Lab 02: PUF Modeling

Ganesh Veluru and Jwala Sri Hari Badam
University of South Florida
Tampa, FL 33620

I. Introduction

Machine Learning (ML) stands as a powerful tool in the realm of data analysis, allowing computers to independently decipher intricate patterns and behaviours from data [1]. This ability to identify patterns becomes especially pertinent when we consider modeling Physical Unclonable Functions (PUFs), which play a crucial role in hardware security. PUFs utilize inherent variations in electronic components to create unique identifiers and cryptographic keys, offering robust security. However, these very variations that make PUFs secure can also expose them to modeling attacks. In these attacks, ML algorithms repeatedly train mathematical models to mimic a PUF's behavior, typically using a dataset of Challenge-Response Pairs (CRPs) collected from the PUF. The success of modeling attacks depends on how accurately they can replicate the PUF and the amount of data required for training. As obtaining CRPs can be a challenging and time-consuming process, the goal is to create precise PUF models using minimal data. Additionally, in the case of attacks on Arbiter PUFs (APUFs) using Deep Neural Networks (DNNs), the challenge parity emerges as a vital feature for calculating input data, given its linear correlation with APUF delay. The steps to create a Machine Learning model is shown in Fig 1.

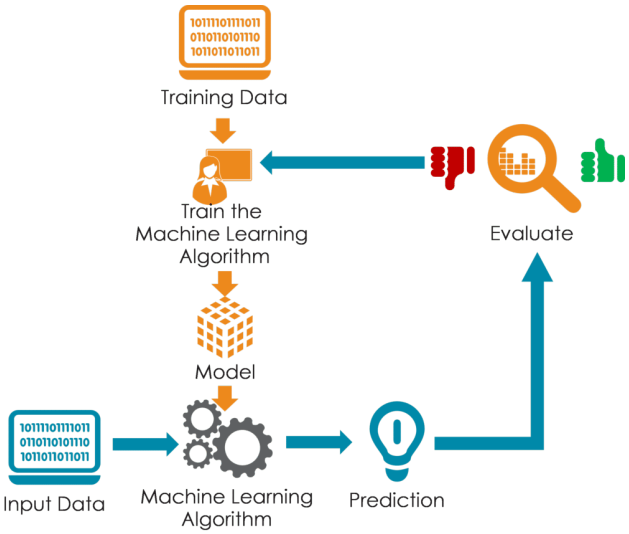


Fig. 1: Steps to Build ML Model

In this project, our primary objective is to investigate the modeling of various PUF types, including APUF, 2-XOR APUF, and 4-XOR APUF, using a range of Machine Learning techniques such as Logistic Regression (LR), Support Vector Machines (SVMs), and Deep Neural Networks (DNNs). We aim to construct these models and carefully evaluate their accuracy. Furthermore, we will explore the resilience of these models against attacks that exploit Machine Learning. Throughout our research, we will also delve into strategies to enhance PUF security in response to these ML-based threats. Our ultimate goal is to provide valuable insights into the ever-evolving field of hardware security.

II. Reading Check

Question 1: What makes neural networks good for modeling PUFs?

Answer: Artificial Neural Networks (ANNs) are computer models that draw inspiration from the structure and operations of the human brain. They consist of layers of interconnected nodes, called neurons, including input, hidden, and output layers. These neurons communicate through weighted connections, and as the network learns from data, it fine-tunes these weights to recognize patterns and generate predictions.

A Neural Network's core functionality relies on the intricate operations of its individual neurons. Each neuron gathers input values from the previous layer, multiplies them by associated weights, and sums them together, adding a bias term. This summed value then undergoes transformation through an activation function. This process, repeated across all neurons within the network, culminates in an output that aims to predict the desired outcome. Although initial predictions may be random, with each training iteration, the network progressively refines its predictions, converging towards the correct values [2].

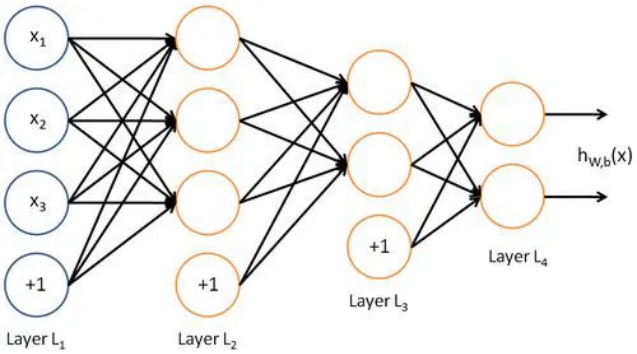


Fig. 2: Representation of a Neural Network

The entire neural network functions cohesively through forward propagation, wherein information flows from input units to hidden layers and ultimately to the output layer. This systematic propagation process allows the network to understand complex data relationships and make accurate predictions. To further enhance its performance, backpropagation is employed to optimize the neural network's parameters, particularly the weights. This iterative refinement process empowers the network to learn how to effectively map input data to desired output, ultimately improving its predictive accuracy.

Therefore, neural networks are advantageous for modeling PUFs because they can effectively capture the intricate and often nonlinear behavior of PUFs. They adapt well to different types of PUFs and can handle datasets of various sizes. Their ability to generalize and process data in parallel makes them suitable for real-time PUF modeling. However, it's important to carefully collect representative data, choose appropriate network architectures, and implement security measures to safeguard against potential threats.

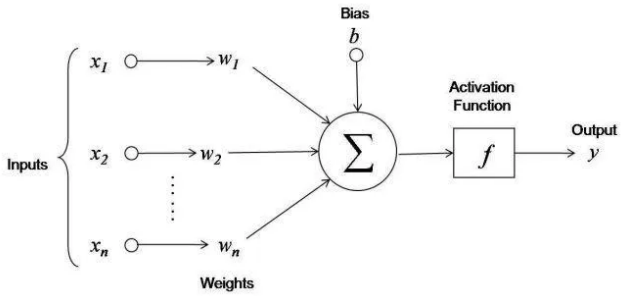


Fig. 3: Operations performed in a neuron

Question 2: Why is it important to split the data into separate training, validation and testing sets?

Answer: The train-validation-test split methodology is a fundamental approach in machine learning, and it involves dividing a dataset into three crucial parts: the training set, validation set, and test set. The training set serves as the foundation for model development. It's where machine learning models learn the intricate patterns, relationships, and nuances within the data. Through techniques like gradient descent, models adjust their parameters to minimize the gap between their predictions and actual outcomes. Essentially, the training set equips the model with the knowledge needed to make meaningful predictions beyond the data it's seen.

However, there's a risk of over-fitting during training, where models become too tailored to the training data and struggle with new data. The validation set acts as an impartial judge, assessing the model's performance on unseen data. By closely observing the model's behavior on this separate dataset, practitioners can fine-tune parameters, select relevant features, and make architectural choices that enhance performance without overfitting. Finally, the test set is entirely separate from training and validation. It represents real-world scenarios where the model encounters new, unseen data. Evaluating the model on this fresh data allows practitioners to measure its ability to make accurate predictions and generalize effectively in practical applications, providing the ultimate test of the model's real-world utility.

Question 3: What features/data are used by the neural network to model the PUF?

Answer: Machine Learning is employed to model PUFs, which are inherently unclonable but can be learned through ML techniques. PUFs generate binary outputs (0 or 1) based on the delays in their circuits and are characterized by CRPs used for learning model parameters. The dataset includes N-bit challenges and 1-bit responses. Mathematically, the response (r) of a PUF to an n-bit challenge (C) is determined by:

$$\text{r} = \begin{cases} 1 & \text{if } \Delta c < 0 \text{ (top signal arrived first)} \\ 0 & \text{otherwise (bottom signal arrived first)} \end{cases}$$

The APUF delay properties can be mathematically modeled using a linear delay model, where each multiplexer (mux) stage contributes delay $\delta 0/1$, based on the presence or absence of switching. The PUF's behavior relies on stage delays and the count of signal switches during the delay. The challenge vector (C) directly influences this switching behavior, and the response depends on the parity vector (Φ) computed from C [3].

To model the PUF using Neural networks, that consists of interconnected neurons. DNNs include multiple layers of neurons, with the output (y) of a neuron determined by the equation:

$$y = \sigma\left(\sum_{i=0}^{n} w_i \cdot x_i + b_i\right) \tag{1}$$

In modeling APUF delay differences $\Delta c = \mathbf{w^T}\Phi$, neural networks can approximate the weights $\mathbf{w^T}$ using parity vectors (Φ) from CRPs. This approach allows neural networks to effectively model APUFs and learn their behavior from CRP data.

III. Methods

A. Software Setup

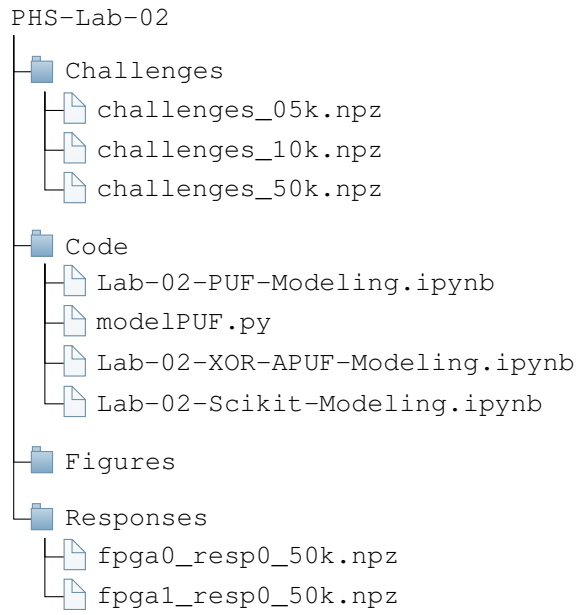
Our procedure for Software setup will take the following steps:

Step 1: Open the PHS VM and Access Jupyter Notebook-To start our investigation, we can open a web browser on our host machine and enter "http://localhost:8888/." This will allow us to access Jupyter Notebook locally. Inside Jupyter Notebook, we'll find various directories, and we should focus on the "PHS-Lab-02" directory.

Step 2: Unzip PHS-Lab-02 Content-Within the "PHS-Lab-02" folder, we can upload the provided "PHS-Lab-02.zip" file. Run the unzip code located in the "RUN-

TOZIP.ipynb" file to extract the contents of "PHS-Lab01.zip." Make sure that the name of the zipped file matches the current directory for successful extraction.

Step 3: Organizing the Extracted Content - After unzipping the provided archive, we'll observe the folder structure illustrated in the accompanying figure. The directory is structured as follows:



a) The "Challenges" folder contains three challenge files in '.npz' format. '.npz' files are NumPy archive files in Python. They serve the purpose of serializing and deserializing NumPy arrays. Among these files, there are challenges of varying sizes: 5,000, 10,000, and 50,000 challenges, respectively. For the current task, we will use the responses generated based on the file containing 50,000 challenges.

b) The "Responses" folder contains the responses that we are going to use to train our ML model along with challenges. We are going to use 2 responses fpga0_resp0_50k and fpga1_resp0_50k which are the reponses of FPGA0 and FPGA1 for 50k challenges.

c) Figures generated during the completion of the task, such as plots and graphical representations, will be stored in the "Figures" directory.

d) Inside the "Code" folder, we discover two important files. The first one is modelPUF.py which contains 2 classes. Firstly, ChallengeResponseSet which is used to convert our CRPs into parity vectors and it also has methods to divide the dataset to train, test and validation sets and the second class is pufModel which has the methods to build, train and test the Neural network model. The second file is "Lab-02-PUF-Modeling.ipynb " which we are going to use to interface with modelPUF.py to get data, to train the model and to get accuracy for each partition ratio and followed by displaying statistical results and plots. In addition to above files we are going to create 2 more .ipynb files. Lab-02-XOR-APUF-Modeling.ipynb for creating model for 2-xor PUF and 4-XOR PUF. Lab-02-Scikit-Modeling.ipynb for creating classifiers using scikit learn library by using the same data that we used to build neural network.

B. Experimental Procedure

Part A: Building ANN model using 50k challenges and responses for FPGA0 and FPGA1

Step 1: We need to import modelPUF, load the challenges and responses for both FPGA0 and FPGA1.

Step 2: We need to create a method that returns accuracy using the classes and methods in modelPUF.py. The code snippet for the get_accuracy() is shown in the Fig 4.
ChallengeResponseSet() will setup the dataset fomat and we will give the dataset to pufModel() followed by splitting the dataset, training the model and testing the model and returning the accuracy.

Step 3: Using the above method we are calculating the accuracy for each split of dataset for the FPGA0 and FPGA1. We are storing the accuracies by appending to the lists as shown in fig. with this we will get accuracies for the split ratios we defined.

```
def get_accuracy(cfile: str, rfile: str, p: float) -> float:
    # Setup CRP dataset format for the PUF model
    crpData = ChallengeResponseSet(cfilename=cfile, rfilename=rfile)

    # Input the CRP dataset to the pufModel
    puf = pufModel(crpData)

    # Splits the CRP dataset in train/test subsets. p (float): percentage of data to use for training
    x=crpData.train_test_split(p)

    # Trains the models with the CRP train subset
    puf.train(crpData)

    # Test the models with the CRP test subset and computes its accuracy.
    accuracy = puf.test(crpData)
    return accuracy
```

Fig. 4: Python Script to get Accuracy based on split ratio

Part B: Building ANN model for 2-XOR and 4-XOR PUFs for FPGA0 and FPGA1

Step 1: We are loading the responses of FPGA0 and FPGA1 and need to convert the 16 bit output to 8 bits using xor operation on consecutive bits.

Step 2: We are iterating through all responses and considering one response at a time and for that response we are running a loop with a step of 2 to perform xor operation [4]. The code snippet is shown in the Fig 5.

Step 3: After performing 2-xor operation on all responses we are saving all responses to a file.

```python
import numpy as np
#converting responses to 2-XOR using actual responses.

data_0 = np.load("../Responses/fpga0_resp0_50k.npz")
data_0=list(data_0['response'])

data_1= np.load("../Responses/fpga1_resp0_50k.npz")
data_1=list(data_1['response'])

fpga0_xor_2_puf=[]
fpga1_xor_2_puf=[]

# outer loop to read all responses.
for i in range(len(data_0)):
    xor_str_0=""
    xor_str_1=""
    #inner loop to perform xor operation on consecutive bits.
    for j in range(0,len(data_0[i]),2):

        xor_str_0+=str(int(data_0[i][j])^int(data_0[i][j+1]))
        xor_str_1+=str(int(data_1[i][j])^int(data_1[i][j+1]))


    fpga0_xor_2_puf.append(xor_str_0)
    fpga1_xor_2_puf.append(xor_str_1)

#saving the 2-xor responses in the files.
responses_0=np.array(fpga0_xor_2_puf)
out_filename_0 = "../Responses/fpga0_resp0_50k_xor_2_puf.npz"
np.savez(out_filename_0, response=responses_0)

responses_1=np.array(fpga1_xor_2_puf)
out_filename_1 = "../Responses/fpga1_resp0_50k_xor_2_puf.npz"
np.savez(out_filename_1, response=responses_1)

print('Output File =', out_filename_0,out_filename_1)
```

Fig. 5: Python Script to Convert Responses to 2-XOR

Step 4: Similarly we need to get 4-xor responses by performing xor on 2-xor responses which we obtained above.

Step 5: Now similar to ANN model for APUF we need to get accuracies for 2-xor PUFs and 4-xor PUFs of FPGA0 and FPGA1.

Part C: Building Model using scikit-learn

Step 1: We need to load the 6 response files (normal, 2-xor, 4xor for each FPGA).

Step 2: Here we are building classifiers using scikit Learn Library where we are importing Logistic Regression, K-Nearest Neighbours, Support Vector Machine, Decision Tree, Random Forest and Naïve Bayes algorithms.

Step 3: Instantiated the models using respective classes and storing the objects in models={ } with keys as respective algorithm. It is shown in the Fig 7.

Step 4: Creating 6 dictionaries for each data file with keys as algorithms and values as list of accuracies for split_ratio.

Step 5: Creating a function that will accept the cfile, rfile, split ratio and classifier as arguments and return accuracy.

Step 6: Creating nested loop where outer loop iterates for models={} and inner loop for split ratio. For each algorithm we are calculating accuracy (with split ratios) for all 6 data sets. The code snippet for this is shown in the Fig 8.

Step 7: After the execution of nested loop we will get 6 dictionaries populated with accuracies of 6 ML algorithms for each split ratio value.

Step 8: The 6 dictionaries are appended to a nested dictionary and using json module the dictionary is stored in the file. As the training of all models takes times the accuracies are stored for further analysis and using json load we can retrieve the dictionary.

IV. Results

Firstly the calculated accuracies for ANN with all split ratios for all datasets are shown in Table I.

The accuracies are calculated for 7 ML Algorithms and the statistical results for each Algorithm are calculates using pandas describe().

We are using the dictionary that is saved in json format and calculated the average value for each algorithm and displaying it after styling the dataframe. Similarly displaying the maximum accuracy recorded for each data set when trained with each algorithm.

The average and maximum values are shown in Table II and III.

The model trained using ANN algorithm using many splits for all datasets are plotted using matplotlib.

For FPGA0 and FPGA1 with rfile0 and rfile1 which are actual responses are plotted in single graph as shown in Fig 9.

Similarly for 2-xor for FPGA0 and FPGA1 in single plot and for 4-xor in another plot using the same code as shown in Fig 10.

For the scikit learn algorithms the average accuracies are plotted using bars for all datasets and plot is shown in Fig 11.

V. Discussion

The Average accuracy for the both FPGA0 and FPGA1 is around 68% and maximum of 70% is recorded when trained using ANN for APUF. When we consider 2-XOR APUF the average accuracy is around 70% and maximum recorded is 73% for both devices. For the 4-XOR we are observing less accuracy compared to APUF and 2-XOR APUF which is 57% for both devices. The results suggest that the complexity of the PUF architecture has an impact on the accuracy of the models. The 2-XOR APUF, which involves additional XOR operations, yielded higher accuracy than the basic APUF which is not expected.

In contrast, the 4-XOR APUF, which introduces further complexity, results in lower accuracy compared to both APUF and 2-XOR APUF which is the expected result. Increasing the number of XOR gates in an APUF makes it more resistant to ML attacks [5]. This added resistance is due to the increased complexity and unpredictability introduced by the additional XOR gates. These gates create more intricate and nonlinear relationships between the input challenges and the responses generated by the APUF. As a result, it becomes more challenging for attackers to develop accurate ML models to predict APUF outputs, as they need larger and more diverse datasets for training. However, it's important to note that there's a trade-off between security and practicality, so finding the right balance is crucial when designing APUFs for real-world applications.

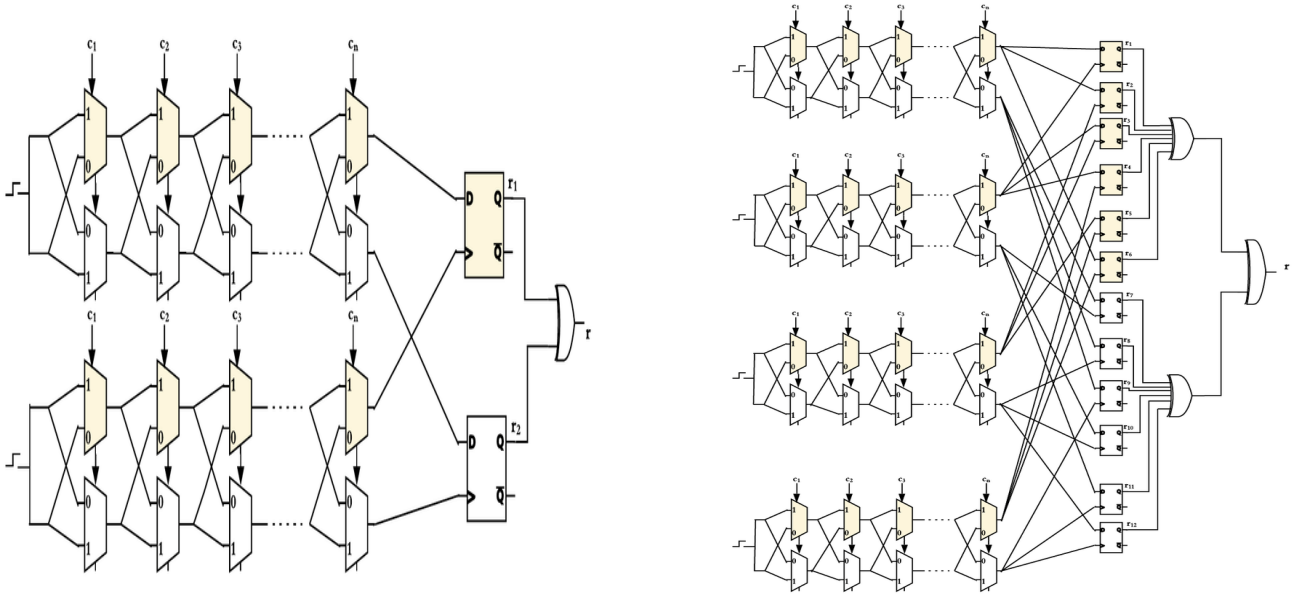


Fig. 6: Conventional Design of 2-XOR and 4-XOR APUF

TABLE I: Statistical Results for APUF, 2-XOR PUF and 4-XOR PUF

Parameter	**FPGA-0**			**FPGA-1**		
PUF	**APUF**	**2-XOR APUF**	**4-XOR APUF**	**APUF**	**2-XOR APUF**	**4-XOR APUF**
Minimum	63.2	66.4	56.5	63.2	66.8	53.8
Maximum	70.8	73	59	59.0	73.6	58.9
Mean	68.1	70.7	57.8	57.8	70.6	57.0
SD	2.83	2.1	0.8	2.20	2.3	1.3

TABLE II: Average Accuracies for APUF, 2-XOR APUF and 4-XOR APUF with Algorithms Used for FPGA-0 and FPGA-1

Dataset	**LR**	**KNN**	**SVM**	**NB**	**DT**	**RF**	**ANN**
rfile0	63.87	60.39	64.42	64.32	55.97	64.58	68.1
rfile1	63.29	59.62	63.43	63.29	55.65	63.48	68.37
xor-2-rfile0	68.04	63.65	67.94	67.80	57.49	68.2	70.68
xor-2-rfile1	67.07	63.13	67.42	67.28	56.96	67.21	70.64
xor-4-rfile0	57.58	53.4	57.98	57.46	51.52	57.86	57.82
xor-4-rfile1	56.73	53.33	57.12	56.48	51.35	56.87	57.00

Coming to the results when we used other Machine Learning Algorithms the highest average accuracy of 64% recorded with Logistic Regression, SVM and Random Forest. Therefore we can confirm that ANN performs well compared with other Algorithms and for 4-XOR APUF the other Algorithms gave the accuracy similar to ANN. Some of the challenges faced is the training time the model taking for some algorithms like SVM and we can even check with other Algorithms and hyper parameter tuning for better results to get good model.

VI. Conclusion

In summary, this laboratory experiment has provided us with practical insights into modeling PUFs using ML algorithms and the potential vulnerabilities that attackers can exploit. Additionally, we've learned that enhancing the complexity of PUFs by increasing the number of XOR gates can enhance their resistance against ML-based attacks. During this experiment, we gained hands-on experience with the scikit-learn library, enabling us to train various classification algorithms and compare their accuracies with those achieved using ANNs.

Moving forward, there is a need to explore the development of even more robust PUFs that can effectively withstand ML-based attacks. Additionally, it's crucial to investigate how attackers might employ different techniques to model N-XOR PUFs, further enhancing our understanding of potential security risks.

TABLE III: Maximum Accuracies for APUF, 2-XOR APUF and 4-XOR APUF with Algorithms Used for FPGA-0 and FPGA-1

Dataset	LR	KNN	SVM	NB	DT	RF	ANN
rfile0	64.37	60.81	64.5	64.82	56.93	64.84	70.88
rfile1	63.93	60.48	63.50	63.69	56.63	63.84	70.44
xor-2-rfile0	68.32	64.14	68.48	68.21	58.07	68.31	73.02
xor-2-rfile1	67.58	64.07	67.76	68.36	58.21	67.43	73.62
xor-4-rfile0	58.34	54.03	58.21	58.31	51.87	58.08	59.00
xor-4-rfile1	57.57	53.96	57.26	57.31	52.04	57.09	58.89

```python
# Importing models from sklearn
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# creating dictionary of objects for each algorithm

models = {
    'LR': LogisticRegression(),
    'KNN': KNeighborsClassifier(n_neighbors=5),
    'SVM': SVC(kernel='poly', degree=5),
    'NB': GaussianNB(),
    'DT': DecisionTreeClassifier(criterion='entropy'),
    'RF': RandomForestClassifier(n_estimators=50,
          criterion='entropy', max_depth=10, max_leaf_nodes=10)
}

#saving split ratios
split_ratio= [i / 100.0 for i in range(5, 100, 20)]

# creating dictionaries for each file to store accuracies of each model for each split
keys=['LR','KNN','SVM','NB','DT','RF','ANN']
acc_rfile0={}
acc_rfile1={}
acc_xor_2_rfile0={}
acc_xor_2_rfile1={}
acc_xor_4_rfile0={}
acc_xor_4_rfile1={}

#Initializing dictionaries with empty lists as values for each algorithm
for item in keys:
    acc_rfile0[item]=[]
    acc_rfile1[item]=[]
    acc_xor_2_rfile0[item]=[]
    acc_xor_2_rfile1[item]=[]
    acc_xor_4_rfile0[item]=[]
    acc_xor_4_rfile1[item]=[]

#sample dictionary.
print(acc_rfile0)
```

Fig. 7: Python Script to Import Scikit Learn Models

```python
# function to return accuracy for ML model after fitting the model

def accuracy_ML(cfile,rfile,p,classifier):
    #getting training and testing data using get_data()
    trainX,trainY,testX,testY=get_data(cfile,rfile,p)
    classifier.fit(trainX,trainY)
    predY=classifier.predict(testX)
    acc=accuracy_score(predY,testY)*100

    return acc
```

```python
# Nested Loop for calculating accuracy for each model and for each split ratio
#: for APUF, 2-XOR, 4-XOR ; FPGA0, FPGA1

for key,obj in models.items():
    print(key,"running")
    for i in split_ratio:
        acc_rfile0[key].append(accuracy_ML(cfile,rfile0,i,obj))
        acc_rfile1[key].append(accuracy_ML(cfile,rfile1,i,obj))
        acc_xor_2_rfile0[key].append(accuracy_ML(cfile,xor_2_rfile0,i,obj))
        acc_xor_2_rfile1[key].append(accuracy_ML(cfile,xor_2_rfile1,i,obj))
        acc_xor_4_rfile0[key].append(accuracy_ML(cfile,xor_4_rfile0,i,obj))
        acc_xor_4_rfile1[key].append(accuracy_ML(cfile,xor_4_rfile1,i,obj))
```

Fig. 8: Python Script to get Accuracies for all scikit learn Algoritms

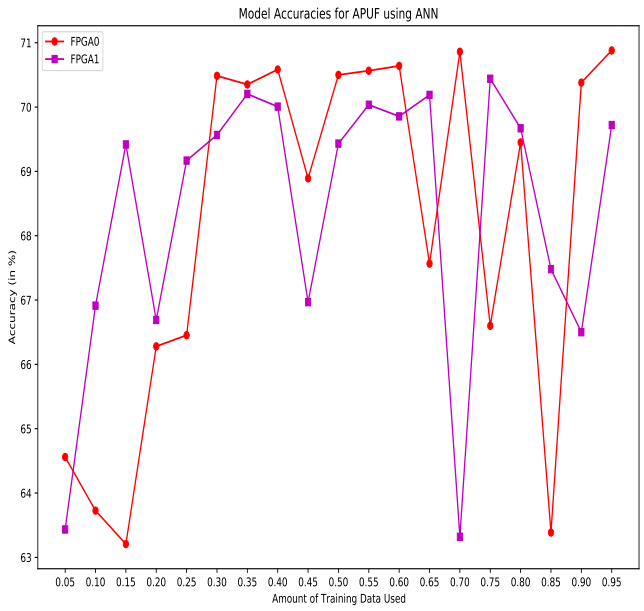


Fig. 9: Accuracy of Model for various amount of Training Data using ANN

REFERENCES

[1] R. Karam, S. Katkoori, and M. Mozaffari-Kermani, "Lecture Note 4: Machine Learning in Hardware Security/Machine Learning," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Sep 2023.

[2] ——, "Experiment 1: PUF Modeling," in *Practical Hardware Security Course Manual*. University of South Florida, Sep 2023.

[3] ——, "Lecture Note 4: Machine Learning in Hardware Security/Cloning the Unclonable," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Sep 2023.

[4] ——, "Lecture Note 5: Attacks and Countermeasures on Strong PUFs/The Arbiter PUF Family," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Sep 2023.

[5] ——, "Lecture Note 5: Attacks and Countermeasures on Strong PUFs/Strong PUFs," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Sep 2023.

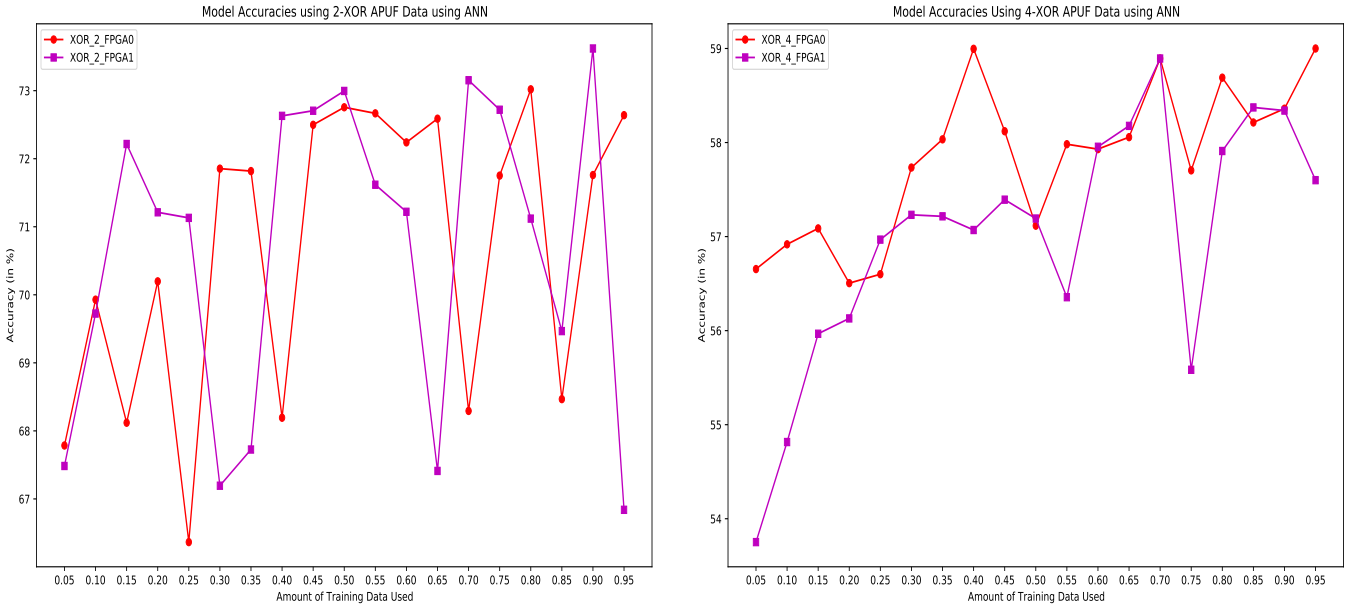


Fig. 10: Accuracy of Model for 2-XOR and 4-XOR APUF Dataset using ANN

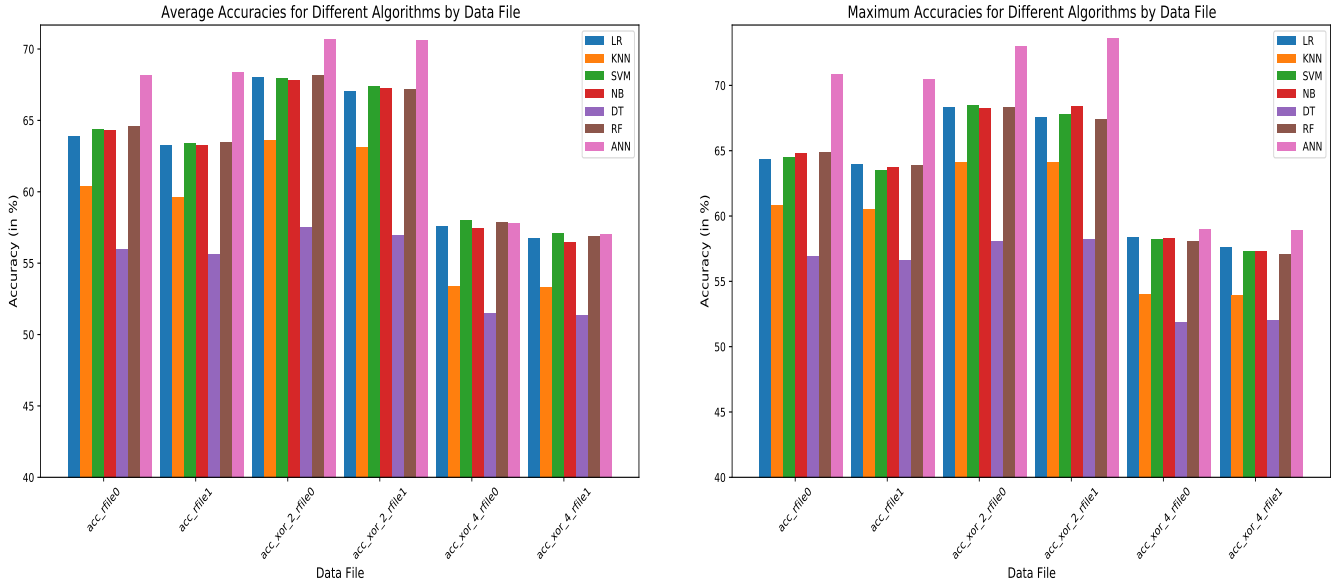


Fig. 11: Average and Maximum Accuracies for each model with all Datasets