ulm university universität
uulm

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Mikroelektronik

Breaking Physical Unclonable Functions using Neural Networks

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Jonas Otto
edwin.otto@uni-ulm.de
982249

Gutachter:
Prof. Dr.-Ing. Maurits Ortmanns
Prof. Dr.-Ing. Robert Fischer

Betreuer:
Holger Mandry

2020

Version November 23, 2020

© 2020 Jonas Otto

This work is licensed under the Creative Commons Attribution 4.0 International (CC BY 4.0) License. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Typesetting: PDF-LaTeX 2_ε

Contents

1 Introduction **1**

2 Theory and Background **3**
2.1 Physical Unclonable Functions 3
2.1.1 Examples of Weak PUFs 5
SRAM-PUF 5
Ring-Oscillator based PUFs 6
2.1.2 Examples of Strong PUFs 6
Arbiter PUF 7
Loop-PUF 8
2.2 Artificial Neural Networks 8
2.2.1 Perceptron 9
Perceptron Learning Algorithm 10
2.2.2 Multi Layer Perceptron 10
Activation Functions 11
MLP Learning 12

3 Implementation **15**
3.1 Simulation 15
3.2 Preprocessing 16
3.3 Modeling 17
3.4 Hyperparameter Tuning 17
3.4.1 Evaluation 19

4 Results **21**
4.1 Arbiter PUF 22
4.1.1 Error prone training data 24
4.2 XOR Arbiter PUF 25

4.3 Staged Arbiter PUFs 27
4.3.1 V1 Arbiter PUF 28
Special case $k = n$ 29
Special case $k \cdot m = n$ 32
4.3.2 V2 Arbiter PUF 34
4.3.3 V3 Arbiter 36
4.4 Loop PUF 38
4.5 Comparison 40

5 Conclusion and Outlook **43**

Bibliography **45**

List of Acronyms

PUF physical unclonable function 1

FPGA field-programmable gate array 3

CRP challenge-response-pair 4

ANN artificial neural network 1

MLP multi layer perceptron 3

SGD stochastic gradient descent 12

ReLU rectified linear unit 11

MSE mean squared error 12

BER bit error rate 16

CSV comma separated values 15

1 Introduction

In recent years, not only the number of mobile computing devices has grown significantly, but the amount of very small, low power embedded devices such as identification tags and cards in daily use is higher than ever.

Almost all of those devices perform some sort of security functions as their primary or secondary purposes. Functions such as device and user authentication, verification and generation of digital signatures, encryption and protection of user data are ubiquitous. Those all rely on traditional cryptographic measures, often relying on the secrecy of a private key. This presents an issue especially on devices which are very resource constrained in terms of chip area and power usage. In those cases, it may not be possible to provide non volatile memory for secure storage of keys, especially if the security requirements call for special anti-tamper measures to protect the storage against invasive attacks.

Physical unclonable functions (PUFs) are a proposed solution to those problems. A PUF allows derivation of a secret key without the need for additional storage in a way that is inherently tamper sensitive, or provides a direct authentication mechanism to the device. Those authentication mechanisms have however been shown to be vulnerable against modeling attacks. Both the construction of a mathematical model using in-depth knowledge about the PUFs design, and modeling using machine learning techniques including artificial neural networks (ANNs) have been demonstrated [13] and pose a significant limitation on the security and usability of PUFs.

The goal of this thesis is to implement a program capable of executing those attacks, which allows testing of a PUF design with regards to this specific vulnerability. The vulnerability or resistance of specific designs should also be quantifiable to give an estimate of the required effort an attacker would have to undertake in order to succeed in impersonating such a device. This program will then be used to evaluate and compare the modeling resistance of multiple PUF designs.

In chapter 2, an introduction into the working principle of PUFs is given, as well as an overview of the machine learning techniques used. Chapter 3 goes into detail about the implementation of the program and the metrics used to evaluate the modeling efforts. The results of testing various PUF architectures against ANN based modeling are presented in chapter 4. A conclusion and aspects of possible future work are given in chapter 5.

2 Theory and Background

In this chapter, an overview of the fundamentals of both physical unclonable functions and artificial neural networks is given. This will provide the necessary background on PUFs, as well as introduce the fundamental concept behind the specific PUFs used in later chapters. Furthermore, the principle behind ANNs will be introduced with a focus on multi layer perceptrons (MLPs), as this is the type of ANN used in this thesis.

2.1 Physical Unclonable Functions

The ever rising number of mobile and embedded electronic devices prompts a need for lightweight authentication methods. Currently, this is provided using on-device secret keys, which are placed in non-volatile memory, alongside cryptographic hardware capable of encryption or digital signing. This is however not the most area- and power efficient method of authentication. PUFs aim to not only improve efficiency, but also resistance against invasive attacks, which traditionally required specialized anti-tamper hardware [5, 22].

PUFs can be described as systems, whose behavior is dependent on random internal variations, which inevitably occur during manufacturing of the device. It is crucial that those variations can not be deliberately fabricated, for example after analyzing another instance of the PUF. Although the concept is not limited to electronic circuits, as introduced in [4], the PUFs considered in this work are all “silicon PUFs”, that may be implemented on field-programmable gate arrays (FPGAs), in silicon on a dedicated chip, or adjacent to the device using it.

The PUF generally has a digital output of one or multiple bit, which is referred to as the response r. It may also accept additional input in the form of a challenge c. This makes it possible to model the PUF as a black-box challenge-response system. The response is given as $r = f(c)$, where $f()$ is unique to each instance of the PUF. A tuple containing a challenge and its corresponding response (c, r) is called a challenge-response-pair (CRP).

Physical unclonable functions can be categorized into two categories, the main difference being the domain of $f()$:

Weak PUF Weak PUFs are a way of storing secret keys other than in non-volatile memory. They generally allow for only a small set of challenges, commonly only one. This requires some level of robustness in the PUF output or appropriate error correction, since it is not possible to compensate for an unreliable output by applying a large amount of challenges. It is also required to protect the PUF against unauthorized readout, as a single readout may reveal the entirety of the stored key [22].

Strong PUF The Strong PUF is characterized by a large amount of possible challenges. It is desired that the number of possible challenges grows exponentially while the circuit area grows linearly, which prevents an attacker from reading out all possible challenge-response-pairs. It is also important that an attacker might not infer the response to any challenge even with knowledge of every CRP used so far [5]. Desirable properties of a PUF are also to maximize both the intra-hamming-distance, which describes the hamming distance of responses of a single PUF to different challenges, and the inter-hamming-distance, which measures the distance of the responses of multiple PUF instances. Those measures prevent inherent bias of the response both in each individual instance and across multiple instances of the same design. This makes strong PUFs suitable for direct use in authentication flows. A naive strong PUF authentication protocol consists of two phases:

During *enrollment*, the device is inside a trusted area in possession of the issuing body. A large number of (uniformly random generated) CRPs is extracted from the device and stored in a secure database. It is important that the CRPs extracted at this stage stay private, and the readout process is not observed by an attacker. This phase happens only once.

In the *authentication* phase, the server wants to authenticate the device over an insecure channel. The server sends a challenge c from the CRP database to the device, which calculates the response $\tilde{r} = f(c)$. The server then evaluates if the response matches the one in the database ($\tilde{r} = r$). This step may be repeated by sending multiple challenges or deriving further challenges from the first one on the device, which enables the server to authenticate the device even if the PUF is unreliable, which is generally assumed.

It has already been shown that this approach to authentication is vulnerable to modeling attacks, with the goal of predicting PUF responses after observing previous successful authentication sequences. Further PUF-based authentication protocols have been proposed. Those are however also limited in the amount of security improvement they provide, or negate the advantages in circuit-area or complexity of PUFs described above [2]. Therefore, the naive authentication protocol described here is assumed in the following.

2.1.1 Examples of Weak PUFs

In this section, two common examples for approaches to constructing weak PUFs shall be given. Those will not be covered in detail, as the rest of this work relies on the challenge-response behavior as exhibited by strong PUFs. Nonetheless, a clear distinction between PUF types is essential.

SRAM-PUF

During power-up of an SRAM cell, the cell will transition from an unstable state into either one of the two stable 1 or 0 states. Which of those states is assumed, is dependent on both noise during power-up, but also on process variation of the transistors. Each cell has therefore a tendency (or “skew”) towards one state, which

is consistent on each power-up. This produces a unique fingerprint for each SRAM chip. Identification is done by computing the Hamming distance between the to be identified fingerprint and all known fingerprints, and choosing the device with the smallest distance. It has been shown that with a large enough fingerprint size, the device can be accurately identified among a population of SRAM chips [7]. This type of PUF has no challenge, as the response is present as soon as the SRAM is powered on.

Ring-Oscillator based PUFs

A type of PUF which is especially suited for implementation on FPGAs is the Ring-Oscillator- or RO-PUF [25]. It consists of N identically laid-out ring oscillators, which differ slightly in frequency due to manufacturing variation. An output bit is obtained by selecting two ring oscillators based on the challenge, and comparing the frequencies of the selected oscillators. While there are $\frac{N(N-1)}{2}$ possible choices of two different oscillators to compare, the number of independent bits that can be generated is only $\log_2(N!)$. Correlations are present as soon as two oscillators a and c are compared, where they have both previously been compared to an oscillator b with the results $a < b$ and $b < c$. Additional restrictions on comparable oscillators apply, as it has been shown that details about the specific placement of individual oscillators introduce bias in the measured frequencies [6]. Because of this relatively small challenge-set, the RO-PUF is classified as a weak PUF.

2.1.2 Examples of Strong PUFs

Strong PUFs are the main aspect of this work, as the inherent property of a large set of possible CRPs makes machine-learning based attacks possible. In this section, both the arbiter PUF and the Loop-PUF are introduced. The arbiter PUF represents the fundamental building block for the PUF designs evaluated in sections 4.2 and 4.3, and the Loop-PUF is one of multiple approaches of utilizing ring-oscillators in strong PUFs.

Arbiter PUF

The arbiter PUF works on the principle of a race condition between two circuit paths of identical nominal length. A rising edge on the input is split into two paths, which travel through a chain of delay elements. In each element of the PUF, the paths may be swapped if the challenge bit applied to the corresponding element is 1. This results in a different total path for each applied challenge, while keeping the length of both paths the same relative to each other. In reality, the paths in each element differ uncontrollably due to process variations in manufacturing. This results in a difference in signal propagation through the PUF, at the end of which the arbiters output is only dependent on which signal arrives first.

Figure 2.1 shows an arbiter PUF with an applied challenge, and it can be observed how the paths are swapped every time the challenge bit c_i is 1. It is also apparent that the number of challenges $n = 2^N$ grows exponentially with the number of elements N in the PUF.

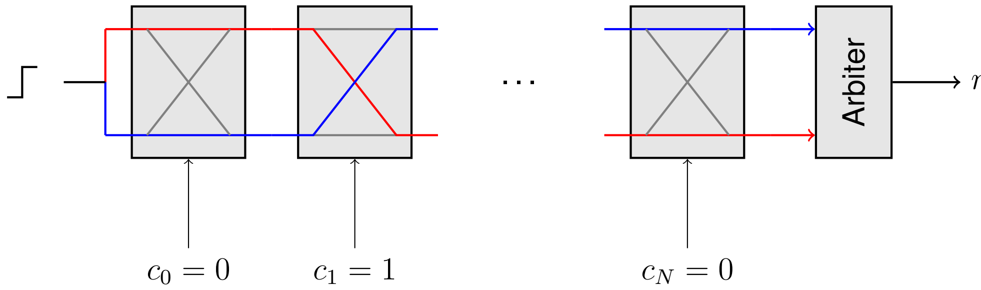


Figure 2.1: Schematic of the arbiter PUF.

When describing the N-stage arbiter PUF using an additive delay model [3], only $N + 1$ parameters are necessary, after eliminating delay components common to both paths. The PUF response is then linearly dependent on the parity vector Φ [24]:

$$\Phi_i = \prod_{k=i}^{n-1}(1 - 2c_k),\ i = 0, \dots, n-1,\ \Phi_n = 1 \tag{2.1}$$

$$\Delta_c = \vec{w}^T \Phi \tag{2.2}$$

Where Δ_c is the delay difference of both paths when applying challenge c, and $\vec{w} \in \mathbb{R}^{N+1}$ the weight vector containing the PUF parameters.

Loop-PUF

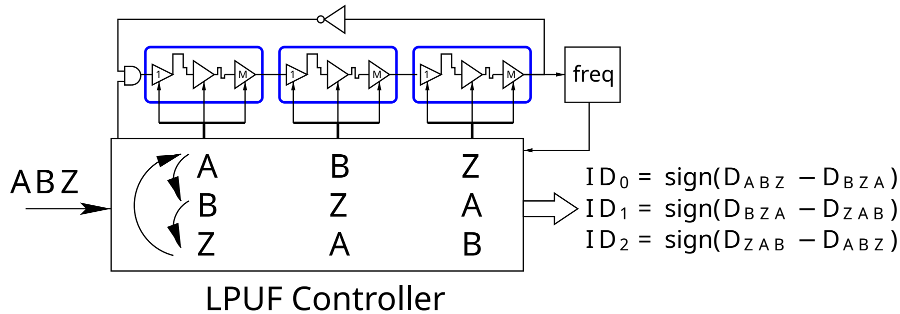


Figure 2.2: Schematic of the Loop PUF with controller taken from Zhoua Cherif Jouini et al. [1]

Although the ring-oscillator based PUF has been described as a weak PUF (section 2.1.1), there exist strong PUF designs utilizing ring-oscillators. One example is the Loop-PUF [1]: The Loop-PUF is based on a single ring-oscillator, which is comprised of configurable delay chains. Each delay chain consists of multiple configurable delay elements. The delay elements are configured by one challenge bit each. During a single readout process, the ring oscillator is measured multiple times with different configurations, which a controller derives from the initial challenge. By comparing the frequency measurements, multiple response bits are generated. Figure 2.2 illustrates this with a controller that divides the challenge into three parts A, B and Z. By rotating the challenge, it is applied to the ring-oscillator three times, resulting in three measurements D, from which the response is generated. It is important that the frequency measurement itself is kept secret, as this would greatly simplify modeling of the ring oscillator.

2.2 Artificial Neural Networks

In this thesis, ANNs are used to attack PUFs by modeling the relationship between challenge and response based on previously seen CRPs. In this chapter, the specific type of ANN used is described.

ANNs are networks made up of neurons, which are modeled after the human brain. They are capable of providing solutions to problems in computer science that are hard to solve manually. This is done by a method of teaching, which makes ANNs part of the field of machine learning.

Use of an ANN is usually split into two phases: In the training phase, the network is presented with example data, and adjusts its parameters in order to learn the relationship between the input and output data.

During the inference phase, new input data is presented to the network, which then predicts the output. In the following, the perceptron and multi layer perceptron are introduced:

2.2.1 Perceptron

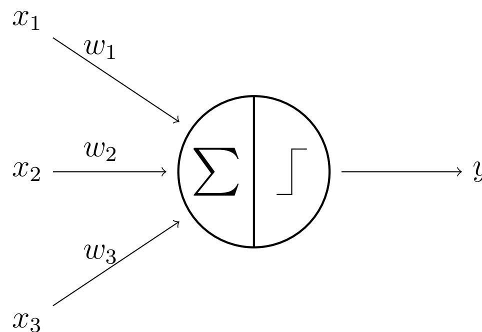


Figure 2.3: Schematic illustration of a single perceptron with input vector $\vec{x}$, weights $\vec{w}$ and output y (Source: own illustration)

The perceptron is modeled after a single biological neuron. Given an input vector $\vec{x} \in \mathbb{R}^m$, its output is

$$y = \begin{cases} 1 & \text{if } \sum_{i=0}^{m} w_i \cdot x_i + b > 0, \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

using the weight-vector $\vec{w}$ and bias b.

The perceptron is an algorithm capable of solving simple binary classification problems. Figure 2.3 shows the typical schematic representation of the perceptron including the summation of the weighted inputs and the threshold determining the output.

Perceptron Learning Algorithm

The perceptron learning algorithm was first presented in 1985 [21]. Given a training set $T = \{(\vec{x_1}, t_1), \ldots, (\vec{x_s}, t_s)\}$ containing s feature vectors $\vec{x_i}$ with the corresponding output $t_i \in \{0, 1\}$, the goal of the learning algorithm is to find the weight-vector $\vec{w}$ and bias b which satisfy the constraints in T.

The algorithm is initialized by choosing random weights $\vec{w}$ and bias b, as well as a learning rate η. The following steps are then repeated as long as T contains training data:

1. Choose a vector $\vec{x}$ and the corresponding output t from the training set. Calculate the actual output $y = f(\vec{x})$ using the current weights $\vec{w}$ and bias b.
2. Calculate the difference of the expected and actual output $\delta = t - y$.
3. Adapt the weights and bias: $\vec{w}_i(t+1) = \vec{w}_i + \eta\delta\vec{x}_i$ and $b(t+1) = b + \eta\delta$

If the data in the training set is linearly separable, this algorithm is guaranteed to converge to a solution which correctly separates the feature vectors in T [16].

2.2.2 Multi Layer Perceptron

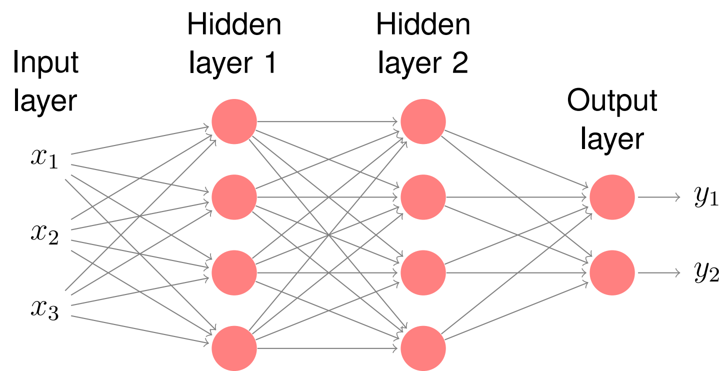


Figure 2.4: Multi Layer Perceptron with Input $\vec{x} \in \mathbb{R}^3$, two hidden layers with four neurons each, and two output neurons. (Source: "Example: Neural network" by Kjell Magne Fauske on texample.net, used under CC BY 2.5 / Layer count and colors adjusted)

In order to solve separation problems which are not linearly separable, multiple neurons can be connected to form a MLP. Neurons in the MLP are arranged in layers, with at least one hidden layer and an output layer. The output layer can have multiple neurons, in this case the network output is a vector $\vec{y} \in \mathbb{R}^n$. MLPs are fully connected networks, which means that every node in a particular layer is connected to every node in the following layer. Figure 2.4 shows a MLP featuring two hidden layers with four neurons each, three input neurons and two output neurons. The Universal Approximation Theorem [8] states that even with a single hidden layer the network can approximate any continuous function, given enough neurons in the hidden layer. Similar observations have been made in the case of a fixed number of neurons per layer, but arbitrary network depth [11].

Activation Functions

In contrast to the formal definition of the perceptron (equation 2.3), the neurons that make up the MLP use activation functions different to the Heaviside step function. In the following, the weighted sum over the neuron inputs will be applied to the activation function f resulting in an output

$$y = f(\vec{w} \cdot \vec{x} + b) = f\left(\sum_{i=0}^{m} w_i \cdot x_i + b\right). \tag{2.4}$$

Different activation functions are in use by state-of-the-art machine learning applications, an overview is given in [18]. Commonly used examples include sigmoid functions such as the logistic function

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

which is commonly used in the output layer, where its output can be interpreted as a classification probability. Activation functions in the output layer can be chosen to fit the desired output vector format, for example such that the output represents a discrete probability distribution in a classification task.

The rectified linear unit (ReLU) function, defined as

$$f(x) = \mathrm{ReLU}(x) = \max\{0, x\} \tag{2.6}$$

has gained popularity [20] due to the low computational complexity when forwarding and calculating gradients, and is often used in hidden layers.

MLP Learning

For the training phase of the MLP, a training set $T = \{(\vec{x_1}, \vec{t_1}), \ldots, (\vec{x_s}, \vec{t_s})\}$ is used analogous to the perceptron learning algorithm.

Training is performed using stochastic gradient descent (SGD), in which the weights are adjusted according to the gradient of the error function.The required gradients for weights in other layers than the output layer are calculated using error-backpropagation [23].

SGD minimizes the error between expected and actual network output according to an error function. The error function is usually chosen depending on the output type of the network. A popular example is the mean squared error (MSE), defined as:

$$e = \|\vec{t} - \vec{y}\|^2 = \sum_{i=1}^{m} (t_i - y_i)^2 \tag{2.7}$$

For multi-class classification problems, cross-entropy and Kullback–Leibler divergence are measures of the similarity of the resulting probability densities. For binary classification tasks, binary cross-entropy may be used [9]:

$$e = t \cdot \log(y) + (1 - t) \cdot \log(1 - y) \tag{2.8}$$

Many current developments in machine learning focus on ANN types other than the MLP used here. Those architectures are usually derived by considering properties of the specific tasks they are trying to solve. Convolutional Neural Networks (CNNs) for example optimize convolution kernels which are applied to images. This introduces location invariance as the same kernels are applied across the whole image, and utilizes the spatial relation present in images [17]. This reduces the number of weights to be trained compared to the MLP, reducing training time. Other networks are specialized for processing time-series data or 3D point-clouds. Many of those architectures such as the CNN can be constructed as special cases of

the fully-connected MLP, where weights are shared or connections missing. As the networks used in this work are comparatively small, and training time is not the limiting factor, those specializations are not required, leaving the generalized MLP approach as a viable option.

3 Implementation

One objective of this work is to implement a framework for experimental evaluation of existing and new PUF designs against modeling attacks using ANNs. This framework should allow a researcher to quickly test the modeling capabilities by providing a set of CRPs. While this is of course not capable of determining that a design is truly resistant against modeling, it is able to indicate if the design is vulnerable against common attacks and provides metrics for comparison. This chapter gives an insight into the implementation of this framework, and the additional simulation capabilities that were added.

3.1 Simulation

All experiments in this work have been carried out using simulated datasets. While some simulation of the arbiter PUF was already available as a Matlab script, it proved convenient to re-implement this in python and integrate it with the simulations required for the other types of PUFs. The arbiter-based PUFs are simulated using an additive delay model similar to [3]: The total delay of each path is the sum of the delays of the connecting wires the signal passes through, with the switch itself assumed to not add any delay. The simulation program does not model noise or other factors contributing to an incorrect response readout. The resistance of modeling against readout error has been evaluated, but to avoid having to create a large number of datasets, the readout error is applied just before training (see section 4.1.1). Simulation was implemented for the arbiter-, XOR-arbiter-, staged-arbiter- and Loop-PUF, and was provided for the arbiter variants V1 - V3, which are introduced in section 4.3. The output of the simulation is a list of CRPs in the comma separated values (CSV) format, which was chosen for its widespread compatibility with other programs.

3.2 Preprocessing

As the PUF architectures used in this work are mainly based on the arbiter PUF, the capability of transforming the challenges into parity-vector representation (equation 2.1) is required. As stated above, the simulated CRP data is not processed in any way, so this preprocessing happens directly after loading the (simulated) dataset from disk. Since the same dataset is likely to be used for many experiments, it proved useful to cache the calculated parity-vectors on disk, since the calculation may take some time especially on datasets containing thousands of CRPs. This computation was also implemented to run parallelized, resulting in a more than $7\times$ speedup on the 6-core, 12-thread system used for most computations.

Another preprocessing option is to introduce random error in the response data. The user can specify a probability with which a response bit will be flipped, simulating noise during readout or error during transmission of the challenge or response. This option of adding noise to the PUF readout was preferred to directly modeling the noise during the simulation stage, since this allows the user to directly specify the bit error rate (BER), which may be easily measured when analyzing a hardware implementation of a PUF. This also allows comparison with different types of PUFs, which may be difficult if measured using noise figures of internal structures that may differ between designs.

A similar approach has been taken in [24]: During simulation, the readout noise for each delay is explicitly simulated. Readout of a single challenge then happens eleven times, using majority voting to determine the final response. Then, additional error is applied later the same way as explained above. In this thesis, it is believed to be beneficial to simulate the PUF in an ideal way, and not make any assumptions about readout noise and the corresponding error correction techniques to compensate for the noise. Instead, all those factors are combined into the total BER.

3.3 Modeling

Modeling is done using ANNs, specifically fully-connected MLPs. For all machine learning related tasks, the widely popular TensorFlow library [15] with the Keras API is used. It provides means of specifying the network architecture layer by layer, and already implements the used activation functions (section 2.2.2). In all experiments, the hidden layers use the ReLU activation function, while the output layer (containing only one node unless explicitly stated) is using the sigmoid function (equation 2.5). Since the network output is a continuous value between 0 and 1, a threshold has to be applied in order to generate the predicted binary PUF response. In the case of successful modeling the output has been observed to be close to zero or one in almost every case, making the choice of threshold less relevant to the result. The built-in accuracy calculation uses a threshold of 0.5 for binarization.

For the training phase, an optimization algorithm has to be chosen. This then minimizes the loss function, which can also be varied. Due to good performance in previous works regarding arbiter- [24] and ring-oscillator [13] PUFs, the Adam optimizer [12] has been chosen. Adam introduces individual learning rates for each parameter, which are continually adapted during training. This improves its performance compared to classical SGD learning. The loss function to be minimized is binary cross-entropy (equation 2.8).

Since we are working with simulated datasets which are far bigger than necessary for modeling, the number of CRPs used for training is selectable by the user, either by the total number or as a percentage of the total dataset.

3.4 Hyperparameter Tuning

The hyperparameters of a neural network include the number of layers and neurons, choice of activation functions and parameters of the learning phase such as learning rate, loss function and optimization algorithm. Because it is not generally possible to choose those parameters intuitively or optimal, efforts have been made to find near-optimal hyperparameters using iterative optimization algorithms. One

of those algorithms is Hyperband [14], which will be used here. Hyperband focuses on speeding up random search over all possible configurations by adaptively allocating resources to candidates. The authors observe a speedup of $5\times$ to $30\times$ over other approaches such as Bayesian optimization [14].

Since this approach does still use considerable resources and time, as many neural network configurations have to be trained, it is important to limit the search space and define the parameters being optimized. The main parameter used here is the network layout, meaning the number of neurons in each layer (see section 2.2.2). The Hyperband implementation used is built into `keras-tuner`, which is the library we use for hyperparameter tuning, as it integrates well with TensorFlow Keras. The implementation allows conditional hyperparameters, which are parameters that may not be present in all configurations. This feature is used in order to optimize the number of neurons in each layer at the same time as the layer count.

Additionally to the number of parameters, the numerical range and step of each parameter influences the search space. Unless explicitly specified, the hyperparameter tuning is performed to test one to four hidden layers, with one to 20 neurons each. This covers the network configurations used in previous work (refer to section 4.1 and following), without increasing the search space significantly, which would in turn vastly increase the runtime required. It is possible to specify a `step` parameter, which can reduce the granularity at which the parameter is modified, but in our case this has been set to the default value of 1.

During the tuning process it is also possible to optimize parameters relating specifically to the training phase of the network. In some cases, we used this to optimize the learning rate, although instead of an integer range a set of possible values has been provided to reduce the search space, since the exact value of this parameter is assumed to be less important than its order of magnitude.

The objective of the tuner has been set to maximize the validation accuracy, to prevent converging on an overfitting model, and an additional early stopping callback is used to stop a trial if the validation loss does not improve over ten consecutive epochs.

3.4.1 Evaluation

There are multiple standard measures for evaluating the performance of an ANN, especially in the special case of a single binary output, which is common for binary classification tasks and is also the case in the networks used here. In those cases, metrics such as the F_1-score are commonly used in order to avoid problems introduced by highly unbalanced datasets, in which one of the two classes is underrepresented. For modeling PUFs however, this is not the case, since PUF designers already aim to create a PUF with an output that is uniformly distributed and does not show a bias towards either response (see section 2.1). This allows the use of the prediction accuracy directly, which provides a much more intuitive metric. The accuracy is defined as

$$acc = \frac{n_{TP} + n_{TN}}{n_{Total}} \tag{3.1}$$

where n_{TP} ("true positives") is the number of responses correctly predicted as one, n_{TN} ("true negatives") the number of correctly predicted zeroes, and n_{Total} the total number of predictions. If a balanced dataset is assumed, the remaining "false negatives" and "false positives" are just the difference to the total number of ones or zeros, which should be both approximately $n_{Total}/2$. This provides the accuracy as a simple metric of "how many responses were predicted correctly".

While this is a useful metric if the number of available CRPs is known, it is of greater importance to the designer of a PUF, how many CRPs an attacker would have to collect, in order to successfully impersonate the PUF [19]. This directly affects the level of security provided by the PUF and is used in this work as a measure to compare different types of PUFs. Determining this measure requires a threshold for determining when the modeling is successful, and may be compared over different values of a parameter of the design, such as the challenge size or number of circuit elements (section 4.3.1). For selection of the threshold we referred to results from hardware implementations of comparable designs, it may also be required to choose this based on the modeling capabilities of the ANN.

In order to evaluate modeling performance over a range of training set sizes, a mode of operation was introduced that performs training with different numbers of CRPs using the same ANN architecture. The results are both displayed in a diagram and output for further analysis. It is also possible to additionally iterate over different values for the BER, as explained in section 3.2. Iteration over model parameters

was omitted, not only since that requires multiple datasets, but also because it was found that it may be advantageous to choose a different network architecture, which may be found using hyperparameter tuning (section 3.4). It should also be noted that evaluation is performed on a dataset different to the training set, to detect problems like overfitting. Since the simulated datasets usually contain vastly more CRPs than needed for training, it was possible to use the remaining part of the dataset as the validation set.

4 Results

In this chapter, the implementation described above is used in order to evaluate different PUF designs. First, the arbiter PUF is modeled as a baseline measurement, to which the other designs can be compared. The main metric will be the number of CRPs required to achieve a sufficient prediction accuracy, as this would most likely be the deciding factor on how feasible an attack would be. Factors such as availability of computational resources and time are considered to be of smaller importance, as the experiments have been executed on desktop computer hardware, with training usually taking less than a few minutes. The only time intensive process is the hyperparameter tuning, but results of that may be applied to many PUF instances of the same design.

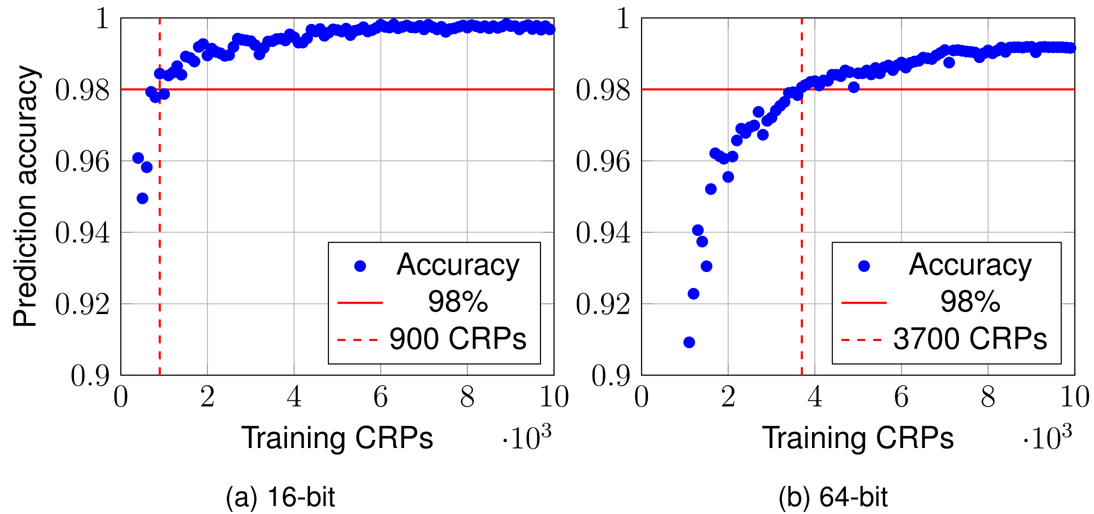
The scope of this work is limited to a specific type of attack. The provided set of CRPs is assumed to be fixed, and the challenges are uniformly random. Other than the parity-vector preprocessing (section 3.2), the training set is not altered before use. Specifically, the challenges are not well chosen in advance. This scenario may present itself when an attacker is able to eavesdrop on the authentication process, but does not possess control over the device in a way that allows readout of responses to arbitrary challenges. In two of the cases presented below, the ability to choose special challenges, in particular those where the challenge bits controlling a specific part of the PUF are constant, may lead to additional vulnerabilities. Those make it possible to separate the PUF into smaller blocks that are easy to model. Those attacks are outside the scope of this evaluation, but have to be considered for a full understanding of the presented PUFs. It will be mentioned explicitly wherever such a vulnerability is suspected.

4.1 Arbiter PUF

Although it has been demonstrated already that the arbiter PUF is not resistant to modeling attacks using neural networks, those results should be confirmed in order to use them as a baseline to compare further results to.

Experiments with a 64-bit arbiter PUF on ASIC produced using 65nm CMOS technology [10] show an expected BER in the range of at least 2% up to 5% under varying environmental conditions such as temperature, supply voltage and nearby active components. Other experimental results [25] show a lower error rate (0.7% probability) but also indicate significant decreases in reliability under non-ideal environmental conditions. Those measurements do not take BER of transmission channels or possible error correction of the response into account. Thus a threshold of 98% modeling accuracy has been chosen to indicate successful modeling of the arbiter PUF.

Firstly, we want to reproduce and verify the results in [24]. There, simulated training datasets are used equivalent to our simulation, assuming normal distributed delays. The challenges used in training are transformed to parity-vector format (equation 2.1) before applying them to the neural network. The neural network used for modeling consists of a single hidden layer with five neurons, using the ReLU activation function. Using the same training process, we achieve a 98.86% modeling accuracy using the same number of 6800 training CRPs. This differs from the claimed 99.50% but is nonetheless sufficient for convincingly modeling the PUF. Since the number of training CRPs is an important metric, this model is evaluated using different training set sizes. Figure 4.1b shows that an accuracy of 98.06% is reached using only 3700 CRPs. Since the learning process is influenced by random weight initialization and does not converge to the exact same result over multiple runs, modeling has been repeated multiple times with the same dataset. Figure 4.1 shows only the maximum of all attempts. In figure 4.1a the same graph is shown for the 16-bit variant, which shows a similar curve for the prediction accuracy, although the entire graph is shifted to the left resulting in a much lower number of CRPs required to reach the threshold.



(a) 16-bit

(b) 64-bit

Figure 4.1: Prediction accuracy for 16- and 64-bit arbiter PUF for varying training set size.

Since the number of neurons and the single hidden layer seem to be chosen somewhat arbitrarily, we employ the automated hyperparameter tuning described in section 3.4 to test if any other ANN configuration may lead to better results. The hyperparameter tuning was repeatedly executed with varying training set sizes. It was found that just targeting the highest prediction accuracy using a training set that is already known to be sufficient to reach the desired minimum accuracy did not necessarily result in a configuration that achieves similar results using a smaller training set. Thus, the hyperparameter tuning was repeatedly executed with smaller training sets as long as the predefined threshold for the prediction accuracy was reached.The same procedure has been repeated for challenge sizes from 16- to 256-bit, the results of which are presented in table 4.1, with the last column showing the number of neurons in each layer for the configuration used.

Figure 4.2 shows the relation between the challenge length and the minimum number of CRPs with which it was possible to achieve a modeling accuracy of at least 98%. It is apparent that the relationship is almost perfectly linear, while the number of possible challenge grows exponentially with the challenge length. This confirms the inherent susceptibility of the arbiter PUF towards this attack, and provides a baseline that should be improved upon.

Table 4.1: Required CRPs for 98% prediction accuracy of the arbiter PUF after hyperparameter tuning.

Length	CRPs	Accuracy	ANN Architecture
16-bit	900	98.43%	[5,4,2]
32-bit	1600	98.33%	[4,4]
64-bit	3700	98.06%	[5]
128-bit	6400	98.31%	[7, 5]
256-bit	11700	98.01%	[5]

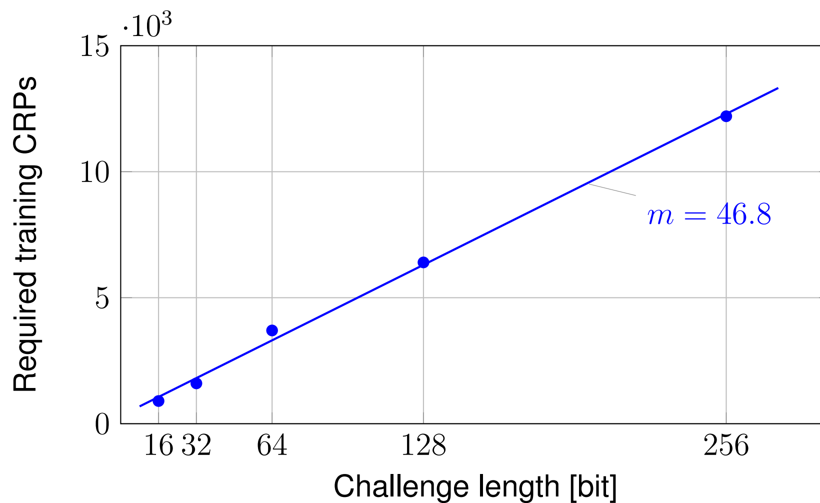


Figure 4.2: Required training set size for 98% prediction accuracy by challenge size of the arbiter PUF

4.1.1 Error prone training data

In the previous section, it has been established that the arbiter PUF is not 100% reliable in reality. While this has been used to justify the modeling threshold, the simulated arbiter PUF did not contain any noise-sources and produced a perfectly reliable output. In the following, bit-errors will be introduced into the response training data with a predefined rate, and modeling accuracy will be evaluated.

Figure 4.3 shows the prediction accuracy with different levels of error-rate in the training data. The accuracy is measured using the validation set, which does not contain erroneous responses. This implies that the ANN is tolerant against a noisy PUF response, as the prediction accuracy for example still reaches 97% even if

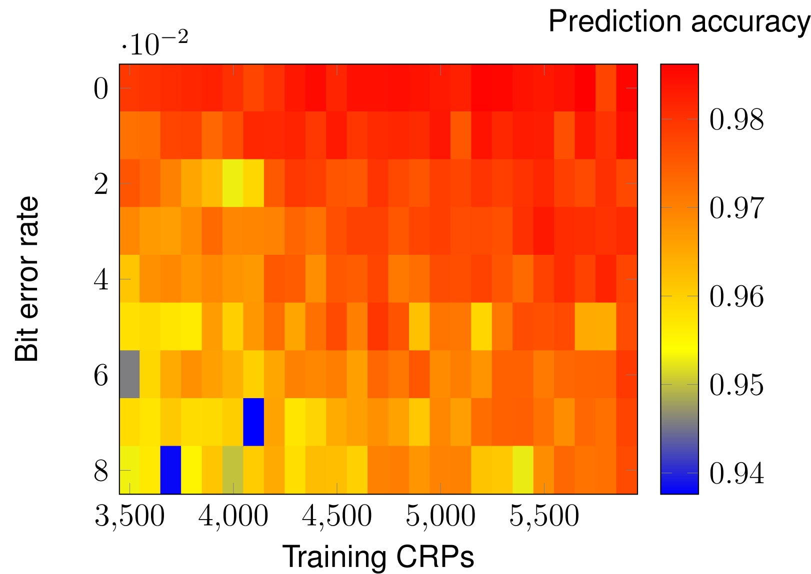


Figure 4.3: Accuracy of 64-bit arbiter PUF modeling with error in training data.

5% of the responses in the training set are erroneous. It is however evident that the reliability of the modeling attempts decreases rapidly with a rising BER, and a larger number of CRPs is required. This metric will not be the focus of the following PUF designs, but the implications of varying amounts of readout error and possible error correction measures during the authentication process on modeling attacks could be the subject of future work.

4.2 XOR Arbiter PUF

One approach that has been shown to improve the modeling resistance of arbiter PUFs is the XOR-arbiter PUF [25]. It employs multiple parallel arbiter PUFs, which all receive the same challenge. The outputs are connected to an XOR gate, which produces the final PUF response. The size of the PUF is directly controlled by the length of the challenge and the number of parallel arbiter PUFs used, both of which are also parameters to our simulation of this design. It has been suggested that the number of arbiter PUFs chosen should always be odd, as the output may exhibit bias otherwise [26].

The 64-bit XOR PUF has been chosen as a candidate for comparison to the plain arbiter PUF. Before attempting to optimize the results using hyperparameter tuning, a model which already promises some results shall be used. In [24], the 3-XOR PUF has successfully been modeled using a MLP with two hidden layers, containing seven and five neurons each. Figure 4.4 shows the modeling accuracy of the network after training with a varying number of CRPs. It can be observed that although the modeling is not perfectly reliable, using at least 22000 CRPs almost always results in a prediction accuracy of more than 98%. This is lower than the 36800 CRPs used in [24], which can be explained by the additional readout noise applied in those experiments, and by the higher achieved prediction accuracy of 99.22%. As already described in section 4.1, the necessary prediction accuracy can be chosen lower than that, which is why we keep the 98% threshold chosen there.

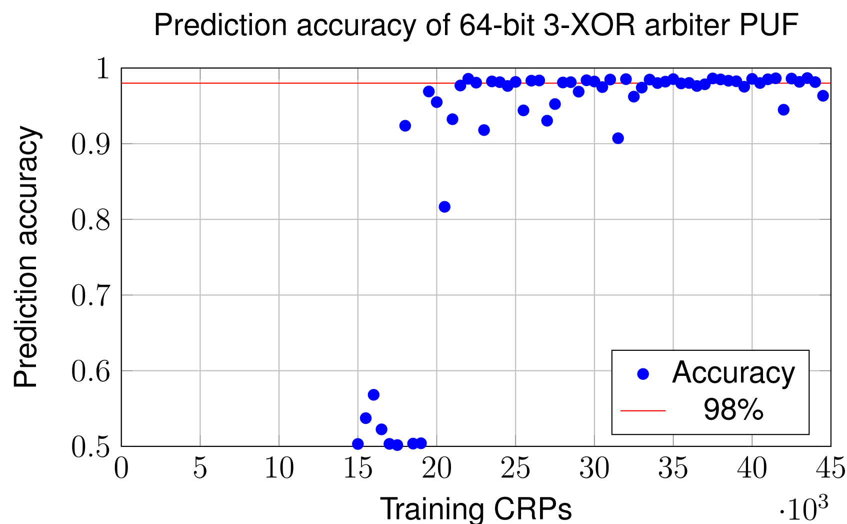


Figure 4.4: Prediction accuracy for 64-bit 3-XOR arbiter PUF.

The resistance of the XOR arbiter PUF towards modeling attacks with neural networks rises steeply when increasing the number of parallel arbiter PUFs. Using the approach of setting a threshold for the required modeling accuracy, and then applying the hyperparameter tuning technique did not lead to improved results over those in [24] when using four or more parallel arbiter PUFs. Table 4.2 shows the improved results for the 2- and 3-XOR PUFs, and the already available results for the 4- and 5-XOR PUFs. The improvements in the first two variants can be mainly attributed to the predefined target accuracy of 98%, which allowed the use of a sig-

Table 4.2: Required CRPs for learning the XOR arbiter PUF. Results for 4- and 5-XOR variants adopted from [24], since no further improvement was achieved.

Nr. of XORs	CRPs	Accuracy	Architecture
2	4700	98.67%	[2,18,5]
3	22000	98.57%	[7,5]
4	41200	98.60%	[15,15]
5	320000	97.23%	[29,29]

nificantly smaller training set of 4700 instead of 32.000 CRPs for the 2-XOR variant and 22 000 instead of 36 800 for the 3-XOR PUF. Even larger XOR PUFs require significantly more training data and larger neural networks to model, [24] proposes using a fully connected network of five layers with an average number of 230 neurons per layer to model the 6-XOR PUF with 97.42% accuracy. This does not only not reach the accuracy target used here, but also vastly exceeds the size of the search-space of the hyperparameter tuning efforts undertaken in this thesis. The number of trainable parameters in such a MLP sums up to over 220.000, which is far more than the number of variables used to describe the PUF completely in simulation. While this does not present an inherent problem, it may be an area of future work to employ special non-fully-connected networks which take advantage of properties of the PUF in order to simplify training of the ANN.

This shows that the XOR PUF does succeed in the goal of improving modeling resistance compared to the single arbiter PUF by introducing a nonlinearity. It is also easy to improve this resistance further by adding parallel arbiters at the cost of additional circuit area.

4.3 Staged Arbiter PUFs

In this section, three different types of PUF shall be evaluated. All of them are derived from the arbiter PUF, by combining multiple arbiter PUFs in a way that either obfuscates the PUF response, similar to the XOR-arbiter PUF, or the challenge. These measures are intended to circumvent the inherent modeling vulnerabilities of the arbiter PUF, arising from the linear delay model (see section 2.1).

4.3.1 V1 Arbiter PUF

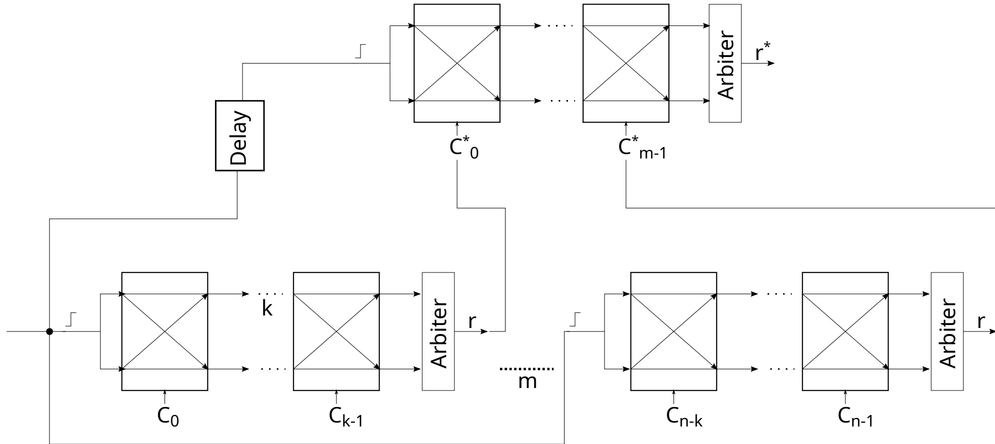


Figure 4.5: Circuit diagram of the V1 arbiter PUF with m first stage PUFs of length k, and n-bit total challenge size

The first arbiter PUF variant is very similar to the XOR arbiter PUF. As before, m arbiter PUFs in parallel form the first stage. In order to get a single response bit, another arbiter PUF is employed, whose m-bit challenge consists of the responses from the first stage PUFs. Contrary to a m-bit XOR, this introduces an additional circuit component containing inherent randomness. While a single arbiter PUF may be easy to model, this is usually under the assumption that the challenge is always known and it is possible to calculate a parity vector in order to solve the arbiter PUF as a linearly separable problem (equations 2.1 and 2.2). This is not possible in the V1 arbiter, as long as the assumption holds that the (internal) responses of the first stage are not accessible to the adversary. An additional parameter is whether the same challenge is applied to each first-stage PUF or if the challenge is split and the total challenge represents the concatenation of all first-stage challenges. The parameter k describes the length of each first-stage arbiter PUF. This means that in the case $k \cdot m = n$ (n being the total challenge length) each first-stage arbiter is controlled by a separate part of the challenge, and for $k = n$ that each first-stage arbiter PUF receives the same (entire) challenge. Other cases may be realized by overlapping the challenges, but are not considered here. A configuration will be described as $\mathrm{PUF}(k, n, m)$.

Special case $k = n$

In this design, the same challenge of length n is applied to m PUFs. The responses of those PUFs then form the challenge for the final m-bit PUF. The differences of this design compared to the XOR arbiter PUF shall be explored, especially with regards to different values of m, starting with $m = 2$ arbiter PUFs in the first layer.

In order to facilitate fair comparison to the other PUF designs, the targeted prediction accuracy should ideally be identical. It was however observed during initial experiments with this architecture, that modeling was very inconsistent, and a clear relationship between PUF parameters and required CRPs could not be determined.

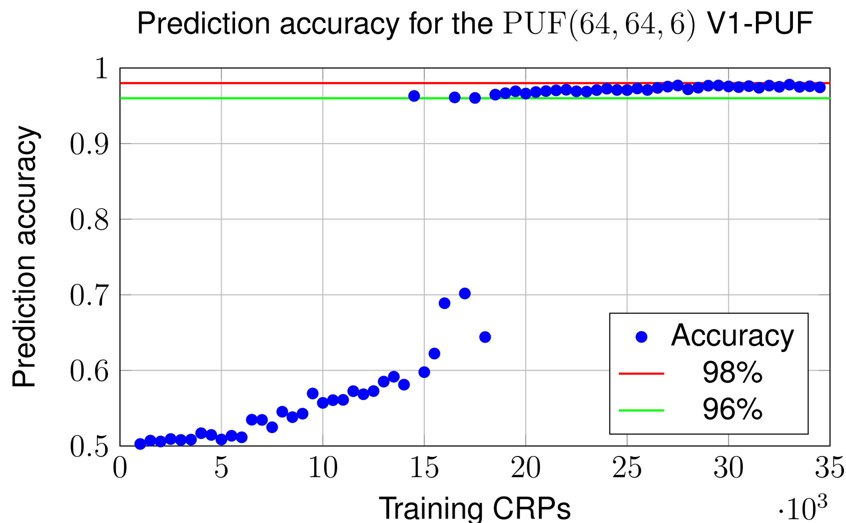


Figure 4.6: Prediction accuracy for the $\mathrm{PUF}(64, 64, 6)$ V1-PUF, comparing the previous threshold for successful modeling of 98% to the new threshold of 96%

This was due to the chosen accuracy threshold, which was chosen to match the arbiter PUF evaluation (section 4.1) at 98%. As apparent in figure 4.6, the modeling characteristics are still typical in the sense that a clear threshold for the number of required CRPs exists, the scale however is slightly different. It can be observed that the modeling accuracy plateaus before it reaches the threshold of 98%, which makes this value a poor choice for comparison. This is why in the following, the

threshold at which a modeling attempt is considered successful has been lowered to 96%. While this has to be taken into account when comparing this architecture to others, this allows a much clearer evaluation in terms of how the PUF responds to changing the aforementioned parameters.

Comparing the V1 configuration $\mathrm{PUF}(n, n, m)$ to the single arbiter PUF is done in terms of required circuit area, which is estimated using the number of required switching elements. For the arbiter PUF, this is equivalent to the challenge length, while for the V1 PUF, this is equal to $m \cdot n + m$, since m n-bit arbiter PUFs are needed in the first stage, as well as one m-bit arbiter PUF in the second stage. This way, both the cost of implementing and the number of random parameters are identical, and the modeling results are to be attributed to the difference in architecture. Similar to the XOR PUF, the challenge length n of the V1 PUF is kept constant, and only the number of parallel arbiter PUFs in the first stage (m) is varied.

The $\mathrm{PUF}(32, 32, 5)$ V1 variant consisting of five parallel 32-bit arbiter PUFs surpasses the single 128-bit arbiter PUF in terms of CRPs required, while using a comparable number of circuit elements (132 versus 128 switching elements). This comes at the cost of reduced challenge length however.

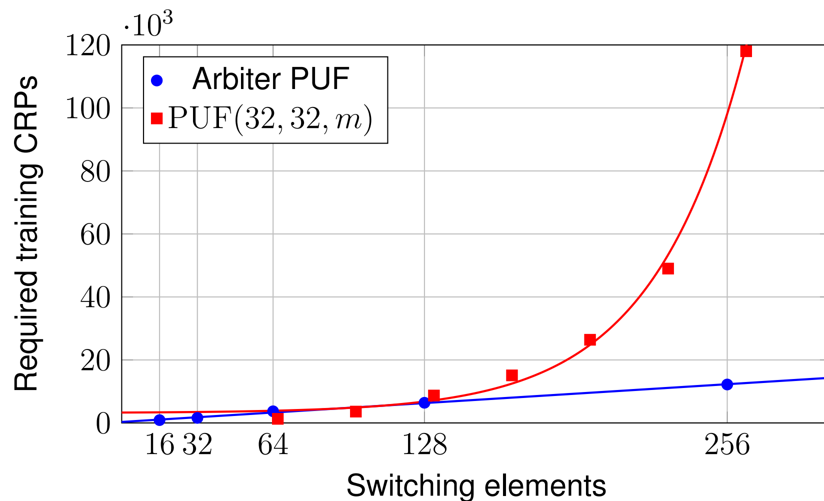


Figure 4.7: Comparison of required training set size by number of switching elements between the single arbiter PUF and the special case "$k = n$" of the V1 PUF

Table 4.3: Modeling results of the $\mathrm{PUF}(32, 32, m)$ V1 PUF, 96% accuracy threshold.

m	CRPs	Accuracy	Architecture
2	1300	96.12%	[3,8]
3	3600	96.53%	[4,11,10]
4	8700	96.61%	[5,16,17,11]
5	15100	96.11%	[9,12,15]
6	26400	97.02%	[11,16,17]
7	49000	96.94%	[18,17,17]
8	118000	96.85%	[17,20,16,6]

Figure 4.7 shows that in this special case the added delay elements in the form of parallel arbiter PUFs in the first stage do not have the same effect on learnability as adding the same number of switching elements to the end of an arbiter PUF. The number of required CRPs for successful modeling using ANNs increases at a higher rate for the V1 PUF compared to the arbiter PUF. It is also apparent that the increase does not happen strictly linear as observed with the single arbiter PUF: For the 32-bit variant, the number of required CRPs grows approximately exponentially, although especially with the 64-bit version it is apparent that this is only the case for $m \geq 5$, for smaller m the increase follows a more linear trend (figure 4.8).

It has already been mentioned that this configuration is very similar in its layout to the XOR PUF, with the only difference being that the m-input XOR is replaced by a m-bit arbiter PUF. Compared to the 64-bit XOR PUF with the same number of PUFs in the first stage (section 4.2), the V1 PUF can however be more easily modeled, using a smaller amount of CRPs. Compared to the XOR PUF it was also viable to model the PUF for values $m > 5$ using fewer CRPs and smaller ANNs. The modeling results for the V1 PUF of both 32- and 64-bit challenge length are listed in tables 4.3 and 4.4, and visualized in figure 4.8. The comparison in figure 4.8 also highlights that between the 32- and 64-bit V1 PUFs, the influence of the number of parallel first-stage PUFs m outweighs the impact of the challenge length n on modeling difficulty.

Table 4.4: Modeling results of the $\mathrm{PUF}(64, 64, m)$ V1 PUF, 96% accuracy threshold.

m	CRPs	Accuracy	Architecture
2	1600	96.33%	[3]
3	6200	96.16%	[6,10]
4	15100	99.59%	[2,13,9]
5	12600	96.53%	[7,10]
6	24300	96.56%	[11,13,18,12]
7	47000	96.65%	[12,17,17,20]
8	160600	96.94%	[15,20,18,16]

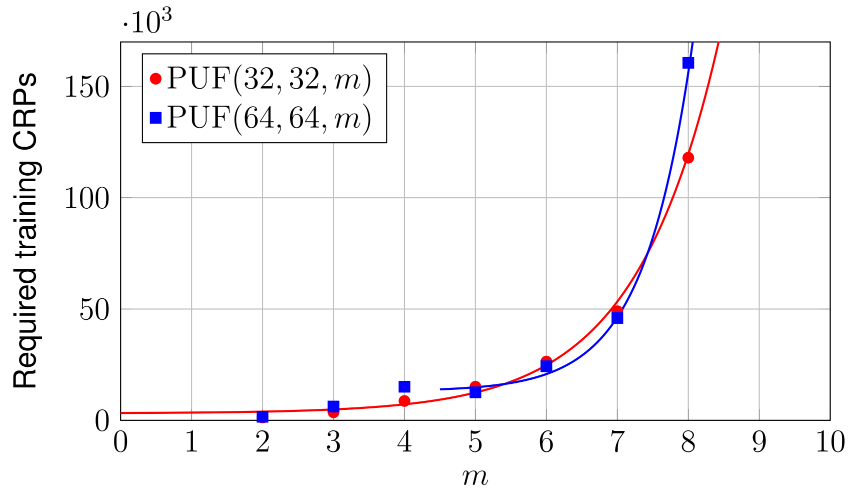


Figure 4.8: Required training set size for 98% prediction accuracy of the V1 PUF configuration $\mathrm{PUF}(n, n, m)$

Special case $k \cdot m = n$

The second special configuration option $k \cdot m = n$ explores the option to apply different parts of the challenge to different first-stage PUFs. The n-bit challenge is split into m challenges of k bit each. It is possible to either use many small first-stage PUFs, forming the challenge to a larger second-stage PUF, or to use a small number of larger first-stage PUFs, similar to the $k = n$ case. All tested configurations are listed in table 4.5.

Table 4.5: V1 PUFs with parameters $k \cdot m = n$.

Configuration $\mathrm{PUF}(k, n, m)$	CRPs	Accuracy	Architecture
$\mathrm{PUF}(1, 16, 16)$	300	96.66%	[7]
$\mathrm{PUF}(1, 32, 32)$	800	96.29%	[3,9]
$\mathrm{PUF}(1, 64, 64)$	1700	96.54%	[4]
$\mathrm{PUF}(2, 16, 8)$	1900	96.10%	[16]
$\mathrm{PUF}(2, 32, 16)$	threshold not reached		
$\mathrm{PUF}(2, 64, 32)$	threshold not reached		
$\mathrm{PUF}(64, 128, 2)$	2800	96.47%	[3]
$\mathrm{PUF}(32, 128, 4)$	95000	98.25%	[18]

The first three tested configurations consist of just a single switching element and arbiter in each first stage PUF ($k = 1$). Each challenge bit has the possibility of getting inverted by the 1-bit first stage element. The evaluation shows that even though the challenge input to the second stage PUF is not directly accessible, this configuration does not provide any increased modeling resistance when compared to the arbiter PUF. Increasing the size of the first stage PUFs to 2-bit challenges, while keeping the total challenge length consistent, seems to provide a much improved modeling resistance. While 16-bit version of this configuration is still easily modeled, both the 32-bit and 64-bit challenge length versions were not able to be modeled using the same approach as for the arbiter, XOR and first V1 PUFs with a comparable amount of CRPs. Since the second stage consists of a single 16- respectively 32-bit arbiter PUF which has been shown to be easily modeled in previous works and confirmed in section 4.1, it can be assumed that the first stage sufficiently obfuscates the input to that arbiter PUF to impede the modeling of the second stage. The last two configurations both accept a challenge of 128-bit length, and are constructed using a first stage of either two 64-bit arbiter PUFs of four 32-bit arbiter PUFs. Despite a difference in the number of switching elements of only two between both configurations, the $\mathrm{PUF}(64, 128, 2)$ setup with a smaller second stage is much more susceptible to modeling. This continues the trend of an increased modeling difficulty with an increasing number of parallel first-stage structures, as already observed when comparing the V1 PUF to the single arbiter PUF above and in figure 4.7.

Considering this specific PUF configuration however, concerns about the second type of attack mentioned in the introduction to chapter 4 have to be raised: In the $n = k \cdot m$ case, all of the PUF elements in the first stage are directly and individually accessible. This could open an attack vector in which the attacker only varies the challenge to a single first stage PUF, which keeps the remaining $m - 1$ intermediate responses constant (C^* in figure 4.5), reducing the problem to multiple m-bit arbiter PUFs with only one changeable challenge bit determined by the k-bit first stage PUF. As this requires well chosen challenges, the feasibility of such an attack is not examined here, but may present an area of future work.

4.3.2 V2 Arbiter PUF

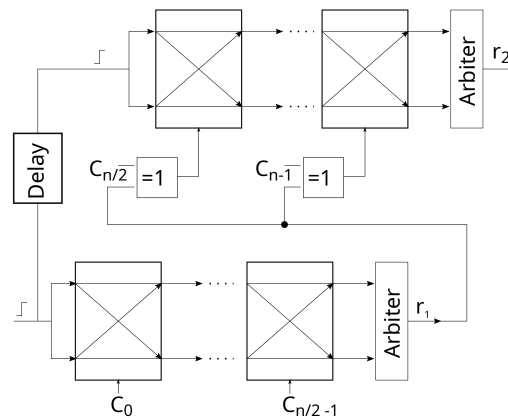


Figure 4.9: Circuit diagram of the V2 arbiter PUF variant with n-bit challenge

The approach to obfuscate the input to the last PUF stage is continued in the second arbiter PUF variant: The first half of the challenge is applied to the first arbiter PUF. The output of the first PUF then influences the challenge of the second PUF, by XOR with the second half of the initial challenge, as shown in figure 4.9. Again, it is important that the intermediate response r_1 is kept secret.

The second arbiter variation employs the same number of switching elements as the original arbiter PUF. To add an additional layer of indirection, the PUF is divided into two identical arbiter PUF modules, each with a size of half the total challenge length. The first half of the challenge is applied to the first stage, while the second half gets applied to the second arbiter PUF stage, but only after getting XORed with the response of the first PUF.

Table 4.6: V2 PUF modeling results.

Length	CRPs	Accuracy	Architecture
16-bit	1100	98.24%	[14]
32-bit	7400	98.68%	[5,16,11]
64-bit	12200	98.02%	[6,16]
128-bit	24700	98.11%	[6,13,12]
256-bit	38100	98.16%	[5,15,17]

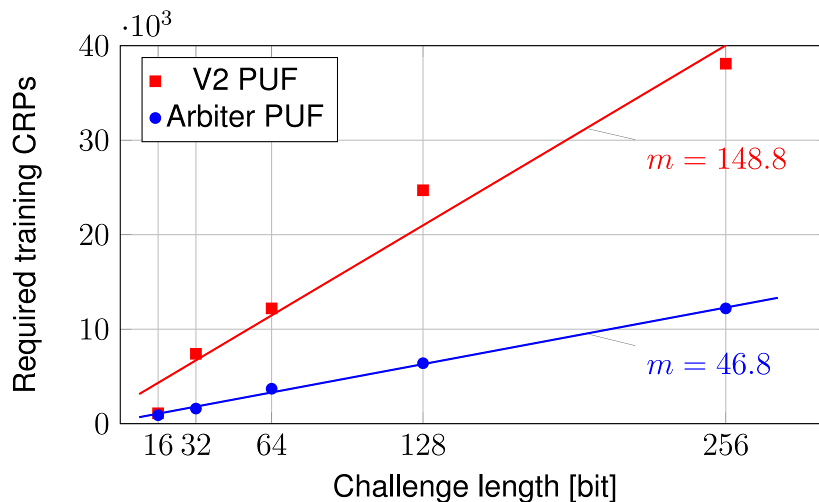


Figure 4.10: Required training set size for 98% prediction accuracy of the V2 PUF compared to the single arbiter PUF

As the final response results from an $n/2$-bit arbiter PUF, the V2 PUF is expected to perform at least better than that. Compared to the $n/2$-bit single arbiter PUF, the challenge is not applied directly, but is XORed with the response of the first PUF. This obfuscates the inner challenge to the second stage, as the response of the first stage is not exposed.

This design does seem to provide a direct improvement in modeling resistance compared to the arbiter PUF, as evident in figure 4.10. For each challenge length configuration from 16- to 256-bit, the V2 PUF required more than three times as many CRPs in order to achieve a similar level of prediction accuracy. Unlike the XOR or V1 variants however, the number of required CRPs does not grow faster than linear in the number of challenge bits or circuit elements.

It should also be mentioned again that attacks other than the one used here are possible with this specific design. Just as with some of the V1 variants, if the challenge can be arbitrarily chosen during the attack, it is possible to model the individual parts of the PUF by keeping the corresponding challenge bits constant. In the V2 architecture in particular, the intermediate result r_1 may be observed at the output with a well chosen second half of the challenge, which is kept constant. The second stage arbiter could also be modeled by varying only the second half of the challenge, as the influence of the first half only consists in flipping the entire challenge to the second stage if r_1 is one, which could be accounted for. Again, as this requires specially chosen challenges, such an attack is not covered in this evaluation.

4.3.3 V3 Arbiter

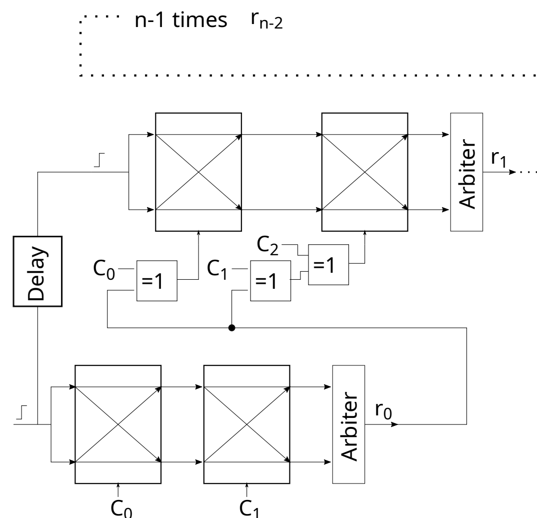


Figure 4.11: Arbiter PUF variation V3 for n-bit challenge.

The third variant consists of $n-1$ 2-bit arbiter PUFs for n-bit challenges. The response of each stage is used in the input of the following stage, along with one additional challenge bit. The simulation is only parameterized in terms of challenge length, but the size of each stage (length of each individual arbiter PUF) could also be adjusted. With $f_i(\cdot)$ representing the single arbiter PUF i, r_i the corresponding response and c_k bit k of the input challenge, the V3 PUF can be described as follows, where $\oplus$ describes XOR:

Table 4.7: Modeling results of the arbiter PUF variant V3, 98% accuracy threshold.

Length	CRPs	Accuracy	Architecture
16-bit	300	99.0%	[3,5]
32-bit	175	99.66%	[2,2]
64-bit	225	98.81%	[3]
128-bit	600	100%	[1,7]
256-bit	1100	98.6%	[2]

$$
\begin{aligned}
r_0 &= f_0(c_0, c_1) \\
r_1 &= f_1(r_0 \oplus c_0, r_0 \oplus c_1 \oplus c_2) \\
r_2 &= f_2(r_1 \oplus c_1, r_1 \oplus c_2 \oplus c_3) \\
&\vdots \\
r &= f_{n-2}(r_{n-3} \oplus c_{n-3}, r_{n-3} \oplus c_{n-3} \oplus c_{n-2} \oplus c_{n-1})
\end{aligned}
\tag{4.1}
$$

This architecture combines the individual arbiter PUFs in a more sequential approach, with the aim that a neural network capable of modeling the PUF needs a greater depth than previously which requires a greater training set to optimize.

Additionally, the challenge to all stages except the first is not directly known. This should improve security as it is not possible to calculate the parity feature-vectors for these stages, which usually enable efficient modeling of the arbiter PUF.

In order to evaluate these hypotheses, simulated CRP data is generated for the PUF architecture as shown in figure 4.11, for challenge lengths n from 16 to 256. The same iterative modeling approach of using the automated hyperparameter tuning until the prediction accuracy reaches the threshold of 98% is used. Training was again performed using the parity feature vectors (equation 2.1).

Table 4.7 displays the results of this approach. The second column displays the minimum number of training CRPs that we were able to perform successful modeling with. As evident in table 4.7, the hypothesis of an intrinsic modeling resistance due to the sequential "deep" layout can be firmly rejected. Not only does the proposed architecture not improve modeling resistance compared to the single arbiter

PUF, our experiments show that modeling requires a significantly lower number of training CRPs to achieve the same modeling accuracy (figure 4.12). These numbers and the results of the hyperparameter tuning indicate a systematic flaw in this particular design, and further investigation into the reason behind this is needed.

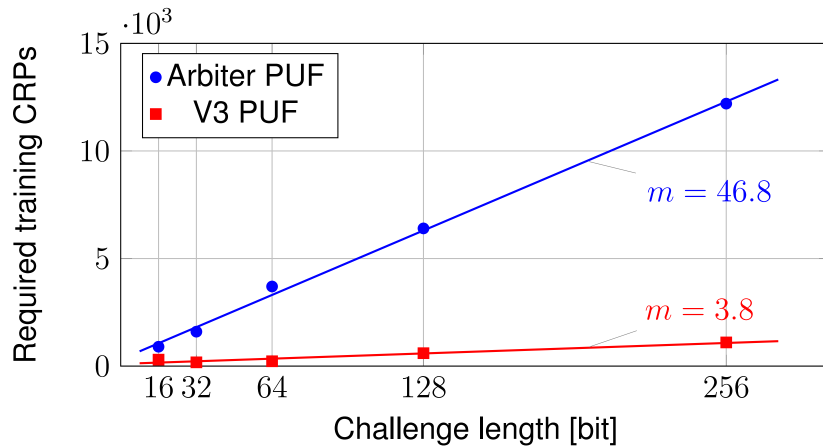


Figure 4.12: Required training set size for 98% prediction accuracy of the V3 PUF

4.4 Loop PUF

The Loop PUF as described in section 2.1.2 has the distinct difference to the other PUF designs described here that depending on parametrization, multiple response bits can be generated for one challenge. The PUF contains N delay chains of M delay elements each, which each gets controlled by one challenge bit resulting in a $n = N \cdot M$ bit challenge. N is also the number of response bits when using the control strategy suggested in [1]. Since the ANN response can no longer be regarded as a binary classification problem, the previously used loss function of binary cross-entropy can no longer be used. Instead, the MSE over all response bits is minimized in this case. Accuracy is still used as a metric, representing the fraction of correct versus total predictions. Following the approach of the authors in [1], N is chosen as three, while M is varied. This keeps the number of response bits consistent and allows variation of the challenge length.

Table 4.8: Modeling results of the $N = 3$ Loop PUF.

M	n	CRPs	Accuracy	Architecture
6	18	30000	98.46%	[11,9,2]
11	33	67500	98.07%	[5,12,3,1]
16	48	313600	98.70%	[17,18,2]
22	66	347600	98.17%	[4,14,1]
32	96	712500	98.15%	[20,9,17,6]
43	129	$\geq$900000	$\leq$97.72%	

The distinct property of the Loop PUF of having a response of more than one bit makes direct comparison to previous results difficult. While the modeling results in table 4.8 suggest that the $M = 22$ Loop PUF, which accepts a 66-bit challenge, is similar to the 5-XOR 64-bit PUF (section 4.2) in modeling difficulty, it has to be considered that the arbiter-based PUF variants only produce one response bit per challenge. Depending on the requirements of the authentication process, this might mean that for a single authentication attempt, the arbiter PUF requires more read-outs and thereby more challenges transmitted, which benefits the eavesdropping attacker.

Figure 4.13 shows the required number of CRPs for modeling in relation to the challenge length. It can be seen that for larger challenges, this number grows much faster than observed with the arbiter PUF or the improved V2 PUF. In contrast to the V1 and XOR PUF, the increase in modeling difficulty also occurs alongside an increase in challenge length, which may be desired. Since the sources of randomness in this architecture are not more than in the arbiter PUF of same challenge length, these results can be attributed to the controller which defines how the same challenge is applied to the PUF multiple times and how multiple response bits are generated. Since the exact functionality of the controller is known to an attacker, an area of future work could be to include that knowledge into the modeling process and thus constrain modeling to the unknown, random parts of the PUF.

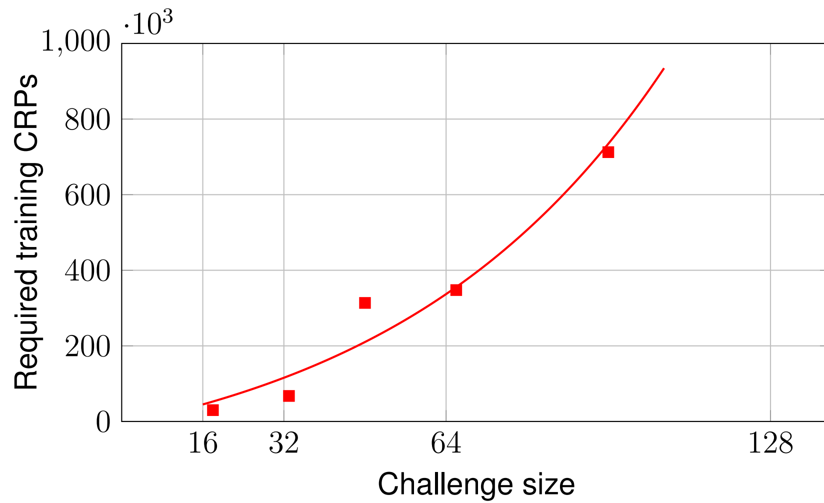


Figure 4.13: Modeling results of the $N = 3$ Loop PUF.

4.5 Comparison

To provide a quick overview over the results of the selected PUF architectures, figure 4.14 and figure 4.15 are provided. Figure 4.14 compares the single arbiter PUF (section 4.1) with the V2 (section 4.3.2), and V3 (section 4.3.3) architecture, which have in common that the circuit area requirements are directly proportional to the challenge size. Only architecture V2 was found to provide an improved modeling resistance over the arbiter PUF.

Figure 4.15 compares the V1 PUF (section 4.3.1) to the XOR PUF of equivalent challenge length. Both use a varying number of parallel elements in the first stage, and it is apparent that the modeling resistance grows quickly when increasing this parameter. Despite the additional random elements in the second stage, XOR PUF still required a greater number of CRPs to model than the V1 PUF in a similar configuration.

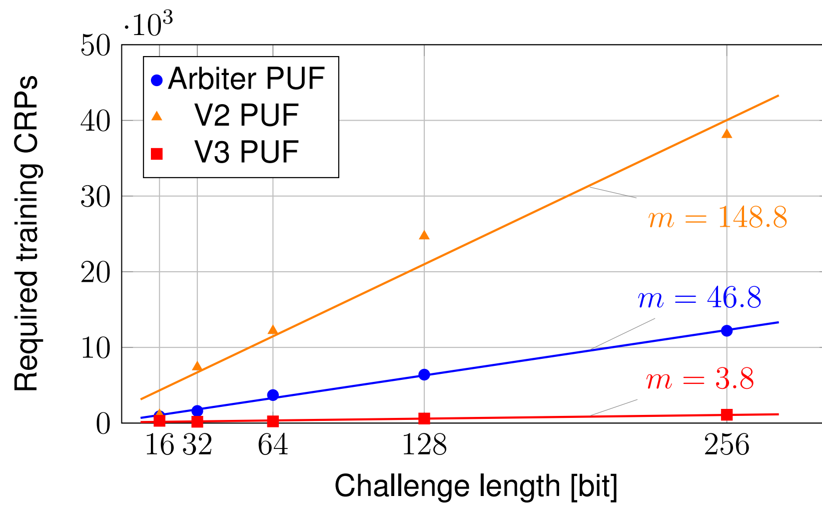


Figure 4.14: Comparison of the V2 and V3 PUFs with the single arbiter PUF by challenge length.

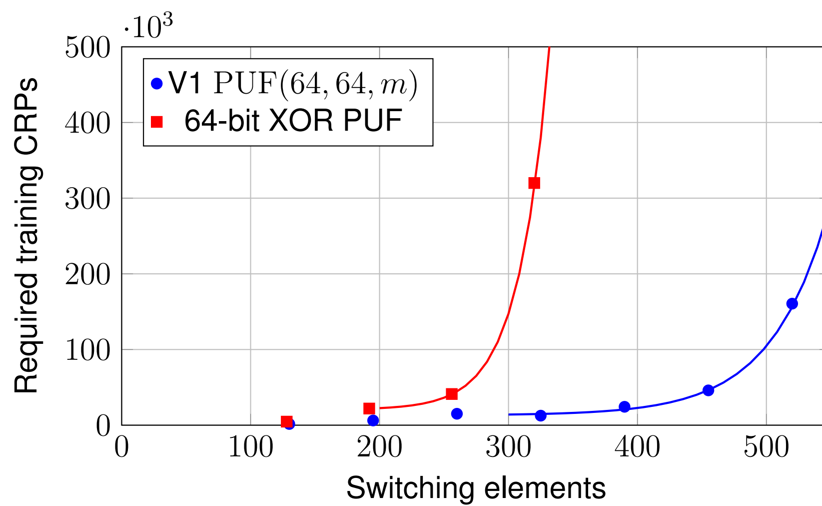


Figure 4.15: Comparison of the XOR and V1 PUFs, both with 64-bit challenge length, by required switching elements.

5 Conclusion and Outlook

The goal of this thesis was the implementation of a program for attacking physical unclonable functions using neural networks, as well as examining a way of measuring the resistance to such attacks in a comparable manner. The attacks are executed from the point of an adversary that can eavesdrop on CRPs during authentication and has knowledge of the PUF design, but no direct access to the device under attack.

For the first part, a framework has been realized in python, utilizing tensorflow as a standard library for machine learning tasks. It allows a researcher to use simulated or measured challenge-response datasets in order to examine the vulnerability of the PUF design against modeling. The artificial neural networks used are fully connected multi layer perceptrons, which require few assumptions about the function being modeled and are still a viable choice here due to the small size of the required networks. Automated hyperparameter tuning has been used and made available to the user to remove the variable of the network architecture from any comparisons.

In the second part, multiple PUF designs have been compared in terms of the required amount of challenge-response-pairs that an attacker would have to acquire in order to succeed at modeling the PUF using the presented techniques. Four of the evaluated designs are variants and combinations of the arbiter PUF architecture: The V1 variant (section 4.3.1) which is similar in design to the XOR arbiter PUF (section 4.2) also increases modeling difficulty compared to the single arbiter PUF in a similar manner, where the relationship between required CRPs and circuit area of the PUF is exponential rather than linear. The V2 PUF also increases modeling difficulty above the arbiter PUF, although the number of required CRPs still grows linearly with the challenge length. No improvement has been found in the V3 architecture, which could be modeled using even less CRPs than the single arbiter PUF. The last PUF architecture tested is the Loop-PUF, which differs from the other

architectures in that it is based on a ring oscillator with configurable delay elements. It also employs additional control structures which enables generating multiple response bits for each challenge. While this makes comparison to the other PUFs difficult, modeling was only achieved using a large amount of CRPs, which required expensive configurations of the XOR or V1 arbiter architectures.

The measure of the minimum number of CRPs required to achieve a predefined prediction accuracy has been used throughout the evaluation and provides a good indication of the security of a PUF against this kind of attack.

Areas of future work could include using in-depth knowledge about a PUF design in order to include fixed architectural features in the ANN architecture, reducing the number of parameters to be learned. Additionally, for some of the evaluated PUF designs other attacks may be more feasible if the possibility of applying well chosen challenges is given, which should be considered.

Bibliography

[1] Zouha Cherif et al. "An Easy-to-Design PUF Based on a Single Oscillator: The Loop PUF". In: *2012 15th Euromicro Conference on Digital System Design*. 2012 15th Euromicro Conference on Digital System Design (DSD). Cesme, Izmir, Turkey: IEEE, Sept. 2012, pp. 156–162. ISBN: 978-0-7695-4798-5 978-1-4673-2498-4. DOI: 10.1109/DSD.2012.22. URL: http://ieeexplore.ieee.org/document/6386887/ (visited on 07/19/2020).

[2] Jeroen Delvaux et al. "Secure Lightweight Entity Authentication with Strong PUFs: Mission Impossible?" In: *Advanced Information Systems Engineering*. Ed. by Camille Salinesi, Moira C. Norrie, and Óscar Pastor. Red. by David Hutchison et al. Vol. 7908. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 451–475. ISBN: 978-3-642-38708-1 978-3-642-38709-8. DOI: 10.1007/978-3-662-44709-3_25. URL: http://link.springer.com/10.1007/978-3-662-44709-3_25 (visited on 07/20/2020).

[3] Blaise Gassend et al. "Identification and authentication of integrated circuits". In: *Concurrency and Computation: Practice and Experience* 16.11 (Sept. 2004), pp. 1077–1098. ISSN: 1532-0626, 1532-0634. DOI: 10.1002/cpe.805. URL: http://doi.wiley.com/10.1002/cpe.805 (visited on 10/11/2020).

[4] Blaise Gassend et al. "Silicon physical random functions". In: *Proceedings of the 9th ACM conference on Computer and communications security - CCS '02*. the 9th ACM conference. Washington, DC, USA: ACM Press, 2002, p. 148. ISBN: 978-1-58113-612-8. DOI: 10.1145/586110.586132. URL: http://portal.acm.org/citation.cfm?doid=586110.586132 (visited on 11/10/2020).

[5] Charles Herder et al. "Physical Unclonable Functions and Applications: A Tutorial". In: *Proceedings of the IEEE* 102.8 (Aug. 2014), pp. 1126–1141. ISSN: 0018-9219, 1558-2256. DOI: 10.1109/JPROC.2014.2320516. URL: http://ieeexplore.ieee.org/document/6823677/ (visited on 08/17/2020).

[6] Andreas Herkle et al. "In-depth Analysis and Enhancements of RO-PUFs with a Partial Reconfiguration Framework on Xilinx Zynq-7000 SoC FPGAs". In: (2019). In collab. with Universität Ulm. Publisher: Universität Ulm. DOI: 10.18725/OPARU-17952. URL: https://oparu.uni-ulm.de/xmlui/handle/123456789/18009 (visited on 06/07/2020).

[7] Daniel E. Holcomb, Wayne P. Burleson, and Kevin Fu. "Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers". In: *IEEE Transactions on Computers* 58.9 (Sept. 2009). Conference Name: IEEE Transactions on Computers, pp. 1198–1210. ISSN: 1557-9956. DOI: 10.1109/TC.2008.212.

[8] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (Jan. 1989), pp. 359–366. ISSN: 08936080. DOI: 10.1016/0893-6080(89)90020-8. URL: https://linkinghub.elsevier.com/retrieve/pii/0893608089900208 (visited on 10/16/2020).

[9] Katarzyna Janocha and Wojciech Marian Czarnecki. "On Loss Functions for Deep Neural Networks in Classification". In: *arXiv:1702.05659 [cs]* (Feb. 18, 2017). arXiv: 1702.05659. URL: http://arxiv.org/abs/1702.05659 (visited on 11/11/2020).

[10] Stefan Katzenbeisser et al. "PUFs: Myth, Fact or Busted? A Security Evaluation of Physically Unclonable Functions (PUFs) Cast in Silicon". In: *Cryptographic Hardware and Embedded Systems – CHES 2012*. Ed. by Emmanuel Prouff and Patrick Schaumont. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 283–301. ISBN: 978-3-642-33027-8.

[11] Patrick Kidger and Terry Lyons. "Universal Approximation with Deep Narrow Networks". In: *arXiv:1905.08539 [cs, math, stat]* (June 8, 2020). arXiv: 1905.08539. URL: http://arxiv.org/abs/1905.08539 (visited on 10/16/2020).

[12] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *arXiv:1412.6980 [cs]* (Jan. 29, 2017). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980 (visited on 10/19/2020).

[13] Sharad Kumar and Mohammed Niamat. “Machine Learning based Modeling Attacks on a Configurable PUF”. In: *NAECON 2018 - IEEE National Aerospace and Electronics Conference*. NAECON 2018 - IEEE National Aerospace and Electronics Conference. Dayton, OH: IEEE, July 2018, pp. 169–173. ISBN: 978-1-5386-6557-2. DOI: 10.1109/NAECON.2018.8556818. URL: https://ieeexplore.ieee.org/document/8556818/ (visited on 05/20/2020).

[14] Lisha Li et al. “Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization”. In: *arXiv:1603.06560 [cs, stat]* (June 18, 2018). arXiv: 1603.06560. URL: http://arxiv.org/abs/1603.06560 (visited on 10/14/2020).

[15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: http://tensorflow.org/.

[16] Marvin Minsky and Seymour Papert. *Perceptrons: an introduction to computational geometry*. Expanded ed. Cambridge, Mass: MIT Press, 1988. 292 pp. ISBN: 978-0-262-63111-2.

[17] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com (visited on 11/11/2020).

[18] Chigozie Nwankpa et al. “Activation Functions: Comparison of trends in Practice and Research for Deep Learning”. In: *arXiv:1811.03378 [cs]* (Nov. 8, 2018). arXiv: 1811.03378. URL: http://arxiv.org/abs/1811.03378 (visited on 10/10/2020).

[19] Ulrich Rührmair et al. “Modeling attacks on physical unclonable functions”. In: *Proceedings of the 17th ACM conference on Computer and communications security - CCS ’10*. the 17th ACM conference. Chicago, Illinois, USA: ACM Press, 2010, p. 237. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866335. URL: http://portal.acm.org/citation.cfm?doid=1866307.1866335 (visited on 08/23/2020).

[20] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. "Searching for Activation Functions". In: *arXiv:1710.05941 [cs]* (Oct. 27, 2017). arXiv: 1710.05941. URL: http://arxiv.org/abs/1710.05941 (visited on 10/09/2020).

[21] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 1939-1471, 0033-295X. DOI: 10.1037/h0042519. URL: http://doi.apa.org/getdoi.cfm?doi=10.1037/h0042519 (visited on 10/10/2020).

[22] Ulrich Ruhrmair and Daniel E. Holcomb. "PUFs at a glance". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*. Design Automation and Test in Europe. Dresden, Germany: IEEE Conference Publications, 2014, pp. 1–6. ISBN: 978-3-9815370-2-4. DOI: 10.7873/DATE.2014.360. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6800561 (visited on 08/17/2020).

[23] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/323533a0. URL: http://www.nature.com/articles/323533a0 (visited on 10/08/2020).

[24] Pranesh Santikellur, Aritra Bhattacharyay, and Rajat Subhra Chakraborty. *Deep Learning based Model Building Attacks on Arbiter PUF Compositions*. 566. 2019. URL: http://eprint.iacr.org/2019/566 (visited on 05/24/2020).

[25] G. Edward Suh and Srinivas Devadas. "Physical unclonable functions for device authentication and secret key generation". In: *Proceedings of the 44th annual conference on Design automation - DAC '07*. the 44th annual conference. ISSN: 0738100X. San Diego, California: ACM Press, 2007, p. 9. ISBN: 978-1-59593-627-1. DOI: 10.1145/1278480.1278484. URL: http://portal.acm.org/citation.cfm?doid=1278480.1278484 (visited on 10/11/2020).

[26] Nils Wisiol and Niklas Pirnay. “Short Paper: XOR Arbiter PUFs Have Systematic Response Bias”. In: *Financial Cryptography and Data Security*. Ed. by Joseph Bonneau and Nadia Heninger. Vol. 12059. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 50–57. ISBN: 978-3-030-51279-8 978-3-030-51280-4. DOI: 10.1007/978-3-030-51280-4_4. URL: http://link.springer.com/10.1007/978-3-030-51280-4_4 (visited on 10/19/2020).

Name: Jonas Otto Matrikelnummer: 982249

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den ....

Jonas Otto