SATConda: SAT to SAT-Hard Clause Translator

Rakibul Hassan, Gaurav Kolhe, Setareh Rafatirad, Houman Homayoun, Sai Manoj Pudukotai Dinakarrao
George Mason University, Fairfax, VA, USA Email: {rhassa2,gkolhe,srafatir,hhomayou,spudukot}@gmu.edu

***Abstract*—Logic obfuscation emerged as an efficient solution to strengthen the security of integrated circuits (ICs) from multiple threats including reverse engineering and intellectual property (IP) theft. Emergence of Boolean Satisfiability (SAT) attacks and its variants have shown to circumvent the security mechanisms such as obfuscation and a plethora of its variants. A plethora of advanced security defenses to thwart the SAT attacks are introduced. Despite the effectiveness, the imposed overheads in terms of area and power are unacceptably high. In contrast, our current work focuses on devising an iterative, dynamic and intelligent SAT-hard clause generator for a given SAT-prone problem, termed as SATConda. The SATConda is a SAT-hard clause generator that utilizes a bipartite propagation based neural network model. The utilized model comprises multiple layers of artificial neural networks to extract the dependencies of literals and variables, followed by long short term memory (LSTM) networks to validate the SAT hardness. The SATConda is trained with conjunctive normal form (CNF) of the IC netlist that are both SAT solvable and SAT-hard. Further, the SATConda is equipped with a SAT-clause generator to convert a CNF from satisfiable (SAT) to unsatisfiable (unSAT) with minor perturbation (which translates to minor overheads) so that the SAT-attack cannot decrypt the keys. To the best of our knowledge, no previous work has been reported on neural network based SAT-hard clause or CNF translator for circuit obfuscation. We evaluate our proposed SATConda's empirical performance against MiniSAT, Lingeling and Glucose SAT solvers on ISCAS'85 benchmark circuits.**

I. Introduction

With the semiconductor industries are inclining towards fabless business model i.e., outsourcing the fabrication to offshore foundries, to cope-up with the operational and maintenance costs, the hardware security threats are exacerbating [1]. This hardware threat could be in the form of intellectual property (IP) theft, hardware Trojan (HT) insertions, integrated circuit (IC) tampering, and over production and cloning [2], [3]. What is worse, the threat could occur during any phase of the IC production cycle ranging from design phase, fabrication phase or even after releasing the design to the market (in the form of side-channel attacks) [1], malware attacks [4], [5], and adversarial attacks [6].

To thwart the prevalent security threats, many hardware design-for-trust techniques have been introduced such as split manufacturing [7], IC camouflaging, and logic locking *a.k.a* logic obfuscation [8]. Among multiple aforementioned techniques, logic locking can thwart the majority of the attacks at various phases in the IC Production chain [9]. This is because logic locking requires the correct keys to unlock the true functionality of the design. Additionally, as a part of the post-manufacturing process, the activation of IC (i.e., providing correct keys) will be accomplished in a trusted regime to hide the functionality from the untrusted foundry and other attacks. Having key-programmable gates allows the designer or user to control the functionality using these key inputs.

Although logic locking schemes enhance the security of the IP, the advent of Boolean satisfiability (SAT) based attack [10], also known as "oracle-guided" threat model shows that by applying a few stimuli to the design and analyzing the output, the key value and functionality of an IC could be extracted in the order of a few minutes. To implement SAT attack, the attacker needs access to (a) an obfuscated netlist of IC (obtained after de-layering fabricated IC or constructed from layout), and (b) a functional/activated IC, to which the attacker can apply inputs and monitor the output and functionality. The extracted netlist is converted into a conjunctive normal form (CNF[1]), fed to a SAT solver to determine the keys (assignment to each boolean variable in the CNF) to decrypt and reverse engineer the IC/IP. It has been seen that modern SAT solvers can solve a SAT-problem with up to million variables [11].

To mitigate SAT attack several logic locking [9] techniques have been proposed. A recently proposed mechanism on logic locking was presented to mitigate SAT attack by introducing an additional logic block that makes SAT attack computationally infeasible [12]. Recent literature reported SPS attack [13] against Anti-SAT defense [12] which can break the Anti-SAT defense within few minutes.

One of the major challenges in adopting the existing defenses such as obfuscating large number of gates is the overheads imposed in terms of area and power [14]. Previous works [12], [13] consider developing Anti-SAT solutions through embedding different metrics (properties of netlist that cannot be translated into CNF) or through heuristic intuitions. Such defenses involve challenges including complexity, incompleteness and high probability to exclude parameters that were not explored in literature. To address these concerns, we introduce SATConda[2], a neural network with bipartite message passing mechanism that can automatically learn and determine the properties of a CNF and distinguish SAT and unSAT problems. To perform this, we trained our neural network model with both SAT and unSAT CNFs. Thereby the network learns the SAT and and unSAT clauses and provides the SAT and unSAT distributions. This learnt features are further utilized to form a CNF generator that can convert the provided SAT prone CNF into unSAT (SAT-hard) through minimal modification to the netlist such as flipping a literal (through addition of inverter gate or using XNOR instead of XOR are some of the naïve possibilities) in a clause of CNF

[1]A CNF is a conjunction (i.e AND) of one or more clauses, where a clause is a disjunction (i.e OR) of literals.

[2]SATConda is a SAT framework developed with Anaconda tool.

978-1-7281-4207-4/20/$31.00 ©2020 IEEE

i.e., converts the SAT distribution to unSAT distribution by learning the distributions yet preserving the functionality. The amount of clauses added or the perturbations introduced can be controlled to determine the trade-off between overheads and security. The two main contributions of this work are:

- we exploit message passing neural network technique (MPNN) to convert a CNF file(extracted from the circuit netlist) of an IC/IP from SAT to unSAT through SATConda. To the best of our knowledge no previous work has taken the advantage of message passing algorithm to learn how to translate a SAT satisfiable CNF file to an unsatisfiable one and extract the distributions.
- We successfully defend the SAT-attack by introducing an unSAT block and encrypting that block the the original circuit. Using SATConda, we showcase the existing obfuscation schemes can be made robust with minimal modifications.

The idea of exploring deep neural network in circuit obfuscation, especially for converting SAT to unSAT is unexplored and is novel. We evaluated our proposed model using ISCAS'85 benchmark circuits. We also evaluated the translated obfuscated file with three different state-of-the-art SAT solvers. Our translated CNF file remains unsatisfiable in all of them. In addition to evaluating the SAT hardness, we have evaluated the area and power overheads incurred with additional security deployment through SATConda. The proposed SATConda introduces an area and power overhead of <5% on an average for ISCAS'85 benchmarks to make them unSAT, which is significantly smaller compared to other existing SAT defense techniques.

II. Background

Here, we discuss the basic information regarding the logic locking and the SAT attack.

A. Logic Locking

Logic locking mechanism is implemented in a design by adding additional gates a.k.a "key-gates" to secure the circuit (IC/IP) by inducing the randomness in the observable output. To achieve the desired output from the design, all the key-gates must be set to their proper input. Any incorrect insertion to any of the key-gates leads to incorrect output. So an attacker needs to know the correct assignment of those keys-gates to decode the actual functionality of the design.

Figure 1a depicts the original circuit and the Figure 1b shows a logic-locked circuit of the C17 circuit from ISCAS'85 benchmark. The original circuit consists of five inputs with six NAND gates and two outputs. The encryption is done by adding three additional gates, termed as key-gates. For Figure 1b, if one assigns (k2, k1, K0) as (1, 0, 1), only then the circuit will function as desired. Complexity of determining the key inputs will increase exponentially with the number of key-gates when attacker performs brute-force search.

B. SAT Attack

Despite logic locking being successful in securing the IC from reverse engineering, the Boolean Satisfiability-based

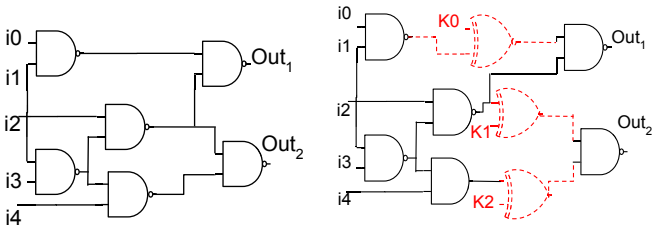


Fig. 1: Logic locking on c17 benchmark circuit. (a) Original Circuit Design, (b) Encrypted circuit with additional key-gates (dotted lines/gates) [15]. Desired circuit behavior is achieved when (k2, k1, k0) = 101

attack commonly known as SAT-attack [10] proposed in 2015 has successfully broke six state-of-the-art logic-locking defense mechanism proposed. The results have shown that the circuits can be successfully deobfuscated despite deploying logic locking solutions within few seconds. To mitigate this SAT-based attack(s), researchers have proposed several counter-measures from time to time and new attack model(s) has also been proposed to counterfeit that defense. We present a glimpse of SAT-attack here:

1) Attack Model: The attack model was established under the assumption that the attacker has

- a gate-level netlist extracted from the obfuscated IC.
- an activated functional chip for observing the output pattern for a given input.

2) Attack Methodology: SAT attack generates a carefully crafted input patterns and observed the corresponding output from the activated functional chip. The goal of SAT attack is to eliminate incorrect key-values at each iteration by observing the outputs for a given pair of inputs. This input/output pairs are called Discriminating Input Patterns (DIP). By observing this DIP, SAT-attack iteratively eliminates numerous wrong keys and this step is iterated until it eliminates all the wrong keys and determines the correct key.

III. Related Work

Several techniques have been proposed to defend and secure the design from reverse engineering threats from SAT-attack. Here, we review some of the relevant defense techniques to thwart SAT attacks.

EPIC [16] inserts XOR/XNOR gates randomly as key-gates to the original netlist to achieve a locked-netlist. One can decrypt the key-values by inspecting the XOR/XNOR gates and configuring them as buffers or inverters using the key-inputs [8].

An additional circuit block, Anti-SAT block [12] was proposed to add with the encrypted circuit to mitigate the SAT attack. They showed that the time required to expose all the key-values is an exponential function of the key gates in the Anti-SAT block. By making the key-size large enough, the SAT attack becomes computationally complex and infeasible.

Similar work has been reported in SARLock [9]. In this work they proposed a SARLock block with the encypted

circuit that maximizes the number of DIPs, thus making the SAT-attack runtime exponential with the number of secret-key bits. This method was shown to be vulnerable to a SAT-based attack, double-DIP attack reported in [17].

In another work, advanced encryption standard (AES) circuit [8] was proposed into an encrypted circuit to prevent the SAT-attack. By adding this AES circuit, [8] makes the attack computationally intractable as the attacker cannot retrieve the input of the AES block by observing the output patterns. However, this techniques suffers from large area overhead since implementation of AES circuit requires for significant number of logic gates [12].

In [18] a SAT-resilient cyclic obfuscated circuit design was proposed by adding dummy paths to the encrypted circuit which make the combinational loop non-reducible. This defense mechanism was prone to another type of SAT-based attack named CycSAT [19] which can effectively decrypt the cyclic encryption.

Unlike the existing defenses discussed above, our proposed methodology utilizes a neural network and extracts the feature variables automatically to learn the SAT and unSAT distributions. These distributions will be further utilized to translate a CNF from SAT to unSAT by adding additional circuitry with least overhead compared.

IV. SATConda: SAT to UNSAT Translator

In this section, we present the overview of our proposed SATConda. SATConda is a hybrid Message Passing Neural Network (MPNN) [20] framework that is able to learn the SAT-based deobfuscation by message passing across the literals and clauses for the CNF. SATConda learns at a clause-by-clause basis rather than all clauses at once. This learning aids to learn the distribution of SAT and unSAT clauses. Further, the proposed SATConda is also equipped with a mechanism to perturb the netlist i.e., update the CNF in a slight manner to convert the SAT problem (distribution) into a unSAT (distribution). Figure 2 depicts the operational flow of the proposed framework.

A. Generating Training Data

In order to train the MPNN of SATConda, we generate a pair of SAT problems, one of which is satisfiable (SAT) and the other one is unsatisfiable (unSAT). To do so, we start with a random clause generator that generates clauses with varying size (number of literals in each clause). We query a SAT solver (miniSAT solver [21] in our case) to check whether the clause is satisfiable or not. Thus, the CNF clauses and its corresponding SAT hardness i.e., SAT or unSAT is utilized to train the MPNN. During the training phase, the utilized MPNN extracts the features from the CNF and learns the distribution of literals and their relationship from the SAT hardness perspective. For an efficient training, we train with a pair of SAT-problems that have difference in one literal with one being satisfiable and the other one being unsatisfiable, as shown in bottom row of Figure 2.

B. Neural Network Architecture

We encode the IC obfuscation problem as a SAT problem, which is further converted as an hierarchical undirected graph, where each clause is represented with one node and each literal inside a clause is represented as one node.

SATConda consists of a literal vector (L_{init}) and a clause vector (C_{init}) extracted from the CNF, which is fed to a three-layer fully connected MPNN $(L_{msg}, C_{msg}, L_{sat})$, and a two-layer long-short term memory (LSTM) (C_u, L_u) network. Hidden states for literals and clauses are denoted by L_h and C_h respectively. An adjacency Matrix (M) defines the relation between literals and clauses. This relationship between literals and clauses are established by connecting edges among them.

Message is passed back and forth along the edges of the network [20]. At first, a message is passed to a clause from its neighbouring literals and the clause gets updated. In the next step, a literal gets message from its neighbouring clause(s) and also from its complements. This message passing event occurs back and forth until the model refines a vector space for every node. The MPNN takes L_{init} and C_{init} as its input and passes its output (L_{msg}, C_{msg}) to the LSTM network which updates the literals L^{t+1} and clauses C^{t+1} at each iteration, as follows:

$$C_u([M^T L_{msg}(L^t)]) \rightarrow C^{t+1} \tag{1}$$

$$C_u([C_h^t]) \rightarrow C_h^{t+1} \tag{2}$$

$$L_u([Flip(L^t), MC_{msg}(C^{t+1})]) \rightarrow L^{t+1} \tag{3}$$

$$L_u([L_h^t]) \rightarrow L_h^{t+1} \tag{4}$$

L_{sat} votes SAT or unSAT for a particular literal and taking the average vote of all the literals after T iteration, SATConda predicts whether a problem is SAT or unSAT.

This message-passing architecture lets the SATConda to learn the features that can distinguish the SAT solvable CNFs from unSAT CNFs i.e., learn the distribution of SAT and unSAT CNFs.

Post training, the proposed SATConda first verifies for the SAT hardness i.e., SAT solvable or unSAT for a given CNF. If the CNF is SAT-hard, it terminates. However, if it is SAT solvable, the SATConda starts perturbing the literals (such as flipping or adding auxiliary variables) beginning from the last clause of a given CNF, thus translating a given CNF from SAT solvable to SAT-hard.

C. SAT to UnSAT Translator

After training our model, the netlist in the form of a CNF is provided as an input (after extracting from the circuit netlist) to verify SAT-hardness and convert it to SAT-hard. The design flow of SATConda is presented in Fig. 2. At first, our SATConda checks whether this CNF file is satisfiable or not by querying the state-of-the-art SAT solver (MiniSAT [21]). If the CNF file is not satifsiable (unSAT) then SATConda terminates. On the other hand, if the CNF file is satisfiable, then the SATConda invokes the clause generator to add additional clauses or perturb the CNF based on the learnt distribution from the neural network to make the CNF unSAT.

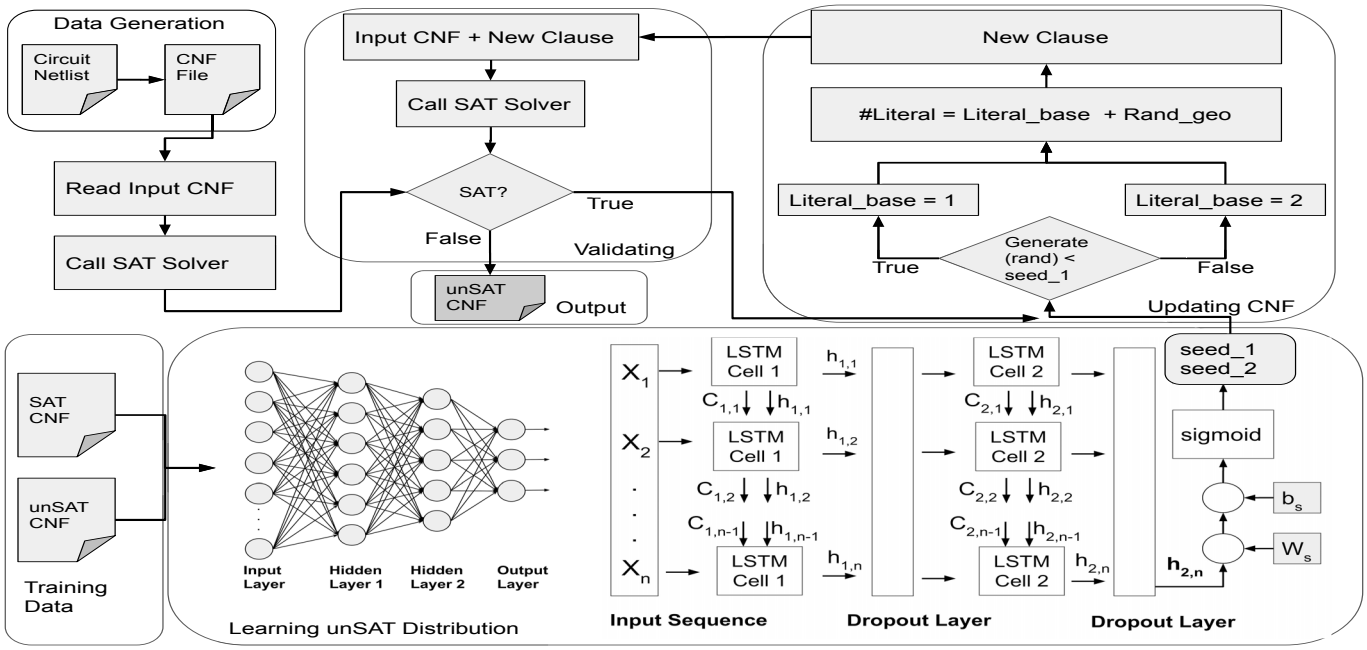


Fig. 2: Architecture of proposed SATConda.

Additional clause generation process starts with getting the seed values from the trained neural network and are passed to $seed_1$ and $seed_2$ variables (see Fig. 2). The values of the $seed_1$ and $seed_2$ are determined by fitting the learnt distribution to Bernoulli and Geometric distributions, respectively. Then, the clause generator generates a decimal number randomly between 0 and 1 and checks whether the randomly generated number is greater than the $seed_1$ or not. If it is greater than the $seed_1$ value then another variable $Literal\_base$ is assigned to one otherwise two. The variable $Literal\_base$ initializes the number of literals that a newly generated clause will have. Then the generator draws samples from a geometric distribution at a probability of $seed_2$ and assigns that value to a variable($Rand\_geo$). Adding both the $Literal\_base$ and $Rand\_geo$ we have the length of a new clause (how many literals in the clause). After that, SATConda samples a variable from the variable list, present in the original CNF file, without replacement. This variable sampling is done with a 0.5 probability of negating that variable or not. For example if a CNF file consists of 10 variables and SATConda picks the 9th variable for clause generation, then there is a 50% chance of taking that variable as "9" or as "-9". This perturbation aids in minimizing the imposed overheads and observe the SAT-hardness, verified through minSAT solver.

When a new clause is generated then SATConda append that new clause to the original CNF. After appending the new clause, SATConda again queries the SAT-solver to determine whether the addition of the new clause or perturbed variable makes the CNF SAT-hard. If the CNF still remains satisfiable then SATConda keeps adding or perturbing clauses until the CNF becomes SAT-hard. Thus, adding or perturbing the clauses makes the CNF i.e., IC netlist SAT-hard.

D. Summary of SATConda

The clause generation processes mentioned above follow the Algorithm presented in Algorithm 1. From Algorithm 1 it can be observed that the solvability of a given CNF file is checked first, as given in Line 2 of Algorithm 1. Given the condition that the CNF file is satisfiable, the algorithm generates a new clause with varying size and adds this newly generated clause with the previous clauses, as shown in Line 13 of Algorithm 1. Again the SATConda checks for the SAT solvability for all the clauses and keeps adding clauses until the CNF file becomes unSAT (as in Line 15). Once the CNF becomes SAT-hard, the algorithm provides the modified SAT-hard CNF as output, given in Line 34 of Algorithm 1.

V. Experiments and Results

In this section we describe the experimental tools used for performing our experiment and evaluate the impact of SATConda in terms of SAT hardness and the incurred overheads.

A. Experimental Setup

In this work, we used the MPNN model from [22] to get the seed required for our random clause generator. We trained the model with 10,000 CNF files. Out of them 5000 were SAT and 5000 were unSAT. The training-cost is 0.6930 and the validation cost is 0.6932, which is sufficiently good enough and converging to ensure that the model generalizes well with no over or under-fitting. We evaluated the performance on ISCAS'85 benchmark circuits shown in Table I. We verified the satisfiability of CNF files using three different SAT solvers, MiniSAT [21], Lingeling [23], and Glucose [24]. The rationale for choosing these solvers is that these solvers form basis for numerous SAT attacks crafted for deobfuscation in the

Algorithm 1 SATConda Algorithm

Input : $solve\_clauses, seed\_1, seed\_2$
Output : $unSAT - cnf$

$is\_sat := solve(solve\_clauses)$
if $is\_sat == True$ **then**
  while true **do**
    $rand := gen\_decimal(0 - 1)$
    if $rand < seed_1$ **then**
      $lit\_base := 1$
    else
      $lit\_base := 2$
    end if
    $rand\_geo = rand.geometric(seed\_2)$
    $literal = lit\_base + rand\_geo$
    $new\_clause := generate\_clause(n\_var, literal)$
    $solve\_clauses+ = new\_clause$
    $is\_sat := solve(solve\_clauses)$
    if $is\_sat == True$ **then**
      $solve\_clauses+ = new\_clause$
    else
      $break$
    end if
  end while
  $solve\_clauses+ = new\_clause$
end if
function GENERATE_CLAUSE(n_var, $literal$)
  $array\_size := minimum(n\_var, literal)$
  $clause\_gen := gen.rand\_array(n\_var, array\_size)$
  $rand := gen\_decimal(0\ 1)$
  if $rand < 0.5$ **then**
    $new\_clause := clause\_gen + 1$
  else
    $new\_clause := -(clause\_gen + 1)$
  end if
  return new_clause
end function
$unSAT - cnf := solve\_clauses$

past few years. The area and power overheads are calculated using Synopsys Design Compiler, Version: L-2016.03-SP3. SAED90nm EDK Digital Standard Cell Library is used for logic synthesis. All the experiments were performed on a server with 8-core Intel Xeon E5410 CPU, running at 2.33 GHz, with 16 GB RAM. The operating system installed on the server is CentOS Linux 7.

B. Evaluation

Here, we present the evaluation in terms of SAT-hardness and the overhead analysis in addition to our empirical findings of SATConda.

1) SAT-hardness: Table II presents the satisfiability of the ISCAS'85 benchmark circuits before and after the conversion through SATConda. It can be seen from this table that all the original benchmark circuits were satisfiable initially with all the three SAT-solvers [21], [23], [24]. This indicates that an attacker could perform a SAT-attack with any of these SAT-solvers and reverse engineer the IP/IC. Table II also shows that once the IC/IP design is converted using SATConda, it becomes SAT-hard, meaning none of the three experimented SAT-solvers could solve a satisfying assignment for the literals. This clearly proves that the SATConda can effectively translate a SAT problem into a SAT-hard problem.

2) Overhead Analysis: In addition to SAT-hardness, we evaluate the imposed overheads through the conversion. Table III reports the area and power overhead (in terms of %) for the best-fit model achieved in our experiment. From Table III it is observed that except for the c17 and c1355 circuits the area overhead is around 5% with the proposed SATConda

TABLE I: ISCAS'85 Benchmark Circuits

Circuit Name	#Inputs	#Outputs	#Gates
c17	5	2	6
c432	36	7	160
c499	41	32	202
c880	60	26	383
c1355	41	32	546
c1908	33	25	880
c2670	233	140	1193
c3540	50	22	1669
c5315	178	123	2307
c7552	207	108	3512

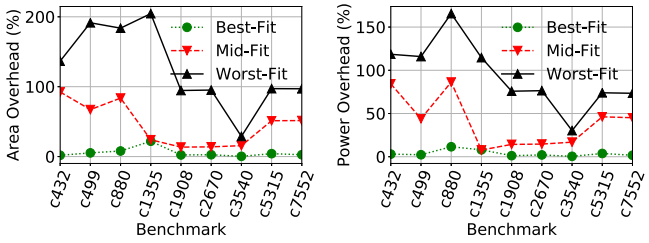
TABLE II: SATConda evaluation on different SAT-solvers

Circuit Name	miniSAT		Lingeling		Glucose	
	Before Conversion	After Conversion	Before Conversion	After Conversion	Before Conversion	After Conversion
c17	✓	✗	✓	✗	✓	✗
c432	✓	✗	✓	✗	✓	✗
c499	✓	✗	✓	✗	✓	✗
c880	✓	✗	✓	✗	✓	✗
c1355	✓	✗	✓	✗	✓	✗
c1908	✓	✗	✓	✗	✓	✗
c2670	✓	✗	✓	✗	✓	✗
c3540	✓	✗	✓	✗	✓	✗
c5315	✓	✗	✓	✗	✓	✗
c7552	✓	✗	✓	✗	✓	✗

✓= Corresponding CNF was satisfiable by the SAT solver
✗= Corresponding CNF was unsatisfiable by the SAT solver

TABLE III: Report on Area and Power Overhead

Circuit Name	Area Overhead (%)				Power Overhead (%)			
	SAT-Conda	LUT + LUT [14]	SAR-Lock + SLL [9]	SLJI + TI [8]	SAT-Conda	LUT + LUT [14]	SAR-Lock + SLL [9]	SLJI + TI [8]
c17	29.32	-	-	-	32.86	-	-	-
c432	1.85	-	-	21	3.03	-	-	70
c499	5.27	-	-	-	2.29	-	-	-
c880	7.97	-	-	-	11.54	-	-	-
c1355	21.96	-	-	-	8.01	-	-	-
c1908	2.31	-	-	-	1.21	-	-	-
c2670	2.62	580	-	-	2.09	96	-	-
c3540	0.35	-	-	-	0.47	-	-	-
c5315	4.16	-	8	3	3.76	-	9	12
c7552	2.63	265	6	21.5	1.72	14	6	20



(a) (b)

Fig. 3: Overhead analysis for ISCAS'85 benchmark circuit. (a) Area overhead for different cases, (b) Power overhead for different cases

technique. The area overhead for c17 is 29.32% and for c1355 is 21.96%. In case of the power overhead, SATConda adds around 5% power overhead to all the circuits except for c17 and c880. The c17 has a power overhead of 32.86% and c880 has a power overhead of 11.54%.

In addition, we compare our SATConda induced overheads with the state-of-the-art key-gate based logic-locking obfuscation techniques such as LUT+LUT [14], SARLock+SLL [9], and SLJI+TI [8] in Table III. We report the area and power overheads as reported in the respective papers and avoid comparison by speculating the results, as a result, they are left blank. Our proposed design is not key-gate based, rather addition of auxiliary gates to the original design or perturbing the existing design. Our model requires $160.95\times$, $2.06\times$, and $6.26\times$ lower area overhead and shows 63.46%, 91.65%, and 96.53% power saving on average compared to LUT+LUT, SARLock+SLL and SLJI+TI techniques, respectively.

3) Analysis: As the clauses generated are based on the model learnt by the MPNN, different training data can lead to different learnt models and different seed values. We analyze the impact of the seed on the overheads here, as all of them lead to unSAT in terms of security. We compare three different models that we achieved from SATConda. We named them as best-fit model (for $seed_1 = 1.0$ and $seed_2 = 1.0$ - leading to lowest overhead), mid-fit model(for $seed_1 = 0.5$ and $seed_2 = 0.5$), and worst-fit model(for $seed_1 = 0.3$ and $seed_2 = 0.4$). Figure 3 shows the overhead compared to the base circuit with the three different model fits for the benchmark circuits. Figure 3a depicts the area overhead (%) and Figure 3b shows the power overhead (%) for different models. On average, the area and power overhead ranges from 5.45% to 125.44% and 3.79% to 93.81% respectively, irrespective of the model fit used for generating or perturbting the clauses, which is still significantly lower than [14]. As can be seen that the worst-fit model gives relatively large area overhead for all the benchmark circuits. The reason behind this is the worst-fit model adds a significant number of clauses to the original CNF file. Similar trend can be observed for power overhead from Figure 3b where the best-fit model outperforms the other two models with a significant margin.

VI. Conclusion

In this work we introduce a neural network based SAT-hard problem generator, SATConda, towards an intelligent and stronger logic-locking based obfuscation for hardware security. We have observed a successful conversion of SAT to unSAT through SATConda for multiple variable size CNFs. We evaluate our framework on the state-of-the-art benchmarks such as ISCAS'85 and validate with three traditional SAT solvers regarding the SAT hardness. Our model shows $160.95\times$, $2.06\times$, and $6.26\times$ area savings and 63.46%, 91.65%, and 96.53% power savings on average compared to LUT+LUT, SARLock+SLL, and SLJI+TI based techniques, respectively.

References

[1] M. A. Mak, "Trusted defense microelectronics: future access and capabilities are uncertain," GOVERNMENT ACCOUNTABILITY OFFICE WASHINGTON DC, Tech. Rep., 2015.

[2] R. Karri *et al.*, "Trustworthy hardware: Identifying and classifying hardware trojans," *Computer*, vol. 43, no. 10, pp. 39–46, 2010.

[3] M. Rostami *et al.*, "A primer on hardware security: Models, methods, and metrics," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1283–1295, 2014.

[4] S. Shukla *et al.*, "Stealthy malware detection using rnn-based automated localized feature extraction and classifier," in *Int. Conf. on Tools with Artificial Intelligence*, 2019.

[5] S. Shukla *et al.*, "Rnn-based classifier to detect stealthy malware using localized features and complex symbolic sequence," *Image*, vol. 90, no. 96, pp. 102–108.

[6] S. M. P. Dinakarrao *et al.*, "Adversarial attack on microarchitectural events based malware detectors," in *Proceedings of the 56th Annual Design Automation Conference*, 2019.

[7] J. J. Rajendran *et al.*, "Is split manufacturing secure?" in *Conf. on Design, Automation and Test in Europe*, 2013.

[8] M. Yasin *et al.*, "On improving the security of logic locking," *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 9, pp. 1411–1424, 2015.

[9] M. Yasin *et al.*, "SARLock: SAT attack resistant logic locking," in *Int. Symp. on Hardware Oriented Security and Trust*, 2016.

[10] P. Subramanyan *et al.*, "Evaluating the security of logic encryption algorithms," in *Int. Symp. on Hardware Oriented Security and Trust*, 2015.

[11] J. Franco and J. Martin, "Handbook of satisfiability frontiers in artificial intelligence and applications," 2009.

[12] Y. Xie and A. Srivastava, "Mitigating sat attack on logic locking," in *Int. Conf. on Cryptographic Hardware and Embedded Systems*, 2016.

[13] M. Yasin *et al.*, "Security analysis of anti-sat," in *Asia and South Pacific Design Automation conf.*, 2017.

[14] G. Kolhe *et al.*, "On custom LUT-based obfuscation," in *Great Lakes Symp. on VLSI*, 2019.

[15] S. Dupuis and M.-L. Flottes, "Logic locking: A survey of proposed methods and evaluation metrics," *Journal of Electronic Testing*, pp. 1–19, 2019.

[16] J. A. Roy *et al.*, "Ending piracy of integrated circuits," *Computer*, vol. 43, no. 10, pp. 30–38, 2010.

[17] Y. Shen and H. Zhou, "Double dip: Re-evaluating security of logic encryption algorithms," in *Great Lakes Symp. on VLSI*, 2017.

[18] K. Shamsi *et al.*, "Cyclic obfuscation for creating sat-unresolvable circuits," in *Great Lakes Symp. on VLSI*, 2017.

[19] H. Zhou *et al.*, "Cycsat: Sat-based attack on cyclic logic encryptions," in *36th Int. Conf. on Computer-Aided Design*, 2017.

[20] J. Gilmer *et al.*, "Neural message passing for quantum chemistry," in *Int. conf. on Machine Learning*, 2017.

[21] N. Sorensson and N. Een, "Minisat v1. 13-a sat solver with conflict-clause minimization," *SAT*, vol. 2005, no. 53, pp. 1–2, 2005.

[22] D. Selsam *et al.*, "Learning a SAT solver from single-bit supervision," *arXiv preprint arXiv:1802.03685*, 2018.

[23] A. Biere, "Lingeling, plingeling and treengeling entering the sat competition 2013," 2013.

[24] G. Audemard and L. Simon, "GLUCOSE: a solver that predicts learnt clauses quality," *SAT Competition*, 2009.