UNIVERSITÉ LIBRE DE BRUXELLES

BRUFACE BRUSSELS FACULTY OF ENGINEERING

VRIJE UNIVERSITEIT BRUSSEL

Graduation thesis submitted in partial fulfilment of the requirements for the degree of engineering sciences: Master of Electrical Engineering

Implementation of physical unclonable functions on FPGA circuits

Brent De Weerdt

Master thesis submitted under the supervision of

Prof. Dragomir Milojevic

The co-supervision of

Prof. Jean-Michel Dricot

In order to be awarded the Master's Degree in

Electrical Engineering

Academic year

2020-2021

Abstract

Implementation of physical unclonable functions on FPGA circuits

Brent De Weerdt

Academic year 2020-2021

To be awarded the degree of: *Master of Science in Electrical Engineering*

Keywords: *FPGA, Physical Unclonable Function (PUF), ROPUF*

The goal of this Master's thesis is to design a physical unclonable function and implement it from scratch on an FPGA, such that it can be used to generate cryptographic keys. It should provide a starting point for further research. First some popular PUF designs for FPGAs are described and compared, as well as the properties used to evaluate the performance of a PUF. The ROPUF was chosen for its ease of implementation, flexibility and desirable properties. The design is developed in VHDL and tested in simulation and on real hardware. Analysis of its properties show comparable performance with respect to the state of the art. A BCH decoder is added to the basic design to greatly increase the reliability, such that the responses can be used for cryptographic key generation. The increase in reliability is shown to be sufficient for this purpose, and the code size can be chosen based on the expected error rate of the PUF. Directions for future extensions or improvements are pointed out. Finally, a practical demonstration has been implemented as a starting point for further applications and as a proof of concept that the PUF can indeed be used as cryptographic key generator. All source code, constraint files, scripts to gather and analyze data and the demo programs are made publicly available, such that others can build upon this basis.

List of acronyms

(AMBA) AXI: (Advanced Microcontroller Bus Architecture) Advanced eXtensible Interface

APUF: Arbiter PUF

ASIC: Application-Specific Integrated Circuit

BCH (code): Bose-Chaudhuri-Hochquenghen (code)

CMOS: Complementary Metal Oxide Semiconductor

CRP: Challenge-Response Pair

ECC: Error Correction Code

FF: Flip-Flop

FPGA: Field-Programmable Gate Array

HD: Hamming Distance

IBS (code): Index-Based Syndrome (code)

IC: Integrated Circuit

I/O: Input/Output

IP: Intellectual Property

LFSR: Linear Feedback Shift Register

LUT: LookUp Table

MOSFET: Metal-Oxide-Semiconductor Field-Effect Transistor

PDL: Programmable Delay Line

PUF: Physical Unclonable Function

RFID: Radio-Frequency IDentification

RO: Ring Oscillator

ROPUF: Ring Oscillator PUF

UART: Universal Asynchronous Receiver Transmitter

VHDL: VHSIC Hardware Description Language

Contents

Chapter 1

Introduction

A Physical Unclonable Function (PUF) is a hardware primitive that generates a sequence of bits, that is derived from the physical properties of an Integrated Circuit (IC) or external media, thus unique to that specific device or medium. In 2000 it was shown in [1] that mismatches in MOSFETs for any submicron CMOS process can be used to uniquely identify ICs, without any special manufacturing techniques or post-processing. In [2] from 2002 it was showed that also variations in wire delays on an FPGA can be used to distinguish devices from each other, even in the case of temperature and voltage variations. These hardware fingerprints can be used for authentication of devices, to aid pay-per-device schemes by restricting the execution of software to a specific device, or even to derive cryptographic keys.

The classical methods for authentication, Intellectual Property (IP) protection or data encryption consist of storing a secret key in some memory on the chip. The problem with this approach is that an adversary can read out this memory to obtain the secret key and put it on a counterfeit chip or decrypt sensitive data. Protecting a key that resides in memory from extraction requires complex and expensive anti-tampering techniques.

These two problems of chip counterfeiting and key extraction are tackled by PUFs. The copying of secret keys is prevented by the fact that the PUF's response is derived from slight differences in physical properties caused by uncontrollable manu-

facturing variations. Therefore, the PUF response of the chip cannot be replicated, even by careful manufacturing. The second problem, which is the extraction of the key by adversaries, is alleviated by the fact that the secret information is contained in the hardware itself and can be evaluated at any time, such that there is no need to store the key in some persistent memory which can be read out by attackers. Trying to probe the hardware itself is harder, since invasive operations on the chip cause modifications to the hardware, destroying the information that resided in this hardware. In conclusion, PUFs are appealing because they make securing of authentication IDs or cryptographic keys easier and/or less expensive.

The focus of this thesis is on the implementation of PUFs on FPGA circuits, which primarily use variations in wire delay as many other properties such as threshold voltages, capacitances, etc. cannot be measured on an FPGA due to its constrained layout. The goal is to create a practical PUF implementation from scratch, which can be used to generate cryptographic keys, to be used for example in IoT devices. It is meant as a starting point for further research, by providing a clear guide down to the technical details on how to actually implement a PUF design from start to finish. Some choices such as the PUF measurement time or the specific error correction code to be used are highlighted and analyzed. The resulting implementation is compared to the current state of the art in terms of PUF performance, FPGA resource consumption and power usage.

A small demonstration is created where an encryption key is derived from the generated PUF response, as an example of a possible application. Directions in which to further improve or extend the design are provided too. Finally, all source code, constraint files and scripts are made available such that others can analyze, improve or build on our basic designs.

The contributions of this thesis start with the listing of a number of properties used to evaluate the performance of PUF implementations and the selection of which ones are useful to analyze our design. Then a few possible applications of PUFs are given and the most common designs used for FPGA PUFs are described. From these designs the ring oscillator PUF is chosen and implemented with the goal to generate cryptographic keys. The whole development process starting from

the initial VHDL design up to the placement on real hardware and analysis of the physical implementation is explained in detail. Multiple small Python scripts to process the obtained data are written and the calculated performance properties of the implemented design are compared with the current state of the art. Lastly a practical demonstration is created in order to illustrate a potential use case of our PUF.

The original contribution of this work consist mainly of a detailed description of the complete implementation in VHDL together with the necessary placement constraints and its technicalities, along with scripts to analyze the PUF output data. This framework is also made publicly available in a GitHub repository: `https://github.com/brentdewe/ropuf`. Additionally a comparison is made to highlight the differences in properties between the listed PUF designs, such as the number of response bits, ease of implementation and analysis or potential pitfalls. The power usage of the designs is also estimated, aided by simulations of the circuit activity. These two tables are not typically found in other papers.

Chapter 2

Background and state of the art

In this chapter some main concepts and properties of PUFs are listed, along with a few typical applications. The sole focus here is on creating a PUF with an FPGA, without any other hardware. Following the taxonomy in [3], this can be classified as an intrinsic PUF, because both the source of randomness as the evaluation of it are embedded on the chip itself.

Due to the fixed layout of FPGA chips, the most used technique to derive PUF bits on FPGAs is the measurement in some way of wire delays, as many properties such as threshold voltage, power lines, etc. used on ASICs cannot be measured with an FPGA.

First a distinction between strong and weak PUFs is made, after which the statistical properties of the PUF response bits used to described the quality of a PUF are listed. Next a few applications are described, before giving more details on specific designs and their operation.

2.1 Characteristics

A PUF can be used to generate a single unique fingerprint for the device, or can give different responses to each provided challenge. These challenge response pairs (CRPs) can be used for example for authentication.

2.1.1 Strong and weak PUFs

The difference between strong and weak PUFs is described in [4] and lies primarily in the number of CRP pairs. For a strong PUF the amount of CRPs is very large, in most cases exponential with respect to some parameters. In addition, the CRPs must not be predictable when a subset of CRPs is known. This makes enumeration of the whole CRP space unfeasible. Therefore, for a randomly chosen challenge, the response can only be determined by having access to the PUF when the challenge is given.

A weak PUF has a limited CRP space, which makes enumeration of all CRPs possible. Hence for most applications, access to the PUF has to be restricted such that an attacker does not have direct access to the PUF responses even when in physical possession of the device.

2.1.2 Properties

Most measures to compare PUF performances are based on the statistical properties of their responses. In [5], metrics from different papers are listed and compared, after which they propose a coherent set or properties to evaluate PUFs, which are discussed below. Here only a single PUF identity is considered, instead of multiple responses to different challenges. The only difference is that most properties are averaged over one dimension less, namely the number of different challenges K. All properties listed below are from [5]:

Uniformity

Uniformity describes the ratio between the number of zeros and ones in the PUF response bits. In order for the bits to be fully random, this ratio must be near 50%. It is defined as the fraction of ones in an L-bit PUF response:

$$uniformity = \frac{1}{L}\sum_{l=1}^{N} r_l$$

with r_n the n-th bit in the response.

Reliability

The reliability is the ability of a PUF to generate the exact same response each time. This is especially important when using the response to generate cryptographic keys, as slight variations in the response make the generated key useless. Given a reference L-bit response R, the reliability is one minus the fractional Hamming Distance (HD) averaged over T responses measured at different time instants:

$$reliability = 1 - \frac{1}{T}\sum_{t=1}^{M}\frac{HD(R, R_t)}{L}$$

When the PUF is perfectly reliable, each response is equal and the reliability is 100%. Each additional error will reduce this value.

Steadiness

Steadiness is related to the reliability in that it measures the probability of individual response bits to be zero or one. p_l is defined as the probability that response bit l is a one: $p_l = \sum_{t=1}^{T} r_{l,t}$. In order to obtain reliable responses, each p_l should be as close to 0 or 1 as possible, which is reflected in the definition of steadiness:

$$steadiness = 1 + \frac{1}{L}\sum_{l=1}^{L}\log_2(\max\{p_l, 1 - p_l\})$$

As for the reliability, a perfect PUF would result in a steadiness of 100% and deviations from the reference response will reduce the steadiness.

Uniqueness

To quantify the ability of a PUF to distinguish different chips, uniqueness is used. This is the average fractional Hamming distance between each pair of chips, for N different chips.

$$uniqueness = \frac{2}{N(N-1)}\sum_{n=1}^{N-1}\sum_{m=n+1}^{N}\frac{HD(R_n, R_m)}{L}$$

with R_n the L-bit response of chip n. In the ideal case, the responses for different chips are completely independent from each other, such that the average Hamming distance and thus uniqueness would be 50%.

Diffuseness

The diffuseness has almost the same definition as the uniqueness, with the difference in the fact that not different chips are compared regarding the responses to a single challenge, but that responses to different challenges within a single chip are compared, which leads to the following definition with K the number of different PUF challenges:

$$diffuseness = \frac{2}{K(K-1)} \sum_{k=1}^{K-1} \sum_{m=k+1}^{K} \frac{HD(R_k, R_m)}{L}$$

As for uniqueness, the diffuseness should be close to 50%. As the PUF design in this paper does not use multiple challenges, this property will not be measured.

Bit-aliasing

Bit-aliasing describes the bias of the l-bit in the response over different chips. It is similar to the uniformity in that the bits should be random and the bit-aliasing should be close to 50%. The bit-aliasing of bit l is determined as follows:

$$(bit - aliasing)_l = \frac{1}{N} \sum_{n=1}^{N} r_{l,n}$$

where $r_{l,n}$ is the l-th bit in the response of chip n. A bit-aliasing value close to zero or one also effects the uniqueness, because bits that most often have the same value effectively shorten the device signature length.

Probability of misidentification

Given reference PUF responses R_X and R_Y of chips X and Y respectively, and measurement R'_X of chip X. The probability of misidentification is the probability that $HD(R_Y, R'_X) < HD(R_X, R'_X)$, such that chip X is recognized as chip Y due to noise in the PUF responses. Since in this work the PUF responses are not compared directly but only used to derive cryptographic keys, this metric is not useful here.

2.2 Applications

2.2.1 Authentication

A basic method for authentication is described in [6], where the assumption is that the PUF has an exponential number of CRPs and that a model building attack on the PUF is unfeasible. In this case, the authenticating party can record a number of challenge-response pairs in a database before giving out the PUF. An illustration of the process is given in figure 2.1. Due to the very large number of possible CRPs, an attacker cannot reliably guess which responses are recorded, while prerecording all pairs is impossible. It is then necessary to have access to the PUF when the challenge is given in order to be able to answer with the correct response.

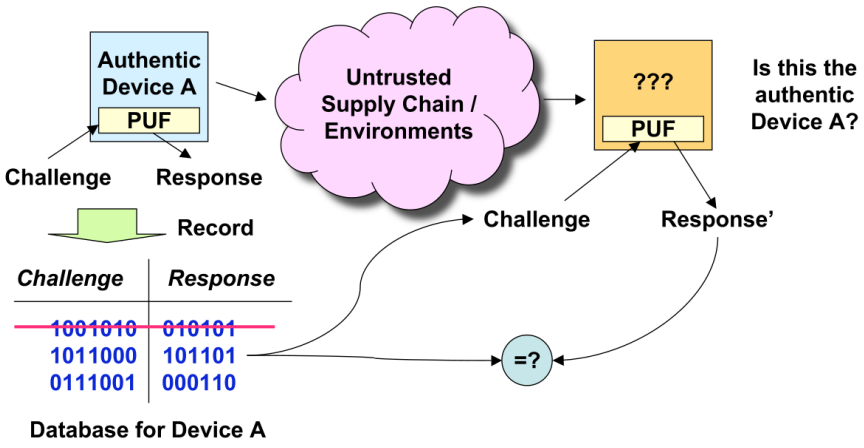

Figure 2.1: Illustration of CRP authentication of a PUF. Figure from [6]

The first commercial implementation of such a scheme was developed by Verayo Inc. and Toppan Printing Co., Ltd., where a PUF was embedded in an RFID tag. This RFID PUF was used in a challenge-response fashion to determine the authenticity of consumer products. Users whose smartphone supported Near Field Communication (NFC) could scan the RFID tag to check whether a product was genuine or a counterfeit. The performance of this PUF is studied in [7], which concluded that this PUF could operate in a large temperature range without identification errors.

2.2.2 Cryptographic key generation

Another application explained in [6] is the generation of cryptographic keys from the PUF response. In this case the response is hashed down to increase its entropy and used as is, or used as the seed for a key generation algorithm if the final key needs to have special properties, e.g. RSA key pairs.

These processing steps require that the PUF response is exactly the same each time it is measured, which is generally not the case due to noise and changes in the environment such as temperature and power supply voltage. Therefore, some error correction code (ECC) needs to be used to enhance the reliability of the responses. After ECC decoding, the obtained output is hashed and used as is or processed further.

An example of this key generation can be found on the Xilinx Zynq Ultrascale+ MPSoC, where an RO based PUF is used to generate a Key Encryption Key (KEK) [8, p. 279]. This KEK is unique to each device and cannot be read out. It is used to encrypt the cryptographic keys of the software or user, such that they can be safely stored in non-volatile memory. When a stored key is needed again, the KEK is extracted from the PUF measurement and used to decrypt the user's keys.

2.2.3 IP protection

A last example application is the protection of Intellectual Property (IP), for instance on FPGAs. In a pay-per-device licensing scheme, a company sells their IP to be used on a specific number of FPGAs, instead of providing an expensive license to program an unlimited number of devices using their IP. However, apart from legal obligations there is no reason a customer cannot create more devices than allowed, or extract the bitstream from a chip and use it to make illegal copies.

A PUF is here used to limit the execution of the IP to specific devices using the unique PUF fingerprint of each device. The company gives out customized bitstreams that can only be used on a specific set of FPGAs by means of some kind of locking mechanism that is heavily intertwined with the actual IP, to avoid

copying or reverse engineering of the IP. Such a scheme is studied for instance in [9], where additional state transitions are added to the IP and the correct transition is only picked when their Anderson PUF outputs the correct response each time.

2.3 Designs

In [3] a broad range of PUF types and categories is enumerated. Only a small selection of designs applicable to FPGAs is described.

2.3.1 Ring oscillator PUF (ROPUF)

As the name implies, a ring oscillator PUF or ROPUF uses variations in the frequencies of ring oscillators to derive PUF bits. It was introduced together with the notion of a silicon PUF itself by [2]. First the ring oscillator itself will be defined before describing how this is used to generate PUF bits.

Ring oscillator

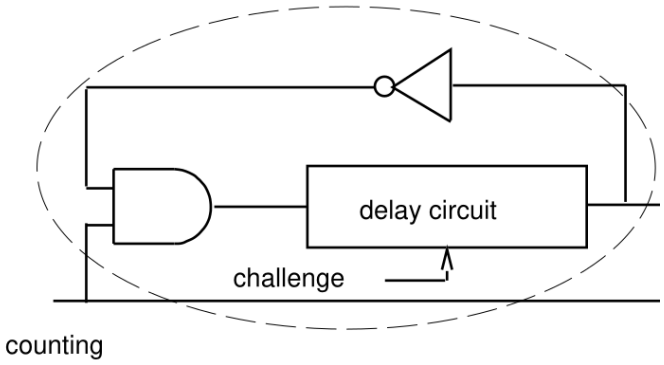

Figure 2.2: Schematic of a ring oscillator. Figure from [2]

The schematic of a ring oscillator is shown in figure 2.2. It consists of a closed loop with an inverter and a delay circuit, typically consisting of a chain of LUTs. Such a circuit will oscillate on its own since the inverter input and its output of opposite value are connected through the delay line. Most often an AND gate is added with an extra enable signal, such that the oscillator can be started or

stopped at any time. Optionally additional challenge bits are incorporated into the delay line, which is shown in figure 2.3. These Programmable Delay Lines (PDLs) are realized on an FPGA by adding challenge bits as extra inputs to a LUT. The challenge bits do not change the output of the LUT but do cause a different SRAM cell to be selected, resulting in a slightly different delay path.

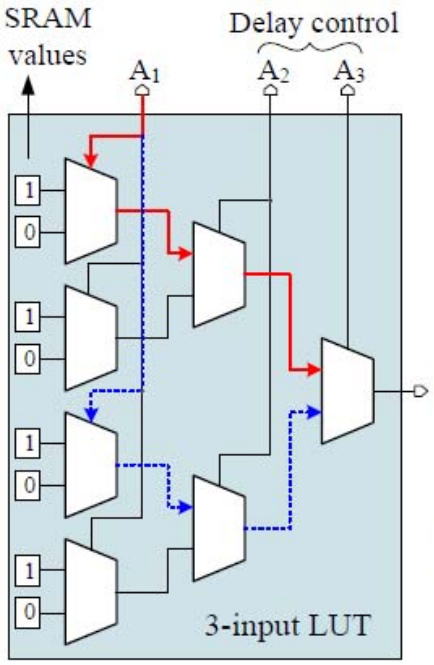

Figure 2.3: Example of a PDL for a 3-input LUT. Figure from [10]

ROPUF

When putting multiple ring oscillators on a chip, each one will oscillate at a different frequency, even for equal challenge inputs, due to manufacturing variations. The outputs of two ring oscillators are then connected to counters, which count the rising edges of the oscillators, yielding a measure of the frequency of each one. This mechanism is visualized in figure 2.4. After a fixed time period, most often a certain number of cycles of a reference clock, the two counters are compared. Depending on which of the two oscillators has the highest frequency, a zero or one is emitted. By comparing then multiple pairs of ring oscillators in sequence, a complete PUF response is generated.

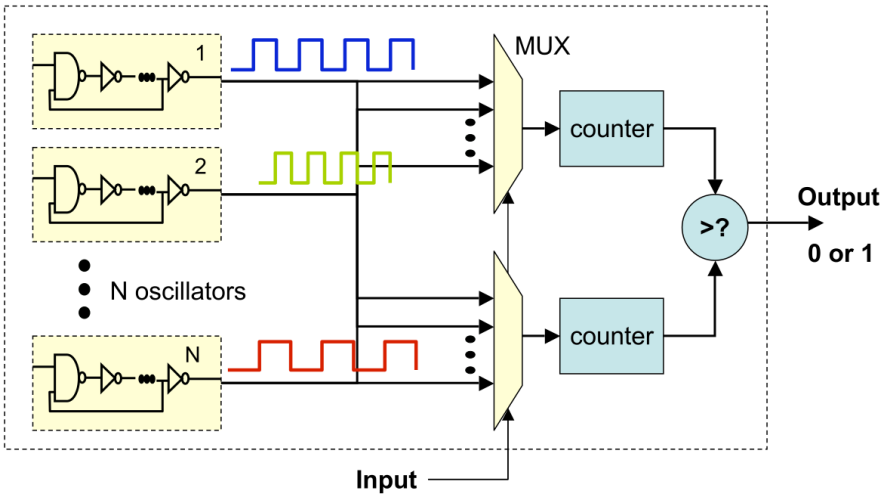

Figure 2.4: Main building block of an ROPUF. Figure from [6]

2.3.2 Anderson (Glitch) PUF

The Anderson PUF [11] was the first PUF specifically designed for FPGAs. It uses the specific internal layout of the Xilinx Virtex-5 to generate glitches due to differences in signal propagation delays. More in detail, it uses two LUTs on the same carry chain as shown in figure 2.5. The output of LUT A will toggle between zero and one each clock cycle, while LUT B will output the inverse pattern. When LUT A and its multiplexer are faster than LUT B and that multiplexer, a short positive pulse will appear at N2. This happens when LUT A switches from multiplexer input 0 to input 1, while signal N1 is still one because the multiplexer of B will not yet have switched from input 1 to input 0. The presence or absence of the pulse is then used as the PUF response bit.

The first disadvantage of this design is that it is specific to the internal FPGA layout, such that it even required some changes to work on the Virtex-7 series as shown in [12]. Another problem studied in [13] is that the response changes over time, such that it has to be measured at a fixed time after power-on of the FPGA. A last remark is that the Anderson PUF does not incorporate challenges, such that the only way to generate multiple responses is to implement multiple PUFs on the same chip.

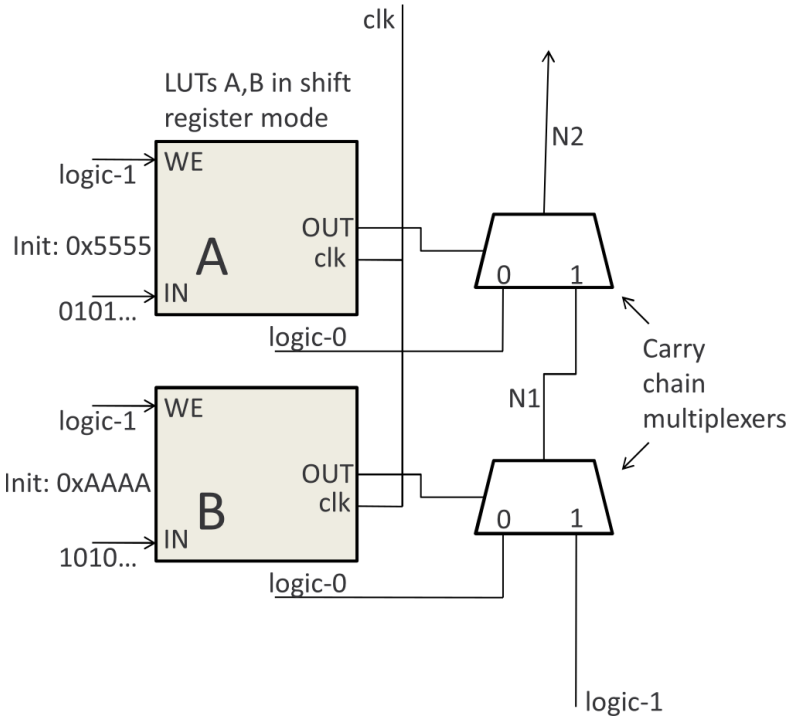

Figure 2.5: Main idea of the Anderson PUF. Figure from [11]

2.3.3 Arbiter PUF (APUF)

The arbiter PUF was introduced by [14] to reduce the long measurement times of ring oscillator PUFs. Instead of comparing the timing of delay circuits by putting them inside ring oscillators and comparing the resulting frequencies, a single rising edge is sent over two symmetric delay paths and an arbiter signals on which of the two paths the edge arrives first. The idea is sketched in figure 2.6, where the delay paths consist of chained switch blocks that take challenge bits as extra input. Depending on the challenge bit, the switch block either let the two paths pass through or crosses them over. This results in an amount of possible responses that is exponential in the number of challenge bits.

However, as already shown in [14] it is clear that the delays of the arbiter PUF can be modeled using an additive delay model, in which the delays of different wires and primary elements can be estimated separately and added together, such that the PUF output bits can be accurately predicted.

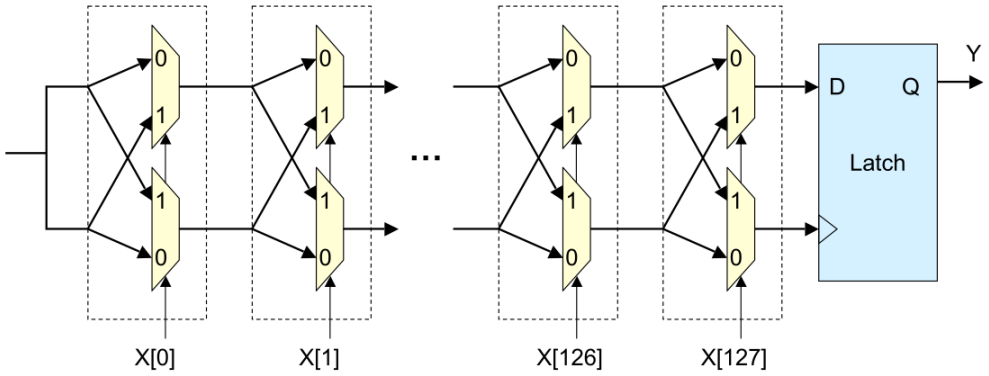

Figure 2.6: Schematic of an APUF. Figure from [2]

XOR'ing multiple arbiter PUF bits together has been proposed in [6] in order to reduce predictability. Experiments in [15] have shown that this strategy does not work and that is can even have a negative effect on predictability and uniqueness. They proposed the double arbiter PUF, which compares the corresponding signal paths between different arbiter PUFs instead of the symmetric paths of a single switch block chain. This design and its variants, the 2-1, 3-1 and 4-1 double arbiter PUFs, do have a significant positive effect on the predictability and uniqueness of the outputs.

2.4 Comparison of PUF designs

	Response length w.r.t. area used	Specific to FPGA layout	Repeated measurement	Uniqueness
ROPUF	quadratic	no	yes	high
APUF	exponential	no	yes	potentially low
Anderson PUF	linear	yes	no	high

Table 2.1: Comparison of properties between different PUF designs

As an overview, some high-level characteristics of the different PUF designs are given in table 2.1. The first property is the number of different PUF bits that

can be generated using a certain FPGA area. This number is quadratic for the ROPUF as we do pairwise comparisons of ROs. The APUF has an exponential number of challenges since each additional switch block adds another bit to the challenge, doubling the amount of CRP pairs. However, these responses can be accurately predicted using machine learning attacks as in [15]. Anderson PUF bits, on the other hand, are entities on their own which do not support challenge inputs. Therefore each added element only adds a single extra PUF bit.

In addition, the Anderson PUF needs a specific internal layout of the slices on the FPGA, a constraint which is not shared by the RO- and APUF. Another problem with the Anderson PUF is the fact that its response can only be measured once per power-on cycle, which makes conducting experiments more cumbersome.

A final property that is worth mentioning is the uniqueness of responses. The uniqueness value for the ROPUF and Anderson PUF is generally close to the ideal value of 50%, while for the APUF good uniqueness is not at all guaranteed, as shown in [5] and [15].

2.5 Error correction

When cryptographic keys are to be derived from PUF responses, these responses must be identical each time the PUF is evaluated. However, the responses are inherently noisy, even without the larger effects of environmental variations such as temperature and operating voltage or the effect of chip aging. Therefore some error correction method is needed to increase the reliability of the PUF responses.

Often [16] is cited as the base work on deriving keys from noisy data without revealing more information than necessary about the secret data. The Bose-Chaudhuri-Hochquenghen (BCH) code is discussed there, which is popular in PUF Error Correction Coding (ECC), used for example in [6], [12], [17]. A BCH code is characterized by the tuple (n, k, t). In the case of a PUF, for this BCH (n, k, t) code the PUF generates k bits, to which $n-k$ generated syndrome bits are added, resulting in a code block of n bits in total. t is the maximum number of errors the code is able to correct. As an example, the code BCH(255, 155, 13) is able to correct up

to 13 errors in a code block of 255 bits. This codeblock is made up of 155 secret PUF bits and 100 publicly known syndrome bits. The syndrome is generated by the BCH encoder using a reference PUF response obtained by averaging multiple measurements.

Another ECC scheme, this one specifically designed for PUFs, is Index-Based Syndrome (IBS) coding introduced by [18]. When the PUF bits are independent and identically distributed (i.i.d.), IBS coding can be proven to be information-theoretically secure. In addition, if the PUF does not only output the hard bit values, but can also provide confidence values, IBS supports soft decoding which yields a coding gain with respect to hard decoding that is used in other coding schemes. In the case that the i.i.d. assumption does not hold however, information about the secret key is leaked through the IBS syndrome and the effects on security are even worse than originally expected, as discussed in [19].

Chapter 3

PUF implementation

3.1 Basic design

The first choice to be made when implementing the PUF, is the the variety of PUF to use (ROPUF, APUF, etc.). In section 2.4 the different types are already compared to each other. Here the choice was made for a ring oscillator PUF, due to ease of implementation on an FPGA. The Anderson PUF is very dependent on the internal layout of the FPGA and the time of measurement, while there is no possibility for a challenge-response scheme. The Arbiter PUF is also harder to implement as the two racing path should be of exactly the same length, which is not a trivial task on an FPGA.

Another important point is the communication with the outside world, in our case mostly a laptop on which the development happened. The two ports that were available both on the laptop as on the Nexys A7 board were the Ethernet and USB ports. As the main focus lies on the development of a PUF and not on the communication protocol, the simplest possible option was chosen, in this case the UART protocol over USB. With the Nexys A7 board the UART communication happens over the same cable that is used for programming of the FPGA, so no extra cables or setup were needed. In addition, Vivado comes with an IP core for UART communication, which makes it even easier to implement.

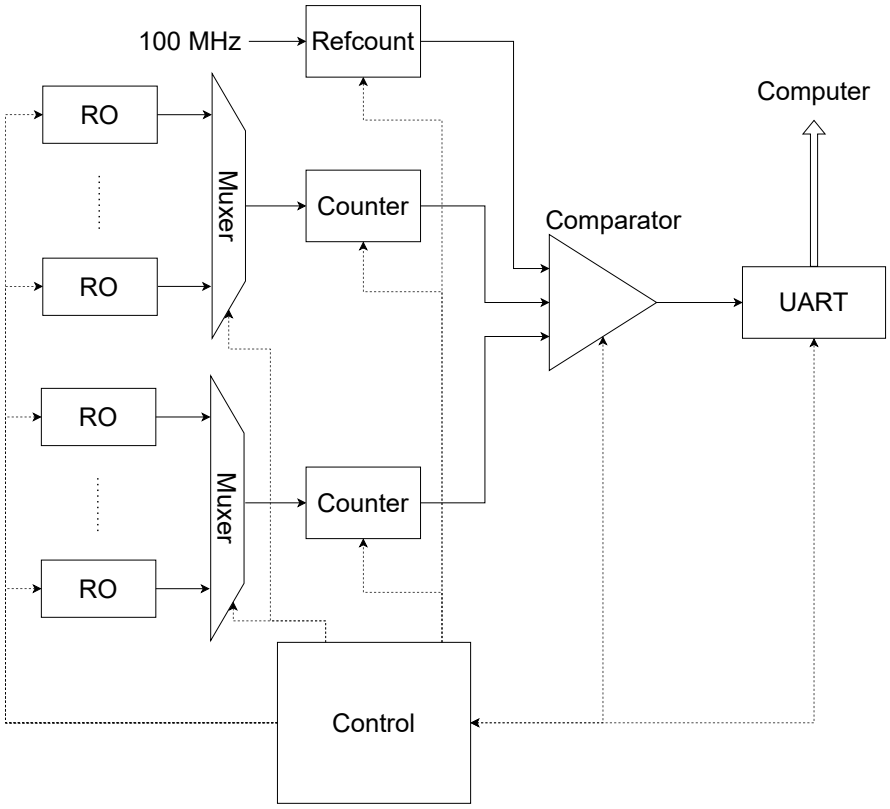

Figure 3.1: High level overview of the basic PUF

Our ROPUF is inspired by the design shown in [10]. An overview is given in figure 3.1, the differences with [10] are listed in the following paragraph. It consists of two ring oscillator arrays of 32 ROs each, such that 32×32 different PUF bits can be generated. The control block activates one oscillator from each block at a time and selects the appropriate RO output using the multiplexer. The two oscillator outputs are then each fed into a counter, while another reference counter has as input the main clock signal. After a fixed number of clock cycles, the values of the two counters are compared and a zero or one is emitted depending on which of the values is higher. The resulting PUF bits are then sent to a connected computer over UART. The individual blocks will be discussed more in detail in the following sections.

The are a few differences between our design and the one of [10]. To start, our ring oscillators do not accept any challenge bits. It is unclear why [10] added them in

the first place, as their challenge is fixed for each RO and therefore does not increase the number of different PUF bits. Consequently, they are left out in our design. Furthermore the number of ROs in each array is doubled w.r.t. [10], such that the number of different PUF bits is increased by a factor 4. The other differences are mainly subtilities in the control logic. The value of the reference counter is not read by the control block, but the comparator block which determines the time of comparison instead. Also the LFSR, described in section 3.1.4, is increased in size to accommodate the increased number of ring oscillators. The initial state of this LFSR is not determined by input from the computer, but is a fixed value. After startup the PUF then continually cycles through all possible states of the LFSR, outputting a continuous periodic stream of PUF bits instead of a fixed block of 256 bits. Therefore, the PUF bits are not saved in a large shift register but are packed per 8 in a byte and sent directly to the computer by the UART block. Further details are given in the following sections.

3.1.1 Ring oscillator

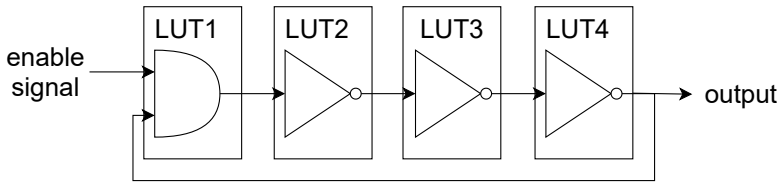

Figure 3.2: Schematic of the used ring oscillator

Our ring oscillators consist of 3 inverters and an AND gate, each realized by separate LUTs, as illustrated in figure 3.2. These 4 LUTs then fit perfectly inside a single FPGA slice for a compact design and to avoid inter-slice routing. Once the routing would leave the slice, additional constraints would be necessary to keep these routes identical for each oscillator.

The first LUT functions as an AND gate on the oscillator output which is fed back and the enable signal used to activate the RO. The next three LUTs each act as an inverter through which the signal propagates, in order to create a delay path. The output of the third LUT is sent to the multiplexer and is fed back to the input.

3.1.2 Muxers, counters and comparator block

The multiplexer block just select the active ring oscillator from the array, whose index is given by the control block. All counter blocks in the design simply count rising edges on their input, with the difference for the refcounter that its input is not an RO output but the main clock signal. When the reference counter hits a certain fixed value, the comparator block will compare the values of both RO counters. Based on these two values the comparator will output a zero or a one and it will set a 'finished' signal high to indicate that the result is available. The control block is responsible to reset the counters and comparator after each PUF bit result.

3.1.3 UART block

The UART block is connected to the comparator and waits for PUF bit results. Each time a result is ready it is stored in an 8-bit shift register. Once this register is full, this is signaled to the control unit so the generation of bits is paused until the buffer is available again. The byte is then sent to the Xilinx AXI UART Lite IP core [20] which handles the actual UART communication.

3.1.4 Control block

The control block manages the operation of the whole PUF design. It starts by generating a challenge using a 10-bit Linear Feedback Shift Register (LFSR), which generates a pseudorandom sequence. The linear feedback part means that the input to the shift register is determined by a linear function on the bits currently in the register, in this case the function is successive XOR'ing of bits at certain fixed locations. The state of the LFSR is the value of the shift register. Since the number of possible states is finite and the state transitions are deterministic, the LFSR will cycle through a number of states periodically. A maximal length LFSR will cycle through all possible states with the exception of the invalid zero state, from which no other state can be reached. The LFSR used is our design takes the XOR of the seventh and tenth bit in the register as shown in figure 3.3, which can be proven by simple brute force to be a maximal length LFSR which cycles

through the $2^{10} - 1 = 1023$ possible states.

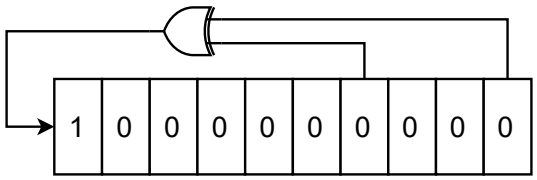

Figure 3.3: The specific LFSR used in our design

For each new PUF bit, the control block generates a new challenge by shifting the LFSR by one bit. The first five bits in the register are used to select a ring oscillator in the first array, while the last five bits select one in the second array. The same values are sent to the multiplexers to select the correct RO output. At the same time the reset signal on the counters and comparator will be lifted such that the counting starts too. When the PUF bit is ready, the control block disables counting and generates the next challenge. If the UART block is ready to accept new bits, generation of the next bit is started, otherwise the control will wait for the UART block to become ready.

This generation of PUF bits will continue indefinitely, starting right after the reset signal deactivates, without any other external impulses. Due to the periodicity of the LFSR the output stream will also be periodic, apart from noise, with a period of 1023 bits. The stream can then be observed for a certain amount of time and the split into blocks for further analysis.

3.2 Adding error correction (ECC Design)

After the basic design, error correction capability was added. Now there is two-way communication with the computer. First, the PUF waits for a command from the computer. Either this is the character 'p', which stands for provision. In the provisioning step the PUF responds with a raw response without error correction decoding. From such multiple raw responses a reference response can be determined, which is then used to calculated the syndrome bits. When the syndrome is available, the computer can send an 's', followed by the syndrome.

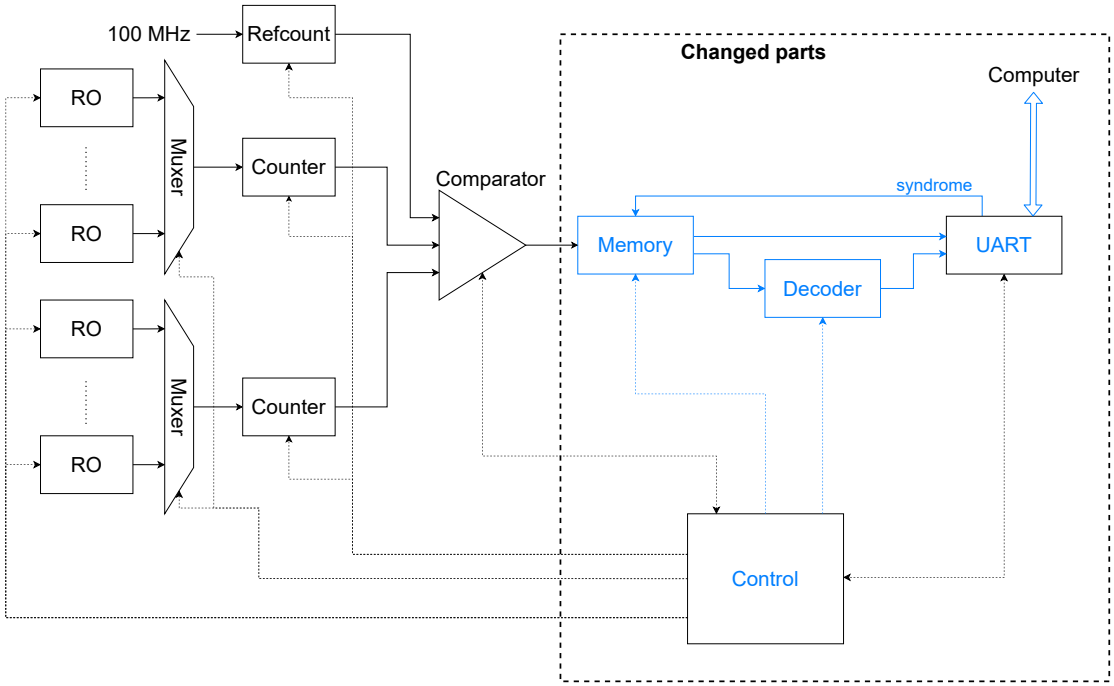

Figure 3.4: PUF design with BCH decoding added. New or changed elements with respect to the basic design are in blue.

The PUF will then use the syndrome for error correction on the raw response using the decoder. The new design with the added or changed blocks is illustrated in figure 3.4.

For the error correction part a BCH decoder is used, because of the popularity in literature and the availability of a ready-to-use BCH code design. The tool from [21] was chosen as it can generate decoders in VHDL for all possible codes with a block size up to 1023. More details on the specific BCH code used are given in section 4.3.

3.2.1 Changes w.r.t. the basic design

Both the control and UART block are changed and made more complex to incorporate the ECC part. The resulting state diagrams and communication between the two blocks are sketched in figure 3.5. More details on their operation are given in the following paragraphs.

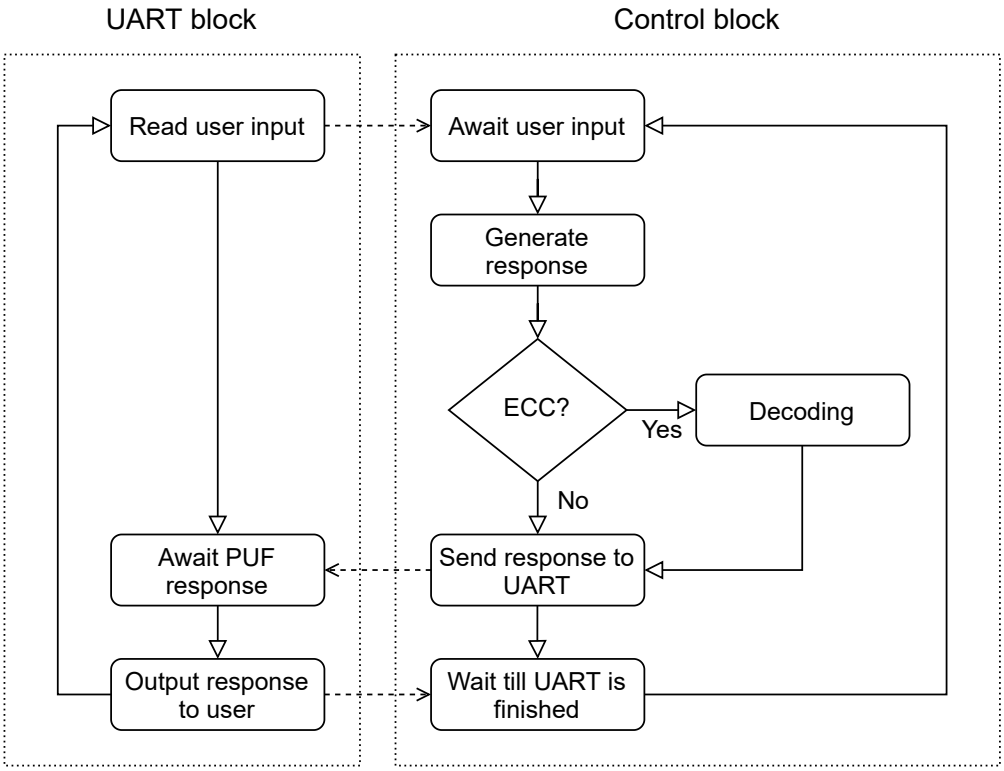

Figure 3.5: State flowchart of the control and UART blocks in the ECC design

Instead of immediately starting with PUF bit generation, the control block waits for the UART block to read the input from the computer. The wanted type of response is read, which is either a 'p' for a raw response or 's' for an error corrected response. In the case of error correction also the syndrome is read from the input. When all input is processed, the syndrome is sent to the memory block and the type of wanted response is signaled to the control block.

The control block then starts to generate the PUF bits in the same way as the basic design, as described in section 3.1.4. The control block counts the number of generated bits and stops the generation when the correct number of bits is ready. At this point there is a difference between the scenarios with and without error correction.

If no error correction is needed, the bits are saved directly in the UART block and are sent back to the computer directly after the generation of the response

is finished. In the case of error correction decoding, the response bits are saved in the memory block alongside the syndrome. When the raw response is ready, the control activates the decoder and commands the memory block to send the response and syndrome bits to the decoder block. The output of the decoder is then saved in a shift register in the UART block. After the decoding step the output is then sent to the computer.

3.3 Development and functional validation

All functionalities were written in VHDL and synthesized, implemented and simulated using the freely available Xilinx Vivado Design Suite HL WebPACK Edition installed on Arch Linux. Everything was tested on a Nexys A7 board with Artix 7 FPGA (part XC7A100T-1CSG324C). The first step was installing the Vivado software, USB drivers and Digilent board files, where the guide from [22] was really helpful. The Digilent board files enable easy selection of the correct part and automatic configuration for several board components such as the clock or I/O pins.

The first part that was developed is the communication with the computer. Since the UART Lite IP core should work out of the box with the UART USB bridge on the Nexys board, the main problem was the usage of this IP core, which uses the Advanced Microcontroller Bus Architecture (AMBA) AXI4-Lite protocol to interface with other blocks. The UART Lite IP core documentation [20] did not specify this to communicate with the core, apart from some technical details, and also Xilinx's AXI reference guide [23] did not clarify the working of the AXI4 protocol. Only by digging into the AMBA AXI specification [24] itself, it became clear which signals to use and when. After further experimentation we were able to read and write in UART both in simulation and on the real board. Detailed information on how to use the IP core is given in appendix A.

The next step is the writing of all the control logic for the basic design. Since the behavior of the ring oscillators depend on the real hardware, they cannot be easily simulated. Therefore the RO arrays are represented by external inputs in the simulation who all oscillated continuously with different periods. The control

block, multiplexers, counters and comparator were written and tested on these artificial ROs in behavioral simulations. Once all these components worked as expected, The ring oscillators were added together with enable signals etc. At this point only timing simulations can be used because functional simulations cannot handle the closed loops of the ROs. After some last timing simulations to check if everything still functioned correctly, the focus was shifted towards the implementation and constraining of the ROs on the FPGA, which is the topic of section 3.4.

After the basic design was fully functional and analyzed, the design was extended to include ECC capability to increase the reliability of responses. The encoder and decoder of [21] were first simulated separately on their own, to test if they worked correctly and to figure out the exact input and output protocols of the blocks, as this was not specified in detail in [21]. When performing initial tests on a BCH(15, 5, 3) code, the output of the encoder block could not be correctly decoded by the decoder. After cross-referencing with the Matlab implementation of BCH it became clear that the encoder did not provide the correct syndrome bits. The output of the Matlab BCH encoder could however always be decoded correctly by the VHDL decoder, also for larger BCH codes. Therefore only the decoder was incorporated in the design.

In addition to the decoder block from [21], a memory block was added to temporarily store the raw PUF response and the syndrome bits. Furthermore the changes described in section 3.2.1 were made to the UART and the control block to accommodate for the ECC part. The extended design was validated in timing simulations and by analyzing the output on the real hardware.

The following section will focus on a critical point of the PUF design, namely the layout of the ring oscillators on the hardware.

3.4 Implementation on the FPGA

The PUF bit extraction relies on the fact that the different ring oscillator have slightly different frequencies due to manufacturing variations that result in non-

identical delays paths in each RO. However, in order to measure these delay differences, each RO must be laid out in exactly the same way on the chip, otherwise these hardware variations are overshadowed by the differences in wire lengths due to varying layouts of the ring oscillators. Another point to consider is that the synthesis or implementation tool might try to optimize the ROs, for example by merging the three inverter LUTs into a single inverter, which is logically equivalent. This is completely unwanted behavior in this case, as we explicitly want to introduce an extra delay path and want to exercise full control over how the RO is realized on the hardware.

3.4.1 Preventing optimizations

In the VHDL code of the ring oscillator block, the four LUTs are explicitly instantiated using Xilinx's UNISIM library. Unfortunately, this does not prevent the optimizers of the implementation tool to merge multiple LUTs into one. After synthesis, the RO block still contains four distinct LUTs as shown in figure 3.6, exactly as we would like. However, after implementation the four LUTs are merged

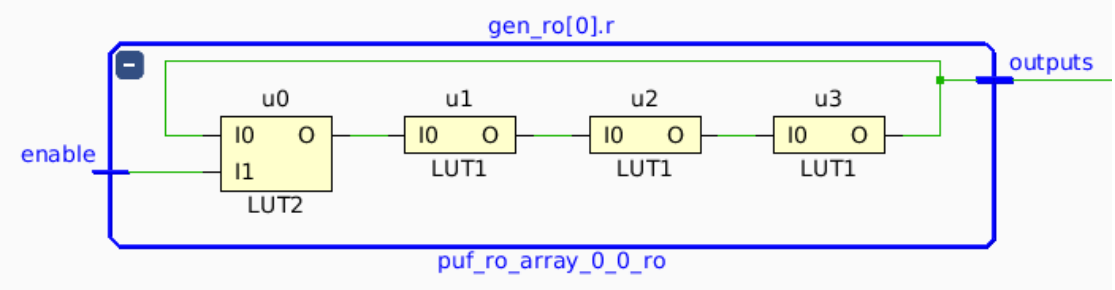

Figure 3.6: Ring oscillator schematic after synthesis

into a single one, visible in figure 3.7. Even if this structure is logically equivalent, it does not have the timing properties or hardware layout we want to achieve.

The merging of these LUTs can be prevented by using the DONT_TOUCH attribute in the VHDL code or in a separate constraint file. More details on this can be found in appendix B.

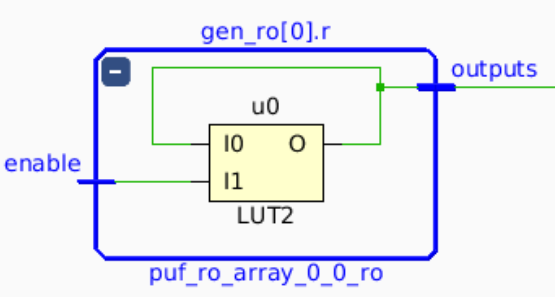

Figure 3.7: Ring oscillator schematic after implementation

3.4.2 Explicit hardware layout

Even when the four LUTs of each RO pass untouched through the optimization phases, they are still placed randomly on the FPGA. This leads to the second requirement, being the identical layout of each RO. This is achieved by further addition of several location constraints. First the LUTs are put inside a single slice on the FPGA by setting their relative (slice) location, then another constraint specifies the exact LUT to use within that slice.

Furthermore all ROs are put close to each other in a rectangular block on the FPGA. This makes it easy to put this block in different locations on the chip in order to mimic measurements on multiple FPGAs. This is accomplished by assigning each RO block a specific slice location on the FPGA. As this is quite tedious to do for 64 RO blocks, a Python script was written to automatically generate this list of constraints given the number of ring oscillators and a starting position. The specific attributes and commands used to constrain the ROs are given in appendix B.

The final hardware layout of both designs are visualized in figure 3.8 and 3.9. The orange 8 by 8 rectangle contains the 64 constraint ring oscillators, where each small orange block is a slice containing a single RO. The blue rectangles signify all the slices used by the rest of the PUF design. Their size depends on the number of elements within the slice that are in use, being the number of LUTs, FFs, muxers and carry chain. It is also clear that the ECC design needs more area, mainly for the BCH decoder which is located at the light blue 'island' on the bottom right in

figure 3.9.

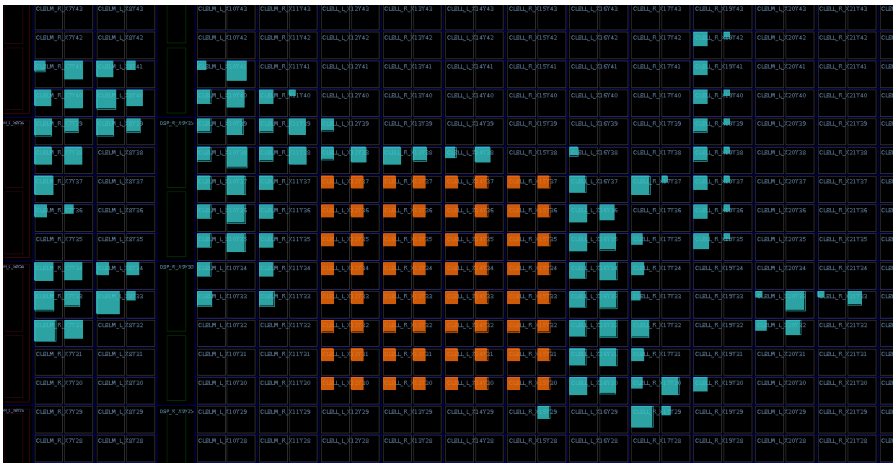

Figure 3.8: The layout of the basic design on the FPGA. Each rectangle represents a single slice, while the size indicates how many elements inside the slice are used.

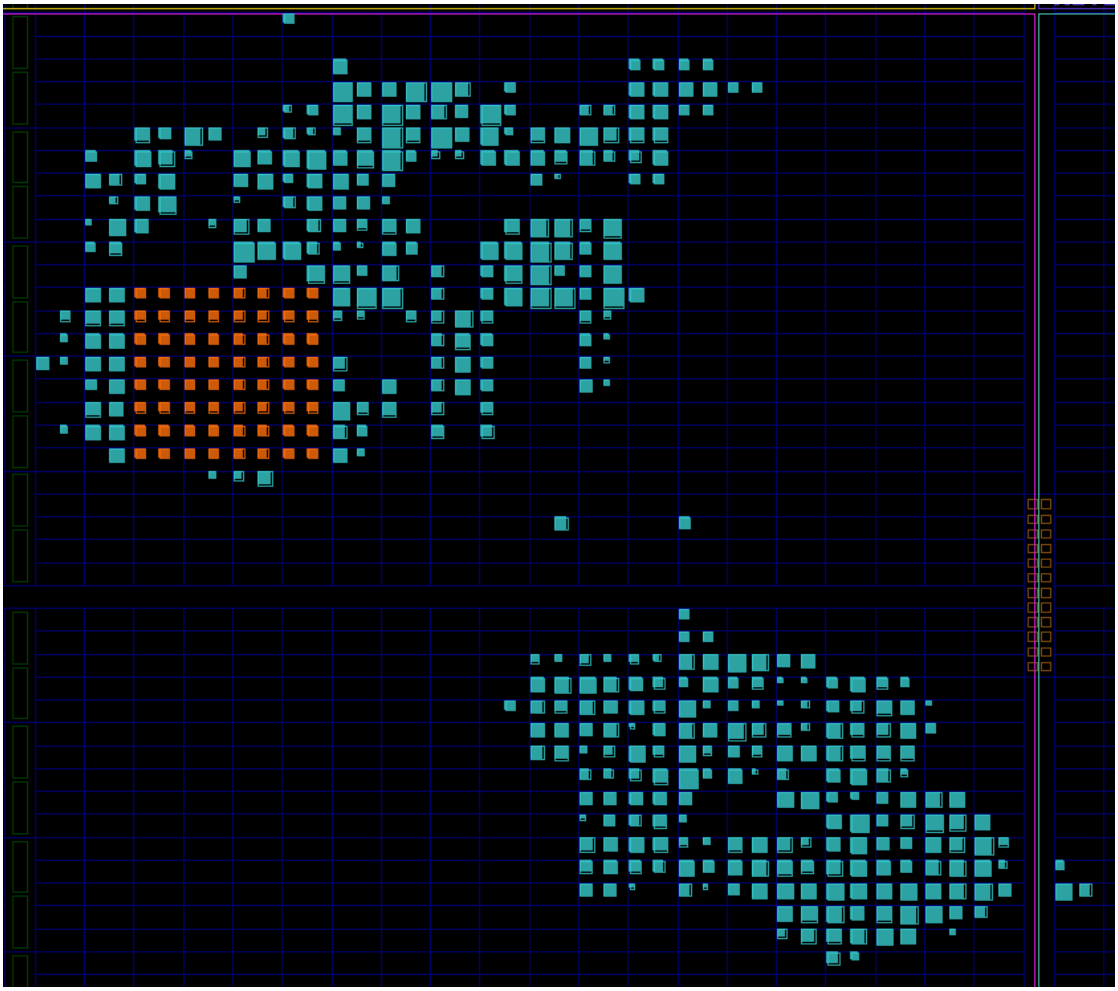

Figure 3.9: The layout of the ECC design on the FPGA. Each rectangle represents a single slice, while the size indicates how many elements inside the slice are used.

In figure 3.10 a more detailed layout of the top right corner of the RO array block in the basic design is provided. There we can see the individual slice elements that are occupied by the design. On the left side of each slice there are 4 LUTs, next to it we have multiplexers, then a carry chain and finally on the right 8 FFs. In this zoomed picture we can clearly see the rigid structure of the ring oscillators in orange, where each RO occupies all 4 LUTs of the slice.

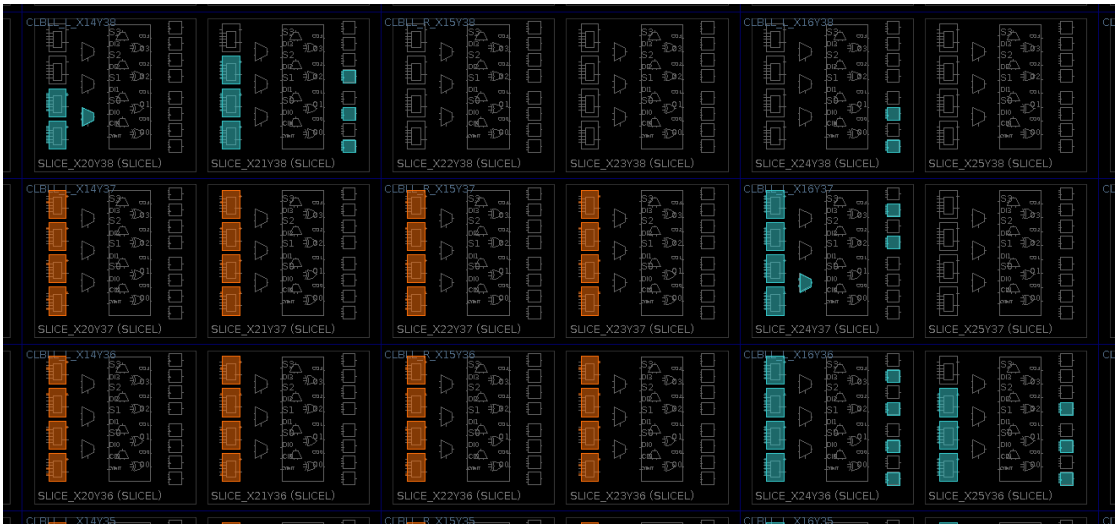

Figure 3.10: Zoom of the FPGA layout of the basic design

Chapter 4

Experiments and Results

4.1 Determination of the measurement time

Before conducting further experiments on the PUF, the time during which the frequency of each ring oscillator is measured has to be determined. The goal is to achieve more that 99% reliability as in the literature, in the least amount of measurement time. Therefore the reliability in function of the time after which the counters of the ROs are compared is studied. Measurement times of 100, 200, 500, 1000, 2000, and 5000 reference clock cycles were studied, with the reference clock operating at 100 MHz.

For each measurement time, at least 100 000 blocks of 1023 bits were obtained to get a good estimate of the reliability. The blocks are averaged to get the most likely reference response, which is then compared to each block. The results are plotted in figure 4.1. The first value where the reliability exceeds 0.99 is at 200 clock cycles. For a 100 MHz clock this means it takes at least 10 ns $\times$ 200 $\times$ 1023 $\approx$ 2 ms to generate a PUF response of 1023 bits. A higher value than this 200 clock cycles was not chosen, because it is much more efficient to add error correction coding on top of the raw response. As an example, the BCH(255, 171, 11) code needs 537 clock cycles to correct a *block of 171 PUF bits*, which yields orders of magnitude better error rates than increasing the measurement time by multiple

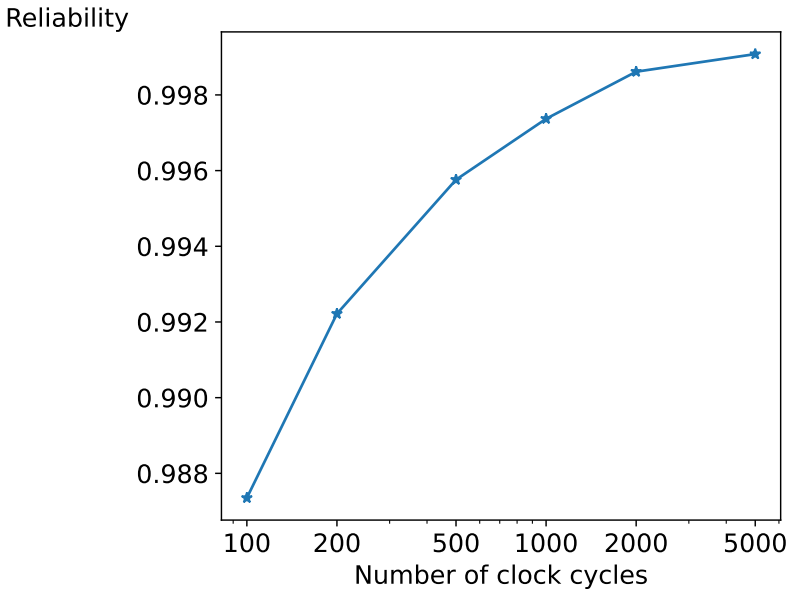

Figure 4.1: PUF reliability for different measurement times

hundreds of clock cycles for *each individual PUF bit*.

4.2 PUF Properties

Once the basic design was complete the properties were determined by extracting 1000 times the 1023 bit response from the FPGA. In order to simulate the measurement of different chips, the ring oscillator arrays were put successively in 8 different locations spread over the FPGA.

	This work	[10]	[5]	Ideal value
Uniformity	51.01%	50.61%	50.56%	50%
Reliability	99.19%	99.16%	99.14%	100%
Steadiness	98.64%	-	98.51%	100%
Uniqueness	47.86%	47.13%	47.24%	50%
Bit-aliasing	51.01%	-	50.56%	50%

Table 4.1: Performance of the designed PUF and comparison with other papers

The results are listed in table 4.1 together with the results from two other papers. The first compared paper is [10] whose design was the inspiration for this work. Their ROPUF was tested on 5 45 nm Spartan-6 FPGAs. The other paper is [5], which defined these PUF properties and used them to evaluate an ROPUF on 193 Spartan-3E FPGAs with 90 nm technology. We see that the uniformity and average bit-aliasing is about half a percent worse, while the reliability and steadiness are nearly equal and the uniqueness is more than 0.6% better in comparison with the other papers. Overall we can conclude that our design's performance is comparable with the state of the art.

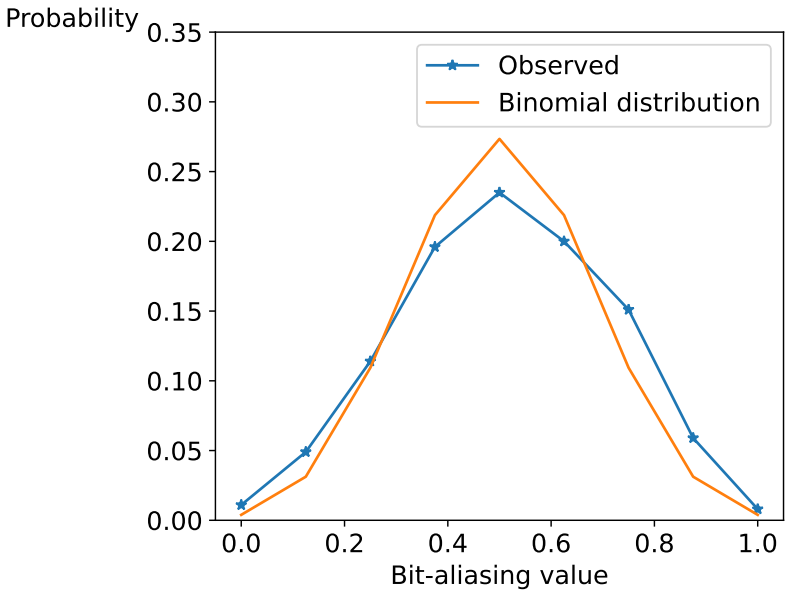

Figure 4.2: Distribution of the individual bit-aliasing values

In table 4.1, only the average bit-aliasing is mentioned. However, it is possible that there are considerable biases in individual bits, which cancel out each other. The bit-aliasing values of each bit were therefore put in a histogram which is shown in figure 4.2. Since only 8 different positions are measured, there are 9 possible values for the bit-aliasing (0/8 up to 8/8), with an ideal value of 50%. If the bits would be perfectly random, we would obtain a binomial distribution of 8 experiments with a 'success' probability (probability of a 1) of 0.5. This distribution $B(8, 0.5)$ is also plotted in figure 4.2. As we can see, the distribution of bit-aliasing values is

comparable to what can be expected from the ideal case. It follows that there is no problematic bias in individual PUF bits along different devices or device locations.

4.3 Error correction

Now that the basic PUF properties are determined, the reliability is increased by adding BCH coding. The goal is to have an error probability of less than 1 ppm. In order to limit the footprint of the decoder on the FPGA and to ease experimentation, the blocks are cut to 255 bits.

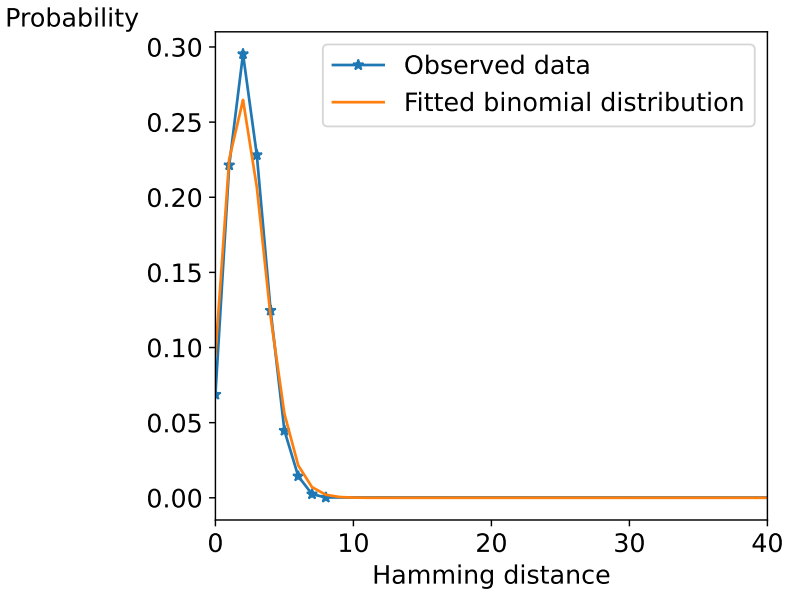

Figure 4.3: Intra-chip Hamming distance histogram of a 255-bit response of the basic PUF

From the measurements obtained previously, the histogram of Hamming distances with respect to the cut reference response are plotted in figure 4.3. A binomial distribution is fitted on top of it, with n the block size of 255 bits, and the calculated error probability of $p = 0.0092$. Using this binomial distribution, the smallest Hamming distance that occurs with probability less than 1 ppm, is 12. This is thus the maximum number of errors that the BCH code should be able to handle.

However, for the BCH(255, 163, 12) code of block size 255 which can correct up to 12 errors, there are $255 - 163 = 92$ syndrome bits and only 163 actual PUF bits, such that the maximum number of errors decreases. Therefore a block size of 163 was then considered, which results in a lower number of expected errors, such that the number of PUF bits can increase again. This iterative process converges after two steps, giving the final code BCH(255, 171, 11) with 171 PUF bits and 84 syndrome bits.

4.3.1 Reliability with BCH decoding

The reference response needed to calculate the syndrome for BCH decoding was determined by averaging 1000 raw PUF responses. The syndrome was then obtained using Matlab's built-in BCH encoding function `bchenc`. This syndrome was then put in a script `getdata.py` that continually sends the syndrome to the PUF in order to obtain an error corrected response.

For the BCH(255, 171, 11) code, not a single error was observed in multiple batches of more than 10 million bits each, spread over multiple days, for a total of more than 100 million bits. Therefore, the error correction capability of the BCH code was lowered to (255, 179, 10), but even then no error was found in more than 40 million bits. The next code in the row was then tried, namely BCH(255, 191, 9). In this code there were some errors in the 47 million extracted bits, giving a reliability of 99.9991455% or an error rate of 8.545×10^{-6}. When looking at individual responses, 41 out of 246309 responses had errors in them, which is less than 2 per 10 000 responses.

We can conclude that the initial estimation of the error rates was too pessimistic, and that the BCH decoding is capable to significantly increase the reliability of responses, beyond what is needed for practical purposes. However, the effect of environmental variations is not studied here. In [25], temperature variations did not have a large impact, but difference in the power supply voltage could cause up to 15% change from the reference response. To account for this case a more robust BCH code should be chosen.

4.4 FPGA area and power usage

4.4.1 Area consumption

The area consumption on the FPGA of the basic design and the ECC design are summarized in table 4.2. As a reference, the design from [10], on which our basic design is based, is added for comparison. The total number of slices in our design more is than twice as large as the amount of slices in [10], although the number of response bits generated by our design is four times larger. By halving the number of ring oscillators in our design, which already takes up 90 slices, the response length would become a comparable 255 bits and the total area would decrease 45 slices in the RO arrays, plus additional reductions in the surrounding control logic. It can be concluded that our design does not use significantly more area than a comparable design.

	LUTs	FFs	Total slices	Response bits
Basic design	481	322	182	1023
Design from [10]	-	-	82	256
BCH decoder	541	627	196	171
Total ECC design	1248	1495	478	171
BCH decoder	1269	1621	423	798
Total ECC design	2423	4037	1045	798
PUFKY design [17]:				
decoding	-	-	149	2226
ROPUF + decoding	-	-	1101	2226

Table 4.2: Area utilization of both designs in comparison with other papers

For our ECC design a comparison was made with the PUFKY in [17], which also uses a ROPUF and BCH decoder (+ repetition code). The decoder is mentioned separately for each design, and the ROPUF plus the decoding parts are summed to be compared with our total ECC design area consumption. The PUFKY uses a larger BCH code (318, 174, 17) than our BCH(255, 171, 11), while our decoder takes more area. This is probably due to our use of an older, less optimized BCH

decoder. Also the response bit length is much larger for the PUFKY design. It realizes this larger response length by decoding multiple blocks in series, while our design is restricted to a single BCH block.

By enlarging our BCH code to (1023, 798, 23), we use more of the bits available in the RO arrays. However, while the area consumption for our design with response length 798 is comparable to the PUFKY, the PUFKY is still able to generate much more response bits for the same area. In order to alleviate this problem we should use a more optimized decoder and decode multiple smaller blocks in order to achieve the PUFKY's area efficiency.

4.4.2 Power usage

The power usage of both designs was estimated using the built-in Vivado power estimator. The switching activity of the designs was determined by a post-implementation timing simulation run for 1 ms. The results are shown in table 4.3, where the total power is also split into the static power consumption, the power consumed by the clock management module and the dynamic power used by the BCH decoder and the rest of the PUF.

	Basic design	ECC design
Total power	208 mW	219 mW
Static power	97 mW (47%)	97 mW (44%)
Clock manager	106 mW (51%)	106 mW (48%)
PUF (without decoder)	4 mW (2%)	5 mW (2%)
BCH decoder	-	11 mW (5%)

Table 4.3: Power usage of both designs

The PUF power consumption is most likely higher than predicted by the software as the behavior of the ring oscillators is not simulated well. In simulation the RO periods are not below 2 ns while according to the Artix 7 data sheet [26] the period should be close to 0.5 ns. A small experiment indeed showed that the ROs oscillate around 20 times faster than the 100 MHz clock. This higher switching activity

results in an increase in actual power consumption, although the total PUF power usage will still be negligible compared to the static power and the power consumed by the clock management module.

4.5 Demo

In order to demonstrate the usage of our PUF and to provide a possible starting point for further applications, a demo was created to showcase the FPGA PUF. A screenshot of the demo's output can be seen in figure 4.4. There are two scripts involved, `demo_puf.py` starts by collection 100 raw responses from the PUF, which are then averaged to obtain a reference response. Since no suitable BCH encoder implementation in Python was found, a call to an external Matlab script is made to calculate the syndrome for the reference response.

Next the syndrome is sent to the FPGA and used by the PUF correct potential errors in the generated raw response using this syndrome. The error corrected response is then sent back to the computer. This response is received by the computer and hashed using the built-in SHA256 algorithm to obtain a 256-bit encryption key.

The encryption key is then input to the `demo_send.py` script, which uses it to encrypt the message 'Hello World!' and print out the encrypted result. This encrypted message is then sent (copied) back the to `demo_puf.py` script which uses the derived key to decrypt the message again.

In a real world application the two scripts would of course be replaced by two different devices and the key and syndrome would be stored in a more permanent way. The demo script can also be found in the GitHub repository.

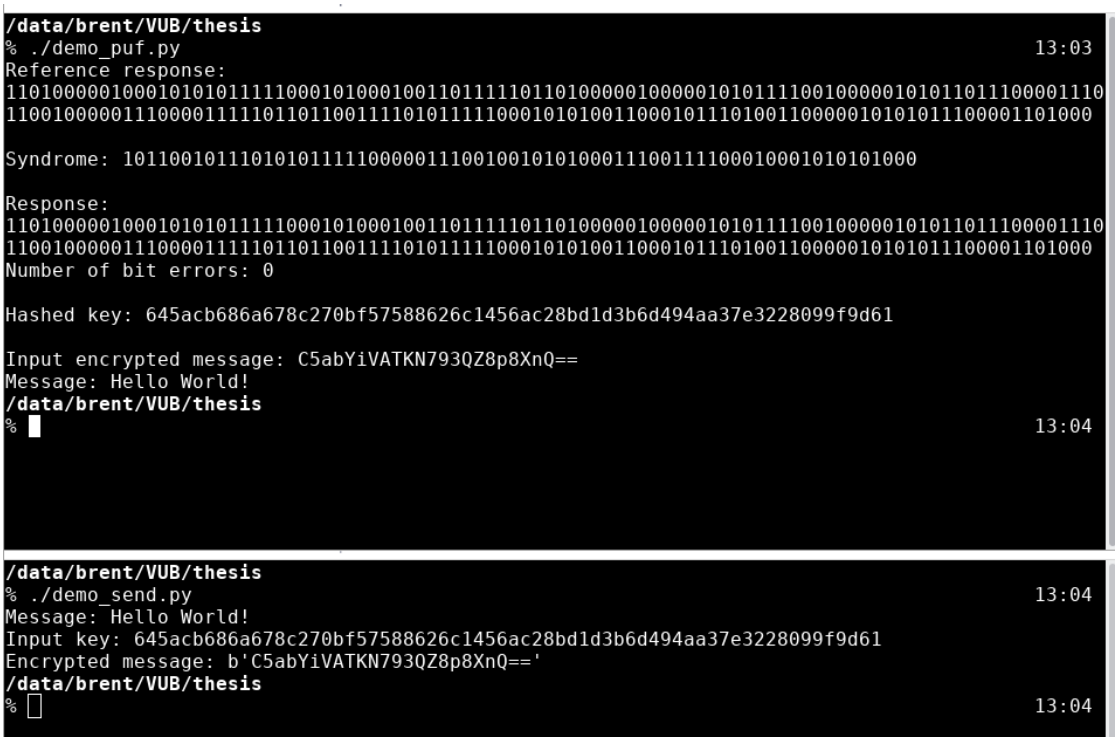

Figure 4.4: Example output of the demo scripts

Chapter 5

Conclusions

In this work, the concept of a physical unclonable function is explained, along with an overview of some possible applications. Three popular PUF types, being the ROPUF, APUF and Anderson PUF are described. Their characteristics and drawbacks are listed and compared to each other. Furthermore the properties used to analyze the performance of a PUF are listed.

One of the three PUF types was then chosen to be implemented from scratch, inspired by the design of [10]. The PUF was developed in VHDL, simulated and tested on an Artix 7 FPGA. We obtained a uniformity of 51.01%, a reliability and steadiness of 99.19% and 98.64%, 47.86% uniqueness and a bit-aliasing of 51.01%. These values are all comparable with the current state of the art, signifying a successful implementation.

Additionally, a BCH code was added to the design, to greatly increase the reliability of the PUF responses. For the BCH(255, 191, 9) code, the reliability is increased to 0.999991455%, or an error rate of 8.545×10^{-6}. More robust codes can be selected if needed, for which not a single error was observed in more than 100 million bits of measurements.

The area usage of both the basic and the ECC design is then compared to other papers, which showed that our basic PUF area is comparable with other compact designs, while the ECC design needs improvement before being able to compete

with the state of the art in terms of area consumption. Also the estimated power usage of both designs was tabulated. Unfortunately the needed power is underestimated due to incorrect simulation of the PUF's ring oscillators. Still, the total power consumed by the PUF would be negligible compared to the static power and the power consumption of the clock management.

5.1 Future work

A continuation of this work could consist of further analysis of the current designs, by testing them on multiple devices or studying the effect of environmental variations such as temperature or voltage on the PUF reliability. Another interesting angle would be looking at the security, either from an information-theoretical point of view or by trying to attack the PUF using machine learning or side-channel attacks.

Instead of using ring oscillators, one could adapt the Anderson PUF for Artix 7 FPGAs, implement the Arbiter PUF or another PUF type to replace the ring oscillators. It could be interesting to compare the different PUFs using the same setup, eliminating guesswork around the effects of differences in hardware or methodologies when comparing for example the ROPUF and APUF.

On the other hand, there are various possible improvements that could be added to our design. One main area of focus is the BCH decoder, as the decoder used here is quite outdated. A more optimized BCH decoder could certainly be developed. The BCH code could also be completely replaced with IBS coding, although the security implications should be carefully considered here. The design can also be extended by moving the provisioning phase onto the FPGA, such that the syndrome bits do not need to be calculated externally. Likewise the hashing of the final PUF response can be moved inside the FPGA, such that is it able to output ready to use cryptographic keys.

References

[1] K. Lofstrom, W. R. Daasch, and D. Taylor, "IC identification circuit using device mismatch," *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, no. February, pp. 372–373, 2000, ISSN: 01936530. DOI: 10.1109/ISSCC.2000.839821.

[2] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas, "Silicon physical random functions," *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 148–160, 2002, ISSN: 15437221. DOI: 10.1145/586131.586132.

[3] T. McGrath, I. E. Bagci, Z. M. Wang, U. Roedig, and R. J. Young, "A PUF taxonomy," *Applied Physics Reviews*, vol. 6, no. 1, 2019, ISSN: 19319401. DOI: 10.1063/1.5079407.

[4] U. Rührmair and D. E. Holcomb, "PUFs at a glance," *Proceedings -Design, Automation and Test in Europe, DATE*, 2014, ISSN: 15301591. DOI: 10.7873/DATE2014.360.

[5] A. Maiti, V. Gunreddy, and P. Schaumont, "A systematic method to evaluate and compare the performance of physical unclonable functions," *Embedded Systems Design with FPGAs*, vol. 9781461413, pp. 245–267, 2013. DOI: 10.1007/978-1-4614-1362-2_11.

[6] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," *Proceedings - Design Automation Conference*, pp. 9–14, 2007, ISSN: 0738100X. DOI: 10.1109/DAC.2007.375043.

[7] H. Kang, Y. Hori, and A. Satoh, "Performance evaluation of the first commercial PUF-embedded RFID," *1st IEEE Global Conference on Consumer Electronics 2012, GCCE 2012*, pp. 5–8, 2012. DOI: 10.1109/GCCE.2012.6379926.

[8] Xilinx Inc., "Zynq UltraScale+ Device Technical Reference Manual," 2020. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf.

[9] J. Zhang, Y. Lin, Y. Lyu, and G. Qu, "A PUF-FSM Binding Scheme for FPGA IP Protection and Pay-Per-Device Licensing," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 6, pp. 1137–1150, 2015, ISSN: 15566013. DOI: 10.1109/TIFS.2015.2400413.

[10] N. N. Anandakumar, M. S. Hashmi, and S. K. Sanadhya, "Compact Implementations of FPGA-based PUFs with Enhanced Performance," *Proceedings - 2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems, VLSID 2017*, pp. 161–166, 2017. DOI: 10.1109/VLSID.2017.7.

[11] J. H. Anderson, "A PUF design for secure FPGA-based embedded systems," *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, pp. 1–6, 2010. DOI: 10.1109/ASPDAC.2010.5419927.

[12] M. A. Usmani, S. Keshavarz, E. Matthews, L. Shannon, R. Tessier, and D. E. Holcomb, "Efficient PUF-based key generation in FPGAs using per-device configuration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 2, pp. 364–375, 2019, ISSN: 10638210. DOI: 10.1109/TVLSI.2018.2877438.

[13] P. Grabher, D. Page, and M. Wójcik, "On the (re) design of an FPGA-based PUF," *IACR Cryptology ePrint Archive*, pp. 1–4, 2013. [Online]. Available: http://eprint.iacr.org/2013/195.pdf.

[14] B. Gassend, D. Lim, D. Clarke, M. Van Dijk, and S. Devadas, "Identification and authentication of integrated circuits," *Concurrency Computation Practice and Experience*, vol. 16, no. 11, pp. 1077–1098, 2004, ISSN: 15320626. DOI: 10.1002/cpe.805.

[15] T. Machida, D. Yamamoto, M. Iwamoto, and K. Sakiyama, "A New Arbiter PUF for Enhancing Unpredictability on FPGA," *Scientific World Journal*, vol. 2015, 2015, ISSN: 1537744X. DOI: 10.1155/2015/864812.

[16] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. Technical Report 2003/235, Cryptology ePrint archive, http://eprint.iacr.org, 2006. Previous version appeared at EUROCRYPT 2004," pp. 79–100, 2004. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.148.5971&rank=10.

[17] R. Maes, A. Van Herrewege, and I. Verbauwhede, "PUFKY: A fully functional PUF-based cryptographic key generator," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7428 LNCS, pp. 302–319, 2012, ISSN: 03029743. DOI: 10.1007/978-3-642-33027-8_18.

[18] M. D. Yu and S. Devadas, "Secure and robust error correction for physical unclonable functions," *IEEE Design and Test of Computers*, vol. 27, no. 1, pp. 48–65, 2010, ISSN: 07407475. DOI: 10.1109/MDT.2010.25.

[19] G. T. Becker, A. Wild, and T. Guneysu, "Security analysis of index-based syndrome coding for PUF-based key generation," *Proceedings of the 2015 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2015*, pp. 20–25, 2015. DOI: 10.1109/HST.2015.7140230.

[20] Xilinx Inc., "AXI UART Lite v2.0 - PG142," 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_uartlite/v2_0/pg142-axi-uartlite.pdf.

[21] E. Jamro, "The Design of a VHDL Based Synthesis Tool for BCH Codecs," M.S. thesis, University of Huddersfield, 1997, pp. 1–132. [Online]. Available: http://home.agh.edu.pl/~jamro/bch_thesis/bch_thesis.html.

[22] Digilent, *Installing Vivado, Xilinx SDK, and Digilent Board Files*. [Online]. Available: https://reference.digilentinc.com/vivado/installing-vivado/start (visited on 10/28/2020).

[23] Xilinx Inc., "AXI Reference Guide, v4.0," *Xilinx Technical Documentation*, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

[24] Arm Limited, "AMBA AXI & ACE Protocol Specification," 2020. [Online]. Available: https://developer.arm.com/documentation/ihi0022/latest.

[25] A. Maiti, J. Casarona, L. McHale, and P. Schaumont, "A large scale characterization of RO-PUF," *Proceedings of the 2010 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2010*, pp. 94–99, 2010. DOI: 10.1109/HST.2010.5513108.

[26] Xilinx Inc., "Artix-7 FPGAs Data Sheet: DC and AC Switching Characteristics," 2021. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds181_Artix_7_Data_Sheet.pdf.

[27] ——, "Vivado Design Suite Properties Reference Guide-ug912," 2020. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

[28] ——, "Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide," *Xilinx Technical Documentation*, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug953-vivado-7series-libraries.pdf.

Appendix A

UART Lite IP Core usage

The exact operation of the IP core was understood after analysis of the AMBA AXI4 specification [24] and some simulation experiments with the core. A single method for a read and a write operation will be described here, which is applicable in these specific FPGA designs, but not necessarily the only correct one. The core's input and output signals will be in *italic*, with the prefix *s_axi_* common to all the AXI signals removed for brevity.

When a byte is sent over the UART connection, the core will store this byte in its internal RX buffer, which can then be read out using the AXI protocol. Inversely, for a write operation the byte will be stored in the internal TX buffer, which is continuously emptied by sending those bytes out over the UART. How exactly these RX and TX buffers can be read and written to will be discussed in the following sections. It is assumed all signals are set to zero at the start of operation.

A.1 Read operation

The AXI protocol works with addresses, so first the address of the RX buffer should be selected, by setting the 4-bit input signal *araddr* to 0b0000. To start a read operation, *arvalid* is set high, which means the value in *araddr* is valid and a read operation is requested. Simultaneously *rready* is set to one, to indicate we

are ready to accept results immediately.

The core then answers with *arready* set high, meaning that the read request is accepted. *arvalid* must be put to zero again now to avoid triggering a new read request.

When *rvalid* is set to one by the core, the data is available to be read from *rdata*. Only the 8 least significant bits in *rdata* will contain information, the other 24 bits are always zero. Since we put *rready* to one, the data will only be available for a single clock cycle. *rready* can be left high if desired. While *rvalid* is high, the 2-bit output *rresp* will signal if the read was successful or not. A value of 0b00 means a successful read, a value of 0b10 indicates that the RX buffer was empty.

A.2 Write operation

To start the write operation, five signals need to be set. The address of the TX buffer is selected by setting *awaddr* to 0b0100 and marking this address as valid by setting *awvalid* high. In the same clock cycle also the information byte can be put in the 8 least significant bits of *wdata* and signaling this valid data with a high *wvalid* signal. Lastly setting *bready* to one means the block is ready to accept the write response.

The core will accept the write request with putting *wready* to high. Now *awvalid* and *wvalid* must be set low again.

The write response will follow shortly after with a high *bvalid*. If *bresp* now has the value 0b10, the TX buffer was still full and the byte could not be accepted. Success is indicated by the value 0b00. Analogous to a read operation, *bready* may be left high if wanted.

For further details the VHDL file `uart.vhd` in the GitHub repository can be consulted.

Appendix B

RO placement constraints

The VHDL architecture for our ring oscillator block is given below. All the attributes used can be found in the Vivado Properties Reference Guide [27, pp. 154, 203, 342]. The first objective is to make sure the optimizations do not merge or otherwise alter the four LUTs of the design, u0 up to u3. This is done by setting the DONT_TOUCH attribute, which is quite self-explanatory, to true for all four LUTs.

Secondly, the ring oscillator should be put into a single slice, with a fixed ordering of the LUTs within. The relative location (RLOC) constraint works at the slice level to position elements, such that by specifying the same relative location for each LUT, they will all end up in the same slice. In order to fix the internal ordering of the LUTs, there is the BEL (Basic ELement) constraint, which is used to assign VHDL objects to specific elements (LUTs, FFs, ..) within a slice. The "6LUT" of the BEL attribute values refers to a 6-input LUT, while the "A", "B", "C", "D" are the 4 different LUTs in a slice. The combination of the RLOC and BEL constraints forces a fixed single-slice layout for the ring oscillators.

As a last remark, the explicit LUT instantiations are done using the Xilinx UNISIM library by mapping the inputs and output and providing a truth table. More information on these instantiations can be found in the Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide [28, pp. 421–425].

```
architecture Behavioral of ro is
        signal chain : std_logic_vector(3 downto 0);
        attribute DONT_TOUCH : boolean;
        attribute RLOC : string;
        attribute BEL : string;
        attribute DONT_TOUCH of u0 : label is true;
        attribute DONT_TOUCH of u1 : label is true;
        attribute DONT_TOUCH of u2 : label is true;
        attribute DONT_TOUCH of u3 : label is true;
        attribute RLOC of u0 : label is "X0Y0";
        attribute RLOC of u1 : label is "X0Y0";
        attribute RLOC of u2 : label is "X0Y0";
        attribute RLOC of u3 : label is "X0Y0";
        attribute BEL of u0 : label is "A6LUT";
        attribute BEL of u1 : label is "B6LUT";
        attribute BEL of u2 : label is "C6LUT";
        attribute BEL of u3 : label is "D6LUT";
begin
        u0 : LUT2 generic map (INIT => "1000") -- and port for enable
        port map (O => chain(0), I0 => chain(3), I1 => enable);
        u1 : LUT1 generic map (INIT => "01") -- inverter
        port map (O => chain(1), I0 => chain(0));
        u2 : LUT1 generic map (INIT => "01") -- inverter
        port map (O => chain(2), I0 => chain(1));
        u3 : LUT1 generic map (INIT => "01") -- inverter
        port map (O => chain(3), I0 => chain(2));
        output <= chain(3);
end Behavioral;
```

Finally the absolute location of each RO is fixed such that they are all put together in a rectangular space on the FPGA. This is done with the absolute location (LOC) constraint. Since it is not convenient to manually create a constraint line for each of the 64 ROs, a Python script was written to automate this task. This script is

also available in the GitHub repository. As an example the output for 4 ROs is shown below.

A last problem is that Vivado does not allow combinatorial loops in the design by default. This will cause an error message when trying to generate a bitstream. Luckily the solution is to add the property ALLOW_COMBINATORIAL_LOOPS. The wildcard '*' makes it easy to apply this property to all oscillators in the arrays in two lines, which are put under the following location constraints.

```
set_property LOC SLICE_X10Y20 \
        [get_cells {puf_i/ro_array_0/U0/gen_ro[0].r/u0}]
set_property LOC SLICE_X10Y22 \
        [get_cells {puf_i/ro_array_1/U0/gen_ro[0].r/u0}]
set_property LOC SLICE_X10Y21 \
        [get_cells {puf_i/ro_array_0/U0/gen_ro[1].r/u0}]
set_property LOC SLICE_X10Y23 \
        [get_cells {puf_i/ro_array_1/U0/gen_ro[1].r/u0}]
set_property ALLOW_COMBINATORIAL_LOOPS true \
        [get_nets {puf_i/ro_array_0/U0/gen_ro[*].r/O}]
set_property ALLOW_COMBINATORIAL_LOOPS true \
        [get_nets {puf_i/ro_array_1/U0/gen_ro[*].r/O}]
```