Graph Neural Network based Netlist Operator Detection under Circuit Rewriting

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

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

ABSTRACT

Recently graph neural networks (GNN) have shown promise in detecting operators (multiplication, addition, comparison, etc.) and their boundaries in gate-level digital circuit netlists. Unlike formal approaches such as NPN Boolean matching, GNN-based methods are structural and statistical. This means that making structural changes to the circuit while maintaining its functionality may negatively impact their accuracy. In this paper, we explore this question. We show that indeed the prediction accuracy of GNN-based operator detection does fall following simple circuit rewriting. This means that custom rewrites may be a way to hamper operator detection in applications such as logic obfuscation where such undetectability is a security goal. We then present ways to improve the accuracy of prediction under such transforms by combining functional/semi-canonical information into the training and evaluation of the ML model.

CCS CONCEPTS

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

KEYWORDS

graph convolutional network, circuit rewriting, circuit reverse engineering, hardware security

ACM Reference Format:
Guangwei Zhao and Kaveh Shamsi. 2022. Graph Neural Network based Netlist Operator Detection under Circuit Rewriting. In *Proceedings of the Great Lakes Symposium on VLSI 2022 (GLSVLSI '22), June 6–8, 2022, Irvine, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3526241.3530330

1 INTRODUCTION

Modern real-world digital designs are generated by a multi-stage process akin to software compilation. A description of the functionality of the design is described first in a hardware-description-language (HDL) programming language. This HDL code is then typically translated into a data-flow-graph (DFG) with operators (e.g. addition, multiplication, Boolean, etc.) and operands as nodes, and edges as connections. Each operator is then mapped to a specific gate-level circuit structure. The resulting gate-level netlist can be further optimized during which different operator logic may mix with each other until a final gate-level netlist is produced.

In this paper, we focus on the problem of recovering the operations in the original HDL/RTL, i.e. the original DFG, from a gate-level netlist. This problem has important applications. The first is in "reverse-engineering for trust". Here, an end-user is given a gate-level netlist design by an untrusted party that may include malicious off-spec functionality, i.e. a hardware Trojan. Recovering a DFG for such a gate-level netlist can help assist the end-user in verifying that the design is non-malicious.

Second, understanding the limits of operator detection can help in assessing the security of circuit obfuscation schemes. Circuit obfuscation schemes are techniques that aim to deliberately obscure the structure or functionality of the circuit which in turn can hamper successful malicious tampering of the design. If operator detection is possible in an obfuscated netlist, then the obfuscation scheme is leaking important information about the functionality of the design and maybe insecure.

Formal techniques such as equivalence checking or NPN matching [7] can be used to provably find a given operator in a netlist. However, their complexity is super-polynomial as they require NP-hard problem solving such as SAT and QBF. Hence, heuristic and statistical techniques have been used as a less accurate but faster alternative. Recent work [2] has shown how graph neural networks (GNNs) trained on labeled netlist examples can detect common operators with high accuracy (above 80%).

Since the GNN in this approach is only given the graph structure and node gate types, there is the question of whether it can accurately classify functionally-equivalent yet structurally-different subcircuits. In this paper, we study this question. Specifically:

- We generate a dataset of operator circuits subject to rewriting, i.e. replacing circuit subgraphs with functionally equivalent alternatives. We demonstrate how this rewriting can drastically diminish the accuracy of GNN-based operator detection.
- We in response add a set of functionality-aware features to the model and show how this improves the detection accuracy of the model. We also demonstrate how training on synthetic rewritten data can improve accuracy.

The paper is organized as follows: Section 2 presents preliminaries and background. Section 3 presents our technical approach to the studying the issue. Section 4 presents experimental results and Section 5 concludes the paper.

GLSVLSI '22, June 6–8, 2022, Irvine, CA, USA

ACM ISBN 978-1-4503-9322-5/22/06…$15.00
https://doi.org/10.1145/3526241.3530330

2 PRELIMINARIES

Graph Neural Network (GNN). Traditional neural networks receive vector/matrix data. Hence they cannot directly operate on compact representations of graphs. Graph Neural Networks (GNNs) are models built for this task from the ground up.

A Graph Convolutional Network (GCN) [6] is one of the most prominent GNN structures. The layer performs the following operation on node encodings $H^{(l)}$ updating them to $H^{(l+1)}$:

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

Here $\tilde{A} = A+I_n$ is the adjacency matrix of the graph A plus I_n self-connections. $\tilde{D}$ is the degree matrix of the graph, W is the layer's weights, with σ as the activation function of the layer. Therefore, the previous node encodings are updated via combining them with weights and information from their neighbors through the matrix A. This corresponds to passing data around in the graph, without the need for operating on the explicit full-rank adjacency matrix A. L GCN layers correspond to each node sensing information from L hops away in the graph. After the L rounds, one can use the node encodings $H^{(l)}$ for node-level regression/classification. Graph-level prediction can be done by aggregating the final node encodings $H^{(l)}$ using various aggregation techniques.

GNN-RE: GNN-based Reverse Engineering (Operator Detection). As first introduced in [2], one can take a set of RTL examples with known operations, synthesize them down to the gate-level netlist while maintaining what operator each gate belongs to. Then the operator membership of each gate is used as a node-level label that a GNN will be trained to predict. On datasets consisting of several operations (multiplication, addition, multiplexing, etc.) GNNs were shown to achieve higher than 80% accuracy.

In GNN-RE, the circuit is encoded as a graph with gates as nodes and connections between the gates as edges. The initial node encodings in GNN-RE consist of several fields. These fields include the count of gates with specific functionalities in the h-hop neighborhood of the node. So the initial encoding in GNN-RE already includes some form of neighborhood information. The GraphSAGE library was used in GNN-RE which samples fixed-length walks of the graph rather than applying a GCN directly. The authors argued that this would assist in reducing the runtime complexity, by allowing the user to train on a sampled subset of the graphs in the circuits. Since the goal here is classification, a *softmax* layer at the output of the model was used to classify each node to one of the available operator categories.

3 OPERATOR-DETECTION UNDER CIRCUIT REWRITING

3.1 Circuit Rewriting

Circuit rewrites are used extensively in logic synthesis and DFG optimization. Consider the fact that $a + b - a = b$. This means that if we search the DFG of a program/circuit and are able to find an expression matching $a + b - a$, we can simply replace it with b. Such a rewrite will reduce the complexity/size of the graph. The same can be done for Boolean logic circuits. ABC for instance, as one of the fastest yet most effective logic minimization tools, uses an And-Inverter-Graph (AIG) rewriting algorithm [8]. Here the circuit is first converted to an AIG. Then the algorithm attempts to replace 4-input subcircuits in the AIG with a set of pre-compiled optimal alternative structures and see which replacement produces the most area/delay savings.

It is possible to generalize and formalize the notion of rewriting for Boolean circuits. Here, given a circuit $C(x) = y$ with primary inputs x and outputs y consisting of gates g_i that can take on functions from a basis B (i.e. $B = \{\text{AND/OR/NAND/NOR/BUF}\}$), a rewriting procedure R is a (probabilistic) algorithm that alters the structure of C while maintaining its functionality. i.e. $C' \leftarrow R(C)$ where $\forall x \in X,\ C(x) = C'(x)$.

3.2 Theoretical Limits

Consider a synthesis flow in which the DFG of the HDL is converted to a gate-level netlist by a one-for-one simple replacement of the operators in the DFG with cells from a pre-compiled public circuit library. Here a reverse-engineer with knowledge of the library can in theory recover the precise DFG. In the worst-case, the reverse-engineer may use subgraph-isomorphism (SGI), which is an NP-complete algorithm, against the circuit library. A trained GNN while not as complete as the SGI approach, will learn the library structures over time and hopefully predict the right operators.

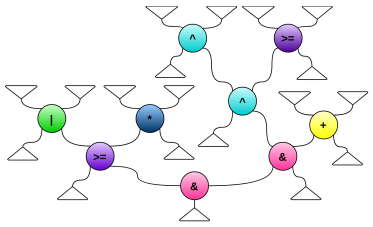
Both the SGI and GNN approach however can fail as soon as even simple rewriting is applied. Consider the case of a bit-wise XOR operation applied to two 8-bit words $y = a \oplus b$. One can randomly pick 4 of the bits in the operation, replace the XOR gate with an XNOR gate plus an inverter at one of its inputs. SGI will no longer match this as an 8-bit XOR. Instead, it may detect it as a 4-bit XOR and a 4-bit XNOR. If the inverter on one of the bits is pushed into neighboring logic it can hamper SGI's matching of that logic as well. A GNN that is trained on recognizing 8-bit XORs, may run into issues here too. Although the statistical nature can make it more robust than the SGI approach.

If an equivalence-checking (EQ) approach is used instead, it may be able to detect that the functionality of the circuit has been preserved despite the rewritings introduced. The broader question of whether rewriting can in an information-theoretic manner hide the original DFG remains. One can envision cases where a DFG is rewritten in a manner where no algorithm can successfully distinguish between alternative choices of original DFGs. Finding a rewriting procedure R that can with provable security hide maximally that which is "hide-able" in the circuit in the first place will be an indistinguishability-obfuscator iO. Developing iO has been a holy grail in cryptography for more than a decade [5]. iO with low-overhead has been quite elusive. Hence, there are some practical limits to how much rewriting/structural-obfuscation can hide high-level functionality.

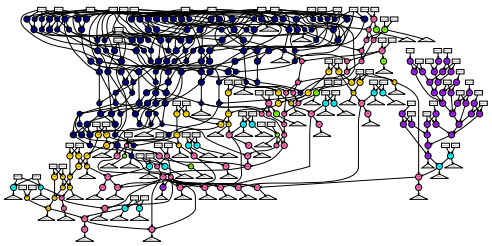
In this paper, we show that practical simple rewriting does hurt the accuracy of GNN-based prediction. Secondly, we show that as intuited here incorporating features that capture functionality rather than mere structure do indeed improve model accuracy in the presence of rewrites.

3.3 Dataset Generation

Operator-DAGs. In order to train and evaluate our models in this paper, we need training datasets of circuits. The circuit gates need to be labeled according to the operator that they belong to. In [2] authors created several handcrafted HDL circuits and synthesized and



(a) The behavior-level operator DAG



(b) The gate-level operator DAG

Figure 1: An example of a random generated operator DAG with 10 inputs and 9 operations. The operators (&:and, ∧:xor, |:or, >=:comparison, +: add, ∗: multiply) are colored in the DAG the same color as gates that belong to them in the netlist.

labeled their gate-level netlists. In our work, we automatically generate HDL (Verilog) designs. These are built via a DAG construction algorithm.

The algorithm is as follows. The user selects the number of input words x_i and their width $|x_i|$. A list $L = \{x_0, ..., x_n\}$ is constructed first that includes these words. At random two elements of L are picked, and a binary operator is applied between them. The operators are picked at random from a set of possible operators with user-specified selection likelihood weights. We use the following in our experiments: {multiplication:0.3, addition:1, xor:1, or:1, and:1, comparison:1}.

Each time an operator is applied its resulting word is added back into the list L. To ensure the connectedness of the inputs, intermediate DAG nodes are not picked until all input words have been picked at least once. The user can specify how many nodes to add. In our experiments, we stop once $n - 1$ operators have been created with n being the number of input words. The resulting HDL is then synthesized to gate-level.

In order to label the gate-level design we first use a directive to keep the word wires alive during the synthesis process. Then, given a binary operator $y = op_t(x_i, x_j)$, starting from the wires in y, backward BFS exploration until we reach x_i and x_j, will help identify all the gates that belong to the operator op_t.

An example of an operator DAG and its gate-level synthesized and labeled netlist can be seen in Fig. 1.

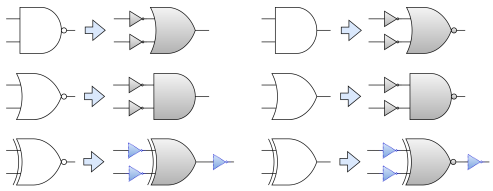
Rewriting. As for rewriting, while the space of possible rewrite transforms is prohibitively large, in this paper, we use a simple set of rewrite rules revolving around inverter introduction and absorption. The rules operate on individual gates rather than k-cuts as done in AIG-rewriting. We leave the exploration of more complex rewriting rules to future work.

Our first category of rewrite rules consists of DeMorgan's law applied to AND/NAND or OR/NOR gates. Per Fig.2a an AND gate may be converted to a NOR gate with two inverters introduced at its inputs: i.e. $a + b = \overline{(\bar{a} + \bar{b})}$. A similar rule can be applied to OR/NOR gates. The second rewrite rule is for XOR/XNOR gates. Here, one can take an XOR gate and convert it to an XNOR gate with a single inverter added to one of its inputs. i.e. $a \oplus b = \overline{(\bar{a} \oplus b)}$, or convert an XNOR to an XOR.

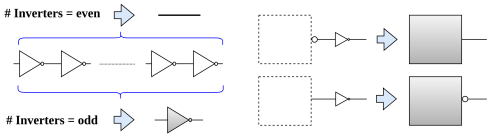
While the above rules primarily introduce inverters into the design, our third rule category tries to absorb them. Here we replace an even number of back-to-back inverters with none or absorb an inverter that follows a gate into the gate (e.g. changing AND+NOT to NAND) per Fig. 2b.

The above rules are applied by first randomly selecting a subset of the circuit gates. This subset is sized as an Rp (rewrite percentage) portion of the gates in the circuit. The first and second rules (inverter introduction) are applied to these gates. Then, the third rule (inverter absorption) is applied to the entire circuit eating up extra inverters.

Since we do not outsource the rewriting to ABC or another synthesis tool we are able to keep track of the node (gate) labels through the rewriting process. An example rewriting of an operator DAG with different Rp values can be seen in Fig. 3.



(a) Rewrite rule one/two (Inverter introduction)



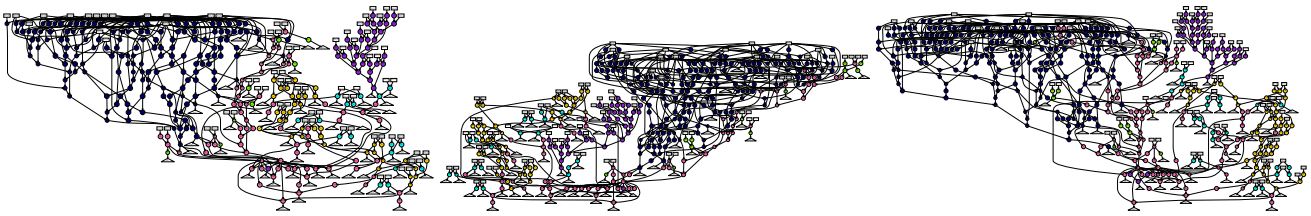
(b) Rewrite rule three (Inverter Absorption)

Figure 2: Simple Rewriting Rules

3.4 Functional Features

As we will demonstrate in Section 4, the above simple rewrite rules can reduce the GNN's accuracy significantly. In order to improve upon the model's accuracy in the face of rewriting we experiment with adding functional features to the model in this paper. We assume the baseline node feature to simply be (a one-hot encoding of) the functionality of the individual gate itself. We denote this node encoding/feature with *gf*. We may append to this baseline vector additional features as discussed here.

GNN-RE features. We adopt a set of features from GNN-RE [2] which we denote with *gnnre*. This includes a 12-dimensional vector. The first two fields capture the number of PIs and POs that the gate is immediately connected to. The last two fields capture the input and output degree of the gate in the circuit graph. The intervening 8 fields document the number of different functions/cell-types in the 1-hop neighborhood of the gate (given that we have 8 different functions/cells in our cell library). The *gnnre* features described above are still considered structural features as they do not directly equate to functionality.



(a) 10% rewritten (b) 20% rewritten (c) 40% rewritten

Figure 3: An operator DAG of 10 inputs and 9 operations rewritten with multiple percentages (*Rp*s).

Truth Table. The truth table is an explicit representation of the functionality of a circuit. Hence, it may serve as a candidate functional feature. Note that for an individual gate, its functionality as defined by the *gf* feature is simply a representation of its truth table. So a truth table here would be useful only if it is capturing a larger subcircuit. In this paper, we experiment with taking the truth table of multiple subcircuits rooted at a target node as *tt* features.

This is done by first performing a k-cut-enumeration on the target gate g_i. A k-cut of g_i is a connected subcircuit rooted at g_i with precisely k inputs (k-input logic cone). k-cut-enumeration is a procedure that finds all the k-cuts rooted at g_i with $k \leq s$ with s as a user-given parameter. This is done by recursively constructing new cuts from the existing cut set by replacing a given cut's ith input with the gate that is connected to it, and stopping the process once the s size limit is reached. The *tt* feature consists of the truth table of the smallest 3 cuts in the cut-enumeration with $s = 4$. The table is extracted by simulating the cut for every possible input pattern, with the nodes being ordered according to their node ID in the graph. The truth table is represented as an integer that captures the index of the given truth table in a set of all seen truth tables in the current circuit database. One can alternatively use the binary truth table as an integer or as a binary vector.

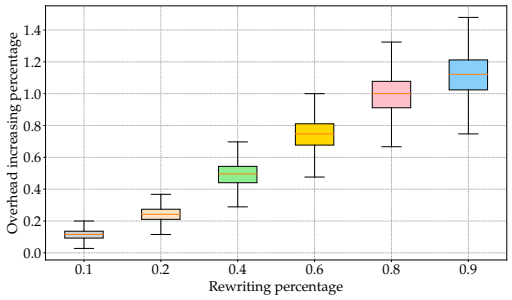


Figure 4: Overhead statistics on each rewriting percentage. The orange line in each box represents the average value.

NPN Class. The set of all circuits that can be made functionally equivalent to one another by permuting or negating their inputs and outputs belong to the same Negation-Permutation-Negation (NPN)-class. For 4-input single-output circuits, there are $2^{2^4} = 65536$ different truth tables, but only 255 NPN classes. NPN classes are used in AIG-rewriting [8] to replace 4-input structures with alternative ones. NPN-matching is the problem of finding an NPN-equivalent subcircuit in a larger circuit and has been used traditionally for operator detection [10].

Truth table features are sensitive to the ordering of the input wires. However, the NPN class of a cut is not. Hence we experiment with including NPN classes in the node feature vector. The *npn* features in our experiments consist of the NPN class of the smallest 3 cuts in the cut set rooted at gate g_i. The classes are represented simply by their index in the set of all seen NPN classes in the dataset which is extracted beforehand using the `testnpn` ABC command. This command given the set of all cut truth tables seen in the dataset will find their NPN class. A map is pre-compiled from truth tables to NPN classes which is then used during feature construction.

Signal Probability. Interestingly our most successful feature appeared to be signal probability (SP). This is defined simply as the probability of a given net being 1. In a given circuit the signal probability of internal wires depends on the signal probability of the inputs themselves. One typically assumes the primary inputs to be independent random variables with 0.5 signal probabilities. Given this, computing the precise signal probability of an internal wire is a #P-complete problem which is believed to be harder than NP-complete problems [3]. However, it is possible to estimate with somewhat good accuracy the signal probability of Boolean circuit nets by simply simulating the circuit for a large number of input patterns, or by assuming that the input nodes for each gate are independent variables (not true for convergent paths) and propagating input probabilities to the output. e.g. the probability of the output of an AND gate $y = AND(a, b)$ would be calculated as $sp(y) = sp(a) \times sp(b)$.

In our work, we incorporate signal probability into the feature vector in different ways. These are denoted as 1) *gsp*: Global SP uses the probability of an internal node g_i as seen from the PIs, 2) *csp*: Cut SP consists of the signal probability of g_i as seen from the smallest 3 non-trivial cuts of the gate, and 3) *lsp*: Local SP consisting of the signal probability of g_i as obtained by backward BFS chunk of the circuit starting from node g_i.

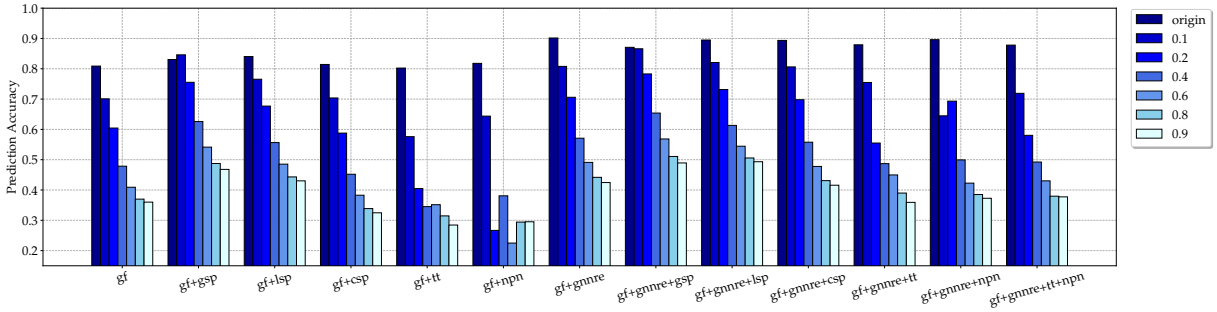
Note that *tt* and *npn* features are inexpensive to acquire when the cut size is small ≤ 4, but as it increases, the input space of a cut to obtain a truth table or NPN class becomes unmanageable. The global or local signal probability features do not have this issue. In addition, the input ordering sensitivity that *tt* features suffer from is not an issue here since the signal probability of all inputs is assumed to be 0.5 for the circuit or cut. The probability propagation also takes less time than truth table or NPN-class extraction. Signal probability features perform surprisingly well in our experiments.

4 EXPERIMENTS

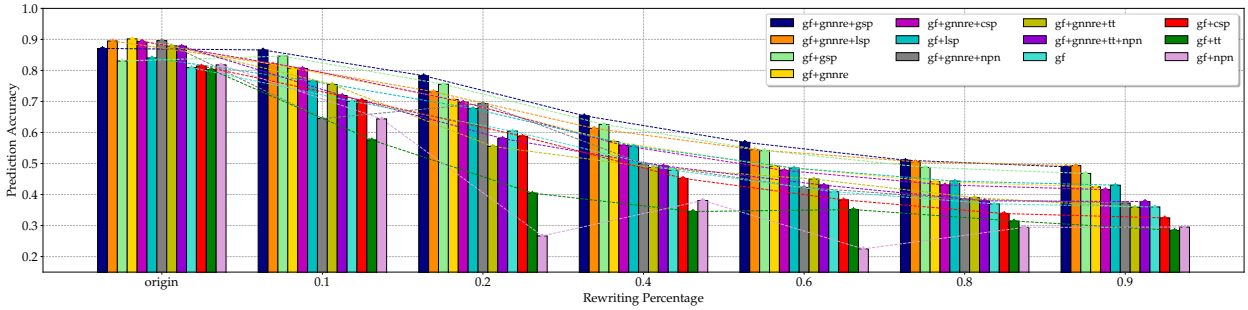
Experiments were run on an AMD 5900 CPU with 24 threads, 32GB of memory, running Ubuntu 20.04, with an RTX 3070 TI GPU. We use `spektral` [4] for graph learning and implement our rewriting and feature collection in Python using the `pyneos` framework [9].

Method / Rewrite	gf	gf+gsp	gf+tt	gf+lsp	gf+csp	gf+npn	gf+gnnre	gf+gnnre+gsp	gf+gnnre+tt	gf+gnnre+lsp	gf+gnnre+csp	gf+gnnre+npn	gf+gnnre+tt+npn
origin	0.8089	0.8305	0.8024	0.8403	0.8143	0.8179	0.9016	0.8708	0.8791	0.8949	0.8936	0.8960	0.8781
10%	0.7008	0.8460	0.5760	0.7655	0.7037	0.6438	0.8079	0.8660	0.7548	0.8207	0.8062	0.6448	0.7190
20%	0.6045	0.7554	0.4047	0.6769	0.5878	0.2667	0.7060	0.7830	0.5550	0.7315	0.6979	0.6932	0.5804
40%	0.4786	0.6258	0.3452	0.5565	0.4520	0.3808	0.5709	0.6539	0.4870	0.6132	0.5577	0.4993	0.4923
60%	0.4091	0.5415	0.3515	0.4853	0.3828	0.2249	0.4911	0.5684	0.4497	0.5445	0.4778	0.4227	0.4302
80%	0.3699	0.4875	0.3146	0.4432	0.3387	0.2941	0.4418	0.5107	0.3899	0.5054	0.4309	0.3851	0.3796
90%	0.3600	0.4680	0.2844	0.4302	0.3248	0.2952	0.4246	0.4893	0.3592	0.4933	0.4158	0.3727	0.3774

Table 1: Prediction accuracy on original dataset and rewriting datasets



(a) Prediction accuracy trend under features' combinations. Legends in the rightmost represent the rewriting percentage.



(b) Prediction accuracy rankings under a different rewriting percentage

Figure 5: Prediction accuracy on original dataset and rewriting datasets

Rewriting %	Avg Overhead %	Best Pred Acc	Unlearn/Overhead
10%	0.1129	0.8660	0.3153
20%	0.2402	0.7830	0.4938
40%	0.4940	0.6539	0.5014
60%	0.7465	0.5684	0.4463
80%	0.9999	0.5107	0.3909
90%	1.1253	0.4933	0.3628

Table 2: Unlearnability per overhead for different rewriting percentage

Dataset Generation. We use the operator-DAG generation procedure discussed in Section 3 to generate a pristine/original dataset. In our original dataset, there are 2590 operator-DAGs, with input word counts ranging from 4 to 16 and input bit widths ranging from 2 to 8.

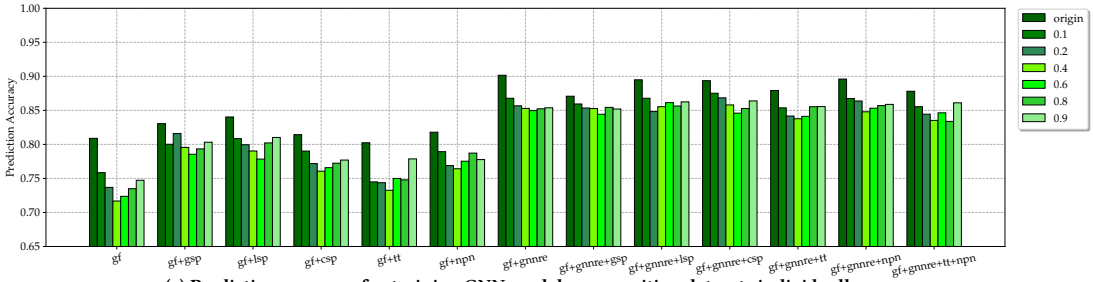
We use `Yosys` [**?**] to create gate-level netlists from the generated Verilog DAGs. We use the "keep" directive in Yosys to keep word wires alive and label node operators by scanning the logic between words as described in Section 3.

We generate rewritten datasets by taking the 2590 DAGs in the original dataset and rewriting each with a number of different rewriting percentages (*Rp*): 10%, 20%, 40%, 60%, 80%, and 90% (see 3.3). The area overhead as measured by gate count inflation can be seen in Fig. 4.

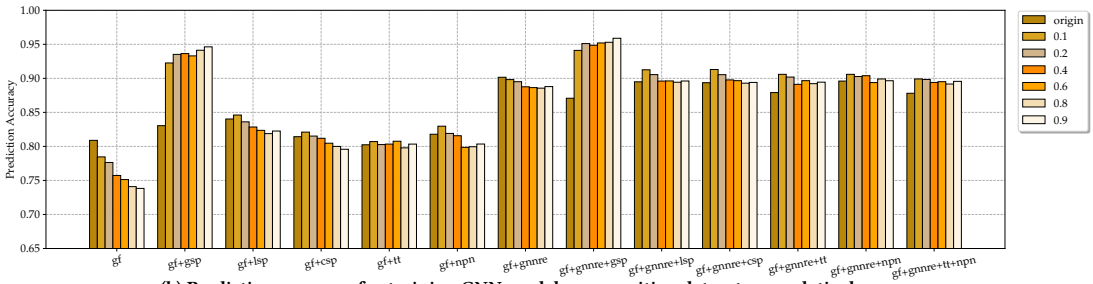
GNN Model. We use a GCN with 3 layers as our GNN with the first layer having 256 units and the other two having 128 units followed by a *softmax* layer with 6 classes for our operators. We trained the models with the Adam optimizer for 300 epochs and a batch size of 20. We use 20/80 test/train cross-validation. Accuracy is reported as the number of correct node label guesses divided by all guesses. This means that for our 6-class classification random guessing will achieve a 16.6% accuracy. A model that performs worse than this threshold obviously has no predictive power.

Training on Pristine and Evaluating on Rewritten Circuits. We first trained the GNN model on the pristine dataset and used the trained model to predict the rewritten datasets. In this scenario, the model will not have encountered any rewritten circuits. The prediction accuracy of different node feature encodings under this approach is reported in Figures 5a, 5b, and Table 1.

First, we note that the inclusion of *gnnre* features dramatically improves the model accuracy. This suggests that authors in [2] made a clearly advantageous choice in engineering these features as compared to just individual gate functions (*gf*). With respect to rewriting, as can be seen from this data, the prediction accuracy drops significantly as more nodes in the circuit are rewritten regardless of the feature encoding scheme. However, if we compare *gf* (purely structural) to *gf+(g/l)sp* (structural + functional) we see that the signal probability feature improves the prediction accuracy against rewritten circuits despite not having seen them during



(a) Prediction accuracy for training GNN models on rewriting datasets individually



(b) Prediction accuracy for training GNN models on rewriting datasets cumulatively

Figure 6: Train and Predictions on rewriting datasets. Legends in the rightmost represent the rewriting percentage.

training at all. This confirms that functional features increase the resilience of the model against rewriting.

To our surprise, cut-based features, such as *tt*, *npn*, or *csp* performed quite poorly. This could be due to the fact that the order of the cuts in the feature vector is somewhat arbitrary. This can hurt the performance of neural networks that expect fixed data ordering. We aim to explore this further in our future work.

Training and Predicting on Rewritten Circuits. We then performed both training and prediction on the datasets that include rewritten circuits. Fig. 6a shows accuracy for when the model is trained and evaluated on individual datasets (pristine, 10% rewritten, 20% rewritten, ...). The U-shape profile here suggests that when the model is shown rewritten circuits, it is hardest for it to accurately predict those circuits that are rewritten with mid *Rp*s. i.e. if every gate in the circuit is rewritten with the same rule, the model may perform better than when half of the gates have been rewritten. This could be because half-rewriting may produce more irregular circuits than either no-rewriting or full-rewriting. Table 2 also suggests that rewriting half of the gates may produce optimal overhead/unlearnability ratios. Unlearnability here is measured as the distance between the best prediction accuracy for that overhead and the pristine prediction accuracy.

Fig. 6b shows models trained on accumulated data, e.g. on {pristine+10%+20%} datasets. These produce the highest accuracy values showing the effectiveness of training on synthetic rewritten circuits. Also, it can be seen that purely structural features like *gf* even when shown rewritten circuits during training still perform poorly as the rewriting level increases. This is as opposed to functional features such as *sp* where training on more rewritten circuits seems to improve their accuracy.

5 CONCLUSION

In this paper, we showed how simple circuit rewriting can diminish the accuracy of GNN-based operator detection in gate-level netlists. We demonstrated then how including functional features or training on synthetic rewritten datasets can overcome some of this. We aim to explore more complex rewriting rules and machine learning techniques in our future work.

REFERENCES

[1]]yosys [n. d.]. Yosys verilog synthesis framework. https://github.com/YosysHQ/yosys.

[2] Lilas Alrahis, Abhrajit Sengupta, Johann Knechtel, Satwik Patnaik, Hani Saleh, Baker Mohammad, Mahmoud Al-Qutayri, and Ozgur Sinanoglu. 2021. GNN-RE: Graph Neural Networks for Reverse Engineering of Gate-Level Netlists. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021).

[3] S Ercolani, M Favalli, M Damiani, P Olivo, and B Ricco. 1989. Estimate of signal probability in combinational logic networks. In *Proceedings of the 1st European test conference*. IEEE Computer Society, 132–133.

[4] Daniele Grattarola and Cesare Alippi. 2021. Graph neural networks in tensorflow and keras with spektral [application notes]. *IEEE Computational Intelligence Magazine* 16, 1 (2021), 99–106.

[5] Máté Horváth and Levente Buttyán. 2020. *Cryptographic Obfuscation: A Survey*. Springer.

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

[7] Wenchao Li, A. Gascon, P. Subramanyan, Wei Yang Tan, A. Tiwari, S. Malik, N. Shankar, and S.A. Seshia. 2013. WordRev: Finding word-level structures in a sea of bit-level gates. In *Proc. IEEE Int. Symp. on Hardware Oriented Security and Trust*. 67–74.

[8] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. 2006. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *Design Automation Conference, 2006 43rd ACM/IEEE*. IEEE, 532–535.

[9] Kaveh Shamsi, Meng Li, David Z Pan, and Yier Jin. 2019. KC2: Key-Condition Crunching for Fast Sequential Circuit Deobfuscation. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 534–539.

[10] Pramod Subramanyan, Nestan Tsiskaridze, Wenchao Li, Adria Gascón, Wei Yang Tan, Ashish Tiwari, Natarajan Shankar, Sanjit A Seshia, and Sharad Malik. 2013. Reverse engineering digital circuits using structural and functional analyses. *IEEE Transactions on Emerging Topics in Computing* 2, 1 (2013), 63–80.