Scalable Attack-Resistant Obfuscation of Logic Circuits

Abdulrahman Alaql  Swarup Bhunia
Dept. of ECE, University of Florida, Gainesville, FL 32608
alaql89@ufl.edu, swarup@ece.ufl.edu

Abstract—Hardware IP protection has been one of the most critical areas of research in the past years. Recently, attacks on hardware IPs (such as reverse engineering or cloning) have evolved as attackers have developed sophisticated techniques. Therefore, hardware obfuscation has been introduced as a powerful tool to protect IPs against piracy attacks. However, many recent attempts to break existing obfuscation methods have been successful in unlocking the IP and restoring its functionality. In this paper, we propose SARO, a Scalable Attack-Resistant Obfuscation that provides a robust functional and structural design transformation process. SARO treats the target circuit as a graph, and performs a partitioning algorithm to produce a set of sub-graphs, then applies our novel Truth Table Transformation (T3) process to each partition. We also propose the $T_3$ metric, which is developed to quantify the structural and functional design transformation level caused by the obfuscation process. We evaluate SARO on ISCAS85 and EPFL benchmarks, and provide full security and performance analysis of our proposed framework.

I. INTRODUCTION

Hardware IPs form a crucial component in the semiconductor industry. A single SoC comprises one or more IP blocks that are usually purchased from a third-party. Hardware IPs are typically available as RTL or gate-level netlists and can be integrated at any stage in the design flow. However, hardware IPs have been plagued by security concerns like IP piracy and counterfeiting in recent years. IP protection techniques can be categorized as authentication based, or obfuscation based. Authentication based techniques rely on the insertion of a unique signature (watermark) to prove ownership of the IP. Obfuscation techniques, on the other hand, rely on preventing the attacker from understanding the design-intent thereby preventing an unauthorized third-party from gaining access to, or replicating the design. The IP life cycle is shown in Fig. 1 alongside some attack vectors and their possible countermeasures.

Obfuscation techniques have significantly been evolving over the past years, where some are applied to the early stages of the IP’s life cycle (functional), others are applied in the fabrication stage (physical) [1]. Physical obfuscation techniques are post-silicon measures that are applied to the basic structure of logic elements. By adding dummy contacts, these logic elements are not easily distinguishable, which adds a layer of protection against physical attacks. Functional obfuscation techniques (also known as logic locking) are applied to Register-Transfer Level (RTL) or gate-level netlists. An example of functional obfuscation and a simple overview of the locking process are shown in Fig. 2. Various attacks have been proposed to break these obfuscation techniques. Over the years, these attacks have grown in sophistication as the attackers utilize information gleaned from every stage in the design process in order to bypass existing countermeasures.

Unfortunately, most obfuscation techniques are developed with a singular goal of mitigating one type of attacks, while leaving the IP vulnerable to unconsidered and unexplored attacks. This philosophy has caused a critical limitation, where existing obfuscation techniques lack a true design transformation. For instance, some techniques focus on adding a single monolithic module at the circuit’s output, leaving the rest of the design unchanged. In other words, most existing techniques have failed to address the three essential properties of hardware obfuscation, which are the functional alteration, the structural alteration, and the scalability to large designs.

In this paper, we propose SARO, a Scalable Attack-Resistant Obfuscation of logic circuits. In this approach, we split the design into smaller partitions, and then perform a systemic Truth Table Transformation (T3) process to each partition. The T3 process is an RTL-based locking mechanism implemented that is highly randomized in many aspects. The main benefit of obfuscating a small and local partition is to help in altering the structure of the netlist as well as the graphical representation of the overall design when applied to all partitions. Instead of inserting key-gates, we have developed SARO to perform a true design transformation to the entire circuit using functional and structural modifications that are able to completely hide the design intent of the circuit. SARO is extremely flexible, as it allows for choosing the size and number of partitions, as well as the type of transformation for each partition, which can help manage area and power overheads. Moreover, the proposed approach is shown to be scalable, as it requires polynomial time to perform the locking.
Finally, Section V concludes this study.

Section IV discusses the implementation results of the proposed and the existing approaches. Section III introduces the methodology for the obfuscation techniques and attacks, and the issues of existing approaches. Section II provides background regarding obfuscation and security analysis of the proposed and the existing approaches.

This paper makes the following major contributions:

- Analyzes the shortcomings of existing techniques, where security and performance issues deem these techniques unpractical or insecure.
- Introduces SARO, a pre-silicon structural and functional hardware obfuscation, where the design is divided into small partitions, and each partition is obfuscated individually. This approach provides two main benefits, structural alteration of the circuit, and the ability to lock extremely large IPs, which makes the method scalable.
- Performs a robust Truth Table Transformation (T3) process that offers a verity of locking mechanism functions and adds a key dependency to both the original and the added gates to form symmetric partition with hard-to-find dummy functions.
- Introduces $T_3^{metric}$, a pre-silicon obfuscation evaluation metric that can quantify the level of structural and functional alteration caused by the obfuscation process.

To the best of our knowledge, there has not been any work that focuses on providing a scalable framework to apply functional obfuscation with the focus on the structural alteration of the circuit. The rest of the paper consists of the following: Section II provides background regarding obfuscation techniques and attacks, and the issues of existing approaches. Section III introduces the methodology for the proposed obfuscation approach as well as the obfuscation metrics that can help evaluate the quality of the applied approach. Section IV discusses the implementation results of the approach in terms of runtime, overhead, and thorough security analysis of the proposed and the existing approaches. Finally, Section V concludes this study.

1) Node Selection Techniques: Multiple node selection approaches have been developed; the following list overviews some of these approaches.

   a) Random Insertion (RN): A random node selection process is performed, where no heuristics are implemented. This approach is considered one of the original obfuscation techniques, which has been introduced in [2]. Although this method is scalable, there is no guarantee that the provided security for this approach is robust.

   b) Secure Logic Locking (SLL): The node selection process introduced in [3] is an iterative approach that avoids creating muting key-gates, which are defined as gates that are masked by other surrounding key-gates.

   c) Logic Cone-Size (CS): This selection process measures the logic cone-size of each node in the design [3]. The cone-size is calculated as the summation of the total number of nodes in the fan-in and fan-out of the considered node.

   d) Controlability-Based Selection: The node selection process in the technique analyzes the switching activity of the circuit and quantifies the probability for each node being switched based on a set of input patterns. This analysis is referred to as the controllability factor. The locking technique in [4] places key-gates in low controllability locations in order to prevent Trojan insertions.

   e) Observability-Based Selection: The node selection process analyzes the circuit for the probability of affecting the observed output, this analysis is referred to as the observability factor. The locking technique in [5] selects nodes to obfuscate based on balanced observability values in order to control the output corruption level. This approach is designed to tolerate slight errors when the obfuscation key source is unreliable or noisy.

II. BACKGROUND AND MOTIVATION

A. Background on Obfuscation Techniques

1) Types of Obfuscation Key-Gates: When considering functional obfuscation, there are different types of key-gates and dummy functions that can be added to selected nodes to be obfuscated. The following list shows the different types of obfuscation gates used for the selected nodes.

   a) XOR/XNOR Gates: In this approach, an XOR/XNOR gate is inserted to the node and add a key signal to one of the inputs of the gate [2].

   b) Multiplexers: A multiplexer is inserted in the selected node [3], one of the inputs to the multiplexer is the original route, and the other inputs are for dummy functions and routes.

   c) Functional-Stripping Operations: Instead of placing a key-gate, a functional-stripping operation is performed to a set of nodes followed by a key-controlled functional restoring unit. The performed arithmetic operation is reversible, such that applying the correct key to the functional restoring unit restores the correct functionality. The best example of this approach is the Stripped-Functionality Logic Locking (SFLL), which has been introduced in [4].

2) Node Selection Techniques: Different node selection approaches have been introduced in the literature; the following list elaborates on these approaches.

   a) Random Insertion (RN): The node selection in this approach is non-deterministic and random [2]. The process is quick and scalable as no analysis of the circuit is needed to insert key-gates. The generated list of selected nodes is changed every time the process is run.

   b) Secure Logic Locking (SLL): The node selection process introduced in [3] is an iterative approach that avoids creating muting key-gates, which are defined as gates that are masked by other surrounding key-gates.

   c) Logic Cone-Size (CS): This selection process measures the logic cone-size of each node in the design [3]. The cone-size is calculated as the summation of the total number of nodes in the fan-in and fan-out of the considered node.

   d) Controlability-Based Selection: The node selection process in the technique analyzes the switching activity of the circuit and quantifies the probability for each node being switched based on a set of input patterns. This analysis is referred to as the controllability factor. The locking technique in [4] places key-gates in low controllability locations in order to prevent Trojan insertions.

   e) Observability-Based Selection: The node selection process analyzes the circuit for the probability of affecting the observed output, this analysis is referred to as the observability factor. The locking technique in [5] selects nodes to obfuscate based on balanced observability values in order to control the output corruption level. This approach is designed to tolerate slight errors when the obfuscation key source is unreliable or noisy.

B. Attacks on Obfuscation

In this subsection, we discuss the existing attacks that can fully or partially break the locking of the obfuscated IPs. Based on our threat model, we focus on attacks that are applied to the locked gate-level netlist of the IP.
1) Satisfiability Attack (SAT): SAT attack is a powerful tool that is applied to any logic-locked combinational circuit to extract the obfuscation key \[9\]. To perform the attack, the obfuscated gate-level netlist, as well as an unlocked circuit, are required. The attack is applied by performing the SAT-solving algorithm; this algorithm applies different input patterns and key values, then compares the outputs of both the obfuscated and the unlocked circuits. The algorithm then uses these distinguishing input patterns (DIPs) to identify wrong key sub-sets, and the process iterates until the entire key-space is covered and the correct key is revealed. The attack is only applicable to purely combinational circuits, or sequential circuits with scan-chain capabilities. Which limits the number of IPs that SAT can be applied to. Another issue with the attack lays upon the applicability of SAT-complex designs (such as block-ciphers and multipliers), where the solvers are not able to reach a solution polynomial time.

2) Key Sensitization Attack (KSA): Key Sensitization Attack (KSA) is a technique that aims to force a primary output to leak a key-value \[5\]. The attacker looks for a pattern that sensitizes a key input to the output. By identifying key-gates that can block the effect of other key-gates (muting gates), the attack can focus on sensitizing the key values of those gates. For the attack to be applied, an activated circuit is required alongside an obfuscated gate-level netlist. The major issue of this attack is scalability, where large and complex designs require a long time to sensitize the key values successfully. Additionally, there is no guarantee that all key values can be sensitized to a primary output of the design, especially in sequential circuits.

3) Hill-climbing Attack: Hill-climbing is an iterative process that aims to maintain and improve the quality of the output. In the context of hardware obfuscation, this process can be used as an attack to obtain the obfuscation key \[9\]. The hill-climbing attack requires a locked netlist, as well as a functional unlocked system. The golden test-patterns used for the system validation in the development stage can also be used instead of an unlocked system.

4) Template Matching Attack: Template matching is a structural-based analysis attack that works in the manner of looking for recognizable patterns in the gate-level netlist \[10\]. Attackers aim to use this approach to reverse engineer a system, where a high-level description of the system is publicly available, and recognizable patterns in the gate-level netlist are identified. The other use for this attack is to identify the critical parts (such as s-boxes in cryptographic modules), then modify the design to disable security modules, insert hardware Trojans \[11\], or leak sensitive information to the primary output.

5) Structural Analysis using Machine Learning Attack (SAIL): SAIL is a machine learning structural analysis attack that aims to restore the obfuscated design into its original pre-synthesis form \[12\]. The attack helps expose the intended key value for each key-gate inserted during the obfuscation process. SAIL is scalable and design-independent, as it can be applied to any obfuscated gate-level netlist that uses XOR/XNOR key-gates. In addition to the gate-level netlist, the attacker needs to obtain the obfuscation tool used on the target system. An enhancement to the attack algorithm, referred to as SURF, was introduced in \[13\]. SURF proceeds the structural analysis by a hill-climbing process that is able to improve the accuracy of the attack.

6) Constant Propagation Attack (Sweep): Sweep is a structural analysis-based attack on logic locking \[14\]. The attack is performed by leveraging the information given by the synthesis tool to expose the correct key. The sweep attack analyzes each key input individually by replacing it with a logic zero in one instance, and a logic one in the other instance. In each instance, the modified locked netlist is synthesized, and the two synthesis reports are compared. The difference between the two reports (deltas) in terms of power, area, and critical path delay are observed and investigated for any correlation with the correct value of the key.

7) Attacks on SFLL: The functional analysis attack on logic locking (FALL) has been introduced in \[15\] to break the obfuscation of the SFLL technique. The attack is specifically designed to identify the functionally striped parts of the design, then it applies a functional analysis algorithm that can restore the original circuit. The attack was able to break SFLL with 90% of successful attempts. Additionally, the attack in \[16\] was able to break the obfuscation by introducing two algorithms that can find the protected patterns of SFLL.

III. Scalable Attack-Resistant Obfuscation

We propose SARO, a scalable attack-resistant obfuscation process that can be applied to protect systems against known attacks. In SARO, we provide a highly randomized design transformation solution that partitions the target circuit and locks each partition individually by employing our novel Truth Table Transformation (T3). The two major benefits of this approach are 1) a strong structural alteration of the circuit is obtained, and 2) a framework that is scalable to large circuits. Our approach is designed to be flexible to any system type, and to any key size. The power and area overheads are also controllable as the process accounts for the size of each partition and the total number of dummy functions to be inserted into the design. The following multi-stage process is shown in Algorithm 1.

**Algorithm 1 Scalable Attack Resistant Obfuscation**

```plaintext
1: procedure SARO
2: Input: org_netlist: Design netlist to be obfuscated
3: Input: k_size: Size of the obfuscation key
4: Input: aig_flag: AIG flag
5: Input: SARO_netlist: Obfuscated netlist
6: G_org ← Hyper_Graph_Converter(org_netlist)
7: TopoG ← Topological_Sorting(G_org)
8: partition_size ← initial_analysis(G_org, k_size)
9: if aig_flag then
10: G_org = AIG(G_org)  // optional
11: G_org[p] = partition(G_org.partition_size)
12: for partition in G_org[p]
13: SARO_netlist[p] = T3(partition)
14: end for
15: locked_netlist = merge(SARO_netlist[p])
16: Output: locked_netlist: Obfuscated netlist
17: end procedure
```

The inputs to the algorithm are the original netlist `org_netlist`, the size of the key, the value of the key (inserted or random),
and overhead constraints (area, power, and timing). The netlist `org_netlist` of the target circuit is converted to its equivalent `G_org`. The hypergraph is then topologically sorted, and each vertex is ranked based on its logic level. The topological dictionary `TopoG` is generated using the function introduced in [17]. Then, we perform a partitioning algorithm as follows: `G_{org}[p] = partition(G_{org}, p)`, where `p` is the number of partitions. Each partition in `G_{org}[p]` is obfuscated individually using our design transformation scheme. Each step in Algorithm [1] is designed to be highly randomized. For instance, after the partitioning is performed, a shuffling step (shown in line 12) is applied. This step ensures that non-deterministic design transformation solutions are offered.

A. Major Steps of SARO

1) Initial Analysis and Partition Size: In the initial analysis stage, the original netlist is compiled using a synthesis tool; the generated gate-level netlist is recomputed multiple times for optimization purposes. The optimized netlist `org_netlist` is considered the reference design, where area, power, and timing reports are used to measure the overheads for the obfuscated netlist produced at the end of the SARO process. Moreover, the size of partitions is specified in this stage based on the size of the circuit and the number of key-bits. The partition size is calculated using Eqn. [1]

\[
\text{partition size} = \text{round down}\left\{\frac{\text{Gates}_{\text{total}}}{k_{\text{size}}}\right\}
\]

where `Gates_{total}` is the total number of gates (vertices), `k_{size}` is the key size in bits, and `partition size` is the number of gates specified for each partition.

2) Universal Gate Transformation: This is an optional stage, where the design is transformed into its equivalent form of only AND gates and inverters, which is referred to as the And-Inverter Graph (AIG). These two gates are considered universal, as any logic function can be produced from these two. Although the generated netlist of this stage will be heavily unoptimized, applying this transform increases the ability to perform the design transformation to parts of the design that might not be accessible when using traditional libraries.

3) Hypergraph Partitioning: In this stage, the gate-level netlist is transformed into its equivalent hypergraph format. A hypergraph is a generalization of a graph in which a structure is represented with vertices and edges. Vertices are the basic elements (gates) in the hypergraph, while edges are the links (wires) that connect these vertices, hyperedges refer to edges that connect more than two vertices. The SARO tool assigns each gate and wire with a numerical value and adds gates to the vertices list, and wires to the edges list. The output of this stage is a hypergraph format file `G_{org}` of the original netlist. The obtained hypergraph file is run through the partitioning process. The used partitioning algorithm in SARO is called hMETIS [18], which is a partitioning framework that produces high-quality partitions in terms of the locality. The algorithm is designed to reduce the number of cut edges, where cut edges can be considered as the outputs of each partition. The partitioning framework uses a multi-level partitioning algorithm, which is described in the following 4-phase process:

- Coarsening Phase: During this phase, the main hypergraph is split into a set of smaller subgraphs. This phase is applied to balance the number of hyperedges in each produced hypergraph.
- Initial Partitioning Phase: The initial partitioning phase computes the bisection of the coarsened hypergraphs generated in the first phase. Since the coarsened hypergraphs are relatively small (about 100 vertices each), the partitioning algorithm will not affect the runtime or the quality of partitions.
- Uncoarsening Phase: In this phase, the partitions of the previous phase are used to reconstruct the main hypergraph. These partitions can be considered the vertices of a higher level graph.
- Refinement Phase: A refinement algorithm is used to create the partitions for the main hypergraph. The objective of the refinement algorithm is to reduce the number of cuts in the main partitions. This phase produces the final list of partitions and their corresponding vertices.

The parameters for the partitioning tool are set based on the overhead constraints which dictate the average number of gates in each partition. The tool produces the file of the partitioned hypergraph `G_{org}[p]`. The obfuscation tool then takes the partitions file, each line in this file represents a vertex, and the value in that line refers to the partition number to which that vertex is assigned to. The tool then reconstructs each partition in the gate-level and identifies the gates, inputs, wires, and outputs for each partition. Each partition is treated as a standalone design module.

4) Truth Table Transformation (T3): In SARO, each partition is obfuscated by altering the overall functionality, this alteration is performed in RTL, where a key-based systemic T3 process is performed to genetically alter the partition’s functionality. Moreover, this transformation aims to maximize randomness while maintaining a robust locking mechanism. This is essential to combat any attack that focuses on machine learning training sets to extract the keys (such as SAIL). Hence, we focus on providing a non-deterministic obfuscation approach that is hard to predict. Given the `i`th partition `P^i(I'[m]) = O'[m]`, where `I` is the local input bus to the partition with the size of `n` bits, and `O` is the local output bus of `P` with the size of `m` bits. For each output bit `t` in `P`, where `0 ≤ t < m`, the Boolean function `f^i_t` can be expressed as `O^i_t = f^i_t(I_{t}^i, I_{t+1}^i, ..., I_{n-1}^i)`. When performing the T3 process, the Boolean function `f^i_t(P = 1)` is transformed to `f^i_t(P', K^i, D') = O^i_t`, with the correct key being `K`, and the wrong key being `K'.

Dummy inputs `D` are taken from other partitions in the circuit, where the algorithm in SARO aims to take dummy inputs from other partitions in the same logic depth. This way a grid-like shape between partitions is formed, which causes a high graphical alteration of the circuit. One challenging aspect of choosing dummy inputs is the accidental formation of combinational loops. These loops are formed when a combinational logic chain drives itself. In other words, if the output of a set of gates in a chain is connected to the input of the chain, then an oscillation behavior is formed. Good design practices encourage avoiding combinational loops, as they cause rapid switching and inaccurate timing analysis of the circuit. In SARO, we avoid forming these loops by only choosing dummy inputs that do not exist in the fan-out cone of the output `O^i_t`, which can help to avoid these loops. Moreover, we perform a
Topological ordering analysis of the circuit, where each node is ranked based on its logic depth from primary ports and memory elements (flip-flops). Candidate dummy wires are selected to be less than or equal to the lowest-ranked partition inputs $I'[n]$. This process provides two benefits, firstly, it helps eliminate the formation of accidental combinational loops during the T3 process, and secondly, it reduces the chance of lengthening the critical path of the circuit, which is a critical factor to preserve the circuit’s performance. The T3 subroutine process outlined in Algorithm 2 shows how the transformation for partition $P'$ is performed.

Algorithm 2 Truth Table Transformation (T3) Process

1: procedure T3
2: Input: $P'$: Target Partition $i$
3: Input: TopoG: Topologically Sorted Vertices Dictionary
4: $T3_{flag} ← partition\_evaluator(P')$ \triangleright \text{optional}
5: if $T3_{flag}$ then
6: return $P'$
7: else
8: $P'' ← \{\}$ \triangleright \text{initialize transformed partition}
9: $f_i ← function\_extractor(P')$
10: $D^i ← dummy\_inputs\_assignment(TopoG, P')$
11: for $q = 1$ to $n$ \triangleright \text{partition output location}
12: $f_q^i(I^{i}, K^i, D^i) ← Transform(f_i(I^{i}))$
13: $v_q^i ← RTL\_generator(f_q^i)$
14: $P'' ← append(v_q^i)$
15: return $P''$
16: end procedure

The T3 process takes both the partition $P'$ and the topologically sorted dictionary $TopoG$ as input. The process starts with evaluating the partition topologically, where two major properties are investigated. Firstly, the fan-in cone for each output is analyzed, where cones that cover more inputs are more suitable for T3 than the others. Secondly, a logic depth analysis for the entire partition is performed, where the maximum logic depth is reported. Based on these two properties, the T3 process is either performed or skipped. The user can adjust the requirements for allowing T3 to be applied, or skip the evaluation altogether. Two partition examples that illustrate suitable and non-suitable partitions for T3 are shown in Fig. 3.

The example in Fig. 3(a) shows a well-connected partition, where fan-in cones cover a large number of inputs. Additionally, the logic depth for this example is high, which allows for a more successful design transformation process. On the other hand, the example in Fig. 3(b) shows a disconnected partition, these types of partitions are generated due to the limitation of the partitioning algorithm, and the type of target design. In this example, most fan-in cones are completely disconnected, and the logic depth of the overall partition is too low. Although T3 can still transform this partition and introduce a robust design modification, it may introduce additional overhead. When the $T3_{flag}$ is true, a functional representation $f_i^j$ for each output $O_i^j$ in the partition is then generated by extracting all logic components controlling the considered output in the partition.

Dummy inputs $D^i$ are also generated by selecting wires in the design that are less than or equal to the lowest-ranked vertex in $f_i^1$. The number and location of selected wires of $D^i$ are randomized, ranging from 0 to $n$ the size of the original partition input $I'[n]$. The transform process is then performed to $f_q^i$, where one of the candidate operations is added to the function to form $f_q^i$. After the function is transformed, a separate file of the modified function $f_q^i$ (expressed in Verilog RTL) is generated. The algorithm concludes when all functions are transformed, and the modified partition $P''$ is obtained by grouping all of the created $f_q^i$ functions. The following list in Table I different types of functional and structural altering operations that are added to the CASE statement:

**TABLE 1: A Summary of Transformation Operations used in T3**

| Modification Type        | Correct Key Applied         | Wrong Key Applied                  |
|--------------------------|-----------------------------|------------------------------------|
| Shuffled Outputs         | $f_i^j(I^{i}, K^i, D^i) = O_i^j$ | $f_i^j(I^{i}, K^i, D^i) = O_{j+1}$ |
| Arithmetic Operation     | $f_i^j(I^{i}, K^i, D^i) = O_i^j$ | $f_i^j(I^{i}, K^i, D^i) = O_{j+1} - K$ |
| Inverted Outputs         | $f_i^j(I^{i}, K^i, D^i) = O_i^j$ | $f_i^j(I^{i}, K^i, D^i) = \text{inv}(O_i^j)$ |
| Dummy Substitution       | $f_i^j(I^{i}, K^i, D^i) = O_i^j$ | $f_i^j(I^{i}, K^i, D^i) = f_i^j(D^i)$ |
| Random Function          | $f_i^j(I^{i}, K^i, D^i) = O_i^j$ | $f_i^j(I^{i}, K^i, D^i) = R_i(1, K, D^i)$ |

The listed transformation operations offer a wide range of modification options to choose from. For example, shuffled outputs can be implemented using different output sequences. On the other hand, arithmetic operations can corrupt the output by performing a key-based randomly selected function. Simple operations such as inverting the outputs are also used, as well as introducing a random function $R_i$ that provides a unique behavior. The application of these modification operations results in a strong functional and topological impact on the partition.

**B. Complexity Analysis for SARO’s T3**

The Boolean function $f_i^j$ can be represented as a look-up table consisting of $n$ select-lines, one output, and $2^n$ configuration entries. The entries of the $f_i^j$ look-up table express the behavior of the function across all possible combinations of $I'$. However, when performing SARO’s design transformation to $P'$, the Boolean function
functions, fixed, SARO is able to control the size of $D^i$, the number of allocated key inputs to the partition
respectively. The number of entries $E$ where possible corrupted functions $F$ in Table II for a 3-input partition shows the total number of

The beauty of incorporating dummy inputs alongside the added key inputs is to extend the size of the truth table of $f_i^n$, which in turn allows for a wide range of corrupted functions to be incorporated, which suggests that the number of possible functions increases exponentially as the total number of inputs increase. The total number of possible functions can be expressed in Eqn. 2 and 3:

$$E = (2^{n+k+d} - 2^{n+d})$$

$$F_{count} = 2^E$$

where $E$ is the number of entry locations for the corrupted functions, $n$, $k$, and $d$ are the input, key, and dummy input bus sizes respectively. The number of entries $E$ is computed by excluding the entries allocated for the original function, which is shown in the subtracted in the equation. $F_{count}$ is computed as $2^E$, which is a high-order function that represents the range of possible configurations that SARO can implement in the locked mode. Since the original partition inputs $I^i$, and the number of allocated key inputs to the partition $K^i$ are usually fixed, SARO is able to control the size of $D^i$ in order to expand the choices for corrupted functions. A simple analysis in Table II for a 3-input partition shows the total number of possible corrupted functions $F_{count}$:

### TABLE II: A Summary of Attacks on Obfuscated Gate-level Netlists

| Input Size | Key Input | Dummy Input | Output Size | $E$ | $F_{count}$ |
|------------|-----------|-------------|-------------|-----|------------|
| 3          | 1         | 0           | 1           | 8   | 256        |
| 3          | 1         | 1           | 1           | 16  | 6.55e+5    |
| 3          | 1         | 2           | 1           | 32  | 4.29e+10   |
| 3          | 2         | 0           | 1           | 24  | 1.67e+7    |
| 2          | 1         | 1           | 1           | 48  | 2.91e+14   |
| 3          | 2         | 2           | 1           | 96  | 7.92e+28   |
| 3          | 3         | 0           | 1           | 56  | 7.20e+16   |
| 3          | 3         | 1           | 1           | 112 | 5.19e+33   |
| 3          | 3         | 2           | 1           | 224 | 2.09e+67   |

The computed values in Table II suggest that introducing more dummy inputs $d$ allows for an extremely wide range of choices for the corrupted function to implement. The major benefit of introducing this level of flexibility is to allow for a robust structural and functional design transformation process. Fig. 4(a) provides two examples that show how the T3 process offers a higher number of corrupted functions to choose from.

Fig. 4(a) shows AND and XOR functions represented in a look-up table format, where input values are highlighted in green, and their corresponding outputs are highlighted in blue. Fig. 4(b) shows both functions being obfuscated using traditional key-gate insertion, where the key input (highlighted in red) determines the behavior of the function and outputs highlighted in red are for the locked mode. When using SARO’s T3 process, additional dummy inputs (highlighted in green) are utilized to widen the range of possible corrupted functions. Fig. 4(c) and (d) show how the design transformation process is applied to AND and XOR gates respectively, where the additional dummy inputs (digits highlighted in green) added a wider range of possibilities.

### C. Obfuscation Evaluation Metrics

In this subsection, we use three metrics that can quantify the structural and functional impact caused by the obfuscation. These metrics can be used to evaluate the existing obfuscation schemes, as well as SARO.

1) **Scatter Index**: The first metric we use is the scatter index of key-entry nodes, which are defined as the first node that is connected to a key input. We focus on these nodes because they determine the location at which the design modification is performed. The proposed scatter metric measures how well-distributed key-entry nodes are in the obfuscated design. The index is calculated by performing a scoring process that evaluates each node in the design and determines whether the node is within an acceptable logic-level-distance from the closest inserted key-entry node. The process is described in Algorithm 3.

#### Algorithm 3 Scatter Index Evaluation

1: **procedure** Scatter Index Scoring
2: **Input**: $L_{depth}[n]$: Logic depth between node $n$ and the closest key-entry node
3: **Input**: $n$: Number of nodes
4: **Input**: $max_{depth}$: Maximum acceptable logic depth
5: scatter score $= 0$
6: for $i = 1$ to $n$
7: if $L_{depth}[i] \leq max_{depth}$ then
8: scatter score $= scatter score + 1$
9: end if
10: end for
11: $T_{index} = scatter score \times 100\%$
12: **Output**: $T_{index}$: Scatter index
13: **end procedure

where $T_{index}$ is the scatter index, $n$ is the number of nodes in the design, and $L_{depth}[n]$ is the logic depth between node $n$ and the closest key-entry node. The parameter $max_{depth}$ is the maximum acceptable logic-level-distance between a node and
the closest key-entry node; this parameter is set by the user to determine the acceptable level of distributed key-entry nodes. The main two factors affecting max\_depth are the key size and the number of nodes n. In our analysis, we set max\_depth = \( \frac{n}{k\_size} \). The function in Algorithm 3 performs a scoring process that evaluates each node in the obfuscated design. For each node i, the variable scatter\_score is incremented when the logic-depth-distance between it and the closest key-entry node is less than or equal to max\_depth.

2) Obfuscation Coverage Index: The second metric we introduce is the obfuscation coverage, which is the ratio of gates affected by the obfuscation to the total number of gates in the design. This coverage metric will quantify the propagation of data corruption caused by locked key-entry nodes. The metric is measured by identifying all gates that are included in the fan-out logic cones of all key-entry nodes, and a percentage of coverage is calculated by Eqn. 4:

\[
C\_index = \frac{\sum_{k=1}^{i\_keys} G\_covered}{G\_total} \times 100\% \quad (4)
\]

where \( C\_index \) is the coverage index, \( i\_keys \) is the number of key-entry nodes in the design, \( G\_covered \) is the number of gates inside the fan-out cones of key-entry node \( k \), and \( G\_total \) is the number of gates in the design. Obfuscated designs with high coverage indexes make it difficult for attackers to analyze the behavior of the circuit when in locked mode and make the obfuscated function hard to identify.

3) Formal Verification: Formal verification is a process that compares two versions of a design and confirms if they are functionally equivalent. This process applies a set of input patterns that are selected to trigger unique and hard-to-reach areas of the design. These patterns are applied to both designs, and the outputs of the designs are compared. The test pattern passes the test if both designs provide an equivalent output, while unmatched outputs are considered a failure. Formal verification can be used to quantify the amount of functional change between the original and the obfuscated circuits. We use the formality index metric, which represents the percentage of corruption out of the total set of patterns applied by the formal verification tool. Eqn. 5 and Eqn. 6 show how the formality metric \( F\_index \) is calculated:

\[
Pattern\_match = \frac{O\_size - [\text{Hamming}(O\_obf, O\_golden)]}{O\_size} \quad (5)
\]

\[
F\_index = \frac{\sum_{k=1}^{P\_total} (1 - \frac{2 \times Pattern\_match[k] - 1)}{P\_total} \times 100\% \quad (6)
\]

the matching rate \( Pattern\_match \) is used to examine each input-output pattern. The golden output \( O\_golden \) is obtained from the unlocked circuit, while the output for the obfuscated circuit \( O\_obf \) is obtained after a randomly selected key is applied. The operation \( \text{Hamming}(O\_obf, O\_golden) \) performs the hamming distance calculation between the two obtained outputs, where the maximum Hamming distance value is \( O\_size \), which is the size of the output bus. The formality index \( F\_index \) is obtained by computing the sum of the mismatch rate, divided by the total number of applied input patterns \( P\_total \).

All three metrics are used to create the \( T^3\_metric \), which quantifies the structural and functional design transformation level of the obfuscation process. Eqn. 7 shows how the \( T^3\_metric \) metric is calculated:

\[
T^3\_metric = \frac{F\_index + C\_index + T\_index}{3} \quad (7)
\]

The indexes \( T\_index \), \( C\_index \), and \( F\_index \) are obtained from Algorithm 3 Equations 4, 5, and 6 respectively. Each of these indexes is equally weighted.

IV. RESULTS AND ANALYSIS

In this section, runtime, overhead, and security analysis are provided when applying the proposed approach to multiple ISCAS85, EPLF, and CEP benchmarks to ensure that our proposed approach provides the necessary protection at an affordable overhead. We use both Synopsys Design Compiler and ABC compiler for synthesis, and we map the benchmark designs on a 250nm LEDA standard cell library. Both ModelSim and Synopsys VCS are used for simulation. The testing hardware used in the analysis consists of a 4.4GHz CPU with 6 cores and 12 threads, and 32GB of DDR4 RAM.

A. Scalability Analysis

To confirm that our approach is scalable, the relation between runtime and circuit size (in terms of vertices) should not be exponential. Fig. 5 shows the runtime comparison between our proposed process and some existing approaches such as random insertion, SLL, and cone-size-based insertion.

![Fig. 5: Runtime vs number of vertices for our implementation of the existing and the proposed obfuscation approaches.](image)

The runtime analysis shows that SLL and cone-size based approaches require an exponentially increasing runtime as the number of vertices increases, up to a point where some benchmarks cannot be practically obfuscated. On the other hand, the runtime analysis shows that our approach has a linear relationship between the number of vertices and the polynomial-time needed to complete the obfuscation process. This linear relation suggests that large IPs will be obfuscated without any scalability issues. We have also applied our approach to extremely large IPs (100,000+ gates), and the runtime results show that the approach is scalable. Table IV shows the runtime results when applying our approach to some large EPFL and CEP IPs.
TABLE III: Comparison of Structural Differences between Various Obfuscation Techniques

| Benchmark | Formality Index (%) | Scaler Index (%) | Coverage Index (%) |
|-----------|---------------------|-----------------|-------------------|
| c432      | 47.2                | 13.4            | 72.8              |
| c499      | 65.3                | 14.3            | 65.7              |
| c800      | 49.3                | 17.8            | 81.2              |
| c1355     | 58.3                | 18.7            | 72.6              |
| c1908     | 55.6                | 15.6            | 64.2              |
| c2670     | 67.6                | 17.8            | 80.6              |
| c3540     | 63.4                | 19.5            | 79.7              |
| c3515     | 68.2                | 48.7            | 74.3              |
| c2688     | 61.2                | 41.8            | 68.4              |
| c5315     | 68.4                | 69.6            | 66.3              |
| ALU*      | N/A                 | N/A             | N/A               |
| mem_ctrl* | N/A                 | N/A             | N/A               |
| FIR*      | N/A                 | N/A             | N/A               |
| AES192    | 302432              | 21734           | 9002.34           |

Average: 60.56% 27.72% 72.49% 49.73% 14.81% 35.56% 51.38% 16.63% 33.32% 81.84% 94.25% 75.51%

1) Functional and Structural Analysis: Based on the results in Table III, it is clear that our approach causes the highest impact on the structure of the design. Getting high values for both formal verification and coverage index indicates that the obfuscation process causes a high level of functional alteration, while a high structural alteration is observed when getting a high scatter index value. The $T_{3\text{metric}}$ metric has been calculated for the existing and the proposed approaches and shown in Table VI.

Since our approach focuses on local obfuscation, a strong structural transformation impact on the entire design is obtained. The optimization process can further alter the structural representation of the circuit. Hence, the design transformation solution offered by SARO would increase the complexity of pattern recognition attempts and hide all design signatures that might expose sensitive information about the system. One way to measure the structural differences between the original and the obfuscated designs is to analyze the proposed $T_{3\text{metric}}$ metric. The following analysis shows the formal verification test being applied to different benchmarks. Table VI shows the functional and structural impact of the existing obfuscation techniques as well as SARO.

The $T_{3\text{metric}}$ metric can be controlled based on the size of the partition. Smaller partitions would increase the $T_{3\text{metric}}$ score as a larger structural impact is caused to the obfuscated circuit. However, the smaller partitions are, the more overhead the design will have. Our obfuscation approach gives the user the ability to choose the size of the partitions, which can be useful when limited overhead is allowed. Fig. 7 shows the relation between the overhead (based on c1355) and $T_{3\text{metric}}$ when changing the number of gates per partition.

A visual comparison between the existing methods and SARO for the ISCAS85 c1355 benchmark is shown in Fig. 6. The comparison shows that all existing methods insert key-entry nodes (highlighted in red) in a clustered fashion, where most of them are grouped in one area. However, key-entry nodes are well-distributed in SARO.

2) Resiliency Against SAT Attack: Due to the distributed SAT resistance, the obfuscated designs are well protected against SAT attack. The SAT attack runtime analysis results for existing obfuscation techniques and SARO are shown in Table VII.

All benchmarks using SARO obfuscation have not been broken through SAT until the set timeout (10 hours). While traditional obfuscation techniques have all been broken quickly.

B. Overhead Analysis

Although overhead constraints can be controllable when using our obfuscation tool, we aim to balance the quality of obfuscation and the cost in terms of area and power overheads. The tool always constrains the latency overhead not to exceed 5% since latency dictates the maximum clock speed that can be used in the IP. Table VII shows the overhead analysis when applying our approach and setting the partition size to be 25 gates. It is clear from the overhead analysis that our obfuscation approach can maintain a low overhead value. The overall average for the tested benchmarks is 23.1% for area overhead and 27.9% for power overhead.

TABLE IV: Runtime Results for SARO applied to large IPs

| Benchmark | Number of Vertices | Number of Partitions | Run Time (seconds) |
|-----------|--------------------|----------------------|--------------------|
| ALU       | 9264               | 458                  | 111.25             |
| mem_ctrl  | 28759              | 1424                 | 277.05             |
| FIR       | 30797              | 256                  | 282.31             |
| hypotenuse| 21438              | 10571                | 7421.81            |
| AES192    | 302432             | 21734                | 9002.34            |

Average: 60.56% 27.72% 72.49% 49.73% 14.81% 35.56% 51.38% 16.63% 33.32% 81.84% 94.25% 75.51%

The comparison shows that all existing methods insert key-entry nodes (highlighted in red) in a clustered fashion, where most of them are grouped in one area. However, key-entry nodes are well-distributed in SARO.

C. Security Analysis

To ensure that our approach provides the necessary IP protection, a thorough security assessment is applied to the obfuscated benchmarks. Various attacks have been applied to these benchmarks under different obfuscation settings. In this subsection, we will discuss the functional and structural alteration caused by SARO, as well as the security analysis against SAT, KSA, hill-climbing, SWEEP, desynthesis, and SAIL attacks.
Fig. 6: A graphical overview of the c1355 benchmark is used to demonstrate the existing and proposed locking methods. The implemented methods are stripped-functionality logic locking [4], full-lock [19], cone-size based locking [3], and SARO. The obfuscation gates are highlighted in red. It is clear that in all existing methods key-entry nodes are clustered in one region of the circuit. However, SARO performs an obfuscation that distributes the key-entry nodes across the entire design and alters the graphical representation of the locked circuit.

Fig. 7: The relation between structural alteration (represented as $T_3$ metric) and the average overhead when changing the number of gates per partition. This analysis is based on c1355.

Even without the addition of the distributed SAT resistance, our approach is designed to obfuscate extremely large IPs (100,000+ gates), and the complexity of SAT grows as the design and the key-size increase. Moreover, large designs are more likely to contain SAT hard blocks that prevent the solver from converging. Hence, SAT attack should not cause a security threat when using our approach.

We have also tested the resiliency of SARO against App-SAT, which is an approximated SAT attack that converges when it finds a key that provides the correct functionality most of the time with minimal corruption that is often tolerable (e.g., 99% of correct functionality) [20]. When comparing our SAT resistance against existing methods, SARO can provide the necessary protection while maintaining high output corruptibility. Table VIII shows a brief comparison between SARO and existing SAT resistance techniques.

### TABLE VI: Functional and Structural Changes Analysis Using the Proposed $T_3$ metric

| Benchmark | CS | SLL | RN | SARO |
|-----------|----|-----|----|------|
| c432      | 44.33 | 35.40 | 25.53 | 84.16 |
| c499      | 48.43 | 38.06 | 27.20 | 83.40 |
| c880      | 49.43 | 22.16 | 24.36 | 81.03 |
| c1355     | 49.86 | 17.73 | 44.93 | 75.50 |
| c1908     | 54.53 | 5.43  | 3.35  | 4.41  |
| c2670     | 55.16 | 5.55  | 3.38  | 5.37  |
| c3540     | 55.16 | 47.16 | 39.20 | 83.96 |
| c5315     | 63.73 | 38.36 | 34.16 | 83.83 |
| c6288     | 68.16 | 40.16 | 39.70 | 82.66 |
| Average   | 53.39 | 33.36 | 33.77 | 83.86 |

### TABLE VII: Runtime for SAT Attack to Break the Obfuscation

| Benchmark | CS | SLL | RN | SARO |
|-----------|----|-----|----|------|
| c432      | 1.17 | 2.25 | 1.21 | timeout |
| c499      | 2.22 | 2.21 | 1.17 | timeout |
| c880      | 4.35 | 1.11 | 3.29 | timeout |
| c1355     | 2.24 | 2.25 | 3.47 | timeout |
| c1908     | 4.34 | 2.22 | 3.35 | timeout |
| c2670     | 14.42 | 11.38 | 13.27 | timeout |
| c3540     | 5.43 | 3.35 | 4.41 | timeout |
| c5315     | 8.55 | 4.38 | 5.37 | timeout |
| c6288     | timeout | timeout | timeout | timeout |
| c7552     | 15.59 | 8.41 | 8.37 | timeout |
| Average   | 6.47 | 4.17 | 4.87 | timeout |

Excluding the runtime of c6288.

3) Resiliency Against Hill-climbing Attacks: Due to the key-size and the high output corruption of the obfuscated designs, hill-climbing attacks will not be easily performed. The accuracy of the initial guess of the key has to be very
high before any observable functional behavior is achieved. The hill-climbing attack has been applied to an obfuscated c1908 benchmark using a 64-bit key; the attack algorithm has timed out (at 10-hours) without converging.

4) Resiliency Against SWEEP and Desynthesis Attacks: SWEEP and desynthesis attacks exploit the synthesis tool to determine whether the selected key is correct or wrong. However, SARO is designed to insert random dummy functions. The type and size of these functions vary, where it might be larger than the original functions in some instances, and smaller in others. This property reduces the accuracy of the attacks, where no common features can be extracted when the guessed key is correct. We have implemented the attacks on a SARO obfuscated c1908 benchmark, and the accuracy of the guessed key does not exceed 50% in all the attempts that we have performed. The attack accuracy analysis results for existing obfuscation techniques (mux-based with randomly selected dummy routes) and SARO are shown in Table IX.

![Table IX: Attack accuracy for SWEEP attack applied to existing techniques and SARO](image)

| Benchmark | CS  | SLL | RN | SARO |
|-----------|-----|-----|----|------|
| c432      | 64.0| 58.0| 75.0| 0.0  |
| c499      | 75.0| 88.2| 72.5| 5.0  |
| c880      | 88.2| 100 | 88.2| 0.0  |
| c1355     | 95.0| 92.5| 72.5| 5.0  |
| c1908     | 48.5| 66.5| 32.6| 2.5  |
| c2670     | 72.5| 67.5| 68.0| 0.0  |
| c3540     | 98.0| 70.0| 78.5| 0.0  |
| c5315     | 100 | 80.8| 86.0| 8.0  |
| c2670     | 72.5| 67.5| 68.0| 0.0  |
| c6288     | 92.5| 82.5| 88.2| 0.0  |
| c7552     | 78.6| 64.0| 77.5| 2.5  |
| Average   | 81.2| 77.0| 73.9| 2.3  |

V. CONCLUSION

We have proposed SARO, a novel approach that applies a strong obfuscation that not only locks the design but also hides any structural signatures that might help attackers gain information about the system. Additionally, the proposed approach applies a design modification process that is randomized in different design aspects, which significantly reduces the accuracy of structural analysis attacks that are based on machine learning and pattern recognition. We introduced a distributed attack resistance scheme that, unlike existing approaches, provides a well-hidden and hard-to-remove protection against all major attacks, while maintaining high output corruption when the wrong obfuscation key is applied. Moreover, the approach we propose proves to be very scalable compared to other existing techniques; runtime analysis shows that our approach is suitable for extremely large IPs. The tool applies the obfuscation technique that uses a customizable key size, and manageable latency, power, and area overheads. We have also introduced the $T_{3}^{metric}$, which is an evaluation metric that quantifies the structural and functional design transformation level caused by the obfuscation process. Security assessment shows that the obfuscated circuits are protected against SAT, KSA, hill-climbing, and SWEEP. Further investigation should focus on incorporating the proposed tool to apply sequential obfuscation schemes.

REFERENCES

[1] R. S. Chakraborty and S. Bhunia, “Hardware protection and authentication through netlist level obfuscation,” in Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design. IEEE Press, 2008, pp. 674–677.
[2] J. A. Roy, F. Koushanfar, and I. L. Markov, “Epic: Ending piracy of integrated circuits,” in Proceedings of the conference on Design, automation and test in Europe, 2008, pp. 1069–1074.
[3] Y.-W. Lee and N. A. Touba, “Improving logic obfuscation via logic cone analysis,” in 2015 16th Latin-American Test Symposium (LATS). IEEE, 2015, pp. 1–6.
[4] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. Rajendran, and O. Sinanoglu, “Provocative logic locking: From theory to practice;” in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 1601–1618.
[5] M. Yasin, J. J. Rajendran, O. Sinanoglu, and R. Karri, “On improving the security of logic locking,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 9, pp. 1411–1424, 2015.
[6] S. Dupuis, P.-S. Ba, G. Di Natale, M.-L. Flottes, and B. Rouzyere, “A novel hardware logic encryption technique for thwarting illegal overproduction and hardware trojans,” in 2014 IEEE 20th International On-Line Testing Symposium (IOLTS). IEEE, 2014, pp. 49–54.
[7] A. Alaqi, T. Hoque, D. Forte, and S. Bhunia, “Quality obfuscation for error-tolerant and adaptive hardware ip protection,” in 2019 IEEE 37th VLSI Test Symposium (VTS). IEEE, 2019, pp. 1–6.
[8] P. Subramanyan, S. Ray, and S. Malik, “Evaluating the security of logic encryption algorithms,” in Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on. IEEE, 2015, pp. 137–143.
[9] S. M. Plaza and I. L. Markov, “Solving the third-shift problem in ic piracy with test-aware logic locking,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 34, no. 6, pp. 961–971, 2015.
[10] W. Li, “Formal methods for reverse engineering gate-level netlists,” CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, Tech. Rep., 2013.
[11] R. S. Chakraborty and S. Bhunia, “Security against hardware trojan through novel application of design obfuscation,” in Computer-Aided Design-Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on. IEEE, 2009, pp. 113–116.
[12] P. Chakraborty, J. Cruz, and S. Bhunia, “Sail: Machine learning guided structural analysis attack on hardware obfuscation,” in 2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST). IEEE, 2018, pp. 56–61.
[13] ——, “Surf: Joint structural functional attack on logic locking,” in 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2019, pp. 181–190.
[14] A. Alaqi, D. Forte, and S. Bhunia, “Sweep to the secret: A constant propagation attack on logic locking,” in 2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST). IEEE, 2019, pp. 1–6.
[15] D. Sironi and P. Subramanyan, “Functional analysis attacks on logic locking,” IEEE Transactions on Information Forensics and Security, vol. 15, pp. 2514–2527, 2020.
[16] F. Yang, M. Tang, and O. Sinanoglu, “Stripped functionality logic locking with hamming-distance-based restore unit (sfll-hd)-unlocked,” IEEE Transactions on Information Forensics and Security, vol. 14, no. 10, pp. 2778–2786, 2019.
[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to algorithms, MIT press, 2009.
[18] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, “Multilevel hypergraph partitioning: applications in vlsi domain,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 7, no. 1, pp. 69–79, 1999.
[19] H. M. Kamali, K. Z. Azar, H. Homayoun, and A. Sasan, “Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks,” in Proceedings of the 56th Annual Design Automation Conference 2019, 2019, pp. 1–6.
[20] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, “Appasat: Approximately obfuscating integrated circuits;” in 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 2017, pp. 95–100.
[21] Y. Xie and A. Srivastava, “Mitigating sat attack on logic locking,” in International Conference on Cryptographic Hardware and Embedded Systems. Springer, 2016, pp. 127–146.
[22] M. Yasin, B. Mazumdar, J. J. Rajendran, and O. Sinanoglu, “Sarlock: Sat attack resistant logic locking,” in 2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 2016, pp. 236–241.