A Black-Box Sensitization Attack on SAT-Hard Instances in Logic Obfuscation

Isaac McDaniel[1], Michael Zuzak[2], and Ankur Srivastava[1]
[1]Department of Electrical and Computer Engineering, University of Maryland, College Park, MD, USA
[2]Department of Computer Engineering, Rochester Institute of Technology, Rochester, NY, USA
ilm@umd.edu, mjzeec@rit.edu, ankurs@umd.edu

***Abstract*—Logic obfuscation is a prominent approach to protect intellectual property within integrated circuits during fabrication. In response to logic obfuscation, the Boolean satisfiability attack was developed and demonstrated to unlock a great deal of existing obfuscation configurations. This drove the development of new SAT-resistant obfuscation countermeasures. Some of these, including Full-Lock and InterLock, resist SAT attacks by inserting SAT-hard instances, rapidly scaling the runtime of each SAT attack iteration. In this work, we demonstrate that while such countermeasures resist SAT-style attack strategies, an attacker with access to the inputs and outputs of the SAT-hard instance Full-Lock has inserted into an oracle circuit can infer the design's intended functionality in linear time, thereby unlocking the circuit. We also observe that this class of obfuscation leaves most of the original design topology intact and show how this enables an attacker to sensitize the SAT-hard instance within a black-box oracle and make inferences about the instance's input-output relationship from the oracle's primary inputs and outputs. We develop a novel attack which uses this leakage to allow an attacker to efficiently unlock designs obfuscated with Full-Lock without the special assumption of access to the SAT-hard instance's inputs and outputs. This recovers the intellectual property and renders these obfuscation techniques insecure. We empirically demonstrate the potency of our novel sensitization attack against benchmark circuits obfuscated with SAT-hard instances. Our proposed attack was able to unlock all 6 benchmark circuits containing 384-bit keys and 3 out of 4 benchmarks with a 960-bit key within 48 hours. In comparison, the conventional SAT attack was only able to unlock 3 of 6 benchmarks with 384 key bits and none of the 4 benchmarks with 960 key bits in the same 48 hour timeout period.**

***Index Terms*—Logic Obfuscation, Full-Lock, Untrusted Foundry, Reverse Engineering**

I. 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 [1]. 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, whose values are collectively referred to as a key. In this way, the functionality of the design becomes dependent on the value of this key, and the design house protects the secret key, the key value which produces the intended functionality. By withholding this secret key from an untrusted fabrication partner, the intended functionality of a design is hidden. Such an approach mitigates security threats during fabrication. See [2], [3] for a comprehensive survey of logic obfuscation research.

In response to logic obfuscation, a family of Boolean satisfiability attacks, known as SAT attacks, were developed to unlock them [4]–[6]. These iterative attacks make the assumption that the attacker has access to a black-box oracle which can be queried for primary output values corresponding to the applied primary inputs. In one iteration of the original SAT attack, the attacker formulates a Boolean satisfiability (SAT) problem which is satisfied by two key values which 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 its next oracle query. This ensures that at least one of the keys which 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 potency of SAT-style attacks against logic obfuscation has driven the development of SAT-resilient obfuscation techniques. 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 [7], [8]. This family of approaches includes prominent techniques such as [9]–[15]. 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 [7], [16]. 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 [17], [18]. This family includes techniques such as Full-Lock [19] and Interlock [20]. The advantage of such an approach is that sizable error rates can be injected while maintaining resilience to SAT-style attacks [19], [20]. In this work, we narrow our scope to obfuscation techniques using this second approach.

A. Contributions

In this work, we explore Full-Lock and InterLock [19], [20], which are logic obfuscation techniques which resist SAT-style attacks by inserting SAT-hard instances to increase the runtime of the SAT problem solved in each attack iteration. We observe that the only obfuscation provided by these techniques

This work was supported by NSF grant 1953285.

is the functionality of the inserted SAT-hard instance, with the rest of the circuit topology remaining unchanged. We show that for Full-Lock and InterLock, the functionality of the SAT-hard instance, and by extension that of the obfuscated design, can be learned from a polynomial number of its input-output pairs. We exploit the topological rigidity of this type of obfuscation, paired with a black-box oracle, to partially leak these input-output pairs to a SAT-capable attacker attempting to unlock Full-Lock. We develop an attack which uses this leakage to reverse engineer the intended functionality of a Full-Lock-obfuscated netlist faster than the SAT attack alone. The contributions of this work can be summarized as follows:

- We introduce a method for an attacker to learn the intended functionality of the SAT-hard instance Full-Lock places in a design using a linear number of observations of its inputs and outputs in an oracle. Knowing this functionality allows an attacker to de-obfuscate the design.
- We extend this method to the more recent InterLock, which is similar to Full-Lock but uses a more complex SAT-hard instance. We describe how an attacker can manipulate the inputs to make the SAT-hard instance in InterLock equivalent to the one in Full-Lock so that the same analysis could be used to determine part of its functionality. We then show how the remaining functionality can be extracted in polynomial time.
- We formulate solvable problems whose solutions allow a SAT-capable attacker to make partial observations of Full-Lock's SAT-hard instance by 1) applying specific inputs to the SAT-hard instance in the oracle and 2) inferring possible instance outputs from the oracle's primary outputs. These are made possible through analysis of the obfuscated netlist. The problems we solve do not include the SAT-hard instance itself, bypassing the security guarantees of Full-Lock.
- We formalize an attack methodology using these partial observations. Each observation which is leaked completely exponentially reduces the functionality search space and, if all observations are leaked in their entirety, the attack completely de-obfuscates Full-Lock. Even if observations cannot be determined precisely, a partial solution is produced which the attacker can use to greatly accelerate a secondary attack using an existing method. Since Full-Lock resists current state of the art methods such as the SAT attack, this allows counterfeit copies of the unobfuscated design to be produced by an untrusted foundry previously unable to do so.
- We empirically evaluate our attack against benchmark circuits obfuscated with Full-Lock. After completing our proposed attack to produce a partial solution, we launch a secondary attack using an existing attack method to recover the total functionality of the design. During the experiment, our proposed attack unlocked benchmark circuits within 48 hours even with very large SAT-hard instances, including 3 of 4 benchmarks with 960 key bits and all 6 benchmarks with 384 key bits. Conversely, the conventional SAT attack could not unlock any benchmarks with 960 key bits and only 3 of 6 benchmarks with 384 key bits within the same 48 hours.

II. Preliminaries

A. Attacker Model

In this work, we assume a SAT-capable adversary common in recent logic obfuscation research, such as [7], [9]–[13], [19], [20]. 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.

B. 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.

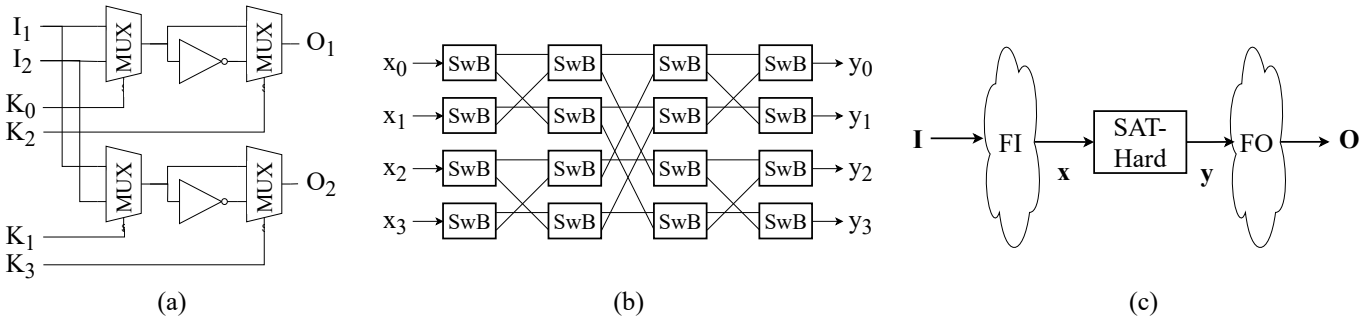
In Full-Lock [19], which we primarily focus on in this work, this module is a switching network whose functionality is made key-dependent through the use of programmable logic and routing (PLR) blocks. Each node in the network is a switch-box (SwB) which may exchange the input to output routing and invert each of 2 signals according to 3 key inputs. As a result, the outputs of the 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 [19]. 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 [7], [8]. Rather than hindering the SAT attack by reducing the number of inputs which 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.

III. Attacking Full-Lock by Querying SAT-Hard Instance

In a design obfuscated with Full-Lock, there is in the worst case one permutation of Full-Lock inputs which is functionally equivalent to the black-box oracle [19]. In this section, we show how an attacker can learn this single correct permutation from an N-input SAT-hard instance with $2N$ queries of the exposed inputs and outputs of the instance. While our attacker model does not allow direct access to these signals, in subsequent sections we will develop a method for an attacker to methodically leak the results of the SAT-hard instance queries from a black-box oracle through analysis of the obfuscated netlist.

This will require two steps: 1) sensitization of the SAT-hard instance, covered in Section IV, and 2) inference of



(a) (b) (c)

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.

instance outputs, covered in Section V. 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. Once this leakage is established, we formalize an attack on Full-Lock in Section VI which makes a sequence of queries of the oracle and attempts to leak the instance functionality. This produces a partial solution, which can be completed by a secondary attack using an existing method.

A. Signal Definitions

Our attacks depend on manipulation of the primary inputs and outputs of the circuit as well as the inputs and outputs of the SAT-hard instance. We define vectors of the latter as is commonly done for primary inputs and outputs: the instance input vector **x** and output vector **y**. These have the same length N, since the function of the SAT-hard instance is to permute its inputs. We will refer to the primary input vector as **I** and the primary output vector as **O**. The primary inputs and outputs are part of the design topology before obfuscation, so the lengths of **I** and **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 4 of these signals labeled.

B. 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 which can be solved individually to provide a partial solution. 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 one correct permutation of the inputs from all $N!$ permutations of the inputs. 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. After solving this smaller problem, finding the 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 solving these smaller problems 1 at a time, the attacker must consider only $\sum_{i=1}^{N} N-i$ solutions in total to determine the functionality of the instance, compared to the $N! = \prod_{i=1}^{N} N-i$ solutions considered by the holistic attack.

Now, we describe how the attacker can find the destination of an input with 2 queries of the SAT-hard instance in an oracle. Since the attacker can observe the inputs and outputs of the instance directly, we will define these to also be the primary inputs and outputs (**I** = **x**, **O** = **y**). First, we note that each output is a function of only 1 input, and that this output is the permutation destination of that input. Therefore, if the attacker makes 1 query of the SAT-hard instance to learn any input-output pairing, then changes 1 input bit, input $i \in 1, 2, \ldots, N$, and makes another query, the only output bit to change will be the changed input's permutation destination, output $j \in 1, 2, \ldots, N$. We represent the input and output of the first query as $\mathbf{x}_1^i$ and $\mathbf{y}_1^j$, though j is not known from this first query. The second query results are $\mathbf{x}_2^i$ and $\mathbf{y}_2^j$.

TABLE I: Sample I/O relationships showing how the functionality of a fully exposed SAT-hard instance can be learned in linear time. If the instance has N permutation inputs, then the functionality is learned after checking the oracle $2N$ times.

	x		**y**	j	Learned Mapping
$\mathbf{x}_1^i$	0000	$\mathbf{y}_1^j$	1101		
$\mathbf{x}_2^1$	0001	$\mathbf{y}_2^j$	1111	2	$x_1 \rightarrow y_2$
$\mathbf{x}_2^2$	0010	$\mathbf{y}_2^j$	0101	4	$\neg x_2 \rightarrow y_4$
$\mathbf{x}_2^3$	0100	$\mathbf{y}_2^j$	1100	1	$\neg x_3 \rightarrow y_1$
$\mathbf{x}_2^4$	1000	$\mathbf{y}_2^j$	1001	3	$\neg x_4 \rightarrow y_3$

Here, for a Boolean vector **u**, we use the notation $\mathbf{u}_1^n$ and $\mathbf{u}_2^n$ to represent 2 values of **u** which have a Hamming distance of 1, with the single differing bit in the nth place. Thus, the second input value x_2^i has $x_{2,k}^i = x_{1,k}^i$ for $k \neq i$ and $x_{2,i}^i = \neg x_{1,i}^i$ and the second output has $y_{2,k}^j = y_{1,k}^i$ for $k \neq j$ and $y_{2,j}^j = \neg y_{1,j}^j$ for input i's destination output j. The attacker chooses i, the input bit which changes, but must observe j, the output bit which shows a response. With this process, the attacker has learned the destination of an obfuscated signal after observing just two SAT-hard instance input/output pairs.

After the attacker has learned the destination of 1 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. The same process is repeated iteratively until the total functionality is known. Thus, the attacker can learn the functionality of the SAT-hard instance in $2N$ oracle queries.

As an example, we examine the SAT-hard instance queries in Figure 2, which produce an initial input-output pair $\mathbf{x}^i = 0000$, $\mathbf{y}^j = 1101$. Setting $i = 1$, the attacker then toggles bit 1 of **x**, and finds the I/O pair $\mathbf{x}_2^1 = 0001$, $\mathbf{y}_2^j = 1111$, revealing that the changing output is $j = 2$. From these two data points,

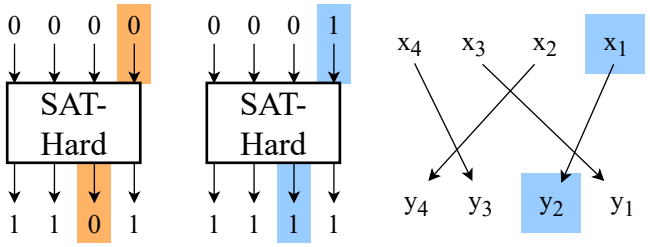


Fig. 2: An example of learning the functionality of an exposed SAT-hard instance. In (a) the attacker generates a reference I/O pair, and in (b) the attacker is able to compare a second I/O pair and learn that the instance passes input I_0 to output O_1. (c) shows the correct permutation after observing the example data from Table 1.

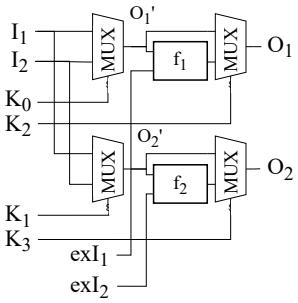


Fig. 3: Diagram of the SAT-hard instance used in InterLock, showing the switch-box (SwB) circuit modified from Full-Lock to include two extra inputs going to function blocks. These replace the inverters originally present in Full-Lock.

the attacker infers that x_1, the first bit of $\mathbf{x}$, is permuted to y_2, the second bit of $\mathbf{y}$. Now setting $i = 2$, the attacker can use the same first input $\mathbf{x}_1^2 = 0000$ and add a new query for $\mathbf{x}_2^2 = 0010$. Finding $\mathbf{y}_2^4 = 0101$ reveals $\neg x_2$ is permuted to y_4.

After iterating through each bit of $\mathbf{x}$, the attacker has learned the permutation of all 4 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 I shows all of the I/O pairs the attacker finds and the information learned from each one, while Figure 2 graphically shows the first step of the process as well as the mapping from $\mathbf{x}$ to $\mathbf{y}$ that the attacker constructs.

C. Extension to InterLock

The authors of Full-Lock have since introduced InterLock [20], which improves Full-Lock by increasing the complexity of the switch-boxes used in the SAT-hard instance. Recall that after possibly switching their two inputs, Full-Lock switch-boxes have a second stage of multiplexers which give them the option to pass or invert each signal. InterLock replaces the inverters with two-input gates, such as AND, OR, XOR, etc. Each gate with logic function f_i and inputs O_i', the output of the first multiplexer, and exI_i, a new input to the SAT-hard instance added by InterLock. The latter pass from the original circuit to each individual switch-box. The new gate add 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 3 shows the new switch-box diagram introduced by InterLock.

This introduces 2 difficulties we must overcome to extend the method we have developed in Section III-B 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_i with which other inputs exI_i are applied to the intermediate signals O_i'.

Difficulty 1 can be addressed by assigning each exI_i to a certain value. Importantly, the attacker knows the function f_i of the gate each exI_i goes to. Furthermore, for any 2-input f_i, there is some value of exI_i such that $f_i = O_i'$ or $f_i = \neg O_i'$. We say that this value of exI_i sensitizes f_i to O_i'. The attacker can set exI_i to its sensitizing value so the output of each switch-box, and therefore the entire SAT-hard instance, is a permutation and possible inversion of its inputs, as was the case for Full-Lock. The attacker can now use the method previously described for Full-Lock to determine the permutation destination of each SAT-hard instance input. However, in this case, this does not give the total functionality of the instance, since this analysis has been done for a special case with constraints on the extra inputs exI_i. To recover the total functionality, the attacker must now drop these constraints and address difficulty 2.

To learn which f_i and exI_i are applied to each input, the attacker must manipulate each exI_i to see which, if any, output y_j is affected. The attacker can toggle each exI_i one at a time to see whether it affects any output y_j. If it does, the attacker knows to apply f_i to the corresponding input signal after removing the SAT-hard instance. Otherwise, there are 2 explanations: 1) this exI_i is unused and the corresponding f_i is bypassed, or 2) f_i is not sensitized to exI_i at the current value of I_i. To distinguish between these, a second pass through exI_i is needed. The attacker first inverts the value of every instance input x_i then again toggles each exI_i which didn't affect any y_j in the first pass. If some exI_i produces no change to the outputs this time, the attacker knows that this exI_i is not actually used, since every combination of O_i' and exI_i has been tried but the value of exI_i has not affected the output.

Once this process is complete, the attacker knows every f_i and exI_i which 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 4 oracle queries per switch-box. Since each stage of the network has $O(N)$ switch-boxes and there are $O(\log(N))$ stages in the network, this increases the complexity of the attack from $O(N)$ for Full-Lock to $O(N \log(N))$ for InterLock. While this does increase complexity, it can still be completed in polynomial time, unlocking InterLock efficiently.

The methodologies in this section have assumed an attacker with access to the inputs and outputs of the SAT-hard instance, which is stronger than the SAT-capable attacker model. In further sections, we will describe how our weaker attacker model can leak enough information from a black-box oracle to use these methods to attack Full-Lock even without direct access to the inputs and outputs of the SAT-hard instance. Although the rest of this paper is focused on an attack on a design obfuscated with Full-Lock, the methods we describe in

Sections IV-VII can be adapted to apply to InterLock.

IV. 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 in constant time, and the block's total functionality in linear time. 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.

Applying our technique to the SAT-capable attacker model requires analysis of the obfuscated netlist to select inputs for an oracle query which will apply inputs $\mathbf{x}_1^i$ and $\mathbf{x}_2^i$ to the SAT-hard instance. To do this, we slightly generalize the attacker in the previous section by allowing the primary inputs to differ from the instance inputs ($\mathbf{I} \neq \mathbf{x}$), but still requiring the primary outputs to be the same as the instance outputs ($\mathbf{O} = \mathbf{y}$).

We define a sensitizing input as a pair of primary input values $\mathbf{I}_1^i, \mathbf{I}_2^i$ which produce the SAT-hard instance inputs $\mathbf{x}_1^i, \mathbf{x}_2^i$. We say that i is the sensitized input. The instance, and in this case primary, outputs for the same inputs are $(\mathbf{y}_1^j, \mathbf{y}_2^j)$, where we say that j is the sensitized output. As in the case where the attacker is able to directly query the SAT-hard instance, j is the permutation destination of instance input i. A successful attack in this case must find a sensitizing input $(\mathbf{I}_1^i, \mathbf{I}_2^i)$ for every SAT-hard instance input $i \in [1, N]$ 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 i, we create two copies of the logic between the instance and the primary inputs with input vectors $\mathbf{I}_1^i$ and $\mathbf{I}_2^i$ and output vectors $\mathbf{x}_1^i$ and $\mathbf{x}_2^i$. We will add logic to these to build a miter circuit. The values of $\mathbf{I}_1^i, \mathbf{I}_2^i$ are the solution to the problem, so these are not altered and remain the inputs of the miter circuit. Logic added to the outputs $\mathbf{x}_1^i, \mathbf{x}_2^i$ need to force them to meet the requirements $x_{1,i}^i = \neg x_{2,i}^i$ and $x_{1,k}^i = x_{2,k}^i, k \neq i$. This can be done by adding N logic gates $G_k = f_k(x_{1,k}^i, x_{2,k}^i), k \in [1, N]$, 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 in order to satisfy the single output of the miter circuit. Since the miter, shown in Figure 4(a), is only satisfied when the instance is sensitized at input i, any primary input values $\mathbf{I}_1^i, \mathbf{I}_2^i$ which satisfy the miter must be a sensitizing input. Once the attacker has found ($\mathbf{I}_1^i$ and $\mathbf{I}_2^i$), they can be applied to the black-box oracle to observe the outputs $\mathbf{O}_1^j = \mathbf{y}_1^j$ and $\mathbf{O}_2^j) = \mathbf{y}_2^j$, which reveal j is the permutation destination of 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 which differ only at the desired bit. When this happens, it is not possible to sensitize the SAT-hard instance to 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 a much 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 1 SAT-hard instance input, each time reducing the effective number of signals permuted by the obfuscation by 1 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 which 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 [21], [22]. They have also seen use in other security applications [10], [11], [23].

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.

V. Leakage of 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 the outputs of the instance can be partially leaked 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) which 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

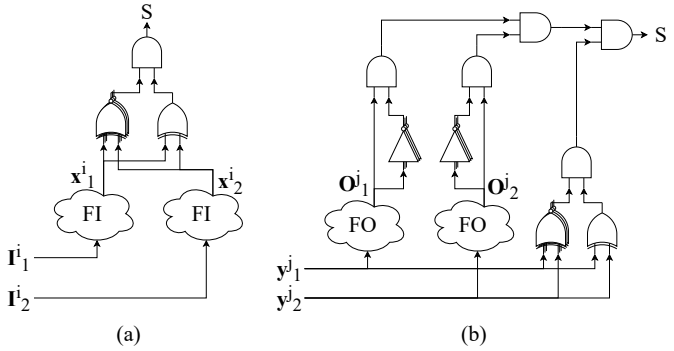


Fig. 4: (a) Miter circuit for finding a sensitizing input for a SAT-hard instance input. (b) Miter circuit which tests an observed output pattern for sensitivity to a particular input bit. In both miter circuits, the inclusion of one XOR gate forces the condition that $\mathbf{x}^i_1$ and $\mathbf{y}^j_1$ have a Hamming distance of 1 from $\mathbf{x}^i_2$ and $\mathbf{y}^j_2$.

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 j has inverted while all other outputs remain the same. While the value of j is unknown, the attacker can build a list of candidate outputs by testing each instance output $n \in [1, N]$ 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 n could be the sensitized output 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 2 copies of the SAT-hard instance fanin. To test whether an instance output could be sensitized, the miter is constructed around copies of the instance fanout, with $\mathbf{y}^j_1$ and $\mathbf{y}^j_2$ as their inputs and $\mathbf{O}^j_1$ and $\mathbf{O}^j_2$ as their outputs. Since the attacker is interested in whether any value of $\mathbf{y}^j_1, \mathbf{y}^j_2$ is consistent with the oracle query, these are also the input to the overall miter circuit. To enforce the condition that output n is sensitized, i.e., $y^j_{1,n} = \neg y^j_{2,n}$ and $y^j_{1,k} = y^j_{2,n}, k \neq n$, logic gates $G_k = f_k(y^j_{1,k}, y^j_{2,k}), k \in [1, N]$, where f_n is XOR and $f_k, k \neq i$ is XNOR. In addition, the attacker requires the satisfying value of $(\mathbf{y}^j_1, \mathbf{y}^j_2)$ to produce the observed primary output $\mathbf{O}^j_1, \mathbf{O}^j_2$. This is done by adding a $2N$-input AND gate which 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 4(b). If the miter is satisfiable, then the instance output n must be added to the list of candidates for j.

We use this analysis repeatedly to prune the search space of the functionality of the SAT-hard instance. N oracle queries, one which 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 i, i.e. for each oracle query, if the attacker determines that m_i of the N outputs could not be the permutation destination j for 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 1 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. However, as in the previous section, the functionality of the SAT-hard instance has already been extensively pruned, so the 2 subsequent attacks still obtain a total solution more quickly than an attack using only the secondary method. As when finding sensitizing inputs, the efficiency of this leakage 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.

VI. Full-Lock Functionality Recovery by a SAT-Capable Attacker

Finally, let us consider our total attack surface. Generally, an attacker has access only to the primary inputs **I** and outputs **O** of a black-box oracle and there is logic between these and inputs **x** and outputs **y** of the SAT-hard instance, so $\mathbf{x} \neq \mathbf{I}$ and $\mathbf{y} \neq \mathbf{O}$. This describes any Full-Lock implementation targeted by any SAT-capable attacker.

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 leak its possible outputs. This can be done by combining the prior two algorithms. First, the attacker analyzes the instance fanin with the miter in Figure 4(a) to find a sensitizing primary input $\mathbf{I}^i_1, \mathbf{I}^i_2$ for each instance input $i \in \{1, ..., N\}$. This can be performed exactly as described in Section IV, since the nonempty fanout cone of the SAT-hard instance does not affect the topology or function of the fanin. The attacker queries the oracle for each sensitizing input, but unlike in IV, 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 V, 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 4(b) to evaluate which instance outputs $n \in \{1, ..N\}$ 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. Additionally, our attack increases efficiency by not including the SAT-hard instance in a SAT formulation. Avoiding the SAT solver's worst-case scenario this way is what allows it to reduce the size of the problem faster than existing attacks.

VII. 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 2 reasons:

1) Our attack sensitizes SAT-hard instance inputs by learning 2 primary input vectors I_1^i, I_2^i which produce 2 instance inputs X_1^i, X_2^i which 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 V and VI, 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 j could not be the permutation destination of i and True otherwise. We have developed a tool which 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 which were previously the SAT-hard instance inputs which 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 which 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 secondary attack can be launched using this modified netlist and the existing black-box oracle.

VIII. 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.

TABLE II: Sensitization attack and SAT attack durations for various key sizes. Both attacks resulted in a timeout if the circuit was not unlocked after 48 hrs (170,000 s). All times are in seconds.

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	8458.36	8527.19	8708.47
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	1733.37
	960	619.47	53.63	673.10	timeout

The source code for our attack used the ABC synthesis tool [24] to parse and model benchmark circuits locked with Full-Lock [19]. We then extended the tool's functionality to implement the attack on 5 benchmarks selected from the ISCAS '85 suite [25] and 1 benchmark from MCNC20 [26], each obfuscated with Full-Lock using 3 or 4 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 leakage of SAT-hard instance outputs. This is the most general form of our attack, which can be launched by any SAT-capable attacker.

To perform the attack, we prepared black-box oracles and benchmark circuits obfuscated with Full-Lock in the Berkeley Logic Interchange Format [27]. We created an extension of ABC which extracts the SAT-hard instance fanin cone from the obfuscated netlist, constructs the miter circuit in Figure 4(a), and uses ABC's SAT solver to find sensitizing inputs. It then queries the oracle circuit and uses those results, along with the fanout cone of the SAT-hard instance, to form the miter in Figure 4(b) and leak information about the SAT-hard instance functionality. Our ABC extension concludes by producing a Boolean matrix recording which permutation destinations are possible for each instance input, as described in Section VII. We also built a tool which modifies the obfuscated netlist to replace the SAT-hard instance with multiplexers, also described in Section VII. We obtained a total solution from the modified netlists by launching a secondary SAT attack with the lazy-sat tool [4]. Our control data using the conventional SAT attack was also taken with this tool.

We tested our attack against 6 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 largest 4 benchmarks, we also tested with 960 bits.

Table II 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. However, at these sizes our attack's longest runtime is 27 s among all benchmarks, which is very small compared to the length of the 48 hr timeout window. Furthermore, there is a noticeable acceleration of the secondary SAT attack, which uses the output of the sensitization tool, compared to the time taken by the standard SAT attack.

At larger sizes, the sensitization attack becomes much more efficient, as with 384 key bits, the sensitization attack unlocked every benchmark with 384 key bits. For even the largest Full-Lock size with 960 key bits, 3 of the 4 large benchmark circuits were unlocked, each with a runtime of approximately 12 minutes (720 s). In contrast, 3 of the 6 benchmark circuits 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 existing attack methodologies. Our results remain consistent across several circuit topologies with only one outlier benchmark.

IX. Conclusion

In this paper, we introduced a novel sensitization attack to recover the intended functionality of a design obfuscated with Full-Lock, which is resilient against attacks by existing methodologies such as the SAT attack. Our novel attack uses structure of the logical function of the obfuscation to infer functionality by comparing sections of the circuit not altered by the obfuscation to the inputs and outputs of a black-box oracle.

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 for our largest key size reduced attack runtime for nearly every circuit from over 48 hours for the traditional SAT attack to less than 15 minutes for our novel sensitization attack.

References

[1] M. Rostami, F. Koushanfar, and R. Karri, "A primer on hardware security: Models, methods, and metrics," Proceedings of the IEEE, vol. 102, no. 8, pp. 1283–1295, 2014.

[2] S. Dupuis and M.-L. Flottes, "Logic locking: A survey of proposed methods and evaluation metrics," Journal of Electronic Testing, vol. 35, no. 3, pp. 273–291, 2019.

[3] A. Chakraborty, N. G. Jayasankaran, Y. Liu, J. Rajendran, O. Sinanoglu, A. Srivastava, Y. Xie, M. Yasin, and M. Zuzak, "Keynote: A disquisition on logic locking," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 10, pp. 1952–1972, 2019.

[4] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the security of logic encryption algorithms," in 2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 2015, pp. 137–143.

[5] M. El Massad, S. Garg, and M. V. Tripunitara, "Integrated circuit (ic) decamouflaging: Reverse engineering camouflaged ics within minutes." in NDSS, 2015, pp. 1–14.

[6] K. Z. Azar, H. M. Kamali, H. Homayoun, and A. Sasan, "Smt attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the sat attacks," IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 97–122, 2019.

[7] M. Zuzak, Y. Liu, and A. Srivastava, "Trace logic locking: Improving the parametric space of logic locking," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 40, no. 8, pp. 1531–1544, 2020.

[8] H. Zhou, A. Rezaei, and Y. Shen, "Resolving the trilemma in logic encryption," in 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2019, pp. 1–8.

[9] Y. Xie and A. Srivastava, "Anti-sat: Mitigating sat attack on logic locking," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 2, pp. 199–207, 2018.

[10] A. Sengupta, M. Nabeel, M. Yasin, and O. Sinanoglu, "Atpg-based cost-effective, secure logic locking," in 2018 IEEE 36th VLSI Test Symposium (VTS). IEEE, 2018, pp. 1–6.

[11] A. Sengupta, M. Nabeel, N. Limaye, M. Ashraf, and O. Sinanoglu, "Truly stripping functionality for logic locking: A fault-based perspective," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 12, pp. 4439–4452, 2020.

[12] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. Rajendran, and O. Sinanoglu, "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.

[13] B. Shakya, X. Xu, M. Tehranipoor, and D. Forte, "Cas-lock: A security-corruptibility trade-off resilient logic locking scheme," IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 175–202, 2020.

[14] M. Yasin, J. J. Rajendran, O. Sinanoglu, and R. Karri, "On improving the security of logic locking," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 9, pp. 1411–1424, 2015.

[15] Y. Liu, M. Zuzak, Y. Xie, A. Chakraborty, and A. Srivastava, "Strong anti-sat: Secure and effective logic locking," in 2020 21st International Symposium on Quality Electronic Design (ISQED). IEEE, 2020, pp. 199–205.

[16] M. Zuzak and A. Srivastava, "Obfusgem: Enhancing processor design obfuscation through security-aware on-chip memory and data path design," in The International Symposium on Memory Systems, 2020, pp. 260–271.

[17] A. Saha, S. Saha, S. Chowdhury, D. Mukhopadhyay, and B. B. Bhattacharya, "Lopher: Sat-hardened logic embedding on block ciphers," in 2020 57th ACM/IEEE Design Automation Conference (DAC). IEEE, 2020, pp. 1–6.

[18] K. Shamsi, M. Li, D. Z. Pan, and Y. Jin, "Cross-lock: Dense layout-level interconnect locking using cross-bar architectures," in Proceedings of the 2018 on Great Lakes Symposium on VLSI, 2018, pp. 147–152.

[19] H. M. Kamali, K. Z. Azar, H. Homayoun, and A. Sasan, "Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks," in Proceedings of the 56th Annual Design Automation Conference 2019, 2019, pp. 1–6.

[20] ——, "Interlock: An intercorrelated logic and routing locking," in 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD). IEEE, 2020, pp. 1–9.

[21] M. Prasad, P. Chong, and K. Keutzer, "Why is atpg easy?" in Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361). IEEE, 1999, pp. 22–28.

[22] R. Drechsler, S. Eggergluss, G. Fey, A. Glowatz, F. Hapke, J. Schloeffel, and D. Tille, "On acceleration of sat-based atpg for industrial designs," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 7, pp. 1329–1333, 2008.

[23] J. Cruz, F. Farahmandi, A. Ahmed, and P. Mishra, "Hardware trojan detection using atpg and model checking," in 2018 31st international conference on VLSI design and 2018 17th international conference on embedded systems (VLSID). IEEE, 2018, pp. 91–96.

[24] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in International Conference on Computer Aided Verification. Springer, 2010, pp. 24–40.

[25] M. C. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the iscas-85 benchmarks: A case study in reverse engineering," IEEE Design & Test of Computers, vol. 16, no. 3, pp. 72–80, 1999.

[26] S. Yang, "Logic synthesis and optimization benchmark user guide version 3.0," MCNC, Jan. 1991, 1991.

[27] U. Berkeley, "Berkeley logic interchange format (blif)," Oct Tools Distribution, vol. 2, pp. 197–247, 1992.