A Machine Learning-based Security Vulnerability Study on XOR PUFs for Resource-Constraint Internet of Things

Ahmad O. Aseeri*, †, Yu Zhuang*, Mohammed Saeed Alkatheiri§
*Department of Computer Science, Texas Tech University, United States
†Department of Computer Science, Prince Sattam bin Abdulaziz University, Saudi Arabia
*ahmad.aseeri@ttu.edu, †yu.zhuang@ttu.edu
§Faculty of Computing and Information Technology, University of Jeddah, Saudi Arabia
§msalkatheri@uj.edu.sa

Abstract—Physical unclonable functions (PUFs) are emerging as a promising class of hardware primitives for delivering security for IoT devices. Cryptographic key-based security mechanisms are heavy-weight by demanding resources more than many resource-constraint IoT devices can provide. and are also vulnerable to side-channel invasive attacks. PUFs utilize integrated circuits’ manufacturing variations to produce responses unique for individual devices, and hence cannot be reproduced. An important goal of security research is to discover all possible insecure risks, which can provide secure application developers useful information so that they can avoid the risk-containing components or mechanisms. While physically unclonable, some PUFs have been found to be mathematically clonable by machine learning methods. Large XOR arbiter PUFs is one group of PUFs that were shown to withstand existing attack methods unless long training time is used in the machine learning process. In this paper, we investigate the effectiveness of a neural network method in attacking large XOR PUFs, a neural network method modified to handle training datasets possibly larger than memory capacity. Our study shows that the modified neural network method attains high prediction accuracy while consuming substantially less time for large XOR PUFs than the fastest machine learning code used in all earlier studies known to us. Some of the large XOR PUFs that took existing machine learning codes several days of parallel computing time on high-performance computing servers have been broken by our method in less than two hours, indicating vulnerability of even large XOR PUFs. Discovery of all potential vulnerabilities of a PUF is important since secure application developers need such information for deciding which PUF to choose, and an unidentified vulnerability can lead to security risks for IoT devices.

Keywords: Physical Unclonable Functions, Hardware Security, Machine Learning, Internet of Things

I. INTRODUCTION

Communication security is of fundamental importance for internet of things (IoTs), and authentication is one of the methods of choice for securing the communications between IoT devices. Classical authentication techniques rely on cryptographic algorithms that utilize secret keys stored in non-volatile memory. Keys in non-volatile memory are vulnerable to sophisticated invasive side-channel attacks. In addition, many IoT devices are of limited resource, impractical to adopt resource-demanding security protocols. Cryptographic algorithms are heavy-weight as pointed out by a 2016 study [1], since a cryptographic algorithm needs a processor, at least a task-specific processor, to execute the software that implements the algorithm. Physical unclonable functions (PUFs) utilize variations in integrated circuits to produce responses unique for individual PUFs, and hence are not reproducible by manufacturers as well as by attackers even if they have obtained the exact PUF circuit design, possessing great potential for implementing secure mechanisms. Implementable with very simplistic circuits with extremely low energy and other resource requirements, PUFs are particularly promising for delivering high security for resource-constraint IoT devices.

An important part of work in security is to discover all possible insecure risks. Information on security vulnerabilities is useful for PUF designers for developing new PUFs to overcome these existing vulnerabilities, as well as for PUF-utilizing IoT device developers who can avoid some PUFs special vulnerabilities the IoT device cannot eliminate at application level, while adopt other PUFs with different security features or risks the IoT can take care of. While physically unproducible, PUFs are reported to be “mathematically clonable” by machine learning-based modeling methods which can accurately predict the responses of PUFs. Mathematical clonability allows attackers to develop malicious software to impersonate the identity of trusted PUF-embedded devices by producing the same responses PUFs would give.

Since 2010, there has been quite some work on studying the vulnerability of PUFs under machine learning attacks [2], [3]. As shown in those studies, the XOR Arbiter PUFs (XOR PUFs) with large numbers of component arbiter PUFs (six or more component arbiter PUFs) are among the PUFs that possess high resistance against machine learning attacks. For instance, it took a memory-optimized parallel machine learning code [3] on 16 cores almost 3 days (or about 44 days of sequential computing time) to break the 128-bit 7-XOR arbiter PUF. Most of the existing machine learning methods used for attacking PUFs are based on
the logistic regression with resilient propagation (RProp-LR) [2]. Recently, two of the authors of this paper investigated a neural network-based method on attacking the feed-forward arbiter PUFs (FF PUFs) [4], and were able to break the FF PUFs which were unbreakable by other existing methods by [2].

Due to the high effectiveness of neural network methods in revealing the previously undiscovered vulnerability of FF PUFs as shown in the study [4], we are motivated to investigate if large XOR PUFs with five or more component arbiter PUFs can resist neural network-based attacks, and in particular, in this paper we are interested in finding out if large XOR PUFs can be broken by neural network-based attack methods on laptop computers. Since existing machine learning attack studies were using high performance computing servers, and since laptop computers are of limited computing resources and also are more easily accessible and obtainable by potential attackers, the breakability of PUFs by laptop computers would be a more severe risk that deserves investigation. The neural network-based attack method employed in [4] is a machine learning procedure adapted to the peculiar architecture of FF PUFs, and hence is not applicable to XOR PUFs. In addition, breaking XOR PUFs with large numbers of component arbiter PUFs may need large training datasets of challenge-response pairs (CRPs), which could exceed memory capacity of a laptop computer. To handle large number of CRPs, we have modified an optimization method employed in the neural network to enable the feasibility of machine learning on datasets larger than memory capacity. The modified neural network method is applied to 64-bit and 128-bit XOR PUFs, showing its ability to break large XOR PUFs substantially faster than existing machine learning attack methods. And in one case it took less than two hours of sequential computing time to break a PUF which took the fastest existing method 3 days of 16-core parallel computing time, or equivalently 44 days sequential computing time. The study presented in this paper reveals a vulnerability of large XOR PUFs which has not been discovered by earlier studies, providing IoT secure application developers valuable information for deciding which PUF to avoid for potential secure risks.

II. OVERVIEW OF PHYSICAL UNCLONABLE FUNCTIONS

Physical Unclonable Functions (PUFs) are increasingly used for the purpose of authentication and cryptographic key generation [5]. The motivation behind the development of PUFs emanates from the fact that classical cryptographic techniques, which mostly rely on the use of secret keys, fall short into two main shortcomings. First, cryptographic secret keys are traditionally stored in digital nonvolatile memories thus vulnerable to different type of attacks such as side-channel [6] and invasive physical attacks [7]. Second, due to resource constrains, power consumption, and manufacturing costs, electronic devices may not suitable to use secret key-based cryptographic algorithms in certain commercial scenarios [5] [8]. On the other hand, PUFs provide an alternative approach to achieving security and privacy, especially for resource-constrained devices like smart cards and sensors, which cannot support computationally-expensive cryptographic techniques.

Physical unclonable functions leverage manufacturing variations that occur in physical chips at very small scales. These variations make each chip unique and hence can be exploited to prevent semiconductor re-fabrication. Although this phenomenon is regarded as a side effect in integrated circuits design (ICs) [8], it is considered advantageous for security purposes in physical systems. That is, instead of storing secrets in a digital nonvolatile memory, PUFs retrieve the secret information present in the inevitable physical variations which are treated as unique signatures of the silicon chips. Note that PUF manufacturing variability can be defined in multiple ways. Gate delays, power-on state of SRAM, and threshold voltages are among the most common ways of utilizing physical characteristics to derive the secrets [5].

Fig. 1 shows the generic diagram of a silicon PUF circuit. In a nutshell, PUF is an input-output mapping in the form $T : \{0, 1\}^n \rightarrow \{0, 1\}^m$ where the input is $n$-bit binary vector (called challenge) and the output is $m$-bit binary vector (called response). A PUF circuit can receive $2^n$ different possible $n$-bit challenge vectors, each of which produces an $m$-bit output. Silicon PUFs can generate unique challenge-response pairs (CRPs) for different integrated circuit instances [9].

A. Arbiter PUFs

Arbiter PUF is a common implementation of delay-based silicon PUFs introduced in [10]. It is based on measuring the time delay difference between two symmetric paths. The signals race against each other simultaneously through a
sequence of \( n \) stages, each of which consists of two multiplexers (MUXes) [5] [8]. At each stage, the signal transition as well as the delay difference between the two paths depend on the challenge bit which controls how the MUXes behave, i.e., either straight or crossing as illustrated in Figure 2. The delay introduced by the multiplexers varies with the input challenge bits, and an arbiter, mostly implemented by a latch, determines the final output to be either 0 or 1. Mostly, if the top path arrives first, then the arbiter output would be 1, otherwise the output is 0.

A model of arbiter PUFs that all machine learning attack methods are based upon was introduced in [11]. It is a model of additive delays of the signals at all stages of the PUF circuit. The output of the arbiter is determined by the delay difference between the two paths that enter the arbiter. The arbiter (latch), as illustrated in Figure 2, outputs 1 if the signal reaches the D input bit earlier than the signal reaches the C input bit, and outputs 0 otherwise. For an arbiter with \( n \) stages, where each stage consists of two multiplexers, the elapsed times of the two signals arriving at the arbiter from the time the signal starts entering the first stage of the PUF are the summations of the delays incurred at each stage of the PUF.

Figure 3 illustrates how arbiter PUFs are modeled. For the \( i^{th} \) stage, we denote by \( D_t(i) \) the elapsed time of the signal arrives at the top output bit from the time the signal starts entering the first stage, and denote by \( D_b(i) \) the elapsed time of the signal arrives at the bottom output bit. For the \( i^{th} \) stage, depending on the value of the challenge bit to this stage, the signal of top output bit could come from the top output bit or the bottom output bit of the \( (i-1)^{th} \) stage. Similarly, the signal of bottom output bit of the \( i^{th} \) stage could come from the top output bit or the bottom output bit of the \( (i-1)^{th} \) stage. Thus, we denote by \( \delta_t(i), \delta_b(i), \delta_{tb}(i), \text{ and } \delta_{bt}(i) \), respectively, the incurred time delay for a signal goes from top bit of stage \( (i-1) \) to top bit of stage \( i \), the incurred delay for a signal goes from bottom bit of stage \( (i-1) \) to bottom bit of stage \( i \), and the incurred delay for a signal goes from bottom bit of stage \( (i-1) \) to bottom bit of stage \( i \). We use \( c_i \) to denote the challenge bit at the \( i^{th} \) stage with \( c_i \) taking the value of 0 or 1.

With the notations given above, the signal delays at the \( i^{th} \) stage has the following relation with delays at the \( (i-1)^{th} \) stage:

\[
D_t(i) = c_i[D_t(i-1) + \delta_t(i)] + (1 - c_i)[D_b(i-1) + \delta_{bt}(i)]
\]

\[
D_b(i) = c_i[D_b(i-1) + \delta_{tb}(i)] + (1 - c_i)[D_t(i-1) + \delta_b(i)]
\]

meaning that the delays at stage \( i \) are equal to the delays at stage \( (i-1) \) plus the newly incurred delays. Then, the difference between top and bottom signal delays at the \( i^{th} \) stage, denoted by \( \Delta_i \), i.e.,

\[
\Delta_i = D_t(i) - D_b(i),
\]

is:

\[
\Delta_i = c_i[\Delta_{i-1} + \delta_t(i) - \delta_{bt}(i)] + (1 - c_i)[\Delta_{i-1} + \delta_b(i) - \delta_{tb}(i)]
\]

Let \( \alpha(i) = \delta_t(i) - \delta_{bt}(i) \) and \( \beta(i) = \delta_b(i) - \delta_{tb}(i) \), the equation above can be re-written into:

\[
\Delta_i = (2c_i - 1) \left[ \Delta_{i-1} + \frac{\alpha(i) - \beta(i)}{2} \right] + \frac{\alpha(i) + \beta(i)}{2}
\]

from which it can be derived that:

\[
\Delta_n = \phi(1)w(1) + \phi(2)w(2) + \cdots + \phi(n)w(n) + \frac{\alpha(n) + \beta(n)}{2},
\]

where

\[
\phi(i) = (2c_i - 1)(2c_{i+1} - 1) \cdots (2c_n - 1),
\]

and

\[
w(i) = \begin{cases} 
\frac{\alpha(1) - \beta(1)}{2}, & i = 1 \\
\frac{\alpha(i) - \beta(i)}{2} + \frac{\alpha(i-1) + \beta(i-1)}{2}, & i = 2, \ldots, n
\end{cases}
\]

A comment about \( \phi(i) \)'s is that they take value of 1 or \(-1 \). Since the arbiter's response is 1 if \( \Delta_n > 0 \) and is 0 otherwise, the response \( r \) can be expressed as:

\[
r = \text{sgn} \left( \phi(1)w(1) + \cdots + \phi(n)w(n) + \frac{\alpha(n) + \beta(n)}{2} \right)
\]

where \( \text{sgn}(\cdot) \) is the sign function.
Equation (2) is usually called the linear additive delay model for arbiter PUFs because of the linear term with respect to $\phi$'s inside the sign function. Actually, it is the linearization transformation (Equation 1) that transforms the highly nonlinear relationship between response $r$ and challenge vector $C$ into a linear function of $\phi$'s inside the sign function. This linear transformation is one of the key steps that makes machine learning attacks of PUFs feasible, since otherwise the highly nonlinear relationship between $r$ and $C$ is formidable for any machine learning method. Since the term inside the $\text{sgn}(\cdot)$ function is linear with respect to the transformed challenges $\phi$'s, a machine learning attack method of arbiter PUFs is to identify the hyperplane that separates the high-dimensional space of transformed challenges $\phi$'s into two subspaces with $r = 0$ in one subspace and $r = 1$ in the other subspace.

B. XOR Arbiter PUFs

An XOR arbiter PUFs [2], [12], or XOR PUFs for short, is built from multiple arbiter PUFs. As illustrated in Fig. 5, the $k$-XOR arbiter PUF uses $k$ arbiter PUFs as components, where all of the $k$ arbiter PUFs use the same challenge $C$ as the challenge input. The responses of all individual arbiter PUFs are XORed together to produce the final response $r$ for the corresponding input challenge $C$. Thus, the response of the $n$-stage $k$-XOR arbiter PUF in Figure 5 can be expressed as:

$$ r = \bigoplus_{j=1,...,k} r_j, \quad (3) $$

where $r_j$ is the internal output of the $j^{th}$ component arbiter PUF.

The XOR operation increases non-linearity of the relationship between the response $r$ and the transformed challenges $\phi$'s. It is obvious that adding arbiter PUFs increases the chip area as well as cost needed for silicon implementation of the PUFs. But, every additional arbiter PUF increases nonlinearity as well as the dimensionality of the parameter space to be machine-learned by attackers [2], leading to higher resistance against machine learning attacks [11]. For more details on PUF designs, the types, and their specific variations, we refer the reader to [2], [5].

III. MACHINE LEARNING ATTACK OF XOR ARBITER PUFs

There have been studies [2], [3] that successfully revealed vulnerability in mathematical clonability of XOR arbiter PUFs using machine learning procedures. The machine learning method employed in [2], [3] is the logistic regression with resilient propagation (RProp-LR). The RProp-LR method used in [3] was optimized for memory access performance and parallel processing, leading to reduced training time for breaking XOR PUFs. Figure 4 illustrates the generic process to perform machine learning-based attack on XOR PUF. It consists of two self-explanatory stages: preprocessing and machine learning. In preprocessing stage, linear transformation of the input challenge set is applied, as shown in equation (1). In machine learning stage, the model (3) is trained using a machine learning method. After the training has successfully completed, the response of given challenges can be predicted using the trained model.

Since neural network (NN) is one of the most powerful machine learning approaches for supervised learning. In
this paper, we decide to investigate the effectiveness of NN methods for attacking XOR PUFs. In particular, we plan to look into Multi-Layered Perceptron (MLP), one of the highly effective yet easy-to-use neural network methods [13]. Another reason that has motivated us to investigate MLP-based attack methods is that very recently an MLP method was used to break feed-forward arbiter PUFs which were unbreakable by other methods [2]. The MLP-based attack method introduced in [4] is a machine learning procedure adapted to the peculiar architecture of FF PUFs, and hence not applicable to XOR PUFs. In addition, breaking XOR PUFs with large numbers of component arbiter PUFs may need large training datasets of challenge-response pairs (CRPs), which could exceed memory capacity of a laptop computer. To handle large number of CRPs, in this paper we have modified an optimization method employed in the MLP method to enable the feasibility of machine learning on datasets larger than memory capacity. Existing machine learning attack studies by [2], [3] were using high performance computing servers, which usually have large memory capacity for each compute node. In this paper, we assume limited memory capacity for our machine learning attack method so that our method is effective even on laptop computers. Since laptop computers are usually of limited computing resources and also are more affordable and easily obtainable by potential attackers, the breakability of PUFs by laptop computers would be a more severe risk that deserves investigation.

Our attack method uses multi-layered perceptron (MLP) method. For an MLP, given a training set data \((x_i, y_i)\) for \(i = 1, 2, \ldots, n\), where \(n\) is the number of data items in the training set, the output of each neuron in one layer of the neural network is the result of an activation function whose input is the linear combination of all outputs of neurons at the earlier layer, with the first layer of neurons having the linear combination of all scalar entries of the vector \(x_i\), where the coefficients of the linear combination at one layer is called the weight vector for that layer of neurons. This process is proceeded from the input layer towards the output layer where the final model output, \(\hat{y}_i\) for \(i = 1, 2, \ldots, n\), is generated. When the model outputs \(\hat{y}_i\) for \(i = 1, 2, \ldots, n\) is not close to the observed result \(y_i\) for \(i = 1, 2, \ldots, n\), the weight vectors are tuned in such a way that would reduce the difference between the model output and the observed result in the upcoming iterations.

In the multi-layered perceptron algorithm, the values of \(w(i)\)'s, as in equation (2) for each of the component arbiter PUFs whose output \(r_j\)'s are fed to the XOR gate of the XOR PUF that is modeled by (3), are iteratively refined, by training the delay model (2) on a set of challenge-response pairs (CRPs). After iterations of training, as the weights \(w(i)\)'s are successfully determined if the training dataset is adequately large, the correct response for a given challenge \(C\) can be

---

### Algorithm 1 XOR PUF Breaker

**Variables Description:**
- \(\mathbb{C}\) is the set of CRPs.
- \(\gamma\) is a chunk, and \(\eta\) is the chunk size.
- \(\alpha\) is training time.
- \(\theta\) is predication accuracy.
- \(\Phi\) is weight vector.

```plaintext
1: procedure XOR_BREAKER(\mathbb{C}, \eta)
2: \(\Phi\) ← initialize weights
3: if size(\mathbb{C}) \leq memory_capacity then
4: \(tr_set, ts_set\) ← generate(\mathbb{C}, \mathbb{C} \times 0.2)
5: \(\theta, \Phi, \alpha\) ← train_model(tr_set, ts_set, \Phi)
6: else
7: \(num\_chunks = \mathbb{C}/\eta\)
8: \(round\) ← 1
9: while \(\theta < 98\%\) do
10: for \(j \leftarrow 1\) to \(num\_chunks\) do
11: \(\text{if \(\text{round}\) is \(1\) then}
12: \(tr\_set, ts\_set\) ← generate(\(\gamma_j, \gamma_j \times 0.2\))
13: \(\text{shuffle\_and\_store(tr\_set, ts\_set)}\)
14: \(\text{else}
15: \(\text{load(tr\_set, ts\_set)}\)
16: \(\text{end if}
17: \(\theta, \Phi, \alpha\) ← train\_model(tr\_set, ts\_set, \Phi)
18: \(\text{if \(\theta \geq 98\%\) then exit procedure}
19: \(\text{end for}
20: \(\text{round} ++\)
21: \(\text{end while}
22: \(\text{end if}
23: end procedure
```

---

### Algorithm 2 Train Model

```plaintext
1: procedure TRAIN\_MODEL(train\_set, test\_set, \Phi)
2: model ← train(train\_set, \Phi)
3: \(\theta\) ← evaluate(test\_set, model)
4: \(\Phi\) ← get\_updated\_weights(model)
5: \(\alpha\) ← get\_train\_time(model)
6: return \(\theta, \Phi, \alpha\)
7: end procedure
```
dataset into smaller *chunks* and perform an accumulative training process in such a way that the weight vector resulted from the training on a chunk is passed as the input weight vector for training on the next chunk, as illustrated in Algorithm 1, in which we denote by one *round* that all chunks have been visited once in the training procedure but each chunk may be visited multiple times, or multiple iterations, during the training on each chunk. Algorithm 2 is the method applied to the training on a chunk that can fit in the memory. The working mechanism of the two algorithms are elaborated in the following.

The procedure `XOR_BREAKER` in Algorithm 1 takes two parameters: the size of challenge-response dataset and the chunk size which should be less than or equal to the machine’s available memory. The second procedure `TRAIN_MODEL` in Algorithm 2 is to train our model stated in equation (3). It takes three parameters: train set, test set, and the weight vector $\Phi$ to be tuned during the machine learning process. At the beginning of `XOR_BREAKER` procedure, the algorithm decides if the input challenge-response dataset $C$ can fit into the memory or not (line 3). If it can, the training algorithm splits the dataset into a train set, $tr\_set$, and a test set, $ts\_set$, followed by performing model training via the call of `TRAIN_MODEL` procedure on the train set $tr\_set$. This procedure returns three values: percentage accuracy $\theta$, training time $\alpha$, and the updated weight vector $\Phi$ to be used as the input weight vector for training on the next chunk. On the other hand, if the challenge-response dataset $C$ cannot fit into the memory (line 6), the dataset $C$ is partitioned into smaller chunks, each less than or equal to the available memory capacity (line 7). The outer loop (while loop) from line 9 to 21 iterates over all $n$ chunks, possibly multiple times (i.e., multiple rounds), until the percentage accuracy is greater than or equal to 98%. The inner loop (the for loop) from line 10 through line 19 iteratively performs model training and evaluation on each chunk. The first round is a bit special from the subsequent rounds because it involves generating each of $n$ chunk (line 12) followed by storing/appending it (after being shuffled) into a file on the disk (line 13). The subsequent rounds involve only loading a chunk from the file (line 15). Note that when the percentage accuracy becomes $\geq 98\%$ during the training of any chunk, we don’t need to wait for the entire round to finish as stated in (line 18).

### IV. EXPERIMENTAL STUDIES

#### A. Experimental Setup

In our simulation environment, we designed a challenge-response pair (CRPs) generator such that the challenges are randomly drawn from the range $(0, 2^n)$ where $n$ is the number of bits of a challenge. The arbiter delays are drawn using a Gaussian distribution with mean 300 and standard deviation 40. For machine learning, we use scikit-learn [14], a library that comes with an implementation of various neural network methods. We modified the optimization method employed in scikit-learn to handle datasets larger than memory capacity.

Multi-layer Perceptron (MLP) can iteratively reduce the loss function using different optimizers like gradient descent, Adaptive Moment Estimation (ADAM) [15], and Limited-memory BFGS [16]. We have found out that ADAM optimizer works well, with relatively large CRPs, in both training time and validation score. Hence, we selected the ADAM optimizer. Table I summarizes selected MLP parameters for training our model.

To evaluate the trained model’s predictive power as a binary classifier, the prediction accuracy is used that measures the percentage (of 100% top accuracy) of the responses correctly predicted by the model. For each $k$-XOR PUF, we run the experiment 10 times and report the average accuracy, average training time, and the best accuracy where the model of the best accuracy is extracted. In addition, to examine the performance and predictability of our approach, we compare our results against existing two works on XOR arbiter PUFs, Ruhrmair at el., (2010) [2] and Tobisch and Becker (2015) [3]. The training time reported by both works is the training time for the best accuracy. It is worth mentioning that Tobisch and Becker method employs parallel computation using 16 cores, thus their training times shown in this paper is multiplied by 16 in order to be consistent with the sequential model training times by our approach as well as by Ruhrmair at el. approach.

### V. RESULTS AND DISCUSSION

Table II contains the results of our experiment. The first table II-(a) contains the results for 64-bit XOR PUF, while the second table II-(b) shows the results for 128-bit XOR PUF. In addition, we use letters A, B, and C to denote the three different methods. That is, A indicates the work by Tobisch and Becker (2015), B indicates the work by Ruhrmair at el., (2010), and finally our work is denoted by the letter C. The experiments were performed on a MacBook.
Table II: Results of XOR Arbiter PUF Using Laptop Computer of 16GB Memory.

(a) 64-bit PUFs

<table>
<thead>
<tr>
<th>Stages</th>
<th>XOR Size</th>
<th>Method</th>
<th>Training Set (CRPs (\times 10^6)) (GB)</th>
<th>Chunk Size (CRPs (\times 10^6)) (GB)</th>
<th>Accuracy</th>
<th>Training Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Best</td>
<td>Average</td>
<td></td>
<td>Best</td>
<td>Average</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>42 (\times 10^3)</td>
<td>n/a</td>
<td>98%</td>
<td>n/a</td>
<td>32 sec</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>12 (\times 10^3)</td>
<td>n/a</td>
<td>99%</td>
<td>n/a</td>
<td>3.42 min</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0.4 (\times 10^6) (0.2 GB)</td>
<td>–</td>
<td>98.86%</td>
<td>98.42%</td>
<td>19.2 sec</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>A</td>
<td>260 (\times 10^3)</td>
<td>n/a</td>
<td>98%</td>
<td>n/a</td>
<td>2.13 min</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>80 (\times 10^3)</td>
<td>n/a</td>
<td>99%</td>
<td>n/a</td>
<td>2.08 hrs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0.8 (\times 10^6) (0.4 GB)</td>
<td>–</td>
<td>98.83%</td>
<td>95.55%</td>
<td>58 sec</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>1.4 (\times 10^3)</td>
<td>n/a</td>
<td>98%</td>
<td>n/a</td>
<td>16.16 min</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>200 (\times 10^3)</td>
<td>n/a</td>
<td>99%</td>
<td>n/a</td>
<td>31.01 hrs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2 (\times 10^6) (1 GB)</td>
<td>–</td>
<td>99.24%</td>
<td>99.15%</td>
<td>7.4 min</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>A</td>
<td>20 (\times 10^6)</td>
<td>n/a</td>
<td>98%</td>
<td>n/a</td>
<td>14.49 hrs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>5 (\times 10^6) (2.5 GB)</td>
<td>–</td>
<td>99.28%</td>
<td>99.21%</td>
<td>11.8 min</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>A</td>
<td>150 (\times 10^6)</td>
<td>n/a</td>
<td>98%</td>
<td>n/a</td>
<td>4.2 days</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>30 (\times 10^6) (15 GB)</td>
<td>14 (\times 10^6) (7 GB)</td>
<td>99.17%</td>
<td>98.74%</td>
<td>23.3 min</td>
<td></td>
</tr>
</tbody>
</table>

(b) 128 bit PUFs

<table>
<thead>
<tr>
<th>Stages</th>
<th>XOR Size</th>
<th>Method</th>
<th>Training Set (CRPs (\times 10^8)) (GB)</th>
<th>Chunk Size (CRPs (\times 10^8)) (GB)</th>
<th>Accuracy</th>
<th>Training Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Best</td>
<td>Average</td>
<td></td>
<td>Best</td>
<td>Average</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>200 (\times 10^3)</td>
<td>n/a</td>
<td>98%</td>
<td>n/a</td>
<td>1.33 min</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>24 (\times 10^3)</td>
<td>n/a</td>
<td>99%</td>
<td>n/a</td>
<td>2.53 hrs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0.8 (\times 10^6) (0.8 GB)</td>
<td>–</td>
<td>98.69%</td>
<td>98.46%</td>
<td>1.32 min</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>A</td>
<td>2.2 (\times 10^6)</td>
<td>n/a</td>
<td>98%</td>
<td>n/a</td>
<td>14.4 min</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>500 (\times 10^3)</td>
<td>n/a</td>
<td>99%</td>
<td>n/a</td>
<td>16.36 hrs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>3 (\times 10^6) (3 GB)</td>
<td>–</td>
<td>98.88%</td>
<td>98.70%</td>
<td>5 min</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>15 (\times 10^9)</td>
<td>n/a</td>
<td>98%</td>
<td>n/a</td>
<td>71.2 hrs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>2 (\times 10^8) (20 GB)</td>
<td>14 (\times 10^6) (14 GB)</td>
<td>99.22%</td>
<td>99.03%</td>
<td>19 min</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>A</td>
<td>40 (\times 10^8)</td>
<td>n/a</td>
<td>98%</td>
<td>n/a</td>
<td>44.3 days</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>40 (\times 10^8) (40 GB)</td>
<td>14 (\times 10^6) (14 GB)</td>
<td>99.19%</td>
<td>99%</td>
<td>1.5 hrs</td>
<td></td>
</tr>
</tbody>
</table>

Pro laptop computer with an Intel i7 processor 2.2GHz and 16GB of memory. We used a single core in our experiments.

In Table II, the column “Training Set” lists the sizes of training datasets used that attained successful machine learning training of the models, the column “Chunk Size” lists the chunk size upper bounds used in our MLP method, and the column “Speedup” is ratio of the shorter training time of A and B to the training time of our approach, which is an indicator of how many times “faster” or “slower” our approach as compared with the better of the two existing methods. Because the model training is performed on a laptop computer of relatively small memory capacity (16GB), our trials showed that the maximum number of CRPs couldn’t exceed 28 million CRPs per chunk for 64-bit PUFs (that is about 14 GB of data without counting the memory overhead of the Python scikit-learn package) and 14 million per chunk for 128-bit PUFs (that is about 14 GB of data).

As seen in both tables, our approach has produced prediction accuracies as high as the other two. On the other hand, for large XOR sizes with XOR size greater than 6, ours significantly outperforms A and B in both training times and the size of the training data set needed for successful training. And especially in terms of the training time, for the PUFs with XOR sizes of 7 or larger, the experimental data show that our method is over 70 times faster to 700 times faster.

Since XOR PUFs with large sizes of XOR gates (7 or beyond) were shown to able to resist machine learning attacks in earlier studies known to us, as indicated by the large training datasets and long training times listed in Tables II for A and B. The substantial reduction by our method in
both dataset size and training time show that even XOR PUFs with 7 or 8 component arbiter PUFs are not secure enough to withstand machine learning attacks. Note that although we carried out experiments up to only 8-XOR PUFs, we anticipate that we could get similar outstanding results if we continue to train our model for PUFs with XOR size larger than 8. However, it is unlikely that this many PUFs will be used in practice due to the expected response unreliability and the overhead associated with PUFs of large size.

VI. Conclusion

In this paper, we have presented a new machine learning procedure for attacking XOR Arbiter PUF, which requires smaller training datasets and is of much higher efficiency for large XOR PUFs. The machine learning procedure employs a multi-layer perceptron method whose internal optimizer is modified to handle training datasets larger than memory capacity, allowing the machine learning procedure to run on laptops of limited size of memory. Experimental tests were performed on XOR PUFs of various sizes and the results show that large XOR PUFs can be broken in one and half hours or less on a laptop computer, uncovering a new vulnerability of these PUFs.

VII. Acknowledgement

The research was supported in part by National Science Foundation under Grant No. CNS-1526055.

References


