Lorax: Machine Learning-Based Oracle Reconstruction With Minimal I/O Patterns

Wei Zeng, Azadeh Davoodi
Department of Electrical and Computer Engineering
University of Wisconsin–Madison
Madison, WI, USA
{wei.zeng, adavoodi}@wisc.edu

Rasit Onur Topaloglu
IBM
Hopewell Junction, NY, USA
rasit@us.ibm.com

***Abstract*—This paper proposes a new attack model where the attacker tries to reconstruct a combinational logic circuit without having full oracles access. This means due to limited access time to the product, the attacker has access to only a limited number of input-output (I/O) pairs and does not have any information about the design. The goal of the attacker is to reconstruct a circuit to be deployed with simulation or emulation, in order to act as an efficient surrogate to perform fast attacks.**

We propose Lorax, the first automated framework to reconstruct circuits from limited access, using tree-based machine learning (ML) models of different configurations. It features early estimation of accuracy of the reconstructed oracle using cross-validation, as well as approximation techniques for efficient synthesis of the learned logic. For cases that are difficult to learn, Lorax applies a special function matching phase utilizing an explanatory analysis of a tree-based ML model to identify bit importance. Our experiments show that with a training set of 6400 I/O pairs, Lorax can successfully approximate commonly-used functions from a range of sources, including arithmetic circuits, industrial designs, and computer vision problems, with an accuracy of 79–84% on average and near 100% for some arithmetic functions.

***Index Terms*—machine learning; logic reconstruction**

I. Introduction

Oracle access to a logic circuit is a core step that is evoked in important problems in hardware security such as logic locking [1], [2], logic camouflage [3], gate-level reverse engineering [4], physically unclonable function (PUF) attacks [5], etc. It assumes an attacker can query the circuit and obtain the output corresponding to any desired inputs patterns. As one example, AppSAT [1] exploits oracle access to derive an approximate key using a Boolean satisfiability solver during logic locking. As another example, Ganji *et al.* [6] show how oracle access can be utilized on top of machine learning (ML) techniques to learn logic locking schemes and PUFs.

The above cases often assume availability of *full* oracle access, where the attacker is assumed to have unlimited access to the circuit so they can query the output for any input pattern, as many times as desired. However, this assumption may not be realistic in many scenarios because access to the product may be limited to a short duration of time.

Recently, Shamsi *et al.* include a brief study of oracle learnability in the context of logic locking [7]. However, the study on oracle learnability itself was quite limited and not the focus of their work [7]. Similarly, the work [5] studied oracle learnability but with restriction to assumptions about PUFs.

The work [8] uses deep RNNs to attack logic locking with less than 0.5% of input-output (I/O) pairs in the input space. However, the number of patterns may still be very large even for small circuits (e.g., 21 million for a circuit with 32 inputs, which is impractical if attacker has access for a limited time).

In this paper, we focus on the scenario where the attacker does not have full oracle access to a target circuit, nor do they have access to design information such as port names, netlist or layout. This attack scenario corresponds to the case when the time an attacker can access a product is limited. This occurs, for example, when the product is getting tested in the lab. They only have access to the I/O pairs which because of the short duration of access time is tiny-sized ($\ll 2^m$ for m inputs). The I/O pairs are uniformly sampled at random from the input space to capture the fact that no specific knowledge about I/O pairs is known in advance by the attacker. The goal of the attacker is to reconstruct a gate-level circuit that is as close as possible to the target in functionality. This is done while imposing a limit on delay of the reconstructed circuit, so it can serve as an efficient surrogate for simulation or emulation. For example, if some applications allow errors or low precision, a reconstructed circuit could be sold on the market with slightly lower performance criteria than the original.

This work is the first to propose and study the above attack model. We propose Lorax, an oracle reconstruction framework that utilizes novel techniques. Lorax automatically explores tree-based ML models of different configurations. It features early estimation of accuracy of a reconstructed oracle using cross validation, and approximation techniques for efficient synthesis of the learned logic. Lorax is also able to evoke a special matching phase to check for existence of standard functions which are known to be hard to learn. This matching phase is guided by an explanatory analysis of the ML model to identify bit importance. None of the techniques which are utilized by Lorax have been explored in prior work.

We experiment with functions from a variety of domains, including arithmetic circuits, industrial designs, symmetric functions, and computer vision problems, with an average of 179 inputs. We show they can be reconstructed with only 6400 I/O patterns, with an average accuracy of 79%–84% depending on the desired delay. Especially, we achieve an average of 94%

This research was supported by Award #1812600 from National Science Foundation and by Task #2845 from Semiconductor Research Corporation.

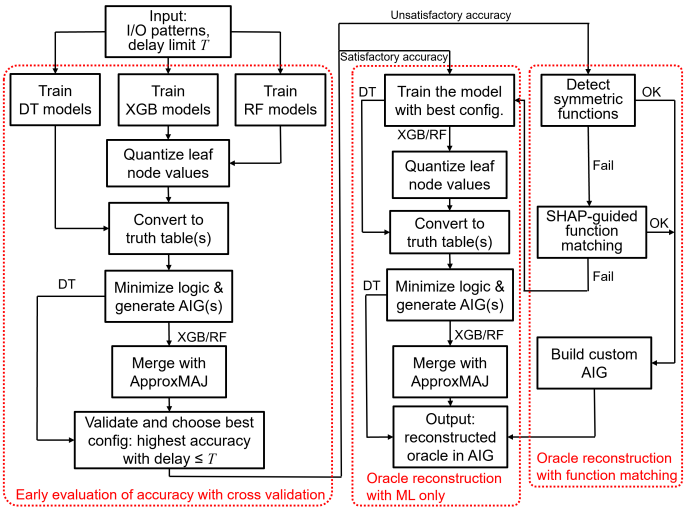


Fig. 1. Overall workflow of ML-based oracle reconstruction with Lorax.

reconstruction accuracy for functions from industrial designs. We show function matching with explanatory analysis of ML model is effective for some hard-to-learn functions, of which the reconstruction accuracy is further improved to near 100%.

We note, the scope of our work (including validation) does not yet include the case when the *function* of the accessible circuit is locked or otherwise altered. The novelty of our work is proposing the limited-access attack model, as well as illustrating the effectiveness of Lorax for a variety of application domains.

II. Attack Model

We define the following attack model for oracle reconstruction. For an m-input unknown Boolean function $f : \mathbb{B}^m \rightarrow \mathbb{B}$ where $\mathbb{B} = \{0, 1\}$, the attacker is given a set of n uniformly random I/O pairs $Q = \{(\mathbf{x}_i, f(\mathbf{x}_i))\}_{i=1}^{n}$ where $n \ll 2^m$.

The attacker tries to find a reconstructed circuit $\hat{f}$ that has the highest possible accuracy with delay no more than T. The accuracy acc is defined as the probability the output of the reconstructed oracle coincides with the original oracle in the entire input space, formally $acc = \frac{1}{2^m} \sum_{\mathbf{x} \in \mathbb{B}^m} \mathbb{1}(f(\mathbf{x}) = \hat{f}(\mathbf{x}))$, which can be estimated using a uniformly random test set $S \subseteq \mathbb{B}^m$ independent from the input patterns in Q, i.e., $acc \approx \frac{1}{|S|} \sum_{\mathbf{x} \in S} \mathbb{1}(f(\mathbf{x}) = \hat{f}(\mathbf{x}))$.

The above attack model assumes all I/O pairs are accessible but because of the limited time to access them, the attacker can only obtain $|S| \ll 2^m$ pairs. Also, the scope of this paper is limited to the cases when the function of the accessible circuit is not locked or otherwise altered.

III. Overview of Lorax

Fig. 1 shows the overall workflow of Lorax. Starting from the left panel, Lorax first evaluates ML models with different configurations, synthesizes each one to identify the delay of the corresponding oracle candidate, and then applies cross validation to identify the most accurate model among the ones which have a synthesized circuit with delay lower than the imposed time limit. If the accuracy is found to be satisfactory by the attacker, then the identified best ML model is reconstructed as the final oracle, as shown in the middle panel of Fig. 1. If the accuracy is unsatisfactory, in the right panel, Lorax deploys a special function matching step based on explanatory analysis of an ML model. This step checks the match with specific functions that are known to be hard to identify with ML only.

More specifically, the early evaluation phase (left panel) consists of three steps. First, three different "tree-based" ML models of nine different configurations are evaluated in parallel to learn the underlying logic implied by the available, tiny-sized I/O pairs. This step is also referred to as logic regression. (The details of logic regression is explained in Sec. IV-A.) Second, it reconstructs each ML model as an oracle candidate by synthesizing the learned logic into a compact circuit as an and-inverter graph (AIG). This is done by applying a unified flow of logic minimization, quantization of leaf nodes in the tree-based ML model, and performing a gate approximation technique. The delay of each reconstructed oracle candidate is then estimated from its AIG and the ones with delay higher than the imposed time limit are discarded. (This step is explained in Sec. IV-B.) Third, for the remaining reconstructed oracles, Lorax applies cross validation given the specific set of tiny-sized I/O patterns to identify the model/configuration of the highest accuracy. (Details of this step is explained in Sec. IV-C.)

Next, the middle panel acts as the core of Lorax which retrains the best-identified ML model (this time using the entire set of available I/O pairs for training, as opposed to the rolling 90%/10% splits for training/testing during the cross validation), and synthesizes it as a compact circuit.

Finally, in the right panel, Lorax applies special function matching by first checking for symmetric functions and if there is no match, it guides the function matching by exploiting **SH**apley **A**dditive ex**P**lanations (SHAP) which aims to improve the accuracy further by examining the importance of each input bit from the learned logic (explained in Sec. IV-D).

IV. Details

Here we explain more details about components of Lorax.

A. Logic Regression with Tree-Based ML Models

We perform logic regression using three tree-based supervised learning models: decision tree (DT) [9], XGBoost (XGB) [10], and random forest (RF) [11]. The advantages of tree-based models include the straightforward conversion from tree nodes to truth tables of the learned logic, as well as the availability of SHAP tree explainer [12], an efficient, specially-optimized algorithm to evaluate SHAP values that we can use for function matching, which will be detailed in Sec. IV-D.

We show the process of logic regression using an example. Fig. 2(a) shows a trained XGB or RF model, which consists of many decision trees (only showing the first two trees). Numbers in leaf nodes (rectangular boxes) indicate the possible outputs of a tree[1]. Fig. 2(b) shows the first tree in (a) after leaf node quantization. The resulting tree can then be transformed to a truth table. Fig. 2(c) shows the corresponding truth table in PLA format of the tree in (b). As can be seen, each quantized leaf node of the tree corresponds to a product term in the truth table, and the conversion is straightforward. We then use the logic minimization tool `espresso` [13] to simplify the truth table, and use the AIG synthesis tool `abc` [14] to generate the

[1]The DT model is similar except that it has a single tree with binary leaves.

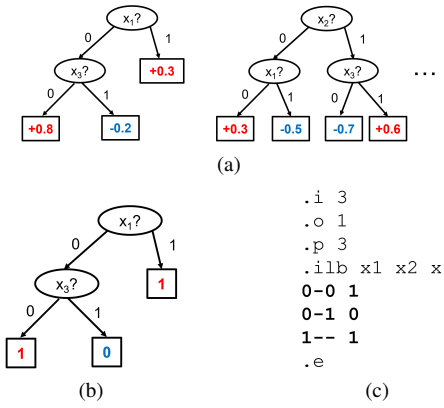


Fig. 2. (a) Illustration of underlying trees in a trained XGB or RF model. (b) The first tree in (a) after quantization. (c) The corresponding truth table in PLA format for the tree in (b). A dash for an input means "don't care".

reconstructed circuit. The above process works naturally for a single tree (among the trees in XGB or RF) and for DT (which does not require quantization because the leaves are already binary values). Next, we explain how the trees in XGB and RF can be combined/approximated to convert into a compact AIG.

B. Model Approximation and Conversion

The explained logic regression process was applicable to a single decision tree but by definition, both XGB and RF consist of multiple decision trees, where the final prediction is derived by the sum of leaf node values from each tree. Depending on the implementation of an RF or XGB model, the value in a leaf node can be any real number. Specifically in XGB, leaf values are expressed in logit odds, where a positive value means the function is more probable to output a 1 than a 0, and vise versa. This means for an XGB model with 100 trees (which is a nominal number in practice), we would need an adder to sum up 100 real numbers, and determine the prediction by the sign of the sum, if we wanted the exact prediction. However, this is neither feasible nor necessary—even if we somehow managed to do it, the resulting circuit would be extremely complex.

Therefore, we first quantize the leaf node values to one bit (i.e. binary) as shown in Fig. 2(b). This can reduce the adder to a majority gate, which simply compares the numbers of ones and zeros in the inputs. However, a majority gate with a large number (e.g. 100) of inputs is still very complex and prone to very large delay. To address it, we further approximate this majority gate as a hierarchical network of simpler majority gates. Fig. 3 shows an example that approximates a 125-input majority gate as a 3-level network of 5-input majority gates. Though demonstrated with XGB, the proposed quantization and gate approximation work for RF as well. Our experiments show that the proposed approximate majority gate works well.

C. Early Estimation of Accuracy With Cross Validation

Before the attacker performs actual reconstruction, they can use cross validation to get an early idea of accuracy and delay of the reconstructed oracle, and choose the best model and hyperparameter set. Specifically, we randomly divide the available I/O patterns into 10 equally-sized groups and use a 10-fold cross validation to compare configurations with different models, numbers of trees (for XGB and RF only) and tree depths. For XGB and RF, we go through the same quantization and majority gate approximation process as explained before.

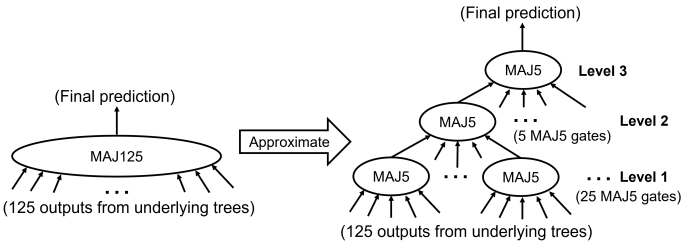


Fig. 3. A three-layer network of 5-input majority gates as an approximation of a large, complex 125-input majority gate.

Specifically, everything in cross validation is the same as in Sec. IV-A, except that we run the process 10 times. In each pass, we reserve one of the 10 groups (which traverses through the 10 groups in each pass) for evaluation and use the remaining 9 groups for ML training and reconstruction. The accuracy and delay of the reconstructed oracle are averaged over the 10 passes for each configuration. The configuration that has the best accuracy and satisfies the limit on delay is used for actual reconstruction.

D. SHAP-Guided Function Matching

By nature, tree-based ML models may not work well to predict the outputs of unseen inputs for functions formed by many XOR gates, which is often the case for arithmetic functions and some symmetric functions. To alleviate this issue, we propose SHAP-guided function matching that identifies important input bits and insert multiplexers (MUX) to reduce the problem into simpler cases.

SHAP tree explainer [12] is a recent advance in explanatory analysis of tree-based ML models (e.g. DT, XGB, RF), focusing on efficient and exact evaluation of Shapley values (a.k.a. SHAP values) of each bit in the prediction made by a tree-based ML model. SHAP decomposes the prediction output as $\hat{f}(\mathbf{x}_i) = \mathbb{E}[\hat{f}(x)] + \sum_j s(i,j)$, where $\mathbb{E}[\hat{f}(\mathbf{x})]$ is the average ML prediction based on the training set, and $s(i,j)$ is the SHAP value of input pattern i and bit j, which reveals the expected marginal contribution of bit j in predicting the output of pattern i. Therefore, it can serve as a measure of bit importance or the "weight" of the input bit in the predicted output.

To get this information, the attacker would first train an initial tree-based ML model using all available I/O patterns (as training samples), preferably XGB with a large number of trees (e.g., 216) and a large depth (e.g., 8) for the best possible accuracy. Note that for SHAP analysis, we use the model *without* quantization and approximation, nor do we consider the limit on delay. Although this initial model may not be very accurate, the attack can get the most important input bits by examining the mean absolute SHAP values[2] S_j of each input bit j across patterns i, i.e. $S_j = \mathbb{E}_{i \in \mathbb{B}^m}(|s(i,j)|)$, and taking the bit j^* with the highest S_j, i.e. $j^* = \arg\max_j S_j$.

Once the attacker identifies the the most important bit j^*, they can assume the function as the output of a 2-1 MUX, where the select input is $x[j^*]$, and data inputs are connected to two separate functions $\hat{f}_0(\mathbf{x})$ and $\hat{f}_1(\mathbf{x})$ that are trained with two subsets of the available I/O patterns where $x[j^*] = 0$

[2]We elaborate the process with an example in our experiments.

and $x[j^*] = 1$, respectively. In case of two equally important bits are identified, a 4-1 MUX with two select inputs can be assumed similarly, and the available I/O patterns would be divided into four subsets. It works similarly for more than two important bits. This process can be applied recursively as long as each subset includes enough number of patterns. In practice, however, with a limited number (e.g. 6400) of I/O patterns, a limited number (e.g. 6) of important bits can be identified before the subsets become too small to train accurate functions.

In case that the attacker achieves a high accuracy (e.g. $> 99\%$) for some function, they can generalize the learned function into a function *template* that describes the functionality. An example template is "XOR of the highest bit and the middle bit", more precisely, $x[n-1]$ XOR $x[n/2-1]$ (zero-based indices) for an n-input function. These templates can be used to match against the available I/O patterns for validation, whose underlying function may be otherwise difficult to learn.

V. Experimental Results

We use 100 functions and corresponding data sets from the IWLS 2020 programming contest benchmark [15]. We divide the functions into function groups with similar functionalities and/or sources, whose profile is shown in Table II. Each function in the benchmark comes with a training set, a validation set, and a test set. Regardless of the number of input bits in the underlying function, each data set includes 6400 I/O pairs that are randomly sampled from the input space. This training set is a tiny fraction of the input space in most cases (e.g. around 10^{-6} for a 32-input function) and is solely used in our experiments to reconstruct the oracle.

Recall, in our attack model, the goal of the attacker is to use the reconstructed oracle for accelerated simulation or emulation. Therefore, in our experiments, we report the delay and accuracy of the AIG circuit corresponding to the reconstructed oracle[3]. We evaluate the accuracy using the provided test set in [15] which is independent of the used training set. All ML models are implemented in C++ with XGBoost APIs. All experiments are carried out on a Linux desktop with an Intel 3.60 GHz CPU and 64 GB memory. Multi-threading is disabled.

A. Oracle Reconstruction with ML Only

Here, we show the results of oracle reconstruction with ML-only Lorax, which corresponds to the two left panels in Fig. 1. For each ML model, we consider a few configurations with the following naming convention. For the **X**GB and **R**F models, we use {model: X/R}{number of trees}-{max tree depth}. (We use "U" to refer to unlimited tree depth.) For DT models, we use DT-{max tree depth}. The cross validation process selects one ML configuration from DT-6, DT-8, DT-U, X125-4, X125-6, X125-8, R125-6, R125-8, and R125-U, after imposing an attacker-provided limit on the delay of the synthesized oracle circuit. Therefore, the selected model/configuration may vary across the functions and limits on the oracle delay.

Table I compares the accuracy and delay of the reconstructed oracle under different limits on delay. The results in each row are averaged over all functions in the corresponding function group. For each group and limit, we also report the most common model (MCM) selected by Lorax. A dash in MCM indicates no prevailing model in the function group. The bottom rows of the table report average, min, and max oracle reconstruction runtimes in Lorax, which includes cross validation, model training and synthesis.

[3]Since the functions used in our experiments have a single output, we report the delay to be proportional to length of the longest path of the AIG circuit.

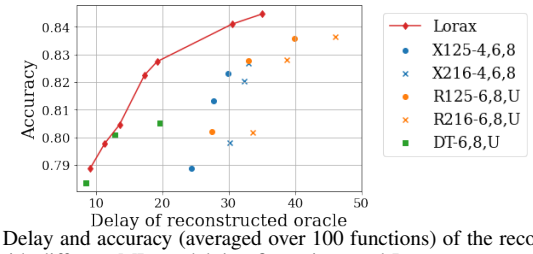


Fig. 4. Delay and accuracy (averaged over 100 functions) of the reconstructed oracle with different ML models/configurations and Lorax.

Looking at the overall average row in Table I, we observe that imposing the smallest limit yields $4\times$ lower oracle delay with 5% degradation in accuracy (from 84% to 79%) and selecting DTs as MCM. This is compared to the case with the largest limit when XGB and often RF will be the selected MCM, yielding higher accuracy but significantly higher delay. This trend exists for each function group in the table.

The runtime to reconstruct the oracle increases when increasing the limit, as reported in the bottom rows of the table. We also report the runtime for cross validation solely, which is included in the runtime of Lorax. In general the runtime of Lorax is dominated by the cross validation process, which is expected, due to the iterative validation passes. However, the overall runtime overhead is still small (the longest runtime is within minutes). Since we disabled multithreading in our experiments, the reported runtimes include all configurations and passes in cross validation performed *in series*. Moreover, these runtimes should be viewed considering the attacker reconstructs the oracle once with the goal to deploy it frequently to simulate or emulate the behavior of the circuit.

To illustrate the effectiveness of Lorax, next, in Fig. 4, we show comparison in delay and accuracy with single ML models/configurations. Each point is averaged over all 100 functions in the benchmark. (The numbers for each single model/configuration is when it is exclusively applied to all the 100 functions.) From the figure we can see that Lorax is *always* better than any single ML configuration with a tradeoff curve above all the points of single configurations.

B. Oracle Reconstruction with SHAP-Guided Matching

We can see from Table I that some functions are difficult to learn by Lorax when the training sample size is small, i.e., arithmetic functions in `ex00-08 even`, `ex20-28 even`, and `ex40-49`, as well as symmetric functions `ex74-79`. In such a case, the attacker can still use Lorax to get an estimated accuracy early in the cross validation process. In case this estimated accuracy is not satisfactory, the attacker can then apply our proposed function matching techniques to check for existence of these special functions in attempt to enhance the accuracy. More specifically, we check for symmetric functions by comparing the number of ones in the input vector and the output bit, and implemented them by adding a side circuit that counts the number of ones in the input bits, and a DT that learns

TABLE I

COMPARISON OF DELAY (D) AND ACCURACY (A) OF RECONSTRUCTED ORACLES, AND THE MOST COMMON ML MODEL (MCM) SELECTED BY LORAX IN EACH FUNCTION GROUP WHEN THE ATTACKER IMPOSES DIFFERENT LIMITS ON DELAY.

Limit on delay →	10			15			20			25			30			40			Unlimited			Fn Match	
Fn group (size) ↓	D	A	MCM	D	A	MCM	D	A	MCM	D	A	MCM	D	A	MCM	D	A	MCM	D	A	MCM	D	A
ex00-08 even (5)	10	52	DT-6	10	52	DT-6	14	53	DT-8	21	53	DT-U	26	55	X125-4	32	60	X125-8	39	61	—	53	100
ex01-09 odd (5)	8	97	DT-8	12	98	DT-U	12	98	DT-U	12	98	DT-U	12	98	DT-U	32	98	R125-U	32	98	R125-U	51	100
ex10-14 (5)	10	82	DT-6	14	85	DT-8	16	85	DT-U	19	86	DT-U	19	86	DT-U	31	87	R125-8	41	88	R125-U, R216-U	—	—
ex15-19 (5)	9	88	DT-6	13	90	DT-8	15	90	DT-8	15	90	DT-8	15	90	DT-8	36	91	R125-U	40	91	R216-U	—	—
ex20-28 even (5)	10	50	DT-6	10	50	DT-6	14	50	DT-8	18	51	DT-8	18	51	DT-8	31	51	—	33	52	—	—	—
ex21-29 odd (5)	10	60	DT-6	11	62	DT-6	17	65	DT-8	23	86	X125-4	23	85	X125-4	27	91	X125-4	34	92	R216-U	2	100
ex30-39 (10)	8	97	DT-8	12	98	DT-U	12	98	DT-U	12	98	DT-U	14	98	DT-U	35	98	R216-U	35	98	R216-U	29	100
ex40-49 (10)	11	57	DT-6	11	58	DT-6	14	60	DT-8	19	61	DT-U	19	61	DT-U	32	61	R125-6	35	62	R125-6/U	—	—
ex50-59 (10)	8	89	DT-6	10	89	DT-8	10	90	DT-8	11	90	DT-8	16	90	DT-6	27	91	R125-8	31	91	R216-U	—	—
ex60-73 (14)	7	95	DT-6	10	96	DT-8	12	97	DT-U	13	97	DT-U	15	97	DT-U	25	97	R125-U	27	97	R216-U	—	—
ex74-79 (6)	8	71	DT-6	11	71	DT-6/8	16	72	DT-U	17	74	DT-U	17	74	DT-U	30	80	X125-8	30	80	X125-8	20	100
ex80-89 (10)	11	91	DT-6	14	92	DT-8	17	93	DT-U	24	94	X125-4	27	96	X125-6	29	97	X125-8	30	97	X216-8	—	—
ex90-99 (10)	10	66	DT-6	11	66	DT-6	11	66	DT-6	24	69	X125-4	28	71	R125-6	34	72	R125-8	55	72	R216-U	—	—
Overall average (100)	9	79	DT-6	11	80	DT-6	14	80	DT-8/U	17	82	DT-U	19	83	DT-U	30	84	R125-8	35	84	R216-U	N/A	
Avg. runtime (s)	2.6			4.1			22.7			70.0			120.2			311.2			397.7			N/A	
– cross validation (s)	2.5			4.0			22.6			69.4			119.1			308.4			392.1				
Min. runtime (s)	1.2			2.2			7.9			15.5			21.2			52.3			52.3				
Max. runtime (s)	6.9			8.3			79.7			289.2			820.1			1130.5			1590.0				

TABLE II

PROFILE OF FUNCTION GROUPS IN IWLS 2020 CONTEST BENCHMARK

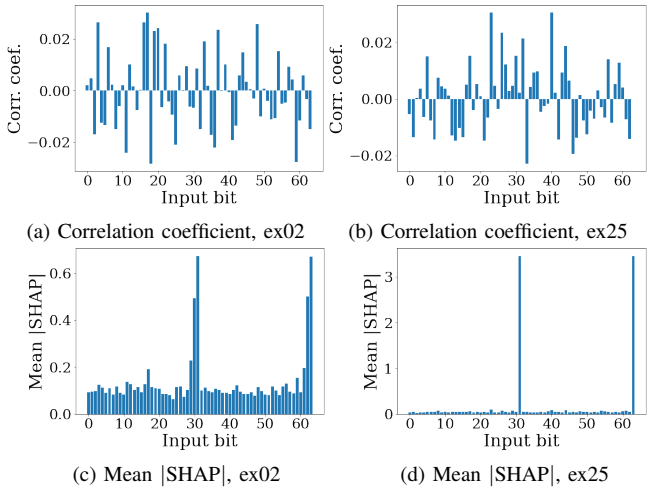
Function group	Group size	Description	# input bits per fn. Min	Max	Avg
ex00-08 even	5	2nd MSB of sum of two integers	32	512	198
ex01-09 odd	5	MSB of sum of two integers	32	512	198
ex10-14	5	LSB of quotient of integer division	32	512	198
ex15-19	5	LSB of remainder of integer division	32	512	198
ex20-28 even	5	Middle bit of product of two integers	16	256	99
ex21-29 odd	5	MSB of product of two integers	16	256	99
ex30-39	10	Comparator of two signed integers	20	200	110
ex40-49	10	LSB of square root	10	256	75
ex50-59	10	Selected outputs of PicoJava design	19	394	84
ex60-73	14	Selected outputs of MCNC designs	16	52	34
ex74-79	6	Selected symmetry functions	16	16	16
ex80-89	10	Binary classifications of MNIST	196	196	196
ex90-99	10	Binary classifications of CIFAR-10	768	768	768
Overall	100		10	768	179

the relationship between the count and the original output. This helps identify 6 symmetric functions `ex74-79`.

For arithmetic functions, we use SHAP-guided analysis to check for patterns in the importance of input bits for adders, comparators, outputs of XOR or MUX, as detailed in Sec. IV-D. Specifically, SHAP value is quite effective to measure the bit importance. We demonstrate this point using an example shown in Fig. 5. The figure compares the correlation coefficients and the mean absolute SHAP values of each input bit with respect to the output bit, for two functions `ex02` and `ex25`, respectively. For each function, we can see two peaks in the distribution of mean absolute SHAP values in Fig. 5(c) and (d), corresponding to the two most important bits in the input vector. In contrast, correlation coefficients in Fig. 5(a) and (b) show small random noises, which do not give any apparent hint of bit importance.

Once the most important bits are identified, the attacker may use a MUX with these important bits as the select bits, divide the training set according to the values of these select bits and re-train models in different modes.

Take `ex02` as an example. Since Fig. 5(c) shows that bits 31 and 63 (denoted $x[31]$ and $x[63]$) are two most important bits, we divide the training set into four subsets with $x[63]x[31] = 00, 01, 10, 11$, respectively, train four separate ML models and generate AIGs, combine them with a 4-1 MUX with $x[63]$ and $x[31]$ being select inputs. In the example of `ex02`, we



(a) Correlation coefficient, ex02 (b) Correlation coefficient, ex25

(c) Mean |SHAP|, ex02 (d) Mean |SHAP|, ex25

Fig. 5. (a)-(b) Correlation coefficients and (c)-(d) mean absolute SHAP values (based on 10k random patterns) of each input bit with respect to the output bit in functions ex02 and ex25, respectively. SHAP-guided analysis allows identifying the important bits while correlation coefficient fails to do so.

get higher accuracy from all four separated models than the original model. In function group `ex21-ex29 odd`, we can even achieve near 100% accuracy from all separated models.

With these function matching techniques, we are able to identify 31 out of 100 functions in the benchmark, on which we can build custom AIGs of the identified functions and achieve close to 100% accuracy. We report the delay and accuracy for these functions in the last two columns of Table I and mark those for other functions with dashes. Note that `ex00-09` have large delay because we implemented the exact function as identified. To reconstruct a faster oracle, input bits of lower importance can be ignored for a negligible error rate. Finally, we do not report runtime for oracle reconstruction using function matching because it is mostly a process performed by the attacker but guided by the SHAP analysis of Lorax.

C. Impact of Quantization and Approximate Majority Gate

Recall, Lorax utilizes approximation techniques including leaf node quantization and approximated majority gate (approx-MAJ) to synthesize the oracle as a compact AIG with low delay.

TABLE III
ACCURACY OF XGB AND RF WITH/WITHOUT QUANTIZATION (Q) AND APPROXIMATED MAJORITY GATE (M). AVERAGE DELAY IS IN THE BOTTOM ROW.

Function group (group size)	X125-4			X125-6			X125-8			X216-4			X216-6			X216-8			R125-6			R125-8			R125-U			R216-6			R216-8			R216-U		
	∅	Q	QM	∅	Q	QM	∅	Q	QM	∅	Q	QM	∅	Q	QM	∅	Q	QM	∅	Q	QM	∅	Q	QM	∅	Q	QM	∅	Q	QM	∅	Q	QM	∅	Q	QM
ex00-08 even (5)	65	56	55	68	61	59	66	62	60	68	58	55	68	61	58	67	62	60	58	54	54	61	59	58	62	60	59	58	54	54	61	59	58	62	60	59
ex01-09 odd (5)	99	96	94	99	97	96	99	97	96	99	95	95	99	97	96	99	97	97	95	94	94	97	97	97	98	98	98	95	94	94	97	97	97	98	98	98
ex10-14 (5)	83	77	74	86	83	80	87	85	83	83	77	78	86	83	83	87	84	85	84	83	83	87	87	87	88	88	88	84	83	83	87	87	87	88	88	88
ex15-19 (5)	86	83	80	89	87	84	90	87	85	87	82	84	89	85	87	90	86	88	88	88	80	90	90	90	91	91	91	88	88	88	90	90	90	91	91	91
ex20-28 even (5)	51	51	50	51	51	51	51	51	51	50	50	50	51	51	51	51	51	52	50	51	50	51	51	51	52	51	51	50	50	50	51	51	51	51	51	51
ex21-29 odd (5)	98	88	85	99	89	85	100	92	88	99	80	78	100	84	82	100	81	80	90	74	73	92	86	83	92	88	86	90	75	74	93	87	84	93	90	86
ex30-39 (10)	99	96	93	99	97	96	99	98	97	99	95	95	99	97	97	99	97	97	95	94	94	97	97	97	98	98	98	95	94	94	97	97	97	98	98	98
ex40-49 (10)	58	57	56	60	59	58	62	60	60	58	57	56	61	59	59	62	60	60	58	58	58	61	61	61	62	62	62	59	58	58	61	61	61	62	62	62
ex50-59 (10)	90	87	85	90	88	85	91	89	87	90	86	87	90	86	88	91	86	89	89	89	89	90	90	90	91	91	91	89	89	89	90	90	90	91	91	91
ex60-73 (14)	97	94	93	98	96	95	98	96	95	97	91	94	98	94	96	98	94	97	96	95	95	97	97	97	98	97	97	96	95	95	97	97	97	97	97	97
ex74-79 (6)	83	77	74	82	79	78	81	80	79	84	74	76	83	79	78	82	79	79	72	72	72	74	74	74	74	74	74	73	72	72	74	73	73	74	74	74
ex80-89 (10)	98	96	95	98	98	97	98	98	97	98	96	95	98	98	97	98	98	97	94	93	93	95	95	95	96	96	96	94	93	93	95	95	95	96	96	96
ex90-99 (10)	74	71	69	74	72	70	74	73	71	74	71	70	75	73	71	75	73	72	70	70	70	72	72	72	73	73	72	70	70	70	72	72	72	73	73	72
Avg. accuracy	85	81	79	85	83	81	86	84	82	85	80	80	86	82	82	86	82	83	82	80	80	84	83	83	84	84	84	82	80	80	84	83	83	84	84	84
Avg. delay	—	192	24	—	196	28	—	198	30	—	329	30	—	331	32	—	332	33	—	195	27	—	201	33	—	208	40	—	333	34	—	338	39	—	345	46

To measure the impact of quantization and approxMAJ for XGB and RF models, we train these models with the relevant configurations introduced earlier and perform quantization and approxMAJ, as given in Sec. IV-B.

We report in Table III the accuracy before quantization (denoted ∅), immediately after quantization (denoted Q), and after both quantization and approxMAJ (denoted QM). The reported accuracy is averaged within each function group. We also report the average delay across all functions in the bottom row of Table III. Delay before quantization (the column of ∅) is not reported because synthesizing the tree with floating point leaf nodes results in complex circuits, and the resulting circuits would be much more complex than that after quantization.

From Table III we make the following observations. For both XGB and RF, approxMAJ significantly reduces the complexity of the generated AIGs, which can be seen as significant reduction in delay (on average by a factor of 8 across all models). This is while the degradation of accuracy is fairly small compared to the original (2% and 0.5% respectively, averaged over all models and functions). We can conclude that both techniques are quite effective in creating an approximate circuit as a faster oracle without much degradation in accuracy.

D. Effect of Training Set Size

We study how training set size affects the accuracy and the delay of the reconstructed oracle with Lorax. We vary the training set size from 1600 to 12800 by taking the first 1600 and 3200 I/O patterns from the training set, the entire training set (of size 6400), and combining the training set and validation set for each function (to build the 12800-sample data set), respectively. As always, the accuracy is evaluated with a separated test set that is different than both the training and validation sets.

Fig. 6 shows tradeoff curves of average accuracy vs delay with different training set sizes. With more training data, we can observe a general increase in accuracy with the same delay. However, the increase is less obvious with smaller delays resulting from simpler configurations (e.g., DTs of smaller depths selected by Lorax). This is expected intuitively, as a single DT has limited capability to express complex functions. In other words, if a low oracle delay is desired, we can reconstruct with few training samples without loss of accuracy.

VI. CONCLUSIONS

We presented Lorax, a framework for oracle reconstruction

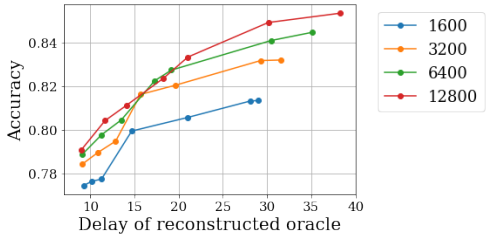


Fig. 6. Average delay vs accuracy of the reconstructed oracle by Lorax using different training set sizes (shown in legend).

with limited random I/O patterns. In our experiments, we showed with a tiny-sized training set, Lorax can approximate a range of functions from arithmetic circuits, industrial designs, and computer vision problems, with an average accuracy of 79%–84%, depending on the desired delay limit. We also showed function matching with explanatory analysis can boost the accuracy of some arithmetic functions to near 100%.

REFERENCES

[1] K. Shamsi *et al.*, "AppSAT: Approximately deobfuscating integrated circuits," in *HOST*, 2017, pp. 95–100.

[2] F. Yang *et al.*, "Stripped functionality logic locking with hamming distance-based restore unit (SFLL-hd) – unlocked," *IEEE Trans. on Information Forensics and Security*, vol. 14, no. 10, pp. 2778–2786, 2019.

[3] G. Di Crescenzo *et al.*, "Boolean circuit camouflage: Cryptographic models, limitations, provable results and a random oracle realization," in *ASHES*, 2017, pp. 7–16.

[4] S. Keshavarz *et al.*, "SAT-based reverse engineering of gate-level schematics using fault injection and probing," in *HOST*, 2018, pp. 215–220.

[5] U. Rührmair *et al.*, "Modeling attacks on physical unclonable functions," in *CCS*, 2010, pp. 237–249.

[6] F. Ganji *et al.*, "Pitfalls in machine learning-based adversary modeling for hardware systems," in *DATE*, 2020, pp. 514–519.

[7] K. Shamsi *et al.*, "On the impossibility of approximation-resilient circuit locking," in *HOST*, 2019, pp. 161–170.

[8] F. Tehranipoor *et al.*, "Deep rnn-oriented paradigm shift through bocanet: Broken obfuscated circuit attack," in *GLSVLSI*, 2019, pp. 335–338.

[9] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[10] T. Chen *et al.*, "XGBoost: A scalable tree boosting system," in *SIGKDD*, 2016, pp. 785–794.

[11] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[12] S. M. Lundberg *et al.*, "From local explanations to global understanding with explainable AI for trees," *Nature Machine Intelligence*, vol. 2, no. 1, pp. 56–67, 2020.

[13] R. L. Rudell *et al.*, "Multiple-valued minimization for PLA optimization," *IEEE Trans. on CAD*, vol. 6, no. 5, pp. 727–750, 1987.

[14] R. Brayton *et al.*, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*, 2010, pp. 24–40.

[15] A. Mishchenko *et al.*, "IWLS 2020 programming contest benchmark," 2020. [Online]. Available: https://github.com/iwls2020-lsml-contest