Characterize the ability of GNNs in attacking logic locking

Wei Li*, Ruben Purdy*, José M. F. Moura*, R.D. Blanton*
*Electrical and Computer Engineering Department
Carnegie Mellon University, Pittsburgh, United States

***Abstract*—Recent research has showcased the capability of Graph Neural Networks (GNNs) to uncover the correct keys from locked circuits without an oracle. However, a comprehensive comprehension and discourse regarding their achievements remain limited. For instance, which specific GNN architectures possess a greater potency in effectively uncovering these keys? In this paper, we model their ability to identify circuit changes that stem from a logic lock as the ability to decide the isomorphism between logic netlists. We examine different netlist graph representations and characteristics of GNNs that improve the ability in graph isomorphism. We show that GNNs are always upper bounded by heterogeneous Weisfeiler Lehman test for netlist isomorphism determination, and give the conditions when GNNs reach the bound. Motivated by these conditions, a specific GNN architecture, NetlistGNN, is proposed. Experimental results align with our theorems, and demonstrate that GNN-based state-of-the-art attacking models can benefit from our findings by simply replacing the original GNN model to NetlistGNN.**

***Index Terms*—Logic locking, Oracle-less attacks, Graph neural networks**

I. Introduction

The significant expense of integrated circuit (IC) manufacturing has driven most design companies to outsource fabrication to third-party foundries. This decades-long shift has introduced a number of threats to IC security. For example, a nefarious third party could engage in malicious circuit alteration, overproduction, or IP theft.

In response to these concerns, a number of design protection techniques have been developed for protecting logic circuits, the "secret sauce" for most ICs. Among these, logic locking has garnered a remarkable level of interest as a method to prevent untrusted third parties from gaining a functional version of an IC through overproduction or reverse engineering. There has been a great deal of work in the area of logic locking such as [1], [2], [3], [4], [5], [6], [7], [8], and many more. Logic locking modifies a logic circuit so that it only operates correctly when it is programmed with secret keys. Thus, a third party without the knowledge of the correct keys would only access the inoperable circuit.

However, many logic locking techniques are vulnerable to attacks that allow untrusted parties to uncover the correct keys. Many attacks also require the use of an oracle, which is a properly programmed version of the circuit in which the keys are stored in a tamper proof memory. Although an attack cannot directly read out the keys from the oracle, they can be uncovered using the input/output behavior of the oracle along with simulation or analysis of the locked logic netlist.

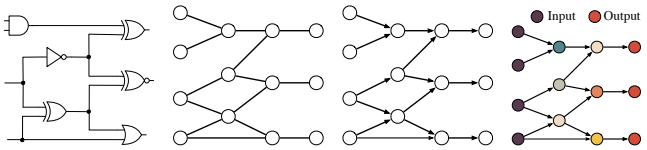
Recent work has shown that Graph Neural Networks (GNNs) are able to uncover the correct keys from locked circuits through analysis of the locked logic netlist alone [6], [8], [7]. Thus, these attacks can be applied even in situations where an adversary is unable to acquire an oracle. Additionally, these attacks can uncover keys from circuits locked with locks that are resistant to traditional oracle-based attacks. The intuition behind the success of GNNs in attacking a locked circuit lies in the fact that the logic netlist exhibits repeated sub-circuits, and GNNs are well-equipped at learning the local structure of the circuit, meaning that GNNs can capture those repeated sub-circuits, and distinguish between sub-circuits that have and have not been altered by a given lock. However, despite the notable success of GNNs in revealing the correct key, there is limited understanding and discussion regarding the "why" of their success, e.g., what types of GNN architectures are more powerful in the attacking problem? In this paper, we examine the characteristics of GNNs that lead to their ability to identify circuit changes stemming from a logic lock. Our framework is inspired by previous works [9] that demonstrate the capability of GNNs in determining graph isomorphism being upper bounded by the Weisfeiler-Lehman (WL) test. We introduce three graph representations of a logic netlist, namely, undirected, directed, and heterogeneous. We explore the relationship between GNNs and the WL test on the three representations. The contributions are as follows:

- We show that GNNs are at most as powerful as the WL test in distinguishing logic netlists under all three graph representations.
- We establish conditions on the GNNs and graph representations to make the GNN as powerful as the WL test or its variations.
- Using these conditions, NetlistGNN, a specific GNN architecture that is maximally powerful is proposed.
- Experiments align with our theoretical analysis and demonstrate that state-of-the-art attacking methods benefit from NetlistGNN in most cases.

II. Preliminary

Graph representation of a logic netlist. In this paper, we focus on the gate-level logic netlist, which is composed of various gates, and interconnects. A netlist can be also modeled as a graph, where each gate is a node, and the interconnect form the edges. Differentiated by the graph type, there are three possible netlist graph representations: undirected, di-

979-8-3503-0955-3/23/$31.00 ©2023 IEEE



(a) (b) (c) (d)

Fig. 1: (a) An example netlist with corresponding (b) undirected, (c) directed, and (d) heterogeneous graph models.

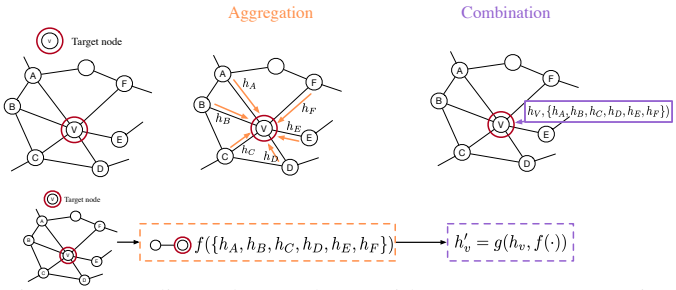


Fig. 2: An undirected GNN layer with two steps: aggregation and combination.

rected, and heterogeneous.[1] Examples of three netlist graph representations are shown in Figure 1.

Graph Neural Networks. Graph Neural Networks (GNNs) learn the node embeddings or graph embedding given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and node attributes $\{\boldsymbol{h}_v^0 : v \in \mathcal{V}\}$, where an embedding is a low-dimensional vector that contains information about the original object. Most GNNs are composed of a sequence of layers where each node *aggregates* features from its neighbors and *combines* the aggregated information. Figure 2 shows how a GNN layer works for an undirected graph. Let $\mathcal{A}$ be a general GNN for undirected graphs, at each layer i, $\mathcal{A}$ computes the node feature $\boldsymbol{h}_v^{(i)}$ for every node $v \in \mathcal{V}$ by:

$$\boldsymbol{h}_v^{(i)} = \text{COM}^{(i)}\left(\boldsymbol{h}_v^{(i-1)}, \text{AGG}^{(i)}\left(\left\{\boldsymbol{h}_u^{(i-1)} : u \in \mathcal{N}(v)\right\}\right)\right), \tag{1}$$

where $\mathcal{N}(v)$ denotes the neighborhood of v, i.e., $\mathcal{N}(v) = \{u : \{u, v\} \in \mathcal{E}\}$, and $\text{AGG}^{(i)}$ ($\text{COM}^{(i)}$) is the *aggregation* (*combination*) function of a GNN $\mathcal{A}$ at i_{th} layer. The output of the layer sequence, $\boldsymbol{h}_v^{(L)}$, is the node embedding of v generated by $\mathcal{A}$. To obtain the graph embedding $\boldsymbol{h}_\mathcal{G}$, a readout function $\text{READ}(\cdot)$ is applied to the node embeddings:

$$\boldsymbol{h}_\mathcal{G} = \text{READ}\left(\left\{\boldsymbol{h}_v^{(L)} : v \in \mathcal{V}\right\}\right) \tag{2}$$

Directed GNNs. When the graph is directed, one possible variation is to aggregate features from successors and predecessors separately. Formally, the input of the aggregation function is:

$$\left(\left\{\boldsymbol{h}_s^{(i-1)} : s \in \mathcal{S}(v)\right\}, \left\{\boldsymbol{h}_p^{(i-1)} : p \in \mathcal{P}(v)\right\}\right) \tag{3}$$

where $\mathcal{S}(v)$ and $\mathcal{P}(v)$ are the successors and predecessors of v, respectively.

Heterogeneous GNNs. For heterogeneous graphs, the aggregation is applied to different metapaths, where a metapath in the netlist graph is defined as a combination of two node types and one edge type. Formally, the input of the aggregation function is:

$$\left(\left\{\boldsymbol{h}_u^{(i-1)} : u \in \mathcal{N}_m(v)\right\} : m \in \text{metapaths}\right) \tag{4}$$

Here, $\mathcal{N}_m(v)$ is the set of neighbors of v that connects v by a specific meta-path $m = i\text{-}e\text{-}j$, i.e., $\mathcal{N}_m(v) = \{u : \{u, v\} \in \mathcal{E}_e,\ type(u) = i,\ type(v) = j\}$.

Directed WL test. Similar to directed GNNs, the WL test is easily extended to a directed graph by aggregating successors and predecessors separately. That is,

$$c_v'^{(t)} = g\left(c_v'^{(t-1)}, \left\{\left\{c_s'^{(t-1)} : s \in S_v\right\}\right\}, \left\{\left\{c_p'^{(t-1)} : p \in P_v\right\}\right\}\right) \tag{5}$$

Heterogeneous WL test. The WL test is also extended to the heterogeneous cases where different metapaths give different multi-sets:

$$c_v^{H(t)} = g\left(c_v^{H(t-1)}, \left(\left\{\left\{c_u^{H(t-1)} : u \in \mathcal{N}_m(v)\right\}\right\} : m \in \text{metapaths}\right)\right)$$

Logic locking. Formally, a logic-locking technique transforms a circuit $\mathbb{C} : \{0,1\}^n \rightarrow \{0,1\}^m$ into a locked circuit $\mathbb{L} : \{0,1\}^n \times \{0,1\}^k \rightarrow \{0,1\}^m$ locked with a key $K : \{0,1\}^k$. Applying the correct key to the locked circuit restores the functionality of the original circuit, such that $\mathbb{L}(K, X) = \mathbb{C}(X)$ for any input $X : \{0,1\}^n$. With an incorrect key applied, the locked circuit may disagree with the original circuit, preventing the use of the locked circuit by unauthorized parties. The first proposed logic locking technique, referred to as Random Logic Locking (RLL) [1], inserts parity gates at random nets in the circuit. The other inputs to the gates are driven by the key inputs so that an incorrect key value will invert the value of the net. LUT lock [2] replaces gates in the circuit with lookup tables, specifically targeting gates along paths that which are difficult to attack using SAT. Besides RLL and LUT-based locking methods, MUX-based methods, which add incorrect paths to the circuit which must be ignored by correctly setting MUX select lines, [4], [5] have also been widely studied.

Attack model. Attack is to predict the correct key value given a locked netlist. Figure 4 shows the general workflow of GNN-based attacks. Given a locked netlist, the first step is to construct the corresponding graph representation, and the target key prediction problem is formulated as either a node classification or edge prediction problem and predicted by GNN. Simple examples of different locking methods and corresponding graph representations are shown in Figure 3. The attack for the parity gate-based and LUT-based locking can be directly translated into a node classification problem, where the white node is the node to be classified. For the parity gate-based locking, the white node is the key node and the label is the key value, i.e., 0 or 1. For the LUT-based locking, the white node denotes the LUT node and the label is the type of the original gate replaced by the LUT module. To attack the MUX-based locking, the problem is naturally an edge prediction problem, where orange edges are to be predicted, i.e., which one exists in the original circuit. Alrahis [6], [7] models the edge prediction as a graph classification problem by extracting the sub-graphs surrounding the target edges, and use the graph embedding to predict the edges.

[1]Formal definitions can be found at the appendix: https://anonymous.4open.science/r/NetlistGNN/proof.pdf

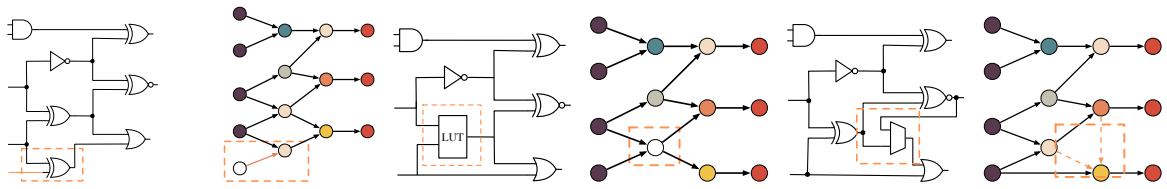


Fig. 3: Given the original logic netlist as in Figure 1a, the logic netlist is locked by (a) parity gate-based (c) LUT-based, and (e) MUX-based logic locking. The right graphs are their corresponding graph representations.

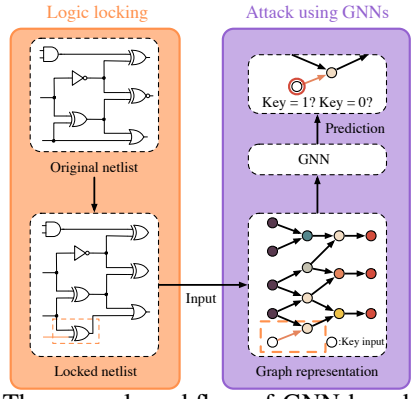


Fig. 4: The general workflow of GNN-based attacks.

Algorithm 1 `AttackTRLL`

Require: $\mathcal{N} \rightarrow$ the locked netlist by TRLL [3];
Require: $f \rightarrow$ A perfect graph discriminator;
$L \leftarrow$ extract all subgraphs from $\mathcal{N}$;
for each key input k in $\mathcal{N}$ **do**
  $\mathcal{G}_k \leftarrow$ extract the subgraph of $\mathcal{N}$ rooted from the input node corresponding to k;
  for each subgraph $\mathcal{G}_l$ in L **do**
    Determine the discrepancy between $\mathcal{G}_k$ and $\mathcal{G}_l$ by $f(\mathcal{G}_k, \mathcal{G}_l)$;
    if the discrepancy is case S_{1a} or case S_{2a} **then**
      vote $k = 1$; ▷ S_{1a} and S_{2a} are from [3]
    else if the discrepancy is is case S_{1b} or case S_{2b} **then**
      vote $k = 0$; ▷ S_{1b} and S_{2b} are from [3]
    end if
  end for
  Determine k by majority vote;
end for

Because GNN-based attacks only rely on structural information from the locked circuit, they do not require attackers to obtain an oracle. Thus, they are applicable even in situations where attackers are unable to purchase an unlocked version of the circuit from the market. This is an advantage over SAT- or ATPG-based attacks.

III. Theoretical Framework

In this section, we analyze when and how GNNs are able to uncover more keys from a locked circuit.

	Random Netlist	Regular half-subtractor
GCN [10]	0.5629	1.0000
GraphSAGE [11]	0.6214	0.9982

TABLE I: The key prediction accuracy of GCN and GraphSAGE on attacking Random Logic Locking (RLL). The random netlist is generated using circuitgraph [12]. For regular circuits, half-subtractors with different bits are generated.

Our intuition comes from the motivating example shown in Table I: GNN-based attack methods can easily uncover keys from half-subtractor, which is a regular circuit, i.e., circuit with many regular and repeated moudules. However, given a randomly generated circuit, GNN-based method fails to attack it. This hints that the regular structure of the circuit is the key to the success of GNN-based attack: Given a locked circuit without an oracle, the only visible information for the attacker is the structure of the locked circuit. At the same time, since the number of locked keys is limited, there is still a significant portion of sub-circuits that are not altered by keys. As long as there exists similarity between locked part and unaltered part, a learning algorithm can learn the structure information from the unaltered sub-circuits, and then use the learned structure information to distinguish the altered sub-circuits from the unaltered ones, thereby uncovering the keys.

The ability to learn graph structure is highly related to the ability to distinguish graphs. If the netlist graph is composed of many repeated sub-graphs, and there is a "perfect" discriminator that can distinguish any two graphs and locate the difference, then the discriminator can learn any trivial difference between graphs, thereby being more capable of "guessing" the key value by comparing the difference caused by the key insertion. Assume we have a "perfect" discriminator, Algorithm 1 is a theoretical algorithm to attack Truly Random Logic Locking (TRLL) [3], which claims to be learning-resistent. The key idea is to extract all sub-circuits from the locked circuit, and compare the with-key sub-circuit with the without-key sub-circuit. The key value is determined by the discrepancy found by the discriminator.

Inspired by this intuition, we characterize the power of GNNs in attacking as its ability to distinguish logic netlists. Formally, we define the statement "*B is at least as powerful as A*" as follows:

Definition 1 ($A \preceq B$)**.** *Assume GNNs and WL test have the same node attributes and initial node labels if the input graphs are from the same logic netlist. Given two logic netlists $\mathcal{N}_1, \mathcal{N}_2$, let the input of $A(B)$ be one of the three netlist graph representations, and defined as $\mathcal{G}_1^A, \mathcal{G}_2^A$ ($\mathcal{G}_1^B, \mathcal{G}_2^B$). We say that B is at least as powerful as A ($A \preceq B$) if and only if the following statement holds: If A decides $\mathcal{G}_1^A$ and $\mathcal{G}_2^A$ are not isomorphic, then B also decides $\mathcal{G}_1^B$ and $\mathcal{G}_2^B$ are not isomorphic.* [2]

Specifically, we say $A = B$ if and only if $A \preceq B$ and $B \preceq A$; $A \prec B$ if and only if $A \preceq B$ holds but $B \preceq A$ does not hold. In the following section, we will construct the

[2]Preliminary of WL test and all proofs can be found at the appendix.

relationships among different GNNs and WL test variations based on the definition above.

A. Undirected vs. Directed vs. Heterogeneous

Undirected [13], [6], [8], [7] and directed [14], [15] graphs were both used to represent the logic netlists. To the best of our knowledge, the heterogeneous netlist graph was never discussed, let alone the comparison with the three representations. In this section, we study the influence of the three graph representations, that is, under which representations are GNNs and WL test the most powerful?

We first discuss the power of WL test and its variations under different graph representations. The following lemma suggests that the directed WL test is at least as powerful as the original undirected WL test in distinguishing the netlist graph.

Lemma 1. *Undirected WL $\prec$ directed WL*

Besides undirected and directed WL test on the netlist graphs, we can also derive the relationship between the directed WL test and heterogeneous WL test:

Lemma 2. *Directed WL $\preceq$ heterogeneous WL*

Unlike the comparison between directed and undirected WL test, it is possible for directed WL test to be as powerful as the heterogeneous WL test. The conditions are specified as follows:

Lemma 3. *Directed WL $=$ heterogeneous WL when nodes with different gate types are assigned different initial node labels.*

We already show that the heterogeneous WL test is the most powerful among three WL test variations. The inequality relationships also hold in GNNs: It is easy to see that heterogeneous GNNs are at least as powerful as other GNN variations, since the heterogeneous GNNs can always reduce to directed GNNs and normal GNNs. Formally, we have:

Lemma 4. *Undirected GNNs $\prec$ directed GNNs*

Lemma 5. *Directed GNNs $\preceq$ heterogeneous GNN*

B. GNNs versus WL test

Xu [9] demonstrates that the power of GNNs is bounded by WL test in determining the isomorphism of undirected graphs. However, as shown in previous lemmas, both GNNs and WL test are not as powerful as their directed and heterogeneous versions in determining the isomorphism of netlists. Here, we compare GNNs and WL test by directly comparing their maximally powerful version: the heterogeneous GNNs and heterogeneous WL test:

Lemma 6. *Heterogeneous GNNs $\preceq$ heterogeneous WL test*

Although GNNs are upper bounded by heterogeneous WL test, they are still possible to reach the bound, i.e., be equally powerful as the heterogeneous WL test:

Theorem 1. *Let $\mathcal{A}$ be a GNN with a sufficient number of GNN layers. $\mathcal{A}$ $=$ heterogeneous WL test if the following conditions hold:*

- *the aggregation and combination functions of $\mathcal{A}$ are injective.*
- *the readout function of $\mathcal{A}$ is injective.*
- *At least one of the two conditions holds: 1) $\mathcal{A}$ separately aggregates features from different meta-paths. 2) $\mathcal{A}$ separately aggregates features from the successors and predecessors, and nodes with different gate types are assigned different initial node attributes.*

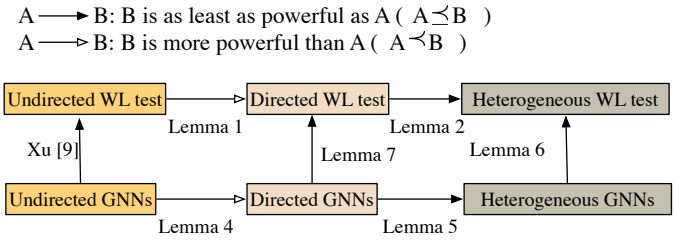


Fig. 5: The power relations of GNNs and WL tests in determining the logic netlist isomorphism. Different colors represent different graph representations.

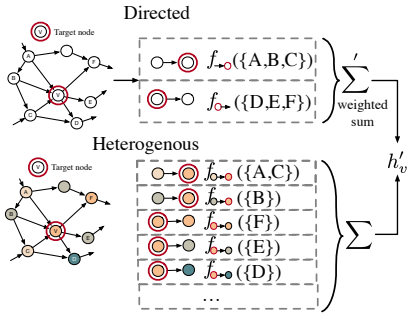


Fig. 6: The visualization of directed NetlistGNN and heterogeneous NetlistGNN, h'_v is the new node embedding of v.

The summary of all relationships are given in Figure 5. Based on Theorem 1, a guidance for building powerful GNNs can be naturally provided. However, the theoretical upper bound is hard to reach since injectivity cannot be guaranteed, and the requirement of layers being sufficiently deep is not feasible in practice. Moreover, injective functions are not unique, meaning that significant efforts are still needed to determine which GNN setting is better. In the appendix, we discuss these problems and propose corresponding concrete solutions.

IV. Method

In this section, we describe how our findings on GNNs are applied to attack the logic lock. We first propose our GNN architecture: NetlistGNN. Then, we introduce how NetlistGNN is used in the attacking flow.

A. NetlistGNN

Based on Theorem 1, both directed GNNs and heterogeneous GNNs can reach the limit bounded by heterogeneous WL test. Correspondingly, we propose two variations. A figure illustration of the two variations are shown in Figure 6. In NetlistGNN, successors and predecessors are aggregated separately (Theorem 1). The layer of the directed NetlistGNN

TABLE II: Attacking results of UNTANGLE[7] and UNTANGLE+ (replacing original DGCNN by directed NetlistGNN) on random MUX-based lock. K is the number of testing keys; ✓ (×) is the number of keys that are predicted correctly (not correctly); ? is the number of keys that cannot be decided in the postprocess.

Benchmark	K	UNTANGLE			UNTANGLE+		
		✓	×	?	✓	×	?
c7552	64	62	**0**	2	64	**0**	0
	128	115	2	9	124	**1**	1
	256	224	3	19	243	**2**	1
c5315	64	58	0	5	60	**1**	2
	128	112	**3**	7	116	6	0
	256	203	16	19	229	**6**	3
c3540	64	56	6	2	62	**2**	0
	128	109	6	12	120	**4**	3
	256	230	18	5	237	**13**	3
c2670	64	55	6	3	58	**4**	2
	128	113	**4**	11	108	9	11
	256	213	**12**	31	223	13	20
b22	512	485	8	19	499	**7**	6
b21	512	482	14	15	500	**4**	7
b20	512	485	14	13	506	**4**	2
b14	512	478	21	13	505	**5**	2

TABLE III: Attacking results of MuxLink[6] and MuxLink+ (replacing original DGCNN by directed NetlistGNN) on D-MUX lock [4]. Columns follow the same definitions in Table II. Time is the training time.

Benchmark	K	MuxLink				MuxLink+			
		✓	×	?	time (s)	✓	×	?	time (s)
c7552	256	246	10	0	98	245	**9**	2	121
c6288	256	255	0	1	86	256	0	0	116
c5315	256	244	**8**	4	88	243	12	1	114
c3540	256	224	**30**	2	78	224	31	1	90
c2670	256	223	31	2	50	225	**29**	2	62
b22	512	492	13	7	1423	494	**11**	7	1812
b21	512	496	**4**	12	912	497	6	9	1056
b20	512	488	19	5	880	486	**16**	10	1031
b14	512	486	24	2	162	490	**19**	3	198

is formalized as:

$$\boldsymbol{h}_v^{(t)} = \Theta_1 \boldsymbol{h}_v^{(t-1)} + \sum_{j \in \mathcal{S}(i)} \epsilon_{i,j}^{s} \mathbf{\Theta_s} \mathbf{h}_j^{(t-1)} + \sum_{j \in \mathcal{P}(i)} \epsilon_{i,j}^{p} \mathbf{\Theta_p} \mathbf{h}_j^{(t-1)} \tag{6}$$

where Θ_i is the trainable weight, and the $\epsilon_{i,j}^{p}(\epsilon_{i,j}^{s})$ is the attention coefficient using the same calculation method in [16].

Similarly, for the heterogeneous NetlistGNN , the layer is formalized as:

$$\boldsymbol{h}_v^{(t)} = \Theta_1 \boldsymbol{h}_v^{(t-1)} + \sum_{m \in \text{metapaths}} \sum_{j \in \mathcal{N}_m(i)} \frac{\mathbf{\Theta_m} \mathbf{h}_j^{(t-1)}}{|\mathcal{N}_m(i)|} \tag{7}$$

In the heterogeneous version, we do not use attention for better generalization: attention makes the model easy to be overfitting, especially when there are $|nodetypes| \times 2 \times |nodetypes|$ metapaths.

B. Workflow for attacking logic lock

For attacking different logic lock methods, GNNs are used in different ways. In this paper, we focus on the GNN architecture, therefore, we only cover a basic and general workflow here (shown in Figure 4). It is recommended to read [17] for details about applying GNNs on various logic lock. Given a locked logic netlist, it is translated into a netlist graph representation. Then, we insert keys to the logic netlist using the same logic lock technique, these newly inserted keys are for training and validation. Depending on the logic lock types, the corresponding attacks can be translated into either node classification or graph classification. Details can be found in Section II.

V. Results

In the experiments, we try to verify our theoretical findings. Therefore, we mainly consider two questions:

- RQ1: Can previous GNN-based attacking methods directly benefit from our theoretical findings by simply replacing the original GNN model to NetlistGNN?
- RQ2: Do our statements hold in practice, even under various scenarios, like synthesis and corruptibility?

Different models are assigned different hidden dimension to make the model complexity comparable: directed NetlistGNN is 32, and heterogeneous NetlistGNN is 16, while DGCNN, GraphSAGE, and our undirected variant are 64. We perform the experiments on a Tesla V100 GPU. ISCAS-85, ITC-99 datasets,and Common Evaluation Platform (CEP) datasets are used in our experiments.

A. RQ1

In RQ1, we keep all the same with the previous attacking methods, except that the GNN model is replaced to directed NetlistGNN with the same complexity and model depth.

Comparison with UNTANGLE[7]. The results are shown in Table II. Generally, after using NetlistGNN, UNTANGLE+ reduces the false predicted number by 40% for the small ISCAS-85 dataset, and 65% for the relatively large ITC-99 dataset. **Comparison with MuxLink[6]**. To integrate NetlistGNN into MuxLink, we preserve the sortpooling used in MuxLink and replace the GCN layers by directed NetlistGNN layer (Equation (6)). The results are shown in Table III. Enhanced with NetlistGNN, MuxLink+ is able to reduce the faulty prediction by 15% on large ITC-99 dataset, with a slight runtime increase.

B. RQ2

For RQ2, we train the model for 1000 epochs, and use the model with the best validation performance for testing. We evaluate our theorems on two representative parity-gate based methods: Truly Random Logic Lock (TRLL) [3] in Table IV and Random Logic Lock (RLL) [1] in Figure 7 and Figure 8. Both of them are implemented by circuitgraph library [12]. Node attributes are one-hot features representing gate function; 30% available locations are selected to insert new keys for training, validation, and testing with ratio 3:1:1. The attacking problem is treated as node classification, and node label is its key value in parity-based methods. More experiments can be found at the appendix.

Comparison with other GNN models. We compare graphSAGE [11] and GATv2 [16] with directed (Equation (6)) and heterogeneous (Equation (7)) NetlistGNN. Our models are much better than previously-used GNN model. Moreover, it is observed that all GNNs perform better in the larger case, which aligns with our intuition that GNNs are successful in attacking because of its ability in capturing local structure since larger case always provides more local structure information.

Undirected vs. Directed vs. Heterogeneous As shown in Table IV, where "Ours w. undirected" is the undirected version of our attention-based layer, i.e., the GATv2 model. When the directed and heterogeneous graph representations are reduced to the undirected graph, the accuracy drops a lot. The directed

TABLE IV: Prediction accuracy of GraphSAGE [11], GATv2 [16], and NetlistGNN on Truly Random Logic lock [3]

Dataset	ISCAS85		CEP					ITC99					
Benchmark	c5315	c7552	des3	sha256	FIR_filter	md5	IIR_filter	b14	b18	b19	b20	b21	b22
# of nodes	3968	4302	13128	53385	16225	21766	26091	8022	114463	223111	20022	20411	29147
GraphSAGE	0.8311	0.8766	0.834	0.9793	0.992	0.9175	0.9692	0.8277	0.8871	0.8754	0.8655	0.8647	0.8659
Ours directed	0.8378	0.9481	0.9538	**0.9948**	0.994	0.9782	0.9835	**0.8986**	**0.9687**	**0.9834**	**0.9556**	0.9499	0.9577
Ours heterogeneous	**0.9527**	**0.9675**	**0.9769**	0.9868	**1.0**	**0.9794**	**0.9956**	**0.8986**	0.9627	0.9739	0.9417	**0.9513**	**0.9622**
Ours w. undirected (GATv2)	0.8649	0.8247	0.8803	0.9807	0.976	0.921	0.9714	0.8243	0.9217	0.9157	0.8308	0.866	0.9415

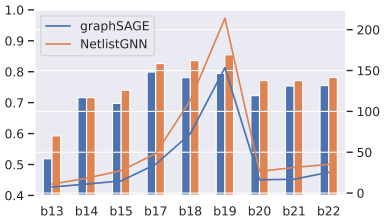


Fig. 7: The GraphSAGE and NetlistGNN results on attacking circuits locked by RLL after synthesis. The bars (left y-axis) are test accuracy, and the lines (right y-axis) are training time in seconds.

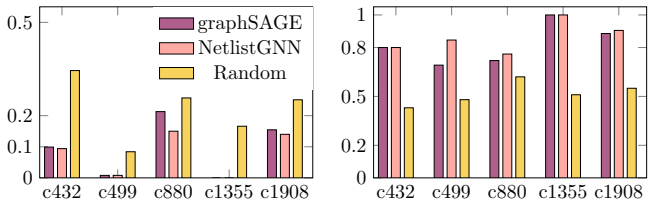


Fig. 8: (a) Corruptibility and (b) Accuracy on attacking RLL. Random is the average result over ten independent random key guesses.

NetlistGNN with gate function as node attribute achieves a similar performance with heterogeneous NetlistGNN.

Impact of Synthesis In order to study the impact of synthesis on the accuracy of the GNN attack, comparing performance after synthesis is performed. We use Synopsys DesignCompiler to synthesize to a generic library. The results are presented in Figure 7. We observe that: 1) Synthesis is an effective method to defend GNN attack. 2) NetlistGNN always performs better than graphSAGE, with a few sacrifices of training time.

Corruptibility Because the GNN attack does not guaranteed the correctness of keys, it is important to quantify how much the generated key degrades security. Corruptibility is the fraction of inputs for which the outputs of the locked circuit and original circuit disagree when a particular key is applied to the locked circuit. We measure bit-wise corruptibility, meaning we calculate corruptibility for each output separately and then average over all outputs. Figure 8 compares the corruptibility of random keys with the corruptibility of keys predicted by GNNs. Since corruptibility calculation is too time-consuming, we only count circuits with number of gates less than 1500. We can see that applying GNNs significantly reduce the corruptibility, where NetlistGNN is still better than graphSAGE. At the same time, the drop of corruptibility is generally related with accuracy: higher accuracy usually indicates lower corruptibility.

VI. Conclusion

In this paper, we developed a theoretical framework for reasoning about the ability of GNNs in identifying circuit changes that stem from a logic lock. We also proved an ability upper bound of GNNs and when GNNs reach the upper bound.

References

[1] J. A. Roy, F. Koushanfar, and I. L. Markov, "EPIC: Ending Piracy of Integrated Circuits," *Design, Automation & Test in Europe*, pp. 1069–1074, 2008.

[2] H. M. Kamali, K. Z. Azar, K. Gaj, H. Homayoun, and A. Sasan, "LUT-lock: A Novel LUT-Based Logic Obfuscation for FPGA-Bitstream and ASIC-Hardware Protection," *IEEE Computer Society Annual Symposium on VLSI*, pp. 405–410, 2018.

[3] N. Limaye, E. Kalligeros, N. Karousos, I. G. Karybali, and O. Sinanoglu, "Thwarting all logic locking attacks: Dishonest oracle with truly random logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 9, pp. 1740–1753, 2020.

[4] D. Sisejkovic, F. Merchant, L. M. Reimann, and R. Leupers, "Deceptive logic locking for hardware integrity protection against machine learning attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[5] A. Alaql, M. M. Rahman, and S. Bhunia, "Scope: Synthesis-based constant propagation attack on logic locking," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 8, pp. 1529–1542, 2021.

[6] L. Alrahis, S. Patnaik, M. Shafique, and O. Sinanoglu, "Muxlink: Circumventing learning-resilient mux-locking using graph neural network-based link prediction," *arXiv preprint arXiv:2112.07178*, 2021.

[7] L. Alrahis, S. Patnaik, M. A. Hanif, M. Shafique, and O. Sinanoglu, "Untangle: Unlocking routing and logic obfuscation using graph neural networks-based link prediction," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[8] L. Alrahis, S. Patnaik, F. Khalid, M. A. Hanif, H. Saleh, M. Shafique, and O. Sinanoglu, "Gnnunlock: Graph neural networks-based oracle-less unlocking scheme for provably secure logic locking," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 780–785.

[9] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[10] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016.

[11] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," 2017, pp. 1024–1034.

[12] J. Sweeney, R. Purdy, R. D. Blanton, and L. Pileggi, "Circuitgraph: A python package for boolean circuits," *Journal of Open Source Software*, vol. 5, no. 56, p. 2646, 2020.

[13] L. Alrahis, A. Sengupta, J. Knechtel, S. Patnaik, H. Saleh, B. Mohammad, M. Al-Qutayri, and O. Sinanoglu, "Gnn-re: Graph neural networks for reverse engineering of gate-level logic netlists," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[14] Z. He, Z. Wang, C. Bail, H. Yang, and B. Yu, "Graph learning-based arithmetic block identification," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–8.

[15] Z. Guo, M. Liu, J. Gu, S. Zhang, D. Z. Pan, and Y. Lin, "A timing engine inspired graph neural network model for pre-routing slack prediction," 2022.

[16] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?" *arXiv preprint arXiv:2105.14491*, 2021.

[17] L. Alrahis, S. Patnaik, M. Shafique, and O. Sinanoglu, "Embracing graph neural networks for hardware security," *arXiv preprint arXiv:2208.08554*, 2022.