MIDAS: an End-to-end CAD Framework for Automating Combinational Logic Locking

Akashdeep Saha, Siddhartha Chowdhury, Rajat Subhra Chakraborty, *Senior Member, IEEE*, and Debdeep Mukhopadhyay, *Senior Member, IEEE*

***Abstract*—Logic locking has surfaced as a notable safeguard against diverse hazards that pose a risk to the integrated circuit (IC) supply chain. Existing literature on logic locking largely encompasses the art of proposing new constructions, on the one hand, and unearthing weaknesses in such algorithms on the other. Somehow, in this race of make and break, the stress on automation of adopting such techniques on real-life circuits has been rather limited. For the first time, we present a generic end-to-end combinational logic locking CAD framework,** MIDAS**. This framework analyses circuit netlists and generates locked netlists. Due to its generic circuit analysis, it bridges the gap, integrates diverse logic locking techniques, and offers a scope of integration of potential future ones.** MIDAS **framework's efficacy has been verified through its application on ISCAS'85 and ISCAS'99 benchmark circuits, locked using six different schemes such as *EPIC, Anti-SAT, SFLL-HD, SFLL-fault, CAS-Lock,* and *LoPher*.** MIDAS **minimizes the hardware overhead requirements of otherwise resource-intensive locking technique *LoPher* by extracting an influential portion of circuit to lock and utilizing a simple fitness function. We also assess the overhead increase for the aforementioned locking methods, thereby complicating the identification of influential nodes within the locked netlists. Finally, we evaluate** MIDAS **by selectively locking parts of a commercially-designed open-source RISC-V core.**

I. Introduction

The majority of IC vendors have switched to fabless operations as a result of the skyrocketing cost of IC fabrication. Most of IC manufacturing is contracted out to foreign foundries, which raises concerns about its reliability. As a result, it has given birth to several challenges to the IC supply chain, including IP infringement, design piracy, IC overproduction, and the introduction of hardware trojans (HTs). Logic locking has become a proactive defense mechanism against the prevalent threats to various locations in the IC supply chain. The idea behind logic locking is to add key-based logic to the original circuit to hide its operation from the adversary. The secret key is applied, and the locked circuit operates as intended while incurring some hardware overheads.

Real-life circuits comprise of a large number of logic gates. It is infeasible to lock large circuits entirely, considering the overheads of logic locking. Yet, the locked circuit should be robust against various known attacks from the designer's perspective. Only a few works, if not none, discuss the implementation aspects of logic locking to make it feasible to reproduce in real-world circuits. Hence, it naturally raises the question: which part of the circuit should one lock to achieve maximal protection while at the same time minimizing the incurred overheads? Thus, circuit analysis to identify a target location to lock in a large circuit is paramount. This motivates us to design the proposed framework, which can effectively analyze circuits and extract the *influential portions* from it to lock to maximize the effect of logic locking.

Initially, logic locking changed throughout time from randomly including key gates in the design to deliberately selecting techniques to obtain the highest level of resilience at the cost of overheads. However, with the advent of the Boolean satisfiability-based (SAT) attacks pioneered by the authors of [1], the majority of logic locking approaches were rendered useless. The SAT-based attack is easy to mount and only needs two things: (i) *a working chip* or *oracle*, and (ii) *a locked netlist*. The SAT-based attack proceeds to prune the wrong keys by extracting the Distinguishing Input Patterns (DIPs). A DIP is an input pattern that produces different outputs for different key values; the oracle helps distinguish the correct one. With the emergence of the SAT-based attack, the community turned towards designing SAT-based attack-resistant strategies, also known as post-SAT logic locking techniques. The emergence of SAT-resilient post-SAT logic locking is also motivated by the fact that the security against the input-output query-based SAT attacks ensures the provable security of the proposed logic locking technique [2]. The two main categories of post-SAT logic locking strategies are (i) *single point-function schemes*, and (ii) *SAT-hard constructs*. The schemes corresponding to (i) have low output corruptibility [3–5], so the SAT-based attacks are reduced to mere brute-force search on the feasible keyspace. The schemes that correspond to (ii) incorporate SAT-hard structures like look-up tables (LUTs) [6], block cipher [7], etc. The SAT-based attacks are rendered futile since they need exponential time to solve each subsequent iteration.

A plethora of attacks, both oracle-guided [1, 2, 8–10] and oracle-less [11–14] which examine the structural vulnerabilities of the locked netlist and defense strategies [3–7, 15, 16], have been proposed in the literature. Further, in this game of cat and mouse, only a few logic locking techniques have survived to offer protection from known attacks on logic locking. A SAT-hard logic locking scheme called *LoPher* [7] inherits the security features of the well-known block cipher PRESENT into logic locking. It proposes realizing a portion of the circuit without impairing the block cipher's overall structure. The security argument of *LoPher* is based on the fact that unlocking *LoPher* is equivalent to extracting the secret

The authors are members of the Secure Embedded Architecture Laboratory (SEAL), Dept. of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, West Bengal, India 721302.
Email: akashdeep@iitkgp.ac.in, siddhartha.chowdhury92@gmail.com, rschakraborty@cse.iitkgp.ac.in, debdeep.mukhopadhyay@gmail.com.

key of PRESENT[1]. However, configuring the SBox and the inherent permutation layer in realizing circuit components is challenging. Thus, implementing *LoPher* is non-trivial and incurs huge hardware overheads upon locking. Along with other locking techniques implemented in MIDAS as a case study, we present a graph-based implementation strategy of *LoPher*, which aims to minimize the incurred overheads by utilizing a simple fitness function to produce securely locked circuits. The other logic locking techniques have been implemented in MIDAS effectively, maximizing their impact on existing attacks against them, which we also demonstrated experimentally (ref. Sec. V). We emphasize our observation that utilizing the same locking scheme on distinct portions of the target netlist results in varying levels of robustness. The main contribution of this paper can be summarized as follows:

- We propose a methodology to analyze a circuit and extract crucial portions of a circuit that when locked can provide maximal protection to the entire circuit against well-known attacks on logic locking. We have also studied the effects of logic synthesis with different technology libraries and optimization efforts on crucial portions of a given circuit.
- We present a generic end-to-end CAD framework, MIDAS[2] capable of locking the desired portion of the circuit with SAT-hard and point function-based logic locking techniques. It consists of multiple steps, mostly generic and applicable to a wide range of circuits.
- In this work, as a proof of concept, we have adopted *LoPher* as the SAT-hard locking technique, considering the fact that there exist no known attacks against it. However, with certain modifications, other locking techniques can also be realized while keeping the locking target location identification analysis intact. The framework also includes a fitness function, which reports the best possible embedding of the circuit portion (subgraph) into the cipher (represented as a graph) to ensure minimum overheads of locking.
- We further present a case study where we lock a commercially designed open-source RISC-V core from *Syntacore* using our framework. We logic-lock crucial combinational logic modules of the core using the proposed framework, one module at a time.

The rest of the paper is organized as follows. Sec. II discusses the background of different logic locking techniques and prominent attacks against them. We elaborate the MIDAS framework in detail in Sec. III. As a case study, we lock a RISC-V processor presented in Sec. IV. Sec. V contains the experimental results of the application of MIDAS on benchmark circuits followed by a discussion in Sec. VI. We conclude in Sec. VII.

II. Related Works

In this section, we discuss the existing prominent works on logic locking. In particular, we focus on the locking techniques that have been implemented in MIDAS framework.

[1]other substitution–permutation network (SPN)-based ciphers could also be utilized following the same methodology

[2]King MIDAS of Greek mythology could turn anything he touched into gold. The MIDAS Touch or golden touch originates from this [source: Google]. Analogously, the proposed MIDAS framework can convert every circuit it touches from unprotected into a protected version.

A. SAT-Resilient Logic Locking

The highly potent SAT-based attack nullified most of the logic locking techniques then. The first SAT-based attack resilient locking schemes were SAR-Lock [3] and Anti-SAT [4] (ref. Fig. 1(b)). These logic locking techniques utilized single-point functions to produce an incorrect input-output pair for every wrong key in the input-output space. Hence, every faulty input-output pair eliminates a single incorrect key in a SAT-based attack, reducing it to brute-force search based on the key complexity. However, they became vulnerable to *bypass attack* [9] and *removal attack* [12]. To combat the known attacks on logic locking, the *stripped functionality logic locking (SFLL)* technique [5] and its variants were proposed. The main crux of *SFLL* is that a portion of the original circuit design is modified using a *Functionality Stripper* circuit. The resultant structure is restored to its actual functionality using a *Functionality Restore Unit*, (ref. Fig. 1(c)). Additionally, it provides increased output corruption. However, the functional analysis attack on logic locking (*FALL*) [11] exploits the structural traces of the hard-coded secret key from the locked netlist to report the same. A provably secure variant of SFLL introduces stuck-at faults to remove specific functionalities from the target netlist [17]. The resulting failing patterns due to the fault are identified and stored in a tamper-proof look-up table (LUT). A comparator, paired with the LUT, detects these failing patterns and restores functionality using a XOR gate.

CAS-Lock [18], (ref. Fig. 1(d)), is an advanced logic locking approach that was resistant to a union of existing attacks on logic locking. It is a modified version of *Anti-SAT* with improved features. However, a couple of attacks [8, 19] that successfully circumvent the *CAS-Lock* have been proposed. Hitherto, it is evident that the locking techniques based on ad hoc principles would ultimately fail.

The other genre of SAT-resilient logic locking schemes involves inserting additional SAT-hard structures into the original design. *Full-Lock* [6] and *LoPher* [7] fall into the category of SAT-hard locking techniques. *Full-Lock* proposes to insert certain key programmable routing blocks that are SAT-hard by construction. While *LoPher* proposes to realize a portion of the circuit to be protected using the components of a block cipher. As block ciphers are inherently SAT-based attack resistant, *LoPher* is SAT-based attack immune. These locking techniques make the SAT-based attack take exponential time to solve each SAT iteration. However, with the advent of the neural-network-guided SAT attack (*NNgSAT*) [20], the SAT-hard locking techniques simply inserting additional complex SAT-hard instances [6] were attacked. However, [7] harnesses the SAT-based attack resistance of block ciphers to devise the SAT-hard logic blocks. To the best of our knowledge, *LoPher* is one of the very few locking techniques with no reported attacks to date. Keeping all this in mind, we have focused more on [7] in our framework.

Some so-called "generic attacks" on logic locking exist [21, 22]. The attack in [21] proposed to probe the secret key from an activated chip. However, as noted in [2], the adversary model for provably secure logic locking restricts probing the

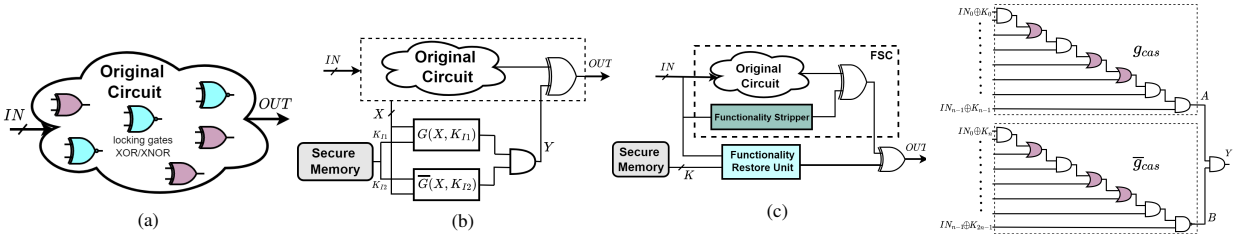


Figure 1: Locking techniques belonging to different genres of logic locking. (a) *EPIC*, (b) *AntiSAT*, (c) *SFLL-HD*, and (d) *CAS-Lock* locking technique. (a) is a pre-SAT whereas, (b)-(d) are post-SAT locking techniques.

oracle[3]. Moreover, this technique entails sophisticated and costly setups, meticulous sample preparations, and skilled adversaries with high precision probing capabilities, raising concerns about its practicality. The fault-based key sensitization attack [22] is rendered infeasible in *LoPher*. Due to the PRESENT cipher's confusion and diffusion properties [24], *LoPher* is secure against such key sensitization-based attacks.

B. *The* LoPher *Locking Technique*

LoPher [7] utilizes block cipher PRESENT [24] to produce SAT-hardness. PRESENT is a lightweight block cipher having 31 rounds with a block size of 64 bits, and a key size of 80 or 128 bits. Block ciphers are cryptographic constructs that realize a mapping $\mathcal{B} : \mathcal{P} \times \mathcal{K} \rightarrow \mathcal{T}$, where $\mathcal{P} \in \{0,1\}^n$ is the plaintext space of $n-$bits; $\mathcal{T} \in \{0,1\}^n$ ciphertext space of $n-$bits and $\mathcal{K} \in \{0,1\}^k$ is the secret key of dimension $k-$bits. The realized mapping $\mathcal{B}$ is a bijection from $\mathcal{P}$ to $\mathcal{T}$ for a particular key $k \in \mathcal{K}$. There are sixteen 4×4 SBoxes, a bit permutation layer, and a key mixing layer in each round of PRESENT[4]. The authors in [7] propose using SBoxes to realize various logic gates by fixing specific input bits to chosen binary values, allowing the SBox to function as the desired gate. Similarly, routing in a circuit is achieved by configuring the SBox as a switch box and using the permutation layer in PRESENT. A PRESENT structure modification allows setting SBox input bits of intermediate layers to perform desired functionalities (logic gates, buffer, or switch boxes). This involves introducing a layer of 2-input MUXes to fix the necessary input bits, with the select lines of these MUXes determining the fixed bits. The authors demonstrate that this modification does not compromise PRESENT's security [7]. Overall, *LoPher* resists existing logic locking attacks.

C. *Contemporary CAD Framework*

A pre-silicon vulnerability identification framework called *Vigilant* [25] uses graph-theory-based algorithms to find probable targets in a locked netlist for fault-injection attacks. Likewise *Valkyrie* [2] is a security diagnostic tool that locates weaknesses related to structural attacks. Another attack framework utilizing graph neural networks (GNN) to capture structural and functional vulnerabilities is *Titan* [26]. The main focus of each of the aforementioned frameworks is structural vulnerability detection. For instance, *Vigilant* validates the vulnerabilities in [4, 6, 18], whereas *Valkyrie* targets the single-point function techniques [3–5].

The proposed framework implements a SAT-hard logic locking technique using ciphers and other diverse locking methods. It can also integrate any cipher to lock an extracted portion of the original circuit design.

The existing CAD frameworks have their primary objective as reverse engineering netlists to uncover logic or functional components [27–30]. They do not perform logic locking. For offensive and defensive objectives, gate-level netlists can be altered by *HAL* [31]. A recently proposed interactive graphical user interface for circuit analysis is *Netviz* [32]. However, the proposed MIDAS framework concentrates on creating locked circuits by evaluating designs for maximum impact, making it more challenging to unlock the locked netlists.

III. MIDAS: The Proposed Framework for Logic Locking

In this section, we discuss in detail the basic building blocks of the proposed framework MIDAS. We particularly highlight the generic and the *LoPher*-specific portions of MIDAS.

The framework begins by parsing the circuit netlist into a graph $G_1 = (V_1, E_1)$, where nodes V_1 represent gates and edges E_1 represent circuit wires. Each node includes a characteristic vector with information like gate name, inputs, outputs, topological level, input/output nodes, and a colour indicating the number of primary output nodes influenced by the circuit. This is a generic step in MIDAS (see Fig. 2. Likewise, We represent a block cipher in a graph-based form as $G_2 = (V_2, E_2)$. Here, nodes V_2 are the SBoxes in each round and edges E_2 denote the permutation layer connecting SBoxes across rounds.

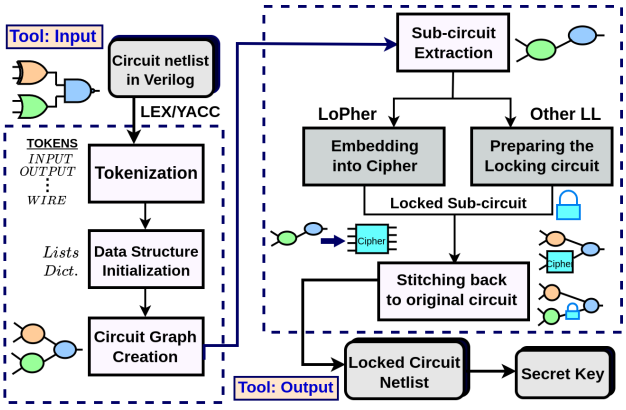


Figure 2: An overview of the MIDAS framework.

[3]The attack model utilizing HTs for key extraction [23] also deviates from established threat models in provably secure logic locking.

[4]Without the loss of generality, the round key layer exists, but for simplicity has not been highlighted in this mathematical representation

A. Circuit Extraction

In a real-world setting, a circuit consists of thousands of gates. Hence, it is infeasible to lock the entire netlist with acceptable overhead. Hence, only a portion is extracted and locked using the desired locking technique. As discussed, we represent the circuit as a graph, which aids us to involve various graph-based algorithms for analyzing netlists. The MIDAS framework extracts a portion (known as the *influential portion*) of circuit (C_O) that needs to be obfuscated. The *influential portion* can be defined as that part of the circuit that affects the maximum circuit outputs, or results in the highest output corruptibility when obfuscated. Since output corruptibility is a crucial security metric in logic locking [33–35]. The work in [33] asserts that locked circuits should exhibit high output corruptibility to increase the adversary's ambiguity. Likewise, Fig. 6 in [34] includes output corruptibility as part of quality metrics, alongside resilience to various attacks on logic locking.

Algo. 1 outlines our proposed approach of extracting the *influential portion*. It takes input the circuit netlist (C_O), where O denotes the number of primary outputs and the set of target outputs (G_O), to produce the graph-based representation G_1 as an interim step utilizing the LEX/YACC parser (line 3). Next, every primary output in G_1 is collected in the set G_O and is associated with a unique colour (line 5, Algo. 1). Subsequently, utilizing graph traversal algorithms, all the dependent nodes for each primary output in the set G_O are assigned the corresponding colour of that particular primary output. This process is iterated for every primary output node (lines $7-16$, Algo. 1). The data structure $ColorsDict[\mathcal{N}]$ stores the colours assigned to all the nodes $(\mathcal{N})$ in G_1. The number of colours assigned to a particular node signifies the number of primary outputs in G_1 it influences. It handles the colour attribute of node *characteristic vector*. Please note that the framework facilitates the locking of specifically chosen user-defined outputs. The user is required to input the desired selection of outputs for obfuscation (ref. as set G_O) in Algo. 1, where *influential nodes* are subsequently extracted based on set G_O.

Next, we define a couple of properties that have been utilized in designing the extraction of *influential portion* (G_1'), in MIDAS.

Property 1. *Extracting a portion of the circuit is equivalent to selecting a subset of nodes and their corresponding edges as in G_1, i.e., a subgraph (G_1'), from the graphical representation of the same.*

Property 2. *The node(s) in the graphical representation of the netlist having the maximum colour assignment are probable candidates for the root node of subgraph G_1'.*

The node (or set of nodes) with maximum influence or assigned colours are identified and represented using the set N_{max_C}. $Find\_Max\_Color()$ is used to construct N_{max_C}. It is a simple function that finds out the node/s that have the maximum colour assigned to it, implying presence of maximum cardinality in $ColorsDict[j]$, and $j \in N_{max_C}$. Informally, it is synonymous with finding the maximum element/s in an array.

Identifying the set N_{max_C} is a generic step, and it is crucial for executing any locking techniques implemented in MIDAS. Tab. I illustrates the number of nodes in the set N_{max_C} for ISCAS-85 benchmark circuits. The primary inputs are not considered to be a part of N_{max_C}. Further, N_{max_C} can consist of more than one node. Algo. 1 then iterates over the elements of N_{max_C} and extracts a subgraph of the desired depth. In Sec. V, we analyze how logic synthesis and optimizations of benchmark circuits influence the size of N_{max_C}.

The subgraph extraction is a *LoPher*-specific step in the MIDAS framework. Since, considering all the other locking techniques implemented in MIDAS, the original circuit remains intact. Additional locking circuitry is appended to the original circuit to produce a locked netlist. However, in *LoPher*, a portion of the circuit (extracted subgraph) is actually replaced with block cipher components.

Example 1. *The set N_{max_C} for* c17 *circuit will have a couple of nodes $[N16, N11]$ as returned by the $Find\_Max\_Color()$ function in Algo. 1. Since both of them have $|ColorsDict[N16]| = 2$ and $|ColorsDict[N11]| = 2$,*

Algorithm 1: Influential Portion Extraction algorithm

```
1  Input: Circuit Netlist (C_O), Target Outputs (G_O)
   // O ← number of outputs; Default G_O has all C_O outputs
2  Output: subgraph of maximum influence (G_1') of desired depth
3  G_1 = parser(C_O) // Graph-based representation of C_O
4  Initialization: Output nodes in G_1 (i.e., G_O)
5  Assign: Unique colours to every element in G_O
6  Define: Queue Q, Cone_nodes[], ColorsDict[N] // N ← # of nodes in G
7  for each i in G_O do
8      Enqueue(Q, i)
9      while (!Empty(Q)) do
10         tmp ← Dequeue(Q)
11         for each j in parents[tmp] do
12             ColorsDict[j] ← colour[i]
13             Enqueue(Q, j)
14         end
15     end
16 end
17 N_maxC = Find_Max_Color(ColorsDict[N])
   // Returns the node with maximum colour
18 if N_maxC ≥ 2 then
19     for each i in N_maxC do
20         if fan_in(i) ≤ 3 then
21             Cone_nodes ← Cone_nodes ∪ BFS(i) upto three levels upward // Here the desired depth is three
22             return Cone_nodes
23             break
24         end
25     end
26 end
```

Circuit	c432	c499	c880	c1355	c1908	c2670	c3540	c5315	c6288	c7552
I/O	36/7	41/32	60/26	41/32	33/25	233/140	50/22	178/123	32/32	207/108
# Nodes	19	80	2	296	435	4	54	5	7	4

Table I: The number of nodes with maximum colour assignments for each benchmark circuit.

which is the maximum colour assignment among all c17 *circuit nodes. Next, the nodes of the subgraph (*$Cone\_nodes$ *elements), illustrated in Fig. 3, are extracted considering* $N16$ *of* N_{max_C} *as root node.*

1) Security Analysis: The extraction of *influential nodes* from a circuit intended for locking is performed using Algo. 1. Concerns may arise about adversaries exploiting Algo. 1 on a locked netlist to extract *influential nodes* and identify the locking circuitry. However, we demonstrate the difficulty of identifying actual *influential nodes* from a locked netlist. The number of *influential nodes* in a locked netlist significantly increases compared to the original circuit due to the integration of locking circuitry. This integration causes multiple nodes from the added circuitry to be considered members of *influential nodes* extracted by Algo. 1, making it challenging to distinguish actual *influential nodes* from those related to the locking circuitry. Next, we outline the specific increase in gate counts for various locking circuitry in MIDAS.

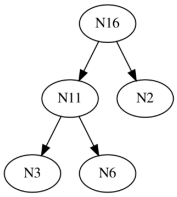


Figure 3: Influential portion of the *c17* circuit extracted with $N16$ as root node.

Estimation of Additional Nodes: For the pre-SAT locking technique *EPIC*, which integrates key-based XNOR/XOR gates into the target circuit, it is easy to infer that the number of introduced gates scales directly with the number of key bits.

The gate count for the introduced locking circuitry in *CAS-Lock* is the same as that of *AntiSAT* [18]. Assuming a $K-$bit key, the additional gates in the complementary AND chains are $2(K-1)$ two-input AND/NAND gates. Additionally, the AND chain precedes $2K$ additional two-input XOR/XNOR key gates and finally, an AND gate to merge the output of the two complementary blocks. Hence, the total number of additional gates introduced by *CAS-Lock* and *AntiSAT* becomes:

$$= 2K + 2(K-1) + 1 \tag{1}$$

SFLL-HD consists of two blocks $Restore\ Unit$ and $Stripe\ Unit$ that assess the relationship between the input and the protected cube. These blocks function as comparator logic, employing XOR and AND gates [11]. We are reiterating the same example mentioned in [11], illustrated in Fig. 4. The functions implemented by the $Restore$ and $Stripe$ units in Fig. 4 with HD parameter $h = 1$ and $N = 4$ are as follows:

$$\begin{aligned} Restore\ Unit &= (\overline{a} \wedge \overline{b} \wedge \overline{c} \wedge \overline{d}) \vee (a \wedge b \wedge \overline{c} \wedge d) \vee \\ &\quad (a \wedge \overline{b} \wedge c \wedge d) \vee (a \wedge \overline{b} \wedge \overline{c} \wedge \overline{d}) \\ Stripe\ Unit &= \sim ((p \vee q \vee r) \wedge (p \vee r \vee s) \wedge (p \vee q \vee s) \\ &\quad \wedge (q \vee r \vee s)) \end{aligned} \tag{2}$$

Next, we generalize the aforementioned functions for an $N-$input *SFLL-HD* circuit with HD parameter h to estimate the additional gates as follows:

$$\begin{aligned} Restore\ Unit &= {}^{N}C_h(N-h-1) + ({}^{N}C_h - 1) + (N+1) + 1 \\ Stripe\ Unit &= ({}^{N}C_h - 1) + {}^{N}C_h(N-1) + {}^{N}C_h\lceil N/2 \rceil + 1 \end{aligned} \tag{3}$$

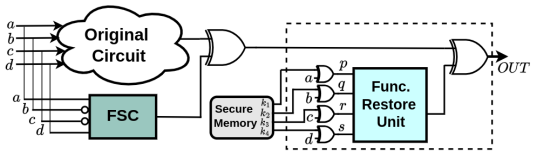


Figure 4: An example [11] of *SFLL-HD* with secret key $(k_1, k_2, k_3, k_4) = 1001$, HD ($h$) = 1, and inputs (a, b, c, d).

where, ${}^{N}C_h$ represents the number of failing patterns. The CNF representation of the $Restore\ Unit$ [11] for a protected pattern includes ${}^{N}C_h(N-h-1)$ OR gates, $({}^{N}C_h - 1)$ AND gates, $(N+1)$ XOR gates, and a NOT gate. Similarly, the $Stripe\ Unit$ comprises $({}^{N}C_h - 1)$ OR gates, ${}^{N}C_h(N-1)$ AND gates, ${}^{N}C_h\lceil N/2 \rceil$ NOT gates, and a XOR gate. While estimating NOT gates, we assume a uniform distribution of the protected pattern with an equal number of 0s and 1s at $h = N/2$, which generates the highest output corruptibility [5]. The total number of additional gates is the sum of those in the $Restore$ and $Stripe$ units.

LoPher comprises multiple rounds of the PRESENT cipher with an additional MUX layer in each round. We calculate the gate overheads for each layer to estimate the additional gates. Assuming ANF representation for each output bit of the PRESENT S-Boxes [7], the first three S-Box output bits require 14 two-input AND and XOR gates each, while the last output bit requires four such gates. With 16 S-Boxes in each round, the total additional two-input gates per S-Box layer amount to $16 \times \{3(14) + 4\} = 736$. Additionally, the $64-$bit MUX layer in each round necessitates 256 additional gates (assuming the standard $2:1$ MUX equation of $(A.S' + B.S)$), and the *add round key* layer requires 64 XOR gates. Thus, the total additional two-input gates introduced due to *LoPher* are:

$$= (736 + 256 + 64)R = 1056R \tag{4}$$

where R is the number of cipher rounds utilized in embedding a circuit in PRESENT. A 32-bit key utilized for *CAS-Lock* and *AntiSAT*, the augmentation in *influential nodes* in contrast to the original ones detailed in Tab. I is ≈ 127. Meanwhile, the count of failing patterns (${}^{N}C_h$) for a 16-bit *SFLL-HD* with $h = 4$ is 1820 and the increase in *influential nodes* is ≈ 65520. Lastly, four PRESENT rounds for locking integrate ≈ 4200 additional *influential nodes*.

The number of additional XOR gates used in stitching increases linearly with the number of influential nodes. The aforementioned estimates consider all two-input gates except the one-input NOT gate. As discussed earlier, these additional gates are identified as *influential nodes* by Algo. 1, when the locking circuit is integrated with the original one.

Moreover, it is a standard practice to resynthesize locked netlists. Logic synthesis and optimization play crucial roles in shaping the circuit's graph structure, altering the number and positions of *influential nodes* in the locked design. These new influential nodes, identified post-synthesis of the locked design, bear no correlation with those in the original circuit netlist and hence, as mentioned in [2], cannot be traced. The technology library-based representation of the two input gates used in the estimation is proprietary to the fabrication unit, often employing complex gate representations. Even if

the locked design is released without optimization or logic synthesis, the adversary lacks a standard reference point (original netlist) to distinguish the original *influential nodes* from the potential *influential nodes* extracted by Algo. 1 in the locked netlist. Next, we present an example to demonstrate the aforementioned arguments.

Example 2. *For c432, Algo. 1 identifies 18 influential nodes. Locking three randomly chosen* influential nodes *with a 32-bit SFLL-HD locking scheme utilizing* MIDAS *results in 1380* influential nodes *when analyzed with Algo. 1. Hence, the adversary has to find three* influential nodes *from the set of 1380 nodes. Post-synthesis, the same locked c432 netlist yields 773 influential nodes.*

B. Embedding

This section discusses the most critical step specific to the *LoPher* locking technique. The embedding step is required in *LoPher* since it replaces a circuit's extracted subgraph with block cipher components. The main idea is to embed the sub-circuit, extracted utilizing Algo. 1 (ref. Sec. III-A), into the cipher. The sub-circuit functionality is realized using the block cipher components.

We first discuss the NP-completeness of the embedding problem and then propose a heuristic algorithm for it. We can reduce the well-established *Subgraph Isomorphism problem* (Problem 1) to the proposed embedding problem (Problem 2).

Problem 1. ***Subgraph isomorphism problem.*** *Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, let $G'_1 = (V'_1, E'_1)$ be the subgraph of G_1, the problem determines whether G'_1 is isomorphic to G_2.*

The subgraph isomorphism problem is NP-Complete. The Subgraph Isomorphism problem takes as input two graphs G_1 and G_2 and checks whether a subgraph of G_1 is isomorphic to a subgraph of G_2. Let us establish the connection between the proposed embedding problem with the above subgraph isomorphism problem.

Problem 2. ***Embedding problem.*** *Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the problem determines whether G_1 can be realized using G_2 components (vertices and edges).*

The graph G_2 can be *embedded* using components of G_1 since G_2 is a subgraph of G_1. *LoPher* proposes to embed an extracted subgraph using a block cipher. However, in our case, the challenge is that the structure of the block cipher is entirely different from the extracted subgraph of the circuit. To embed the same, we have to fix several constraints in the block cipher structure, which serves as secret keys of the *LoPher* locking technique. Next, we formally prove the NP-completeness of the *embedding problem*.

Lemma 1. *The embedding problem for two graphs G_1 and G_2 is NP-complete.*

Proof. We first establish that the subgraph isomorphism problem is polynomial-time reducible ($\leq_P$) to the embedding problem quite trivially, and then proceed further to establish their relationship. Let the graphical representation of the original circuit be G_1, the extracted sub-circuit be G'_1. Let the graphical representation of the cipher be G_2.

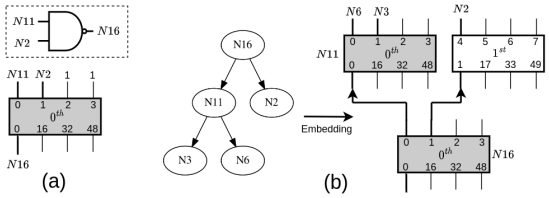


Figure 5: (a) Embedding the root node $N16$ in PRESENT SBox layer. (b) Embedding remaining subgraph of *c17*.

The subgraph isomorphism problem can be directly mapped into the proposed embedding problem, if we consider graphs G_1 and G_2 in the subgraph isomorphism problem to be the graphical representation of the circuit and the cipher, respectively. Likewise, the subgraph G'_1 is the extracted influential portion (subgraph) of the circuit. Hence, this establishes the Subgraph isomorphism problem $\leq_P$ embedding problem. Next, we establish the subgraph isomorphism problem $\Leftrightarrow$ embedding problem.

$\Rightarrow$ Suppose G'_1 is isomorphic to G_2. This implies G'_1 is equivalent to G_2. Hence, G'_1 can be realized using components of G_2. Thus, if G'_1 and G_2 are isomorphic, then G'_1 can be successfully embedded in G_2.

$\Leftarrow$ Suppose a successful embedding exists for G'_1 into G_2. G'_1 can be realized using components of G_2, implying that G'_1 and G_2 are equivalent. Thus, G'_1 and G_2 are isomorphic. This shows that the embedding problem is NP-hard.

Next, we prove that the embedding problem belongs to NP. The embedding problem can be mapped into subgraph isomorphism problem in polynomial time. Since the subgraph isomorphism problem is NP-Complete, its certificate can be verified in polynomial time. Thus, a certificate of embedding G'_1 into G_2 should also be verified in polynomial time. This completes the proof[5]. So, the embedding problem using cipher graphical representations and the circuit is NP-complete. ■

Security Analysis Based on Logical Cryptanalysis: Before introducing a heuristic algorithm for embedding, we formally analyze the security of the key recovery problem in *LoPher*.

Block ciphers, which serve as the core of *LoPher*, are generally considered heuristically secure, as there is no formal mathematical proof of their security. Their construction is instead grounded in crucial cryptographic principles such as confusion and diffusion. In works [37, 38], the encryption process of a block cipher is formulated as a Boolean satisfiability (SAT) problem, with SAT solvers then applied to evaluate the cipher's robustness. Solving the SAT problem associated with the encryption process is effectively equivalent to a cryptanalytic attack where the goal is to recover the cipher's secret key using known plaintext-ciphertext pairs, a technique referred to as logical cryptanalysis [38]. However, if the block cipher is sufficiently secure and resistant to such

[5]The work in [36] reveals that graph isomorphism problem, while theoretically exponential in complexity, are often efficiently resolved due to the 'rarity' and 'fragility' of truly problematic cases. Our graph-based method leverages these insights, utilizing practical heuristics to adeptly tackle real-world scenarios effectively.

attacks, encoding its encryption algorithm as a SAT problem results in a hard, unsolvable instance of the SAT problem [39].

Formally, a block cipher can be represented as $\mathcal{B}(\mathcal{P}, \mathcal{K}, \mathcal{T})$, where $\mathcal{P}$ and $\mathcal{T}$ are known plaintext and its corresponding ciphertext, respectively. In logical cryptanalysis, the solver aims to extract the secret key $\mathcal{K}$ using plaintext-ciphertext $(\mathcal{P} - \mathcal{T})$ pairs. *LoPher* can be formally expressed as $\mathcal{B}'(\mathcal{P}', \mathcal{K}', \mathcal{T}')$, where $\mathcal{K}'$ includes the two sets of keys of the MUX layer and add round key $\mathcal{K}$ of $\mathcal{B}$.

The block cipher is transformed into a SAT problem in logical cryptanalysis, where each layer is expressed as a conjunctive normal form (CNF) formula. For *LoPher*, this process involves converting the intermediate layers of each round — such as addRoundKey, the MUX layer, the SBox layer, and the permutation layer into their corresponding propositional formulae [38]. The CNF formulae for the SBox and permutation layer are derived by leveraging the inherent relationships between the input and output bits of the cipher's SBox and permutation. In contrast, the CNF expressions for the MUX layer and addRoundKey are determined using the Tseytin transformation [40] applied to logic gates. Finally, the overall CNF for a round is constructed by combining the CNF formulae for all the intermediate layers.

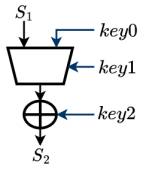


Figure 6: Parts of *LoPher* keys.

In the following lemma, we argue the hardness of key recovery from $\mathcal{B}'$.

Lemma 2. *The key recovery of LoPher ($\mathcal{B}'$) implies the key recovery of the underlying block cipher ($\mathcal{B}$).*

Proof. LoPher is constructed by augmenting the underlying block cipher ($\mathcal{B}$) with a keyed 2:1 MUX layer. The only additional keys introduced in *LoPher* are the bit that must be set at specific SBox inputs for the desired functionality as logic gates or switch boxes ($key0$), and the select line ($key1$) for each of the MUXes (ref. Fig. 6). S_1 represents the signal from the SBox at the upper layer, and S_2 denotes the binary bit set at the SBox input at the subsequent layer.

Finally, the round keys of the block cipher ($key2$) are generated for each round. The CNF representation of the keyed 2:1 MUX can be expressed as follows[6]:

$$(key1 \vee \overline{S_1} \vee S_2) \wedge (key1 \vee S_1 \vee \overline{S_2}) \wedge (\overline{key1} \vee \overline{key0} \vee S_2) \wedge (\overline{key1} \vee key0 \vee \overline{S_2}) \quad (5)$$

Upon setting $key1$ to 0, $key0$ is never selected, causing the bit from the previous round to pass through unaltered. In this scenario, the above equation (Eq. 6) simplifies to the following CNF formula:

$$(\overline{S_1} \vee S_2) \wedge (S_1 \vee \overline{S_2}) \quad (6)$$

which is the CNF representation of a buffer gate. Thus, fixing $key1$ to 0 effectively renders $key0$ redundant, simplifying the MUX layer in $\mathcal{B}'$ to a buffer gate. Under this configuration, the complexity introduced by the MUX layer disappears, and the cipher $\mathcal{B}'$ behaves identically to the underlying block cipher $\mathcal{B}$. Therefore, recovering the key from $\mathcal{B}'$ under logical cryptanalysis is at least as difficult as recovering the key from $\mathcal{B}$. This is because all other layers remain intact between the two structures ($\mathcal{B}'$ and $\mathcal{B}$) and the additional keys in $\mathcal{B}'$ further increase the complexity of the CNF equations. ■

[6]every layer consists of 64 such CNF equations considering PRESENT.

We now propose a heuristic algorithm for the same. The embedding of the extracted subgraph begins in a bottom-up manner. First, the root node ($N16$ in Fig. 5(a)) is embedded in an SBox layer. More specifically, we configure a chosen SBox ($SBoxR$) from the SBox layer of the block cipher and fix certain specific input bits such that $SBoxR$ starts behaving as the corresponding logic gate of the root node.

Example 3. *In Fig. 5(a), we show the root node $N16$ embedding of the extracted subgraph. $N16$ is a NAND gate. To embed $N16$ in PRESENT, we choose the 0^{th} SBox as $SBoxR$ from the SBox layer. When the last two input bits of $SBoxR$ are fixed to 1, the $SBoxR$ becomes a NAND gate, and we successfully embed the root node. The fixing of $1's$ in the last two input bits of $SBoxR$ is handled by Algo. 3.*

Before discussing the strategy of embedding the child nodes of the root node, let us look into the crucial data structure and variables required for designing Algo. 2. We will be referencing the sample embedding in Fig. 5 for describing each of the following.

- $SBox\_gates$ – It is the library that stores the input-output bit positions of SBoxes involved in realizing different logic gates and switches.
  Example: For NAND gate using a PRESENT SBox: $SBox\_gates$ = [Inputs = 1, 2; Output = 1]. It is interpreted as if NAND gate inputs are provided to the first two input bits of an SBox, the NAND gate output will be obtained from the first output bit of that SBox (ref. Fig. 5(a) inset).
- P_Layer – This data structure stores the bit-permutation mapping of the block cipher utilized for embedding.
  Example: The first or the 0^{th} SBox (let $SBox_0$) has input bits $\{0, 1, 2, 3\}$ and the P_Layer output of these bits of $SBox_0$ are $\{0, 16, 32, 48\}$ with PRESENT permutation.
- $SBox\_used\_layer$ – This data structure stores information of SBoxes that have been utilized in each layer for embedding the extracted subgraph. This is one of the outputs of Algo. 2.
  Example: $SBox\_used\_layer$ = $\{\mathbf{1} : [0, 0, 1, 0]\}$ after embedding the root node $N16$ in Fig. 5(a). Here, $\mathbf{1}$ denotes the $layer\_count$. The first value (i.e. 0) is the SBox number (here $SBoxR$), the second value and the third values are $SBox_0$ bit input positions for the NAND gate, and the last value is the output bit position that is the input to the next layer SBox input after passing through P_Layer.
  Likewise, $SBox\_used\_layer$ stores information for those specific SBoxes utilized in constructing gates and switches. For SBox as switches or buffers and NOT gates, the second input bit is marked as $' - 1'$ and $' - 2'$, respectively.
- $SBox\_used\_layer\_type$ – This data structure is used to store whether the utilized SBoxes in $SBox\_used\_layer$ are logic gate 'G' or simply a buffer or switch box 'D'. This is

another output of Algo. 2.
Example: $SBox\_used\_layer\_type = \{\mathbf{1} : [\text{'}G\text{'}]\}$ *on embedding* $N16$ *in PRESENT SBox layer.*

- Out – Variables that store the gate output bit positions of the SBox, which gets permuted to the input of the underlying SBox in the subsequent layer.
Example: In Fig. 5, $Out_1 = 0$ *and* $Out_2 = 1$*, the first output bit of the* 0^{th} *SBox (grey SBox embedding* $N11$ *in Fig. 5(b)) and the* 1^{st} *SBox (white in Fig. 5(b)), which serves the input bits to* $SBoxR$ *embedding* $N16$*, after application of PRESENT* P_Layer*.*
- $temp\_SBox$ – Variables that store the SBox number from which the output bit (Out) arises.
Example: in Fig. 5, $temp\_SBox1 = 0$*, and* $temp\_SBox2 = 1$ *since* Out_1 *arises from* 0^{th} *SBox and* Out_2 *arises from the* 1^{st} *SBox.*

Please note that there are other data structures and variables used in the algorithms. Their utility has been specified as comments alongside.

Next, we embed the child nodes of the root. The variables Out_1 and Out_2 hold the positions of the SBox output bits permuted to $SBoxR$, the SBox used for embedding the root node. Analyzing the $SBox\_gates$ library reveals that for an SBox implemented as a logic gate, the output is obtained through a specific fixed output bit. This leads to four cases for an SBox implemented as a gate in the subsequent layer (lines $15-26$, Algo. 2). If both Out_1 and Out_2 correspond to the SBox gate output of the desired gate for the child node to be embedded, we label them as $'G','G'$ in the $SBox\_used\_layer\_type$ data structure. However, if one or both Out_1 and Out_2 positions do not match the SBox gate output of the desired gate, we designate them as $'D'$. The search for SBoxes concludes when all gate child nodes in the extracted subgraph are embedded, or the total number of $'G's$ in $SBox\_used\_layer\_type$ equals the number of gate nodes in $Cone\_nodes$.

Example 4. *While embedding* $N11$ *in the extracted subgraph, we see that the bit position* Out_1 *and the SBox gate output bit position for the NAND gate is the same. Hence, we append a* $'G'$ *for the* $temp\_SBox1$ *in* $SBox\_used\_layer\_type$*. The* $temp\_SBox2$ *(marked as white SBox in Fig. 5(b)) is a simple switch box that we label as* $'D'$ *since it is a primary input and not any logic gate. This aids us in applying circuit inputs to the extracted subgraph at the same level in the cipher layer. The Final output of Algo. 2 after embedding the extracted Subgraph in PRESENT is* $SBox\_used\_layer = \{\mathbf{1} : [0,0,1,0], \mathbf{2} : [0,0,1,0,1,4,-1,1]\}$ *and* $SBox\_used\_layer\_type = \{\mathbf{1} : [\text{'}G\text{'}], \mathbf{2} : [\text{'}G\text{'}, \text{'}D\text{'}]\}$.

When two different SBox input bits are generated from the same SBox in the subsequent layer post-permutation, it leads to a "collision". Algo. 2 can handle collisions of SBoxes. The nature of the colliding SBoxes is examined in the event of a collision. A multiple fan-out switch box is employed in the subsequent layer if both are of type 'G' and use the same input signal. This 'M' type SBox allows routing a single signal to multiple SBox inputs and can manage multiple fan-outs in a circuit. Otherwise, if both SBoxes are 'G' but use different input signals, one is converted to 'D' to ensure the integrity of the already formed cipher layers below. The embedding of child nodes while avoiding collision is described in lines $28-51$ of Algo. 2. The aforementioned process is repeated until all the nodes or gates in the subgraph are placed. The final outputs of Algo. 2 are data structures $SBox\_used\_layer$ and $SBox\_used\_layer\_type$.

Fitness Assessment Function: The proposed heuristic algorithm achieves the optimal embedding of a circuit subgraph within a cipher, aiming to minimize the number of layers and reduce overhead. A fitness assessment function is incorporated to evaluate the feasibility of an embedding solution. The total number of layers needed for embedding a subgraph while preserving the cipher structure depends on the choice of the initial root SBox ($SBoxR$). Rather than exhaustively checking all possible $SBoxR$ choices, we minimize comparisons by considering d groups of SBoxes in a bit-permutation-based SPN cipher, where d represents the dimensions of the SBoxes [41]. This approach allows efficient determination of optimal embedding for a given $SBoxR$.

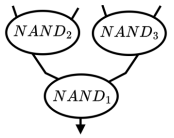


Figure 7: An extracted influential portion of NAND gates.

Example 5. *In Fig. 7, selecting* $SBoxR$ *as* $0, 4, 8,$ *or* 12 *in Algo. 2 necessitates three layers of PRESENT for embedding, whereas any other* $SBoxR$ *choice requires five layers. Thus, we can determine which one requires the minimum embedding layers by comparing the layer requirements of SBoxes* $0, 1, 2,$ *and* 3*. For example, if the* 0^{th} *SBox (*$SBoxR = 0$*) requires the minimum layers, SBoxes* $0, 4, 8,$ *and* 12 *will also require the minimum.*

It is easy to infer that different ciphers for embedding the subgraph are possible by simply feeding the framework with the SBox input-output bit mapping library ($SBox\_gates$) and the bit-permutation mapping (P_Layer) of the new cipher.

C. Stitching it Back

This is a generic step in the framework. For any logic locking technique, one has to integrate the locking circuitry with the original one. Stitching can be defined as the process of integrating the locking circuitry with the original circuit to produce the final locked circuit. We shall first discuss the stitching procedure of *LoPher* followed by the remaining locking techniques in MIDAS.

Once the extracted subgraph embedding with minimum layers, as suggested by the fitness function, has been found, the framework proceeds to stitch back the locked portion to the host circuit. Stitching in *LoPher* implies connecting the input and output wires of the obfuscated block with corresponding open signals in the original netlist without the influential portion. A challenge exists in the case of *LoPher*: the subgraph output is a one-bit signal (or a few bits considering multiple fan-outs from upper layer nodes), whereas each round of block cipher produces a fixed number of signals, for example, PRESENT produces a 64-bits signal in each round.

For the correct functioning of the obfuscated circuit, the dimensions of the input/output of the obfuscated block and

Algorithm 2: Algorithm for embedding the extracted influential portion into the Cipher

Input: influential portion nodes // sub-graph extracted using Algo. 1
P_Layer // permutation layer of the Cipher
$SBox\_gates$ // input-output bit configuration for various Logic gate, switches, from the SBox
Output: $SBox\_used\_layer$, $SBox\_used\_layer\_type$
Initialization:
Assign: $layer\_count \leftarrow 1$
Define: $SBox\_used\_layer$ // data structure (DS) that stores the #SBox details used in each layer
Define: $SBox\_used\_layer\_type$ // DS that stores SBox used whether as Gate 'G' or Switch box 'D'
/* Embedding is done in bottom-up manner */
$SBoxR \leftarrow$ #SBox for embedding the root node gate // for optimizing, $SBoxR$ can be changed
$SBox\_used\_layer[layer\_count] \leftarrow SBoxR$ // #SBox configured as root node gate
$SBox\_used\_layer\_type[layer\_count] \leftarrow$ 'G'
$layer\_count++$
$temp\_SBox1 \leftarrow$ #SBox whose output bit (Out_1) is the $SBoxR$ first input bit, after applying P_Layer
$temp\_SBox2 \leftarrow$ #SBox whose output bit (Out_2) is the $SBoxR$ second input bit, after applying P_Layer
/* following SBox configurations for Logic gate are obtained from $SBox\_gates$ */
if $Out_1 \leftarrow$ *SBox gate configuration output bit* && $Out_2 \leftarrow$ *gate configuration output bit* **then**
  $SBox\_used\_layer\_type[layer\_count] \leftarrow$ 'G', 'G'
end
else if $Out_1 \leftarrow$!*(SBox gate configuration output bit)* && $Out_2 \leftarrow$ *SBox gate configuration output bit* **then**
  $SBox\_used\_layer\_type[layer\_count] \leftarrow$ 'D', 'G'
end
else if $Out_1 \leftarrow$ *SBox gate configuration output bit* && $Out_2 \leftarrow$!*(SBox gate configuration output bit)* **then**
  SBox_used_layer_type[layer_count] $\leftarrow$ 'G', 'D'
end
else
  SBox_used_layer_type[layer_count] $\leftarrow$ 'D', 'D'
end
SBox_used_layer[layer_count] $\leftarrow temp\_SBox1, temp\_SBox2$
while !*(all nodes placed)* **do**
  if !*(collision in* $SBox\_used\_layer[layer\_count]$*)* **then**
    choose #SBox from $SBox\_used\_layer[layer\_count]$ as initial SBox // choice is done based on the topology of the nodes
  end
  else
    if *Colliding SBoxes operate on the same signal* **then**
      Use multiple fan-out switch SBox configurations for the input bits
      Add switch SBox for all remaining SBoxes in $SBox\_used\_layer[layer\_count--]$
    end
    else if *Colliding SBoxes operate on different signal* **then**
      if *Both SBoxes are 'G'* **then**
        Change one of the 'G' SBoxes in $SBox\_used\_layer[layer\_count--]$ to 'D' and form that 'G' SBox in $SBox\_used\_layer[layer\_count]$
      end
      else if *One SBox is 'G'* **then**
        Prioritize the 'G' SBox input bit // first assign 'D' SBox for gate input
        Add 'D' SBox for all remaining SBoxes in $SBox\_used\_layer[layer\_count--]$
      end
      else
        Change the input bit positions of anyone 'D' SBox in $SBox\_used\_layer[layer\_count--]$ layer such that there is no collision in the current layer.
      end
    end
  end
  repeat steps 12 - 27
end

Algorithm 3: Key Generation Algorithm

Input: $SBox\_used\_layer$ // SBox details used in each layer
$SBox\_used\_layer\_type$ // SBox used whether as Gate or Switch box
$SBox\_gates$ // input-output bit configuration for various Logic gate, switches, from the SBox
$SBox\_Bit\_Config$ // SBox bit configuration for logic gates and switch box
Output: $Key_f[i]$ // Keys for each layer i
Define: $key0$ // configuration bits for gate and switch boxes for SBox
Define: $key1$ // Select line for each MUX
Define: $key2$ // round keys for each round of the block cipher
for ***each*** i *in # layers* **do**
  for ***each*** j *in* $SBox\_used\_layer[i]$ **do**
    if $SBox\_used\_layer\_type[i][j] ==$ *Gate* **then**
      $assign\ key0[i][j] \leftarrow$ Configuration bits for that Gate from $SBox\_Bit\_Config$
      $assign\ key1[i][j] \leftarrow 1$
    end
    else if $SBox\_used\_layer\_type[i][j] ==$ *switch gate* **then**
      Input_index $\leftarrow$ input bits of the switch box
      Output_index $\leftarrow$ output bits of the switch box
      $assign\ key0[i][j] \leftarrow$ Configuration bits for that switch box from $SBox\_Bit\_Config$
      $assign\ key1[i][j] \leftarrow 1$
    end
  end
  for ***each*** j *not in* $SBox\_used\_layer[i]$ **do**
    $assign\ key0[i][j] \leftarrow rand(0,1)$
    $assign\ key1[i][j] \leftarrow 0$
  end
end
for ***each*** i *in # layers* **do**
  $assign\ key2[i] \leftarrow round\_key_{block\_cipher}[i]$
  $Key_f[i] \leftarrow (key0[i]||key1[i]||key2[i])$ // The final key for each layer of *LoPher*
end

the open signals should be equal; only then can successful stitching of the circuit be executed. A key-based AND gate layer addresses the same [7]. It helps to propagate only those signals which are of relevance. This simple yet effective solution to the aforementioned challenge reduces the output signals of the block cipher to the desired dimension as required for integrating the same with the host circuit.

For the other locking techniques implemented in MIDAS (*AntiSAT, CAS-Lock, and SFLL-HD*), first, the *influential nodes* are identified using Algo. 1. Locking circuitry are produced following these locking strategies. Then the output of the locking circuitry is stitched with these nodes. For *EPIC*, XOR/XNOR-based key gates are integrated with *influential nodes* instead of randomly inserting them throughout the target circuit. Sec. V presents the advantage of integrating the locking circuitry with the *influential nodes*.

D. Key Generation

This is also a generic step in the framework. The key generation process for locking techniques *EPIC, AntiSAT, CAS-Lock, SFLL-HD, and SFLL-fault* is straightforward as they do not require any post-processing of the locking circuitry for the secret key generation. However, in *LoPher*, the embedding algorithm generates two data structures $SBox\_used\_layer$ and $SBox\_used\_layer\_type$, which are post-processed to generate the secret key of *LoPher*. We now elaborate on the key generation technique.

In *LoPher*, the extracted sub-circuit is embedded in unrolled layers of the block cipher, necessitating layerwise secret keys. The number of layers depends on the gate type and topology of the desired sub-circuit. Algo. 3 summarises the key generation process of *LoPher*. It takes input the data structure $SBox\_Bit\_Config$, which stores the binary bits that must be set to SBox inputs for it to behave as the desired gate. *Example:* $SBox\_Bit\_Config = [1,1]$ *for NAND and* $SBox\_Bit\_Config = [1,0]$ *for NOR using PRESENT SBox.*

The bit positions which need to be fixed can be obtained from the $SBox\_gates$ library, which is also an input to the algorithm. It creates the first two parts of the secret key by traversing the layerwise information of the embedding ($SBox\_used\_layer$ and $SBox\_used\_layer\_type$) of the extracted subgraph generated by Algo. 2. The remaining bits of $key0$ and $key1$ are assigned with randomly chosen binary bits and 0, respectively (lines $22-25$, Algo. 3).

The three variables $key0, key1$, and $key2$ store the configuration bits, select line bits, and the round keys of the block cipher, respectively. The final key ($Key_f[i]$) for each layer i results from appending the three parts $key0, key1$, and $key2$. Next, we discuss a case study of locking a RISC-V processor utilizing the MIDAS framework.

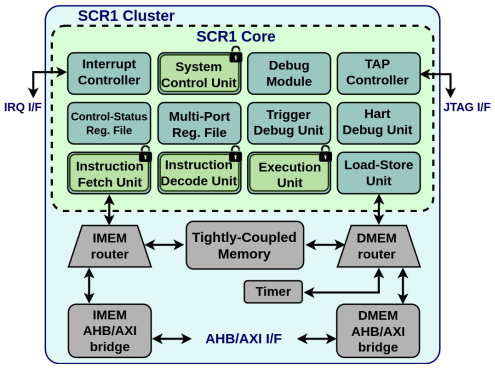


Figure 8: RISC-V Core overview [42] and locked modules.

IV. Case Study: Locking RISC-V

This section demonstrates the application of *LoPher* on a commercial RISC-V processor core, specifically the *Syntacore* RISC-V core [42]. The chosen core is industry-grade, silicon-proven, and compatible with major EDA flows and Verilator. SCR1 features a primary in-order pipeline, configurable into 2, 3, or 4 stages based on frequency requirements. Fig. 8 illustrates the processor cluster, highlighting the SCR1 core. We used the *Dhrystone* test suite binary file included in the *Syntacore* SCR1 repository [42] to test the processor core's integrity. We explored the design space of crucial RISC-V core modules to assess potential locking targets. The

Design Description	LUTs	FFs	DSPs	BRAMs
Unlocked SCR1-Core	**21594**	**20009**	**4**	**32**
SCR1-Core with IFU Locked	21978	20009	4	32
SCR1-Core with SCU Locked	22038	20009	4	32
SCR1-Core with IEU Locked	21978	20009	4	32
SCR1-Core with IDU Locked	21818	20009	4	32

Table II: Description of Resource Consumption when the SCR-1 RISC-V core is implemented on Nexys4-DDR board.

focus was on the *Instruction Fetch Unit (IFU)*, *Instruction Decoder Unit (IDU)*, *Instruction Execution Unit (IEU)*, and *System Control Unit (SCU)*. We fed gate-level netlists of these modules into our framework, which identified influential combinational blocks within each module. To ensure integrity, we replaced each module's behavioural Verilog file with a gate-level netlist and tested input-output characteristics using a separate testbench.

We replaced the original gate-level netlist of the module with the logic-locked netlist generated by the framework. The interface of the locked module now includes key bits. No additional memory elements were used to store the key during experimental verification. When the correct key is entered, the SCR bootloader prompt appears, confirming the RISC-V core's integrity. With an incorrect key, the prompt does not appear, indicating the core remains locked. Application of SAT-based attacks [1, 43] on the synthesized locked modules, utilizing the original module as an oracle, failed to extract the secret keys within a 2-day time-out period. As discussed in [7], the high output corruptibility of [7] makes the addition of *Bypass* circuitry infeasible. Further, *Removal* of the *LoPher* locking circuitry results in an unconnected and non-functional recovered netlist. *FALL* is not applicable due to the absence of hard-coded keys in its designed locking netlists.

As already suggested, using our framework, we have attempted to lock four different modules *IFU, IDU, IEU)*, and *SCU* as highlighted in Fig. 8. The hardware resource utilization of implementing locked RISC-V core in FPGA is presented in Tab. II. Please note that the resource consumption of the RISC-V core, as depicted in Tab. II, is shown in terms of LUTs, FFs, DSPs, and BRAMs. We synthesized the RISC-V core [42] on a Digilent Nexys4-DDR board, which houses an Artix-7 FPGA. However, the RISC-V core consists of Xilinx-specific Hard-IPs, so it could not be synthesized with ASIC libraries. However, we have synthesized the individual modules with our 180nm SCL library before and after locking with *LoPher* to convey an idea about resource consumption

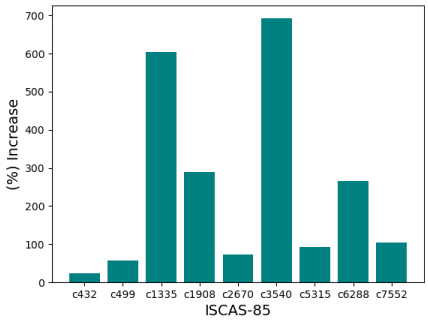


Figure 9: Percentage increase in time required (in second) to perform a successful SAT attack [1] for the original and our proposed method for insertion.

circuits	s1423	s5378	s9234	s13207	s15850	s35932	s38584
original	70	675	1549	702	458	7808	880
LoPher	2296	2766	2298	1931	2299	1386	2258

Table III: Gate equivalents of ISCAS-89 circuits locked using MIDAS with *LoPher*, synthesized with Cadence Genus tool.

regarding gate equivalents (GE). For example, GE before and after locking the *SCU* is 4679 and 7622, respectively.

V. Experimental Validation

In this section, we evaluate the MIDAS framework. Although the work's primary focus is the implementation aspect of *LoPher*, we demonstrate the framework's effectiveness across different locking methodologies belonging to the prominent genres of logic locking. We have implemented random or XOR/XNOR-based logic locking (*EPIC*) [15], a pre-SAT locking technique. We have then implemented *Anti-SAT [4], SFLL-HD [5], and CAS-Lock [18]*, which are advanced post-SAT point function-based locking techniques. Finally, the post-SAT locking technique *LoPher* [7], for which there are no reported attacks, has been implemented using MIDAS. It belongs to the SAT-hard genre of post-SAT logic locking. The locking scripts have been implemented using Python programming language. The graphical representation of the circuit, crucial nodes, and sub-circuit extraction has been performed using networkX and Graphviz Python packages.

The locked netlists were optimized using *ABC* [44] synthesis and verification framework for obtaining the locked designs in BENCH format. The locked circuits were re-synthesized, and the area overheads were calculated using industrial tools like the Cadence Genus. The framework can handle input design files specified in both BENCH and structural Verilog formats. Two sets of locked benchmarks have been prepared to demonstrate the framework's efficacy. The first set has been locked by following the strategy proposed in Sec. III-A. It inserts the locking circuitry on influential nodes. On the other set, it has been inserted following the approach presented in the corresponding publications. This aids us in comparing the results obtained for extracting the secret key in terms of timing requirements. Please note that these locking techniques have already been broken. So, we compare the timing requirements to extract the secret keys. We have implemented them just to highlight the generality of the framework.

Circuit	c432	c499	c880	c1355	c1908	c2670	c3540	c5315	c6288	c7552
Original	158	340	328	342	363	540	969	1343	2201	1387
AntiSAT	287	580	555	600	642	786	1245	1594	2386	1664
CAS-Lock	291	588	568	609	644	794	1254	1589	2391	1646
LoPher	2160	2863	2261	2304	3795	1930	2863	2295	2308	2298

Table IV: The gate equivalents of the synthesized locked circuit of ISCAS-85 with *Anti-SAT* and *CAS-Lock* with 64-bit key, and *LoPher*, utilizing Cadence Genus Synthesis tool.

Lib.	Circuit	c432	c499	c880	c1355	c1908	c2670	c3540	c5315	c6288	c7552
Restricted	**Original**	179	367	476	382	452	621	1281	1662	2964	1870
	AntiSAT	313	646	720	644	735	894	1415	1948	3255	2153
	CAS-Lock	331	701	761	703	817	936	1511	2163	3283	2199
	SFLL	1208	1327	1406	1340	1493	2149	2585	4187	4934	4648
	LoPher	2204	2784	2338	2382	3019	2441	3141	3454	5468	5912
Un-Restrict.	**Original**	297	619	476	649	459	538	735	1727	1774	1799
	AntiSAT	425	813	561	808	642	759	812	1735	2066	1913
	CAS-Lock	442	822	574	824	645	811	946	1820	2068	1775
	SFLL	986	1039	1082	1051	1098	1747	1864	2298	3519	3358
	LoPher	1513	1589	1857	1949	1659	2202	2583	3587	4656	4889

Table V: Gate equivalents (GE) when synthesized with different libraries having restricted and un-restricted (all) gates.

Fig. 9 presents the percentage increase in the time required for extracting the secret key by applying the SAT-based attack [1] on ISCAS-85 benchmark circuits locked with *EPIC* [15]. There is a significant increase in the time required to extract the secret key using our technique. This is attributed to the greater number of SAT variables, clauses, and subsequent Boolean decisions needed for representing locked circuits in CNF form. In most cases, the time required to extract the correct key is more than double the existing duration. The overheads for *LoPher*, in terms of the number of GE, have been presented in Tab. IV and Tab. III, for ISCAS-85 [45] and ISCAS-89 [46] benchmarks, respectively. It conforms with the claim that overhead for *LoPher* remains fairly constant [7] through benchmark circuits.

Next, we implement the *SFLL-HD* locking technique. The output of *SFLL-HD* locking circuitry is stitched with the crucial nodes and is subjected to the *FALL* attack that could successfully extract the *SFLL-HD* key. Tab. VII compares the timing and the overhead area detail of the two sets of locked circuits. It can be observed that the *FALL* attack requires more time to resolve the correct key by analyzing the circuit for our case compared to the existing locked benchmarks.

We demonstrate that inserting stuck-at faults at influential nodes leads to an increased occurrence of failing patterns compared to random insertion strategies. MIDAS strategically identifies and targets influential nodes for applying the locking, which improves practical robustness at a lower cost. The traditional *SFLL-fault* technique inserts faults across all nodes in a target netlist to identify the suitable target location [17]. Tab VIII presents a detailed comparison of the failing patterns observed in different subgraphs of the benchmark circuits.

Hence, our proposed technique of choosing suitable targets or locations in a circuit is found to be effective compared to the existing insertion strategy of locking circuitry by increasing the time required significantly. Further, the locking strategies *AntiSAT* and *CAS-Lock* have been implemented. Tab. IV lists the overhead detail of the same.

We now discuss the impact of logic synthesis on the number of *influential nodes* and overheads of the locked circuits.

Effect of Logic Synthesis and Optimizations. We investigated how logic synthesis affects the number of *influential nodes* within circuits, employing various technology libraries and synthesis optimization commands. The primary aim of these optimizations was to assess their impact on extracted *influential nodes* in a circuit. As each synthesis process succeeded, the circuit's graph structure alteration significantly affected the number of influential nodes. We synthesized the benchmark circuits using two distinct libraries featuring different logic gate types. One library encompassed all possible gates, while the other was constrained to fundamental gates such as AND, OR, NOT, and XOR. Tab. IX (Col. 2 and 5) overviews the influential nodes' count for ISCAS-85 benchmarks synthesized utilizing two distinct libraries.

Lib.	Circuit	c432	c499	c880	c1355	c1908	c2670	c3540	c5315	c6288	c7552
Un-Restrict.	**Original**	0.147	0.154	0.195	0.151	0.293	0.265	0.327	0.895	0.929	0.908
	AntiSAT	0.149	0.157	0.203	0.152	0.306	0.322	0.334	0.932	0.983	1.001
	CAS-Lock	0.150	0.157	0.216	0.154	0.311	0.358	0.366	0.927	0.992	0.955
	SFLL	0.152	0.155	0.207	0.157	0.328	0.321	0.338	0.919	0.996	1.041
	LoPher	0.155	0.178	0.239	0.164	0.374	0.399	0.385	1.027	1.039	1.083
Restricted	**Original**	0.152	0.160	0.205	0.153	0.311	0.338	0325	0.954	0.964	0.752
	AntiSAT	0.153	0.160	0.206	0.154	0.311	0.342	0.337	1.001	0.988	1.014
	CAS-Lock	0.153	0.159	0.216	0.154	0.316	0.357	0.369	0.955	0.998	0.868
	SFLL	0.154	0.158	0.198	0.161	0.297	0.375	0.329	0.984	1.001	0.781
	LoPher	0.156	0.181	0.234	0.184	0.308	0.401	0.353	1.035	1.024	1.041

Table VI: Dynamic power consumptions (in watt) when synthesized with un-restricted (all) and restricted gate libraries.

Circuit	*FALL* Attack (sec)		Area Overhead (GE)	
	Existing	This work	Existing	This work
c432	5.4	6.4	707	753
c880	6.0	6.7	897	830
c1355	7.7	10.5	825	915
c1908	6.9	8.7	863	918
c2670	6.5	7.9	1278	1324
c3540	7.0	10.8	1565	1570
c5315	11.4	10.0	1688	1765
c6288	16.0	18.6	2642	2696
c7552	11.2	13.0	1678	1666

Table VII: SFLL implementation details with 32−bit key, utilizing MIDAS. GE represents gate equivalents of synthesized locked circuit utilizing the Cadence Genus Synthesis tool.

Circuits	c17	c432	c3540	c5315	c6288	c7552
Random	20/32	24/64	36/128	40/128	24/64	36/128
MIDAS	24/32	36/64	64/128	72/128	48/64	96/128

Table VIII: Failing patterns of extracted subgraphs in SFLL-fault for random and Influential Nodes-based insertion of stuck-at-0 faults.

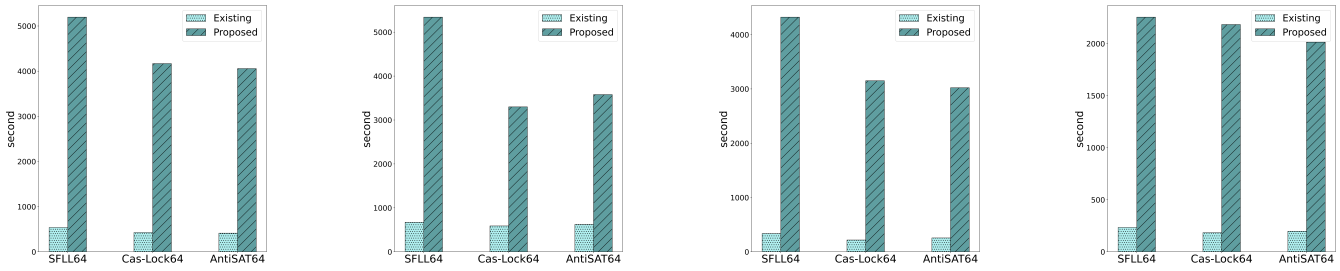
To examine the impact of synthesis commands, we synthesized the benchmarks using the `"synthesize -to_mapped -effort high"` and `"synthesize -to_mapped -effort low"` commands in Cadence Genus. The resulting variations in the count of *influential nodes* are presented in Tab. X.

Choice of Circuit Location to Lock. In Tab. IX, we also present the variations in the overall count of influential nodes, impacting all outputs and a randomly chosen 50% of the total outputs for the circuits. The designer retains control over the percentage of specific output nodes targeted for locking and can guide the algorithm based on their domain-specific knowledge of which parts to lock. It further illustrates the decrease in runtime associated with identifying influential nodes only for selected circuit outputs compared to all outputs.

MIDAS is designed to complement the designer's expertise rather than replace it. Our framework provides an automated analysis to identify critical nodes, enhancing security through a systematic and thorough approach. The designer controls the overall locking strategy and can guide the algorithm based

ISCAS-85 Circuits	Un-Restricted Library			Restricted Library		
	All Outputs	Selected Outputs	Time Decrease (%)	All Outputs	Selected Outputs	Time Decrease (%)
c432	12	14	**47.81**	35	35	**65.06**
c499	27	42	**52.46**	80	82	**36.83**
c880	1	4	**93.19**	1	3	**41.47**
c1355	113	114	**53.26**	80	81	**61.56**
c1908	85	113	**47.96**	100	100	**69.92**
c2670	4	4	**11.39**	3	6	**37.18**
c3540	6	7	**76.83**	9	10	**80.09**
c5315	2	2	**85.86**	7	17	**55.95**
c6288	3	3	**22.16**	5	7	**38.32**
c7552	1	3	**43.59**	2	3	**77.29**

Table IX: Number of *Influential nodes* that affect all outputs and selected outputs to lock, when ISCAS-85 benchmark circuits are synthesized with un-restricted (all) and restricted gate libraries. The % decrease shows the reduction in time required to find the *Influential nodes* for selected outputs compared to all outputs present in a target circuit.



(a) c7552 (b) c6288 (c) c5315 (d) c3540

Figure 10: Time (in seconds) taken by the *Valkyrie* security diagnostic tool to identify the vulnerabilities present in the circuits locked with *SFLL-HD, CAS-Lock,* and *AntiSAT*.

on their domain-specific knowledge of which parts to lock. As evident from the experimental results furnished in Tab. IX and the input requirement of Algorithm 1 (*Target Outputs*), MIDAS allows the designer to select specific outputs to lock instead of locking all outputs. Similarly, MIDAS enables locking specific subcircuits within a larger circuit, as demonstrated in our Case Study (Sec. IV), where it is applied to lock chosen combinational blocks within RISC-V modules. The complexity of modern circuits can make it challenging for designers to identify all potential critical nodes manually. Our algorithmic approach can systematically analyze the circuit to uncover vulnerabilities that might be overlooked due to the intricate nature of the design. This combined approach ensures that the most critical parts are securely locked.

Security Evaluation with *Valkyrie* Tool. We conducted a vulnerability analysis of circuits locked with the proposed *influential nodes* compared to the existing integration methods. Fig. 10 illustrates the time taken to identify structural vulnerabilities for *SFLL-HD, CAS-Lock*, and *AntiSAT*, referred to as Flip Locking Techniques (FLT) in [2], for both approaches. Circuits locked with the proposed method require a much longer execution time than conventional methods, attributed to the complexity and ambiguity of identifying structural vulnerabilities [2], reinstating the robustness of our approach. Please note that *LoPher* is not an FLT; hence, unlike other FLTs, it is not vulnerable to *Valkyrie*.

We employed MIDAS to lock the circuits synthesized using various libraries through different locking techniques. The locked circuits' corresponding GE and power consumptions are outlined in Tab. V and Tab. VI, respectively.

The MIDAS framework can simultaneously lock an individual netlist using various locking techniques. We implemented this by employing a straightforward method that involves locking different *influential nodes* with distinct locking schemes. Tab. XI details the overhead of the circuit locked using a combination of *SFLL-HD*, *CAS-Lock*, and *Anti-SAT*.

Opti.	Circuit	c432	c499	c880	c1355	c1908	c2670	c3540	c5315	c6288	c7552
High Low	All Gates	35	116	5	114	91	7	6	7	4	4
	Restricted	14	80	6	83	93	9	11	10	7	5
	All Gates	34	29	1	109	83	4	5	3	3	1
	Restricted	12	79	1	77	90	2	7	7	5	2

Table X: Number of *Influential nodes* for different optimization efforts (*high* and *low*), in *Cadence Genus Synthesis tool.*

Library	No Restriction				Restricted Gates			
Circuits	c3540	c5315	c6288	c7552	c3540	c5315	c6288	c7552
Area (GE)	3213	3357	3536	3765	4539	4619	5261	4813
Power (W)	0.204	0.987	0.896	0.726	0.298	1.018	0.902	0.877

Table XI: Overhead in terms of GE and Dynamic power consumption, when locked with a combination of *CAS-Lock*, *SFLL-HD*, and *AntiSAT* applied at different positions of the target circuits.

VI. Discussion

While formal locking theories, such as those discussed in [47–51], provide a solid foundation, our primary focus with MIDAS is on practical implementation aspects, addressing real-world challenges. Recent advancements, including the *Valkyrie* tool, have emphasized the need to consider practical attack scenarios and theoretical guarantees, as they reveal insights into the applicability of provably secure logic locking (PSLL) techniques.

Techniques like row-activated LUTs [51] offer a pathway to achieve exact-functional secrecy (EFS)[7] by leveraging their functional versatility. Approaches such as *Meerkat* [50], which utilize reduced ordered binary decision diagrams (ROBDD) for canonical representations to achieve indistinguishability obfuscation (iO), showcase the potential for scalable and secure obfuscation methodologies when aligned with efficient implementation techniques. These developments collectively emphasize the importance of balancing theoretical precision with practical feasibility, as MIDAS aims to achieve.

Quasi-universal circuits, achieved by fixing gate functionality and universal routing, have been proposed for designing indistinguishability Logic Obfuscation (iLO). It offers strong theoretical guarantees by implementing every possible depth d and size s function. This ensures that an attacker learns nothing beyond these bounds, making UCs a highly expressive and robust logic locking.

While ciphers may theoretically restrict the space of possible functions more than UCs, they provide robust defenses in practice through well-analyzed security primitives, such as S-Boxes and permutation layers. These trade-offs between theoretical and practical security highlight the need for designs that balance universality with real-world implementability. Building on this foundation, *LoPher* achieves iLO by combining the universality of circuit structures with cryptographic primitives, leveraging SBoxes and permutation layers to enhance resilience against real-world attacks. This approach exemplifies a practical yet secure solution for modern obfuscation challenges.

[7]EFS is achieved when the attacker cannot determine the original circuit's precise functionality. EFS permits arbitrary approximations of the original circuit [51].

VII. Conclusion

In this work, we present the framework MIDAS which can implement combinational pre-SAT and post-SAT point functions and SAT-hard logic locking techniques. MIDAS can identify a crucial portion of the circuit based on a simple colour-based analysis and lock the same to maximize the effect of the implemented logic locking. For the first time, a SAT-hard locking technique *LoPher* has been implemented in a logic locking framework. We present a heuristic algorithm for embedding a circuit subgraph into the cipher netlist graph. We show the versatility of MIDAS by incorporating locking techniques belonging to different genres of logic locking. We validate MIDAS on a couple of benchmark suites and report the implementation of existing attacks on those logic locking schemes. Further, we have presented a case study of the application of MIDAS with an open-source commercially designed RISC-V processor.

References

[1] P. Subramanyan and et al., "Evaluating the security of logic encryption algorithms," in *HOST*. IEEE, 2015, pp. 137–143.

[2] N. Limaye and et al., "Valkyrie: Vulnerability assessment tool and attack for provably-secure logic locking techniques," *IEEE Trans. on Information Forensics and Security*, vol. 17, pp. 744–759, 2022.

[3] M. Yasin and et al., "Sarlock: Sat attack resistant logic locking," in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2016, pp. 236–241.

[4] Y. Xie and A. Srivastava, "Anti-sat: Mitigating sat attack on logic locking," *IEEE Trans. CAD*, vol. 38, no. 2, pp. 199–207, 2018.

[5] M. Yasin and et al., "Provably-secure logic locking: From theory to practice," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1601–1618.

[6] H. M. Kamali and et al., "Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks," in *2019 56th Annual Design Automation Conference (DAC)*, pp. 1–6.

[7] A. Saha and et al., "Lopher: Sat-hardened logic embedding on block ciphers," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, pp. 1–6.

[8] ——, "Dip learning on cas-lock: Using distinguishing input patterns for attacking logic locking," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 688–693.

[9] X. Xu and et al., "Novel bypass attack and bdd-based tradeoff analysis against all known logic locking attacks," in *International conference on cryptographic hardware and embedded systems*. Springer, 2017, pp. 189–210.

[10] M. Yasin and et al., "Security analysis of anti-sat," in *ASP-DAC*. IEEE, 2017, pp. 342–347.

[11] D. Sirone and P. Subramanyan, "Functional analysis attacks on logic locking," *IEEE Trans. on Information Forensics and Security*, vol. 15, pp. 2514–2527, 2020.

[12] M. Yasin and et al., "Removal attacks on logic locking and camouflaging techniques," *IEEE Trans. on Emerging Topics in Computing*, 2017.

[13] L. Alrahis and et al., "Gnnunlock: Graph neural networks-based oracle-less unlocking scheme for provably secure logic locking," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, pp. 780–785.

[14] ——, "Muxlink: Circumventing learning-resilient mux-locking using graph neural network-based link prediction," in *2022 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, pp. 694–699.

[15] J. A. Roy and et al., "Epic: Ending piracy of integrated circuits," in *Proceedings of the conference on Design, automation & test in Europe*, 2008, pp. 1069–1074.

[16] A. Alaql and S. Bhunia, "Saro: Scalable attack-resistant logic locking," *IEEE Trans. on Information Forensics and Security*, vol. 16, pp. 3724–3739, 2021.

[17] A. Sengupta and et al., "Atpg-based cost-effective, secure logic locking," in *2018 IEEE 36th VLSI Test Symposium (VTS)*. IEEE, 2018, pp. 1–6.

[18] B. Shakya and et al., "Cas-lock: A security-corruptibility trade-off resilient logic locking scheme," *IACR Trans. on Cryptographic Hardware and Embedded Systems*, pp. 175–202, 2020.

[19] A. Sengupta and et al., "Breaking cas-lock and its variants by exploiting structural traces," *IACR Trans. on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 3, pp. 418–440, 2021.

[20] K. Z. Azar and et al., "Nngsat: Neural network guided sat attack on logic locked complex structures," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.

[21] S. Engels and et al., "The end of logic locking? a critical view on the security of logic locking," *Cryptology ePrint Archive*, 2019.

[22] A. Jain and et al., "Atpg-guided fault injection attacks on logic locking," in *2020 IEEE Physical Assurance and Inspection of Electronics (PAINE)*. IEEE, 2020, pp. 1–6.

[23] ——, "Taal: tampering attack on any key-based logic locked circuits," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 26, no. 4, pp. 1–22, 2021.

[24] A. Bogdanov and et al., "Present: An ultra-lightweight block cipher," in *9th Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2007, pp. 450–466.

[25] L. Mankali and et al., "Vigilant: Vulnerability detection tool against fault-injection attacks for locking techniques," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2023.

[26] ——, "Titan: Security analysis of large-scale hardware obfuscation using graph neural networks," *IEEE Trans. on Information Forensics and Security*, 2022.

[27] N. Albartus and et al., "Dana-universal dataflow analysis for gate-level netlist reverse engineering," *Cryptology ePrint Archive*, 2020.

[28] T. Meade and et al., "Gate-level netlist reverse engineering for hardware security: Control logic register

identification," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, pp. 1334–1337.

[29] J. Geist and et al., "Relic-fun: Logic identification through functional signal comparisons," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, pp. 1–6.

[30] W. Li and et al., "Wordrev: Finding word-level structures in a sea of bit-level gates," in *2013 IEEE international symposium on hardware-oriented security and trust (HOST)*. IEEE, pp. 67–74.

[31] M. Fyrbiak and et al., "Hal—the missing piece of the puzzle for hardware reverse engineering, trojan detection and insertion," *IEEE Trans. on Dependable and Secure Computing*, vol. 16, no. 3, pp. 498–510, 2018.

[32] J. Geist and et al., "Netviz: A tool for netlist security visualization," in *2023 24th International Symposium on Quality Electronic Design (ISQED)*. IEEE, pp. 1–8.

[33] R. Karmakar and et al., "A new logic encryption strategy ensuring key interdependency," in *2017 International Conference on VLSI Design*. IEEE, pp. 429–434.

[34] S. Dupuis and M.-L. Flottes, "Logic locking: A survey of proposed methods and evaluation metrics," *Journal of Electronic Testing*, vol. 35, pp. 273–291, 2019.

[35] D. Sisejkovic and R. Leupers, "Security metrics: One problem, many dimensions," in *Logic Locking: A Practical Approach to Secure Hardware*. Springer, 2022.

[36] M. Anastos M., Kwan and M. B., "Smoothed analysis for graph isomorphism," 2024. [Online]. Available: https://arxiv.org/abs/2410.06095

[37] M. Gwynne and O. Kullmann, "Towards a better understanding of hardness," in 17^{th} *International Conference on Principles and Practice of Constraint Programming*, 2011, pp. 37–42.

[38] F. Massacci and L. Marraro, "Logical cryptanalysis as a sat problem," *Journal of Automated Reasoning*, vol. 24, pp. 165–203, 2000.

[39] S. Cook and D. Mitchell, "Finding hard instances of the satisfiability problem: A survey." *Satisfiability Problem: Theory and Applications*, pp. 1–17, 1996.

[40] G. S. Tseitin, "On the complexity of derivation in propositional calculus," *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pp. 466–483, 1983.

[41] S. Patranabis and et al., "SCADFA: Combined SCA+ DFA attacks on block ciphers with practical validations," *IEEE Trans. Computers*, 2019.

[42] Syntacore. SCR1 RISC-V Core. https://github.com/syntacore/scr1.

[43] K. Shamsi and et al., "AppSAT: Approximately deobfuscating integrated circuits," in *2017 HOST*. IEEE, 2017, pp. 95–100.

[44] B. L. Synthesis and V. Group, "ABC: A system for sequential synthesis and verification, release 10216," 2018. [online] http://www.eecs.berkeley.edu/ alanmi/abc/.

[45] F. Brglez and et al., "A neutral netlist of 10 combinational benchmark circuits and a target translator," in *Fortran. ISCAS'85*, 1985.

[46] ——, "Combinational profiles of sequential benchmark circuits," in *Circuits and Systems, 1989., IEEE International Symposium on*, May 1989, p. vol.3.

[47] K. Shamsi and et al., "On the impossibility of approximation-resilient circuit locking," in *IEEE International Symposium on Hardware Oriented Security and Trust*. IEEE, 2019, pp. 161–170.

[48] H. Zhou and et al., "Resolving the trilemma in logic encryption," in *IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 2019, pp. 1–8.

[49] P. Beerel and et al., "Towards a formal treatment of logic locking," *Cryptology ePrint Archive*, 2022.

[50] M. Massad and et al., "Logic locking for secure outsourced chip fabrication: A new attack and provably secure defense mechanism," *arXiv preprint arXiv:1703.10187*, 2017.

[51] K. Shamsi and Y. Jin, "In praise of exact-functional-secrecy in circuit locking," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 5225–5238, 2021.