IMPLEMENTATION AND ASSESSMENT OF
STRONG LOGIC ENCRYPTION TECHNIQUES

by

Shervin Roshanisefat

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Electrical and Computer Engineering

Committee:

_____	Dr. Avesta Sasan, Dissertation Director
_____	Dr. Kris Gaj, Committee Member
_____	Dr. Bijan Jabbari, Committee Member
_____	Dr. Liang Zhao, Committee Member
_____	Monson Hayes, Department Chair
_____	Kenneth Ball, Dean, Volgenau School of Engineering

Date: 07/26/2021

Summer Semester 2021
George Mason University
Fairfax, VA

Implementation and Assessment of Strong Logic Encryption Techniques

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Shervin Roshanisefat
Master of Science
University of Tehran, 2015
Bachelor of Science
Qazvin Azad University, 2009

Director: Dr. Avesta Sasan, Professor
Department of Electrical and Computer Engineering

Summer Semester 2021
George Mason University
Fairfax, VA

Table of Contents

List of Tables

List of Figures

Abstract

IMPLEMENTATION AND ASSESSMENT OF STRONG LOGIC ENCRYPTION TECHNIQUES

Shervin Roshanisefat, PhD

George Mason University, 2021

Dissertation Director: Dr. Avesta Sasan

The increasing cost of building, operating, managing, and maintaining state-of-the-art silicon manufacturing facilities has pushed several stages of the semiconductor devices manufacturing supply chain offshore. However, many of these offshore facilities are identified as untrusted entities. Processing and fabrication of ICs in an untrusted supply chain pose a number of challenging security threats such as IC overproduction, Trojan insertion, Reverse Engineering, Intellectual Property (IP) theft, and counterfeiting.

To counter these threats, various hardware design-for-trust techniques have been proposed. Logic obfuscation, as a proactive technique among these techniques, has been introduced as a technique that obfuscates and conceals the functionality of IC/IP using additional key inputs that are driven by an on-chip tamper-proof memory.

Shortly after introducing the primitive logic locking solutions, a very strong attack based on the satisfiability solvers (SAT) was shown that could break all previously proposed locking mechanisms in almost polynomial time. To thwart this attack, researchers have investigated many directions, such as formulating locking solutions that significantly increase the number of required SAT iterations, or formulating the locking solutions such that it is not translatable to a SAT problem.

However, further investigations demonstrated that some of these locking techniques are vulnerable to other types of attacks such as removal attacks, approximate attacks, bypass attack, and Satisfiability Module Theories (SMT) attack. Besides, these techniques suffer from very low output corruption. Hence, an inactivate IC behaves almost identical to an unlocked IC except one or a few inputs.

For addressing these challenges, first, we will characterize the SAT attack, which shows that how using different SAT solvers can produce different results with large deviations which demonstrates that long execution time or high memory usage in one SAT solver may not be a problem in another solver. Next, we discuss a branch of SAT-resilient methods called cyclic locking and propose efficient methods to introduce feedbacks into a circuit in a way that SAT and its improved versions for cyclic circuits could not find the correct key. Then, we discuss a new branch of obfuscation techniques that tries to restrict access to the scan chain and thus circumvent the SAT attack. In there, we discuss a new attack method called unrolling SAT that potentially could be used for breaking obfuscated scan chains and recover the protected design. Last, we propose an open source framework that replaces SAT solvers with formal verification tools and could be used on a broader range of problems.

Chapter 1: Introduction

1.1 Why Hardware Obfuscation

Cost of building a new semiconductor fab was estimated to be US $5.0 billion in 2015, with large recurring maintenance costs [1], and sharply increases as technology migrates to smaller nodes. Due to the high cost of building, operating, managing, and maintaining state-of-the-art silicon manufacturing facilities, many major U.S. high-tech companies have been always fabless or went fabless in recent years. To reduce the fabrication cost, and for economic feasibility, most of the manufacturing and fabrication is pushed offshore [1]. However, many offshore fabrication fabs are untrusted, which has raised concern over potential attackers that include the manufacturers, with an intimate knowledge of the fabrication process, the ability to modify and expand the design prior to production, and an unavoidable access to the fabricated chips during testing. Hence, fabrication in untrusted fabs has introduced multiple forms of security threats from supply chain including that of overproduction, Trojan insertion, Reverse Engineering (RE), Intellectual Property (IP) theft, and counterfeiting [2].

Hardware obfuscation is the process of hiding the functionality of an IP by building ambiguity or by implementing post-manufacturing means of control and programmability into a netlist [3–5]. Gate camouflaging [6–8] and logic locking [9, 10] are two of the widely explored obfuscation mechanisms for this purpose. A camouflaged gate is a gate that after reverse engineering (using delayering and lithography) could be mapped to any member of a possible set of gates or may look like one logic gate (e.g., AND), however, functionally perform as another (e.g., XOR). In locking solutions, the functionality of a circuit is locked using several key inputs such that only when a correct key is applied, the circuit resumes its expected functionality. Otherwise, the correct function is hidden among many of the 2^K

(K being the number of keys) circuit possibilities. The claim raised by such an obfuscation scheme was that to break the obfuscation, an adversary needs to try a large number of inputs and key combinations to extract the correct key, and the difficulty of this process increases exponentially as the number of keys and primary inputs increases. Hence, if enough gates are obfuscated, an adversary faces an unacceptably long time (claimed as years to decades) to break the obfuscation scheme. Note that the availability of scan chains, which is inserted following Design for Test (DFT) recommended flow, allows an adversary to access combinational logic in each stage of a sequential circuit, load the desired input, execute the stage for one cycle, and readout the output.

1.2 Challenges in Hardware Obfuscation

The validity and strength of the state-of-the-art logic locking solutions to protect IPs/ICs against adversaries in the manufacturing supply chain was seriously challenged in recent years after the introduction of the Boolean satisfiability attack (SAT-Attack) [11–14]. SAT attack is an oracle-guided attack that requires access to the reverse engineered netlist and also a working copy of the IC to apply and capture interesting I/O pairs from it. After introduction of the SAT attack, researchers investigated a body of locking solutions with the objective of resisting the SAT attack. The revelation of SAT attack redirected the attention of the researchers to find harder obfuscation schemes that protect acyclic Boolean logic and resist the SAT attack. These methods have targeted a number of weaknesses in the SAT attack and could be categorized into three categories:

Weaker Distinguishing Inputs

Original SAT attack was powerful because it could iteratively rule out several wrong keys and constrain the key space effectively by finding interesting inputs called distinguishing inputs (DIP). The SARLock and Anti-SAT [15, 16] logic locking methods were proposed to mitigate this vulnerability. In a circuit protected by these solutions, a wrong key produces

a wrong output only for one input. This will restrict the SAT attack effectively since it can only rule out one wrong key per iteration. Hence, a SAT attack will be reduced to a Brute-force attack as it requires an exponential number of inquires to find the correct key. A design protected by these mechanisms, regardless of the key used for its activation, behaves very similar to the original design (except for one input). Hence, this group of obfuscation solutions suffers from low output corruption. To increase the output corruption, they could be augmented with other (output corruption oriented) obfuscation mechanisms. However, by using approximate SAT attack [17] almost all key values for the augmented obfuscation mechanism could be correctly identified.

Further research revealed that these obfuscation techniques are vulnerable to removal [18], Bypass [19] and FALL [20] attacks. In a removal attack, these SAT hard blocks are identified using Signal Probability Skew (SPS) attack [18] and removed. In Bypass attack [19], an auxiliary circuit that recovers the wrong output in these locking schemes is created. This attack identifies the input combinations that produce the wrong output for a wrong key; then it adds a bypass circuit to flip the wrong output when that specific input is applied. In FALL attacks, a functional analysis of the circuit will be performed and have two stages. In the first stage, it analyses the functionality of the obfuscated circuit and tries to identify the locking keys. If there was more than one candidate for the locking key, it tries to use the SAT to find the correct locking key from a list of alternatives and using simulations on the unlocked circuit.

Increasing Circuit-SAT Complexity

Another feature that makes the SAT attack powerful is the fast execution time of the underlying SAT solver in solving the circuit SAT and extracting DIPs. For locking schemes in this category, the netlist is designed in a way that translates to a large circuit-SAT with possibly a SAT-hard portion and thus requires more time to solve. Cross-Lock [21] exploited this vulnerability by adding cross-bars to the netlist and obfuscates circuit connections. Equivalent circuit-SAT in this method requires large multiplexers and the symmetric nature

of this block will make it a SAT-hard problem [22–24]. Without any additional clauses, any SAT solver requires a long execution time to find a single distinguishing input.

Netlists with camouflaged or memory-based blocks could also be used for this purpose. For these blocks, an equivalent circuit should be used that replaces them. For blocks with a large number of input and key ports, the equivalent circuit could be very large. This is especially true in the case of a locked circuit with large LUTs. This could lead to a large circuit-SAT with lots of SAT clauses. Recently, machine learning-based [25] and modeling-based [26] attacks were proposed that could break these obfuscation techniques.

SAT Unsolvable Structures

SAT attack needs to translate the reverse-engineered netlist into CNF clauses to be able to use the underlying SAT solver. Memory-blocks and Boolean gates could be easily translated into CNF clauses using equivalent circuits and Tseitin [27] transformation. Boolean limitation of SAT solvers could be used as a vulnerability to implement non-Boolean structures to counter the SAT attack.

Delay Locking [28] is one of such methods. It uses key-gates to lock both the functionality and the timing behavior of the obfuscated circuits. The logic aspect of the locking could be easily translated to CNF, however, the behavioral (timing) aspect of circuit operation can not be easily translated into a SAT friendly CNF. Hence, formulating a SAT attack on a delay-locked netlist will produce a circuit of correct functionality, but the timing violations will make the circuit malfunctioning. This method could potentially prevent overproduction or any reuse of fabrication materials like masks, but it can not prevent reverse engineering and IP-theft of the design. Also, an attack called TimingSAT [29] was later proposed to break this obfuscation method.

1.3 Organization of the Report

The rest of this thesis is organized as follows. In chapter 2, we cover the background on threat models and formal definition of the SAT attack. Then, in chapter 3, we evaluate

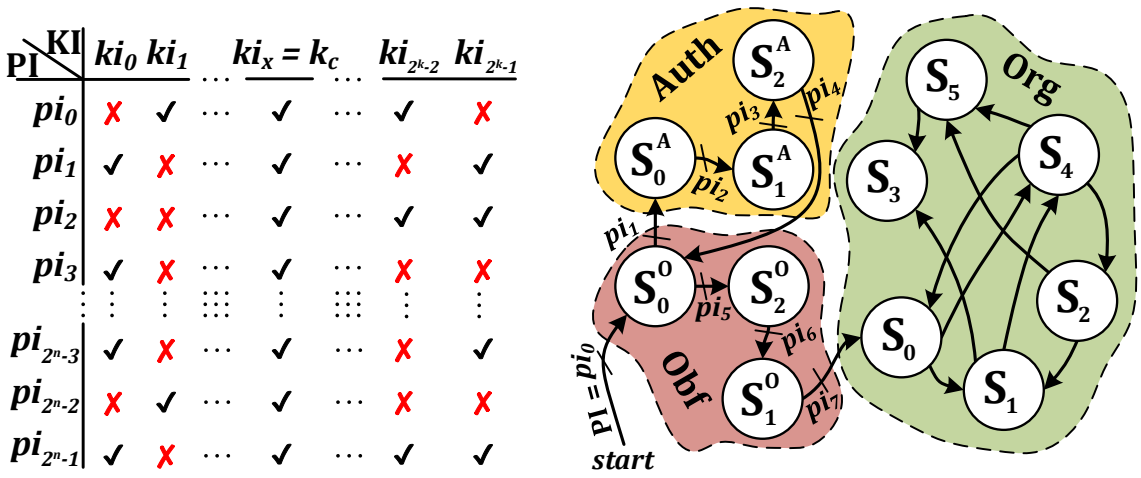
and benchmark the capabilities of SAT solvers when specifically dealing with hardware obfuscation problem. It provides insights on capabilities and limitation of different classes of SAT solvers, helping the researchers in choosing the most able SAT solver for evaluating the effectiveness and hardness of their proposed obfuscation solutions and prevents researchers from generalizing the failing result of a poor choice of SAT solver solution, to all SAT solvers. Then, in chapter 4, we elaborate on the limitation of cyclic SAT attacks and our approach for breaking/preventing these attacks. We introduce our techniques for building an exponential relation between the number of feedbacks and the number of created cycles in a circuit. We also introduce three mechanisms for building a cyclic Boolean function to further increase the complexity of pre-processing in cyclic attacks. In chapter 5, we introduce DFSSD, a novel logic locking solution for sequential and FSM circuits with a restricted (locked) access to the scan chain. This technique is specifically designed to resist against sequential SAT attacks based on bounded model checking. In chapter 6, we introduce RANE framework, an open-source CAD-based toolbox for evaluating the security of logic locking mechanisms that implement a unique interface to use formal verification tools without a need for any translation or simplification. The RANE attack not only performs better compared to the existing de-obfuscation attacks, but it can also receive the library-dependent logic-locked circuits with no limitation in written, elaborated, or synthesized standard HDL, such as `Verilog`. In chapter 7, we present our future research direction to sequential obfuscation and discuss scan chain obfuscation and the new unrolling SAT attack.

Chapter 2: Background on Threat Models and Boolean Satisfiability Attacks

Logic locking is the process of hiding the functionality of a circuit by implementing post-manufacturing means of programmability into the netlist. Logic locking has been widely studied in the literature [5, 10, 15, 16, 22, 30–35], in which, the functionality of a circuit is locked using two major techniques. It could be implemented as (1) a set of dedicated key inputs, driven from a Tamper-Proof Memory (TPM), such that only when the correct key is applied, the circuit resumes its expected (correct) functionality [5, 9, 10, 15, 16, 22, 30–37], or (2) a set (sequence) of input patterns through PIs that requires to be traversed to lead the state of the circuit to the normal (correct) mode [38–40].

2.1 Key-based Versus Key-less Locking

Fig. 2.1 depicts how both key-based and key-less logic locking techniques work. In key-based logic locking shown in Fig. 2.1(a), there exist two sets of inputs, i.e., primary inputs (PI) and key inputs (KI). In key-based logic locking, the key values are stored and initiated in TPM. However, after reverse-engineering, the content of TPM will be wiped out, and the adversary has to evaluate them as extra inputs to the circuit (KI). In key-less logic locking, on the other hand, as depicted in Fig. 2.1(b), new state modes, such as *obfuscation* mode and *authentication* mode, are added in the circuit's FSM required to be traversed using a specific set of PI patterns. For example, by applying patterns as $\text{pi}_1 \rightarrow \text{pi}_7$ to the PI of 2.1(b), the circuit will reach its normal (correct) initial state. In this case, there exists no extra (dedicated) wires/inputs as the KI, which potentially could make the reverse engineering attacks much harder for the adversary.

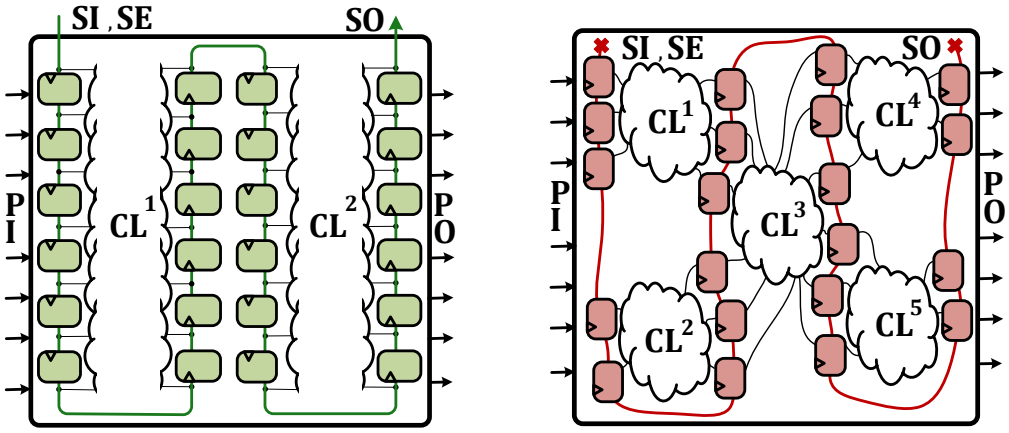


(a) Key-based Example (b) Key-less Example

Figure 2.1: Explicit Key-based vs. Implicit Sequence-based.

2.2 Restricted Versus Open Scan Chain Access

Over the last decade, numerous studies have evaluated and challenged the validity and robustness of existing logic locking techniques against different forms of attacks, especially attacks on key-based logic locking techniques [3]. Depending on the threat models defined for the attacks, these attacks could be categorized into different groups. One central assumption in these threat models is the availability of the DFT infrastructure (i.e., scan chain pins) for the adversaries [41]. When the access to the scan infrastructure is *OPEN*, as shown in Fig. 2.2(a), the adversary would be able to have separate access to inputs/outputs of each combinational logic (CL), separately. By controlling the scan enable (SE), this access could be achieved through the scan in (SI) and the scan out (SO). Assuming that the access to the scan pins is *OPEN*, Boolean satisfiability (SAT)-based attack could break most forms of the logic locking techniques in a matter of minutes [11, 12].



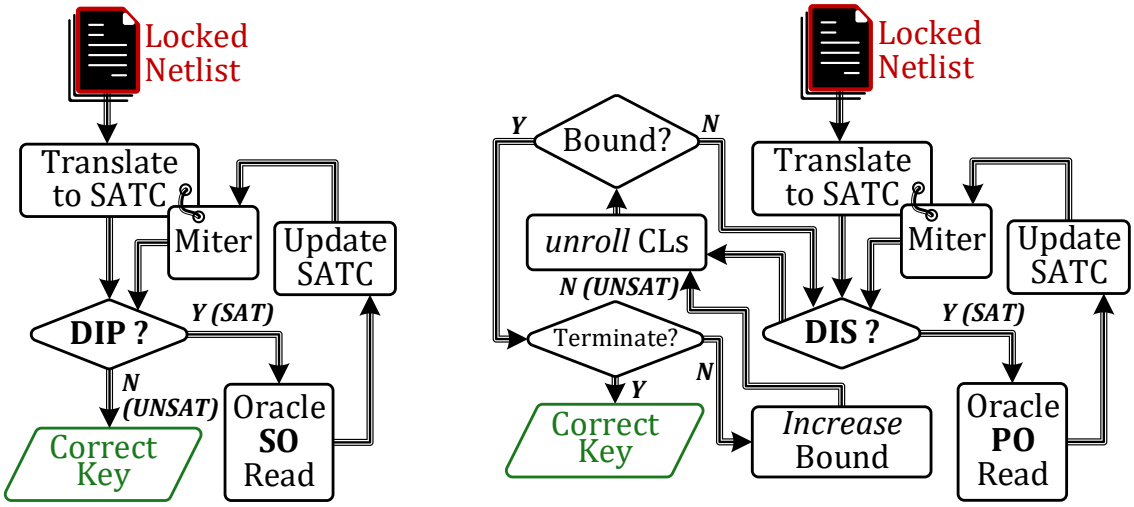
(a) *OPEN* Scan Chain (b) *BLOCKED* Scan Chain

Figure 2.2: Scan availability in de-obfuscation attacks.

2.3 Boolean Satisfiability Attack

In the SAT attack, as illustrated in Fig. 2.3(a), the adversary first transforms each CL of the reverse-engineered circuit to SAT circuit (SATC). Then, by building the miter circuit $(CL(pi, K_1) \oplus CL(pi, K_2))$, the adversary revokes the SAT solver to find pi which distinguish between two keys (different outputs for K_1, K_2), called *discriminating input pattern* (DIP). Then, this DIP will be queried on the oracle (through DFT), and the SI/SO constraint for the CL will be stored back in the SAT solver, and the miter circuit would be solved again in the next iteration. When the miter+constraints problem no longer has a satisfying assignment in one iteration, the list of added constraints is a complete set that uniquely characterizes a correct key. Finding a correct key is then straightforward. Any key that satisfies this set of constraints is correct, and it could be found using a single query to the SAT solver. With access to the scan pins and using this flow, the adversary would target each CL separately and apply the combinational de-obfuscation attack.

Since the validity and strength of most logic locking techniques are undermined by combinational SAT attacks, numerous studies show how the scan access could be limited to avoid the feasibility of such attacks [4, 41–45]. When the scan chain access is *BLOCKED*, as demonstrated in Fig. 2.2(b), the adversary's access is limited to PI/PO, and she cannot



(a) Combinational SAT Attack (b) Sequential SAT Attack

Figure 2.3: Combinational SAT vs. sequential SAT attack.

build the SATC for each CL. However, even while the access to the circuit is limited to PI/PO, the SAT solver could be used to model an attack on the sequential circuits as a whole. The attack procedure on sequential circuits with no scan access is shown in Fig. 2.3(b). Similar to the SAT attack, it has an iterative structure for pruning the search space. However, due to the restricted access to the internal registers, to formulate the internal state, unfolding could be merged with the combinational SAT attack to extend it for sequential circuits [46–48]. Hence, rather than finding a DIP in each iteration, it finds a sequence of inputs denoted as *discriminating input sequence* (DIS) that can generate two different outputs for two different keys. After finding each DIS, the miter+constraint will be updated with a new constraint to ensure that the next onset of keys produces the same output for previously found DIS. This process continues until no further DIS is found within the given boundary.

Both combinational and sequential SAT-based de-obfuscation attacks could be applied to key-based logic locking techniques, and they are not directly applicable to key-less logic locking. The key-less logic locking was first introduced in HARPOON [38]. HARPOON is a sequential logic locking technique that modifies the FSM of a circuit. In such key-less

techniques, one or few extra sets (modes) of state, such as obfuscation mode or authentication mode as shown in Fig. 2.1(b), is merged with the original FSM of the circuit. In such an FSM locking solution, a specific *unlocking* sequence is required (and applied in multiple cycles) to drive the FSM from a locked state to reach the active FSM's original initial state [38, 49]. The target of an adversarial attack against such FSM locking solutions is to find a sequence of input patterns in the result of which the initial state of the original FSM is reached and IP is activated.

2.3.1 SAT Solvers and Circuit SAT Problem

The Boolean satisfiability for a Boolean function $F(X_1, X_2, ..., X_n)$ is the problem of determining if there exists an assignment $\hat{X} = (x_1, x_2, ..., x_n)|x \in \{0, 1\}^n$, such that $F(\hat{X}) = 1$. Related to Boolean satisfiability, [50] proposed the Davis-Putnam (DP) algorithm, a deductive approach based on iterative existential quantification by resolution, which justifies if a satisfiable solution exists. [51] suggested a backtracking search mechanism, known as Davis-Logemann-Loveland (DLL), that exhaustively searches for a satisfying assignment. DLL is a depth-first search algorithm, with the added capability to derive the implication graph of a Boolean logic, and upon detecting an unjustifiable scenario, backtracks to the previous node in the search tree to explore other branches. [52] improved the DLL by means of Conflict Driven Clause Learning (CDCL), capable of pruning the search space in result of two fundamental changes to the DLL's DFS search mechanism: (1) using the concept of clause learning, where a conflicting scenario, reached after visiting a branch of search tree, is converted into a conflict clause and is added to the list of input clauses and by changing the backtracking mechanism to a non-chronological mechanism to speed up the discovery of new conflict clauses. (2) adding the concept of restart to allow abandoning the current search tree and to reconstruct a new one if the rate of discovery of conflict clauses is low. The modern SAT solvers improved the efficiency of CDCL SAT solvers by using techniques such as Boolean constraint propagation, two literal watching, modified decision heuristics, and by implementing locality based search [53][54][55].

2.3.2 Preparing Obfuscated Netlists for SAT Attack

A SAT solver takes a Boolean function in Conjunctive Normal Form (CNF) as input and finds a valid assignment for input variables to satisfy the function. To attack an obfuscated netlist using a SAT solver, a working copy of the chip and its obfuscated netlist is required. The adversary can acquire the working chip after it is unlocked by the manufacturer and shipped to the market and could gain access to the obfuscated netlist by means of RE. In case of supply chain adversary, the obfuscated netlist is readily available to the attacker. Then, the obfuscated netlist should be transformed into a circuit SAT problem. This process is explained next.

Converting Obfuscated Gates to Key-Programmable Gates

Let us refer to the functional black-box copy of the obfuscated circuit as C_F. The C_F is used to find the correct output for any given input. When using K keys, random assignment of key could create at most 2^K instances of a circuit. Similar argument applies to camouflaged cells, where each of K camouflaged gates could assume one of the M different possibilities (for simplicity, let us consider $M = 2$). Let us denote obfuscation scheme obtained by means of using K keys or obfuscated gates by K-obfuscation. A circuit C with N_X inputs that is subjected to K-camouflaging could be represented with an equivalent C_K circuit with $N_X + K$ inputs. Let us denote the circuit C with input X and output Y by $C(X, Y)$ and its K-obfuscated netlist by $C(X, K, Y)$. If the correct set of keys $\hat{K} = (k_0, k_1, ..., k_{K-1})$ is applied to the obfuscated circuit, for every input the obfuscated circuit reduces to the original circuit $C(X, \hat{K}, Y_K) \triangleq C(X, Y)$.

For a SAT attack the key signals in $C(X, K, Y)$ should be available as input. Hence, obfuscation cells should be represented as *Key-Programmable Gate* (KPG), where insertion of the correct key converts them to the correct gate. The cells used for obfuscation could be divided into two categories: (1) key-controlled gates [30][36] in which the key is an input signal (e.g. XOR, MUX based obfuscation). (2) keyless-gates [10][8] where functionality is

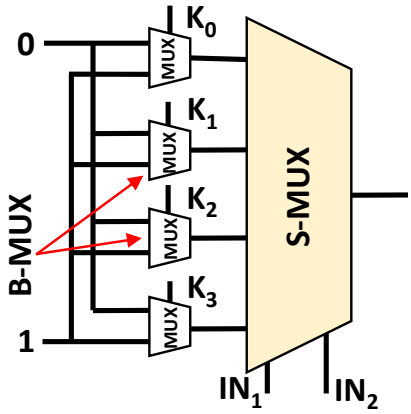


Figure 2.4: Converting a LUT to a KPG.

hidden in the ambiguous structure or by use of internal memory elements (e.g. camouflaged gates and LUTs). When using key-controlled gates, the key is stored in an internal memory or a burned fuse. Hence, in a reverse-engineered netlist the key inputs could be identified by tracking their connectivity to memory/fuse elements. To prepare the $C(X, K, Y)$ netlist, the memory/fuse element is removed and key inputs are connected to input port(s).

When using keyless-gates, the gate has to be transformed to a key-programmable gate before invoking a SAT attack. For a L-input LUT, the number of functional possibilities is 2^{2^L}. To build a KPG for a LUT, the circuit illustrated in Fig. 2.4 is deployed. The inputs to the LUT are connected to the select lines of the S-MUX and keys are the select lines of B-MUXes. Then, each key is connected to an input port adding 2^L keys to the $C(X, K, Y)$.

A camouflaged cell relies on hiding the gate functionality by keeping the structure of several gates similar. Even in the best camouflaging cells, the number of gate possibilities is limited and it could be treated similarly to programmable cells, where the camouflaged cell is replaced by a MUX and each of the gate possibilities is fed to a different input of the MUX, while using the select lines of the MUX as key inputs that are routed to the input pins of the $C(X, K, Y)$.

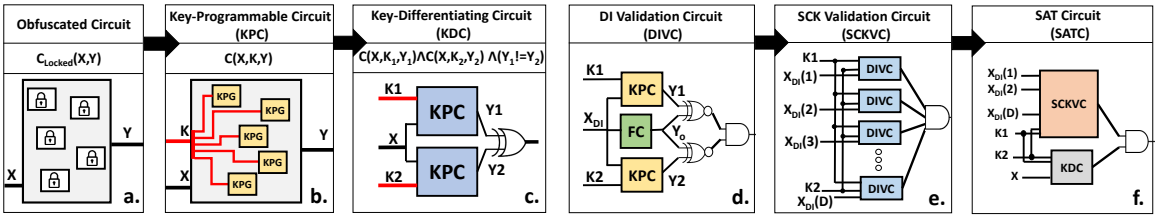


Figure 2.5: (a) Transforming an obfuscated circuit to (b) Key-Programmable Circuit and (c) Key-Differentiating Circuit. (d) DIVC circuit for validating that two input keys produce the correct output with respect to a previously discovered DI. (e) SCKVC circuit for validating that both input keys are in SCK set and produce the correct output for all previously discovered DIs. (f) SATC circuit for finding a new DI.

Converting an Obfuscated Circuit Into a SAT Problem

Before invoking the SAT solver, every key input combination is considered as a candidate key. Let's denote the *Set of Candidate Keys* by SCK. If we can find an input x_d, and two distinct key values K_1 and K_2 in SCK such that $C(x_d, K_1, Y_1) \neq C(x_d, K_2, Y_2)$, the input x_d is denoted as a *Discriminating Input* (DI) [11]. This is because the selected input has the ability to prune the SCK and find at least one incorrect key that is removable from SCK. In addition each time a new DI is found, the SCK search space for function F_{DI} should be updated. This could be achieved by forcing the F_{DI} to check each pair of new keys K_1 and K_2 against all previously founds DIs. A Complete-DI-set is a set of DI inputs that reduces the SCK to the *Set of Valid Keys* (SVK). SCK reduces to SVK when we no longer can find a DI using the updated F_{DI}. At this point if a key is valid across the Complete-DI-Set, it is the correct key for all other inputs [11].

In this chapter, as suggested in Fig. 2.5.b, a reverse-engineered netlist, where all obfuscated cells are replaced with KPG cells, is denoted by *Key-Programmable Circuit* (KPC). To build the F_{DI}, two copies of the KPC are used, their non-key inputs (X) are tied together, and their outputs are XORed. This circuit produces logic 1 when the output of two instantiated KPCs for the same input X but different keys $K1$ and K_2 are different. This circuit, as suggested in Fig. 2.5.c is denoted as *Key-Differentiating Circuit* (KDC).

The candidate keys in the SCK are capable of producing the correct output for all DIs

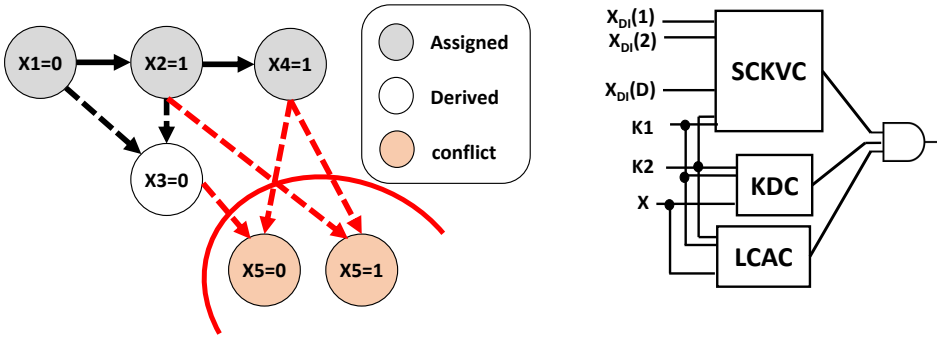


Figure 2.6: (left) Process of arriving at a conflict clause. (right) the SATC circuit augmented with a block to check for non-occurrence of learned conflict clauses.

that have previously been discovered and tested on the KPC circuit. In order to test the keys for one DI, the circuit in Fig. 2.5.d is instantiated. In this figure, FC is the working copy of the chip, and its output is used for testing the correctness of both KPCs for a given DI and two key values. This circuit is denoted as *DI-Validation Circuit* (DIVC). To test the keys for all DIs, as illustrated in Fig. 2.5.e, the DIVC circuit is duplicated D times, with D being the number of current DIs tested, and the output of all DIVC circuits ANDed together. The resulting circuit is a validation circuit for SCK set denoted as SCKVC.

If two keys K_1 and K_2 produce the correct output for all previously tested DIs (SCKVC evaluates to true), but produce different results for a new input X_{test}, then X_{test} is a DI that further prunes the SCK. This, as illustrated in Fig. 2.5.f, could be tested by using an AND gate at the output of SCKVC and KDC circuits. The resulting circuit forms a SAT solvable circuit denoted by SATC. When SATC evaluates to true, the KDC has tested a pair of keys K_1 and K_2 that produce two different results for an input X_{test}, and SCKVC circuit has confirmed that both K_1 and K_2 belong to SCK set. Hence, the input X_{test} is yet another DI. Each time a new DI is found, the SCKVC should be updated by adding yet another DIVC circuit for testing the newly discovered DI. This process is continued until SAT solver no longer finds a solution to the final SAT circuit. In this case, any key remaining in the SCK set is a correct key for the circuit. On the SAT solver side, every time the SAT solver is executed, it learns a new set of conflict clauses. It is essential to store

Algorithm 1 SAT Attack on Obfuscated Circuits

1: $KDC = C(X, K_1, Y_1) \wedge C(X, K_2, Y_2) \wedge (Y_1 \neq Y_2)$;
2: $SCKVC = 1$;
3: $SATC = KDC \wedge SCKVC$
4: $LCAC = 1$
5: **while** $((X_{DI}, K_1, K_2, CC) \leftarrow SAT_F(SATC) = T)$ **do**
6: $\quad Y_f \leftarrow C_F(X_{DI})$;
7: $\quad DIVC = C(X_{DI}, K_1, Y_f) \wedge C(X_{DI}, K_2, Y_f)$;
8: $\quad SCKVC = SCKVC \wedge DIVC$;
9: $\quad LCAC = LCAC \wedge CC$
10: $\quad SATC = KDC \wedge SCKVC \wedge LCAC$;
11: $KeyGenCircuit = SCKVC \wedge (K_1 = K_2)$
12: $Key \leftarrow SAT_F(KeyGenCircuit)$

the learned clauses and use them in the next invocation of the SAT solver to prevent SAT solver from re-learning these clauses. Hence, as illustrated in Fig. 2.6, a *Learned-Clause Avoidance Circuit* (LCAC) is added to the SATC to check for the occurrence of learned conflict clauses.

2.3.3 SAT Attack Algorithm

The SAT attack, as illustrated in Algorithm 1, follows the SATC construction process explained in section 2.3.2. In the first iteration, the SCKVC circuit does not contain any logic, since there is no previously tested DI. Hence, it is set to 1 (true). The KDC circuit is simply built based on its definition by using the equation in Fig. 2.5.c. The SATC circuit is constructed by using an ANDing the KDC and SCKVC circuits. SAT_F function is a call to SAT solver. Considering the to-be-assigned variables in SATC circuit are X, K_1 and K_2, the SAT solvers return an assignment to these variables and a list of *conflict clauses* (CC) learned during SAT execution. SAT_F return UNSAT if no such assignment exists. The while loop is controlled by the return status of the SAT solver. In every pass through the while loop, a new DI is found. Hence, the SATC circuit should be modified (lines 7-10). The parts of SATC circuit that is updated are the SCKVC and LCAC. After finding each DI, an additional DIVC is added to SCKVC to validate the keys generated in the next invocation of SAT solver with respect to the newly found DI. In addition, the newly learned CCs are added to LCAC. The C_F is a call to the functional circuit that returns the correct

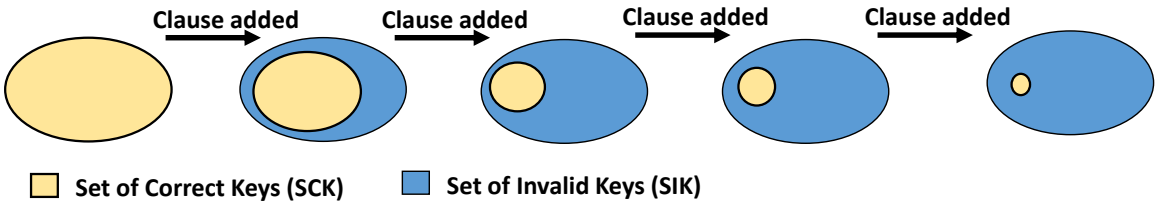


Figure 2.7: SCK set reduces in each pass through the while loop in Algorithm 1 as a new DI is discovered and is added to SATC circuit.

output for each newly found DI. Finally, the SATC circuit is formulated at line 10 for the next invocation of SAT solver.

The while loop is executed until no other DI is found. At this point, any key in the SCK set is a correct key. To obtain a correct key, the DIVC circuit is modified to take a single key denoted as KeyGenCircuit. Hence, KeyGenCircuit has input K, and its output is valid if K satisfy all previous constraints imposed by previously found DIs. A simple call to a SAT solver at this point returns a correct key assignment. If the SAT solver does not return a valid key, it means the obfuscation, locking, or camouflaging technique is invalid. Note that the SAT attack in each iteration, as explained in Algorithm 1 and illustrated in Fig. 2.7, reduces the SCK by constraining the SATC with new clauses added to the SCKVC and LCAC. But it does not explicitly check to find the keys in SCK.

Chapter 3: Benchmarking the Capabilities and Limitations of SAT Solvers in Defeating Obfuscation Schemes

Today, many off-the-shelf SAT solvers with various capabilities are freely and openly available, and each year many new and more capable solutions are being developed. Some of the most efficient and most powerful SAT solvers are the winners of the International SAT competition [56], where solvers participating in the competition are required to test the performance of the proposed SAT solvers on a large number of SAT problems in various categories. Different SAT solutions have offered widely varying performance dealing with various SAT problems, illustrating that the choice of a SAT solver and its underlying features could have a significant impact on the solver's success and also on the time it takes to solve a specific problem. In addition, different SAT solvers require various amounts of memory resources, and assuming powerful SAT solvers may perform extremely poorly or fail to find the solution for larger benchmarks, even if they may outperform other SAT solvers for solving a large number of small SAT problems. In this chapter, we investigate the limitations and capabilities of different classes of SAT solvers when specifically dealing with the problem of circuit obfuscation.

3.1 Evaluated SAT Solvers and Obfuscation Techniques

The selected SAT solvers and obfuscation schemes are described next.

3.1.1 Studied SAT Solvers

MiniSat

This solver [53] is developed as a modifiable SAT solver with conflict-driven backtracking, watched literals and dynamic variable ordering. Most of the later SAT solvers are a modified

version of this solver. It could be used as a baseline for evaluating the effectiveness of added features in other solvers for obfuscated circuit benchmarks.

Glucose

Glucose [54] is an extension of MiniSat code with a special focus on removing useless clauses as soon as possible and a new restart scheme. It uses the idea of Literal Block Distance (LBD) to estimate the quality of learned clauses. Other SAT solvers incorporated its restart policies. This solver adapts itself according to four predefined outlier benchmark characteristics, but in none of the tested benchmarks, default strategy has changed.

Lingeling

Lingeling [55] is based on the idea of interleaving search and pre-processing. It uses various techniques to reduce the search space. Binary and ternary clauses are stored separately from large clauses. Large clauses are kept using literal stacks and references to them are simplified from pointers to stack position. Binary and ternary clauses are kept in occurrence lists. Occurrence lists are defined using stacks and are referenced by stack position. It also uses a modified version of restart mechanism used in Glucose. Number of variables and clauses are also monitored during execution and the number of learned clauses is controlled using their variance.

Maple (MiniSat/Glucose)

Maple SAT variants [57] use a new branching heuristic in place of Variable State Independent Decaying Sum (VSIDS) called Learning Branching Heuristic (LRB). Two variants of MapleSat are MapleMiniSat and MapleGlucose, respectively based on MiniSat and Glucose. MapleGlucose uses LRB for 2500 seconds of the execution, and then switches to VSIDS. In MapleMiniSat, VSIDS is replaced with LRB.

CryptoMiniSat

CMSAT or CryptoMiniSat [58] is a SAT solver that compiled from SatELite, PrecoSat, Glucose and MiniSat features. It has special mechanisms for XOR clause handling and separates watch lists for binary clauses. It can detect distinct sub-problems in clause list and try to solve them with sub-solvers.

3.1.2 Studied Obfuscation Techniques

A random obfuscation scheme was proposed by Roy et al. in [30]. In this scheme, which is one of the earliest work on obfuscation, the XOR/XNOR gates are randomly inserted in the netlist. We refer to this obfuscation as rnd. A major weakness of this scheme was the ability of an attacker to sensitize the circuit, by application of carefully selected inputs, and to propagate the obfuscation keys to the primary outputs of the circuit. Rajendran et al. [31] proposed a more sophisticated obfuscation mechanism to avoid such sensitization attacks by preventing insertion of isolated and mutable key-gates. We refer to this scheme as dac12. An important metric in logic obfuscation is increasing the output corruption when a wrong key is used. Rajendran et al. [59] proposed an obfuscation method that uses fault propagation analysis to maximize Hamming distance between correct and incorrect outputs when attacker applies a wrong key. They proposed two variants of their obfuscation technique based on using XOR and MUX gates. We refer to these obfuscation schemes as toc13xor and toc13mux. Wires with low controllability are susceptible to Trojan insertion. To obfuscate the degree of controllability of wires in a netlist, in [60] Dupuis et al. tried to minimize the wires with low controllability. This was achieved by inserting AND/OR gates attempting to balance the signal probabilities. We refer to this obfuscation method as iolts14.

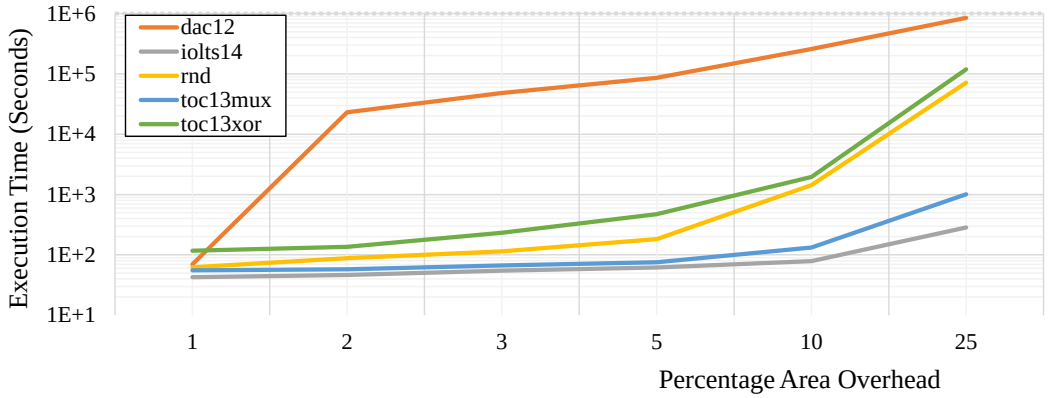


Figure 3.1: The difficulty of investigated obfuscation solutions across all SAT solvers. The execution time reported is the sum of the execution times of all SAT solvers for finding the key at each reported obfuscation overhead percentage point.

3.2 Experimental Results

3.2.1 Benchmarking Platform

For benchmarking of selected SAT solvers, we used a farm of 20 Dell Latitude-7010 desktops equipped with Intel Core-i5 processor and 8GB of RAM. For fair comparison, and to reduce the impact of of the operating system background processes, we dedicated one machine to each SAT solver at a time, and installed Ubuntu Server 16.04.3 LTS operating system in shell mode. We used the ISCAS-85 and MCNC benchmark suites in our study and obfuscated each benchmark with $\{1\%, 2\%, 3\%, 5\%, 10\%$ & $25\%\}$ area overhead. To account for run-to-run variation in performance, we ran the SAT solver 15 times for each obfuscated benchmark.

3.2.2 Results

Fig. 3.1 illustrates the difficulty of defeating each obfuscation method across all SAT solvers. To generate this graph, the execution times for finding the keys to all obfuscated benchmarks are added together at each obfuscation overhead percentage point. The figure illustrates that the complexity of benchmarks obfuscated by dac12 is considerably higher than that for all other investigated obfuscation schemes. We should also note that the time needed

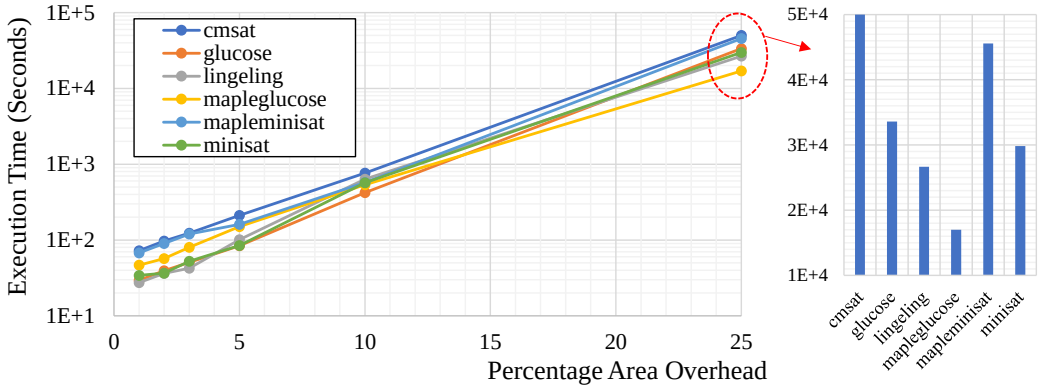


Figure 3.2: Total execution time of each SAT solver for finding the correct key for all benchmarks and all obfuscation schemes except dac12 as a measure of solver strength for low-to-mid complexity problems.

for obfuscating a design using the dac12 methodology is considerably longer than the time required by earlier obfuscation methods. The simulation results confirm that increasing the controllability of internal signals, as done in iolts14, or increasing the output corruption, as implemented in toc13, significantly reduces the strength of obfuscation scheme against SAT attacks. Hence, obfuscation schemes that produce the lowest possible output corruption, or reduce the controllability of internal signals pose a harder problem for SAT solvers. However, please note that the aforementioned options for making the obfuscation problem harder for SAT solvers is completely against the reasons why these obfuscation schemes were introduced in the first place (high corruption for higher protection, and high controllability for Trojan prevention).

For the rest of this chapter, we separate the discussion of dac12 and the other investigated benchmarks as, depending on the percentage area overhead used for obfuscation, they represent two groups of low-to-mid and mid-to-high complexity SAT problems. Note that we have not used the SAT-hard obfuscation schemes such as SARLock [15] for two reasons. First, they are prone to a simpler SPS attack for detection and removal of key-forming-cones. Second, to study the effectiveness of SAT solvers, we deliberately chose to work with medium to semi-difficult problems that are still solvable by SAT solvers in a reasonable time, so that the execution time of SAT solvers is a measure of their efficiency. Otherwise,

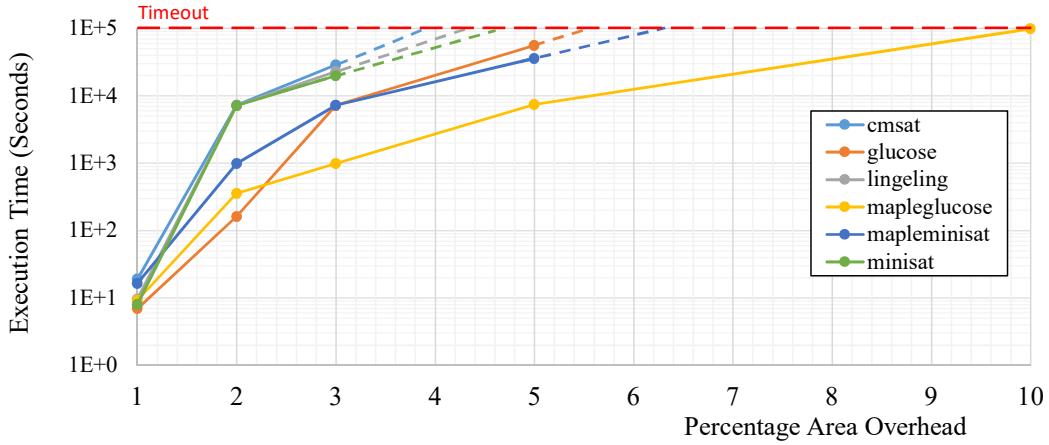


Figure 3.3: The execution time of SAT solvers for finding the correct key for all dac12 obfuscated benchmarks as a measure of solver strength for mid-to-high complexity problems.

if the operation of SAT solvers is reduced to brute-force attacks by working on a non-SAT or extreme SAT-hard problems, the execution time of all SAT solvers will be similar, as they will run until they are timed out, or they exhaustively try all possible inputs, thus reducing the SAT solver to a brute-force depth first search solver.

Fig. 3.2 (left) illustrates the ability of SAT solvers in defeating the obfuscation scheme across all low-complexity obfuscation schemes (all obfuscation methods except dac12). As illustrated in this figure, the relationship between execution time and area overhead is exponential. However, note that the execution time grows at a very different pace for different SAT solvers, leading to a significant difference in runtime at higher obfuscation percentages. This is illustrated in Fig. 3.2 (right), where the runtime of SAT solvers under study, benchmarked at 25%, is plotted. As shown in this figure, MapleGlucose, although not the best SAT solver at smaller percentages, outperforms all other solvers by a considerable margin for high percentages, to the point that its runtime is about 3x smaller than that of CryptoMiniSat. Fig. 3.3 illustrates the ability of investigated SAT solvers to find the key for the netlists obfuscated using dac12. As the obfuscation complexity increases, the runtime of SAT solvers widely varies. In this experiment, a 24-hour limit was imposed on the SAT solvers to break the obfuscated benchmarks. MapleGlucose outperformed all other

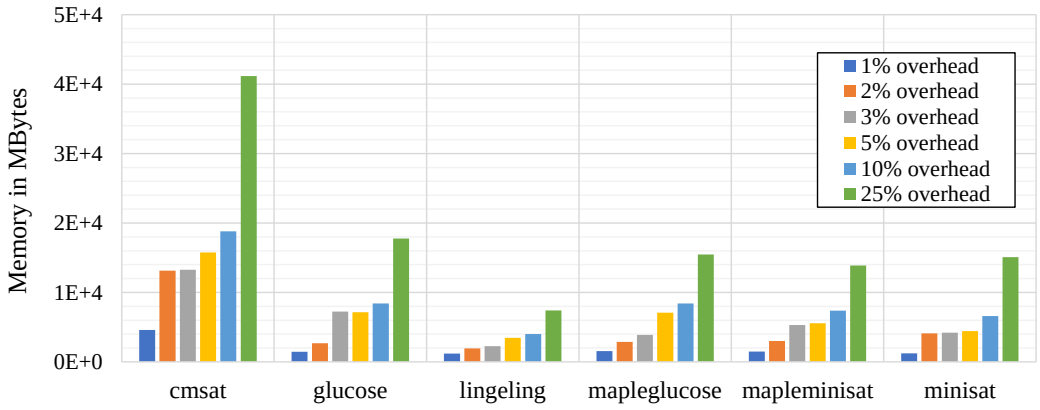


Figure 3.4: Memory usage of each SAT solver across all benchmarks for each obfuscation percentage point as a measure of solver efficiency.

SAT solvers in this experiment.

Fig. 3.4 illustrates the peak of the memory usage for each SAT solver across all benchmarks and at each obfuscation area overhead percentage point. As illustrated in this figure, Lingeling has the lowest memory requirements across all SAT solvers. Hence, it is the most efficient solver in a memory constrained environment, or when the size and percentage of obfuscation considerably increases. As illustrated in Fig. 3.2, Lingeling is also the fastest solver at small obfuscation percentages. At the same time, the CryptoMiniSat is the most memory demanding solver across all obfuscation overheads.

In our study, on average, dac12 produced the hardest obfuscation problems for all investigated SAT solvers. However, when it comes to individual benchmarks, we found a few exceptions to this finding, which prevented us from generalizing the result. For example, as illustrated in Fig. 3.5, the total execution time of all SAT solvers for finding keys to benchmarks C2670 and C3540 (being a part of the ISCAS-85 benchmark suite) is compared. The toc13xor obfuscation in circuit C3540 produces a much harder problem for SAT solvers across different obfuscation overheads when compared to dac12, whereas in C2670 the behavior is reversed. Hence, the netlist characteristic (number of inputs, number of gates, connectivity, topology, number of outputs) plays a significant role in the strength of the applied obfuscation, suggesting the use of hybrid obfuscation methods to defend various

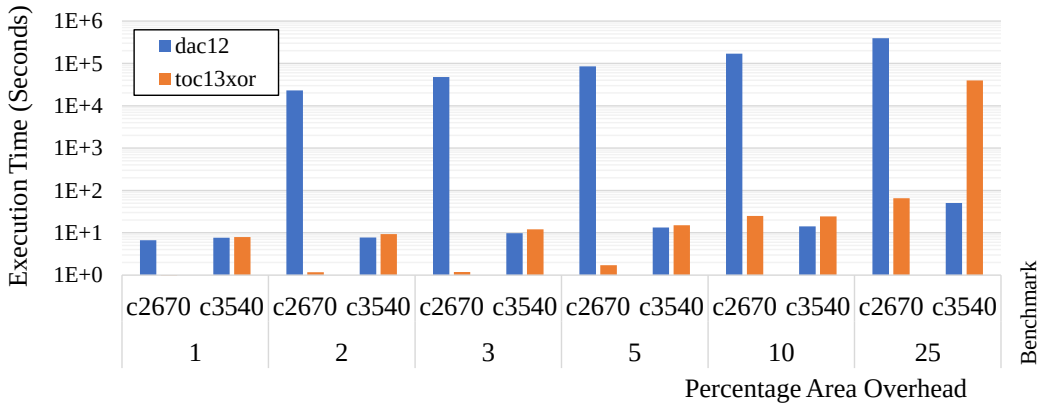


Figure 3.5: Execution time for deobfuscating c2670 and c3540 which are are obfuscated with toc13xor and dac12.

netlists.

3.3 Discussion and Takeaways

When it comes to finding keys for a k-obfuscated circuit, the choice of the best SAT solver depends on the netlist characteristics (number of inputs, number of gates, connectivity, topology, number of outputs) and the level of difficulty of implemented obfuscation methods and the available resources of the system executing the SAT solver.

Across the studied solvers, Lingeling provides acceptable performance for small k-obfuscation problems and has the lowest overall memory demand. Our study reveals that Lingeling is best suited for attacking small to midsize obfuscation problems, considering its shorter execution time for these problems, or for attacking extremely large obfuscated circuits, due to its memory efficiency in cases when other solvers become memory-bounded and thus useless. The memory efficiency of Lingeling is the result of a special implementation of data references for 64-bit machines with a specialized memory allocator and garbage collector.

MapleGlucose, a variation of MapleSat, although not as efficient as Lingeling at small to midsize k-obfuscation problems, still provides the acceptably-good performance. However,

in terms of runtime, it significantly outperforms other solvers for large and more difficult k-obfuscation problems. Our investigation revealed that the hybrid branching heuristic used in MapleGlucose proved to be its most useful feature for reducing the solver's execution time. The secondary feature that was observed to be helpful in reducing the MapleGlucose execution time is using the restart policies in Glucose solver. MapleGlucose, however, may not be suited for extremely large problems, as it may fail to execute in a memory-bounded environment, as its memory demands grow faster than for Lingeling.

Our study revealed that the CryptoMiniSat has the worst performance for k-obfuscation, both in terms of execution time, and memory efficiency. CryptoMiniSat incorporates many interesting features and has proven to be powerful, especially for problems that could be partitioned and solved by separate solvers, but the added features do not help with the efficiency of the solver to deal with k-obfuscation problems.

We experimentally observed that, although different SAT solvers' execution times for a given k-obfuscation problems widely vary, their runtime tracks the obfuscation problems' difficulty. Meaning, if a problem is made more challenging for one solver, it becomes more challenging for all solvers. However, such relationship is not linear. This was especially observed in dealing with the dac12 obfuscation method. Meaning, if a k-obfuscation problem is hardened and SAT solver's execution time is doubled, the problem may cause a much higher or much lower increase in the execution time for another solver. Hence, the results of one solver for a given k-obfuscation cannot be generalized across all SAT solvers.

Across various k-obfuscation methods studied in this chapter, dac12 proved to be generally the most difficult. The learned conflict clauses for a dac12 k-obfuscated circuit are usually less constraining as they rely on a larger number of literals. This provides us with a hint to design harder obfuscation problems by exploiting the SAT solver's clause learning behavior and enforcing mechanisms to increase the number of literals in the learned conflict clauses. Such a defense not only reduces the solver's ability to quickly prune the search space but also increases the memory requirements of the solver for keeping longer clauses. This leads to a faster increase in the size of less effective learned clauses and could degrade

the solver in two different ways: (1) The solver memory requirement is pushed towards the system memory bound, (2) the solver's ability to shrink the size of learned clauses based on identification of shorter and more effective (more pruning) clauses is reduced.

3.4 Conclusion

Our investigation revealed that the Glucose and Lingeling solvers are best suited for small to midsize k-obfuscation problems, while MapleGlucose provides the best execution time for large k-obfuscation problems. When dealing with extremely large k-obfuscation problems, Lingeling again becomes the best choice due to its efficient and less memory demanding database implementation. In terms of testing the hardness of k-obfuscation methods, especially for mid-to-hard size problems, we observed that the increase in the k-obfuscation difficulty affects the runtime of each solver quite differently. Hence, although the increase in difficulty could be verified by one SAT solver, a pace of the increase in difficulty is dependent on the choice of a SAT solver and the results from one solver cannot be generalized. Finally, from a defender's perspective, the results of this benchmarking study suggest that targeting the clause-learning process by means of k-obfuscation, to increase the size of each learned conflict clause, directly affects the effectiveness of SAT solvers in pruning the search space and is a possible promising area for further investigation.

Chapter 4: SAT-hard Cyclic Logic Obfuscation

Cyclic obfuscation [32] is an approach that was considered as a defense mechanism against SAT attacks. However, this technique was later broken by CycSAT attack [61]. CycSAT added a pre-processing step to the original SAT attack for detection and avoidance of cycles in the netlist before deploying an SAT attack. In this chapter, we illustrate that the pre-processing step of CycSAT attack has to process a cycle avoidance condition for every cycle in the netlist, otherwise, the subsequent SAT attack could get stuck in an infinite loop or returns UNSAT. Hence, the runtime of the pre-processing step is linearly related to the number of cycles in a netlist. Besides, we illustrate that the generation of a cycle avoidance clause for a netlist of cyclic Boolean nature is far more time consuming than an acyclic Boolean logic.

From this observation, we first propose several mechanisms for cyclification of a non-cyclic Boolean netlist. Then, we propose two design techniques by which a linear increase in the number of inserted feedbacks in a netlist would exponentially increase the number of generated cycles. Since a successful SAT attack on a cyclic circuit requires the generation of a per-cycle avoidance clause and considering that our proposed techniques make the time it takes to generate such avoidance clauses an exponential function of the number of inserted feedbacks, CycSAT attack faces exponential runtime at its processing step. Hence, when deploying CycSAT, the complexity of the pre-processing of the resulting cyclic netlist goes beyond a reasonable time limit. On the other hand, skipping the prepossessing result in an unsuccessful SAT attack. Hence, cyclic obfuscation, when constructed using the methodology proposed in this chapter, proves to be a strong defense against the SAT and CycSAT attack.

4.1 Previous Cyclic Methods

A method that could render SAT solvers ineffective is to invalidate the acyclic nature of netlist by using cyclic logic obfuscation. Cyclic logic obfuscation was first proposed in [32] whereby introducing feedbacks in the netlist, the netlist is no longer a Directed Acyclic Graph (DAG). In their approach, each intentionally created cycle had more than one way to be opened, making such cycle irreducible by structural analysis, claiming that the existence of such a cycle breaks the original SAT attack in [11, 12].

Attacks previously proposed for breaking logic locking solutions are not effective on cyclically obfuscated circuits. The brute-force attack on obfuscated circuits (even those that are not SAT-hard) will face exponential difficulty. The sensitization attack would not work on cyclic circuits since key values control the multiplexers' select line and the select values can not be sensitized to output pins. The pure SAT attack does not work on cyclic circuits as cycles could either trap the SAT solver or make it exit with an incorrect key, a problem that also occurs in approximate SAT attacks (i.e., AppSAT); the approximate attacks address the issue of separating the keys between SAT-hard and conventional obfuscation. Considering that cyclic circuits trap the SAT solver, this group of attack is also would not work. Removal and SPS attacks are aimed at detecting and removing point functions which are used as a means of building SAT hard solutions in the DAG-based network. Considering that the cyclic obfuscation does not use a point function, SPS and removal attacks are not applicable.

Cyclic obfuscation was later broken with introduction of cyclic (cycle-aware) attacks in [61–63]. CycSAT was the first cyclic attack, details of which are shortly discussed. Later, Chen [62] introduced an enhanced SAT-attack that considers structural cycles. From a functional standpoint, this attack acts similar to the structural attack in CycSAT.

In CycSAT attack, before invoking the SAT solver, the netlist is checked for key conditions that may result in the creation of cycles. These conditions are translated to a set of cycle avoidance clauses and are added to the list of clauses that represent the circuit SAT problem. The Algorithm 2 illustrates the flow of utilizing the cycle avoidance-clauses in

Algorithm 2 CycSAT Attack on Cyclic Obfuscated Circuits

1: Find a set of feedback signals $(w_0, w_1, ...w_m)$;
2: Compute "no structural path" formulas $F(w_0, w_0')$, ..., $F(w_m, w_m')$;
3: $NC(K) = \wedge_{i=0}^{m} F(w_i, w_i')$
4: $C(X, K, Y) = C(X, K, Y) \wedge NC(K)$
5: $SAT_{circuit} = C(X, K_1, Y_1) \wedge C(X, K_2, Y_2) \wedge (Y_1 \neq Y_2)$;
6: **while** $((X_{DI}, K_1, K_2) \leftarrow SAT_F(SAT_{circuit}) = T)$ **do**
7: $\quad Y_f \leftarrow C_{BlackBox}(X_{DI})$;
8: $\quad DIVC = DIVC \wedge C(X_{DI}, K_1, Y_f) \wedge C(X_{DI}, K_2, Y_f)$;
9: $\quad SAT_{circuit} = SAT_{circuit} \wedge DIVC$;
10: $KeyGenCircuit = DIVC \wedge (K_1 = K_2)$
11: $Key \leftarrow SAT_F(KeyGenCircuit)$

CycSAT.

In this algorithm, $(w_0, w_1, ...w_m)$ is a collection of feedback signals whose break will make the encrypted circuit acyclic and w_i' is a signal that feeds to w_i before the break. The function $F(w_i, j)$ is a function that construct the condition for *having no structural path* between signal w_i to signal j. The $F(w_i, j)$ is computed by starting from a feedback signal w_i and constructs a string of clauses that satisfy the following condition while traversing a cycle:

$$F(w_i, j) = \bigwedge_{l \in NK(j)} F(w_i, l) \vee bk(l, j) \tag{4.1}$$

In this function, the $NK(j)$ are the non-key inputs of signal j, and $bk(l, j)$ is the condition on the key assuring key does not affect j. This function is initiated with condition $F(w_i, w_i) = 0$ and finishes after completing the loop. In this case, the condition for no structural path is tested on all discovered feedback signals in line 3 of the algorithm.

Subsequently, Rezaie et al. proposed two solutions [64, 65] to counter CycSAT attack. In the first solution [64], by adding hard cycles to the original netlist, they create a situation that any traversal of the feedback signals will miss a cycle. Also, for this method, dependent cycles are added to the original circuit such that two nested cycles should be closed to create a working circuit. In the second solution [65], a method is introduced to create cycles that behave non-combinational in unreachable states. However, in the next section,

after providing further detail on these locking mechanisms, we illustrate that these solutions are still vulnerable and by making a simple modification to CycSAT attack, they could be easily broken.

Finding all cycles in a cyclic circuit (a requirement for CycSAT attack) is not an easy task. Recently, Shen et al. introduced a new attack called BeSAT [63]. Authors of this attack argue that "it is impossible to capture all cycles in any graph with any set of feedback signals as done in CycSAT algorithm". To address this problem, BeSAT first adds "no structural path" (CycSAT-I) conditions for a "set of feedback signals". This is similar to the pre-processing step in CycSAT attack. Then, it performs SAT while monitoring the behavior of the attack: during the DIP generation process, due to the missing NC clauses, it is possible that solving the circuit-SAT problem results in repeated DIPs. Under the original SAT attack, this could trap the attack in an infinite loop. In BeSAT, every new DIP is compared with previous DIPs and if it was generated before, the algorithm uses it to determine the stateful key K_s. BeSAT compares the output of the new DIP for the two found keys with the oracle circuit. The output of the stateful key disagrees with the oracle circuit. Then, the found stateful key will be explicitly banned by adding $(K1 \neq K_s \ \wedge \ K2 \neq K_s)$ condition to the circuit-SAT problem. After finding all DIPs and banning all stateful keys, BeSAT begins pruning oscillating keys by employing ternary SAT.

4.2 Analyzing the Weaknesses of Cyclic Obfuscation

In this section, we first show that the nested cycles could not guarantee a secure cyclic obfuscation. Furthermore, we propose a new attack mechanism to break the hard cycles. Then, we investigate the weaknesses of CycSAT attack, according to which we propose a new mechanism for cyclic obfuscation.

4.2.1 Breaking Nested Cycles

An obfuscation method that was previously proposed to counter CycSAT attack is the use of nested cycles [64]. In this method, the original circuit is augmented with a pair of nested

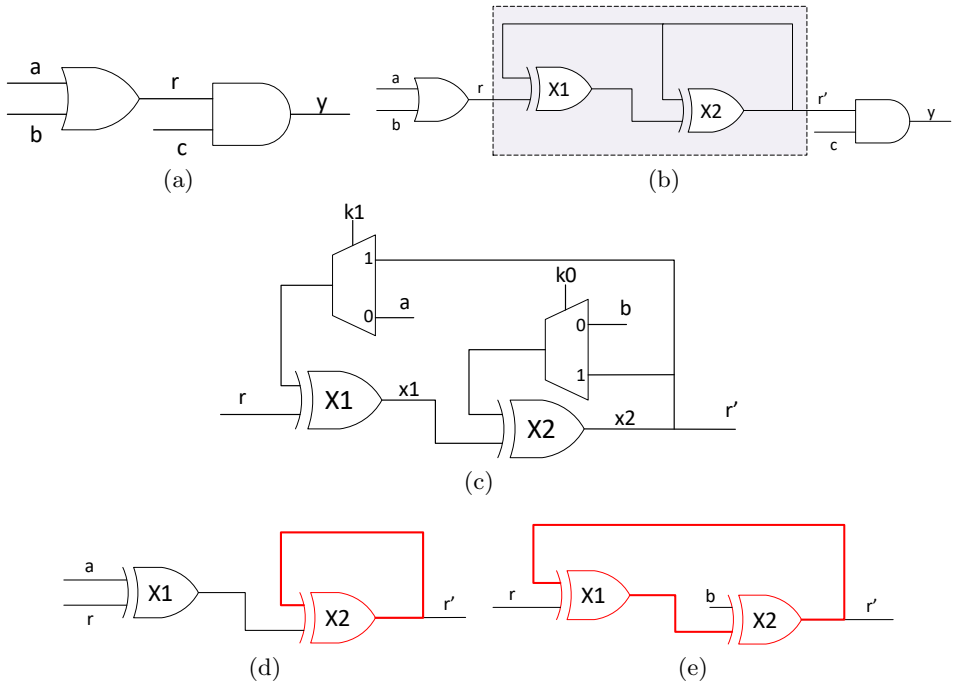


Figure 4.1: Cyclification with dependent cycles: (a) Original circuit, (b) cyclified with an auxiliary-circuit that acts as a buffer, (c) Obfuscated auxiliary circuit, (d) auxiliary circuit with broken outer cycle, (e) auxiliary circuit with broken inner cycle.

cycles such that for correct operation, both cycles should be closed. An example of such a transformation is shown in Fig. 4.1.b for the original circuit in Fig. 4.1.a. After the transformation, the nested cycles are a needed and valid part of the original circuit and attempting to remove one or both cycles will affect the correct functionality of the circuit. A designer may try to obfuscate these cycles using multiplexers as depicted in Fig. 4.1.c.

Direct application of structural CycSAT attack, as claimed by authors in [64] results in breaking each of nested cycles separately, creating an oscillating and un-SAT-isfiable circuit. However, as claimed earlier, we can still deploy a successful attack against this variant of cyclic obfuscation using a simple modification to the pre-processing step of CycSAT attack.

For this purpose, during the pre-processing step, in addition to composing the "no sensitizable path" clauses (as proposed in [61]), we compose and include a new set of clauses that consider "reducibility" as an alternative option to opening the loops. In this picture, the cycle could either be opened (using no sensitizable path clauses) or could be reduced using

Algorithm 3 Generating RC Clauses for Dependent Cycles

```
1: procedure REDUCTION_ATTACK(circuit K)
2:     Find and sort all cycles in K by their length C = (c_0, c_1, ...c_m);
3:     for all c_i in C do
4:         RC(c_i) = φ;
5:     for all c_i in C do
6:         if IS_COMB_CYCLE(c_i) == False then;
7:             RC(c_i) = RC(c_i) ∨ opened(c_i);
8:             while c_j ← next outer cycle do
9:                 sub_circuit ← sub-circuit of closed c_i and c_j;
10:                if IS_COMB_CYCLE(sub_circuit) then
11:                    RC(c_i) = RC(c_i) ∨ (closed(c_i) ∧ closed(c_j));
12:                    RC(c_j) = RC(c_j) ∨ (closed(c_i) ∧ closed(c_j));
13:            RC(K) = RC(K) ∧ RC(c_i);

1: procedure IS_COMB_CYCLE(sub_circuit S)
2:     r, r' ← input and output of auxiliary-circuit;
3:     if SAT(S_opened ∧ (r ≠ r')) then
4:         return False;
5:     else
6:         return True;
```

newly added reducible clauses. The *reducible* clauses are defined for possible dependent cycles that implement specific functions between their inputs and outputs. These clauses will be generated for each cycle by pairing it with matching outer cycles. The process of generating the reducible clauses is captured in the Algorithm 3. The reduction attack procedure, first, sorts all cycles according to their length and then begins processing them from the shortest to the longest cycle. For each cycle, it checks if the cycle is combinational if it is not, it tries to find an outer cycle that makes its behavior combinational. In this algorithm, the $IS\_COMB\_CYCLE()$ validates if a sub-circuit containing a cycle is combinational or not. For this purpose, the function disconnects the cycles by breaking the feedback into two disconnected wire segments r and r'. Then by using a SAT solver, it checks if there are any values for the wires that $r \neq r'$. If such a scenario was not found, it classifies the sub-circuit as a combinational circuit. Otherwise, a non-combinational circuit, according to which the necessary clauses are generated.

This algorithm could be applied to any netlist obfuscated using the auxiliary-circuit such as the one in Fig. 4.1.c. This circuit has two cycles c_1 ={X2} and c_2 ={X1, X2}.

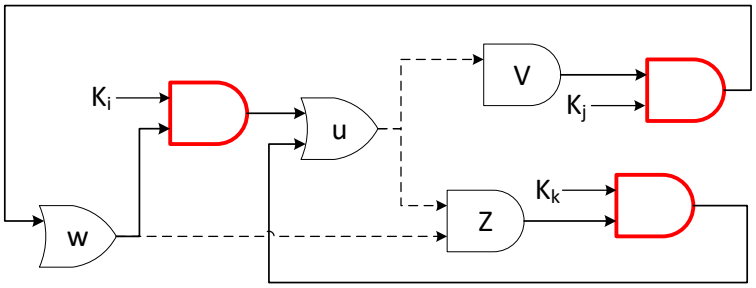


Figure 4.2: An example for a circuit obfuscated with a hard cycle. Added Key-gates are shown in red, and the original wires are shown with dotted lines.

The smallest cycle c_1 is oscillating and oscillates when X1 output is 1 as shown in Fig. 4.1.d. By considering this cycle as closed and pairing it with its only outer cycle c_2 we will have $RC(c_1) = k_0' \vee (k_0 \wedge k_1)$. The outer cycle c_2 as shown in Fig. 4.1.e is also non-combinational and the reducible clauses will be $RC(c_2) = k_1' \vee (k_0 \wedge k_1)$. Thus, by closing both cycles, as shown in Fig. 4.1.b it can be derived that $r' = r \oplus r' \oplus r' = r$ and the circuit does act as a buffer with no oscillation. The reducible clause for this circuit will be $RC(K) = (k_0' \vee (k_0 \wedge k_1)) \wedge (k_1' \vee (k_0 \wedge k_1))$ for closing both cycles or opening both cycles since non of them has combinational behavior independently.

It should be noted that these auxiliary-circuits could be in the form of partially intercepted cycles, where more than one outer cycle is partially intercepted with another outer cycle. We acknowledge that for partially intercepted cycles, our proposed algorithm would not work, and an alternative algorithm that generates the NC condition by considering the partially intercepted combinational cycles is required.

4.2.2 Breaking Hard Cycles

Hard cycles were proposed in [64] to create a situation that any traversal of feedback signals will miss a cycle. An example is shown in Fig. 4.2, where the original circuit consists of gates U, V, W, and Z. In the obfuscated netlist, the gate U is connected to V and Z, and W is connected to Z. By creating a hard cycle, new connections using AND gates have been added. These new wires connect (V, W), (W, U) and (Z, U) and shown with thicker lines.

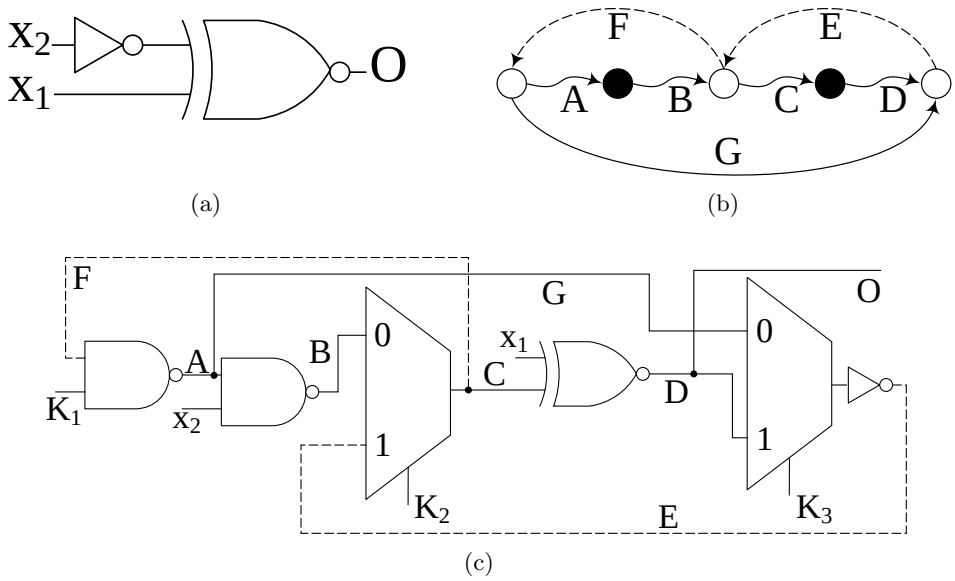


Figure 4.3: (a) Original circuit (b) Flow diagram of the netlist (c) Obfuscated circuit.

Feedback sets for the new circuit are {V, W} and {Z, U}. Application of CycSAT attack on this circuit misses the larger cycle {U, V, W, Z, U}, and the attack fails.

Hard cycles could be easily broken by modifying the mechanisms used for the computation of $F(w_i, j)$. The $F(w_i, j)$ could be computed in two ways: (i) traversing through a cycle starting from w_i until w_i is visited again and ignoring the cycle break conditions imposed by fanins of other nested cycles; or (ii) traversing through one cycle and adding the cycle break conditions imposed by other nested cycle. As shown for the example in Fig. 4.2 and the next example, the first choice results in missing some "no cycle" (NC) conditions, leaving cycles in a design that could break the subsequent SAT attack. By choosing the condition (ii), we show that it is possible to build the *NC* condition by visiting all cycles in the netlist without missing any of the hard cycles. To better illustrate this concept, let us provide a simple example.

For the obfuscated netlist in Fig. 4.3 and a topological sort from gate A, the edge E and F are identified as feedbacks. When following rule (i), and after building the *NC* condition we will have:

1: $F(F, A) = F(F, F) \vee bk(k_1) = k'_1$
2: $F(F, F') = F(F, A) \vee bk(k_2) = k'_1 \vee k_2$
3: $F(E, C) = F(E, E) \vee bk(k_2) = k'_2$
4: $F(E, E') = F(E, C) \vee bk(k_3) = k'_2 \vee k'_3$
5: $NC = F(F, F') \wedge F(E, E') = (k'_1 \vee k_2) \wedge (k'_2 \vee k'_3)$

The problem with this assignment is when $(k_1, k_2, k_3) = (1, 1, 0)$. In this case, the NC condition is satisfied, however, the larger nested cycle $\{E, F, G, E\}$ is not broken. Hence, the NC condition would not resolve the cycles if nested or multi-path scenarios exist. In this case, if the wrong key $(k_1, k_2, k_3) = (1, 1, 0)$ is chosen by SAT solver, it will enter a loop. Depending on whether the cycle is oscillating or stateful, the SAT solver will either be trapped in an infinite loop or will exit UNSAT. Note that this infinite loop happens during the execution of the SAT solver and not during the topological sort used in the original SAT attack proposed in [11, 12].

To avoid the problem imposed by rule (i), we need to follow the rule (ii) where the key contribution of all fanins in all stages are considered. When using rule (ii) for building the NC condition for the same circuit we have:

1: $F(F, A) = F(F, F) \vee bk(k_1) = k'_1$
2: $F(F, F') = (F(F, A) \vee bk(k_2)) \wedge (F(F, E) \vee bk(k_2)) = (k'_1 \vee k_2) \wedge (k'_1 \vee k_3 \vee k'_2)$
3: $F(E, C) = F(E, E) \vee bk(k_2) = k'_2$
4: $F(E, E') = (F(E, C) \vee bk(k_3)) \wedge (F(E, G) \vee bk(k_3)) = (k'_2 \vee k'_3) \wedge (k'_2 \vee k'_1 \vee k_3)$
5: $NC = F(C, C') \wedge F(E, E') = (k'_2 \vee k'_3) \wedge (k'_1 \vee k'_2 \vee k_3) \wedge (k'_1 \vee k_2)$.

By following the rule (ii), the previous assignment of keys $(k_1, k_2, k_3) = (1, 1, 0)$ will no longer be a valid assignment, preventing the SAT solver from being stuck or exiting with a wrong key. However, in this case, *all cycles in the design have to be traversed and conditioned*. As a matter of fact, given the way the NC is formulated in [61], to derive the "no structural path" condition, some of the combinational cycles (such as $\{E, F, G, E\}$ in Fig. 4.3) have been visited more than once. Hence, the number of times the key conditions have to be generated is even larger than the number of cycles in a netlist.

The problem of visiting nested cycles more than once in CycSAT attack could be resolved by a slight modification to CycSAT pre-processing step. In the modified attack, instead

of applying rule (ii) on one-cycle-per feedback, we could apply the rule (i) on all cycles. It is intuitive to see that both approaches produce the same *NC* clauses. For example, in Fig. 4.3 when following condition (i), and traversing cycle $\{E, F, G, E\}$, the condition $(k_1' \vee k_2' \vee k_3)$ is generated. Hence, by ANDing the generated condition to the two clauses generated by applying the rule (i), the *NC* condition of rule (ii) is generated. However, in this case, the combinational cycle $\{E, F, G, E\}$ is only visited once. Even by considering the improvement suggested in CycSAT formulation, it still requires visiting all cycles in a netlist to compose the *NC* clauses. This necessity, as described in the next section, becomes one of the key features which is used in this chapter to break CycSAT attack.

A different method of introducing complexity is by eliminating the DAG nature of the original netlist and by transforming it into a Boolean cyclic function, which could be represented using a Directed Cyclic Graph (DCG), before subjecting it to cyclic obfuscation. If the original netlist is not a DAG, CycSAT pre-processing step has to build the *NC* condition by checking for "*no sensitizable path*" condition [61], instead of "*no structural path*" condition. The no sensitizable path condition from [61] is recited in equation 4.2:

$$F(w_i, j) = \bigwedge_{l \in fanin(j)} F(w_i, l) \vee ns(l, j) \tag{4.2}$$

The "no sensitizable path" condition generates a clause for each multi-input gate in a cycle. As a result, *NC* clauses are much longer and much weaker. Hence, adding even a small number of feedbacks to such circuits (that have valid Boolean cycles) for obfuscation, will significantly increase the size of the circuit-SAT problem, as the "no sensitizable path" condition has to be generated for all cycles. To illustrate the weaker and longer nature of the *NC* clauses, the "no sensitizable path condition" for the circuit in Fig. 4.3 is constructed below:

1: $F(F,A) = F(F,F) \vee ns(F,A) = \ k_1'$
2: $F(F,B) = F(F,A) \vee ns(A,B) = k_1' \vee x_2'$
3: $F(F,F') = (F(F,B) \vee ns(B,F')) \wedge (F(F,E) \vee ns(E,F')) = (k_1' \vee x_2' \vee k_2) \wedge (k_1' \vee k_3 \vee k_2')$
4: $F(E,C) = F(E,E) \vee ns(E,C) = k_2'$
5: $F(E,D) = F(E,C) \vee ns(C,D) = k_2'$
6: $F(E,E') = (F(E,D) \vee ns(D,E')) \wedge (F(E,G) \vee ns(G,E')) = (k_2' \vee k_3') \wedge (k_2' \vee k_1' \vee x_2' \vee k_3)$
7: $NC = F(F,F') \wedge F(E,E') = (k_1' \vee x_2' \vee k_2) \wedge (k_1' \vee k_3 \vee k_2') \wedge (k_2' \vee k_3') \wedge (k_2' \vee k_1' \vee x_2' \vee k_3)$

4.3 SRCLock: The Proposed Cyclic Obfuscation

The issue with the original method of generating cycle avoidance (NC) clauses using CycSAT was shown and discussed in section 4.1 using two simple examples in which traversal of wires based on a single topological sort of gates resulted in a missing cycle. When using the original CycSAT, because of the missing *NC* clauses for such cycles and due to the randomness of assigned key and input values by the SAT solver, the SAT attack can be stuck in an infinite loop or exit with a wrong key. The possibility of facing an oscillating or stateful cycle greatly increases as the number of generated cycles in the design increases to a point that majority of key-space (to be tested by SAT solver) could result in oscillating or stateful cycles, vanishing the chances of a successful attack to unreasonably small probability. On the other hand, attacks such as BeSAT [63] that can track the behavior of the SAT attack at runtime, could detect oscillating or stateful scenarios (due to missing cycles in pre-processing time) and eliminate the incorrect key. However, at runtime, the BeSAT eliminates one key at a time. Hence, it is successful if such key combinations are small. In other words, the BeSAT attack rune time is linearly dependent on the number of such keys. When such key combinations is (exponentially) large (which is the case in our to-be-proposed obfuscation solution), the BeSAT attack's runtime becomes unacceptably large.

$$T_{NC} = \sum_{i=1}^{N} t_{NC} \ \mid \ N = 2^m \tag{4.3}$$

CycSAT pre-processing time is characterized in equation 4.3. As illustrated, the processing time is linearly related to the number of discovered cycles N and the time for composing

the NC condition t_{NC} per cycle. Our approach for breaking CycSAT is to exponentially increase the time needed for composing the NC condition in the pre-processing step of CycSAT beyond acceptable. This is achieved by exponentially increasing the number of cycles N in a design with respect to the number of inserted feedbacks m, and increasing the time required for processing each cycle (t_{NC}) by forcing the pre-processing step to consider the "no sensitizable path" condition instead of "no structural path" condition. Next, we provide two solutions for building an exponential relation between the number of feedbacks and the number of generated cycles, and three solutions for converting an acyclic circuit to a valid cyclic circuit.

4.3.1 Exponentially increasing the number of cycles in a netlist

In order to exponentially increase the number of cycles in a given netlist with respect to the number of inserted feedbacks, we introduce two approaches: (1) building Super Cycles (SC) and, (2) building Logarithmic Feedback Networks (LFN).

Building Super Cycles (SC)

The process of building a SC is illustrated in Fig. 4.4. Before that, let us first define a Micro Cycle (MC). A MC is a cycle created by following the cycle creation conditions adopted from [32], which are recited below:
MC Condition 1: Any created cycle has to be non-reducible,
MC Condition 2: At least $n \geq 2$ edges in each small cycle have to be removable.

A reducible cycle has a single entry point. Hence, the depth-first-search (DFS) traversal of a netlist that only contains reducible cycles is unique. This allows the reducible cycles to be easily opened by removing a unique set of feedback edges which can be found efficiently [32]. By having multiple entries into each MC, the non-reducible condition is satisfied, forcing an adversary to use CycSAT pre-processing step to generate the necessary cycle avoidance clauses before invoking the SAT solver. In graph theory, a strongly connected graph is defined as a graph with at least one path between any two pairs of its vertices.

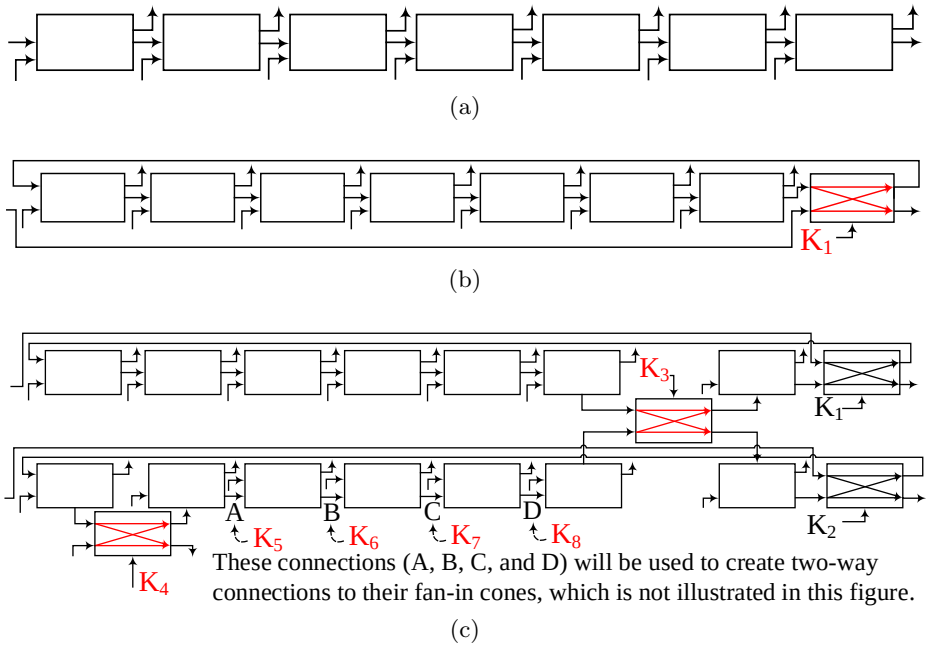


Figure 4.4: Building a Super Cycle from 7 gate MC. (a) A path segment containing 7 gates, (b) building a Micro Cycle, (c) building a SC by strongly connecting multiple MCs.

adopting from this definition, in our solution, a SC is defined as a strongly connected graph of MCs. To substantially increase the number of generated cycles, in the last step of SC generation, the edge density of the generated strongly connected graph is increased, creating additional paths between MCs. The process of building a SC is summarized in Algorithm 4.

Both approaches use a switch to create a direct or cross connection between two points which is shown in Fig. 4.5. Switch inputs will be defined by SC and LFN methods and a

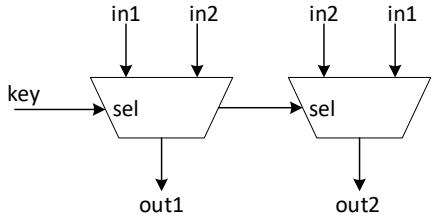


Figure 4.5: Switch structure.

Algorithm 4 Steps for building a Super Cycle

1: Construct MCs in the fanin of smallest possible number of primary outputs.
2: Strongly connect all generated MCs (this, as illustrated in Fig. 4.4.b, is done by creating a two-way connection between each newly created MC, and the existing SC).
3: Select signals in MCs (A, B, C, D in Fig. 4.4.c) that are not used for SC connectivity and provide a two way path from them to unused edges in other MCs or random signals in their fanin cone.

single key input will determine connections between input and output.

In this algorithm, the requirement of generating the MCs in the fanin of the smallest number of primary outputs increases the likelihood of shared and/or connecting edges between created MCs. By having all MCs strongly connected, we create the possibility of larger combinational cycles. And finally, adding the random connections, increase the density of the edges in the strongly connected graph, increasing the number of resulting cycles. In the results section, we illustrate that the number of created cycles, generated from following these steps as described in Algorithm 4, becomes an exponential function of the number of inserted feedbacks.

Lemma. *The lower bound on the number of cycles created when using SC is* 2^m*, when* m *is the number of inserted feedbacks.*

Informal Proof. The proposed SC method adds two paths (from and to paths) to connect each new cycle to the existing SC. This way, the new cycle could be added or not added to any of the previously existing cycles. Hence, the addition of a new cycle at least double the number of potential cycles. Note that the number of connecting edges between the new cycle and the existing cycle could be more than one, resulting in an increase in the number of cycles with a much higher rate. From this discussion, after inserting m feedbacks and connecting them, at least 2^m cycles will be created. ■

Building Logarithmic Feedback Networks (LFN)

In this method, as illustrated in Fig. 4.6.a, several logic paths (preferably from the fanin cone of a single primary output) are selected. Then, by breaking a wire in the midpoint of each logic path, we create two smaller logic segments. The signal entering and the signal

exiting each half segment is marked as its start point (SP) and endpoint (EP) respectively. Then, the SP and EP of multiple such logic path segments are used to build a logarithmic switching network (e.g., Omega, Butterfly, Benes, or Banyan network). When connecting M number of EPs to M number of SPs, for Ms of power of 2, we need $M(1 + log_2(M))$ multiplexers for a logarithmic network. In this case, when the correct key is applied, the switching network is configured correctly, otherwise, invalid connectivity obfuscates the netlist functionality.

Lemma. *The lower bound on the number of cycles when using LFN is* $\sum_{l=1}^{m} \binom{m}{l}(l-1)!$*, when* m *is the number of inserted feedbacks and* l *is the cycle size divided by 2.*

Informal Proof. The proposed LFN is a special case of a complete bipartite graph that contains no odd cycles. Suppose that SE_{ij} indicates a vertex from SP_i to EP_j. Similarly, ES_{ij} indicates a vertex from EP_i to SP_j. For $l = 2$, the cycles are all paths from a SP to its corresponding EP and return path $\{SE_{ii}, ES_{ii}\}$. If we start from SP_i, the second visited node is its EP (EP_i). Since each EP is connected to all SP_s, for intermediate nodes, we have all permutations as alternative possible paths. Cycles with $l = 2$, have no intermediate node. So, there are $\binom{m}{1}0!$ cycles when $l = 2$. For $l = 4$, the cycles are paths like $\{SE_{ii}, ES_{ij}, SE_{jj}, ES_{ji}\}$. There is only one intermediate node in cycles when $l = 4$ resulting in $\binom{m}{2}1!$ cycles. Similarly, for $l = 6$, the cycles are paths like $\{SE_{ii}, ES_{ij}, SE_{jj}, ES_{jk}, SE_{kk}, ES_{ki}\}$. Since, we have two intermediate nodes, j and k, we should consider their permutation as a new cycle, i.e. $\{SE_{ii}, ES_{ik}, SE_{kk}, ES_{kj}, SE_{jj}, ES_{ji}\}$. So, for $l = 6$, we have $\binom{m}{3}2!$. With similar relation, for $l = 8$, we have $\binom{m}{4}3!$ cycles. We can extend this relation to all cycles with different length. The summation of these cycles indicates the number of cycles in our logarithmic network, which is $\sum_{l=1}^{m} \binom{m}{l}(l-1)!$. ■

Note that $\sum_{l=1}^{m} \binom{m}{l}(l-1)!$ is the lower bound of the number of simple and nested cycles created by using the logarithmic network. The number of paths from each SP to each EP could be more than 1, and there are possibilities of having a connection between SPs and EPs of the different paths in the original circuit, increasing the number of cycle possibilities

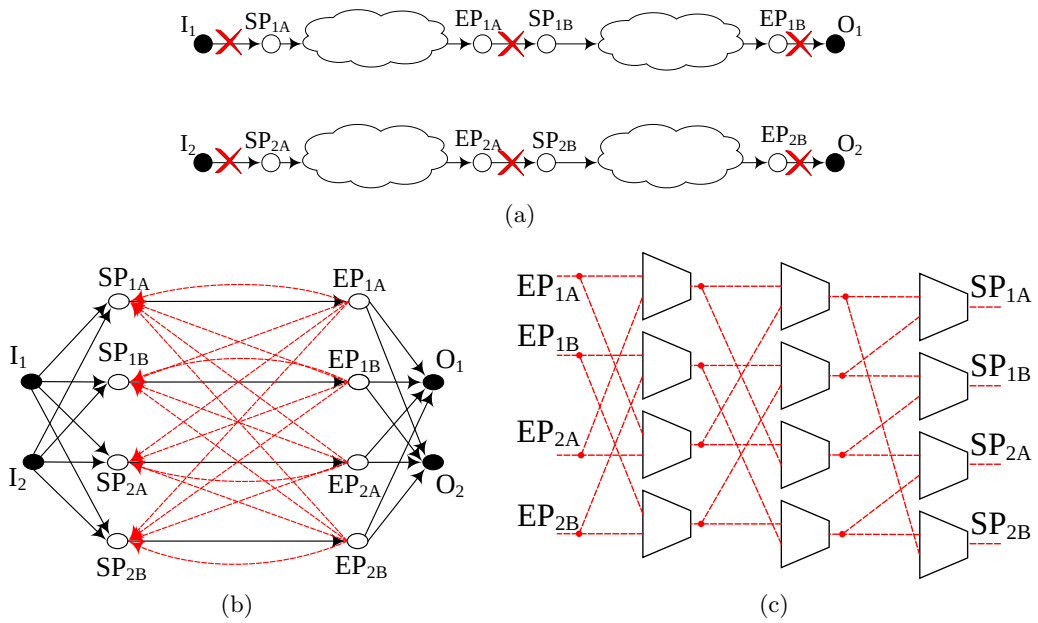


Figure 4.6: Building a logarithmic feedback network in which the number of cycles exponentially increase with the number of feedbacks.

to a far larger number. Based on the lower bound formula, the number of created cycles is $O(\sum_{l=1}^{m} \binom{m}{l}(l-1)!) \leq O(m!) = O(m^m)$. Hence, there exists an exponential relation between the number of inserted feedbacks and the number of resulting cycles in the netlist.

4.3.2 Building Cyclic Boolean Functions

A Boolean function does not need to be acyclic. Furthermore, it is possible to reduce the number of gates in a circuit if a function could be implemented in its acyclic form [66–69]. For example, the work in [69] presents an n-input 2n-output positive unate Boolean function which can be realized with $2n$ two-input gates when feedback is used but requires $3n-2$ gates if the feedback is not used. Hence, cyclification of a circuit in addition to forcing CycSAT pre-processing step to consider the "no sensitizable path", could also remedy the area overhead of introducing new gates for cyclic obfuscation. To cyclify a netlist and to increase the t_{NC} in Equation 4.3, we suggest three approaches: (1) Template-based cyclic-function mapping, (2) Input-dependency based cycle generation and, (3) Node-merging

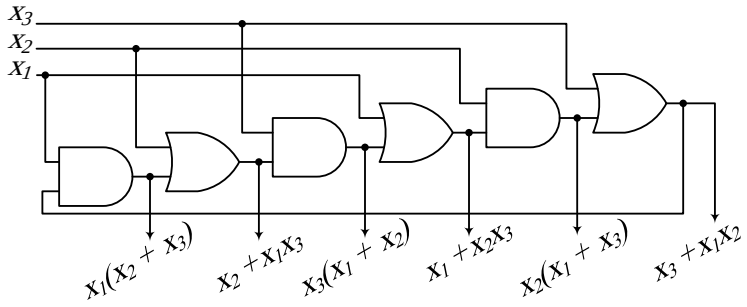


Figure 4.7: 3-input Rivest circuit implementing six functions.

cycle generation.

Template-based cyclic-function mapping

In this approach, many small cyclic Boolean circuits are collected as templates in our obfuscation library. Then, a netlist is scanned for opportunities (with and without logic manipulation) to replace a cluster of logic gates with such templates. An example of such feedback template is the circuit introduced in [69] where a special case of it (for 3 inputs) is illustrated in Fig. 4.7. To introduce cycles, the circuit could be modified to introduce at least one of the possible functions in this circuit. The candidate logic cluster is then replaced by the template. To prevent template scanning and removal attacks, in a subsequent camouflaging step (using the gate and route obfuscation) the template will be hidden. Note that many such templates could be made [66–69], and by not knowing the template type and the camouflaged technique used to hide the connection, an attacker has no prior information to identify and remove these templates.

Input-dependency based cycle generation

This method explores the correlations between signals that share common primary inputs in their fanin cone. Considering N such signals in an arbitrary stage of a DAG, some of the 2^N inputs may never occur. For example, when tracking 4 signals A, B, C, and D in Fig. 4.8, we may find that $ABCD = \{0010\}$ could not occur. A SAT solver could be used

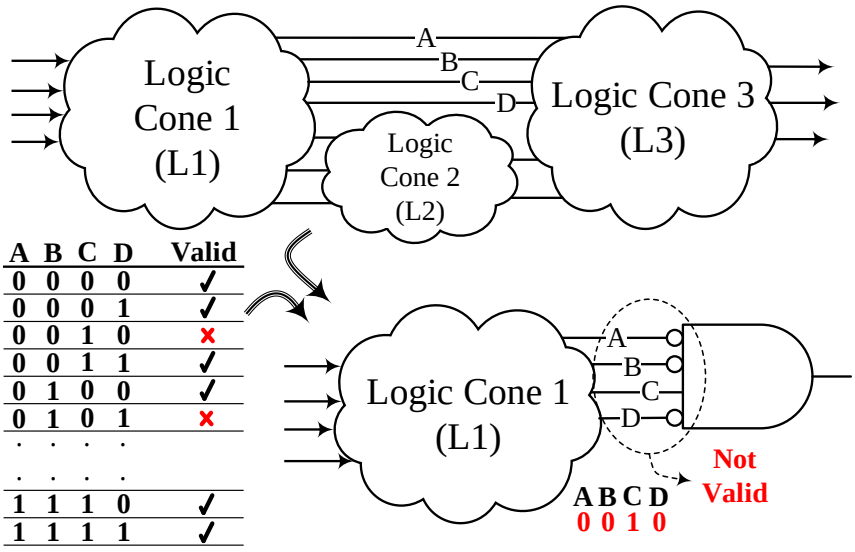


Figure 4.8: Input-dependency based cycle generation flow. Due to correlation of intermediate signals, certain signal combinations may never occur.

for finding the non-occurring input scenarios; This process is illustrated in Fig. 4.8, where the logic clusters L2 and L3 are removed, and the 4 signals are ANDed together such that for a certain case, for example, $ABCD = 0010$, the output of AND gate is evaluated to 1. Then, this circuit is given to a SAT solver to find a satisfying input assignment. If SAT solver returns UNSAT, this combination of input is chosen since it would never happen, otherwise, a different combination is checked.

In the next step, we use a sequential element and tie the discovered non-occurring input scenario to the state preserving input of the sequential element. For example, by using an SR-latch in Fig. 4.9.a, If $SR = 11$ doesn't happen, the Q_{next} is the inverse of input S. Hence, we can build a circuit that ties the discovered non-occurring input scenario to the $SR = 11$. For example, let's assume wires A, B, C and D have a non-occurring combination $ABCD = 0010$ and these signals construct the signal $Y = A + B + CD$. Fig 4.9.c illustrates the signal Y reconstructed when the non-occurring combination of the inputs is tied to SR input of the latch. After generating the cyclic logic, to hide the correlation between input signals, the wire selection is obfuscated. Finally, the SR-latch feedback is obfuscated using a set of multiplexers. This assures that CycSAT can only generate the correct NC clauses if

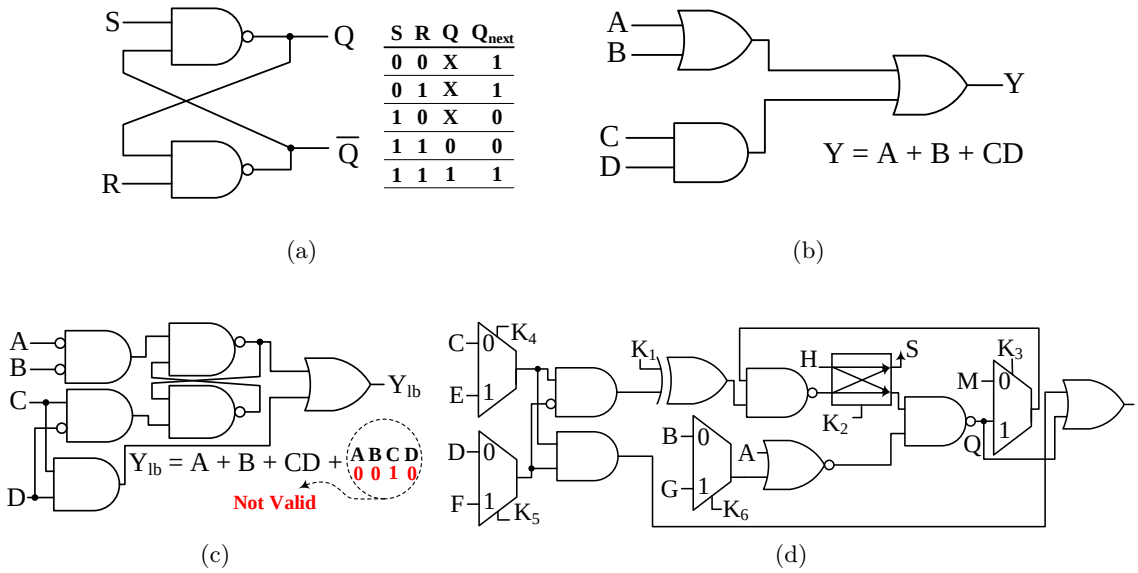


Figure 4.9: Input-dependency-based cyclification of a Boolean function. (a) SR-latch (b) original circuit (c) cyclified circuit when $ABCD = 0010$ is non-occurring. (d) obfuscated cyclified circuit using additional random inputs E, F, G, H and M.

the "no sensitizable path" condition is processed, otherwise, it breaks the SR-latch feedback and invalidates the netlist.

Node-merging based cycle generation

The third approach for cyclification of a netlist is based on the work in [66] where the logic implication is used to identify cyclifiable structure candidates directly, or to create them aggressively in circuits. At its core, the work in [66] introduces active combinational feedback cycles by merging two nodes in the original DAG. To check the validity of the generated cyclic netlist, they use a SAT-based algorithm and validate whether the formed cycles are combinational or not.

Algorithm 5 Timing Aware Cyclic Obfuscation

```
1: procedure SWITCH_INSERTION(int required_paths, circuit K)
2:     largest_cone ← output port with largest cone;
3:     b = BFS(largest_cone);
4:     while (number of inserted feedbacks < required_paths) do
5:         tail ← pop(b);
6:         if (slack(tail) > delay of a keygate and tail not marked) then
7:             path ← DFS on tail considering slacks;
8:             mark path as selected in the circuit;
9:             add feedback to the path;
10:            update the circuit's timing using STA/EDA;
11:    for (each selected path) do
12:        for (each gate in the path) do
13:            if (slack(gate) > delay of a multiplexer) then
14:                disconnect gate output;
15:                insert multiplexer;
16:                connect gate and multiplexer based on SC/LFN;
17:                update circuit timing using STA/EDA;
```

4.4 Timing Aware Cyclic Obfuscation

During logic locking, each modification to the original netlist affects the timing characteristics of the original circuit. A timing oblivious obfuscation solution could result in changes to the delay of one or more timing critical path(s) (via insertion of key gates), leading to a slower design. In this section, we argue that our proposed obfuscation solution could be designed to be timing-aware, minimizing (or removing) the impact of obfuscation on circuit timing. This can be achieved by incorporating a simple static timing analysis (STA) in our obfuscation procedure.

Our proposed solution for timing-aware cyclic obfuscation is presented in Algorithm 5. Both SC and LFN methods (supported in this algorithm) require selection of non-overlapping logic paths in the circuit for intertwined cycle creation. In our solution, presented in Algorithm 5, we find these non-overlapping logic paths in the fan-in cone (FIC) of a single primary output. The reason for selecting the logic paths in the same FIC is to take advantage of existing connections between selected logic sub-paths when one sub-path is in the FIC of at least one of the gates in the other sub-path. This condition results in the generation of many additional cycles, on top of those generated by LFN or SC. This is

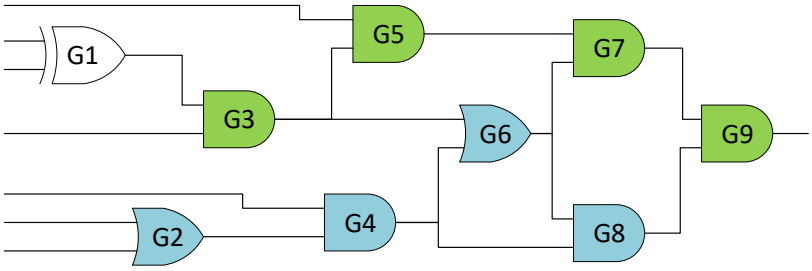


Figure 4.10: Selected paths for cyclic obfuscation in an output cone.

because each feedback could create a cycle when combined with each of path forward edges. After selection of a logic sub-path and before committing to the insertion of a new switch, the netlist is assessed for timing violation. If there is no violation, the cycle is generated and the slack of affected timing paths are updated. Finally, the logic gates in the selected sub-path are marked as **used**, removing them from future searches.

Our proposed algorithm selects new logic paths in the FIC of the selected primary output until there are no more viable sub-paths. The algorithm could be modified to continue finding new paths by selecting the next primary output candidate that has the largest number of un-**used** gates.

4.5 Experimental Results

In this section, we analyze the effectiveness of our proposed defense against SAT, CycSAT, and BeSAT attacks. For finding cycles in a netlist (after cyclic obfuscation), we implemented the cycle identification algorithm proposed in [70] using C++. Considering that the source code for BeSAT was not openly available, we implemented the BeSAT attack based on the description in [63] using Yices SAT solver [71]. Our computational platform is a Dell PowerEdge R620 equipped with Intel Xeon E5-2670 and 64GB of RAM. We used ISCAS-85 benchmarks listed and described in Table 4.1 to evaluate our solution and to compare it with the prior work. The timeout limit in our experiments is set to 10 hours: If an experiment does not conclude within the timeout limit, its table entry is marked as "t/o".

Table 4.1: Description of ISCAS-85 circuits used in this chapter.

Circuit	#Gates	#PIs	#POs	Circuit	#Gates	#PIs	#POs	Circuit	#Gates	#PIs	#POs
c432	160	36	7	c1355	546	41	32	c3540	1669	50	22
c499	202	41	32	c1908	880	33	25	c5315	2307	178	123
c880	383	60	26	c2670	1269	233	140	c7552	3513	207	108

Table 4.2: The number of cycles reported during CycSAT attack. The exponential fitting function is in form of $c = 2^{mX}$.

Circuit	N=1	N=2	N=3	N=5	N=10	N=15	m
c432	3,384	23,879	$4.6 * 10^5$	NiS	NiS	NiS	6.3
c499	10	331	1528	$1.4 * 10^6$	NiS	NiS	4.1
c880	67	1,601	1,903	$5.0 * 10^6$	t/o	t/o	4.5
c1355	59	636	$5.7 * 10^5$	$1.9 * 10^9$	t/o	t/o	6.2
c1908	13	294	12,594	$1.3 * 10^7$	t/o	t/o	4.8
c2670	273	1,570	8,912	$2.9 * 10^5$	t/o	t/o	3.6
c3540	1,215	5,991	$8.7 * 10^5$	$4.9 * 10^8$	t/o	t/o	5.8
c5315	162	4,869	6,650	$1.2 * 10^9$	t/o	t/o	6.0
c7552	11	124	1,558	$2.6 * 10^5$	$1.2 * 10^9$	t/o	3.0

In an experiment, if the netlist is too small for insertion of the number of required feedbacks, its table entry is marked as "Netlist is Small (NiS)".

4.5.1 Exponential Growth in The Number of Cycles

Cyclification Using Super Cycles (SC)

The number of cycles created in ISCAS-85 benchmarks, when using N=1, 2, 3, 5, 10, and 15 MCs of size 7 (i.e., 7 gates in a cycle) for building a SC is reported in Table 4.2. Using curve fitting techniques, the number of cycles in each netlist is also reported as a function of the number of feedbacks X, in form of 2^{mX}, in which m is the netlist-specific exponential acceleration factor. The minimum bound for m (according to the discussion in section 4.3.1) when using SC is one. However, as reported in Table 4.2, the value of m is usually far larger than one, meaning there would be a far larger number of cycles than that expected from the SC-imposed minimum bound.

Table 4.3: Percentage of area overhead for SC creation when using different number of MCs (N) of length 7.

Circuit	N=1	N=2	N=3	N=5	N=10	N=15	N=20
	Area Overhead Percentages (%)						
c432	7.50	13.75	20.00	NiS	NiS	NiS	NiS
c499	5.94	10.89	15.84	25.74	NiS	NiS	NiS
c880	3.13	5.74	8.36	13.58	26.63	39.69	52.74
c1355	2.20	4.03	5.86	9.52	18.68	27.84	37.00
c1908	1.36	2.50	3.64	5.91	11.59	17.27	22.95
c2670	0.95	1.73	2.52	4.10	8.04	11.98	15.92
c3540	0.72	1.32	1.92	3.12	6.11	9.11	12.10
c5315	0.52	0.95	1.39	2.25	4.42	6.59	8.76
c7552	0.34	0.63	0.91	1.48	2.90	4.33	5.75

As illustrated in Table 4.2, increasing the number of feedbacks exponentially increases the number of cycles, such that with only 15 feedbacks, the cycles in none of the netlists could be counted in 10-hour limit. Note that, the designer can exponentially increase CycSAT attack's pre-processing time, by linearly increasing the number of feedbacks. For executions resulted in timeout, we also confirmed that initiating CycSAT with incomplete NC clauses traps the SAT solver in an infinite loop. Hence, the attacker can not complete the pre-processing in a reasonable time, and incomplete pre-processing traps the subsequent invocation of the SAT solver. The area overhead for building the SC in terms of the number of switches depends on the number of MCs and the number of gates in each MC. The area overhead for having various numbers of MCs of 7 gates when building a SC is reported in Table 4.3.

Cyclification using Logarithmic Feedback Networks (LFN)

As discussed and proved in section 4.3.1, the lower bound on the number of generated cycles, when the LFN method for cyclification flow is adopted, is an exponential function of the number of feedbacks. Furthermore, similar to SC, the edge density of the original netlist may substantially increase the number of created cycles. This is because of the gates with fan-outs greater than one in selected logic path segments. If the output of a gate in the LFN is connected to the input(s) of another gate(s) in the same network, the resulting net

Table 4.4: Number of cycles reported during CycSAT attack using LFN method. N is the number of selected paths for creating LFN.

	N=2	N=4	N=8	N=16	N=32
Lower Bound	3	24	16072	3.8×10^{12}	2.3×10^{32}
c432	26,578	NiS	NiS	NiS	NiS
c499	192	278,577	1.3×10^{10}	t/o	NiS
c880	8,836	4.5×10^{8}	t/o	t/o	NiS
c1355	8.3×10^{6}	t/o	t/o	t/o	NiS
c1908	8.4×10^{7}	t/o	t/o	t/o	t/o
c2670	1.2×10^{7}	t/o	t/o	t/o	t/o
c3540	8.5×10^{9}	t/o	t/o	t/o	t/o
c5315	1.2×10^{9}	t/o	t/o	t/o	t/o
c7552	2.9×10^{9}	t/o	t/o	t/o	t/o

counts as an additional forward path. Then, each forward path could be matched with a feedback, resulting in an additional cycle. Considering that path segments are selected from the FIC of the same primary output, there exist many such connections (forward edges), resulting in the generation of a far larger number of cycles than the guaranteed minimum bound expected from using LFN. To illustrate this, both the number of created cycles for each benchmark and the theoretical lower bound (calculated using $\sum_{l=1}^{N} \binom{N}{l}(l-1)!$ as proved in section 4.3.1) is reported in Table 4.4. As illustrated, for most of the obfuscated benchmarks with a LFN larger than 4, cycle enumeration results in timeout after 10 hours due to the exponential number of created cycles. This indicates an exponential runtime at the CycSAT pre-processing stage. The area overhead for creating LFNs of different sizes (different number of input paths) is reported in Table 4.5. Note that in both SC and LFN, the area overhead scales with the number of inserted feedbacks and not the size of the circuit. Hence, the area overhead is smaller in larger circuits.

Capturing the power overhead of cyclic obfuscation is more involved. The leakage component of power overhead is a function of the area overhead of the obfuscation solution, and threshold voltage (VT) of inserted multiplexers. Using a high-VT switch cell reduces the leakage impact, however, it introduces additional delay [72]. In a simple implementation where standard cells are selected from a single VT, the increase in the leakage power is

Table 4.5: Percentage of area overhead for an inserted LFN for different number of selected paths (N).

Circuit	N=2	N=4	N=8	N=16	N=32
	Area Overhead Percentages (%)				
c432	5.00	NiS	NiS	NiS	NiS
c499	3.96	11.88	27.72	79.21	NiS
c880	2.09	6.27	14.62	41.78	NiS
c1355	1.47	4.40	10.26	29.30	NiS
c1908	0.91	2.73	6.36	18.18	43.64
c2670	0.63	1.89	4.41	12.61	30.26
c3540	0.48	1.44	3.36	9.59	23.01
c5315	0.35	1.04	2.43	6.94	16.64
c7552	0.23	0.68	1.59	4.55	10.93

Table 4.6: The power overhead of SC and LFN of size N=16.

Circuit	SC (N=16)		LFN (N=16)	
	Switching (%)	Leakage (%)	Switching (%)	Leakage (%)
c432	NiS	NiS	NiS	NiS
c499	NiS	NiS	212.64	75.13
c880	38.09	44.85	56.67	38.82
c1355	12.79	32.77	13.26	24.6
c1908	8.42	19.1	13.38	15.66
c2670	14.14	15.96	13.17	12.32
c3540	8.76	10.79	3.86	8.88
c5315	5.75	8.51	6.13	6.7
c7552	2.88	5.78	7.79	4.4

similar to the increase in the area. The dynamic power consumption, on the other hand, depends on the switching activity of the inserted switches. After proper activation, the switching activity of the inserted multiplexers depends on the toggling rate of the correct input net to the multiplexer. The net toggling activity, in turn, depends on the level of controllability of that net and the probable input scenario to the netlist. The power consumption of both the LFN and SC-based solutions of size N=16 is provided in Table 4.6. However, note that the power consumption could improve (at the expense of timing and security) by modifying the SC or LFN algorithm to choose nets with small toggling rate to reduce the overhead of dynamic power consumption.

Table 4.7: SAT attack, CycSAT, and BeSAT execution time.

Circuit	N=2			N=2 + SR-L=10					N=15 + SR-L=10				
	SAT	#Cycles	CycSAT-I	SAT	#Cycles	CycSAT-I	CycSAT-II	BeSAT	SAT	#Cycles	CycSAT-I	CycSAT-II	BeSAT
c432	Inf	23,879	2.56s	Inf	1.65×10^5	UNSAT	11.69s	35.48s	Inf	t/o	UNSAT	t/o	t/o
c499	0.56s	236	0.10s	Inf	397	UNSAT	0.11s	0.79s	Inf	t/o	UNSAT	t/o	t/o
c880	Inf	1,601	0.24s	Inf	7.87×10^6	UNSAT	793.12s	t/o	Inf	t/o	UNSAT	t/o	t/o
c1355	Inf	636	0.12s	Inf	5.00×10^5	UNSAT	53.21s	134.56s	Inf	t/o	UNSAT	t/o	t/o
c1908	0.28s	294	0.10s	Inf	6,467	UNSAT	0.73s	170.74s	Inf	t/o	UNSAT	t/o	t/o
c2670	Inf	1,570	0.23s	Inf	7,412	UNSAT	0.92s	17.22s	Inf	t/o	UNSAT	t/o	t/o
c3540	Inf	5,991	0.75s	Inf	6,026	UNSAT	0.75s	22.67s	Inf	t/o	UNSAT	t/o	t/o
c5315	Inf	4,869	0.61s	Inf	2.59×10^5	UNSAT	26.04s	370.08s	Inf	t/o	UNSAT	t/o	t/o
c7552	Inf	124	0.189s	Inf	164	UNSAT	0.19s	18.30s	Inf	t/o	UNSAT	t/o	t/o

4.5.2 SAT, CycSAT and BeSAT Resilience

Table 4.7 captures the result of SAT, CycSAT, and BeSAT attacks on ISCAS-85 benchmarks that are obfuscated using our proposed solution. For generating the data in this table, we prepared three sets of obfuscated benchmarks. The first set of benchmarks is obfuscated with only two MCs using the SC approach for obfuscation method. This group of obfuscated benchmarks represents cyclification with a small number of dummy cycles, with no real cycles. The netlists in the second set, are first obfuscated using 10 SR-latches (by using the input-dependency based obfuscation as described in section 4.3.2) and then are cyclified by inserting two MCs. The second group represents the case where there are some real cycles in the design, while the total number of cycles is still small. The third group is similar to the second group, however, the number of inserted MCs is increased to 15. It represents obfuscated solutions with both real and exponentially large number of dummy cycles. The results of running SAT, CycSAT, and BeSAT is captured in Table 4.7. For c432 and c499, generating large number of MCs (15) was not possible, hence, the largest number of possible MCs were used in the generation of SC.

The first group introduces a small number of removable cycles. As reported in Table 4.7, even the existence of simple cycles traps the original SAT attack in an infinite loop in most cases (except for two benchmarks that SAT solver luckily chooses a sequence of inputs that avoid or exit the trap). However, CycSAT, when uses the "no structural path" condition

(CycSAT-I) for generating the cycle avoidance clauses, easily breaks all obfuscated netlists. As illustrated in this table and predicted in Equation 4.3, CycSAT runtime (which includes the runtime for both pre-processing step and SAT solver's invocation) almost linearly varies with the number of cycles in each netlist.

For the second group, where the original circuit is also cyclified (using real cycles), the usage of CycSAT-I returns UNSAT as it produces *NC* clauses that breaks the real Boolean cycles. However, when CycSAT uses the "no sensitizable path" conditions (CycSAT-II), it breaks the obfuscation in all cases. Most notable in this data is the increase in the runtime of CycSAT attack (when compared to the first group) as the time it takes to compose the *NC* condition for each cycle based on "no sensitizable path" condition is longer. This validates the impact of logic cyclification on the runtime of CycSAT attack. Another attack possibility is BeSAT attack. However, the BeSAT attack should be slightly modified: considering that the design contains real Boolean cycles, the "no sensitizable path" condition (instead of "no structural path" in the BeSAT attack as described in [63]) should be used for the generation of the *NC* clauses. Hence, the attack could be carried by generating a set of *NC* clauses (given a deadline) and then use BeSAT to attack the obfuscation and recover from oscillating and stateful cycle conditions. To model this attack, we set "no sensitizable path" pre-processing deadline to 2 hours, and BeSAT attack time to 8 hours (total of 10 hours attack time). As shown for "N=2+SR-L=10", all but one benchmark was successful and in general, BeSAT underperform compared to CycSAT-II attack. This is because there exists a small number of cycles, and both CycSAT-II and BeSAT have found and conditioned all cycles, however, BeSAT due to the runtime monitoring of DIPs is slower compared to CycSAT-II attack.

Finally, for the third group, where the number of inserted feedbacks is increased to 15, all three attacks fail. The CycSAT-I is not applicable, as it will open real cycles, resulting in netlist malfunction, and even if pre-processing of this attack finishes (which does not) it will exit as UNSAT. The CycSAT-II fails as it can not finish the pre-processing on time. Note that by increasing the number of feedbacks, the designer can easily and

exponentially increase the required pre-processing time unreasonably long. The remaining attack possibility is the BeSAT attack. In this case, the pre-processing of NC clauses is carried until the time limit (2 hours) and then BeSAT attack is carried out. Note that in this condition, the BeSAT starts the SAT attack with a partial set of clauses generated in the pre-processing step. However, as illustrated in Table 4.7, BeSAT will reach the deadline after invalidating 100s of thousands of keys. This is when there exist millions (or larger) other keys that cause oscillating behavior which BeSAT has not yet examined and pruned (one at a time) in the time limit.

As explained previously, BeSAT only works when the number of undetected cycles (and un-conditioned keys) is small. The BeSAT attack is slow and eliminates one-incorrect-key at a time. This is when, in our proposed obfuscation solution, there exists an exponentially large number of invalid keys even after partial pre-processing: As a part of our obfuscation solution (and to create real cycles), we are using (diffused) SR-latches. To prevent stateful behavior, through careful input-logic section (as described in section 4.3.2), we ensure that the value of SR input can not evaluate to 11 (condition for statefulness). For this purpose, the input logic cone to S and R input is constructed by exploiting the interdependency of selected wires in the netlist. However, the selection of inputs is further hidden through routing obfuscation. In this case, only with the application of the correct key, the inter-dependence of the input wires will render the SR-latch non-stateful (by skipping the 11 input). Let's assume $S = g(K_1, X)$ and $R = f(K_2, X)$, where the g and f are the logic representing the input cone of S and R input to the SR-latch, K_1 and K_2 are the key gates in the fan-in cone of S and R, and the X is the choice of primary input. In this scenario, any choice of K_1, K_2 and X that could make the $SR = 11$ will result in a stateful circuit. From this analysis, the worst-case scenario for BeSAT is a function of the size of primary input X, and key selection K_1 and K_2 for which the wire S and R evaluate to 1, which is an exponential function of the key-length $K = (K_1 \bigcap K_2)$. Considering that our solution builds a strongly-connected graph, the FIC of S and R could span to all the key-gates. Hence, the number of invalid keys that should be banned is exponentially large. Considering this

discussion, and for a large number of key combinations that should be banned (one at a time), as shown in the results for "N=15+SR-L=10", BeSAT attack does not work against our proposed solution.

4.5.3 SAT, CycSAT and BeSAT Resiliency of Previous Methods

In this section, we study the effectiveness of previously proposed cyclic logic solutions and compare them with our proposed solution. To attack the prior art solutions we use the modified CycSAT attack as described and formulated in section 4.2.2 of this chapter. The modified CycSAT attack works similar to the original CycSAT attack, however, instead of composing the *NC* clauses per detected feedback, it composes the *NC* clauses per detected cycle.

The original cyclic locking method was introduced in [32] where authors proposed inserting multiplexers in the circuit to create cycles. This obfuscation solution attempts to create irreducible cycles. This method can only create dummy cycles as it does not affect the DAG nature of a combinational netlist and is referenced in this chapter as *glsvlsi17*. The second method discussed here [64] considers CycSAT attack and tries to defeat CycSAT-I using an auxiliary-circuit. This method was discussed in section 4.2.1. By adding the proposed auxiliary-circuits to a design, real cycles are formed, converting the DAG nature of the netlist to a DCG. The netlist is then augmented with additional dummy cycles (similar to the glsvlsi17 method), making the netlist to contain both real and dummy cycles. In this chapter, we use the name *date18* to refer to this cyclic obfuscation solution.

To assess the effectiveness of prior art solutions, we modeled each of the glsvlsi17 and date18 to obfuscate the ISCAS-85 benchmarks. To compare the evaluation results of prior art to that of our proposed solution (in Table 4.7), the glsvlsi17 method is implemented using 15 randomly selected feedbacks of length 7, while the benchmarks prepared using date18 solution are obfuscated using the same number of feedbacks (15) and 10 real cycles (for DAG to DCG transformation), implemented using the auxiliary-circuit as described in [64]. For smaller benchmarks, where insertion of this many feedbacks was not feasible,

Table 4.8: SAT attack, modified CycSAT, and BeSAT results for evaluation of glsvlsi17 method.

Circuit	#Cycles	SAT		CycSAT-I		BeSAT	
		Time	Iteration	Time	Iteration	Time	Banned
c432	32	t/o	-	0.02	1	0.45	0
c499	282	t/o	-	0.05	1	0.88	0
c880	36	1.35	61	0.13	15	3.59	0
c1355	t/o	t/o	-	t/o	-	7220.83	3
c1908	1,625	t/o	-	0.95	83	8.44	0
c2670	129	t/o	-	2.26	19	55.11	0
c3540	606	0.63	41	0.70	14	10.03	0
c5315	4,216	1.7	33	1.19	45	32.75	0
c7552	1,117	2.35	105	1.77	73	43.24	0

we have inserted the largest feasible number of feedbacks. To show the effectiveness of our solution in increasing the runtime of the CycSAT pre-processing step, we have also evaluated the number of generated cycles for each of the prior cyclic obfuscation (glsvlsi17 and date18) solutions.

Table 4.8 captures our evaluation results for glsvlsi17 when attacked using SAT, CycSAT-I, and BeSAT. As expected the success of SAT attack on selected benchmarks is random, as generated cycles could trap the SAT solver. Note that by increasing the number of feedbacks, the chances of trapping the SAT solver increases. CycSAT-I breaks the obfuscation and finds the key to all but one obfuscated benchmark. For c1355, cycles could not be processed within the 10-hour time limit, and the attack is timed out. But this case is a great showcase to see the power of BeSAT. As expected, BeSAT could also break this obfuscation. Considering that the pre-processing for most of the benchmarks could be done in less than 2-hours, and all cycles could be found for such small obfuscations, the number of banned keys for all cases but one is zero. For this reason and for the additional overhead of runtime monitoring of SAT execution time, the BeSAT takes longer than CycSAT-I. The only interesting scenario is for c1355, where the CycSAT-I is timed out and can not finish the pre-processing of all cycles. In this case, the incomplete set of NCs is used in BeSAT, and with only 3 banned keys, BeSAT skips the traps and finds the correct key. Note that the reason why BeSAT does work is that the number of oscillating keys generated in this

Table 4.9: Evaluating date18 obfuscation against SAT, CycSAT and BeSAT.

Circuit	#Cycles	SAT		CycSAT-I		CycSAT-II		BeSAT	
		Time	Iteration	Time	Iteration	Time	Iteration	Time	Banned
c432	62	t/o	-	0.02	UNSAT	0.3	18	9.19	0
c499	1,157	t/o	-	0.06	UNSAT	0.14	18	2.00	0
c880	56	t/o	-	0.04	UNSAT	0.31	23	5.11	0
c1355	t/o	t/o	-	t/o	-	t/o	-	7268.77	12
c1908	1,645	1.99	144	0.02	UNSAT	0.88	68	205.02	0
c2670	149	t/o	-	0.03	UNSAT	0.53	41	10.53	0
c3540	626	6.49	187	0.1	UNSAT	1.53	37	18.19	0
c5315	4,236	t/o	-	0.05	UNSAT	2.06	60	30.45	0
c7552	1,137	t/o	-	0.08	UNSAT	1.9	31	40.75	0

obfuscation solution is small. This is unlike our proposed solution that there exists an exponentially large number of such keys, and if given to BeSAT, they have to be eliminated one at a time.

Table 4.9 captures evaluation results for date18 method. Aware of the shortcomings of glsvlsi17, the date18 solution was proposed as a CycSAT-resistant obfuscation solution. The proposed auxiliary-circuit by itself has a minimal impact on the number of cycles. However, this method is expected to have a larger number of stateful cycles, and when the original SAT attack used there are higher chances for trapping the SAT solver in an infinite loop. The results in Table 4.9 support this hypothesis, as only two benchmarks are successfully attacked using the base SAT attack. When attacked using CycSAT-I, the date18 solution remains resistant as the pre-processing step of CycSAT-I incorrectly opens the real cycles during *NC* clause generation. However, when the modified CycSAT-II attack, as described in section 4.2, is deployed, could easily break all instances of obfuscated solutions except c1355 (that could not be pre-processed in a reasonable time for having a very large number of cycles). However, in the case of BeSAT and after limiting the pre-processing time to two hours, the key for c1355 could be recovered in 68.77s after 2 hours of *NC* clause generation. Other benchmarks that previously was broken by CycSAT-II is also broken by BeSAT with zero banned keys since the generated *NC* clauses cover all undesirable cycle conditions. Note that for this attack, the *NC* clauses for BeSAT are generated using the

"no sensitizable path" condition, otherwise the attack will return as UNSAT.

Comparing the glsvlsi17 and date18 data in Tables 4.8 and 4.9 with that of our proposed solution in Table 4.7 illustrate the effectiveness of our solution: none of the obfuscated netlists using our solution could be broken by SAT, CycSAT-I, CycSAT-II, or BeSAT (original and modified) attacks, as it includes a solution to trap both the SAT solver and pre-processing step of CycSAT/BeSAT. Note that, when deploying SAT or CycSAT attack to break glsvlsi17 or date18, the runtime, in addition to the number of inserted feedbacks, also depends on the selection of feedbacks. Hence, a random selection of feedbacks in glsvlsi17 and date18 results in considerable variation in the attack time. Therefore, these solutions, unlike our proposed solution, can not guarantee a monotonic increase in the runtime of the attack as the number of randomly selected feedbacks increases. Note that in our solution, the runtime is dominated by CycSAT's or BeSAT's pre-processing step, and this runtime is linearly dependent on the number of cycles, and the number of cycles is an exponential function of the number of inserted feedbacks. Hence, we can guarantee a monotonic increase in the overall runtime of the attack against our proposed solution as the number of inserted feedbacks increases.

4.5.4 Timing Aware Cyclification

As described in section 4.4, inserting logic gates in timing-critical paths would increase the critical path of the netlist resulting in a performance penalty. To minimize the performance penalty to the extent possible, we proposed a timing aware cyclic obfuscation flow in section 4.4. This solution would only affect the timing if it can no longer use non-critical timing paths for feedback insertion.

Table 4.10 captures the result of our proposed timing aware cyclic obfuscation when allowing 0% and 5% delay overhead for cyclic obfuscation. Using this delay constraint, the algorithm tries to insert the maximum number of feasible feedbacks in each benchmark using the SC solution proposed in section 4.3.1. In this table, we have provided a measure of the maximum number of MCs that could be implemented in each benchmark for building

Table 4.10: Timing-aware obfuscation results for the Super Cycle method.

Circuit	Slack = 5%				Slack = 0%				
	#Cycles	SAT(s)	#Keys	#MCs	#Cycles	SAT(s)	#Keys	#MCs	Area %
c432	303,476	0.14	15	2	NiS	NiS	NiS	NiS	NiS
c499	NiS	NiS	NiS	NiS	NiS	NiS	NiS	NiS	NiS
c880	t/o	t/o	95	18	t/o	0.23	51	12	26.63
c1355	t/o	t/o	109	23	2,766	0.55	25	8	9.16
c1908	t/o	t/o	187	38	t/o	t/o	111	24	25.23
c2670	t/o	t/o	335	70	t/o	t/o	244	53	38.46
c3540	t/o	t/o	378	75	t/o	t/o	274	57	32.83
c5315	t/o	t/o	448	110	t/o	t/o	446	95	38.66
c7552	t/o	t/o	729	183	t/o	t/o	632	158	35.98

a strongly connected graph before running out of usable gates. The key count is the sum of the number of key values needed for managing the MCs and the number of key values needed for managing the additional multiplexers (used for creating outgoing edges from internal gates in each MC). As illustrated, the maximum number of MCs and key values is a function of the netlist size and the acceptable delay overhead. Note that in larger benchmarks, even without incurring a time penalty we can insert a large number of MCs, pushing CycSAT attack to be trapped in its pre-processing step until timeout. In addition, note that with 10 MCs, our C++ implementation of pre-processor can not finish counting the number of generated cycles, and according to SC and LFN lemmas proved in sections 4.3.1 and 4.3.1, the number of generated cycles exponentially grows with each added feedback. Hence, we can make the attack-time unreasonably long with no or limited timing impact.

The number of MCs and the number of gates in each MC (e.g., cycle length) could affect the number of created cycles and defines the SAT resiliency of the circuit. Parameters like targeted frequency and area overheads should also be considered during cyclic obfuscation. However, this could create a trade-off on how SAT-resilient a circuit is versus how efficiently it could be implemented.

4.6 Conclusion

In this chapter, we proposed a new mean of cyclic obfuscation that is immune to SAT, CycSAT and BeSAT attacks. To make the pre-processing step of CycSAT and BeSAT attacks ineffective, we proposed two mechanisms (SC and LFN) for exponentially increasing the number of generated cycles with respect to the number of inserted feedbacks. In addition, we proposed three mechanisms to cyclify the circuit with real cycles (Cyclic Boolean Logic). The addition of real cycles forces an attacker to generate the "no sensitizable path" conditions during the pre-processing step of CycSAT or BeSAT attacks, which is considerably more time consuming than "no structural path" generation. The exponential increase in the number of feedbacks prevents the attacker from generating NC conditions for all cycles in a reasonable amount of time. This breaks CycSAT attack. The BeSAT attack can proceed to its SAT stage with an incomplete set of NC clauses, however, it has to ban remaining invalid keys one at a time, and there exists an exponentially large number of such keys. Hence, it also fails to break the proposed solution.

Chapter 5: A Strong Obfuscation Solution for Circuits with Restricted Access to Scan Chain

The original SAT attack was only applicable to the combinational circuits. However, the existence of the scan chain, allows an adversary to treat the FSM and sequential circuits as a combinational circuit; using the scan chain, the attacker to load desired input into scan registers, carry the attack for one cycle, and readout the output through the scan chain. Hence, to prevent SAT attack on obfuscated sequential and FSM solutions, various means for restricting access to the scan chain [37, 42, 43] was investigated. In this approach, which is illustrated in Fig. 5.1, an obfuscation solution is constructed using two key values: (1) a key for obfuscating the functional logic, and (2) a key for obfuscating the scan chain.

5.1 New Structures for SAT Resiliency

Restricting access to (or locking of) the scan chain, however, did not stop the researchers from developing variants of SAT attack solution capable of attacking an obfuscated circuit. Lack of access to the scan chain was addressed in [46] by changing the attack model to find a sequence of inputs (rather than a single input) resulting in incorrect output. This attack, so-called unrolling-based SAT (UB-SAT) attack, expands the given FSM in time to be able to find a sequence of distinguishing inputs.

To defend against UB-SAT and model checker based attacks in design with restricted access to the scan chain, in this chapter, we introduce a new obfuscation solution denoted as Deep Faults and Shallow State Duality (DFSSD). The DSFFD obfuscation scheme exploits the weaknesses of the existing attacks in obfuscating FSM and sequential circuits and prevents these attacks from satisfying their early exit conditions, forcing them to become unbounded. To build the DFSSD solution, we propose a combination of two concepts:

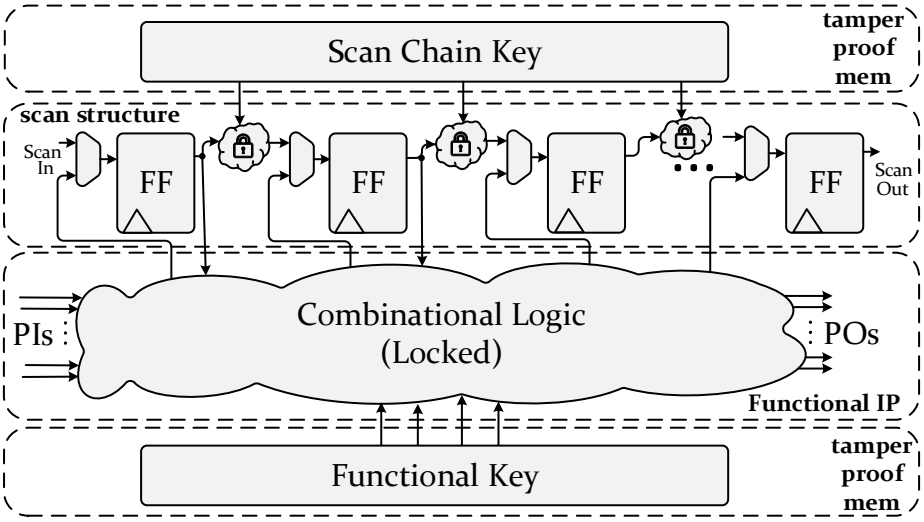


Figure 5.1: An obfuscated IC with restricted access to scan chain. In such circuit, there are two separate keys: one for unlocking the scan chain (for test), and one for unlocking the function.

(1) Encrypting **Deep Faults (DF)**, the discovery of which requires specific traversal patterns with a large enough depth that cannot be reached by bounded model checkers or unrolling based SAT attacks. (2) Encoding **Shallow State Duality (SSD)**, in which by implementing key-controlled duplicate states, the early termination conditions of the UB-SAT are violated.

As described earlier, limiting access to the scan chain removes the ability of the attacker to deploy a pure SAT attack on the combinational logic between internal scan registers, and has to revert to the weaker variant of SAT attacks such as UB-SAT (working with only primary input and primary output). Following is a short background on Scan chain obfuscation and proposed attack solutions for de-obfuscating such solutions.

5.2 Securing Scan Chain Structure

Several methods have been recently proposed in the literature to obfuscate the scan chains [73–75]. To secure the test and debug operations, [42] proposed a design-for-security (DFS) flow that deploys a structure, denoted as *Secure Cell* (SC). However, SC was compromised via

the *shift-and-leak* attack [43]. Another early attempt in this domain was the *Encrypt Flip-Flop* (EFF) [73] scheme. In EFF the output of each scan flop is obfuscated based on a key value such that either the Q or Q_{bar} output is propagated in the scan chain, and accordingly, the *scan-in* sequence is also modified. The EFF was also tackled by the *ScanSAT* attack [48]. Later, the *SeqL* obfuscation scheme [75] extended the EFF by separating scan chain keys from the logic locking keys, and by creating functional isolation through locking a subset of flip-flop inputs and scan-output pairs. The *Dynamically Obfuscated Scan* (DOS) [74] scheme obfuscates the scan chain while periodically changing the obfuscation key during the test process. Assuming a hard to break scan chain obfuscation, the pure SAT attack could be no longer applied. Hence, an attacker should resort to SAT attack variants designed for attacking scan-access restricted obfuscation solutions by only relying on controllability (observability) of primary inputs (outputs).

5.3 Deobfuscation Methods Without Scan Chain Access

El Massad et al. [46] extended the SAT attack to circuits with no scan chain access, proposing an attack that only required access to the primary input/outputs of an activated chip. The attack procedure is shown in Algorithm 6. Similar to the SAT attack, it has an iterative process for pruning the search space. However, due to the restricted access to the internal registers, rather than finding a Discriminating Input (in each iteration), it finds a sequence of inputs X denoted as *Discriminating Input Sequence* (X_{DIS}) that can generate two different outputs for the same input sequence for two different keys. In this algorithm, $C(X, K, Y)$ refers to the obfuscated circuit producing output sequence Y using input sequence X and key vector K, and $C_{BlackBox}(X)$ refers to the output sequence of the activated circuit for the same input sequence. After transforming the obfuscated circuit to a circuit SAT ($Model$) problem, the attack instantiates a Bounded Model Checker (BMC) to find the X_{DIS}. After the discovery of each X_{DIS}, the $Model$ is updated with a new condition to make sure that the next onset of keys, that will be discovered in the subsequent attack

Algorithm 6 Sequential Attack on Obfuscated Circuits

1: $b = initial\_boundary$, $Terminated = False$;
2: $Model = C(X, K_1, Y_1) \wedge C(X, K_2, Y_2) \wedge (Y_1 \neq Y_2)$;
3: **while** not $Terminated$ **do**
4: **while** $(X_{DIS}, K_1, K_2) \leftarrow BMC(Model, b) = T$ **do**
5: $Y_f \leftarrow C_{BlackBox}(X_{DI})$;
6: $Model = \wedge\, C(X_{DIS}, K_1, Y_f) \wedge C(X_{DIS}, K_2, Y_f)$;
7: **if UC**$(Model, b) \vee$ **CE**$(Model, b) \vee$ **UMC**$(Model)$ **then**
8: $Terminated$;
9: $b = b + boundary\_step$;

iterations, produce the same output for previously discovered X_{DIS}. This process continues until no further X_{DIS} is found within the boundary of b.

After reaching the boundary, the algorithm checks three criteria to determine if the attack can be terminated:

(1) Unique Completion (UC): This criterion checks for the uniqueness of the key. If there is only a single key that satisfying all previous *DIS*es, the attack is terminated.

(2) Combinational Equivalence (CE): If there is more than one key that agrees with all previously found X_{DIS}, the attack checks the combinational equivalency of the remaining keys. In this step, the input/output of FFs are considered as pseudo primary outputs/inputs allowing the attacker to treat the circuit as combinational. The resulting circuit is subjected to a SAT attack, and if the SAT solver fails to find a different output or next state for two different keys, it concludes that all remaining keys are correct and the attack terminates.

(3) Unbounded Model Check (UMC): If UC and CE fail, the attack checks the existence of a DIS for the remaining keys using an unbounded model checker. This is an exhaustive search with no limitation on bound (or the number of unrolls). If no DIS is discovered, the existing set of DIS is a complete set, and the attack terminates. Otherwise, the bound is increased and previous steps are repeated. The original implementation of this attack [46] uses NuSMV as the model checker and is not scalable for larger circuits. Shamsi et al. improved this attack via implementing several tweaks in the attack procedure [47].

The practicality of UB-SAT attack (proposed in [46]) is grounded on the use of a fast

bounded model checker (BMC) [76] and the implementation of early termination strategies to avoid the exhaustive search. This allows the attacker to avoid using time-consuming and exhaustive unbounded model checking runs for the discovery of DISes and to find the obfuscation key in a reasonable time. For having an effective obfuscation technique against this attack, we need an obfuscation solution that 1) prevent the UC and CE early termination, and 2) pushes the required bound for a BMC solver to an unreasonably large bound (which is defined at design time), resulting in unreasonable attack time against the proposed obfuscation solution. These two objectives are the future direction of our research.

5.4 Proposed Methods

The practicality of UB-SAT attack (proposed in [46]) is grounded on the use of a fast bounded model checker (BMC) [76] and the implementation of early termination strategies to avoid the exhaustive search. This allows the attacker to avoid using time-consuming and exhaustive unbounded model checking runs for the discovery of DISes and to find the obfuscation key in a reasonable time. In this section, we describe an obfuscation solution that 1) prevent the UC and CE early termination, and 2) pushes the required bound for a BMC solver to an unreasonably large bound (which is defined at design time), resulting in unreasonable attack time against the proposed obfuscation solution.

5.4.1 Shallow State Duality

The first termination criterion (UC) relies on the uniqueness of the key and it fails if there is more than one valid key for the obfuscated circuit. In the sequential attack proposed in [46], UC was the main termination criterion for most of the benchmarks. For the second termination criteria (CE), successful termination relies on the equality of all next state and output values for remaining candidate keys for all input and state combinations.

Our proposed solution for breaking both UC and CE termination checks is simply adding duplicate key controlled, yet valid states such that more than one valid key exists. We refer

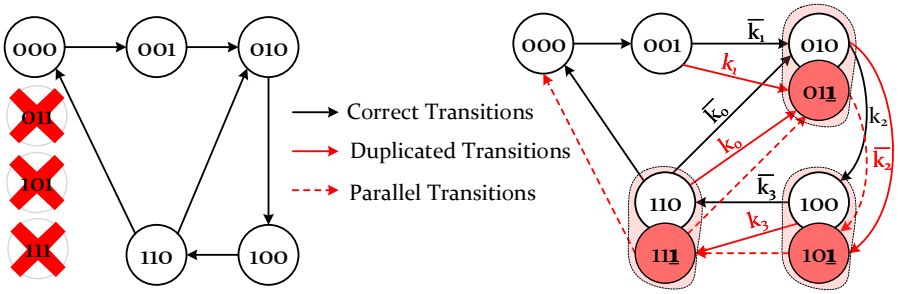
Algorithm 7 Extracting an unreachable state with minimum hamming distance from a reachable state

1: $boundary = limit$, $i = 1$, $hd = 1$
2: $Model = C_{comb}(X, S, O, S_{next}) \wedge C_{comb}(X_1, S_{init}, O_1, S_1)$
3: **while** $true$ **do**
4: $A = (\forall(S, X), \exists S_{urs}, S_{next} \neq S_{urs}) \wedge (HD(S_{urs}, S_i) == hd)$
5: **if** $(S_{i-1}, S_i, S_{urs}) \leftarrow QBF(Model \wedge A) = T$ **then**
6: return S_{i-1}, S_i, S_{urs}
7: **else if** $i < boundary$ **then**
8: $i = i + 1$
9: $Model = \wedge\ C_{comb}(X_i, S_{i-1}, O_i, S_i)$
10: **else if** $i == boundary$ and $hd < output\_width$ **then**
11: $i = 1$, $hd = hd + 1$

to this scheme as Shallow State Duality (SSD). This concept is illustrated in Fig. 5.2. In this example, the original FSM has five Reachable States (RS) and three Un-Reachable States (URS). In the modified FSM, the unreachable states are used to replicate three of the existing states such that the transition to the original or replicated state is controlled by a key. In this example, all the replicated states produce the same outputs as the original state and key bits are correct for both values of 0 and 1, although, it might not be the case in a different implementation. Therefore, the UC check fails as more than one correct key exists. In addition, in the CE check, the input to the registers is considered as a primary output. Hence, for duplicated states, two different key values do not generate the same output as they do not reach the same state. Note that the SSD is not a form of obfuscation as multiple keys are correct keys and it should be combined with our obfuscation solution which is described next. However, it is an effective and low-overhead technique to prevent early termination of the UB-SAT attack and its variants.

The duplicate states can be added during the state encoding (design time) or after logic synthesis (physical design time). Fig. 5.2 shows an example of a state transition graph encoded with duplicated states at design time. The circuit functionality remains unchanged while the circuit has $2^4 = 16$ correct keys.

For adding duplicate states to a synthesize netlist, we first need to find a few unreachable states (S_{urs}). The Algorithm 7 describes our approach for finding such states using a quantified Boolean formula (QBF) solver. To minimize the logic (overhead) needed for



Current State	Next State I=0	Next State I=1	output		Current State	Next State I,k_i=00	Next State I,k_i=10	Next State I,k_i=01	Next State I,k_i=11	output
000	001	001	A	⟶	000	001	001	001	001	A
001	010	010	B	⟶	001	010	010	011	011	B
010	100	100	C	⟶	010	101	101	100	100	C
011	✗ NOT USED ✗			✗	011	101	101	101	101	C
100	110	110	D	⟶	100	110	110	111	111	D
101	✗ NOT USED ✗			✗	101	111	111	111	111	D
110	000	010	E	⟶	110	000	010	000	011	E
111	✗ NOT USED ✗			✗	111	000	000	011	011	E

Figure 5.2: (a) Original state transition graph, (b) Modified state transition graph with duplicate states (shallow state duality).

encoding the duplicate states, we search for S_{urs} with minimum hamming distance (HD) from one of reachable (existing) states. In this Algorithm, inputs, states, outputs, and next states are defined as X, S, O, and S_{next}, respectively, and C_{comb} refers to the combinational representation of the original circuit in which the input/output of FFs are considered as pseudo primary outputs/inputs (similar to CE check in UB-SAT attack).

After initializing the boundary limit and defining the desired hamming distance (e.g. hd=1), a model consisting of two C_{comb} instances is created. To find a S_{urs}, one instance of C_{comb} is used as $C_{comb}(X, S, O, S_{next})$ with *for-all* condition on its primary inputs (X) and current states (S) to generate all the outputs (O) and next states (S_{next}) that could be produced by the C_{comb}. By assuming $S_{next} \neq S_{urs}$, the QBF solver will attempt to find a set of values for S_{urs} that is not a part of the generated S_{next}. Then, to select a URS from the set of S_{urs} that has a hamming distance of hd from a RS, another instance of C_{comb} as $C_{comb}(X_1, S_{init}, O_1, S_1)$ is used. In the QBF solver, this instance produces RSes (S_1) that are reachable from the initial state (S_{init}). Any URS in S_{urs} with HD of one from the S_1

could be considered as the answer. If such a URS was not found, a new copy of C_{comb} as $C_{comb}(X_2, S_1, O_2, S_2)$ is added to the model to produce RSes that are reachable from the initial state in two cycles. If necessary, this unrolling continues until the boundary limit to check S_{urs} with all RSes reachable in i cycles. When a URS is found, it is added to the netlist by adding the logic to make the transition between the URS and the original states based on the key value. The URS will produce the same output as the original RS it was duplicated from and will transition to the same next state(s) (or the duplicate of the next states).

5.4.2 Deep Faults

The sequential attack [46] relies on a bounded search space for finding a discriminating input sequence X_{DIS}, and it keeps increasing the boundary if the termination checks fail. The X_{DIS} is a sequence of inputs, each forces a transition to a new state until a discriminating state is reached, where a discriminating state refers to a state whose output is different for the same input with two different keys (a DIP condition). This state traversal (based on X_{DIS}) will not include any other discriminating state transition or repeated state. *Such a discriminating state could only be found if the shortest state traversal path from the initial state to that state is shallower than the boundary condition (the number of transitions) specified when invoking the BMC solver.*

The traversal depth of a sequential/FSM circuit is defined as the maximum number of state traversals (starting from initial state) where no state is visited twice. However, the sequential/FSM circuits may have a limited traversal depth [77]. This makes a BMC a plausible attack for finding all possible DIS in such circuits, as all states can be visited within a reasonably small bound. Our solution to protect against BMC formulated attack (e.g., [46]) is to increase the traversal depth of the FSM/sequential circuits and push the impact of wrong keys into deep states beyond reach of the BMC (with reasonable bound). This makes the discovery of such DIS unreasonably time consuming. We refer to such faults as deep faults.

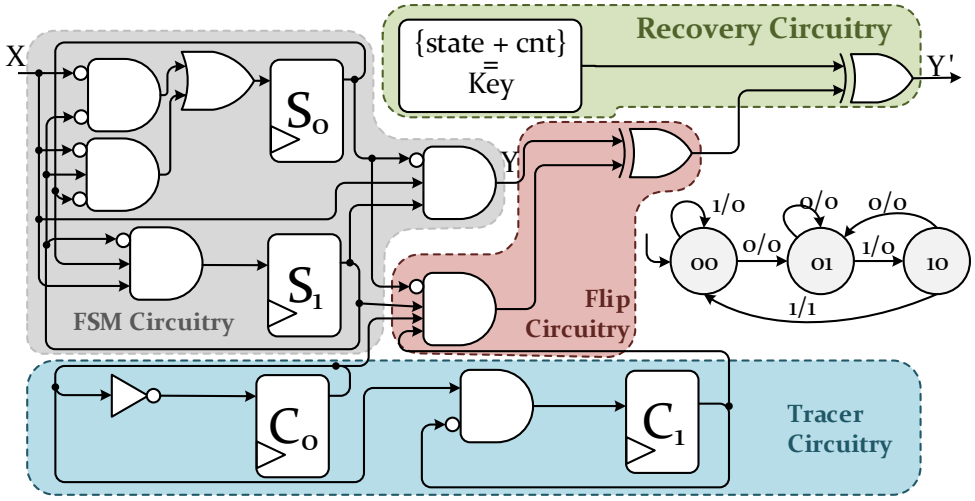


Figure 5.3: Implementation of Deep Fault (DF) obfuscation for a 011 detector. The protected pattern include the counter (C_1C_0) FFs, and the (S_1S_0) state FFs. The state transition graph of the circuit is shown in the middle right.

Our obfuscation methodology for creating Deep Faults (DF) is described via the example shown in Fig. 5.3. The circuit targeted for obfuscation is a simple '0-1-1' input sequence detector. As illustrated, the DF is implemented by adding 1) a tracer circuit, 2) a flip circuit, and 3) a recovery circuit to the original circuit. The tracer, as described earlier, is a function-modified counter or a LSFR that changes its state each time a triggering event is observed. The triggering events can be selected state transitions, state visits, or simply the rising edge of the clock. For simplicity, in Fig. 4, the triggering event is the clock and the tracer is a 2-bit counter. The flip circuit toggles the value of a single primary output of the original circuit when a protected pattern is observed. The protected pattern is a predefined pattern generated by combining selected state registers (from the original design) and tracer's state register. In Fig. 5.3, the flip circuit is shown as a four-input AND gate that fires when the protected '10,11' pattern for 'S_1S_0, C_1C_0' is observed. The last component of the Deep Fault obfuscation is the recovery circuit that toggles the output signal when the inserted obfuscation key agrees with the protected pattern. Hence, when the correct key is applied (1011 in this case), the previously flipped output related to the protected

Table 5.1: Truth table for the 011 detector obfuscated using a two-bit counter.

$S_1S_2C_1C_2$	Y	K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_9	K_{10}	K_{11}	K_{12}	K_{13}	K_{14}	K_{15}
0000	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
0001	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
0010	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
0011	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
0100	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
0101	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
0110	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
0111	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓
1000	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
1001	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
1010	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
1011	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
11xx	S=11 is unreachable																

pattern will flip back (recovered) by the recovery circuit, however, insertion of a wrong key will result in flipping a correct output.

Table 5.1 shows the truth table of the circuit in Fig. 5.3 for all key-combinations. In this circuit, when the DF is subjected to UB-SAT attack, each input can only rule out a single wrong key. Thereby, the pruning power of each discovered DIS is very limited, and the correct key is found only when the protected input is tested. This concept is similar to obfuscation solutions using point functions (e.g. SARLock [15] and Anti-SAT [16]). However, there is a fundamental difference. In point functions, the adversary uses a random input, and although the average case or worse case attack time is an exponential function of the key size, the attacker can potentially discover the correct key with a single lucky attempt. However, in deep faults, the discovery of DISes is conditioned on the tracer state, which cannot be directly controlled by input. Hence, it can guarantee a minimum bound on the number of required DISes before the discovery of the fault, which is at least equal to the number of cycles needed for the tracer to reach the fault generation state. This is a necessary condition for the generation of the fault, but it is not enough. For the fault to occur, the selection of state registers of the circuit that are included as a part of a protected pattern should also reach the fault generating pattern. Hence, the number of

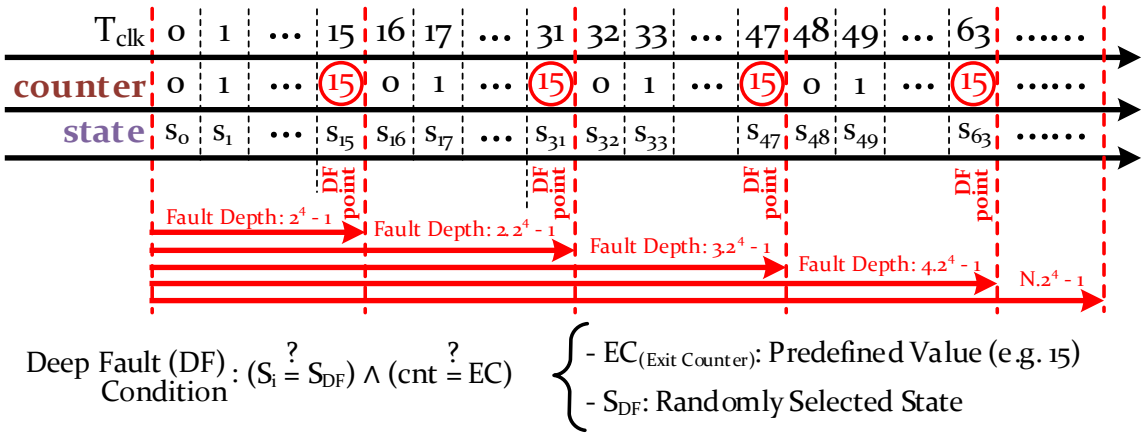


Figure 5.4: Deep faults mechanism for an always-on counter.

required DISes, which is equal to the number of cycles to reach the protected pattern, can be far larger.

The lower bound for finding the protected pattern could be defined based on the tracer event counting mechanism. For simplicity, let's assume a counter is used as the tracer circuit.

Lemma 1. *The bound requirement for a BMC solver to find the protected pattern of a deep fault which is implemented using a simple clocked counter is $C = 2^w$ where w is width of the counter.*

Proof. The protected pattern consists of two parts: 1) w bits of tracer (counter) register bits, and s bits of state registers. As illustrated in Fig. 5.4, the portion of protected pattern implemented by counter is reached every $C = 2^w$ cycles. However, the state transition does not have a predefined traversal order, and the fault is only generated when the protected pattern (state-tracer) is observed. If the s bits of state registers, that are selected for inclusion in the protected pattern, do not take the needed value to build the protected pattern at cycle C, the fault is not generated. The next viable cycle for reaching the protected pattern will be at $2 \times C$ or in general at $N \times C$. Hence, the minimum bound requirement for a BMC attack to discover the fault is $C = 2^w$. ■

Lemma 2. *The bound requirement for a BMC solver to find the protected pattern for*

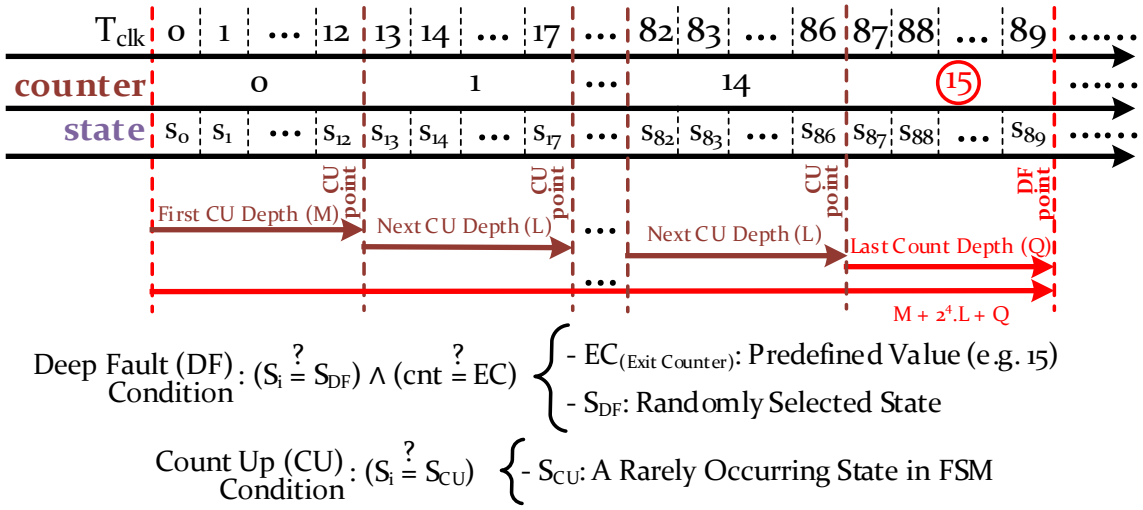


Figure 5.5: Deep faults mechanism for transition-triggered counter.

a deep fault which is implemented using a tracer that counts a selected ***state transition*** *is $M + C \times L + Q$, where M is the shortest path to reach the selected state transition from the initial state, $C = 2^w$, w is width of the counter, L is the shortest sequence of state transitions to visit the selected triggering transition twice (shortest cycle including the triggering transition), and Q is the number of state transitions to reach a state whose encoding completes the protected pattern signature.*

Proof. As shown in Fig. 5.5, the tracer only counts up if a specified state transition occurs. It takes at least M cycles for the first triggering event to occur. After this transition, the shortest sequence of transition that could result in a count-up is L, where the state transition is repeated. The number of times the triggering state transition should be visited is $C = 2^w$ times. After $M + (C \times L)$ cycles, the tracer portion of the protected pattern is ready for fault generation. However, we still need another Q cycles to reach a state whose encoding completes the protected pattern. If the target state could not be met in $M + (C \times L) + Q$ cycles, it might need to repeat $(C * L)$ for N times to be able to reach the target state. So the number needed cycles for generating the deep fault is $M + N(C \times L) + Q$. The lower bound of required cycles (equal to the number of DIS) occurs at $N = 1$, thus the

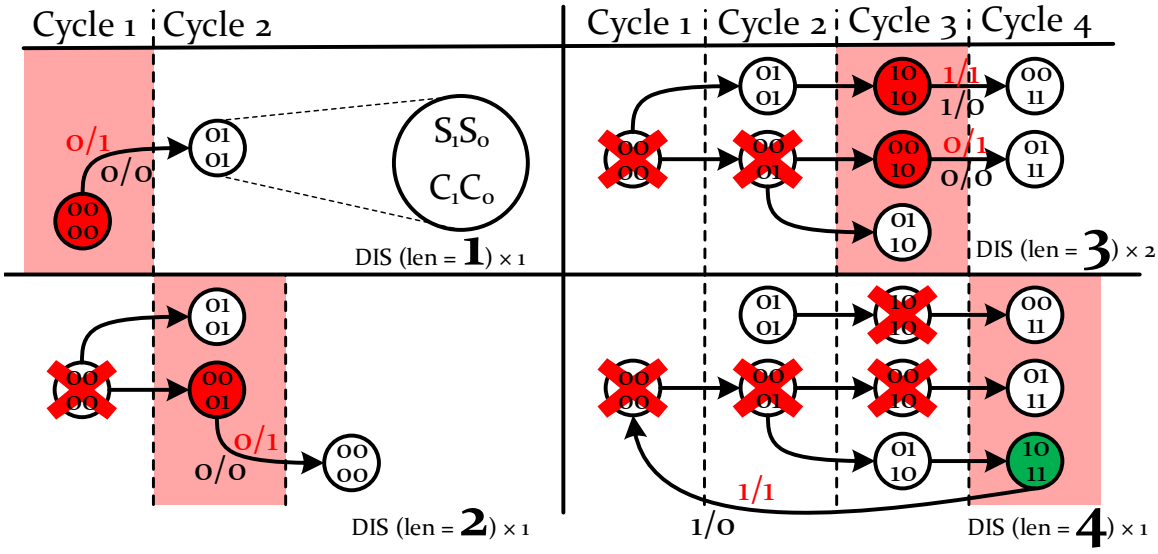


Figure 5.6: Cycle by cycle DIS discovery for the 011 detector of Fig. 5.3, which is obfuscated using DF with a 2-bit counter. The deep fault is discovered when the deep state representing the protected state (shown in green) is reached.

minimum required BMC bound for the discovery of fault is $M + (C \times L) + Q$. ■

Fig. 5.6 shows the result of the UB-SAT attack on the 011 detector of Fig. 5.3. With a 2-bit counter, according to Lemma 1, the BMC min-bound of discovering faults is $2^2 = 4$. As expected, in each cycle at least one fault is discovered, while the deep fault is discovered at the expected boundary of 4.

5.4.3 Preventing the Removal of the Tracer

A simple mechanism to implement the DF-tracer is using a counter. However, counters can be easily identified by structural analysis [78] as they have well-defined structure and are loosely connected to the rest of the circuit. Note that, for implementing the DF, the exact counter behavior is not needed; We only need a tracer circuit for tracking cycles or events. Hence, we can use an event tracking LFSR (i.e. where LFSR state is updated based on a state transition) or a function-modified (with different encoding) counter to implement the tracer. The repetition period of LFSR (number of non-repeated LSFR state values) would serve as the depth that the fault could be delayed. In addition, to prevent

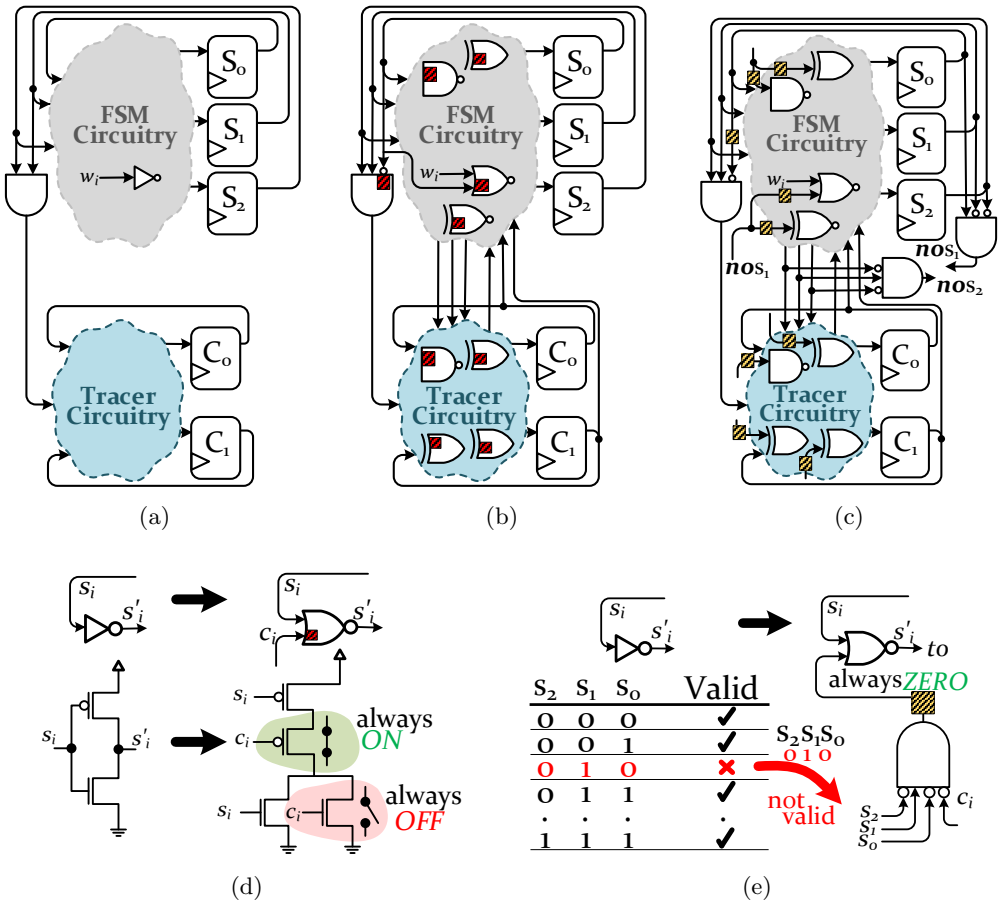


Figure 5.7: Camouflaged (Covert) gates and non-occurring states could be used for merging the two strongly-connected graphs of state and counter FFs. a) DF circuit, b) DF circuit hidden by covert gates, c) DF circuit hidden by building dummy logic from non-occurring signal combinations, d) a covert gate implementation with dummy input, e) non-occurring signal used for implementing dummy logic.

the attacker from structural analysis using asynchronous signals, the enable/rest signal of the tracer circuit should not be separated from the rest of the circuit. furthermore, the tracer could be designed to exhibit a pseudo-counter behaviour such that the update of the tracer's different register state values relies on both the existing tracer register values and other registers selected from the sequential circuit or FSM.

With the changes discussed earlier, the inserted tracer (modified-counter or LFSR) can not be functionally identified. However, it is still prone to detection by structural analysis of the data flow graph. As Fig. 5.7.(a) shows, the tracer is still loosely connected to the

rest of circuit. To resolve this issue, the data flow graph of the obfuscated circuit should be modified such that tracer circuit can not be easily isolated. In other words, the tracer's registers' values should be also in the input logic cone of the other FFs. However, this should not affect the functionality of the tracer. One solution for modifying the data flow graph, as illustrated in Fig. 5.7.(b) is through the usage of Covert Gates [79]. These gates have dummy inputs, connected to always on or always off transistors, that don't affect the gates' function. In practice, the gates in the logic cone of the state registers can be replaced by Covert gates, and the tracer registers' output can be connected to the dummy inputs of the covert gates. With this change, without modifying the circuit functionality, the tracer circuit will be strongly connected to the rest of the circuit when the data flow graph is extracted. The covert gates can be also used to bring additional dummy inputs from the state machine or sequential circuit to tracer without affecting its functionality. The problem with this method is that it can only protect the design against adversaries attempting to fully reverse engineer an existing ASIC, and it does not protect the IP against an untrusted manufacturing facility. In fact, the manufacturing facility has access to the layout represented via the GDSII file. Hence, the Covert gates are not hidden from the foundry.

To protect against adversarial reverse engineering at untrusted foundries, one can also utilize non-occurring signal combinations in the netlist for building dummy connections to/from the tracer circuit. As Fig. 5.7 shows, the non-occurring signal combinations can be found using a QBF solver [5] and utilized to design an always-zero (or one) signal combined from counter FFs and signals in input cone of other state FFs.

5.5 Experimental Results

We have implemented the UB-SAT attack using Yices SMT solver by creating the combinational equivalent circuit and unrolling it for finding DISes and checking UC and CE terminations. For UMC termination, we have used SuperProve from Berkeley ABC package [80]. The experiments were performed on an Intel Core i5 with 64GB RAM.

Table 5.2: Experimental results for deep faults.

Circuit	Circuit Info				Deep Faults											
	URS count from [81]				DF3			DF4			DF5			DF7		
	FF	PI	PO	URS	D/S	Time	Term	D/S	Time	Term	D/S	Time	Term	D/S	Time	Term
s344	15	9	11	9,536	29/8	39	UC	56/16	139	UC	119/32	2849	UC	-	TO	-
s382	21	3	6	2,073,412	17/48	9393	UMC	-	TO	-	-	TO	-	-	TO	-
s386	6	7	7	51	8/8	9	UC	29/16	45	UC	70/32	482	UC	-	TO	-
s526	21	3	6	1,695,692	27/16	463	UMC	49/32	1580	UMC	92/64	4684	UMC	-	TO	-
s713	19	35	23	517,625	15/8	28	UC	53/16	317	UC	-	TO	-	-	TO	-
s832	5	18	19	7	32/16	86	UC	44/16	148	UC	99/32	2549	UC	-	TO	-
s838	32	34	1	$> 2^{30}$	20/72	1096	UC	43/80	2194	UC	-	TO	-	-	TO	-
s1196	18	14	14	259,492	27/8	77	UC	64/16	377	UC	135/32	3146	UC	-	TO	-
s1423	74	17	5	$> 2^{72}$	8/8	26	UC	49/16	2924	UC	-	TO	-	-	TO	-
s1494	6	8	19	16	21/8	46	UC	78/16	369	UC	187/32	4333	UC	-	TO	-
s5378	179	35	49	$> 2^{176}$	42/8	1653	UC	-	TO	-	-	TO	-	-	TO	-
s9234	211	36	39	$> 2^{227}$	-	TO	-	-	TO	-	-	TO	-	-	TO	-
s38584	1426	38	304	$> 2^{1449}$	8/8	1293	UC	-	TO	-	-	TO	-	-	TO	-

Table 5.2 captures the results of attacking circuits, which are obfuscated using deep faults with varying counter widths (3, 4, 5, and 7 bits). The first few columns of the table describe the characteristics of these benchmarks in terms of number of flip-flops (FF), number of primary I/O (PI and PO), and number of unreachable states (URS) according to [81]. In this table DFw represent a DF obfuscation, constructed using a counter of width w. For each obfuscated circuit, number of discovered DISes, and the number of inputs in the last DIS (its length) is reported as D/S. The maximum attack time is set to eight hours. Attacks that take longer are reported as *TO*. From this table, following observations are made: 1) the number of discovered DISes grow exponentially with respect to the size of the counter. This is consistent with the Lemma 1: at each cycle we can produce at least 1 DIS until the protected pattern (which in this case is encoded using the highest value of the counter) is reached. Hence, we should at least have 2^w DISes. 2) The size of the largest input sequence (S) in which the deep fault is discovered is $N \times 2^w$ (N being an integer). This is consistent with Lemma 1, where the protected pattern could be potentially (but not necessarily) observed at every N$\times 2^w$ cycles; 3) the runtime of the attack increases exponentially as the depth of DF tracer circuit (counter) increases; and 4) when the circuit

Table 5.3: Experimental results for shallow state duality (SSD) and combined (DFSSD) methods.

Circuit	Shallow State Duality			DFSSD					
	SSD			SSD + DF5			SSD + DF7		
	D/S	Time	Term	D/S	Time	Term	D/S	Time	Term
s344	0	9	UMC	115/32	5231	UMC	-	TO	-
s382	0	5	UMC	-	TO	-	-	TO	-
s386	0	6	UMC	65/32	1232	UMC	-	TO	-
s526	0	5	UMC	90/64	4676	UMC	-	TO	-
s713	0	223	UMC	-	TO	-	-	TO	-
s832	0	12	UMC	-	TO	-	-	TO	-
s838	-	TO	-	-	TO	-	-	TO	-
s1196	0	12	UMC	-	TO	-	-	TO	-
s1423	-	TO	-	-	TO	-	-	TO	-
s1494	0	15	UMC	-	TO	-	-	TO	-
s5378	0	65	UMC	-	TO	-	-	TO	-
s9234	0	3065	UMC	-	TO	-	-	TO	-
s38584	-	TO	-	-	TO	-	-	TO	-

is solely protected by DF, the UC termination is the most reoccurring termination strategy.

Table 5.3 also captures the results of attacking the ISCAS'89 benchmarks when encoded using duplicated states (SSD) and protected using a combination of both techniques (DFSSD). The SSD column of Table 5.3 captures the result of UB-SAT attack against circuits protected only by Shallow State Duality. As expected, when UB-SAT is deployed against a SSD encoded circuit, the UC And CE termination strategies become useless. As reported, for all SSD-encoded benchmarks, either the attack is terminated by UMC or prematurely terminated for lack of memory resources.

The last two columns of Table 5.3, capture the impact of combining the DF and SSD (DFSSD) which is the main solution proposed in this chapter. The DFSSD combines the best feature of the two solutions. The SSD prevents early UC and CE termination, while the DF pushes the faults down into deep states, resulting in an exponential increase in the number of required DISes and the attack time with respect to the counter size. Note that by preventing the UC and CE terminations, and by forcing the attack to UMC termination check in every iteration, the SSD+DF5 has considerably larger runtime compared to DF5.

5.6 Conclusion

In this chapter, we proposed DFSSD, an obfuscation solution for FSM and sequential circuits with restricted (locked) access to the scan chain. The DFSSD deployed two mechanisms, specifically designed to resist against BMC-based attacks such as UB-SAT: 1) it uses shallow state duality to prevent early termination of such attacks by invalidating the unique completion and combinational equivalence checks, forcing the attack to rely on exhaustive and time-consuming UMC for assessing the attack's termination condition; 2) it injects fault into deep and hard to reach (by a BMC) states. The DFSSD allows the designer to precisely control the depth of the fault at design time using a low overhead circuit solution and make the attack time unreasonably long.

Chapter 6: RANE Framework

Over the recent years, many de-obfuscation attacks subvert the trustworthiness of logic locking. In particular, *Boolean* satisfiability (SAT)-based attack and its derivatives show how well-formulated attack models could break the existing logic locking techniques [3] with fast convergence. Depending on the assumptions of the threat models used in de-obfuscation attacks, these attacks may be able to target the combinational parts of the circuit, separately through the design-for-test (DFT) infrastructure (i.e., scan chain pins) [11–14, 17, 19, 20, 25, 61], or they have to target the sequential circuit as a whole (through PI/PO) [46–48, 82]. Nevertheless, many of these attacks have been developed over the basic capabilities of different solvers that have lots of limitations, making them less practical on real applications. For instance, almost all open-source de-obfuscation attack tools [11,14,47, 83] receive the locked circuit in `Bench` or *translated and remapped* `Verilog` format, converted by open-source synthesis tools like ABC and Yosys [84, 85]. So, many real applications with complex macros require a heavy library-dependent conversion and simplification before exploiting these attacks. In many cases, these attacks fail to evaluate the robustness of existing logic locking techniques in such scenarios.

In this chapter, we introduce a unified Reverse Assessment of Netlist Encryption (**RANE**), as an open-source CAD-based toolbox for evaluating the security of different logic locking techniques. Unlike the existing open-source de-obfuscation tools on logic locking, RANE has been developed based on a unified framework with a unique interface to exploit the capability/scalability of formal verification tools for different stages of the attack. The RANE framework allows the users to exploit any formal verification tool, either open-source or commercial, such as Cadence JasperGold, Synopsys Formality, SymbiYosys, etc. The establishment of such formal verification tools allows RANE to support circuits written,

elaborated, synthesized in standard HDL, like `Verilog`, with no limitation on the technology library used for the design/implementation. The flexibility and deployability of formal verification tools also allow us to model different threat models in RANE. With more concentration on sequential-based attacks on logic locking, in this chapter, we will evaluate the capability/performance of RANE on *FOUR* different case studies: (1) an oracle-less attack on key-less (implicit key) sequence-based logic locking (HARPOON), (2) An oracle-guided attack on HARPOON, (3) An oracle-guided attack on sequential logic locking (scan is BLOCKED), and (4) An oracle-guided attack on combinational logic locking (scan is OPEN). We also demonstrate how the RANE framework could be extended to support any form of de-obfuscation attack that relied on formal verification tools.

6.1 Proposed Attack Framework: RANE

The existing attack frameworks on logic locking with available source codes are developed by exploiting pre-compiled Binary and modulo theory solvers that accept the netlists in `Bench`, which is a minimal language for the description of hardware [11, 14, 47]. This requirement introduces a burden for modeling and assessing logic locking as all complex structures have to be re-synthesized and expressed in the simplest logic structures compatible by the solver. For instance, a 2-bit Full Adder (FA) in acceptable `Bench` format is depicted in Fig. 6.1 and its corresponding `Verilog` format demonstrated in Fig. 6.2. As illustrated, the FAs, although available in the standard cell library, cannot be interpreted by the solver's native macros and have to be translated to basic logic gates/macros. Problems become more complicated when complex standard syntax declarations such as vectors, inouts, and aliasing are used. More precisely, the limitations faced during the modeling of a complex netlist in simplified `Bench` format include (1) limited availability of available macros with inherent support only for the description of basic gates, (2) static syntax declaration for available macros with no possibility of extension, (3) requirement for having/writing a dedicated parser for such format that is library- and language-dependent, (4) incompatibility

```
INPUT(a[0])
INPUT(a[1])
INPUT(b[0])
INPUT(b[1])
INPUT(cin)

OUTPUT(s[0])
OUTPUT(s[1])

OUTPUT(cout)

s[0] = XOR(a[0], b[0], cin)
cout_01 = AND(a[0], b[0])
cout_02 = AND(a[0], cin)
cout_03 = AND(b[0], cin)
cout_0 = OR(cout_01, cout_02, cout_03)
s[1] = XOR(a[1], b[1], cout_0)
cout_11 = AND(a[1], b[1])
cout_12 = AND(a[1], cout_0)
cout_13 = AND(b[1], cout_0)
cout = OR(cout_11, cout_12, cout_13)
```

Figure 6.1: Acceptable `BENCH` format in existing and available SAT and sequential SAT attacks' source codes [11, 47] for a 2-bit FA.

```
`timescale 1ns / 1ps
//////////////////////
// Lib.v
//////////////////////
module FA_1bit (
  input a,
  input b,
  input cin,
  output s,
  output cout );

  assign {cout,s} = a + b
                    + cin;
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////
// Top.v
//////////////////////
module FA_2bit(
    input [1:0] a, b,
    input cin,
    output [1:0] sum,
    output carry );
  FA_1bit s0(a[0], b[0], cin,
             sum[0], cr0 ) );
  FA_1bit s1(a[1], b[1], cr0,
             sum[1], carry ) );
endmodule
```

Figure 6.2: Standard `Verilog` format acceptable by RANE for a 2-bit FA.

with many standard syntax declarations, like vector, inout, aliasing, etc. The complexity involved in building a translator and having to model and account for these complexities significantly raises the bar for the application of the existing attacks.

To overcome this shortcoming, we propose RANE as a CAD-based toolbox for evaluating the security of logic locking that applies to a much broader set of applications and circuits. By exploiting open-source toolkits for design analysis and code generation of RTL designs written in standard HDLs, RANE supports parsing and analyzing circuits written, elaborated, or synthesized in standard HDL, such as `Verilog`. This also allows us to use formal verification tools for the de-obfuscation modeling instead of using pre-compiled solvers

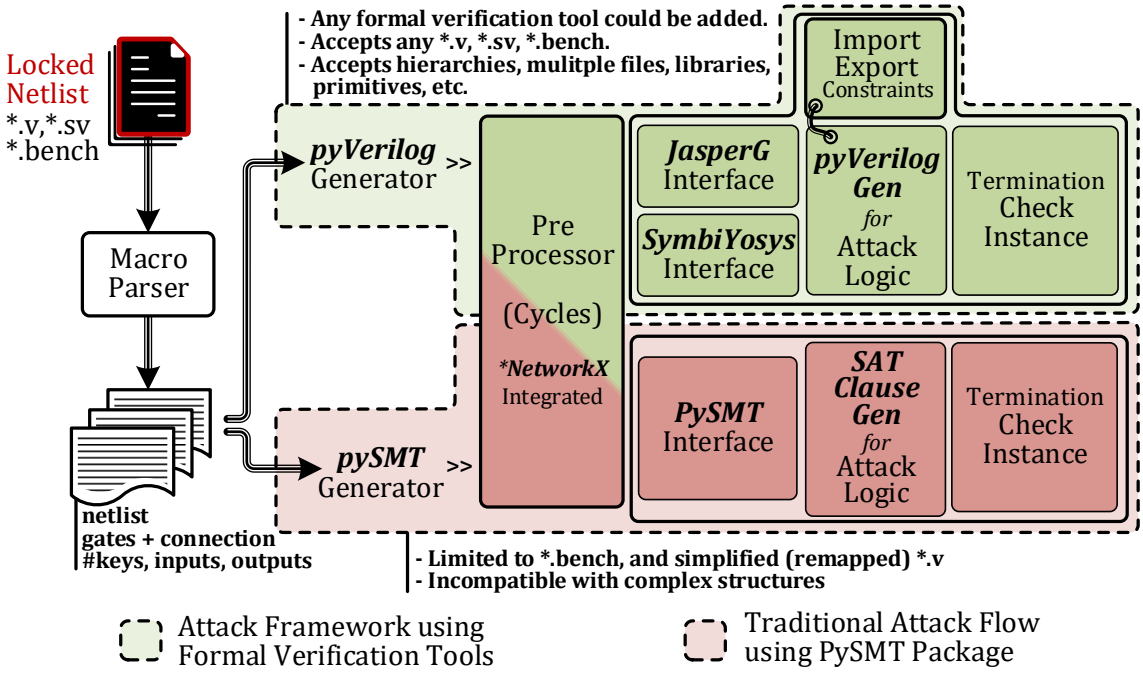
as the core of de-obfuscation. The usage of formal verification tools allows RANE to be extended based on the inherent features of these formal tools. Besides, the RANE applicability is seamlessly improved as formal tools are revised and upgraded to parse and interact with new libraries and complex macros without having to do any additional translation or modeling.

6.1.1 RANE Framework

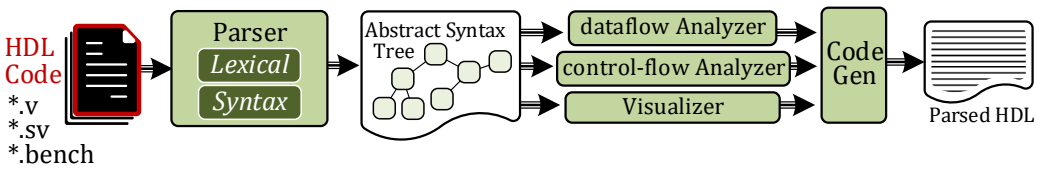
Fig. 6.3(a) shows the overview of the RANE framework. In the RANE framework, we provide two different solutions: (1) a formal-based interface through *Pyverilog* generator, and (2) a pre-compiled static-model tool using *PySMT* generator. In the first solution, we use Pyverilog [86] as the open-source HDL analyzer for code parsing, static analysis, and code translation. Pyverilog framework is captured in Fig. 6.3(b). The parser, dataflow analyzer, control-flow analyzer, and `Verilog` code generator are the four major features in Pyverilog. Pyverilog also provides a dataflow and control-flow graph visualizer for interpreting the hardware. In RANE, we implement and integrate different interfaces to support different verification and solver tools. As demonstrated in Fig. 6.3(a), by getting the benefit of Pyverilog, Cadence JasperGold and SymbiYosys are integrated as the formal tools. Also, using Pyverilog, any model like miter circuit, equivalency check, etc., could be generated using behavioral `Verilog` code, making the model generation for de-obfuscation much easier. RANE also supports features like exporting/importing constraints, automated cycle pre-processing, and `Verilog`-based attack model generation.

6.1.2 RANE Application

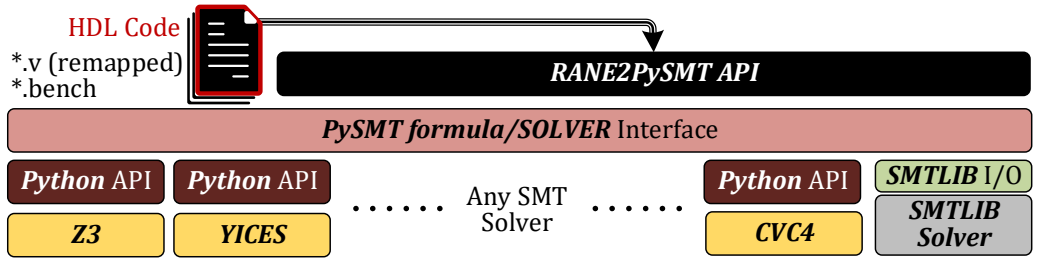
For the second solution, we implement and integrate an interface for embedding PySMT into the RANE framework. PySMT is a solver-agnostic library for fast prototyping of satisfiability modulo theory (SMT)-based algorithms. As demonstrated in Fig. 6.3(c), by using different APIs, PySMT provides the possibility of invoking well-known SMT solvers, such as Z3 [87], Yices[88], and Boolector[89]. By integrating the PySMT framework, similar



(a) RANE Framework



(b) PyVerilog Framework in RANE



(c) PySMT Framework in RANE

Figure 6.3: RANE overall framework.

to the existing de-obfuscation attack tools, it could be engaged on `Bench` and remapped `Verilog` files over pre-compiled solvers.

Using this framework, RANE can model different threat models on logic locking and formulate various attacks with much less effort than the existing de-obfuscation attack tools. In the following section, we will evaluate the application of RANE on *FOUR* different case studies: (1) **oracle-less attack on HARPOON**, in which HARPOON is the key-less FSM logic locking, (2) **oracle-guided attack on HARPOON**, (3) **oracle-guided attack on sequential logic locking**, in which random-based logic locking is engaged, and as an assumption of the threat model, the scan chain accessibility is *BLOCKED*, and (4) **oracle-guided attack on combinational logic locking**, in which random-based logic locking is engaged, and the scan chain accessibility is *OPEN*.

Case Study 1: Oracle-less Attack on HARPOON

In this case study, we assume that the adversary might have only a single working copy of the chip. They can first apply a sequence of input patterns, and by observing the outputs, they can build a database of such I/O pairs. Alternatively, the I/O pair also could be obtained by the adversary at the foundry from the pre-generated functional test patterns or post-layout verification test[1]. Then, by reverse-engineering the chip or having access to the layout at the foundry, the netlist could be extracted. We refer to this attack model as *bronze model*. The adversary in this model does not have access to the scan chain. By using this threat model, the de-obfuscation attack on HARPOON could be accomplished in two main steps: (1) Finding the initial value of FFs (*init state*), e.g., S_0 in Fig. 2.1(b), such that if the init state initializes the circuit, it would produce the same output if the input patterns are applied to the oracle; (2) Formulating the formal verification problem to find the correct sequence of input patterns, allowing us to reach the previously found init state, referred to as *unlocking sequence*. For example, in Fig. 2.1(b), $\text{pi}_1 \rightarrow \text{pi}_7$ is the unlocking sequence to reach the init state.

[1]Since HARPOON is a key-less logic locking, functional test patterns or post-layout verification tests could be used to build and extend the database of I/O pair.

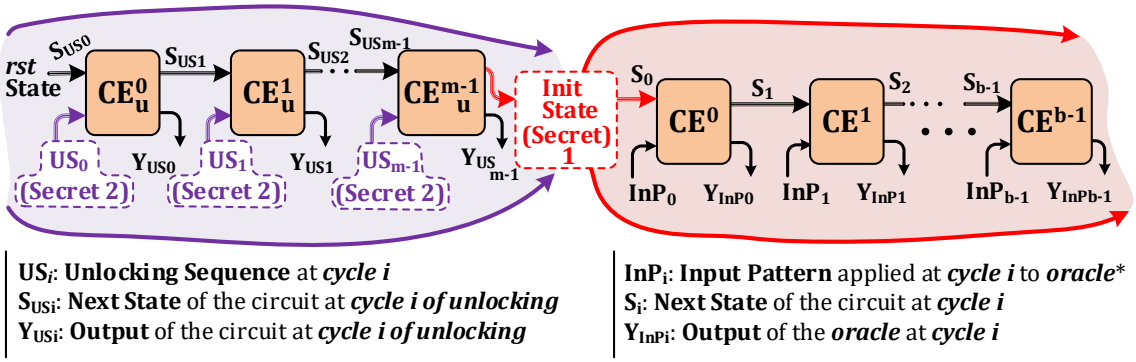


Figure 6.4: Oracle-Less Attack Model on HARPOON.

Fig. 6.4 shows the oracle-less attack model on HARPOON based on the bronze threat model. For the first step of the attack, i.e., formulating *secret 1* of Fig. 6.4, the init state is considered as the key. Then, by applying the I/O sequences from the pre-built database to the (unrolled) combinational equivalent (CE) netlist[2], i.e. $InP_0 \rightarrow (CE^0)$, $InP_1 \rightarrow (CE^1)$, etc., constraining that the output values (Y_{InP0}, Y_{InP1}, etc.) must match with oracle outputs, the init state could be found by formal tool. After formulating the *secret 1*, then the second step will be formulated for finding the unlocking sequence, i.e. finding *secret 2* of Fig. 6.4. In this step, unlocking sequence is the key, i.e., $US_{0:m-1}$, and with constraining that state of the circuit that must reach the valid init state, the formal tool integrated with the RANE framework could find the unlocking sequence. Algorithm 8 also illustrates the flow of this case study in the RANE framework. It consists of three steps: (1) formulating of *secret 1*, (2) formulating of *secret 2*, and (3) invoking the formal tool for finding both secrets. Note that all unrolling steps are implicitly done by the formal tool.

It is worth mentioning that if the adversary aims to *ONLY* reverse engineer the chip, formulating and performing part 1 and part 3 of the Algorithm 8 would be enough. After finding the init state, the adversary could then insert their own scan chain into the reverse-engineered netlist to provide the possibility of loading the proper init state to the

[2]all CE^is/CE^i_us are the same each represents the combinational equivalent of the locked netlist. Each CE^is/CE^i_us is implicitly generated for one cycle by the formal tool.

Algorithm 8 Oracle-less Attack Model on HARPOON using RANE

——— Formulating of Secret 1 (init state) ———

1: Get an I/O sequence $(\{InP_0, Y_{InP0}\}, ..., \{InP_{b-1}, Y_{InPb-1}\})$ from $C_{BlackBox}$;

2: $Model \leftarrow CE(InP_0, \hat{S}_{init}, Y_{InP0}, \hat{S}_1) \wedge_{i=1}^{|I/O|} CE(InP_i, \hat{S}_i, Y_{InPi}, \hat{S}_{i+1})$;

——— Formulating of Secret 2 (unlocking sequence) ———

3: $Model \wedge = \ CE_u^0(US_0, S_{rst}, Y_{US0}, S_{US1})$;

4: $i \leftarrow 1$;

——— Invoking the Formal Tool: Finding Secret 1, 2 ———

5: **while** $Formal(Model \wedge (S_{USi} = \hat{S}_{init})) \rightarrow Fail$ **do**

6: $\quad Model \wedge = \ CE_u^i(US_i, S_{USi}, Y_i, S_{USi+1})$;

7: $\quad i \leftarrow i + 1$;

8: **return** $Formal(Model \wedge (S_{USi} = \hat{S}_{init}))$ ▷ {init state, unlocking sequence}

FFs, bypassing the need to go through the second step for finding the unlocking sequence. Furthermore, assuming that for the pre-built I/O pairs, there exists multiple init states satisfying the formal mode, and each init state could be reached via a unique unlocking sequence, formulating and solving both steps together will automatically constrain finding the valid init state whose unlocking sequence is the shortest one. This case study in RANE is the first of its kind in de-obfuscation attacks that target key-less logic locking techniques like HARPOON with no need for oracle. In our experimental results, we show how the success rate of this model depends on the length/size of I/O pairs available in the pre-built database and the size of the unlocking sequence (numbers of obfuscation/authentication FSMs). Note that since the adversary's capability is limited to only using the available and pre-built sequence of I/O pairs, the existing combinational/sequential SAT can not be used for this attack. This is because the solver can no longer constrain the input patterns freely. But this attack could be easily modeled and carried by RANE.

Case Study 2: Oracle-guided Attack on HARPOON

In this case study, we target the same logic locking technique evaluated in case study 1, i.e., HARPOON. We assume that the adversary has access to the reverse-engineered netlist and the functional chip (oracle). Other assumptions are the same as the threat model of case study 1. Fig. 6.5 illustrates this attack model on HARPOON. Similar to case study 1, it could be done in two steps, i.e., finding init state (secret 1) and finding the unlocking

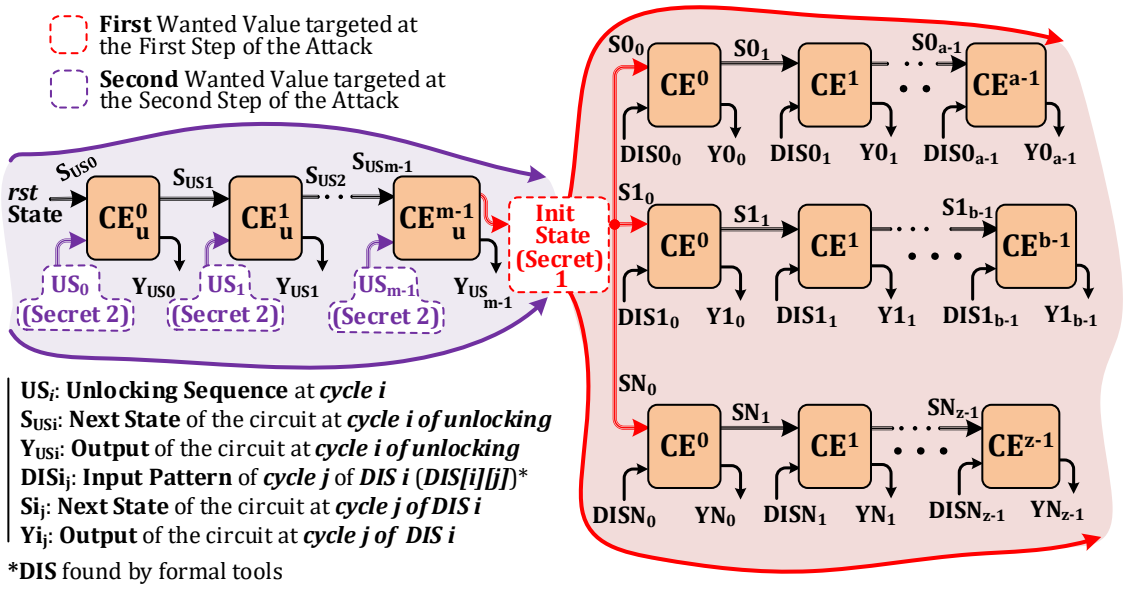


Figure 6.5: Oracle-Guided Attack Model on HARPOON.

sequence (secret 2). However, these two steps will be accomplished in sequence. Regarding the first step (secret 1), since the oracle is available for the adversary, the generation of the sequence for finding the init state will be done by the formal tool. Similarly, all unrolling operations will be done implicitly in this case study. The availability of the oracle also allows us to expand the number of sequences from one to many ($InP1_{1:a}$, $InP2_{1:b}$, ..., $InPN_{1:z}$). Unlike the attack model in case study 1, since the adversary's capability is not limited to a fixed I/O pair database, this model's success rate does not depend on the length/size of the sequences. Algorithm 9 depicts the flow of case study 2. Note that unlike case study 1 that finds both secrets at once, in this case, multiple formal tool invocation will be accomplished for finding secret 1 followed by finding secret 2.

Since formal tool is employed for finding the DISes, the termination condition would be adopted from conventional sequential SAT attack [46]: (1) **Unique Completion (UC):** This criterion checks for the uniqueness of the secret. If there is only a single secret that satisfying the defined constraints, the attack is terminated. (2) **Combinational Equivalence (CE):** If there is more than one secret that agrees with the constraints, the attack

Algorithm 9 Oracle-guided Attack Model on HARPOON using RANE

——— Finding Secret 1 (init state) ———

$Model \leftarrow C_{seq}(X, S_{init1}, Y_1) \wedge C_{seq}(X, S_{init2}, Y_2)$;
while !**UC**$(Model)$ $\wedge$!**CE**$(Model)$ $\wedge$!**UMC**$(Model)$ **do**
    $DIS_i \leftarrow Formal(Model \wedge (Y_1 \neq Y_2))$;
    $Y_i \leftarrow C_{BlackBox}(DIS_i)$;
    $Model \wedge = \ C_{seq}(DIS_i, S_{init1}, Y_i) \wedge C_{seq}(DIS_i, S_{init2}, Y_i)$;

——— Finding Secret 2 (unlocking sequence) ———

$Model \wedge = \ CE^0_u(US_0, S_{rst}, Y_{US0}, S_{US1})$;
$i \leftarrow 1$;
while $Formal(Model \wedge (S_{USi} = \hat{S}_{init})) \rightarrow Fail$ **do**
    $Model \wedge = \ CE^i_u(US_i, S_{USi}, Y_i, S_{USi+1})$;
    $i \leftarrow i + 1$;
return $Formal(Model \wedge (S_{USi} = \hat{S}_{init}))$ ▷ {init state, unlocking sequence}

checks the combinational equivalency of the remaining secrets. In this step, the input/output of FFs is considered as pseudo primary outputs/inputs allowing the attacker to treat the circuit as combinational. The resulting circuit is subjected to an SAT attack. If the SAT solver fails to find a different output or next state for two different secrets, it concludes that all remaining secrets are correct, and the attack terminates. (3) **Unbounded Model Check (UMC):** If UC and CE fail, the attack checks the existence of a satisfying assignment for the remaining secrets using an unbounded model checker. This is an exhaustive search with no limitation on bound.

Case Study 3: Oracle-guided Attack on Sequential Logic Locking

RANE attack could also be used for breaking conventional key-based logic locking solutions, such as random logic locking (RLL) [30], or strong logic locking (SLL) [31] applied on the sequential circuit, where access to the scan chain is *BLOCKED*. In this case study, we target to model the conventional SAT-based sequential de-obfuscation attack, shown in Fig. 2.3(b). The adversary has access to the PI/PO of the oracle and reverse-engineered *locked* netlist. Unlike the existing sequential de-obfuscation attacks [46, 47] that handle the unrolling explicitly by the framework, RANE could accomplish it both implicitly handled by the formal verification tool encapsulated in the RANE framework or explicitly by the defined attack model. Also, supporting `Verilog` in the RANE framework allows us to get

Algorithm 10 Oracle-guided Attack Model on Logic Locking with *BLOCKED* scan chain (Sequential Logic Locking) using RANE

1: $Model = C_{seq}(X, K_1, Y_1) \wedge C_{seq}(X, K_2, Y_2)$;
2: **while** !**UC**$(Model)$ $\wedge$!**CE**$(Model)$ $\wedge$!**UMC**$(Model)$ **do**
3: $\quad X_{dis} \leftarrow Formal(Model \wedge (Y_1 \neq Y_2))$;
4: $\quad Y_f \leftarrow C_{BlackBox}(X_{dis})$;
5: $\quad Model \wedge = \ C_{seq}(X_{dis}, K_1, Y_f) \wedge C_{seq}(X_{dis}, K_2, Y_f)$;
6: **return** $Formal(Model)$ ▷ Return Correct Key

the benefit of behavioral `Verilog` helping to build any model with much less effort.

The support of implicit unrolling provides the RANE framework to use any of the available either open-source or commercial verification tools. Hence, RANE can get the benefit of the scalability, stability, and adaptability of these tools to handle a much richer set of input formats, handle a wider range of gates[3]. This is the main aim of the RANE framework that be easily adaptable in any flow, without the need for input format translation, remapping, Decoding, re-synthesis. Algorithm 10 shows the flow of case study 3 in the RANE framework with implicit unrolling. As demonstrated, the attack formulation is first initiated using the miter circuit (XORed double-circuit). Then, per each iteration, the formal tool looks for a DIS and two keys that produce different outputs for that DIS. In the next iterations, the previously found DISes must match with the oracle, and the attack model termination conditions will be checked when no more DIS is found. As shown, unrolling operations for finding DISes are not formulated in the model (implicit unrolling), and it will be handled automatically by the formal tool.

Case Study 4: Oracle-guided Attack on Combinational Logic Locking

In this case study, RANE emulates the most well known SAT-based attack on logic locking proposed by Subramanyan *et al.* [11], which is oracle-guided on logic lockings with *OPEN*

[3]Formal tools could support any type of macros defined in the standard cell library, as opposed to very limited basic gates available in `Bench` format (used in the existing sequential de-obfuscation attack, i.e., KC2).

Algorithm 11 Oracle-guided Attack Model on Logic Locking with *OPEN* scan chain (Combinational Logic Locking) using RANE

1: $Model = C_{comb\_lock}(X, K_1, Y_1) \wedge C_{comb\_lock}(X, K_2, Y_2)$;
2: **while** $Formal(Model \wedge (Y_1 \neq Y_2))$ **do**
3: $\quad X_{dip} \leftarrow Formal(Model)$;
4: $\quad Y_f \leftarrow C_{BlackBox}(X_{dip})$;
5: $\quad Model \wedge = \ C_{comb\_lock}(X_{dip}, K_1, Y_f) \wedge C_{comb\_lock}(X_{dip}, K_2, Y_f)$;
6: **return** $Formal(Model)$ ▷ Return Correct Key

scan chain access, referred to as SAT-based combinational de-obfuscation attack. As demonstrated in Algorithm 11, in the SAT-based combinational de-obfuscation attack, a (distinguishing) miter circuit needs to be built as $miter \equiv C_{comb\_lock}(X, K_1) \neq C_{comb\_lock}(X, K_2)$ for any arbitrary locked combinational logic C_{comb_lock}. Based on the miter circuit, the formal tool will be invoked and will return a DIP that produces different outputs for two different keys. Then, this DIP is queried on the oracle, $C_{BlackBox}$, $eval \leftarrow C_{BlackBox}(X_{dip})$ and the I/O-constraint for the equivalency check, $C_{comb\_lock}(X_{dip}, K_1) = C_{comb\_lock}(X_{dip}, K_2) = eval$ will be added as a new constraint to the formal tool, and after this update, the *miter* circuit would be solved again. When the $miter + constraints$ problem has no satisfying assignment (no more DIP), it could identify the correct key.

From the formal tool perspective in RANE, the formulation of both key-based oracle-guided combinational and sequential de-obfuscation attacks are very similar. The only difference is that for the attack model on the sequential circuits, the formal tool looks for DIS (with implicit unrolling), but in the model on the combinational circuit, finding DIP is the main objective of the formal tool.

6.2 Experimental Results

With exploiting packages like PySMT [90] and Pyverilog [86], the proposed RANE framework has been implemented in `Python3`. The current version of the RANE framework, available in [91], has been built over different formal verification tools configurable by the

Table 6.1: Description of ISCAS-85/89* circuits.

Circuit	#Gates	#PIs	#POs	Circuit	#Gates	#PIs	#POs	Circuit	#Gates	#PIs	#POs
c432	160	36	7	c1355	546	41	32	c3540	1,669	50	22
c499	202	41	32	c1908	880	33	25	c5315	2,307	178	123
c880	383	60	26	c2670	1,269	233	140	c7552	3,513	207	108
Circuit	#FFs	#PIs	#POs	Circuit	#FFs	#PIs	#POs	Circuit	#FFs	#PIs	#POs
s344	15	9	11	s832	5	18	19	s5378	179	35	49
s382	21	3	6	s838	32	34	1	s13207	638	62	152
s386	6	7	7	s1196	18	14	14	s15850	534	77	150
s526	21	3	6	s1423	74	17	5	s35932	1,728	35	320
s713	19	35	23	s1494	6	8	19	s38584	1,426	38	304

*s9234 *MUST* be ignored since it has some FFs that have no path to POs.

users, including Cadence JasperGold as the commercial formal verification tool and SymbiYosys as a formal open-source tool. The formal tools are responsible for major operations of attack modeling, such as unrolling, building miter, finding sequences, DIPs, and DISes. In this chapter, the experiments are accomplished using the open-source SymbiYosys formal verification engine[4]. We evaluate and verify the feasibility/performance of the RANE framework, based on all *FOUR* case studies previously discussed in Section 6.1.2, on a set of ISCAS-{85/89} benchmark circuits, as listed in Table 6.1. For sequential-based experiments, i.e., case studies 1, 2, and 3, since the circuits have a sequential depth of fewer than 100 cycles, with skipping UMC check, the boundary/depth is set to 100 cycles. The integration of PySMT and SymbiYosys allows RANE to get the benefit of different solvers. In the experiments, and based on our observation to get the most benefit, we use *Yices* for case studies 1 and 2, and the best performance achieved by *Boolector, MathSAT,* or *Yices* for case studies 3 and 4. All experiments are carried on ARGO cluster computing [92] as a computing cluster equipped with Intel Xeon E5-2670, with 16 core CPUs and 512GB of RAM.

Table 6.2 demonstrates the performance of the RANE framework when it is configured

[4]To facilitate re-producing the results by the community and remove the dependency on commercial tools, the results are generated on available open-source tools. Our preliminary investigation shows that the results could improve significantly (in terms of runtime and memory) when a commercial tool, such as Cadence JasperGold, is configured as the utilized formal method tool.

Table 6.2: RANE performance in Case Study 1 - Oracle-less on HARPOON.

Circuit	{3, 18}*		{5, 30}		{10, 60}		{20, 120}	
	time	#I/O	time	#I/O	time	#I/O	time	#I/O
s344	4	20	4	30	5	60	22	120
s382	1	20	-	-	4	60	172	120
s386	4	20	5	30	7	60	22	120
s526	-+	-	-	-	-	-	-	-
s713	5	20	6	30	8	60	23	120
s832	5	20	5	30	-	-	-	-
s838	-	-	3	30	6	60	22	120
s1169	6	50	20	230	8	60	-	-
s1423	4	30	10	110	11	100	65	120
s1494	224	20	226	30	249	60	1,468	120
s5378	6	20	7	30	15	60	76	120
s13207	15	20	22	30	56	60	246	120
s15850	-	-	18	30	40	60	-	-
s35932	-	-	53	30	111	60	-	-
s38417	72	50	-	-	-	-	-	-
s38584	49	20	108	60	-	-	1,117	120

*{number of obfuscation/authentication FSMs, The length of unlocking sequence}
time: in Seconds #I/O: number of input/output patterns
+Failed to find the correct init state by using 3,000 I/O pairs.

for case study 1, in which the circuits are locked with HARPOON [38]. When the circuit's state is in obfuscation/authentication modes, random POs are selected to be corrupted. We also used random input patterns to build the database of I/O pairs for this case study. In this experiment, the number of authentication/obfuscation FSMs (unlocking sequence size) is swept. As shown, for different circuits, with a different number of obfuscation FSMs (different unlocking sequence sizes), the threat model defined in case study 1 can retrieve the secrets with a small number of I/Os.

Since the output of this model is based on a limited set of pre-built I/O pairs, this threat model cannot guarantee the uniqueness of the init state (secret 1) generated by the framework. However, increasing the size of I/O pairs, or applying different sets of random I/O pairs, results in restricting the different {secret 1, secret 2} possibilities that match with the oracle pre-generated I/O pairs. In this experiment, we limit the size of the pre-built I/O database to 3,000 cycles and by using this size, our observation shows that, on average,

Table 6.3: RANE performance in Case Study 2 - Oracle-guided on HARPOON.

Circuit	{3, 18}*		{5, 30}		{10, 60}		{20, 120}	
	time	size	time	size	time	size	time	size
s344	3+1	1/2	5+2	3/4	11+9	4/4	20+98	7/4
s382	8,484+107	107/44	8,101+218	93/44	4,449+398	65/67	to	to
s386	3+1	3/3	12+3	11/4	30+9	12/4	100+195	15/4
s526	1,644+28	59/44	17,749+275	129/44	2,677+275	48/50	6,4441+to	48/49
s713	102+3	7/8	144+6	9/10	496+17	9/10	1,322+619	43/6
s832	10+2	7/10	6+2	4/5	59+23	17/11	34+113	12/13
s838	3,175+82	45/66	293+29	43/4	701+140	63/6	4,319+3,742	112/18
s1196	67+10	13/15	45+4	10/11	79+42	14/15	79+148	11/13
s1423	26,300+556	157/16	to	-	to	-	to	-
s1494	11+3	9/6	12+5	7/8	41+36	13/18	26+99	9/5
s5378	323+8	26/9	532+67	34/10	722+148	32/9	1,840+829	34/9
s13207	68,817+2,839	74/29	77,102+3,772	73/29	to	-	mem	-
s15850	to	-	to	-	to	-	to	-
s35932	1,167+1,189	21/7	1,234+1,815	22/7	mem	-	mem	-
s38417	mem	-	mem	-	mem	-	mem	-
s38584	mem	-	mem	-	mem	-	mem	-

*{number of obfuscation/authentication FSMs, The length of unlocking sequence}
time: in Seconds timeout (to): 24 hours size: #DISes/Depth mem: Out of memory

for 73.4% of the cases in Table 6.2, the extracted secrets are the correct expected ones. Note that, since this model generates the attack model once (without iterative structure)[5], increasing the number of I/O pairs does not affect the execution time significantly (almost linearly *w.r.t.* the number of I/O pairs).

Table 6.3 depicts the performance of the RANE framework on the same logic locking technique, i.e., HARPOON[6], but in this case (case study 2), we assumed that the oracle is available. In this case, the formal tool can generate different DISes for finding init state. Unlike case study 1, finding the unlocking sequence (secret 2) will be started when the init state (secret 1) is found. Hence, in this experiment, the execution time of the RANE framework is divided into two parts, $t_1 + t_2$, in which t_1 is the RANE execution time for finding the secret 1, and t_2 is the time required to find secret 2. The *size* indicates the

[5]The model defined in Fig. 6.4 will be generated at once. The attack process will be accomplished for two secrets simultaneously. It will find the init state (secret 1) and the unlocking sequence (secret 2) at once.

[6]The same locked circuits are used for the experiments on case studies 1 and 2.

Table 6.4: RANE performance in Case Study 3 - Oracle-guided on Random-based Sequential Logic Locking.

Circuit	RANE					KC2 (neos) [47]				
key size	100	150	200	250	300	100	150	200	250	300
s344	5	7	n/a	n/a	n/a	1	5	n/a	n/a	n/a
s386	6	30	n/a	n/a	n/a	3	27	n/a	n/a	n/a
s832	11	40	196	504	n/a	3	50	644	5,771	n/a
s1196	4	14	9	15	70	1	5	4	10	82
s1494	20	31	69	310	511	3	10	18	63	144
key size	10	20	30	40	50	10	20	30	40	50
s382	6	183	to	to	to	68	246	to	to	to
s526	15	4,932	to	to	to	to	to	to	to	to
s713	2	2	1	2	3	0	0	0	0	1
s838	12	211	to	16	to	53	to	to	to	to
s1423	7	21	921	to	to	10	77	9,041	to	to
s5378	16	14	62	25	28	8	10	40	7	9
s13207	779	817	784	820	839	2,840	3,686	2,881	2,790	2,414
s15850	772	738	767	735	921	688	490	507	703	795
s35932	895	660	985	850	811	777	882	3,576	1,095	8,056
s38417	5,571	6,036	6,287	to	to	to	to	to	to	to
s38584	to	to	to	to	to	to	to	to	to	to

time: in Seconds timeout (to): 4 hrs n/a: Circuits are too small for that key size
KC2 is executed with different configurations, and the best performance is reported.

attack model size in terms of {*#DISes/Depth*}. This experiment reveals one of the biggest limitations of unrolling-based attacks. The problem size will be grown in two dimensions: (1) increasing the number of DISes, (2) increasing the depth of unrolling. Thus, for larger circuits, this model faces a larger execution time. For cases with the memory bound, switching to commercial formal tools, e.g. Cadence JasperGold, will resolve the issue.

Table 6.4 shows the performance of the RANE framework in case study 3. In this case, we assume the (XOR-based) key gates are inserted at random places, the access to the scan chain is *BLOCKED*, and the attack model evaluates the circuit as a whole. To provide comparative results, we engage the PySMT generator for building the model for this case study. The unrolling has been accomplished statically/explicitly, and similar to KC2 (neos), the locked circuits are in `Bench` format. In this experiment, the best performance achieved by *Boolector, MathSAT,* and *Yices* has been reported [89]. As demonstrated, with

Table 6.5: RANE performance in Case Study 4 - Oracle-guided on Combinational Logic Locking.

Circuit	RANE				SAT (sld) [11]			
overhead	%5	%10	%25	%50	%5	%10	%25	%50
c432	1	1	1	1	0	0	0	0
c499	0	1	1	2	0	2	2	12
c880	0	1	2	7	0	0	1	4
c1355	1	2	8	107	0	1	7	169
c1908	1	2	20	689	1	2	21	377
c2670	1	to	to	to	to	to	to	to
c3540	2	4	9	201	4	2	6	122
c5315	7	45	to	9,804	5	20	to	to
c7552	44	2,227	to	to	43	to	to	to

time: in Seconds timeout (to): 4 hrs

outperforming KC2 (neos) [47] for larger circuits, the RANE framework promises better scalability. Note that, since KC2 (neos) is a pre-compiled C++ platform, it outperforms RANE for the smaller circuits[7].

Table 6.5 compares the performance of the RANE framework, once it models case study 4 using PySMT generator, with the conventional SAT attack on combinational logic locking by Subramanyan *et al.* [11]. Similarly, since the conventional SAT attack has been deployed using a compiled binary file and uses a pre-compiled SAT solver, it outperforms the RANE framework in some parts of the experiment. However, for larger circuits, we observe that the RANE framework can outperform the conventional SAT attack by reconfiguring it to use the most suitable SAT solver for each circuit.

6.3 Conclusion

In this chapter, we introduced the Reversal Assessment of Netlist Encryption (RANE) Attack, an open-source framework for evaluating the security of logic locking techniques. The RANE framework integrates packages like Pyverilog and PySMT with formal verification tools to support circuits described in standard languages, like `Verilog`. We evaluated the effectiveness of the RANE framework on *FOUR* different case studies. We illustrated how

[7]compared to RANE, KC2 provides faster parsing/solving on the small circuits.

the RANE attack could model different de-obfuscation attacks with much less effort. We also demonstrated how the RANE attack could use either a golden chip as a reference or a set of pre-recorded I/Os for the unlocking process. Moreover, we also illustrated how the RANE framework could formulate the first attack model on key-less FSM obfuscation solutions. Our experimental results show that the high scalability of formal tools allows RANE to outperform the existing de-obfuscation attacks on larger circuits while eliminating the shortcomings of prior art de-obfuscation solutions for dealing with translation and modeling of complex structures.

Chapter 7: Conclusion and Future Work

7.1 Future Work

RANE, introduced in chapter 6, integrates formal verification tools with packages like Pyverilog and PySMT to provide rich extensibility, deployability, scalability, and performance, especially compared to the existing pre-compiled and optimized but static de-obfuscation attacks:

1. Almost all attacks with much more capabilities and assumptions can be modeled using the RANE framework. For instance, the uncommon usage of latches for latch-based or clock-gated logic locking techniques [93] requires a troublesome and hard-to-be-achieved transition to be acceptable by the existing attacks. However, RANE can support a much more comprehensive range of digital building blocks and macros with full support on standard library cells.

2. The futuristic support of different features/capabilities in formal tools could be easily engaged in the RANE framework. For instance, the support of range equivalent circuits in the formal tools will allow us to formulate a much more scalable attack model on sequential logic locking. Range equivalent circuits are compressed circuits representing the circuit's τ^{th} unroll, accepting inputs and generating outputs in the range of the τ^{th} unroll circuit. Although the concept of range equivalent circuits is an open research problem, the RANE framework can support and engage such features once the formal tools support it.

3. Parallelism could be engaged much more appropriately when CAD formal tools are in place. For instance, the joint Cadence and AWS proof-of-concept for utilizing various

degrees of parallelism using JasperGold on AWS (JAWS) show how verification could achieve huge speed-up on parallel computing systems [94].

7.2 Conclusion

Logic obfuscation is a fast-changing field of research. Through its short period from its inception in 2008 until now, numerous promising solutions were proposed, but, almost all of these defensive mechanisms later were broken by a variant of SAT attack or structural analysis. By providing this framework, we tried to make the evaluation of the future methods easier and to assist the hardware security community to move from a direct SAT solver interface to a more high-level interface using formal verification tools.

Bibliography

[1] DIGITIMES, "Trends in the global ic design service market," *online http://www.digitimes.com/news/a20120313RS400.html?chid=2*, vol. 2013, 2013.

[2] U. Guin, D. Forte, and M. Tehranipoor, "Anti-counterfeit Techniques: From Design to Resign," in *14th Int. Workshop on Microprocessor Test and Verification*, Dec 2013, pp. 89–94.

[3] K. Zamiri Azar, H. Mardani Kamali, H. Homayoun, and A. Sasan, "Threats on Logic Locking: A Decade Later," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*. New York, NY, USA: ACM, 2019, pp. 471–476.

[4] K. Zamiri Azar, F. Farahmand, H. Mardani Kamali, S. Roshanisefat, H. Homayoun, W. Diehl, K. Gaj, and A. Sasan, "COMA: Communication and obfuscation management architecture," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Sep. 2019, pp. 181–195.

[5] S. Roshanisefat, H. Mardani Kamali, H. Homayoun, and A. Sasan, "SAT-hard Cyclic Logic Obfuscation for Protecting the IP in the Manufacturing Supply Chain," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (TVLSI)*, ser. TVLSI '20. IEEE, 2020.

[6] J. Rajendran, M. Sam, O. Sinanoglu, and R. Karri, "Security Analysis of Integrated Circuit Camouflaging," in *Proceedings of the 2013 ACM SIGSAC Conf. on Computer & Communications Security*. ACM, 2013, pp. 709–720.

[7] B. Erbagci, C. Erbagci, N. E. C. Akkaya, and K. Mai, "A secure camouflaged threshold voltage defined logic family," in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, May 2016, pp. 229–235.

[8] M. Li, K. Shamsi, T. Meade, Z. Zhao, B. Yu, Y. Jin, and D. Z. Pan, "Provably Secure Camouflaging Strategy for IC Protection," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 2017.

[9] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. J. Rajendran, and O. Sinanoglu, "Provably-Secure Logic Locking: From Theory To Practice," in *Proceedings of the 2017 ACM SIGSAC Conf. on Computer and Comm. Security*, 2017, pp. 1601–1618.

[10] H. Mardani Kamali, K. Zamiri Azar, K. Gaj, H. Homayoun, and A. Sasan, "LUT-Lock: A Novel LUT-based Logic Obfuscation for FPGA-Bitstream and ASIC-Hardware Protection," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, July 2018, pp. 1–6.

[11] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the Security of Logic Encryption Algorithms," in *2015 IEEE Int. Symp. on Hardware Oriented Security and Trust (HOST)*, May 2015, pp. 137–143.

[12] M. El Massad, S. Garg, and M. V. Tripunitara, "Integrated Circuit (IC) Decamouflaging: Reverse Engineering Camouflaged ICs within Minutes," in *NDSS*, 2015.

[13] S. Roshanisefat, H. K. Thirumala, K. Gaj, H. Homayoun, and A. Sasan, "Benchmarking the Capabilities and Limitations of SAT Solvers in Defeating Obfuscation Schemes," in *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, July 2018, pp. 275–280.

[14] K. Zamiri Azar, H. Mardani 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*, vol. 2019, no. 1, pp. 97–122, Nov. 2018.

[15] M. Yasin, B. Mazumdar, J. J. V. Rajendran, and O. Sinanoglu, "SARLock: SAT Attack Resistant Logic Locking," in *2016 IEEE Int. Symp. on Hardware Oriented Security and Trust (HOST)*, May 2016, pp. 236–241.

[16] Y. Xie and A. Srivastava, "Mitigating SAT Attack on Logic Locking," in *Int. Conf. on Cryptographic Hardware and Embedded Systems*. Springer, 2016.

[17] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, "AppSAT: Approximately Deobfuscating Integrated Circuits," in *IEEE Int'l Symp. on Hardware Oriented Security and Trust (HOST)*, 2017, pp. 95–100.

[18] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran, "Security Analysis of Anti-SAT," in *22nd Asia and South Pacific Design Automation Conf.*, Jan 2017.

[19] X. Xu, B. Shakya, M. M. Tehranipoor, and D. Forte, "Novel Bypass Attack and BDD-based Tradeoff Analysis Against All Known Logic Locking Attacks," in *Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2017.

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

[21] 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*. ACM, 2018, pp. 147–152.

[22] 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*. ACM, 2019, pp. 89:1–89:6.

[23] G. Kolhe, H. M. Kamali, M. Naicker, T. D. Sheaves, H. Mahmoodi, S. M. P. Dinakarrao, H. Homayoun, S. Rafatirad, and A. Sasan, "Security and Complexity Analysis of LUT-based Obfuscation: From Blueprint to Reality," in *Proceeding of the International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[24] G. Kolhe, S. M. PD, S. Rafatirad, H. Mahmoodi, A. Sasan, and H. Homayoun, "On Custom LUT-Based Obfuscation," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, ser. GLSVLSI 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 477482.

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

[26] J. Sweeney, M. J. H. Heule, and L. Pileggi, "Modeling techniques for logic locking," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[27] G. S. Tseitin, *On the Complexity of Derivation in Propositional Calculus.* Springer Berlin Heidelberg, 1983, pp. 466–483.

[28] Y. Xie and A. Srivastava, "Delay Locking: Security Enhancement of Logic Locking Against IC Counterfeiting and Overproduction," in *Proceedings of the 54th Annual Design Automation Conference 2017.* ACM, 2017, pp. 9:1–9:6.

[29] A. Chakraborty, Y. Liu, and A. Srivastava, "TimingSAT: Timing Profile Embedded SAT Attack," in *Proceedings of the Int. Conference on Computer-Aided Design.* ACM, 2018, pp. 1–6.

[30] J. A. Roy, F. Koushanfar, and I. L. Markov, "Ending Piracy of Integrated Circuits," *Computer*, vol. 43, no. 10, pp. 30–38, Oct 2010.

[31] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Security Analysis of Logic Obfuscation," in *Proceedings of the 49th Annual Design Automation Conf.* ACM, 2012, pp. 83–89.

[32] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, "Cyclic Obfuscation for Creating SAT-Unresolvable Circuits," in *Proc. of the on Great Lakes Symposium on VLSI 2017.* ACM, 2017, pp. 173–178.

[33] 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*, vol. 2020, no. 1, pp. 175–202, Nov. 2019. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8397

[34] H. M. Kamali, K. Z. Azar, H. Homayoun, and A. Sasan, "Interlock: An intercorrelated logic and routing locking," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[35] K. Zamiri Azar, H. M. Kamali, S. Roshanisefat, H. Homayoun, C. P. Sotiriou, and A. Sasan, "Data flow obfuscation: A new paradigm for obfuscating circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 4, pp. 643–656, 2021.

[36] S. Roshanisefat, H. Mardani Kamali, and A. Sasan, "SRCLock: SAT-Resistant Cyclic Logic Locking for Protecting the Hardware," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3194554.3194596

[37] S. Roshanisefat, H. Mardani Kamali, K. Zamiri Azar, S. Manoj Pudukotai Dinakarrao, N. Karimi, H. Homayoun, and A. Sasan, "DFSSD: Deep Faults and Shallow State Duality, A Provably Strong Obfuscation Solution for Circuits with Restricted Access to Scan Chain," in *Proceedings of the IEEE VLSI Test Symposium (VTS)*, ser. VTS '20. New York, NY, USA: IEEE, 2020.

[38] R. S. Chakraborty and S. Bhunia, "Harpoon: An obfuscation-based soc design methodology for hardware protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1493–1502, 2009.

[39] A. R. Desai, M. S. Hsiao, C. Wang, L. Nazhandali, and S. Hall, "Interlocking obfuscation for anti-tamper hardware," in *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, ser. CSIIRW '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2459976.2459985

[40] J. Dofe and Q. Yu, "Novel dynamic state-deflection method for gate-level design obfuscation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 2, pp. 273–285, 2018.

[41] H. M. Kamali, K. Z. Azar, H. Homayoun, and A. Sasan, "On designing secure and robust scan chain for protecting obfuscated logic," in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, ser. GLSVLSI '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 217222. [Online]. Available: https://doi.org/10.1145/3386263.3407655

[42] U. Guin, Z. Zhou, and A. Singh, "Robust Design-for-Security Architecture for Enabling Trust in IC Manufacturing and Test," *IEEE Tran. on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 5, 2018.

[43] N. Limaye, A. Sengupta, M. Nabeel, and O. Sinanoglu, "Is Robust Design-for-Security Robust Enough? Attack on Locked Circuits with Restricted Scan Chain Access," *CoRR*, vol. abs/1906.07806, 2019.

[44] H. Mardani Kamali, K. Zamiri Azar, H. Homayoun, and A. Sasan, "Scramble: The state, connectivity and routing augmentation model for building logic encryption," in *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2020, pp. 153–159.

[45] N. Limaye, E. Kalligeros, N. Karousos, I. G. Karybali, and O. Sinanoglu, "Thwarting all logic locking attacks: Dishonest oracle with truly random logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2020.

[46] M. El Massad, S. Garg, and M. Tripunitara, "Reverse Engineering Camouflaged Sequential Circuits without Scan Access," in *IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, 2017, pp. 33–40.

[47] K. Shamsi, M. Li, D. Z. Pan, and Y. Jin, "KC2: Key-Condition Crunching for Fast Sequential Circuit Deobfuscation," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019.

[48] L. Alrahis, M. Yasin, H. Saleh, B. Mohammad, M. Al-Qutayri, and O. Sinanoglu, "ScanSAT: Unlocking Obfuscated Scan Chains," in *Proc of the Asia and South Pacific Design Automation Conf.*, 2019, pp. 352–357.

[49] F. Koushanfar, *Active Hardware Metering by Finite State Machine Obfuscation.* Cham: Springer International Publishing, 2017, pp. 161–187.

[50] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, Jul. 1960. [Online]. Available: http://doi.acm.org/10.1145/321033.321034

[51] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962. [Online]. Available: http://doi.acm.org/10.1145/368273.368557

[52] J. P. Marques-Silva and K. A. Sakallah, "Grasp: a search algorithm for propositional satisfiability," *IEEE Trans. on Computers*, vol. 48, no. 5, pp. 506–521, May 1999.

[53] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *Int. Conf. on theory and applications of satisfiability testing*, 2003, pp. 502–518.

[54] G. Audemard and L. Simon, "Glucose and Syrup in the SAT Race 2015," *SAT Race*, 2015.

[55] A. Biere, "Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013," *Proceedings of SAT Competition*, vol. 2013, 2013.

[56] T. Balyo, M. J. Heule, and M. Jarvisalo, "Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions," 2017.

[57] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning Rate Based Branching Heuristic for SAT Solvers," in *Int. Conf. on Theory and Applications of Satisfiability Testing.* Springer, 2016, pp. 123–140.

[58] M. Soos, K. Nohl, and C. Castelluccia, "CryptoMiniSat," *SAT Race solver descriptions*, 2010.

[59] J. Rajendran, H. Zhang, C. Zhang, G. S. Rose, Y. Pino, O. Sinanoglu, and R. Karri, "Fault Analysis-Based Logic Encryption," *IEEE Trans. on Computers*, vol. 64, no. 2, pp. 410–424, Feb 2015.

[60] S. Dupuis, P. S. Ba, G. D. Natale, M. L. Flottes, and B. Rouzeyre, "A Novel Hardware Logic Encryption Technique for Thwarting Illegal Overproduction and Hardware Trojans," in *2014 IEEE 20th Int. On-Line Testing Symposium (IOLTS)*, July 2014, pp. 49–54.

[61] H. Zhou, R. Jiang, and S. Kong, "CycSAT: SAT-based Attack on Cyclic Logic Encryptions," in *IEEE Int. Conf. on Computer-Aided Design*, 2017, pp. 49–56.

[62] Y.-C. Chen, "Enhancements to SAT Attack: Speedup and Breaking Cyclic Logic Encryption," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 23, no. 4, May 2018.

[63] Y. Shen, Y. Li, A. Rezaei, S. Kong, D. Dlott, and H. Zhou, "BeSAT: Behavioral SAT-based Attack on Cyclic Logic Encryption," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ACM, 2019, pp. 657–662.

[64] A. Rezaei, Y. Shen, S. Kong, J. Gu, and H. Zhou, "Cyclic locking and memristor-based obfuscation against cycsat and inside foundry attacks," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 85–90.

[65] A. Rezaei, Y. Li, Y. Shen, S. Kong, and H. Zhou, "CycSAT-unresolvable Cyclic Logic Encryption Using Unreachable States," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ACM, 2019, pp. 358–363.

[66] J. H. Chen, Y. C. Chen, W. C. Weng, C. Y. Huang, and C. Y. Wang, "Synthesis and verification of cyclic combinational circuits," in *IEEE Int'l System-on-Chip Conf. (SOCC)*, 2015, pp. 257–262.

[67] V. Agarwal, N. Kankani, R. Rao, S. Bhardwaj, and J. Wang, "An efficient combinationality check technique for the synthesis of cyclic combinational circuits," in *Proc. of the ASP-DAC*, 2005, pp. 212–215.

[68] M. D. Riedel and J. Bruck, "The synthesis of cyclic combinational circuits," in *Proc. 2003. Design Automation Conf.*, 2003, pp. 163–168.

[69] R. L. Rivest, "The Necessity of Feedback in Minimal Monotone Combinational Circuits," *IEEE TC*, vol. 26, no. 6, pp. 606–607, 1977.

[70] K. A. Hawick and H. A. James, "Enumerating Circuits and Loops in Graphs with Self-Arcs and Multiple-Arcs." in *FCS*, 2008, pp. 14–20.

[71] B. Dutertre, "Yices 2.2," in *Computer Aided Verification*. Springer, 2014, pp. 737–744.

[72] A. Vakil, H. Homayoun, and A. Sasan, "IR-ATA: IR Annotated Timing Analysis, a Flow for Closing the Loop Between PDN Design, IR Analysis & Timing Closure," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ACM, 2019, pp. 152–159.

[73] R. Karmakar, S. Chatopadhyay, and R. Kapur, "Encrypt Flip-Flop: A Novel Logic Encryption Technique For Sequential Circuits," 2018.

[74] X. Wang, D. Zhang, M. He, D. Su, and M. Tehranipoor, "Secure Scan and Test Using Obfuscation Throughout Supply Chain," *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 9, Sep. 2018.

[75] S. Potluri, A. Kumar, and A. Aysu, "SeqL: SAT-attack Resilient Sequential Locking," *IACR Cryptology ePrint Archive*, vol. 2019, p. 656, 2019.

[76] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, *Model Checking and the State Explosion Problem.* Springer, 2012, pp. 1–30.

[77] M. Mneimneh and K. Sakallah, "SAT-based Sequential Depth Computation," in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference.* ACM, 2003, pp. 87–92.

[78] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, "Reverse engineering digital circuits using functional analysis," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2013, pp. 1277–1280.

[79] B. Shakya, H. Shen, M. Tehranipoor, and D. Forte, "Covert Gates: Protecting Integrated Circuits with Undetectable Camouflaging," *IACR Tran. on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 3, May 2019.

[80] B. Sterin, N. Een, A. Mishchenko, and R. Brayton, "The benefit of concurrency in model checking," in *Proceedings of the International Workshop on Logic Synthesis, IWLS*, 2011, pp. 176–182.

[81] H. Yotsuyanagi and K. Kinoshita, "Undetectable fault removal of sequential circuits based on unreachable states," in *Proceedings. 16th IEEE VLSI Test Symposium*, April 1998, pp. 176–181.

[82] N. Limaye and O. Sinanoglu, "Dynunlock: Unlocking scan chains obfuscated using dynamic keys," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 270–273.

[83] T. Meade, J. Portillo, S. Zhang, and Y. Jin, "Neta: When ip fails, secrets leak," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 9095. [Online]. Available: https://doi.org/10.1145/3287624.3288739

[84] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 24–40.

[85] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[86] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Applied Reconfigurable Computing*, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds. Cham: Springer International Publishing, 2015, pp. 451–460.

[87] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[88] B. Dutertre, "Yices2.2," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 737–744.

[89] R. Brummayer and A. Biere, "Boolector: An efficient smt solver for bit-vectors and arrays," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Kowalewski and A. Philippou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 174–177.

[90] M. Gario and A. Micheli, "Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms," in *SMT Workshop 2015*, 2015.

[91] GATE Lab, "RANE framework v.1.0.0," https://github.com/gate-lab/RANE, https://cadforassurance.org/tools/evaluation-of-obfuscation/rane, 2021.

[92] GMU Office of Research Computing, "ARGO Cluster," http://wiki.orc.gmu.edu/mediawiki, 2021.

[93] J. Sweeney, V. Mohammed Zackriya, S. Pagliarini, and L. Pileggi, "Latch-based logic locking," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 132–141.

[94] A. Elzeftawi *et al.*, "Running Cadence JasperGold formal verification on AWS at scale," https://aws.amazon.com/blogs/industries/tag/cadence-jaspergold/, 2021.

Curriculum Vitae

Shervin Roshanisefat received his B.Sc. degree in computer engineering from Qazvin Azad University, Iran in 2009. He received his M.Sc. degree in computer engineering from the University of Tehran, Iran in 2015. He has worked earlier on telecommunication devices for SRD in Iran. Since Fall 2016, he has been a Ph.D. student at the Electrical and Computer Engineering department at George Mason University. His research interests are in the areas of approximate computing, low power design, hardware-level functional safety, and security.