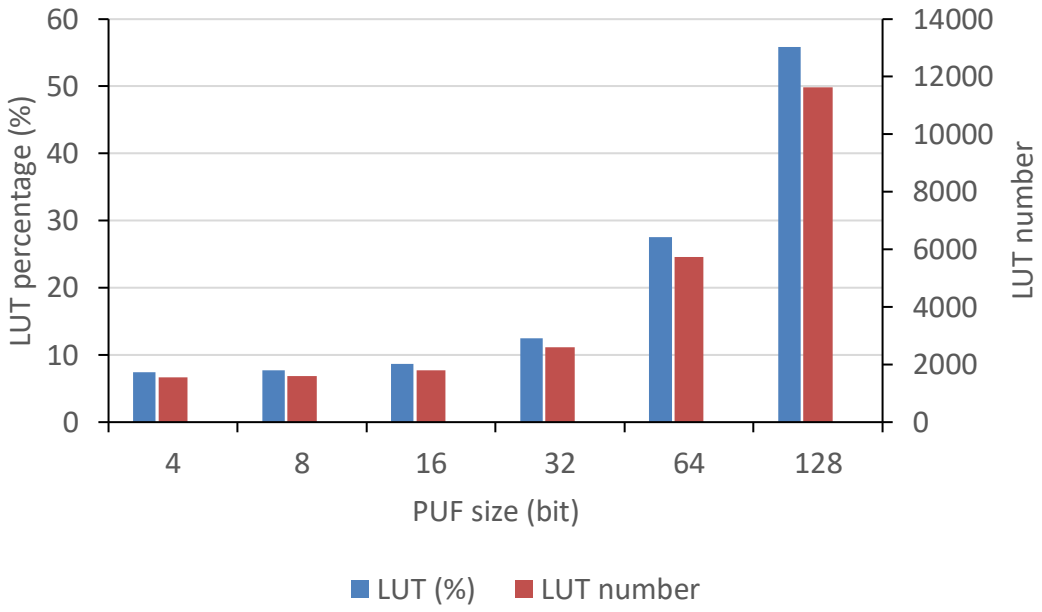
PUF size	Look-up Tables (LUT)	Flip Flop (FF)
4	1545	1545
8	1598	1561
16	1796	1593
32	2596	1689
64	5735	1785
128	11627	1861

Table 5. Look-Up Table (LUT) and Flip Flop (FF) Utilization of the FPGA across different PUF sizes for Arbitrary PUF

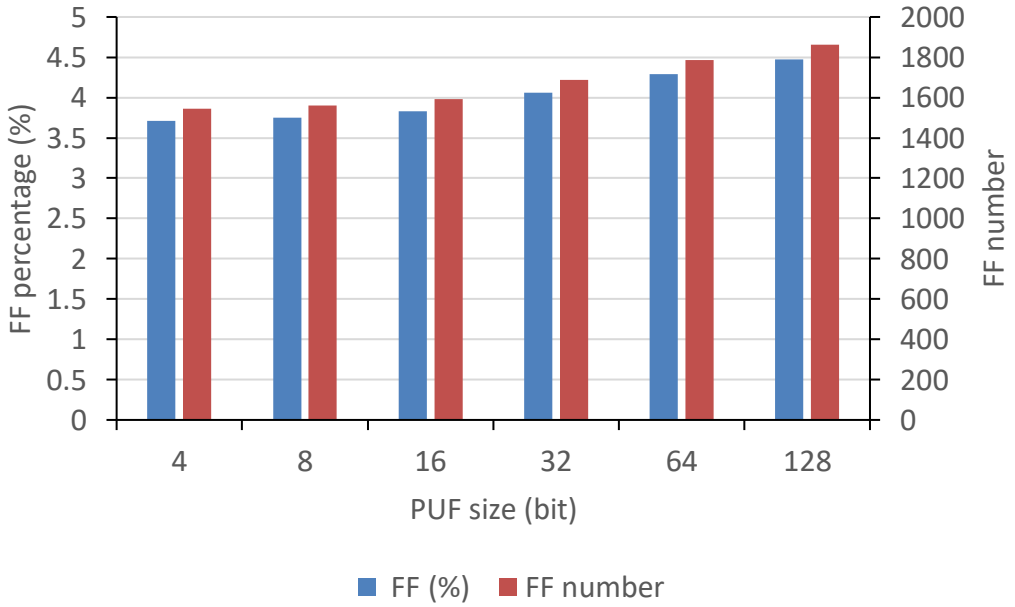
Table 5 presents the FPGA resource utilization in terms of Look-Up Tables (LUTs) and Flip-Flops (FFs) for various PUF sizes. It provides insights into how the complexity of the design scales as the PUF size increases.

A clear trend emerges where the number of required LUTs and FFs increases with longer PUF sizes, but the rate of growth differs between the two types of resources. LUT utilization shows a sharp rise, particularly for larger PUF size. For example, moving from a 4-bit PUF size (1545 LUTs) to a 128 –bit PUF size (11627LUTs) results in an approximately 7.5× increase in LUT usage.

In contrast, FF utilization exhibits a more gradual increase, growing from 1545 FFs in the smallest PUF size to 1861 FFs in the largest. This suggests that while the sequential elements (FFs) grow with increasing response length, their impact is less pronounced compared to LUTs. The relatively modest growth in FF utilization indicates that the design remains primarily LUT-driven, likely due to the combinational nature of the PUF logic and XOR operations. Notably, the LUT utilization jumps significantly when transitioning from 64-bit PUF size (5735 LUTs) to 128-bit PUF size, resulting in the highest LUT count (11,627).



a)



b)

Figure 16. FPGA a)LUT and b)FF utilization percentage of the complete circuit across different PUF sizes

Figure 16 illustrates the FPGA resource utilization trends in terms of LUT and FF percentages for different PUF sizes. As observed in Table 5, LUT utilization increases significantly with larger PUF size, reflecting the growing complexity of combinational

logic in the design. The diagram highlights this trend, showing a steep rise in LUT usage, especially for configurations beyond 32-bit challenges.

In contrast, FF utilization exhibits a much more gradual increase across different PUF sizes. The figure confirms that while FF usage grows slightly as PUF size increases, the overall percentage remains relatively stable compared to LUTs. This indicates that the design is primarily LUT-driven, with Flip-Flops contributing minimally to overall resource consumption.

By visualizing these trends, Figure 16 provides a clear representation of how FPGA resource allocation scales with increasing PUF size complexity, emphasizing the high LUT demand associated with larger challenge sizes.

PUF size	Look-up Tables (LUT)	Flip Flop (FF)
4	31	27
8	84	43
16	282	75
32	1082	171
64	4221	267
128	10113	343

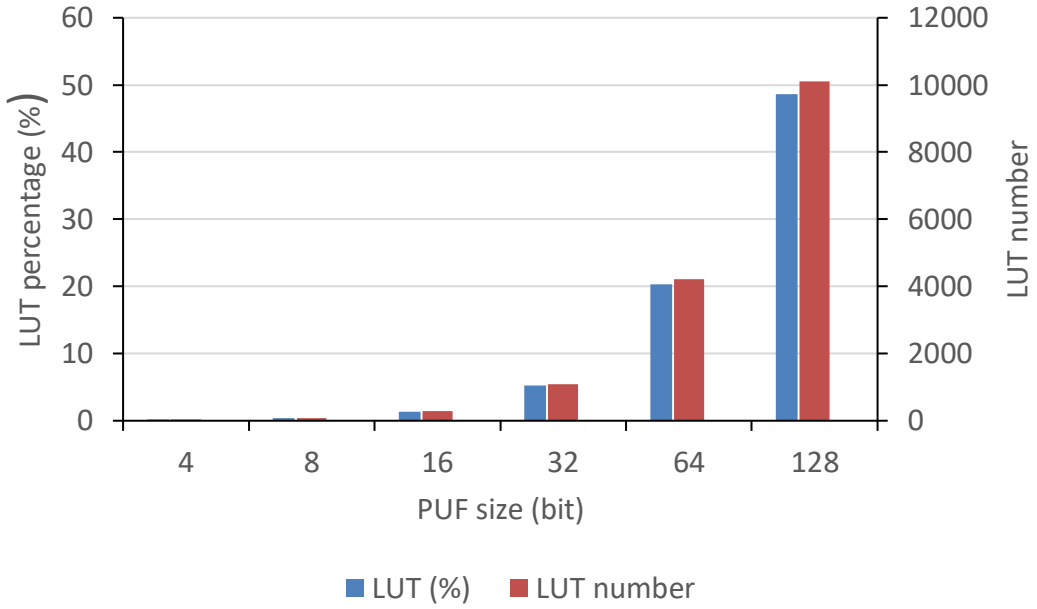
Table 6. PUF only resource utilization on the FPGA across different PUF sizes

Table 6 presents the resource utilization of the PUF core in terms of Look-Up Tables (LUTs) and Flip-Flops (FFs) for various PUF sizes. The table highlights how resource consumption scales as the PUF size increases.

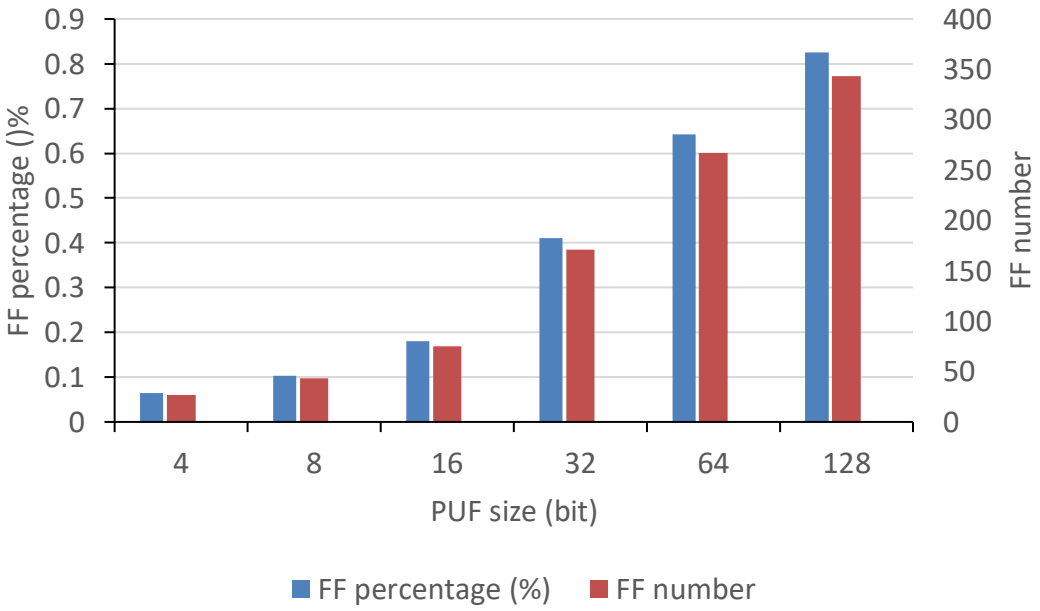
LUT utilization shows a significant rise with increasing PUF size, reflecting the growing complexity of the PUF logic. The smallest configuration (4-bit) requires only 31 LUTs, while the largest (128-bit) consumes 10,113 LUTs, demonstrating a substantial increase in combinational logic requirements. Notably, the jump from 32-bit (1,082 LUTs) to 64-bit (8,400 LUTs) marks a sharp escalation in resource demand, indicating that larger response sizes contribute significantly to LUT consumption.

FF utilization follows a more gradual growth pattern. While it increases with longer PUF size, the rise is less pronounced compared to LUTs. The FF count starts at 27 for 4-bit and reaches 343 for 128-bit, showing that the design remains primarily LUT-driven. The relatively moderate growth in FF usage suggests that the sequential elements play a secondary role compared to the combinational logic in the PUF's operation.

These findings emphasize that the complexity of the PUF is predominantly dictated by LUT requirements, with FF utilization remaining relatively stable. This distinction is crucial for understanding the scalability of the design on FPGA platforms.



a)



b)

Figure 17. FPGA a)LUT and b)FF utilization percentage of the PUF across different PUF sizes

Figure 17 shows LUT and FF utilization across different PUF sizes for the PUF core. LUT usage increases significantly with larger PUF size, particularly beyond 32-bit, reflecting

the growing combinational logic demands. In contrast, FF utilization rises more gradually, indicating that sequential elements play a smaller role in scaling complexity.

The diagram highlights how the PUF's resource needs grow with PUF size, emphasizing LUTs as the primary factor. This insight helps in optimizing FPGA implementations for efficiency and performance.

Following is the resource utilization of the circuit after the addition of the Balance XOR is presented. The following tables show the number of Look-Up Tables (LUTs) and Flip-Flops (FFs) used across different PUF sizes. Additionally, the diagrams illustrate the percentage of resource utilization relative to the total available FPGA resources. Comparisons are made between the circuit before and after adding the Balance XOR, as well as across different PUF sizes to analyze the impact on resource usage. Other FPGA resources such as LUTRAM, BRAM, IO, BUFG, and MMCM remain constant and will not be discussed further in this subsection.

PUF size	Look-up Tables (LUT)	Flip Flop (FF)
4	1559	1545
8	1637	1561
16	1952	1593
32	3145	1689
64	7899	1785

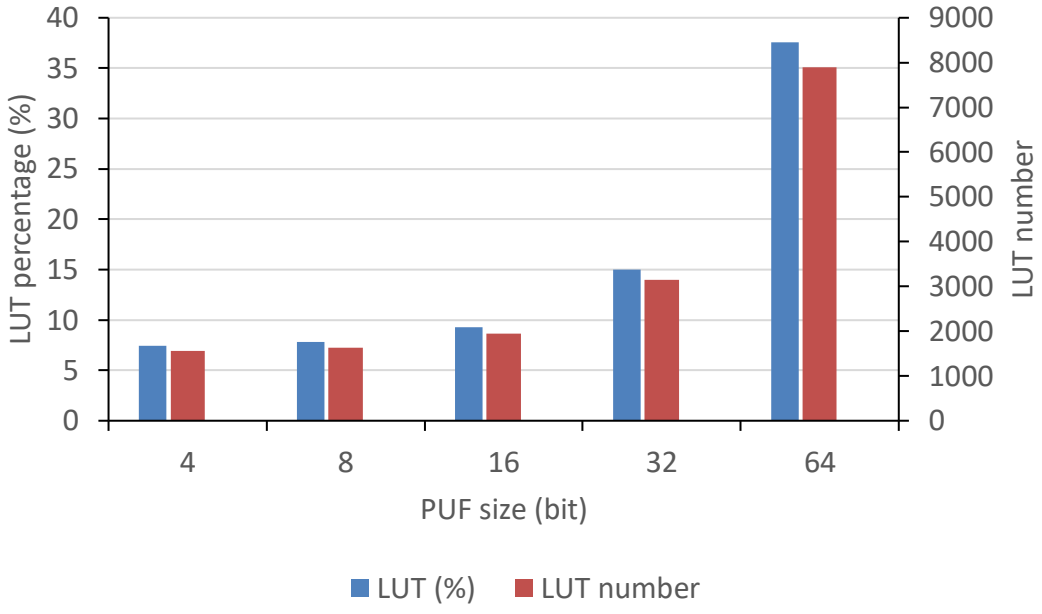
Table 7. Look-Up Table (LUT) and Flip-Flop (FF) Utilization of the FPGA across different PUF sizes Arbitrary PUF and the new XOR

This table presents the FPGA resource utilization after incorporating the Balance XOR into the design. It shows the Look-Up Table (LUT) and Flip-Flop (FF) counts for various PUF sizes, allowing for a comparison with the original implementation.

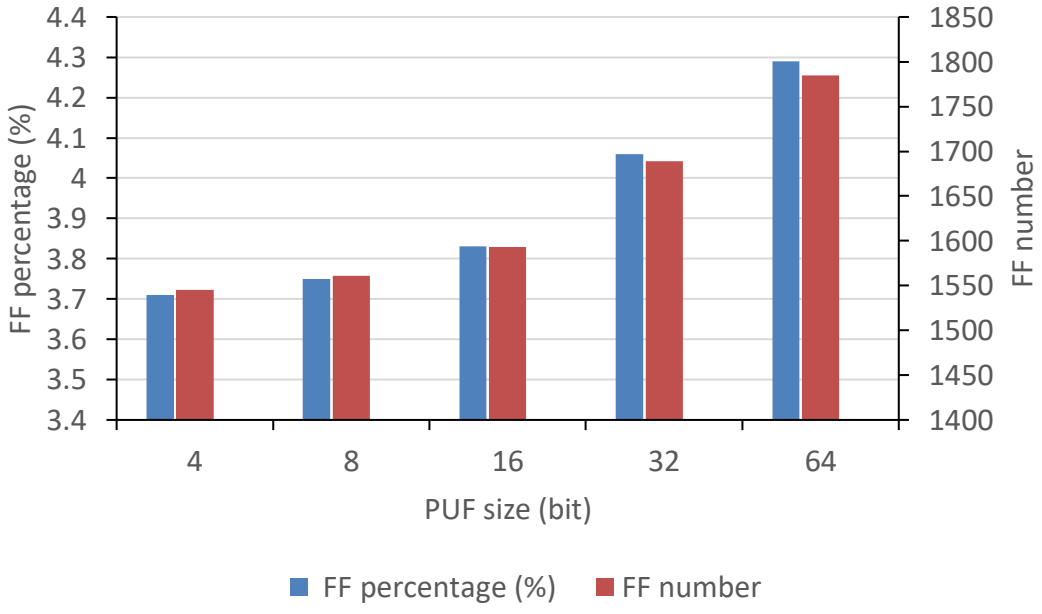
A key observation is the increased LUT utilization across all configurations, which is expected due to the additional XOR logic. The increase is particularly notable in larger PUF size. For example, the 64-bit configuration now requires 7899 LUTs compared to the previous 5735 LUTs.

In contrast, the FF utilization remains unchanged for most configurations. This indicates that while the XOR logic increases the combinational complexity of the design (affecting LUTs), it does not significantly impact the sequential elements (FFs), which remain largely tied to the challenge and response storage mechanisms.

Overall, the addition of the Balance XOR introduces a noticeable resource overhead, particularly in LUT usage, which may influence design choices when implementing this approach on FPGA hardware with limited resources.



a)



b)

Figure 18. FPGA a)LUT and b)FF utilization percentage of the complete circuit with the Balance XOR across different PUF sizes

Figure 18 presents the percentage utilization of LUTs and FFs across various PUF sizes following the addition of the Balance XOR. The diagram reveals a notable rise in LUT

usage, reflecting the increased combinational complexity introduced by the XOR logic. This effect is more pronounced in larger PUF sizes, where LUT consumption grows significantly.

Conversely, FF utilization remains largely stable, indicating that the Balance XOR primarily influences logic operations rather than sequential elements. The diagram highlights the trade-off between enhanced response properties and additional FPGA resource demands, which must be considered for efficient hardware implementation.

PUF size	Look-up Tables (LUT)	Flip Flop (FF)
4	45	27
8	123	43
16	438	75
32	1631	171
64	6385	267

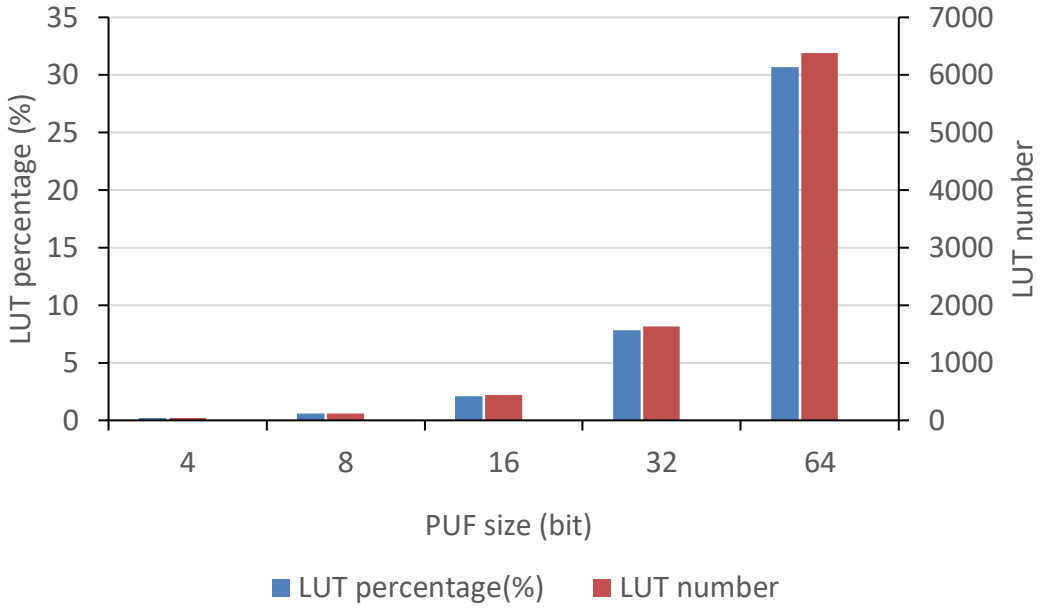
Table 8. PUF only utilization on FPGA across different PUF sizes after the addition of Balance XOR

Table 8 presents the FPGA resource utilization of the PUF module after integrating the Balance XOR, demonstrating the effect of the additional logic on LUT and FF usage. Compared to the original implementation, LUT utilization has increased across all PUF size configurations, reflecting the additional combinational complexity introduced by the XOR logic.

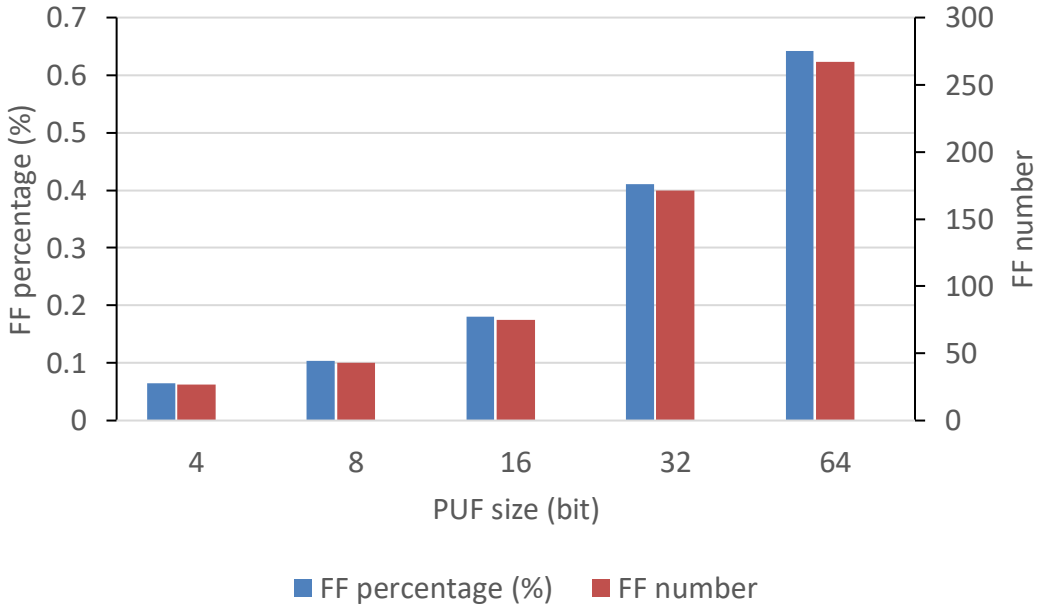
The impact of the Balance XOR is especially noticeable in larger PUF size configurations. For example, , the 64-bit configuration sees a significant rise from 4221 to 6385 LUTs, reinforcing how the XOR logic scales with increased PUF size. This trend highlights that as the PUF grows in complexity, the added combinational logic further amplifies resource consumption.

Conversely, FF utilization remains largely unchanged across all configurations, with values identical to those in the original design. This suggests that while the Balance XOR alters the combinational processing of responses, it does not impact the storage elements responsible for holding challenge and response data. The relatively stable FF count confirms that the sequential elements of the PUF remain unaffected, and the increase in resource usage is primarily due to the expansion of logic operations within the PUF.

These findings emphasize a trade-off between improved response properties and FPGA resource constraints. The additional XOR logic enhances uniformity and unpredictability but demands greater LUT utilization, which may influence design considerations when implementing the PUF on FPGAs with limited resources.



a)



b)

Figure 19. FPGA a)LUT and b)FF utilization percentage of the PUF with the Balance XOR across different PUF sizes

Figure 19 visualizes the LUT and FF utilization percentages across different PUF sizes for the PUF after implementing the Balance XOR. The diagram emphasizes the increasing demand for LUTs as the PUF size grows, with larger configurations showing a steeper rise in resource consumption.

Unlike LUTs, FF utilization remains nearly constant, suggesting that the Balance XOR's impact is confined to combinational logic rather than altering the sequential components. The diagram provides insight into how the design scales with the added logic and highlights the trade-offs between improved response properties and FPGA resource constraints.

Chapter 5. Conclusion

In this work, the design, implementation, and evaluation of an Arbiter PUF on FPGA as a hardware security primitive were presented. The Uniformity and Uniqueness of the traditional Arbiter PUF was analyzed across different PUF sizes. The introduction of the Balance XOR gate improved the uniformity of the original Arbiter PUF while maintaining its uniqueness. However, this enhancement came at the cost of increased resource utilization. Specifically, Look-Up Table (LUT) usage saw a significant rise, highlighting a trade-off between improving security properties and maintaining hardware efficiency.

Despite the added overhead, the Balance XOR implementation had minimal impact on Flip-Flop (FF) utilization, indicating that the additional logic primarily affected combinational operations rather than sequential elements. This balance between security enhancement and resource constraints is a critical consideration for deploying Arbiter PUFs in real-world FPGA-based trusted computing applications.

Further testing is required to fully evaluate the effects of the Balance XOR on the circuit's performance, stability, and resistance to various environmental and operational factors. Additionally, future work could explore optimization techniques to mitigate the resource overhead while preserving or further enhancing PUF characteristics. Investigating the resilience of the modified Arbiter PUF against machine learning-based attacks would also provide valuable insights into its security robustness.

Bibliography

[1]. **Pieter Maene, Johannes Götzfried, Ruan de Clercq, Tilo Müller, Felix Freiling, and Ingrid Verbauwhede.** Hardware-Based Trusted Computing Architectures for Isolation and Attestation.

[2]. **Luigi Coppolino, Salvatore D'Antonio, Giovanni Mazzeo, , Luigi Romano**. A Comprehensive Survey of Hardware-assisted Security: from the Edge to the Cloud.

[3]. **SONIA AKTER, KASEM KHALIL AND MAGDYBAYOUMI.** A Survey on Hardware Security: Current Trends and Challenges

[4]. **PAUL D. ROSERO-MONTALVO, ZSOLT ISTVÁN, and WILMAR HERNANDEZ.** A Survey of Trusted Computing Solutions Using FPGAs

[5]. **G. Edward Suh and Srinivas Devadas**. Physical Unclonable Functions for Device Authentication and Secret Key Generation

[6]. **Jeroen Delvaux, Roel Peeters, Dawu Gu and Ingrid Verbauwhede**. A Survey on Lightweight Entity Authentication with Strong PUFs

[7] **D. A. Patterson and J. L. Hennessy**, Computer Organization and Design: The Hardware/Software Interface, 5th ed. Morgan Kaufmann, 2013.

[8] **Xilinx**, "MicroBlaze Soft Processor Core," [Online]. Available: https://www.xilinx.com/products/design-tools/microblaze.html. [Accessed: 17-Feb-2025].

[9] **P. Chu**, *FPGA Prototyping by VHDL Examples: Xilinx MicroBlaze MCS SoC*, Wiley, 2017.

[10] **AMD-Xilinx**, "MicroBlaze Processor Reference Guide," UG984.

[11] **R. Tessier and W. Burleson,** "Reconfigurable Computing for Digital Signal Processing: A Survey," *Journal of VLSI Signal Processing*, vol. 28, no. 1–2, pp. 7–27, 2001.

[12] **Thomas Eisenbarth, Tim Güneysu, Christof Paar, Ahmad-Reza Sadeghi, Dries Schellekens, Marko Wolf**. Reconfigurable Trusted Computing in Hardware

[13] **Kusum Lata and Linga Reddy Cenkeramaddi**. FPGA-Based PUF Designs: A Comprehensive Review and Comparative Analysis

[14] **Alexander A. Ivaniuk and Siarhei S. Zalivaka**. FPGA Based Arbiter Physical Unclonable Function Implementation with Reduced Hardware Overhead

[15] **Ulrich Rührmair and Heike Busch and Stefan Katzenbeisser** . Strong PUFs: Models, Constructions and Security Proofs

[16] **Ulrich Rührmair, Jan Sölter, Frank Sehnke, Xiaolin Xu, Ahmed Mahmoud, Vera Stoy anova, Gideon Dror, Jürgen Schmidhuber, Wayne Burleson, and Srinivas Devadas**. PUF Modeling Attacks on Simulated and Silicon Data

[17] **S. HEMAVATHY , V. S. KANCHANA BHAASKARAN**. Arbiter PUF- A Review of Design, Composition, and Security Aspects

[18] **Nils Wisiol, Christoph Graebnitz, Marian Margraf, Manuel Oswald, Tudor A. A. Soroceanu, and Benjamin Zengin Moradi.** Why Attackers Lose: Design and Security Analysis of Arbitrarily Large XOR Arbiter PUFs

[19] **PHUONG HA NGUYEN, DURGA PRASAD SAHOO, RAJA TSUBHRA CHAKRABORTY, and DEBDEEP MUKHOPADHYAY.** Security Analysis of Arbiter PUF and Its Lightweight Compositions Under Predictability Test

[20] **Simranjeet Singh , Srinivasu Bodapati , Sachin Patkar , Rainer Leupers , Anupam Chattopadhyay , Farhad Merchant.** PA-PUF: A Novel Priority Arbiter PUF

[21] **Ji-Liang Zhang, Qiang Wu, Yi-Peng Ding, Yong-Qiang Lv, Qiang Zhou, Zhi-Hua Xia, Xing-Ming Sun, and Xing-Wei Wang.** Techniques for Design and Implementation of an FPGA-Specific Physical Unclonable Function

[22] **Christopher Vega.** Resource-efficient PUF Implementation Through FPGA Resource Re-utilization

[23] **Anita Aghaie, Amir Moradi.** TI-PUF: Toward Side-Channel Resistant Physical Unclonable Functions

[24] **Mario Barbareschi, Biagio Boi, Franco Cirillo, Marco De Santis and Christian Esposito.** Securing the Internet of Medical Things using PUF-based SSI Authentication

[25] **Jae W. Lee, Daihyun Lim, Blaise Gassend, G. Edward Suh, Marten van Dijk, and Srinivas Devadas.** A Technique to Build a Secret Key in Integrated Circuits for Identification and Authentication Applications

[26] **Athanasios Xynos, Vasileios Tenentes, Yiorgos Tsiatouhas.** SiCBit-PUF: Strong in-Cache Bitflip PUF Computation for Trusted SoCs

Appendix

Verilog modules

```
module Multi_instance_APUF_AXI #(
    parameter C_LENGTH = 64,  // Length of challenge in bits
    parameter R_LENGTH = 64
)(
    input wire       S_AXI_ACLK,     // AXI Clock
    input wire       S_AXI_ARESETN,  // AXI Reset
    input wire [31:0] S_AXI_AWADDR,  // Write Address
    input wire       S_AXI_AWVALID,  // Write Address Valid
    output reg       S_AXI_AWREADY,  // Write Address Ready
    input wire [31:0] S_AXI_WDATA,   // Write Data
    input wire [3:0]  S_AXI_WSTRB,   // Write Strobes
    input wire       S_AXI_WVALID,   // Write Data Valid
    output reg       S_AXI_WREADY,   // Write Data Ready
    output reg [1:0]  S_AXI_BRESP,   // Write Response
    output reg       S_AXI_BVALID,   // Write Response Valid
    input wire       S_AXI_BREADY,   // Write Response Ready
    input wire [31:0] S_AXI_ARADDR,  // Read Address
    input wire       S_AXI_ARVALID,  // Read Address Valid
    output reg       S_AXI_ARREADY,  // Read Address Ready
    output reg [31:0] S_AXI_RDATA,   // Read Data
    output reg [1:0]  S_AXI_RRESP,   // Read Response
    output reg       S_AXI_RVALID,   // Read Data Valid
    input wire       S_AXI_RREADY    // Read Data Ready
);

    // Internal Registers
    reg reg_ipulse;                  // Pulse signal to trigger the PUF
    reg [C_LENGTH-1:0] reg_challenge;    // Challenge register
    reg [R_LENGTH-1:0] reg_response;     // Response register

    wire [R_LENGTH - 1:0] wire_response;
    wire [R_LENGTH - 1:0] wire_responsen;

    // AXI Write Address Ready
    always @(posedge S_AXI_ACLK) begin
        if (!S_AXI_ARESETN) begin
            S_AXI_AWREADY <= 1'b0;
        end else begin
```

The module defines two parameters. C_LENGTH: Specifies the bit-width of the challenge input to the PUF. R_LENGTH: Specifies the bit-width of the response output from the PUF.

The module receives the following AXI signals as inputs:

- **Clock and Reset inputs.** S_AXI_ACLK: AXI clock signal. S_AXI_ARESETN: Active-low reset signal.
- **Write Channel Inputs.** S_AXI_AWADDR: Write address. S_AXI_AWVALID: Indicates a valid write address. S_AXI_WDATA: Write data. S_AXI_WSTRB: Write strobes (specifies which bytes in S_AXI_WDATA are valid). S_AXI_WVALID: Indicates that valid write data is available. S_AXI_BREADY: Indicates that the master is ready to receive a write response.
- **Read Channel Inputs**. S_AXI_ARADDR: Read address. S_AXI_ARVALID: Indicates a valid read address. S_AXI_RREADY: Indicates that the master is ready to receive read data.

The module provides the following AXI outputs:

- **Write Channel Outputs.** S_AXI_AWREADY: Indicates the module is ready to accept a write address. S_AXI_WREADY: Indicates the module is ready to accept write data .S_AXI_BRESP: Write response status (OKAY, SLVERR, etc.). S_AXI_BVALID: Indicates a valid write response.
- **Read Channel Outputs.** S_AXI_ARREADY: Indicates the module is ready to accept a read address. S_AXI_RDATA: Read data output. S_AXI_RRESP: Read response status .S_AXI_RVALID: Indicates valid read data.

The module defines the following internal registers. reg_ipulse: A signal used to trigger the PUF module. reg_challenge: Stores the challenge bits sent to the PUF. reg_response: Stores the response bits received from the PUF.

The module uses a wire for connecting the response output. wire_response: Carries the PUF's response before storing it in reg_response.

These components collectively implement an AXI-based interface for interacting with the PUF module using memory-mapped reads and writes.

The first always block is responsible for controlling S_AXI_AWREADY. When S_AXI_ARESETN is 0, S_AXI_AWREADY is also set to 0. Otherwise, S_AXI_AWREADY is assigned the result of the logical AND operation between S_AXI_AWVALID and the negation of S_AXI_AWREADY. This means that when the AXI reset is asserted (0), the system signals that the write address is not ready. In the normal operation

(S_AXI_ARESETN = 1), S_AXI_AWREADY toggles based on the validity of the write address.

The second always block is responsible for enabling the writing of data. When S_AXI_ARESETN is 0, meaning the AXI reset is asserted, S_AXI_WREADY is also set to 0, indicating that the system is not ready to accept write data. When the reset is deasserted (S_AXI_ARESETN = 1), S_AXI_WREADY follows the same logic as S_AXI_AWREADY, meaning it is assigned the result of S_AXI_WVALID AND NOT(S_AXI_WREADY), ensuring that write data is processed correctly.

The third always block is for the process of writing data. Specifically, while the AXI reset (S_AXI_ARESETN) is 0, the ipulse signal and the challenge are both set to 0. However, when the write data is ready (S_AXI_WREADY) and both the write data (S_AXI_WVALID) and write address (S_AXI_AWVALID) are valid (1), different cases apply depending on the challenge bit length (C_LENGTH) of the PUF. This allows the system to be parameterizable for a challenge bit length of 32 bits or lower, 64 bits, or 128 bits.

Since ipulse is a single-bit signal, it remains unchanged across all scenarios and is assigned the least significant bit (LSB) of the write address. Meanwhile, the reg_challenge register is divided into smaller segments, each receiving 32-bit portions of the AXI write data to accommodate different challenge bit lengths.

The fourth always block is responsible for handling the AXI write response. Specifically, when the AXI reset (S_AXI_ARESETN) is deasserted (0), the S_AXI_BVALID signal, which validates the write response, and the S_AXI_BRESP signal, which holds the write response status, are both set to 0.

If S_AXI_WREADY (indicating that the module can accept write data) and S_AXI_WVALID (indicating that write data is available) are both 1, while S_AXI_BVALID is 0, then S_AXI_BVALID is asserted (1), and S_AXI_BRESP is set to 0.

Finally, if S_AXI_BREADY (indicating that the master is ready to receive the write response) is asserted (1), then S_AXI_BVALID is deasserted (0).

The fifth always block is responsible for indicating that the AXI read address is ready. When the AXI reset is deasserted (0), the read address is not ready, meaning S_AXI_ARREADY is set to 0. In any other case, S_AXI_ARREADY is assigned the result of the logical AND between S_AXI_ARVALID and !S_AXI_ARREADY.

The sixth always block is responsible for handling read operations in the AXI interface. When the AXI reset (S_AXI_ARESETN) is deasserted (0**)**, the S_AXI_RVALID signal, which indicates valid read data, and S_AXI_RRESP, which represents the read response status,

are both initialized to 0. However, when a valid read address is received (S_AXI_ARVALID = 1) and the read address channel is ready (S_AXI_ARREADY = 1), but **r**ead data is not yet valid (!S_AXI_RVALID), the system proceeds to handle the read request.

At this point, different cases apply depending on the response bit length (R_LENGTH) of the PUF. The logic ensures that the read data is valid before sending it. This allows the system to support configurable challenge bit lengths of 32 bits or lower, 64 bits, or 128 bits. The response register (reg_response) is divided into smaller 32-bit segments, which are sequentially assigned to S_AXI_RDATA to accommodate different PUF response bit lengths.

Finally, once the master is ready to receive the read data (S_AXI_RREADY = 1), the system resets S_AXI_RVALID to 0, indicating that the transaction has completed.

The seventh always block defines the behavior of the reg_response register. When the AXI reset signal (S_AXI_ARESETN) is low (0), reg_response is also reset to 0. Otherwise, reg_response takes the value of wire_oresponse, which is the output of the PUF module.

Following this, the Multi-instance Arbiter PUF module is instantiated. The module has two parameters: C_LENGTH and R_LENGTH, which define the challenge and response sizes, respectively. The register reg_ipulse serves as the input pulse signal and is connected to the ipulse port of the PUF module. The register reg_challenge holds the input challenge signals and is connected to the challenge port of the PUF module. The wire wire_response is connected to the module's response output.

This ensures that the AXI interface properly interacts with the Multi-instance Arbiter PUF, enabling challenge-response operations.

```
`timescale 1ns / 1ps

(* dont_touch = "yes" *)
module Multi_instance_APUF
#(parameter C_LENGTH=64,
  parameter R_LENGTH=64)(
    input ipulse,
    input [C_LENGTH-1 : 0] challenge,
    output [R_LENGTH - 1 : 0] response
    );

 //   (* dont_touch = "yes" *) wire temp;

    generate
    genvar i;
    for (i =0; i < R_LENGTH; i = i +1)
    begin
    (* dont_touch = "yes" *)
    arbiter_puf #(.ID(i),.C_LENGTH(C_LENGTH)) inst_arbiter_puf(
        .ipulse(ipulse),
        .challenge(challenge),
        .response(response[i])
    );

    end

    endgenerate

endmodule
```

Textbox 2. Multi-instance Arbiter PUF Verilog code

In Textbox 2, the Verilog code shown corresponds to the Multi-instance Arbiter PUF module. This module is responsible for generating multiple instances of the Arbiter PUF using a for-generate loop.

The inputs to this module are:

- **ipulse** – The input pulse signal that is fed into each instance of the Arbiter PUF (arbiter_puf).
- **challenge** – A challenge input vector, which is passed to the challenge input of each Arbiter PUF instance.

The output of this module is:

- **response** – A response vector where each element, response[i], represents the output of an individual Arbiter PUF instance.

The for-generate loop iterates R_LENGTH times, creating R_LENGTH independent instances of the Arbiter PUF module. Each generated instance takes the same ipulse and challenge signals as inputs. The output response[i] captures the response from the i-th Arbiter PUF instance, effectively generating a set of responses for different instances. Each Arbiter PUF instance processes the same challenge (challenge), but due to physical variations in real hardware, different instances may produce different responses.

```
`timescale 1ns / 1ps

(* dont_touch = "yes" *)
module arbiter_puf
#(parameter C_LENGTH=32, ID = 0)(
input ipulse,
input [C_LENGTH-1:0] challenge,
output response

   );
   wire delay_line_out_1;
   wire delay_line_out_2;

   (* dont_touch = "yes" *)
   delay_line  #(
      .C_LENGTH(C_LENGTH)
   ) inst_delay_line (
   .ipulse(ipulse),
   .challenge(challenge),
   .out_1(delay_line_out_1),
   .out_2(delay_line_out_2)
   );
  (* dont_touch = "yes" *)
   dff inst_dff1 (
   .id(odelay_line_out_1),
   .clk(odelay_line_out_2),
   .oq(response)
   );
endmodule
```

Textbox 3. Arbiter PUF Verilog code

In Textbox 3, the Verilog code shown corresponds to the Arbiter PUF module. This module integrates the Delay Line and the Arbiter, which in this case is implemented using a D Flip-Flop (DFF) instead of a latch for the sake of simplicity and predictability. This choice does not affect the overall functionality of the PUF.

The inputs to this module are:

- **ipulse** – A pulse signal that is connected to the ipulse input of the Delay Line module (delay_line).
- **ichallenge** – A challenge input that is passed directly to the ichallenge input of the Delay Line module.

The outputs of the Delay Line module are:

- **odelay_line_out_1** and **odelay_line_out_2**, which serve as the two signal paths that propagate through the delay elements.

These outputs are then directly fed into the D Flip-Flop (dff**)**, where:

- **odelay_line_out_1** is connected to the D input of the flip-flop.
- **odelay_line_out_2** is connected to the clock input of the flip-flop.

Finally, the D Flip-Flop determines the output response (oresponse), which represents the final PUF response based on the signal race through the delay path.

```
`timescale 1ns / 1ps

(* dont_touch = "yes" *)
module delay_line
#(parameter C_LENGTH=32)(
input ipulse,
input [C_LENGTH - 1 : 0] challenge,
output out_1,
output out_2

  );

  wire  [C_LENGTH - 1 : 0] net;
  wire  [C_LENGTH - 1 : 0] net2;
  assign net [0] =ipulse;
  assign net2 [0] =ipulse;

  generate
  genvar i;
  for (i =0; i < C_LENGTH-1; i = i +1)
  begin

  (* dont_touch = "yes" *)
   mux inst_mux_1(
  .ia(net[i]),
  .ib(net2[i]),
  .sel(challenge[i]),
  .out(net[i+1])
  );



  (* dont_touch = "yes" *)
   mux inst_mux_2(
  .ia(net2[i]),
  .ib(net[i]),
  .sel(challenge[i]),
  .out(net2[i+1])
  );
```

Textbox 4. Delay Line Verilog code

In Textbox 4, the Verilog code for the Delay Line module is presented. The design leverages a for-generate loop to instantiate and connect multiple switch blocks dynamically.

- The net and net2 signals represent the internal wiring of the delay line. Initially, both are assigned to ipulse (the input pulse).
- Inside the generate loop, each switch block is created using two 2-to-1 multiplexers (MUXes). These multiplexers determine the propagation path of the pulse based on the challenge bits (challenge).
- net[i] and net2[i] act as intermediate signals, passing data from one switch block to the next.
- A temporary wire (temp) is introduced inside the generate loop, computing the Balance XOR of net[i] and net2[i] at each stage, though it is not used in this version of the module.
- The final outputs (out_1 and out_2) correspond to the last elements of net and net2, which are then fed into the Arbiter (a D Flip-Flop in this case).
- At the end of the module, additional temporary wires (temp1 and temp2) are introduced: temp1 computes the Balance XOR of net[C_LENGTH-1] and net2[C_LENGTH-1]. temp2 computes the Balance XOR of out_1 and out_2.

These values are not utilized in the current module but are kept for potential future modifications. This modular implementation allows for easy scalability, as the **le**ngth of the delay line (C_LENGTH) can be adjusted to modify the complexity and uniqueness of the PUF.

```
`timescale 1ns / 1ps


(* dont_touch = "yes" *)
module dff(
   input id,
   input clk,
   output reg oq
   );



   always @ (posedge clk)
   begin
      oq <= id;
   end
endmodule
```

Textbox 5. D Flip Flop code

In Textbox 5, the Verilog code for the D Flip-Flop (DFF) and the 2-to-1 Multiplexer (MUX) is presented.

The D Flip-Flop (DFF) has two inputs and one output:

- **clk** is the clock signal that triggers changes at the rising edge.
- **id** is the data input, whose value is stored in the flip-flop.
- **oq** is the output, which takes the value of id at the rising edge of clk.

```
`timescale 1ns / 1ps


(* dont_touch = "yes" *)
module mux
#(parameter ID = 0)(
   input ia,
   input ib,
   input sel,
   output out
   );
   assign out = (sel) ? ia : ib;
endmodule
```

Textbox 6 2 to 1 Multiplexer Verilog

In Textbox 5, the Verilog code for the 2-to-1 Multiplexer (MUX) is presented.

The Multiplexer (MUX) has three inputs:

- **sel** (selector input) determines which of the two inputs is forwarded to the output.
- **ia** and **ib** are the two possible input values.
- The output **out** takes the value of either ia or ib, depending on the value of sel.

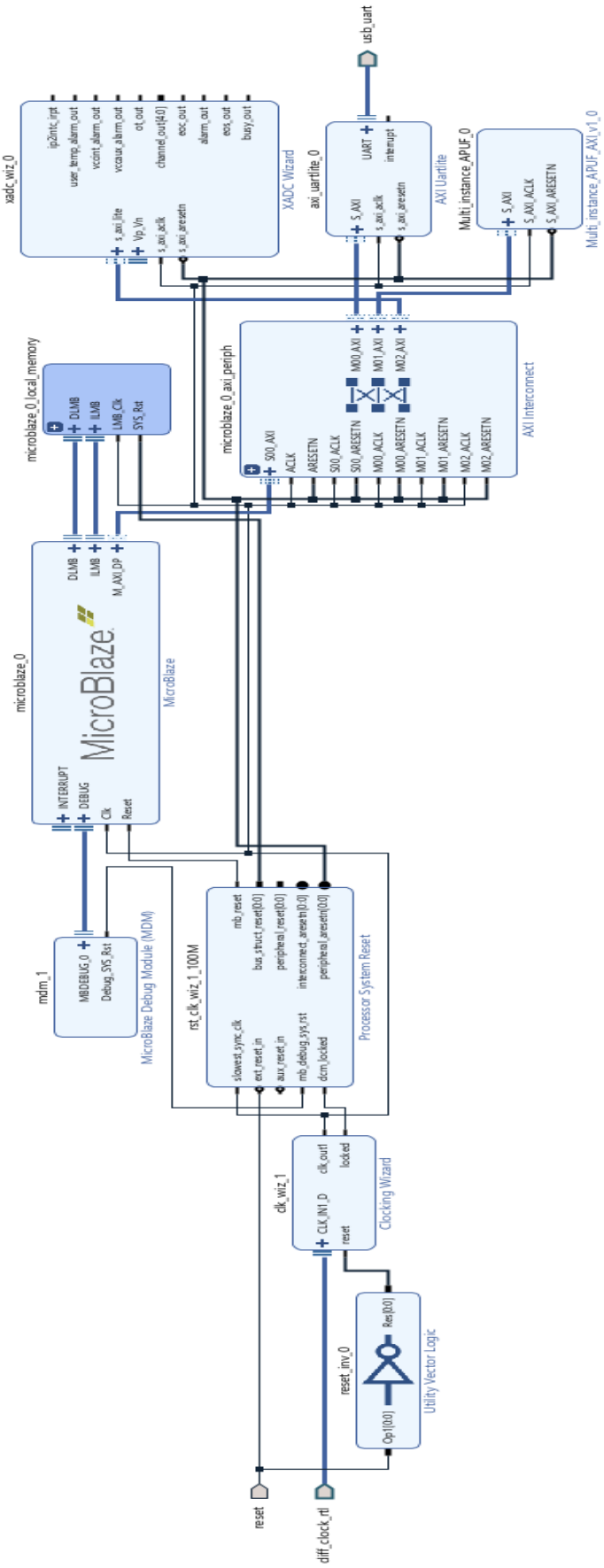


Figure 10 Final Block Design

The final block design, including all major components, is shown in Figure 10. In addition to the **MicroBlaze** processor and the components previously discussed, several other essential blocks are present in the design: AXI Interconnect, MicroBlaze Local Memory (microblaze_0_local_memory), MicroBlaze Debug Module, Processor System Reset, Clocking Wizard**,** and Utility Vector Logic.

- The **AXI Interconnect** enables direct communication between the MicroBlaze microprocessor and peripheral components such as the AXI UART, PUF module**,** and XADC Wizard, ensuring efficient data transfer by managing multiple AXI-based connections.
- The **MicroBlaze Local Memory** provides on-chip memory for the processor to store instructions and data, allowing it to function independently without external memory.
- The **MicroBlaze Debug Module** facilitates debugging by enabling features such as breakpoints, real-time status monitoring, and processor control during execution.
- The **Processor System Reset** module ensures that all components start in a known state by handling reset signals and synchronizing them across the design.
- The **Clocking Wizard** generates and manages the required clock signals for different components in the system, ensuring they operate at appropriate frequencies.
- The **Utility Vector Logic** block is used to process control signals. In this design, its input is connected to an external input pin, and its output is connected to the reset input of the Clocking Wizard, allowing external control over the clock reset functionality.

Embedded C code

```c
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "xil_io.h"
#include "xil_printf.h"
#include "puf_driver.h"

#define chall_number 3


int main()
{
    init_platform();
    u64 challenge_array[chall_number]={0xA7808D288B771F80,
             0x4FAD8CD61663915A,
             0x20D14C52E36B21F6,
             };

    u64 oresponse;


    for (int i=0; i<chall_number ; i++) {

       write_challenge(challenge_array[i]);

       oresponse = read_response();

       printf(" 0x%016llX\n", oresponse);
       for (volatile int delay = 0; delay < 500000; delay++);

    }


    cleanup_platform();
    return 0;
}
```

Textbox 7. Embedded C code, main program

In Textbox 7, the main function of the test program is presented. Specifically, this version is designed for a 64-bit challenge and a 64-bit response. Due to Vitis' capabilities, approximately 400 challenges can be processed at the same time. However, fewer challenges are shown in the figure to better illustrate the code's functionality and structure. The chall_number variable defines how many challenges are used in a single run and is later referenced in the for-loop to ensure each challenge is processed sequentially, with a corresponding response generated when running the program on

the FPGA. For cases where fewer than 64 challenges are needed, u32 is used instead of u64.

The write_challenge and read_response functions were added to the generic driver of the PUF instead of creating a custom driver from scratch due to issues with Vivado. To facilitate platform and application building, a Makefile was used. These functions handle writing challenges to the PUF module and reading back the generated responses. Once write_challenge and read_response are executed, the program prints the oresponse value returned by read_response to the Vitis terminal. Additionally, an inner for loop introduces a delay between challenges to ensure the FPGA has enough time to process each challenge and generate the correct response.

```
#include "puf_driver.h"
#include "xil_io.h"  // Xilinx I/O functions
#include "xparameters.h"  // For PUF base address definition

void write_challenge(u64 challenge) {
       u32 challenge_low, challenge_high;
       challenge_low=challenge;
       challenge_high=challenge>>32;

    Xil_Out32(PUF_BASEADDR + PUF_CHALLENGE_REG_OFFSET1, challenge_low);
    Xil_Out32(PUF_BASEADDR + PUF_CHALLENGE_REG_OFFSET2, challenge_high);
    Xil_Out32(PUF_BASEADDR + PUF_PULSE_REG_OFFSET, 1);
    for (volatile int delay = 0; delay < 10000; delay++);
    Xil_Out32(PUF_BASEADDR + PUF_PULSE_REG_OFFSET, 0);
}

u64 read_response() {
       u64 response_low, response_high;

       response_low = Xil_In32(PUF_BASEADDR + PUF_RESPONSE_REG_OFFSET1);
       response_high= Xil_In32(PUF_BASEADDR + PUF_RESPONSE_REG_OFFSET2);

    return (response_high<<32) | response_low;
}
```

Textbox 8. puf_driver.c file

In Textbox 8, the code of the puf_driver.c file, which contains the write_challenge and read_response functions, is presented.

- The write_challenge function is responsible for sending challenges to the PUF module. It writes the challenge values to specific addresses in the AXI wrapper, as defined in the Verilog implementation. After writing the challenge, the

function sends an input pulse to trigger the response generation. A brief delay is introduced between setting the pulse high (1) and resetting it to low (0) to ensure proper operation.

Since the AXI data width is 32 bits, a 64-bit challenge must be split into two 32-bit parts. The lower 32 bits are stored in challenge_low, and the upper 32 bits are stored in challenge_high. The function then writes these values separately using Xil_Out32(), a Xilinx function that performs memory-mapped I/O writes. If the challenge length is 32 bits or lower, the challenge_high part is unnecessary, so the function argument changes from u64 to u32, and the second Xil_Out32() call is removed.

- The read_response function retrieves the output response from the PUF. Like write_challenge, it handles 64-bit responses by reading two 32-bit parts separately, since the AXI read data width is also 32 bits. The Xil_In32() function, which performs memory-mapped I/O reads, is used to fetch the lower and upper parts of the response. The function then reconstructs the full 64-bit response by shifting response_high left by 32 bits and combining it with response_low. If the response length is 32 bits or less, response_high is omitted, and only response_low is returned.

```
#ifndef PUF_DRIVER_H
#define PUF_DRIVER_H

#include <stdint.h>

#define PUF_BASEADDR XPAR_MULTI_INSTANCE_APUF_0_BASEADDR

#define PUF_PULSE_REG_OFFSET 0x00
#define PUF_CHALLENGE_REG_OFFSET1 0x08
#define PUF_CHALLENGE_REG_OFFSET2 0x18
#define PUF_RESPONSE_REG_OFFSET1  0x10
#define PUF_RESPONSE_REG_OFFSET2  0x20

typedef unsigned long u32;
typedef unsigned long long u64;


void write_challenge(u64 challenge);
u64 read_response();

#endif  // PUF_DRIVER_H
```

Textbox 9. puf_driver.h header file

In Textbox 9, the code of the puf_driver.h file is presented. This header file defines the necessary elements for interfacing with the PUF module. It includes function declarations for write_challenge and read_response, which are used by the main program. Additionally, it defines the memory addresses that these functions use to communicate with the PUF via the AXI interface.

The base address of the PUF is defined as 0x80000000 (or XPAR_MULTI_INSTANCE_APUF_0_BASEADDR, which should be replaced with the correct base address). The remaining address offsets correspond to specific registers in the PUF's AXI wrapper:

- PUF_PULSE_REG_OFFSET: Controls the input pulse signal.
- PUF_CHALLENGE_REG_OFFSET1 and PUF_CHALLENGE_REG_OFFSET2: Store the challenge input values.
- PUF_RESPONSE_REG_OFFSET1 and PUF_RESPONSE_REG_OFFSET2: Hold the response values that will be read by the system.

Since the AXI interface operates on 32-bit data, a 64-bit challenge must be split into two 32-bit parts, which are stored at PUF_CHALLENGE_REG_OFFSET1 and PUF_CHALLENGE_REG_OFFSET2. Similarly, the response is retrieved in two parts using PUF_RESPONSE_REG_OFFSET1 and PUF_RESPONSE_REG_OFFSET2.

The file also defines custom data types for 32-bit (u32) and 64-bit (u64) values:

- typedef unsigned long u32;
- typedef unsigned long long u64;

If the challenge size is 32 bits or lower, only PUF_CHALLENGE_REG_OFFSET1 is used, and the second register is unnecessary. In that case, the addressing aligns with the configuration set in the AXI wrapper module, which was discussed earlier.

Python programs

```
def generate_challenges(num_challenges, challenge_length):
    challenges=[]
    hex_length = challenge_length // 4  # Convert bit length to hex
length
    for i in range(num_challenges):

 challenges.append(format(random.getrandbits(challenge_length),
f'0{hex_length}X') )

    return challenges
```

Textbox 10. Challenge generation function from python file

In Textbox 10, the code for challenge generation is presented. The function generate_challenges takes two integer inputs:

- num_challenges: The total number of challenges the user wants to create.
- challenge_length: The length of each challenge in bits.

When the function is called, it performs the following steps:

a. Initializes an empty list called challenges.
b. Calculates hex_length, which determines how many hexadecimal digits are needed to represent challenge_length bits (challenge_length // 4). Since each hex digit represents 4 bits, dividing by 4 gives the exact number of hex characters needed.
c. Generates num_challenges random binary values of length challenge_length using random.getrandbits(challenge_length).
d. Converts each binary value to a zero-padded uppercase hexadecimal string of length hex_length and appends it to the list.
e. Finally, the function returns the list of hexadecimal challenges.

Finally, the challenges were stored in a text file and grouped into sets of 400 to facilitate the process of running the embedded C program on the FPGA

```python
def calculate_uniformity(responses):
    uniformity = []
    for r in responses:
        total_bits = len(r)
        ones_count = r.count('1')
        u = (ones_count / total_bits)
        uniformity.append(u)
    total_uniformity = (sum(uniformity) / len(uniformity)) * 100
    return total_uniformity
```

Textbox 11. Uniformity calculation function from python file

In Textbox 11, the code for calculating Uniformity is presented. The function calculate_uniformity takes a list of Strings that are the responses as input and performs the following steps:

1. For each response in the list, count the number of '1'**s** and divide it by the total number of bits in the response to calculate the fraction of '1's.
2. Append this fraction to the uniformity list.
3. Compute the final Uniformity by summing all elements in the uniformity list, dividing by the total number of responses, and multiplying by 100 to express it as a percentage.

The final output represents the average Uniformity across all challenge-response pairs.

```python
def calculate_uniqueness(responses, segment_length):
    n = len(responses)
    l = len(responses[0])  # Total length of each response (e.g., 64 bits)

    # Ensure segment length divides response length evenly
    if l % segment_length != 0:
        raise ValueError("Segment length must evenly divide response length.")

    num_segments = l // segment_length  # Total segments per response
    total_hd = 0
    total_comparisons = 0

    # Iterate over each segment position
    for seg in range(num_segments):
        # Extract the segment from each response
        segments = []
        for response in responses:
            segment = response[seg * segment_length : (seg + 1) * segment_length]
            segments.append(segment)

        # Pairwise comparison of segments
        for i in range(n):
            for j in range(i + 1, n):
                hd = sum(a != b for a, b in zip(segments[i], segments[j]))
                total_hd += hd
                total_comparisons += segment_length

    # Normalize by total bits compared
    uniqueness = (total_hd / (total_comparisons)) * 100
    return uniqueness
```

Textbox 12. Uniqueness calculation function from python file

In Textbox 12, the code for calculating Uniqueness is presented. The function calculate_uniqueness takes two inputs:

1. A list of strings representing the responses.

2. An integer specifying the segment length, which will be used to divide each response into smaller segments.

Why Use a Multi-instance Arbiter PUF Instead of Two Separate PUFs?

Traditionally, Uniqueness is calculated by comparing responses from two physically separate PUFs. However, in this case, a single Multi-instance Arbiter PUF is used to simulate multiple instances. This is valid because:

- Each response from a Multi-instance Arbiter PUF originates from a different arbiter circuit within the design, ensuring independent response variations, similar to having separate PUFs.
- Each arbiter circuit introduces different path delays, leading to unique challenge-response behavior for each instance.
- The segmentation approach allows pairwise comparisons between different segments of the response, effectively mimicking inter-device comparisons.
- This method saves hardware resources by eliminating the need for multiple physical PUFs, making the evaluation more efficient.

Since each segment acts as an independent response from a unique arbiter instance, comparing them provides a valid Uniqueness measurement.

Uniqueness Calculation Process

The function follows these steps:

1. Check if the segment length divides the response length evenly. If not, an error is raised to ensure valid segmentation.
2. Determine the total number of segments per response.
3. Extract segments from each response and store them in a list.
4. Compute pairwise Hamming Distance (HD) between segments from different responses. For each bit in a segment, HD increases if the bits differ.
5. Accumulate total HD and count the number of bit comparisons made.
6. Compute the final Uniqueness as:

$$Unqueness = \frac{Total\ HD}{Total\ Comparisons} * 100$$

This percentage represents the average difference between PUF instances, with an ideal value close to 50%.