An Oracle-Less Machine-Learning Attack against Lookup-Table-based Logic Locking

Kaveh Shamsi
University of Texas at Dallas, ECE Department
Richardson, Texas, USA
kaveh.shamsi@utdallas.edu

Guangwei Zhao
University of Texas at Dallas, ECE Department
Richardson, Texas, USA
guangwei.zhao@utdallas.edu

ABSTRACT

Replacing cuts in a circuit with configurable lookup-tables (LUTs) that are securely programmed post-fabrication is a logic locking technique that can be used to hide the complete design from an untrusted foundry. In this paper, we study the security of basic LUT-based locking against a set of oracle-less attacks, i.e. attacks that do not have access to a functional oracle of the original circuit. Specifically we perform cut graph/truth-table prediction using deep and graph neural networks with various data encoding strategies. Overall we observe that naive LUT-based locking with small cuts with 2 or 3 inputs may be vulnerable to oracle-less approximation whereas such attacks become less feasible for higher cut sizes. We open source our software for this attack.

CCS CONCEPTS

• **Security and privacy → Security in hardware**.

KEYWORDS

logic locking, circuit obfuscation, circuit deobfuscation, oracle-less

ACM Reference Format:
Kaveh Shamsi and Guangwei Zhao. 2022. An Oracle-Less Machine-Learning Attack against Lookup-Table-based Logic Locking. In *Proceedings of the Great Lakes Symposium on VLSI 2022 (GLSVLSI '22), June 6–8, 2022, Irvine, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3526241.3530377

1 INTRODUCTION

Logic locking is a technique that can help hide the full design of an integrated circuit (IC) from untrusted foundries or end-users. Locking is done by adding extra "key" inputs into the circuit design such the IC may not function correctly absent a post-fabrication configuration of these key inputs to a secret value.

Locking can be done by adding key-controlled XOR/XNOR-AND/OR-MUX logic [9] into the circuit in what we deem "additive" locking. "Reductive" logic locking conversely is based on replacing native parts of the original circuit with key-dependent logic. A prominent example here is Lookup-Table (LUT)-based locking in which a k-input cone/cut in the circuit is replaced with a k-input key-controlled lookup-table with 2^k key bits (see Fig. 1). This makes it such that from the attacker's perspective the LUT functionality can be any of the 2^{2^k} different k-input Boolean functions.

Several oracle-less (OL) attacks against locking have been proposed in recent years. These are attacks that use only the structure/functionality of the locked semi-programmable circuit to learn the correct key, as opposed to an oracle-guided (OG) attack which can use correct input-output patterns collected from an oracle of the original circuit. SAIL [5] was first to propose analyzing the surrounding logic (neighborhood/locality) of a key-dependent ambiguous part of the locked circuit using machine-learning techniques to infer the original inserted gate type in XOR/XNOR locking. This was later on extended in OMLA [4] to use graph neural networks (GNNs) to process the key-logic neighborhood. Recently in [3] GNN-based link-prediction was used to attack MUX-based interconnect obfuscation with great success. SWEEP/SCOPE [2] are OL attacks that find correct keys by using the fact that loading the correct key into the locked circuit is not expected to lead to unreasonable area/delay/power reduction.

To the best of our knowledge, reductive LUT-based locking has not been extensively studied in the context of machine-learning oracle-less attacks. In this paper we focus on this problem and deliver the following:

- We propose CUT-SAIL, a SAIL-based approach to inferring the functionality/structure of missing k-cuts from their surrounding logic using machine-learning.
- We explore several techniques for encoding the missing k-cut, including using its truth-table, its adjacency matrix/list, or using canonical graph representations. We develop a self-referencing training and prediction scheme as well.
- We perform the above machine-learning attacks on a set of ISCAS benchmarks and report the results. Our results demonstrate that LUT-based locking with small LUTs (2, or 3 inputs) may be highly susceptible to OL approximation, whereas for larger LUTs the prediction accuracy drops dramatically. We release the source code for our analysis [1].

The paper is organized as follows: Section 2 covers preliminaries, Section 3 presents the proposed methodology. Section 4 presents experimental results. Section 5 concludes the paper.

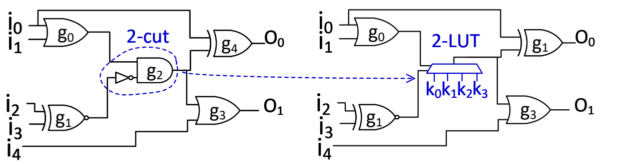

Figure 1: LUT-based locking. A k-cut is replaced with a k-LUT, which can be realized as a mux with 2^k configuration bits and k select lines.

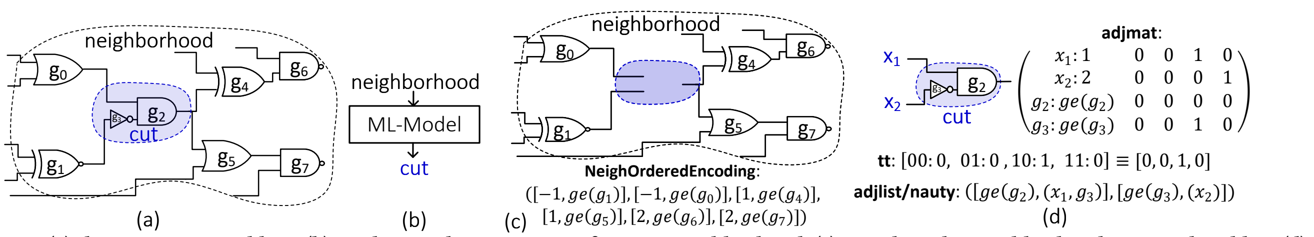

Figure 2: (a) the CUT-SAIL problem, (b) predicting the missing cut from its neighborhood, (c) encoding the neighborhood as an ordered list, (d) encoding the cut using its truth-table, or its adjacency matrix/list.

2 PRELIMINARIES

Formally a circuit locking scheme $c\mathcal{L}$ for a family of circuits C_o is an algorithm that converts an original circuit $c_o \in C_o : I \rightarrow O$ where I/O are the n/m-bit input/output space respectively, to an obfuscated/locked circuit $c_e : K \times I \rightarrow O$, where a polynomial number of key inputs l have been added to the circuit, and there must exist a correct key k_* such that $\forall x \in I\ c_e(k_*, x) = c_o(x)$. $C_e = \{c_e(k, x) | k \in K\}$ is the possible circuit space.

An oracle-guided attacker here in addition to access to the netlist for c_e (obvious), can query c_o on chosen points and obtain $y_i = c_o(x_i)$. An oracle-less (OL) attacker is tasked with the much more difficult problem of recovering the original functionality absent oracle queries. This makes oracle-less attacks approximate in nature. It is difficult if not impossible to achieve key correctness guarantees in an oracle-less setting.

3 CUT-SAIL: MISSING-CUT-INFERENCE

SAIL [5] initially targeted random XOR/XNOR insertion. In traditional XOR/XNOR locking l wires like w are randomly selected, opened to a left wire w_l and a right wire w_r. Then an X(N)OR gate is inserted w_r = X(N)OR(k_i, w_l). The correct key bit is the value that makes $w_r = w_l = w$ as it was in the original circuit. An XOR/XNOR hence leads to the correct key 0/1 respectively. This means that the type of the inserted gate readily reveals the correct key bit. Therefore, XOR/XNOR insertion must be followed by post-insertion mixing/resynthesis. During resynthesis and technology mapping, the type of gates can change, and the inverter operator at the output of an XNOR can get pushed around onto other wires. This can dilute the direct association between the correct key and the key inputs' neighborhood.

SAIL is based on the following flow: the post-resynthesis locality/neighborhood of a key input is taken, and a machine-learning (ML) model is used to try to predict the pre-resynthesis neighborhood. From the pre-resynthesis neighborhood, the value of the correct key bit is then obvious. The ML model can be trained on thousands of examples of insertion+resynthesis using a given synthesis flow and cell library. It was later shown that one can simply directly predict the correct key bit (as a Boolean ML problem) from the post-resynthesis neighborhood [4]. We employ a similar approach in this paper but rather than predicting pre-resynthesis circuit structures, or XOR/XNOR key-bits, we aim to infer a missing k-cut from its surrounding logic. Formally, we are given a k-cut $c_i(x_1, ..., x_k)$ embedded in a potentially-locked larger circuit c_e. The task is to predict the functionality of c_i given c_e. Per Fig. 2a we aim to employ an ML model F that takes in some representation of the circuit minus the missing cut $c_e \setminus c_i$, and predicts some representation of the functionality of the cut c_i, i.e. $Rep_c(c_i) \leftarrow F(Rep_n(c_e \setminus c_i))$.

3.1 Neighborhood Encoding

The first step is to encode and process the cut's neighborhood. An l-neighborhood (denoted as L_l) of a gate g_i in the circuit is defined as gates that have a distance of $\leq l$ to g_i. In SAIL the l-neighborhood of the key-controlled XOR/XNOR gate is used as the input to the ML model. In our case, the l-neighborhood of a k-lut can be obtained by a breadth-first-search (BFS) traversal of depth l starting from the union of the LUT's inputs and root (output) as the source.

The l-neighborhood of a key-controlled LUT is itself a graph. The ML model has to hence take in this graph in some format and operate on it to use it for prediction. Passing graphs to machine-learning models has been a topic of extensive research for quite some time with various frameworks being proposed. We discuss two main approaches that are applicable herein:

Ordered List Encoding: Conventional neural networks (NNs) are geared towards matrix data with fixed sizes. This makes it difficult to directly pass an unstructured l-neighborhood graph to an NN. One can convert a graph to its dense adjacency matrix and pass it to a neural network model. This however quadratically inflates the data size, since a non-sparse matrix representation of a graph with N nodes is of size $O(N^2)$. Converting the graph to its adjacency list reduces this size complexity to $O(|V| * |E|)$ (for a graph $G = (V, E)$). The order in which the nodes in the graph are written down in an adjacency list/matrix can vary in a topology-independent manner. This topology-divorced repositioning of input features can be challenging for a conventional neural network.

If the order in the adjacency list is tied to the topology of the graph somehow, this can alleviate the issue here. In an ordered list encoding of the kind we use in our work, the l-neighborhood graph of a cut is converted to a list with nodes near the cut being written down before distant nodes.

For the gates $g_i \in L_l$, the ordered-list neighborhood encoding creates a vector OrdNList(L_l) by first appending all the gates at distance 1 from the cut, then those at distance 2, and so on. A negative sign can be added to the encoding of nodes that are in the fanin of the cut to differentiate them from those in its fanout. In this encoding approach, the explicit connection between the gates in L_l is lost. Only their closeness to one another is captured within the order of the list.

For each gate $g_i \in G_i$, its gate-feature-vector is included in the neighborhood ordered list encoding. The gate feature that we use, captured by ge in Fig. 2, consists of an integer denoting the functionality of the gate by its unique index in the gate library. This is then followed by the number of inputs to the gate. Neighborhoods that do not reach the l gate count are padded with zeros to the maximum size l in the dataset. The above L_l is in the form of a

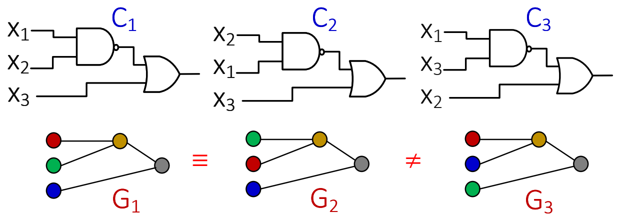

Figure 3: Despite C_1 and C_2 having different adjacency lists/matrices, G_1 and G_2 are colored isomorphic graphs. Canonical labeling can be used to store them as the same vector. The color of the nodes helps preserve the correct order of the cut inputs.

vector and can hence be passed to a neural network or other vector-receiving ML models.

Graph-Neural-Network: In the above ordered-list encoding the explicit connections between the nodes in the graph of L_l are lost. The closeness of two nodes is carried on the order of the list. While there are ways to increase topological information in the neighborhood encoding[1], a modern alternative to the above problem is using graph-neural-networks (GNNs). These are ML models that operate directly on graphs and hence can capture the topology of a graph natively. A common such model is a graph-convolutional-network (GCN) [6]. A GCN layer performs the following operation on node encodings $H^{(l)}$ in a graph to update them:

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}} A \hat{D}^{-\frac{1}{2}} H^{(l)} W^{(l)})$$

With $\tilde{A} = A + I_n$ being the adjacency matrix of the graph added with I_n denoting self-connections for each node, $\tilde{D}$ the degree matrix of the graph, W the layer's weights, and σ the activation function of the layer. In each application of the layer, the previous node encodings are updated via combining them with weights and encodings from their neighbors through the matrix A. Since the operation corresponds to passing data around in the graph, the explicit full rank adjacency matrix A is not needed. l GCN layers correspond to each node sensing information from l hops away in the graph.

With a GCN we can pass the neighborhood graph L_l directly to the graph ML model. OMLA [4] applied a similar approach to XOR/XNOR OL deobfuscation and observed superior results. Our results confirm that the GCN is indeed superior to the ordered list neighborhood encoding in the cut-prediction task as well.

3.2 Cut Encoding

The ordered list or GCN can both process the OL attack problem input, L_l. Now the question is how to capture/encode the missing cut's functionality, i.e. the output of the ML model. We explored several approaches in our work that we discuss here:

Truth-Table: To our surprise, we discovered that the best performing encoding for the cut in our experimental setup appeared to simply be its truth-table. Here per Fig. 2d given a cut $c_i(x_1, ..., x_k)$, we simulate the cut on all 2^k possible input patterns and record its output bit in a 2^k vector. This encoding grows exponentially with the size of the cut inputs k^2.

[1][10] for instance reports beating graph-neural-networks by passing sorted encoding lists along with other list-based topological information to a transformer.

[2]note that this is not an entirely symmetric challenge for the attacker. i.e. a defender trying to incur exponential encoding size cost on the attacker by raising k, may cost himself exponentially in terms of area overhead if using 2^k-entry fully-programmable LUTs. Semi-programmable structures may break this tie but are outside the scope of this paper [7]

Adjacency Matrix: We can encode the cut using its adjacency matrix maintaining reversibility. Per Fig. 2d, for a cut with s nodes (k inputs plus gates) an $(s) \times (s+1)$ matrix is built. The 0th column stores the features for each node, i.e. the input index for input nodes, and the gate-feature for gate nodes. The 1-to-$s+1$ columns are set to all zeros except for 1s in each $(i, j+1)$ locations where node i has an outgoing connection to node j. This produces an encoding of size $O(s^2)$ which is manageable for the k-cuts ($k < 8$) explored in our experimentation. Smaller matrices are padded with zeros to the maximum adjacency matrix size in the dataset.

Adjacency List: Another way to encode the cut reversibly is using its adjacency list per Fig. 2d. Here starting from the root node in the cut, its gate feature is added to the encoding vector, followed by listing its fanins. The gate feature includes the number of fanins, which signals to a reverse parser where the fanin description ends and the next node starts. Smaller cuts are padded with zeros to the maximum adjacency list size in the dataset.

Canonical Graph Encoding: Two graphs G_1 and G_2 are isomorphic if it is possible to relabel the nodes of G_1 to obtain G_2. Graph isomorphism (GI) is an NP problem with no known polynomial-time solver. However, there are software libraries that can compute GI efficiently for reasonably sized graphs. The `nauty` [8] library is one of such tools. In addition to solving GI, `nauty` has functionality for "canonical labeling" of a graph. If two graphs are isomorphic, DFS/BFS on the two graphs will produce the same sequence of nodes if nodes with higher/lower canonical labels are explored first in the DFS/BFS tree. Hence, performing the adjacency list encoding while respecting the ordering set by the canonical labeling produced by `nauty` can create an encoding that unlike the adjacency list/matrix is preserved under a set of node permutations. Per Fig. 3 the first two cuts C_1 and C_2 have different adjacency list/matrix encodings, yet their canonical list encoding is the same.

An important functionality in `nauty` is the ability to color the nodes in the graph. Graph isomorphism between G_1 and G_2 with colored nodes requires that the relabeling-map that takes G_1 to G_2 to be color-preserving. We color the input nodes in the cut with k different colors and the gates with g different colors for each gate functionality. This can be seen in Fig. 3.

3.3 Constructing and Training the Model

Model Output Layer. The output layer of the network is hence heavily dependent on the type of cut encoding. We use a dense layer followed by a sigmoid activation function for predicting truth-tables bit-vectors. For adjacency-matrix/list/nauty encodings the output is a sequence of numbers. We one-hot encode these numbers and use a dense+softmax layer to produce each one.

Categorizing Cut Encodings. A technique that showed significant accuracy improvement was *output categorization*. Here we take a hash of the cut encoding (bit-vector for a truth-table or number sequence in for adjacency-list/matrix encodings) and add it to a hash-set. The hash-set will map each possible output/cut encoding to a single index in the set. Denote the size of this set as $|CE|$. Now rather than having the model predict individual fields in the cut encoding vector, we train the model to only predict an index into

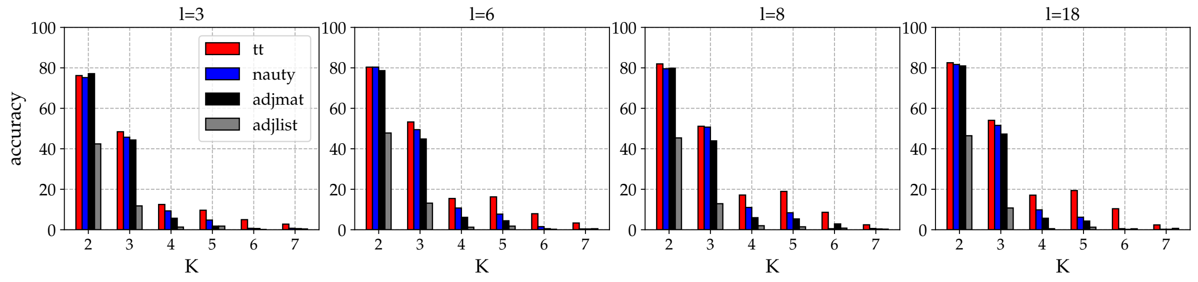

Figure 4: GCN LUT-prediction accuracy (percentage of perfect matches) on pristine combined benchmark set for different values of l and k and different cut encodings with output categorization activated.

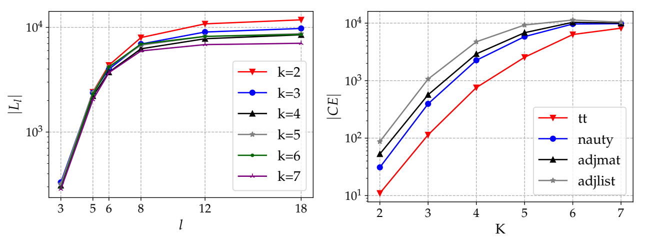

Figure 5: (a) number of unique l-neighborhoods in dataset per increasing l. (b) number of unique k-cut encodings per increasing k and different cut encoding schemes.

$|CE|$ via a softmax. If $|CE|$ is prohibitively large, this can create an equally large NN/GCN model. However, for small values of k (less than 8 in our experiments) this appeared manageable (less than 10K).

Training via Cut-Enumeration and Self-Referencing. We can mount the above CUT-SAIL OL attack under two different threat models. *Model1*: the attacker trains the model on a large set of unobfuscated/pristine benchmark circuits that are synthesized using the same library and synthesis flow as the locked designs. *Model2*: the attacker can use the obfuscated circuit itself as a training data generator. In this "self-referencing" approach (per the terminology in [4]) the attacker does not need access to a replica of the defender's synthesis flow+library.

In both cases, training data requires generating a set of neighborhoods $\{L_l\}$ along with their *correct* k-cut functionality encoding. This cannot be done on cuts in the obfuscated circuit that are missing, i.e. have been mapped entirely to a key-controlled k-LUT. However, it can be done on every *present* k-cut in the circuit, by simply *imagining* that it was missing and using its neighborhood to predict it. We can generate a large amount of such data from obfuscated or pristine circuits via cut-enumeration. The cut-enumeration procedure works by recursively finding new cuts rooted at a node by replacing cuts in the current cut set, with cuts with more inputs by moving the cut's boundary.

Poisoned Data. As the circuit is obfuscated with more and more LUTs, the self-referenced training data will have more and more missing functionalities in the set of l-neighborhoods. During training, we deal with this by first trying to avoid other k-LUTs in the vicinity of the target cut as much as possible. If this is not possible for some cuts, we set the missing functions in the l-neighborhood to the most common k-cut in the rest of the circuit in a greedy manner. More intelligent approaches such as A-star search are possible but not explored here.

		l=3		l=6		l=8		l=18	
k	catCE	NN	GCN	NN	GCN	NN	GCN	NN	GCN
2	-	73	74.4	74.1	78.9	74.8	79.6	74.9	78.3
	✓	74.9	76.2	76.6	80.4	78.1	82	78.2	82.5
3	-	32.7	40.1	35.2	38.8	37	37.7	36.8	39.1
	✓	42.9	48.4	48.6	53.3	50.8	51.1	50.1	54.1
4	-	0.965	0.851	2.24	0.738	1.62	1.65	1.53	1.59
	✓	10.9	12.5	16.2	15.4	20.6	17.1	20	17.1
5	-	0.892	1.78	4.24	0.964	4.29	4.14	4.99	1.16
	✓	8.48	9.64	18.4	16.2	25.6	18.9	26	19.4
6	-	0.768	0.791	1.54	1.02	1.33	0.605	1.51	0.931
	✓	5.63	4.98	7.8	7.91	10.3	8.61	27.2	10.4
7	-	0.664	0.387	0.609	0.996	0.387	0.277	0.692	0.498
	✓	3.21	2.77	6.5	3.32	6.72	2.43	5.84	2.38

Table 1: GCN vs NN 90/10-train/test split prediction accuracy over unobfuscated combined benchmark circuits with different l and k and truth-table encoding. catCE denotes w/output categorization.

4 EXPERIMENTS

We implemented the above CUT-SAIL framework in Python using `tensorflow`, `spektral`, and `pynauty`. Tests were run on an AMD Ryzen Threadripper 3990X with 128 logical cores and 256 GB of memory running Linux with no GPU acceleration. As benchmarks, we used a random selection of ISCAS combinational and sequential benchmarks as seen in Table 2 (42 original circuits and 608 locked circuits in total). The benchmarks were resynthesized using ABC with a cell library with basic primitive gates. Our script supports designs in the NanGate45nm library as well.

Combined Dataset (Model1). Fig. 5 shows the neighborhood and cut encoding count over all k-cuts enumerated for all 42 circuits in the unobfuscated benchmark set for different k and l (neighborhood depth) values. For $k \leq 7$ the number of different cuts remains below 10K allowing for manageable output categorization.

We use a graph neural network with 3 GCN layers with 32 channels before an output layer. For ordered-list neighbor encodings we used a neural network with 3 layers each with $\sqrt{n_{out} * n_{in}}$ hidden neurons before the output layer which is constructed following 3.3. Inferring cuts with different k requires different ML models in our current implementation. We train both NN and GCN models for 250 epochs with an Adam optimizer and a batch size of 50 with early stopping enabled. No training task on the combined dataset took more than an hour to finish.

Fig. 4 shows the GCN's LUT-prediction accuracy (number of perfectly recovered LUTs with 90/10-train/test split) over all k-cuts (limit of 20 per node) enumerated in the unobfuscated benchmark set. It can be seen that the truth-table cut encoding is the best performer achieving above 80% accuracy (translates to 90% > key-bit accuracy) on 2-LUT recovery followed by nauty and the adjacency

	stats		perc=5%					perc=10%					perc=15%					perc=20%				
bench	i/o	g	nl	k=2	k=3	k=4	k=5	nl	k=2	k=3	k=4	k=5	nl	k=2	k=3	k=4	k=5	nl	k=2	k=3	k=4	k=5
s298	17/20	101	6	50	56.2	45.8	43.8	11	68.2	55.7	48.9	46.9	16	62.5	58.6	39.2	44.4	21	71.4	72	40.4	39.6
s349	24/26	116	6	100	54.2	59.4	66.1	12	87.5	61.5	55.7	62.2	18	90.3	63.2	48.3	56.2	24	93.8	64.6	57.2	59.6
s386	13/13	132	7	67.9	60.7	57.1	56.2	14	69.6	65.2	56.7	46.7	20	72.5	64.4	45.9	53.3	27	76.9	67.1	54.6	43.4
s641	54/42	148	8	87.5	79.7	48.4	53.9	15	70	59.2	62.5	57	23	69.6	55.4	59	60.4	30	68.3	70.4	57.6	53.1
s713	54/42	149	8	50	60.9	39.1	64.5	15	78.3	61.7	54.2	58.5	23	76.1	63.6	57.7	46.2	30	71.7	75	53.7	63.1
s526	24/27	156	8	46.9	51.6	54.7	58.6	16	71.9	53.9	53.9	53.1	24	79.2	62	54.9	43.6	32	68	59.4	52.3	56.1
c432	36/7	198	10	82.5	73.8	61.3	60	20	66.2	60.6	55	64.1	30	85.8	72.1	61.7	67.1	40	73.1	66.2	55.9	61.9
s510	25/13	209	11	77.3	61.4	53.4	60.2	21	82.1	79.2	63.7	69.9	32	72.7	84	65.6	71.4	42	78.6	79.5	66.9	63.8
s499	23/44	222	12	54.2	90.6	64.8	74.2	23	21.7	82.1	65.6	73.9	34	52.2	82	65.6	73.9	45	50	82.8	56.2	73.9
s820	23/24	265	14	67.9	80.4	56.7	65.4	27	66.7	77.8	60.9	63.4	40	74.4	77.2	62	60.8	54	67.6	71.1	60.7	65.7
s991	84/36	297	15	90	79.2	70.4	57.3	30	78.3	70.4	72.1	64	45	74.4	65.3	69.9	67	60	75	74.4	72.2	71.8
c880	60/26	308	16	65.6	70.3	53.5	56.6	31	86.3	66.9	65.9	65	47	81.4	68.1	59	60.1	62	82.7	69.8	60.2	57.6
s953	45/52	373	19	64.5	74.3	63.8	74.3	38	65.1	72.7	73.7	65.8	56	60.7	75.7	65.7	65.8	75	63.3	84.8	70.8	69.8
s967	45/52	390	20	80	60	60.9	75.8	40	82.5	72.8	70.9	70.1	59	73.7	75.2	64	73	79	79.7	77.8	69.6	74.8
c1908	33/25	398	20	78.8	75	56.2	51.1	40	81.9	73.1	54.4	55.4	60	83.8	81.5	60.2	64.4	80	82.2	79.7	62.5	55.8
s1269	55/47	408	21	90.5	85.7	71.4	70.1	41	95.7	72.9	69.2	67.4	62	88.3	71	68	58.4	82	89.3	70.1	64.3	67.7
s1196	32/32	444	23	91.3	84.2	63.3	65.4	45	78.3	76.4	64.4	65.8	67	82.1	71.6	64.1	63.8	89	78.1	80.8	66.2	63.3
s1512	86/78	459	23	88	72.3	61.4	53.4	46	87	71.2	59.8	52.7	69	89.5	71.6	53.1	52.9	92	80.4	69.4	50.9	51.9
c1355	41/32	481	25	100	83	60.2	57.9	49	95.4	70.9	61.7	41.9	73	98.3	72.6	54.5	36.5	97	96.9	76.4	54.7	37.2
s1238	32/32	485	25	76	83	64	59.9	49	86.2	81.1	57.4	58.3	73	79.5	82.9	63.3	62.3	98	82.9	76	60.9	58.9
c499	41/32	487	25	100	74	72.8	63.2	49	99.5	73	56.1	42.7	74	95.9	77.2	59	40.5	98	94.6	76.8	52	40.3
s1423	91/79	513	26	65.4	55.3	50.5	54.2	52	67.8	57.2	53.4	53.8	77	75.3	57.6	51.5	53.8	103	72.6	58.7	48.7	53.9
s1488	14/25	575	29	86.2	63.4	62.1	60.9	58	88.4	67.7	58.9	64	87	90.2	67.2	58.8	63.8	116	86.9	68.1	63.5	60.1
c2670	157/64	628	32	77.3	71.9	54.1	59.2	63	88.1	69.6	60.6	62.1	95	82.4	63.8	59.6	64.1	126	82.9	65	56.7	62.9
c3540	50/22	938	47	82.4	69.9	61.4	58.6	94	88	70.3	59.8	60	141	86.5	72.6	59.2	62.6	188	85.4	75.7	59	61.1
s3384	226/209	1049	53	70.8	59.4	59.1	54.1	105	81.9	61.8	55.8	53.5	158	77.4	58.9	54	53.3	210	71	58	54.3	52.7
s3271	142/130	1106	56	92.4	60	57.1	60.5	111	86.7	62.6	59.7	63.6	166	86	62.1	56.3	62	222	93.6	65.7	53.4	57.9
s5378	214/213	1169	59	86	67.8	53.9	62.3	117	79.9	70.1	60.7	65.7	176	81.7	73.2	61.2	66.2	234	82.2	71.9	61.6	66
c5315	178/123	1264	64	89.5	85.4	66.2	65.8	127	84.3	76	67.1	63	190	86.4	77	66.4	63.3	253	87.4	84.7	65.8	62.6
s4863	153/88	1458	73	91.8	76.4	69.3	58.9	146	95.4	72.3	62.6	55.3	219	87.9	74.9	62.4	55.8	292	88.3	72.9	61.9	55.5
s9234	247/250	1504	76	87.5	70.6	62.3	60	151	81.3	65.5	58.1	61	226	85.5	71.8	57.3	60.3	301	82.4	68.3	56.5	62.1
c7552	206/107	1510	76	83.2	70.7	59.8	50.6	152	86.7	65.5	56.3	53.7	227	85.8	67.5	55.2	51.2	303	84.2	65	55.1	54.6
s6669	322/269	1854	93	86.8	66.5	62	55.9	186	78.1	69	58.4	56.8	279	84.3	68	59.3	56	371	83	68	61.7	56.1
c6288	32/32	1917	96	97.4	98.8	79.6	57.3	192	99	96.5	73.3	57.3	288	97.8	90.9	75.2	57.8	384	97.8	75.5	73.5	57.3
s13207	700/790	2600	131	74.4	71	62.2	53.9	261	76.6	64.2	52.6	52	391	71.2	69.9	45.5	51.8	521	73.4	69.4	48.6	53.8
s15850	611/684	3080	155	83.7	67.8	58.5	57	309	82.7	67.7	55.5	54.7	463	81.2	64.8	49.4	52.8	617	80.9	63.9	48.5	51.9
s35932	1763/2048	8614	431	94.7	80.9	77.3	62.4	862	91.6	83.1	66.1	72.7	1293	90.6	82.8	67.8	70.3	1723	87.9	82.6	72.4	65.4
s38417	1664/1742	9107	456	82.2	64.6	63.2	62.7	911	82	66.3	65	63.1	1367	80.7	64.6	55.4	61.8	1822	79.8	64.6	53.7	62.5
avg-acc				80	71.3	60.5	60.1		80.4	69.6	60.7	59.4		80.9	70.6	58.8	58.7		80.1	71.6	58.8	58.6

Table 2: Single circuit self-referenced CUT-SAIL accuracy. nl is the number of LUTs inserted in the circuit which is derived from perc × num_gates. k is the number of LUT inputs (k-cut).

matrix representation, while the non-canonical adjacency list performs the worst barely achieving 45% on the same task. Table 1 compares the GCN to the NN with various l and k on the same dataset with a truth-table encoding. The effect of output categorization on the performance can be seen as a near order of magnitude difference in higher k values. The GCN outperforms the NN slightly confirming the results in [4]. In some higher k tasks the NN seems to beat the GCN which we aim to explore in our future work.

Single Circuit Deobfuscation (Model2). Table 2 shows CUT-SAIL in a single-circuit self-referencing mode using the GCN with a truth-table encoding and $l = 20$. An average 80% key-bit prediction accuracy is observed across the benchmarks with 2-LUT insertion. No deobfuscation task took longer than 10 minutes. The point at which data poisoning becomes dominant is likely at higher locking percentages than 20%. Combining other locking schemes with LUT insertion can also hurt our current CUT-SAIL implementation, which a multi-task OL framework may be able to attack.

5 CONCLUSION

We presented CUT-SAIL, an OL ML-based attack that predicts missing k-cuts in a circuit. Our results demonstrate that insertion of 2 and 3-input LUTs for obfuscation without any post-processing can be vulnerable to such OL approximation with 70 − 80% accuracy reaffirming the point that using fewer larger LUTs may be superior to using many smaller ones [7].

REFERENCES

[1] [n. d.]. Code release. http://www.github.com/kavehshm/cutsail.

[2] Abdulrahman Alaql, Md Moshiur Rahman, and Swarup Bhunia. 2021. SCOPE: Synthesis-Based Constant Propagation Attack on Logic Locking. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 29, 8 (2021), 1529–1542.

[3] Lilas Alrahis, Satwik Patnaik, Muhammad Abdullah Hanif, Muhammad Shafique, and Ozgur Sinanoglu. 2021. UNTANGLE: Unlocking Routing and Logic Obfuscation Using Graph Neural Networks-based Link Prediction. *arXiv preprint arXiv:2111.07062* (2021).

[4] Lilas Alrahis, Satwik Patnaik, Muhammad Shafique, and Ozgur Sinanoglu. 2021. OMLA: An Oracle-less Machine Learning-based Attack on Logic Locking. *IEEE Transactions on Circuits and Systems II: Express Briefs* (2021).

[5] Prabuddha Chakraborty, Jonathan Cruz, Abdulrahman Alaql, and Swarup Bhunia. 2021. SAIL: Analyzing Structural Artifacts of Logic Locking Using Machine Learning. *IEEE Transactions on Information Forensics and Security* 16 (2021), 3828–3842.

[6] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[7] Gaurav Kolhe, Hadi Mardani Kamali, Miklesh Naicker, Tyler David Sheaves, Hamid Mahmoodi, PD Sai Manoj, Houman Homayoun, Setareh Rafatirad, and Avesta Sasan. 2019. Security and complexity analysis of lut-based obfuscation: From blueprint to reality. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[8] Brendan D McKay and Adolfo Piperno. 2014. Practical graph isomorphism, II. *Journal of symbolic computation* 60 (2014), 94–112.

[9] Muhammad Yasin and Ozgur Sinanoglu. 2017. Evolution of logic locking. In *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 1–6.

[10] Jiawei Zhang, Haopeng Zhang, Congying Xia, and Li Sun. 2020. Graph-bert: Only attention is needed for learning graph representations. *arXiv preprint arXiv:2001.05140* (2020).