What Makes A Good PUF?

PUF-Modeling Attacks Final Report

Arya Shetty
Professor: Sheng Wei

***Abstract*—This development project demonstrates how effective modeling attacks are on a PUF (Physically Unclonable Function). It also discusses the results from various tests as well as recommendations and findings for the future of PUF security.**

***Index Terms*—PUF, Hardware Security, Machine Learning**

I. Introduction

Physically Unclonable Functions, PUFs, are still an essential part of hardware security. Ensuring the technology can still stand as a useful authenticator when it comes to data ownership requires testing. Of course, recreating a PUF physically is almost impossible, but mimicking the structure and using machine learning based attacks can predict and even replicate its responses. After learning about the security in Lecture 2, I decided to delve deeper into the topic. Since machine learning attacks continued to pose a threat, I decided to make my development project around the modeling attacks themselves, and what we could learn from them to improve PUF security.

II. Technical background

A research paper conducted a similar experiment. They used the architecture from numerous PUFs to get the challenge-response data, CRP, and then they tested on a variety of machine learning attacks. [2] The paper mainly used Linear Regression for its modeling attacks. If the PUF was too difficult to represent, they used Evolution Strategies, mimicking how the general population evolves, a sort of black box strategy. Most of their attacks were successful given enough time, data, and pre-processing. Eventually, they came to some conclusions to help improve PUF security, which I will review in my own results evaluation.

I did not copy the exact experiment from the paper since I used my own attack and PUFs. I used the Arbiter, XOR, and Lightweight PUFs for my experiment. The Arbiter PUF is the first simple PUF we learned in class. To briefly go over it, the hardware architecture is comprised of two paths of multiplexers, and at the end of the paths is a flip-flop that determines which signal reached the end first. To use PUFs for authentication, a challenge is sent, made from a set of bits, and the PUF takes in the challenge and sends out a response. The bits determine whether the signal stays on the same path or switches, which changes the final output response to 0 or 1. The XOR PUF is similar in that it is made of multiple ”k” chains of Arbiter PUFs, but before the response is outputted, the signals are sent to an XOR gate, and the output is changed again, adding a layer of complexity. Finally, the lightweight PUF keeps the same format as the XOR PUF, but each chain uses a different set of challenge bits, further adding to the non-linear complexity.

I used linear regression, a perceptron, and a simple neural network for the attacks. Linear regression, in this case, is trying to find the best-fit straight line for the challenge-response data. Perceptron attacks are also linear, but their output is based on binary classification, so the responses don’t need to be fitted since they will be 0 or 1, accurate to the response. Finally, the neural network is multiple layers of perceptrons, which allows it to learn more complex patterns since it updates its weights as it passes through the layers, helping identify patterns more easily. More iterative learning helps it adjust the loss for predictions to be more accurate to the ground truth.

III. System setup and implementation

My setup was all software-based for this experiment. Since I didn’t have access to an actual PUF and didn’t have the actual timing delays like the paper, I found a Python library to simulate a PUF. [1] The pypuf library provides a toolbox for simulating a variety of PUFs, generating CRPs for training, and even testing PUFs on user-created challenges. Regarding the actual machine learning of the attack, I used PyTorch since I have used it for previous machine learning projects. For the basic pipeline, I split the attacks into three different scripts. In each script, I construct the model, and then, to determine which PUF I use to run the attack on, I have a parameter that the user can choose before they run the script. Once set, the script first creates the PUF with the library with some set parameters I have in the code, which are for deciding the chains, noise, bits, and such. Then I create the CRPs for the specific PUF. Next, if input mapping is turned on, the CRPs are preprocessed for the PUF’s architecture. For example, for the Arbiter and XOR PUF, the raw challenge bits are looped through, and for each position in the challenge set, the feature is calculated by multiplying all the subsequent bit positions. The end result is the same length bit set, but with the parity features calculated for each position, indicating the effect of the delay on the final response. For the lightweight input mapping, I am still working on it, but for now, I have set up the input mapping to run twice to simulate the sort of challenge transformation the PUF goes through for each chain. Next, the attack is trained on the challenges and responses for a certain number of epochs, and the graph is saved. Then the accuracy testing function is run on a new set of challenges the

model has not trained on; again, the results are saved. From here, there are a few folders in the directory that correspond to the attack and the number, so you can easily check the training results. This same process is held for each attack, and they all have their own folders for graphs, training parameters, results, and saved models. To create the attacks, I used Python libraries, sklearn and pytorch, and the framework they have to train linear regression. For the perceptron attack, I create a single-layer network with sigmoid activation, and use SGD (Stochastic Gradient Descent) optimizer and BCE (Binary Cross Entropy) loss. This was effective since the responses were just 1 or 0 for the challenge bits. Finally, the neural network was similar, but I had four fully connected layers with dropout to try and increase the patterns the model would learn and prevent overfitting.

IV. Evaluation Results

When beginning the PUF modeling attacks, I started with the Arbiter PUF without input mapping. You can see from the results that it performed very poorly.

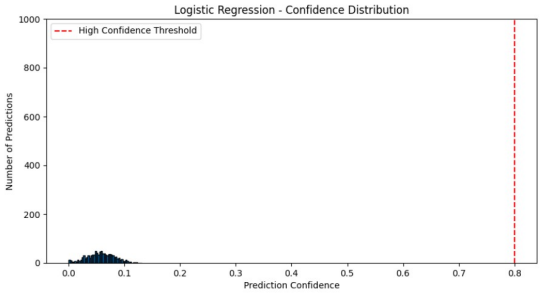

Fig. 1: Logistic Regression attack on Arbiter PUF. Trained without input mapping, the model performed poorly with a 0.5440 attack accuracy, and none of its predictions were confident.

For reference, the confidence is calculated by taking the prediction, subtracting 0.5, then multiplying by 2 to get back on the response scale, which is 0 or 1. If it's 0 or 1 before the conversion, it will remain "confident", while being in between would mean it's not; I checked by making sure it's at least greater than 0.8 (the red line on the graph).

I soon realized that when the paper was trained on PUFs, the data was preprocessed before it started. [2] Since I knew the architecture of the actual Arbiter PUF, I tried to employ its unique architecture and give the data more of a recognizable pattern, which can be seen in the system setup section. After retraining with the input mapped data, the results were much better since the model was able to learn what was actually affecting the responses of the PUFs.

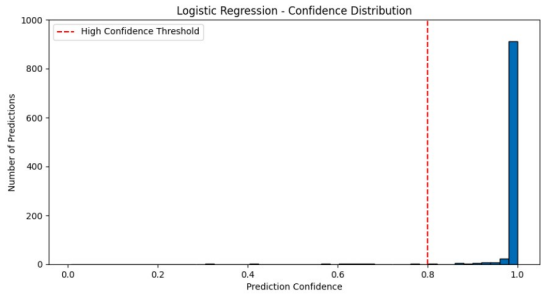

Fig. 2: Logistic Regression attack on Arbiter PUF. Trained with Arbiter input mapping, the model performed great with 968 high-confidence predictions.

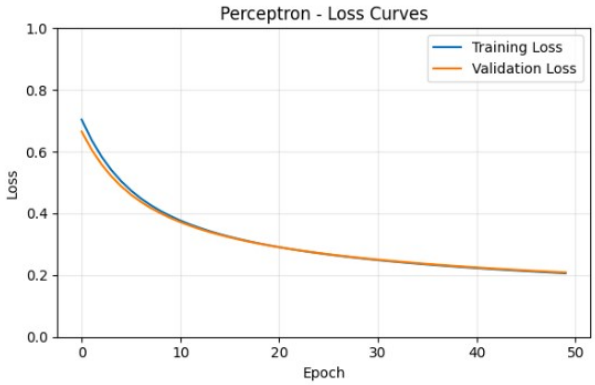

Fig. 3: Perceptron attack on Arbiter PUF. Trained with Arbiter input mapping, the model was good with 0.9970 attack accuracy.

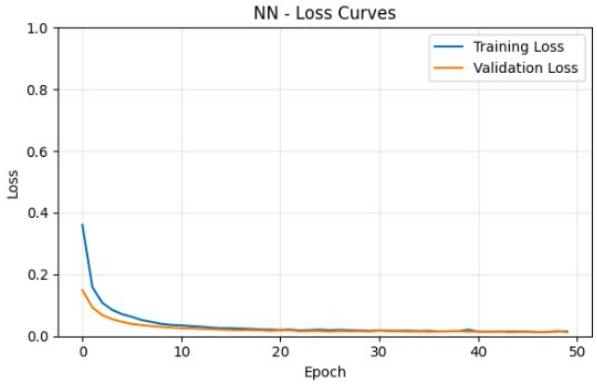

Fig. 4: Neural Network attack on Arbiter PUF. Trained with Arbiter input mapping, had 988 high-confidence predictions.

Since the Arbiter PUF was working well, I saved the best results and moved on to the XOR PUF. Here, the input mapping remained the same as the architecture was similar. However, the additional XOR gate was not fully covered in this mapping. The data remained non-linear and too complex to learn for the first two attacks; however, the neural network had no trouble learning and was accurate in testing.

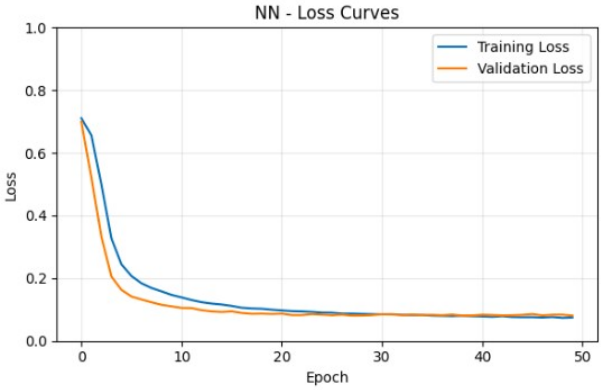

Fig. 5: Neural Network attack on XOR PUF. Trained with XOR input mapping, had 0.9560 attack accuracy.

This was most likely due to the neural network's advantage, the iterative learning. The multiple layers helped the model learn the XOR gate's effect on the response. After cracking two PUFs, I moved to the lightweight PUF. This PUF's complexity would already be too much for the first two attacks, but without proper input mapping, the neural network could not seem to replicate its behavior either. I am still working on an appropriate input mapping for this PUF since each chain's additional random challenge sets are complex to recreate, as you can see by the training graph.

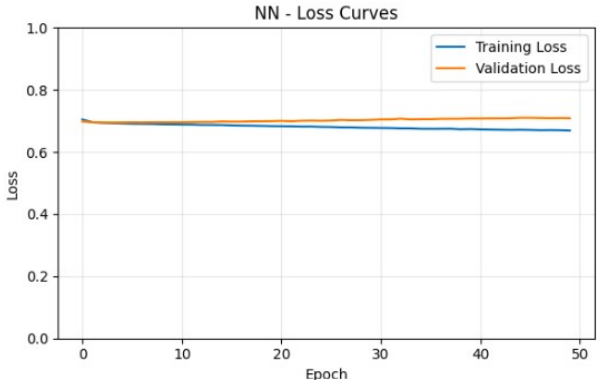

Fig. 6: Neural Network attack on Lightweight PUF with 4 chains. Input mapping is still not great, as it doesn't recognize the challenge bit transformation.

I ran some additional tests to check whether changing specific parameters would affect the overall learning of the models. Increasing the dataset was beneficial, but it was quite slow to run on my hardware, so it wasn't a viable option, but I'm sure it would be better to learn from. Increasing the bit size of the challenge sets to 128 bits did decrease the training accuracy, but it wasn't enough when the model was already performing effectively with the correct pre-processing. Inserting generated noise did not affect results all too much either. However, using additional chains in the XOR PUF, specifically four and above, was too much for the neural network to handle, and it could not learn the behavior of the PUF.

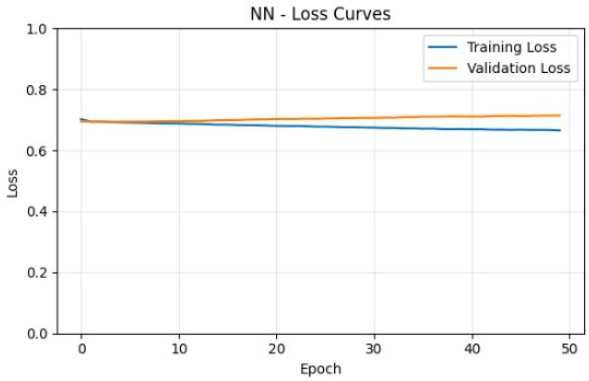

Fig. 7: Neural Network attack on 8 chain XOR PUF as opposed to the 2 chain. Performed very poorly with 0.4990 test accuracy.

TABLE I: Attack Results on Arbiter PUF (100000 CRPs)

	Linear Regression	**Perceptron**	**NN**
Train Loss	0.03	0.21	0.02
Test Accuracy (%)	100	99.7	99.7
High Confidence Predictions (x/1000)	1000	444	988

TABLE II: Attack Results on XOR PUF (4 chains)

	Linear Regression	**Perceptron**	**NN**
Train Loss	0.7	0.69	0.25
Test Accuracy (%)	51.6	52.2	95.6
High Confidence Predictions (x/1000)	0	0	675

TABLE III: Attack Results on Lightweight PUF

	NN
Train Loss	0.67
Test Accuracy (%)	48.4
High Confidence Predictions (x/1000)	0

Overall, I came to conclusions similar to those in the paper when conducting these tests. Firstly, from the attack perspective, having to preprocess the data and define features is very important because replicating PUF behavior becomes nearly impossible without it. With this set, most other parameters like noise and such would not affect the training results too much.

In terms of the defense, there are a number of things PUF engineers can do to thwart attacks. I found the same results as the paper in that increasing the bit size of the challenge sets and the number of parallel chains in PUFs can significantly hurt simpler training models. However, this does come with the increased overhead of more hardware to run. I believe another effective way would be to use different challenge bits

since it adds the same layer of complexity but does not require additional delay gates.

I will continue testing different PUFs and improving the attacks, but concerning the security of PUFs, there are still many ways to fend off simple machine learning attacks.

V. Source code

If you want to check more of the results or even try running the training scripts, I have attached them with this report, and they can also be accessed from my GitHub repository. [3] The README markdown should briefly explain how to run the code and reproduce the results from this experiment, as well as bug fixes. When you get familiar with running the scripts, you can easily check the "params.txt" files in each of the folders to find out how to get the specific results from their respective graphs.

Acknowledgment

I want to thank you for teaching this class. I enjoyed your lectures and hope you continue to offer this course for future classes.

References

[1] PyPUF Contributors. Pypuf documentation - simulation overview, 2024.

[2] U. Ruhrmair, F. Sehnke, J. Solter, G. Dror, S. Devadas, and J. Schmidhuber. Modeling attacks on physical unclonable functions. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 237–249. ACM, 2010.

[3] Arya Shetty. Puf modelling attacks. https://github.com/AryaShetty08/PUF-Modelling-Attacks, 2024.