LOOPLock: Logic Optimization-Based Cyclic Logic Locking

Hsiao-Yu Chiang, Yung-Chih Chen, De-Xuan Ji, Xiang-Min Yang, Chia-Chun Lin, and Chun-Yao Wang, *Member, IEEE*

***Abstract*—SAT Attack, CycSAT, and Removal Attack have demonstrated their abilities to break most existing logic locking methods. In this article, we propose a new cyclic logic locking method to invalidate these attacks simultaneously. Our main intention is to create noncombinational cycles to lock a circuit. Specifically, the noncombinational behavior in the noncombinational cycles that is unobservable at the primary outputs (POs) needs to be preserved when the correct key-vector is fed to resist CycSAT, and the noncombinational behavior in the noncombinational cycles affecting POs needs to be preserved when the incorrect key-vector is fed to invalidate SAT Attack. Furthermore, some nodes will be removed when applying our locking method, which is able to defend Removal Attack. The experimental results show the effectiveness and low area overhead of the proposed method.**

***Index Terms*—Cyclic logic locking, CycSAT, hardware security, logic optimization, SAT Attack.**

I. Introduction

THE GLOBALIZATION of IC design and manufacturing flow enables fabless IC companies to purchase intellectual property (IP) cores from third-party vendors for reducing SoC design effort and send their designs to foundries for fabrication. Although this model is time and cost-effective, the IC/IP providers could expose their designs to threats, such as piracy, overproduction, and counterfeiting. To deal with this issue, a protection technique, called *logic encryption/locking* [8], was proposed recently to enable IC designers to get control on their designs. It works by inserting extra *key-gates* like XOR/XNOR-based gates and the primary inputs (PIs), called *key inputs*, into a design to hide the functionality. A locked design is functionally correct only if a correct secret key, known to the designer only, is fed into the *key inputs*. With increasing the number of *key-gates*, the difficulty of the decryption grows exponentially. Several logic locking methods have been proposed and they are primarily different in the locations for inserting the extra logic gates and key inputs [7], [8], [11], [21]–[23], [27].

Conversely, *logic decryption* is an attacking technique that identifies the secret key of a locked design, and is an important manner for evaluating the security of logic locking methods. Satisfiability (SAT) Attack [15] is an attacking method based on Boolean satisfiability algorithms to decrypt several traditional locking methods. The SAT Attack constructs a miter-like circuit with two copies of the locked netlist, which shares the same PIs but has independent key inputs, to identify the differences between their primary outputs (POs). Then SAT solvers will be called to find the distinguishing input patterns (DIPs) iteratively. The DIPs are used to filter out all the incorrect key-vectors, which is the main concept of SAT Attack. When there is no DIP found by the SAT solver, all the incorrect key-vectors are filtered out and the attack is successful. Because a DIP can rule out multiple incorrect key-vectors, once SAT Attack decrypts the locked circuits quite efficiently. Some attacking methods [1], [6], [9], [14], [16]–[20], [28] based on the SAT Attack have been proposed in different purposes against locking methods.

In spite of the effective decryption from SAT Attack, SAT Attack still has some shortcomings. For example, if the DIP in SAT Attack can rule out multiple incorrect key-vectors once, SAT Attack is very efficient. However, for some new locking methods [22], [23], the DIP can only rule out one incorrect key-vector once, which lowers the performance of the SAT Attack, and SAT Attack acts like a brute force method under this situation.

One of the objectives in these methods was to drag out the time required in the process of the SAT Attack [22], [23]. However, these methods have vulnerabilities to Removal Attack [25], [26], which is a method locating and then removing or bypassing security structures to restore the original functions. Removal Attack has the advantage of being able to identify the locked structures in the circuit. The positions of the locked structures are obtained by calculating the probability of a signal value of 1 and 0 in the circuit. However, even if Removal Attack can find and remove all the locked structures, it may still face a challenging scenario. That is, when the defender also removes a certain subcircuit from the original circuit during the locking process, Removal Attack

Manuscript received July 12, 2019; revised October 2, 2019; accepted November 24, 2019. Date of publication December 17, 2019; date of current version September 18, 2020. This work was supported in part by the Ministry of Science and Technology of Taiwan under Grant MOST 107-2221-E-155-046, Grant MOST 108-2221-E-155-047, Grant MOST 106-2221-E-007-111-MY3, and Grant MOST 108-2218-E-007-061. This article was recommended by Associate Editor R. Karri. *(Corresponding author: Chia-Chun Lin.)*

Hsiao-Yu Chiang, De-Xuan Ji, Xiang-Min Yang, Chia-Chun Lin, and Chun-Yao Wang are with the Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan (e-mail: johnny19941007@gmail.com; jidx1994@gmail.com; yhm19930125@gmail.com; chiachunlin@gapp.nthu.edu.tw; wcyao@cs.nthu.edu.tw).

Yung-Chih Chen is with the Department of Computer Science and Engineering, Yuan Ze University, Taoyuan 32003, Taiwan (e-mail: ycchen.cse@saturn.yzu.edu.tw).

Digital Object Identifier 10.1109/TCAD.2019.2960351

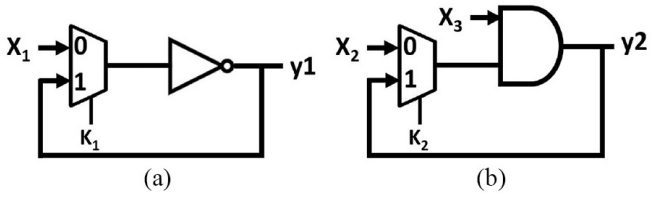

Fig. 1. Example demonstrating the cyclic logic locking. (a) Inverter locked into a cyclic structure. (b) AND gate locked into a cyclic structure.

cannot recover the original circuit by just removing the locked structures after the unlocking process.

Another known characteristic that SAT-based attacking methods usually cannot well deal with is transitional data, like delay or timing. Hence, a timing-based logic locking method with a tunable delay key-gate (TDK) scheme called delay logic locking (DLL) was proposed in [21]. However, there are other methods based on the SAT Attack model to simulate the behavior of the delay-locked logic to crack the DLL [1], [6].

Cyclic logic locking [13] is another defense technique proposed against the SAT Attack. This technique is to destroy the directed acyclic graph (DAG) structure by randomly generating feedback loops (cycles) in the circuit. Despite the superior effectiveness of cyclic logic locking, this technique was still cracked by CycSAT [9], [20], [28]. CycSAT consists of two versions, CycSAT-I and CycSAT-II, for different purposes.

CycSAT-I aims to find a correct key-vector that breaks the cycles in the circuit and restore to the original acyclic circuit. For achieving this, CycSAT-I needs to compute the "no structural cycle" condition on the key values and find a correct key-vector to break the cycles. For example, Fig. 1(a) shows a cyclic locked circuit, and its original circuit is an inverter. To break the cycle in Fig. 1(a), CycSAT-I will compute the no structural cycle condition for the cycle in Fig. 1(a), and obtain the correct key value of $\boldsymbol{K_1} = 0$, then the resultant circuit will be restored to the original inverter.

CycSAT-II assumes that when a correct key-vector is applied to the circuit, the remaining cycle in the resultant circuit is combinational rather than noncombinational. For achieving this, CycSAT-II needs to compute the "no sensitizable cycle" condition, which is different from the no structural cycle condition of CycSAT-I. In addition to the key values, the no sensitizable cycle condition considers the side inputs of each cycle. The reason for considering side inputs additionally is that CycSAT-II does not allow any noncombinational cycle in the circuit after decryption, which means that it will check whether the side inputs of a cycle have input-noncontrolling values simultaneously under an input pattern. If so, it will break the noncombinational cycle by selecting an appropriate key value. For example, Fig. 1(b) shows the cyclic locked circuit, and its original circuit is an AND gate. To break the cycle in Fig. 1(b), CycSAT-II will compute the no sensitizable cycle condition for the cycle in Fig. 1(b). First, CycSAT-II will observe the values of the keys and the side inputs that are able to cause the noncombinational cycle in the circuit. In Fig. 1(b), when $\boldsymbol{K_2} = 1$ and $\boldsymbol{X_3} = 1$, the cycle in Fig. 1(b) will be a noncombinational cycle. Then, CycSAT-II will force $\boldsymbol{K_2} = 0$ to avoid this situation and break the condition about the noncombinational cycles in the circuit.

The advantage of CycSAT is that both its versions consider all aspects of cyclic locking techniques. Specifically, it considers the condition of transforming the cyclic locked circuit to its original acyclic one and considers the condition of having the cycles in the locked circuit and the original one. It also can efficiently find out all the cyclic structures. However, the disadvantage of CycSAT is that it does not consider a situation: noncombinational cycles are still reserved when the correct key-vector is fed. When this situation happens, CycSAT-II will obtain an incorrect key-vector. This is because it assumes that noncombinational cycles are illegal and have to be ruled out.

Thus, cyclic unresolvable using unreachable states [10] utilizes unreachable states to create noncombinational cycles under any correct key-vector, which obfuscates CycSAT for obtaining incorrect key-vectors. Furthermore, SRClock [11] builds super cycles to exponentially prolong the time required in CycSAT for analyzing all the cycles in the circuit.

Although the locking methods [10]–[12], [22]–[24] have been demonstrated to defend SAT Attack, Removal Attack, or CycSAT successfully, these methods either cannot invalidate two of them or can be attacked by their extensions. Thus, in this article, we propose a cyclic logic locking method based on logic optimization that can invalidate all the SAT Attack, Removal Attack, and CycSAT. Our main intention is to create cycles into the circuit and these cycles will not be broken when the correct secret key-vector is applied. These cycles are constructed by adapting a logic optimization technique, node merging (NM) [4], [5]. Since the created cycles will be preserved under the correct secret key-vector, the proposed approach can invalidate SAT Attack and CycSAT.

For defending SAT Attack, we create a cycle pair (Type-I) with the noncombinational cycles, which affects POs when the incorrect key-vector is fed. This noncombinational behavior will let the SAT solver obtain multiple solutions or no solution to invalidate SAT Attack. For defending CycSAT, we create a cycle pair (Type-II) with the noncombinational cycles, which are unobservable at the POs of the circuit when the correct key-vector is fed. Furthermore, Removal Attack cannot unlock the locked circuit by simply removing the locked subcircuit due to the absence of location information of merged nodes. This is because the merged nodes are disappeared in the locked circuit.

The main contributions of this article are twofold.

1) We propose a cyclic logic locking method using a logic optimization technique, which is different from previous locking methods.
2) The proposed method is able to defend all the SAT Attack, CycSAT, and Removal Attack.

The remainder of this article is organized as follows. Section II discusses the similar work with our method. Section III introduces the combinational cycles created by the NM technique. Section IV elaborates on our cyclic logic locking method. Section V evaluates the security under different attacking methods. The experimental results are shown in Section VI. Finally, Section VII concludes this article.

II. Related Work and Comparison

In this section, since this article is related to cyclic logic locking, we will discuss the pros and cons of our method compared with other related cyclic locking techniques, including cyclic logic locking [13], SRClock [11], and cyclic unresolvable using unreachable states [10].

A. Cyclic Logic Locking

Cyclic logic locking is proposed to defend SAT Attack. This technique destroys the DAG structure by randomly generating feedback loops (cycles) in the circuit. Our approach is an extension of this technique. However, our approach sophisticatedly inserts cycle pairs into the circuit such that cycles exist under correct or incorrect key-vectors.

1) Pros: Since the cyclic logic locking technique simply adds a MUX to create the cycles for locking, it becomes invalid when an attacker breaks all the cyclic structures. CycSAT is the method that can attack this cyclic logic locking technique. However, our approach cannot be unlocked by removing all the cyclic structures in the circuit. This is because our locking approach still reserve cycles in the circuit when the correct key-vector is applied.

2) Cons: Our approach exploits logic optimization techniques to create Type-I and Type-II cycle pairs. Unfortunately, only a limited number of cycle pairs are found in some small-size benchmarks. The number of identified cycle pairs to be inserted is various and circuit-dependent for different circuits.

B. SRClock

SRClock is proposed to defend CycSAT. This technique is to build an extremely large amount of cycles called "super cycles" in the circuit. With having these cycles in the circuits, it is a challenge for attackers to search and break all the cycles efficiently.

1) Pros: Although SRClock creates lots of cycles in the circuit to prolong the time required in CycSAT for breaking all the cycles in the circuit, the locked circuit still could be unlocked. If the time for searching the cyclic structures in the circuit could be reduced, the cycles would be broken within one day by CycSAT. On the other hand, the area overhead of this technique is high due to super cycle construction. However, the concept of this article is to invalidate CycSAT fundamentally. Only one cycle pair is enough to resist CycSAT, with little area overhead.

2) Cons: The advantage of SRClock is that it can arbitrarily build super cycles in the circuit. However, our method exploits NM and NM-based cycle generation techniques to create Type-I and Type-II cycle pairs. Unfortunately, an only limited number of cycle pairs are found in some small-size benchmarks. The number of identified cycle pairs is various and circuit-dependent for different circuits.

C. Cyclic Unresolvable Using Unreachable States

The authors proposed a method using unreachable states to create noncombinational cycles under any correct key-vector in the circuit, which obfuscates CycSAT. The main idea of our method is also to reserve noncombinational cycles under the correct key-vector.

1) Pros: Most logic locking techniques add locked structures to the original circuits. If the locked structures are identified and removed, the resultant circuit is the same as the original one. Cyclic unresolvable using unreachable states technique is also no exception. However, our proposed method uses logic optimization techniques to lock the circuit, which is unique to other locking techniques. When the locked structures by our method are identified and removed, the attacker still faces a challenge that how to restore the removed subcircuit in the original circuit.

2) Cons: Our method exploits NM and NM-based cycle generation techniques to create Type-I and Type-II cycle pairs. Unfortunately, only a limited number of cycle pairs are found in some small-size benchmarks. The number of identified cycle pairs is various and circuit-dependent for different circuits. Conversely, cyclic unresolvable using unreachable states technique uses a condition where some inputs are always 0 in the reachable states, creating another situation that these inputs are 1 in the unreachable states and noncombinational cycles are generated.

III. Preliminaries

Since this article is based on NM [4], [5] and NM-based cycle generation [3] techniques, and there are various VLSI testing concepts in these two works, we first introduce some terminologies related to VLSI testing in this section. Then we explain the cycle generation technique [3] based on NM.

A. Background

An *input-controlling value* of a gate g is the value once applied to its inputs, the output value of g can be determined. An *input-noncontrolling value* is the inverse of input-controlling value. A gate h is said to be in the *transitive fanout cone* of a gate g if there exists a path connecting from g to h. On the contrary, a gate g is in the *transitive fanin cone* of a gate h when there is a path leading from g to h.

The *dominators* of a gate g are defined as the gates that paths start from g to any PO would pass through. A *side input* of a dominator is a fanin of a dominator that is not on the path from a gate g to its dominator.

A *stuck-at fault* is a fault model that is used to model manufacturing defects on wires or logic gates. A *stuck-at 1 (0) fault* on a wire or a gate means that the signal on the wire or gate is stuck to a fixed logic value 1 (0) due to manufacturing defects. A *stuck-at fault test* is a process to generate test patterns that are capable of distinguishing a faulty circuit from the fault-free one. The requirements for a stuck-at fault test are to *activate* the fault effect and then to propagate the fault effect to any POs. If there exists no pattern for the fault, the fault is an untestable fault. For an untestable fault, the corresponding wire or gate is redundant and can be replaced by a constant value, either 1 or 0, depending on the faulty value.

The mandatory assignments (MAs) are necessary values assigned to some wires to generate a test pattern for a fault on a wire w. Consider a stuck-at fault on a wire w, the assignments obtained by setting w to the fault-activating value and by setting the side inputs of dominators of w to the fault-propagating values are MAs. Then, these assignments can be propagated

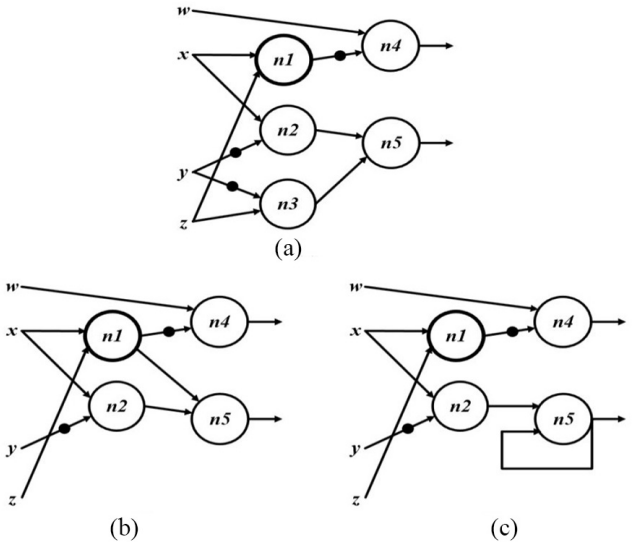

Fig. 2. Example for identifying node mergers to minimize the circuit. (a) Original circuit. (b) Resultant circuit after merging $n1$ and $n3$. (c) Cyclic circuit after merging $n3$ and $n5$.

forward or backward to infer additional MAs by performing logic implications. For example, to activate the fault effect of a stuck-at 0 fault on a wire w, w needs to be assigned a value of 1; to propagate the fault effect, the side inputs of all dominators need to be assigned input-noncontrolling values. If the MAs are inconsistent, no test pattern exists for this fault.

B. Node Merging

NM [4], [5] is a logic optimization technique, which is capable of detecting mergers and thus achieving minimized resultant circuits with considering observability don't cares (ODCs). In [4] and [5], the process of merging two nodes was modeled as a misplaced-wire error. Take Fig. 2(a) as an example, the functionalities of $n1$ and $n3$ only differ when $z = 1$ and $x = y$. Replacing $n3$ with $n1$ causes a misplaced-wire error, which is the functional difference under $z = 1$ and $x = y$. However, $x = y$ implies $n2 = 0$, and $n2 = 0$ blocks the error effect so that the error effect is not observable at $n5$, which means that the error is undetectable. Thus, when this error is undetectable, merging the two nodes will not affect the overall functionality of the circuit. For detecting this error, an input pattern has to cause different values on n_t and n_s for error activation, and propagate the error-effect to any POs. If there does not exist any input pattern that can detect the error, the error is undetectable and the replacement of n_t with n_s is safe in terms of functionality. Condition 1 is the sufficient condition for finding node mergers.

Condition 1 [4], [5]: Let f denote an error of replacing n_t with n_s. If $n_s = 1$ or D, and $n_t = D$ are MAs for the stuck-at 0 fault test on n_t, and $n_s = 0$ or $\overline{D}$, and $n_t = \overline{D}$ are MAs for the stuck-at 1 fault test on n_t, f is undetectable.

The D and $\overline{D}$ symbols in Condition 1 are often used in ATPG algorithms. D means that the value is $1/0$, where 1 is the fault-free value, and 0 is the faulty value. Contrarily, $\overline{D}$ means that the value is $0/1$, where 0 is the fault-free value, and 1 is the faulty value. From the viewpoint of functionality, because $n_t = D$ ($\overline{D}$) actually activates n_t to 1 (0) for the stuck-at 0 (1) fault test, and $n_s = 1$ (0) or D ($\overline{D}$) is also an MA, the replacement of n_t with n_s does not alter the functionality based on Condition 1.

In Condition 1, n_t denotes a target node, and n_s denotes a substitute node. Merging n_t and n_s is equivalent to replacing n_t with n_s, that is, connecting n_s to n_t's fanout node and then removing n_t. Thus, the process of identifying the node mergers is transformed into performing two logic implications by Condition 1: deriving the MAs for the stuck-at 0 and stuck-at 1 fault tests on n_t.

We use the circuit in Fig. 2 to illustrate the merger identification algorithm. For simplicity, the example circuits in the rest of this article are represented in and-inverter graphs (AIGs). Vertices in an AIG represent two-input AND gates; edges represent the connections among the gates; the dots on edges represent inverters. In Fig. 2(a), we compute MAs for the stuck-at 0 and stuck-at 1 fault tests on $n_t = n3$ by activating and propagating the fault effects. The resultant MAs for the stuck-at 0 fault test on $n3$ are $\{n3 = D,\ y = 0,\ z = 1,\ n2 = 1,\ x = 1,\ n1 = 1,\ n4 = 0,\ n5 = D\}$ and that for the stuck-at 1 fault test on $n3$ are $\{n3 = \overline{D},\ n2 = 1,\ x = 1,\ y = 0,\ z = 0,\ n1 = 0,\ n5 = \overline{D}\}$. According to Condition 1, $n1$ and $n5$ satisfy the requirements and can be mergers. However, only $n1$ was used as n_s to replace $n3$ in [4] and [5]. This is because using $n5$, a cyclic structure that [4], [5] did not deal with will be introduced. Fig. 2(b) shows the resultant circuit, which is smaller in terms of the node count, after merging $n3$ with $n1$.

C. NM-Based Cycle Generation

From the previous example in Section III-B, Chen and Wang [4], [5] did not choose $n5$ in Fig. 2(a) as a substitute node n_s to replace the target node $n3$. The reason is that $n5$ lies in the transitive fanout cone of $n3$, and the replacement of $n3$ with $n5$ forms a cycle. Fig. 2(c) shows the circuit after replacing $n3$ with $n5$, where $n2$ is a side input of $n5$. When $n2$ is 1, which is an input-noncontrolling value to $n5$ in Fig. 2(c), the value on the cycle will depend on the previous value on the cycle, which leads to noncombinational behavior. This cycle is called a noncombinational cycle. Hence, Chen *et al.* [3] proposed Theorem 1 to describe the requirement about being combinational cycles.

Theroem 1 [3]: Let n_t denote a target node and n_s denote a substitute node in the transitive fanout cone of n_t. Replacing n_t with n_s forms a set of cycles L. If the value changes on n_t are never propagated to n_s, which means all the side inputs of L do not have input-noncontrolling values simultaneously, L is combinational.

According to Theorem 1, to check if the value changes on n_t are propagated to n_s or not, we can generate an input pattern that propagates the fault effect from n_t to n_s. If no such an input pattern exists, it satisfies Theorem 1 and the formed loop is combinational. The n_s is a cyclic substitute node (CSN).

However, if we would like to find all CSNs, we need to test if there exists a pattern that propagates the fault effect for each pair of n_t and n_s. This process would be computation-intensive. Thus, [3] proposed Condition 2 based on Condition 1 in

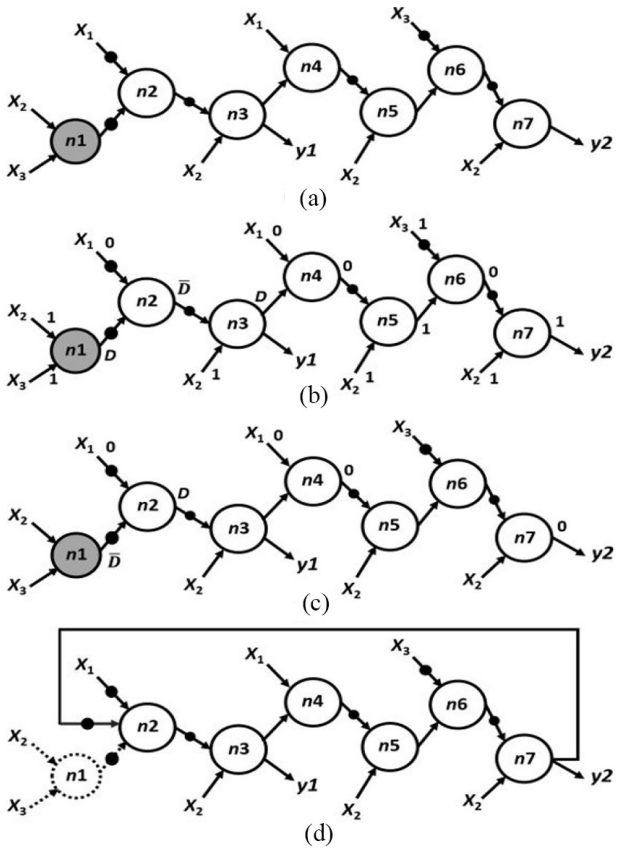

Fig. 3. Example demonstrating the procedure of candidate CSN identification. (a) Circuit before $n1$ and $n7$ are merged. (b) MAs for the stuck-at 0 fault test on $n1$. (c) MAs for the stuck-at 1 fault test on $n1$. (d) Circuit after $n1$ and $n7$ are merged.

Section III-B to efficiently identify candidate CSNs, which are possibly able to form combinational cycles after merging.

Condition 2: Let n_t denote a target node, and n_s denote a substitute node in the transitive fanout cone of n_t. Replacing n_t with n_s forms a set of cycles L. If $n_s = 1$ and $n_t = D$ are MAs for the stuck-at 0 fault test on n_t, and $n_s = 0$ and $n_t = \overline{D}$ are MAs for the stuck-at 1 fault test on n_t, n_s is a candidate CSN.

Different from Condition 1, Condition 2 did not include the situation, "$n_s = D$ and $n_s = \overline{D}$ are MAs for the stuck-at 0 and stuck-at 1 fault tests on n_t, respectively," in it. The reason is that if this situation is included, it will conflict with "the value changes on n_t are never propagated to n_s" in Theorem 1.

We use an example in Fig. 3 to show how to exploit Condition 2 to facilitate candidate CSN identification. In Fig. 3(a), suppose that we choose $n_t = n1$. We first perform the stuck-at 0 fault test on $n1$, the corresponding MAs are $\{X_2 = 1, X_3 = 1, n1 = D, X_1 = 0, n2 = \overline{D}, n3 = D, n4 = 0, n5 = 1, n6 = 0, n7 = 1\}$, as shown in Fig. 3(b). Then, for the stuck-at 1 fault test on $n1$, the corresponding MAs are $\{n1 = \overline{D}, X1 = 0, n2 = D, n4 = 0, n7 = 0^1\}$ as shown in Fig. 3(c). Thus, only $n7$ is identified as a candidate CSN, which is possible to replace $n1$ to form a combinational cycle.

However, Condition 2 only partially satisfies Theorem 1, which means $n7$ is only a candidate CSN rather than a CSN. The reason is that Condition 2 only can ensure the combinationality of the formed cycles under the input patterns $n_t = D$ and $n_t = \overline{D}$. As a result, Chen *et al.* [3] used an SAT-based algorithm to validate whether a candidate CSN is a CSN under other input patterns. In fact, $n7$ is a CSN indeed in this example. Fig. 3(d) shows the cyclic circuit after replacing $n1$ with $n7$. We observe that the side inputs, X_1, $\overline{X_1}$, X_2, and $\overline{X_3}$ of the nodes in the cycle, cannot have input-noncontrolling values simultaneously. Thus, the cycle formed by merging $n1$ and $n7$ is combinational.

[1]One may be curious about why $n7 = 0$ is an MA for the stuck-at 1 fault test on $n1$. The reason is as follows: to activate the fault effect on $n1$, one of X_2 and X_3 has to be 0. If $X_2 = 0$, which implies $n7 = 0$. On the other hand, if $X_2 = 1$ and $X_3 = 0$, accomplished with $n4 = 0$, it will infer $n5 = 1$, $n6 = 1$, and finally $n7 = 0$. Therefore, $n7 = 0$ is also an MA for the stuck-at 1 fault test on $n1$.

IV. Our Method

In this article, we exploit the techniques mentioned in Section III to develop our logic locking method. The locked circuit is cyclic with *combinational* and *noncombinational* cycles. However, when the correct key-vector is fed, the unlocked circuit is still cyclic. In Section III, we have explained how to create a functionally correct combinational cycle. Next, we will discuss how to create noncombinational cycles based on two different conditions either−affecting the POs, or unobservable at the POs, to invalidate SAT Attack and CycSAT. Furthermore, the proposed method will remove target nodes based on NM technique to obfuscate Removal Attack.

According to Theorem 1, the sufficient condition ensuring the formed cycle L is combinational is that the value changes on n_t are never propagated to n_s. That is, there exists a node n_b in the path from n_t to n_s, which blocks the effect of the value changes. Based on the insight, if we would like to create a noncombinational cycle, we can replace n_t with a node in between n_t and n_b. Additionally, if there is any PO located at a node prior to n_b, the value changes on n_t will affect the PO; otherwise, the value changes on n_t is unobservable at the PO.

However, if we create a noncombinaitonal cycle affecting POs, the functionality of the circuit will be changed. Hence, to preserve the functionality of the circuit when the correct key-vector is fed, we propose a scheme that creates a cycle pair (Type-I): one is a noncombinational cycle affecting POs (L_1), the other is a functionally *correct* combinational cycle (L_2). Then we use a MUX (M_1) and a key input (K_1) to configure them, as shown in Fig. 4(a).

In Fig. 4(a), the dotted node T_1 is the target node n_t to be replaced. Using the method in [3]–[5], we can identify that the node S_1 is a substitute node n_s for T_1. Next, let us consider the input pattern $\{X_1 = 0, X_2 = 1, X_3 = 0\}$. Under this pattern, the effect of removing T_1 is blocked at n_3 since X_1 is 0 ($n_b = n3$). Thus, we can choose $n1$ ($n1$ is in between n_t and n_b) to create a noncombinational cycle L_1 that affects POs (the PO $y1$ is located at node n_2 prior to $n_b = n3$). We also choose S_1 to create a functionally *correct* combinational cycle L_2. We use the MUX M_1 controlled by the key input K_1 to connect these two cycles, and K_1 determines that one of $n1$ and S_1 will be selected for substituting T_1. On the other hand, since T_1 is a multiple fanout node, considering the removal of T_1 changing the functionality of other fanout's subcircuit, we add another

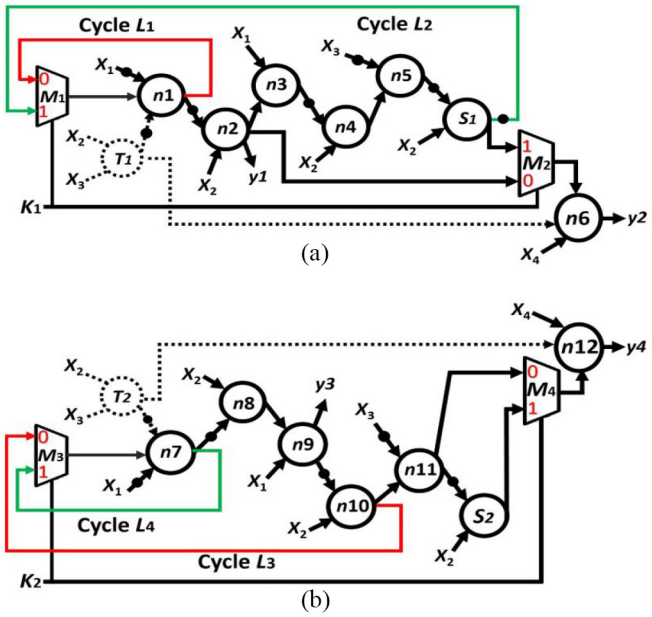

Fig. 4. Example for demonstrating two types of cycles in LOOPlock. (a) Type-I cycle pair. (b) Type-II cycle pair.

MUX M_2, which is also controlled by K_1, for restoring the functionality by selecting the substitute node S_1 with $K_1 = 1$. Hence, if $K_1 = 0$, node $n2$ will be selected by M_2 but cannot restore the functionality. This is because $n2$ is prior to n_b, which satisfies the condition of generating noncombinational cycles, the value change will be propagated to PO $y2$ to alter the functionality of the circuit.

The mission of Type-I cycle pair is to defend SAT Attack. Its effectiveness is explained as follows: Since there is a noncombinational cycle affecting POs in the Type-I cycle pair, at least one PO has indefinite value. When SAT Attack uses SAT solvers to find the DIP, the indefinite value at POs will make the SAT solving calls never terminate. The detailed evaluation of the Type-I cycle pair under SAT Attack will be discussed in Section V.

After creating the Type-I cycle pair to defend SAT Attack, we create another cycle pair (Type-II): one is a noncombinational cycle that is unobservable at POs (L_4), the other is a functionally *incorrect* combinational cycle (L_3), to invalidate CycSAT. We also use a MUX (M_3) and a key input (K_2) to configure them, as shown in Fig. 4(b). We have explained how to create a functionally correct combinational cycle in Section III. If we want to create a functionally incorrect combinational cycle, we just choose a node that has different values with n_t under one input pattern as the substitute node, like $n10$ in Fig. 4(b).

In Fig. 4(b), the dotted node T_2 is the target node n_t to be replaced and node S_2 is a substitute node n_s for T_2. We consider the same input pattern $\{X_1 = 0, X_2 = 1, X_3 = 0\}$. Under this pattern, the effect of removing T_2 is blocked at $n9$ since X_1 is 0 ($n_b = n9$). Thus, we can choose $n7$ ($n7$ is in between n_t and n_b) to create a noncombinational cycle L_4 that is unobservable at POs (no PO is located at nodes prior to $n_b = n9$), and choose $n10$ to create a functionally *incorrect* combinational cycle L_3. We use another MUX M_3 controlled by a key input K_2 to connect these two cycles, and K_2 determines that one of $n7$ and $n10$ will be selected for substituting T_2. Furthermore, we also need to restore the functionality as Type-I cycle pair does. As a result, we use another MUX (M_4), also controlled by K_2, for restoring the functionality by selecting the substitute node S_2 with $K_2 = 1$. However, if $K_2 = 0$, $n11$ will be selected, which cannot restore the functionality. This is because $n11$ has a different value with T_2 under one input pattern, and this effect will be propagated to the PO y_4 such that the functionality of the circuit is changed.

In fact, the Type-II cycle pair is exactly opposite to the Type-I cycle pair, and it is able to resist CycSAT due to the following reason. For CycSAT, it ensures that all the cycles are combinational after decryption, which means that noncombinational cycles are not allowed. However, there exists a noncombinational cycle, which is unobservable at POs, in Type-II cycle pair. Since the behavior of noncombinational cycle is unobservable at POs, it is harmless to the functionality of the circuit. Thus, CycSAT will filter out the noncombinational cycle, and select the functionally incorrect combinational cycle in Type-II cycle pair. As a result, the obtained key is incorrect. The detailed evaluation of the Type-II cycle pair under the attack of CycSAT will also be discussed in Section V.

When creating cycles in the Type-I and Type-II cycle pairs, the target node n_t will be removed permanently. Hence, for Removal Attack, it cannot break the locking by just removing the feedback loops and the added MUXes, $M_1 \sim M_4$. The absence of location information about the original target node n_t invalidates the decryption from Removal Attack.

V. Evaluation

In this section, we evaluate the security of a locked circuit by the proposed logic locking method under the attacks of SAT Attack, CycSAT, and Removal Attack.

A. SAT Attack

SAT Attack transfers the decryption problem into the SAT problem by constructing a miter-like circuit. In each iteration, it identifies a DIP by an SAT solving call, and then uses the DIP and its corresponding output pattern to prune incorrect keys. To demonstrate the effectiveness of our method, we construct a miter-like circuit, which consists of two copies of Type-I cycle pair, as shown in Fig. 5. Then we use SAT solvers to find the DIP. Note that we only construct Type-I cycle pair in this example for simplicity since its mission is for defending SAT Attack.

In the example of Fig. 5, the correct key value is $K_1 = 1$. We will see that SAT Attack cannot obtain this value successfully. In the first iteration of SAT solving call, we may find a DIP ($X_1 = 0, X_2 = 1, X_3 = 1, X_4 = 1$). Then, this DIP and its corresponding outputs of the locked circuit, which are ($y1 = 1$, $y2 = 1$), will be added as a constraint in the SAT solving formula. This constraint will force $K_1 = 1$. Next, in the second iteration, it may find another DIP ($X_1 = 0, X_2 = 1, X_3 = 0$, $X_4 = 0$). Similarly, this DIP and the corresponding outputs of the locked circuit, which are ($y1 = 0, y2 = 0$), will be added as another constraint in the SAT solving formula. However, this

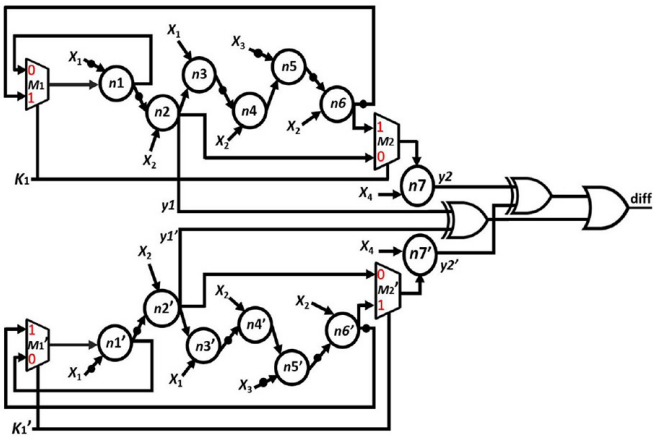

Fig. 5. Miter-like circuit with Type-I cycle pair.

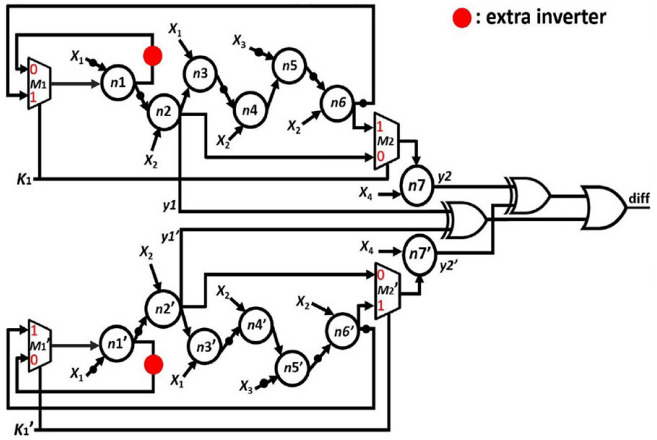

Fig. 6. Miter-like circuit with Type-I cycle pair after adding an extra inverter in each noncombinational cycle in Fig. 5.

constraint will force $K_1 = 0$. Then, in the third iteration, it may find the same DIP ($X_1 = 0, X_2 = 1, X_3 = 0, X_4 = 0$) and force $K_1 = 0$ again. We have known that a DIP in SAT Attack can prune incorrect keys. From the iterations mentioned, we observe that the constraints cannot force $K_1 = 1$ and $K_1 = 0$ simultaneously, which means the incorrect key value ($K_1 = 0$) cannot be pruned. In other words, the SAT Attack will be never terminated since the DIPs cannot prune the incorrect keys. Hence, SAT Attack fails to unlock the circuit locked by our method.

Additionally, we observe a variant of Type-I cycle pair, which is also useful for defending SAT Attack, in this article. That is, we add an extra inverter in the noncombinational cycle of Type-I cycle pair. As shown in Fig. 6, the extra inverters are added at the output of $n1$ and $n1'$ of miter-like circuit of Fig. 5. After adding extra inverters in Fig. 5, unfortunately, SAT Attack cannot find any DIP, which means that SAT solvers return UNSAT in the first solving call. This is because the noncombinational behavior oscillates after adding the inverters in the noncombinational cycles of Fig. 5. That is, SAT solvers cannot find a definite value on each node in these cycles, which leads to UNSAT. Without having the DIP, SAT Attack cannot prune incorrect keys. Thus, we can either use the original Type-I cycle pair or its variant in the locked

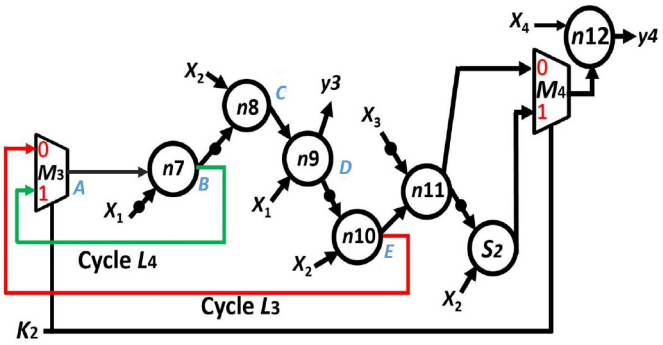

Fig. 7. Example circuit with Type-II cycle pair.

circuit, which further obfuscates attackers. In summary, logic locking by creating Type-I cycle pair invalidates SAT Attack.

B. CycSAT

CycSAT [28] contains CycSAT-I and CycSAT-II versions for different purposes. We will discuss how the proposed method is able to defend both of them. We use the example in Fig. 7, which contains Type-II cycle pair only for simplicity, to demonstrate the ability of our method.

For CycSAT-I, it needs to compute the key values to break cycles. Equation (1) is the result that the CycSAT-I algorithm applied on the example of Fig. 7

$$\begin{aligned} F(A, A') &= (F(A, B) \vee bk(B, A')) \wedge (F(A, E) \vee bk(E, A')) \\ &= -K_2 \wedge K_2 = 0. \end{aligned} \tag{1}$$

In Fig. 7, there exist two cycles L_3 and L_4, where L_4 is inside L_3. To find the key values for breaking cycles L_4 and L_3, the function of key value $F(A, A')$ is derived, where A and A' are the start point and end point of both L_3 and L_4. Since there are two cycles in Fig. 7, $F(A, A')$ is composed of two terms, $F(A, B) \vee bk(B, A')$ represents cycle L_4, and $F(A, E) \vee bk(E, A')$ represents cycle L_3, where $bk(B, A')$ and $bk(E, A')$ are the conditions of key values such that B cannot affect A', and E cannot affect A', respectively. As a result, for breaking L_4, K_2 has to be 0 while K_2 is 1 for breaking L_3. The function of key value for breaking both cycles L_4 and L_3 is $-K_2 \wedge K_2$, which leads to contradiction. Thus, no key value can be obtained to attack the locked circuit of Fig. 7 by applying CycSAT-I.

For CycSAT-II, in addition to the key values, the side input values have to be considered as well for breaking cycles. This is because CycSAT-II assumes that there does not exist any noncombinational cycles in the circuit after decryption. If the side inputs of nodes in a cycle are all input-noncontrolling values simultaneously under an input pattern, CycSAT-II will break the cycle by selecting appropriate key values. In the example of Fig. 7, the function of key values and side inputs $F(A, A')$ for breaking cycles is derived as (2)

$$\begin{aligned} F(A, A') &= (F(A, B) \vee ns(B, A')) \wedge (F(A, E) \vee ns(E, A')) \\ &= (X_1 \vee -K_2) \wedge (X_1 \vee -X_2 \vee -X_1 \vee -X_2 \vee K_2) \\ &= (X_1 \vee -K_2) \wedge 1 \\ &= (X_1 \vee -K_2). \end{aligned} \tag{2}$$

Similarly, the first term, $F(A, B) \vee ns(B, A')$ represents cycle L_4, and $F(A, E) \vee ns(E, A')$ represents cycle L_3, where $ns(B, A')$ and $ns(E, A')$ are the conditions on keys and side input values that B cannot affect A' and E cannot affect A', respectively. As a result, for breaking L_4, X_1 has to be 1 or K_2 has to be 0. For breaking L_3, $X_1 = 1$, or $X_2 = 0$, or $X_1 = 0$, or $X_2 = 0$, or $K_2 = 1$. The resultant key value and side input for breaking both L_4 and L_3 are shown in (2). That is, $X_1 = 1$ or $K_2 = 0$.

The next step is to check that whether there is any input pattern that could sensitize a cycle under the current constraints. We found that $X_1 = 0$ sensitizes cycle L_4. Hence, $X_1 = 0$ enforces $K_2 = 0$ for breaking cycle L_4. $K_2 = 0$ implies that L_3 will be selected as a correct cycle. However, L_4 is a noncombinational cycle, but unobservable at POs. Thus, L_4 is harmless for the overall functionality. Conversely, L_3 is a functionally incorrect combinational cycle. Thus, CycSAT-II will obtain an incorrect key value of $K_2 = 0$ rather than the correct one of $K_2 = 1$. In summary, CycSAT-I and CycSAT-II both cannot crack the proposed locking method.

C. Removal Attack

For Removal Attack, assume that the locked structures in our method have been identified by attackers, as shown in Fig. 8(a). Then, the attackers may directly remove the locked structures of Type-I and Type-II cycle pairs, like MUXs $M_1 \sim M_4$ and related wires, which leads to the remaining circuit as Fig. 8(b). Hence, if the attackers cannot restore the subcircuits, expressed as dotted lines, in the fanins of $n1$, $n6$, and $n7$, the overall functionality of the original circuit will be changed. Fortunately, the locations of target nodes, T_1 and T_2, are invisible to attackers after removing the locked structures. Thus, we can ensure that Removal Attack cannot easily decrypt the proposed method by removing the locked structures.

Furthermore, to convincingly show that Removal Attack is invalid to decrypt our locking technique, we assume that the attacker has identified all the cyclic locking structures and has removed them. Next, the attacker aims to restore the functionality of circuit. If the attacker chooses the wire from one input of a MUX for circuit restoration, the effect is the same as guessing the key value at the selection line of MUX. Hence, this restoration method is with high complexity. Alternatively, the attacker may aim to restore the original circuit by modifying the incomplete circuit resulted from Removal Attack. Since some wires and nodes are disappeared from the original circuit due to NM in the proposed logic locking method, it is quite challenging to attackers to guess the locations for reconnecting wires or adding nodes. If the attacker does try to restore the circuit to its original version, i.e., without cyclic circuit, he may encounter trial-and-error processes and pay much effort in verification. In summary, Removal Attack does not exactly know the locations of the merged (disappeared) nodes such that circuit restoration becomes quite difficult and computation intensive. Thus, the proposed method can defend Removal Attack.

D. Combination of Type-I and Type-II Cycle Pairs Against SAT Attack and CycSAT

In Sections V-A and V-B, we have already demonstrated the capability of Type-I cycle pair against SAT Attack, and the capability of Type-II cycle pair against CycSAT. However, the result of the combination of Type-I and Type-II cycle pairs against SAT Attack and CycSAT has not been mentioned.

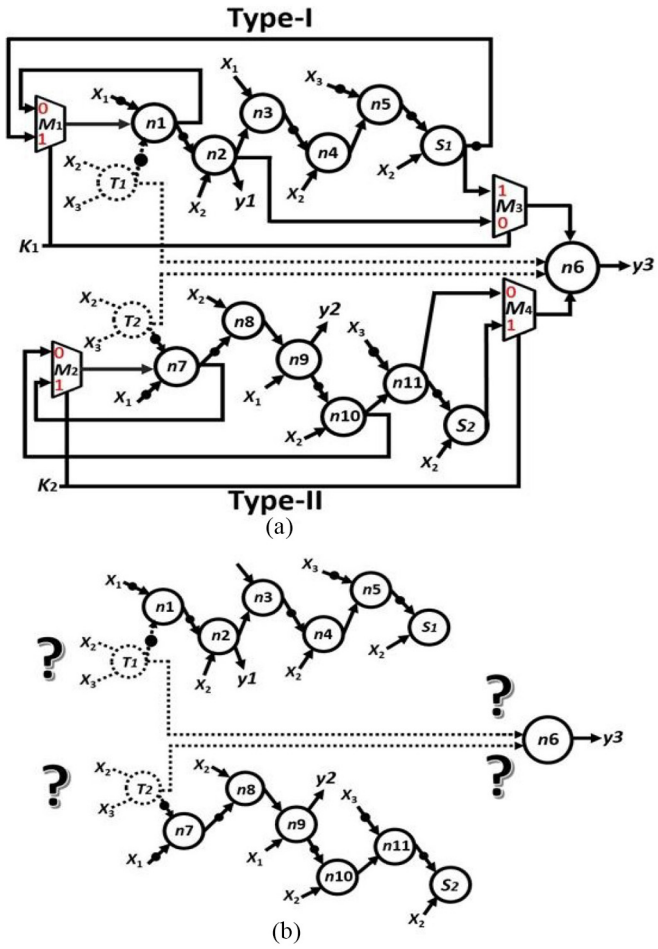

Fig. 8. Example circuit for demonstrating defending Removal Attack. (a) Identified locked circuit with the combination of Type-I and Type-II cycle pairs. (b) Remaining circuit after removing the locked structures.

First of all, an example circuit having the combination of Type-I and Type-II cycle pairs is shown as Fig. 8(a). For SAT Attack, it needs to construct the miter-like circuit from Fig. 8(a) and transforms the circuit into CNF to find the DIP. However, there exists a noncombinational cycle affecting the POs in Type-I cycle pair, which means SAT Attack will never be terminated or cannot find any DIP. As a result, when there exists a Type-I cycle pair in the circuit, it can defend SAT Attack. On the other hand, once there does not exist any Type-I cycle pair in the circuit, it is vulnerable to SAT Attack due to lack of noncombinational cycle affecting the POs.

For CycSAT, it needs to apply CycSAT-I and CycSAT-II to find the keys to break cycles. When CycSAT decrypts a circuit with the combination of Type-I and Type-II cycle pairs, CycSAT-I will fail because the circuit always has cycles under any key value. Then CycSAT-II will rule out all the noncombinational cycles in the circuit. In Type-I cycle pair, the noncombinational cycle affecting the POs is *harmful* to the functionality of circuit, which means CycSAT-II correctly rules out functionally incorrect cycles of Type-I cycle pair. However, in Type-II cycle pair, the noncombinational cycle

unobservable at POs is *harmless* to the functionality of circuit, which means CycSAT-II incorrectly rules out functionally correct cycles of Type-II cycle pair.

After decryption, CycSAT also has to apply SAT Attack to find the remaining key values in the circuit. However, CycSAT-II has ruled out the functionally correct cycles in Type-II cycle pair, which means that the functionality of the resultant circuit has been changed. If SAT Attack directly decrypts the resultant circuit, it is equivalent to finding DIPs in a functionally incorrect circuit. A functionally incorrect circuit means that the values of POs will be incorrect under any key value, and it implies that SAT Attack will always find DIPs in each iteration and be never terminated. Thus, once there exists a Type-II cycle pair in the circuit, it can defend CycSAT. On the other hand, once there does not exist any Type-II cycle pair in the circuit, it is vulnerable to CycSAT due to the lack of noncombinational cycle unobservable at POs.

VI. Experimental Results

We conducted six experiments and show the results in this section. The first experiment is to show the locking capability of the proposed method. The second one is to show the results when considering the tradeoff between security level and area overhead in a benchmark. The third one is to show results of our selection algorithm against the exhaustive attack. The fourth one is to show the area overhead of different benchmarks under a specific number of keys. The fifth one is to show the results of applying SAT Attack and CycSAT to the locked circuits. The last one is to show the results of the output error rate (OER) and averaged hamming distance (Avg. HD) to defend Removal Attack.

We implemented the proposed methods within ABC [2] environment using C language. Our experiments were conducted on a 3.0 GHz Linux platform (CentOS 4.6). The benchmarks are from the IWLS 2005 suite [29]. Every benchmark was transformed to the AIG in blif format, and only its combinational part was considered in the experiments.

We show the algorithm for finding Type-I and Type-II cycle pairs in Fig. 9. Given a circuit $\boldsymbol{C}$, for each target node $\boldsymbol{n_t}$ in $\boldsymbol{C}$, we compute the MAs for the stuck-at 0 fault and stuck-at 1 fault on n_t. Then, according to the MAs, we can find the node n_m satisfying Condition 1 and the node n_{cyclic} satisfying Condition 2. If n_{cyclic} exists, we find the node n_b which blocks the effect of value changed from n_t to n_{cyclic}. Then we check if there exists any PO between n_t and n_b. If so, we find a Type-I cycle pair; otherwise, we find a Type-II cycle pair. However, if there exists n_m but not n_{cyclic}, we check each path P_{nt} in the fanout cone of n_t and find the n_b which blocks the effect of value changed from n_t in P_{nt}. If there exists any PO between n_t and n_b, which means that we cannot create the noncombinational cycle unobservable at POs; otherwise, we can find a Type-II cycle pair. After finding all the Type-I and Type-II cycle pairs, we lock the circuit using the structure in Section IV, and we transfer the circuit into the bench format for running SAT Attack and CycSAT.

Table I shows the results of the first experiment. That is, the number of Type-I and Type-II cycle pairs that the proposed method can identify in each benchmark. Columns 1 and 2 list benchmark name and the number of nodes in it, N. Columns 3 and 4 list the number of identified Type-I, Type-II cycle pairs, respectively. Column 5 shows the corresponding CPU time for identifying all these cycle pairs measured in second. According to Table I, we can see that the proposed method is quite feasible. It can create at least two pairs of cycles for each type among all the benchmarks. Although these results are circuit-dependent, most benchmarks have double digit number of cycle pairs. The average numbers of Type-I and Type-II cycle pair for one benchmark are 40 and 100, respectively. Note that, in theory, only one pair of cycles for each type is enough to defend SAT Attack, CycSAT, and Removal Attack.

Although using one cycle pair of each type in the locking is capable of defending SAT Attack, CycSAT, and Removal Attack, it is fragile under exhaustive attack in practice. Hence, in the second experiment, we select at most 16 cycle pairs of each type to lock each benchmark. Using 16 cycle pairs of each type in the locking means that the length of key-vector is 32 bits, and the total number of key-vectors is 2^{32}, which is an intractable problem to attackers using exhaustive attack. The experimental results are shown in Table II. For one cycle pair of each type, our method will remove the target node n_t as well as single-fanout nodes in the fanin cone of n_t. Meanwhile, our method will add two MUXes for one cycle pair of each type. This node count increase is quite stable for one cycle pair. Nenc column in Table II represents the resultant node count when at most 16 cycle pairs of each type are added for

Algorithm 1 Type-I&Type-II Cycle Pair Identification:

Input: Circuit $\boldsymbol{C}$
Output: A set $\boldsymbol{P_{set}}$ containing Type-I & Type-II cycle pairs
1: **for** each node $\boldsymbol{n_t}$ in $\boldsymbol{C}$
2: Compute MAs for the stuck-at 0 fault test on $\boldsymbol{n_t}$;
3: Compute MAs for the stuck-at 1 fault test on $\boldsymbol{n_t}$;
4: $\boldsymbol{n_{cyclic}} \leftarrow$ Find a node satisfying Condition 2;
5: $\boldsymbol{n_m} \leftarrow$ Find a node satisfying Condition 1;
6: **if** ($\boldsymbol{n_{cyclic}} \neq$ null) **then**
7: $\boldsymbol{n_b} \leftarrow$ Find a node blocking the effect of value
8: changed from $\boldsymbol{n_t}$ to $\boldsymbol{n_{cyclic}}$;
9: **if** (there exists any PO between $\boldsymbol{n_t}$ and $\boldsymbol{n_b}$) **then**
10: Find a Type-I cycle pair $\boldsymbol{P_{typeI}}$;
11: Add $\boldsymbol{P_{typeI}}$ into the set of cycle pairs $\boldsymbol{P_{set}}$;
12: **end**
13: **else**
14: Find a Type-II cycle pair $\boldsymbol{P_{typeII}}$;
15: Add $\boldsymbol{P_{typeII}}$ into the set of cycle pairs $\boldsymbol{P_{set}}$;
16: **end**
17: **else if** ($\boldsymbol{n_{cyclic}}$ == null) && ($\boldsymbol{n_m} \neq$ null) **then**
18: **for** each path $\boldsymbol{P_{nt}}$ in the fanout cone of $\boldsymbol{n_t}$
19: $\boldsymbol{n_b} \leftarrow$ Find a node blocking the effect of value
20: changed from $\boldsymbol{n_t}$ in $\boldsymbol{P_{nt}}$;
21: **if** (there exist no POs between $\boldsymbol{n_t}$ and $\boldsymbol{n_b}$) **then**
22: Find a Type-II cycle pair $\boldsymbol{P_{typeII}}$;
23: Add $\boldsymbol{P_{typeII}}$ into the set of cycle pairs $\boldsymbol{P_{set}}$;
24: **end**
25: **endfor**
26: **endfor**
27: return $\boldsymbol{P_{set}}$;

Fig. 9. Proposed algorithm for finding Type-I and Type-II cycle pairs.

TABLE I
RESULTS OF OUR METHOD IN IDENTIFYING ALL THE TYPE-I AND TYPE-II PAIRS

Benchmark	N	Type-I	Type-II	Time (s)
aes_core	21513	140	228	89.48
b17	52920	74	253	1050.92
b20	12219	36	129	4563.66
b21	12782	30	135	3459.66
b22	18488	45	196	3923.80
C432	209	11	14	0.10
C1908	414	8	22	0.76
C3540	1038	30	20	5.20
C5315	1773	6	6	0.32
C7552	2074	18	27	0.64
dalu	1740	10	34	7.88
des_area	4857	2	3	35.88
i2c	1306	2	7	0.78
i8	3310	64	66	10.81
i10	2673	70	53	39.44
mem_ctrl	15641	53	166	6387.37
pci_bridge32	24369	26	126	296.49
pci_spoci_ctrl	1451	16	29	7.82
rot	1063	9	21	0.53
s9234	1958	14	39	1.27
s13207	2719	11	54	4.08
s38417	9219	35	204	3.94
s38584	12400	27	131	130.32
sasc	784	3	7	0.14
systemcaes	13054	25	138	189.31
tv80	9609	128	213	2126.12
usb_funct	15894	22	129	43.22
wb_conmax	48429	232	351	363.65
Avg.	—	40.96	100.04	812.27

TABLE II
RESULTS OF OUR METHOD IN IDENTIFYING 16 PAIRS OF CYCLES

Benchmark	N	Type-I	Type-II	Nenc	Time (s)	OH (%)
aes_core	21513	16	16	21657	40.07	0.67
b17	52920	16	16	53064	622.77	0.27
b20	12219	16	16	12375	107.71	1.28
b21	12782	16	16	12942	1259.34	1.25
b22	18488	16	16	18646	89.41	0.85
C432	209	11	14	322	0.13	54.07
C1908	414	8	16	523	0.99	26.33
C3540	1038	16	16	1181	1.29	13.78
C5315	1773	6	6	1833	0.38	3.38
C7552	2074	16	16	2228	0.83	7.43
dalu	1740	10	16	1862	7.15	7.01
des_area	4857	2	3	4882	48.40	0.51
i2c	1306	2	7	1347	0.67	3.14
i8	3310	16	16	3444	6.86	4.05
i10	2673	16	16	2813	3.52	5.24
mem_ctrl	15641	16	16	15794	541.57	0.98
pci_bridge32	24369	16	16	24511	87.69	0.58
pci_spoci_ctrl	1451	13	16	1576	5.78	8.61
rot	1063	9	16	1170	0.51	10.07
s9234	1958	14	16	2094	1.15	6.95
s13207	2719	11	16	2844	3.56	4.60
s38417	9219	16	16	9363	2.30	1.56
s38584	12400	16	16	12547	117.46	1.19
sasc	784	3	7	829	0.13	5.74
systemcaes	13054	16	16	13202	57.26	1.13
tv80	9609	16	16	9757	49.30	1.54
usb_funct	15894	16	16	16047	39.60	0.96
wb_conmax	48429	16	16	48573	31.03	0.30
Avg.	—	—	—	—	111.67	6.20

locking. Column 6 is the required CPU time. The last column is the corresponding percentage of area overhead.

According to Table II, we can see that the average area overhead is 6.20% for all the benchmarks. If we exclude the benchmarks with less than 1000 nodes, e.g., *C*432, *C*1908, and *sasc*, the percentage of area overhead will be even lower. On the other hand, in our method, the security level can be elevated by adding more cycle pairs if available, with a little more area overhead. Note that the required CPU time for benchmarks with similar size is various a lot possibly. For example, *b*20 and *b*21 have similar size, but the required CPU time for adding the same amount of Type-I and Type-II cycle pairs is significantly different. The main reason about this difference is the functionality of benchmark. The proposed method is based on NM technique, which exploits logic implications to find the CSNs. Hence, each benchmark has its complexity for finding cycle pairs, regardless of the circuit size.

It is worthy mentioning that the proposed method has the following shortcoming. Since the effectiveness of the proposed method is circuit-dependent, it is probably that only few cycle pairs can be constructed in a circuit, like the benchmarks *des_area*, *i*2*c*, and *sasc*. Nevertheless, good news is that our method can identify many (>16) Type-I and Type-II cycle pairs for benchmarks with larger node count (>10 000) due to the existence of many ODCs in these benchmarks. Thus, for large VLSI designs, our method is still promising and applicable.

Furthermore, to demonstrate the defensiveness of the proposed locking scheme against the exhaustive attack, in addition to the experiment in Table II, we have also conducted another experiment, which is a defense evaluation about assuming that an attacker divides a circuit into logic cones and then guesses the partial key-vector exhaustively. If we did not consider any selection strategy in the locking procedure, the locations of inserted key gates may not be in the fanin cone of the same PO, which lowers the security level of locked circuits. Thus, we propose a selection algorithm for the cycle pair insertion such that the cycle pairs will be clustered in the fanin cone of one PO. The new results of our approach with the proposed selection algorithm are shown in Table III. In Table III, the number of inserted Type-I and Type-II cycle pairs are both 16. The last column shows the maximal number of cycle pair in the fanin cone of one PO. According to Table III, we can see that clustering 32 cycle pairs into the fanin cone of one PO is possible for almost all benchmarks. The pseudo code of the selection algorithm for cycle pair insertion is shown in Fig. 10. In Fig. 10, given a circuit $\boldsymbol{C}$, for each cycle pair $\boldsymbol{C_p}$, we calculate the number of POs in the fanout cone of $\boldsymbol{C_p}$. Then, we can obtain the number of cycle pairs in the fanin cone of each PO. We select the PO that has the most cycle pairs, **max_*cp***, in its fanin cone. If max_*cp* is larger than or equal to 32, we arbitrarily select 32 cycle pairs

TABLE III
RESULTS OF OUR SELECTION ALGORITHM AGAINST THE EXHAUSTIVE ATTACK

Benchmark	Type-I	Type-II	|max. cycle pairs in one PO|
aes_core	16	16	32
b17	16	16	32
b20	16	16	32
b21	16	16	32
b22	16	16	30
C3540	16	16	32
C7552	16	16	32
i8	16	16	32
i10	16	16	32
mem_ctrl	16	16	32
pci_bridge32	16	16	32
s38417	16	16	32
s38584	16	16	32
systemcaes	16	16	32
tv80	16	16	32
usb_funct	16	16	32
wb_conmax	16	16	32

Algorithm 2 Cycle Pair Selection:

Input: Circuit $\boldsymbol{C}$
Output: A set of cycle pairs $\boldsymbol{P_{max}}$ that are clustered within the fanin cone of fewest number of POs
1: **for** each cycle pair $\boldsymbol{C_p}$ in $\boldsymbol{C}$
2: Label POs that can be reached from $\boldsymbol{C_p}$;
3: **endfor**
4: Count the number of cycle pairs in the fanin cone of each PO;
5: Sort the POs by the number of cycle pairs in a descending order;
6: $\boldsymbol{PO_{max}} \leftarrow$ the PO having the most cycle pairs, $\boldsymbol{max\_cp}$, in its
7: fanin cone;
8: Select $\boldsymbol{PO_{max}}$;
9: **if** ($\boldsymbol{max\_cp} \geq 32$) **then**
10: Arbitrarily select 32 cycle pairs from $\boldsymbol{PO_{max}}$'s fanin cone as
11: the set $\boldsymbol{P_{max}}$;
12: **end**
13: **else**
14: Include cycle pairs in the fanin cone of $\boldsymbol{PO_{max}}$ into $\boldsymbol{P_{max}}$;
15: **while** (($|\boldsymbol{P_{max}}| < 32$) and (the next PO $\neq$ null))
16: Select the next PO having the most cycle pairs in its fanin
17: cone;
18: **if** (($|\boldsymbol{P_{max}}|$ + number of cycle pairs in the fanin cone of
19: the next PO) <32) **then**
20: Include cycle pairs in the fanin cone of the next PO into
21: the set $\boldsymbol{P_{max}}$;
22: **end**
23: **else**
24: Arbitrarily select the number of cycle pairs that can
25: reach 32 into the set $\boldsymbol{P_{max}}$;
26: **end**
27: **endwhile**
28: **end**
29: return $\boldsymbol{P_{max}}$;

Fig. 10. Proposed new selection algorithm for inserting cycle pairs.

from its fanin cone as the final cycle pairs $P_{\max}$. If max_*cp* is smaller than 32, we include these cycle pairs into the set $P_{\max}$. Next, we repeat this operation until the size of the set $P_{\max}$ reaches 32. If the size of $P_{\max}$ is less than 32 when selecting all the number of cycle pairs, we select all of them as $P_{\max}$.

Next, we demonstrate the fourth experiment about the area overhead of different benchmarks under specific numbers of keys. Here, we choose some benchmarks from Table II that can be locked by at least 16 cycle pairs of each type, such as *C*7552, *s*38417, and *b*17, to show the tradeoff between the total number of key-vectors and area overhead in Fig. 11. We observed that the area overhead positively related to the total number of key-vectors. Among the benchmarks, there are three groups with similar node count, one is the small-size group with *C*7552, another is the medium-size group with *s*38417, the other is the large-size group with *b*17.

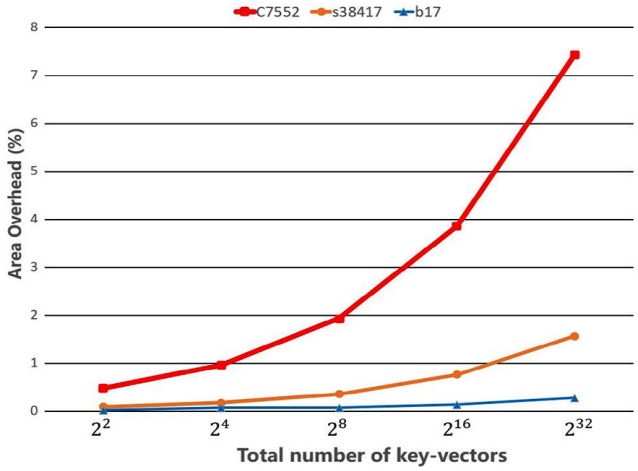

Fig. 11. Experiments for demonstrating the area overhead of different benchmarks under specific numbers of key-vectors.

For all the groups, the area overhead is similar when the total numbers of key-vectors are 2^2 and 2^4, which are all less than 1%. However, when the number of key-vectors increases to 2^{32}, the area overhead of small-size group grows to near 8%, while for large-size group, the area overhead is less that 1%. This result shows that our method can achieve a high security level with a little area overhead for large-size circuits.

In addition to the experiment of area overhead, we also conducted another experiment about timing and power overhead. Table IV shows the experimental results of timing and power overhead under at most 16 cycle pairs of each type. Columns 4 and 5 show the level of critical path in the original circuits and locked circuits, respectively. For inserting each cycle pair, we add at least two MUXes in our locking structure. If the cycle pair is inserted on the critical path, it will affect the delay of the locked circuit. According to Table IV, we found that for some benchmarks, like ***C7552*** and ***s38584***, the levels of critical path do not increase a lot. However, for some benchmarks like ***b20*** and ***b22***, the levels of critical path increase a lot. We realized that the reason for this difference is the circuit structure. As we know, to have Type-I and Type-II cycle pairs, the circuit structure has to satisfy some requirements like the conditions in NM [4], [5] and NM-based cycle generation [3]. If the critical path satisfies the requirements for having cycle pairs, we may still insert the cycle pairs on it. Thus, the level of critical path could be increased. Columns 6 and 7 show the dynamic power of the original circuits and locked circuits, respectively.

According to Table IV, most benchmarks only have little dynamic power overhead due to the minor increase of area after cycle pair insertion. However, for some benchmarks like

TABLE IV
RESULTS OF OUT METHOD ABOUT TIMING AND POWER OVERHEAD

Benchmark	Type-I	Type-II	Init_level	Lock_level	Init_power (uW)	Lock_power (uW)
aes_core	16	16	26 (1)	42 (1.62)	11390 (1)	11562 (1.02)
b17	16	16	43 (1)	46 (1.07)	6787 (1)	7135 (1.05)
b20	16	16	66 (1)	104 (1.58)	7738 (1)	7988 (1.03)
b21	16	16	67 (1)	76 (1.13)	7838 (1)	8213 (1.05)
b22	16	16	69 (1)	100 (1.45)	11391 (1)	11612 (1.02)
C432	11	14	42 (1)	43 (1.02)	68 (1)	143 (2.10)
C1908	8	16	32 (1)	46 (1.43)	307 (1)	626 (2.03)
C3540	16	16	41 (1)	72 (1.76)	505 (1)	928 (1.84)
C5315	6	6	38 (1)	38 (1)	834 (1)	1073 (1.29)
C7552	16	16	29 (1)	29 (1)	1270 (1)	2190 (1.72)
dalu	10	16	39 (1)	40 (1.03)	139 (1)	760 (5.47)
des_area	2	3	33 (1)	33 (1)	2855 (1)	5089 (1.78)
i2c	2	7	16 (1)	16 (1)	174 (1)	207 (1.19)
i8	16	16	27 (1)	45 (1.67)	262 (1)	612 (2.34)
i10	16	16	51 (1)	64 (1.25)	706 (1)	1064 (1.51)
mem_ctrl	16	16	36 (1)	46 (1.28)	1846 (1)	2410 (1.31)
pci_bridge32	16	16	31 (1)	34 (1.10)	5202 (1)	5499 (1.06)
pci_spoci_ctrl	13	16	19 (1)	49 (2.58)	113 (1)	398 (3.52)
rot	9	16	51 (1)	51 (1)	199 (1)	283 (1.42)
s9234	14	16	36 (1)	49 (1.36)	473 (1)	579 (1.22)
s13207	11	16	34 (1)	55 (1.61)	629 (1)	706 (1.12)
s38417	16	16	30 (1)	43 (1.43)	3738 (1)	3828 (1.02)
s38584	16	16	36 (1)	38 (1.05)	3630 (1)	3803 (1.05)
sasc	3	7	9 (1)	17 (1.89)	218 (1)	325 (1.49)
systemcaes	16	16	47 (1)	70 (1.49)	5124 (1)	5443 (1.06)
tv80	16	16	52 (1)	67 (1.29)	2228 (1)	2331 (1.05)
usb_funct	16	16	27 (1)	27 (1)	5104 (1)	5480 (1.07)
wb_conmax	16	16	27 (1)	30 (1.11)	13072 (1)	13351 (1.02)

TABLE V
RESULTS OF OUR METHOD FOR DEFENDING SAT ATTACK AND CYCSAT

Benchmark	Type-I	Type-II	SAT Attack	Time (s)	CycSAT	Time (s)
aes_core	1	1	UNSAT	0.0149	Inf. loop	Inf. loop
b17	1	1	UNSAT	0.0469	Inf. loop	Inf. loop
b20	1	1	UNSAT	0.0139	Inf. loop	Inf. loop
b21	1	1	UNSAT	0.0079	Inf. loop	Inf. loop
b22	1	1	UNSAT	0.0149	Inf. loop	Inf. loop
C432	1	1	Inf. loop	Inf. loop	Inf. loop	Inf. loop
C1908	1	1	Inf. loop	Inf. loop	Inf. loop	Inf. loop
C3540	1	1	UNSAT	0.0019	Inf. loop	Inf. loop
C5315	1	1	UNSAT	0.0039	Inf. loop	Inf. loop
C7552	1	1	UNSAT	0.0029	Inf. loop	Inf. loop
dalu	1	1	Inf. loop	Inf. loop	Inf. loop	Inf. loop
des_area	1	1	Inf. loop	Inf. loop	Inf. loop	Inf. loop
i2c	1	1	UNSAT	0.0009	Inf. loop	Inf. loop
i8	1	1	Inf. loop	Inf. loop	Inf. loop	Inf. loop
i10	1	1	UNSAT	0.0019	Inf. loop	Inf. loop
mem_ctrl	1	1	UNSAT	0.0169	Inf. loop	Inf. loop
pci_bridge32	1	1	UNSAT	0.0149	Inf. loop	Inf. loop
pci_spoci_ctrl	1	1	Inf. loop	Inf. loop	Inf. loop	Inf. loop
rot	1	1	UNSAT	0.0001	Inf. loop	Inf. loop
s9234	1	1	UNSAT	0.0049	Inf. loop	Inf. loop
s13207	1	1	UNSAT	0.0029	Inf. loop	Inf. loop
s38417	1	1	UNSAT	0.0069	Inf. loop	Inf. loop
s38584	1	1	UNSAT	0.0089	Inf. loop	Inf. loop
sasc	1	1	UNSAT	0.0039	Inf. loop	Inf. loop
systemcaes	1	1	UNSAT	0.0069	Inf. loop	Inf. loop
tv80	1	1	UNSAT	0.0059	Inf. loop	Inf. loop
usb_funct	1	1	UNSAT	0.0119	Inf. loop	Inf. loop
wb_conmax	1	1	UNSAT	0.0329	Inf. loop	Inf. loop

dalu and ***des_area***, their power overhead is large. The reason for this large power overload in the locked circuits of ***dalu*** and ***des_area*** is oscillating behavior. Since our locking scheme allows having noncombinational cycles under the correct key-vectors, the signals in these noncombinational cycles may be continuously switched such that the power overhead increases. However, for large circuits, which have many Type-I and Type-II cycle pairs to be inserted, we can choose the cycle pairs whose noncombinational cycles do not have oscillating behavior. Thus, the proposed locking scheme is still promising and applicable from the viewpoint of dynamic power overhead.

Furthermore, we also show the results of applying SAT Attack and CycSAT to the locked circuits in Table V. In this experiment, we only generated one Type-I cycle pair and one Type-II cycle pair for each benchmark as shown in Columns 2 and 3 since one Type-I cycle pair and one Type-II cycle pair are enough to defend SAT Attack and CycSAT. Columns 4 and 5 show the results and corresponding CPU time after applying SAT Attack. The results are either UNSAT or Inf. loop (infinite loop), which is consistent with the security evaluation mentioned in Section V. This is because the noncombinational cycle in Type-I cycle pair of our method makes SAT solvers never be terminated or UNSAT. The required CPU time is also very little. Columns 6 and 7 show the results and corresponding CPU time after applying CycSAT. The results are all Inf. loop, this is because the algorithm of CycSAT cannot effectively find the condition to break the cycle, which causes the follow-up SAT Attack fail to find DIPs due to the noncombinational cycle.

Finally, in the last experiment, to show high complexity of restoring the functionality after Removal Attack, we conducted the experiment that compares the hamming distance (HD) and OER between the original circuit and the recovered one. The experimental setting is as follows: we chose a key-vector and applied it to the locked circuit. Then we observe the differences at the outputs of the original circuit and recovered one under 1000 random patterns. If any PO has a different value under a pattern, we count one. Then, we can obtain the OER of the recovered circuit. We also calculate the averaged HD of the outputs of two circuits for all the simulated patterns. The experimental results are shown in Table VI. In Table VI, Column 2 shows the benchmarks and the number of POs. Columns 3 and 4 show the numbers of inserted Type-I and Type-II cycle pairs. Columns 5 and 6 show the results of OER and the averaged HD between the original circuit and recovered one. Column 7 shows the ratio of the averaged HD and the number of POs in each benchmark. According to Table VI, we observed that most circuits have 100% OER, which means that Removal Attack cannot easily restore the functionality of the original circuit. However, for some benchmarks like ***C1908***, ***C5315***, and ***dalu***, they have lower OER. The reasons are that the number of cycle pairs is not 32, or the number of outputs is few. These factors affect OER significantly. For the averaged HD, the result shows that this value varies a lot, and its reasons are the same as for the OER. The ratio of averaged HD and the number of POs is up to 30% for more than half benchmarks, which means that many outputs are still incorrect when Removal Attack is applied. Thus, our method is still promising and applicable.

TABLE VI
RESULTS OF OER AND AVG. HD TO DEFEND REMOVAL ATTACK

Benchmark	|PO|	Type-I	Type-II	OER (%)	Avg.HD	$\frac{HD}{\|PO\|}$ Ratio (%)
aes_core	659	16	16	100	323	49
b17	1512	16	16	100	366	24
b20	512	16	16	100	254	50
b21	512	16	16	100	254	50
b22	757	16	16	100	508	67
C432	7	11	14	67	2	29
C1908	25	8	16	49	3	12
C3540	22	16	16	84	5	23
C5315	123	6	6	38	2	2
C7552	107	16	16	100	12	11
dalu	16	10	16	16	2	13
des_area	192	2	3	100	123	64
i2c	142	2	7	100	18	13
i8	81	16	16	99	27	33
i10	224	16	16	100	8	4
mem_ctrl	1235	16	16	100	214	17
pci_bridge32	3566	16	16	100	1590	45
pci_spoci_ctrl	73	13	16	100	22	30
rot	107	9	16	100	5	5
s9234	250	14	16	100	109	44
s13207	790	11	16	100	64	8
s38417	1742	16	16	100	64	4
s38584	1730	16	16	100	65	4
sasc	129	3	7	100	56	43
systemcaes	799	16	16	100	238	30
tv80	391	16	16	100	146	37
usb_funct	1867	16	16	100	655	35
wb_commax	2186	16	16	100	882	40

VII. CONCLUSION

In this article, we proposed a cyclic logic locking method to invalidate the state-of-the-art attacking methods. The experimental results show that the proposed method is able to lock the general combinational benchmarks with low area overhead. It is promising to push forward the progress of logic locking techniques with the proposed method.

ACKNOWLEDGMENT

The authors would like to thank the NuLogiCS research group led by Prof. H. Zhou for providing the program of CycSAT. They also thank Y. Li for his quick responses to their questions about the program.

REFERENCES

[1] K. Z. Azar, H. M. Kamali, H. Homayoun, and A. Sasan, "SMT attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the SAT Attacks," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2019, no. 1, pp. 97–122, 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/7335

[2] Berkeley Logic Synthesis and Verificaiton Group. Accessed: Jul. 12, 2019. *ABC: A System for Sequential Synthesis and Verification*. [Online]. Available: http://www.eecs.berkeley.edu/alanmi/abc

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

[4] Y.-C. Chen and C.-Y. Wang, "Fast node merging with don't cares using logic implications," *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design Dig. Tech. Papers (ICCAD)*, 2009, pp. 785–788.

[5] Y.-C. Chen and C.-Y. Wang, "Fast node merging with don't cares using logic implications," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 29, no. 11, pp. 1827–1832, Nov. 2010.

[6] A. Chakraborty, Y. Liu, and A. Srivastava, "TimingSAT: Timing profile embedded SAT attack," *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design Dig. Tech. Papers (ICCAD)*, 2018, pp. 1–6.

[7] M. Chen, E. Moghaddam, N. Mukherjee, J. Rajski, J. Tyszer, and J. Zawada, "Hardware protection via logic locking test points," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 12, pp. 3020–3030, Dec. 2018.

[8] J. A. Roy, F. Koushanfar, and I. L. Markov, "Ending piracy of integrated circuits," *Computer*, vol. 43, no. 10, pp. 30–38, 2010.

[9] A. Rezaei, Y. Shen, S. Kong, J. Gu, and H. Zhou, "Cyclic locking and memristor-based obfuscation against CycSAT and inside foundry attacks," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, 2018, pp. 85–90.

[10] A. Rezaei, Y. Li, Y. Shen, S. Kong, and H. Zhou, "CycSAT-unresolvable cyclic logic encryption using unreachable states," in *Proc. Asia South Pacific Design Autom. Conf. (ASPDAC)*, 2019, pp. 358–363.

[11] S. Roshanisefat, H. M. Kamali, and A. Sasan, "SRClock: SAT-resistant cyclic logic locking for protecting the hardware," in *Proc. Great Lakes Symp. VLSI (GLSVLSI)*, 2018, pp. 153–158.

[12] A. Sengupta and M. Rathor, "Security of functionally obfuscated DSP core against removal attack using SHA-512 based key encryption hardware," *IEEE Access*, vol. 7, pp. 4598–4610, 2018.

[13] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, "Cyclic obfuscation for creating SAT-unresolvable circuits," in *Proc. Great Lakes Symp. VLSI (GLSVLSI)*, 2017, pp. 173–178.

[14] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, "AppSAT: Approximately deobfuscating integrated circuits," in *Proc. IEEE Int. Symp. Hardw. Oriented Security Trust (HOST)*, 2017, pp. 95–100.

[15] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the security of logic encryption algorithms," in *Proc. IEEE Int. Symp. Hardw. Oriented Security Trust (HOST)*, 2015, pp. 137–143.

[16] Y. Shen, A. Rezaei, and H. Zhou, "A comparative investigation of approximate attacks on logic encryptions," in *Proc. Asia South Pacific Design Autom. Conf. (ASP-DAC)*, 2018, pp. 271–276.

[17] Y. Shen, A. Rezaei, and H. Zhou, "SAT-based bit-flipping attack on logic encryptions," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, 2018, pp. 629–632.

[18] Y. Shen and H. Zhou, "Double DIP: Re-evaluating security of logic encryption algorithms," in *Proc. Great Lakes Symp. VLSI (GLSVLSI)*, 2018, pp. 179–184.

[19] Y. Shen, Y. Li, S. Kong, A. Rezaei, and H. Zhou, "SigAttack: New high-level SAT-based attack on logic encryptions," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, 2019, pp. 940–943.

[20] Y. Shen, Y. Li, A. Rezaei, S. Kong, D. Dlott, and H. Zhou, "BeSAT: Behavioral SAT-based attack on cyclic logic encryption," in *Proc. Asia South Pacific Design Autom. Conf. (ASP-DAC)*, 2019, pp. 657–662.

[21] Y. Xie and A. Srivastava, "Delay locking: Security enhancement of logic locking against IC counterfeiting and overproduction," in *Proc. ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, 2017, pp. 1–6.

[22] Y. Xie and A. Srivastava, "Mitigating SAT Attack on logic locking," in *Proc. Int. Conf. Cryptograph. Hardw. Embedded Syst.*, 2016, pp. 127–146.

[23] M. Yasin, B. Mazumdar, J. Rajendran, and O. Sinanoglu, "SARlock: SAT attack resistant logic locking," in *Proc. IEEE Int. Symp. Hardw. Orient. Security Trust (HOST)*, 2016, pp. 236–241.

[24] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. Rajendran, and O. Sinanoglu, "Provably-secure logic locking: From theory to practice," in *Proc. ACM Conf. Comput. Commun. Security*, 2017, pp. 1601–1618.

[25] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran, "Removal attacks on logic locking and camouflaging techniques," *IEEE Trans. Emerg. Topics Comput.*, to be published, doi: 10.1109/TETC.2017.2740364.

[26] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran, "Security analysis of anti-SAT," in *Proc. Asia South Pacific Design Autom. Conf. (ASP-DAC)*, 2016, pp. 342–347.

[27] G. L. Zhang, B. Li, B. Yu, D. Z. Pan, and U. Schlichtmann, "TimingCamouflage: Improving circuit security against counterfeiting by unconventional timing," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, 2018, pp. 91–96.

[28] H. Zhou, R. Jiang, and S. Kong, "CycSAT: SAT-based attack on cyclic logic encryptions," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design Dig. Tech. Papers (ICCAD)*, 2017, pp. 49–56.

[29] IWLS2005 Benchmarks. Accessed: Jul. 12, 2019. [Online]. Available: http://iwls.org/iwls2005/benchmarks.html

Hsiao-Yu Chiang received the B.S. degree from the Department of Computer Science, National Taipei University of Education, Taipei, Taiwan, in 2017. He is currently pursuing the M.S. degree with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan.

His current research interests include hardware security and electrical design automation.

Yung-Chih Chen received the B.S., M.S., and Ph.D. degrees from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2003, 2005, and 2011, respectively.

He is currently an Assistant Professor with the Department of Computer Science and Engineering, Yuan Ze University, Taoyuan, Taiwan. His current research interests include logic synthesis, design verification, and design automation for emerging technologies.

De-Xuan Ji received the B.S. and M.S. degrees from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2016 and 2018, respectively.

His current research interests include hardware security and electrical design automation.

Xiang-Min Yang received the B.S. degree from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2018. He is currently pursuing the M.S. degree with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan.

His current research interests include hardware security and electrical design automation.

Chia-Chun Lin received the B.S. and M.S. degrees from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2011 and 2013, respectively, where he is currently pursuing the Ph.D. degree.

His current research interests include logic synthesis, optimization, verification for VLSI designs, and automation for emerging technologies.

Chun-Yao Wang (Member, IEEE) received the B.S. degree from the Department of Electronics Engineering, National Taipei University of Technology, Taipei, Taiwan, in 1994, and the Ph.D. degree from the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, in 2002.

Since 2003, he has been an Assistant Professor with the Department of Computer Science, National Tsing Hua University, Hsinchu, where he is currently a Distinguished Professor. He has published over 70 technical papers in these areas and is a named inventor in 9 patents. His current research interests include logic synthesis, optimization, and verification for very large-scale integrated/system-on-chip designs and emerging technologies.

Dr. Wang was the recipient of the Best Paper Award in 2018 IEEE International Symposium on VLSI Design, Automation and Test. Two of his research results were nominated as Best Papers in the 2009 IEEE Asia and South Pacific Design Automation Conference and the 2010 IEEE/ACM Design Automation Conference, respectively.