Full-Lock: Hard Distributions of SAT instances for Obfuscating Circuits using Fully Configurable Logic and Routing Blocks

Hadi Mardani Kamali, Kimia Zamiri Azar, Houman Homayoun, Avesta Sasan
George Mason University
Farifax, VA, USA
{hmardani,kzamiria,hhomayou,asasan}@gmu.edu

ABSTRACT

In this paper, we propose a novel and SAT-resistant logic-locking technique, denoted as Full-Lock, to obfuscate and protect the hardware against threats including IP-piracy and reverse-engineering. The Full-Lock is constructed using a set of small-size **full**y Programmable Logic and Routing b**lock** (PLR) networks. The PLRs are SAT-hard instances with reasonable power, performance and area overheads which are used to obfuscate (1) the routing of a group of selected wires and (2) the logic of the gates leading and proceeding the selected wires. The Full-Lock resists removal attacks and breaks a SAT attack by significantly increasing the complexity of each SAT iteration.

KEYWORDS

Reverse Engineering, Logic Locking, SAT Attack, Logarithmic Network.

1 INTRODUCTION AND BACKGROUND

To reduce the cost of manufacturing a new chip, to take advantage of new technology nodes, or to meet the market demand, the manufacturing supply chain of Integrated Chips (IC) is globalized, distributing steps of IC manufacturing over different facilities in different countries [32]. This is when the lack of trust and monitoring mechanisms has raised concerns over manufacturing supply chain security threats such as overproduction, Trojan insertion, reverse engineering, IP theft, and counterfeiting [20]. To combat these threats, *logic locking* 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 [26]. When using logic locking, an attacker in the manufacturing supply chain cannot re-generate the correct functionality of an IC/IP without the correct key. After the chip is fabricated, it is then programmed in a trusted facility. Considering a large number of key possibilities (e.g. 2^{20} possibilities with only 20 keys), a brute force attack on logic locking faces a runtime that is on average exponentially related to the number of key values.

Shortly after introducing the primitive logic locking solutions [9, 18, 19, 21], a very strong Boolean attack, the *Satisfiability* (*SAT*) attack, was proposed [14]. In this attack model, the attacker has access to the obfuscated, but reverse engineered netlist. In addition, the attacker is able to obtain a functional/unlocked IC, apply a desired input and observe its output (oracle-based attack). The SAT attack can extract the functionality of locked circuit by applying and testing only a few smartly selected input queries. It was shown that the SAT attack could break all previously proposed primitive locking mechanisms in almost polynomial time.

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317831

To thwart the strength of SAT attack, researchers have investigated two main directions (1) formulating locking solutions that significantly increase the number of required SAT iterations (inputs to be tested), (2) formulating the locking solutions such that it is not translatable to a SAT problem. The first approach in which formulating obfuscation and locking solutions significantly increase the number of SAT iterations was assumed to be a perfect anti-SAT solution, such as SARLock, Anti-SAT, SFLL, and LUT-Lock [4, 11, 27, 30]. In extreme case, using these techniques, each tested input (each iteration) invalidate a single key combination. Hence, by using these techniques, a SAT attack, similar to a brute force attack, faces an exponential runtime. However, further investigations demonstrated that some of these locking techniques are vulnerable to other types of attacks such as Signal Probability Skew (SPS) attack [13], removal attack [12], approximate-based attack(s) [6, 22], bypass attack [29], and *Satisfiability Module Theories* (SMT) attack [8]. In addition, these techniques suffer from very low output corruption. Hence, an unactivated IC behaves almost identical to an unlocked IC with exception of one or few inputs.

The second approach investigated by researchers was formulating obfuscation and locking mechanisms that were not translatable to SAT problems. Example of such techniques includes the use of cyclic Boolean logic for locking [16] or behavioral locking of the logic [28]. The cyclic obfuscation creates combinational cycles in the design. This invalidates the Directed Acyclic Graph (DAG) nature of logic and forces a SAT attack to either be trapped in an infinite loop or to generate an incorrect key upon termination [16]. Alternatively, in [28] a behavioral (non-Boolean) locking scheme was introduced where the locking mechanism targeted the setup and hold properties (timing properties) of the circuit. However, shortly after the introduction of these obfuscation techniques, researchers revealed stronger and more advanced modeling and attack solutions such as cycSAT [5], and *Satisfiability Module Theories* (SMT) attack [8] that were able to model the cyclic or behavioral locking into a SAT or SAT+theory solvable logic problems.

A new (and third) direction for building SAT-hard solutions, which is thoroughly discussed in this paper, is to significantly increase the run-time of *each iteration* of the SAT solver. The only existing solution that somewhat fits this category is the Cross-lock [7], in which a one-time programmable interconnect mesh is used to obfuscate the routing of a netlist, and the resulting obfuscated netlist substantially increase the runtime of each iteration of the SAT attack. However, we will illustrate that obfuscation solution in [7], although a step in the right direction, is not a strong solution in this space, and by following the principles and design guidelines discussed in this paper, it is possible to construct obfuscated circuits that translate into far harder SAT circuits than Cross-lock.

In this paper, we explore the characteristics and principles of designing this new category of SAT-hard obfuscation solutions, where the goal is to exponentially increase the time required for each iteration of the SAT attack. As a strong representative member of this class of obfuscation techniques, we introduce *Full-Lock*. The Full-Lock is constructed using a set of cascaded fully programmable logic and

routing blocks (PLR) networks that replace parts of the logic and routing in the desired netlist. The PLRs are SAT-hard instances designed to construct a desired ratio between the number of clauses and the number of variables with PLRs are translated to their Conjunctive Normal Form (CNF). The cascaded and non-blocking design of PLR pushes the SAT solver's algorithm to build a very deep decision tree and to spend significant time in hopeless regions of the decision tree, causing a significant increase in each iteration of SAT attack.

2 A NEW PERSPECTIVE OF SAT HARDNESS

A SAT attack, in each of its iterations, finds a *Discriminating Input Patterns* and rules out one or more incorrect key value(s). Hence, many SAT-resilient locking schemes tried to weaken the pruning power of one DIPs, making sure each DIP can only rule out one incorrect key. This forces the number of needed iterations to exponentially increase with respect to the number of keys as a mean of exponentially increasing the required execution time of the SAT attack, although, the execution time of each iteration of SAT solver could be quite short.

The strength of SAT solvers come from their Conflict-Driven Clause Learning (CDCL) ability. In each iteration of the SAT attack, a new SAT problem is defined. The goal of the SAT solver is to finds a satisfying value for all its literals. The literal values are either assigned or derived. Each assignment of value to a literal pushes the solver down into one of the branches of its decision tree implemented using a recursive call. During this recursive procedure, if the solver reaches a state where the derived value of a literal is different from its previously derived or assigned value, a *conflict* is detected. This is when the solver investigates how the conflict was driven, identifies a set of literal assignments that cause the conflict, and generates a clause that prevents the identified literal assignment. The newly learned conflict-clause is then added to the original problem set, aiding the solver to prune its decision tree and to avoid reaching the same conflict in the future. Then, the decision tree is backtracked a safe point prior to the conflict.

Davis-Putnam-Logemann-Loveland (*DPLL*) algorithm (or one of its derivatives), which is used to perform CDCL, is illustrated in Algorithm 1. Each SAT iteration invokes the DPLL function. In addition, DPLL may also call itself. As it can be seen in *line* 12 and 16, new recursive call adds a new variable, l or $\bar{l}$, to Φ. Hence, an increase in the number of recursive calls (*line* 12 and 16) increases the complexity of the next DPLL call. So, the number and complexity of recursive DPLL calls could be a dominant factor for each invocation of SAT solver (a SAT Attack iteration).

Algorithm 1 DPLL Algorithm Pseudo-code

```
1:  function DPLL(Φ)
2:      if Φ has an empty clause then
3:          return "UNSAT";
4:      if Φ is [] then                                   ▷ Φ is empty
5:          SAT_assign ← Current Assignment;
6:          return "SAT";
7:      if Φ contains a unit clause l then                ▷ Unit Propagation
8:          Φ ← Φ - all clauses with l;
9:          Φ ← Φ with eliminating all l̄;
10:         return DPLL(Φ);
11:     if Φ contains a pure literal l then               ▷ Purification
12:         return DPLL(Φ ∪ l);
13:     if DPLL(Φ ∪ l) is SAT then                        ▷ Branching
14:         return "SAT";
15:     else
16:         return DPLL(Φ ∪ l̄);                           ▷ (One more level in Tree)
```

The runtime of a SAT attack could be obtain from:

$$T_{Attack} = \sum_{i=1}^{N} T(i) = \sum_{i=1}^{N} (t + T_{DPLL}(\Phi)) \quad (1)$$

A difficult problem requires a very large runtime. The first solution is weakening the DIP and increasing the number of iteration (N) to a very large number [4, 11, 27, 30, 31]. In spite of very shallow DPLL recursive tree, and for having a very large N these solution exhibit resistance against SAT attack. However, this type of defense, as suggested previously is prone to SPS [27], Approximate-based [6, 22], bypass [29], and possibly removal attack [12].

Based on the discussion on DPLL, an alternative solution is smaller N but larger recursive trees. Hence, as illustrated in equation 2, the attack time could also increase beyond acceptable if the number of recursive calls (M) grows to a very large number.

$$T_{Attack} = \sum_{i=1}^{N} (t + T_{DPLL}(\Phi)) \simeq \sum_{i=1}^{N} \sum_{j=1}^{M} (T_{DPLL}^{Avg}) \simeq MN \times T_{DPLL}^{Avg} \quad (2)$$

The very strong aspect of this form of building SAT-hard solutions is that (1) the problems posed at each iteration of SAT attack is a SAT-hard problem, (2) the output corruption of this methods is significantly higher than obfuscating solution relying on increasing the N, (3) it is not susceptible to SPS, removal or approximate attack.

Motivated from this discussion, in this paper we present the Full-Lock. Full-Lock is able to considerably and exponentially increase the number (M) and computational complexity (T_{DPLL}^{Avg}) of recursive calls in DPLL function via replacing some of the logic and routing in the circuit by one or more SAT-hard obfuscation instance(s) in the circuit.

3 FULL-LOCK

Many SAT-hard problems (instances) are introduced annually in SAT competition. These problems aim to trap *Davis-Putnam-Logemann-Loveland* (*DPLL*) or generate extremely complex and time-consuming computational models for this algorithm. Although, none of them is directly convertible to a logic circuit, feature and tricks used in these SAT-hard problems could be used in designing SAT-hard circuit (SATC) obfuscation problems.

In [17], the SAT hardness of formulas produced using fixed-length clause generator was investigated. This work concluded that "*For formulas that are either relatively short, in which the number of clauses per variable is less than 3, or relatively long, in which the number of clauses per variable is larger than 6, DPLL finishes quickly, but the formulas of medium length, between 3 to 6, take significantly longer*". This is because formulas that have few clauses are *under-constrained*, and have several satisfying assignments. Providing under constrained clauses to the algorithm 1 increases the chances of one satisfying assignment to be found early in the search using *unit propagation* or *purification*. Note that these two steps of DPLL algorithm are used to simplify the size of formula before *branching*, while *branching* assigns a value to an unassigned variable, making the DPLL tree one level deeper. Formulas that have many clauses on the other hand are *over-constrained*. In over-constrained clauses, the contradictions are found easier and the search is quickly concluded.

SAT hardness of medium length formulas is higher than under or over-constrained formulas. This is because they only have relatively few (if any) satisfying assignments. Hence, throughout the search and after assigning values to many variables, many empty clauses will be generated. This results in a deep DPLL recursive tree for testing each assumption [1]. Fig. 1 demonstrates the number of recursive calls made by DPLL for solving the formula for fixed-length 3-SAT formulas, where the ratio of clauses to variables is varied from 2 to 8. As illustrated, the ratio from 3 to 6 provides much higher DPLL calls, and 4.3 clauses per variable is the best ratio, generating the most computational challenging SAT instances with the highest number of DPLL calls. For example, a 100-variable 300-clause instance (clause/variable = 3 "under-constrained"), or a 100-variable 5000-clause instance (clause/variable = 50 "over-constrained") is easily solvable within few seconds. However, the SAT solver takes a very long time solving a 3-SAT instance which is constructed with 100 variables and 450 clauses. From this discussion, an obfuscated circuit

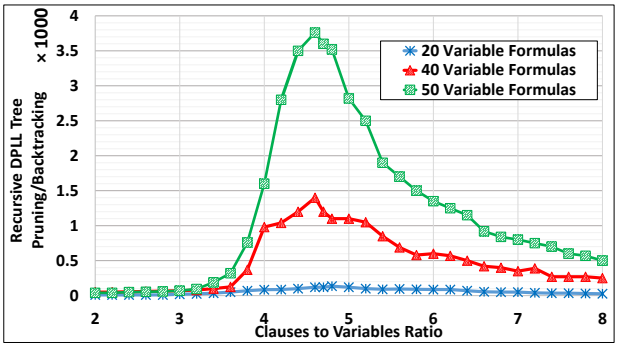

Figure 1: Median Number of Recursive DPLL Tree Pruning/Backtracking for Random 3-SAT Formulas, based-on the Ratio of Clauses to Variables [17].

Table 1: Tseytin Transformation of Basic Logic Gates.

Gate	Operation	CNF (sub-expression)
C = AND(A,B)	$C = A.B$	$(\overline{A} \vee \overline{B} \vee C) \wedge (A \vee \overline{C}) \wedge (B \vee \overline{C})$
C = NAND(A,B)	$C = \overline{A.B}$	$(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee \overline{C}) \wedge (B \vee \overline{C})$
C = OR(A,B)	$C = A + B$	$(A \vee B \vee \overline{C}) \wedge (\overline{A} \vee C) \wedge (\overline{B} \vee C)$
C = NOR(A,B)	$C = \overline{A + B}$	$(A \vee B \vee C) \wedge (\overline{A} \vee \overline{C}) \wedge (\overline{B} \vee \overline{C})$
C = BUFF(A,B)	$C = A$	$(A \vee \overline{C}) \wedge (\overline{A} \vee C)$
C = NOT(A,B)	$C = \overline{A}$	$(\overline{A} \vee \overline{C}) \wedge (A \vee C)$
C = XOR(A,B)	$C = A \oplus B$	$(\overline{A} \vee \overline{B} \vee \overline{C}) \wedge (A \vee B \vee \overline{C}) \wedge (A \vee \overline{B} \vee C) \wedge (\overline{A} \vee B \vee C)$
C = XNOR(A,B)	$C = \overline{A \oplus B}$	$(\overline{A} \vee \overline{B} \vee C) \wedge (A \vee B \vee C) \wedge (A \vee \overline{B} \vee \overline{C}) \wedge (\overline{A} \vee B \vee \overline{C})$
C = MUX(S,A,B)	$C = A.\overline{S} + B.S$	$(S \vee \overline{A} \vee C) \wedge (S \vee A \vee \overline{C}) \wedge (\overline{S} \vee \overline{B} \vee C) \wedge (\overline{S} \vee B \vee \overline{C})$

is SAT-hard when its *Conjunctive Normal Form (CNF)* has medium-length clauses with a ratio of clauses to variables between 3 to 6 (best if close to 4)

3.1 Logarithmic Networks for SAT-Hardness

Table 1 lists the Tseytin transformation [25] of various logic gates into their respective CNF expression. From this table, only XOR/XNOR and MUX have 4 clauses per gate. This is when the clauses to variables ratio is 1 and 4/3 in MUX and XOR/XNOR respectively. In spite of the observation that for a single gate the XOR/XNOR has a larger clause to variables ratio, MUXes provides a better building block for constructing SAT-hard circuits. This is because: (1) with no unit propagation and purification, for having four variables, a MUX can make the recursive DPLL tree one level deeper, (2) unit propagation and purification steps in DPLL algorithm provide more simplified and smaller formula using enhanced Gaussian elimination while the contribution of XOR/XNOR gates are much higher [10]. Hence, MUXes needs more DPLL recursive tree prunings/backtrackings compared to XORs/XNORs. Moreover, since unit propagation and purification satisfy less formula, the clause to variable ratio will increase while MUXes have more contribution.

The next step for building a SAT hard problem, and to push the clause to variable ratio to the desired range of 3 to 6 (4.3 as the best), is preventing the propagation and purification from simplifying the circuit before branching into recursive DPLL tree. This could be achieved by building a switching network using MUXes, where none of the variable related to a given MUX in a switching network could be resolved, unless their cascaded variables (related to cascaded MUXes in the original circuit) are resolved, a requirement that is recursively continued. This would prevent purification and simplification prior to reaching the leaves of the decision tree, as each variable in an intermediate layer of switching network is cascaded, while pushing up the clause to variable ratio to the desired range. This is consistent with the finding in the [3], in which investigating Boolean formulations of global detailed interconnect constraints, authors concluded that the CNF of symmetric switching networks is a hard problem for SAT solvers. In addition, using N-by-M switch-boxes, with back-to-back interconnection, illustrated in Fig. 2 creates hard satisfiable instances

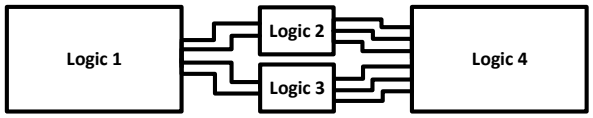

Figure 2: N-by-M switch-boxes for Building Hard Satisfiable Instances [2].

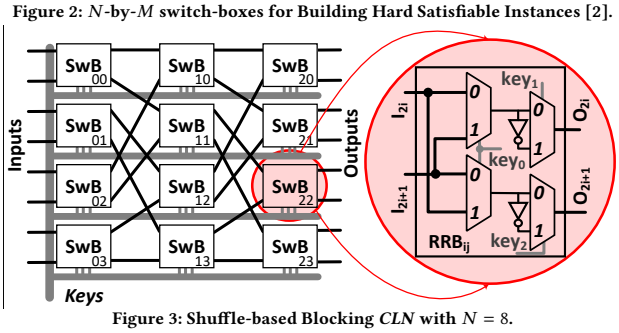

Figure 3: Shuffle-based Blocking *CLN* with $N = 8$.

that trap even the best solvers in hopeless regions of their solution space for a long time before a satisfying solution can be found [2].

In Full-Lock we achieve this by constructing a key-configurable logarithmic-based network (CLN) for obfuscation of routes. For this purpose, we create small and lightweight switch-boxes (SwB) that are implemented easily using only MUXes. These small and lightweight SwBs allow us to create large logarithmic switching ($log_2 N$) network to (1) increase the clauses to variables ratio using MUXes that are independently interconnected back-to-back (cascaded) to each other, and (2) benefit from the hardness of switch-boxes while the power, performance, and area overhead remains reasonable.

Across all switching networks, a set of self-routing logarithmic networks, $log_2 N$ networks, provides configurable interconnection with less overhead compared to conventional networks such as mesh or crossbar. There are numerous self-routing networks in this category, such as *banyan*, *baseline*, *shuffle*, etc. Fig. 3 demonstrates a simple implementation of a 8×8 CLN using the blocking *shuffle* network [24]. This CLN is constructed using small SwBs, where each SwB is built using MUXes. In each SwB the outputs can be an arbitrary permutation of the inputs. In addition, as shown, we add key-configurable inverters for each wire, allowing an output to be shuffled and negated based on the key value. The CLN has N inputs, and due to its structure N is a power of 2. Numbers of SwBs in a CLN depends on the number of inputs as well as the model of $log_2 N$ networks. In all aforementioned blocking CLN, the number of SwBs is the same, i.e. $N/2 * logN$, and the only difference between them is the topology of SwBs interconnections.

The previously discussed self-routing logarithmic networks are blocking networks as they cannot propagate all permutations of their inputs to the outputs. In the result section of this paper, we illustrate that the blocking feature of these networks, eliminate a large number of permutations and significantly reduce the SAT hardness of these networks. This could change by building a non-blocking network.

According to [23], a non-blocking logarithmic network is characterized by $LOG_{N,M,P}$. In this equations N denotes the number of inputs/outputs, M is the number of extra (cascaded) stages, and P indicates there are $P-1$ additional copies *vertically cascaded*. Exploration on N, M, and P shows that the minimum feasible values of P and M, which makes the network strictly non-blocking, results in constructing a much larger network than a blocking CLN. As an instance, for $N = 64$, these values are $M = 3$ and $P = 6$. It means that a $LOG_{N,M,P}$, with $N = 64$, has more than 5× area overhead compared to a blocking CLN with the same input size, i.e. $N = 64$.

To substantially increase the permutations possibilities without incurring large area overhead, we used the near non-blocking logarithmic network suggested in [23] for constructing a key-Configurable Logarithmic-based Network (CLN). This network is able to generate

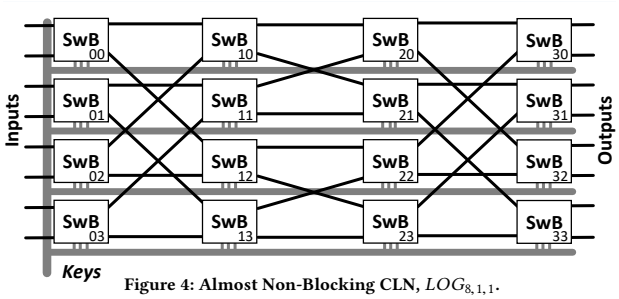

Figure 4: Almost Non-Blocking CLN, $LOG_{8,1,1}$.

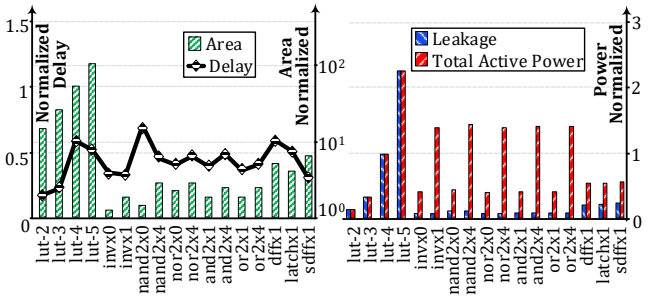

Figure 5: Power, Delay, and Area of STT-LUT and Standard Cells in 28nm CMOS.

not all, but *almost all* permutations, while it could be implemented using a $LOG_{N,log_2(N)-2,1}$ configuration, meaning it has only $log_2(N)-2$ extra stages and no additional copy. Fig. 4, demonstrates an example of such an almost non-blocking CLN with $N = 8$. As it can be seen, the topology of SwBs interconnections is different with *shuffle*-based, shown in Fig. 3. This topology is a *banyan*-based interconnection that matches with our proposed $LOG_{N,log_2(N)-2,1}$.

Since an almost non-blocking CLN has only $log_2(N)-2$ extra stages, its area and power overhead is roughly 2x compared to a blocking CLN with the same N. However, this almost non-blocking CLN is far more resistant against SAT attack compared to a blocking network. For example, an N=64 input non-blocking CLN allow only 5 iterations of SAT attack to be completed within 2×10^6 seconds, while the same size blocking network resist the SAT attack for only ~17 Seconds, or a much larger blocking network of $N = 512$ inputs (4 times the number of inputs, 16 times the area) complete 6 iterations of SAT attack in 2×10^6 seconds.

3.2 Strongly Twisted CLN into LUT/Logic

CLN provides an interconnect locking scheme that is able to generate a SAT-hard instance which significantly increases the execution time for each SAT iteration. However, in order to enhance this strength, and especially resist against other types of attacks, such as removal attack, we try to twist CLN into the logic of the gates around it. For this purpose, we suggest two methods. First, as was mentioned, we add key-configurable inverters within CLN. These inverters allow us to combine the CLN with the logic of the gates leading its inputs. In fact, both logic and interconnect locking is embedded into the CLN. For instance, suppose that one of the inputs of CLN is derived using an *OR* gate. So, we can change it to *NOR*, and configure the CLN to generate its negate on its corresponding output. These key-configurable inverters within CLN allow us to change the logic of the gates leading it. So, even removing CLN and finding the correct permutation provided by CLN will not generate correct functionality. In addition, since adding these inverters has no impact on simplification steps in DPLL, i.e. unit propagation and purification, the clause to variable ratio generated by CLN will not change.

Second, we replace the gates preceding the CLN with small Spin Transfer Torque- (STT)-based LUTs with the same input. Combining CLN with LUTs provide a *fully Programmable Logic and Routing blocks* (PLRs) that bears a resemblance to FPGA architecture. From SAT attack perspective, since each LUT will be translated to MUXes, for a LUT with R inputs, it adds up to R level to recursive DPLL tree. Moreover, since LUTs are directly connected to the output of CLN, these extra R level will be added to the large recursive DPLL tree of CLN. Hence, by massively increasing the size of recursive DPLL tree of CLN using small LUTs, PLR boosts the security of Full-Lock against SAT.

It should be noted that we use STT-based LUTs that are similar in functionality to FPGAs, however, they provide significantly higher speed running at GHz frequency, near zero leakage power, high thermal stability, and highly integrative with CMOS [4]. Since, each gate, located at the output of *CLN*, will be replaced with a LUT with the same input size, investigation on sizes of gates in different benchmarks such as ISCAS-85 and MCNC, shows that the maximum fan-in size is 5. It means that the largest required LUT has 5 inputs. Hence, using STT-based LUTs with a maximum size of 5 relatively has no delay overhead compared to CMOS-based basic gates. In addition, the power and area overhead is considerably low in these LUTs with size less than 5. As shown in Fig. 5, LUTs with size 2, 3, 4, and 5, have negligible overhead compared to CMOS-based basic gates. In addition, the size of all gates leading the CLN can be decreased to be 2. For instance, an *AND3* gate can be changed to two *AND2* while the outputs of one of them is an input for the second one. Hence, the overhead of STT-LUT can be even lower while only LUTs with size 2 is required.

3.3 Inserting SAT-hard *PLR*s into Design

Using these PLRs provides a big advantage compared to other locking schemes. Since inserting a PLR in a circuit provides a SAT-hard instance in the circuit, it is not required to employ a specific insertion to enhance the strength of PLRs. However, due to the topological structure of circuits it may be beneficial to have an insertion policy. But, we demonstrate that even using random insertion/replacement strategy for these PLRs creates extremely large recursive DPLL tree that makes the circuit resilient against SAT.

Additionally, in comparison with Cross-lock [7] that is a layout-based interconnect locking scheme, Full-Lock has no restriction on selection of wires and logic gates to replace them with PLRs. In Cross-lock, since they used high-density cone-based selection strategies, such as k-cut and wire-cut, to decrease the possibility of using removal attack, it has a restriction in selecting the wires to insert the crossbar. However, since we strongly twisted the CLN into the logic of the gates leading and preceding the selected wires, even removing the CLN using removal attack does not generate correct functionality. Hence, there is no limitation for wire selection in Full-Lock.

Fig. 6 demonstrates two simple examples that how Full-Lock inserts PLRs in the circuit. As shown in Fig. 6(a) and (b), the selected gates are highlighted in red, i.e. g_{14}, g_{15}, g_{16}, and g_{17}. Since these gates have no impact on each other, replacing them with PLR, including CLN and LUTs, does not generate any cycle in the design. However, Fig. 6(a) and (c) show that replacing the gates, which are highlighted in blue, i.e. g_2, g_5, g_7, and g_9, generates cycle in the circuit. Additionally, some of the leading gates of CLN is changed (negated), all highlighted in purple, i.e. $\overline{g_5}$, $\overline{g_{12}}$, g_{new} in Fig. 6(b), and $\overline{g_1}$, $\overline{g_6}$ in Fig. 6(c), which shows that how twisting leading gates into CLN is working. For instance, g_5 in Fig. 6(a), an *XOR*, has been replaced with $\overline{g_5}$ in Fig. 6(b), an *XNOR*. In this case, CLN will recover the functionality of this gate using key-configurable inverters that are embedded into CLN.

4 EXPERIMENTAL RESULTS

To show the efficiency of Full-lock, it is evaluated using different SAT-based attacks, including SAT for acyclic [14, 15], cycSAT for cyclic

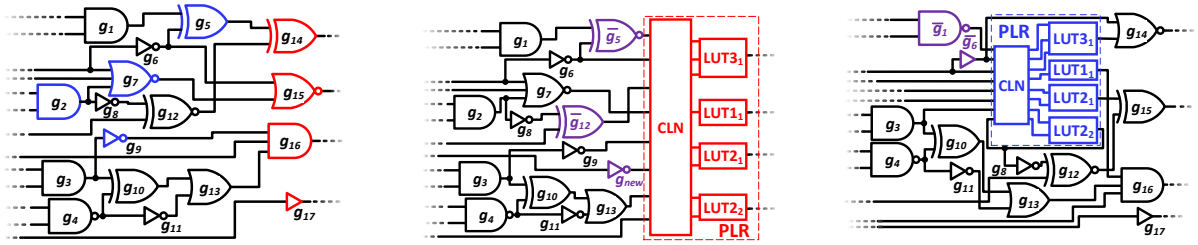

Figure 6: PLR Insertion Example: (a) Gate-level of Original Circuit. (b) Adding PLR and Negating leading Gates with (b) *Acyclic* Structure, (c) *Cyclic* Structure.

Table 2: SAT Execution Time on *shuffle*-based Blocking CLN for Different Sizes.

CLN Size (N)	4	8	16	32	64	128	256	**512**
Shuffle-based Blocking CLN								
SAT Iterations	7	8	9	13	15	27	28	**TO**
SAT Execution Time $(Seconds)$	0.01	0.04	0.22	1.22	17.4	154.7	2329.3	**TO**
Almost non-Blocking CLN								
SAT Iterations	14	18	25	32	**TO**	**TO**	**TO**	**TO**
SAT Execution Time $(Seconds)$	0.01	0.15	2.35	79.18	**TO**	**TO**	**TO**	**TO**

TO: *Timeout* = 2×10^6 *Seconds*

Table 3: Power/Area/Delay and SAT-based Resiliency of Blocking and almost non-Blocking CLNs for Different Sizes.

CLN	Area (um^2)	Power (nW)	Delay (ns)	SAT-Resilient
Shuffle ($N = 32$)	10.1	448.1	0.82	✗
$LOG_{32,3,1}$	17.8	2137.5	0.98	✗
Shuffle ($N = 64$)	22.8	1071.1	0.89	✗
$LOG_{64,4,1}$	**38.6**	**8451.4**	**1.06**	✓
Shuffle ($N = 128$)	50.8	2503.6	0.93	✗
Shuffle ($N = 256$)	113.6	5791.4	0.99	✗
Shuffle ($N = 512$)	**254.3**	**2308**	**1.04**	✓

Table 4: Execution Time of SAT Attack on Full-Lock with Different Sizes of PLRs.

Circuit	16×16				32×32		
	1	2	3	4	1	2	3
c432	28.8	1506.8	TO	TO	TO	TO	TO
c499	40.7	786.2	TO	TO	TO	TO	TO
c880	34.1	847.6	TO	TO	TO	TO	TO
c1355	64.9	1158.3	TO	TO	TO	TO	TO
c1908	45.5	1022.6	TO	TO	TO	TO	TO
c2670	79.8	1766.2	11791.5	184993.6	TO	TO	TO
c3540	67.2	429.6	7924.7	TO	TO	TO	TO
c5315	66.8	887.2	5748.1	TO	TO	TO	TO
c7552	90.3	1109.4	7638.6	66808.2	273367.4	TO	TO
apex2	38.4	633.1	TO	TO	TO	TO	TO
apex4	40.1	348.9	3670.9	18539.1	58467.6	380449.5	TO
i4	55.8	1604.8	TO	TO	TO	TO	TO
i7	84.6	1330.8	TO	TO	TO	TO	TO

TO: *Timeout* = 2×10^6 *Seconds*

[5], and AppSAT for approximate-based [6, 22], all implemented in C++, and were run on a Dell PowerEdge R620 equipped with Intel Xeon E5-2670 2.50GHz and 64GB of RAM.

4.1 Blocking *vs.* almost non-Blocking CLN

As was mentioned previously, Since not all but almost all permutations can be generated using non-blocking CLN, $LOG_{N,log_2(N)-2,1}$, it is far more resistant against SAT attack compared to a blocking network, especially with less power/performance/area overhead. We evaluate a shuffle-based CLN and an almost non-blocking with different sizes using SAT. As it can be seen in Table 2, increasing the CLN size, exponentially increases SAT execution time for either blocking or almost non-blocking. However, the SAT execution time is at least one order of magnitude higher in almost non-blocking. In addition, SAT is not able to break almost non-blocking CLN with a size larger than $N = 64$, however, for blocking CLN, it is easily broken for all sizes less than $N = 512$.

Since CLNs is the main part of PLRs as a SAT-hard instance that have medium length clauses while translated to CNF, the execution time of each iteration is significantly high, particularly for large sizes that cannot be broken using SAT. For blocking CLN with size $N = 512$ and non-blocking with size $N = 64$, after $2{\times}10^6$ Seconds, the number of completed iterations in SAT is only 7 and 5, respectively. It means that, on average, each iteration at least takes 2.8×10^5 Seconds in blocking and 4×10^5 in almost non-blocking CLNs.

Table 3 demonstrates power/area/delay of blocking and almost non-blocking CLNs for different sizes using Synopsys generic 32nm educational libraries. As it can be seen, the incurred overhead by the smallest almost non-blocking CLN, which is resilient against SAT ($N = 64$), is approximately one-third of the smallest SAT-resilient blocking CLN ($N = 512$) in terms of power consumption. Additionally, the overhead imposed by CLN is significantly low compared to area and power of even small-scale benchmark circuits.

4.2 Full-Lock Security Against Various Attacks

As was mentioned previously, in Full-Lock, the gates and their driving wires will be selected randomly to be replaced with PLRs. After selecting the required wires and their leading gates, Full-lock replaces them with PLR. Furthermore, the logic of some gates leading the selected wires will be negated. One or few PLR(s) can be added into the design based on the design criteria in terms of power/area/delay or security.

4.2.1 ***Security Against SAT-based Attack.*** Since random insertion is implemented for inserting PLRs in Full-Lock, it may generate cycle into the design. So, cycSAT has been used instead of SAT to support having potential cycles in locked circuits. In addition, to check resiliency against approximate-based attack, the cycSAT is enabled using AppSAT to extract the approximate key and corresponding error rate. Table 4 shows cycSAT execution time while different numbers of PLRs with different sizes have been inserted into ISCAS-85 and MCNC benchmark circuits. As it can be seen, for all circuits, having three PLRs contain 32×32 CLNs makes all locked circuit resistant against SAT. However, for each benchmark circuit, even smaller PLRs can break cycSAT.

In order to show the SAT-hardness of PLRs, we explore different sizes/numbers of PLRs to find the smallest size and the smallest number of PLRs (the lowest power/area overhead) that is required to provide resiliency against SAT. Table 5 shows the best solution of Full-Lock in terms of area/power/delay for each benchmark circuits. As shown, in all benchmark circuits, Full-Lock needs smaller/fewer PLRs compared to the required numbers of crossbar in Cross-Lock. As an instance, in *apex4*, only having two PLRs with a 32×32 CLN and another PLR with a 8×8 CLN can break SAT while its timeout

Table 5: PLRs Size in SAT-resilient Full-Lock compared to Cross-Lock.

Circuit	# Gates	# I/Os	Full-Lock	Cross-Lock [7]
c432	160	36/7	2×16×16 + 1×8×8	1×32×36
c499	202	41/32	2×16×16 + 1×8×8	1×32×36
c880	386	60/26	2×16×16 + 1×8×8	1×32×36
c1355	546	41/32	2×16×16 + 1×8×8	2×32×36
c1908	880	33/25	3×16×16	2×32×36
c2670	1193	157/64	1×32×32	3×32×36
c3540	1669	50/22	3×16×16 + 1×8×8	3×32×36
c5315	2307	178/123	3×16×16 + 2×8×8	3×32×36
c7552	3512	206/107	1×32×32 + 1×16×16	3×32×36
apex2	610	39/3	2×16×16 + 1×8×8	2×32×36
apex4	5360	10/19	2×32×32 + 1×8×8	11×32×36
i4	338	192/6	2×16×16 + 1×8×8	1×32×36
i7	1315	199/67	2×16×16 + 2×8×8	3×32×36

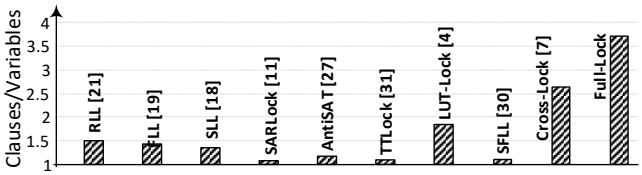

Figure 7: Average Clauses to Variables Ratio for Different Logic Locking Schemes.

is set to $2{\times}10^6$ Seconds. However, for the same circuit, Cross-Lock inserts 11 32×36 crossbars to make it resilient against SAT.

In addition, in order to show that PLRs are SAT-hard instances that significantly increase the number (M) and computational complexity (T^{Avg}_{DPLL}) of DPLL calls in each SAT iteration, we calculate the average clauses to variables ratio using MiniSAT for different logic locking schemes during deobfuscation. As it can be seen in Fig. 7, clauses to variables ratio in Full-Lock is 3.77. However, for all other methods this value is much lower. Across all logic locking schemes, LUT-Lock and Cross-Lock have higher clauses to variables ratio. Since LUT-Lock uses key-programmable LUTs for obfuscation, the translated CNF is MUX-based. However, since they have no back-to-back connection, the depth of MUX tree is low, which results in reducing the value of this ratio. The only technique with a close clauses to variables ratio is Cross-Lock, which is an interconnect locking with a tree of MUX. However, this ratio is almost 4 (3.77) in Full-Lock.

4.2.2 ***Security Against Removal Attack.*** As was mentioned previously, Cross-lock [7] as a layout-based interconnect locking scheme, used high-density cone-based selection strategies, such as k-cut and wire-cut, to decrease the possibility of using removal attack, which restricts in selecting the wires to insert the crossbar. However, since the logic of the gates leading each CLN can be negated, even having the possibility of removing CLN, and finding the functionality of LUTs does not produce correct functionality, which shows that Full-Lock has no vulnerability against removal attacks.

4.2.3 ***Security Against Algebraic Attack.*** CLN can be expressed as an affine transformation function of the data input X, of the form $y = A \cdot X + B$, where A is an $N \times N$ matrix and B is an $N \times 1$ vector, with all elements dependent on the key input. Although recovering A and B is not equivalent to finding the key input, it may enable the successful deobfuscation of CLN. Since Full-Lock replaces the the preceding gates of selected wires with LUTs, it cannot be transformed to an affine function. So, it is safe against SAT-based algebraic attacks.

5 CONCLUSIONS

In this paper, we proposed Full-Lock as a SAT-resistant logic locking solutions. Full-Lock creates a SAT-hard obfuscated netlist by replacing parts of logic and routing in the design with one or more sets of fully programmable logic and routing blocks (PLRs). The PLRs are designed to push the clauses to variables ratio in their CNF representation close to 4 to create insanely hard circuit SAT problems. With this mechanism, Full-lock SAT resistance comes from forcing the number of required recursive DPLL calls in each iteration of the SAT solver to a very large number, forcing each iteration to take a very long time to complete. Unlike previously SAT-hard solutions, Full-Lock exhibit high output corruption if a wrong key is used for activation. Finally, Since logic locking is twisted into interconnect locking in Full-Lock, it is resilient against removal and algebraic attacks.

REFERENCES

[1] P. C. Cheeseman, B. Kanefsky, and W. M. Taylor. 1991. Where the really hard problems are.. In *IJCAI*, Vol. 91. 331–340.
[2] F. A. Aloul *et al.* 2002. Solving difficult SAT instances in the presence of symmetry. In *Proc. of the Design Automation conf. (DAC)*. 731–736.
[3] G.-J. Nam *et al.* 2004. A comparative study of two Boolean formulations of FPGA detailed routing constraints. *IEEE Trans. Comput.* 53, 6 (2004), 688–696.
[4] H. M. Kamali *et al.* 2018. LUT-Lock: A Novel LUT-Based Logic Obfuscation for FPGA-Bitstream and ASIC-Hardware Protection. In *2018 IEEE Computer Society Annual Symp. on VLSI (ISVLSI)*. 405–410.
[5] H. Zhou *et al.* 2017. CycSAT: SAT-based attack on cyclic logic encryptions. In *Proc. of the Int'l Conf. on Computer-Aided Design (ICCAD)*. 49–56.
[6] K. Shamsi *et al.* 2017. AppSAT: Approximately deobfuscating integrated circuits. In *Hardware Oriented Security and Trust (HOST), IEEE Int'l Symp. on.* 95–100.
[7] K. Shamsi *et al.* 2018. Cross-Lock: Dense Layout-Level Interconnect Locking using Cross-bar Architectures. In *Proc. of the Great Lakes Symp. on VLSI (GLSVLSI)*. 147–152.
[8] K. Z. Azar *et al.* 2019. SMT Attack: Next Generation Attack on Obfuscated Circuits with Capabilities and Performance Beyond the SAT Attacks. *IACR Trans. on Cryptographic Hardware and Embedded Sys.s (TCHES)* 2019, 1 (2019), 97–122.
[9] K. Z. Azar *et al.* 2019. Threats on Logic Locking: A Decade Later. In *Proc. of the Great Lakes Symp. on VLSI (GLSVLSI)*. 6.
[10] M. Soos *et al.* 2009. Extending SAT solvers to cryptographic problems. In *Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. 244–257.
[11] M. Yasin *et al.* 2016. Sarlock: Sat attack resistant logic locking. In *Hardware Oriented Security and Trust (HOST), IEEE Int'l Symp. on.* 236–241.
[12] M. Yasin *et al.* 2017. Removal attacks on logic locking and camouflaging techniques. *IEEE Trans. Emerging Topics in Computing* 1 (2017), 1–1.
[13] M. Yasin *et al.* 2017. Security analysis of anti-sat. In *Design Automation conf. (ASP-DAC), Asia and South Pacific.* 342–347.
[14] P. Subramanyan *et al.* 2015. Evaluating the security of logic encryption algorithms. In *IEEE Int'l Symp. on Hardware Oriented Security and Trust (HOST)*. IEEE, 137–143.
[15] S. Roshanisefat *et al.* 2018. Benchmarking the capabilities and limitations of SAT solvers in defeating obfuscation schemes. In *IEEE Int'l Symp. on On-Line Testing And Robust System Design (IOLTS)*. 275–280.
[16] S. Roshanisefat *et al.* 2018. SRCLock: SAT-Resistant Cyclic Logic Locking for Protecting the Hardware. In *Proc. of the Great Lakes Symp. on VLSI (GLSVLSI)*. 153–158.
[17] D. Mitchell, B. Selman, and H. Levesque. 1992. Hard and easy distributions of SAT problems. In *AAAI*, Vol. 92. 459–465.
[18] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri. 2012. Security analysis of logic obfuscation. In *Proc. of the Design Automation conf. (DAC)*. 83–89.
[19] J. Rajendran, H. Zhang, C. Zhang, G. S. Rose, Y. Pino, O. Sinanoglu, and R. Karri. 2015. Fault analysis-based logic encryption. *IEEE Transactions on computers* 64, 2 (2015), 410–424.
[20] M. Rostami, F. Koushanfar, and R. Karri. 2014. A primer on hardware security: Models, methods, and metrics. *Proc. of the IEEE* 102, 8 (2014), 1283–1295.
[21] J. A. Roy, F. Koushanfar, and I. L. Markov. 2010. Ending piracy of integrated circuits. *Computer* 43, 10 (2010), 30–38.
[22] Y. Shen and H. Zhou. 2017. Double dip: Re-evaluating security of logic encryption algorithms. In *Proc. of the on Great Lakes Symp. on VLSI (GLSVLSI)*. 179–184.
[23] D.-J. Shyy and C.-T. Lea. 1991. Log/sub 2/(N, m, p) strictly nonblocking networks. *IEEE Trans. Commun.* 39, 10 (1991), 1502–1510.
[24] H. S. Stone. 1971. Parallel processing with the perfect shuffle. *IEEE trans. comp.* 100, 2 (1971), 153–161.
[25] G. Tseitin. 1968. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic* (1968), 115–125.
[26] P. Tuyls, G.-J. Schrijen, B. Škorić, J. V. Geloven, N. Verhaegh, and R. Wolters. 2006. Read-proof hardware from protective coatings. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. 369–383.
[27] Y. Xie and A. Srivastava. 2016. Mitigating sat attack on logic locking. In *Int'l Conf. on Cryptographic Hardware and Embedded Sys.s (CHES)*. 127–146.
[28] Y. Xie and A. Srivastava. 2017. Delay locking: Security enhancement of logic locking against ic counterfeiting and overproduction. In *Proc. of the Design Automation conf. 2017 (DAC)*. 9.
[29] X. Xu, B. Shakya, M. M. Tehranipoor, and D. Forte. 2017. Novel bypass attack and BDD-based tradeoff analysis against all known logic locking attacks. In *Int'l Conf. on Cryptographic Hardware and Embedded Sys.s (CHES)*. 189–210.
[30] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. V. Rajendran, and O. Sinanoglu. 2017. Provably-secure logic locking: From theory to practice. In *Proc. of the ACM SIGSAC Conf. on Computer and Communications Security (CCS)*. 1601–1618.
[31] M. Yasin, A. Sengupta, B. C. Schafer, Y. Makris, O. Sinanoglu, and J. V. Rajendran. 2017. What to lock?: Functional and parametric locking. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*. 351–356.
[32] A. Yeh. 2012. Trends in the global IC design service market. *DIGITIMES research* (2012).