FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit in Informatik

Side-Channel Analysis of Physical Unclonable Functions (PUFs)

Dieter Schuster

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit in Informatik

Side-Channel Analysis of Physical Unclonable Functions (PUFs)

Seitenkanalanalyse von Physical Unclonable Functions (PUFs)

Author:	Dieter Schuster
Supervisor:	Prof. Dr. Claudia Eckert
Advisors:	Dipl. Ing. (FH) Dominik Merli Dipl. Inf. Christian Schneider
Date:	December 9, 2010

I assure the single handed composition of this diploma thesis only supported by declared resources.

December 9, 2010 Dieter Schuster

Abstract

Physical Unclonable Functions (PUFs) are an emerging technology, which makes it possible to base secure identification, authentication or generation of keys on unclonable hardware tokens. Side-channel attacks, on the other hand, have been successfully used to break various cryptographic devices. However, publications on the topic are dealing almost exclusively with the attacks on cryptographic algorithms, e. g. DES and AES. Although, PUF attacks based on machine learning have been published, there are no surveys regarding side-channels, yet.

This thesis is meant to fill this gap and provide an analysis and demonstration of side-channel attacks against PUFs. The theoretic possibility of such attacks is discussed and rated with respect to feasibility. Furthermore, the electromagnetic emissions of an RO-PUF prototype were measured and analysed. Also, a proof-of-concept implementation is presented, which enables the extraction of a 64 bit secret key of the RO-PUF prototype, based on the gathered measurement data. Additionally, the measurements are used to evaluate the viability of the theoretical attack scenarios. Finally, countermeasures helping to prevent such side-channel attacks against PUFs are suggested.

Zusammenfassung

Physical Unclonable Functions (PUFs) bieten, als neuartige Technologie, sichere Identifikation, Authentifizierung und Erzeugung von Schlüsseln auf einem nicht nachbildbarem physikalischem Medium. Andererseits wurden Seitenkanalattacken erfolgreich genutzt, um verschiedenartige kryptographische Geräte anzugreifen. Jedoch konzentrierten sich Veröffentlichungen zu diesem Thema fast ausschließlich auf Verschlüsselungsalgorithmen, wie z. B. DES und AES. Obwohl es veröffentlichte Angriffe auf PUFs gibt, welche sich Methoden des maschinellen Lernens zunutze machen, sind keine Studien im Bezug auf Seitenkanäle bekannt.

Ziel dieser Arbeit ist es, durch eine Analyse und Durchführung von Seitenkanalattacken auf PUFs, diese Lücke zu füllen. Dazu wurden die theoretischen Möglichkeiten dieser Attacken erörtert und auf ihre Durchführbarkeit hin beurteilt. Weiterhin wurden die elektromagnetischen Abstrahlungen eines RO-PUF Prototyps gemessen und analysiert. Darauf basierend wurde eine Attacke implementiert, die es ermöglicht einen 64 Bit langen geheimen Schlüssel aus den zu Grunde liegenden Messungen zu gewinnen. Zusätzlich wurde die Machbarkeit der zuvor vorgeschlagenen theoretischen Attacken, anhand der vorliegenden Messdaten, evaluiert. Abgeschlossen wird die Arbeit mit Vorschlägen für Gegenmaßnahmen, die helfen sollen, Seitenkanalattacken gegen PUFs zu verhindern.

Contents

List of Figures

List of Tables

1. Introduction

In today's world a lot of security mechanisms rely on hardware tokens, e. g. smartcards, as a secure way of authentication or key storage. However, it is possible to clone those tokens by extracting secrets, for example, with invasive attacks. This is possible, because secrets, like a cryptographic key, have to be stored in some kind of memory, which can be read by an attacker. Therefore, new ways of secure key storage have to be investigated.

Physical Unclonable Functions (PUFs) are a relatively new approach of achieving secure identification, authentication or generation of cryptographic keys. Given a challenge as input, a PUF yields a response which is unique to the instance of the PUF. Every entity has a different behaviour, which is determined during production and not manipulatable by the manufacturer. As the name suggests, they are physically unclonable, which is contrary to traditional practices implementing secure operations in software or easily replicable hardware. Furthermore, PUFs are very hard to characterise, which impedes simulating their behaviour in software.

Using this kind of technology makes it possible to create unclonable hardware tokens, which have a unique identity bound to the hardware and can be securely authenticated in the field. Since PUFs can be used to generate cryptographic keys, they can also be used instead of complex and expensive secure storage systems for storage of private keys. Instead of having to keep a private key stored in external secure memory, it can be generated by a PUF on demand and discarded after use. Hence the key only exists during the cryptographic operations, which restricts the attack surface for an adversary immensely.

Recent research deals not only with the improvement of currently known PUFs [13] or the discovery of new types, but also with proofs of the assumed security properties. Among the latter were attacks based on machine learning, in order to characterise the behaviour of a PUF [16] and information theoretical proofs that there's no significant leakage of secret bits in data used for the operation of the PUF [23].

This thesis gives an analysis of side-channels, like power consumption or electromagnetic emissions during operation of the PUF, which an attacker could use to gain knowledge of concealed data of the PUF. For our analysis we used a Ring-Oscillator PUF (RO-PUF) [20] implemented on a Field Programmable Gate Array (FPGA) . However the attacks may also be generalised for other types of PUFs, because they target properties, which are present in most implementations of PUFs.

In Chapter 2 some background information on PUFs and the different types thereof as well as a short overview on side-channels is given. Further, related work concerning current attacks and analysis of weaknesses of PUFs are presented in Chapter 3. The afore mentioned PUF prototype that was used for this analysis is described in depth in Chapter 4. In Chapter 5, the theoretical attacks are proposed. The implementation of one those attacks is detailed in Chapter 6. Chapter 7 evaluates every of the previously suggested theoretical attacks regarding feasibility, based on measurements conducted with the prototype. Next to that, the effectiveness of the actual implemented attack is analysed. Chapter 8 concludes the thesis.

2. Background

This chapter introduces Physical Unclonable Functions (PUFs), their functional principles and applications. Furthermore side-channels and attacks related to them will be described.

2.1. Physical Unclonable Functions

A Physical Unclonable Function is defined as physical structure which delivers a response when challenged, as displayed in Figure 2.1. These responses can be easily generated by challenging the structure. Because PUFs are hard to characterise, it is highly improbable to repeat the behaviour of a PUF for any given challenge. Furthermore, the characteristics of the PUF, which are different for each instance built, cannot be controlled even by the manufacturer. This fact implicates that PUFs are physically unclonable.

2.1.1. Types of PUFs

There are different physical properties, which can be exploited to implement a PUF.

Optical PUFs

Doping a transparent material with light scattering particles is one example of a manufacturing process which cannot be fully controlled. The particles may move in unpredictable ways while the surrounding material is curing. When directing a laser beam at the material, which was suggested in [14], the randomly positioned particles will scatter the light resulting in a unique speckle pattern. Different challenges would be represented by aiming the laser beam at the material in different angles.

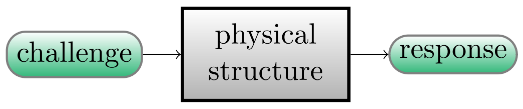

Figure 2.1.: Basic principle of a PUF

Coating PUFs

The basic principle of randomly placed particles in a curing material can also be applied to electrical based PUFs. In [21], Tuyls et al. presented this approach, known as the Coating PUF. A Coating PUF is built as the top layer of an Integrated Circuit (IC) by applying circuit paths, laid out in a comb shape, on top of the given IC. These will be encased by a material, which is randomly doped with dielectric particles of different size and dielectric strength. Each pair of circuit paths now forms a capacitor with random capacitance, which again is highly improbable to be controllable by the manufacturer. Furthermore, placing the PUF on the top of the initial IC yields protection against invasive attacks for the underlying circuits. Hence if an attacker tries to disassemble or remove the cover of the device in any way, the PUF will lose its original response behaviour and therefore its secret.

Silicon PUFs

In contrary to the above mentioned PUFs, which utilise randomness explicitly introduced during the manufacturing process, Silicon PUFs take advantage of intrinsic randomness.

One example of this intrinsic randomness is the behaviour of the bistable latching circuitry in Static Random Access Memory (SRAM) as mentioned in [5]. Each of these SRAM cells starts in the same state on power-up with high probability, yet different SRAM cells behave randomly and independently from each other due to variations in the manufacturing process.

Another example of intrinsic randomness are the geometric properties of conducting paths on ICs. Independent of the manufacturing technology of semiconductors, there are always inevitable variations in the width of conducting paths. This results in differences in signal propagation delays, which again are random and not manipulatable by the manufacturer. One can take advantage of these differences by e. g. implementing two symmetrical digital delay lines, which get triggered simultaneously. Because of the above mentioned variations the outcome, i. e. at which end the transition occurs first, is different between multiple ICs. This is the working principle of the so called Arbiter PUF [12], where a digital arbiter indicates which of both signals was the fastest and therefore produces a one-bit result.

Another example for a silicon PUF is the Ring-Oscillator PUF (RO-PUF) [20]. Although exploiting the same variations as the Arbiter PUF, the working principle is different. As the name suggests, a lot of identical ring oscillators are implemented on an IC and get excited by a signal at the same time. Due to said variations during production, the ring oscillators resonate in different frequencies, although they are built identical. A pair of oscillators can be compared by counting falling and rising edges for each and comparing the result. Further details on RO-PUFs are given in Chapter 4, where the RO-PUF prototype used for this thesis is described.

2.1.2. Responses of PUFs

All of the above mentioned types of PUFs share the property of having a noisy response. This is due to environmental influences affecting the physical system in question. A higher temperature will result in extension of the material and hence in a different behaviour of the PUF. Another influence on the characteristics of a PUF may be ageing processes of the involved materials resulting in diversifying responses during longer periods of time. For the PUFs based on electrical circuits, variations in supply voltage also induce major noise in responses.

Being sensitive to all those inevitable environmental factors causes the responses of PUFs to be useless for applications that cannot cope with a certain level of errors. Therefore, it is necessary for all types of PUFs to have some sort of error correction, to assure a constant response for a given challenge.

Another property of responses of PUFs is not being uniformly distributed over the possible space of responses. This is usually alleviated by the use of universal hashing after the error correction step to redistribute the responses.

2.1.3. Applications of PUFs

There are two basic modes in which PUFs are being used. Either the PUF at hand is able to generate a very large quantity of challenge-response pairs or it only yields few usable challenge-response pairs. This characteristic determines the field of application.

It is possible to built an authentication scheme, described in [14], based on a great number of challenge-response pairs, which the manufacturer generates and saves to a secure database. When the device is out in the field and needs to authenticate itself a challenge gets chosen out of the database and is presented to the PUF. The response yielded by the PUF is compared to the one stored in the database, to check whether it was the original PUF. Each challenge-response pair is only used once to prevent man-in-the-middle attacks. Figure 2.2 outlines this procedure.

If the goal is just a simple identification, it suffices being able to generate just few different challenge-response pairs. In this scenario the challenge will mostly stay the same and the response is interpreted as an ID. Another setting, where only a single challenge is used, is the generation of a cryptographic private key. Without the usage of a PUF, a private key in an embedded device would be stored in a secure memory. This is a very expensive approach, because the memory has to be tamper-proof. An example of a device implementing such a secure tamper-proof memory system is the cryptographic coprocessor card IBM 4764 [7]. Besides hardware for cryptographic operations and a random number generator, this device features protective shields, sensors and control circuitry to protect against physical attacks. The control circuitry must be powered at all times, using batteries if the device is not in a powered-on machine, and humidity, temperature and barometric pressure ranges must be strictly maintained according to the devices specification.

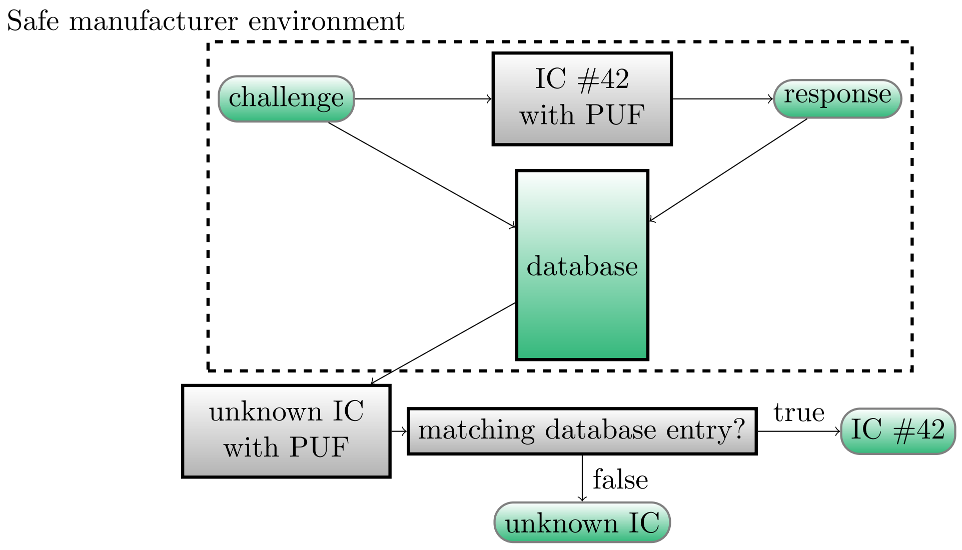

Figure 2.2.: Authentication using Physical Unclonable Functions: The manufacturer creates a database of CRPs in a safe environment. A IC with PUF in the field is authenticated by being challenged with a challenge from the database. If the response matches the corresponding entry in the database, the IC passes the authentication.

It is obvious, that the usage of such a device is not just expensive in acquisition, but also complex during operation. Additionally, all the hardware needed to protect a device like this cannot be fit into a small, embedded device. However, it is possible not to store the private key, but to generate it on demand using a PUF. Because a PUF is hard to characterise, the secret is implicitly stored in a secure way. Furthermore, it is protected against invasive attacks, because the PUF would get faulty or even destroyed. This makes the system inherently tamper-proof.

Another field of application for PUFs is using them in protection of brand and Intellectual Property (IP) . Since it became common for the semiconductor industry to have shared foundries or outsourcing the production, counterfeiting of devices and IP is an increasing problem. As already mentioned it is possible to derive cryptographic key material from PUFs, which in turn can be used to encrypt and decrypt IP and thereby bind it to a certain FPGA chip. Such an approach using bitstream encryption was suggested in by Kean [8]. Apart from that, there where also solutions suggested, where it is the aim to authenticate the hardware platform on which third party IP runs, using PUFs [19]. An improvement of this approach can be found in [5], where secret-key material gathered from the PUF is used both for encryption and MAC-based authentication.

Closely related to IP protection is the topic of remote feature activation, as mentioned in [6]. This means that a certain feature of the device may be unlocked after the device is already in possession of, an often untrusted, third party. As mentioned above, the responses of a PUF, from which keys may be derived, is noisy and some kind of error correction mechanism is needed. Those mechanism also require certain helper data, in order to restore the correct response from a noisy PUF response. The way that keys get derived from PUFs implies that both key and helper data are unique for every PUF. If such helper data is given to a customer to unlock a certain feature of a hardware with built-in PUF, the customer is only able to unlock it on the given device. An explanation of the way the key and helper data can be generated is given in Chapter 4.

2.2. Side-Channel Attacks

Side-channel attacks are a form of physical attacks on devices, which use information gained from the device while it is operating. Every implementation causes additional effects while in operation, e. g. power consumption or electromagnetic radiation. Putting information gained by these side-channels in correlation with the supposed activity, makes it possible to gain information about internally used information. When attacking cryptographic devices the common goal is the acquisition of a secret key.

2.2.1. Timing Attack

Computational systems usually need a different amount of time to perform an operation for different inputs. This is due to many reasons like branching optimisations, cache behaviour or simply processor instructions that take different amount of cycles. In [10], Kocher showed that it is possible to use these timing variations to find fixed Diffie-Hellman exponents or factor RSA keys.

A simple example is a function, which checks a passphrase character wise from beginning to end and reporting an error as soon as the first mismatch occurs. This kind of implementation makes it possible for the attacker to gradually approximate her input to the correct passphrase. By measuring the time it takes the function to report the error, the attacker is able to conclude how many characters were guessed correctly at the beginning of the current input.

2.2.2. Power Attack

Another possibility that enables inference of operations executed within a device is the measurement and analysis of the device's power consumption. Just like some operations taking more time than others, they have a different energy usage pattern as well. This is due to the power consumption characteristics of Complementary Metal Oxide Semiconductor (CMOS) technology, which is prevalent in today's digital circuits. Each time a CMOS gate changes its state, there is measurable power consumption at the V_{dd} pin. In the case, where gates are clocked and working synchronously, they all change at the same time and their power consumption adds up. Approximately 85% of

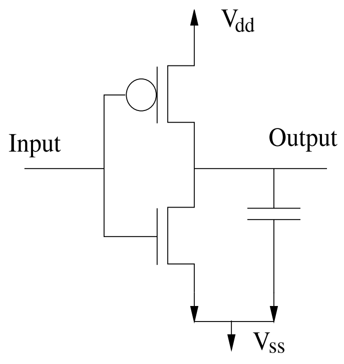

Figure 2.3.: Circuit diagram of an inverter

power consumption is caused by the dynamic charge or discharge. As can be seen using an inverter as an example (Figure 2.3), at the output of each gate there is a parasitic capacitance, caused by the connected wires and following gates. During a change in the output, this capacitance is either charged or discharged, leading to a current, which flows to V_{dd} or V_{ss}. Further information regarding this topic can be found in [22].

Simple Power Analysis

The basic form of power analysis attacks is the Simple Power Analysis (SPA). After measuring the supply current during the operation of the attacked devices, the power traces are inspected visually. Hardly any conditioning takes place beforehand, except frequency filters or averaging functions in order to eliminate noise.

Using this approach, it is for example possible to see the 16 rounds of DES in the power traces of a device using this encryption algorithm or break RSA implementations by observing differences in multiplication and squaring operations [9]. Usually SPA is used for a first analysis of the device's operation and to find regions of interest. Furthermore conditional jumps can be visible because of different power consumption in the different branches of execution. Hence SPA can be used to break cryptographic implementations in which branching depends on the data being processed.

Differential Power Analysis

Going one step further from there and using statistical methods for the analysis of the gathered data, leads to Differential Power Analysis (DPA) [9]. This type of attack makes it feasible to exploit smaller variations in the measurements, which are potentially overshadowed by measurement errors. An often used approach in DPA is to record power measurements while letting the attacked device operate on different inputs. The attacker calculates hypothetical intermediate results for those inputs and possible values for the secret, he tries to find out. After that, based on a power consumption model of the device, he maps those results to hypothetical power consumptions.

Both the actual measurements and the hypothetical results are then correlated. The hypothetical calculation with the highest correlation to the real measurements is the one using the correct secret value.

While data collection in SPA is limited to the recording of one or a few power traces, it is necessary for DPA to record a lot of traces with different inputs.

2.2.3. Electro-Magnetic (EM) Attack

Whenever there is a current, there is also an induced electro-magnetic field with a field intensity that is directly proportional to the current. Hence the methods for SPA and DPA can also be used on data, which was gathered by an EM sensor.

However the use of an EM probe yields some advantages over the plain measurement of power consumption, as demonstrated in [15]. First of all, it is less invasive than power measurement, where a measuring resistor has to be placed in the circuitry. Furthermore the data gathered contains more information. This is due to the EM field having a positive and negative direction depending on the direction of the current, whereas the data from power measurements are only absolute values. Another advantage of using an EM probe is the potential of regional measurements, making it possible to analyse functional groups apart from each other, if they are placed on different regions in the device.

3. Related Work

Apart from fundamental research on PUFs, like finding new types or improving existing ones, there have been several considerations regarding the security related properties of PUFs. This chapter will provide an overview of the security concerning work on this topic.

3.1. Duplication of a PUF

One possible attack scenario for PUFs is the attempt to duplicate a PUF by building an IC identical to the PUF. Although the manufacturer is not able to produce an identical copy of a given PUF at will, because of the already described statistical variations in the manufacturing process, a clone could be obtained by chance. This attack could be already seen as successful, if the counterfeit PUF yields the correct responses for most of the challenges and doesn't even have to be fully identical.

However the inevitable variations are big enough that each PUF can be identified, even when produced in the same batch or even on the same wafer. This was for example shown in [12] where inter-chip variation was measured on a single wafer and across wafers. Hence an attacker would have to produce a huge amount of PUFs and conduct thorough measurements to obtain two similar PUFs. The complexity for this undertaking, both economical and computational, is very high and can be considered as unfeasible.

3.2. Modelling of PUFs

As explained above, it is very hard to directly clone a PUF. Another approach would be to model the behaviour of a given PUF and simulate it, in order to obtain the correct answer to a given challenge.

3.2.1. Building a Delay Model

By measuring the delays of all the circuit component, it is possible to create a time-accurate model of a PUF. This model can then be used to simulate the behaviour of the original PUF to predict the responses of given challenges.

Exact measurement, with the needed accuracy, is a difficult task, though. An attacker can choose between invasive measurement, where the package of the IC has to be removed to insert probes, or non-invasive measurements, like DPA or EM analysis. Invasive procedures might damage the PUF and therefore yield wrong results, because the behaviour of the PUF will change. Moreover, the probes themselves might interfere with the exact measurements of timings, because of capacitive coupling between the probing equipment and the device to be measured.

Non-invasive techniques have disadvantage of not being able to measure small delay variations as precisely as invasive methods and therefore not being a valid alternative for gathering the required data to time-accurate modelling the PUF.

3.2.2. Building a Challenge-Response Model

Considering the drawbacks of exact measurements to be able to simulate the PUF and calculate an answer for a given challenge, it is a promising approach to characterise the PUF by characterising the responses. Exhaustively mapping all Challenge Response Pairs (CRPs) is not a viable option for most types of PUFs, since there is a huge number of them and measuring all of them is impossible in reasonable time. This can be seen in the classification of [17], where only for the RO-PUF a full read-out is given as practically feasible.

A much more promising approach is using machine learning techniques, to learn the corresponding responses to challenges by only using a small fraction of all possible CRPs. Lim showed in [12] how the delays in Arbiter PUFs can be modelled by a Support Vector Machine (SVM) with examples of CRPs. The prediction error rate of the SVM was below the inter-chip variation, which means that the software model could not be distinguished from the original PUF. Because of this variants of the Arbiter PUF were developed in order to be more resilient against machine learning model building attacks. However, even XOR Arbiter, Lightweight and Feed Forward Arbiter PUFs were successfully attacked with techniques like logistic regression or evolution strategies in [16]. Recently model building attacks were improved further in [18], where it was shown that policy gradients are an even more effective machine learning approach.

3.3. Protocol Attacks

Besides the PUF itself, the protocols using the PUF are susceptible to attacks. If an attacker is able to break such a protocol, she may also gain knowledge about the behaviour of the PUF or even secret keys the PUF should conceal.

As mentioned in [4], the most significant variant in this context is the Man-In-The-Middle (MITM) attack (MITM), in which the attacker intercepts e. g. the communication between a user and the PUF. Being able to do so leverage the adversary to impersonate the PUF without other parties noticing it. The impact of this depends on the usage scenario. If the goal is to prove that one has possession of the PUF or access to it, it is just part of the protocol and not a real attack.

3.4. Information Leakage in Helper Data

Since the operation of the PUF depends on helper data, which is stored in insecure memory and is in a certain amount dependent on the secret data, there might be an information leakage in helper data.

An often suggested construction for error-correction for PUFs involves the Bose-Chaudhuri-Hochquenghen (BCH) code. An example would be BCH(255,63,30), where 255 bits are generated by the PUF and 63 bits are the actual key size. The other 192 bits are exposed, from an information theoretical view, as the helper data. In this example, 30 of the 255 bits can be corrected at most. Because of environmental variations some PUFs show very high error rates. In an attempt to keep the error correction simple, it is preferable to reduce the error beforehand. Two examples of error reduction are the use of repetition coding and a preselection of the bits, that are less likely to be noisy. Both approaches again need helper data bits, which are stored publicly and might leak information about the PUF response.

The first attempt in finding an error correcting code, which is information theoretically secure, i. e. whose helper data doesn't leak any information about the response, was mentioned in [23] with the suggestion of Index-Based Syndrome (IBS) coding.

4. The RO-PUF Prototype

The used prototype was implemented on a Xilinx Spartan 3E FPGA and is based on the work in [2] and [13]. As suggested by [2], it uses code-offset construction [3] for the helper data algorithm with a construction of BCH codes for error correction and hashing based on a Toeplitz matrix [11].

In this chapter, a description of the basic functional principle of the RO-PUF is given. Afterwards, the methods for the needed error correction step and hashing will be explained.

4.1. Functional Principle

A RO-PUF can be categorized as a Silicon PUF. As mentioned above, Silicon PUFs utilise intrinsic randomness of ICs, which is caused by variation in the manufacturing process. In this case signal propagation delays are exploited to yield a response based on the physical properties of the IC. As the name Ring-Oscillator PUF suggests, there are a number of identical ring-oscillators layed out across the chip. Although being identical, there are inevitable differences in their frequencies, because of said physical variations.

In order to generate one bit for the response, two oscillators are chosen and run for a specified duration. The positive edges are counted and after the time has elapsed, both counters are compared. Depending on which counter has the higher value, the result is either 0 or 1. Figure 4.1 illustrates this approach. The prototype uses 511 ROs to yield 510 response bits.

4.2. Helper Data Algorithm

Because of environmental influences, like temperature, it cannot be guaranteed, that a given oscillator pair always behaves the same in comparison. Therefore the application using it either may be error tolerant, or it is necessary to correct the occurring errors. In a scenario for generation of private key material, every faulty bit results in a different key, which is unacceptable. In addition to the noisy nature of PUFs, their responses are not uniformly distributed. Hence, error correction alone is not sufficient to form a cryptographically secure key, nor to form the seed for the generation of a key.

To deal with the above mentioned issues, a helper data algorithm implements an information reconciliation phase followed by a privacy amplification phase. Both need

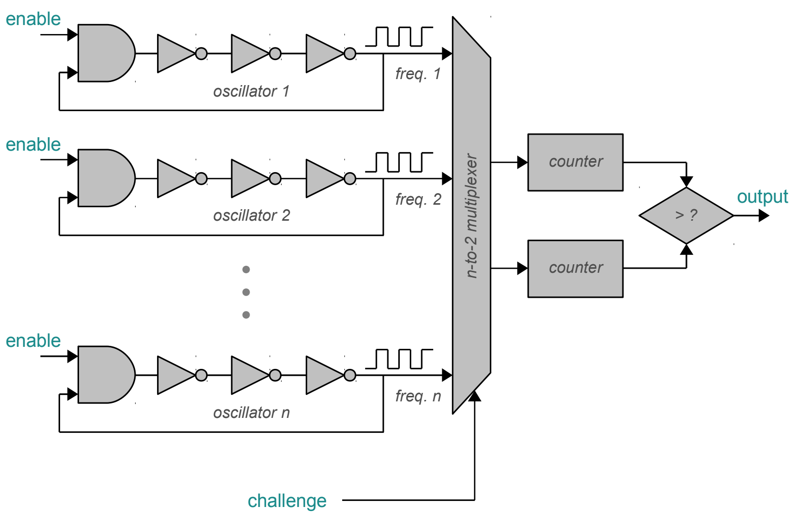

Figure 4.1.: Generating a response in an RO-PUF

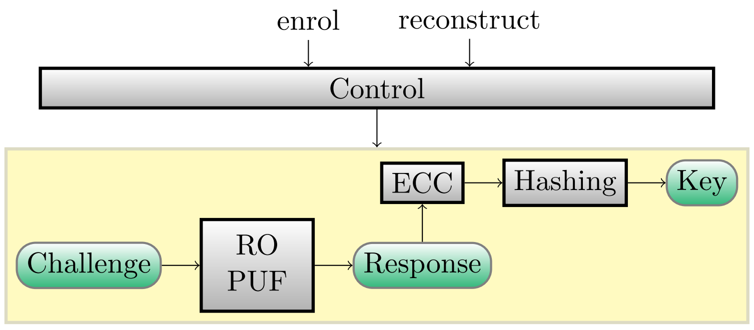

Figure 4.2.: Functional schematic of the prototype

helper data to work, which is previously generated once, in a trusted environment, during enrolment. The data obtained in this phase can then be stored in non-secure memory, where it is accessible for the reconstruction of the key.

The whole helper data algorithm is capsuled from the application interface, which can only issue the instructions *enrol* and *reconstruct* (Figure 4.2). Usually *enrol* is only used by the manufacturer during enrolment and deactivated on deployment.

Being based on offset-code construction [3] the implementation of the helper data algorithm for the prototype requires an error correcting code C, with parameters $[n, k, d]$, as well as a set H of universal hash functions. The parameter n describes the length of the code C, whereas k denotes the length of the messages it encodes, both in bits. The minimum distance of the code words is given by d, thus the code can correct up to $t = \lfloor \frac{d-1}{2} \rfloor$ bits.

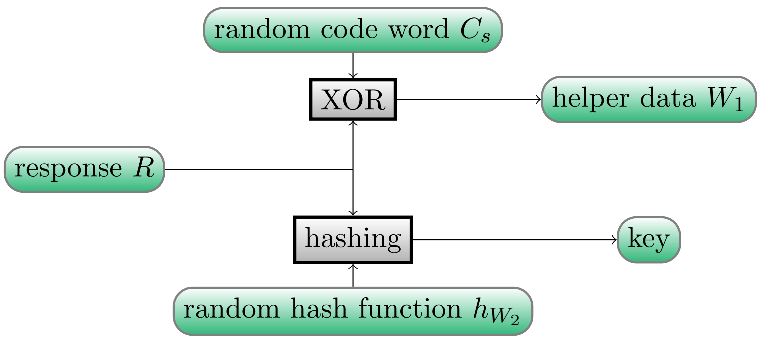

Figure 4.3.: Enrolment

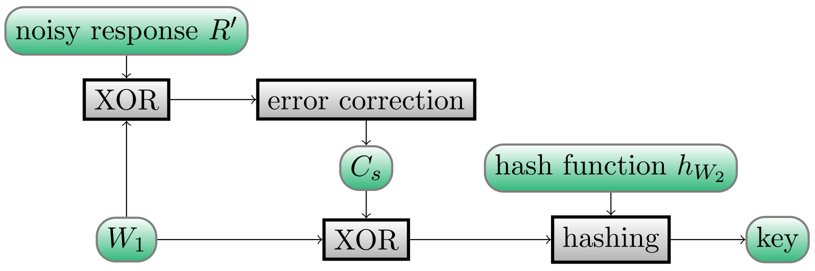

Figure 4.4.: Reconstruction

When issuing *enrol* a pair of helper data $W = (W_1, W_2)$ and the key K are generated. To achieve this, first, a code word C_s out of C is chosen at random. Based on C_s the first part of the helper data is calculated by $W_1 = C_s \oplus R$. Also, a hash function h_i gets randomly chosen out of the set H. The key, resulting from *enrol*, is defined as $K = h_i(R)$, with the index i forming the second part of the helper data W_2. An overview of the enrolment is given in Figure 4.3.

During key reconstruction a noisy response R' is obtained from the PUF. Using the helper data W the key K, as generated by *enrol*, gets restored. As the first step the noise induced errors have to be corrected. Using W_1, a noisy code word can be calculated with $C'_s = W_1 \oplus R'$, from which C_s can be obtained by using the decoding algorithm of C. Analogous to the creation of W_1 during *enrol*, the original R results from $R = W_1 \oplus C_s$. Applying the hash function h_i, with the index resulting from W_2, yields the key $K = h_{W_2}(R)$ again. Figure 4.4 shows a summary of the key reconstruction.

4.2.1. Error Correction

As mentioned above, an error correcting code C is needed. Among the suggestions for different error correcting codes on FGPA hardware given in [2], a BCH code design was chosen for the prototype. This decision was based on the strong error correction

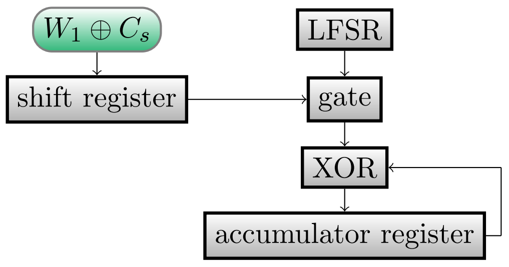

Figure 4.5.: LFSR-based hashing with Toeplitz matrix

properties of the BCH codes, which requires less input bits are required and hence less ROs need to be implemented.

The actual code used is BCH(255,37,45). This means that it processes 255 input bits, generated by the PUF, and yields 37 error free bits, if there are at most 45 errors. Therefore, the error correcting code is needed to be applied twice, in order to correct the errors in the 510 bit long input bit string.

4.2.2. Hashing

The code-offset construction for the helper data algorithm requires a function implementing universal hashing. For the prototype, this is provided by hashing based on a Toeplitz matrix as described in [11]. This so called Toeplitz hashing is an often used algorithm, when hashing has to be implemented in hardware, because it has the advantage of being simple to implement and hence not using much space on an FPGA. Figure 4.5 shows the principle of operation of the hashing as used in the prototype. The Toeplitz matrix is not directly stored in memory, which would waste a lot of space on the FPGA, but instead is simulated by a Linear Feedback Shift Register (LFSR) .

During operation, the result of the error correction phase $(W_1 \oplus C_s)$ is shifted to a gate, one bit per cycle. Every time this bit is 1, the current value of the LFSR gets XORed onto the value of an accumulator register. The LFSR gets shifted and the procedure is repeated for the remaining input bits. The initial value of the LFSR defines the Toeplitz matrix and corresponds to the second part of the helper data W_2.

The hashing operation on the prototype processes the first 74 bits, as returned from the error correction phase, and yields a uniformly distributed, 64 bit long hashed bit string.

5. Possible Side-Channel Attacks

In the following section several possible approaches to side-channel attacks against the RO-PUF prototype are discussed. First, possibilities for attacks, which are specific to RO-PUFs are mentioned. Then, alternatives which also may work for other types of PUFs are explained. It is hereby assumed, that the design and used algorithms of the PUFs are known beforehand. The focus is set on EM-based attacks, because of the mentioned advantages of this method in Chapter 2.2.3.

5.1. Ring Oscillators

Foremost, the core part of the PUF prototype, which consists of oscillators, counters and comparators, is investigated.

5.1.1. Oscillators

All implemented oscillators get excited by a stimulus and start to oscillate with their respective frequencies. This oscillation itself causes EM emissions. Being able to detect the frequency of every single oscillator results in a full model of the PUF. Given a challenge, i. e. a list of which pairs of oscillators are compared in which order, it is trivial to simulate the comparison and the following phases of the key generation, given the obtained model.

Apart from EM emissions generated by the oscillation, an oscillator would show a difference in power consumption dependent on his frequency. Since the oscillators consist of NOT-gates, there will be more switching activity on higher frequencies, which in turn results in a higher power consumption. The possibility to measure the power consumption for each oscillator in sufficiently high accuracy results in the same full model of the PUF as mentioned above.

However both approaches suffer in terms of feasibility. Depending on the implementation, there may be more than two oscillators active at the same time, which results in high interference while trying to measure a single oscillator. Additionally, a close proximity between the oscillators increases such effects. Another problem is that the emissions of the oscillators are very low, which makes it hard to find the correct position for the EM probe to measure single oscillators.

5.1.2. Counters

In addition to the power consumption of the oscillators themselves being directly proportional to their frequency, the power consumption of the counters relates to the frequency of the oscillator being monitored. Every operation that increments the counter results in power consumption. The higher the frequency of the monitored oscillator, the more incrementation steps of the counter are executed.

Knowing the count of both counters, at the time of the comparison step, means knowing the result of the comparison as well. Hence, it is possible to measure the noisy response of a PUF for a given challenge. To obtain the key, it is still necessary to compute the error-correction and hashing steps afterwards. This is less powerful than the full model, which the previous attack scenario would yield, because the result is restricted to the current challenge. If it is desired to get a model for more than one challenge, measurements for every challenge in question had to be conducted. Although, it might be possible to get a full model with fewer measured challenges by carefully selecting them to infer relations between oscillators.

As with the measurement of the oscillators above, the increments of a counter can hardly be measured, because it is just a small single operation. However there are just two positions of the counters that have to be found, compared to over a thousand positions of oscillators. Given that it is possible to find the position of those and measure with a sufficiently high accuracy, this approach seems to be more practical than the measurement of every single oscillator.

5.1.3. Comparator

In contrast to the attacks suggested so far, where a time period has to be monitored, stands the attack of the comparator. The comparator only gets used on known points in time. That is because the time span for which the oscillations are counted is constant. That means, that after the first point in time a peak caused by the comparator is found, it is easy to find the remaining comparator peaks as well. Different comparison operations should show a different power consumption pattern. This may result from leakage current, when toggling the output or different switching operations during the comparison, depending on the implementation of the comparator. An investigation of those peaks may directly result in the noisy response of the PUF for a given challenge, which is basically the same result obtained as by attacking the counters as mentioned above.

5.2. Error Correction

Regarding the error correction mechanism, this analysis is restricted to the helper data algorithm used in the prototype, which was already explained in Chapter 4.2. Furthermore, only the reconstruction of the key is considered, because an attacker does not have access to the PUF during initial generation of the key, in a realistic scenario.

As Figure 4.4 showed, there are two XOR operations involved, independent of the used error correction algorithm. The attacker has no influence on the noisy PUF response R', but can manipulate the helper data W_1 at will, because it is stored in unsecured memory. Also, it is not possible to directly influence the corrected code word C_s, because it is the result of the error correction algorithm. However, there is an indirect influence of C_s due to the fact, that the input of the error correction algorithm is the result of $R' \oplus W_1$, with W_1 being under the control of the attacker.

Power consumption of XOR-gates can be modelled with the hamming weight assumption, which is e. g. mentioned in [15]. Since power consumption depends on switching activity, it can be estimated, that the power consumption will be lower when fewer bits have to be switched. Looking at an XOR operation, there are no bits switched, if the inputs are identical and all bits get switched, if one input is the complement to the other. All this can be summed up in the hamming weight of the output, because, regarding an XOR operation, the hamming distance of the inputs equals the hamming weight of the output.

When attacking such a XOR operation, one of the inputs should be fixed and the other variable. Measurements of power consumption or EM emission could be conducted for every possible value of the variable input. After that, it is possible to compare the traces at the point in time, where the XOR operation was executed. Given the above mentioned power consumption model, the variable input of the XOR-gate was identical to the fixed input, when the lowest power consumption was measured. Hence it is possible to obtain the value of one operand by brute-forcing all possible values of the other input.

In the scenario of the actual prototype, there is the choice between two XOR operations to attack, i. e. $W_1 \oplus R'$ or $W_1 \oplus C_s$. Due to the fact, that W_1 is fully controllable by the attacker, the operands R' and C_s would be the respective targets. However R' is afflicted with noise and hence not fixed, which makes it unsuitable as a target for the attack. During a normal run of the helper data algorithm C_s is always fixed, because it is the result of the error correcting function, which eliminates possible noise. Choosing C_s as the target value leaves the attacker with two problems. The first problem is that a brute-force attack may not be feasible or too costly, because of the large number of measurements that have to be conducted. The second barrier is that C_s can not be reconstructed any more, given that W_1 may take every possible value. Because C_s results out of error correcting $W_1 \oplus R'$, it gets indirectly influenced by W_1. Choosing a value for W_1 other than the original one can be seen as additional noise to that in R'. Thus the error correction algorithm may result in a different code word than C_s, if the hamming distance of the chosen value and the original value of W_1 gets too big.

However, both problems can be alleviated by the same measure. By only brute-forcing a certain amount of bits at one time and keeping the others at the original value of W_1, it is possible to reduce the complexity of trying all possible values as well as keeping C_s fixed. The aim is not testing every value in the possible number range, but to get measurements for every value of every single bit. Keeping everything fixed and toggling just single bits might however not yield a notable change in the side-channel. That is why it is proposed to divide the n bits in groups of x bits, with x being, for example, 4. Instead of conducting 2^n measurements, it gets reduced to just $\frac{n}{x}2^x$, with x being considerably smaller than n. Choosing a small value for x also implies staying below the error correction threshold. After taking those measurements, the measurements of the single groups are compared against each other, with the one containing the lowest power consumption being the candidate for that group. The combination of all of those candidates then yields the candidate for the whole bit string and hence the most probable value for C_s.

5.3. Hashing

As mentioned in Chapter 4.2.2, the hashing implementation of the prototype relies on Toeplitz hashing. To understand the idea behind the attack, it is important to remember, that each of the input bits from $W_1 \oplus C_s$ is getting shifted to the gate, one bit per cycle. The gate will then cause a XOR operation of the current value of the LFSR and an accumulator register, depending on the value of the input bit.

The branching at the gate allows to execute a side-channel attack, because the XOR operation with the value of the LFSR and the output register only occurs when the input bit is 1. In the other case, no operation is executed at all, apart from the LFSR-shift, which is executed in both cases. Hence, there is a great difference in power consumption dependent on the input bits. This way the attacker gains knowledge of the value of the error corrected PUF response and only needs to calculate the hash himself, with the knowledge of W_2, to get the key.

5.4. Summary

As seen above the suggested attacks vary in difficulty and prospect of a successful attack as well as the target PUF they can be conducted on. Attacking RO-PUFs by trying to measure the different frequencies of every single oscillator would have a great impact, because it results in a full model of the PUF. However a successful attack is hardly feasible, because of the many oscillators that have to be characterised and brought in the right order. The attack on the counters offers better prospects, because the attacker only has to monitor two counters, which lowers the effort compared to measuring every single oscillator. The last attack on the RO-PUF, comparing the peaks of the comparator in measurement traces, should be the easiest one to conduct, because clear peaks can be expected in comparison to the previous mentioned approaches.

Attack	Prospect	Target PUF
Oscillators	$--$	RO-PUF
Counters	$-$	RO-PUF
Comparator	$+$	RO-PUF
Error correction with different helper data	$++$	generic
Toeplitz hashing	$++$	generic

Table 5.1.: Overview of suggested side-channel attacks

The other suggested attacks are not targeting specifics of the RO-PUF and hence are generic attacks, also suitable for other types of PUFs. Clear peaks in measurement traces can be expected from both, attacking the XOR of the error correction and the toeplitz hashing, which makes both approaches promising.

6. Implementation

In this chapter the proceeding while implementing a side-channel attack for the FPGA PUF prototype is discussed. The description will emphasise on the hashing phase for which the attack was successfully implemented.

6.1. Measurement Setup

As mentioned above, an EM probe (Langer ICR HH 150) was used for obtaining the measurements in connection with a Langer PA 303 amplifier. Traces were recorded at a rather high sample rate of 20 GS/s with a LeCroy wavePro 715Zi oscilloscope. The prototype itself was implemented on a Digilent Nexys2 board hosting a Xilinx Spartan3E-1200 FPGA, running at 50 MHz. Figure 6.1 shows a photo of the measurement setup, with the EM probe being aligned over the FPGA IC.

6.2. Software

6.2.1. Measurement

For controlling the PUF prototype, i. e. issuing *enrol* and *reconstruct* commands, a program written in C was used. The used prototype does not read its helper data from external insecure memory, but receives them as parameters of the *reconstruct* function. This makes it easy to log the generated helper data from enrolment, as well as to manipulate helper data for every run of *reconstruct*. Because this software also decides when to execute which command on the prototype, it is also used for controlling the oscilloscope and storing the traces. This simplifies tasks like saving traces with different used helper data to different folders or different file naming conventions. For the interface between the computer and the FPGA a serial RS-232 connection was used, because a USB connection induced too much noise. Measurement data is stored in binary files as concatenated unsigned 8 bit integers without any headers.

6.2.2. Analysis

In order to analyse the recorded measurements, a custom plotting tool was utilized. This software is coded in Python (>2.5) with PyQt (4.7.3) front end and uses the libraries numpy (1.4.1) and matplotlib (0.99.3) for calculations and plotting.

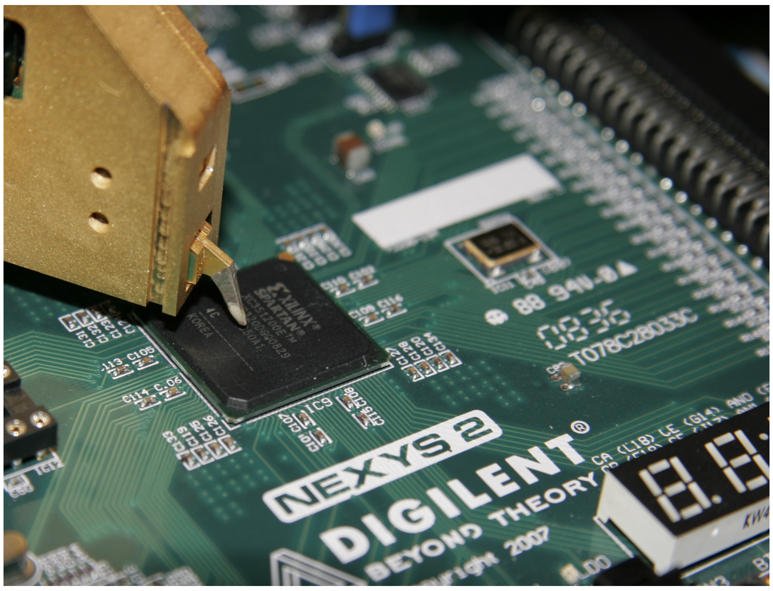

Figure 6.1.: Photo showing the Langer ICR HH 150 probe over the Xilinx Spartan3E-1200 FPGA IC

Apart from this tool, a lot of analysis was done using above named libraries in an interactive Python shell. This allowed for a fast overview over gathered data, interactive calculations and prototyping of the proof-of-concept code.

6.3. Evaluation of Measurements

The first step was evaluating where to position the probe for getting meaningful traces. Therefore the prototype was running reconstruction operations in an infinite loop, while the position of the EM probe was changed and the results were observed on the oscilloscope. First of all, it was necessary to to get a clean signal with a good signal to noise ratio. Those position candidates were narrowed down further, so they showed a pattern that fitted the model of the reconstruction operation. This pattern should show periodic peaks for the oscillator/comparator phase, as well periodic behaviour during hashing at the end. Furthermore, the cycle period of the comparator peaks should match the known time period, for which the oscillators are running. Additionally the FPGA was configured for emitting a trigger signal to aid in differentiating the three phases. An example of such a trace is given in Figure 6.2, where mostly the oscillator phase with the characteristic comparator peaks can be seen. In order to see the error correction and hashing phase, it is necessary to zoom in at the very end of the trace. This magnification is shown in Figure 6.3, where the first three peaks are the last

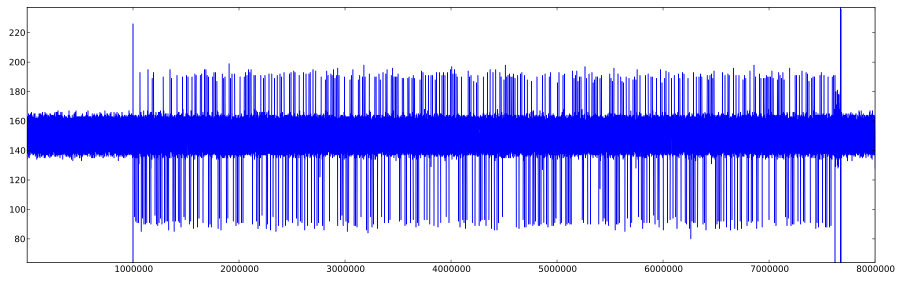

Figure 6.2.: Complete trace of the reconstruction operation

comparator peaks. After a trigger, the error correction appears in three peak groups, followed by a hardly recognisable hashing phase between the last two trigger signals.

6.4. Evaluation of the Hashing Phase

After obtaining a suited placement for the EM probe, the time frame of the hashing operation was measured relatively to the trigger at the end, which signalised the successful reconstruction of a key. In order to get clean traces without much noise, 3.000 measurements of 100.000 samples each were conducted and the traces were aligned and averaged. The result is given in Figure 6.4, where you can see the hashing occurring between the major spike after 20.000 samples, which appears at the end of error correction, and the trigger peak after 80.000 samples.

As mentioned in Chapter 5.3, the used hashing implementation is mainly based on a LFSR and an XOR operation. Since only the power consumption of the XOR is relevant, the shifting of the LFSR can be seen as additional noise. However, since the LFSR operations are always the same across all measurements, they remain in the averaged trace and are not eliminated like the noise. Given that the initial value of the LFSR is determined by helper data W_2, it is possible to set it to a new random value on every measurement. In this way, the results of the XOR operation are different in every measurement run and the end result is not the correct key. However, the decisions at the gate, whether there is a XOR operation or not, stay the same. Hence, operations which are not relevant for the attack get varied and because of that eliminated in the averaged trace, while relevant operations are kept constant and are reinforced in an averaged trace.

The result of the measures above is a cleaner trace with less noise, but it is still not possible to notice obvious patterns, which allow to draw conclusions about the input bits coming from the error correction. Although there are obvious spikes, there are also smaller varying peaks between them and it is not sure which of them contain information about the input bits. Therefore, it is preferable to know, which of the peaks

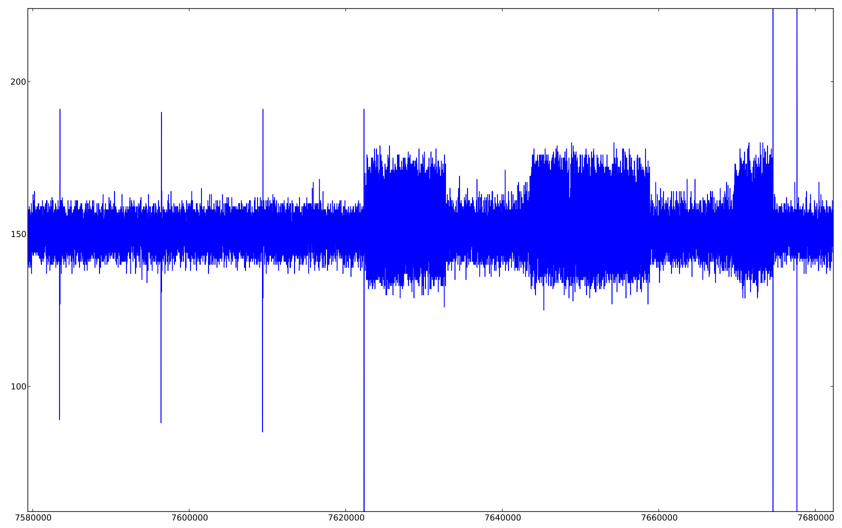

Figure 6.3.: Magnification of the last part of the complete trace. Three trigger signals can be identified: The first between oscillator phase and error correction phase, the second between error correction and hashing and the last at the end of hashing.

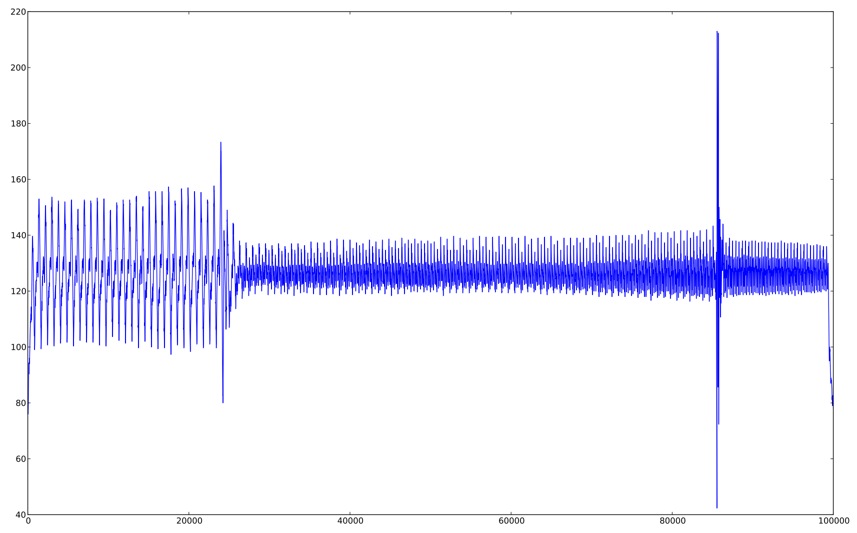

Figure 6.4.: Averaged trace of 3000 measurements of the hashing phase preceded by the end of the error correction phase. A trigger signals the end of the hashing operation.

are caused by processing input data. So far, noise, varying between measurements, was eliminated by averaging. Now, the inverse approach is used to obtain a trace, primarily showing everything, which consumes power regardless of the input bits. However, it is not possible to directly change those input bits. It is possible to vary helper data W_1, which has an influence on the bits, we want to change. Changing W_1 in a completely random manner has enough impact on the error correction, so that it corrects to different code words, yielding varying input bits for the hashing phase. There were 30.000 measurements with randomized W_1 conducted and averaged, leaving a trace that characterises everything apart the influence of the input. Calculating the difference between the averaged trace without noise and the newly obtained trace, leaves a trace, where peaks show at which points in time input bits play a role. An example of those traces is given in Figure 6.5. At face of it there is no gain of information, since for every peak in the trace to analyse, there is a peak as well in the other two traces. This would mean every single peak should be of importance. However, it is noticeable, that every other peak varies in height, although the corresponding peaks of the other traces do not show this variation. Hence, there is more information contained in these peaks compared to the others. Another fact, which is assuring that only every other peak is important for the analysis, is having 148 positive peaks in the trace, whereas only 74 bits are used for hashing.

An explanation for alternating two types of peaks lies in the way the Toeplitz hashing is implemented in the prototype. One input bit is processed in two cycles, one for shifting the LFSR and the other for the XOR operation. Note well, that this is an implementation detail of the prototype and might be as well implemented together in one cycle. The interval in samples between the peaks fits the clock rate of 50 MHz of the used FPGA and confirms that the varying half of the peaks is caused by the XOR operation.

Comparing the values of the peaks to the error corrected bits from a debugging log file of the prototype's operation showed that the higher those peaks are relatively to each other, the more likely the processed bit was 1. This confirmed the proposed model of the power consumption regarding the XOR operation. However, all of the traces obtained showed that the height of the peaks was increasing towards the end of the hashing operation. Although an explanation for this was not found, there is a way of compensating this effect to still being able to extract the bit sequence. Since half of the 148 peaks are almost constant, except for the increasing trend towards the end, those peaks can be used as a reference. By using the ratio of the information bearing peaks to the peaks that follow them instead of using the direct peak value, the influence of the increasing trend is compensated.

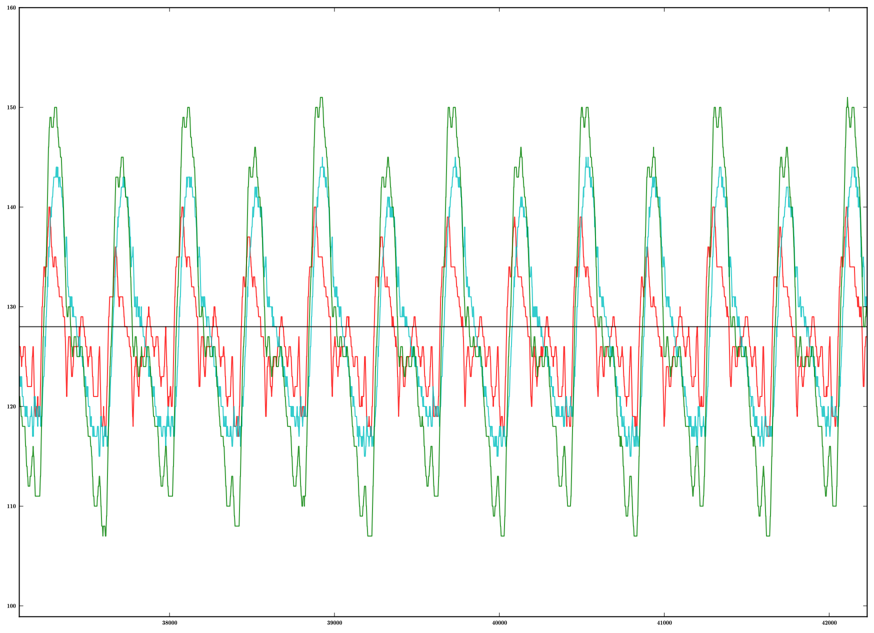

Figure 6.5.: Trace with indication of relevant peaks. The actual trace with randomised W_2 (red) is overlayed by the trace with randomised W_1 (green) and the differential trace (bright blue).

6.5. Proof-of-Concept for Attacking the Hashing Phase

A proof-of-concept software was implemented, to present that it is possible to extract the bits being hashed exclusively by using measurements gathered by normal operation of the PUF.

Because there are many parameters that need to be adapted for different measurement runs and measurement setups, this proof-of-concept has the form of a wizard. It aids to tune those parameters step by step and yields the most probable bit string according to those parameters and the given traces.

The program is called with the path to the measurement traces as a parameter. In the first step, the traces are read into memory before they get aligned. The alignment is necessary, because there are always small variations in the traces. Additionally, when randomising helper data, some operations, like error correction, might need a different amount of time, hence adding to shifts in the traces. Since every trace features the trigger peak at the end, the function first tries to find that peak in every trace and then aligning them according to that peak by prepending or appending zeros to the

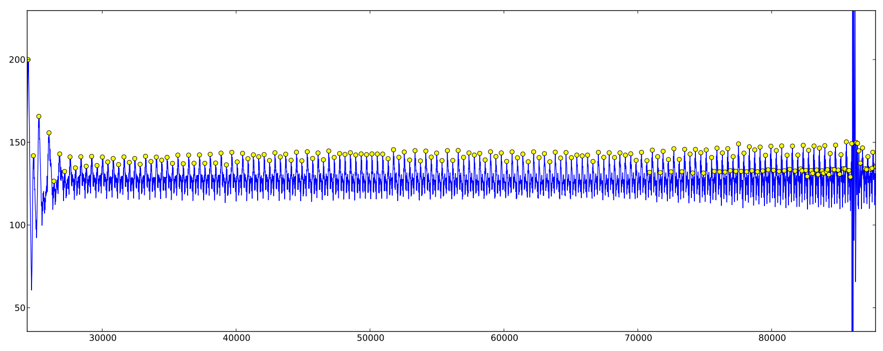

Figure 6.6.: Proof-of-concept step 1: Preview of peak detection. In this case the parameter for delta was set to 13.5.

traces. After being aligned, the mean of every data point gets calculated, which leaves one averaged trace.

The next step consists of detecting the distinct peaks in the averaged trace. The helper function, which accomplishes this, is inspired by the peak detection mechanism mentioned in [1]. This approach is suited for peak detection in spiky and noisy signals, where the well-known zero-derivate method to find maxima and minima is not suited. The problem that is to be solved here, is the automatic detection of the peaks and valleys, which are obvious for the eye and not maxima and minima in the mathematically sense. Basically, peaks are defined here as the highest point surrounded by points that are a certain delta lower. This delta is one of the above mentioned parameters, which need manual tuning. As an aid for finding the correct value, the wizard shows a preview of the trace, where the peaks are marked, which were found using the given delta value. Such a preview window is presented in Figure 6.6. There might be the possibility of smaller peaks being detected (e.g. at the end of the trace of Figure 6.6), which will be eliminated in the next step. The main focus of this step should be, that all relevant peaks are being detected without too much false positives.

The knowledge, that there have to be 148 peaks, which have an equal interval between them, can be used to find the relevant peaks in the set of already detected peaks. The wizard already suggests a value, which is the median of the distances between all peaks. Depending on the number of false positives found, it is already a good estimation of the actual interval. The value of the interval can also be found by measuring the number of samples the hashing operation takes and dividing that by 148. The peaks which are left after applying this method are shown in Figure 6.7.

In the next step, the information bearing peaks have to be weighted to compensate for overall variations in power consumption during the hashing operation. Hence, like already described above, every other odd peak is divided by the following even peak.

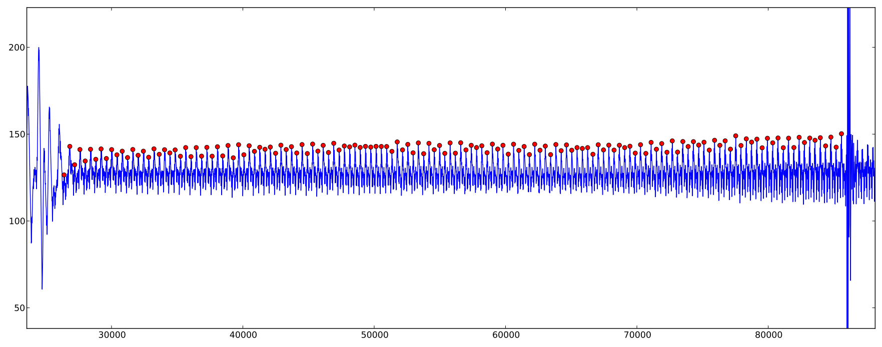

Figure 6.7.: Proof-of-concept step 2: Preview of significant peaks. Here, the assumed value of the interval was set to 400.

Each of the resulting values being below a certain threshold means there is a 0 at that position in the bit string, every value being above that threshold indicates a 1. Given enough measurements to eliminate the noise, such a threshold could always be found manually, when knowing the actual bit string. But because a potential attacker wants to gain knowledge of this exact string, this trivial method is not an option. Assuming a balanced number of zeros and ones in the string, the mean of all values is a good approximation of the threshold. This was confirmed by applying the proof-of-concept on real measurement data, where the approach of taking the mean as the threshold only failed with too noisy data. Performance and error rate of the proof-of-concept code will be discussed in Chapter 7.

7. Evaluation

In this chapter the suggested attacks are evaluated based on conducted measurements of the RO-PUF prototype. Performance and error rates of implemented attacks are given and the problems are discussed.

7.1. Attack of Toeplitz Hashing

First, the implemented proof-of-concept attack of the Toeplitz hashing phase is evaluated. Here, the focus is on how accurate the, to be hashed, bit string can be recovered from the measurement data. It is obvious, that the accuracy of the attack increases with the accuracy of the gathered data and lack of noise it contains. Since noise can be reduced by averaging the traces, clean data can be obtained by measuring a large number of traces. However, gathering a lot of data is time and disk space consuming, therefore it is preferable to keep the number of measurements to be conducted small.

As already discussed in Chapter 6, an optimising approach that can be taken while conducting the measurements is randomising the helper data W_2. Since it alters the operations regarding the LFSR, the power consumption for those operations varies as well and can be cancelled in the traces, like noise, by averaging. This results in a clearer signal of the operations of interest.

Because the proof-of-concept code only evaluates the positive peaks, it is possible to get a more distinct trace by cutting the negative values already at the oscilloscope and using the full available range of values for the positive peaks. This causes the ratio between peaks to get more pronounced, which is advantageous for the weighing of the peaks. However, those traces might be more prone to noise, since spikes caused by noise will also have a higher influence.

A comparison between the suggested approaches regarding error rate can be seen in Table 7.1. There, it is shown, that all measurement methods should work fine with over 3000 traces. If measurements are restricted to around 1000 traces, the naive approach is getting less efficient and guesses 5 bits wrong out of the 74 bit string, which is to be hashed. Compared to that number of errors, measurements can be reduced to around 200, when randomising helper data W_2. The only measurement method yielding error free results for more than 1000 measurements is randomising W_2 and additionally using the whole available range for positive values. However that approach performs even poorer than the other two alternatives, when noise cannot be faded out enough, because of an insufficient number of traces. This is caused by noise peaks being scaled

Number of Traces	Normal	Randomised W_2	Randomised W_2 ”fullrange”
3000	1 (1.4%)	0 (0%)	0 (0%)
2000	2 (2.7%)	1 (1.4%)	0 (0%)
1000	5 (6.8%)	1 (1.4%)	0 (0%)
500	6 (8.1%)	3 (4.1%)	1 (1.4%)
200	9 (12.2%)	4 (5.4%)	6 (8.1%)
100	13 (17.6%)	11 (14.9%)	13 (17.6%)
50	18 (24.3%)	13 (17.6%)	21 (28.4%)
20	24 (32.4%)	16 (21.6%)	24 (32.4%)

Table 7.1.: Errors and error rates of the hash attack with different number of measurements

and confused with the actual peaks when trying to find the relevant peaks. Hence, if there is no possibility to obtain more than 500 traces, just randomising W_2 is the most accurate method. If it is possible to record more measurements, it is advisable to scale the positive positive part of the traces using the whole value range, because then the proof-of-concept will most certainly have error free results.

7.2. XOR Attack in Code-Offset Algorithm

As discussed in Chapter 5, the XOR operation after the error correcting step could be used to attack the error correction phase. However, there is a difference of the attacked XOR during hashing and the XOR after error correction. In contrast to the hashing phase, where the varying power consumption was caused by executing an XOR operation at some points in time and not executing it at other points in time, there is only the one attacked XOR operation, always being executed, after error correction.

To still be able to attack this operation, the power consumption depending on inputs has to be modelled, like with the suggested hamming weight assumption. Traces obtained from operation of the prototype, however, did not show the behaviour, which is expected considering the hamming weight model. This is probably because there are no actual XOR gates in the implementation of the FPGA based prototype, but the functionality is implemented via look-up tables. Furthermore, comparisons of runs with different inputs did not result in an alternative power consumption model for the FPGA prototype. Additionally, there is no single peak representing the XOR of the error correction, but a group of three peaks. Such post-pulse oscillations can be caused when a device operates at high clock frequencies, where strong peaks affect also the following cycles. Figure 7.1 is zoomed in on the first two peaks and shows, that there are no clear patterns between the chosen extreme cases. Cases where certain patterns arose, were negated by counterexamples of other bit groups.

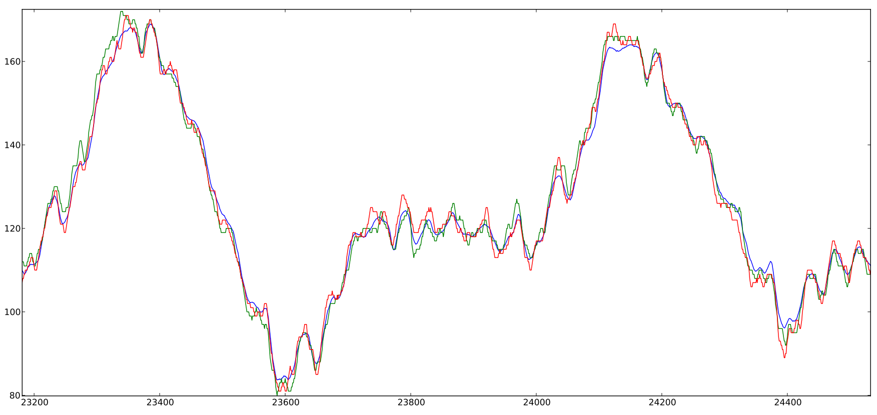

Figure 7.1.: Zoomed view on the first two peaks of the XOR operation during error correction. The blue line shows the mean of all the traces with unaltered helper data, as a reference. The single trace, where the first 4 bits of helper data W_1 are identical to the first 4 bits of C_s (which is the other input for the XOR), is displayed in green. In contrast the red line shows the opposite case, i. e. the first 4 bits of W_1 are the inverted first 4 bits of C_s.

Although the attack could not be implemented for the given FPGA prototype, because of the above mentioned reasons, it is still considered a valid attack for other implementations. Furthermore, the attack would still be possible for an FPGA-based RO-PUF, assuming a power model for the XOR operation of look-up tables can be found. This, however, was out of the scope of this work.

7.3. Attack of the RO-PUF

Like with the attacks of the error correction mechanism, a successful proof-of-concept of the attacks suggested for the actual RO-PUF part of the prototype was not implemented. This is due to the already discussed bad prospects of those attacks (cf. Chapter 5), which are now confirmed, in the following, by actual measurements.

7.3.1. Oscillators

Besides high interference between oscillator pairs, the biggest concern, regarding the feasibility of attacks against the oscillators, was the strength of measurable emissions. Actual measurements confirmed this concern, as can be seen in the left half of Figure 6.3. The oscillator phase could only be identified by the comparator peaks, which occurred with the expected period. During the time, where the oscillators are running, there is no signal recognisable, which is different to noise. This is illustrated in

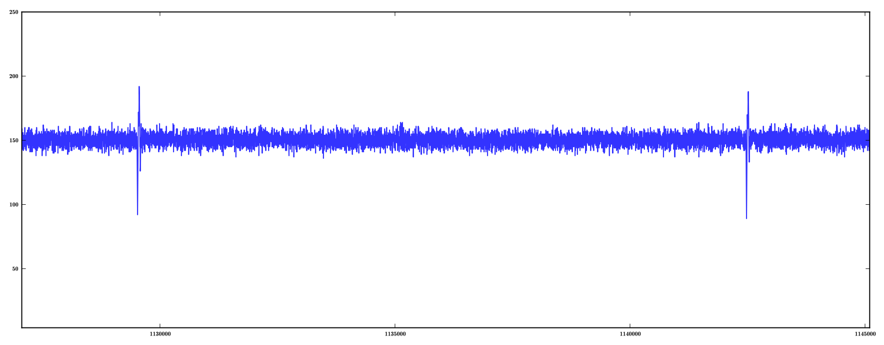

Figure 7.2.: Zoomed view on the interval, when oscillators are running.

Figure 7.2, which shows a magnification of the trace between two comparator peaks. Without a refined measurement setup and higher measuring sensitivity, this attack cannot be considered realisable. Even if the emissions of the oscillators can be measured with sufficient accuracy, there is still the problem of highly probable interferences. Additionally, this is even getting more complicated, because it is hard to locate every single oscillator. In the case, where oscillators are located very close to each other, interferences are also increased.

7.3.2. Counters

Although the attack of the counters seems more likely possible, than the attack on the oscillators, the power consumption and EM emission of the counters is comparable to those of the oscillators. This can also be seen in Figure 7.2, where no signal emerges from the noise. Hence, in regard of practicability, the same conclusion as with the oscillators applies. However, targeting the counters still means less measurement effort, because there are only two counters compared to the number of oscillators, which is almost two orders of magnitude higher. Also, those two counters can be located more easily compared to the location of every single oscillator, like mentioned above.

7.3.3. Comparator

In contrast to other RO-PUF related attacks, there are clearly visible peaks in the traces at those points in time, when the comparator operations are supposed to be executed. Apart from the trigger peaks, they are the only clearly recognisable spikes in the full trace, given in Figure 6.2.

Similar to the attack of the XOR operation during error correction, it was not possible to extract a power model for the comparison implemented in look-up tables, given the scope of this work. However, assuming the knowledge of such a model, the attack of the comparison operations is to be considered as a feasible way of obtaining the noisy response to a given challenge.

7.4. Possible Countermeasures

The proof-of-concept of one suggested theoretical side-channel attack against PUFs shows, that those attacks are of practical relevance. Hence, it is necessary to reconsider the designs of the PUFs and their fuzzy extractor operations, like error correction and hashing, in order to protect them against such attacks. Therefore, this section gives suggestions for possible countermeasures.

The hashing phase is the most promising attack surface for the prototype in question. Since the operation of the Toeplitz hashing only executes the XOR operation, when input bits are set to 1, a detailed power consumption model of the actual XOR operation is rendered unnecessary. However, it is of great importance to have a hardware efficient hash function, because the PUF and its surrounding logic should not waste valuable space on the IC or FPGA. This is why more complicated hash functions are not desired in practice. Yet, it is necessary to alter this hashing mechanism in order to prevent information leakage through this side-channel.

One suggestion for concealing the power consumption differences between the cycles is given in Figure 7.3. The idea is to introduce a second register, on which the XOR operation with the LFSR is executed, whenever the real operation does not get executed. This is implemented by controlling the fake logic circuit with the inverted bits of the shift register. Hence, there is a XOR operation executed in every cycle, independent of the input bits. This alleviates the noticeable deviations in power consumption. Even if a detailed power consumption model of the XOR operation is known, an attacker would need to know which circuit is used on which cycle. However, this is equivalent to the knowledge of the input bits, which are the actual target of the attack. Although being a valid countermeasure for FPGAs as well as for ASICs, there is the drawback, that the original Toeplitz hashing logic gets almost doubled.

A slightly different approach is presented in Figure 7.4, where a multiplexer is used instead of a gate. In the cases, where the input bit is set to 1, the value of the LFSR is used in the XOR operation. Instead of no XOR operation being executed in case of a 0 as the input bit, there will also be an operation, but with a predefined static value. Considering a static value of all bits set to 0, the value of the accumulator register wont be changed after the XOR operation. This way, the hashing operation will yield the exact same result as the original approach, with the exception of having a XOR operation executed in every cycle. Therefore, the information leaked through the side-channel is not as distinct as compared to the original hashing. Although, it has to be noted, that since there is no change in output bits during the inserted operation, it should show a different power consumption behaviour and hence be distinguishable from the normal XOR operation. However, a change in output bits can be easily

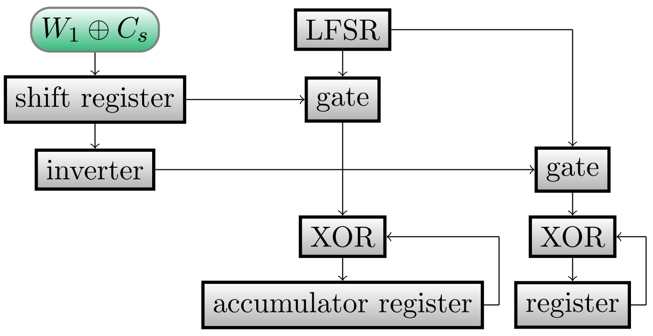

Figure 7.3.: Toeplitz hashing with additional inverted XOR operation

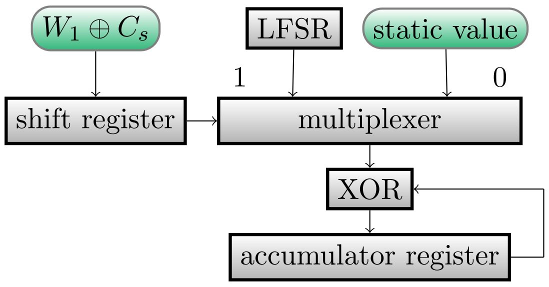

Figure 7.4.: Toeplitz hashing with static alternative value

achieved with that approach by choosing the static value as all bits set to 1. The result of a XOR with this value equals inverting every bit of the input. Taking the hamming weight assumption as a basis, this does not seem to be an improvement, because the alternative operation will have a high power consumption every time, if all static bits are set to 1, and a low power consumption, if all static bits are set to 0. Given that there is no accurate model known, like in the case of the FPGA-based prototype, it can be an improvement to the vanilla Toeplitz hashing regarding side-channels. Depending on the implementation details it could also be more efficient in terms of gates compared with the approach, where the logic gets doubled. It has to be noted, that setting all bits of the static value to 1 will not result in the original Toeplitz hash any more. This is caused by an inversion to be on hand, when the number of zeros in the input string is odd. However, since the same function is executed during enrolment, where the key is defined, it will still yield the same key.

Regarding side-channel countermeasures for the error correction phase, it is more difficult to find a solution without changing the whole proven concept of code-offset construction [3]. However, since the suggested attack relies on changing groups of bits in helper data W_1, which can be seen as introducing noise, a fine tuning of the error correction parameters helps making the execution of this attack more difficult. By minimising the safety margin between actual errors and the maximal number of errors that can be correct by the error correction code, the attacker gets limited in the number of bits in W_1, which he is able to change at once. This should be a general goal, because the use of simpler error correction codes, which only correct fewer errors, is beneficial in terms of hardware complexity as well.

Looking at the actual measurements and initial considerations, there is practically less need for countermeasures regarding the attacks on the oscillators and counters of the RO-PUF. This leaves the comparator as a possible target for side-channel attacks. As discussed above, a matching power consumption model and further analysis would make an attack of the comparator feasible. However, the comparator is an intrinsic part of the design of current RO-PUFs and not avoidable as such. The only obvious solution would be timed operations, which run in parallel to the comparator, inducing noise into measurements and therefore concealing the actual power consumption of the comparator.

8. Conclusion

The contribution of this thesis is an analysis and demonstration of side-channel attacks against Physical Unclonable Functions. It hereby fills the gap between other analyses regarding the security of PUFs. Previous studies were mainly focused on functional improvements, modelling of PUFs and improvement of error correction mechanism and hashing, regarding information theoretical security standpoints as well as hardware complexity. Although side-channels were mentioned in some previous work, analyses focusing on them were not conducted.

Firstly, attacks against PUFs based on side-channels were developed in theory. Here, it became obvious, that attacks are possible, which are not only applicable to a specific kind of PUF, but also appropriate for a range of different types of PUF. This is because, in most usage scenarios, PUFs need additional logic to be useful. Examples for this are error correction mechanisms, to cope with the noisy nature of responses generated by the PUF, and hashing, to pre-process the response in order to make it usable for cryptographic operations. Hence, not only the PUF itself provides attack surface in the context of side-channel attacks, but also its ambient logic.

Secondly, the theoretical results were practically evaluated using an FPGA-based RO-PUF prototype. In order to conduct the necessary measurements, an EM probe was used, to be able to perform the attacks as non-invasive as possible. The analysis of different measurements showed the respective feasibility of the theoretical attack scenarios with standard measurement equipment.

Furthermore, it was possible to develop a proof-of-concept for the most promising attack, exploiting the design of the so called Toeplitz hashing, which is commonly used because of its hardware efficiency. With the help of the proof-of-concept code, it is possible to recover all of the bits in the 74 bit, to be hashed, bit string by analysing the data of 1000 power traces.

Finally, different approaches for countermeasures against the suggested attacks were suggested. Although those countermeasures could not be evaluated in terms of their practical effectiveness, they should nevertheless show, how side-channel based attacks could be complicated or even avoided.

In conclusion, it is important to state that side-channels have to be considered in the development and further improvement of PUFs. Given that all security features that PUF-based technology offers can be lost by leakage through a side-channel, it is necessary to not only improve hardware complexity or maximise entropy, but also include countermeasures avoiding leakage in future designs.

Bibliography

[1] Eli Billauer. peakdet: Peak detection using matlab. http://billauer.co.il/peakdet.html, September 2008.

[2] Christoph Bösch, Jorge Guajardo, Ahmad-Reza Sadeghi, Jamshid Shokrollahi, and Pim Tuyls. Efficient helper data key extractor on fpgas. In *CHES '08: Proceedings of the 10th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 181–197, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM J. Comput.*, 38(1):97–139, 2008.

[4] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Controlled physical random functions. *Computer Security Applications Conference, Annual*, 0:149, 2002.

[5] Jorge Guajardo, Sandeep S. Kumar, Geert Jan Schrijen, and Pim Tuyls. Fpga intrinsic pufs and their use for ip protection. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 63–80. Springer, 2007.

[6] Jorge Guajardo, Sandeep S. Kumar, Geert Jan Schrijen, and Pim Tuyls. Brand and ip protection with physical unclonable functions. In *ISCAS*, pages 3186–3189. IEEE, 2008.

[7] IBM Corporation. IBM 4764 Model 001 Specification Sheet, 2008.

[8] Tom Kean. Cryptographic rights management of FPGA intellectual property cores. In *FPGA*, pages 113–118, 2002.

[9] Kocher, Jaffe, and Jun. Differential power analysis. In *CRYPTO: Proceedings of Crypto*, 1999.

[10] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, pages 104–113, 1996.

[11] Hugo Krawczyk. Lfsr-based hashing and authentication. In *CRYPTO '94: Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, pages 129–139, London, UK, 1994. Springer-Verlag.

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

[13] Dominik Merli, Frederic Stumpf, and Claudia Eckert. Improving the quality of ring oscillator pufs on fpgas. In *5th Workshop on Embedded Systems Security (WESS'2010)*, Scottsdale, AZ, USA, October 2010. ACM.

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

[15] Eric Peeters, François-Xavier Standaert, and Jean-Jacques Quisquater. Power and electromagnetic analysis: Improved model, consequences and comparisons. *Integration*, 40(1):52–60, 2007.

[16] Ulrich R ührmair, Frank Sehnke, Jan S ölter, Gideon Dror, Srinivas Devadas, and J ürgen Schmidhuber. Modeling attacks on physical unclonable functions. In *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security*, pages 237–249, New York, NY, USA, 2010. ACM.

[17] Ulrich Rührmair, Jan Sölter, and Frank Sehnke. On the foundations of physical unclonable functions. Cryptology ePrint Archive, Report 2009/277, 2009. http://eprint.iacr.org/.

[18] Frank Sehnke, Christian Osendorfer, Jan Sölter, Jürgen Schmidhuber, and Ulrich Rührmair. Policy gradients for cryptanalysis. In Konstantinos I. Diamantaras, Wlodek Duch, and Lazaros S. Iliadis, editors, *ICANN (3)*, volume 6354 of *Lecture Notes in Computer Science*, pages 168–177. Springer, 2010.

[19] Eric Simpson and Patrick Schaumont. Offline hardware/software authentication for reconfigurable platforms. In *Cryptographic Hardware and Embedded Systems - CHES 2006*, pages 311–323. 2006.

[20] G. E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 9–14, 2007.

[21] Pim Tuyls, Geert-Jan Schrijen, Boris Škorić, Jan van Geloven, Nynke Verhaegh, and Rob Wolters. Read-proof hardware from protective coatings. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, chapter 29, pages 369–383. Springer Berlin Heidelberg, 2006.

[22] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design a Systems Perspective*. Addison-Wesley, Reading, 1985.

[23] Meng-Day (Mandel) Yu and Srinivas Devadas. Secure and robust error correction for physical unclonable functions. *IEEE Des. Test*, 27(1):48–65, 2010.

A. Appendix

A.1. Source Code of Proof-Of-Concept for Attacking the Hashing Phase

```
#!/usr/bin/env python

import os
from optparse import OptionParser
import numpy as np
import matplotlib.pyplot as plt
from helperfuncs import *

def get_sig_peaks(peaks, num_of_points=148, margin=50, med_dist=None):
    """
    Extracts relevant peaks for the analysis out of previously detected
    peaks and returns their positions.

    Arguments:
    peaks -- detected maximum peaks as returned from the
             get_peaks helper function
    num_of_points -- the number of points relevant for the analysis
                     (number of points = number of secret bits * 2)
    margin -- safety margin for detection of relevant peaks
    med_dist -- expected approximate distance between relevant peaks
    """
    if not med_dist:
        med_dist = np.median([peaks[i+1][0]-peaks[i][0]
            for i in xrange(len(peaks)-1)])
    trigger=np.argmax([item[1] for item in peaks])
    res = []
    for i in xrange(trigger,0,-1):
        if peaks[trigger][0]-peaks[i][0] >= med_dist+margin:
            res.append(i)
            num_of_points -= 1
            break
    for i in xrange(res[0], 0, -1):
        if peaks[res[-1]][0]-peaks[i][0] >= med_dist-margin:
            res.append(i)
            num_of_points -= 1
            if num_of_points==0:
                return res[::-1]
    assert "get_sig_peaks failed"

def weigh_peaks(peaks, sig_peaks):
    """
    Weights peaks according to the ratio between peak to analyse
    and the following unimportant peak.

    Arguments:
    peaks --  maximum peaks as returned from the
              get_peaks helper function
    sig_peaks -- positions of relevant peaks as returned
                 by get_sig_peaks
    """
    dpeaks=sig_peaks[::2]
    npeaks=sig_peaks[1::2]
    return [float(peaks[dpeaks[i]][1])/peaks[npeaks[i]][1]
        for i in xrange(len(dpeaks))]

def convert(values, threshold=0.9):
    """
    Converts the weighted values to a binary bit string
    according to the given threshold.

    Arguments:
    values -- list of values as returned from weigh_peaks
    threshold -- the threshold value,
                 which decides whether a value gets
                 converted to 0 or 1
    """
    return ''.join([item<threshold and '0' or '1'
        for item in values])

if __name__=='__main__':
    parser =\
        OptionParser(usage="usage: %prog [options] \
            path_to_measurements")
    parser.add_option("-d", "--peakdelta",
        dest="peak_delta",
        help="delta for peak detection",
        metavar="PEAKDELTA",
        type="int",
        default=25)
    parser.add_option("-p", "--prefix",
        dest="file_prefix",
        help="prefix of the filenames of the measurement",
        metavar="PREFIX",
        default="0")
    parser.add_option("-l", "--puflog",
        dest="puf_log",
        help="path to puf.log",
        metavar="PUFLOG",
        default="puf.log")
    parser.add_option("-s", "--startsample",
        dest="start_sample",
        help="at which sample to start in the files",
        metavar="START",
        type="int",
        default=0)
    parser.add_option("-n", "--numpoints",
        dest="num_of_points",
        help="how many significant points there are in hashing",
        metavar="POINTS",
        type="int",
        default=148)
    parser.add_option("-m", "--margin",
        dest="margin",
        help="margin for detection of significant peaks",
        metavar="MARGIN",
        type="int",
        default=50)
    parser.add_option("-i", "--interval",
        dest="med_dist",
        help="interval between significant peaks",
        metavar="INTERVALL",
        type="int",
        default=None)
    parser.add_option("-t", "--conv_threshold",
        dest="conv_threshold",
        help="threshold to convert weighted values to bits",
        metavar="TH",
        type="float",
        default=None)

    (options, args) = parser.parse_args()
    data = None

    # read possibly previously generated average trace
    if len(args)!=1:
        if os.path.exists('./tmp.dat'):
            data = np.fromfile('./tmp.dat', np.uint8)
        else:
            parser.error("path to data dir missing")


    # read in measurements, align them and
    # calculate the mean to eliminate noise
    if data is None:
        data_dir = args[0]
        data = read_aligned_mean(data_dir,
                 options.file_prefix,
                 options.start_sample)
        save_nparray(data, './tmp.dat')

    # extract important peaks
    # (higher than delta compared to vicinity)
    peak_delta = options.peak_delta
    while peak_delta!="":
        plt.plot(data, 'b-')
        peak_delta=float(peak_delta)
        print "extracting peaks with delta", peak_delta
        peaks,_ = get_peaks(data, peak_delta)
        plt.plot(*(zip(*peaks)),
            linestyle='None',
            marker='o',
            markerfacecolor='yellow')[0]
        plt.show()
        peak_delta = raw_input('new peak delta \
            (empty to continue):')

    # try to find the peaks significant for the hashing
    margin, med_dist = options.margin, options.med_dist
    if not med_dist:
        med_dist = np.median([peaks[i+1][0]-peaks[i][0]
            for i in xrange(len(peaks)-1)])
```

```
while margin != "" and med_dist != "":
    plt.plot(data, 'b-')
    margin, med_dist = float(margin), float(med_dist)
    print "filtering significant peaks with margin", \
        margin, "and approx. interval", med_dist
    sig_peaks = get_sig_peaks(peaks,
        options.num_of_points,
        margin, med_dist)
    plt.plot([peaks[item][0] for item in sig_peaks],
        [peaks[item][1] for item in sig_peaks], 'ro')
    plt.show()
    margin = raw_input('new margin (empty to continue):')
    med_dist = raw_input('new approx. interval \
        (empty to continue):')

# weigh the peaks to compensate for different
# power consumption
bit_values = weigh_peaks(peaks, sig_peaks)

# convert the weighted values to bit values
# according to a threshold
if not options.conv_threshold:
    options.conv_threshold = np.mean(bit_values)
calc_bitstr = convert(bit_values, options.conv_threshold)
print "supposed key\t\t", calc_bitstr

# print actual bit string, if debug log of PUF is given
# and calculate number of errors
logdata = None
if os.path.exists(options.puf_log):
    with open(options.puf_log) as f:
        logdata = f.read().split('\n')[2:]
        print "key from puf.log\t", logdata[2][:74]
if logdata:
    print "difference\t\t", \
        sum(ch1 != ch2
        for ch1, ch2 in zip(calc_bitstr, logdata[2][:74]))
```

A.2. Source Code of Helper Functions used in the Proof-Of-Concept

```
#!/usr/bin/env python

import os, glob, struct
import numpy as np

def read_aligned_mean(directory, prefix='0', start=0):
    """
    Read in measurement traces out of the given directory,
    align them according to a trigger peak and return
    an averaged trace.

    Arguments:
    directory -- the directory of the traces,
                 given as a string
    prefix -- the prefix of the filenames to read in
    start -- starting sample of from where to start searching
             for the trigger peak
    """
    l = []
    peak = 0
    files = sorted(glob.glob(os.path.join(directory, prefix+'*.dat')))
    for infile in files:
        print "reading", infile
        x = np.fromfile(infile, np.uint8)
        temp = np.ndarray.tolist(x)
        if not l:
            peak = temp[start:].index(255)+start
            l.append(x)
        else:
            try :
                curpeak = temp[start:].index(255)+start
            except:
                print "no peak found in", infile
                continue
            diff = peak-curpeak
            if diff <= 0:
                diff = -diff
                l.append(temp[diff:]+[0]*diff)
            else:
                l.append([0]*diff + temp[:-diff])
        if len(l[-1]) == 0:
            print l[-1], curpeak, diff

    return np.mean(np.array(l, np.uint8), axis=0)

def get_peaks(a, delta, x=None):
    """
    Returns a tuple of maximum peaks and minimum peaks found in
    a trace.

    Arguments:
    a -- array representing a trace
    delta -- delta value of when to recognize a point as a peak
    x -- alternative labeling of axis
    """
    maxtab = []
    mintab = []

    if x is None:
        x = np.arange(len(a))

    a = np.asarray(a)

    if len(a) != len(x):
        sys.exit('Input vectors a and x must have same length')

    if not np.isscalar(delta):
        sys.exit('Input argument delta must be a scalar')

    if delta <= 0:
        sys.exit('Input argument delta must be positive')

    mn, mx = np.Inf, -np.Inf
    mnpos, mxpos = np.NaN, np.NaN

    lookformax = True

    for i in np.arange(len(a)):
        this = a[i]
        if this > mx:
            mx = this
            mxpos = x[i]
        if this < mn:
            mn = this
            mnpos = x[i]

        if lookformax:
            if this < mx-delta:
                maxtab.append((mxpos, mx))
                mn = this
                mnpos = x[i]
                lookformax = False
        else:
            if this > mn+delta:
                mintab.append((mnpos, mn))
                mx = this
                mxpos = x[i]
                lookformax = True

    return maxtab, mintab

def save_nparray(a, fn):
    """
    Saves a numpy array in a file
    (binary uint8 without any header).

    Arguments:
    a -- array to be saved to disk
    fn -- filename for the saved array, given as a string
    """
    with open(fn,'w') as outfile:
        for item in a:
            outfile.write(struct.pack('B',item))
```

B. Glossary

BCH	Bose-Chaudhuri-Hochquenghen
CMOS	Complementary Metal Oxide Semiconductor
CRP	Challenge Response Pair
DPA	Differential Power Analysis
EM	Electro-Magnetic
FPGA	Field Programmable Gate Array
IBS	Index-Based Syndrome
IC	Integrated Circuit
IP	Intellectual Property
LFSR	Linear Feedback Shift Register
MITM	Man-In-The-Middle
PUF	Physical Unclonable Function
RO-PUF	Ring-Oscillator PUF
SPA	Simple Power Analysis
SRAM	Static Random Access Memory
SVM	Support Vector Machine