UNIVERSITY OF PIRAEUS

School of Information and Communication Technologies

Department of Informatics

Thesis

Thesis Title: Τίτλος Διατριβής:	**(Implementation of a key generator system using Ring Oscillator PUFs)** (Υλοποίηση ενός συστήματος παραγωγής κλειδιών με την χρήση PUF ταλαντωτή δακτυλίου)
Student's name-surname:	**(Nikolaos Pilichos)**
Father's name:	**(Christos)**
Student's ID No:	**(Π22144)**
Supervisor:	**Mihalis Psarakis, Professor**

June 2026/ Ιούνιος 2026

Copyright ©

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή μου για την καθοδήγηση, τις πολύτιμες συμβουλές και τη συνεχή υποστήριξη που μου παρείχε καθ' όλη τη διάρκεια εκπόνησης της παρούσας πτυχιακής εργασίας.

Περίληψη

Η ραγδαία αύξηση των εφαρμογών του Διαδικτύου των Πραγμάτων (IoT) έχει εισαγάγει δισεκατομμύρια διασυνδεδεμένες συσκευές, δημιουργώντας σημαντικές προκλήσεις στη διασφάλιση της εμπιστευτικότητας, της ακεραιότητας των δεδομένων και της αυθεντικοποίησης συσκευών. Οι παραδοσιακοί μηχανισμοί ασφαλείας βασίζονται συχνά στην ασφαλή αποθήκευση κρυπτογραφικών κλειδιών, τα οποία όμως είναι ευάλωτα σε φυσικές επιθέσεις, κλωνοποίηση και εξαγωγή κλειδιών, ιδιαίτερα σε περιβάλλοντα IoT με περιορισμένους πόρους. Για την αντιμετώπιση αυτών των προκλήσεων, οι Φυσικά μη Κλωνοποιήσιμες Συναρτήσεις (PUFs) έχουν αναδειχθεί ως μια πολλά υποσχόμενη υλικοκεντρική λύση, η οποία παράγει κρυπτογραφικά χρήσιμη πληροφορία παραμένοντας παράλληλα ανθεκτική στις προαναφερθείσες επιθέσεις. Η παρούσα εργασία παρουσιάζει τον σχεδιασμό και την υλοποίηση ενός συστήματος παραγωγής κλειδιών βασισμένου σε PUF Ταλαντωτή Δακτυλίου (Ring Oscillator – RO), εμπνευσμένου από την αρχιτεκτονική που προτείνεται στη σχετική βιβλιογραφία [1]. Η προτεινόμενη αρχιτεκτονική παράγει κρυπτογραφικά κλειδιά μέσω ενός μηχανισμού ζευγών ερωτήματος-απόκρισης (Challenge-Response Pairs), σε συνδυασμό με στάδια μετεπεξεργασίας που διασφαλίζουν τη μοναδικότητα και την επαναληψιμότητα υπό διαφορετικές περιβαλλοντικές συνθήκες. Παράλληλα με την υλοποίηση σε υλικό του γεννήτριου κλειδιών, η εργασία περιλαμβάνει και λογισμικό διεπαφής που υλοποιείται στο υποσύστημα επεξεργασίας (Processing System) της χρησιμοποιούμενης πλακέτας. Τέλος, παρουσιάζεται ένα σύνολο σεναρίων αξιολόγησης και εξειδικευμένων εργαλείων που αναπτύχθηκαν στο πλαίσιο της εργασίας.

Επιστημονική Περιοχή: Ασφάλεια Υλικού / Ενσωματωμένα Συστήματα (Hardware Security / Embedded Systems)

Λέξεις Κλειδιά: PUFs, Ταλαντωτές Δακτυλίου, Παραγωγή Κλειδιών, FPGA, Ασφάλεια Υλικού

Abstract

The rapid rise of internet of things (IoT) applications has introduced billions of interconnected devices, creating significant challenges in ensuring data confidentiality, integrity and device authentication. Traditional security mechanisms often rely on secure storage of cryptographic keys that can be vulnerable to physical attacks, cloning and key extraction, particularly in resource-constrained IoT environments. To address these challenges, physical unclonable functions have emerged as a promising hardware solution that generates cryptographically useful information while being resistant to the aforementioned attacks. This work presents the design and implementation of a Ring Oscillator (RO) PUF key generator based on the design proposed in [1]. The proposed architecture produces cryptographic keys based on a challenge-response pair mechanism, combined with post-processing to ensure uniqueness and reproducibility under various environmental conditions. Along with the hardware implementation of the key generator, this work also includes a software interface implemented in the processing system of the used board. Furthermore, a set of evaluation scripts and custom tools is also presented as part of this work.

Scientific Area: Hardware Security / Embedded Systems
Key words: PUFs, Ring oscillators, Key generation, FPGA, Hardware Security

Εκτεταμένη Περίληψη

Η ραγδαία εξάπλωση συσκευών Διαδικτύου των Πραγμάτων (IoT) έχει καταστήσει την ασφαλή αποθήκευση κρυπτογραφικών κλειδιών σε μη πτητική μνήμη (π.χ. EEPROM, SRAM ή flash) ένα σημείο ευπάθειας, καθώς τα δεδομένα αυτά παραμένουν εκτεθειμένα σε φυσικές επιθέσεις ακόμη και όταν η συσκευή είναι εκτός λειτουργίας. Οι Φυσικά μη Κλωνοποιήσιμες Συναρτήσεις (Physical Unclonable Functions, PUFs) αποτελούν μια εναλλακτική προσέγγιση υλικού, η οποία αξιοποιεί τις αναπόφευκτες μικροδιαφοροποιήσεις της διαδικασίας κατασκευής ολοκληρωμένων κυκλωμάτων για να παράγει ένα μοναδικό, δυσαναπαράγωγο «ψηφιακό αποτύπωμα» για κάθε φυσική συσκευή, χωρίς να απαιτείται μόνιμη αποθήκευση του κλειδιού.

Αντικείμενο της παρούσας εργασίας είναι ο σχεδιασμός, η υλοποίηση και η πειραματική αξιολόγηση ενός πλήρους συστήματος παραγωγής κρυπτογραφικών κλειδιών βασισμένου σε PUF τύπου Ταλαντωτή Δακτυλίου (Ring Oscillator – RO), με στόχο πλατφόρμα Zynq-7000 (πλακέτα Zybo Z7-10). Η προσέγγιση που ακολουθήθηκε είναι “bottom-up”: ξεκινώντας από την ακατέργαστη μέτρηση της συχνότητας ταλάντωσης κάθε δακτυλίου, η εργασία προχωρά σε στάδια κανονικοποίησης και συμπίεσης εντροπίας, καταλήγοντας σε μια πλήρη αλυσίδα παραγωγής κλειδιού προσβάσιμη μέσω υλικολογισμικού (firmware) στο υποσύστημα επεξεργασίας (Processing System) της πλακέτας, καθώς και μέσω αντίστοιχης αλυσίδας εργαλείων λογισμικού στην πλευρά του υπολογιστή-οικοδεσπότη (host). Για τις ανάγκες της εργασίας αναπτύχθηκαν επιπλέον εξειδικευμένα εργαλεία, όπως ένας μηχανισμός τοποθέτησης (placer) που διασφαλίζει τη συμμετρική τοποθέτηση των ταλαντωτών δακτυλίου πάνω στο υλικό του FPGA, καθώς και σενάρια συλλογής και ανάλυσης δεδομένων για τον χαρακτηρισμό του σχεδιασμού σε πολλαπλές φυσικές πλακέτες.

Η αξιολόγηση απόδοσης που πραγματοποιήθηκε ανέδειξε καθαρά τις σχεδιαστικές προτεραιότητες του συστήματος. Η αξιοπιστία (reliability) τέθηκε ως πρωταρχικός στόχος σχεδίασης, με αποτέλεσμα το μετρούμενο ποσοστό σφαλμάτων bit (Bit Error Rate) να είναι αρκετά χαμηλό ώστε να διορθώνεται αποτελεσματικά με έναν απλό κώδικα Hamming χαμηλής πολυπλοκότητας, χωρίς να απαιτείται η χρήση πιο σύνθετου fuzzy extractor. Το κόστος αυτής της επιλογής αντικατοπτρίστηκε στους άλλους δύο βασικούς δείκτες ποιότητας: η μετρούμενη μοναδικότητα (uniqueness) του σχεδιασμού ήταν περίπου 24%, αισθητά χαμηλότερη από την ιδανική τιμή του 50%, υποδεικνύοντας αυξημένη συσχέτιση μεταξύ των αποκρίσεων διαφορετικών πλακετών, ενώ και η ομοιομορφία (uniformity) της παραγόμενης ακολουθίας bit απέκλινε από την ιδανική ισορροπία 50/50, παρά τα στάδια κανονικοποίησης και συμπίεσης εντροπίας που εφαρμόστηκαν. Και οι δύο αποκλίσεις αποδίδονται στο γεγονός ότι το στάδιο κανονικοποίησης εκτιμά τις τιμές αναφοράς του από ένα σχετικά μικρό δείγμα πλακετών, το οποίο πιθανότατα δεν επαρκεί για την πλήρη διάκριση συστηματικής (ανεξάρτητης πλακέτας) μεροληψίας από τη γνήσια, εξαρτώμενη από την πλακέτα, διακύμανση της διαδικασίας κατασκευής.

Μία ακόμη σημαντική παρατήρηση αφορά το υποσύστημα διόρθωσης σφαλμάτων: παρότι το σχήμα διόρθωσης βασισμένο σε κώδικα Hamming σχεδιάστηκε, υλοποιήθηκε και αξιολογήθηκε επιτυχώς σε επίπεδο λογικής, δεν κατέστη δυνατή η ενσωμάτωσή του στο τελικό συνθεμένο bitstream λόγω τεχνικών περιορισμών (packaging conflicts) του εργαλείου Vivado που χρησιμοποιήθηκε. Ως αποτέλεσμα, το τελικό σύστημα βασίζεται προσωρινά σε ένα στάδιο υπό μορφή λογισμικού (SHA-256 conditioning) ως πρακτική, αν και ατελή, εναλλακτική λύση αντί της διόρθωσης σφαλμάτων σε υλικό. Επιπλέον, τα πειραματικά δεδομένα συλλέχθηκαν από μικρό αριθμό πλακετών — συμβατό με τους περιορισμούς μιας προπτυχιακής πτυχιακής εργασίας, αλλά μικρότερο από την κλίμακα αντίστοιχων μελετών της βιβλιογραφίας — και συνεπώς τα αποτελέσματα θα πρέπει να ερμηνεύονται ως ενδεικτικά της συμπεριφοράς του σχεδιασμού και όχι ως οριστικός στατιστικός χαρακτηρισμός.

Συνολικά, η αρχιτεκτονική και τα εργαλεία που αναπτύχθηκαν στο πλαίσιο της παρούσας εργασίας αποτελούν μια στέρεη και επεκτάσιμη βάση για ένα σύστημα παραγωγής κλειδιών RO-PUF υλοποιημένο σε FPGA. Οι πλέον άμεσες κατευθύνσεις για μελλοντική εργασία είναι η μείωση του χάσματος μεταξύ των τρεχουσών και των ιδανικών τιμών μοναδικότητας και ομοιομορφίας — ενδεχομένως μέσω κανονικοποίησης με μεγαλύτερο δείγμα πλακετών — καθώς και η ολοκλήρωση της ενσωμάτωσης του κυκλώματος διόρθωσης σφαλμάτων σε υλικό,

μαζί με μια αξιολόγηση μεγαλύτερης κλίμακας για την επιβεβαίωση των παρατηρούμενων μετρικών.

Table of contents

Contents

List of Figures

Tables

Listings

Diagrams

1 Introduction

As the world becomes more interconnected, the demand for secure data storage and transmission continues to increase, reflecting the ongoing transformation toward more lightweight and compact electronics. Traditionally, security mechanisms rely extensively on secret keys, kept as binary, encrypted data in non-volatile memory like EEPROM, SRAM or flash. Although these types of memory are essential parts of conventional design, in the case of more power- and area-restricted applications, they can be problematic, as well as weak points in terms of security. Furthermore, when a circuit is powered down, hardwired data remains at risk of unauthorized access and theft. To address these fundamental vulnerabilities, Physically Unclonable Functions (PUFs) have emerged as a promising hardware-based root-of-trust, providing a secure, energy-efficient, and cost-effective alternative for generating and storing a system's private master keys without the need for permanent, vulnerable storage.

A PUF can be most easily described as a unique electronic fingerprint for integrated circuits. It is a hardware security primitive that exploits manufacturing process induced variations to produce chip specific signatures. These variations, occurring at the nanoscale in the logic and interconnects of a chip, include the random concentration of dopants in transistors, variations in gate oxide thickness, and differences in wiring delays. Because these manufacturing variations are random and practically impossible to predict or replicate even by the original manufacturer using the same process, the resulting functions are considered physically unclonable. Mathematically, a PUF maps an external stimulus, known as a challenge, to a unique output, known as a response. The combination of these inputs and outputs is referred to as Challenge-Response Pairs (CRPs), allowing for the extraction of secrets that are inseparable from the physical device itself.

Field Programmable Gate Arrays (FPGAs) have emerged as a promising platform for the development of PUFs as they combine the versatility of microprocessors and the parallel execution and performance advantages of ASICs while being several times less expensive. While PUFs can be realized in ASICs, the escalating fabrication costs of advanced process technologies make FPGAs a more attractive option for many developers; for instance, the chip design cost in 28 nm technology is estimated to be almost double that of 45 nm technology. FPGAs offer significant advantages, including shorter time-to-market, lower design costs, and the ability to instantiate PUFs directly from the hardware fabric's standard components.

Among the use cases for PUF technology, the most common ones include authentication, random number generation and secret key generation. For authentication applications, a PUF can be used to extract a chip-specific signature based on the unique characteristics of the chip it is implemented in. The mechanism involves a challengeable PUF that receives the external stimulus and then produces its device-specific response; the PUF's output is then compared to a reference one, and if and only if they match is the device authenticated. In the case of Random Number Generation, a PUF can act as a high entropy seed provider. Unlike pseudo random number generators that rely on secret seed inputs, PUFs have the ability to extract randomness directly from complex physical systems, making them completely unpredictable. By leveraging metastable challenges or the inherent randomness and noise in the startup patterns of memory cells, PUFs generate streams of truly random bits. Researchers have recently proposed the combination of PUFs with chaotic PRNGs where the initial seed is secured from the PUF. Finally, for the key generation use case, the PUF acts as a complete key generator,

bypassing the need for storing the key locally and potentially compromising security; such a design is the main topic of the following pages.

The following sections of this work will focus on the detailed analysis of the implemented key generator and its comparison with common practices for similar systems, both in terms of PUFs used and architectures implemented. The approach chosen for the presentation of the system is a bottom-up one in terms of abstraction. The first chapter will present some basic PUF-related information on which later sections will rely. In section 2, the basic Ring Oscillator architecture used will be presented and explained, while in section 3 the architectural details of its deployment will be discussed. Section 4 will mainly focus on the post-processing of the system and how it handles the raw PUF response to produce a key, while section 5 will handle the topics relating to error correction. Section 6 then presents a performance evaluation of the design against standard PUF quality metrics. Finally, the last two sections of this work are dedicated to the custom tooling and evaluation scripts used in the development of the presented system, and the software interaction developed on the processing system of the board.

2 The Ring Oscillator PUF

As mentioned before, a physical unclonable function is a hardware security primitive that uses manufacturing process variation to produce cryptographic secrets. For this implementation, the chosen PUF type is the Ring Oscillator. To better understand its workings, it is useful to categorize it among the different PUF types. The RO is a **silicon-based**, **delay-based**, **weak** PUF. A silicon-based PUF is a PUF implemented in semiconductor hardware that exploits the inherent and uncontrollable manufacturing variations present in silicon devices. In our case, the device used is a Zybo Z7-10 board that integrates the AMD/Xilinx XC7Z010 device. These variations affect electrical properties such as transistor thresholds, propagation delays, and memory cell characteristics, creating a unique hardware fingerprint for each chip. A **weak PUF** is a PUF that provides a limited number of challenge-response pairs (CRPs), often only one or a small set of stable responses. Weak PUFs are primarily used for device authentication, secret-key generation, and secure key storage as in this design. Finally, a **delay-based PUF** is a class of PUFs that derives its responses from manufacturing-induced variations in signal propagation delays through integrated circuits. Although the circuit paths are designed to be identical, microscopic process variations cause slight timing differences between them. By measuring and comparing these delays, the PUF generates responses that are unique to each device. Another example is the Arbiter PUF.

The structure of a Ring Oscillator circuit is fairly simple. It consists of an odd number of inverters and an AND or NAND gate to enable and disable the chain as shown in figure 2.1. The output of the circuit is an oscillating signal whose frequency depends on the structure of the inverters and connections that drive it.

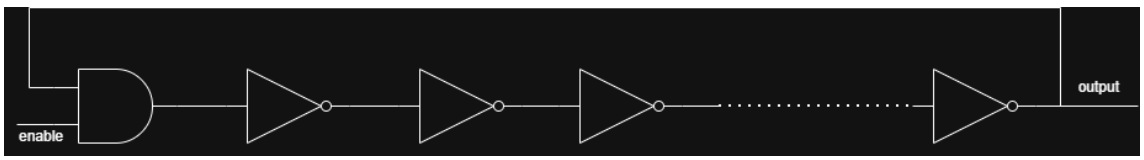

Figure 2.1: Schematic of a Ring Oscillator.

In an ASIC implementation of an RO, the inversion would be carried out by a CMOS inverter circuit consisting of two MOSFET devices in a vertical configuration, as shown in fig 2.2. In the case of an FPGA implementation, the components of the chain described above are look-up tables (LUTs) (fig 2.3) - configurable memory that stores the output values for all possible input combinations of a logic function - configured to operate the same way as an actual inverter while still preserving the uniqueness.

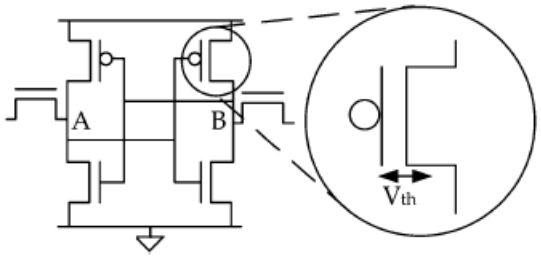

Figure 2.2: Schematic of a CMOS inverter.

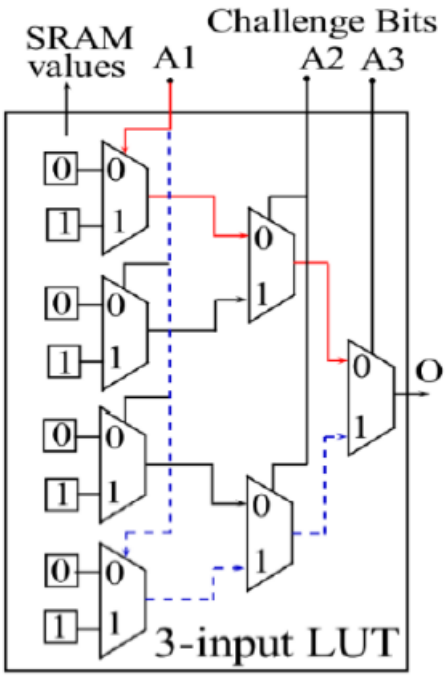

Figure 2.3: Schematic of a LUT.

Specifically, for our design, the number of stages is configurable. As observed by experimentation, the number of stages is a factor that strongly affects the average variation of the RO frequencies. Below are two instances with different numbers of stages.

Figure 2.4: 9-stage RO device view.

For the simultaneous delivery of the enable signal to all ROs, and the avoidance of timing mismatches and thus compromised stability and uniqueness of the response, we latch the enable signal. To achieve this, we use a flip-flop that passes the enable signal to the ROs synchronously.

For better placement of the ROs and easier implementation of the placer script (presented in later sections), the VHDL entity we designed includes Xilinx primitives instead of a behavioral description of the inversion. Furthermore, for the avoidance of Vivado removing the PUFs during implementation (as they are combinatorial loops), it is necessary to modify the xdc file accordingly. In the next section, the main topic is the surrounding logic that makes the utilization of the RO frequency possible.

The complete VHDL source of this design, together with all Python tooling discussed later in this work, is publicly available in the project's repository at https://github.com/Zipnx/PUFfin. The excerpt below, taken from the RO entity of that repository, shows how the NAND gate and the first inverter of the chain are instantiated directly as Xilinx LUT6_L primitives rather than described behaviorally; the remaining eight inverters follow the exact same pattern, each consuming one additional 6-input LUT.

Listing 1: RO.vhd: NAND gate and inverter chain as Xilinx LUT6_L primitives (RO.vhd, PUFfin repository).

```
-- NAND gate: t0 = NAND(en, t9)
LUT6_L_NAND0 : LUT6_L
  generic map (
    INIT => X"8888888888888888"
  )
  port map (
    I0 => enable_int,
    I1 => t9,
    I2 => c(0),
    I3 => '0',
    I4 => '0',
    I5 => '0',
    LO => t0
  );

-- Inverter 1
LUT6_L_INV0 : LUT6_L
  generic map (
    INIT => X"5555555555555555"
  )
  port map (
    I0 => t0,
    I1 => c(1),
    I2 => '0',
    I3 => '0',
    I4 => '0',
    I5 => '0',
    LO => t1
  );

-- ... inverters 2 through 8 repeat the same LUT6_L_INV pattern,
-- each chaining from the previous stage's output (t1 -> t2 -> ... -> t9)
```

3 Raw count extraction

This section's main topic is the implementation of the logic that extracts the unique Ring Oscillator frequency from each implemented instance. At this point, the implemented design starts to differ from the traditional ring oscillator architecture. To better understand the philosophy of our design, a presentation of the common key generator is given, followed by an analysis of our design and a comparison of the two.

3.1 Traditional RO-based key generator architecture

The vast majority of the existing literature regarding RO key generators [4] presents a fairly simple structure that uses the RO as the main source of entropy that derives the final raw key. The idea behind the architecture is the generation of a single bit via the comparison of two RO frequencies. The design consists of the following elements:

1. Ring Oscillator bank: A pool of ROs whose frequencies are measured and then compared

2. multiplexers: these blocks are the selection mechanism that chooses which ROs will be measured
3. Two counters: the frequencies are extracted via these two counters that receive the oscillating signals as their clock
4. A comparator: The final bit producing component

To produce a single bit of the final key, the system first receives a challenge. This challenge is essentially a bit string that contains the select signals for the multiplexers. The number of these multiplexers is based on the number of RO banks; usually there are either two n-to-1 or two n-to-2 multiplexers in the design. Based on these signals, the multiplexers select two ring oscillators (ROs) from the RO array and connect them to the clock inputs of two binary counters. Since each RO oscillates at a frequency determined by its physical characteristics, the counters increment according to the oscillating signals generated by the selected ROs.

The counting process can be terminated using two different techniques. In the first approach, a fixed time window is employed. During this interval, the enable signals of both the counters and the ring oscillators are asserted, allowing the counters to accumulate the number of oscillations produced by the selected ROs. At the end of the measurement window, the counter values represent the frequencies of the corresponding oscillators and can therefore be compared.

The second approach relies on small-size counters and overflow detection logic. Instead of counting for a predetermined period of time, both counters start from zero and increment according to the frequencies of their corresponding ROs. The overflow detection circuitry monitors the counters and determines which one reaches its maximum value first. Since the counter driven by the faster RO accumulates counts more rapidly, the first overflow event indicates which oscillator has the higher frequency.

Both approaches ultimately lead to the final stage of the architecture, which consists of comparing the frequencies of the two selected ring oscillators. In the time-window method, this comparison is performed using the final counter values, whereas in the overflow-based method it is inferred from the first overflow event. If RO1 is determined to be faster than RO2, the system outputs a logic ‘1’; otherwise, it outputs a logic ‘0’ (or vice versa, depending on the specific implementation). Due to unavoidable manufacturing process variations, identically designed ring oscillators exhibit slightly different frequencies, making the comparison result unique to each device and suitable for PUF-based key generation. Block-diagram representations of the architecture are shown in Fig. 3.1 and 3.2.

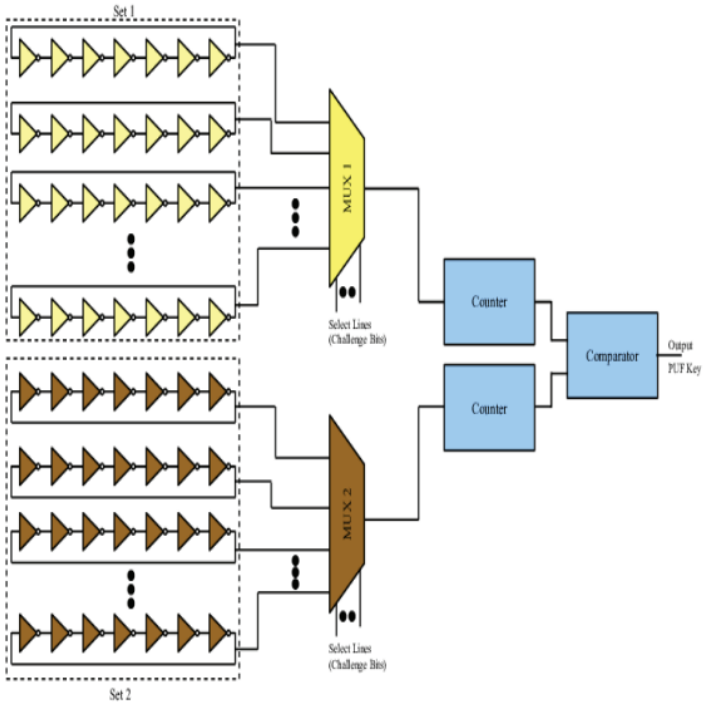

Figure 3.1: common RO-based Key generator architecture with two n-to-1 MUXs.

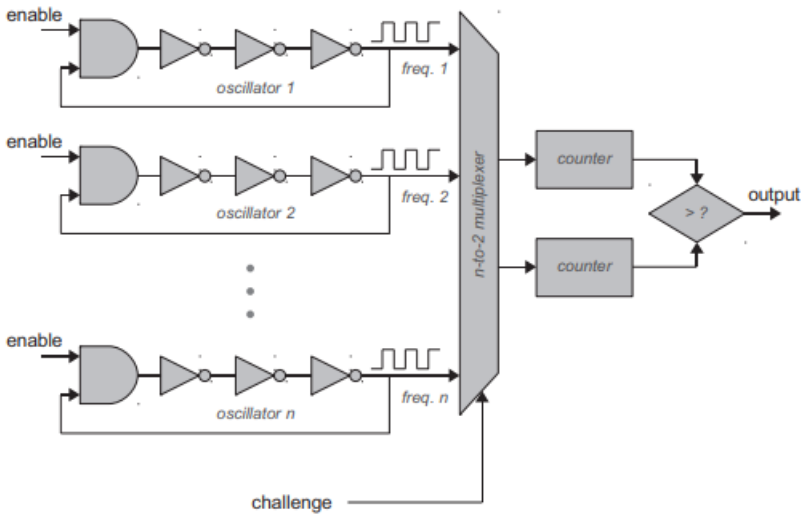

Figure 3.2: common RO-based Key generator architecture with one n-to-2 MUX.

When relating the number of ring oscillators (ROs) to the achievable key length, a chaining technique can be used. When generating a secret response from a number of oscillators, a response length of bits can be achieved by applying a chaining method with comparisons. In

this approach, the second RO used in one comparison is reused as the first RO in the following comparison, forming a sequential chain of RO pairings.

This chaining rule ensures that all ROs are connected in a structured sequence rather than being selected independently for each comparison. As a result, each comparison produces one response bit based on the relative behavior of two adjacent oscillators in the chain. By repeating this process across the full set of ROs, a total of comparison outcomes can be obtained.

In this way, the key length scales linearly with the number of ring oscillators, since each additional oscillator contributes one additional comparison in the chain. Block-diagram representations of the chaining-based RO-PUF architecture are shown in Fig. 3.3. The formula relating the number of ROs in the bank to the length of the final key is therefore the following.

Where N is the number of ROs

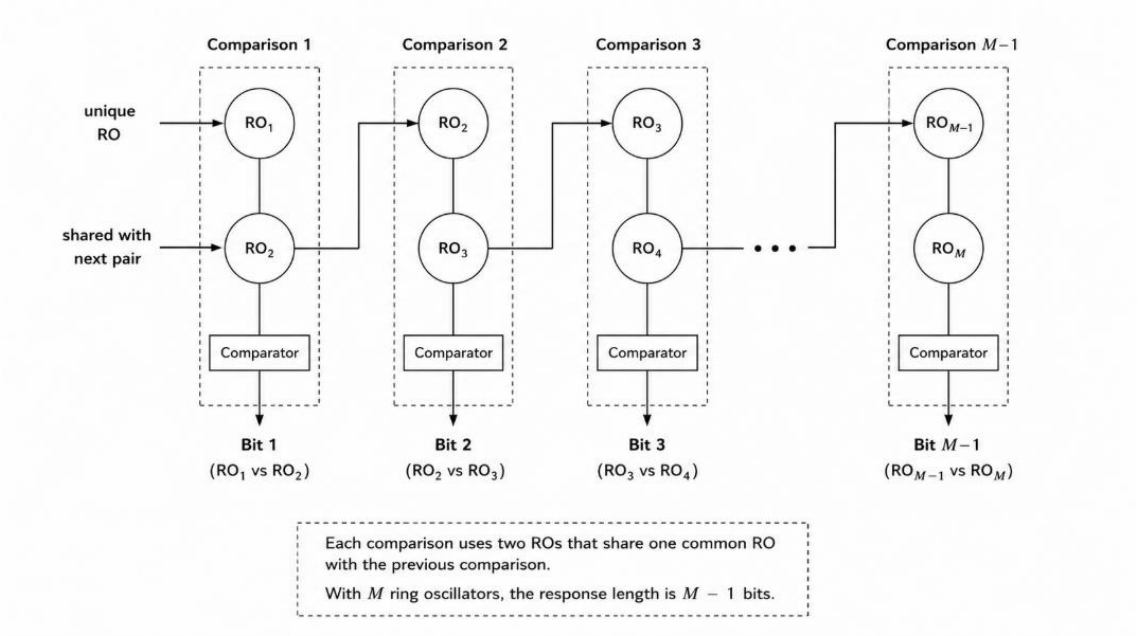

Figure 3.3: visualization of the chaining technique.

3.2 The implemented architecture

As mentioned before, the implemented architecture differs significantly from the traditional one presented in the previous section. The main difference is the processing of the extracted frequencies to produce the raw key bits. The design of this work is heavily based on that of [1] and [2]. The basic components are the following:

1. ROs: the Ring Oscillators of our design are organized in batches
2. MUXs: the multiplexer's main purpose is the selection of a Ring Oscillator from each batch (one MUX per batch)
3. Edge Counters: same as in the traditional design, these counters are used to measure the RO frequencies
4. Enabler: this is the control logic of the whole system; it consists of the following sub-blocks
    i. Enable FSM
    ii. Enable counters

The distinguishing feature of our design is that it can produce the entire key-useful PUF response with one challenge input, removing the need for repetitive challenge inputs, one for each bit of the final key. This is achieved via the RO batch technique where all the ROs of the design are organized in B batches of A ROs each. In the implemented design, the numbers of A and B are configurable and can be changed according to the desired key length. To produce the key, the system receives two control signals. One is the enable of the ROs and the second is the select. The two following sections describe the enabling process, which determines the activation of the system, as well as the main RO bank.

The idea. These four components are not independent modules that happen to sit in the same project; they are wired together by a single top-level entity that instantiates one Enabler for the whole system and sixteen RO/MUX/Edge-Counter chains around it, so that all sixteen batches share the same enable window and busy logic rather than each duplicating it. This is precisely the amortization referred to earlier: the expensive, per-batch hardware (the enable and busy logic) exists once, while only the inexpensive per-oscillator logic (the ROs themselves and their MUX) is replicated sixteen times.

The implementation. The *rocount* entity generates sixteen instances of a *TOP* component (itself wrapping one batch's MUX and edge counter), each driven by its own slice of the shared enable/reset vectors coming out of a single *Enable_counter* instance, and each writing its 24-bit count into its own slice of a flattened 384-bit output bus. A small combinational process ORs together the busy flags of the first few instances to produce one overall busy signal for the AXI wrapper described in the Software Interaction chapter.

Listing 2: rocount.vhd: sixteen batch instances sharing a single Enable_counter (PUFfin repository).

```
gen_instances : for i in 0 to 15 generate
    TOP_inst : TOP
      port map (
        clk   => clk,
        reset => RO_reset_int(i),
        en    => RO_enable_int(i),
        sel   => sel,
        count => count((i*24 + 23) downto i*24)
      );
end generate;

Enable_inst : Enable_counter
generic map (
    TARGET_VALUE => 500_000,
    COUNTER_WIDTH => 19
)
port map (
    clk => clk,
    reset     => reset,
    enable    => cen,
    RO_enable => RO_enable_int,
    RO_reset  => RO_reset_int,
    busy      => are_busy
);

BUSY_EVAL: process (are_busy)
    variable temp: std_logic := '0';
begin
    temp := '0';
    for i in 0 to 4 loop
        temp := temp or are_busy(i);
    end loop;
    busy <= temp;
end process;
```

3.2.1 Enabling

As mentioned previously, there are two main ways to activate the ROs for the counting process. In our design, we opted for the second way, the enable window one. To achieve this, we have designed the enabler block. The purpose of this block is the creation of a fixed-length window in which the ROs will be active and their edges will be counted. Such a task could only be implemented in hardware as the precision offered by an external stimulus directly to the ROs' AND gate is not sufficient due to the much greater delay of the PS compared to the PL. The external system (PS of the board or other) drives two signals of the Enabler, one external enable (iTrigger) signal and the reset (oRoRst) of the system; based on these, the enabler FSM works as follows:

Table Raw count extraction.1: Enable FSM transitions

Current state	Condition	Next state	Output
IDLE	iTrigger = ‘0’	IDLE	No changes
IDLE	iTrigger = ‘1’	ACTIVE	Enable ROs, counter = ‘0’
ACTIVE	Counter < target	ACTIVE	Counter ++
ACTIVE	Counter = target	DONE	oRoRst = ‘1’
DONE	iTrigger = ‘1’	DONE	
DONE	iTrigger = ‘0’	IDLE	Counter = ‘0’

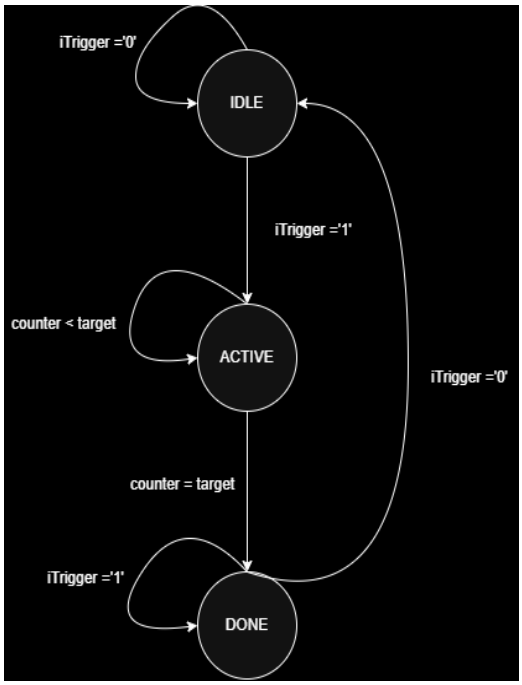

Diagram 3.1: Enable FSM transition diagram.

The Enable FSM’s transitions as described above are affected by the state of the counter value. These counter values are determined by the action of the enable counters. These components are designed to increment based on the board's clock. Their size is configurable and selected based on the desired length of the activation window. The main window-length-determining parameter is the target, and it is computed based on the frequency of the used clock and the window length. Once the external enable arrives, the counters start incrementing to the target value. As long as their counts are less than the target, the enabler drives the RO enable signal high and the downstream Ring Oscillators are active and their frequencies are measured. This way we ensure a fixed-length window of operation and avoid metastability due to software and ARM signal delays.

The idea. The activation window has to be timed by hardware rather than software, since the propagation and interrupt latency of the PS is many orders of magnitude larger than the sub-microsecond precision the window needs; a software-driven enable would make the window length jitter far more than the RO frequency differences the design is trying to measure. The Enabler is therefore a small, self-contained state machine, clocked entirely on the PL side,

that only needs a single external pulse to start and produces a fixed-length enable window every time, regardless of when the PS happens to raise that pulse.

The implementation. The FSM has exactly the three states of Table 3.1: it sits in *IDLE* until *iTrigger* goes high, at which point it drives the reset and enable outputs high and moves to *ACTIVE*; from there, a free-running counter increments every clock cycle until it reaches the *TARGET_VALUE* generic (which encodes the desired window length for the board's clock frequency), at which point the enable output drops and the FSM moves to *DONE*; it only returns to *IDLE* once the external trigger itself drops, preventing a second window from starting immediately on the same pulse.

Listing 3: Enable_counter.vhd: the enable-window FSM of Table 3.1 (PUFfin repository).

```
type state_t is (IDLE, ACTIVE, DONE);
signal cur_state: state_t := IDLE;
constant TARGET : unsigned(COUNTER_WIDTH-
1 downto 0) := to_unsigned(TARGET_VALUE, COUNTER_WIDTH);

FSM: process (clk, reset)
begin
    if reset = '1' then
        cur_state <= IDLE;
        enable_reg <= (others => '0');
    elsif rising_edge(clk) then
        if cur_state = IDLE then
            if enable = '1' then
                rorst_reg  <= (others => '1');
                enable_reg <= (others => '1');
                cur_state  <= ACTIVE;
                counter    <= (others => '0');
            end if;
        elsif cur_state = ACTIVE then
            rorst_reg <= (others => '0');
            if counter = TARGET then
                enable_reg <= (others => '0');
                cur_state  <= DONE;
            else
                counter <= counter + 1;
            end if;
        elsif cur_state = DONE then
            if enable = '0' then
                cur_state <= IDLE;
            end if;
        end if;
    end if;
end process;
```

3.2.2 Selection

The ROs of the system are organized in B batches of A ROs each, as can be seen in figure 3.4. After the activation of the oscillators, we need to select B of them to be measured. This is achieved via the B MUXs of the system and the select signal. This signal is an external vector that contains the B individual select signals for all the batches. In the earlier version of this design, the selection determined only the connection of some of the ROs to the edge counters, meaning that all ROs were active in the enable window but only some of them were measured. This choice, while it simplified the design, was harmful with regard to the power consumption and the embedded application prospects of the system. This was later fixed, so the select signal affected the enabling of the chosen ROs.

The batch mechanism is the one that creates the main difference of this system compared to common ones. The key generator relies on the parallel operation of B ROs and the simultaneous measurement of their frequencies. This approach provides a significant advantage by amortizing the substantial hardware overhead of frequency counters, as multiple oscillators within each batch share a single counter through a multiplexer rather than requiring dedicated measurement logic for every individual oscillator. Furthermore, simultaneous measurement drastically reduces latency, allowing for rapid key generation, such as the reference implementation's 5.62 ms runtime. Furthermore, this parallel running feature acts as a countermeasure for side channel attacks as it makes the frequency extractions significantly harder while also creating a matching problem between observed frequencies and implemented ROs.

The idea. Building one dedicated edge counter per oscillator would waste most of the fabric on measurement logic rather than on the oscillators being measured, since a counter wide enough to distinguish process-variation-level frequency differences is considerably larger than a single RO. Instead, each batch of oscillators shares one counter, and a small multiplexer decides, per batch, which one oscillator's edges actually reach that counter for a given challenge - exactly the sharing that makes the batch technique cheaper than a fully parallel design.

The implementation. At the batch level, a small combinational MUX (*batch8*, shown below for a 3-oscillator batch) instantiates its oscillators and selects one of their outputs with a simple *with -select* statement driven by the batch's 3-bit *sel* input; unselected batches simply output '0' and contribute nothing to the shared counter. That selected, still-oscillating signal is then fed directly as the clock of a free-running binary counter, *RO_EdgeCounter*: rather than edge-detecting the RO output with separate logic, the RO signal itself clocks the counter, so every rising edge of the oscillator increments the count by construction, and the accumulated value after the enable window closes is exactly the oscillator's raw frequency count used throughout the rest of this work.

Listing 4: batch8.vhd: the per-batch selection MUX (PUFfin repository).

```
entity batch8 is
  port (
    clk  : in  std_logic;
    en   : in  std_logic;
    sel  : in  std_logic_vector(2 downto 0);  -- select which RO to output
    o    : out std_logic
  );
end batch8;

architecture Behavioral of batch8 is
  signal ro_outputs : std_logic_vector(2 downto 0);
begin
  gen_ro: for i in 0 to 2 generate
      ro_inst: RO
        port map (clk => clk, en => en, o => ro_outputs(i));
  end generate;

  with sel select
      o <= ro_outputs(0) when "0000",
           ro_outputs(1) when "0001",
           ro_outputs(2) when "0010",
           '0'            when others;
end Behavioral;
```

Listing 5: RO_EdgeCounter.vhd: using the selected RO signal directly as a counter clock (PUFfin repository).

```
entity RO_EdgeCounter is
  generic (
    WIDTH : integer := 24
  );
  port (
    rstn     : in  std_logic;                          -- Active-low reset
    ro_clk   : in  std_logic;                          -- RO signal used as clock
    count    : out std_logic_vector(WIDTH-1 downto 0)  -- Output count
  );
end entity;

architecture Behavioral of RO_EdgeCounter is
  signal counter : unsigned(WIDTH-1 downto 0) := (others => '0');
begin
  process(ro_clk, rstn)
  begin
    if rstn = '1' then
      counter <= (others => '0');
    elsif rising_edge(ro_clk) then
      counter <= counter + 1;
    end if;
  end process;

  count <= std_logic_vector(counter);
end architecture;
```

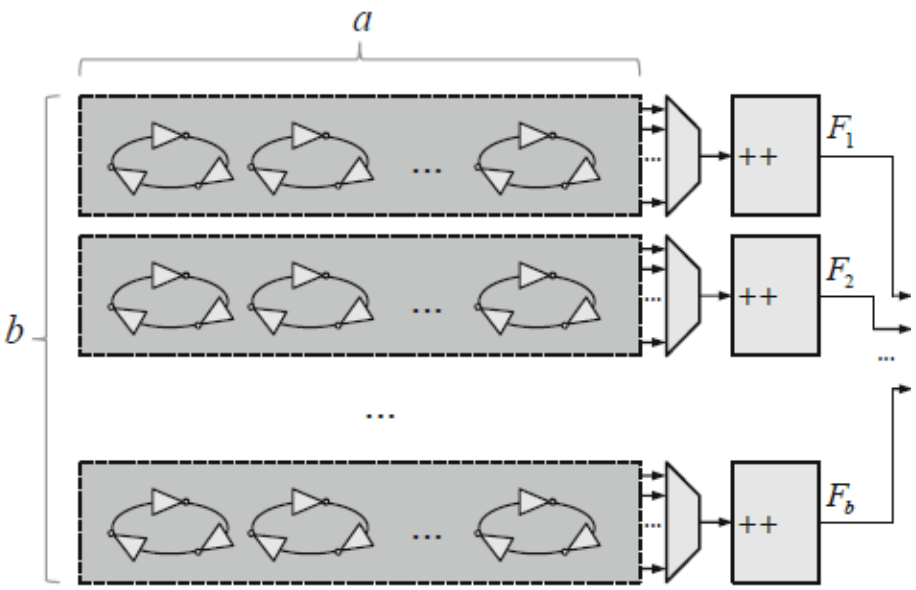

Figure 3.4: Batch system schematic.

3.2.3 Architecture comparison

The following table highlights some of the basic differences of our system to the one described in section 3.1.

Table 3.2: architecture comparison

feature	Our design	Common design
Basic mechanism	Measurement of B oscillators simultaneously	Pairwise comparison
Information extracted	B counts ready to be processed	One Bit of the final key
Hardware Architecture	Hybrid architecture: shares frequency counters across oscillators in each batch to reduce AND gate count	Often sequential or requires substantial counter overhead for parallel designs.
Encoding Method	Section 4	No specific method
Hardware Performance	Optimized for speed, area and security	High cost per bit; up to 8x less efficient in terms of NAND gates per generated bit

4 Post-processing

In this section, the processing of the raw PUF data extracted based on the previously described architecture will be described. The parallel running of the ring oscillators requires a special type of encoding of the frequencies instead of simply comparing them pairwise. The section includes a description of each of the three blocks used to process the data. First, frequency

normalization cleans the data by removing structural biases or "fingerprints" inherent to the specific hardware components, ensuring that the results are truly random rather than predictable. Next, order encoding translates this frequency ordering into a stable digital format designed to stay consistent even when environmental conditions like temperature change slightly. Finally, entropy compression packs the generated information more efficiently, maximizing the density of the secret randomness to produce a stronger and more secure final output. The section will follow the order of the processing flow as implemented in the design (figure 4.1), starting from the earliest stage and closing the section with the explanation of the entropy compressor.

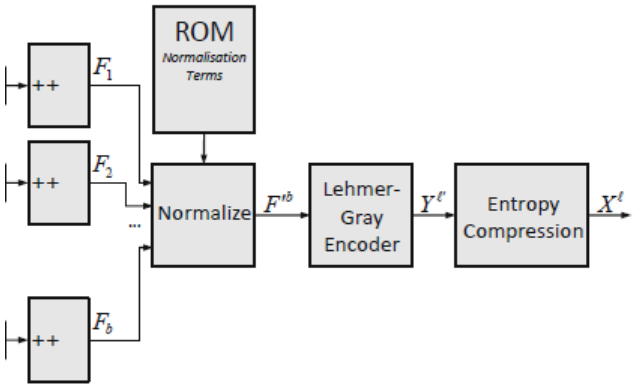

Figure 4.1: Post-processing flow.

4.1 Frequency normalization

The **Frequency Normalization** stage is the first step in the PUFKY post-processing pipeline, specifically designed to remove structural bias from the raw oscillator measurements. Research indicates that raw frequencies are subject to both device-dependent and **oscillator-dependent structural bias**; while device-dependent bias is uniform across a single chip, oscillator-dependent bias can severely compromise entropy by making the frequency ordering deterministic rather than random.

From a statistical perspective, although the oscillators are independent, they cannot be considered identically distributed because each has a different expected value (μFi). Consequently, the frequency ordering would be largely determined by these deterministic expected values rather than the desired random process variations. To resolve this, the system implements the following.

- **Estimation:** An accurate estimate of the expected frequency for each oscillator is obtained by averaging measurements across five different devices. In each of them, the design was placed identically using the custom placer. From each of the devices, we opted for 1024 samples under normal operating conditions. To increase the effectiveness of the normalization process, it is beneficial to increase the number of devices, samples, and also add variation in the operation conditions under which the samples are taken (temperature, voltage, aging etc).

- **Transformation:** This estimate is subtracted from the measured frequency to produce a normalized frequency, defined by the formula:

- **Result:** By subtracting the oscillator-specific bias, the resulting normalized frequencies become independent and identically distributed. This ensures that when the frequencies are later sorted, each of the possible orderings is equally likely to occur, maximizing the available entropy.

The hardware side of this stage is implemented by the RO_Normalizer entity, reproduced in simplified form below. The oscillator-specific averages computed offline, as described above, are programmed into the avg_rom constant (shown here with placeholder values), and the process subtracts the appropriate average from each of the sixteen raw counts every time a new batch selection is presented.

Listing 6: RO_Normalizer.vhd: per-oscillator average subtraction (PUFfin repository).

```
constant avg_rom : avg_array_t := (
    x"000001",  -- avg0
    x"000002",  -- avg1
    x"000003"   -- avg2
);

process(sel, counts)
    variable count_word : unsigned(23 downto 0);
    variable avg_word   : unsigned(23 downto 0);
    variable norm_word  : unsigned(23 downto 0);
begin
    avg_word := unsigned(avg_rom(to_integer(unsigned(sel))));

    for i in 0 to 15 loop
        count_word := unsigned(counts((i*24+23) downto i*24));

        -- absolute difference
        if count_word >= avg_word then
            norm_word := count_word - avg_word;
        else
            norm_word := avg_word - count_word;
        end if;

        results(i) := std_logic_vector(norm_word);
    end loop;
    ...
end process;
```

The characterization process is facilitated by a dedicated Python script that automates the **Estimation** phase by aggregating raw frequency data from the population of test devices. By parsing multi-board datasets, the script calculates the **oscillator-specific mean frequency** for every individual ring oscillator in the design's grid. This script effectively isolates the deterministic structural bias- the unique "fingerprint" of each oscillator's physical implementation - by averaging out the random process variations observed across multiple hardware instances. The resulting population averages represent the precise normalization terms that are subsequently programmed into the system's ROM, allowing the hardware **Normalize block** to perform real-time subtraction and ensure that the final frequency orderings are driven by unique physical entropy rather than predictable layout artifacts.

4.2 Order encoding

Order encoding is the second part of the post-processing pipeline, this stage, as mentioned before, is the main difference between the implemented design and the traditional one presented in the previous section. The main philosophy of order encoding is to extract entropy by the relative order of all the oscillators measured, bypassing the need for sequential activation and measurement.

The first level of this process is **Lehmer encoding**. Its main advantage is that it provides a unique numerical "signature" for a specific frequency sequence without requiring the hardware to perform a complex and time-consuming full sorting operation. Instead, it creates a series of coefficients where each reflects the relationship of an oscillator with all those measured before it in the same group. A key benefit of this approach is its **inherent stability**: if environmental noise causes a "swap" in the ranking between two oscillators with very similar frequencies, it results in only a minimal change to a single Lehmer coefficient.

To further protect the system from errors (bit flips) caused by noise or temperature changes, the architecture applies a layer of **Gray encoding** to these coefficients. Gray codes are specifically designed so that any two consecutive values differ by only a single digit. By combining Lehmer and Gray encodings, the system ensures that a small physical variation in the hardware will result in only a single-bit change in the final output vector, which significantly increases the reliability of the generated key.

The corresponding VHDL, from the encoder entity, computes each Lehmer coefficient as a running count of how many earlier oscillators had a lower normalized value, then converts every coefficient to Gray code with a single XOR-shift operation before concatenating them into the final 49-bit vector, as shown below.

Listing 7: encoder.vhd: Lehmer coefficient computation and Gray conversion (PUFfin repository).

```
-- Gray code conversion
function bin_to_gray(bin_val : unsigned) return unsigned is
begin
  return bin_val xor (bin_val srl 1);
end function;

-- L1: how many of the values seen so far are smaller than C(2)
cnt := 0;
if C(2) > C(1) then cnt := cnt + 1; end if;
vL1 := to_unsigned(cnt, vL1'length);

-- L2
cnt := 0;
for j in 1 to 2 loop
  if C(3) > C(j) then cnt := cnt + 1; end if;
end loop;
vL2 := to_unsigned(cnt, vL2'length);

-- ... L3 through L15 follow the same pattern with a growing comparison window

G1  <= bin_to_gray(vL1);
G2  <= bin_to_gray(vL2);
-- ... G3 through G15 follow the same pattern

code_out <= std_logic_vector(G1)  &
            std_logic_vector(G2)  &
            -- ... continues through G15
            std_logic_vector(G15);
```

Ultimately, this sophisticated encoding allows the PUF system to extract nearly the maximum possible amount of information from the oscillators. While traditional methods waste most of the physical variations by examining only pairs, order encoding captures the complexity of the entire group arrangement, leading to a much higher entropy density.

The equations describing the above process are these:

- $Lj = \sum_{i=1}^{j} gt(F_{j+1}, F_i)$
- $g(x, y) = \begin{cases} 0, & x < y \\ 1, & x \geq y \end{cases}$

The implementation of the order encoding module, as is the case with every part of this system, is configurable in order to accommodate for all possible numbers of ROs the system can use.

4.3 Entropy compression

Entropy compression is the final step of the post-processing pipeline, designed to maximize the **entropy density** (ρ) of the generated response before it is used for key generation. While the preceding Lehmer-Gray encoding successfully captures the physical ordering of the oscillators, the resulting bit vector is not perfectly uniform because many Lehmer coefficients have ranges that are not integer powers of two, leading to biased or dependent bits. To correct this, the architecture utilizes a simple yet effective compression method by selectively XOR-ing the bits that suffer most from these biases. This process significantly increases the "purity" of the randomness; for example, compressing a 49-bit response to 42 bits can raise the entropy density from approximately 90% to nearly 98% according to the authors of the original design. However, this gain in entropy comes with a minor trade-off, as XOR-compression typically causes a slight increase in the bit error probability, which must be accounted for in the subsequent error-correction stages.

The entropy compression stage is implemented as a flat list of XOR operations between specific bit indices of the 49-bit code, each pairing a bit identified as biased by the analysis script described in the Custom Tooling chapter with a less-biased bit elsewhere in the vector; an excerpt is shown below.

Listing 8: ent_compressor.vhd: XOR-based folding of biased bit positions (PUFfin repository).

```
code_out <= code_in;
code_out(0)  <= code_in(0)  xor code_in(5);
code_out(2)  <= code_in(2)  xor code_in(6);
code_out(3)  <= code_in(3)  xor code_in(8);
code_out(4)  <= code_in(4)  xor code_in(9);
code_out(7)  <= code_in(7)  xor code_in(11);
code_out(10) <= code_in(10) xor code_in(14);
code_out(12) <= code_in(12) xor code_in(15);
code_out(13) <= code_in(13) xor code_in(16);
code_out(20) <= code_in(20) xor code_in(21);
code_out(24) <= code_in(24) xor code_in(26);
code_out(28) <= code_in(28) xor code_in(29);
code_out(32) <= code_in(32) xor code_in(33);
code_out(35) <= code_in(35) xor code_in(37);
code_out(36) <= code_in(36) xor code_in(38);
```

To identify the problematic bits of the Lehmer encoding process, a python script was created which by generating a large sample (N=100,000) of random orderings for a batch of oscillators, models the transformation of raw physical entropy into digital form through **Lehmer and Gray encoding**. Most Lehmer coefficients have ranges that are not integer powers of two, which inherently results in a suboptimal binary encoding where certain bits are biased or dependent. The script identifies these "problematic bits" by calculating the absolute deviation of each bit's mean from the ideal 0.5 probability; any bit exceeding a specific bias threshold (set at 0.15 in the code) is flagged for correction. This simulation provides the essential data required for the Entropy Compression stage, allowing the system to selectively XOR these biased bits to maximize the final entropy density which reached up to 98.78% in the original work.

It is worth noting that the simulation results, although accurate, are setup-specific and need to be calculated again for a different configuration of the system.

5 Error correction

Error correction is the final part of a PUF-based system's response generation, since physical responses are inherently noisy and not perfectly reproducible due to environmental variations such as temperature changes, humidity, and chip aging. These factors result in a **bit error rate (Pe)** that can cause bits to flip between evaluations, preventing the reliable reconstruction of a static cryptographic key. In the original work we based our design on, the architecture utilizes a secure sketch construction, specifically the syndrome construction which uses public helper data to reliably recover the original response from a noisy measurement without fully disclosing the secret's entropy. Specifically, PUFKY employs a concatenated code configuration consisting of an efficient **repetition code** acting as the inner code and a highly resource-optimized BCH code as the outer code. This dual-layer approach allows the system to achieve an extremely high key reliability.

In our case, the raw key we produced was tested for reproducibility before applying error correction, to determine the initial BER and evaluate the best-suited Fuzzy extractor. The evaluation happened using 5 identical devices sampling 1024 keys and testing the inter device variation between samples. The results indicated a bit error rate lower than 0.03.

The low BER allowed for a simpler error correction logic to be sufficient for the implemented design. The architecture used is a multiple Hamming encoder and decoder. **Hamming codes**

represent a foundational class of linear block codes employed to mitigate the inherent unreliability (Pe) of PUF responses caused by environmental noise. Generally, Hamming codes are highly efficient for correcting single-bit errors within a block, making them a practical choice for hardware-constrained designs where error patterns are relatively sparse. In our case, the design splits the raw key into seven 7-bit words and implements an (11, 4) Hamming code on each segment. This modular logic allows for a maximum correction of **7 errors** across the final key, provided that no more than one error occurs in any single segment. This results in an encoded 77-bit word (comprised of 49 data bits and 28 parity bits), offering a structured alternative to the syndrome-based BCH decoding favored by PUFKY for handling multiple errors within a single larger block.

The VHDL implementation splits the 49-bit key into seven 7-bit blocks and applies the (11,4) encoding to each block independently inside a single process, as shown below; the four parity bits of each block are computed directly as XOR combinations of its data bits, following the standard Hamming parity-check equations.

Listing 9: hamming_encoder.vhd: per-block (11,4) Hamming encoding of the 49-bit key (PUFfin repository).

```
for b in 0 to 6 loop
    idx := b*7;
    block_data := message(idx+6 downto idx);

    block_enc(2)  := block_data(0);
    block_enc(4)  := block_data(1);
    block_enc(5)  := block_data(2);
    block_enc(6)  := block_data(3);
    block_enc(8)  := block_data(4);
    block_enc(9)  := block_data(5);
    block_enc(10) := block_data(6);

    -- compute parity bits
    block_enc(0) := block_enc(2) xor block_enc(4) xor block_enc(6) xor block_enc(8) xor block_en
c(10);
    block_enc(1) := block_enc(2) xor block_enc(5) xor block_enc(6) xor block_enc(9) xor block_en
c(10);
    block_enc(3) := block_enc(4) xor block_enc(5) xor block_enc(6);
    block_enc(7) := block_enc(8) xor block_enc(9) xor block_enc(10);

    -- copy into output variable
    temp_enc((b*11)+10 downto b*11) := block_enc;
end loop;
encoding <= temp_enc;
```

6 Performance Evaluation

PUFs are typically characterized using four standard quality metrics [3], each capturing a different, complementary aspect of how well a PUF behaves as a security primitive: uniqueness and bit-aliasing describe how well the design distinguishes between physically distinct devices, uniformity describes the internal balance of a single response, and reliability describes how consistently one device reproduces its own response over time and operating conditions. Each of these is discussed in turn below, together with the metric that was actually prioritized for this design and the two that remain open for improvement.

6.1 Uniqueness

Uniqueness measures the variation of responses obtained from different chips for the same set of challenges. If X_i and X_j are the n-bit responses of the i-th and j-th chips respectively for the same challenge, then uniqueness is expressed as the average inter-chip Hamming distance among k devices:

where $HD(X_i,X_j)$ is the Hamming distance between the n-bit strings X_i and X_j, and k is the number of devices under test. The ideal value of uniqueness is 50%, meaning that, on average, exactly half of the bits differ between the responses of any two distinct devices to the same challenge - anything lower indicates that devices are producing more similar responses than pure randomness would predict, which weakens the ability to tell devices apart from their responses alone.

6.2 Uniformity

Uniformity determines how uniform the proportion of 0's and 1's are within the response of a single chip, i.e. whether the response bits are balanced rather than skewed toward one value. For the i-th chip it is calculated as:

where $r_{i,j}$ is the j-th bit of the n-bit response of the i-th chip. The ideal value of uniformity is 50%, corresponding to an equal proportion of 0's and 1's in the response. A uniformity value that deviates significantly from 50% indicates that the post-processing pipeline is producing a biased bitstream, which reduces the effective entropy of the derived key even if the raw physical measurements themselves are unbiased.

6.3 Bit-aliasing

Bit-aliasing captures a related but distinct effect: it occurs when different chips produce nearly identical PUF responses at the same bit position, which is undesirable because it means that bit position carries little device-distinguishing information across the population. It is calculated as:

where $r_{i,j}$ is again the j-th bit of the n-bit response of the i-th chip, and k is the number of chips. The ideal value of bit-aliasing is 50%; a bit position whose aliasing value drifts toward 0% or 100% is effectively "stuck" across the tested population and contributes little to the uniqueness of the overall response, regardless of how random that same bit appears when looking at a single device in isolation.

6.4 Reliability

Reliability determines how efficiently a PUF can reproduce the same response at different operating conditions, such as ambient temperature or supply voltage variation, over a period of time for a given challenge. For the i-th chip, the average intra-chip Hamming distance is first estimated as:

where X_i is the reference response of the i-th chip, $X_{i,?}$ is the response generated by that same chip at time t, and s is the number of repeated measurements of the same challenge; this intra-chip Hamming distance is also the quantity used to compute the bit error rate (BER) referenced in the Error Correction section. Reliability is then defined as:

The ideal value of reliability is 100%, i.e. an ideal value of 0% for the intra-chip Hamming distance, meaning the chip reproduces the exact same response to the exact same challenge every time regardless of environmental conditions.

6.5 Discussion

For the RO-PUF design presented in this work, reliability was treated as the priority metric among the four, which is precisely the reasoning behind the error correction approach adopted in the previous section: with a measured bit error rate low enough to be handled by a lightweight Hamming code rather than a full fuzzy extractor, the design deliberately traded some architectural complexity for a reliability figure that comfortably meets practical key-regeneration requirements. Uniqueness and uniformity, however, remain areas with clear room for improvement: the current design measures a uniqueness of only 24%, well below the ideal 50%, indicating that responses across different boards are more correlated than desired and that further work on normalization or challenge design is needed to decorrelate device-to-device responses. Uniformity likewise falls short of the ideal 50/50 balance of 0's and 1's, suggesting that, despite the frequency normalization and entropy compression stages described earlier, some residual bias remains in the final response and would benefit from a dedicated uniformity-improvement pass, similar in spirit to the biased-placement and phase-calibration approaches reported for other RO-PUF designs in the literature [4].

SHOW SOME FIGURES HERE

7 Custom Tooling and Evaluation Scripts

Beyond the hardware and firmware described in the preceding sections, the development of the RO-PUF key generator relied on a set of custom Python scripts, developed to address three recurring needs: obtaining a placement of the ring oscillators on the fabric that preserves their symmetry, acquiring large samples of raw responses from physical boards, and analysing those samples to derive the normalization and entropy-compression parameters referenced in section 4. This section presents each of these tools in turn.

7.1 Placement tooling

The idea. As mentioned in section 2, leaving the placement of the ring oscillators to the standard Vivado placer is undesirable: even small asymmetries between otherwise identical inverter chains introduce additional, layout-dependent bias into the measured frequencies, on top of the process variation the design is meant to capture. The most direct way to guarantee that every RO instance is built from the same relative arrangement of logic cells is to pin every one of its ten LUTs to an explicit fabric location by hand, but doing this manually for sixteen oscillators per batch, across several batches, is impractical and error-prone. The placer script automates exactly this: given a chosen strategy and orientation, it walks over every RO instance in the design and emits the Xilinx Design Constraints (XDC) needed to place it identically to all the others, without the developer ever touching the Vivado floorplanner by hand.

The implementation. The script builds one XDC line pair per LUT using a small helper, *place_lut()*, which formats the cell's hierarchical path together with a BEL (the specific LUT slot inside a slice: A6LUT, B6LUT, C6LUT or D6LUT) and a LOC (the SLICE_X/SLICE_Y coordinate on the die). The main generator function, *generate_constraints()*, then iterates over all sixteen oscillator instances of a batch and, for each one, calls this helper once per LUT of the RO's NAND-plus-nine-inverter chain, advancing a running (x, y) cursor across the fabric between oscillators according to the chosen layout direction (horizontal or vertical). Two placement

strategies are implemented as two branches of the same function: in the *one-slice-per-stage* branch, the NAND gate and the first three inverters of a given RO share one slice, the next four inverters share a second slice, and the final two inverters share a third slice, so a single 9-inverter RO spans three slices in total. In the *optimal* branch, the ten LUTs are instead distributed one-per-slice across a 2-wide by 5-tall grid of adjacent slices, cycling through all four BELs of each slice column before moving down to the next row. The whole tool is wrapped in a small Tkinter interface that exposes the strategy, orientation and starting coordinates as GUI controls and writes the generated XDC directly to a file, which made it practical to regenerate constraints on the fly while iterating on the stage-count comparison of section 2.

Listing 10: roplacer.py: one-slice-per-stage BEL/LOC constraint generation (PUFfin repository).

```
BELS = ["A6LUT", "B6LUT", "C6LUT", "D6LUT"]

def place_lut(cell_path, bel, slice_x, slice_y):
    return (
        f"set_property BEL {bel} [get_cells {cell_path}]\n"
        f"set_property LOC SLICE_X{slice_x}Y{slice_y} [get_cells {cell_path}]\n"
    )

# one-slice-per-stage: NAND + 3 inverters share a slice, then 4, then 2
slice_x, slice_y = current_x, current_y
output += place_lut(f"{prefix}LUT6_L_NAND0", "A6LUT", slice_x, slice_y)
for i, bel in zip(range(3), BELS[1:]):
    output += place_lut(f"{prefix}LUT6_L_INV{i}", bel, slice_x, slice_y)
for i, bel in zip(range(3, 7), BELS):
    output += place_lut(f"{prefix}LUT6_L_INV{i}", bel, slice_x, slice_y + 1)
for i, bel in zip(range(7, 9), BELS[:2]):
    output += place_lut(f"{prefix}LUT6_L_INV{i}", bel, slice_x, slice_y + 2)
```

Listing 11: roplacer.py: optimal-layout BEL/LOC constraint generation (PUFfin repository).

```
# optimal: one LUT per slice, spread over a 2 x 5 grid of adjacent slices
lut_positions = [
    (0, 0), (1, 0), (0, 1), (1, 1), (0, 2),
    (1, 2), (0, 3), (1, 3), (0, 4), (1, 4)
]
bels = ["A6LUT", "B6LUT", "C6LUT", "D6LUT",
        "A6LUT", "B6LUT", "C6LUT", "D6LUT", "A6LUT", "B6LUT"]
cells = ["LUT6_L_NAND0",
         "LUT6_L_INV0", "LUT6_L_INV1", "LUT6_L_INV2",
         "LUT6_L_INV3", "LUT6_L_INV4", "LUT6_L_INV5",
         "LUT6_L_INV6", "LUT6_L_INV7", "LUT6_L_INV8"]
for (dx, dy), bel, cell in zip(lut_positions, bels, cells):
    px, py = current_x + dx, current_y + dy
    output += place_lut(f"{prefix}{cell}", bel, px, py)
```

A direct, quantitative comparison of resource utilization and RO frequency spread obtained from Vivado implementation runs with each strategy is given in the Implementation Results chapter.

7.2 Data acquisition scripts

Two complementary sampling scripts were used to collect the raw response data needed for characterization. The first targets an early debug build of the firmware that reports raw oscillator counts as plain ASCII-encoded hexadecimal over the serial port; it repeatedly requests a sample for each of the three batch selects and stores the resulting counts as a JSON file,

reporting progress via a simple text progress bar since a full characterization run consists of thousands of individual requests. The second script targets boards running the finished HCM firmware described in the Software Interaction chapter, and is built directly on top of the puffinpy host library rather than talking to the serial port itself: it issues repeated key-generation requests through the HCMCommander class, extracts the 49 bits of each response that are actually used by the key generator (the lower 32 bits of the response together with the 17 least-significant bits of the following word, matching the bit width produced by the post-processing pipeline of section 4), and again stores the collected samples per batch select as JSON. A small companion conversion script performs the same bit-extraction step on raw hexadecimal dumps collected outside of these two scripts, so that data gathered at different points in the project's development could be normalized to the same 49-bit representation before being analyzed.

7.2.1 rosampler.py

The idea. Before the HCM firmware and its binary command protocol existed, an early debug build of the firmware simply printed the sixteen raw oscillator counts of a batch as plain ASCII-encoded hexadecimal whenever a select value was written to the serial port. This script exists to drive that early protocol automatically, requesting a large, fixed number of samples for each of the three batch selects, rather than having a person sit at a terminal and retype the same request thousands of times, which is what a full characterization run of the kind described in section 4.1 (1024 samples per select, per board) would otherwise require.

The implementation. A single low-level function, *execute()*, encapsulates one request/response cycle: it writes the select value terminated by a newline to the serial connection, reads back one line of the response, and slices that line into sixteen 6-character hexadecimal fields, one per oscillator, converting each to an integer. A second function, *benchmark()*, simply calls *execute()* in a loop for the requested number of samples, collecting the sixteen-element lists into a larger result set and printing a simple ASCII progress bar every 32 iterations so that a run of thousands of requests remains observable. The collected samples are ultimately serialized to a JSON file, one array of lists per batch select, which is the exact input format expected by the averaging and Lehmer-bias analysis scripts described later in this section.

Listing 12: rosampler.py: request/response cycle over the early ASCII debug protocol (PUFfin repository).

```
def execute(con, select: int) -> list[int]:
    payload = f'{select}\n\r'
    con.write(payload.encode())

    resp = con.readline().decode().strip('\r\n')
    con.readline()
    results = []
    for i in range(16):
        cut = resp[i*6:(i+1) * 6]
        results.append(int(cut, 16))

    return results

def benchmark(con, select: int, samples: int = 1024) -> list[list[int]]:
    results = []
    for i in range(samples):
        resp = execute(con, select)
        if i % 32 == 0:
            prog = i / samples
            bar = ('#'*int(prog * 70)).ljust(70)
            print(f'<{bar}> {prog*100:.2f}%', end='\r', flush=True)
        if len(resp) != 16:
            error('Invalid response')
            quit()
        results.append(resp)
    return results
```

7.2.2 rosampler_hcm.py

The idea. Once the board-side firmware was replaced by the finished HCM implementation described in the Software Interaction chapter, the raw ASCII protocol above no longer applied: samples now have to be requested through the binary HCM command protocol, and the response format changed to match the register layout of the RO-PUF core rather than a human-readable hex dump. This script is the direct successor of rosampler.py for that new firmware, with one added responsibility: rather than storing the full raw response, it immediately extracts only the 49 bits that the post-processing pipeline of section 4 actually consumes, so that its output is directly usable by the analysis scripts without any further conversion step.

The implementation. Communication is delegated entirely to *HCMCommander* from the puffinpy host library described in the Software Interaction chapter, rather than talking to the serial port directly: *get_resp()* issues a single RO-PUF request for a given batch select and returns the raw 16-byte response, or *None* if the response was malformed. The bit-extraction logic lives in *get_used_bits()*, which reinterprets the lower 4 bytes of the response as the least-significant 32 bits of the key and the next 4 bytes, masked to only its lowest 17 bits, as the most-significant part, combining the two into a single 49-bit integer that is then formatted as a zero-padded binary string. As with rosampler.py, a *benchmark()* function repeats this for all three batch selects over the requested number of samples, with the same progress-bar behaviour, and the resulting per-select lists of 49-bit strings are serialized directly to JSON.

Listing 13: rosampler_hcm.py: sampling and 49-bit extraction over the production HCM protocol (PUFfin repository).

```
from puffinpy import HCMCommander

def get_resp(hcm: HCMCommander, select: int) -> bytes | None:
    resp = hcm.ropuf(select)
    if len(resp) != 16: return None
    return resp

# Checking for the 49 bits that we use right now, this is temporary
def get_used_bits(resp: bytes) -> str:
    lsb = int.from_bytes(resp[0:4], byteorder='big')
    msb = int.from_bytes(resp[4:8], byteorder='big') & 0x1ffff
    value = (msb << 32) | lsb
    return f'{value:049b}'

def benchmark(hcm: HCMCommander, samples: int = 1024):
    results = {0: [], 1: [], 2: []}
    for i in range(samples):
        r0, r1, r2 = get_resp(hcm, 0), get_resp(hcm, 1), get_resp(hcm, 2)
        if r0 is None or r1 is None or r2 is None:
            error('Invalid response')
            quit()
        results[0].append(get_used_bits(r0))
        results[1].append(get_used_bits(r1))
        results[2].append(get_used_bits(r2))
    return results
```

7.2.3 convert.py

The idea. Some of the raw response data used during development was collected before the 49-bit extraction convention used by rosampler_hcm.py was settled on, and exists only as raw hexadecimal dumps of full HCM responses. Rather than re-running a full, multi-hour sampling campaign against physical hardware just to obtain data in the newer format, this small utility performs the same bit-extraction step offline, on already-collected JSON files, so that older and newer datasets can be normalized to the same 49-bit representation before being handed to the analysis scripts.

The implementation. The *convert()* function reproduces the exact same bit-packing logic as rosampler_hcm.py's *get_used_bits()* - reassembling the lower 4 bytes and the next 4 bytes of each raw hex sample into a single 49-bit value - but operates on an entire list of hex strings at once, so it can be applied directly to one of the three per-select arrays stored in an existing JSON result file. The script's *main()* loads such a file from the path given on the command line, replaces the raw hex arrays under keys *"0"*, *"1"* and *"2"* with their converted 49-bit binary-string equivalents, and writes the result to a new file whose name has *"converted"* inserted before the extension, leaving the original raw dump untouched.

Listing 14: convert.py: offline bit-extraction for pre-existing raw hex dumps (PUFfin repository).

```
def convert(samplelist: list[str]) -> list[str]:
    result = []
    for sample in samplelist:
        sample = bytes.fromhex(sample)
        lsb = int.from_bytes(sample[0:4], byteorder='big')
        msb = int.from_bytes(sample[4:8], byteorder='big')
        final = (msb << 32) | lsb
        result.append(f'{final:049b}')
    return result

def main():
    target = sys.argv[1]
    data = load_data(target)

    data['0'] = convert(data['0'])
    data['1'] = convert(data['1'])
    data['2'] = convert(data['2'])

    new_filename = target.split('.')
    new_filename.insert(1, 'converted')
    new_filename = '.'.join(new_filename)

    with open(new_filename, 'w') as f:
        json.dump(data, f, indent=4)
```

7.3 Analysis scripts

The samples collected as described above feed two analysis scripts, corresponding to the two post-processing stages discussed in section 4. The first computes, for every oscillator, the mean of its measured counts across the JSON result files supplied for each of the boards under test; run over the five-device, 1024-sample dataset described in section 4.1, its output is the set of oscillator-specific averages that are subsequently programmed into the Normalize block's reference ROM, i.e. the practical implementation of the Estimation step of the frequency normalization stage. The second script addresses the bit-level bias introduced by Lehmer encoding, discussed in section 4.3: rather than operating on measured data, it generates a large number of uniformly random permutations of the sixteen oscillators in a batch, encodes each with the same Lehmer-Gray procedure used in hardware, and accumulates, bit by bit, how far the average value of each output bit deviates from the ideal probability of 0.5. Bit positions whose deviation exceeds a fixed threshold are reported as biased, and it is this list that determines which bits the Entropy Compression stage selects for XOR-based folding. Because this simulation only depends on the number of oscillators per batch and the resulting Lehmer coefficient ranges, and not on any physical measurement, its results remain valid for a given batch size independently of the specific boards used, but must be regenerated whenever that batch size, and therefore the underlying coefficient ranges, is changed.

7.3.1 averaging.py

The idea. The Estimation step of frequency normalization, described in section 4.1, needs an accurate per-oscillator average frequency, obtained by combining measurements from several physical boards, before that average can be programmed into the hardware Normalize block's reference ROM. Doing this arithmetic by hand across five boards, three batch selects and sixteen oscillators per batch is tedious and error-prone, so this script automates the one specific piece of arithmetic the Estimation step actually needs: turning a set of per-board JSON sample files into one averaged value per oscillator.

The implementation. The script is a thin command-line wrapper around NumPy. *load_data()* reads a single board's JSON result file (as produced by rosampler.py or rosampler_hcm.py) and validates that it is well-formed before returning it as a dictionary keyed by batch select. *get_averages()* then takes that dictionary and, for every batch select, converts the corresponding list of samples into a NumPy array and computes its mean, returning one averaged value per oscillator in the batch. The *main()* function ties this together using argparse: it accepts one or more board result files on the command line, computes the per-board averages for each, and prints them grouped by board, giving a first, unweighted view of the oscillator-specific bias before those numbers are transcribed into the *avg_rom* constant shown in the Post-processing chapter.

Listing 15: averaging.py: per-oscillator mean computation across board result files (PUFfin repository).

```
def load_data(path: str) -> dict | None:
    if not fileExists(path) or isDirectory(path):
        error(f'Invalid filepath: {path}')
        return None
    with open(path, 'r') as f:
        return json.load(f)

def get_averages(data: dict) -> list[float]:
    results = []
    for i, sel in enumerate(data.keys()):
        if str(i) != sel:
            error('Invalid data file')
            return []
        arr = np.array(data[sel])
        results.append(arr.mean())
    return results

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('files', nargs='+', help='JSON Results of sampler from multiple boards')
    args = parser.parse_args()

    averages = []
    for file in args.files:
        data = load_data(file)
        if data is None: return
        avg = get_averages(data)
        if len(avg) == 0: return
        averages.append(avg)

    for i, avg in enumerate(averages):
        print(f'Board {i}:')
        print(f'\t{avg}')
```

7.3.2 lehmer_bias_check.py

The idea. As explained in section 4.3, most Lehmer coefficients have ranges that are not integer powers of two, which means some bits of the encoded, Gray-converted response are inherently more biased than others — but which specific bit positions are biased, and by how much, depends only on the combinatorics of the Lehmer encoding itself, not on any particular physical device. Rather than discovering this bias empirically from noisy hardware measurements, it can be computed directly by simulating the encoding process over a very

large number of random orderings and measuring, bit by bit, how far each output bit's average value drifts from the ideal 0.5 probability. This is exactly what the script does, and its output is the list of bit positions that the Entropy Compression stage of section 4.4 selects for XOR-based folding.

The implementation. The script first defines, in *max_bits_per_L*, how many bits each of the fifteen Lehmer coefficients of a 16-element batch needs (ranging from 1 bit for the first coefficient up to 4 bits for the later ones, mirroring the hardware bit widths of the encoder.vhd entity shown earlier), giving a total encoded width of 49 bits. It then generates *N = 100,000* uniformly random permutations of the sixteen oscillator indices; for each permutation, *lehmer_encode()* computes the Lehmer coefficients by counting, for every position, how many earlier elements are larger, and *gray_encode()* converts each coefficient to Gray code with the same single XOR-shift used in hardware. Every resulting bit is written into a large N×49 NumPy matrix, and once all permutations have been processed, the mean of each column is compared to 0.5: any bit whose absolute deviation from 0.5 exceeds a fixed threshold, set to 0.15 in the code, is reported as a problematic, biased bit position. Because this simulation only depends on the batch size and the resulting Lehmer coefficient ranges rather than on any physical measurement, its result is reusable for any board using the same batch size, but must be regenerated if that batch size changes.

Listing 16: lehmer_bias_check.py: Monte-Carlo estimation of per-bit encoding bias (PUFfin repository).

```
M = 16          # oscillators per batch
BIT_WIDTH = 24
N = 100000      # random permutations to sample

def lehmer_encode(perm):
    n = len(perm)
    code = []
    for i in range(n-1):
        smaller = sum(1 for x in perm[:i] if perm[i] > x)
        code.append(smaller)
    return code

def gray_encode(n):
    return n ^ (n >> 1)

max_bits_per_L = [1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4]
total_bits = sum(max_bits_per_L)
bits_matrix = np.zeros((N, total_bits), dtype=int)

for i in range(N):
    perm = list(range(1, M+1))
    random.shuffle(perm)
    L = lehmer_encode(perm)
    bit_idx = 0
    for coeff, bits_needed in zip(L, max_bits_per_L):
        g = gray_encode(coeff)
        for k in range(bits_needed):
            bits_matrix[i, bit_idx] = (g >> k) & 1
            bit_idx += 1

bias_per_bit = np.abs(bits_matrix.mean(axis=0) - 0.5)
threshold = 0.15
problematic_bits = np.where(bias_per_bit > threshold)[0]
```

8 Software Interaction

Having presented the hardware architecture responsible for the generation and post-processing of the Ring Oscillator response, this section addresses the boundary between that hardware and the rest of the system, i.e. the mechanisms through which the RO-PUF response is made available to a using entity. As is common for PUF-based key generators intended for embedded and IoT applications, the design is targeted at a hard processing system rather than an external microcontroller, specifically the ARM Cortex-A9 based Processing System (PS) of the Zybo Z7-10's Zynq-7000 XC7Z010 device. The following subsections describe the register-level interface exposed to the PS, the firmware developed to operate it, and the way this firmware exposes the RO-PUF, in turn, to an external host machine.

8.1 AXI-Lite register interface

For the RO-PUF core described in the previous sections to be usable by software, it is wrapped by a lightweight AXI4-Lite slave interface, following the same pattern used for the other peripherals of the design, namely the Arbiter PUF core(not described in this work) and the AES accelerators. AXI4-Lite was preferred over the full AXI4 protocol because access to the core is infrequent and register-oriented rather than burst-based, and the reduced protocol avoids unnecessary bus overhead while remaining compatible with the standard Zynq interconnect and driver infrastructure generated by Vivado. The exposed register map is summarized in Table 8.1.

Offset	Name	Access	Description
0x00	CTRL	R/W	bit0: trigger, bits[3:1]: batch select, bit31: reset
0x04	KEY0	RO	four 32-bit words comprising the raw 128-bit RO-PUF response
0x10	KEY3		
0x14	STAT	RO	bit0: busy

Table 8.1: RO-PUF AXI-Lite register map

The reset and trigger bits of the CTRL register drive the Enabling FSM described in section 3.2.1, while the three-bit select field addresses the desired oscillator batch. Once triggered, the calling logic is required to poll the busy bit before the four key registers can be considered to hold a valid response, mirroring the busy semantics of the enable window described previously. In the combined configuration used for the full demonstration project, the same AXI core additionally exposes a small inter-core signalling path (apuf_req / apuf_res / apuf_obf) reserved for coordinating access between the RO-PUF and the Arbiter PUF core when both peripherals are instantiated on the same board.

The idea. The RO-PUF core itself, and the post-processing pipeline wrapped around it, are pure VHDL entities with no awareness of a bus protocol; something has to translate a handful of software-visible registers into the trigger/select/busy signals those entities actually expect, and expose that translation over whatever bus the PS side already knows how to talk to. On a Zynq device that bus is AXI, so a minimal AXI4-Lite slave wrapper is placed around the RO-PUF core purely to give it a memory-mapped address the PS can read and write, without requiring any change to the RO-PUF entity itself.

The implementation. The wrapper instantiates the *keygen* entity once and drives its inputs directly from bit-fields of a single 32-bit control register, latching its outputs into three further read-only registers on every clock edge. The excerpt below shows this core translation: the top three bits of the control word become the batch select, bit 0 becomes the trigger, bit 31

becomes a reset, and the 128-bit key output is simply split across four consecutive registers, exactly matching the register map of Table 8.1.

Listing 17: syskeygen_axi_v1_0_S00_AXI.vhd: mapping the CTRL/KEY/STAT registers onto the RO-PUF core's ports (PUFfin repository).

```
RAWAPUF_INST: keygen_0 port map (
    clk => S_AXI_ACLK,
    rst => sig_rst,
    sel => sig_sel,
    trigger => sig_trigger,
    key => sig_key,
    busy => sig_busy
);

process (S_AXI_ACLK)
begin
    if rising_edge(S_AXI_ACLK) then
        sig_rst     <= slv_reg0(31);
        sig_trigger <= slv_reg0(0);
        sig_sel     <= slv_reg0(3 downto 1);

        slv_reg1 <= sig_key(31 downto 0);
        slv_reg2 <= sig_key(63 downto 32);
        slv_reg3 <= sig_key(95 downto 64);
        slv_reg4 <= sig_key(127 downto 96);

        slv_reg5(0) <= sig_busy;
    end if;
end process;
```

8.2 Processing system firmware

On the PS side, the register interface described above is accessed through a small bare-metal library developed for this work, which provides a common hardware-abstraction layer for every cryptographic peripheral of the board rather than dedicated drivers per project. An enable routine records the AXI base address of a given peripheral, obtained from the Vivado block-design generated address map, into an internal capability bitmask; this allows the same firmware image to adapt to whichever subset of peripherals is present in a given bitstream, from the minimal single-peripheral demonstration projects up to the combined project instantiating the RO-PUF, the Arbiter PUF and both AES cores together.

The RO-PUF driver itself is a thin memory-mapped I/O layer: it writes the batch-select and trigger bits, polls the busy flag, reads back the four key registers and repacks them into a 16-byte array. Because the hardware Hamming error-correction stage discussed in the Error Correction section is not currently active in the synthesized design, the driver additionally offers an optional software-side conditioning step, in which the raw 128-bit response is hashed with

SHA-256 and the two halves of the resulting digest are combined with an exclusive-or operation to yield a 16-byte, more uniformly distributed key. While this conditioning step is not a substitute for a proper fuzzy extractor, it provides a practical, low-cost whitening stage pending resolution of the packaging issues described earlier in this work. The inclusion of the SHA-256 serves a secondary role, specifically the modification of the provided key to match the required length for usage by the AES core.

The idea. Rather than writing a bespoke register-poking routine for each peripheral of the board, the firmware defines a single driver per peripheral type that hides the register offsets behind a small, symmetric API: set a control field, poll a status field, read back a result. For the RO-PUF specifically, the driver additionally has to hide the fact that the hardware error-correction stage discussed in the Error Correction chapter is not currently active, by offering an optional software equivalent instead.

The implementation. A handful of static, inlined helpers wrap the raw *Xil_In32*/*Xil_Out32* calls for the CTRL and STAT registers; *select_set()* and *trigger_set()* read-modify-write the appropriate bits of CTRL without disturbing the others, and *is_busy()* simply masks bit 0 of STAT. The public entry point, *ropuf_execute()*, sequences a full key-generation request: it clears the trigger, programs the batch select, raises the trigger, busy-polls until the core clears it, and only then reads back the four key registers, byte-swapping each 32-bit word into the output buffer. If the caller requests the software conditioning step, the raw 128-bit response is zero-padded to 32 bytes, hashed with SHA-256, and the two 16-byte halves of the digest are folded together with an exclusive-or into the final 16-byte key, exactly as described earlier in this chapter.

Listing 18: ropuf.c: the RO-PUF driver: register sequencing and the SHA-256 conditioning step (PUFfin repository).

```
#define KEYGEN_CTRL(module) (module->hw_addrs.ropuf)
#define KEYGEN_KEY0(module)  module->hw_addrs.ropuf + 0x04
#define KEYGEN_STAT(module)  module->hw_addrs.ropuf + 0x14

void select_set(HCM* module, uint32_t select) {
    uint32_t ctrl = ctrl_get(module) & 0xfffffff1;
    ctrl_set(module, ctrl | ((select & 0x7) << 1));
}

void trigger_set(HCM* module, bool state) {
    uint32_t ctrl = ctrl_get(module);
    if (state) ctrl_set(module, ctrl |= 0x1);
    else       ctrl_set(module, ctrl &= 0xfffffffe);
}

HCMSTATUS ropuf_execute(HCM* module, uint32_t select, uint8_t* out_keybytes, bool pshash) {
    if (!HCMCAP_CHECK(module->capabilities, HCMCAP_ROPUF)) return HCMCAPFAIL;

    trigger_set(module, false);
    select_set(module, select);
    trigger_set(module, true);

    while (is_busy(module));
    uint32_t registers[4];
    trigger_set(module, false);
    resp_get(module, registers);

    for (int i = 0; i < 4; i++) {
        out_keybytes[i*4]     = (registers[i] >> 24) & 0xff;
        out_keybytes[i*4 + 1] = (registers[i] >> 16) & 0xff;
        out_keybytes[i*4 + 2] = (registers[i] >> 8)  & 0xff;
```

```
        out_keybytes[i*4 + 3] = (registers[i] & 0xff);
    }
    if (!pshash) return HCMSUCCESS;

    BYTE plain[32];
    memset(plain, 0, 32);
    memcpy(plain, out_keybytes, 16);

    BYTE hash[SHA256_BLOCK_SIZE];
    SHA256_CTX ctx;
    sha256_init(&ctx);
    sha256_update(&ctx, (const BYTE*)plain, 32);
    sha256_final(&ctx, hash);

    for (int i = 0; i < 16; i++)
        out_keybytes[i] = hash[i] ^ hash[i + 16];

    return HCMSUCCESS;
}
```

8.3 Communication with the host

The firmware exposes the RO-PUF to an external host over UART. It is worth noting that this UART link is provided entirely by the PS: the block design enables the Zynq hard UART peripheral on its dedicated MIO pins, with EMIO routing through the programmable logic explicitly disabled, and no soft UART core is instantiated in the fabric. Consequently, the RO-PUF hardware is never itself connected to the host; the PL is only ever accessed internally, over AXI, by the PS firmware, which is the sole endpoint of the external serial link.

Over this link, the host and the PS exchange fixed-format frames consisting of an opcode byte, a 16-bit payload length and the payload itself, with the firmware replying with a matching response frame. Each incoming command is checked against a per-capability permission bitmask before being dispatched, so that access to a given peripheral, including the RO-PUF, can be selectively enabled or withheld independently of whether the corresponding hardware capability has been detected. A dedicated opcode requests key generation: its payload encodes the desired batch selection, with the most significant bit repurposed as a flag requesting the software hashing step described above. This protocol is mirrored on the host by a companion Python library, which offers the same opcode set together with a command-line and a graphical utility used throughout development to exercise and evaluate the RO-PUF.

The idea. The host needs a way to ask the firmware for a specific operation, with a specific payload, and reliably know where that request ends and the next one begins, over a byte stream that has no built-in notion of message boundaries. The chosen solution is the simplest one that provides this: a fixed 3-byte header (opcode and length) followed by exactly that many payload bytes, mirrored identically on both ends of the link, so that either side can always tell exactly how many bytes to read before a full command or response is available.

The implementation, PS side. *HCM_CommandReceiveDirect()* blocks on the UART until a single opcode byte has been received, then reads exactly two more bytes and unpacks them as a big-endian length, then reads exactly that many payload bytes into a fixed receive buffer before handing the assembled command to the dispatcher described earlier in this chapter. Every read is a small loop rather than a single call, since the underlying *XUartPs_Recv()* is not guaranteed to return all requested bytes in one call.

Listing 19: com.c: framed command reception on the PS side (PUFfin repository).

```
HCMSTATUS HCM_CommandReceiveDirect(HCM* module, Command* out_command) {
    if (!module || module->init_flags.rx_paused || !module->init_flags.uart_initialized)
        return HCMFAILURE;
```

```
    uint8_t opcode;
    int bytes_recv = 0;
    do {
        bytes_recv = XUartPs_Recv(&module->uart_ps, &opcode, 1);
    } while (bytes_recv != 1);

    uint8_t raw_pckt_size[2];
    size_t total_recv = 0;
    while (total_recv < 2) {
        bytes_recv = XUartPs_Recv(&module->uart_ps, raw_pckt_size+total_recv, 2 - total_recv);
        total_recv += bytes_recv;
    }

    uint16_t packet_size = unpack_short(raw_pckt_size);
    if (packet_size >= HCM_COMS_RX_BUFLEN) return HCMFAILURE;

    total_recv = 0;
    while (total_recv < packet_size) {
        bytes_recv = XUartPs_Recv(&module->uart_ps, module-
>rx_buffer + total_recv, packet_size - total_recv);
        total_recv += bytes_recv;
    }
    if (total_recv != packet_size) return HCMMALFORMED;

    out_command->opcode = opcode;
    out_command->size   = packet_size;
    out_command->data   = module->rx_buffer;

    return HCMSUCCESS;
```

The implementation, host side. The companion Python library mirrors this exact framing rather than reinventing it: a *Command* dataclass holds an opcode (drawn from the same enumeration of opcodes as the firmware, including *READ_PUFKY* for RO-PUF key requests) and an optional payload, and its *pack()* method serializes it to bytes using the same layout as the C struct above - one opcode byte followed by a big-endian 16-bit length and the payload - so that a frame built on the host and one built on the PS are byte-for-byte interchangeable.

Listing 20: _command.py: the host-side mirror of the same frame format (PUFfin repository).

```
class Opcode(IntEnum):
    INFO            = 0
    QUERY           = 1
    APUF_SINGLE     = 2
    READ_TEMP       = 3
    READ_PUFKY      = 4
    APUF_BATCH      = 5
    AES_ENCRYPT     = 6
    AES_DECRYPT     = 7

@dataclass(init=True)
class Command:
    opcode: Opcode
    data: Optional[bytes] = None

    def pack(self) -> bytes:
        output  = struct.pack('B', self.opcode)
        output += struct.pack('>H', len(self.data) if self.data is not None else 0)
        if self.data:
            output += self.data
        return output
```

9 Demonstration

For the demonstration of the use of the implemented system, an AES IP core was included in the design. The AES algorithm used is a ten-stage one which requires a 128-bit key. For that purpose, a SHA function was included in order to convert the initial version's 49-bit key to a 256-bit hash, as mentioned in section 8.2. The lower 128 bits were then XORed with the upper 128 ones, creating a key of 128-bit length that was then used by the algorithm to showcase a use case of such a system in a realistic application scenario.

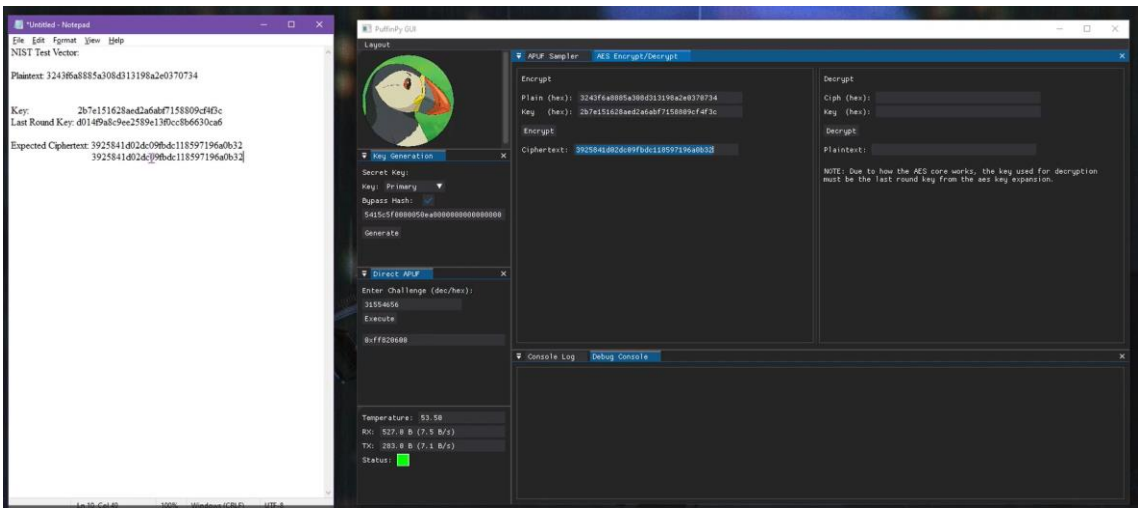

Figure 9.1: AES encryption example.

10 Conclusion and limitations

This work presented the design, implementation and characterization of a Ring Oscillator based Physical Unclonable Function key generator targeted at the Zynq-7000 platform, developed with a bottom-up approach that moved from the raw physical measurement of oscillator frequencies, through normalization and entropy compression, to a complete key-generation pipeline accessible over a processing-system firmware and a companion host-side toolchain. Along the way, custom tooling was developed to address practical needs that are rarely discussed in the PUF literature in equal depth to the core architecture itself, namely a dedicated placer to guarantee symmetric placement of the ring oscillators on the fabric, and a set of acquisition and analysis scripts used to characterize the design across several physical boards.

The performance evaluation carried out in this work shows a design whose priorities and trade-offs are clearly reflected in its measured metrics. Reliability was treated as the primary design goal, and the resulting bit error rate was low enough to be corrected with a lightweight, low-overhead Hamming code rather than a more complex fuzzy extractor, validating that design choice. This came, however, at the cost of the two metrics that were not prioritized to the same degree: the measured uniqueness of 24% falls considerably short of the ideal 50%, indicating

that responses across different boards are more correlated than desired, and the uniformity of the generated bitstream likewise deviates from an ideal 50/50 balance of 0's and 1's despite the normalization and entropy compression stages applied. Both shortcomings point to the same underlying limitation: the current normalization step estimates its reference values from a relatively small set of boards, which is unlikely to be sufficient to fully decorrelate systematic, board-independent bias from genuine, board-specific process variation.

A second limitation concerns the error correction subsystem itself. While the Hamming-based error correction scheme was designed, implemented and evaluated at the logic level, it was not possible to integrate it into the final synthesized bitstream due to packaging conflicts encountered with the version of Vivado used for this work. As a result, the deployed system currently relies on the software-side SHA-256 conditioning step described in the Software Interaction chapter as a practical, if imperfect, substitute for hardware error correction. Resolving this packaging issue and validating the hardware Hamming decoder on real silicon remains an important piece of unfinished work.

Finally, it is worth acknowledging the scale at which this design was evaluated. The characterization data presented in this work was collected from a small number of boards, which is consistent with the constraints of Bachelor Dissertation project but is well below the scale used in comparable studies in the literature, some of which draw on measurements from tens or hundreds of devices. Given how sensitive metrics such as uniqueness and bit-aliasing are to population size, the figures reported here should be read as indicative of the design's behavior rather than as a definitive statistical characterization, and a larger-scale evaluation would be a natural next step to confirm whether the observed uniqueness gap is a property of the architecture itself or an artifact of the limited sample used.

Taken together, these results suggest that the architecture and tooling developed in this work form a solid and extensible foundation for an FPGA-based RO-PUF key generator, but that closing the gap between the current uniqueness and uniformity figures and their ideal values, together with completing the integration of hardware error correction, are the two most concrete directions for continuing this work.

References

[1] Maes, Roel & Herrewege, Anthony & Verbauwhede, Ingrid. (2012). PUFKY: A Fully Functional PUF-Based Cryptographic Key Generator. 10.1007/978-3-642-33027-8_18.

[2]Yin, C.E.D., Qu, G.: LISA: Maximizing RO PUF's Secret Extraction. In: IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pp. 100–105 (June 2010).

[3] N. Nalla Anandakumar, Mohammad S. Hashmi, Mark Tehranipoor,

FPGA-based Physical Unclonable Functions: A comprehensive overview of theory and architectures, Integration, Volume 81, 2021, Pages 175-194, ISSN 0167-9260,

[4] G. E. Suh and S. Devadas, "Physical Unclonable Functions for Device Authentication and Secret Key Generation," *2007 44th ACM/IEEE Design Automation Conference*, San Diego, CA, USA, 2007, pp. 9-14.

All of the tools discussed previously, together with the VHDL sources referenced are available in full in the project's repository at **https://github.com/Zipnx/PUFfin**

Πίνακας Ορολογίας

Ξενόγλωσσος Όρος	Ελληνικός Όρος
Physical Unclonable Function	Φυσικά μη Κλωνοποιήσιμη Συνάρτηση
Ring Oscillator	Ταλαντωτής Δακτυλίου
Challenge-Response Pair	Ζεύγος Ερωτήματος-Απόκρισης
Field Programmable Gate Array	Πίνακας Πυλών Προγραμματιζόμενος στο Πεδίο
Look-Up Table	Πίνακας Αναζήτησης
Finite State Machine	Πεπερασμένη Μηχανή Καταστάσεων
Multiplexer	Πολυπλέκτης
Register	Καταχωρητής
Firmware	Υλικολογισμικό
Entropy	Εντροπία
Uniqueness	Μοναδικότητα
Reliability	Αξιοπιστία
Bit-aliasing	Ψευδο-ομοιότητα bit
Error Correction	Διόρθωση Σφαλμάτων
Encoding	Κωδικοποίηση
Normalization	Κανονικοποίηση
Placement	Τοποθέτηση
Driver	Οδηγός (λογισμικού)

Πίνακας Συντμήσεων – Αρκτικόλεξων – Ακρωνυμίων

Σύντμηση	Πλήρης Ανάπτυξη
PUF	Physical Unclonable Function
RO	Ring Oscillator
CRP	Challenge-Response Pair
FPGA	Field Programmable Gate Array
ASIC	Application-Specific Integrated Circuit
LUT	Look-Up Table
FSM	Finite State Machine
XOR	Exclusive OR
NAND	Not AND
VHDL	VHSIC Hardware Description Language
AXI	Advanced eXtensible Interface
UART	Universal Asynchronous Receiver-Transmitter
SHA	Secure Hash Algorithm
AES	Advanced Encryption Standard
ASCII	American Standard Code for Information Interchange
HD	Hamming Distance
BER	Bit Error Rate
PS	Processing System
PL	Programmable Logic
CMOS	Complementary Metal-Oxide-Semiconductor
IoT	Internet of Things
EEPROM	Electrically Erasable Programmable Read-Only Memory
SRAM	Static Random-Access Memory
JSON	JavaScript Object Notation
ROM	Read-Only Memory

Παράρτημα Α – Πηγαίος Κώδικας

The complete VHDL source code, the processing-system firmware, the host-side Python toolchain, and all placement and evaluation scripts developed and discussed throughout this thesis are maintained in full in the project's public repository, available at: https://github.com/Zipnx/PUFfin. Representative excerpts of the most important source files are already presented in-line within the relevant chapters (see the List of Listings).