2 Machine Learning Attack on PUF

Modeling the Arbiter PUF with Deep Learning

A Introduction

Machine learning (ML) is broad field of data analysis that aims to automate the process of modeling data. A branch of Artificial Intelligence (AI), Machine Learning allows computers to learn *without being explicitly programmed*, by learning to model patterns and behavior from complex data. One of the most influential areas of Machine Learning is Deep Learning, which involves "training" mathematical models known as artificial neural networks (ANNs) to model complex data through a process called *representation learning*.

Inspired by biology (and named as such), ANNs are designed to mimic the behavior of the brain, in that "decisions" are made by an interconnected network of artificial neurons which loosely mimic the functionality of neurons in a brain. As shown in Figure 1(a), each neuron has one or more inputs – which can be from either raw data or a previous neuron – with a *weight* applied to each input and a single *bias* term added. The final value is then applied to an *activation function*, which determines what the final output of the neuron is. One of the most commonly used models in practice is a *deep neural network* (DNN) which has several layers of neurons between the inputs and outputs. An example of a DNN is shown in Figure 1(b).

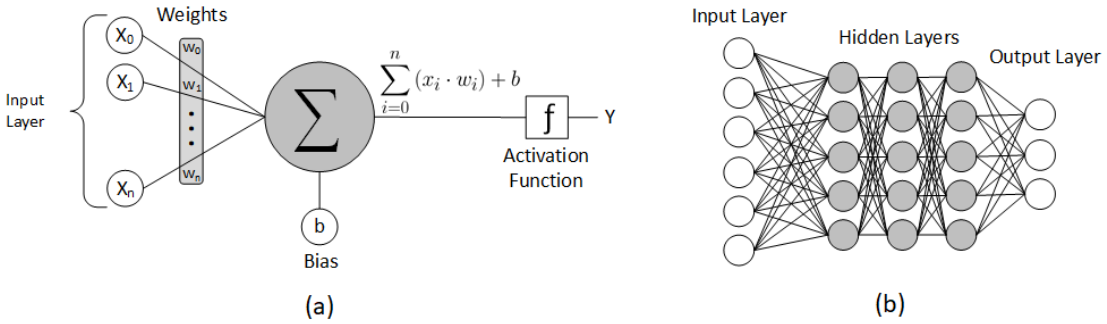


Figure 1: (a) Diagram of simple neuron, where output is a function of the inputs and their respective weights. (b) Diagram of DNN showing the general architecture involving an input layer, multiple hidden layers of various sizes and a final output layer with a neuron for each class.

B Background

Depending on the application, training a DNN can require a relatively large dataset and a wealth of processing power. There are many different methods of training a DNN model, but they generally consist of a few important steps. First, the dataset must be split into 3 different partitions for **training**, **testing**, and **validation**. As their names suggest, the training and testing subsets are used for training the model and then testing it afterward. The validation set, though, is used for evaluating the performance of the model *during* training, rather than afterward with the testing data. This is important because you should always evaluate the model on *unseen* data so that you can be confident in its ability to generalize well in the real world. Second, training the model to be as accurate as possible requires meticulous tuning of the training *hyperparameters* which affect

the model's ability to learn from the training data, as well as the rate at which it does. Some of these parameters are: *batch size* - the number of data samples the model sees at each training step, *number of epochs*, which is the number of times the model works through the entire dataset, *learning rate* - controls the amount that the model's parameters change per training step, and even the architecture of the learning model itself can be considered a hyperparameter.

Putting everything together; we train the model on the training set with some preliminary hyperparameters, evaluate the performance during training on the validation set, make appropriate changes to the hyperparameters then retrain to improve results on the validation data, and ultimately derive a model that can generalize well to the unseen test set.

Because of their excellence in recognizing patterns and learning features from highly complex data, ML-based approaches are the most common ways to attack a PUF. Some of the most commonly used algorithms in literature include Logistic Regression (LR), Support Vector Machines (SVMs), Evolution Strategies (ES), and DNNs [1, 2, 3, 4, 5, 6]. Consequently, this has motivated hardware security researchers to devise enhanced PUF architectures that are resistant to such attacks [7, 8, 9]. In a sense, research efforts to engineer commercially usable PUFs *must* include some sort of countermeasure to mitigate threats of ML-based attacks, lest they be doomed to impracticality.

In general, such attacks are categorized as *modeling attacks* because they involve iteratively training a mathematical construct to *model* the behavior or functionality of a PUF. In practice, a dataset of challenge-response-pairs (CRPs) is extracted from the PUF and then supplied to the learning algorithm for modeling. Modeling attacks then can be evaluated based on how accurately they can model the PUF and how much data is required to train the model. Because extracting CRPs can be difficult and time-consuming, it is preferable to model the PUF accurately with as little data as possible. Additionally, for a DNN-based attack on an Arbiter PUF (APUF), the *challenge parity* can be an important feature to compute for the input data, as it exhibits linear correlation with the delay of the APUF [10]. In this implementation, however, we will instead be using the *parity vector* of each challenge as the input – this conversion is done for you already.

C Hardware and Software Tools

This experiment does not require any hardware itself, but it *does* require a challenge-response-pair (CRP) dataset consisting of (at least) 50,000 CRPs. You should use the dataset you obtained from your PUF in Assignment 1. The CRP data will be formatted by the provided "modelPUF.py" script. You only need to load the respective challenge and response file (one of each). Note that you must pass the challenge .npz file as provided (i.e., file of hex strings of 8-bytes each) and the responses as formatted in Lab 01 (i.e., file of binary strings of 16-bits each).

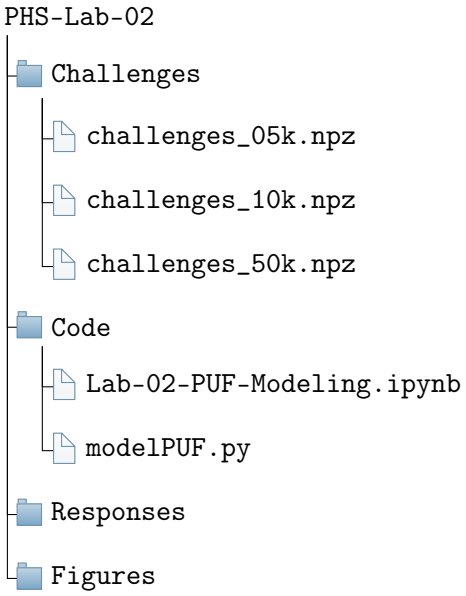
For processing these files, the required software tools are PHS-VM (Appendix C), and the provided lab module files stored in "PHS-Lab-02.zip." The PHS-VM includes Tensorflow v2.0 for Python v3.8-3.10 that provides an easy-to-use interface for rapidly developing AI/ML applications, which we will use to create a *model* of the PUF.

Reading Check

1. What makes neural networks good for modeling PUFs?
2. Why is it important to split the data into separate *training, validation* and *testing* sets?
3. What features/data are used by the neural network to model the PUF?

D Getting Started

Upload and unzip the "PHS-Lab-02.zip" file in the corresponding directory with the same name. The directory is structured as follows:



The **Challenges** directory are the same as Lab 01. You will need to load these files to your model to generate the CRPs dataset. Similarly to Lab 01, you can use any of the challenges files, but **your final results for your report must use the 50k**.

The **Code** directory contains 2 scripts:

1. "Lab-02-PUF-Modeling.ipynb" is the Jupyter Notebook starting scripts that you need to complete. You will use this script to generate the CRPs from the challenges/responses files, interface with the PUF model, collect accuracy list for the model, and then generate plots and statistical results for your report.
2. "modelPUF.py" provides 2 useful interfaces for this experiment:
   (a) First, the **ChallengeResponseSet** class can be used to parse and preprocess the CRP data. This class provides useful functions to sub-sample from the dataset and split it into random train/(validation)/testing subsets based on a percentage split. Additionally, we have included parsing code for your input data that supports two different formats for each file: 1) hex string format in an .npz file for the "challenges" and 2) binary string format in an .npz file for the "responses."
   (b) Second, the **pufModel** class provides a high level interface with which to build, train and test neural network models suited for the given PUF. The script is provided with a command-line-interface (CLI) which allows you to use the script from the terminal. However, it may be better for you to import the modules into your "Lab-02-PUF-Modeling.ipynb" script so you can use them in your own code. Once you have your development environment set up appropriately and are familiar with the functions provided in this script, you may proceed to the lab assignment.

3. **Graduate Only:** you will have to create 2 more scripts called "Lab-02-XOR-APUF-Modeling.ipynb" for step 6 and "Lab-02-Scikit-Modeling.ipynb" for step 7. You can make a copy of the "Lab-02-PUF-Modeling.ipynb" and modify to match the steps' requirements. Make sure to provide sufficient comments silimar to the template.

Store all your response files collected from Lab in the **Responses** directory. You only need 1 response file per FPGA (not 3 each) generated from APUF 01 only (i.e., "APUF_CmodS7_01.bit"). Also, store all your figures generated from "Lab-02-PUF-Modeling.ipynb" in the **Figures** directory.

E Lab Assignment

In this experiment, you will assume the role of the attacker and mount a DNN-based attack on your own PUF from Assignment 1. You will write a Python script which which accomplishes the following:

1) Organize your challenges and response files into the format described earlier, which should generate a total of 50,000 CRPs per FPGA.
2) Utilize the **ChallengeResponseSet** class to load and split the data into different sets for training/testing. *The validation set is not required for this lab.*
3) Create and instantiate a **pufModel** object. Use this object to train a neural network on your CRP data.
4) Repeat and gradually adjust the split ratio p (5% increments from 5% to 95%) and observe the trend in accuracy. What is the ratio (amount of training data) that your model can achieve the highest accuracy with? *Remember to run experiments multiple times and average your results for posterity.*
5) Experiment with different values, and track the performance of the model when it is trained on different splits of the data. *Generally, you should see the performance of the model increase as the amount of training data increases.* Plot your findings in a graph(s) showing the trend of accuracy on the test set versus the size of the training set.

   For graduate students, we additionally require that you:

6) **Graduate Only:** Model a 2- and 4-XOR APUF from the data you have. For example, for the 2-XOR APUF you would take the responses from your CRPs and XOR the first two bits, then the next two, and so on. So your 16-bit response values (which correspond to 16 different APUFs) is then truncated to 8 bits. For the 4-XOR-APUF, the 16-bit values will end up being 4 bits.
7) **Graduate Only:** Instantiate a classifier from Scikit-learn (logistic regression, MLP, SVM, etc.) and train it on the same data for both the APUF and XOR-APUF data in the previous steps. Compare the results to that of the Tensorflow neural network from earlier. The provided PHS-VM does not include "Scikit-learn", you need to install it through the Ubuntu terminal by running:

```
$ source /home/phs/Setup/pyenv.tail
$ pyenv activate cw
$ pip install scikit-learn
```

F Badges and Achievements

This experiment involves **Machine Learning**, **Data Processing**, and **Data Visualization**. By successfully completing the lab you can unlock the following achievements:

- ML1: Machine Learning
- FILE2: File Manipulation
- STAT2: Statistical Analysis
- PLOT2: Data Visualization

Depending on the demonstrated skill, each achievement will be further divided into bronze, silver, and gold levels.

G Deliverables

Set up your experiments along the guidelines described above, and organize your findings into a report according to the guidelines in Appendix A. The report should also contain the following:

- Answers to the reading check questions
- Plots of results from steps 4-7.
- Discuss your findings and how well or poor the model performed.

Submit your report along with all code. All analysis code should be runnable and should produce the output submitted with your report.

References Cited

[1] U. Rührmair, F. Sehnke, J. Sölter, 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.* ACM, 2010, pp. 237–249.

[2] U. Rührmair, J. Sölter, F. Sehnke, X. Xu, A. Mahmoud, V. Stoyanova, G. Dror, J. Schmidhuber, W. Burleson, and S. Devadas, "Puf modeling attacks on simulated and silicon data," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 11, pp. 1876–1891, 2013.

[3] U. Ruhrmair and J. Solter, "Puf modeling attacks: An introduction and overview," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2014.

[4] G. Hospodar, R. Maes, and I. Verbauwhede, "Machine learning attacks on 65nm arbiter pufs: Accurate modeling poses strict bounds on usability," in *2012 IEEE international workshop on Information forensics and security (WIFS).* IEEE, 2012, pp. 37–42.

[5] D. P. Sahoo, P. H. Nguyen, D. Mukhopadhyay, and R. S. Chakraborty, "A case of lightweight puf constructions: Cryptanalysis and machine learning attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, pp. 1334–1343, 2015.

[6] J. Tobisch and G. T. Becker, "On the scaling of machine learning attacks on pufs with application to noise bifurcation," in *International Workshop on Radio Frequency Identification: Security and Privacy Issues.* Springer, 2015, pp. 17–31.

[7] A. Vijayakumar, V. C. Patil, C. B. Prado, and S. Kundu, "Machine learning resistant strong puf: Possible or a pipe dream?" in *2016 IEEE international symposium on hardware oriented security and trust (HOST).* IEEE, 2016, pp. 19–24.

[8] T. Machida, D. Yamamoto, M. Iwamoto, and K. Sakiyama, "Implementation of double arbiter puf and its performance evaluation on fpga," in *The 20th Asia and South Pacific Design Automation Conference.* IEEE, 2015, pp. 6–7.

[9] A. Vijayakumar and S. Kundu, "A novel modeling attack resistant puf design based on non-linear voltage transfer characteristics," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition.* EDA Consortium, 2015, pp. 653–658.

[10] P. Santikellur, A. Bhattacharyay, and R. S. Chakraborty, "Deep learning based model building attacks on arbiter puf compositions," Cryptology ePrint Archive, Report 2019/566, 2019, https://eprint.iacr.org/2019/566.