Removal of SAT-Hard Instances in Logic Obfuscation Through Inference of Functionality

ISAAC MCDANIEL, Department of Electrical and Computer Engineering, University of Maryland College Park, College Park, United States
MICHAEL ZUZAK, Department of Computer Engineering, Rochester Institute of Technology, Rochester, United States
ANKUR SRIVASTAVA, Department of Electrical and Computer Engineering, University of Maryland College Park, College Park, United States

Logic obfuscation is a prominent approach to protect intellectual property within integrated circuits during fabrication. Many attacks on logic locking have been proposed, particularly in the Boolean satifiability (SAT) attack family, leading to the development of stronger obfuscation techniques. Some obfuscation techniques, including Full-Lock and InterLock, resist SAT attacks by inserting SAT-hard instances into the design, making the SAT attack infeasible. In this work, we observe that this class of obfuscation leaves most of the original design topology visible to an attacker, who can reverse-engineer the original design given the functionality of the SAT-hard instance. We show that an attacker can expose the SAT-hard instance functionality of Full-Lock or InterLock with a polynomial number of queries of its inputs and outputs. We then develop a mathematical framework showing how the functionality can be inferred using only a black-box oracle, as is commonly used in attacks in the literature. Using this framework, we develop a novel attack that allows a SAT-capable attacker to efficiently unlock designs obfuscated with Full-Lock. Our attack recovers the intellectual property from these obfuscation techniques that were previously thought secure. We empirically demonstrate the potency of our novel sensitization attack against benchmark circuits obfuscated with Full-Lock.

CCS Concepts: • **Hardware → Combinational circuits**; • **Security and privacy → Hardware reverse engineering;**

Additional Key Words and Phrases: Logic obfuscation, full-lock, untrusted foundry, reverse engineering

ACM Reference Format:
Isaac McDaniel, Michael Zuzak, and Ankur Srivastava. 2024. Removal of SAT-Hard Instances in Logic Obfuscation Through Inference of Functionality. *ACM Trans. Des. Autom. Electron. Syst.* 29, 4, Article 71 (July 2024), 23 pages. https://doi.org/10.1145/3674903

This work was supported by NSF grant 1953285.
Authors' Contact Information: Isaac McDaniel, Department of Electrical and Computer Engineering, University of Maryland College Park, College Park, Maryland, United States; e-mail: ilm@umd.edu; Michael Zuzak, Department of Computer Engineering, Rochester Institute of Technology, Rochester, New York, United States; e-mail: mjzeec@rit.edu; Ankur Srivastava, Department of Electrical and Computer Engineering, University of Maryland College Park, College Park, Maryland, United States; e-mail: ankurs@umd.edu.

ACM 1084-4309/2024/07-ART71
https://doi.org/10.1145/3674903

1 Introduction

The increasing cost and complexity of semiconductor fabrication has driven **integrated circuit (IC)** designers to rely on unaffiliated and untrusted third parties for manufacturing. Such reliance raises security concerns due to the capability of untrusted foundries to reverse-engineer, pirate, and overproduce intellectual property using the design files provided for fabrication [21]. Such an approach exposes IC design houses to substantial financial and security risks.

Logic obfuscation (also known as logic locking) has been developed to mitigate these security threats during fabrication. Techniques within this family integrate auxiliary logic into a combinational circuit driven by both internal logic signals and a number of additional primary inputs, which are referred to as key inputs. For a small subset of all possible key input values, the auxiliary logic does not change the design functionality. However, for most key input values, error is introduced to the circuit with the intention of making pirated chips unreliable. The IC design house knows at least one value in the set that maintains functional correctness. After manufacturing, the IC design house assigns a permanent value to the key inputs from the functionally correct set. This value is stored in tamper-proof memory and is referred to as the secret key or just the key. By withholding the secret key from an untrusted fabrication partner, the correct functionality of a design is hidden. Such an approach mitigates security threats during fabrication. See References [7, 10, 41] for a comprehensive survey of logic obfuscation research.

In response to logic obfuscation, a number of techniques were developed to "unlock" the obfuscated design by either reverse-engineering the circuit by removing the auxiliary logic [2, 34, 38] or identifying a functionally correct key [18]. A family of Boolean **satisfiability (SAT)** attacks, which take the latter approach, are particularly prevalent in the literature [3, 11, 30]. These attacks use SAT solvers to efficiently identify functionally correct key input values. At the time of its introduction, the SAT attack could unlock any existing form of logic obfuscation, such as References [5, 19, 22]. SAT-style attacks are so potent against logic obfuscation that SAT-resilience has become a critical metric for any new logic obfuscation technique [43].

1.1 Contributions

In this work, we explore Full-Lock and InterLock [13, 14], which are logic obfuscation techniques that resist SAT-style attacks by inserting instances of known SAT-hard structures into the circuit. This increases the runtime of the underlying SAT problem solved in SAT-style attacks, making them infeasible. However, we observe that the only information obfuscated by these techniques is the correct routing of signals from the SAT-hard instance inputs to its outputs, with the rest of the netlist topology remaining unchanged. Therefore, if the attacker learns the relationship between the inputs and outputs of the SAT-hard instance, then the obfuscation process can be reversed and an obfuscation-free circuit produced.

In our previous work [16], we have shown that for Full-Lock and InterLock, it is possible to recover the obfuscation-free circuit from a polynomial number of queries of the input-output pairs of the SAT-hard instance. We developed an input stepping attack that exploited the topological rigidity of these types of obfuscation to infer the inputs and outputs of the SAT-hard instance from the obfuscated netlist and black-box oracle, which are available to a SAT-capable attacker. We showed empirically that our methodology allows an attacker to reverse-engineer netlists obfuscated with Full-Lock, which was not possible with prior techniques.

In this work, we review this attack method and justify it with additional mathematical rigor. We refine the attack to significantly reduce both the number of oracle queries and attack runtime. We also prove our assertion that the input-stepping attack is also effective against InterLock.

The contributions of this work can be summarized as follows:

— We review our sensitization attack, which queries the inputs and outputs of the SAT-hard instance placed in a design during obfuscation with Full-Lock. This attack is empirically shown to defeat Full-Lock obfuscation.
— We prove two theorems that, together, show that our sensitization attack defeats Full-Lock, a SAT-proof obfuscation technique, with a linear number of queries of the SAT-hard instance.
— We introduce primary input reuse to the sensitization attack, which reduces runtime by 51.9% and oracle queries by nearly 50%.
— We provably reduce the more recent InterLock to Full-Lock in the context of our sensitization attack. We identify the necessary conditions and how an attacker can achieve them. We describe a method to unlock InterLock with a polynomial number of SAT-hard instance queries.
— We describe the process by which an attacker can remove the SAT-hard instance from the netlist and layout after executing our attack. We also guarantee the timing correctness of the obfuscation-free design. This ensures that the attacker can produce functionally correct counterfeits of the design.
— We discuss how designers can prevent the sensitization attack by embedding functions of multiple obfuscated signals inside the SAT-hard instance.

2 Preliminaries

2.1 Attacker Model

In this work, we assume a SAT-capable adversary common in recent logic obfuscation research, such as References [13, 14, 24–26, 33, 40, 43]. This adversary has access to (1) a locked netlist for the obfuscated circuit, which can be obtained via reverse-engineering the GDSII files provided for fabrication, and (2) a black-box oracle of the obfuscated circuit, which can be obtained from IC test facilities or the open market. While the secret key cannot be read from this oracle circuit, it does allow the adversary to query specific inputs and identify the correct corresponding output for the obfuscated circuit.

We also make the assumption (3) that the attacker can locate the SAT-hard instance inside the obfuscated netlist. This assumption is reasonable, because all key inputs connect to the SAT-hard instance and because the SAT-hard instance contains many copies of the same structure.

2.2 The SAT Attack

The SAT attack was introduced in Reference [30] and defeated all previously developed logic obfuscation techniques. This iterative attack makes the assumption that the attacker has access to a black-box oracle that can be queried for primary output values corresponding to the applied primary inputs. In one iteration, the attacker formulates a Boolean SAT problem that is satisfied by two key values that are consistent with all previous oracle queries but produce a different primary output for at least one primary input value. The attacker then applies that input for that iteration's oracle query. This ensures that at least one of the keys that satisfies the current iteration cannot be used to satisfy the SAT problem in the next iteration, because at most one of these can produce the correct primary outputs when the chosen input is applied. The original SAT attack quickly prompted the creation of many new SAT-resistant obfuscation techniques [20, 32, 33, 37] that were then targeted by new SAT-style attacks [3, 4, 11, 27, 29].

One common approach to achieve SAT resilience is to scale the number of SAT attack iterations required to unlock the circuit by limiting the number of corrupted input-output pairs caused by each wrong key, as derived in References [42, 43]. This family of approaches includes prominent

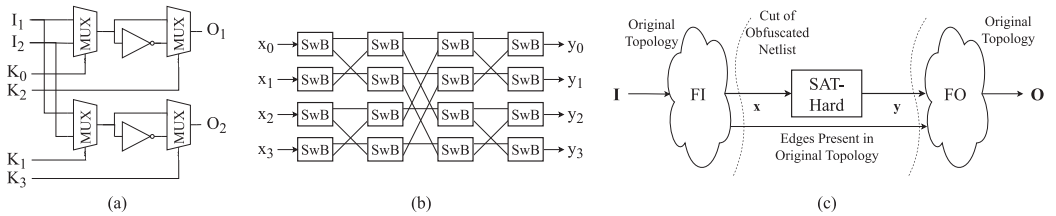

Fig. 1. Diagram of obfuscation with Full-Lock, showing (a) the key-driven switch-box (SwB) circuit, which exchanges the routing of two signals, (b) one possible switching network configuration, which forms the SAT-hard instance, and (c) a visualization of the instance with its fanin and fanout within the obfuscated netlist, with relevant signals labeled.

techniques such as References [15, 24–26, 33, 35, 39, 40]. While such approaches certainly achieve SAT resilience, they are limited in the amount of error they can inject, prompting concerns regarding their efficacy in securing an obfuscated system as a whole [43, 44].

To address these limitations, a second approach to SAT-resilient obfuscation techniques was developed leveraging SAT-hard instances to rapidly scale the runtime of successive SAT attack iterations, rather than increasing the number of iterations required to unlock the design [23, 28]. This family includes techniques such as Full-Lock [13] and Interlock [14]. The advantage of such an approach is that sizable error rates can be injected while maintaining resilience to SAT-style attacks [13, 14]. In this work, we narrow our scope to obfuscation techniques using this second approach.

2.3 Obfuscation with Full-Lock

One class of SAT-resilient techniques exploits characteristics of the **Davis–Putnam–Logemann–Loveland (DPLL)** algorithm used to solve the SAT attack's underlying SAT problem. These techniques place instances in the design of modules known to be SAT-hard, greatly increasing the runtime of successive SAT iterations and resulting in infeasibly long SAT attack runtimes to recover a functionally correct key.

In Full-Lock [13], which we primarily focus on in this work, this module is a switching network whose functionality is made key-dependent through the use of **programmable logic and routing (PLR)** blocks. Each node in the network is a **switch-box (SwB)** that may exchange the input to output routing and invert each of its two input signals according to three key inputs. As a result, the outputs of the SAT-hard instance are a permutation and possible inversion of its inputs. Figure 1 displays the construction of the switch-boxes present in Full-Lock as well as a sample network topology and the placement of the SAT-hard instance in the netlist.

Full-Lock is designed to take advantage of longer runtimes for the DPLL algorithm for problems with a ratio of clauses to variables in a certain range [13]. Multiplexers, which are very numerous in the SAT-hard instance, introduce clauses and variables to the SAT problem at this target ratio, increasing the runtime of the DPLL algorithm. The advantage of such an approach to obfuscation is that it is not fundamentally limited in the amount of error it can inject [42, 43]. Rather than hindering the SAT attack by reducing the number of inputs that produce corrupted outputs, obfuscation methods such as Full-Lock use the structure of the SAT-hard instance to lengthen SAT solve time. This makes the design SAT-resistant while still injecting sufficient error to prevent piracy.

2.4 InterLock

The authors of Full-Lock have since introduced InterLock [14], which improves Full-Lock by increasing the complexity of the switch-boxes used in the SAT-hard instance. Recall that after

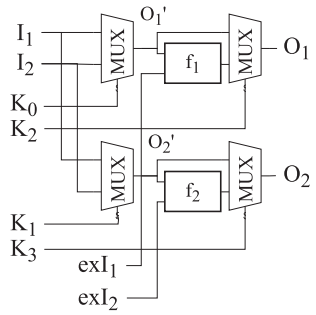

Fig. 2. Diagram of the SAT-hard instance used in InterLock, showing the switch-box (SwB) circuit modified from Full-Lock to include two extra inputs going to function blocks. These replace the inverters originally present in Full-Lock.

possibly switching their two inputs, Full-Lock switch-boxes have a second stage of multiplexers that give them the option to pass or invert each signal. InterLock replaces the inverters with 2-input gates, such as AND, OR, XOR, and so on. Each gate has logic function f_j and two inputs: I_i, the output of the first multiplexer in the switch-box; and "extra input" exI_j, a new input to the SAT-hard instance added by InterLock. The extra inputs pass from the original circuit to each individual switch-box. The new gate moves part of the design functionality to the SAT-hard instance, which prevents an attacker from simply removing the instance to restore the netlist to its state prior to obfuscation. Figure 2 shows the new switch-box structure introduced by InterLock.

InterLock represents a more secure obfuscation technique than Full-Lock, because the incorporation of 2-input gates inside the switch-boxes means that circuit logic is "twisted" into the SAT-hard instance. An attacker must not only learn the permutation order of the SAT-hard instance, but also determine where the 2-input logic gates and extra inputs should be applied to the signals passing through the SAT-hard instance. This makes InterLock more resistant to structural removal attacks than Full-Lock, which can at most invert its inputs. InterLock retains the SAT-resilient nature of its predecessor, so both SAT attack iteration time and output corruption for an incorrect key are high.

3 Attacking Full-Lock by Querying the SAT-hard Instance

In a design obfuscated with Full-Lock, there is in the worst case a single permutation of Full-Lock inputs that is logically equivalent to the intended module functionality implemented by the black-box oracle [13]. With $N!$ total permutations, a brute force attack is unfeasible, so another approach is required. In this section, we show how an attacker can learn the permutation implemented by an N-input SAT-hard instance with $N + 1$ queries of the exposed inputs and outputs of the instance. While our attacker model does not allow the SAT-hard instance to be queried directly, in subsequent sections, we will develop a method to analyze the obfuscated netlist and infer the relevant information from queries of the black-box oracle available to a SAT-capable attacker.

This will require two steps: (1) sensitization of the SAT-hard instance, covered in Sections 4 and 5, and (2) inference of instance outputs, covered in Section 6. The first is achieved through analysis of the SAT-hard instance's fanin cone in the obfuscated netlist, while the second is done through analysis of its fanout cone. We then formalize an attack on Full-Lock in Section 7 that queries the oracle and makes inferences about the instance functionality, producing a partial solution. We complete the solution by integrating our partial results with a conventional SAT-style attack in Section 7.

Table 1. List of Symbols and Their Meanings

Symbol	Meaning
$\mathbf{I}$	Primary input vector
$\mathbf{O}$	Primary output vector
$\mathbf{x}$	Input vector of the SAT-hard instance
$\mathbf{y}$	Output vector of the SAT-hard instance
x_i	Bit i of $\mathbf{x}$
y_j	Bit j of $\mathbf{y}$, the permutation destination of x_i
$\mathbf{x}^{i,1}, \mathbf{x}^{i,2}$	A pair of $\mathbf{x}$ values that differ only at x_i
$\mathbf{y}^{j,1}, \mathbf{y}^{j,2}$	A pair of $\mathbf{y}$ values that differ only at y_j, which is the permutation destination of x_i
$\mathbf{I}^{i,1}, \mathbf{I}^{i,2}$	The primary input values corresponding to $\mathbf{x}^{i,1}, \mathbf{x}^{i,2}$
$\mathbf{O}^{j,1}, \mathbf{O}^{j,2}$	The primary input values corresponding to $\mathbf{y}^{j,1}, \mathbf{y}^{j,2}$

3.1 Vector Definitions

Our attacks depend on observations of the primary inputs and outputs of the circuit as well as inferences about the inputs and outputs of the SAT-hard instance. We define vectors of the latter as the SAT-hard instance input vector $\mathbf{x}$ and output vector $\mathbf{y}$. These have the same length, which we label N, since the function of the SAT-hard instance is to permute its inputs. We will refer to the primary input vector as $\mathbf{I}$ and the primary output vector as $\mathbf{O}$. The primary inputs and outputs are part of the design topology before obfuscation, so the lengths of $\mathbf{I}$ and $\mathbf{O}$ can take any value. Figure 1(c) shows a high-level diagram of the SAT-hard instance and its fanin and fanout cones, with all four of these vectors labeled.

3.2 Revealing Permutation by Stepping Full-Lock Inputs

We can devise a method to learn the functionality of the SAT-hard instance by dividing the problem into sub-problems that can be solved individually. Since the functionality of the instance is to permute its inputs, to solve the problem all at once, as with the SAT attack, the attacker would need to find the correct permutation of the inputs from $N!$ possibilities. However, the attacker can break this down by choosing one input and attempting to find which output it is permuted to, which has N possible solutions.

Theorem 3.1. *Identifying the permutation destinations of input bits of the SAT-hard instance in Full-Lock reduces the solution space exponentially.*

Proof. The solution space of the functionality of the SAT-hard instance in Full-Lock is the number of possible permutations of N inputs, $N!$. Learning the placement of one item in the permutation, i.e., the permutation destination of 1 input bit, means that the number of remaining valid solutions is equal to the number of permutations of the other $N - 1$ input bits, or $(N - 1)!$. Therefore, identifying the permutation destination of 1 input bit prunes the solution space by a factor of N.

Similarly, identifying the permutation destination of a second bit prunes the solution space by a factor of $N - 1$, a third by $N - 2$, and so on. Therefore, the solution space is reduced exponentially as more permutation destinations are found. □

The exponential pruning of the solution space under Theorem 3.1 is sufficient cause for an attacker to search for permutation destinations of the SAT-hard instance. However, we can

deepen our intuition about why permutation destinations are so beneficial to solving the problem by considering the amount of work needed by the attacker to solve the problem using permutation destinations, as compared to solving the Full-Lock functionality without dividing it into sub-problems.

When searching for the permutation destination of the first input bit of the SAT-hard instance, the attacker must evaluate in the worst case N possible solutions, eventually determining the correct answer. After solving the first sub-problem, finding the permutation destination of the next input has only $N-1$ possible solutions, then $N-2$, and so on, until each input's destination has been found. By dividing the problem this way, the number of possible solutions the attacker must consider over the course of the attack is $\sum_{i=1}^{N} N-i$, which is quite low for an obfuscation problem. However, if the entire SAT-hard instance functionality is solved as a single problem, then the attacker must consider $N! = \prod_{i=1}^{N} N-i$ solutions, which is too many for a brute-force attack to be feasible. Therefore, the ability to identify the permutation destination makes solving the problem polynomial instead of exponential.

To determine the permutation destination of a SAT-hard instance input, it is helpful to express the outputs as logical functions. For any SAT-hard instance output y_j, there is some input x_i such that $y_j = f(x_i)$. Since inversion is possible inside the routing network, $f(x_i)$ may be x_i or $\neg x_i$.

Theorem 3.2. *Let $y_j = f(x_i)$ with $f(x_i) = x_i$ or $f(x_i) = \neg x_i$. Then, for two values of $\mathbf{x}$, $\mathbf{x}^{i,1}$ and $\mathbf{x}^{i,2}$, such that $x_i^{i,1} = \neg x_i^{i,2}$ and $x_k^{i,1} = x_k^{i,2}$, $k \neq i$. Then, the corresponding $\mathbf{y}$ values, $\mathbf{y}^{j,1}$ and $\mathbf{y}^{j,2}$, will satisfy $y_j^{j,1} = y_j^{j,2}$ and $y_k^{j,1} = y_k^{j,2}$, $k \neq j$.*

Proof. First, we note that each output is a function of only one input, and each input only propagates to one output, its permutation destination. Therefore, if the attacker makes one query of the SAT-hard instance to learn any input-output pairing, then changes one input bit $x_i \in \{x_0, x_1, \ldots, x_{N-1}\}$ and makes another query, the only output bit to change will be the changed input's permutation destination $y_j \in \{y_0, y_1, \ldots, y_{N-1}\}$. Then, the outputs $\mathbf{y}^{j,1}$ and $\mathbf{y}^{j,2}$ corresponding to $\mathbf{x}^{i,1}$ and $\mathbf{x}^{i,2}$ also have a Hamming distance of 1, and they satisfy $y_k^{j,2} = y_k^{j,1}$ for $k \neq j$ and $y_j^{j,2} = \neg y_j^{j,1}$ for the permutation destination y_j of x_i. Therefore, y_j can be determined by inspection from $\mathbf{y}^{j,1}$ and $\mathbf{y}^{j,2}$. □

Using Theorem 3.2, we describe how an attacker can efficiently unlock Full-Lock by querying the SAT-hard instance $N+1$ times. In the first step of the attack, the attacker makes an initial query of the SAT-hard instance to obtain the input-output pair $\mathbf{x}^{i,1}, \mathbf{y}^{j,1}$. The attacker will make a second query after changing one input bit, which we label $\mathbf{x}^{i,2}, \mathbf{y}^{j,2}$. We represent the position of the changed input bit as i, so $x_i^{i,2} = \neg x_i^{i,1}$. For the attacker, the initial choice of i is arbitrary; any bit in $\mathbf{x}^{i,1}$ could be flipped, and after querying the oracle, the attacker will find that $\mathbf{y}^{j,2}$ differs from $\mathbf{y}^{j,1}$ by one bit, and that y_j is the permutation destination of the toggled input x_i. As discussed previously, knowledge of y_j reduces the problem size by a factor of N, and it has been accomplished with only two SAT-hard instance queries.

After the attacker has learned the destination of one input, the remaining undetermined functionality of the instance can be represented as a permutation of the remaining $N-1$ inputs, since a permutation is a one-to-one mapping from the inputs to the outputs and one of each has just been removed from the problem. Therefore, the attacker chooses a new value of i and makes two more SAT-hard instance queries to learn a second permutation destination, this time reducing the problem size by a factor of $N-1$. After testing each bit of $\mathbf{x}$, the attacker has made $2N$ SAT-hard instance queries and learned the circuit's total functionality. With this information, the attacker

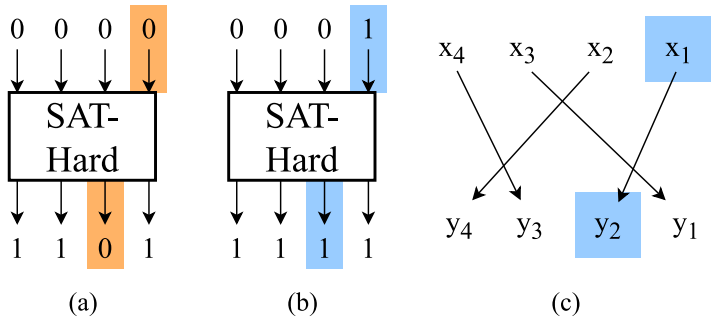

Fig. 3. An example of learning the functionality of an exposed SAT-hard instance. In (a), the attacker generates a reference I/O pair, and in (b), the attacker is able to compare a second I/O pair and learn that the instance passes input I_0 to output O_1. (c) shows the correct permutation after observing the example data from Table 1.

Table 2. Sample I/O Relationships Showing How the Functionality of a Fully Exposed SAT-hard Instance Can Be Learned in Linear Time

i	$\mathbf{x}^i$	$\mathbf{y}^j$	j	Learned Mapping
ref	0000	1101		
0	0001	1111	1	$x_0 \longrightarrow y_1$
1	0010	0101	3	$\neg x_1 \longrightarrow y_3$
2	0100	1100	0	$\neg x_2 \longrightarrow y_0$
3	1000	1001	2	$\neg x_3 \longrightarrow y_2$

If the instance has N permutation inputs, then the functionality is learned after checking the oracle $N + 1$ times.

can remove the SAT-hard instance from the netlist and route the input signals to the appropriate outputs, producing an obfuscation-free netlist.

To reduce the number of queries, the attacker can repeat the same first input $\mathbf{x}^{i,1}$ for each step, so $\mathbf{x}^{0,1} = \mathbf{x}^{1,1} = \cdots = \mathbf{x}^{N-1,1}$, which we represent as $\mathbf{x}^{ref}$ with corresponding output $\mathbf{y}^{ref}$. Then, the other input/output pairs $\mathbf{x}^{0,2}/\mathbf{y}^{j_0,2}, \mathbf{x}^{1,2}/\mathbf{y}^{j_1,2}, \ldots, \mathbf{x}^{N-1,2}/\mathbf{y}^{j_{N-1},2}$ can be represented as simply $\mathbf{x}^0/\mathbf{y}^0, \mathbf{x}^1/\mathbf{y}^1, \ldots, \mathbf{x}^{N-1}/\mathbf{y}^{N-1}$. Since one bit's mapping is now learned with each choice of $\mathbf{x}^i$, the attacker can learn the functionality of the SAT-hard instance with $N + 1$ oracle queries.

As an example, we examine the SAT-hard instance queries in Figure 3, which produce an initial input-output pair $\mathbf{x}^{ref} = 0000$, $\mathbf{y}^{ref} = 1101$. The attacker then toggles bit 0 of $\mathbf{x}$ and finds the I/O pair $\mathbf{x}^0 = 0001$, $\mathbf{y}^0 = 1111$. From these two data points, the attacker infers that x_0, the lowest bit of $\mathbf{x}$, is permuted to y_1, the second lowest bit of $\mathbf{y}$. Then, the attacker can use $\mathbf{x}^{ref}$ and add a new query for $\mathbf{x}^1 = 0010$. Finding $\mathbf{y}^1 = 0101$ reveals x_1 is permuted to $\neg y_3$.

After iterating through each bit of $\mathbf{x}$, the attacker has learned the permutation of all four input bits and therefore knows the logical function of the SAT-hard instance. This allows the attacker to recreate the netlist of the circuit before obfuscation, defeating the IP protection. Table 2 shows all of the I/O pairs the attacker finds and the information learned from each one, while Figure 3 graphically shows the first step of the process as well as the mapping from $\mathbf{x}$ to $\mathbf{y}$ that the attacker constructs.

3.3 Extension to InterLock

InterLock can be attacked in a similar manner to Full-Lock, but the increased complexity of the SAT-hard instance requires additional consideration [14]. Recall that a switch-box in InterLock contains 2-input logic gates with input-symmetric logic function f_j that may be applied to the signals being permuted by the SAT-hard instance. This introduces two difficulties we must overcome to extend the method we have developed in Section 3.2 to InterLock. First, the outputs of the SAT-hard instance added by InterLock are not always a permutation with possible inversion of its inputs. Second, since each stage of the network could cause a signal to pass through a gate, the attacker must learn which functions f_j with extra inputs exI_j are applied to the intermediate signals.

To overcome the first difficulty, we make use of a theorem:

Theorem 3.3. *There exists a vector of exI values such that the output vector* $\mathbf{y}$ *of the SAT-hard instance inserted by InterLock is a permutation with possible inversion of its input vector* $\mathbf{x}$.

Proof. The proof of the theorem is done by showing that each extra input has a value that causes the corresponding switch-box output O_j to be sensitized to I_i, so any change in I_i will be reflected by O_j. Then, if all extra inputs are set to sensitize their corresponding I_i, we show that this results in the SAT-hard instance producing an output that is a permutation with possible inversion of its input.

First, examine the InterLock switch-box diagram in Figure 2. In this diagram, the input-symmetric, non-constant 2-input logic functions f_1 and f_2 represent standard logic gates, such as AND, OR, and so on. Since these gates are included in the obfuscated netlist, they are known to the attacker, as are the values of the extra inputs, since these signals come from outside the obfuscated portion of the circuit. To ensure that O_j is sensitized to I_i, the attacker must use knowledge of the function f_j to set the extra input exI_j to a value such that, while exI_j remains constant, $f_j(I_i, exI_j) \equiv I_i$ or $f_j(I_i, exI_j) \equiv \neg I_i$.

Lemma 3.4. *For an input-symmetric, non-constant 2-input Boolean function* $f(x, y)$*, there exists a Boolean value* p *such that* $f(p, y) \equiv y$ *or* $f(p, y) \equiv \neg y$.

Proof. There are six input-symmetric 2-input Boolean functions: AND, OR, XOR, NAND, NOR, and XNOR. AND and NAND functions are inverted when the inputs both become 1, so for $p = 1$, an AND gate has $f(1, y) \equiv y$ and NAND has $f(1, y) \equiv\neq y$. Similarly, for $p = 0$, an OR gate has $f(0, y) \equiv y$ and NOR has $f(0, y) \equiv \neg y$. XOR and XNOR gates are always sensitized to both inputs; if exactly one input changes, then the gate output will always change. We say that p is both 0 and 1. □

We refer to a vector $\mathbf{p}$ of exI values that satisfies Theorem 3.3 as a pass vector. Similarly, the Boolean value p that satisfies Lemma 3.4 is the pass value of the logic function.

To ensure that O_j is sensitized to I_i, the attacker must use knowledge of the function f_j to set the extra input exI_j to the correct pass value for the function f_j. We will argue next that this collection of individual pass values is also a pass vector of the SAT-hard instance.

The result of sensitization of O_j to I_i (and by extension, of y_j to x_i) is that the extra input can be removed from the attacker's model of the circuit, and f_1 and f_2 can be represented by either a buffer or inverter. This makes the output of each switch-box a permutation with possible inversion of its inputs, which is the same as the switch-boxes in Full-Lock. Since the switching network structure is also the same as in Full-Lock, applying this condition to each switch-box in the SAT-hard instance leads to the conclusion that the functionality of the SAT-hard instance is a permutation with possible inversion, confirming the theorem. □

As described in the proof of the theorem, the attacker can learn each bit p_j of $\mathbf{p}$ by simply inspecting the obfuscated netlist to observe the corresponding gate function f_j. As an example, if

a switch-box has $f_1 = AND(I_i, exI_1)$, then the attacker can set $exI_1 = 1$, and then for $I_i = 0, O_1 = 0$, while for $I_i = 1, O_1 = 1$. Therefore, the pass value of exI_1 is 1.

With a pass vector applied to its extra inputs, the SAT-hard instance of InterLock becomes functionally equivalent to one in Full-Lock. The input stepping method described in Section 3.2 can be used to learn the permutation implemented by the SAT-hard instance.

This is very helpful to the attacker, but unlike in Full-Lock, it does not represent the total functionality of the SAT-hard instance. Since functions are applied to InterLock signals within the SAT-hard instance, we must also match each signal to the 2-input function applied to it (or bypassed) at each stage of the switching network. This can be done by manipulating the inputs and observing changes in the output to determine which intermediate signal passes through each gate and which gates are bypassed.

To most quickly learn which output signal has passed through each f_j, the attacker uses Lemma 3.4 to check whether either value of I_i (which can be changed by changing the SAT-hard instance input x_i) causes an output y_j to be sensitized to an extra input exI_j. By the proof of the lemma, if no y_j is sensitized to exI_j for either value of x_i, then the signal has not passed through a gate.

Assuming the input stepping attack has already been executed on a SAT-hard instance, the attacker begins with the reference input and extra input pass value used for the input stepping attack, then toggles each extra input one at a time. If an output bit y_j changes in response to exI_j, then the attacker knows to apply f_j to the corresponding input signal when reconstructing the obfuscation-free netlist. After testing each extra input, the SAT-hard instance input vector $\mathbf{x}$ is inverted and the process is repeated. Once all inputs have been applied to each gate, any gates that did not produce an output response must be bypassed, since each bit of $\mathbf{x}$ would have been set to its pass value in each 2-input function f_j during one toggle of the corresponding exI_j, yet we did not observe responses in the output to the changing exI_j. Therefore, by Lemma 3.4, f_j could not have been applied to any bit of $\mathbf{x}$.

If output responses are observed, then the attacker knows to apply f_j to the corresponding input signal when reconstructing the obfuscation-free netlist. If no change is observed at the output, then there are two explanations: (1) this exI_j is unused and the corresponding f_j is bypassed, or (2) f_j is not sensitized to exI_j at the current value of I_i. To distinguish between these, a second pass through exI_j is needed. The attacker first inverts the value of every instance input x_i then again toggles each exI_j that did not affect any y_j in the first pass. If some exI_j produces no change to the outputs this time, then the attacker knows that this exI_j is not actually used, since every combination of O'_j and exI_j has been tried but the value of exI_j has not affected the output.

Once this process is complete, the attacker knows every f_j and exI_j that must be applied to each instance input, as well as which instance output the resulting signal is permuted to. Therefore, the attacker is able to replace the SAT-hard instance in the obfuscated netlist with the intended functionality, unlocking the circuit. This attack methodology requires in the worst case four oracle queries per switch-box. Since each stage of the network has $N/2$ switch-boxes and there are approximately $\log_2(N)$ stages in the network (depending on the network topology), this increases the number of queries required for the attack from $N + 1$ for Full-Lock to approximately $(N + 1) + \frac{N}{2}\log(N)$ for InterLock. While we see that InterLock is higher complexity, it can still be completed with a polynomial number of SAT-hard instance queries, unlocking InterLock efficiently.

In subsequent sections, we will describe how a SAT-capable attacker can infer enough information about the SAT-hard instance from a black-box oracle to use the methods described in this section to mount an effective attack on Full-Lock even without direct access to the inputs and outputs of the SAT-hard instance. Although the rest of this article is focused on attacking a design

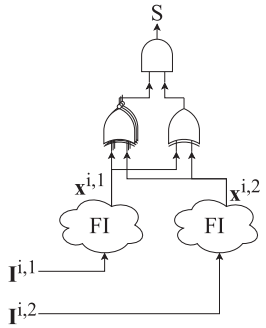

Fig. 4. Miter circuit for finding a sensitizing primary input pattern $\mathbf{I}^{i,1}, \mathbf{I}^{i,2}$ for a SAT-hard instance input. The inclusion of 1 XOR gate and $N-1$ XNOR gates forces the condition that $\mathbf{x}^{i,1}$ has a Hamming distance of 1 from $\mathbf{x}^{i,2}$.

obfuscated with Full-Lock, the methods we describe in Sections 4–8 apply equally to InterLock with the pass vector applied to the extra inputs.

4 Sensitization of the SAT-hard Instance

Using the method in the previous section, an attacker with input and output access to the SAT-hard instance in Full-Lock can learn the permutation destination of one input with a constant number of SAT-hard instance queries, and the instance's total functionality with a linear number of queries. However, our attacker model assumes a black-box oracle, so the only information directly available to the attacker is the primary inputs and outputs of the authenticated circuit. Generally, the SAT-hard instance is not placed at the input or output of the circuit being obfuscated, so the SAT-hard instance inputs/outputs cannot be assumed to be the same as the primary inputs/outputs.

Applying our technique to the SAT-capable attacker model requires analysis of the obfuscated netlist to select inputs for an oracle query that will apply inputs $\mathbf{x}^{i,1}$ and $\mathbf{x}^{i,2}$ to the SAT-hard instance. To do this, we partly generalize the obfuscated circuit in the previous section by allowing a non-empty fanin cone for the SAT-hard instance, so $\mathbf{I} \neq \mathbf{x}$, but still requiring the SAT-hard instance outputs to be the primary outputs ($\mathbf{O} = \mathbf{y}$), meaning that the fanout cone is empty.

We define a sensitizing input as a pair of primary input values $\mathbf{I}^{i,1}, \mathbf{I}^{i,2}$ that produce the SAT-hard instance inputs $\mathbf{x}^{i,1}, \mathbf{x}^{i,2}$. We say that x_i is the sensitized input. The outputs of the SAT-hard instance, and in this case primary outputs, for the same inputs are $\mathbf{y}^{j,1}, \mathbf{y}^{j,2}$, where we say that y_j is the sensitized output. As in the case where the attacker is able to directly query the SAT-hard instance, y_j is the permutation destination of instance input x_i. A successful attack in this case must find a sensitizing input $\mathbf{I}^{i,1}, \mathbf{I}^{i,2}$ for every SAT-hard instance input $i \in [0, N-1]$ and observe its permutation destination to determine the total functionality of the design.

Sensitizing inputs can be found efficiently by constructing a Boolean satisfiability problem around the SAT-hard instance fanin. To find a sensitizing input for instance input x_i, we create two copies of the logic between the instance and the primary inputs with input vectors $\mathbf{I}^{i,1}$ and $\mathbf{I}^{i,2}$ and output vectors $\mathbf{x}^{i,1}$ and $\mathbf{x}^{i,2}$. We will add logic to these to build a miter circuit. The values of $\mathbf{I}^{i,1}, \mathbf{I}^{i,2}$ are the solution to the problem, so these are not altered and remain the inputs of the miter circuit. Logic added to the fanin cone outputs (i.e., SAT-hard instance inputs) $\mathbf{x}^{i,1}, \mathbf{x}^{i,2}$ need to force them to meet the sensitization conditions $x_i^{i,1} = \neg x_i^{i,2}$ and $x_k^{i,1} = x_k^{i,2}, k \neq i$. This can be done by adding N logic gates $G_k = f_k(x_k^{i,1}, x_k^{i,2}), k \in [0, N-1]$, where f_i is XOR and $f_k, k \neq i$ is XNOR. An N-input AND gate, with each G_k as an input, requires all of the output conditions to

be met to satisfy the single output of the miter circuit. Since the miter, shown in Figure 4, is only satisfied when the instance is sensitized at input x_i, any primary input values $\mathbf{I}^{i,1}, \mathbf{I}^{i,2}$ that satisfy the miter must be a sensitizing input. Once the attacker has found $\mathbf{I}^{i,1}$ and $\mathbf{I}^{i,2}$, they can be applied to the black-box oracle to observe the outputs $\mathbf{O}^{j,1} = \mathbf{y}^{j,1}$ and $\mathbf{O}^{j,2} = \mathbf{y}^{j,2}$, which reveal y_j is the permutation destination of x_i.

In an arbitrary circuit design, it is also possible that the miter circuit we have defined is found to be unsatisfiable. This means that there are no two possible output values that differ only at the desired bit. When this happens, it is not possible to sensitize the SAT-hard instance for that bit, and the permutation destination cannot be learned directly. When this occurs, this attack can only partially recover the functionality of the instance, but the reduction in the search space is exponential with the number of SAT miters, allowing a secondary attack using a conventional method to recover the missing functionality. Since the secondary attack solves an exponentially smaller problem than the attacker initially faced, the execution time of our attack combined with a secondary attack is still much smaller than an attack using the same method as the secondary attack from the beginning.

The efficiency of this part of the attack is determined by how quickly sensitizing inputs can be found. Satisfying one miter allows the attacker to learn the correct destination of one SAT-hard instance input, each time reducing the effective number of signals permuted by the obfuscation by one and pruning the functionality search space exponentially. In the example from the previous section, filling in each row of Table 1 would require the attacker to solve one SAT problem. Recovering the oracle's total functionality requires solving N problems, one for each SAT-hard input. This means the amount of time spent on each SAT problem is the primary factor in determining whether the attack is feasible.

Importantly, the miter circuit **does not include the SAT-hard instance** itself, which is designed for attack resilience. In fact, the security provided by Full-Lock depends fundamentally on an attacker using the SAT attack being forced to include the SAT-hard instance in a SAT problem formulation, so our construction of a miter circuit that does not fall prey to this trap bypasses the security guarantees of this logic obfuscation technique. Furthermore, the duration of our novel sensitization attack depends only on the topology of the design before obfuscation, which affects the attacker's ability to find inputs to sensitize specific nodes in the circuit. Solving the latter problem is an important step in IC testing, which has presumably been performed on the target obfuscated design, since it is in production. The attacker is therefore confident that sensitization problems using the netlist are feasible and may even have access to the same or similar commercial **Automatic Test Pattern Generation (ATPG)** tools used to analyze the design for legitimate purposes. Foundry-based attackers, one potential identity of a SAT-capable attacker, are particularly likely to have ready access to these tools. These ATPG tools are very well developed and are highly efficient for these problems [9, 17]. They have also seen use in other security applications [8, 24, 25].

5 Reduction of Oracle Queries

The exclusion of the SAT-hard instance from SAT analysis and the attacker's confidence in the feasibility of the necessary SAT problems make our attack very efficient compared to conventional attacks, such as the SAT attack, which are unaware of Full-Lock functionality, as these must include the SAT-hard instance in their SAT formulations. However, the sensitization process as previously described requires $2N$ oracle queries, though the input step process described in Section 3.2 requires only $N + 1$ queries of the SAT-hard instance. Therefore, we seek to reduce the number of oracle queries in our sensitization attack to match the number required to complete the analysis of the SAT-hard instance without any surrounding circuitry.

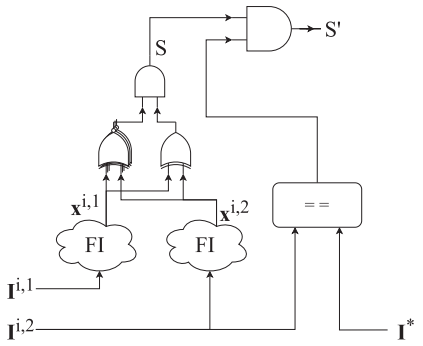

Fig. 5. Miter circuit for finding a sensitizing input for a SAT-hard instance input while reusing one previously generated PI vector $\mathbf{I}^*$, which would have been selected to sensitize a different bit of $\mathbf{x}$.

Comparing the two processes, we see that this difference in query counts comes from the reuse of an initial "reference" input $\mathbf{x}^{ref}$ and output $\mathbf{y}^{ref}$ during the input stepping process, which are compared to each subsequent SAT-hard instance query $\mathbf{x}^i/\mathbf{y}^j$. However, when searching for sensitizing inputs, the circuit-SAT solver returns two PI patterns $\mathbf{I}^{i,1}$ and $\mathbf{I}^{i,2}$ for each bit i of $\mathbf{x}$. To reduce the number of queries we make in our sensitization attack, we would like to similarly add only a single oracle query for each bit i of $\mathbf{x}$. However, for the sensitization attack, we cannot simply choose an arbitrary $\mathbf{x}^{ref}$ and toggle each bit one at a time, because the circuit topology may make some values of $\mathbf{x}$ impossible to achieve for any primary input vector.

Instead, we begin the sensitization process as usual to find the first two primary input values, $\mathbf{I}^{0,1}$ and $\mathbf{I}^{0,2}$, which sensitize a single bit of $\mathbf{x}$, but for subsequent bits of $\mathbf{x}$, we attempt to adapt the circuit-SAT problem of Figure 4 to reuse primary input values already selected to sensitize other bits of $\mathbf{x}$. So, for the second bit of $\mathbf{x}$, we seek to assign $\mathbf{I}^{1,2} = \mathbf{I}^{0,1}$ or $\mathbf{I}^{1,2} = \mathbf{I}^{0,2}$.

The most direct way to implement this is shown in Figure 5, with an equality block enforcing the condition that $\mathbf{I}^{i,2} = \mathbf{I}^*$, where $\mathbf{I}^*$ could be any of the previously generated PI vectors. In practice, this equality block is implemented with an N-input AND gate between all bits of $\mathbf{I}^{i,2}$ or their inversion, depending on the bit's value in $\mathbf{I}^*$. This accounts for the value of $\mathbf{I}^*$ while building the miter circuit and eliminates the need for an explicit $\mathbf{I}^*$ signal in the circuit-SAT problem.

While the first sensitization pair must still be generated in the original way, requiring two initial PIs, this new circuit-SAT problem allows all subsequent sensitization patterns to be produced by adding only one new PI vector. This method still nominally produces a pair of PI vectors to sensitize each of the N bits of $\mathbf{x}$, but accomplishes this with only $N+1$ unique vectors, matching the number of oracle queries used in the original input stepping attack in Section 3.2.

For an arbitrary circuit, there is no guarantee that a previously generated PI vector $\mathbf{I}^*$ can be used to sensitize the chosen bit x_i, even if x_i is sensitizable. If the circuit-SAT solver returns UNSAT using $\mathbf{I}^*$, then the target bit x_i cannot be toggled for any value of $\mathbf{I}$. The attacker must attempt to reuse another previously generated $\mathbf{I}^i$ value, and if no $\mathbf{I}^i$ is successfully reused, then the original circuit-SAT problem must be attempted to see if x_i is sensitizable at all. In this case, sensitizing x_i requires adding two new PI values, and the attack will require more than the theoretical minimum of $N + 1$ oracle queries.

In our experiments, we did not observe any benchmark circuits that required more than $N +$ 1 oracle queries. Additionally, although this version of the attack potentially requires multiple problems to be solved, the solver effectively has fewer bits it needs to assign, making each problem significantly faster. Our experiment shows that the overall attack time does not suffer and in fact improves using this alternate sensitization method.

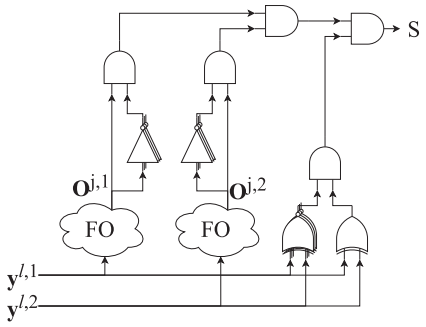

Fig. 6. Miter circuit that tests an observed output pattern for sensitivity to a particular input bit. The inclusion of 1 XOR gate and $N-1$ XNOR gates forces the condition that $\mathbf{y}^{j,1}$ has a Hamming distance of 1 from $\mathbf{y}^{j,2}$.

6 Inferring SAT-hard Instance Outputs

Now that we have established that an attacker can sensitize the SAT-hard instance inside a black-box oracle, we move to show how information about the outputs of the instance can be inferred from the oracle. This problem is more difficult than finding sensitizing inputs, because in the latter, the attacker applies known inputs and can precisely evaluate internal nodes in the fanin of the SAT-hard instance. However, when attempting to determine which SAT-hard instance output has inverted from the change in the primary outputs seen in an oracle query, there may be multiple fanout inputs (i.e., instance outputs) that could produce the same observed results. This limits the attacker to examining each fanout input and determining which ones could have been the inverted signal, rather than solving one problem and producing a definite solution, as when finding a sensitizing input.

The attacker knows that after applying a sensitizing input to the oracle, one instance output y_j has inverted while all other outputs remain the same. While the value of y_j is unknown, the attacker can build a list of candidate outputs by testing each instance output $y_l \in \{y_0, y_1, \ldots, y_{N-1}\}$ to determine whether its sensitization could have produced the primary output $\mathbf{O}^{j,1}, \mathbf{O}^{j,2}$ seen in the black-box oracle. Testing whether y_l could be the sensitized output y_j requires the construction and solving of a miter circuit similar to the one used to find sensitizing inputs.

When finding a sensitizing input, the miter circuit is formed around two copies of the SAT-hard instance fanin. To test whether y_l could be the sensitized bit, the miter is constructed around copies of the SAT-hard instance fanout, with $\mathbf{y}^{l,1}$ and $\mathbf{y}^{l,2}$ as their inputs and $\mathbf{O}^{j,1}$ and $\mathbf{O}^{j,2}$ as their outputs. Since the attacker is interested in whether any values of $\mathbf{y}^{l,1}, \mathbf{y}^{l,2}$ are consistent with the oracle query output $\mathbf{O}^{j,1}, \mathbf{O}^{j,2}$, these are also the input to the overall miter circuit. To enforce the condition that output y_l is sensitized, i.e., $y_l^{l,1} = \neg y_l^{l,2}$ and $y_k^{l,1} = y_k^{l,2}, k \neq l$, we append to the miter the logic gates $G_k = f_k(y_k^{l,1}, y_k^{l,2}), k \in [0, N-1]$, where f_l is XOR and $f_k, k \neq l$ is XNOR. In addition, the attacker requires the satisfying values of $\mathbf{y}^{l,1}, \mathbf{y}^{l,2}$ to produce the observed primary output $\mathbf{O}^{j,1}, \mathbf{O}^{j,2}$. This is done by adding a $2N$-input AND gate that takes as input every bit of $\mathbf{O}^{j,1}, \mathbf{O}^{j,2}$ or its inversion, depending on that bit's value in the oracle query. Finally, the output of this AND gate is passed into an $N+1$-input AND along with the outputs of each gate G_k, which gives the miter output. The structure of this miter is shown in Figure 6. If the miter is satisfiable, then the SAT-hard instance output y_l must be added to the list of candidates for y_j.

We use this analysis repeatedly to prune the search space of the functionality of the SAT-hard instance. N oracle queries, one that sensitizes each instance input, are needed, and each oracle query requires N SAT problems to be solved, one for each instance output. This results in N^2 problems in total. The degree of pruning of the search space depends on the number of permutation

destinations the attacker can rule out for each sensitized input. For a single sensitized input x_i, i.e., for each oracle query, if the attacker determines that m_i of the N outputs could not be the permutation destination y_j for x_i, then the functionality search space is reduced by a factor of $\frac{N}{N-m_i}$. In the best case, the destination is determined exactly when all but one output is ruled out, so $m_i = N-1$ and the search space is reduced by a factor of N from $N!$ to $N-1!$. This is the same reduction as was seen in previous sections when an input's destination was determined.

A secondary attack is necessary when multiple instance outputs could produce the oracle's outputs when sensitized. As in the previous section, the functionality of the SAT-hard instance has already been extensively pruned, so executing both our sensitization attack and a secondary attack using the traditional SAT attack obtains a total solution more quickly than an attack using only the SAT attack. Like the method for finding sensitizing inputs described in Section 4, the efficiency of the output analysis is determined by the efficiency of the available SAT solver. Our attack continues to exclude the SAT-hard instance from the SAT problem, giving it an advantage over existing attacks such as the SAT attack.

7 Full-Lock Functionality Recovery by a SAT-capable Attacker

Finally, let us consider our total attack surface.

Threat Model: A SAT-capable attacker has access to

(1) The primary inputs **I** and outputs **O** of a black-box oracle.
(2) An obfuscated netlist, including knowledge of which inputs are key inputs. Items 1 and 2 are standard in the literature.
(3) The number and location of SAT-hard instance inputs **x** and outputs **y**. This is revealed to the attacker by the concentration of the key inputs in the SAT-hard instance and the regular structure of the switch-boxes.

Knowledge of **x** and **y** is necessary for the attacker to construct miter circuits for sensitivity analysis. In general, there is logic between these vectors and the primary input and output vectors **I** and **O** of the SAT-hard instance, so $\mathbf{x} \neq \mathbf{I}$ and $\mathbf{y} \neq \mathbf{O}$.

With only access to the primary inputs and outputs of the black-box oracle, the attacker must be able to sensitize the inputs of the SAT-hard instance and then infer its possible outputs. This can be done by combining the prior two algorithms, discussed in Sections 4–6. First, the attacker analyzes the instance fanin with the miter in Figure 4 or Figure 5 to find sensitizing primary inputs $\mathbf{I}^{i,1}, \mathbf{I}^{i,2}$ for each instance input $x_i \in \{x_0, x_1, \ldots, x_{N-1}\}$. This can be performed exactly as described in Section 4, since the nonempty fanout cone of the SAT-hard instance does not affect the topology or function of the fanin. The alternate method in Section 5 could also be used to reduce the number of oracle queries made. In either case, the attacker queries the oracle for each sensitizing input, but unlike in Section 4, this does not immediately reveal the permutation that the SAT-hard instance performs.

Instead, though the attacker knows the inputs **x** to the SAT-hard instance, its output **y** must be extrapolated from the primary outputs **O**. This is identical to the scenario described in Section 6, so the same process can be applied here. The attacker compares the oracle outputs $\mathbf{O}^{j,1}, \mathbf{O}^{j,2}$ to the instance fanout and prunes the functionality search space by using the miter in Figure 6 to evaluate which instance outputs $l \in \{1, \ldots N\}$ may have been the stimulated output j.

This attack generally leaves the attacker with only a partial solution to the functionality of the SAT-hard instance, so a secondary attack using an existing methodology is used to identify the exact functionality from the greatly reduced search space. We repeat our earlier argument, that even with the secondary attack, our attack is more efficient than using the existing methodology from the beginning, because our sensitization attack has greatly reduced the size of the remaining

problem. Our attack reduces the problem complexity more efficiently than existing methodologies, because it does not include the SAT-hard instance in a SAT formulation. This is demonstrated in Section 11, where we show the results of our experiment attacking benchmark circuits.

8 Recovery of Complete Functionality through a Secondary Attack

Our sensitization attack can be completed much faster than a traditional SAT attack, but generally produces only a partial solution. This occurs for two reasons:

(1) Our attack sensitizes SAT-hard instance inputs by learning two primary input vectors $\mathbf{I}^{i,1}$, $\mathbf{I}^{i,2}$ that produce two instance inputs $\mathbf{x}^{i,1}$, $\mathbf{x}^{i,2}$ that differ by a Hamming distance of 1, placing the single differing bit in a precise location. This is a heavily constrained problem, and there may be no solution to sensitize some inputs. When this occurs, our attack will not be able to infer the destination of this input, since it cannot observe its effects on the primary outputs without other inputs also changing.
(2) After observing two primary output vectors from the oracle, our algorithm must determine which of the SAT-hard instance output bits could have produced the query results. However, multiple outputs could be capable of this, so the attack is only able to determine a group of candidate outputs, any one of which could be the permutation destination of the sensitized input.

As has been discussed in Sections 6 and 7, the partial solution produced by our methodology reduces the search space by pruning the number of possible permutation destinations of each SAT-hard instance input. While this does not fully unlock the circuit, these results represent an exponential reduction in the functionality search space. To fully unlock the obfuscated circuit, we launch a second attack to recover the remaining functionality. This secondary attack builds on the results of our functional attack and is able to solve the greatly reduced problem.

To set up the secondary attack, we take as output from our novel sensitization attack a matrix S of Boolean values, with rows representing SAT-hard instance inputs and columns representing instance outputs. Matrix element s_{ij} is False if our attack concluded that y_j could not be the permutation destination of x_i and True otherwise. We have developed a tool that uses this information to replace the SAT-hard instance in the obfuscated netlist with N multiplexers, each with an output that replaces an output of the removed SAT-hard instance. A newly added key-driven select signal allows the multiplexer to pass one of the N signals that were previously the SAT-hard instance inputs that our attack did not eliminate as possible sources of that output. The SAT-hard instance was also capable of inverting its inputs, so a 2-input multiplexer is added after each N-input multiplexer that uses another key-driven select signal to choose between the selected instance input and its inversion. Since the SAT-hard instance has been removed, the key-driven select signals are the only key bits remaining in the netlist. The multiplexers are capable of reproducing any functionality in the search space that the original obfuscated netlist was capable of, so this operation preserves the functionality of the design as a whole. The edited netlist is similar to the relaxed models of the SAT-hard instance described in Reference [31], but in our secondary attack, all inputs cannot reach all outputs.

The secondary attack can be launched using this modified netlist and the existing black-box oracle. The key value the attack finds tells the attacker the correct routing of signals from the SAT-hard instance input to its output. With this information, the attacker infers the correct functionality of the obfuscated netlist, bypassing the security guarantees of Full-Lock.

9 Netlist Modification to Remove Obfuscation

After the completion of the secondary attack, the attacker has completely learned the correct functionality of the obfuscated netlist. The simplest way to produce a functional obfuscation-free netlist

from the attack results is to remove the logic for the SAT-hard instance from the design files and add wires from each of its input nodes to the corresponding output node, possibly with an inverter. This netlist modification introduces correct logic to the design with no key inputs but reduces the delay of the signals passing through, which could result in timing violations in a tightly constrained netlist. To mitigate this, the opened up space in the layout that previously hosted the SAT-hard instance can be used to insert delay elements, causing the modified netlist's timing to more closely resemble that of the original obfuscated netlist. Both the netlist editing and timing evaluation can be done with any tools compatible with the IC technology; no special knowledge of the tools used in the original circuit design is required.

10 Countermeasures

As discussed in Section 3.3, InterLock does not provide additional security against our attack compared to Full-Lock. Since each logic gate inside InterLock has only one input that is obfuscated, the attacker can assign the other input so the gate output is the same as the obfuscated signal. Once the embedded gates are made "transparent," the attacker can proceed as if attacking Full-Lock.

Sweeney et al. [31] propose a defense that is very similar to InterLock, but in their version, the logic gates embedded in the switch-boxes take both inputs from the MUXes. The SAT-hard instance in LoPher [23] has the same functionality, but both the permutations and the logic functions are implemented with a cryptographic block cipher. In both techniques, there are no extra inputs to the switch-boxes. This has several implications:

— Without direct access to any gate inputs, the attacker cannot make the gates transparent.
— The attacker can no longer assume a one-to-one mapping from inputs to outputs, since each gate sinks multiple switch-box inputs and produces only one output.
— The value of each SAT-hard instance output depends on more than one of its inputs.
— The sensitivity of the SAT-hard instance to its inputs is unknown. There is no guarantee that toggling any one input will result in an output response.

The interaction between signals traveling through the routing block makes the sensitization attack insufficient to determine the functionality, as the problem the attacker must solve changes from a one-to-one mapping to a circuit with unknown number and type of gates. As a result, these obfuscation methods are secure from our sensitization attack. All future switching-based obfuscation techniques will need to include functions of multiple obfuscated inputs to be considered secure.

Further research into circuit sensitization could produce attacks able to break these techniques. While basic input-stepping is insufficient to learn the functionality of a SAT-hard instance with arbitrary logic, other input patterns may produce enough information for an attacker to manufacture counterfeit chips.

Other types of attacks may also be able to target this kind of obfuscation. The routing network in Full-Lock is designed to resist the SAT attack, but the repeated switch-box structure may make Full-Lock vulnerable to ML-based attacks such as GNNUnlock+ [1, 2]. This attack uses neural networks to classify nodes in an obfuscated netlist as being part of the original circuit or being part of the locking logic.

11 Results

In this section, we discuss the implementation of our attack and present data gathered from testing it against benchmark circuits locked using Full-Lock. We provide runtime data to demonstrate the feasibility of the attack against benchmarks obfuscated with large SAT-hard instances.

11.1 Experimental Setup

Our experiments begin with our benchmark circuits in gate-level Verilog files. We select arbitrary internal signals for obfuscation with Full-Lock and generate the SAT-hard instance with a Python script. The resulting obfuscated Verilog file is saved in the **Berkeley logic interchange format (BLIF)** for maximum compatibility with ABC. The original benchmark file is also converted to BLIF for use as a black-box oracle. During each step, we keep the benchmark circuits as close to the original as possible, preserving their logical structure. This keeps our results consistent regardless of which tools are used to prepare the netlists. Our attack is also technology-independent, since it is purely logical.

Our sensitization attack code extends the ABC synthesis tool [6]. The tool extracts the SAT-hard instance fanin cone from the obfuscated netlist, constructs the miter circuits in Figures 4 and 5, and uses ABC's SAT solver to find sensitizing inputs. Our tool is capable of finding sensitizing inputs using either the method described in Section 4 or the alternate method of Section 5, the latter using fewer oracle queries.

Once the sensitizing inputs are selected, the tool queries the black-box oracle to find the corresponding outputs. With these it forms the miter in Figure 6 and infers a partial solution of the SAT-hard instance functionality.

A Python script modifies the obfuscated netlist to account for the reduced search space as described in Section 8, and then the lazy-sat tool [30] was used to find the total solution.

Our control data was generated using the lazy-sat tool on the first set of obfuscated benchmarks.

We tested our attack on benchmarks from a variety of application areas, selecting five benchmarks from the ISCAS '85 suite [12], one benchmark from MCNC20 [36], and five benchmarks from ITC-99. Each benchmark is obfuscated with Full-Lock using three or four differently sized SAT-hard instances. All benchmarks included logic between the SAT-hard instance and both the primary inputs and outputs, so a successful attack in our experiment required both input sensitization and inference of SAT-hard instance outputs. This is the most general form of our attack, which can be launched by any SAT-capable attacker.

We performed two experiments to demonstrate our attack: (1) We tested our sensitization attack and secondary SAT attack as a compound attack on our benchmark circuits obfuscated with Full-Lock and compared the compound attack duration to that of the traditional SAT attack; and (2) we tested our primary attack using two different sensitization methods, described in Sections 4 and 5 and compared the time required in each case to complete the primary attack, which includes generating the sensitizing inputs and analyzing the primary outputs for input-output relationships.

11.2 Sensitization Attack

We tested our novel sensitization attack against 11 benchmark circuits, first measuring the runtime of the sensitization attack, which produced a partial solution, and then the runtime of the secondary attack, which extracts the remaining functionality. Each benchmark was obfuscated with each of 3 SAT-hard instance sizes with key sizes of 48, 144, and 384 bits. For the 8 larger benchmark circuits, we also tested with 960 bits. In this experiment, we used the sensitization method described in Section 4.

Table 3 shows our results for each benchmark circuit and SAT-hard instance size, as well as the SAT attack runtime data for comparison. For instances with 144 or fewer key bits, the SAT attack is often faster than the proposed attack. This is especially true of the larger benchmarks. However, at these sizes, the attack is still very quick. Except for the largest three benchmarks, our attack completes within 1 minute for 144 or fewer key bits, which is very small compared to the length of the 48-hour timeout window. Furthermore, our sensitization attack clearly accelerates the secondary SAT attack compared to the standard SAT attack.

Table 3. Sensitization Attack and SAT Attack Durations for Various Key Sizes

Circuit	Key Size	Sensitization Attack Runtime	Secondary Attack Runtime	Total Runtime	SAT Attack Runtime
c1908	48	0.55	0.21	0.77	0.84
	144	3.37	0.53	3.90	22.71
	384	14.01	12.91	26.93	timeout
c2670	48	1.58	0.18	1.76	0.48
	144	7.87	0.71	8.59	4.31
	384	68.83	8,458.36	8,527.19	8,708.47
c3540	48	2.46	0.61	3.07	0.56
	144	11.02	2.92	13.94	59.24
	384	48.74	13.16	61.90	timeout
	960	438.07	288.03	726.10	timeout
c5315	48	3.44	0.45	3.89	0.49
	144	15.49	1.49	16.98	15.46
	384	88.83	9.12	97.94	955.38
	960	598.32	109.01	707.32	timeout
c7552	48	4.68	2.16	6.84	5.01
	144	22.00	5.17	27.18	17.85
	384	112.62	117.10	229.72	timeout
	960	702.99	timeout	timeout	timeout
des	48	4.74	0.65	5.38	1.02
	144	20.35	1.67	22.02	10.52
	384	99.89	11.53	111.42	1,733.37
	960	619.47	53.63	673.10	timeout
b14	48	7.10	2.39	9.49	2.14
	144	30.12	6.53	36.65	16.77
	384	141.99	42.01	184.00	768.14
	960	817.04	291.54	1,108.58	timeout
b15	48	11.67	2.07	13.74	2.04
	144	48.75	2.44	51.19	10.95
	384	228.81	11.64	240.45	1,752.04
	960	1,178.61	113.36	1,291.97	timeout
b17	48	55.52	5.47	60.99	5.70
	144	207.54	18.00	225.54	33.19
	384	848.97	35.90	884.87	965.29
	960	1,631.56	271.73	1,903.29	timeout
b18	48	129.43	80.65	210.08	37.92
	144	508.98	68.95	577.93	73.51
	384	1,987.38	221.13	2,208.51	1,071.78
	960	8,236.74	1,018.86	24,517.75	timeout
b19	48	342.45	1,883.71	2,226.16	1,600.44
	144	1,356.16	1,006.17	2,362.33	1,473.56
	384	5,181.09	1,679.80	6,860.89	5,222.07
	960	23,498.89	3,221.14	26,720.03	timeout

Both attacks resulted in a timeout if the circuit was not unlocked after 48 hrs (≈170,000 s). All times are in seconds.

Table 4. Our Method to Reduce the Number of Oracle Queries Drops the Query Count from $2N$ to as Low as $N + 1$

Circuit	Key Size	N	Independent Sensitization Time	Query Reuse Time	Time Reduction (%)	Query Reduction (%)
c1908	48	8	0.55	0.37	33.9	43.8
	144	16	3.37	1.89	43.8	46.4
	384	32	14.01	9.10	35.1	46.4
c2670	48	8	1.58	0.80	49.4	43.8
	144	16	7.87	4.23	45.9	46.9
	384	32	68.83	30.91	55.1	48.4
c3540	48	8	2.46	0.63	74.6	43.8
	144	16	11.02	3.39	69.2	46.9
	384	32	48.72	24.78	49.2	48.4
	960	64	438.07	281.24	35.8	49.2
c5315	48	8	3.44	1.23	64.1	43.8
	144	16	15.49	6.26	59.6	46.9
	384	32	88.83	38.50	56.7	48.4
	960	64	598.32	352.72	41.0	49.2
c7552	48	8	4.68	1.72	63.3	43.8
	144	16	22.00	8.36	62.0	46.9
	384	32	112.62	51.66	54.1	48.4
	960	64	702.99	382.78	45.6	49.2
des	48	8	4.74	1.84	61.1	43.8
	144	16	20.35	8.42	58.6	46.9
	384	32	99.89	50.45	49.5	48.4
	960	64	619.47	410.63	33.7	49.2
Average					51.9	

In our benchmark trials, this minimum bound of queries was achieved for all circuits, with the percentage reduction in oracle queries approaching 50% as the SAT-hard instance size increases. This method also completes the primary attack faster than the original sensitization algorithm.

At larger sizes of the SAT-hard instance, the sensitization attack becomes much more efficient than the SAT attack. Our sensitization attack unlocked every benchmark with 384 key bits and 8 of the 9 benchmark circuits with 960 key bits, the largest size tested. In contrast, 3 of the 11 benchmark circuits with 384 key bits could not be unlocked by the SAT attack within the test window of 48 hours. The SAT attack did not unlock any benchmarks with 960 key bits.

These experimental results show that our novel sensitization attack is able to quickly unlock designs obfuscated with Full-Lock even with sizable SAT-hard instances, which are not efficiently unlockable with traditional attack methods. Our results remain consistent across several circuit topologies with only one outlier benchmark.

11.3 Reduction of Oracle Query Count

We also tested our alternate sensitization method described in Section 5, which reuses oracle queries and so makes fewer of them than the method described in Section 4. To compare these

methods, we executed the primary sensitization attack using the method described in Section 5 on the ISCAS'89 and MCNC20 benchmarks used in the previous experiment: six benchmarks with 48, 144, and 384 key bits and four benchmarks with 960 key bits. We compare the sensitization attack data from the previous experiment to the duration of these new tests.

Table 4 shows our sensitization attack time with and without PI reuse. We see that reusing oracle queries reduces both the number of oracle queries and the time required to execute our attack. Our tool was able to sensitize the SAT-hard instance using the theoretical minimum number of oracle queries for each benchmark circuit, so in key sizes, we see that the percent reduction in oracle queries approaches 50% as $2N$ oracle queries are cut to only $N + 1$. For benchmarks where oracle query reduction is not equal to $\frac{2N-(N+1)}{2N}$, this is the result of some bits of $\mathbf{x}$ not being sensitizable. No oracle queries are made for these bits in either version of the attack, so the percent reduction in oracle queries appears lower than the value of N would suggest.

The attack using fewer oracle queries is also 51.9% faster, on average, and executes as much as 74.6% faster than the other method. This speedup occurs because large sections of the circuit-SAT problem in Figure 5 are simplified out when the value of $\mathbf{I}^{i,2}$ is fixed to an externally assigned value $\mathbf{I}^{*}$. Since $\mathbf{I}^{i,2}$ cannot change, the copy of the SAT-hard instance fanin that has $\mathbf{I}^{i,2}$ as its input has fixed output, and the solver does not need to evaluate this logic for every solution it considers. As a result, the problem size is cut nearly in half compared to the baseline problem in Figure 4, since the solver only needs to consider parts of the problem that are reachable by by $\mathbf{I}^{i,1}$, such as the copy of the fanin cone with $\mathbf{I}^{i,1}$ as its input.

12 Conclusion

In this article, we introduced a novel sensitization attack to recover the intended functionality of designs obfuscated with Full-Lock and InterLock, which are resilient against attacks by existing methodologies such as the SAT attack. Our novel attack infers the input-output relationship of the SAT-hard instance that Full-Lock and InterLock introduce to the circuit and efficiently unlocks the design.

The result is an increase in time efficiency compared to the traditional SAT attack, because the attacker avoids including the SAT-hard instance in the formulation of its Boolean satisfiability problems. The SAT problems the attacker solves are also similar to those used in IC testing, enabling the use of highly optimized algorithms available to design houses and foundry-based attackers. Our experimental data demonstrates the viability of the attack, which breaks nearly every circuit even at our largest key size. Most circuits were unlocked in 20 minutes or less by our novel sensitization attack, even though none were unlocked by the traditional SAT attack. This speedup is achieved with a number of oracle queries one larger than the input size of the SAT-hard instance added by Full-Lock obfuscation.

References

[1] Lilas Alrahis, Satwik Patnaik, Muhammad Abdullah Hanif, Hani Saleh, Muhammad Shafique, and Ozgur Sinanoglu. 2021. GNNUnlock+: A systematic methodology for designing graph neural networks-based oracle-less unlocking schemes for provably secure logic locking. *IEEE Trans. Emerg, Topics Comput.* 10, 3 (2021), 1575–1592.

[2] Lilas Alrahis, Satwik Patnaik, Faiq Khalid, Muhammad Abdullah Hanif, Hani Saleh, Muhammad Shafique, and Ozgur Sinanoglu. 2021. GNNUnlock: Graph neural networks-based oracle-less unlocking scheme for provably secure logic locking. In *Design, Automation & Test in Europe Conference & Exhibition (DATE'21)*. IEEE, 780–785.

[3] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. 2019. SMT attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the SAT attacks. *IACR Trans. Cryptog. Hardw. Embed. Syst.* 2019, 1 (2019), 97–122.

[4] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. 2020. NNgSAT: Neural network guided SAT attack on logic locked complex structures. In *39th International Conference on Computer-Aided Design (ICCAD'20)*. 1–9.

[5] Alex Baumgarten, Akhilesh Tyagi, and Joseph Zambreno. 2010. Preventing IC piracy using reconfigurable logic barriers. *IEEE Des. Test Comput.* 27, 1 (2010), 66–75.
[6] Robert Brayton and Alan Mishchenko. 2010. ABC: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification (CAV'10)*. Springer, 24–40.
[7] Abhishek Chakraborty, Nithyashankari Gummidipoondi Jayasankaran, Yuntao Liu, Jeyavijayan Rajendran, Ozgur Sinanoglu, Ankur Srivastava, Yang Xie, Muhammad Yasin, and Michael Zuzak. 2019. Keynote: A disquisition on logic locking. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 39, 10 (2019), 1952–1972.
[8] Jonathan Cruz, Farimah Farahmandi, Alif Ahmed, and Prabhat Mishra. 2018. Hardware trojan detection using ATPG and model checking. In *31st International Conference on VLSI Design and 17th International Conference on Embedded Systems (VLSID'18)*. IEEE, 91–96.
[9] Rolf Drechsler, Stephan Eggergluss, Gorschwin Fey, Andreas Glowatz, Friedrich Hapke, Juergen Schloeffel, and Daniel Tille. 2008. On acceleration of SAT-based ATPG for industrial designs. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 27, 7 (2008), 1329–1333.
[10] Sophie Dupuis and Marie-Lise Flottes. 2019. Logic locking: A survey of proposed methods and evaluation metrics. *J. Electron. Test.* 35, 3 (2019), 273–291.
[11] Mohamed El Massad, Siddharth Garg, and Mahesh V. Tripunitara. 2015. Integrated circuit (IC) decamouflaging: Reverse engineering camouflaged ICs within minutes. In *Network and Distributed System Security (NDSS'15) Symposium*. 1–14.
[12] Mark C. Hansen, Hakan Yalcin, and John P. Hayes. 1999. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Des. Test Comput.* 16, 3 (1999), 72–80.
[13] Hadi Mardani Kamali, Kimia Zamiri Azar, Houman Homayoun, and Avesta Sasan. 2019. Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks. In *56th Annual Design Automation Conference (DAC'19)*. 1–6.
[14] Hadi Mardani Kamali, Kimia Zamiri Azar, Houman Homayoun, and Avesta Sasan. 2020. InterLock: An intercorrelated logic and routing locking. In *39th International Conference on Computer-Aided Design (ICCAD'20)*. 1–9.
[15] Yuntao Liu, Michael Zuzak, Yang Xie, Abhishek Chakraborty, and Ankur Srivastava. 2020. Strong Anti-SAT: Secure and effective logic locking. In *21st International Symposium on Quality Electronic Design (ISQED'20)*. IEEE, 199–205.
[16] Isaac McDaniel, Michael Zuzak, and Ankur Srivastava. 2022. A black-box sensitization attack on SAT-hard instances in logic obfuscation. In *IEEE 40th International Conference on Computer Design (ICCD'22)*. IEEE, 239–246.
[17] M. K. Prasad, Philip Chong, and Kurt Keutzer. 1999. Why is ATPG easy? In *Design Automation Conference (DAC'99)*. IEEE, 22–28.
[18] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. 2012. Security analysis of logic obfuscation. In *49th Annual Design Automation Conference (DAC'12)*. 83–89.
[19] Jeyavijayan Rajendran, Huan Zhang, Chi Zhang, Garrett S. Rose, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. 2013. Fault analysis-based logic encryption. *IEEE Trans. Comput.* 64, 2 (2013), 410–424.
[20] Shervin Roshanisefat, Hadi Mardani Kamali, and Avesta Sasan. 2018. SRCLock: SAT-resistant cyclic logic locking for protecting the hardware. In *Great Lakes Symposium on VLSI (GLSVLSI'18)*. 153–158.
[21] Masoud Rostami, Farinaz Koushanfar, and Ramesh Karri. 2014. A primer on hardware security: Models, methods, and metrics. *Proc. IEEE* 102, 8 (2014), 1283–1295.
[22] Jarrod A. Roy, Farinaz Koushanfar, and Igor L. Markov. 2008. EPIC: Ending piracy of integrated circuits. In *Conference on Design, Automation and Test in Europe (DATE'08)*. 1069–1074.
[23] Akashdeep Saha, Sayandeep Saha, Siddhartha Chowdhury, Debdeep Mukhopadhyay, and Bhargab B. Bhattacharya. 2020. LoPher: SAT-hardened logic embedding on block ciphers. In *57th ACM/IEEE Design Automation Conference (DAC'20)*. IEEE, 1–6.
[24] Abhrajit Sengupta, Mohammed Nabeel, Nimisha Limaye, Mohammed Ashraf, and Ozgur Sinanoglu. 2020. Truly stripping functionality for logic locking: A fault-based perspective. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 39, 12 (2020), 4439–4452.
[25] Abhrajit Sengupta, Mohammed Nabeel, Muhammad Yasin, and Ozgur Sinanoglu. 2018. ATPG-based cost-effective, secure logic locking. In *IEEE 36th VLSI Test Symposium (VTS'18)*. IEEE, 1–6.
[26] B. Shakya, et al. 2019. CAS-Lock: A security-corruptibility trade-off resilient logic locking scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*. 2020, 1 (Nov. 2019), 175–202.
[27] Kaveh Shamsi, Meng Li, Travis Meade, Zheng Zhao, David Z. Pan, and Yier Jin. 2017. AppSAT: Approximately de-obfuscating integrated circuits. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST'17)*. IEEE, 95–100.
[28] Kaveh Shamsi, Meng Li, David Z. Pan, and Yier Jin. 2018. Cross-lock: Dense layout-level interconnect locking using cross-bar architectures. In *Great Lakes Symposium on VLSI (GLSVLSI'18)*. 147–152.

[29] Yuanqi Shen and Hai Zhou. 2017. Double DIP: Re-evaluating security of logic encryption algorithms. In *Great Lakes Symposium on VLSI (GLSVLSI'17)*. 179–184.
[30] Pramod Subramanyan, Sayak Ray, and Sharad Malik. 2015. Evaluating the security of logic encryption algorithms. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST'15)*. IEEE, 137–143.
[31] Joseph Sweeney, Marijn J. H. Heule, and Lawrence Pileggi. 2020. Modeling techniques for logic locking. In *39th International Conference on Computer-Aided Design (ICCAD'20)*. 1–9.
[32] Yang Xie and Ankur Srivastava. 2017. Delay locking: Security enhancement of logic locking against IC counterfeiting and overproduction. In *54th Annual Design Automation Conference (DAC'17)*. 1–6.
[33] Yang Xie and Ankur Srivastava. 2018. Anti-SAT: Mitigating SAT attack on logic locking. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 38, 2 (2018), 199–207.
[34] Xiaolin Xu, Bicky Shakya, Mark M. Tehranipoor, and Domenic Forte. 2017. Novel bypass attack and BDD-based tradeoff analysis against all known logic locking attacks. In *19th International Conference on Cryptographic Hardware and Embedded Systems (CHES'17)*. Springer, 189–210.
[35] Fangfei Yang, Ming Tang, and Ozgur Sinanoglu. 2019. Stripped functionality logic locking with Hamming distance-based restore unit (SFLL-hd)–Unlocked. *IEEE Trans. Inf. Forens. Secur.* 14, 10 (2019), 2778–2786.
[36] Saeyang Yang. 1991. *Logic Synthesis and Optimization Benchmark User Guide Version 3.0*. Microelectronics Center of North Carolina (MCNC).
[37] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan J. V. Rajendran, and Ozgur Sinanoglu. 2016. SARLock: SAT attack resistant logic locking. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST'16)*. IEEE, 236–241.
[38] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. 2017. Removal attacks on logic locking and camouflaging techniques. *IEEE Trans, Emerg, Topics Comput.* 8, 2 (2017), 517–532.
[39] Muhammad Yasin, Jeyavijayan J. V. Rajendran, Ozgur Sinanoglu, and Ramesh Karri. 2015. On improving the security of logic locking. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 35, 9 (2015), 1411–1424.
[40] Muhammad Yasin, Abhrajit Sengupta, Mohammed Thari Nabeel, Mohammed Ashraf, Jeyavijayan Rajendran, and Ozgur Sinanoglu. 2017. Provably-secure logic locking: From theory to practice. In *ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. 1601–1618.
[41] Muhammad Yasin and Ozgur Sinanoglu. 2017. Evolution of logic locking. In *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC'17)*. IEEE, 1–6.
[42] Hai Zhou, Amin Rezaei, and Yuanqi Shen. 2019. Resolving the trilemma in logic encryption. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD'19)*. IEEE, 1–8.
[43] Michael Zuzak, Yuntao Liu, and Ankur Srivastava. 2020. Trace logic locking: Improving the parametric space of logic locking. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 40, 8 (2020), 1531–1544.
[44] Michael Zuzak and Ankur Srivastava. 2020. ObfusGEM: Enhancing processor design obfuscation through security-aware on-chip memory and data path design. In *International Symposium on Memory Systems (MEMSYS'20)*. 260–271.

Received 6 May 2023; revised 17 November 2023; accepted 16 June 2024