OMLA: An Oracle-less Machine Learning-based Attack on Logic Locking

Lilas Alrahis, *Member, IEEE,* Satwik Patnaik, *Member, IEEE,* Muhammad Shafique, *Senior Member, IEEE,* and Ozgur Sinanoglu, *Senior Member, IEEE*

***Abstract*—Hardware-based attacks on the semiconductor supply chain are emerging due to the globalization of the design flow. Logic locking is a design-for-trust scheme that promises protection throughout the supply chain. While attacks have heavily relied on an oracle to break logic locking, machine learning (ML)-based attacks demonstrate the daunting possibility of breaking locking even without an oracle. Although very potent, current ML-based attacks recover only a subset of the transformations introduced by locking. We aim to address this shortcoming by developing an oracle-less graph neural network-based attack called OMLA, questioning once again the security of logic locking. Our experiments on ISCAS-85 and ITC-99 benchmarks demonstrate that OMLA achieves a key-prediction accuracy up to 97.22% and outperforms state-of-the-art SnapShot and SAIL attacks for all evaluated benchmarks.**

***Index Terms*—Logic Locking, Oracle-less Attack, Graph Neural Networks, Machine Learning**

I. Introduction

With the ever-increasing design complexity and integrated circuit manufacturing costs, outsourcing fabrication offer transcendent financial benefits for design companies. Unfortunately, sophisticated hardware-based attacks are emerging on the globalized supply chain causing various concerns, such as intellectual property (IP) theft. Several *design-for-trust* techniques can offer protection in particular design stages, in contrast to logic locking that promises protection throughout the supply chain. Logic locking hides the functionality of the design using key-controlled logic [1]. Without knowledge of the correct key, untrusted entities cannot access the functionality of the IP. An example of traditional logic locking (TLL) is presented in Fig. 1, where an XNOR key-gate is inserted and controlled by a newly added key-input.

The community has proposed various attacks on logic locking, considering two threat models as follows. (i) Oracle-guided model in which the attacker holds an activated chip in addition to the locked netlist [2]–[4]. (ii) Oracle-less model in which the attacker holds the locked netlist only and is aware of the underlying locking scheme [5]–[8]. We focus on the oracle-less attacks which are considered very potent as they can break logic locking using limited resources. Recently, machine learning (ML)-based approaches have been shown to be successful in breaking logic locking in a oracle-less setting. However, existing ML-based methods suffer from at least one of the following drawbacks (also summarized in Table I).

Rigid Locality Encoding. The rationale behind oracle-less ML-based attacks is that TLL introduces limited local

Lilas Alrahis, Muhammad Shafique and Ozgur Sinanoglu are with the Division of Engineering, New York University Abu Dhabi, AUH 129188 UAE.

Satwik Patnaik is with the Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX 77843 USA.

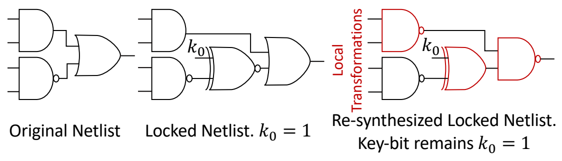

k_0
k_0
Local Transformations
Original Netlist
Locked Netlist. $k_0 = 1$
Re-synthesized Locked Netlist.
Key-bit remains $k_0 = 1$

Fig. 1. Example of X(N)OR logic locking that relies on synthesis to obfuscate the mapping between the type of key-gate and corresponding key-bit value.

TABLE I
Requirements of the ML-based oracle-less attacks on TLL

Attacks	Rigid Locality Encoding	Requires Pre-synthesis Subgraphs	Complex Learning Model
SAIL [5]	✓	✓	✓
SnapShot [6]	✓	✗	✗
OMLA	✗	✗	✗

changes surrounding the key-gates. ML models can learn the induced changes in local enclosing subgraphs around the key-gates and predict the corresponding key-bit values. *SAIL* [5] and *SnapShot* [6] attacks encode the subgraphs as matrix structures to be handled by traditional ML models. A netlist subgraph naturally lacks an ordered tensor representation, thus, the sequence by which gates are encoded affects the overall performance of the underlying attack model.

Pre-synthesis Localities. SAIL [5] targets locking solutions that rely on synthesis to obfuscate the mapping between key-gates and corresponding key-bit values (see Fig. 1). SAIL learns the optimization performed by the synthesis tools, reverts such changes, and recovers the original locked design. To train SAIL, multiple sets of pre-synthesis and post-synthesis encoded localities of different sizes (from 3 gates up to 10 gates) are required. SAIL also requires the synthesis procedure followed on the locked design to be known in advance.

Complex Learning Model. SAIL utilizes two ML models as follows. (i) A change prediction model predicts if a post-synthesis subgraph changed due to synthesis. (ii) A reconstruction model reverts the changes to obtain the original key-gate subgraph. The reconstruction model on its own is an ensemble of multiple neural networks each processing a specific locality size. The attacker can perform a subsequent key-guessing attack based on the recovered subgraph. *In case large and complex datasets are not available, the applicability of SAIL is restricted which may render the attack ineffective.*

Research Challenges: Embedding netlist subgraphs using procrustean rules causes the loss of the insight coming from the original irregular structures. The network of wires between gates in an extracted netlist locality conforms well to the topology of a graph. In this work, we hypothesize that ML approaches which exploit the graph structure, such as graph neural networks (GNNs), are at an advantage over traditional models. Although a GNN is a natural choice to learn on the extracted subgraphs, it comes with its own challenges.

1) The expressive power of GNNs is bounded by the 1-Weisfeiler-Lehmann graph isomorphism test, and thus GNNs can generate identical embeddings for graphs that might be different [9].
2) Representing a netlist as an undirected graph makes the subgraphs denser and facilitates message-passing in the network. However, we lose the notion of IN/OUT-neighborhood of the original netlist.

To test our hypothesis that graph-based learning approaches offer advantages over traditional ML-based methods, we need to overcome the above challenges and develop a robust graph-based learning methodology to attack logic locking.

Our Novel Contribution: Towards this, we propose a GNN-based attack called OMLA, which maps the problem of resolving the key-bit value to subgraph classification. OMLA extracts a small subgraph for each key-gate from the locked netlist. The enclosing subgraphs capture the characteristics associated with the key-bit values of the key-gates. Therefore, the label of a subgraph can also be considered the key-bit value. Our framework employs the following key features:

1) *Node Features and Labels [Sec. III-B]:* To facilitate GNN-based learning on circuits, we propose feature extraction and node labeling methods to capture the IN/OUT-neighborhood information in the original netlist that gets lost when represented as an undirected graph.
2) *Simple Graph Learning Model [Sec. III-D]:* Unlike SAIL [5] and SnapShot [6] attacks that require subgraph encoding into tensor form, OMLA directly operates on the arbitrary graph structure for the extracted localities without setting any limits on the input/output size for the gates. OMLA utilizes a GNN to learn representations for the subgraphs and outputs the predicted key-bit values in an end-to-end manner. OMLA directly predicts the key-bit values from the post-synthesis or post-layout localities, without requiring pre-synthesis localities for training.

The key-prediction accuracy of OMLA on ISCAS-85 and ITC-99 benchmarks locked using random logic locking (RLL) [1] is up to 97.22%. In our experiments we show that OMLA outperforms SnapShot and SAIL attacks for all evaluated benchmarks. We further demonstrate that OMLA (i) can be agnostic of the synthesis script/tool, (ii) can handle different synthesis procedures, (iii) can be launched directly on post-layout generation netlists, and (iv) can process netlists synthesized using a full library. **We also open source OMLA [10].**

II. Graph Neural Networks

We consider an undirected graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{X}, \boldsymbol{A})$, where $\mathcal{V}$ is the vertex set, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the edge set, and $\boldsymbol{X} \in \mathbb{R}^{|\mathcal{V}| \times d}$ is the node feature matrix. The adjacency matrix of the graph is $\boldsymbol{A}$, where $A_{u,v} = 1$ if edge $(u, v) \in \mathcal{E}$. GNNs use the structure of the graph and the node features to learn an embedding of a node, $\boldsymbol{h}_u$, or the entire graph $\boldsymbol{h}_\mathcal{G}$. This is achieved through neighborhood aggregation in which the embedding of a node is iteratively updated based on the embeddings of its neighbors. After L iterations of aggregation, a node's embedding captures the node's features, the neighboring nodes' features, and the topological information within the L-hop neighborhood [9]. The l-th layer of a GNN is abstracted as follows.

$$\boldsymbol{a}_v^{(l)} = \text{AGGREGATE}^{(l)}\left(\left\{\boldsymbol{h}_u^{(l-1)} : u \in \mathcal{N}(v)\right\}\right) \quad (1)$$

$$\boldsymbol{h}_v^{(l)} = \text{COMBINE}^{(l)}\left(\boldsymbol{h}_v^{(l-1)}, \boldsymbol{a}_v^{(l)}\right) \quad (2)$$

$\boldsymbol{h}_v^{(l)}$ denotes the embedding of node v at the l-th layer and $\mathcal{N}(v)$ represents the neighbors of node v. Several architectures for AGGREGATE$(\cdot)$ and COMBINE$(\cdot)$ have been proposed in the literature. We abstract the layer operations as follows, where $\boldsymbol{W}^{(\ell)}$ is a learnable weight matrix, and the inputs to the first layer are the node features $\boldsymbol{H}^{(0)} = \boldsymbol{X}$.

$$\boldsymbol{H}^{(\ell)} = f\left(\boldsymbol{H}^{(\ell-1)}, \boldsymbol{A}; \boldsymbol{W}^{(\ell)}\right) \quad (3)$$

$$\boldsymbol{h}_\mathcal{G} = \text{READOUT}\left(\{\boldsymbol{h}_v^{(L)} \mid v \in \mathcal{G}\}\right) \quad (4)$$

The permutation invariant READOUT function aggregates the final nodes' embeddings to obtain a representation vector for the graph $\boldsymbol{h}_\mathcal{G}$, which is used to predict its label.

III. Proposed OMLA Attack

We assume an oracle-less setting in which the attacker is located in the untrusted foundry and the attacker obtains the locked gate-level netlist by reverse-engineering the GDSII file. The location of the key-gates is determined by tracing the key-inputs from the tamper-proof memory [5], [6], [11], [12]. We assume that the foundry has access to the standard cell library used. This assumption is valid in the following two practical scenarios. (i) The standard cell design must meet the design rules put forth by the foundry. Then, the foundry can reach an agreement with the standard cell library vendor and access the library files. (ii) The foundry itself designs the cells and supplies the standard cell library to the design house. Fig. 2 shows an overview of the main steps of OMLA attack, each of which is discussed in detail below.

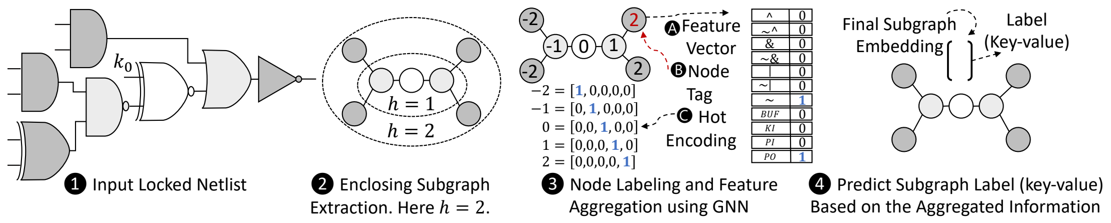
A. Subgraph Extraction

OMLA views the locked netlist as $|\mathcal{V}_k|$ small subgraphs associated with $|\mathcal{V}_k|$ key-gates. After converting the netlist into an undirected graph, OMLA extracts the enclosing subgraphs using h-hop sampling (SAMPLE). Starting from a key-gate $s \in \mathcal{V}_k$, the sampler covers up to h-hops to build the subgraph $\mathcal{G}_{[s]}$ (see ❷ in Fig. 2 for an example of 2-hop sampling).

B. Node Features and Labeling

We assign each node in the extracted subgraph a hot-encoded feature vector of length-d that captures its corresponding Boolean function (see ❸ in Fig. 2). Primary inputs (PIs), primary outputs (POs) and key-inputs (KIs) are not represented as nodes in the subgraphs. However, the feature vector indicates if a gate is connected to a PI, a PO or a KI (see Ⓐ). The length-d depends on the number of Boolean gates in the target technology library. For sequential designs, we can include more entries for the different types of flip flops.

Distance encoding has been proposed in [13] to enhance the structural representation power of GNNs. We follow such an encoding method in which we assign each node in the extracted subgraph a label (tag) that indicates the shortest path distance from the node to the target key-gate in the subgraph



1 Input Locked Netlist
2 Enclosing Subgraph Extraction. Here h = 2.
3 Node Labeling and Feature Aggregation using GNN
4 Predict Subgraph Label (key-value) Based on the Aggregated Information
A Feature Vector
B Node Tag
C Hot Encoding
Final Subgraph Embedding
Label (Key-value)

Fig. 2. OMLA attack stages. The colors dark gray, gray and white represent the 2-hop, 1-hop and 0-hop (root gate) neighborhoods, respectively.

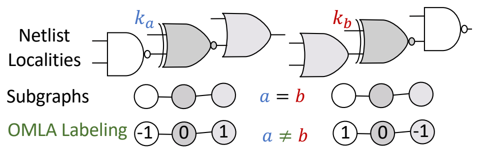


Fig. 3. Processing an undirected graph, a GNN will assign the same embedding to both localities because the corresponding subgraphs are identical. OMLA's annotated subgraphs allow for differentiation between the localities of k_a and k_b and preserve the original direction in the corresponding netlists.

(see B). The target key-gate will always get a unique label of 0, allowing the GNN to distinguish the key-gate from the rest of the nodes. Representing a netlist as an undirected graph causes the loss of IN/OUT-neighborhood notion (see Fig. 3). To overcome such a challenge, we embed the IN/OUT-neighborhood information using node labeling. We additionally assign the label a sign (-/+) to capture the existence of a node in the IN/OUT-neighborhood of the key-gate, while still benefiting from the undirected graph representation (see Fig. 3). The final labels are hot-encoded and concatenated with the original node feature matrix (see C in Fig. 2).

C. Dataset Generation

To train OMLA, we follow the self-referencing model used in SnapShot/SAIL (see Fig. 4). The locked target circuit is used to generate the training/validation set so that the model learns the target design's bias and the behavior of the logic locking method. A dataset is created by copying the target locked netlist N times and then re-locking each copy with a key-size K. The N re-locked netlists are then resynthesized, and the $N \times K$ subgraphs around the newly added key-gates are extracted and split at random using 90:10 ratio for training and validation. During testing, OMLA extracts K subgraphs around the original key-gates and predicts their labels (key-values).

D. Key Prediction as (Sub)graph Classification

The pseudocode of OMLA's inference algorithm is shown in Algorithm 1, where f^L indicates the composition of the function f (3) for the L layers. We build an L-layer GNN to perform graph classification on the extracted subgraphs, where the target labels are the corresponding key-bit values (see 4 in Fig. 2). After obtaining the node embeddings from the final output layer of the GNN ($\boldsymbol{H}_{[s]}^{(L)}$), we can generate the subgraph embedding vector using graph level READOUT function. g is a dense output layer with softmax activation, which maps the subgraph embeddings to the target key-bit values. OMLA is flexible with what GNN to use. Thus, we choose the graph isomorphism network (GIN) architecture [9] as it is one of the most expressive GNNs. GIN updates node representations as follows, where MLP represents a multi layer perceptron.

$$\boldsymbol{h}_v^{(l)} = \text{MLP}^{(l)}\left(\boldsymbol{h}_v^{(l-1)} + \sum\nolimits_{u\in\mathcal{N}(v)} \boldsymbol{h}_u^{(l-1)}\right) \quad (5)$$

Algorithm 1 OMLA Inference Algorithm

```
Input: $\mathcal{G}(\mathcal{V},\mathcal{E},\boldsymbol{X},\boldsymbol{A})$; Target nodes (key-gates) $\mathcal{V}_k$; GNN;
Output: Predicted labels vector $\boldsymbol{Y}$ for $\mathcal{V}_k$ (key-values);
for $s \in \mathcal{V}_k$ do
    Get $\mathcal{G}_{[s]}(\mathcal{V}_{[s]},\mathcal{E}_{[s]},\boldsymbol{X}_{[s]},\boldsymbol{A}_{[s]})$ by SAMPLE on $\mathcal{G}$
    Build $L$-layer GNN with layer operation $f$
    $\boldsymbol{H}_{[s]}^{(L)} \leftarrow f^L(\boldsymbol{X}_{[s]},\boldsymbol{A}_{[s]})$
    $y_{\mathcal{G}_{[s]}} \leftarrow g(\text{READOUT}(\boldsymbol{H}_{[s]}^{(L)}))$
end for
```

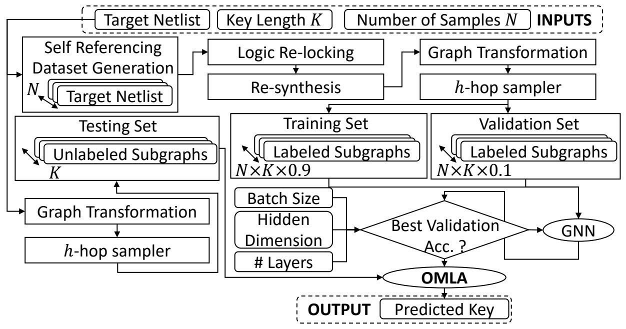


Fig. 4. OMLA flow and experimental setup using the self-referencing model.

GIN considers all structural information from all iterations of the model by replacing (4) with subgraph embeddings concatenated across *all layers* of GIN as follows. The READOUT function adds all node embeddings from the same layer.

$$\boldsymbol{h}_{\mathcal{G}_{[s]}} = \text{CONCAT}\left(\text{READOUT}\left(\left\{\boldsymbol{h}_v^{(l)} | v \in \mathcal{G}_{[s]}\right\}\right) \mid l = 0,1,\ldots,L\right) \quad (6)$$

IV. Experiments

Fairness of Comparison. We evaluate OMLA on the ISCAS-85 [14] and ITC-99 [15] combinational benchmark sets, and directly compare OMLA's results with the results reported for SnapShot and SAIL in [6], making sure we follow the same experimental setup. We implement (i) the same locking scheme (X(N)OR RLL [1]), (ii) lock the same benchmarks, (iii) use the same key-size, and (iv) use the same number of training samples as in [6]. We also use the key-prediction accuracy (KPA) as an evaluation metric similarly to [6], where $KPA = (|K_c|/|K|) * 100$. $|K_c|$ indicates the number of correct key-assignments and $|K|$ denotes the total key-size. The evaluation was performed on an Intel(R) Xeon(R) CPU $X5680$ @$3.33GHz$ with $95GB$ of RAM.

We build a self-referencing training model as discussed in Sec. III-C and set $N = 1000$ and $K = 64$ to be consistent with SnapShot evaluation in [6]. The generated locked netlists are re-synthesized using *Synopsys Design Compiler*. We employ the *Nangate 45nm Open Cell Library* [16] and initially limit the usage of gates with input size greater than two to have

TABLE II
THE HYPERPARAMETERS FOR EACH TARGET DESIGN.

ITC-99 Benchmarks	Batch Size	Hidden Dimension	Layers
b15_C	256	128	6
b21_C	256	128	5
b22_C	256	128	6
b17_C	256	128	6

ISCAS-85 Benchmarks	Batch Size	Hidden Dimension	Layers
c1355	256	64	6
c1908	64	64	6
c2670	128	64	5
c3540	128	64	5
c5315	256	64	5
c6288	128	128	5
c7552	128	64	5

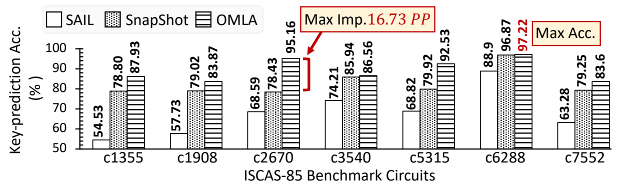


Fig. 5. `OMLA` evaluation on ISCAS-85 benchmarks.

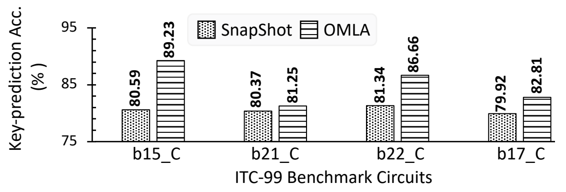


Fig. 6. `OMLA` evaluation on ITC-99 benchmarks.

a fair comparison with state-of-the-art methods. We require 8 entries in each node's feature vector to encode its Boolean functionality. We require additional 3 entries to indicate the connectivity of a gate to PIs, POs and KIs. Lastly, 5 entries are used to hot-encode the gate's tag (See ❸ in Fig. 2). Hence. we initially employ a feature vector length of $d = 16$.

GNN Settings. We use the default parameters for `GIN` [9] and employ 2 layers for all MLPs. We considered $h \in \{1, 2\}$ for each evaluated circuit. However, we consistently observed better performance when using $h = 2$. Setting $L > h$ makes GNNs more sufficiently absorb the entire enclosing subgraph information and learn more expressive key-gate representations [17]. Thus, we experiment with $L \in \{5, 6\}$.[1]

For each benchmark, the training is performed for each batch size, hidden dimension, and number of layers setting. We save the model with the best validation accuracy and use it to evaluate the classification on the testing set. Table II lists the optimized hyperparameters for each circuit.

`OMLA` Performance. The results of `OMLA` on the ISCAS-85 benchmarks are presented in Fig. 5. `OMLA` achieves an avg. KPA of 89.55% and an avg. precision of 83%, outperforming state-of-the-art `SnapShot` and `SAIL` attacks for all evaluated circuits. The results of `OMLA` on ITC-99 benchmarks are consistent with the ISCAS-85 experiment (see Fig. 6); we notice a considerable improvement of KPA for `OMLA` over `SnapShot` in all evaluated cases with an avg. of 84.99%, compared to 80.55% for `SnapShot`. Our results confirm that `OMLA`'s graph-based learning method provides significant improvements over attacks based on traditional ML models.

To further evaluate the effectiveness of `OMLA`, we compare it with the non-ML-based oracle-less redundancy attack of [18] (see Table III). The redundancy attack deciphers a smaller percentage of key-bits compared to `OMLA`. E.g., on the c1355 and c1908 circuits, `OMLA` reports KPA of 87.93% and 83.87%, respectively, while the redundancy attack only deciphers 4.69% and 17.19% of the key-bits, respectively.

[1]For training, Adam optimizer is utilized having an initial learning rate of 0.01. The learning rate is reduced by 0.5 every 50 epochs and a maximum of 500 epochs is set. A dropout ratio of 0.5 is used after the dense layer. The hidden dimension $\in \{64, 128\}$, batch size $\in \{64, 128, 256\}$, and number of layers are tuned for each circuit using a 10% validation set.

Hamming Distance (HD). We measure the HD between the outputs of recovered design by `OMLA` and those of the original design. Additionally, we replace the key obtained by `OMLA` with random keys, which helps us to quantify the functional obfuscation for RLL in general. The ideal goal of the attacker is to get an HD of 0%, while the goal of the defender is to enforce an HD of 50%. For each benchmark, we generate 100 random keys and compute the HD by simulating $100,000$ random input patterns using *Synopsys VCS*. The avg. HD obtained when applying random keys is 23.84%, which indicates a decent amount of obfuscation. On the other hand, the avg. HD value for the ISCAS-85 and ITC-99 benchmarks reconstructed by `OMLA` is a mere 3.9%. Hence, using `OMLA` we (almost) determine the correct functionality. For the c3540 and b22_C benchmarks recovered by `OMLA`, the HD is 0%. An HD 0% implies that `OMLA` fully unlocked the benchmarks. Note that in the case of RLL, multiple keys may activate the correct functionality, and it means that the key reported by `OMLA` was not the one used for locking. However, it happened to be a key that unlocks the design.

Distance Encoding. To evaluate the proposed node labeling (see Sec. III-B), we launch `OMLA` without distance encoding. The avg. KPA on the ISCAS-85 and ITC-99 benchmarks is 87.9% with distance encoding, while it is 78.6% without it. This analysis shows that our proposed distance encoding method enhances the expressive power of the GNN and improves the KPA by an avg. of 9.3 percentage points.

Generic Training Model. To study the effect of following a generic training model on the performance of `OMLA`, we attack one ISCAS-85 benchmark at a time using the rest of the benchmarks for training/validation. The avg. KPA of `OMLA` under this setting is 66.68%. This shows that `OMLA` has better performance when following the self-referencing setup. Since the target netlist is available, there is no reason why the attacker cannot re-lock it and use it for training.

Synthesis-agnostic. We initially assumed access to the original synthesis procedures to generate the dataset for training. But, `OMLA` can be agnostic of the synthesis procedure. Initially, we used one compile_ultra (synthesis command) to synthesize the benchmarks. We refer to this synthesis script as *synthesis-A*. To support our claim, we synthesize the target benchmarks using a synthesis script that uses two compile_ultra commands (*synthesis-B*). Then, we initiate the dataset generation using synthesis-A and attack the netlists synthesized using synthesis-B. The avg. KPA in this case is 85.55% (see Table IV), confirming that `OMLA`'s success does not depend on having access to the original synthesis procedure.

Post-layout Designs. Initially, we launched `OMLA` on the netlists obtained post-synthesis similarly to `SAIL` and `SnapShot`. The ideal approach would be to process post-layout netlists as we assume that the attacker reverse-engineers the GDSII file. After synthesizing the ISCAS-85 benchmarks using the synthesis-B procedure, we perform layout optimization using *Cadence Innovus 17.1*. On avg., `OMLA` achieves a KPA of 85.71% and an HD of 9.63% on the post-layout designs, compared with 87.24% and 9.68% for the corresponding post-

TABLE III
REDUNDANCY ATTACK ON ISCAS-85 BENCHMARKS [18]

Benchmark	**c1355**	**c1908**	**c2670**	**c3540**	**c5315**	**c6288**	**c7552**
Deciphered (%)	4.69	17.19	18.75	68.75	46.88	39.06	40.62

TABLE IV
TRAIN USING *synthesis-A* AND TEST USING *synthesis-B*

Benchmark	**c1355**	**c1908**	**c2670**	**c3540**	**c5315**	**c6288**	**c7552**
KPA (%)	76.79	81.36	93.55	87.69	87.88	95	76.56

synthesis designs, demonstrating OMLA's robustness. *To the best of our knowledge, OMLA is the only structural attack in the literature that is evaluated on post-layout designs.*

Full Library. To demonstrate the generic nature of OMLA, we attack the ISCAS-85 and ITC-99 benchmarks synthesized with a full library. Only the length of the feature vector had to be changed (from 16 to 19) to accommodate the different Boolean functions. The avg. KPA on the benchmarks synthesized with a full library is 84.61%, demonstrating that that OMLA can handle different circuit formats.

To study the **Effect of Synthesis Tool**, we trained and launched OMLA on designs synthesized by the academic *ABC* synthesis tool [19]. The *ABC* tool first transforms a design into and-inverter graph, and then synthesizes it. OMLA achieves an average KPA of 70.98% on the *ABC* synthesized technology-independent ISCAS-85 benchmarks. Even in this challenging setup, OMLA achieves almost the same average KPA reported by SAIL attack (72%) under the more relaxed attack setup assuming access to pre-synthesis localities, synthesis scripts, and following a standard synthesis procedure.

The avg. computational **Runtime** of OMLA on the ISCAS-85 and the ITC-99 benchmarks are $1.85h$ and $4.46h$, respectively, which include subgraphs extraction, distance encoding, training, and inference time. The inference time of OMLA on its own does not take more than $1s$ for all evaluated benchmarks.

Discussion. *GNNUnlock* [7], [20] is an attack on provably secure logic locking that leverage a GNN to identify the protection logic. Unlike OMLA, GNNUnlock is a removal attack and is not applicable for TLL solutions. Note that, apart from the use-case applications, there are also differences concerning the usage of the GNN. OMLA performs a subgraph classification task, whereas GNNUnlock performs a node classification task. OMLA leverages the GIN architecture [9], while GNNUnlock utilizes GraphSAINT [21]. In summary, the goal of the attacks, the GNN models, the classification tasks, the outcomes, and the inner workings of the attacks are distinct from each other.

V. CONCLUSION

This paper proposes OMLA, an oracle-less ML-based attack on logic locking that maps the problem of key-prediction to subgraph classification. OMLA employs a graph neural network to learn a representation for each key-gate by aggregating and updating representation vectors of its neighboring gates, capturing the variations induced by logic locking and revealing the secret key-bit values. Through our experiments on ISCAS-85 and ITC-99 benchmarks, we demonstrate that OMLA outperforms state-of-the-art for all evaluated cases.

REFERENCES

[1] J. Roy, F. Koushanfar, and I. L. Markov, "Ending Piracy of Integrated Circuits," *IEEE Computer*, vol. 43, no. 10, pp. 30–38, 2010.

[2] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Security Analysis of Logic Obfuscation," in *DAC*, 2012, pp. 83–89.

[3] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the Security of Logic Encryption Algorithms," in *HOST*, 2015, pp. 137–143.

[4] L. Alrahis, M. Yasin, N. Limaye, H. Saleh, B. Mohammad, M. Alqutayri *et al.*, "ScanSat: Unlocking static and dynamic scan obfuscation," *IEEE TETC*, 2019.

[5] P. Chakraborty, J. Cruz, and S. Bhunia, "SAIL: Machine learning guided structural analysis attack on hardware obfuscation," in *AsianHOST*, 2018, pp. 56–61.

[6] D. Sisejkovic, F. Merchant, L. M. Reimann, H. Srivastava, A. Hallawa, and R. Leupers, "Challenging the security of logic locking schemes in the era of deep learning: A neuroevolutionary approach," *arXiv preprint arXiv:2011.10389*, 2020.

[7] L. Alrahis, S. Patnaik, F. Khalid, M. A. Hanif, H. Saleh, M. Shafique *et al.*, "GNNUnlock: Graph neural networks-based oracle-less unlocking scheme for provably secure logic locking," in *DATE*, 2021, pp. 780–785.

[8] L. Alrahis, M. Yasin, H. Saleh, B. Mohammad, and M. Al-Qutayri, "Functional Reverse Engineering on SAT-Attack Resilient Logic Locking," in *ISCAS*. IEEE, 2019, pp. 1–5.

[9] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *ICLR*, 2019.

[10] L. Alrahis, S. Patnaik, M. Shafique, and O. Sinanoglu. (2021) OMLA: An Oracle-less Machine Learning-based Attack on Logic Locking. [Online]. Available: https://github.com/DfX-NYUAD/OMLA

[11] Y. Zhang, P. Cui, Z. Zhou, and U. Guin, "TGA: An oracle-less and topology-guided attack on logic locking," in *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop*, 2019, pp. 75–83.

[12] F. A. Petitcolas, "Kerckhoffs' principle." 2011.

[13] P. Li, Y. Wang, H. Wang, and J. Leskovec, "Distance encoding–design provably more powerful GNNs for structural representation learning," *arXiv preprint arXiv:2009.00142*, 2020.

[14] M. C. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering," *IEEE Design & Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.

[15] S. Davidson, "Notes on ITC'99 Benchmarks," https://github.com/squillero/itc99-poli, 1999.

[16] (2011) NanGate FreePDK45 Open Cell Library. Nangate Inc. [Online]. Available: http://www.nangate.com/?page_id=2325

[17] H. Zeng, M. Zhang, Y. Xia, A. Srivastava, R. Kannan, V. Prasanna *et al.*, "Deep graph neural networks with shallow subgraph samplers," *arXiv preprint arXiv:2012.01380*, 2020.

[18] L. Li and A. Orailoglu, "Piercing logic locking keys through redundancy identification," in *DATE*, 2019, pp. 540–545.

[19] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 24–40.

[20] L. Alrahis, S. Patnaik, M. A. Hanif, H. Saleh, M. Shafique, and O. Sinanoglu, "GNNUnlock+: A systematic methodology for designing graph neural networks-based oracle-less unlocking schemes for provably secure logic locking," *IEEE TETC*, 2021.

[21] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "GraphSAINT: Graph Sampling Based Inductive Learning Method," in *ICLR*, 2019.