SAIL: Analyzing Structural Artifacts of Logic Locking Using Machine Learning

Prabuddha Chakraborty, *Graduate Student Member, IEEE*, Jonathan Cruz, Abdulrahman Alaql, and Swarup Bhunia, *Senior Member, IEEE*

***Abstract*— Obfuscation or Logic locking (LL) is a technique for protecting hardware intellectual property (IP) blocks against diverse threats, including IP theft, reverse engineering, and malicious modifications. State-of-the-art locking techniques primarily focus on securing a design from unauthorized usage by disabling correct functionality – they often do not directly address hiding design intent through structural transformations. They rely on the synthesis tool to introduce structural changes. We observe that this process is insufficient as the resulting changes in circuit topology are: (1) local and (2) predictable. In this paper, we analyze the structural transformations introduced by LL and introduce a potential attack, called SAIL, that can exploit structural artifacts introduced by LL. SAIL uses machine learning (ML) guided structural recovery that exposes a critical vulnerability in these techniques. Through this attack, we demonstrate that the gate-level structure of a locked design can be retrieved in most parts through a systematic set of steps. The proposed attack is applicable to most forms of logic locking, and significantly more powerful than existing attacks, e.g., SAT-based attacks, since it does not require the availability of golden functional responses (e.g., an unlocked IC). Evaluation on benchmark circuits shows that we can recover an average of about 92%, up to 97%, transformations (Top-10 R-Metric) introduced by logic locking. We show that this attack is scalable, flexible, and versatile. Additionally, to evaluate the SAIL attack resilience of a locked design, we present the SIVA-Metric that is fast in terms of computation speed and does not require any training. We also propose possible mitigation steps for incorporating SAIL resilience into a locked design.**

***Index Terms*— Hardware obfuscation, logic locking, hardware security, cybersecurity, machine learning.**

I. Introduction

THE use of pre-verified hardware intellectual property (IP) in system-on-chip (SoC) designs has become a prevalent practice in the semiconductor industry. However, the global economic trend that dictates a horizontal business model incorporates many untrusted parties in the modern chip design flow. In particular, most chip designers rely on potentially untrusted 3rd party fabrication facilities. Such a trend diminishes a chip designer's control on the IPs and makes them vulnerable to various forms of attacks, including piracy, reverse-engineering, overproduction, and malicious modifications, i.e., Trojan attacks, leading to serious economic and security threats [1], [2]. To address these security issues, hardware obfuscation or logic locking techniques have been actively studied for the past decade [3]–[5]. Fig. 1, summarizes the two main goals for logic locking: 1) preventing black-box usage, and 2) hiding design intent. This method aims at transforming a design - both functionally and structurally - based on a key, such that the locked design functions correctly only if the right key inputs are provided. Basic functional locking techniques such as XOR gate insertion locking scheme [6]–[8], can prevent black-box usage of the IP/IC. However, an attacker may still reverse engineer the netlist to extract the intent of a design or identify key-gates for a removal attack. In order to address this attack vector, judicious structural transformations need to be introduced in a design. Current locking approaches primarily rely on: 1) insertion of various modification cells (e.g., XOR, XNOR gates) controlled by key bits, and 2) re-synthesis of the design after locking, to obtain any structural changes in the design.

Like any security solutions, IP protection achieved through logic locking largely depends on its robustness against possible attacks. Over the years, several attacks on logic locking have been proposed, which broadly fall into two categories: 1) *functional attacks*, such as SAT attack [9], and 2) *structural attacks*, such as ANTI-SAT block removal attack [10]. However, most reported attacks on logic locking have been functional attacks. The only structural attack [10], is actually not designed to deobfuscate an IP, but to facilitate a subsequent SAT (i.e., functional) attack [9].

Based on a thorough statistical analysis, we make two key observations regarding the technique to introduce structural changes in traditional logic locking: 1) the changes are sparse and local to the inserted gates, and 2) the changes are very deterministic (i.e., follow a set of well-known logic synthesis rules). Motivated by these observations, we introduce a new paradigm of attack on logic locking, called **SAIL** (Structural Analysis using Machine Learning), that exposes a critical vulnerability in logic locking approaches. This attack can retrieve the original design before re-synthesis and hence, the design intent, through structural analysis guided by machine learning. SAIL extracts each logic locking gate locality (subgraph representation of locking and surrounding gates) from a locked design and reverts them to their pre-synthesis state using the reconstruction models trained on [Pre-Synthesis, Post-Synthesis] locality pairs from the training dataset. From the information on the pre-synthesized locked netlist, an attacker

Manuscript received February 2, 2021; revised May 7, 2021; accepted June 14, 2021. Date of publication July 9, 2021; date of current version August 3, 2021. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Ulrich Rührmair. *(Corresponding author: Prabuddha Chakraborty.)*

Prabuddha Chakraborty, Jonathan Cruz, and Swarup Bhunia are with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32603 USA (e-mail: p.chakraborty@ufl.edu).

Abdulrahman Alaql is with the Communication and Information Technology Research Institute, King Abdulaziz City for Science and Technology (KACST), Riyadh 12354, Saudi Arabia.

Digital Object Identifier 10.1109/TIFS.2021.3096028

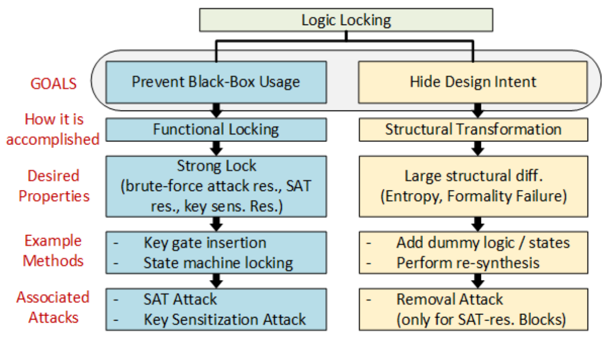

Logic Locking
GOALS
Prevent Black-Box Usage
Hide Design Intent
How it is accomplished
Functional Locking
Structural Transformation
Desired Properties
Strong Lock (brute-force attack res., SAT res., key sens. Res.)
Large structural diff. (Entropy, Formality Failure)
Example Methods
- Key gate insertion
- State machine locking
- Add dummy logic / states
- Perform re-synthesis
Associated Attacks
- SAT Attack
- Key Sensitization Attack
- Removal Attack (only for SAT-res. Blocks)

Fig. 1. Overview of logic locking objectives for IP protection.

can more easily carry out key guessing and reverse engineering attacks by observing the structure [11].

SAIL is a more salient attack than SAT because unlike SAT, SAIL: (1) does not require golden responses from an unlocked IC, which may be difficult to obtain, (2) can be applied to both combinational and sequential designs, (3) works for all digital design types and underlying boolean function (including functions for which SAT fail such as, multiplier), and (4) scalable with respect to key and benchmark size. We propose two variants of SAIL:

- *SAIL-NN:* Uses an ensemble of Neural Network models for reconstructing post-synthesis localities into pre-synthesis localities. It is also equipped with a change prediction model which eliminates the need to reconstruct localities that are predicted to not have been changed.
- *SAIL-RD:* Uses an ensemble of rule dictionaries for reverting post-synthesis localities back to their pre-synthesis state.

We quantitatively and qualitatively analyze the effectiveness of both variants of the SAIL attack using the systematic framework and tool that we have developed. We propose a novel metric called R-Metric which can be used to quantify the total recovery from using SAIL. From our results, we demonstrate that the SAIL attack is very effective against different logic locking techniques and key sizes. We achieve an average of 92%, and up to 97% (Top-10 R-Metric) recovery of the locking structures for a set of benchmark circuits. We also investigate different methods for providing SAIL resilience to locked designs. Exhibiting synthesis-induced changes with near-uniform probability is one of the best ways to defend against SAIL and we design SIVA (**S**tructural **SI**gnature **V**ulnerability **A**nalysis) to capture the degree to which this property is exhibited in a logic locked design. We also prove that the SIVA-Metric is the upper bound of SAIL accuracy for a given locked design.

In particular, we make the following key contributions:

- We analyze, both quantitatively and qualitatively, the nature of structural changes in a gate-level netlist introduced due to different steps of logic locking. We note that the structural changes introduced by logic locking techniques are limited in terms of topology and predictable in nature.
- Based on these observations, we formulate a novel attack paradigm that exploits the limited structural changes and predictable nature of logic locking techniques.
- We propose a complete framework comprising of two different SAIL attack variants with a systematic set of steps. *SAIL-NN* uses (1) a *Change Prediction Model* that can predict whether a key gate locality had undergone changes due to logic locking; and (2) a *Reconstruction Model* that can locally revert the structural changes introduced due to re-synthesis. *SAIL-RD* learns an ensemble of rule dictionaries to revert post-synthesis localities back to their pre-synthesis state.
- We present a comprehensive evaluation of both the proposed SAIL attack variants on a suite of benchmark circuits locked with several logic locking techniques and demonstrate that it is scalable in terms of both effectiveness and speed with respect to the key length, different logic locking techniques, locality size and design size.
- We introduce the R-Metric, a measurement to help understand the effectiveness of SAIL.
- We quantify the fundamental property that provides SAIL resilience for a locked design. This metric, SIVA-Metric, is fast to compute, requires no training and mathematically proven to be the upper bound of SAIL accuracy.
- We investigate different threat models where the attacker does not have knowledge of logic locking technique, synthesis tool, and settings.
- We also investigate different approaches that can potentially lead to SAIL resilience.

The rest of the paper is organized as follows: Section II discusses different state-of-the-art hardware obfuscation and locking techniques and attacks. Section III motivates the work by establishing that logic locking induced changes are local and predictable. Section IV provides the methodologies of the SAIL attack variants. Section V provides quantitative analysis of the proposed attack. Section VI presents the SIVA metric, investigates SAIL efficacy on different threat models, potential SAIL defense tactics, and runtime analysis of SAIL variants. We conclude the paper in Section VII.

II. Background & Related Works

A. Logic Locking & Hardware Obfuscation

Designers use logic locking to prevent the correct functionality of a design by inserting gates that act as a lock and are controlled by key inputs. Correct functional behavior in the design is returned upon entering the correct key or key sequence at the key inputs as shown in Fig. 2. In general, there are two types of logic locking: combinational [8] and sequential [12]. For combinational logic locking, combinational gates (XOR/XNOR, AND, etc.) are inserted into a design with one input serving as a key-bit. To lock a design with an N-bit key at least N combinational key-gates must be added. On the other hand, sequential logic locking modifies the finite state machine by adding key-driven states and state transitions. These transitions are properly traversed by applying a particular pattern of keys at the key inputs which subsequently unlocks the design.

1) Combinational Logic Locking: The focus of this paper is on combinational logic locking [13]. In this logic locking scheme [6]–[8], a design is locked to hide functionality with an N-bit key by inserting appropriate key gates (e.g., XOR

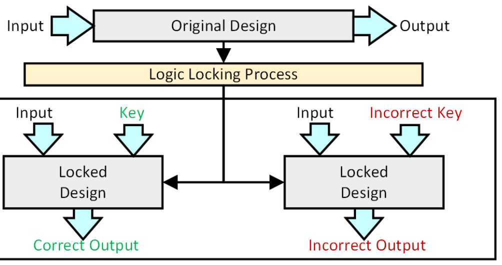


Fig. 2. Overview of the logic locking process. A locked design will only function as intended if the correct key is applied.

for key-bit '0' and XOR followed by an inverter for key-bit '1'). At the most basic implementation, locations of these logic locking gates are randomly selected. But in order to deal with different logic locking attacks, more informed location determination techniques have been proposed [14], [15]. The key gates inserted range from simple XOR/XNOR gates to more complex constructs for dealing with specific attacks [16], [17].

2) Logic Locking Techniques to Throttle Specific Attacks: To deal with SAT-Attack [9] (a SAT-Solver guided key extraction attack, discussed in details later), defenses such as Anti-SAT and SARLock were proposed [16]–[18]. A unique type of locking called, Cyclic obfuscation, was also introduced that inserts combinational loops in the design in order to throttle SAT-Attack [19]. TTlock [20] focuses on modifying the design logic cone to defend against key-sensitization [15], SAT [9] and removal attacks [10]. To defend against key-sensitization attack, Strong Logic Locking was introduced where key gates are inserted to have complex interference [15]. The interference is achieved through the creation of an interference graph where the nodes are key gates and they are connected if the gates interfere. The focus of this type of logic locking is to maximize the number of non-mutable gates in the interference graph. In [21], the authors demonstrate that creating overlaps in the logic cone can effectively counter attacks which isolate and deobfuscate each logic cone separately via brute force techniques. Stripped-Functionality Logic Locking (SFLL) introduced in [14] utilizes a functional-stripping module to lock the function of the IP. This module is followed by a functional restoring unit that restores the circuit's functionality when the correct key is applied.

3) Beyond Traditional Functional Locking: Obfuscation can be carried out at the physical circuit level to prevent doping concentrations or dielectric manipulations [22]. Camouflaging techniques are also proposed to prevent reverse engineering attacks [23]. Delay locking adds an extra layer of security by not only allowing the key to modify the functionality but also the timing profile of the circuit [24]. Error-tolerant logic locking has also been introduced that are designed to provide graceful degradation of output quality as the hamming distance between the correct and applied key increases [25].

4) End-to-End Frameworks and Metrics: End-to-end logic locking frameworks such as MIRAGE has also been introduced which, based on the design, (1) selects the locking technique, (2) locks the design and (3) evaluates the strength of the logic locking [26]. Recently, a learning guided obfuscation framework called LeGO was also introduced which attempts to provide joint resistance against multiple attacks [27]. Metrics to determine the effectiveness of obfuscation/logic-locking has also been proposed [28].

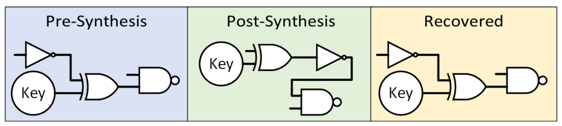
B. Logic Locking Attacks

Over the years, several logic locking attacks have been proposed which target one or more locking techniques. These attacks can be broadly classified into three groups: 1) functional attacks, 2) structural attacks, and 3) joint functional-structural attacks.

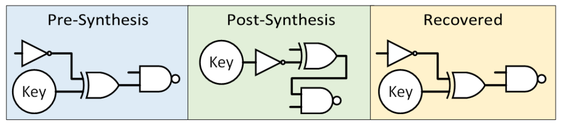
1) Functional Attacks: Most of the existing attacks on logic locking try to retrieve the key, based on functional analysis of the design. These are generally referred to as functional attacks. A key sensitization attack tries to retrieve the key-bits by applying a pattern of non-key inputs such that a key-input is mapped to an observable output (sensitization) [15]. In [9], a SAT-based algorithm is proposed to completely or partially retrieve the key from a combinational design. It requires the locked netlist and an unlocked IC/netlist to carry out the attack. Moreover, SAT formulation is an NP-Complete problem and the heuristic-based implementations are not guaranteed to give a solution within a reasonable time. Different protections [16]–[18] against SAT attack have been proposed that generally involve adding extra logic to increase either the number of iterations of the attack or the time to complete each iteration. A variant of SAT, AppSAT [29], has been shown to bypass typical SAT protections [16] by retrieving an approximate key which provides correct output for most of the input space. Double DIP, a SAT-based attack, was proposed to counter SARLock-enhanced locked design [30]. Cyclic obfuscation [19] was introduced to counter SAT attack but is vulnerable to targeted attacks such as Cycsat [31]. An ML-based attack on logic locking [32] was proposed that focuses on key retrieval, which also requires an unlocked IC, and is expected to suffer from scalability issues.

2) Structural Attacks: Structural attacks use analysis of the design's interconnects and topology to identify and disable locking gates or reverse engineer the design intent. To the best of our knowledge, the only pure netlist-level structural/removal attack on logic locking [10] is aimed at the removal of ANTI-SAT blocks [16] to facilitate a subsequent SAT attack. Contrary, to all the attacks we have mentioned, our work aims at quantitatively and qualitatively analyzing and evaluating the effect of logic locking on the structure of the gate-level netlist and the deterministic heuristics involved in structural hiding. Our attack does not require golden responses (a requirement for most of the attacks) and is highly scalable. Snapshot is an ML-based attack, inspired by SAIL, which predicts the key value for a given key gate locality [33].

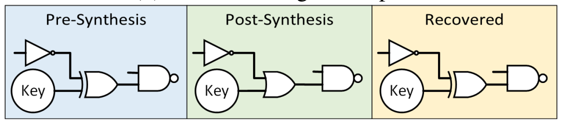
3) Joint Structural-Functional Attacks: Some attacks try to retrieve the key, based on a joint structural and functional analysis. SWEEP attack predicts the key by observers the design features before and after hard-coding a key bit and subsequent synthesis [34]. SURF attack [11], FALL attack [35], redundancy attack [36], TGA [37], and the attack in [38] leverage structural features of the netlist and functional-based analysis to facilitate functionally unlocking the design.



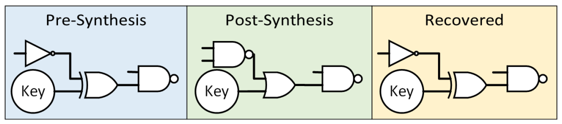
(a) Level-2 change example



(b) Level-3 change example 1



(c) Level-3 change example 2



(d) Level-3 change example 3

Fig. 3. Example structural changes due to synthesis after logic locking (blue→green) and SAIL recovery (green→yellow).

III. Motivation

In this section, we discuss two fundamental observations on logic locking induced structural changes and justify the use of machine learning models for structural analysis based attacks.

A. Structural Changes Due to Logic Locking Are Local

When a netlist is locked using a XOR-based locking scheme, a re-synthesis of the netlist is performed in an attempt to camouflage the key-gates that are inserted. In the best case, the key XOR-Gate itself will combine with local gates and transform into a new structure or cease to be connected to the Key Input. We refer to this as $Level-3$ change. In a slightly worse case, the XOR-Gate inserted due to locking may remain intact but its neighboring gates may change due to logic simplification. We call this a $Level-2$ change. Finally, the third scenario observed involves no change of the inserted locking gates or surrounding gates after re-synthesis, referred to as $Level-1$ changes. As a result, the locked design remains vulnerable to removal and key guessing attacks. Moreover, we observe that this process does not aid in hiding the design intent and fails to obfuscate the design from a reverse engineering standpoint.

In Table I, for 57231 gate localities (of size 3 gates), we note that 15.71% of key inserted localities do not go through any structural change. Another 79.30% of the localities only show Level-2 changes that are very easy to revert using certain rules which can be manually devised or learned through statistical methods. Only the remaining 4.97% of the localities are properly obfuscated. However, most of these changes are predictable and can be locally reverted as well with the help of statistically generated rules/algorithms.

In Fig. 3 (a)-(d), we show some of the most common Level-2 and Level-3 changes that we have observed during our experiments and the resulting localities after we perform the SAIL reconstruction. In Fig. 3 (a), we see the inverter moving from the output of the XOR gate to the input. This transformation is a very common scenario and can confuse an attacker into thinking the *Inverter* is not a locking gate. By observing the locality around the key-gate, our statistically learned rules can determine the state of the locality before synthesis and recover it, thereby removing confusion. In Fig. 3 (b), we observe a very common Level-3 change, where the inverter comes between the key input wire and the XOR gate. This change can also be easily recovered and stitched back into the design. Fig. 3 (c) and Fig. 3 (d) depict more complex Level-3 changes but our models are still able to locally recover from such changes.

TABLE I

Observed Pre Post Synthesis Transformations for Locked Designs Across Different IPs. About 79% Changes Are Local (Level-2) and More Than 15% Localities Have Not Changed at All (Level-1) Due to Resynthesis After LL

	Level - 1	**Level - 2**	**Level - 3**
C1355	26	334	0
C1908	62	292	6
C2670	96	245	19
C3540	283	1124	33
C5315	750	1950	180
C6288	516	2247	117
C7552	481	2257	142
ALU	3404	18570	1057
FIR	3376	18368	1296
Total	**8994 (15.71%)**	**45387 (79.30%)**	**2850 (4.97%)**

B. Synthesis Induced Transformations Are Deterministic

In the previous section, we have established that the changes induced by logic locking gates are local and limited. We can go one step further and try to recover, from these limited changes, a local snapshot of the key-gate inserted region. Recovering a small locality around the locking gate connected to the key input wire is enough to obtain insight into the logic locking that was carried out.

Given the designs before and after synthesis, we enumerate each unique transformation (unique [Input Locality, Output Locality] pair) carried out by the synthesis tool. We observe that very few rules are used by the synthesis tool for carrying out most of the transformations. In Fig. 4, we observe that 50 rules make up about 30% of the transformations. With 300 rules, the synthesis tool does 60.56% of the transformations and only 1000 rules govern 81% of the changes. If we can statistically learn these limited number of rules then we can revert the changes introduced by them. This fact is the main reason why a machine learning reversion attack is possible. Fig. 4 statistics are based on observing 57,231 different localities (of size 3 gates) across multiple iterations of logic locking over the 7 largest ISCAS-85 benchmarks, ALU and FIR. In Fig. 5, we also see a similar trend for a bigger locality size of 5 gates.

The observation that the number of rules or types of change introduced by logic locking is limited alone allows a learning process to generate the required rules for reversion given enough statistical data. This claim is further corroborated by the quantitative recovery results in Section V. In the next section, we will describe our recovery attack models.

IV. Methodology of SAIL Attack

Earlier we have introduced the SAIL attack and presented initial observations [39]. In this paper, we build on the fundamental principles presented in [39] to enhance the potency of the attack, explore possible variants, perform a comprehensive study on its effectiveness and scalability, and finally present potential countermeasures.

We design SAIL to detect and quantify the structural vulnerabilities that exist in current logic locking techniques. Specifically, we aim to reverse the structural changes introduced in a design due to the synthesis step after key-gate insertion. A successful reversal of these changes can make the design vulnerable to reverse engineering attacks and expose the key gates to subsequent removal attacks. The SAIL framework learns the deterministic nature of the synthesis tool for a given logic locking technique and uses the knowledge to expose a fundamental weakness in target designs locked with this same technique. In this work, we propose two separate methods which can be used to realize SAIL. As can be seen in Fig. 6, the main difference between these two methods lies in the learning/knowledge-acquisition component. Next, we will describe these two methods in greater details.

A. Training Dataset Generation

Dataset generation is a common step for both SAIL methods. The dataset must be created to capture the target design's bias as well as the logic locking technique's behaviour.

As we do not have access to the original pre-locked netlists, we take the locked circuit provided to the attacker and treat it as a *Pseudo Golden Circuit* and carry out one more round of logic locking to create the training set. This way, we can capture the circuit-specific information (design's bias) eliminating the need for a *Golden Circuit* and also learn the specific logic locking technique's behaviour. It is important to note that the gates inserted due to the original locking are a minor fraction of the total number of gates in the circuit and hence do not have much of an impact on the subsequent locking. We term this method *Pseudo Self-Referencing*.

B. SAIL Neural Network

The SAIL Neural Network (SAIL-NN) is one of the proposed methods for realizing the SAIL framework. In SAIL-NN (Fig. 6 (a)), we train an ensemble of neural network models for reconstructing the key gate sub-graphs (localities) of a locked design from their post-synthesis to their pre-synthesis state. It may happen that some of these localities have not changed due to the synthesis process and therefore do not require any reconstruction. Hence, we propose a screening technique which comprises of a machine learning model (random forest for our experiments) that can predict if a locality has undergone any change due to synthesis. Avoiding unnecessary reconstruction improves both run-time performance and accuracy. The SAIL-NN method is depicted using the algorithms Algo. 1, Algo. 2, Algo. 3, and Algo. 4.

Algo. 1 is the main procedure for SAIL-NN and starts off by generating vectorized representations of the localities in

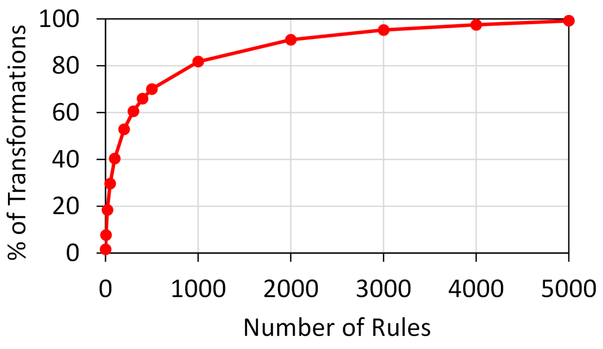


Fig. 4. Observed transformations for pre and post synthesis pairs (locality size 3) over 57,231 samples.

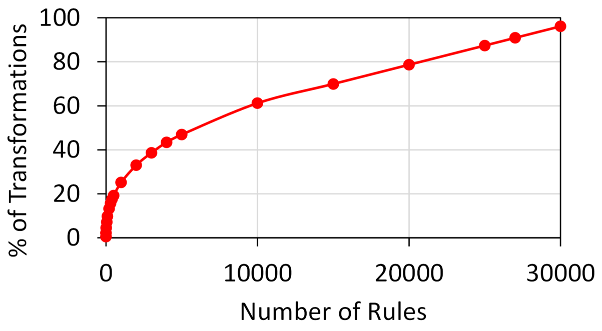


Fig. 5. Observed transformations for pre and post synthesis pairs (locality size 5) over 57,231 samples.

the training dataset, D, in line 2 using Algo. 2. A locality can be translated as a vector in many different ways. For our implementation of SAIL, we represent the connectivity of the gates in the locality in form of an adjacency matrix and for each gate in the locality, the gate-type is expressed using one-hot encoding. The flattened adjacency matrix with the gate-type vectors appended forms the complete locality vector. In Fig. 7, we demonstrate with an example how a locality can be represented in vector form. The size of a locality is denoted by the number of gates which comprise it. Training a machine learning model requires a set of input samples and corresponding expected output (labels). For the given task, input samples are post-synthesis netlist localities of different sizes around selected anchor points (nets) and the expected output (labels) are corresponding pre-synthesis netlist localities of a selected size around the same anchor points (gates). The gates in the locality are determined using a breadth-first search algorithm with a limit on the number of nodes to restrict the locality size. L_i is the list of locality sizes that are used to generate the vectorized input samples ($locI$). The L_o is the locality size of the corresponding vectorized output/labels ($locO$). $locChange$ is a binary indicator for each locality denoting whether they changed during the synthesis or not. The change prediction model used for predicting whether a given locality changed or not is trained using $locI$, $locChange$, and P (line 3 of Algo. 1). P is a set of user-defined machine learning model training parameters. In line 4, the reconstruction machine learning models (for reverting a post-synthesis locality to its pre-synthesis state) are trained with $locI$, $locO$, and P using Algo. 3. The localities of the target design (T) that are to be analyzed/attacked using SAIL are vectorized and stored in T_Locs using Algo. 4 (line 5). In lines 7-17 of Algo. 1,

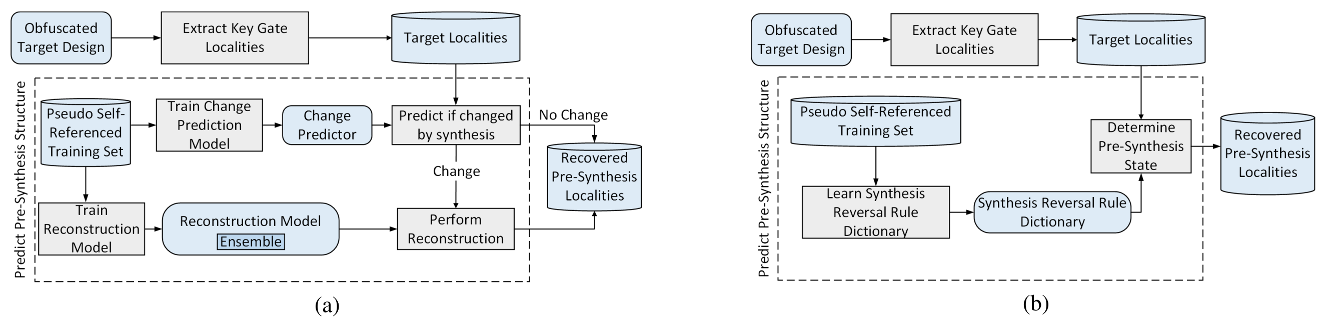
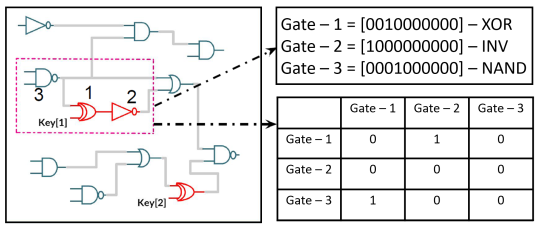

Obfuscated Target Design
Extract Key Gate Localities
Target Localities
Predict Pre-Synthesis Structure
Pseudo Self-Referenced Training Set
Train Change Prediction Model
Change Predictor
Predict if changed by synthesis
No Change
Change
Recovered Pre-Synthesis Localities
Train Reconstruction Model
Reconstruction Model Ensemble
Perform Reconstruction
(a)
Obfuscated Target Design
Extract Key Gate Localities
Target Localities
Predict Pre-Synthesis Structure
Pseudo Self-Referenced Training Set
Learn Synthesis Reversal Rule Dictionary
Synthesis Reversal Rule Dictionary
Determine Pre-Synthesis State
Recovered Pre-Synthesis Localities
(b)

Fig. 6. (a) Overview of the SAIL neural network framework. (b) Overview of the SAIL rule dictionary framework.



Gate – 1 = [0010000000] – XOR
Gate – 2 = [1000000000] – INV
Gate – 3 = [0001000000] – NAND

	Gate – 1	Gate – 2	Gate – 3
Gate – 1	0	1	0
Gate – 2	0	0	0
Gate – 3	1	0	0

Fig. 7. Translation of a locality into its vector form.

Algorithm 1 SAIL-NN
```
1: procedure SAIL_NN($L_i, L_o, D, P, T$)
2:     $locI, locO, locChange = VECTORIZE\_DATASET$
         $(D, L_i, L_o)$
3:     $M_c = learnChangePredModel(locI, locChange, P)$
4:     $M_r = LEARN\_NN\_RECONST\_MODELS$
         $(locI, locO, P)$
5:     $T\_Locs = VECTORIZE\_TARGET\_NETLIST(T, L_i)$
6:     $recoveredSnapshots = \{\}$
7:     for each $l \in T\_Loc$ do
8:         $changeFlag = predictChange(l, M_c)$
9:         if $changeFlag == 1$ then
10:            $predictions = []$
11:            for each $m \in M_r$ do
12:                $p = predictLocality(l, m)$
13:                $predictions.append(p)$
14:            $r = ensembleVoting(predictions)$
15:        else
16:            $r = l[L_o]$
17:        $recoveredSnapshots[l] = r$
18:    return $recoveredSnapshots$
```

the recovered/predicted pre-synthesis localities are obtained and stored in $recoveredSnapshots$ which is returned from the procedure in line 18.

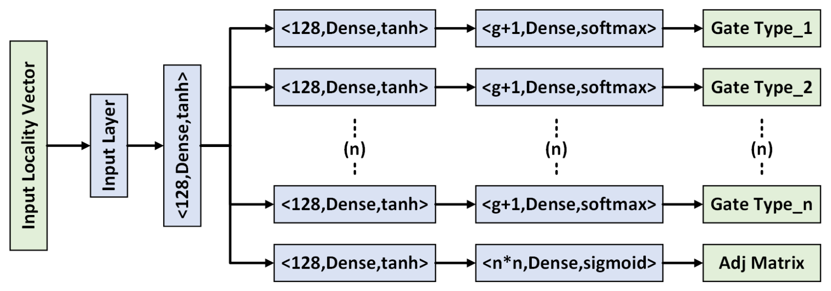
1) SAIL-NN Dataset Vectorization: The training dataset localities must be vectorized and organized before it can be used for training the machine learning models. Algo. 2 depicts this process. D is the training dataset consisting of pre-synthesis netlists ($D.pre$) and post-synthesis netlists ($D.post$). L_i is the list of locality sizes that are used to generate the vectorized input samples ($locI$). locI is initialized as an empty dictionary in line 2. For each $size$ in L_i, we extract the locality vectors (lines 3-8) and store them in locI for $key = size$. The locality vectors are extracted around the anchor points specified for each netlist in $D.post$ (line 6). In line 9, we initialize an empty list $locO$. The vectorized localities of size L_o for $D.pre$ are extracted and stored in $locO$. The locality vectors are extracted around the anchor points specified for each netlist in $D.pre$ (line 11).

Algorithm 2 Vectorize Dataset
```
1: procedure VECTORIZE_DATASET($D, L_i, L_o$)
2:     $locI = \{\}$
3:     for each $size \in L_i$ do
4:         $oneLocI = []$
5:         for each $netlist \in D.post$ do
6:             $locs = vectorizeLocalities$
                 $(netlist, size, netlist.anchors)$
7:             $oneLocI.append(locs)$
8:         $locI[size] = oneLocI$
9:     $locO = []$
10:    for each $netlist \in D.pre$ do
11:        $locs = vectorizeLocalities(netlist, L_o, netlist.anchors)$
12:        $locO.append(locs)$
13:    $index = 0$
14:    $locChange = []$
15:    while $index < len(locO)$ do
16:        $c = detectChange(locO[index], locI[L_o][index])$
17:        $locChange.append(c)$
18:        $index = index + 1$
19:    return $[locI, locO, locChange]$
```

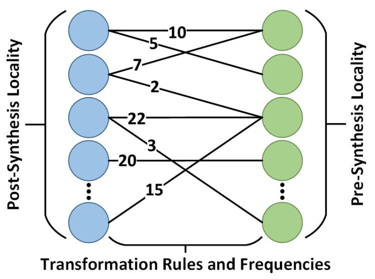
Algorithm 3 Learn NN Reconstruction Models
```
1: procedure LEARN_NN_RECONST_MODELS($locI, locO, P$)
2:     $M_r = []$
3:     for each $(size, oneLocI) \in locI$ do
4:         $index = 0$
5:         while $index < P.NNcount$ do
6:             $trainX, valX, trainY, valY =$
                 $randSplit(oneLocI, locO, P)$
7:             $m = trainNNModel(trainX, valX, trainY, valY)$
8:             $M_r.append(m)$
9:             $index = index + 1$
10:    return $M_r$
```

Algorithm 4 Vectorize Target Netlist
```
1: procedure VECTORIZE_TARGET_NETLIST($T, L_i$)
2:     $T\_Locs = \{\}$
3:     for each $anchor \in T.anchors$ do
4:         $one\_T\_Loc = \{\}$
5:         for each $size \in L_i$ do
6:             $loc\_Vec = vsectorizeLocality(T, size, anchor)$
7:             $one\_T\_Loc[size] = loc\_Vec$
8:         $T\_Loc[anchor] = one\_T\_Loc$
9:     return $T\_Locs$
```

For each locality in $locO$, we refer to the corresponding $locI$ counterpart of size L_o to determine if the locality changed during synthesis. This information is stored in $locChange$ for each locality. The vectorized training dataset components, $locI$, $locO$ and $locChange$ are returned from the procedure.



(a)



(b)

Fig. 8. (a) Neural network architecture for SAIL-NN reconstruction models. (b) Visualized SAIL-RD rule dictionary structure.

Algorithm 5 SAIL-RD

```
1: procedure SAIL_RD($L_i, L_o, D, P, T, N$)
2:     $locI, locO = VECTORIZE\_DATASET(D, L_i, L_o)$
3:     $M_r = LEARN\_RD\_RECOVERY\_MODELS$
       $(locI, locO, P)$
4:     $T\_Locs = VECTORIZE\_TARGET\_NETLIST(T, L_i)$
5:     $recoveredSnapshots = \{\}$
6:     for each $l \in T\_Locs$ do
7:         $predictions = []$
8:         for each $m \in M_r$ do
9:             $post\_Loc\_Dict = m[l]$
10:            $p = post\_Loc\_Dict$ entries with the Top-$N$ frequencies.
11:            $predictions.append(p)$
12:        $r = ensembleVoting(predictions)$
13:        $recoveredSnapshots[l] = r$
14:    return $recoveredSnapshots$
```

Algorithm 6 Learn RD Recovery Models

```
1: procedure LEARN_RD_RECOVERY_MODELS($locI, locO, P$)
2:     $M_r = []$
3:     for each $(size, oneLocI) \in locI$ do
4:         $trainX = oneLocI$
5:         $trainY = locO$
6:         $ruleDictionary = \phi$
7:         for each $(post\_loc, pre\_loc) \in (trainX, trainY)$ do
8:             if $post\_loc$ is a key in $ruleDictionary$ then
9:                 if $pre\_loc$ is a key in $ruleDictionary[post\_loc]$ then
10:                    $ruleDictionary[post\_loc][pre\_loc] + +$
11:                else
12:                    $ruleDictionary[post\_loc][pre\_loc] = 1$
13:            else
14:                $ruleDictionary[post\_loc] = \phi$
15:                $ruleDictionary[post\_loc][pre\_loc] = 1$
16:        $M_r.append(ruleDictionary)$
17:    return $M_r$
```

2) SAIL-NN Learning the Machine Learning Models: As shown in Algo. 1 line 3, the change prediction model is learned based on $locI$, $locChange$ and certain user-defined machine learning parameters (P). For our implementation of SAIL-NN, we use Random Forest [40] classifiers.

The details of training the reconstruction machine learning models is shown in Algo. 3. M_r is the collection of reconstruction machine learning models which forms an ensemble. $oneLocI$ is the list of localities with $size$ number of gates around each anchor points specified. $P.NNcount$ is a user-defined parameter which determines the number of models to be trained for each locality size. As shown in lines 3-9, for each locality size, we train $P.NNcount$ number of models and store them in M_r to be returned from the procedure. A randomized portion of specified size is extracted from the training set to act as the validation set during the neural network training process (line 6).

The neural network architecture used for training each reconstruction model in the ensemble is presented in Fig. 8 (a). There are total $(n + 1)$ channels for predicting a locality of size n. The first n channels are for the n gate predictions and the last channel is for predicting the adjacency matrix of the locality sub-graph. g is the total number of gates in the standard cell library being used.

3) SAIL-NN Pre-Synthesis Locality Prediction: After the training process completes, the machine learning models are used to predict the pre-synthesis locality state for each locality in the target design T as shown in Algo. 1 lines 5-18. However, before these models can process the target design localities, they must be vectorized as shown in Algo. 4. For each anchor point in T, localities of all sizes (as specified by L_i) are extracted and stored in the dictionary T_Locs indexed by the anchor points. T_Locs is returned as the output of the procedure as shown in Algo. 4 line 9.

In Algo. 1, for each entry, l in T_Locs, we predict the pre-synthesis state around the corresponding anchor point (lines 7-18). In line 8, the change prediction model (M_c) is first used to determine whether a reconstruction of the locality is to be carried out or not. If no reconstruction is required, then the input locality l of size L_o is considered as the output as shown in line 16. Otherwise, if reconstruction is deemed necessary, then for each model in M_r a prediction is obtained and stored in the list $predictions$. The final prediction is computed based on a majority voting scheme as shown in line 14.

C. SAIL Rule Dictionary

For a specific input post-synthesis locality, the SAIL-NN variant is designed to predict one candidate for the pre-synthesis locality state of the given input. To break this limitation, we introduce the SAIL Rule Dictionary (SAIL-RD) variant, which can provide, for a given input post-synthesis locality, a set of potential pre-synthesis locality states with varying degree of likelihood/confidence. Additionally, we have designed SAIL-RD to be more efficient, compared to SAIL-NN, both in terms of training and prediction speed. SAIL-RD utilizes a dictionary-based rule learning approach for reverting a post-synthesis locality to its pre-synthesis state. A high-level view of this method is shown in Fig. 6 (b). SAIL-RD is realized using Algo. 5, Algo. 2, Algo. 6, and Algo. 4.

Algo. 5 is the main procedure for SAIL-RD and starts by vectorizing the dataset D in line 2. L_i is the list of locality sizes required for generating the training input samples $locI$ and L_o is the locality size used for generating the expected output/labels $locO$ corresponding to each input sample. In line 3, the rule dictionary ensemble is trained using $locI$, $locO$ and P (a set of machine learning training parameters defined by the user). T is the target locked design that is required to be evaluated and we start by vectorizing the localities inside T as shown in line 4. Subsequently, in lines 5-13, the machine learning models in M_r are used to predict the pre-synthesis state of each specified localities in T. The recovered locality snapshots are returned as the output of the procedure.

1) SAIL-RD Dataset Vectorization: The training dataset D is vectorized as shown in Algo. 2. The explanation of this algorithms is provided in Section IV-B.1.

2) SAIL-RD Learning: In SAIL-RD, the locality reconstruction process is carried out by a set of rule dictionaries that must be learned from the training dataset. Algo. 6 describes the learning process. For each input (post-synthesis) locality size in $locI$, we learn a rule dictionary model and store it in M_r. The final ensemble M_r, is returned as an output of this procedure in line 17. Each rule dictionary model in the ensemble is a dictionary with $\{key$: post-synthesis locality $(post\_loc)\}$ and $\{value$: pre-synthesis dictionary$\}$. Each pre-synthesis dictionary has $\{key$: pre-synthesis locality$(pre_loc)\}$ and $\{value$: observed frequency of the transformation $(pre\_loc \implies post\_loc)$ in the training set$\}$

In Fig. 8 (b), we see the visual representation of a SAIL-RD model. Each post-synthesis locality is mapped to a set of pre-synthesis localities with a specific weight representing the number of times that specific transformation was observed in the training set. During the testing phase, for a given input post-synthesis locality, the model predicts the Top-N choices based on this graphical model.

3) SAIL-RD Pre-Synthesis Locality Prediction: After the training process completes we proceed with the analysis of the target design T as shown in Algo. 5. T is first vectorized using Algo. 4. More details about Algo. 4 is provided in Section IV-B.3. The vectorized target design localities (T_Locs) are reverted back to their pre-synthesis state in lines 6-13 (Algo. 5). For a given locality $l \in T\_Locs$ we obtain the Top-N predictions from each model in M_r and store them in $predictions$. The prediction p for each model is the top-N entries (in terms of frequencies) in the dictionary $m[l]$. The final top-N pre-synthesis states of the specific locality are determined based on a voting scheme and returned as an output of this procedure.

V. Results

Both SAIL-NN and SAIL-RD can provide crucial information regarding the structural vulnerabilities of the locked design. To quantify the effectiveness of these techniques, we perform a series of evaluations on several logic locking schemes and benchmarks.

A. Experimental Setup

From a given logic locked design, the standard procedure for generating the training dataset is to re-lock (pseudo self-referencing) the already locked design using the same logic locking technique we are trying to attack and then synthesize the design once more to create pre and post synthesis pairs. For the following experiments, we use the Synopsys Design Compiler as the synthesis tool, and GSC (Cadence 180 nm Generic Library) as the standard cell library with cells AND2 $\times$ 1, AOI21 $\times$ 1, AOI22 $\times$ 1, INVX1, MX2 $\times$ 1, NAND2 $\times$ 1, NAND3 $\times$ 1, NAND4 $\times$ 1, NOR2 $\times$ 1, NOR3 $\times$ 1, NOR4 $\times$ 1, OAI21 $\times$ 1, OAI22 $\times$ 1, OAI33 $\times$ 1, OR2 $\times$ 1, OR4 $\times$ 1, XOR2 $\times$ 1, DFFSRX1. All synthesis with Synopsys Design Compiler is done with high area and power effort with the remaining settings as default. The base designs are synthesized until the number of each gate type between synthesis rounds has not changed or a maximum number of iterations (50) is reached. We use the following ISCAS85 and other popular open-source benchmarks locked with the specified key sizes for carrying out the experiments unless otherwise mentioned: {c1355:8, c1908:8, c2670:8, c3540:32, c5315:64, c6288:64, c7552:64, ALU:512, FIR:512}. A total of 15 separate locked designs are generated for each of the fully optimized benchmarks (Standard Seed) to generate a diverse testing/evaluation set. Subsequently, to generate the pseudo self-referencing training dataset, we choose one of the previously locked designs for each benchmark, lock them again considering it as golden. This time we generate 45 separate locked designs for each pseudo self-referenced benchmark for the training propose. For our experiments, we use the gates directly connected with the key inputs as anchors due to their proximity to the logic locking gates. We vary the following parameters based on the experiment we perform:

- SAIL Variant (SAIL-RD or SAIL-NN)
- Logic Locking Technique
- Output/Predicted Locality Size (L_y)
- Input Locality Sizes (L_x)

Other experimental parameters remain the same throughout Section V unless specifically pointed out. We use Keras [41] and scikit-learn [40] for implementing the ML models.

B. Metrics

1) Top-N Accuracy: SAIL-NN and SAIL-RD are both designed to perform the same task but in different ways. Because of how SAIL-RD is designed, it can provide multiple candidates for each prediction with varying degree of confidence. The effectiveness of multiple candidate prediction can be captured by the top-N accuracy metric where we consider a prediction to be correct if the pre-synthesis ground truth snapshot matches any of the top-N predicted candidates. SAIL-NN is designed to produce a single candidate for each prediction. It may be possible to extrapolate extra candidates based on the activations of different output neurons but that may also give rise to contentions and ambiguity. In this paper, we limit our study to only Top-1 accuracy for SAIL-NN.

2) R-Metric: Both in case of SAIL-NN and SAIL-RD, it may happen that none of the top-N predicted candidates are completely correct (but they can still be partially correct). In a predicted locality, a gate error is the number of gates that are of the wrong type with respect to the pre-synthesis ground truth locality. A link error for a predicted locality is

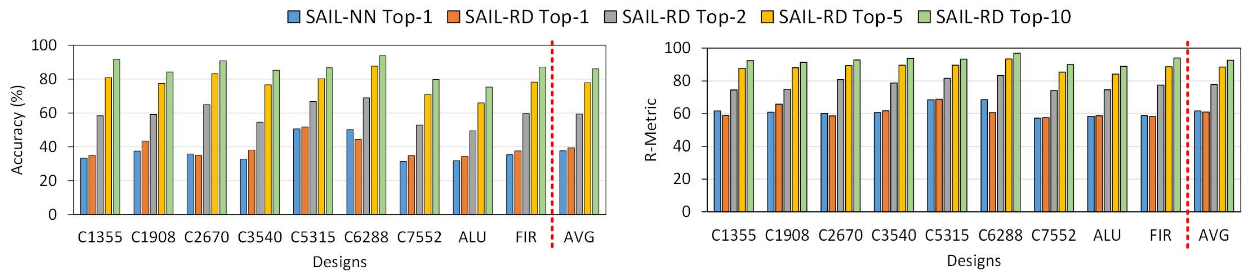


Fig. 9. Comparison between SAIL-NN and SAIL-RD accuracy. Both variants perform equally well in terms of Top-1 accuracy. SAIL-RD is designed to provide multiple prediction candidates and we observe an increase in prediction accuracy as N increases in Top-N. AVG represents the average accuracy across all designs.

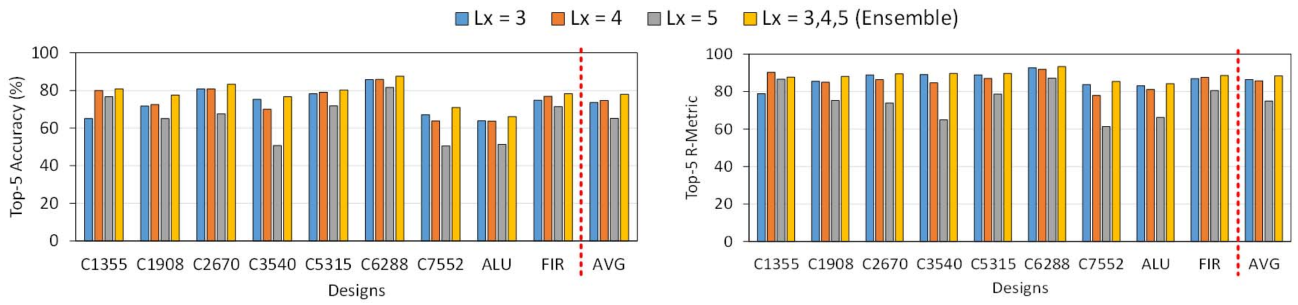


Fig. 10. Comparison of SAIL-RD performance trained across different locality sizes. SAIL-RD ensemble consists of models trained on different input locality sizes (irrespective of the output locality size). SAIL-RD performs best with ensemble.

the hamming distance between the adjacency matrix of the predicted locality and the adjacency matrix of the ground truth locality. In case of a low amount of gate error and no link error, the snapshot prediction is not completely correct but the snapshot remains useful for the next steps of the analysis. So we propose a metric $R \in [0,100]$ as shown in Eqn. 1. For T number of localities predicted during an experiment, GE[i] stands for the number of localities with $GateError = i$ and $LinkError = 0$. For example, in a snapshot with 3 gates, 1 gate types wrong has a weight of 2/3 and 2 gate types wrong has a weight of 1/3. R-Metric captures the overall effectiveness of the structural recovery process.

$$R = \sum_{i=0}^{L_y} \frac{GE[i] \times 100}{T} \times \frac{L_y - i}{L_y} \tag{1}$$

C. Comparison Between SAIL-NN and SAIL-RD

We start off our analysis by first comparing the two different SAIL variants in terms of their structural recovery effectiveness. We use XOR-based logic locking with random key placement to generate a dataset for this particular evaluation. The output/predicted locality size (L_y) is set to 3 and the ensembles are trained on input locality sizes (L_x) [3,4,5]. We train the SAIL-RD and SAIL-NN models on the whole training dataset and evaluate all the 15 iterations of locked designs (test set) for each benchmark. The results are reported in Fig. 9. We observe that SAIL-NN Top-1 accuracy is almost the same as SAIL-RD Top-1 accuracy on average and the accuracy of SAIL-RD increases with the increase in N (for Top-N). XOR-based locking with random key placement appears to be vulnerable to both variants of SAIL with average R-Metric of 92.63 for SAIL-RD Top-10. SAIL-RD also appears to be superior to SAIL-NN due to its ability to predict multiple pre-synthesis state candidates. For training the SAIL-NN Neural Network ensemble we use $P.NNcount = 10$ (Algo. 3), 1000 maximum training epochs for each neural network model and learning rate 0.0001. We train the change prediction model M_c (Random Forest) with input locality size of 5. The Random Forest has 100 trees and the node expansion stops when all leaves are pure or when all leaves have less than 2 samples. Samples are drawn with replacement.

D. Boosting Accuracy With Ensemble

The ensemble we use as the final reconstruction model is made from combining several models, separately trained using datasets with different input locality (Post-Synthesis locality) sizes. As seen in Fig. 10, for different benchmarks the optimal Post-Synthesis locality size varies and the ensemble is the best way to stabilize the variance. We use the SAIL-RD variant for this particular evaluation. The XOR-based logic locking with random key placement is used to generate the dataset and the output/predicted locality size (Ly) is set to 3.

E. Predicting Locality of Larger Sizes

Next we vary the predicted output locality size (Ly) from 3 to 6 as shown in Fig. 11. As the prediction locality size increase, the prediction space increases rapidly. For example, as seen in Table II, there are 512000 possible 3 gate localities for the standard cell library (with the current set of gates) and it is 655360000 for 4 gate localities. So the prediction difficulty increases with the output locality size. Hence we observe the prediction accuracy drop with the increase in prediction locality size. However, even for a locality size of 6 with a prediction space of 6.87195E + 16, we observe SAIL-RD Top-5 R-metric of more than 40 which is indicative of the huge amount of bias in terms of the transformation rules that

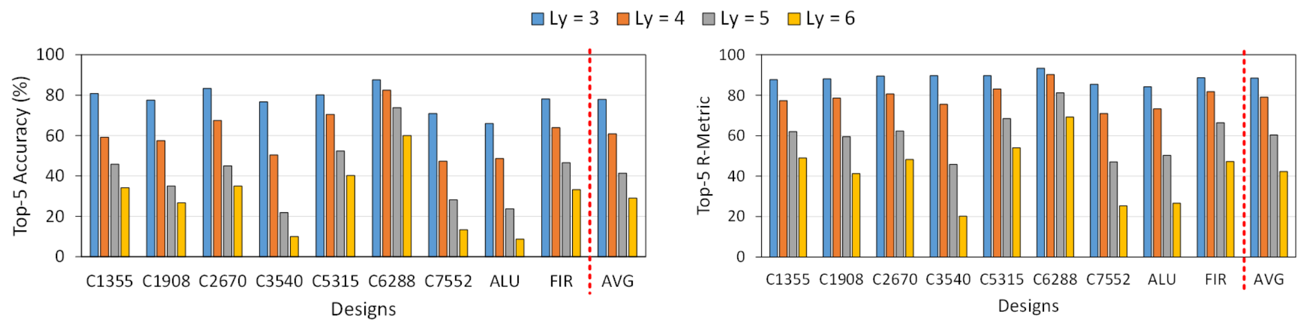


Fig. 11. Comparison of SAIL-RD accuracy for different output locality sizes.

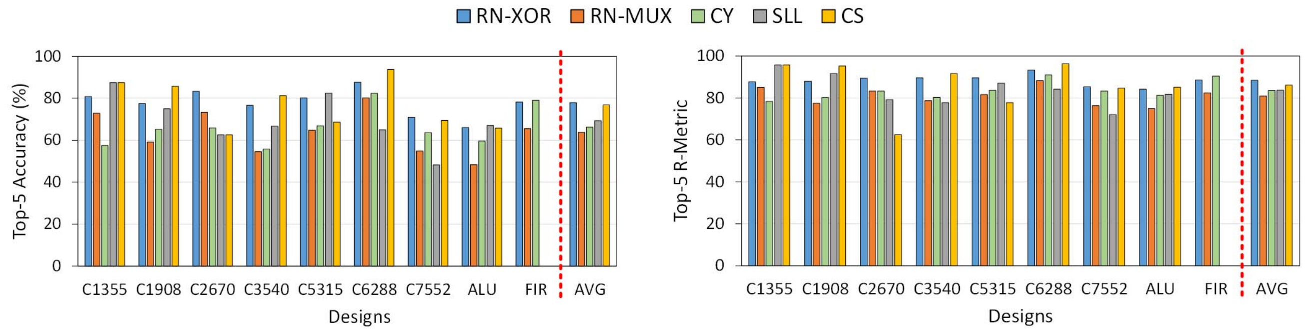


Fig. 12. Comparison of SAIL-RD accuracy for different logic locking techniques.

TABLE II
COMPARISON OF EXPLORATION SPACE FOR DIFFERENT PREDICTED OUTPUT LOCALITY SIZE (Ly)

Ly	3	4	5	6
Exploration Space	5.12E+5	6.55E+5	3.35E+12	6.87E+16
SAIL-RD Avg. Top-5 Acc. (%)	77.91	60.82	41.38	29.02

are being used during the synthesis process. The XOR-based logic locking with random key placement is used for this experiment. The ensembles are trained on input locality sizes $L_x = [3, 4, 5]$ for $L_y = 3$, $L_x = [4, 5, 6]$ for $L_y = 4$, $L_x = [5, 6, 7]$ for $L_y = 5$, and $L_x = [6, 7, 8]$ for $L_y = 6$.

F. Effectiveness Against Different Logic Locking Techniques

The logic locking technique dictates the location and type of locking gate used. In Fig. 12, we see the locality snapshot recovery results for XOR-gate based random locking (RN-XOR), MUX-gate based random logic locking (RN-MUX), AND-gate based Cyclic obfuscation (CY), Secure Logic Locking (SLL), and Logic Cone size based locking (CS) [1]. We observe that these methods are all vulnerable to SAIL and have a Top-5 R-Metric of over 80% for SAIL-RD. The SAIL-RD ensemble is trained on input locality sizes (Lx) [3,4,5] and the output/predicted locality size (Ly) is set to 3.

SLL results for FIR are not reported because the locking algorithm/implementation [1] failed to terminate even after more than a week of running. The SLL and CS techniques are not entirely randomized in nature, hence it is not possible to create multiple iterations of logic locked designs from the same base benchmark. So in order to generate the test set we used 1 set of locked designs for each benchmark with key sizes: {c1355:8, c1908:8, c2670:8, c3540:32, c5315:64, c6288:64, c7552:64, ALU:512}. For generating the training set, we re-run logic locking on the already locked test design again (based on the pseudo self-referencing principle) with key sizes as follows: {c1355:64, c1908:64, c2670:64, c3540:256, c5315:512, c6288:512, c7552:512, ALU:2048}. Note that for RN-XOR, RN-MUX and CY, we follow the standard dataset generation scheme mentioned earlier in Section V-A.

VI. DISCUSSIONS

We have observed that both SAIL-RD and SAIL-NN perform well in exposing the structural vulnerabilities of a logic locked design. Next, we establish a metric (SIVA-Metric) which can quantify the strength of the structural obfuscation of a given design. We also mathematically prove that the SIVA-Metric is the upper bound of SAIL Top-1 Accuracy for a given input/output locality size pair. Based on the intuitions obtained from SIVA, we investigate different defense tactics that can throttle the effectiveness of SAIL. We also perform a runtime analysis for the different SAIL variants proposed and analyze alternative threat models.

A. SIVA: Structural Signature Vulnerability Analysis

SAIL-RD and SAIL-NN can provide an excellent indicator of the strength of structural obfuscation for a given design and technique. However, generating the training dataset and training the models may be time-consuming in a situation where rapid evaluation is necessary. Also, the success of SAIL may vary based on the chosen training process, training dataset, machine learning models and parameters. Hence, through extensive analysis, we have come up with a mathematical metric to quantify the strength of structural obfuscation for a given locked design. Computing this metric (SIVA-Metric) is fast and does not require any training.

In Algo. 7, we describe SIVA (**S**tructural **SI**gnature **V**ulnerability **A**nalysis) in details. *pre* is the locked

Algorithm 7 SIVA

```
1: procedure SIVA($pre, post, anchors$)
2:     $ruleDictionary = \phi$
3:     for each $a \in anchors$ do
4:         $pre\_loc = vectorize(pre, a)$
5:         $post\_loc = vectorize(post, a)$
6:         if $post\_loc$ is a key in $ruleDictionary$ then
7:             if $pre\_loc$ is a key in $ruleDictionary[post\_loc]$ then
8:                 $ruleDictionary[post\_loc][pre\_loc] ++$
9:             else
10:                $ruleDictionary[post\_loc][pre\_loc] = 1$
11:        else
12:            $ruleDictionary[post\_loc] = \phi$
13:            $ruleDictionary[post\_loc][pre\_loc] = 1$
14:    $siva\_metric = 0$
15:    for each $(XStr, XDict) \in ruleDictionary$ do
16:        sortedXDict = sort(XDict, key = lambda x: x[1])
17:        $freq = sortedXDict[-1][1]$
18:        $siva\_metric = siva\_metric + freq$
19:    $siva\_metric = (siva\_metric/len(anchors)) \times 100$
20:    return $siva\_metric$
```

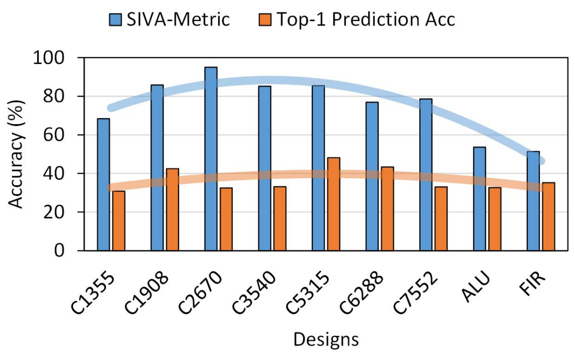


Fig. 13. Comparison of SIVA-Metric and Top-1 Prediction accuracy. An obfuscated design with low SIVA-Metric will be inherently more resilient to SAIL.

pre-synthesis design and $post$ is the locked design after synthesis. $anchors$ are nets in the netlist around which we would like to recover the localities. The first half of the algorithm generates a rule dictionary capturing the transformations that took place between pre and $post$ (lines 2 - 13). $ruleDictionary$ is initialized as an empty dictionary (line 2). For each anchor we vectorize the corresponding localities from pre and $post$ (lines 4, 5). The $ruleDictionary$ is a dictionary with $\{key$: post-synthesis locality $(post\_loc)\}$ and $\{value$: pre-synthesis dictionary$\}$. Each pre-synthesis dictionary has $\{key$: pre-synthesis locality$(pre_loc)\}$ and $\{value$: observed frequency of the transformation $(pre\_loc \implies post\_loc)$ in the data$\}$. The $siva\_metric$ is computed by iterating over the $ruleDictionary$ and sorting each inner dictionary (based on the value) to create $sortedXDict$ (line 16). The maximum frequency entry in $sortedXDict$ is then added to $siva\_metric$ (lines 17, 18). Finally, $siva\_metric$ is normalized over the total number of unique localities (lines 19).

In Fig. 13, we plot the SIVA-metric values for different benchmarks along with their respective SAIL-RD Top-1 accuracy. For this experiment, we use XOR-gate based random logic locking technique and the SAIL-RD model is trained on input locality size (Lx) 3. The output/predicted locality size (Ly) is set to 3.

Contradictory choices in reversal rules is a fundamental property for mitigating structural analysis attacks like SAIL. For example, if localities L_1 and L_2 in the pre-synthesis design both transform into L_3. Then during the SAIL analysis, looking at L_3 we are faced with contradicting choices. In this situation, we will have to pick the statistically most likely solution and proceed. However, if all the locality transformations can be made statistically equally likely then the theoretical upper bound for SAIL top-1 accuracy will also decrease.

B. SIVA-Metric: Upper Bound of SAIL Accuracy Proof

Preliminary Statements:

1) Assume that, there are n unique localities around specified anchor points in a given post-synthesis locked design.
2) For each unique post-synthesis locality LX_i where $i \in [1, 2, \ldots, n]$, D_i is the list of the unique pre-synthesis localities that LX_i transformed from. $D_i = \{LX_{i1} : f_{i1}, LX_{i2} : f_{i2}, \ldots, LX_{im} : f_{im}\}$ where, f_{ij} are the associated frequencies for each of these transformations. $j \in [1, 2, \ldots, m]$, and m is the number of unique pre-synthesis localities that LX_i transformed from.
3) According to the SIVA algorithm (Algo. 7), to compute the SIVA-Metric, we choose for each LX_i, an entry from D_i, $(LX_{ij} : f_{ij})$ such that $\nexists(LX_{ij'} : f_{ij'})$ in D_i where $f_{ij'} > f_{ij}$. We denote the frequency of this chosen D_i entry as, F_i.
4) The SIVA-Metric value for the logic locked design is calculated as follows: SIVA-Metric $= (\sum_{i=1}^{n} F_i) \times \frac{100}{S}$ where S is the total number of localities to be reverted.

Theorem 6.1: SIVA-Metric $= (\sum_{i=1}^{n} F_i) \times \frac{100}{S}$ *implies that the SIVA-Metric is the upper bound of SAIL Accuracy*

Proof: Assume that the premise is true but the consequence is false. Then for some i, $\exists f_{ij'}$ in D_i such that $f_{ij'} > F_i$. This statement creates a contradiction. Hence, the converse of the assumption is true. □

The above proof-by-contradiction establishes an equivalence between the SIVA-Metric and the upper bound of top-1 SAIL accuracy for a specified input and output locality size pair.

C. Transfer Learning: Investigating Alternative Threat Models

SAIL works best if the attacker has knowledge of the locking scheme and the synthesis tool used during locking. This is a fair assumption because according to Kerckhoffs's principle of cryptography a system must be designed so that it is secure even if the knowledge of the locking/crypto algorithm is public knowledge. However, we explore three different scenarios: 1) the locking scheme used by the defender is unknown, 2) the synthesis tool and settings used by the defender are unknown, and 3) both locking scheme and synthesis tool/settings are unknown. All designs are mapped to the standard cells described in Section V-A before attacking. For the first scenario, the attacker can attempt to train a SAIL model with the data from other logic locking techniques and attempt to analyze the target design. In Table III, we observe the

TABLE III
EFFECTIVENESS OF SAIL WHEN THE ATTACKER IS NOT AWARE OF THE LOCKING SCHEME USED BY THE DEFENDER. THE MODELS ARE TRAINED WITH RN-XOR (RANDOM XOR OBFUSCATION) DATA AND ARE USED TO ANALYZE SLL AND CS

IP		C1355	C1908	C2670	C3540	C5315	C6288	C7552	ALU	Avg.
SLL Top - 5	Acc (%)	62.50	25.00	50.00	36.66	52.63	35.08	46.42	46.39	44.33
	R-Metric	79.16	54.16	50.00	63.33	75.43	61.98	71.42	69.93	65.67
SLL Top - 20	Acc (%)	87.50	50.00	50.00	53.33	73.68	59.64	66.07	65.46	63.21
	R-Metric	87.50	66.66	50.00	68.88	82.45	70.17	77.97	77.40	72.62
CS Top - 5	Acc (%)	75.00	42.85	37.50	53.12	49.01	70.31	37.28	36.88	50.24
	R-Metric	91.66	80.95	62.50	69.79	67.32	86.97	76.27	71.03	75.81
CS Top - 20	Acc (%)	75.00	100.0	62.50	75.00	64.70	84.37	76.27	63.31	75.14
	R-Metric	91.66	100.0	70.83	79.16	73.20	91.66	90.96	82.03	84.93

TABLE IV
EFFECTIVENESS OF SAIL WHEN THE ATTACKER IS NOT AWARE OF THE SYNTHESIS TOOL NOR SETTINGS USED BY THE DEFENDER

IP		C1355	C1908	C2670	C3540	C5315	C6288	C7552	ALU	FIR	Avg.
Extreme Top - 5	Acc (%)	57.49	63.33	64.16	67.29	69.27	36.14	55.93	54.34	48.00	57.32
	R-Metric	76.11	79.16	78.33	83.40	82.67	73.29	73.88	75.93	68.60	76.81
Extreme Top - 20	Acc (%)	70.00	76.66	74.16	82.70	77.18	71.04	72.50	74.96	71.39	74.51
	R-Metric	81.11	87.50	83.05	90.97	87.29	86.90	82.36	86.58	82.05	85.31
Medium Top - 5	Acc (%)	57.49	65.83	64.16	65.41	69.06	36.56	53.64	55.36	48.86	57.37
	R-Metric	76.11	78.88	78.61	81.94	81.49	73.19	72.77	75.95	68.97	76.43
Medium Top - 20	Acc (%)	70.00	76.66	74.16	81.66	77.18	70.41	72.29	74.90	71.22	74.27
	R-Metric	81.11	86.38	83.33	89.93	86.11	86.70	81.90	86.32	81.85	84.84

effectiveness of SAIL-RD when trained with RN-XOR dataset and tested on SLL and CS. Although diminished, the effectiveness of SAIL in this setting demonstrates remarkable transfer learning ability. For this experiment, the SAIL-RD ensemble is trained on input locality sizes (Lx) [3,4,5]. The output/predicted locality size (Ly) is set to 3.

For the second scenario, where the attacker is unaware of the synthesis tool or settings used by the defender, the attacker blindly chooses a synthesis tool and setting to mount an attack. In Table IV, we observe the effectiveness of SAIL-RD when the training set was generated using Cadence Genus Synthesis Tool (using both Medium and Extreme optimization efforts) while the defender used Synopsys Design Compiler (using settings described in Section V-A). RN-XOR dataset is used for this experiment and the SAIL-RD ensemble is trained on input locality sizes (Lx) [3,4,5]. The output/predicted locality size (Ly) is set to 3.

For the last scenario, we explore the situation in which both the synthesis tool and obfuscation method is unknown. We train our models on RN-XOR dataset (generated using Cadence Genus Synthesis Tool with extreme settings) and attack (test on) SLL and CS datasets generated using Synopsys Design Compiler (with settings from Section V-A). We observed similar results (-2% accuracy) to Table III. In future works, we will explore more sophisticated transfer learning approaches.

D. SAIL Defense

With SIVA we have a better idea of what drives the success of SAIL and more importantly how we can defend against it. To decrease the SAIL accuracy we must (1) introduce equally likely transformations which will cause contradictions during recovery and (2) create enough randomness in the logic locking process to confusing statistical learning methods. We believe this can be realized through several of the methodologies discussed below.

1) Unoptimized Seed Design: One easy way to introduce more changes in the design due to logic locking is to start the locking process from a highly unoptimized base design. The synthesis process that follows logic locking will have more opportunity to bring about more substantial, unpredictable transformations. As shown in Table V, we compare the SAIL-RD Top-5 accuracy (Acc.) and R-Metric between a dataset generated from our standard procedure (as described in Section V-A) and a dataset generated starting from benchmarks at their highly unoptimized state. We observe a difference of 10% top-5 accuracy and R-Metric of about 5. This improvement in the defense capability indicates that the introduction of diverse changes contributes towards SAIL mitigation. For this experiment, we use XOR-gate based random logic locking technique and the ensemble is trained on input locality sizes (Lx) [3,4,5]. The predicted locality size (Ly) is set to 3.

2) Randomized Synthesis Parameters: The efficiency of SAIL also depends on the success of the training process. If randomness can be introduced during the synthesis step of logic locking then it may be able to confuse the SAIL learning process. We experiment with this idea and generate a dataset where the synthesis parameters are randomized for each locking iteration. In Table V, we observe that the top-5 accuracy for SAIL-RD drops by around 8% due to this technique. The contributing factor here is the randomness of the synthesis process which is throwing off the SAIL models. For this experiment, we use XOR-gate based random logic locking technique and the ensemble is trained on input locality sizes (Lx) [3,4,5]. The predicted locality size (Ly) is set to 3.

3) Randomized Logic Locking Gates: Most logic locking techniques leave an unprecedented amount of structural fingerprints in the design, which is enough for SAIL to make a recovery. For example, in XOR-gate based random logic locking technique the only randomness is in the location where a "predefined construct" is inserted. This "predefined construct" limits the transformation diversity and makes it easier for a statistical attack such as SAIL to succeed. Using "predefined constructs" also makes it easier for an attacker to leverage SAIL results and mount a subsequent key guessing attack [11]. We believe that a logic locking technique which uses different kinds of randomly chosen constructs is ideal for defending against SAIL. Similar techniques for randomizing the key-gate insertion have been leveraged for locking at the gate-level [42] and RTL [43].

TABLE V

TOP-5 SAIL-RD RESULTS FOR DIFFERENT LOCKING METHODS WHICH CAN POTENTIALLY PROVIDE SAIL RESILIENCE

IP		C1355	C1908	C2670	C3540	C5315	C6288	C7552	ALU	FIR	Avg.
Standard	**Acc. (%)**	80.83	77.50	83.33	76.66	80.20	87.60	70.93	66.01	78.17	77.91
	R-Metric	87.70	88.05	89.44	89.65	89.65	93.33	85.38	84.22	88.59	88.44
Unoptimized Seed	**Acc. (%)**	79.16	60.00	71.66	72.08	77.39	56.66	63.12	58.45	64.94	67.05
	R-Metric	89.72	79.44	85.27	85.83	89.40	77.18	81.42	80.98	82.08	83.48
Randomized Params	**Acc. (%)**	84.16	68.06	85.59	61.58	69.74	66.31	59.70	55.80	71.12	69.11
	R-Metric	90.55	81.23	92.37	82.81	84.51	82.20	80.93	79.27	84.95	84.31

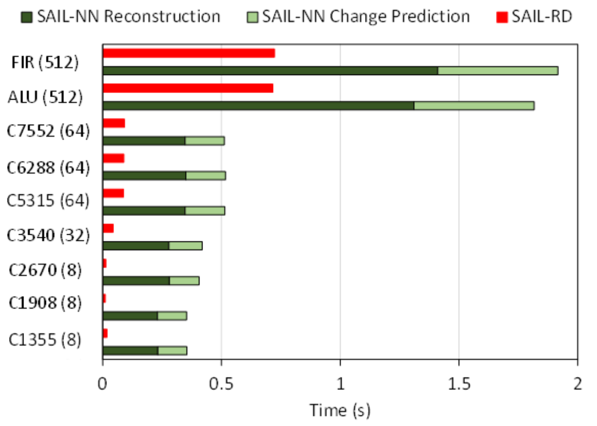


Fig. 14. Comparison of SAIL-NN & SAIL-RD prediction time. SAIL-NN reconstruction time is most costly.

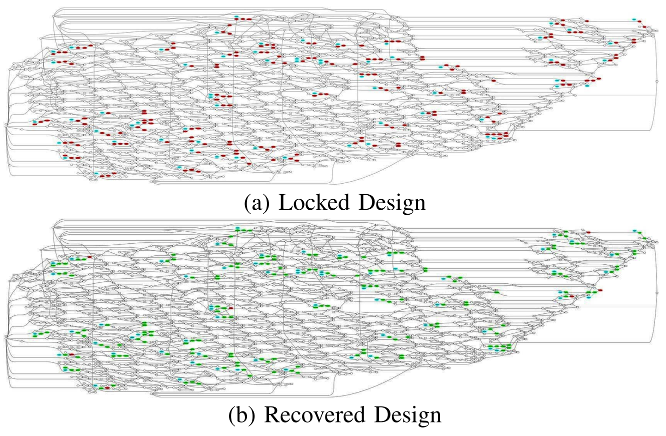


Fig. 15. Graph representation of SAIL-RD (top-5) recovery on RN-XOR locked C6288. Green nodes are correctly recovered gates, and red nodes are incorrectly recovered. Cyan nodes are key inputs.

4) Forced Persistent Randomized Changes: Although the "Unoptimized Seed Design" method was able to provide some resistance to SAIL, it is far from ideal. Unfortunately, state-of-the-art synthesis tools are designed primarily for optimization and not for security. Some level of control should be incorporated in the synthesis tools which can cause the logic locked designs to exhibit transformations that are both optimal and random. A third-party tool can also be used to post-process the synthesized designs in an attempt to introduce more random and big changes. However, such changes must be persistent so that it does not disappear after re-synthesis. In [44], the authors attempt to defend against SAIL by introducing a sufficient amount of conflicting, persistent changes for the same pre/post-synthesis locality pairs to reduce the performance of the ML models.

E. SAIL Runtime Analysis

The time it takes to make predictions with SAIL-NN and SAIL-RD is important for applications where quick turn-around time is essential. As seen in Fig. 14, both SAIL-NN and SAIL-RD are extremely fast even when they are dealing with big designs with tens of thousands of gates and large key size. Also, note that the prediction time is linearly proportional to the key size and almost agnostic to the design size. As an example, FIR has around 30,000 gates and ALU has around 2000 gates for the standard cell library we are using. But there is hardly any difference in the prediction time between ALU (512-bit key) and FIR (512-bit key) for both SAIL-RD and SAIL-NN. The overall training time (trained on all designs from the RN-XOR training dataset) was 4420.74 seconds and 68.70 seconds for SAIL-NN and SAIL-RD, respectively. SAIL-RD appears to be superior to SAIL-NN in terms of both training and prediction time. For this experiment, we use XOR-gate based random logic locking technique and the ensembles for both SAIL-RD and SAIL-NN are trained on input locality sizes (Lx) [3,4,5]. The output/predicted locality size (Ly) is set to 3. For training the SAIL-NN Neural Network ensemble we use $P.NNcount = 10$ (Algo. 3), 1000 maximum training epochs for each neural network model and learning rate of 0.0001.

All results were obtained on a system with the following specifications:

- CPU: Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz
- GPU: GeForce RTX 2080 Ti
- RAM: 64GB

F. Expanding on SAIL Analysis

The information obtained from a successful SAIL analysis can aid in determining the strength of a design's structural obfuscation. However, this information can also be used to more efficiently carry out subsequent attacks such as key recovery or reverse engineering.

1) Key Recovery From Snapshot: Certain fixed-construct XOR-based locking implementations [1], insert only a XOR-gate for $keybit = 0$ and a XOR gate followed by an Inverter for $keybit = 1$. From the locality recovered using SAIL, we can easily predict the value of the key-bit just by observing the gates in the locality. Hence, SAIL can be treated as an intermediate checkpoint from which one can devise other forms of attack. In SURF attack [11], the authors were able to build on top of the SAIL attack results from [39] to mount a subsequent key-recovery attack.

2) Aiding Reverse Engineering: Reverse engineering a netlist to extract its functionality can be severely hampered if a design is logic locked and re-synthesized. Using SAIL attack, we can recover each key-gate locality in the design and make netlist reverse engineering easier. For example, as seen

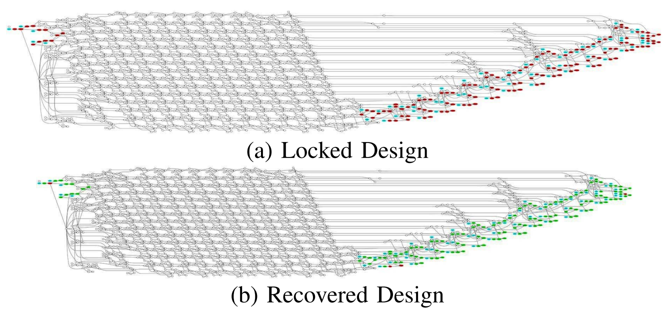

(a) Locked Design
(b) Recovered Design

Fig. 16. Graph representation of SAIL-RD (top-5) recovery on CS locked C6288.

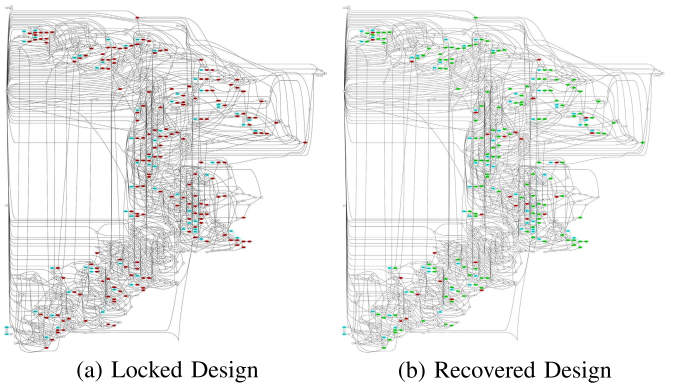

(a) Locked Design (b) Recovered Design

Fig. 17. Graph representation of SAIL-RD (top-5) recovery on RN-XOR locked C7552.

in Fig. 15, if we can locally stitch the predicted pre-synthesis snapshots, most of the changes that logic locking introduced in the locality can be recovered. In Fig. 16 we observe that the CS logic locking technique has left most of the design structurally unmodified and SAIL was able to recover most of the changes that were made in the key gate localities. Fig. 17 shows the behaviour of the RN-XOR logic locking technique on C7552 and highlights the ability of SAIL to recover from the structural changes introduced. Investigating a robust automatic algorithm to replace the post-synthesis locality in the design with the predicted pre-synthesis locality (snapshot stitching) will be a future contribution.

VII. Conclusion

We have reported a powerful and hitherto unexplored attack modality on logic locking, namely SAIL attack, which exposes a critical vulnerability of these methods. Unlike existing functional attack modes, our attack uses a fundamentally different approach that relies on ML models to retrieve structural changes of a locked design with high accuracy. The attack becomes possible since these logic locking methods, due to their reliance on synthesis tools, introduce only small, localized, and predictable transformations in circuit topology. We have presented systematic steps of the attack that lead to the development of an automatic deobfuscation framework. We have studied the efficacy of the attack in terms of both accuracy and computation time for various attacker threat models, logic locking methods, large key sizes, and large benchmarks. While we report a very high accuracy of structural recovery for a set of benchmarks, we believe enhanced feature selection and training approaches can further improve the effectiveness of the attack. SAIL attack, on one hand, is expected to enable robust security analysis of existing logic locking methods and on the other hand, help the development of new methods that are resilient to structural attacks. Accordingly, we designed and presented SIVA, a fast and training-less algorithm that can determine the inherent SAIL-resilience of a logic locked design. We believe one needs to incorporate the following two features to protect against SAIL: (1) distributed and global structural changes, and (2) unpredictable patterns of changes. Future work will include explorations of improved capabilities of SAIL models when making predictions on previously unseen data, generic SAIL models trained across different benchmarks and transformations, and robust SAIL-resistant locking methods.

References

[1] S. Amir *et al.*, "Development and evaluation of hardware obfuscation benchmarks," *J. Hardw. Syst. Secur.*, vol. 2, no. 2, pp. 142–161, Jun. 2018.

[2] R. Torrance and D. James, "The state-of-the-art in IC reverse engineering," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.* Springer, 2009, pp. 363–381.

[3] R. S. Chakraborty and S. Bhunia, "Hardware protection and authentication through netlist level obfuscation," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, Nov. 2008, pp. 674–677.

[4] K. Zamiri Azar, H. Mardani Kamali, H. Homayoun, and A. Sasan, "Threats on logic locking: A decade later," in *Proc. Great Lakes Symp. VLSI*, 2019, pp. 471–476.

[5] B. Tan *et al.*, "Benchmarking at the frontier of hardware security: Lessons from logic locking," 2020, *arXiv:2006.06806*. [Online]. Available: http://arxiv.org/abs/2006.06806

[6] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Security analysis of logic obfuscation," in *Proc. 49th Annu. Design Autom. Conf. (DAC)*, 2012, pp. 83–89.

[7] J. Rajendran *et al.*, "Fault analysis-based logic encryption," *IEEE Trans. Comput.*, vol. 64, no. 2, pp. 410–424, Feb. 2015.

[8] J. A. Roy, F. Koushanfar, and I. L. Markov, "Epic: Ending piracy of integrated circuits," in *Proc. Conf. Design, Autom. Test Eur.*, 2008, pp. 1069–1074.

[9] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the security of logic encryption algorithms," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, May 2015, pp. 137–143.

[10] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran, "Removal attacks on logic locking and camouflaging techniques," *IEEE Trans. Emerg. Topics Comput.*, vol. 8, no. 2, pp. 517–532, Apr. 2020.

[11] P. Chakraborty, J. Cruz, and S. Bhunia, "SURF: Joint structural functional attack on logic locking," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, May 2019, pp. 181–190.

[12] R. S. Chakraborty and S. Bhunia, "HARPOON: An obfuscation-based SoC design methodology for hardware protection," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 10, pp. 1493–1502, Oct. 2009.

[13] K. Shamsi, M. Li, K. Plaks, S. Fazzari, D. Z. Pan, and Y. Jin, "IP protection and supply chain security through logic obfuscation: A systematic overview," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 24, no. 6, pp. 1–36, 2019.

[14] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. Rajendran, and O. Sinanoglu, "Provably-secure logic locking: From theory to practice," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 1601–1618.

[15] M. Yasin, J. Rajendran, O. Sinanoglu, and R. Karri, "On improving the security of logic locking," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 9, pp. 1411–1424, Sep. 2016.

[16] Y. Xie and A. Srivastava, "Anti-SAT: Mitigating SAT attack on logic locking," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 2, pp. 199–207, Feb. 2019.

[17] M. Yasin, B. Mazumdar, J. J. V. Rajendran, and O. Sinanoglu, "SARLock: SAT attack resistant logic locking," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, May 2016, pp. 236–241.

[18] M. Li *et al.*, "Provably secure camouflaging strategy for IC protection," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 8, pp. 1399–1412, Aug. 2019.

[19] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, "Cyclic obfuscation for creating SAT-unresolvable circuits," in *Proc. Great Lakes Symp. VLSI*, May 2017, pp. 173–178.
[20] M. Yasin, A. Sengupta, B. C. Schafer, Y. Makris, O. Sinanoglu, and J. Rajendran, "What to lock? Functional and parametric locking," in *Proc. Great Lakes Symp. VLSI*, 2017, pp. 351–356.
[21] Y.-W. Lee and N. A. Touba, "Improving logic obfuscation via logic cone analysis," in *Proc. 16th Latin-Amer. Test Symp. (LATS)*, Mar. 2015, pp. 1–6.
[22] A. Vijayakumar, V. C. Patil, D. E. Holcomb, C. Paar, and S. Kundu, "Physical design obfuscation of hardware: A comprehensive investigation of device and logic-level techniques," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 1, pp. 64–77, Jan. 2017.
[23] M. E. Massad, S. Garg, and M. Tripunitara, "Integrated circuit (IC) decamouflaging: Reverse engineering camouflaged ICs within minutes," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–14.
[24] Y. Xie and A. Srivastava, "Delay locking: Security enhancement of logic locking against IC counterfeiting and overproduction," in *Proc. 54th Annu. Design Automat. Conf.*, 2017, pp. 1–6.
[25] A. Alaql, T. Hoque, D. Forte, and S. Bhunia, "Quality obfuscation for error-tolerant and adaptive hardware IP protection," in *Proc. IEEE 37th VLSI Test Symp. (VTS)*, Apr. 2019, pp. 1–6.
[26] V. V. Menon *et al.*, "System-level framework for logic obfuscation with quantified metrics for evaluation," in *Proc. IEEE Cybersecurity Develop. (SecDev)*, Sep. 2019, pp. 89–100.
[27] A. Alaql, S. Chattopadhyay, P. Chakraborty, T. Hoque, and S. Bhunia, "LeGO: A learning-guided obfuscation framework for hardware IP protection," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, Apr. 27, 2021, doi: 10.1109/TCAD.2021.3075939.
[28] Y. Hu, V. V. Menon, A. Schmidt, J. Monson, M. French, and P. Nuzzo, "Security-driven metrics and models for efficient evaluation of logic encryption schemes," in *Proc. 17th ACM-IEEE Int. Conf. Formal Methods Models Syst. Design*, Oct. 2019, pp. 1–5.
[29] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, "AppSAT: Approximately deobfuscating integrated circuits," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, May 2017, pp. 95–100.
[30] Y. Shen and H. Zhou, "Double DIP: Re-evaluating security of logic encryption algorithms," in *Proc. Great Lakes Symp. VLSI*, 2017, pp. 179–184.
[31] H. Zhou, R. Jiang, and S. Kong, "CycSAT: SAT-based attack on cyclic logic encryptions," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2017, pp. 49–56.
[32] F. Tehranipoor, N. Karimian, M. Mozaffari Kermani, and H. Mahmoodi, "Deep RNN-oriented paradigm shift through BOCANet: Broken obfuscated circuit attack," in *Proc. Great Lakes Symp. VLSI*, 2019, pp. 335–338.
[33] 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," 2020, *arXiv:2011.10389*. [Online]. Available: http://arxiv.org/abs/2011.10389
[34] A. Alaql, D. Forte, and S. Bhunia, "Sweep to the secret: A constant propagation attack on logic locking," in *Proc. Asian Hardw. Oriented Secur. Trust Symp. (AsianHOST)*, Dec. 2019, pp. 1–6.
[35] D. Sirone and P. Subramanyan, "Functional analysis attacks on logic locking," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 2514–2527, 2020.
[36] L. Li and A. Orailoglu, "Piercing logic locking keys through redundancy identification," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 540–545.
[37] Y. Zhang, P. Cui, Z. Zhou, and U. Guin, "TGA: An oracle-less and topology-guided attack on logic locking," in *Proc. 3rd ACM Workshop Attacks Solutions Hardw. Secur. Workshop*, 2019, pp. 75–83.
[38] F. Yang, M. Tang, and O. Sinanoglu, "Stripped functionality logic locking with Hamming distance-based restore unit (SFLL-hd)–unlocked," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 10, pp. 2778–2786, Oct. 2019.
[39] P. Chakraborty, J. Cruz, and S. Bhunia, "SAIL: Machine learning guided structural analysis attack on hardware obfuscation," in *Proc. Asian Hardw. Oriented Secur. Trust Symp. (AsianHOST)*, Dec. 2018, pp. 56–61.
[40] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.
[41] F. Chollet *et al.* (2015). *Keras*. [Online]. Available: https://keras.io
[42] 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 Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, Oct. 6, 2020, doi: 10.1109/TCAD.2020.3029133.
[43] C. Pilato, A. Basak Chowdhury, D. Sciuto, S. Garg, and R. Karri, "ASSURE: RTL locking against an untrusted foundry," 2020, *arXiv:2010.05344*. [Online]. Available: http://arxiv.org/abs/2010.05344
[44] L. Alrahis *et al.*, "UNSAIL: Thwarting oracle-less machine learning attacks on logic locking," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 2508–2523, 2021.

Prabuddha Chakraborty (Graduate Student Member, IEEE) received the M.Tech. degree from the Indian Institute of Technology (IIT), Kanpur. He is currently pursuing the Ph.D. degree with the University of Florida under the supervision of Dr. S. Bhunia. He has collaborated/worked with companies, such as Xilinx and Texas Instruments. His research interests include computer vision, system security, and applied machine learning in various domains.

Jonathan Cruz received the B.S. (Hons.) and M.S. degrees in computer engineering from the University of Florida. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, University of Florida, under the supervision of Dr. S. Bhunia. His research interests include the secure and efficient IoT, systems security, and reconfigurable computing.

Abdulrahman Alaql received the B.S. degree in electrical engineering from Taibah University, Madinah, Saudi Arabia, in 2011, the M.S. degree in electrical engineering from the University of South Florida, Tampa, FL, USA, in 2016, and the Ph.D. degree from the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA, under the supervision of Dr. S. Bhunia, in 2020. He is currently holding a Research Assistant Professor position with the King Abdulaziz City for Science and Technology (KACST). His research interests include hardware security and trust, IP protection, hardware obfuscation, Trojan detection, and design and testing of reconfigurable devices.

Swarup Bhunia (Senior Member, IEEE) received the B.E. degree (Hons.) from Jadavpur University, Kolkata, India, the M.Tech. degree from the Indian Institute of Technology (IIT), Kharagpur, and the Ph.D. degree from Purdue University, West Lafayette, IN, USA. He is currently a Professor and the Semmoto Endowed Chair with the University of Florida, Gainesville, FL, USA. Earlier, he was appointed as the T. and A. Schroeder Associate Professor of Electrical Engineering and Computer Science with Case Western Reserve University, Cleveland, OH, USA. He has over ten years of research and development experience with over 200 publications in peer-reviewed journals and premier conferences. His research interests include hardware security and trust, adaptive nanocomputing, and novel test methodologies. He received the IBM Faculty Award in 2013, the National Science Foundation Career Development Award in 2011, the Semiconductor Research Corporation Inventor Recognition Award in 2009, and the SRC Technical Excellence Award as a Team Member in 2005, and several best paper awards/nominations. He has been serving as an Associate Editor for IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS (TCAD), IEEE TRANSACTIONS ON MULTI-SCALE COMPUTING SYSTEMS, and *ACM Journal of Emerging Technologies*. He has served as the Guest Editor for *IEEE Design & Test of Computers* in 2010 and 2013 and IEEE JOURNAL ON EMERGING AND SELECTED TOPICS IN CIRCUITS AND SYSTEMS in 2014. He has also served in the organizing and program committee of many IEEE/ACM conferences.