DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2018

On Statistical Properties of Arbiter Physical Unclonable Functions

PHILLIP GAJLAND

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

On Statistical Properties of Arbiter Physical Unclonable Functions

---

Phillip Gajland

A thesis submitted for the degree of

Bachelor of Science

Examiner: Prof. Dr. Elena Dubrova
Supervisor: Prof. Dr. Mark Smith
Mentor: Dr. Felipe Marranghello

School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology

Stockholm 2018

To Mum & Dad

Acknowledgements

I AM ETERNALLY GRATEFUL, for the continued support that I have received during the undertaking of this thesis. I owe my thanks to numerous individuals, without many of whom, this would not have been possible.

First off, this project would never have come about without the tremendous input from Prof. Dr. Elena Dubrova. In fact it is she, whom I owe credit to for introducing me to the concept of PUFs and drawing my attention to the problem at hand. I owe my thanks to Dr. Felipe Marranghello for his guidance and invaluable advice throughout the course of this project. I also wish to thank those individuals that have been kind enough to read multiple draft versions of the thesis.

Abstract

THE GROWING INTEREST IN THE INTERNET OF THINGS (IoT) has led to predictions claiming that by 2020 we can expect to be surrounded by 50 billion Internet connected devices. With more entry points to a network, adversaries can potentially use IoT devices as a stepping stone for attacking other devices connected to the network or the network itself. Information security relies on cryptographic primitives that, in turn, depend on secret keys. Furthermore, the issue of Intellectual property (IP) theft in the field of Integrated circuit (IC) design can be tackled with the help of unique device identifiers. Physical unclonable functions (PUFs) provide a tamper-resilient solution for secure key storage and fingerprinting hardware. PUFs use intrinsic manufacturing differences of ICs to assign unique identities to hardware. Arbiter PUFs utilise the differences in delays of identically designed paths, giving rise to an unpredictable response unique to a given IC.

This thesis explores the statistical properties of Boolean functions induced by arbiter PUFs. In particular, this empirical study looks into the distribution of induced functions. The data gathered shows that only $\sim 3\%$ of all possible 4-variable functions can be induced by a single 4 stage arbiter PUF. Furthermore, some individual functions are more than 5 times more likely than others. Hence, the distribution is non-uniform. We also evaluate alternate PUF designs, improving the coverage vastly, resulting in one particular implementation inducing all 65,536 4-variable functions. We hypothesise the need for n XORed PUFs to induce all 2^{2^n} possible n-variable Boolean functions.

Keywords: PUF, Boolean function, function distribution, arbiter path, digital fingerprint, secure key storage, cryptography, system security, hardware security.

Phillip Gajland (Stockholm, 2018)
On Statistical Properties of Arbiter Physical Unclonable Functions

Contents

List of Figures

List of Tables

List of acronyms

AES Advanced Encryption Standard.

CRP Challenge Response Pair.

FPGA Field-programmable gate array.

IC Integrated circuit.

IoT Internet of Things.

IP Intellectual property.

MAJ Majority Gate.

ML Machine Learning.

MUX Multiplexer.

PUF Physical Unclonable Function.

SRAM Static random-access memory.

XOR Exclusive or.

Chapter 1

Introduction

1.1 Motivation

Due to the growing interest in the Internet of Things (IoT), predictions claim that by 2020 we can expect to be surrounded by 50 billion Internet connected devices [Nor16]. By exposing more entry points to a network, adversaries can potentially use IoT devices as a stepping stone for targeting attacks on the rest of the network. This was successfully demonstrated during the 2016 *Dyn* cyberattack which exploited *Mirai* malware infected IoT devices to perform distributed denial-of-service attacks, causing major internet platforms to be unavailable. Hence, there is an increasing need for providing a secure infrastructure on which the IoT is built, starting with the devices themselves. In the case of communication; confidentiality and authenticity lie at the heart of information security. Whilst encryption provides secure channels through which chips can communicate, authentication guarantees the identity of the communicating partner as well as integrity of data. The underlying cryptographic algorithms rely on secure key storage.

Moreover, globalisation has led to vendors outsourcing the production of Integrated circuits (ICs) to sites located across the globe. Manufacturers may be tempted into producing more than the ordered amount, and selling the surplus illegally [BH13]. In order to combat this, unique device identifiers can be used to verify legitimate chips.

Considering their high entropy, low cost and tamper resistance; PUFs provide an attractive solution to both fingerprinting hardware and secure key storage. Therefore, it is important to be aware of the strengths and weaknesses of PUFs. This thesis addresses the statistical properties of arbiter PUFs.

1.2 Problem

Since their proposal in 2002, PUFs have been used for secure key storage as well as unique identifiers [PRTG02]. However, little is known about the statistical properties of arbiter PUFs. Only by studying these properties, can we ensure the security of PUFs and the depending cryptographic primitives.

The Boolean functions that are induced by PUFs should ideally be uniformly distributed, in order to minimise the risk of attacks on the PUF. Adversaries with knowledge of the non-uniform distribution of functions could potentially use this bias to perform targeted attacks on PUFs, compromising the security of ICs and IoT devices.

1.3 Purpose

Unique device identifiers and secure key storage are only as reliable as the building blocks that they are built on. PUFs can be used to minimise the damage caused by Intellectual property (IP) theft. Vendors contracting manufacturers overseas can integrate PUFs into their chips to ensure that:

- Only the ordered number of chips are produced and sold. (PUFs act as unique device identifiers and vendors publish a whitelist of ID numbers.)
- Designs are loaded onto the chip without being eavesdropped. (PUFs are used for secure key storage and bit stream encryption. However, Trimberger et al. argue that there may be some drawbacks in using PUFs as decryption keys due to their instability and the fact that bitstreams would have to be encrypted uniquely for each chip [TM14].)

Two common alternatives for FPGA key storage are battery backed SRAM and eFuses, with SRAM based key storage being prone to data remanence attacks and fuse based storage being prone to optical inspection [KK99]. As mentioned, maintaining the integrity of encryption keys can be achieved using PUFs and is currently done in the Xilinx Zynq Ultrascale+ and Altera Stratix 10 FPGAs [XI17] [LKA].

1.4 Goal

This thesis aims to study the distribution of Boolean functions that are induced by arbiter PUFs. The intention is to quantify how the distribution of functions differs from a uniform distribution, depending on the structure of PUFs. Furthermore, we wish to investigate multiple improved arrangements of arbiter PUFs along with their evaluation, ensuring increased security.

1.5 Methodology

The presented results have for the most part been formed from empirical data. The simulation on which the study is based, was originally designed by by Prof. Dr. Elena Dubrova. Statistical evaluations have been made on the outcome of multiple trials, in some cases as many as 2 billion [See Figure 4.21]. All experiments were run using a bash shell script and gnuplot for interpreting the data graphically [See Appendix A]. The figures were made using the `draw.io` tool.

1.6 Delimitation

Due to the fact that the number of n-variable Boolean functions is super exponential, namely 2^{2^n}, the study has been limited up to 4-variable functions. Interesting patterns along with the presented hypothesis could be further studied by looking into functions of a larger number of variables. Furthermore, the thesis primarily looks into linear combinations of PUFs using XOR gates. Designs implementing majority gates and other non-linear functions are left for future work.

1.7 Outline

The following chapter gives a brief introduction to PUFs as well as Boolean functions. Furthermore, arbiter PUFs are introduced along with comparable previous studies. Chapter 3 addresses arbiter PUFs in more detail, starting with an example of an XOR function induced by an arbiter PUF. The conditions needed to induce a certain function as well as a proof as to why conflicts occur are presented towards the end of the chapter. Chapter 4 covers an outline of the results from three different experiments as well as a summary. Chapter 5 concludes the thesis. Parts of the source code can be found in the Appendices as well as in the following GitHub repository: `https://github.com/GaPhil/arbiter-pufs-code` [The authors reserve the right to grant access to the repository].

Chapter 2

Background

2.1 Boolean Functions

"Boolean functions are functions of type $f : B^n \to B$, where $B = \{0, 1\}$ is a Boolean domain and n is a non-negative integer called the arity of the function. In the case where $n = 0$, the function is a constant element of B. Every $n-ary$ Boolean function can be expressed as a propositional formula in n variables $x_1 \ldots x_n$, and two propositional formulas are logically equivalent if and only if they express the same Boolean function." [Mor03]

There are 2^{2^n} different n-variable Boolean functions, as shown in Table 2.1.

Table 2.1: There are 2^{2^n} different n-variable Boolean functions.

No. of variables (n)	Number of different functions (f)
1	4 $(0, 1, x, \overline{x})$
2	16 $(0, 1, x_1, x_2, \overline{x_1}, \overline{x_2}, x_1 \oplus x_2$, etc)
3	256 $(0, 1, x_1, x_3, \overline{x_1}, \overline{x_2}, x_2 \oplus x_3$, etc)
4	65,536 $(0, 1, x_1, x_4, \overline{x_1}, \overline{x_2}, x_3 \oplus x_4$, etc)
$\vdots$	$\vdots$
n	2^{2^n} $(0, 1, x_1, x_n, \overline{x_1}, \overline{x_2}, x_3 \oplus x_n$, etc)

Assuming a uniform distribution, the probability of obtaining a particular function is:

$$\frac{1}{2^{2^n}}$$

The probability of obtaining either constant 1 or constant 0 as a function with n variables is:

$$\frac{2}{2^{2^n}}$$

For n variables, the number of functions that do not depend on **any specific** variable where $x_i, i \in 1, ..., n$ is given by:

$$2^{2^n} - 2^{2^{n-1}}$$

2.2 Physical Unclonable Functions

A Physical Unclonable Function (PUF) is a *digital fingerprint* that acts as a unique identifier for a semiconductor device such as an FPGA or a microprocessor. The manufacturing process of semiconductors gives rise to physical variations, making it possible to differentiate between otherwise identical ICs. PUFs are based on these intrinsic manufacturing differences providing a mapping between *challenges* and *responses*, where challenges are applied to the PUF and responses are obtained as an output. A given challenge should give the same response in a particular PUF; hence the term *function* is used. The Challenge Response Pair (CRP) can be evaluated in the form of a Boolean function, which in turn acts as a form of cryptographic key. The term *unclonable* refers to the fact that identical PUFs can not be reproduced. Due to the sheer scale of PUFs and the fact that they are embedded in the IC, tampering changes the challenge-response behaviour.

2.3 Abiter PUFs

There are various different types of PUFs, such as delay PUFs, SRAM PUFs and optical PUFs [MV10] [AMS+09]. However, this thesis is based specifically on Arbiter PUFs, a form of delay PUF that exploits delays in wires and logic gates. This is because the statistical properties result from the restrictions imposed by the delays in a PUF. These manufacturing differences give rise to a so-called race condition between two identical paths. Arbiter PUFs are implemented using multiple switch blocks, each consisting of two multiplexers that decide which path is taken, given a challenge [See Figure 2.1]. The arbiter circuit is usually implemented as a latch or flip-flop [MV10].

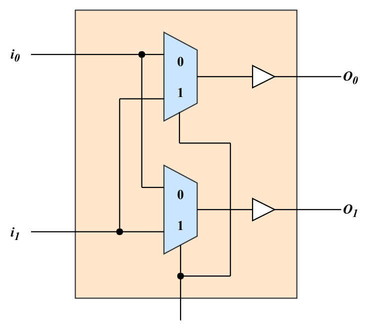
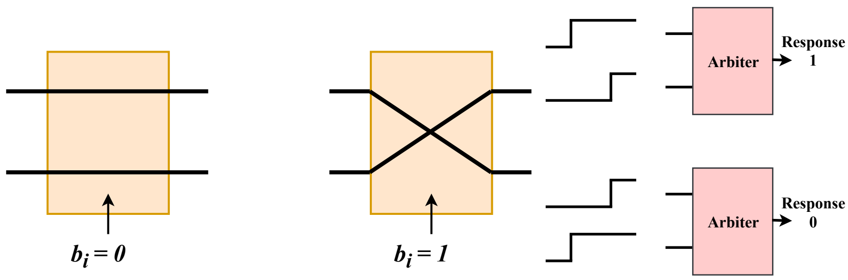


Figure 2.1: Schematic of a switch block

Challenges are applied to the switch blocks, creating a race condition between two paths. Signals travelling through the switch blocks will reach the Arbiter at slightly different times leading to a 0/1 response. If the top signal arrives first then the response will be 0 [See Figure 2.2].



(a) Switch block configuration **(b)** Arbiter operations

Figure 2.2: PUF operations

Multiple switch blocks are connected in series along with an arbiter to form a PUF [See Figure 2.3]. In the case of n stages, a n-bit challenge is applied to the PUF giving rise to a *unique* response in every PUF [Bec15].

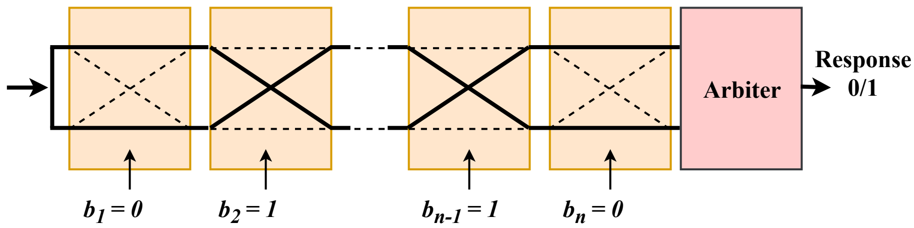


Figure 2.3: Multiple switch blocks in series form a PUF

The relation between challenges and responses is usually referred to as a Challenge Response Pair (CRP). The random output relies on minute manufacturing differences making each chip unique and virtually impossible to reproduce. Hardware authentication schemes, such as those proposed by Abadie et al. can make use of CRPs [AAB+16]. Section 3.2 addresses the Boolean functions mapped by CRPs.

2.4 Machine Learning Attacks

The chapter entitled *"Physically Unclonable Functions: a Study on the State of the Art and Future Research Directions"* in the book *Towards Hardware-Intrinsic Security* provides an informative introduction to Machine Learning (ML) attacks [MV10] [AAB+16]. The authors stress the fact that arbiter PUFs are additive by nature i.e. the delay of a series of switch blocks is the sum of the delays of the individual switch blocks. Using this observation, *model building* *attacks* have achieved prediction errors of 3.55% after observing 5000 CRPs for the ASIC implementation, and a prediction error of 0.6% after observing 90000 CRPs for the FPGA implementation. [MV10]

2.5 Stability

"The lack of stability is one of the limitations that constrains PUFs from being put in widespread practical use."[WYDG18] The paper entitled *"Secure and Reliable XOR Arbiter PUF Design: An Experimental Study based on 1 Trillion Challenge Response Pair Measurements"* addresses the issue of improving PUF security at the expense of reduced stability. The study specifically focuses on arbiter PUFs and shows that an XOR arbiter PUF design is less susceptible to Machine Learning attacks than a single MUX PUF. However, the stability of the responses is reduced significantly [AAB+16].

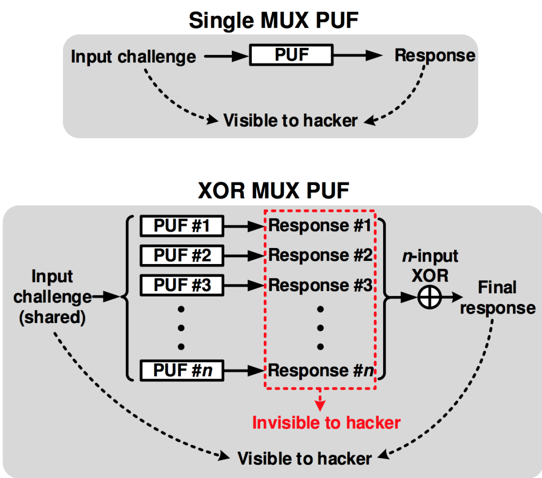


Figure 2.4: XOR MUX PUF [AAB+16]

Chapter 3

Analysis

...of Arbiter PUFs

In this chapter we introduce a simple example of an Arbiter PUF and the Boolean function induced by it [See Section 3.1]. Furthermore we present an overview of the conditions needed for certain functions to be induced [See Table 3.3 and Table 3.4]. We also show a formal proof that certain functions are not induced by single arbiter PUFs.

3.1 Arbiter PUF Example

Consider a single 2-stage arbiter PUF as shown in Figure 3.1 and suppose the stages have the following delays;

$$\begin{array}{llll} d_{11} = 1.1 \text{ ns} & d_{13} = 1.0 \text{ ns} & d_{21} = 1.2 \text{ ns} & d_{23} = 0.8 \text{ ns} \\ d_{12} = 1.3 \text{ ns} & d_{14} = 1.5 \text{ ns} & d_{22} = 1.4 \text{ ns} & d_{24} = 0.9 \text{ ns} \end{array}$$

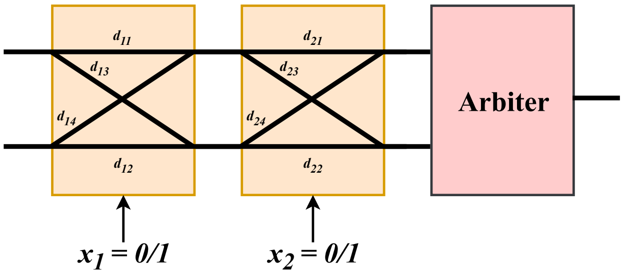


Figure 3.1: Delay Paths

x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	0

Figure 3.2: Truth table

Then the Boolean function induced by the PUF can be evaluated as follows:

$(x_1, x_2) = (0,0)$: $d_{11} + d_{21} < d_{12} + d_{22}$ — The arbiter generates 0 because the upper path is faster than the lower path.

$(x_1, x_2) = (0,1)$: $d_{12} + d_{24} > d_{11} + d_{23}$ — The arbiter generates 1 because the upper path is slower than the lower path.

$(x_1, x_2) = (1,0)$: $d_{14} + d_{21} > d_{13} + d_{22}$ — The arbiter generates 1 because the upper path is slower than the lower path.

$(x_1, x_2) = (1,1)$: $d_{13} + d_{24} < d_{14} + d_{23}$ — The arbiter generates 0 because the upper path is faster than the lower path.

A truth table can be drawn-up as shown in Figure 3.2. The Boolean function induced by the PUF is $f(x_1, x_2) = x_1 \oplus x_2$, where "$\oplus$" denotes XOR.

3.2 PUF Induced Boolean Functions

A truth table can be drawn, displaying the conditions needed for a PUF to induce a particular function, where $d_{11}...d_{14}$ denote the respective delays. [See Table 3.1] In some cases it may be preferable to use delay differences, denoted by Δ, instead of the absolute values [See Table 3.2]. Hence, we adopt the following notation:

$$\Delta_{mj-nk} = d_{mj} - d_{nk}$$

Table 3.1: Truth table showing the 4 Boolean functions induced by a single arbiter PUF with one switch block.

	Challenge		
	$x_1 = 0$	$x_1 = 1$	$f(x_1)$
00	$d_{11} < d_{12}$	$d_{13} > d_{14}$	0
01	$d_{11} > d_{12}$	$d_{13} < d_{14}$	1
10	$d_{11} < d_{12}$	$d_{13} < d_{14}$	x_1
11	$d_{11} > d_{12}$	$d_{13} > d_{14}$	$\overline{x_1}$

Table 3.2: Truth table showing the 4 Boolean functions induced by a single arbiter PUF with one switch block, using delay differences as opposed to absolute values.

	Challenge		
	$x_1 = 0$	$x_1 = 1$	$f(x_1)$
00	$\Delta_{11-12} < 0$	$\Delta_{13-14} > 0$	0
01	$\Delta_{11-12} > 0$	$\Delta_{13-14} < 0$	1
10	$\Delta_{11-12} < 0$	$\Delta_{13-14} < 0$	x_1
11	$\Delta_{11-12} > 0$	$\Delta_{13-14} > 0$	$\overline{x_1}$

3.3 Proof of Conflict

Table 3.3 and Table 3.4 show the conditions required for certain Boolean functions to be induced by a single arbiter PUF with two switch blocks.

In order for $f(x_1, x_2) = x_1$ to be induced by a single arbiter PUF, the following system of inequalities must hold:

$$\begin{cases} d_{11} + d_{21} < d_{12} + d_{22} \\ d_{14} + d_{21} > d_{13} + d_{22} \\ d_{12} + d_{24} < d_{11} + d_{23} \\ d_{13} + d_{24} > d_{14} + d_{23} \end{cases}$$

This can be rewritten using delay differences as follows:

$$\begin{cases} \quad \Delta_{11-12} < -\Delta_{21-22} \\ \quad \Delta_{21-22} > \Delta_{13-14} \\ - \quad \Delta_{11-12} < \Delta_{23-24} \\ \quad \Delta_{13-14} > \Delta_{23-24} \end{cases}$$

Using $-\Delta_{11-12} < \Delta_{23-24}$ and $\Delta_{13-14} > \Delta_{23-24}$ the following inequality can be constructed:

$$-\Delta_{11-12} < \Delta_{23-24} < \Delta_{13-14}$$

Hence,

$$-\Delta_{11-12} < \Delta_{13-14}$$

Using $\Delta_{11-12} < -\Delta_{21-22}$ and $\Delta_{21-22} > \Delta_{13-14}$ the following inequality can be constructed:

$$\Delta_{13-14} < \Delta_{21-22} < -\Delta_{11-12}$$

Hence,

$$\Delta_{13-14} < -\Delta_{11-12}$$

Therefore

$$-\Delta_{11-12} < \Delta_{13-14} < -\Delta_{11-12} \tag{3.1}$$

Equation 3.1 shows a conflict for $f(x_1, x_2) = x_1$. Hence, the function cannot be induced. A similar proof can be used to show why certain functions are not induced by n-variable arbiter PUFs.

However, $f(x_1, x_2) = x_1 \oplus x_2$ and $f(x_1, x_2) = x_2$ are two functions that can be induced according to Table 3.3 and Table 3.4. Hence, when these two functions are XORed $f(x_1, x_2) = x_1$ can be obtained.

$$(x_1 \oplus x_2) \oplus x_2 = x_1 \oplus x_2 \oplus x_1 = x_1$$

Similar reasoning can be applied to other functions and is the theory behind why the experiments presented in this thesis work.

3.4 Logic Simulation of Arbiter PUFs

The results shown in Chapter 4 are based on data gathered from simulating the delays of arbiter PUFs. The simulation can be explained as follows:

1. Select n and number of trials
2. For each trial:
   (a) Assign random values from Gaussian Distribution to the delays.
   (b) Evaluate the resulting truth table.

For more detailed information regarding the simulations see Appendix B.

Table 3.3: Truth table showing the 16 Boolean functions implemented by an arbiter PUF with 2 switch blocks. x_1 and $\overline{x_1}$ cannot be induced.

	Challenge				
	$x_2x_1 = 00$	$x_2x_1 = 01$	$x_2x_1 = 10$	$x_2x_1 = 11$	$f(x_1, x_2)$
0000	$d_{11} + d_{21} < d_{12} + d_{22}$	$d_{14} + d_{21} < d_{13} + d_{22}$	$d_{12} + d_{24} < d_{11} + d_{23}$	$d_{13} + d_{24} < d_{14} + d_{23}$	0
0001	$d_{11} + d_{21} < d_{12} + d_{22}$	$d_{14} + d_{21} < d_{13} + d_{22}$	$d_{12} + d_{24} < d_{11} + d_{23}$	$d_{13} + d_{24} > d_{14} + d_{23}$	x_1x_2
0010	$d_{11} + d_{21} < d_{12} + d_{22}$	$d_{14} + d_{21} < d_{13} + d_{22}$	$d_{12} + d_{24} > d_{11} + d_{23}$	$d_{13} + d_{24} < d_{14} + d_{23}$	$\overline{x_1}x_2$
0011	$d_{11} + d_{21} < d_{12} + d_{22}$	$d_{14} + d_{21} < d_{13} + d_{22}$	$d_{12} + d_{24} > d_{11} + d_{23}$	$d_{13} + d_{24} > d_{14} + d_{23}$	x_2
0100	$d_{11} + d_{21} < d_{12} + d_{22}$	$d_{14} + d_{21} > d_{13} + d_{22}$	$d_{12} + d_{24} < d_{11} + d_{23}$	$d_{13} + d_{24} < d_{14} + d_{23}$	$x_1\overline{x_2}$
0101	$d_{11} + d_{21} < d_{12} + d_{22}$	$d_{14} + d_{21} > d_{13} + d_{22}$	$d_{12} + d_{24} < d_{11} + d_{23}$	$d_{13} + d_{24} > d_{14} + d_{23}$	x_1
0110	$d_{11} + d_{21} < d_{12} + d_{22}$	$d_{14} + d_{21} > d_{13} + d_{22}$	$d_{12} + d_{24} > d_{11} + d_{23}$	$d_{13} + d_{24} < d_{14} + d_{23}$	$x_1 \oplus x_2$
0111	$d_{11} + d_{21} < d_{12} + d_{22}$	$d_{14} + d_{21} > d_{13} + d_{22}$	$d_{12} + d_{24} > d_{11} + d_{23}$	$d_{13} + d_{24} > d_{14} + d_{23}$	$x_1 + x_2$
1000	$d_{11} + d_{21} > d_{12} + d_{22}$	$d_{14} + d_{21} < d_{13} + d_{22}$	$d_{12} + d_{24} < d_{11} + d_{23}$	$d_{13} + d_{24} < d_{14} + d_{23}$	$\overline{x_1 + x_2}$
1001	$d_{11} + d_{21} > d_{12} + d_{22}$	$d_{14} + d_{21} < d_{13} + d_{22}$	$d_{12} + d_{24} < d_{11} + d_{23}$	$d_{13} + d_{24} > d_{14} + d_{23}$	$\overline{x_1 \oplus x_2}$
1010	$d_{11} + d_{21} > d_{12} + d_{22}$	$d_{14} + d_{21} < d_{13} + d_{22}$	$d_{12} + d_{24} > d_{11} + d_{23}$	$d_{13} + d_{24} < d_{14} + d_{23}$	$\overline{x_1}$
1011	$d_{11} + d_{21} > d_{12} + d_{22}$	$d_{14} + d_{21} < d_{13} + d_{22}$	$d_{12} + d_{24} > d_{11} + d_{23}$	$d_{13} + d_{24} > d_{14} + d_{23}$	$\overline{x_1} + x_2$
1100	$d_{11} + d_{21} > d_{12} + d_{22}$	$d_{14} + d_{21} > d_{13} + d_{22}$	$d_{12} + d_{24} < d_{11} + d_{23}$	$d_{13} + d_{24} < d_{14} + d_{23}$	$\overline{x_2}$
1101	$d_{11} + d_{21} > d_{12} + d_{22}$	$d_{14} + d_{21} > d_{13} + d_{22}$	$d_{12} + d_{24} < d_{11} + d_{23}$	$d_{13} + d_{24} > d_{14} + d_{23}$	$x_1 + \overline{x_2}$
1110	$d_{11} + d_{21} > d_{12} + d_{22}$	$d_{14} + d_{21} > d_{13} + d_{22}$	$d_{12} + d_{24} > d_{11} + d_{23}$	$d_{13} + d_{24} < d_{14} + d_{23}$	$\overline{x_1x_2}$
1111	$d_{11} + d_{21} > d_{12} + d_{22}$	$d_{14} + d_{21} > d_{13} + d_{22}$	$d_{12} + d_{24} > d_{11} + d_{23}$	$d_{13} + d_{24} > d_{14} + d_{23}$	1

Table 3.4: Truth table showing the 16 Boolean functions implemented by an arbiter PUF with 2 switch blocks, using deltas. x_1 and $\overline{x_1}$ can not be induced.

	Challenge				
	$x_2x_1 = 00$	$x_2x_1 = 01$	$x_2x_1 = 10$	$x_2x_1 = 11$	$f(x_1, x_2)$
0000	$\Delta_{11-12} < -\Delta_{21-22}$	$\Delta_{21-22} < \Delta_{13-14}$	$-\Delta_{11-12} < \Delta_{23-24}$	$\Delta_{13-14} < \Delta_{23-24}$	0
0001	$\Delta_{11-12} < -\Delta_{21-22}$	$\Delta_{21-22} < \Delta_{13-14}$	$-\Delta_{11-12} < \Delta_{23-24}$	$\Delta_{13-14} > \Delta_{23-24}$	x_1x_2
0010	$\Delta_{11-12} < -\Delta_{21-22}$	$\Delta_{21-22} < \Delta_{13-14}$	$-\Delta_{11-12} > \Delta_{23-24}$	$\Delta_{13-14} < \Delta_{23-24}$	$\overline{x_1}x_2$
0011	$\Delta_{11-12} < -\Delta_{21-22}$	$\Delta_{21-22} < \Delta_{13-14}$	$-\Delta_{11-12} > \Delta_{23-24}$	$\Delta_{13-14} > \Delta_{23-24}$	x_2
0100	$\Delta_{11-12} < -\Delta_{21-22}$	$\Delta_{21-22} > \Delta_{13-14}$	$-\Delta_{11-12} < \Delta_{23-24}$	$\Delta_{13-14} < \Delta_{23-24}$	$x_1\overline{x_2}$
0101	$\Delta_{11-12} < -\Delta_{21-22}$	$\Delta_{21-22} > \Delta_{13-14}$	$-\Delta_{11-12} < \Delta_{23-24}$	$\Delta_{13-14} > \Delta_{23-24}$	x_1
0110	$\Delta_{11-12} < -\Delta_{21-22}$	$\Delta_{21-22} > \Delta_{13-14}$	$-\Delta_{11-12} > \Delta_{23-24}$	$\Delta_{13-14} < \Delta_{23-24}$	$x_1 \oplus x_2$
0111	$\Delta_{11-12} < -\Delta_{21-22}$	$\Delta_{21-22} > \Delta_{13-14}$	$-\Delta_{11-12} > \Delta_{23-24}$	$\Delta_{13-14} > \Delta_{23-24}$	$x_1 + x_2$
1000	$\Delta_{11-12} > -\Delta_{21-22}$	$\Delta_{21-22} < \Delta_{13-14}$	$-\Delta_{11-12} < \Delta_{23-24}$	$\Delta_{13-14} < \Delta_{23-24}$	$\overline{x_1 + x_2}$
1001	$\Delta_{11-12} > -\Delta_{21-22}$	$\Delta_{21-22} < \Delta_{13-14}$	$-\Delta_{11-12} < \Delta_{23-24}$	$\Delta_{13-14} > \Delta_{23-24}$	$\overline{x_1 \oplus x_2}$
1010	$\Delta_{11-12} > -\Delta_{21-22}$	$\Delta_{21-22} < \Delta_{13-14}$	$-\Delta_{11-12} > \Delta_{23-24}$	$\Delta_{13-14} < \Delta_{23-24}$	$\overline{x_1}$
1011	$\Delta_{11-12} > -\Delta_{21-22}$	$\Delta_{21-22} < \Delta_{13-14}$	$-\Delta_{11-12} > \Delta_{23-24}$	$\Delta_{13-14} > \Delta_{23-24}$	$\overline{x_1} + x_2$
1100	$\Delta_{11-12} > -\Delta_{21-22}$	$\Delta_{21-22} > \Delta_{13-14}$	$-\Delta_{11-12} < \Delta_{23-24}$	$\Delta_{13-14} < \Delta_{23-24}$	$\overline{x_2}$
1101	$\Delta_{11-12} > -\Delta_{21-22}$	$\Delta_{21-22} > \Delta_{13-14}$	$-\Delta_{11-12} < \Delta_{23-24}$	$\Delta_{13-14} > \Delta_{23-24}$	$x_1 + \overline{x_2}$
1110	$\Delta_{11-12} > -\Delta_{21-22}$	$\Delta_{21-22} > \Delta_{13-14}$	$-\Delta_{11-12} > \Delta_{23-24}$	$\Delta_{13-14} < \Delta_{23-24}$	$\overline{x_1x_2}$
1111	$\Delta_{11-12} > -\Delta_{21-22}$	$\Delta_{21-22} > \Delta_{13-14}$	$-\Delta_{11-12} > \Delta_{23-24}$	$\Delta_{13-14} > \Delta_{23-24}$	1

Chapter 4

Results

In this chapter the results from three different classes of simulations are presented. The experiments differ in the way that the PUFs are arranged. First off a single PUF is evaluated. In Section 4.2 the output of two PUFs has been XORed. Finally, Section 4.3 presents three XORed PUFs. For each of the following experiments, the distribution is analysed followed by the coverage probability.

4.1 Single Arbiter PUF

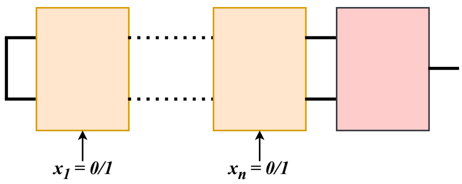


Figure 4.1: Setup for a single arbiter PUF

In Section 4.1 we assume the schematic as given in Figure 4.1, where one single PUF is evaluated for n different variables. The results show that the four 1-variable functions induced are uniformly distributed. The x-axis displays the function number and the y-axis represents the number of occurrences of the particular functions [See Figure 4.2]. However, in the case of 2-variable functions, two functions are not induced, namely x_1 and $\overline{x_1}$ [See Figure 4.3]. Furthermore, x_2 and $\overline{x_2}$ are twice as likely as four of the other functions and four times more likely than the others. For 3-variable functions less than half of the possible functions are induced and we see drastic spikes in four functions [See Figure 4.4]. In the case of 4-variable functions only a minute number $\sim 3\%$ of the functions can be induced [See Figure 4.5].

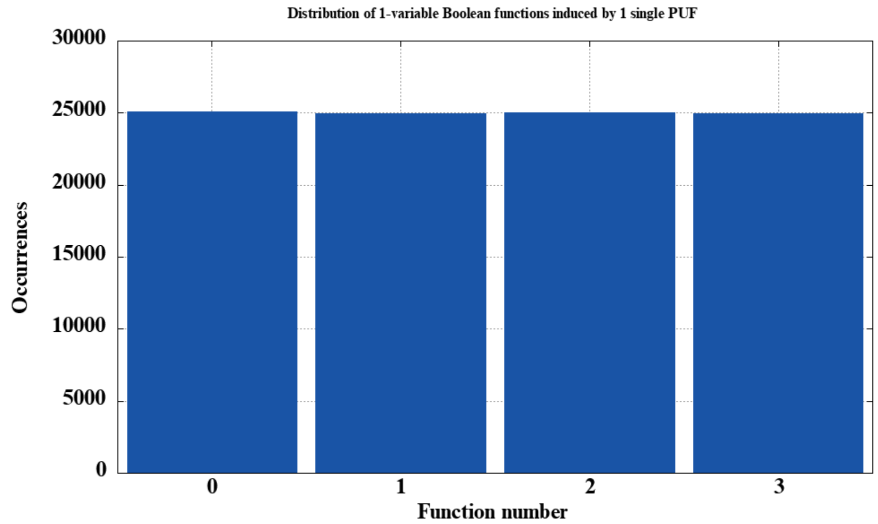


Figure 4.2: Distribution of 1-variable Boolean functions induced by a single 1-stage Arbiter PUF (100,000 trials). All functions are equally probable.

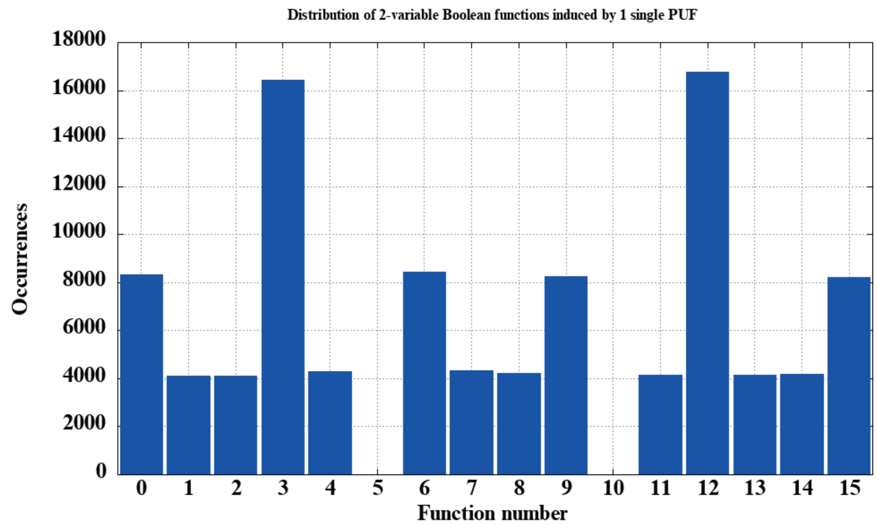


Figure 4.3: Distribution of 2-variable Boolean functions induced by a single 2-stage Arbiter PUF (100,000 trials). x_1 and $\overline{x_1}$ are not induced and x_2 and $\overline{x_2}$ are more likely.

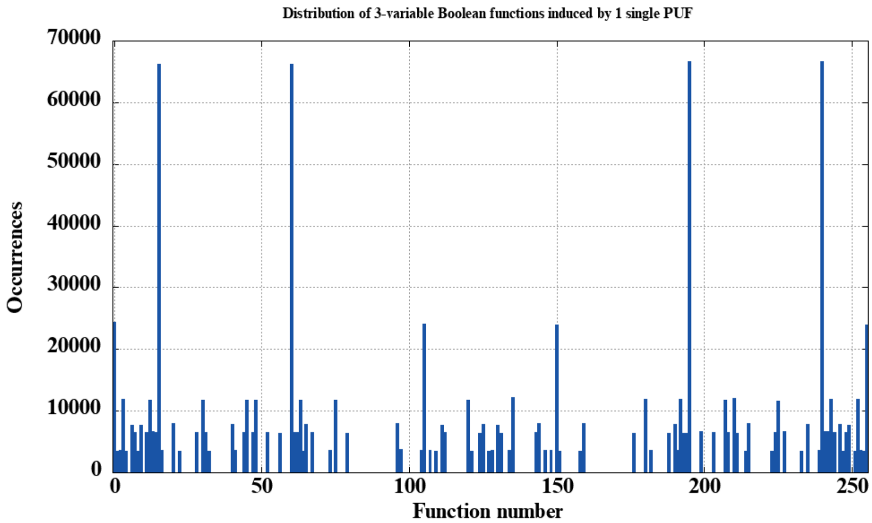


Figure 4.4: Distribution of 3-variable Boolean functions induced by a single 3-stage Arbiter PUF (1 million trials). 152 functions are not induced.

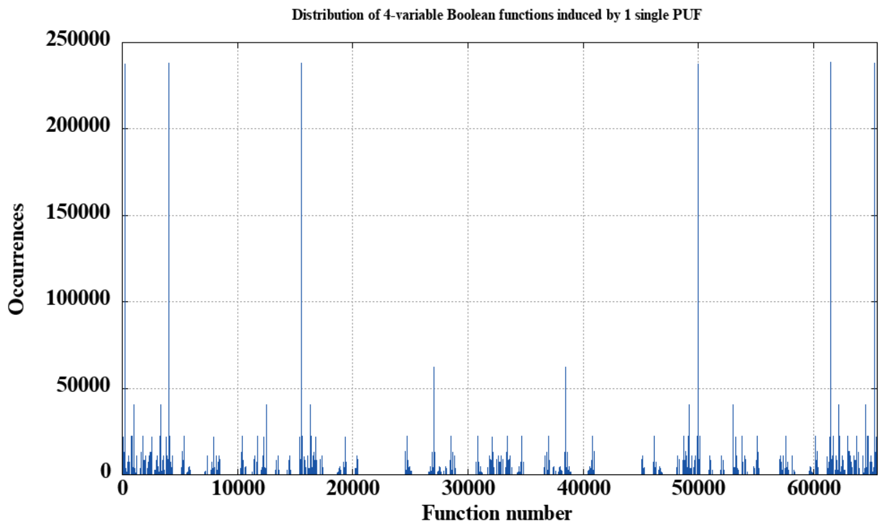


Figure 4.5: Distribution of 4-variable Boolean functions induced by a single 4-stage Arbiter PUF (10 million trials). 63,654 functions are not induced.

4.1.1 Coverage

In this section the functions are ordered from most likely to least likely. Furthermore, the x-axis does not represent the truth table but rather the function count. Since the 1-variable functions are uniformly distributed, there is little gain in studying their coverage. However, in the case of 2-variable functions, 80% coverage is obtained by 10 functions [See Figure 4.6]. Furthermore, for 3-variable functions, 80% coverage is given by 59 functions, which in turn represents only 23% of all possible 3-variable Boolean functions [See Figure 4.7]. Less than 1 percent of all possible 4-variable Boolean functions cover 80% of functions induced by a single 4-stage arbiter PUF, namely 887 [See Figure 4.8].

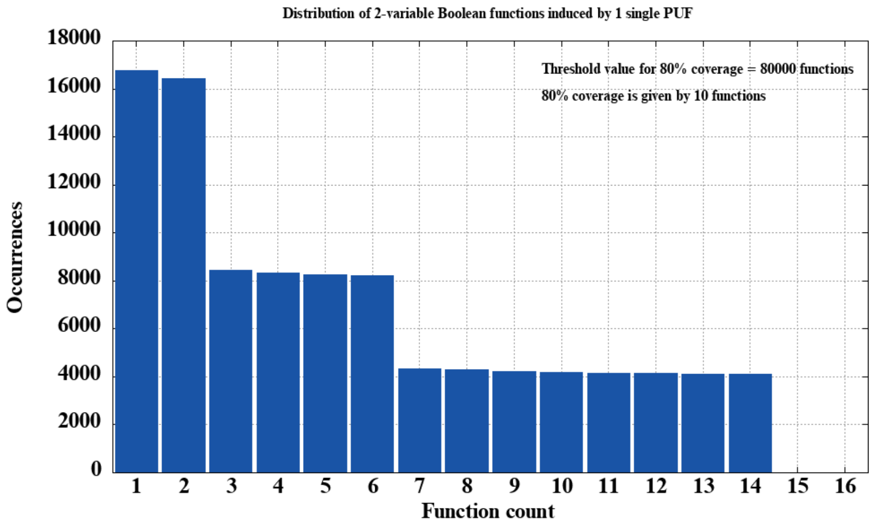


Figure 4.6: Distribution of 2-variable Boolean functions induced by a single 3-stage Arbiter PUF. 80% coverage is given by 10 functions.

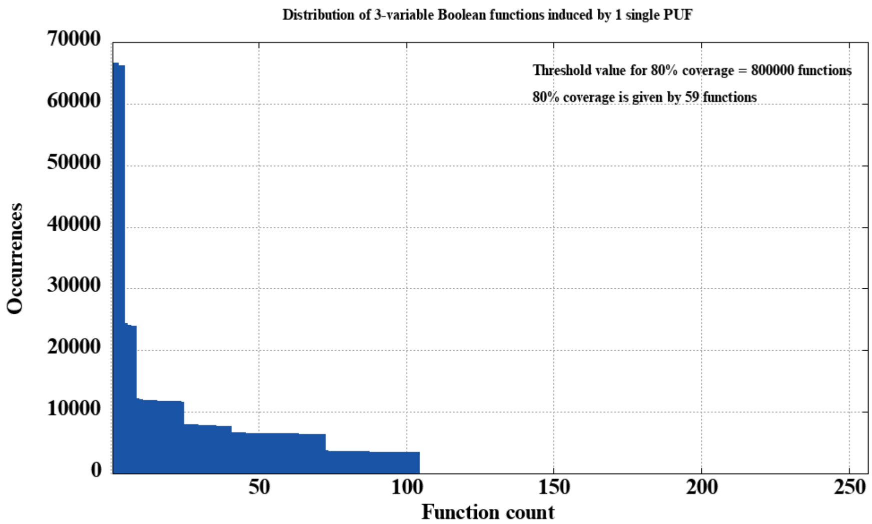


Figure 4.7: Distribution of 3-variable Boolean functions induced by a single 3-stage Arbiter PUF. 80% coverage is given by 59 functions.

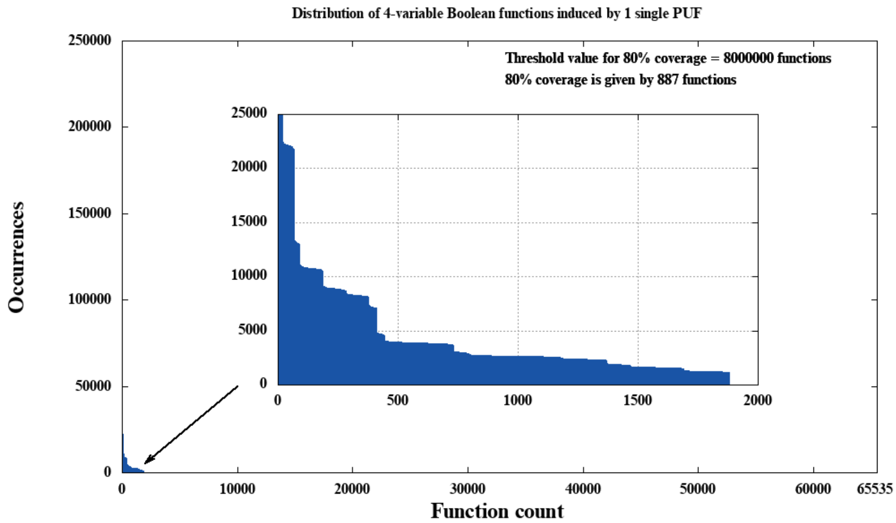


Figure 4.8: Distribution of 4-variable Boolean functions induced by a single 4-stage Arbiter PUF. 80% coverage is given by 887 functions.

4.2 Two XORed Arbiter PUFs

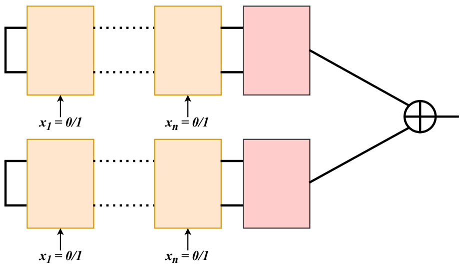


Figure 4.9: Setup for two XORed arbiter PUFs

In this section we assume the schematic as given in Figure 4.9, where the output of 2 PUFs are XORed and evaluated for n different variables. The challenge bits $x_1, ..., x_n$ are the same for each PUF. The distribution of 1-variable functions remains unchanged. However, the two 2-variable functions, x_2 and $\overline{x_2}$ that were previously not induced are now induced. Furthermore, only two 3-variable functions do not occur using this setup, namely $x_1 \oplus x_3$ and $\overline{x_1 \oplus x_3}$. It is also interesting to note the spikes in constant 1 and constant 0 function. Finally, we note a drastic improvement in the number of 4-variable functions that are now induced; almost 18% of all possible Boolean functions.

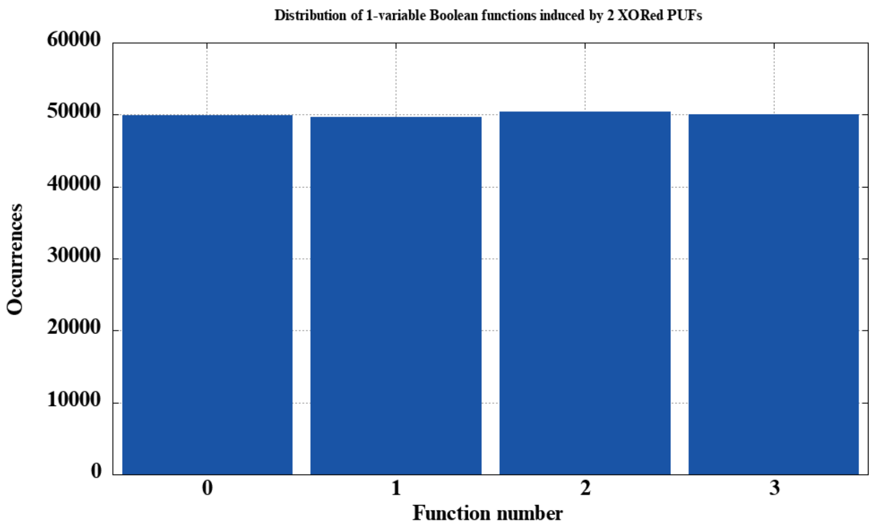


Figure 4.10: Distribution of 1-variable Boolean functions induced by two XORed 1-stage Arbiter PUFs (100,000 trials). All functions are equally probable.

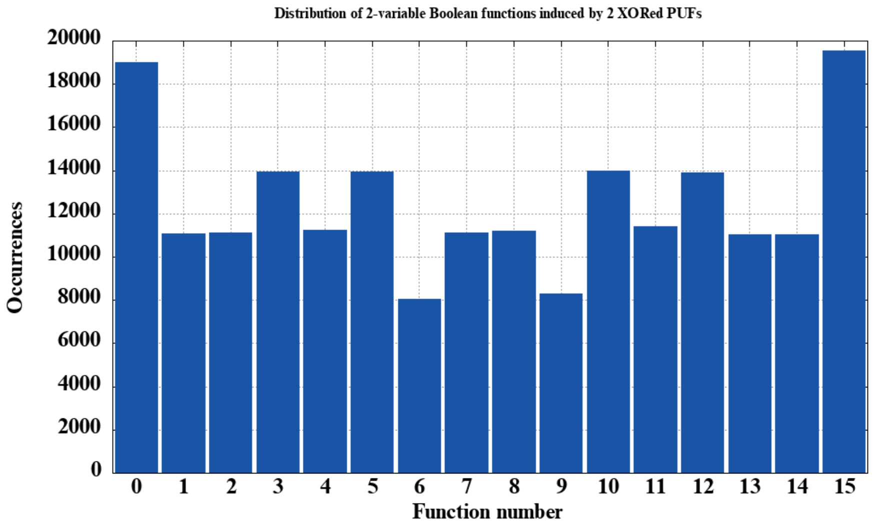


Figure 4.11: Distribution of 2-variable Boolean functions induced by two XORed 2-stage Arbiter PUFs (100,000 trials). All functions are induced.

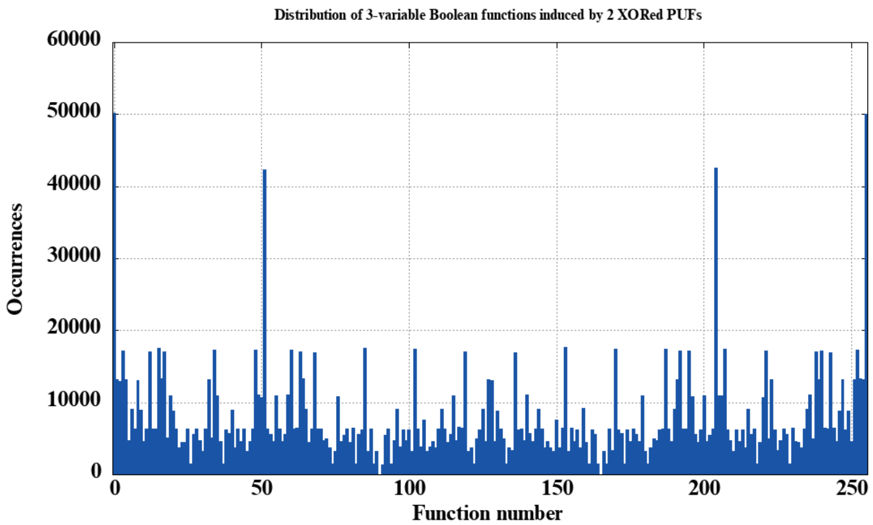


Figure 4.12: Distribution of 3-variable Boolean functions induced by two XORed 3-stage Arbiter PUFs (1 million trials). All functions apart from $x_1 \oplus x_3$ and $\overline{x_1 \oplus x_3}$ are induced.

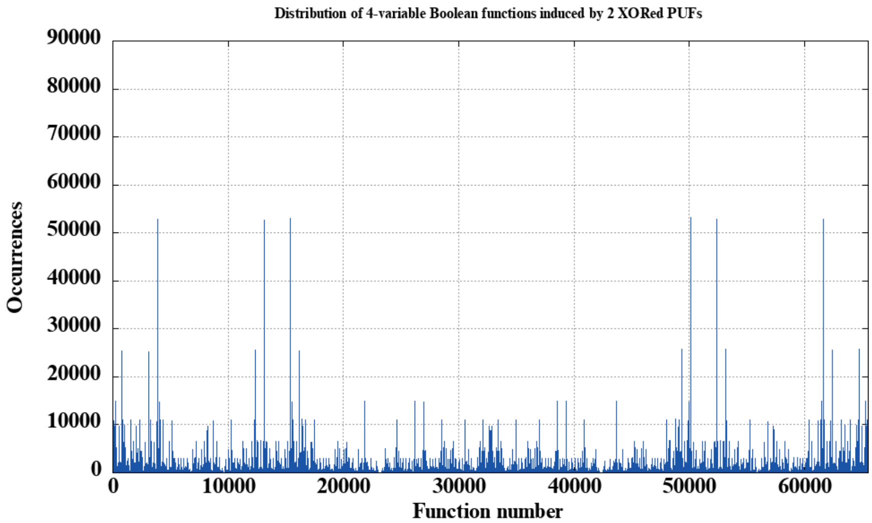


Figure 4.13: Distribution of 4-variable Boolean functions induced by two XORed 4-stage Arbiter PUFs (100 million trials). 11,226 functions are not induced.

4.2.1 Coverage

Again, there is little need to study the 1-variable functions, since these are uniformly distributed. An improvement can be seen in the case of 2-variable functions, for which 12 functions are needed to reach 80% coverage [See Figure 4.14]. Furthermore, the coverage of 3-variable functions improved by factor 2.5, to 151 [See Figure 4.15]. 30% of all possible 4-variable Boolean functions (18,851) are needed to give 80% coverage of two XORed 4-stage PUFs [See Figure 4.16].

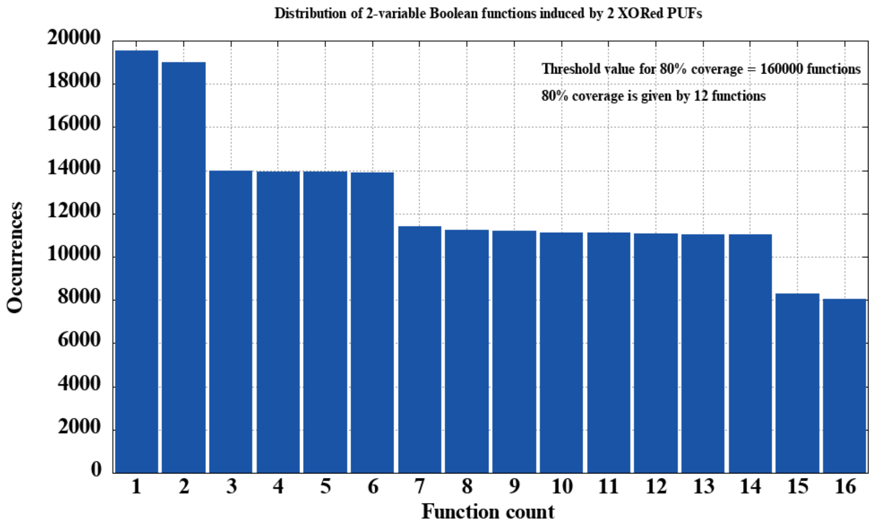


Figure 4.14: Distribution of 2-variable Boolean functions induced by two XORed 2-stage Arbiter PUFs. 80% coverage is given by 12 functions.

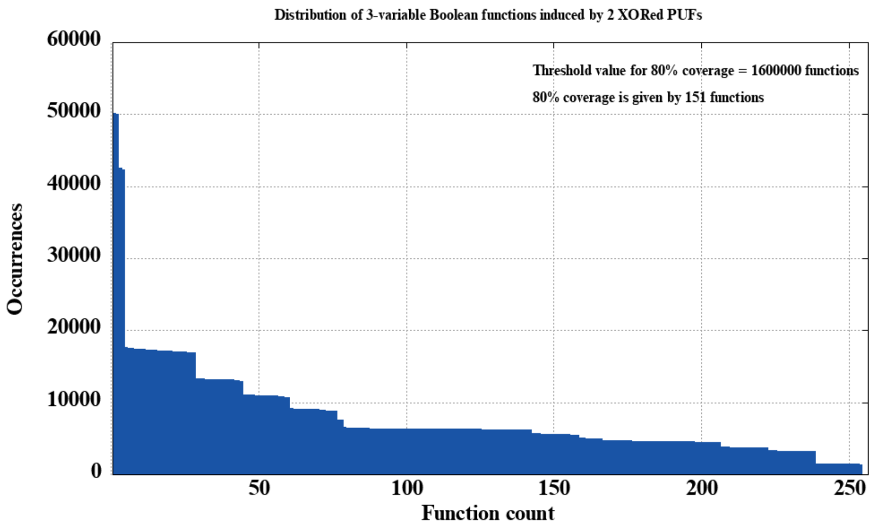


Figure 4.15: Distribution of 3-variable Boolean functions induced by two XORed 3-stage Arbiter PUFs. 80% coverage is given by 151 functions.

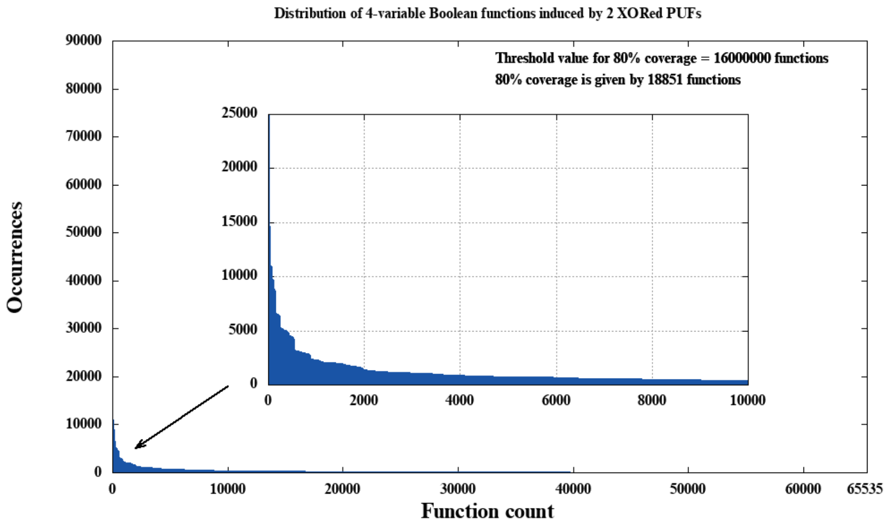


Figure 4.16: Distribution of 4-variable Boolean functions induced by two XORed 4-stage Arbiter PUFs. 80% coverage is given by 18,851 functions.

4.3 Three XORed Arbiter PUFs

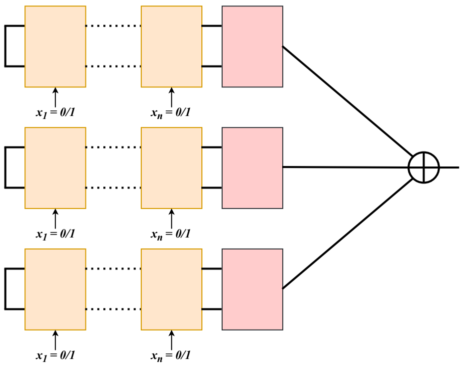


Figure 4.17: Setup for three XORed arbiter PUFs

In Section 4.3 we assume the schematic as given in Figure 4.17, where the output of 3 PUFs are XORed and evaluated for n different variables. The challenge bits $x_1, ..., x_n$ are kept the same for each of the three PUFs. Once again the 1-variable functions remain unchanged and all 2-variable functions are still induced. However, we note the increase in probability of x_1 and $\overline{x_1}$ [See Figure 4.19]. The two 3-variable functions $x_1 \oplus x_3$ and $\overline{x_1 \oplus x_3}$ are now induced. Therefore, all 3-variable Boolean functions can be induced by three XORed PUFs [See Figure 4.20]. We further note the increase in probability of four functions and decrease in constant 1 and 0. Two 4-variable functions, namely $x_1 \oplus x_3$ and $\overline{x_1 \oplus x_3}$ are not induced in this setup either [See Figure 4.21]. However, these were successfully induced using four XORed, PUFs [Please note that those experiments have not been included.]

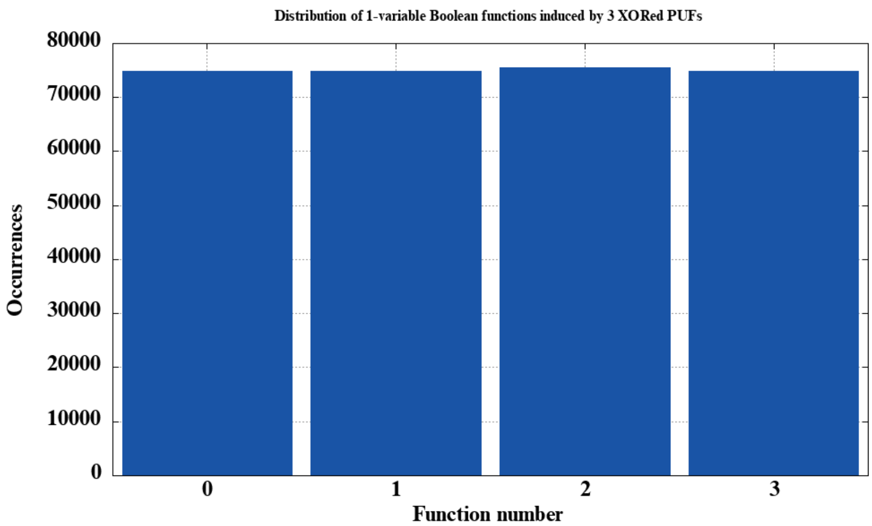


Figure 4.18: Distribution of 1-variable Boolean functions induced by three XORed 1-stage Arbiter PUFs. All functions are equally probable.

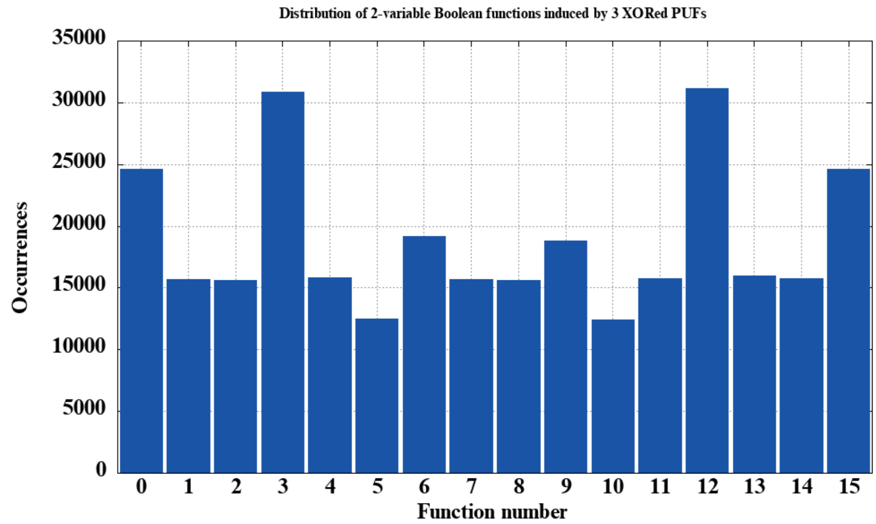


Figure 4.19: Distribution of 2-variable Boolean functions induced by three XORed 2-stage Arbiter PUFs (100,000 trials). All functions are indcued.

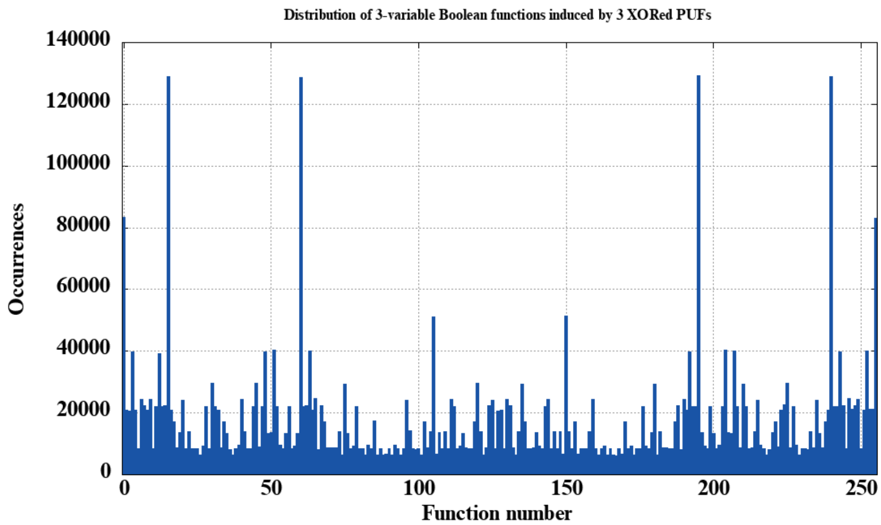


Figure 4.20: Distribution of 3-variable Boolean functions induced by three XORed 3-stage Arbiter PUFs (1.5 million trials). All functions are induced.

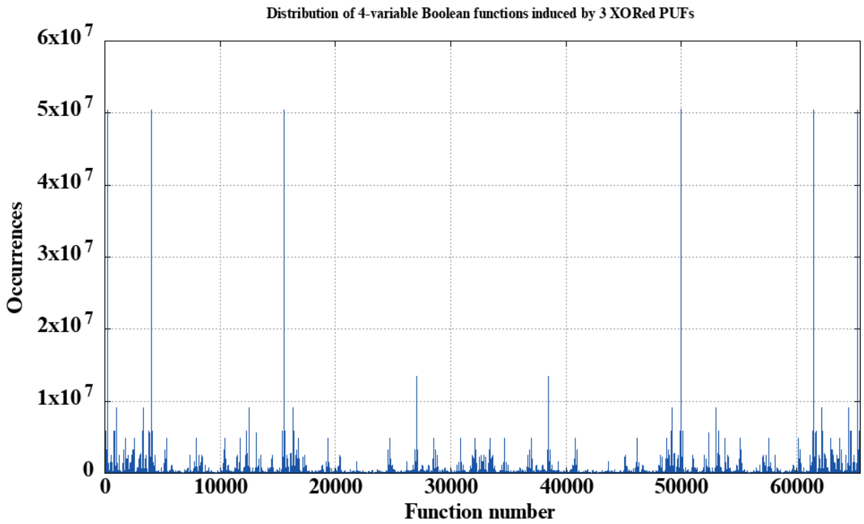


Figure 4.21: Distribution of 4-variable Boolean functions induced by three XORed 4-stage Arbiter PUFs (2 billion trials). All functions apart from $x_1 \oplus x_3$ and $\overline{x_1 \oplus x_3}$ are induced.

4.3.1 Coverage

1-variable functions remain uniformly distributed. Hence, we do not study their coverage. 80% coverage is also obtained by 12 functions in this setup. However, we notice a faster drop in occurrences in Figure 4.22 compared to the previous experiment [See Figure 4.14]. Further analysis shows that this experiment gives 50% coverage with 8 functions whereas the previous requires 9, i.e. the distribution is slightly 'worse'. Similarly, 3-variable functions only needed 145 functions to gain 80% coverage; 6 less than in the previous [See Figure 4.23]. Due to the fact that almost all 4-variable functions are induced, the coverage is significantly better; requiring 23,430 function for 80% coverage [See Figure 4.24].

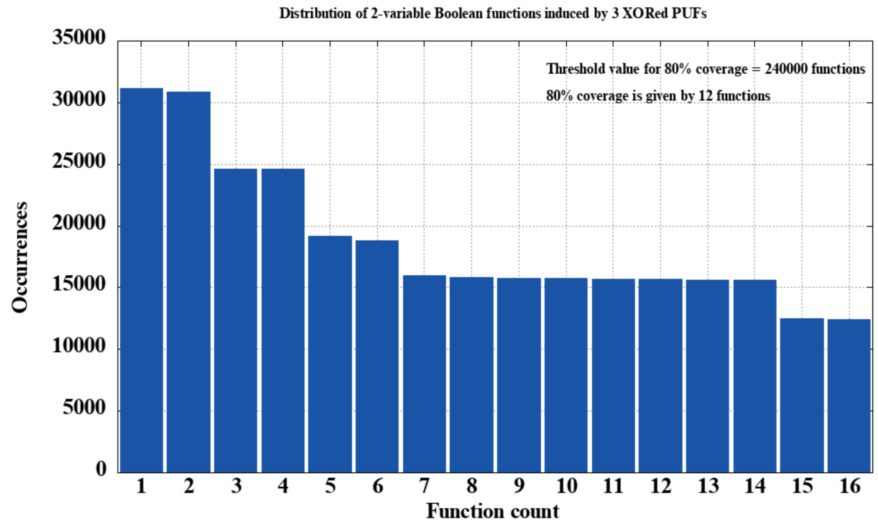


Figure 4.22: Distribution of 2-variable Boolean functions induced by three XORed 2-stage Arbiter PUFs.

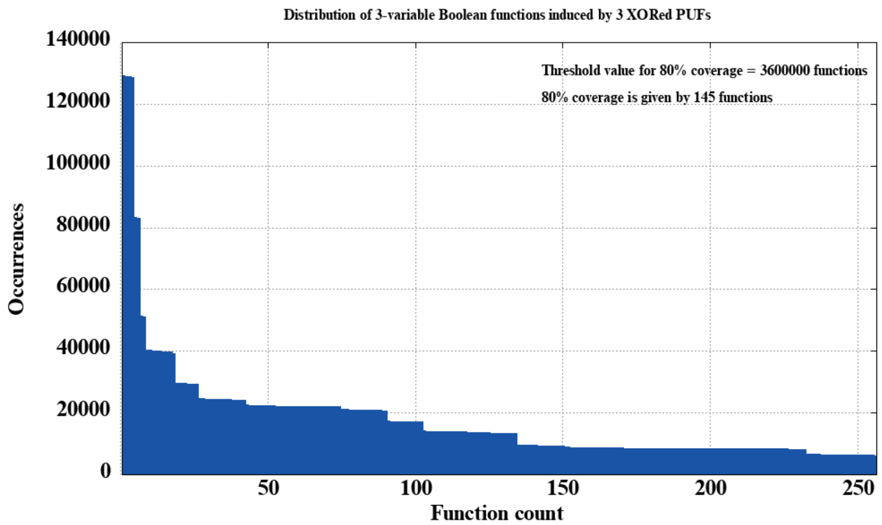


Figure 4.23: Distribution of 3-variable Boolean functions induced by three XORed 3-stage Arbiter PUFs.

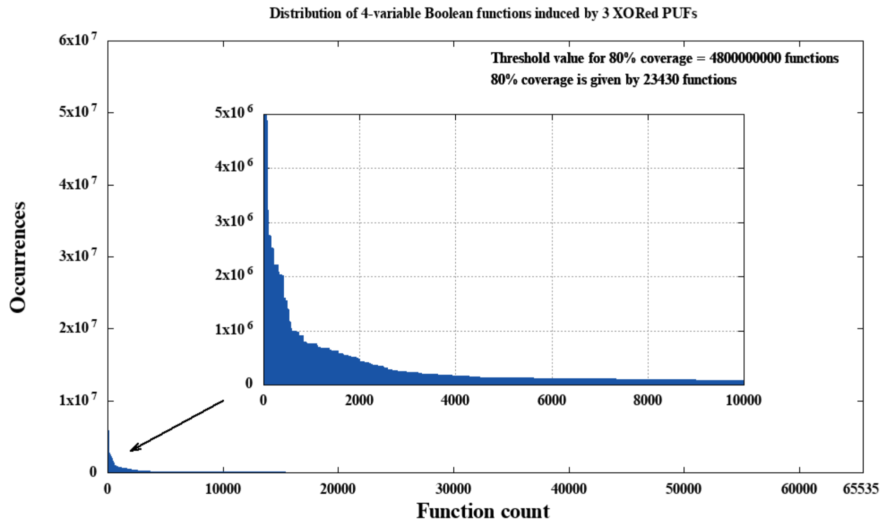


Figure 4.24: Distribution of 4-variable Boolean functions induced by three XORed 4-stage Arbiter PUFs.

4.4 Summary

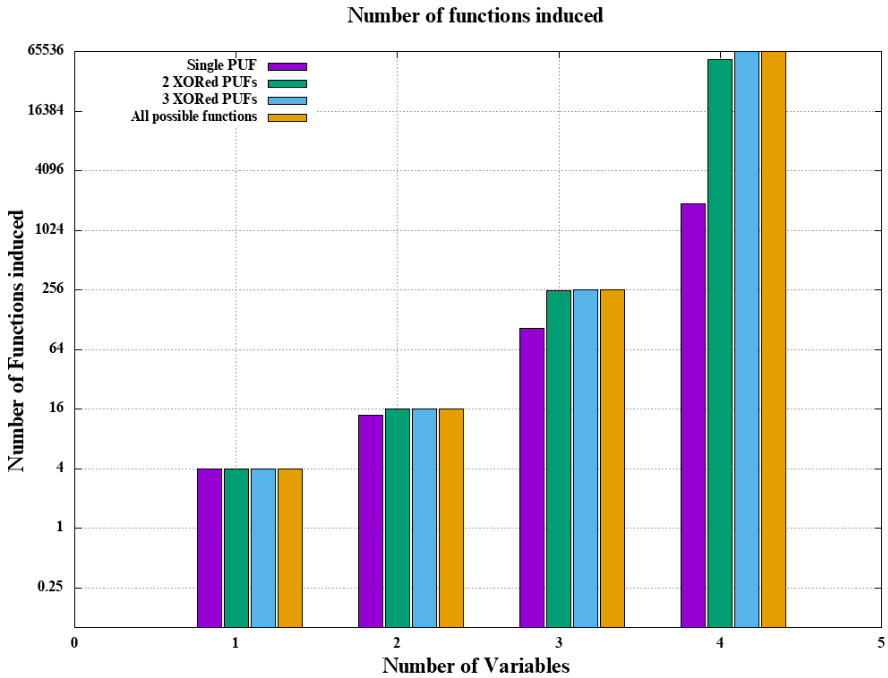


Figure 4.25: XORing the output of PUFs increases the number of functions induced.

The results show that adding multiple PUFs in parallel and XORing their outputs, dramatically increases the number of functions that can be induced. In the case of a single 4-stage PUF less than 3% of all possible functions can in fact be induced. However, by XORing three PUFs in parallel, all but two 4-variable functions are induced [See Figure 4.25 and Table 4.1].

For Table 4.1 we introduce the following notation:
n : number of stages in the arbiter PUF
N_i : total number of Boolean functions for n variables
I_i : number of impossible functions.

Table 4.1: Function Overview

n	N	I
1	4	0
2	16	2
3	256	152
4	65,536	63,654
$\vdots$	$\vdots$	$\vdots$
n	2^{2^n}	$\geq 2^{2^{n-1}} - 2$

(a) One single PUF

n	N	I
1	4	0
2	16	0
3	256	2
4	65,536	11,226
$\vdots$	$\vdots$	$\vdots$
n	2^{2^n}	???

(b) 2 XORed PUFs

n	N	I
1	4	0
2	16	0
3	256	0
4	65,536	2
$\vdots$	$\vdots$	$\vdots$
n	2^{2^n}	???

(c) 3 XORed PUFs

In certain setups only two functions were not induced - a particular function and its complement. The following pairs of functions were not induced:

$$f(x_1, x_2) = \begin{cases} \overline{x_1} \\ x_1 \end{cases} \qquad \text{for } n = 2 \text{ using 1 single PUF}$$

$$f(x_1, x_2, x_3) = \begin{cases} \overline{x_1 \oplus x_3} \\ x_1 \oplus x_3 \end{cases} \qquad \text{for } n = 3 \text{ using 2 XORed PUFs}$$

$$f(x_1, x_2, x_3, x_4) = \begin{cases} \overline{x_1 \oplus x_3} \\ x_1 \oplus x_3 \end{cases} \qquad \text{for } n = 4 \text{ using 3 XORed PUFs}$$

An experiment in which 4 PUFs were XORed, showed that both $x_1 \oplus x_3$ and $\overline{x_1 \oplus x_3}$ were induced. We hypothesise the need for n XORed PUFs to induce all 2^{2^n} n-variable functions. Although adding multiple PUFs increases the number of functions that are induced, the coverage is not necessarily improved. Secure PUFs should strive for a uniform distribution of functions, making it virtually impossible for adversaries to perform attacks based on exploiting biases in the distribution.

Hypothesis 1: *n XORed arbiter PUFs can induce all 2^{2^n} n-variable Boolean functions.*

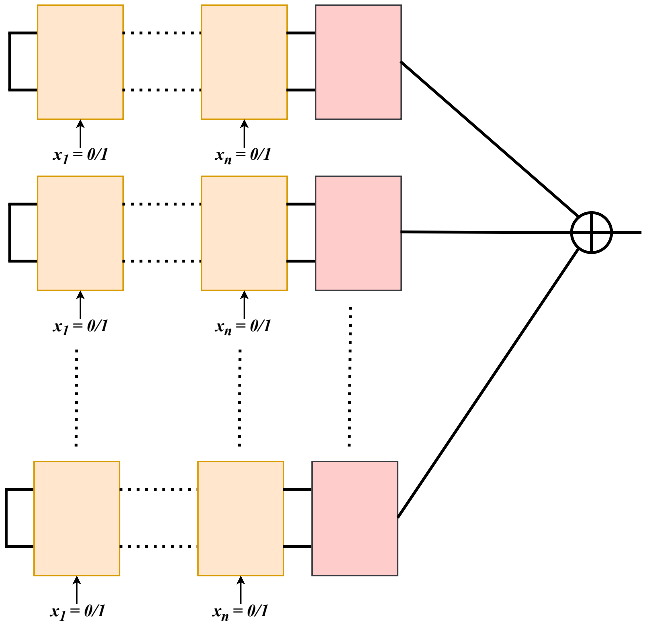


Figure 4.26: Setup for n XORed PUFs to induce all possible n-variable functions.

Chapter 5

Conclusion

The results presented in this thesis allow us to draw some interesting conclusions. In particular, adding multiple PUFs in parallel and XORing their outputs, significantly increases the number of functions that can be induced leading to Hypothesis 1. In order to maximise the security of PUFs, the functions induced should ideally be uniformly distributed. XORing does not necessarily improve the distribution towards a uniform distribution. However, XORing does make brute-force attacks harder due to adversaries needing to compare a larger set of functions.

5.1 Future Work

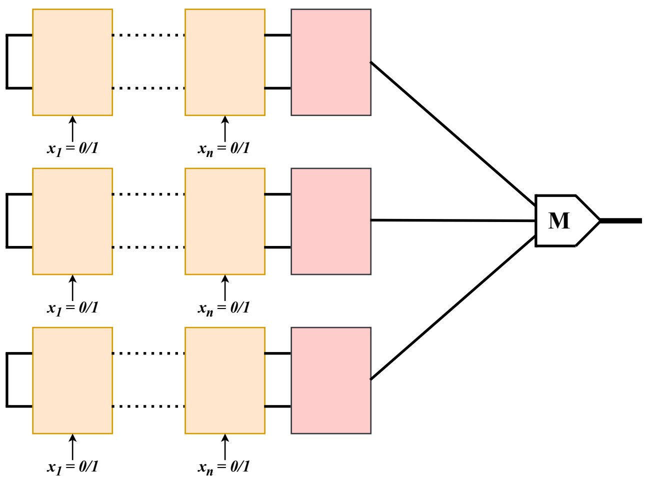


Figure 5.1: Setup for three arbiter PUFs MAJed

This study exclusively looked into designs using XOR gates to increase the number of functions that could be induced and their distributions. According to Hypothesis 1, 128 PUFs would be required to implement a 128-stage arbiter PUF along with all of its Boolean functions - requiring a substantial amount of hardware. Other designs may reduce hardware and further improve coverage. Future studies could address non-linear combinations using majority gates for example [See Figure 5.1].

Furthermore, the coverage was evaluated using all 2^{2^n} Boolean functions. However, it may have been more appropriate to reason about the coverage with regard to only possible functions, as opposed to the theoretical maximum.

Another interesting future study would be to look into parallel PUFs with different challenges for each PUF.

Appendices

Appendix A

Shell Script

```
#!bin/bash

var=1
tries=0
while [ $var -lt 5 ]; do

    if [ $var -lt 3 ];
        then
            let tries=100000
    elif [ $var -lt 4 ]
        then
            let tries=1000000       # 1 mil
        else
            let tries=10000000      # 10 mil
    fi

    g++ simulation_single.cpp
    ./a.out $var $tries >
"./dat/dist/Distr. of ${var}-variable Bool. func. induced by 1 single PUF"

    if [ $var -lt 3 ];
        then
            let tries=100000
    elif [ $var -lt 4 ]
        then
            let tries=1000000       # 1 mil
        else
            let tries=10000000      # 10 mil
    fi

    .
    .
    .

    let var=var+1
done

cd ./dat/dist/
for FILE in *; do
    gnuplot <<- EOF
        # Setup
```

```
            set terminal png size 1024,600 font ",18"
            set term png font "times_new_roman_bold,18"
            set term png font "/usr/local/fonts/times_new_roman_bold.ttf"

        # Axis / titles
            set xlabel "Function number" font ",18"
                set ylabel"Occurrences" font ",18" offset -2.5,0
                set xrange [-0.5:]
                set yrange [0:]

            set key off
            set grid
            set tics font ",18"
                set rmargin 5
                set lmargin 15

        # Style
            set boxwidth 0.9 relative
            set style fill solid 1.0
            set output "../../figs/dist/" . "${FILE}.png"
            set title "${FILE}"
            plot "${FILE}" with boxes linetype rgb "#1954A6"

        # Stats
            set fit logfile '/dev/null'
#            stats "${FILE}" using 2 name "occ"
EOF
done

for FILE in *; do
    sort -k2 -n -r "${FILE}" -o "../sorted/${FILE}"
done

cd ../sorted

for FILE in *; do
    awk '{print NR,$0}' "${FILE}" > temp
    mv temp "${FILE}"
done

for FILE in *; do
    tries_done=`awk '{sum+=$3} END{print sum}' "${FILE}"`
    threshold=$(($tries_done * 80/100))
    total=0
    count=1
    while [ $total -lt $threshold ]; do
        line_val=`awk -v line="$count" 'NR==line {print $3}' < "${FILE}"`
        total=$((total + line_val))
        let count=count+1
    done
    let count=count-1

    gnuplot <<- EOF
        # Setup
            set terminal png size 1024,600 font ",18"
            set term png font "times_new_roman_bold,18"
            set term png font "/usr/local/fonts/times_new_roman_bold.ttf"

        # Axis / titles
            set xlabel "Function count" font ",18"
                set ylabel"Occurrences" font ",18" offset -2.5,0
                set yrange [0:]
```

```
            set key off
            set grid
            set tics font ",18"
                set rmargin 5
            set lmargin 15

        # Stats
            set label "Threshold value for 80% coverage = ${threshold} functions"
            at screen 0.6, screen 0.9 front
            set label "80% coverage is given by ${count} functions"
            at screen 0.6, screen 0.85 front

        # Plot
            set boxwidth 0.9 relative
            set style fill solid 1.0
            set output "../../figs/sorted/" . "${FILE}.png"
            set title "${FILE}"
            plot "${FILE}" using 1:3 with boxes linetype rgb "#1954A6"
EOF
done

# lower case dat/sorted/
for FILE in *; do
     mv "${FILE}" "`echo ${FILE} | tr '[A-Z]' '[a-z]'`";
     mv "${FILE}" "${FILE// /_}";
done

cd ../dist/

.
.
.
```

Appendix B

Simulation

Prof. Dr. Elena Dubrova & Phillip Gajland

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <assert.h>

#include <cstdlib>
#include <cmath>
#include <limits>
#include <cstdio>
#include <iostream>
#include <sstream>

using namespace std;

#define MEAN                    1.0
#define STD_DEV                 0.1
#define NUMBER_OF_TRIES         100000

typedef struct sblock {
    double u;                   /* upper path output*/
    double l;                   /* lower path output */
    double d1;                  /* delay of wire connecting u and u, d11*/
    double d2;                  /* delay of wire connecting l and l, d12*/
    double d3;                  /* delay of wire connecting u and l, d13*/
    double d4;                  /* delay of wire connecting l and u, d14*/
} sblock;

static float compute_upper_delay(
    float up, float low, unsigned x, float d_straight, float d_cross);

static float compute_lower_delay(
    float up, float low, unsigned x, float d_straight, float d_cross);

double generateGaussianNoise(double mu, double sigma);

/* ---------------------------------------------------------------------------- */
int main(int argc, char *argv[]) {

    /* puf size (number of blocks) defined from command line */
    int size = stoi(argv[1]);;
```

```
    /* number of tries defined from command line */
    unsigned long number_of_tries = stoi(argv[2]);;
    int number_of_challenges = (1 << size);
    int number_of_functions = (1 << number_of_challenges);

    unsigned i, j, k, tmp, a, f, function1, function2, function3;
    /* puf consists of multiple sblocks */
    sblock puf[size + 1];
    /* challenge bits = x */
    unsigned challenge_bits[size];
    /* output sequence */
    unsigned output[number_of_challenges];
    unsigned ones_counter = 0;
    /* How many functions???? */
    unsigned function_occurrences[number_of_functions];

    int seed = (int) time(NULL);
    srand(seed);

    for (i = 0; i < number_of_functions; i++) {
        function_occurrences[i] = 0;
    }

    for (a = 0; a < number_of_tries; a++) {

        // NEW PUF IS MADE HERE

        // THREE PUFs IN PARALLEL
        for (int no_of_pufs = 0; no_of_pufs < 3; no_of_pufs++) {

            /* Assign delays of paths d1-d4 to each sblock */
            for (i = 0; i < size; i++) {
                puf[i].d1 = generateGaussianNoise(MEAN, STD_DEV);
                puf[i].d2 = generateGaussianNoise(MEAN, STD_DEV);
                puf[i].d3 = generateGaussianNoise(MEAN, STD_DEV);
                puf[i].d4 = generateGaussianNoise(MEAN, STD_DEV);
            }

            // DONE FOR ONE PUF (multiple sblocks)
            for (k = 0; k < number_of_challenges; k++) {
                for (j = 0; j < size; j++) {
                    challenge_bits[j] = (k >> j) & 1;
                }

                puf[0].u = 0;
                puf[0].l = 0;

                // calculate paths taken
                for (i = 0; i < size; i++) {
                    puf[i + 1].u = compute_upper_delay(
                        puf[i].u, puf[i].l, challenge_bits[i], puf[i].d1, puf[i].d4);
                    puf[i + 1].l = compute_lower_delay(
                        puf[i].u, puf[i].l, challenge_bits[i], puf[i].d2, puf[i].d3);
                }

                // evaluate output of last block
                if (puf[size].l > puf[size].u) {
                    output[k] = 0;
                } else {
                    output[k] = 1;
                }

```

```
                // evaluate which function was implemented by puf
                f = 0;
                for (i = 0; i < number_of_challenges; i++) {
                    f = f + (output[i] << i);   // f = f + (2^i)*output[i];
                    if (output[i] == 1) {
                        ones_counter++;
                    }
                }
            }

            if (no_of_pufs == 0) {
                function1 = f;
            }
            if (no_of_pufs == 1) {
                function2 = f;
            }
            if (no_of_pufs == 2) {
                function3 = f;
            }

            f = function1 ^ function2 ^ function3;
            function_occurrences[f]++;
        }

        function1 = 0;
        function2 = 0;
        function3 = 0;
    }

    for (i = 0; i < number_of_functions; i++) {
        /* Function number: %d occurred %d times*/
        printf("%d\n", function_occurrences[i]);
        printf("%d %d\n", i, function_occurrences[i]);
    }
    printf("\n%d ones \n", ones_counter);

    return (0);
}

static float compute_upper_delay(
    float up, float low, unsigned x, float d_straight, float d_cross) {

    float out;

    if (x == 0) {
        out = d_straight + up;
    } else {
        out = d_cross + low;
    }
    return (out);
}

static float compute_lower_delay(
    float up, float low, unsigned x, float d_straight, float d_cross) {

    float out;

    if (x == 0) {
        out = d_straight + low;
    } else {
        out = d_cross + up;
    }
```

```
    return (out);
}

double generateGaussianNoise(double mu, double sigma) {
    static const double epsilon = std::numeric_limits<double>::min();
    static const double two_pi = 2.0 * 3.14159265358979323846;

    static double z1;
    static bool generate = false;
    generate = !generate;

    if (!generate)
        return z1 * sigma + mu;

    double u1, u2;
    do {
        u1 = rand() * (1.0 / RAND_MAX);
        u2 = rand() * (1.0 / RAND_MAX);
    } while (u1 <= epsilon);

    double z0;
    z0 = sqrt(-2.0 * log(u1)) * cos(two_pi * u2);
    z1 = sqrt(-2.0 * log(u1)) * sin(two_pi * u2);
    return z0 * sigma + mu;
}
```

References

[AAB+16] Martin Abadi, Ross Anderson, Mihir Bellare, Oded Goldreich, Tatsuaki Okamoto, Paul Van Oorschot, Birgit Pfitzmann, Aviel D Rubin, Jacques Stern, Hans Delfs, and Helmut Knebl. *Towards Hardware-Intrinsic Security.* 2016.

[AMS+09] Frederik Armknecht, Roel Maes, A.R. Sadhegi, Berk Sunar, and P. Tuyls. Physically Unclonable Pseudorandom Functions, 2009.

[Bec15] Georg T Becker. The gap between promise and reality: On the insecurity of XOR arbiter PUFs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9293, pages 535–555, 2015.

[BH13] Christoph Böhm and Maximilian Hofer. *Physical unclonable functions in theory and practice*, volume 9781461450. 2013.

[KK99] Oliver Kömmerling and Markus G Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. *Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '99)*, pages 9–20, 1999.

[LKA] Ting Lu, Ryan Kenny, and Sean Atsatt. Introduction The Secure Device Manager for Intel Stratix 10 devices provides a failsafe, strongly authenticated, programmable security scheme for device configuration. Secure Device Manager for Intel® Stratix® 10 Devices Provides FPGA and SoC Security.

[Mor03] Claudio Moraga. Dual Polarity Property. 1(1):71–80, 2003.

[MV10] Maes Roel and Verbauwhede Ingrid. *Towards Hardware-Intrinsic Security.* Number October 2010. 2010.

[Nor16] Amy Nordrum. The internet of fewer things [News]. *IEEE Spectrum*, 53(10):12–13, 2016.

[PRTG02] Ravikanth Pappu, Ben Recht, Jason Taylor, and Neil Gershenfeld. Physical one-way functions. *Science*, 297(5589):2026–2030, 2002.

[TM14] Stephen M. Trimberger and Jason J. Moore. FPGA security: Motivations, features, and applications. *Proceedings of the IEEE*, 102(8):1248–1265, 2014.

[WYDG18] Wei Che Wang, Yair Yona, Suhas N. DIggavi, and Puneet Gupta. Design and Analysis of Stability-Guaranteed PUFs. *IEEE Transactions on Information Forensics and Security*, 13(4):978–992, 2018.

[XI17] Xilinx and Inc. Developing Tamper-Resistant Designs with Zynq UltraScale+ Devices Application Note (XAPP1323). 2017.

TRITA TRITA-EECS-EX-2018:111