Reproduction of GNNUnlock for VLSI Topics

Erin Cold

***Abstract*—Logic locking is a promising design-for-trust technique to protect against IP (Intellectual Property) infringement and unlicensed IC (Integrated Circuit) production. SFLL-HD is a *corrupt and correct* style of logic locking based on Hamming Distance. Although SFLL-HD has been broken before, a comprehensive unlocking technique has not yet been found that works for multiple locking schemes and is not limited by key size selection.**

To solve this need, Lilas Alrahis *et al.* developed a machine learning technique in GNNUnlock: Graph Neural Networks-based Oracle-less Unlocking Scheme for Provably Secure Logic Locking. They report class F1 scores greater than .95 for all circuits analyzed, with many circuits being classified with over 99% accuracy.

This paper reproduces the steps of GNNUnlock up to evaluating the GNN. I was able to reproduce the success of the original authors for netlists locked with SFLL-HD; however, the absence of cross compilation in GNNUnlock is an essential shortcoming of the attack. My reproduction uses the same GNN parameters, training time, and circuit datasets. However, it differs in the cell library used and code to extract node features.

***Index Terms*—reproduction, hardware security, logic locking, machine learning, graph neural network.**

I. Introduction

AS IC feature sizes reduce and the complexity of IC manufacturing increases, there are increasing upfront costs to build a IC foundry. More and more design houses are becoming "fab-less" – where they outsource the actual IC manufacturing process to contracted foundries. However, this exposes the circuit design to untrusted parties, who could insert hardware Trojans, or violate the design patent by overproducing chips to sell or using the IC design itself.

This fundamental lack of trust has spurred the development of hardware security and locking locking techniques, a set of techniques that obfuscate the internal circuit's behavior and produce incorrect output for any chip without a key. This key can only be generated by parties privy to private design information – the IC design house or holders of that chip's production license.

II. Motivation

GNNUnlock was proposed to be a general solution to defeat multiple different locking schemes with different parameters **without using an oracle** - a functioning unlocked chip or simulation that can be queried to find the correct output for a given input. Lilas Alrahis *et al.* summarize the capabilities of current oracle-less attacks[1] in Figure 1.

By developing attacks on logic locking, we can improve our defenses and further the field of hardware security.

E. Cold is a Computer Engineering student at the University of Cincinnati, Ohio, USA. e-mail: colden@mail.uc.edu

Attacks	Different Circuit Formats	Different Locking Schemes	Different Parameter Settings
SPS [13]	✗	✗	✓
RE-based [14]	✗	✗	✓
FALL [5]	✗	✗	✗
SFLL-HD-Unlocked [4]	✗	✗	✗
GNNUnlock	✓	✓	✓

Fig. 1. Oracle-less Attack Capabilities

III. Background

A. SFLL-HD^h

SFLL (Stripped Functionality Logic Locking) is a technique based on the *corrupt and correct* principle. The HD postfix means that the locking modules are based on Hamming Distance. K bits of the PI (primary inputs) are protected. A perturb module introduces an error into the circuit whenever those K bits have an Hamming Distance h away from a secret key.

The restore module introduces a correction whenever those K bits have an Hamming Distance h away from an user provided key (with K bits as well). The circuit will only function correctly for all input patterns when the user provided key matches the secret key. Otherwise there will be corrupted patterns that sneak past the restore module, and corrections that are applied when no corruption exists (thus functioning as an error).

Y_{fs}	IN	k0	k1	k2	k3	k4	k5	k6	k7
✔	0	✖	✔	✔	✔	✔	✔	✔	✔
✔	1	✔	✖	✔	✔	✔	✔	✔	✔
✔	2	✔	✔	✖	✔	✔	✔	✔	✔
✔	3	✔	✔	✔	✖	✔	✔	✔	✔
✔	4	✔	✔	✔	✔	✖	✔	✔	✔
✔	5	✔	✔	✔	✔	✔	✖	✔	✔
✖	6	✖	✖	✖	✖	✖	✖	✔	✖
✔	7	✔	✔	✔	✔	✔	✔	✔	✖

Fig. 2. Corrupt and Correct in SFLL-HD^0

In [6] Muhammad Yasin *et al.* show the *corrupt and correct* principle in Figure 2. The secret key is k6 and $h = 0$ in this example. As the figure shows, whenever the user provides a key is not k6, the input pattern equal to k6 is corrupted. This is because a Hamming Distance of 0 is the same as equality. For each key value tried, it introduces an error for that input pattern as well. The restore logic overlaps with the perturb logic only when k6 is provided. For this simple example it is easy to brute force the correct key, but the complexity increases exponentially with key size[6].

B. GNNUnlock

GNNUnlock converts a netlist to a undirected graph by mapping each gate to a node and each wire between gates to an edge between nodes.

GNNUnlock then trains a GNN to learn the structure of the perturb and restore unit. With this learned model, we can classify a node from a yet-unseen graph as a perturb, restore or original circuit node. After post-processing to further improve accuracy, all the perturb and restore nodes can be removed to obtain the original circuit again. The module and graph representation of a locked circuit is shown in Figure 3.

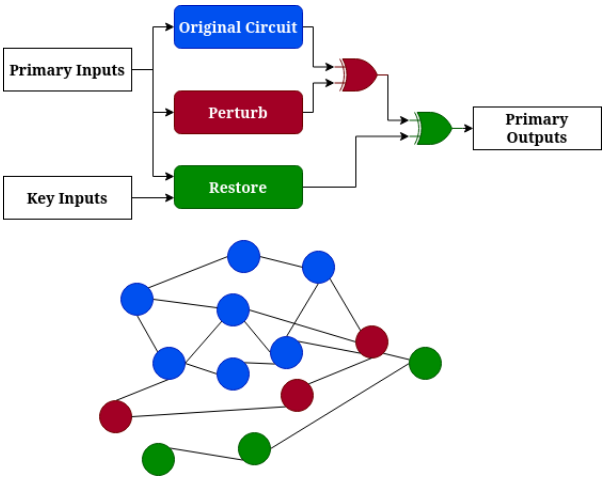

Fig. 3. SFLL Module and Graph Representation

This assumes that each gate in the design has only one class, **which means cross compiling will invalidate this assumption**.

IV. Experimental Setup

All data and code can be found on my GitHub repository page: https://github.com/cynthi8/GNNUnlock [5].

Figure 4 shows the top level flow of information during this GNNUnlock reproduction. I start with ISCAS-85 and ITC'99 benchmarks in the *.BENCH* format. I then convert these to Verilog to be locked with SFLL-HD at the RTL. These netlists are then synthesized back to the gate level using Synopsys Design Compiler™.

From the synthesized netlists, I extract the graph structure, feature vectors, and node class. This is then fed to the GraphSAINT along with the role of each graph (*train*, *validation*, *test*). Because GraphSAINT trains by sampling edges in the adjacency matrix, model performance is significantly improved by adding self-loops to nodes connected to primary outputs. This ensures that a node only connected to PIs and POs will still have an edge to be sampled and trained on.

After the GNN is finished training, I evaluate its performance on the never-seen-before *test* graphs and generate a confusion matrix and F1 scores for each class.

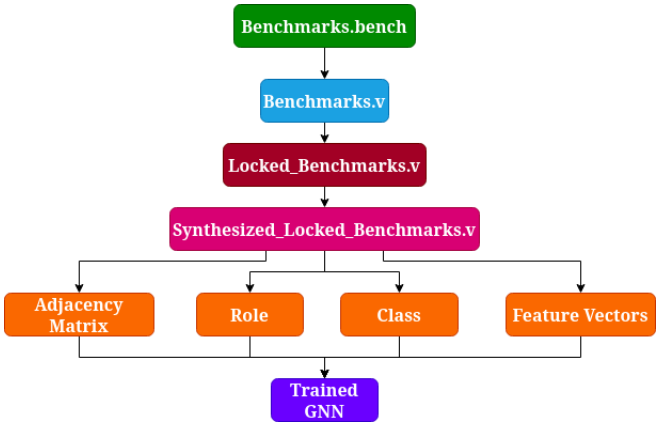

Fig. 4. Top Level Information Flow

A. Requisites

All work is done within an anaconda environment to isolate dependencies and ensure reputability. Key dependencies are $python >= 3.6.8$, $tensorflow >= 1.12.0$, $networkx >= 2.5.0$, and $GraphSAINT$.

The anaconda environment can be setup through the use of the `setup_conda_env.sh` script.

The $GraphSAINT$ repository can be cloned and built through the use of the `setup_GraphSAINT.sh` script.

B. Benchmark Files

The ISCAS-85 benchmarks were taken from http://web.eecs.umich.edu/~jhayes/iscas.restore/benchmark.html. Four ISCAS-85 benchmarks were used: *c2670, c3540, c5315, c7552*.

ITC'99 benchmarks were taken from https://github.com/squillero/itc99-poli. The _C means that it has been unrolled to a combinational circuit such that any DFFs are converted to primary inputs and primary outputs.

Six ITC'99 benchmarks were used: *b14_C, b15_C, b17_C, b20_C, b21_C, b22_C*.

C. Convert Benchmarks to Verilog

The locking code uses only works with netlists in the Verilog format, so the benchmarks need to be converted to Verilog. This is done by parsing the *.BENCH* into an intermediate graph representation. This graph is then written to Verilog using generic gates. An example of this conversion with a simple AND tree is shown going from Figure 5 to Figure 6.

Inferred wires are created, and signal names are renamed to conform to Verilog standards.

D. Lock with SFLL-HD

The benchmarks are then locked at the RTL (register transfer level) using SFLL-HD^2. A perturb and restore module are appended onto the original design. An example of a perturb block is shown in Figure 7.

```
INPUT(1)
INPUT(2)
INPUT(3)
INPUT(4)
INPUT(5)
INPUT(6)
INPUT(7)
INPUT(8)

OUTPUT(9)

10 = AND(1, 2)
11 = AND(3, 4)
12 = AND(5, 6)
13 = AND(7, 8)
14 = AND(10, 11)
15 = AND(12, 13)
9 = AND(14, 15)
```

Fig. 5. AND Tree in .BENCH format

```
// Main module
module and_tree(N1, N2, N3, N4, N5, N6, N7, N8, N9);

  input N1, N2, N3, N4, N5, N6, N7, N8;
  output N9;
  wire N10, N11, N12, N13, N14, N15;

  and ginst1 (N10, N1, N2);
  and ginst2 (N11, N3, N4);
  and ginst3 (N12, N5, N6);
  and ginst4 (N13, N7, N8);
  and ginst5 (N14, N10, N11);
  and ginst6 (N15, N12, N13);
  and ginst7 (N9, N14, N15);

endmodule
```

Fig. 6. AND Tree in Verilog format

```
/*************** Perturb block ***************/
module Perturb (N309, N86, N16, N114, N54, N116, N131, N43, perturb_signal);

  input N309, N86, N16, N114, N54, N116, N131, N43;
  output perturb_signal;
  //SatHard key=11110100
  wire [7:0] sat_res_inputs;
  wire [7:0] keyvalue;
  assign sat_res_inputs[7:0] = {N309, N86, N16, N114, N54, N116, N131, N43};
  assign keyvalue[7:0] = 8'b11110100;

  integer ham_dist_peturb, idx;
  wire [7:0] diff;
  assign diff = sat_res_inputs ^ keyvalue;

  always@* begin
    ham_dist_peturb = 0;
    for(idx=0; idx<8; idx=idx+1)
      ham_dist_peturb = $signed($unsigned(ham_dist_peturb) + diff[idx]);
  end

  assign perturb_signal =  (ham_dist_peturb==2) ? 'b1 : 'b0;

endmodule
/*************** Perturb block ***************/
```

Fig. 7. SFLL-HD^2 (K = 8) Perturb Block for c2670

E. Synthesis

The locked netlists are then synthesized back to a gate level netlist using Synopsys Design Compiler™. The *saed90nm* cell library is used for compilation. **This is a departure from the original GNNUnlock experiment**, which uses the *65nm LPe*, and *Nangate 45nm open cell* technology libraries. However, this is not expected to make a difference because the cell types are similar. There are 18 cell types for *65nm LPe* and 16 for *saed90nm*. GNNUnlock claims to be technology library independent, so this should not break GNNUnlock from working.

To allow *test* nodes to be evaluated, **the locking modules were not cross-compiled**. I reached out to the authors of GNNUnlock and confirmed this was how they collected their data.

The compile script is shown in Figure 8 where the hierarchy is flattened **after compilation**.

```
read_file -format verilog $input_file
compile_ultra -no_autoungroup
ungroup -all -flatten
write_file -format verilog -output $output_file
exit
```

Fig. 8. Design Compiler™ Compile Script

F. Graph Conversion and Feature Extraction

GraphSAINT requires a specific format of data to train on. The full description is shown in Table I.

File	Description
adj_full.npz	A scipy sparse adjacency matrix containing all edges
adj_train.npz	A scipy sparse adjacency matrix only containing edges between training nodes
role.json	A dictionary mapping each node to a role (train, validation, test)
class_map.json	A dictionary mapping each node to a class (main, perturb, restore)
feats.npy	A numpy matrix containing the feature vector for each node

TABLE I
GRAPHSAINT INPUT FILES

The *convert_netlists_to_graph.py* script takes in a directory with roles assigned to each netlist and outputs the required files. Each netlist is converted into a graph and the adjacency matrixes are concatenated into a very large block diagonal sparse matrix. Gates with multiple outputs (i.e. FADDX1) are still considered a as single node.

The class information is extracted from the gate instance name as Design Compiler™ concatenates the original module name to flattened gates.

To collect node features, *convert_netlists_to_graph.py* polls the two-hop neighborhood for each node and tallies up all the surrounding gate types. A gate type is the general type of the gate - independent of operand count and drive strength. For example, NAND2X2 and NAND3X1 are both considered NAND gate types. The number of primary inputs, key inputs, key outputs, as well as the connectivity (in and out degree) is also calculated and included as features. In total, there are 21 different features.

G. GNN Parameters and Training

The GNN was trained using the same parameters as GNNUnlock as described in *DATE21.yml*, the number of epochs was also set to 2000 which was the same number used in GNNUnlock.

H. Cross Compiled Synthesis

To confirm my hypothesis that a cross-compiled netlist could not be trained or tested upon, I flattened the design hierarchy before compiling to result in parallel datasets to the non-cross-compiled versions. The script used is shown in Figure 9.

```
read_file -format verilog $input_file
ungroup -all -flatten
compile_ultra -no_autoungroup
write_file -format verilog -output $output_file
exit
```

Fig. 9. Design Compiler™ Compile Script with Cross Compiling

As expected, Synopsys Design Compiler™ removes any trace of where the gate came from when it is compiled. Because there are no labels to be extracted, I was unable to evaluate the GNN performance on cross-compiled *test* netlists, even if the GNN was trained on non-cross-compiled *train* netlists.

1) Label Deduction: Although Design Compiler™ does not tell us the node labels, I thought there might be a way to deduce the label by comparing the non-cross-compiled and cross-compiled netlists directly. If this worked, we could evaluate GNN performance on cross-compiled *test* circuits.

However, I found that even in the best case scenario, the number of gates that changed would likely make it infeasible to try to piece back together where each gate came from. As a case study, I looked at *c2670_SFLL_HD_2_8_0.v*. This was the smallest circuit locked with SFLL-HD^2 and a key size of 8.

The perturb module was implemented in 11 gates, the restore module was implemented in 21 gates, and the original circuit was implemented in 355 gates. After cross-compiling, at least 50 gates changed their type. For example, there were 32 MUX21X1 gates in the non-cross-compiled circuit and 41 in the cross-compiled circuit so at least 9 new MUX21X1 gates were introduced.

This analysis also showed that logic locking gates were being combined with original design nodes because the gate counts of certain gates changed. 5 FADDX1 gates are used in the logic locking circuitry in the non-cross-compiled netlist, and nowhere else. In the cross-compiled netlist, there are only 4 FADDX1 gates. **The functionality of the 5th FADDX1 gate must have been absorbed into different parts of the circuit**.

V. Experimental Results

I chose to reproduce the SFLL-HD^2 dataset used in the original authors GitHub page [4]. This setup uses 24 netlists of c2670 and c5315 as training graphs, 9 netlists of c3540 as validation graphs, and 12 netlists of c7552 as testing graphs for a total of 45 netlists. Each netlist was locked with various key sizes and different key values. The GNN performance on the test graphs is shown in Table II and III.

		Predicted		
		main	**perturb**	**restore**
True	**main**	8521	1	16
	perturb	22	654	0
	restore	0	0	888

TABLE II
CONFUSION MATRIX FOR SFLL-HD^2 AFTER 2000 EPOCHS

F1 Scores		
main	perturb	restore
0.998	.983	0.991

TABLE III
F1 SCORES FOR SFLL-HD^2 AFTER 2000 EPOCHS

The GNN performs slightly worse on classifying perturb nodes, which is expected as the GNN cannot use Key Inputs as a determining feature like for restore nodes.

However, as a whole, these results support those reported in GNNUnlock. GNNUnlock reports a F1 score of 99.93% for the perturb node classes using c7552 as the test set as shown in Figure 10. My score of 98.3% is lower, but comparable to the F1 scores using different test sets.

Test Set	#Test Graphs	GNN Acc. (%)	F1-Score (%)			#Misclassified Nodes	Removal Success (%)
			RN	PN	DN		
c2670	12	99.53	99.77	97.84	99.73	24 DN as PN 4 PN as RN	100
c3540	9	99.79	99.15	98.63	99.94	5 PN as RN 3 DN as PN 2 DN as RN	100
c5315	12	100	100	100	100	–	100
c7552	12	99.99	100	99.93	99.99	1 PN as DN	100
b14_C	9	99.97	99.94	99.72	99.99	2 PN as RN 5 PN as DN	100
b15_C	9	99.99	99.97	99.84	99.99	3 PN as DN 1 PN as RN	100
b20_C	9	99.98	99.73%	99.52%	99.99%	9 PN as RN 2 PN as DN	100
b21_C	9	100	100	100	100	–	100
b22_C	9	99.96	99.97	98.74	99.98	1 PN as RN 3 PN as DN 27 DN as PN	100
b17_C	9	99.94	99.73	97.52	99.98	3 DN as RN 57 DN as PN 6 PN as RN	100

Fig. 10. Reported Results of GNNUnlock on SFLL-HD^2

VI. Future Directions

Machine learning has the opportunity to improve logic locking attacks and defenses and SFLL-HD appears to be somewhat vulnerable against a direct removal attack like GNNUnlock.

Although I was unable to deduce gate labels for cross-compiled circuits, an approach that works with Design Compiler™ to insert traces on each gate and find where they get compiled to could allow node-classification-based machine learning approaches to train and test on realistic logic locking implementations.

Further research into how EDA (Electronic Design Automation) tools implement logic locking schemes could open up new pathways for hardware security attacks and defenses.

VII. Conclusion

My independent replication of GNNUnlock found the same success as the original authors. The classifier performance was comparable to the values previously reported.

However, the lack of cross-compilation in the *test* circuits invalidates the use of this technique for realistic logic locking implementations. In a practical use case, I conclude that node classification machine learning attacks on SFLL-HD do not have the ability required to effectively remove the locking circuitry.

References

[1] Alrahis, Lilas and Patnaik, Satwik and Khalid, Faiq and Hanif, Muhammad Abdullah and Saleh, Hani and Shafique, Muhammad and Sinanoglu, Ozgur, *GNNUnlock: Graph Neural Networks-based Oracle-less Unlocking Scheme for Provably Secure Logic Locking*, IEEE/ACM Design, Automation and Test in Europe Conference, 780–785, 2021.

[2] Fangfei Yang, Ming Tang, and Ozgur Sinanoglu, *Stripped Functionality Logic Locking with Hamming Distance Based Restore Unit (SFLL-hd) – Unlocked*, 2019

[3] D. Sirone and P. Subramanyan, *Functional Analysis Attacks on Logic Locking*, 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2019, pp. 936-939, doi: 10.23919/DATE.2019.8715163.

[4] Alrahis, Lilas and Patnaik, Satwik and Khalid, Faiq and Hanif, Muhammad Abdullah and Saleh, Hani and Shafique, Muhammad and Sinanoglu, Ozgur, *GNNUnlock*, https://github.com/DfX-NYUAD/GNNUnlock.

[5] Erin Cold, *GNNUnlock Reproduction* https://github.com/cynthi8/GNNUnlock.

[6] Muhammad Yasin, Abhrajit Sengupta, Mohammed Thari Nabeel, Mohammed Ashraf, Jeyavijayan (JV) Rajendran, and Ozgur Sinanoglu. 2017. *Provably-Secure Logic Locking: From Theory To Practice*. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17). Association for Computing Machinery, New York, NY, USA, 1601–1618. DOI:https://doi.org/10.1145/3133956.3133985