Attacking PUF’s

Kyung Hwan ‘David’ Lee

NUID: 001627957

Lee.Kyun@Northeastern.edu

Scope

This project will perform a machine learning attack on a simulated PUF. The data used in this project are generated from a simulated PUF in MATLAB and are provided for this project.

During this document, a detailed description on the execution of a Machine Learning attack on a PUF will be described using tool provided by MATLAB. More specifically, the Classification Learner Tool will be used to execute multiple different types of Machine Learning attacks on the same set of data. The accuracy of the Machine Learning algorithm will be provided in this document.

This document will also compare different Machine Learning algorithms and their effectiveness when applying them to the simulated PUF data given.

This report will be separated into two major sections: *Machine Learning Attack on an Idealized PUF* and *Machine Learning Attack on a Fault Injected PUF*.

This project was completed for *Northeastern University EECE7390 – Computer Hardware Security* class taken in the Spring 2021 semester.

Table of Contents

Terminology

In this document, the following abbreviations will be used.

1. PUF – Physically Unclonable Function
2. ML – Machine Learning
3. MUX - Multiplexer

Software Utilized

In this project, the following software is used.

1. MATLAB 2020a Student Edition

MATLAB is utilized to execute the ML algorithms and a license is provided by the university, *Northeastern University*.

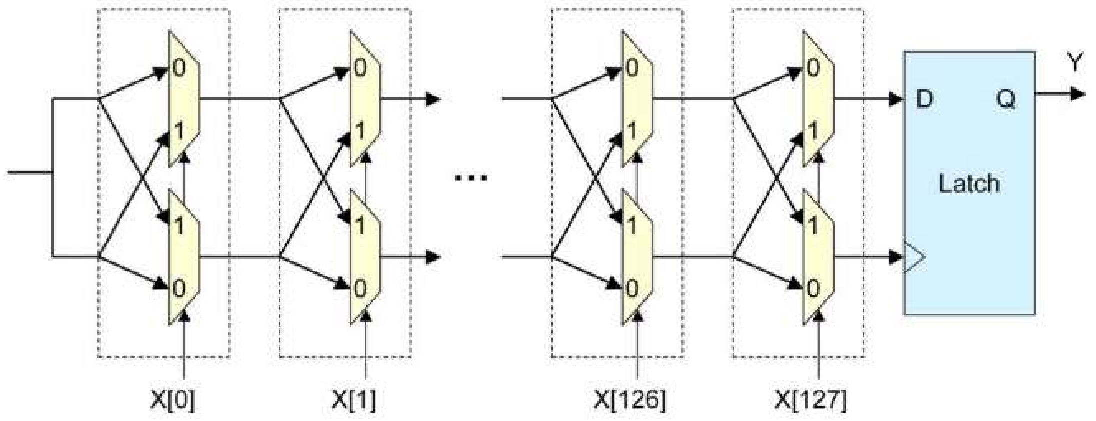
The simulated PUF data set is provided by *Prof. Xiaolin Xu* (www.xiaolinxu.com).

PUF Background

A PUF, or *Physically Unclonable Function*, is a hardware security method where a physical object is given a challenge input and provides a unique response. This is commonly called a *CRP*, or *Challenge and Response Pair*. PUF's are used in many applications where security is needed, such as *RFID*.

PUF's can provide a unique *CRP* because of the process variations that incur during the fabrication process. Because of these process variations, it is too time and resource consuming to replicate a specific PUF.

The specific implementation of the PUF we will be attacking with a ML algorithm is the Arbiter PUF. The below figure is a small subsection of the functionality of an Arbiter PUF.



[1]

An Arbiter PUF functions as a series of multipliers (MUX) in series connected as above. The Challenge bits function as the input of each MUX stage. This will allow the input signal (held high) to propagate through different paths of the PUF. This continues through each stage of the PUF. At the final stage of the PUF, a Latch device is used to see which path of diverging initial path is faster. If the upper path is faster than the lower path, the latch will output a high signal. If the lower path is faster than the upper path, the latch will output a low signal.

When seen above, it may seem like the signals will propagate through the stages at the same rate. However, because of process variations during the fabrication of the semiconductor, the upper and lower path and individual MUX paths will have different propagation delays.

The propagation delay differences may be induced by uneven doping, uneven metal depositions, or uneven oxide deposition. These factors will affect the voltage switching threshold of the gates used within the MUX and may minutely change propagation delays through the MUX's. These fabrication variances can also be seen in the interconnects used to implement the Arbiter PUF. Variances in the Interconnects will change parasitic wire resistance and capacitance values that will affect the propagation delay through different paths.

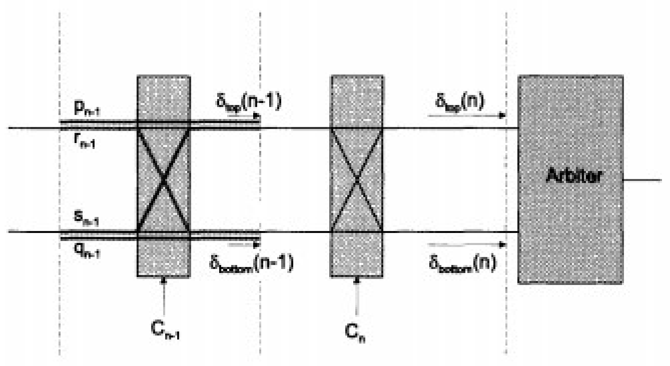
Ideally, the Arbiter PUF will be laid out in a way that each path of the Arbiter PUF is identical to the other as to not induce any artificial propagation delays due to design. We only want to take advantage of variances due to fabrication variances to ensure that there is no CRP skew.

Machine Learning Attack on an Idealized PUF

Additive Delay Model

Because of the design methodology of the Arbiter PUF, it is nearly impossible (time and resource constraining) to physically duplicate an Arbiter PUF. However, it is very susceptible to a Machine Learning attack.

We can implement the *Additive Delay Model* to create a Machine Learning attack that has greater than 99% accuracy. This is accomplished because of the design methodology of the Arbiter PUF. We can calculate and simulate the path delays of the Arbiter PUF using a Machine Learning attack.



[2]

We can denote the path delays that the signal can travel through within the PUF. By providing a small subset of the known CRP's we can utilize a machine learning algorithm to calculate the respective path delays.

$$\delta_{top}(i+1) = \frac{1+C_{i+1}}{2}\left(p_{i+1}+\delta_{top}(i)\right) + \\ + \frac{1-C_{i+1}}{2}\left(s_{i+1}+\delta_{bottom}(i)\right)$$

$$\delta_{bottom}(i+1) = \frac{1+C_{i+1}}{2}\left(q_{i+1}+\delta_{bottom}(i)\right) + \\ + \frac{1-C_{i+1}}{2}\left(r_{i+1}+\delta_{top}(i)\right),$$

[2]

In this delay model, we must first change the known challenge bits from (0, 1) to (-1, 1). This is done so that the above equations can be utilized in the Machine Learning attack. A simple explanation for this change is as follows. For the above equations to function, the challenge bits must modulate which path delay must be added to the 'simulated' signal propagation. If the signal travels through one input of the MUX, it must not add the other MUX input's propagation delay. The -1 and 1 challenge bit acts as a mathematical 'switch', enabling or disabling parts of the equation seen above.

The above equations can be used to derive:

$$\Delta(i+1) = C_{i+1}\cdot\Delta(i) + \alpha_{i+1}C_{i+1} + \beta_{i+1},$$

$$\alpha_n = \frac{p_n - q_n + r_n - s_n}{2}$$
$$\beta_n = \frac{p_n - q_n - r_n + s_n}{2}.$$

[2]

Where *p* is defined as the parity of the challenge bits.

$$p_k = \prod_{i=k+1}^{n} C_i,$$

[2]

The parity vector denotes that later stages of the Arbiter PUF will not effect the previous stages of the Arbiter PUF.

From this, we can denote:

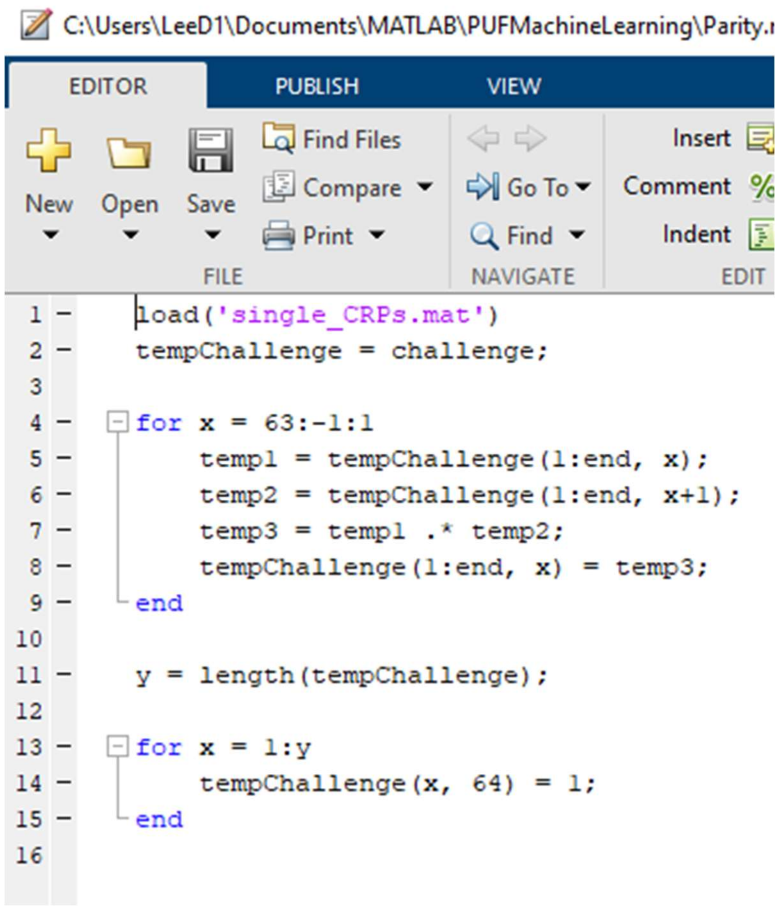
$$\mathbf{p} = (p_0, p_1, \ldots, p_n) \text{ and } \mathbf{d} = (\alpha_1, \alpha_2+\beta_1, \ldots, \alpha_n+\beta_{n-1}, \beta_n)$$ [2]

Please read *Extracting Secret Keys from Integrated Circuits* by *Daihyum Lim*. The thesis is provided in the bibliography of this project report.

From this we can provide the machine learning algorithm with the parity vector of challenge bits with its known response pair and train a ML algorithm. The ML algorithm will automatically compute the *alpha* and *beta* values in training.

Execution

The following script is used to create the parity vector of challenges while preserving their response pair.



```
load('single_CRPs.mat')
tempChallenge = challenge;

for x = 63:-1:1
    temp1 = tempChallenge(1:end, x);
    temp2 = tempChallenge(1:end, x+1);
    temp3 = temp1 .* temp2;
    tempChallenge(1:end, x) = temp3;
end

y = length(tempChallenge);

for x = 1:y
    tempChallenge(x, 64) = 1;
end

```

The classification learner is given the 33 wide parity vector of varying sizes to train the ML algorithm. The classification learner is a tool provided by MATLAB.

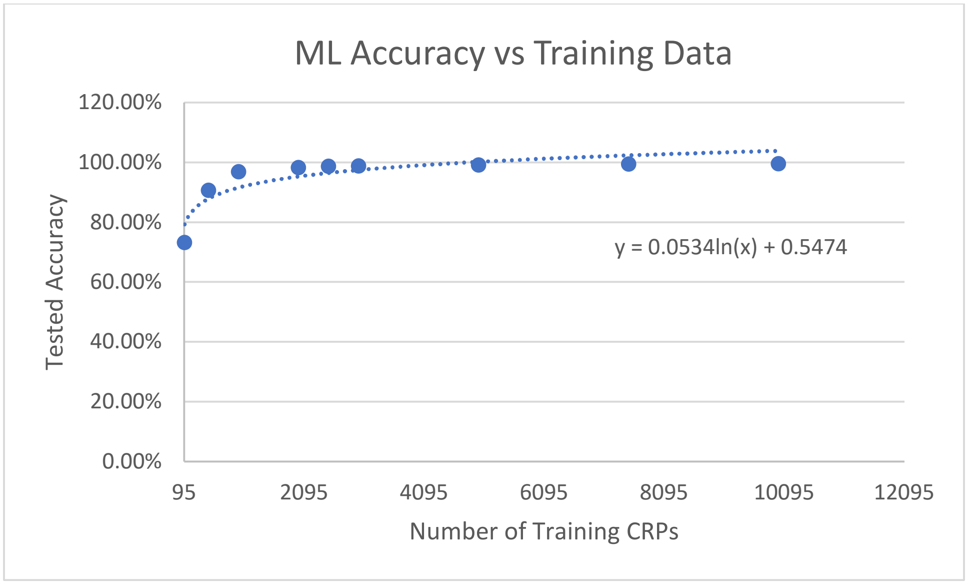
Results

The following tabulates the accuracy of the ML algorithm while changing the number of CRP pairs given as a training data set.

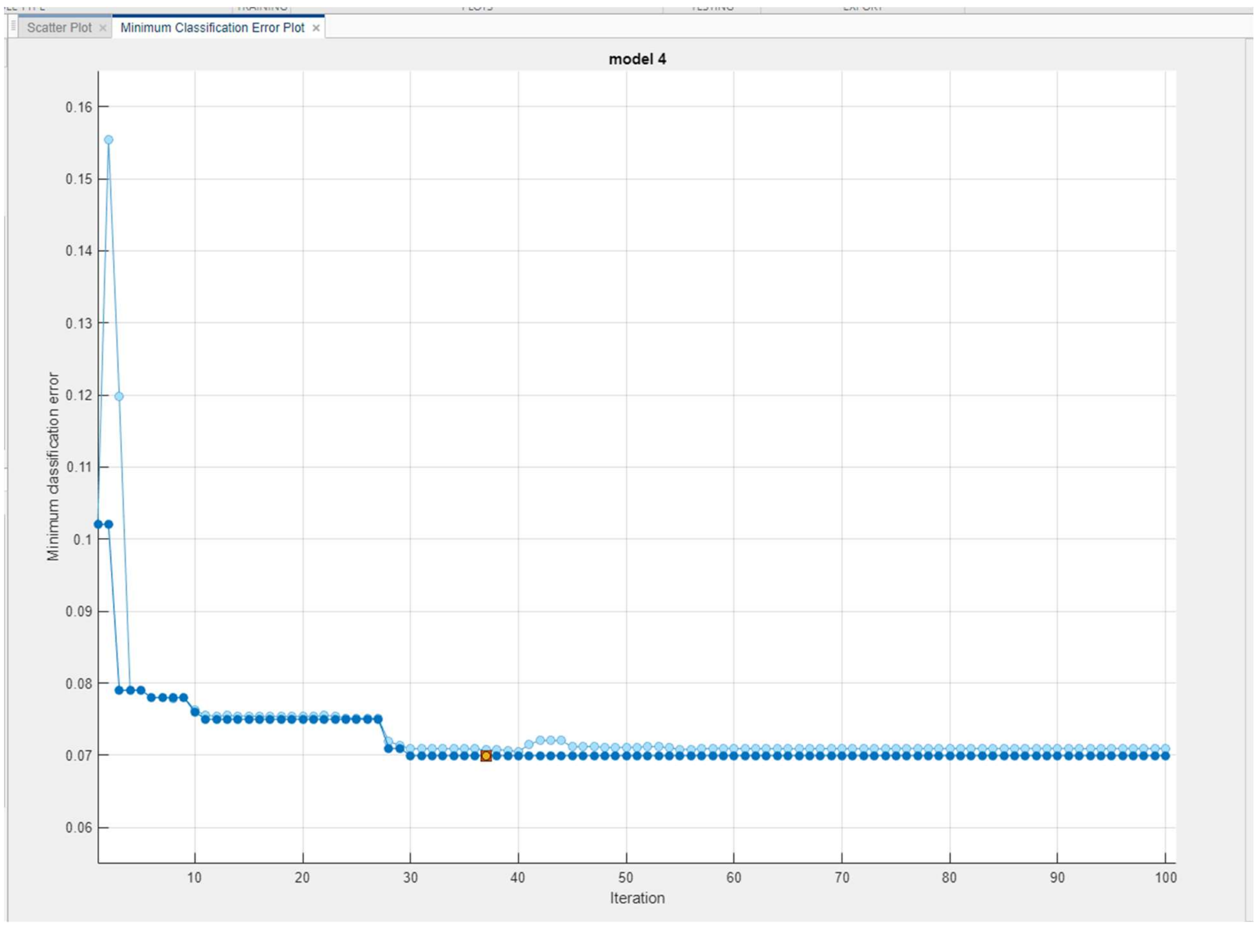
Part 1					
CRP No	Classification Learner	Accuracy	Note	Time	Tested Against Full Response Vector Accuracy
100	Fine Tree	81.0%	Quick to Train	Instant	
100	Medium Tree	81.0%	Quick to Train	Instant	
100	Coarse Tree	75.0%	Quick to Train	Instant	
100	Fine KNN	83.0%	Quick to Train	Instant	
100	Medium KNN	60.0%	Quick to Train	Instant	
100	Coarse KNN	52.0%	Quick to Train	Instant	
100	Cosine KNN	70.0%	Quick to Train	Instant	
100	Cubic KNN	60.0%	Quick to Train	Instant	
100	Weighted KNN	78.0%	Quick to Train	Instant	
100	Linear SVM	80.0%	Standard	Instant	
100	Linear SVM	84.0%	Optimized x30	< 5 Minutes	
100	Linear SVM	**85.0%**	Optimized x100	< 5 Minutes	73.21%
500	Fine Tree	74.6%	Quick to Train	Instant	
500	Medium Tree	75.0%	Quick to Train	Instant	
500	Coarse Tree	70.2%	Quick to Train	Instant	
500	Fine KNN	74.8%	Quick to Train	Instant	
500	Medium KNN	70.4%	Quick to Train	Instant	
500	Coarse KNN	76.8%	Quick to Train	Instant	
500	Cosine KNN	70.6%	Quick to Train	Instant	
500	Cubic KNN	70.6%	Quick to Train	Instant	
500	Weighted KNN	74.4%	Quick to Train	Instant	
500	Linear SVM	87.6%	Standard	Instant	
500	Linear SVM	93.6%	Optimized x30	< 5 Minutes	
500	Linear SVM	**93.6%**	Optimized x100	< 5 Minutes	90.64%
1000	Fine Tree	71.4%	Quick to Train	Instant	
1000	Medium Tree	71.6%	Quick to Train	Instant	
1000	Coarse Tree	67.7%	Quick to Train	Instant	
1000	Fine KNN	74.4%	Quick to Train	Instant	
1000	Medium KNN	75.6%	Quick to Train	Instant	
1000	Coarse KNN	81.5%	Quick to Train	Instant	
1000	Cosine KNN	75.0%	Quick to Train	Instant	
1000	Cubic KNN	75.5%	Quick to Train	Instant	
1000	Weighted KNN	77.8%	Quick to Train	Instant	
1000	Linear SVM	91.5%	Standard	Instant	
1000	Linear SVM	**96.7%**	Optimized x30	< 5 Minutes	96.90%
2000	Fine Tree	70.0%	Quick to Train	Instant	
2000	Medium Tree	68.7%	Quick to Train	Instant	
2000	Coarse Tree	66.8%	Quick to Train	Instant	
2000	Fine KNN	69.3%	Quick to Train	Instant	
2000	Medium KNN	74.0%	Quick to Train	Instant	
2000	Coarse KNN	83.4%	Quick to Train	Instant	
2000	Cosine KNN	74.4%	Quick to Train	Instant	
2000	Cubic KNN	73.9%	Quick to Train	Instant	
2000	Weighted KNN	75.8%	Quick to Train	Instant	
2000	Linear SVM	95.0%	Standard	Instant	
2000	Linear SVM	**98.2%**	Optimized x30	< 5 Minutes	98.21%
2500	Fine Tree	68.3%	Quick to Train	Instant	
2500	Medium Tree	70.1%	Quick to Train	Instant	
2500	Coarse Tree	66.8%	Quick to Train	Instant	
2500	Fine KNN	67.6%	Quick to Train	Instant	
2500	Medium KNN	75.1%	Quick to Train	Instant	
2500	Coarse KNN	84.5%	Quick to Train	Instant	
2500	Cosine KNN	74.7%	Quick to Train	Instant	
2500	Cubic KNN	75.1%	Quick to Train	Instant	
2500	Weighted KNN	76.4%	Quick to Train	Instant	
2500	Linear SVM	95.8%	Standard	Instant	
2500	Linear SVM	**98.3%**	Optimized x30	< 10 Minutes	98.71%
3000	Fine Tree	71.7%	Quick to Train	Instant	
3000	Medium Tree	71.0%	Quick to Train	Instant	
3000	Coarse Tree	67.0%	Quick to Train	Instant	
3000	Fine KNN	67.6%	Quick to Train	Instant	
3000	Medium KNN	76.1%	Quick to Train	Instant	
3000	Coarse KNN	85.5%	Quick to Train	Instant	
3000	Cosine KNN	75.9%	Quick to Train	Instant	
3000	Cubic KNN	76.1%	Quick to Train	Instant	
3000	Weighted KNN	77.4%	Quick to Train	Instant	
3000	Linear SVM	96.1%	Standard	Instant	
3000	Linear SVM	**98.7%**	Optimized 30x	< 10 Minutes	98.80%
5000	Linear SVM	97.5%	Standard	Instant	
5000	Linear SVM	**99.4%**	Optimized 30x	< 10 Minutes	99.17%
7500	Linear SVM	98.1%	Standard	Instant	
7500	Linear SVM	**99.4%**	Optimized 30x	< 60 Minutes	99.44%
10000	Linear SVM	98.3%	Standard	Instant	
10000	Linear SVM	**99.5%**	Optimized 20x	< 90 Minutes	99.61%

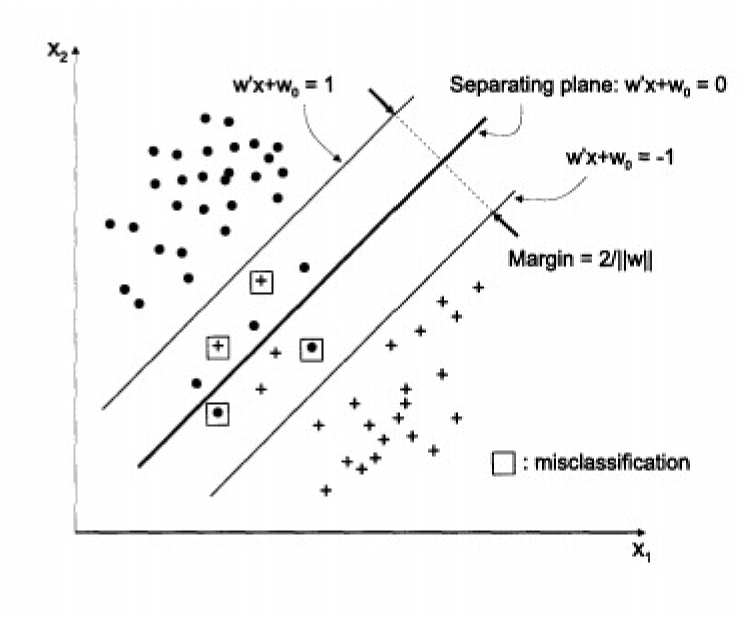
We can see above that even with only 500 CRPs of the available or 0.25% of the available CRPs, we can achieve a prediction accuracy of 90.64%. We can also see that the Linear Support Vector Machine will result in the best accuracy in PUF prediction.

With only 5000 CRPs (2.5% of available CRPs), we can achieve 99.17% prediction accuracy. Even the large training dataset used, the training time for the Linear Support Vector Machine was less than 10 minutes, practically instantaneous.



The optimization graph for 1000 CRPs are shown below.


Scatter Plot
Minimum Classification Error Plot
model 4
Minimum classification error
Iteration
0.16
0.15
0.14
0.13
0.12
0.11
0.1
0.09
0.08
0.07
0.06
10
20
30
40
50
60
70
80
90
100



[2]

We can see that the Support Vector Machine is demonstrably the best ML algorithm for modeling the Arbiter PUF. We rationalized this in a simple way, the Linear Support Vector Machine will compute each *alpha* and *beta* values from the previous section and create a linearly additive relationship with the following predictors. Simply, the path delays of an Arbiter PUF is additive linearly. So, at a cursory glance, the Linear Support Vector Machine is particularly suited for this application.

Machine Learning Attack on a Fault Injected PUF

In the previous section, we assumed an ideal case where external factors were eliminated from the PUF circuitry. The path delays are purely from the fabrication variations due. Realistically, we cannot consider this case practical.

In practical use, the Arbiter PUF will change its behavior because of external factors such as temperature and EMI/EMC interference. We can observe this by delving into the MUX gates and their temperature properties.

All CMOS devices, like the MUX, will have different switching voltage thresholds. This may be due to differing doping levels across the IC or inaccurate MOSFET dimensions. However, this is expected and exploited in the Arbiter PUF design. Another factor exists that can severely impact the MOSFETS performance. The kT/q term often appears in semiconductor devices because it impacts switching thresholds, leakage currents, and the overall performance of the MSOFET. At lower temperature, electrons can move faster (and vice versa). This effectively lowers the propagation delay of the CMOS device. Simply put, a cold IC can run faster than a hot IC.

Changing environments of the same Arbiter PUF device can induce a different CRP. However, this is expected in a practical PUF. We can see this in the application of a Strong PUF in bank ID cards. The bank will issue many random challenges to the PUF. It will compare the response with the bank's know CRP pairs and verify the user. Even in this response, the bank will disregard a small number of incorrect CRPs if most of the CRP are correct. Practically, the bank allows a small amount of response be incorrect and even inserts some incorrect data in its known CRP data set to make the Strong PUF error tolerant without the use of any external error correction [3].

Changes in the Additive Delay Model

With this in mind, we can modify our original additive delay model. We can assume that the external factors affecting the propagation delays within the PUF model to be affected in a Gaussian model. However, most of the CRPs will not be affected significantly enough to flip the response from low to high or high to low. We can modify our model in the following manner.

$$\Delta D = \Delta D_{PUF} + D_{noise} = \vec{w}^T \vec{\Phi} + D_{noise}$$

$$r = \begin{cases} 1, & \text{if } \Delta D_{PUF} + D_{noise} < 0. \\ 0, & \text{if } \Delta D_{PUF} + D_{noise} > 0. \end{cases}$$

[4]

To test the fitness of a given PUF model, we can now compare the reliability of the CRP against the predicted reliability as defined below.

$$\tilde{h_i} = \begin{cases} 1, & \text{if } |\vec{w}^T \vec{\Phi}_i| > \epsilon \\ 0, & \text{if } |\vec{w}^T \vec{\Phi}_i| < \epsilon \end{cases}$$

[4]

This simply means that the ML algorithm must also compute the *e* term, the error threshold.

Execution

In this project, this is practically done by modifying the parity vector used as the predictor in the Classification Learner tool in MATLAB. In the previous section, the parity vector was n+1 wide, n being the number of stages or challenges per response. We will provide the classification learner with n+2 bits, where the n+2 bit is equal to 1. This will allow the ML algorithm to use another variable that will stand for the error term. We can also increase the size of the parity vector to n+3, n+4, and n+5 size to give the ML algorithm more terms to find the error value.

Results

The following the accuracy results from a parity vector of size n+2.

Part 2 parity is n+2				
CRP No	**Classification Learner**	**Accuracy**	**Note**	**Time**
100	Fine Tree	78.0%	Quick to Train	Instant
100	Medium Tree	78.0%	Quick to Train	Instant
100	Coarse Tree	77.0%	Quick to Train	Instant
100	Fine KNN	84.0%	Quick to Train	Instant
100	Medium KNN	91.0%	Quick to Train	Instant
100	Coarse KNN	91.0%	Quick to Train	Instant
100	Cosine KNN	91.0%	Quick to Train	Instant
100	Cubic KNN	91.0%	Quick to Train	Instant
100	Weighted KNN	91.0%	Quick to Train	Instant
100	Linear SVM	88.0%	Standard	Instant
100	Linear SVM	**91.0%**	Optimied 30x	< 5 Minutes
500	Fine Tree	76.4%	Quick to Train	Instant
500	Medium Tree	80.0%	Quick to Train	Instant
500	Coarse Tree	85.4%	Quick to Train	Instant
500	Fine KNN	75.8%	Quick to Train	Instant
500	Medium KNN	85.4%	Quick to Train	Instant
500	Coarse KNN	85.4%	Quick to Train	Instant
500	Cosine KNN	85.4%	Quick to Train	Instant
500	Cubic KNN	85.4%	Quick to Train	Instant
500	Weighted KNN	85.2%	Quick to Train	Instant
500	Linear SVM	85.4%	Standard	Instant
500	Linear SVM	**85.4%**	Optimied 30x	< 5 Minutes
1000	Fine Tree	71.3%	Quick to Train	Instant
1000	Medium Tree	78.2%	Quick to Train	Instant
1000	Coarse Tree	81.6%	Quick to Train	Instant
1000	Fine KNN	72.8%	Quick to Train	Instant
1000	Medium KNN	81.4%	Quick to Train	Instant
1000	Coarse KNN	81.6%	Quick to Train	Instant
1000	Cosine KNN	81.1%	Quick to Train	Instant
1000	Cubic KNN	81.4%	Quick to Train	Instant
1000	Weighted KNN	80.3%	Quick to Train	Instant
1000	Linear SVM	81.6%	Standard	Instant
1000	Linear SVM	**81.6%**	Optimied 30x	< 5 Minutes
2000	Fine Tree	75.3%	Quick to Train	Instant
2000	Medium Tree	83.0%	Quick to Train	Instant
2000	Coarse Tree	83.0%	Quick to Train	Instant
2000	Fine KNN	72.9%	Quick to Train	Instant
2000	Medium KNN	82.7%	Quick to Train	Instant
2000	Coarse KNN	83.0%	Quick to Train	Instant
2000	Cosine KNN	82.7%	Quick to Train	Instant
2000	Cubic KNN	82.7%	Quick to Train	Instant
2000	Weighted KNN	82.2%	Quick to Train	Instant
2000	Linear SVM	83.0%	Standard	Instant
2000	Linear SVM	**83.0%**	Optimied 30x	< 5 Minutes
3000	Fine Tree	77.4%	Quick to Train	Instant
3000	Medium Tree	83.5%	Quick to Train	Instant
3000	Coarse Tree	83.5%	Quick to Train	Instant
3000	Fine KNN	73.3%	Quick to Train	Instant
3000	Medium KNN	83.4%	Quick to Train	Instant
3000	Coarse KNN	83.5%	Quick to Train	Instant
3000	Cosine KNN	83.3%	Quick to Train	Instant
3000	Cubic KNN	83.4%	Quick to Train	Instant
3000	Weighted KNN	82.2%	Quick to Train	Instant
3000	Linear SVM	83.5%	Standard	Instant
3000	Linear SVM	**83.5%**	Optimied 30x	< 5 Minutes

At 1000 CRPs used as training data out of 10000 available CRPs, or 10% of available CRPS, we can achieve a testable 81.6% accuracy.

The following results use a n+3 parity vector.

Part 2 parity is n+3				
CRP No	**Classification Learner**	**Accuracy**	**Note**	**Time**
100	Fine Tree	86.0%	Quick to Train	Instant
100	Medium Tree	86.0%	Quick to Train	Instant
100	Coarse Tree	86.0%	Quick to Train	Instant
100	Fine KNN	84.0%	Quick to Train	Instant
100	Medium KNN	91.0%	Quick to Train	Instant
100	Coarse KNN	91.0%	Quick to Train	Instant
100	Cosine KNN	91.0%	Quick to Train	Instant
100	Cubic KNN	91.0%	Quick to Train	Instant
100	Weighted KNN	90.0%	Quick to Train	Instant
100	Linear SVM	90.0%	Standard	Instant
100	Linear SVM	**91.0%**	Optimied 30x	< 5 Minutes
500	Fine Tree	76.2%	Quick to Train	Instant
500	Medium Tree	79.0%	Quick to Train	Instant
500	Coarse Tree	85.4%	Quick to Train	Instant
500	Fine KNN	76.4%	Quick to Train	Instant
500	Medium KNN	**85.6%**	Quick to Train	Instant
500	Coarse KNN	85.4%	Quick to Train	Instant
500	Cosine KNN	85.4%	Quick to Train	Instant
500	Cubic KNN	85.6%	Quick to Train	Instant
500	Weighted KNN	84.8%	Quick to Train	Instant
500	Linear SVM	85.4%	Standard	Instant
500	Linear SVM	85.4%	Optimied 30x	< 5 Minutes
1000	Fine Tree	70.6%	Quick to Train	Instant
1000	Medium Tree	79.0%	Quick to Train	Instant
1000	Coarse Tree	81.6%	Quick to Train	Instant
1000	Fine KNN	72.3%	Quick to Train	Instant
1000	Medium KNN	81.7%	Quick to Train	Instant
1000	Coarse KNN	81.6%	Quick to Train	Instant
1000	Cosine KNN	81.5%	Quick to Train	Instant
1000	Cubic KNN	**81.7%**	Quick to Train	Instant
1000	Weighted KNN	80.7%	Quick to Train	Instant
1000	Linear SVM	81.6%	Standard	Instant
1000	Linear SVM	81.6%	Optimied 30x	< 5 Minutes
2000	Fine Tree	75.4%	Quick to Train	Instant
2000	Medium Tree	82.8%	Quick to Train	Instant
2000	Coarse Tree	83.0%	Quick to Train	Instant
2000	Fine KNN	73.5%	Quick to Train	Instant
2000	Medium KNN	82.8%	Quick to Train	Instant
2000	Coarse KNN	83.0%	Quick to Train	Instant
2000	Cosine KNN	82.8%	Quick to Train	Instant
2000	Cubic KNN	82.8%	Quick to Train	Instant
2000	Weighted KNN	82.5%	Quick to Train	Instant
2000	Linear SVM	83.0%	Standard	Instant
2000	Linear SVM	**83.0%**	Optimied 30x	< 5 Minutes
3000	Fine Tree	78.0%	Quick to Train	Instant
3000	Medium Tree	83.5%	Quick to Train	Instant
3000	Coarse Tree	83.5%	Quick to Train	Instant
3000	Fine KNN	73.8%	Quick to Train	Instant
3000	Medium KNN	83.3%	Quick to Train	Instant
3000	Coarse KNN	83.5%	Quick to Train	Instant
3000	Cosine KNN	83.3%	Quick to Train	Instant
3000	Cubic KNN	83.3%	Quick to Train	Instant
3000	Weighted KNN	82.7%	Quick to Train	Instant
3000	Linear SVM	83.5%	Standard	Instant
3000	Linear SVM	**83.5%**	Optimied 30x	< 5 Minutes

The following are results from an n+4 and n+5 parity vector.

Part 2 parity is n+4

CRP No	**Classification Learner**	**Accuracy**	**Note**	**Time**
100	Fine Tree	91.0%	Quick to Train	Instant
100	Medium Tree	91.0%	Quick to Train	Instant
100	Coarse Tree	90.0%	Quick to Train	Instant
100	Fine KNN	87.0%	Quick to Train	Instant
100	Medium KNN	91.0%	Quick to Train	Instant
100	Coarse KNN	91.0%	Quick to Train	Instant
100	Cosine KNN	91.0%	Quick to Train	Instant
100	Cubic KNN	91.0%	Quick to Train	Instant
100	Weighted KNN	90.0%	Quick to Train	Instant
100	Linear SVM	91.0%	Standard	Instant
100	Linear SVM	**91.0%**	Optimied 30x	< 5 Minutes
500	Fine Tree	78.2%	Quick to Train	Instant
500	Medium Tree	81.4%	Quick to Train	Instant
500	Coarse Tree	85.4%	Quick to Train	Instant
500	Fine KNN	77.6%	Quick to Train	Instant
500	Medium KNN	85.4%	Quick to Train	Instant
500	Coarse KNN	85.4%	Quick to Train	Instant
500	Cosine KNN	85.4%	Quick to Train	Instant
500	Cubic KNN	85.4%	Quick to Train	Instant
500	Weighted KNN	85.0%	Quick to Train	Instant
500	Linear SVM	85.4%	Standard	Instant
500	Linear SVM	**85.4%**	Optimied 30x	< 5 Minutes
1000	Fine Tree	70.1%	Quick to Train	Instant
1000	Medium Tree	77.2%	Quick to Train	Instant
1000	Coarse Tree	81.6%	Quick to Train	Instant
1000	Fine KNN	71.6%	Quick to Train	Instant
1000	Medium KNN	81.5%	Quick to Train	Instant
1000	Coarse KNN	81.6%	Quick to Train	Instant
1000	Cosine KNN	81.4%	Quick to Train	Instant
1000	Cubic KNN	81.5%	Quick to Train	Instant
1000	Weighted KNN	80.3%	Quick to Train	Instant
1000	Linear SVM	81.6%	Standard	Instant
1000	Linear SVM	**81.6%**	Optimied 30x	< 5 Minutes
2000	Fine Tree	74.6%	Quick to Train	Instant
2000	Medium Tree	82.3%	Quick to Train	Instant
2000	Coarse Tree	83.0%	Quick to Train	Instant
2000	Fine KNN	73.0%	Quick to Train	Instant
2000	Medium KNN	82.8%	Quick to Train	Instant
2000	Coarse KNN	83.0%	Quick to Train	Instant
2000	Cosine KNN	82.5%	Quick to Train	Instant
2000	Cubic KNN	82.8%	Quick to Train	Instant
2000	Weighted KNN	82.2%	Quick to Train	Instant
2000	Linear SVM	**83.0%**	Standard	Instant
3000	Fine Tree	79.2%	Quick to Train	Instant
3000	Medium Tree	83.5%	Quick to Train	Instant
3000	Coarse Tree	83.5%	Quick to Train	Instant
3000	Fine KNN	73.4%	Quick to Train	Instant
3000	Medium KNN	83.3%	Quick to Train	Instant
3000	Coarse KNN	83.5%	Quick to Train	Instant
3000	Cosine KNN	83.3%	Quick to Train	Instant
3000	Cubic KNN	83.3%	Quick to Train	Instant
3000	Weighted KNN	82.8%	Quick to Train	Instant
3000	Linear SVM	83.5%	Standard	Instant

Part 2 parity is n+5

CRP No	**Classification Learner**	**Accuracy**	**Note**	**Time**
2000	Linear SVM	83.0%		
3000	Linear SVM	83.5%		

We can see from the tabulated results that increasing the size of the parity vector from n+2 to n+5 only results in a minor increase the ML algorithm's accuracy (±0.1%). We can see from these results that the increase in size is not an effective way of more quickly attacking the error injected PUF.

We can also see that the ML algorithm has a maximum accuracy of ~83.5%. This is because of the randomly noise injected model used to generate the given PUF data. This is to be expected and as stated in the previous sections, in a practical and realistic application of the Arbiter PUF, we cannot expect 100% accuracy from the PUF itself when generating a CRP. If the noise injected Arbiter PUF (modeling a realistic Arbiter PUF) cannot implement 100% accurate CRP production, the ML algorithm cannot surpass this limitation.

Conclusion

In this project, we have implemented ML algorithms to attack a simulated PUF. The data set was separated into two sections, and ideal case and a fault injected case. The ideal PUF data set was used as a proof of concept. The Arbiter PUF, if a small number of CRPs can be obtained, is very vulnerable to a ML attack.

We have also provided an example of a more realistic PUF application. However, just like the ideal case, if a small number of CRPs can be obtained, it is still vulnerable to a ML attack. Especially given that in realistic applications, it is expected that the PUF itself will not be accurate 100% of the time, we can confidently say that the ML algorithm can effectively model the PUF even with an accuracy of 83.5%.

Bibliography

[1] C. Herder, M.-D. (. Yu, F. Koushanfar and S. Devadas, "Physical Unclonable Functions and Applications: A Tutorial," *Proceedings of the IEEE,* vol. Vol. 102, no. No. 8, August 2014.

[2] D. Lim, "Extracting Secret Keys from Integrated Circuits," 2004.

[3] U. Ruhrmair and D. E. Holcomb, "PUFs at a Glance".

[4] G. T. Becker, "The Gap Between Promise and Reality: On the Insecurity of XOR Arbiter PUFs," Ruhr Universitat Bochum.