Hide & Seek: Seeking the (Un)-Hidden key in Provably-Secure Logic Locking Techniques

Satwik Patnaik, *Member, IEEE*, Nimisha Limaye, *Graduate Student Member, IEEE,* and Ozgur Sinanoglu, *Senior Member, IEEE*

Abstract—Logic locking is a holistic countermeasure that protects an integrated circuit (IC) from hardware-focused threats such as piracy of design intellectual property and unauthorized overproduction throughout the globalized IC supply chain. Out of the several techniques proposed by the hardware security community, provably-secure logic locking (PSLL) has acquired a foothold due to its algorithmic and provable-security guarantees. However, the security of these techniques are regularly questioned by attackers that exploit the vulnerabilities arising from the underlying hardware implementation. Unfortunately, such attacks (i) are predominantly specific to locking techniques and (ii) lack generality and scalability. This leads to a plethora of attacks and researchers, especially defenders, find it challenging to ascertain the security of newly developed PSLL techniques. Additionally, there is no public repository of locked circuits that attackers can use to benchmark (and compare) their developed attacks.

Driven by these challenges, we aim to develop a generalized attack that can recover the secret key across a breadth of PSLL techniques. To that end, we first categorize the existing PSLL techniques into two generic categories. Then, we extract functional and structural properties depending on the underlying hardware construction of the PSLL techniques and develop two attacks based on the concepts of VLSI testing and Boolean transformations. We evaluate our attacks on 30,000 locked circuits across 14 PSLL techniques, including nine unbroken techniques. Our attacks successfully recover the secret key (100% accuracy) for all the considered techniques. Further, our experimentation across different (i) technology libraries, (ii) commercial and academic synthesis tools, and (iii) logic optimization settings provide several interesting insights. For instance, our attacks can recover the secret key by only using the locked circuit when an academic synthesis tool is used. Additionally, designers can use our attacks as a verification tool to ascertain the lower-bound security achieved by hardware implementations. Finally, we shall release our artifacts, which could help foster the development of future attacks and defenses in PSLL domain.

Index Terms—Hardware security, IP protection, key recovery attack, provably secure logic locking

I. INTRODUCTION

THE continual miniaturization of integrated circuit (IC) technology nodes have exacerbated the costs of commissioning state-of-the-art foundries [1]. Designs are regularly outsourced to potentially untrustworthy foundries, and as a result, several hardware-focused threats have emerged, ranging from piracy of design intellectual property (IP) and unauthorized overproduction of ICs to insertion of malicious logic [2].

Logic locking is a holistic countermeasure that protects an IC from several hardware-focused threats such as reverse-engineering, piracy of design IP, and unauthorized overproduction throughout the IC supply chain [3]. Logic locking transforms the original circuit by incorporating additional logic (key-gates) controlled by a secret key. As a result of inserting key-gates, a locked circuit includes additional inputs, referred to as key-inputs apart from regular primary inputs. The secret key is stored in a tamper-proof memory and securely programmed by a trusted facility (e.g., a design house) *after* the fabrication and testing of the ICs. The application of the correct key ensures the locked circuit functions correctly (for all input patterns), while an incorrect key renders the locked circuit to produce corrupted outputs. *The security guarantees offered by logic locking techniques are contingent on the inability of an attacker to recover the secret key.* Prior combinational logic locking techniques focused on (i) finding suitable locations for key-gate insertion [4], [5], and (ii) exploring different key-gates (e.g., multiplexers [6]).

Input/Output-based attacks: The Boolean Satisfiability-based attack (commonly known as SAT-based attack in the logic locking community) [7] broke all known logic locking techniques in 2015. The attack uses a SAT solver to generate *distinguishing input patterns (DIPs)*—these input patterns enable the elimination of incorrect keys from the key search space. The DIPs, along with output responses from a working chip (a.k.a. *oracle*), iteratively eliminate incorrect keys, resulting in the recovery of the secret key. Subsequently, researchers developed approximate-based attacks (*AppSAT* [8] and *Double DIP* [9]) that relax the exactness constraint in the SAT-based attack to yield an approximate key. All the aforementioned attacks utilize input/output (I/O) pairs from an oracle and thus are called *I/O-based attacks*.

I/O-based attack resilient locking: The logic locking community proposed several techniques to thwart I/O-based attacks. These can be categorized under (i) point function-based locking,[1] (ii) SAT-hard locking, (iii) cyclic locking, and (iv) scan locking. Concerning (i), researchers proposed aug-

[1]Also known as provably-secure logic locking, more details in §II-C.

menting the original circuit with logic structures (e.g., point-functions) that ensure I/O-based attacks can prune out exactly one incorrect key in every attack iteration [10], [11]. Adopting this construction necessitates I/O-based attacks to query an exponential number of input patterns (regarding key-size) to recover the secret key. The techniques under (ii), *i.e.,* SAT-hard locking embed structures (e.g., look-up tables, multipliers) that realize complicated SAT formulas, which increase the time taken per attack iteration [12]. The techniques under (iii), *i.e.,* cyclic locking, instantiate feedback cycles to thwart I/O-based attacks [13]. Inserting cycles inhibits the locked circuit from being modeled as a directed acyclic graph, an important requirement for most I/O-based attacks. Finally, the techniques under (iv), *i.e.,* scan locking, obfuscate the scan data, limiting the controllability and observability of internal nets [14], a prime enabler behind the success of I/O-based attacks. We consider the techniques in (i) because of their algorithmic security guarantees in thwarting I/O-based attacks.

A. Arms Race Between Attackers and Defenders in PSLL

SARLock [10] and Anti-SAT [11] were the first techniques to thwart I/O-based attacks. These techniques add point-functions to the original circuit, thereby necessitating an attacker to apply exponential input patterns (regarding key-size) to recover the secret key. However, both techniques were thwarted by *bypass* attack [15] and removal attacks.[2]

Researchers adopted the paradigm of corrupt and correct-based PSLL techniques (also known as stripped-functionality logic locking (SFLL)) where designers enforce controlled corruption for user-specified input pattern(s) by hard-coding them using point-functions. These errors are corrected when the correct key is provided through a key-controlled unit [3]. However, attackers successfully recovered the secret key through structural and functional analysis [16], [17]. A logic removal-based locking approach (SFLL-rem) [18] demonstrated resilience against attackers during a global logic locking competition. However, this technique has been recently circumvented, where researchers demonstrated the intricacies between logic synthesis and logic locking [19]. Researchers proposed improvements over Anti-SAT (viz., CASLock [20]), which thwarted the bypass attack. However, researchers have demonstrated attacks that recovered the secret key [21].

B. Motivation and Research Challenges

As evidenced from the previous sub-section, there has been an arms race between attackers and defenders. Although a plethora of attacks have been proposed; unfortunately, most attacks target specific PSLL techniques, as evidenced next. For instance, the bypass attack [15] demonstrated vulnerabilities in SARLock [10] and Anti-SAT [11] but could not challenge the security of SFLL techniques [3], [18]. The FALL [17] and SFLL-hd-unlocked [16] attacks were successful in recovering the secret key from variants of SFLL-HD but did not apply to SFLL-flex [3] and SFLL-rem [18].

The attacks proposed in [21] broke the security guarantees of CASLock [20] and Anti-SAT [11] but did not consider other PSLL techniques such as SFLL-flex [3], SFLL-rem [18], and corrupt-and-correct (CAC) [22] (to name a few). The sparse prime implicant (SPI) attack [19] recovered the secret key from SFLL-rem and SFLL-HD^0 but did not consider several unbroken PSLL techniques such as CAC [22], diversified tree logic (DTL) [22], Strong Anti-SAT (SAS) [23], and variants of Gen-Anti-SAT [24]. *Despite the existence of all these attacks, nine PSLL techniques have not been tackled from the standpoint of key-recovery attacks.*[3] The aforementioned discussion highlights that state-of-the-art key-recovery attacks are (i) locking technique specific (*i.e.,* the generality is limited) and (ii) unable to challenge the security guarantees of recent PSLL techniques. This leads to our first research challenge.

> **RC1:** Can we formulate generalized attacks that recover the secret key from the hardware implementation of unbroken and broken PSLL techniques?

The I/O-based attacks demonstrated that logic locking techniques having a key-size of k does not necessarily imply k-bit security. The actual security level depends on the mathematical primitive and the scheme construction [26]. Although PSLL techniques are mathematically sound (assuming that DIPs are chosen uniformly at random and are non-repeated, I/O-based attacks require 2^k queries to an oracle) in recovering a k-bit key), the hardware implementation of these techniques leave structural vulnerabilities that attackers exploit to recover the secret key. *Hence, there is a requirement for a security framework that informs a designer regarding the lower-bound security-level attained by the hardware implementation of PSLL techniques.* This leads to our second research challenge.

> **RC2:** Can we develop a security framework that informs designers regarding the lower-bound security-level attained by the hardware implementation of PSLL techniques?

C. Our Research Contributions

Our work addresses the aforementioned research challenges by developing attacks that successfully recover the secret key from the hardware implementation of PSLL techniques. Our attacks (i) apply to a breadth of PSLL techniques, (ii) successfully recover the secret key for five previously broken and nine unbroken PSLL techniques, (ii) support industry-adopted Verilog format, (iv) do not require *a-priori* information, *i.e.,* the functionality of the locked design, (v) are agnostic to the choice of synthesis tool, synthesis commands, technology libraries, and choice of logic gates used to realize the hardware implementation, (vi) are scalable to large-scale designs and key-sizes, and (vii) can be utilized as a diagnostic tool by designers to ascertain the lower bound security-level attained by the hardware implementation of PSLL techniques. The primary contributions of our work are as follows.

---

[2]Removal attacks identify (and isolate) the protection logic and remove it from the locked circuit. Removing the protection logic yields the original circuit to an attacker. Although we acknowledge the existence of removal attacks, we restrict the discussion to key-recovery attacks in this work.

[3]We refer interested readers to our work in [25] where we showcased removal attacks. However, as stated previously (footnote 2), this work aims to recover the secret key from the hardware implementation of PSLL techniques.

TABLE I
EFFICACY OF OUR PROPOSED KEY-RECOVERY ATTACKS AGAINST THE STATE-OF-THE-ART ATTACKS

Attack \ Defense	SARLock	Anti-SAT	SFLL-HD^0	SFLL-flex	SFLL-rem	CASLock	ECE	SAS	Gen-Anti-SAT		CAC	DTL		
									Comp.	Non-comp.		SARLock	Anti-SAT	CAC
SAT [7]	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Bypass [15]	●	●	○	○	○	○	○	○	○	○	○	○	○	○
SFLL-hd-unlocked [16]	○	○	●	○	○	○	○	○	○	○	○	○	○	○
FALL [17]	○	○	●	○	○	○	○	○	○	○	○	○	○	○
SPI [19]	●	●	●	○	●	○	○	○	○	○	○	○	○	○
CASUnlock [21]	○	●	○	○	○	●	○	○	○	○	○	○	○	○
This Work	●	●	●	●	●	●	●	●	●	●	●	●	●	●

●Successful attack ○Unsuccessful/undocumented attack

- We conceptualize and implement two generalized attacks that recover the secret key from the hardware implementation of 14 PSLL techniques, **including nine unbroken techniques.** Our attacks leverage structural and functional properties stemming from the underlying construction of PSLL techniques coupled with VLSI testing principles and Boolean transformations (§III and §IV). Our attacks apply to a breadth of PSLL techniques, as opposed to other attacks that have been PSLL technique-specific (Table I).
- We demonstrate the efficacy of our key-recovery attacks by performing experiments across 30,000 locked circuits. **Our attacks achieve 100% accuracy in recovering the secret key for all locked circuits.** *Our attacks are agnostic to the choice of (i) synthesis tool, (ii) synthesis commands, (iii) technology libraries, and (iv) logic gates used during synthesis.* In short, our analysis illustrates the inadequacies of academic and commercial CAD tools used for realizing hardware implementation of PSLL techniques (§V).
- We present interesting insights from our attacks (§V-C) and suggest that security-enforcing designers and developers of PSLL techniques utilize our attacks as a diagnostic tool (§V-E). Using our attacks, designers can ascertain the lower-bound security level (within a few minutes) achieved by the hardware implementation of a newly developed PSLL technique. Our analysis reveals that structural security of hard-coded[4] PSLL techniques depend on the choice of the secret key, which calls for further investigation on the secure hardware implementation of PSLL techniques.
- Finally, we shall release our artifacts to foster the development of new attacks and defense techniques.

II. BACKGROUND AND PRELIMINARIES

A. Notations and Definitions

Notations. Let $\mathbb{B} = \{0,1\}$ be the Boolean domain. The notation $\{x_0, x_1, x_2\}$ denotes a set of elements x_0, x_1, and x_2. We denote a set A as a subset of set B as $A \subseteq B$. We use italics to denote variables such as primary inputs or $PI = \{I_i\}$, where $i \in \{0 \ldots n-1\}$, primary outputs or $PO = \{O_i\}$, where $i \in \{0 \ldots m-1\}$, protected input ports or $PIP \subseteq PI$, protected output ports or $POP \subseteq PO$, key-inputs or $KI = \{K_i\}$, where $i \in \{0 \ldots k-1\}$, wires (edges) or $E \in \{n_0, n_1, \ldots, n_{p-1}\}$, and gates (vertices) or $V \in \{v_0, v_1, \ldots, v_{q-1}\}$. Constant pattern is denoted as PP for protected pattern, TP for

test pattern, f for fault value, and K for secret key value. A pattern value can be denoted as $\langle p_0, p_1, p_2 \rangle$, where $\{p_0, p_1, p_2\} \in \{0,1\}$. Notation $a \wedge b$ denotes conjunction (AND) of a and b, $a \vee b$ denotes disjunction (OR), $a \oplus b$ denotes exclusive or (XOR), and $\neg a$ denotes logical negation (NOT).

A combinational circuit $\mathcal{C}_{orig}$ is a directed acyclic graph (DAG) having n PIs and m POs implementing a Boolean function $F : PI \rightarrow PO$, where $PI = \{0,1\}^n$ and $PO = \{0,1\}^m$. It contains p wires and q gates. A logic locking technique $\mathcal{L}$ locks $\mathcal{C}_{orig}$ with a secret key K to obtain a locked circuit $\mathcal{C}_{lock}$. $\mathcal{C}_{lock}$ is $L : PI \times KI \rightarrow PO$. $KI = \{0,1\}^{|K|}$, where $|K|$ denotes the cardinality of KI and is called the key-size. $\mathcal{C}_{lock}$ is fabricated by an *untrustworthy* foundry and converted into a chip $\mathbb{C}_{lock}$. After $\mathbb{C}_{lock}$ is tested and packaged, a *trustworthy* facility (e.g., design house) activates the chip by loading the tamper-proof memory with the correct key K to obtain an activated chip $\mathbb{C}_{act}$. This activated chip is also known as an oracle in the logic locking community. $\mathcal{A}^{\mathbb{S}}$ denotes an attacker $\mathcal{A}$ following an attack strategy $\mathbb{S}$. The goal of an attacker $\mathcal{A}^{\mathbb{S}}$ is to recover a key K_{rec} such that $\mathbb{C}_{lock}(i, K_{rec}) = \mathcal{C}_{orig}(i), \forall i \in I$. Upon a successful key-recovery attack, the recovered circuit $\mathcal{C}_{rec}$ is functionally equivalent to the original circuit $\mathcal{C}_{orig}$, *i.e.*, $\mathcal{C}_{rec}(i) = \mathcal{C}_{orig}(i), \forall i \in I$.

Definition 1. *Algorithmic security [3]. A logic locking technique $\mathcal{L}$ is α-secure against an attacker $\mathcal{A}^{\mathbb{IO}}$ making a polynomial number of I/O queries $q(\alpha)$ to a working chip $\mathbb{C}_{act}$, if he/she cannot reconstruct $\mathcal{C}_{rec}$ correctly with a probability P_{succ} greater than $\frac{q(\alpha)}{2^\alpha}$.*

Definition 2. *Structural security [19]. A logic locking technique $\mathcal{L}$ is β-secure against an attacker $\mathcal{A}^{\mathbb{S}}$ performing white-box structural analysis of the locked circuit $\mathcal{C}_{lock}$, if the probability to recover the secret is no greater than $\frac{1}{\beta}$.*

B. Threat Model

Now, we discuss the capabilities of an attacker $\mathcal{A}$, motivation for the attack, and different attack settings. An attacker reverse-engineers the GDSII[5] information and extracts the gate-level netlist of the locked circuit $\mathcal{C}_{lock}$. In addition, she has access to a test pattern generation (TPG) tool $\mathcal{T}$ and a synthesis tool $\mathcal{S}$. She also has access to a working copy of the chip $\mathbb{C}_{act}$ (a.k.a. *oracle*) with the secret key loaded in

---

[4]Hard-coded PSLL techniques are explained in detail in §II-C.

[5]An industry-standard binary file format used by designers for sharing layout-level information (pertaining to an IC) with foundries.

the tamper-proof memory. Note that under our threat model (which is consistent and agreed upon by researchers in the logic locking community), (i) access to $\mathcal{C}_{lock}$ is unrestricted and (ii) access to $\mathbb{C}_{act}$ is restricted, *i.e.,* an attacker can only use $\mathbb{C}_{act}$ to make oracle queries (an attacker can apply input pattern(s) and observe the output response(s)). Additionally, we assume that an attacker cannot insert Trojans or probe the tamper-proof memory or the key-registers to recover the secret key. Furthermore, an attacker (i) knows the type of PSLL technique implemented by the defender, and (ii) can distinguish between PIs and KIs. All the assumptions are consistent with *Kerckhoffs's principle*, which states that everything about the system should be known to an attacker *except* for the secret key. The objective of an attacker is to extract the secret key K from the hardware implementation of a given PSLL technique which would enable her to pirate the design IP and/or engage in overproduction of ICs.

Using the aforementioned resources and capabilities available to an attacker, we define two attack settings.

- **Oracle-less setting:** An attacker $\mathcal{A}^{\mathbb{OL}}$ only uses the locked circuit $\mathcal{C}_{lock}$ to recover the secret key.
- **Oracle-guided setting:** An attacker $\mathcal{A}^{\mathbb{OG}}$ uses the locked circuit $\mathcal{C}_{lock}$ and the oracle $\mathbb{C}_{act}$ to recover the secret key.

C. Classification of PSLL Techniques

A crypto-system exhibits *provable security* when mathematical proofs exist showcasing resilience to certain attacks [27]. A logic locking technique exhibits provable security when it is algorithmically secure against I/O-based attacks under the aforementioned threat model and assumptions discussed next.

- The effort required by an attacker to determine the correct key K, is exponential in the key-size $|K|$, *i.e.,* $\mathcal{O}(2^{|K|})$.
- An attacker is restricted from probing the oracle.

Based on the underlying hardware construction, we categorize PSLL techniques into *hard-coded* and *non-hard-coded* techniques.[6] A hard-coded PSLL technique $\mathcal{L}^{\mathbb{HC}}$ constitutes a pair of algorithms (*Perturb, Restore*). The *Perturb* algorithm takes the circuit $\mathcal{C}_{orig}$ and protected pattern PP as inputs and returns a functionality-stripped (or modified) circuit $\mathcal{C}_{mod}$, and the associated key K. The *Restore* algorithm takes the modified circuit $\mathcal{C}_{mod}$ and augments a key-controlled restore unit $\mathcal{C}_{restore}$, thereby generating a locked circuit $\mathcal{C}_{lock}$ ($\mathcal{C}_{lock}$ = $\mathcal{C}_{mod} \oplus \mathcal{C}_{restore}$). Conversely, a non-hard-coded PSLL technique $\mathcal{L}^{\mathbb{NHC}}$ consists just of algorithm *Restore*. The *Restore* algorithm takes the circuit $\mathcal{C}_{orig}$ as an input, augments a key-controlled restore unit $\mathcal{C}_{restore}$, and returns a locked circuit $\mathcal{C}_{lock}$, and the associated key K; $\mathcal{C}_{lock} = \mathcal{C}_{orig} \oplus \mathcal{C}_{restore}$.

Hard-coded PSLL techniques (Fig. 1(a)) comprise of techniques where the secret is hard-coded (the key is either hard-coded directly through KIs or indirectly through PIs). Examples include SARLock [10], SFLL-HD^0 [3], SFLL-flex [3], SFLL-rem [18], CAC [22], SARLock-DTL [22], CAC-DTL [22], and error-controlled encryption (ECE) [28]. A designer hard-codes the secret either by (i) augmenting hard-coded point-functions [3], (ii) replacing a few logic
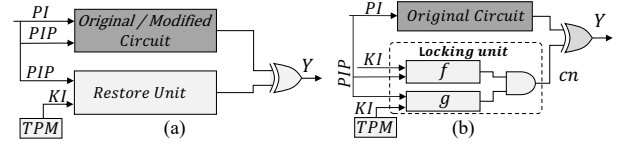
---

[6]Out of the many monikers used for different PSLL techniques, we adopt a simpler and generalized categorization of PSLL techniques.

Fig. 1. High-level construction of (a) hard-coded PSLL techniques (b) non-hard-coded PSLL techniques. PI is primary input, PIP is protected input port, KI is key-input, TPM is tamper-proof memory, and cn is critical wire. For hard-coded PSLL techniques like SARLock [10] and ECE [28], the original circuit is present (as is) and the secret is hard-coded in the restore unit through KIs. For techniques like CAC, CAC-DTL [22], and SFLL variants [3], [18], the construction comprises a modified circuit and a restore unit.

gates in point-functions with OR/NOR gates [22], or (iii) by removing logic [18]. On the other hand, non-hard-coded techniques (Fig. 1(b)) do not hard-code the secret key in the circuit. Examples include Anti-SAT [11], Anti-SAT-DTL [22], CASLock [20], Strong Anti-SAT (SAS) [23], and variants of Gen-Anti-SAT [24]. The construction comprises two logic functions, f and g (Fig. 1(b)), appended to the original circuit through an XOR gate. While f corresponds to $\overline{g}$ for Anti-SAT, Anti-SAT-DTL, CASLock, and Gen-Anti-SAT (Comp.), f can be any function for Gen-Anti-SAT (Non-comp.) [24].

D. Primer on IC Testing

IC testing is a critical step in the supply chain that ensures that a fabricated chip does not possess manufacturing defects. A single stuck-at fault model is widely used to test circuits for faults [29]. A wire is tested for both stuck-at-0 (s-a-0) and stuck-at-1 (s-a-1) fault. Detecting an s-a-0 fault (at a wire) entails generating a test pattern (TP) that sets the wire to logic 1. There are three steps involved in generating TP, viz., (i) fault activation, (ii) path sensitization, and (iii) line justification. Consider Fig. 2(a); we wish to detect an s-a-0 fault on $n1$. $n1$ is output of an AND gate; hence, the input pattern activating $n1$ to logic 1 is $\langle a, b \rangle = \langle 1, 1 \rangle$. The next step is identifying a path to sensitize this value to a PO ($O1$). Since only one gate exists in the fan-out of $n1$, the path includes $n1 \rightarrow O1$. The final step is line justification. This step sensitizes other inputs of the logic gate connected to $n1$ to a known value. As $n1$ fan-outs to NOR gate, the other input of NOR gate ($n2$) must be 0 to propagate the value at $n1$ to $O1$. $n2$ is connected to an OR gate and is set to logic 0 using input pattern $\langle c, d \rangle = \langle 0, 0 \rangle$. Thus, the pattern generated to detect a s-a-0 fault at $n1$ is $\langle a, b, c, d \rangle = \langle 1, 1, 0, 0 \rangle$, as shown in Fig. 2(b). $\mathcal{T}$ generates TP to detect faults at any given wire.

E. Common Functions Used in Our Key-Recovery Attacks

Here we define the functions used in our key-recovery attacks. The PIs of a wire or gate n_0 is extracted using $startpoints(n_0)$; similarly POs of net or gate n_0 is extracted using $endpoints(n_0)$. A logic cone corresponding to a wire or gate n_i is extracted using $extract\_cone(n_i)$. A topologically sorted list of wires lying in the logic cone of PO (O_1) is extracted using $net\_conn(O_1)$. Gate connected to KI (K_0) is obtained using $gate\_conn(K_0)$. The type of logic gate v_0 is obtained using $tech\_mapping(v_0)$. A topologically

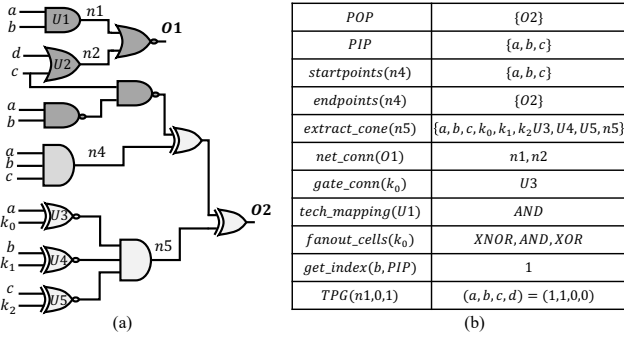| POP	{O2}
PIP	{a, b, c}
startpoints(n4)	{a, b, c}
endpoints(n4)	{O2}
extract_cone(n5)	{a, b, c, k_0, k_1, k_2U3, U4, U5, n5}
net_conn(O1)	n1, n2
gate_conn(k_0)	U3
tech_mapping(U1)	AND
fanout_cells(k_0)	XNOR, AND, XOR
get_index(b, PIP)	1
TPG(n1,0,1)	(a, b, c, d) = (1,1,0,0)

(a) (b)

Fig. 2. Example to illustrate testing principles and common functions.

sorted list of gates in the fanout of input K_0 is obtained using $fanout\_cells(K_0)$. An element at index i in set KI is denoted as $KI[i]$. An index of element k_i in set KI is obtained using $get\_index(k_i, KI)$. A set of test patterns TP_i, where $i \in \{0, \ldots, d\text{-}1\}$ is obtained using an TPG tool $(\mathcal{T})$ to detect stuck-at-fault $f \in \{0, 1\}$ at wire n_0. $\mathcal{T}(n_0, f, d)$ generates d TPs to detect fault f at wire n_0, where d is a user-defined parameter. A synthesis tool $(\mathcal{S})$ is used to perform Boolean transformation using standard logic gates (NAND,AND,OR,NOR,XOR,XNOR,INV). An attacker $(\mathcal{A}^{\mathbb{IO}})$ having access to an oracle $\mathbb{C}_{act}$ can launch the SAT-based attack $SAT()$ to recover the secret key K. The functions are explained using an example in Fig. 2(b).

III. ATTACK ON HARD-CODED PSLL TECHNIQUES

In this section, we conceptualize and develop an attack that recovers the secret key from the hardware implementation of hard-coded PSLL techniques.

Problem formulation. Given access to $\mathcal{C}_{lock}$ locked using $\mathcal{L}^{\mathbb{HC}}$ and black-box access to a working chip $\mathbb{C}_{act}$, recover the secret key K_{rec} such that $\mathbb{C}_{act}(i) = \mathcal{C}_{lock}(i, K_{rec})$, $\forall i \in PI$.

A. Challenges

Our key-recovery attack aims to recover the secret hard-coded protected pattern (PP) that induces output corruption in hard-coded PSLL techniques. Protecting a circuit using a hard-coded PSLL technique becomes ineffective if the hard-coded PP does not influence output corruption. Once the designer converts an algorithmic description of a PSLL technique into its equivalent hardware implementation, the underlying locked circuit becomes a sea of gates and wires. From an attacker's perspective, examining each logic gate for leaking the secret key can be computationally challenging. Moreover, the complexity is further exacerbated when designers utilize intricate logic optimization algorithms to generate locked circuits. Additionally, hard-coded PSLL techniques use varied methods to hard-code the secret (§II-C). Furthermore, some hard-coded PSLL techniques protect multiple PPs [3]. Thus, an attacker faces the following challenges.

C1 How to identify potential vulnerabilities in a locked circuit and subsequently recover the secret key?

C2 How to develop a generic key-recovery attack to challenge the security of hard-coded PSLL techniques? In other words, the attack should be agnostic to the underlying construction of the hard-coded PSLL technique.

B. Methodology

To address **C1**, we articulate the following properties rooted in the construction of hard-coded PSLL techniques.

Property 1. The locked circuit must remain testable, *i.e.,* at least one input pattern exists that detects s-a-0 and s-a-1 faults at every net in the locked circuit. Formally, $\Pr[\mathcal{T}(n_i, f, \mathtt{d}) = \perp] = 0$, $\exists_\mathtt{d} \forall_{i,f}$: $i \in E$, $f \in \{0, 1\}$. This property is directly associated with the principles of IC testing [29] and applies to the hardware implementation of all PSLL techniques.

Recall that in a hard-coded PSLL technique, *Perturb* algorithm generates $\mathcal{C}_{mod}$ from $\mathcal{C}_{orig}$, which can be accomplished either by (i) inserting a hard-coded point-function or (ii) removing functionality corresponding to this PP. $\mathcal{C}_{mod}$ will differ (in functionality) from $\mathcal{C}_{orig}$ only for PPs, *i.e.,* there could exist logic cone(s) inside $\mathcal{C}_{mod}$ which on the application of PPs as inputs would invert the output response of $\mathcal{C}_{orig}$ to induce corruption and obtain $\mathcal{C}_{mod}$. We define the output of such logic cone(s) as *key-revealing logic gate(s)*—these only activate on the application of PPs to induce output corruption. Based on the construction of the hard-coded PSLL techniques considered in this work, we outline two properties that aid in identifying the potential key-revealing logic gate(s).

Property 2. Key-revealing logic gate(s) $(v \in V)$ must be connected to PIP, *i.e.,* they must have primary inputs $ins \subseteq PIP$ as the startpoints. Formally, $ins = startpoints(v), v \in V$, $ins \subseteq PIP$.

Key-revealing logic gate(s) can be either connected to exactly the same number of PIP (as the key-size) or a number lesser than the key-size to account for synthesis-induced transformations and merging with the original design. Following the construction of hard-coded PSLL techniques, PIP must be involved in the construction of the key-controlled restore unit. For example, for techniques where PP is hard-coded (e.g., SFLL-HD^0), $PIP \subseteq PI$, whereas, for techniques where the *key* is hard-coded (e.g., SARLock), $PIP \subseteq KI$.

Property 3. Key-revealing logic gate(s) $(v \in V)$ must influence the corruption of only POP. Simply put, key-revealing logic gate(s) must have only POP as the endpoints. Formally, $POP = endpoints(v), v \in V$.

Note that both properties (i) are derived naturally from the hardware implementation of hard-coded PSLL techniques, and (ii) prune the search space for an attacker. However, with increased design complexity, *i.e.,* designs with larger gate count (e.g., b19_C with 237,962 gates), attackers might end up with a large number of key-revealing logic gate(s). To prune the search space further, we outline an additional property.

Recall that augmenting point functions (with hard-coded PP) to the original circuit ensure I/O-based attacks prune out exactly one incorrect key in every attack iteration, thereby leading to stronger resilience (§I-A). Therefore, the construction of hard-coded PSLL techniques necessitates the activation of key-revealing logic gate(s) only when PP is applied as an input pattern. For instance, if the key-revealing logic gate outputs 0 for any non-PP, then applying PP must toggle this value to 1. From an attacker's perspective, the challenge lies in recovering the secret PP. To recover this secret, an attacker can perform functional simulations (using random input patterns) and observe the output at the key-revealing logic gate(s).

However, obtaining input pattern(s) that justifies key-revealing logic gate(s) to 1 through random simulations is challenging due to exponential complexity regarding the number of inputs. *This challenge is addressed by utilizing the principles of test pattern generation.* More specifically, we use the stuck-at fault model to generate TPs that detect faults for any logic gate. Finally, although most hard-coded PSLL techniques protect one PP, some techniques protect multiple PPs [3]. These points lead to the articulation of our next property.

Property 4. Key-revealing logic gate(s) ($v \in V$) must be activated for exactly N input patterns. In other words, exactly N input patterns should set the value of the key-revealing logic gate(s) to 1. Formally, $\Pr[|\mathcal{T}(v, 0, d)| = N] = 1$, $d > N$, $v \in V$. N is a user-configurable parameter and depends on the underlying construction of the hard-coded PSLL technique. For instance, N is 1 for SFLL-HD^0 while it can be any number for SFLL-flex, which protects multiple PPs.

Generality. The aforementioned properties are dictated by the construction of the considered hard-coded PSLL techniques. Nevertheless, structural and functional properties corresponding to new PSLL techniques can be readily augmented to these properties, which ensures attack upgradability. *Note that these properties are applicable as long as the underlying PSLL technique demonstrates exponential complexity against I/O-based attacks.* All our properties hold irrespective of how the defender constructs a modified circuit, *i.e.,* either by adding logic (e.g., CAC [22]) or by removing logic (SFLL-rem [18]).

Theorem 1. *A partial (or complete) secret can be recovered with test patterns using the stuck-at fault model for a hard-coded PSLL technique satisfying exponential SAT complexity.*

Proof. Consider a PP is hard-coded in the original circuit ($\mathcal{C}_{orig}$) to obtain a modified circuit ($\mathcal{C}_{mod}$) such that the functionality of $\mathcal{C}_{mod}$ differs from $\mathcal{C}_{orig}$ only for PP. Irrespective of how $\mathcal{C}_{mod}$ is constructed to achieve exponential complexity (either by augmenting a point-function, diversifying point-function, or removing logic), it will constitute key-revealing logic gate(s) that will (i) influence corruption of PO(s), (ii) be testable for both faults, and (iii) be activated only for PP. As the effect of key-revealing logic gate(s) can be observed at PO, a stuck-at fault model can be used to generate a TP that controls the output of the key-revealing logic gate(s). Considering Equation 1, a TP is generated to test stuck-at fault f, where $f \in \{0,1\}$ at the output of key-revealing logic gate(s), cn, in $\mathcal{C}_{mod}$. For the correct fault, the computed TP contains partial (or complete) traces of the hard-coded secret.

$$\text{TP} \leftarrow \mathcal{T}(cn, f, 1); \qquad \text{TP} \in \text{K} \qquad (1)$$

Thus, TP generated using the stuck-at fault model can recover (either complete or partial) traces of the hard-coded secret. □

Idea. As discussed, key-revealing logic gate(s) shall induce output corruption at POP on the application of PPs. Therefore, the effect of PPs can also be observed on internal nets of the key-revealing logic gate(s). Such nets shall remain dormant (inactive) for non-PPs and activate only for PPs. Hence, an attacker can resort to generating a TP to detect s-a-0 (s-a-1) on

nets in the key-revealing logic gate(s). Such a TP will include traces of the hard-coded PP.

Example 1. Consider $\mathcal{C}_{orig}$ in Fig. 3(a) and $\mathcal{C}_{mod}$ in Fig. 3(b) that is modified for the input pattern $\langle a, b, c, d \rangle = \langle 1, 0, 0, 1 \rangle$. Using property 3, all the nets are classified as key-revealing logic gate(s) since they affect the POP (Y). Using property 2, we perform a pruning operation on key-revealing logic gate(s). This returns $n0$, $n1$, and $n2$ since they are connected to more than $N/2$ $PIPs$, where N is the length of the PP. Using property 4, we further prune key-revealing logic gate(s) to $n1$ and $n2$. $\mathcal{T}$ generates $\langle a, b, c, d, e \rangle = \langle 1, x, 0, 1, x \rangle$ for net $n1$ and $\langle a, b, c, d, e \rangle = \langle 1, 0, 0, 1, x \rangle$ for net $n2$. x denotes undeciphered bit. Thus, we recover the hard-coded secret $\langle a*, b*, c*, d* \rangle = \langle 1, 0, 0, 1 \rangle$ in an oracle-less setting by testing for s-a-0 fault at $n2$.

Example 2. Consider $\mathcal{C}_{mod}$ in Fig. 3(c) that is modified for the input pattern $\langle a, b, c, d \rangle = \langle 0, 0, 0, 0 \rangle$. All the nets are classified as key-revealing logic gate(s) using property 3. Performing a pruning operation using property 2 returns $n0$ and $n1$. Using property 4, we are only left with $n1$. Invoking $\mathcal{T}$ for $n1$ generates a TP as $\langle 0, 0, x, 0, x \rangle$. Thus, we extracted a partial hard-coded secret key value (K) as $\langle 0, 0, x, 0 \rangle$ by testing for stuck-at faults at $n1$. The undeciphered bit (x) can be revealed by querying an oracle through an I/O-based attack (e.g., SAT-based attack [7]). This example illustrates the scenario where an oracle is required to recover the secret key.

C. Algorithm

We outline our attack on hard-coded PSLL techniques in Alg. 1. It consists of five steps, viz., (i) extraction of nets in the POP, (ii) obtaining mapping between KIs and $PIPs$, (iii) identification of key-revealing logic gate(s), (iv) generation of TPs, and (v) recovery of the secret key. `ExtractNets()` returns the list of nets lying in the POP (lines 1–4). `KeyInputMapping()` extracts the mapping between KIs and $PIPs$ from the key-controlled restore unit (lines 5–13). `CandidateNets()` returns the key-revealing logic gate(s) that satisfy the properties discussed in §III-B (lines 14–20). Next, the algorithm utilizes TPs to recover the hard-coded PP. $\mathcal{T}$ generates TPs that detects s-a-0 and s-a-1 for key-revealing logic gate(s). Given a PSLL technique, $\mathcal{T}$ generates d TPs. For example, when considering SFLL-HD^0, $\mathcal{T}$ is queried to generate *exactly* two TPs for key-revealing logic gate(s). The correct key-revealing logic gate(s) will return *only* one TP and any key-revealing logic gate(s) generating more than one TP is(are) discarded. TP is fed to `KeyExtraction()` to extract key-bits corresponding to $PIPs$ (lines 21–28). If all key-bits are recovered from TPs, the algorithm outputs it as the secret key. However, there might be scenarios where partial key-bits are recovered from multiple TPs. In such cases, partial key-bits can be combined from the multiple TP. Furthermore, if TPs cannot recover some key-bits, $SAT()$ is invoked to recover them. Finally, the algorithm merges the partial key recovered from TPs and the key returned from the $SAT()$ to output the final key.

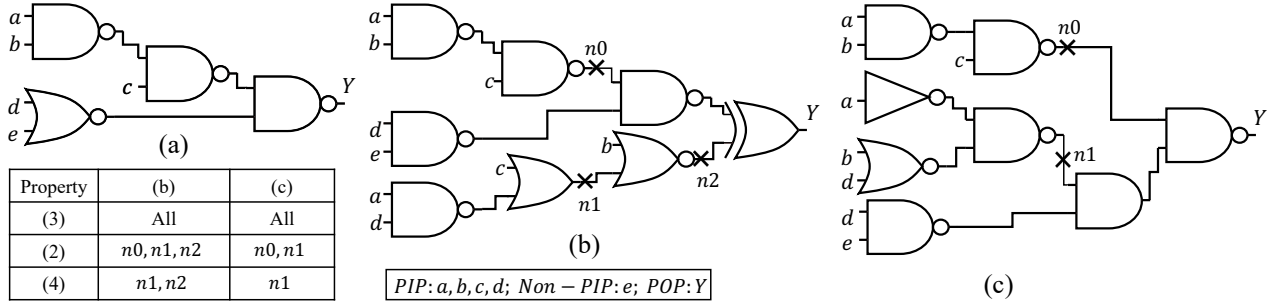| Property	(b)	(c)
(3)	All	All
(2)	$n0, n1, n2$	$n0, n1$
(4)	$n1, n2$	$n1$

$PIP: a, b, c, d;\ Non-PIP: e;\ POP: Y$

Fig. 3. Example showing identification of key-revealing logic gate(s) using properties of hard-coded PSLL techniques. (a) Original circuit (b) Modified circuit (with regards to functionality) for $PP = \langle 1, 0, 0, 1 \rangle$ for PIP $\langle a, b, c, d \rangle$. (c) Modified circuit for $PP = \langle 0, 0, 0, 0 \rangle$ for PIP $\langle a, b, c, d \rangle$. Table outlines the key-revealing logic gate(s) identified per property. For (b), there are two key-revealing logic gate(s) and for (c), there is only one key-revealing logic gate.
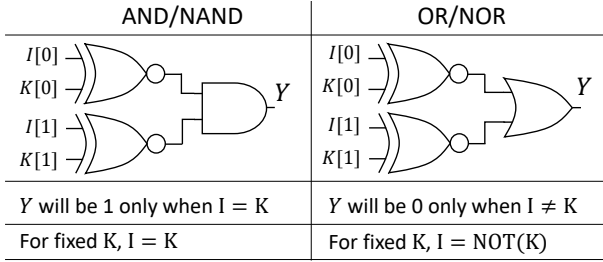

Fig. 4. Key-recovery example where point-functions are diversified.

D. Extension to Other Hard-coded PSLL Techniques

Recall that **C2** outlines the challenge of developing a generic key-recovery attack agnostic to the construction of a hard-coded PSLL technique (§III-A). Next, we discuss the (minor) modifications we implemented to address **C2**.

Some hard-coded PSLL techniques (e.g., CAC-DTL [22], and variants of SFLL [3], [18]) hard-code the secret through PIs in the original circuit, while others (e.g., SARLock-DTL [22], ECE [28]) hard-code the secret through KIs in the restore unit. Our attack successfully recovers the secret key across both classes of techniques. Since the PP corresponds to $PIPs$ for techniques such as CAC, CAC-DTL, and SFLL variants, the attack extracts key-bit corresponding to $PIPs$ from TPs. For SARLock, SARLock-DTL, and ECE, the PP corresponds to KIs; hence, instead of $PIPs$, key-bits corresponding to KIs are extracted from TP. This is achieved by removing lines 22–23 in Alg. 1 and modifying line 24 to $val \leftarrow \text{TP}[k]$.

The second scenario we address towards the generality of our key-recovery attack is to account for the number of PPs. While SFLL-HD^0 protects exactly one PP, SFLL-flex protects multiple PPs. Our attack addresses this using property 4 and utilizes a user-configurable parameter TP that constrains the $\mathcal{T}$ tool to produce exactly N test patterns.

Finally, in scenarios where a designer diversifies the hard-coded point function using OR/NOR gates [22], we recover the secret key as follows. When a designer replaces/diversifies some AND gates in hard-coded point-function with NAND/OR/NOR gates, we slightly modify the key extraction strategy from TPs. Fig. 4 illustrates the relation between I and hard-coded key K for two examples. When AND gates are replaced with NAND gates, there is no change

in `KeyExtraction()`. However, when AND gates are replaced with OR/NOR gates, the relation between K and I changes, as shown in column 2, row 4. Thus, with modifications to `KeyExtraction()`, our attack recovers the secret key for different versions of DTL.

IV. ATTACK ON NON-HARD-CODED PSLL TECHNIQUES

In this section, we develop an attack that recovers the secret key from the hardware implementation of non-hard-coded PSLL techniques. The problem formulation is the same as mentioned in §III and is omitted here.

A. Challenges

The construction of non-hard-coded PSLL techniques consists of a key-controlled locking unit appended to the original circuit via one critical wire cn (Fig. 1(b)). It should be noted that the secret resides in the key-controlled locking unit, and therefore, the first step for an attacker is to structurally analyze the locked circuit to identify the critical wire cn that separates the locking unit from the original circuit. However, as discussed in §III-A, the complexity of identifying this wire (net) is challenging and further exacerbated due to synthesis-guided logic optimizations. The next challenge is how to recover the secret key from the locking unit and how can a generic attack be developed for non-hard-coded PSLL techniques, independent of the construction of the locking unit. To summarize, an attacker faces the following challenges.

C3 How to identify the locking unit from a locked circuit and recover the secret key from the locking unit?

C4 How to develop a generic key-recovery attack for non-hard-coded PSLL techniques?

B. Methodology

To address **C3**, the first step entails identifying the locking unit from the locked circuit. We outline a property stemming from the construction of non-hard-coded PSLL techniques.

Property 5. The wire cn (that separates the locking unit from the original circuit) must be connected to all KI and all PIP. Also, cn must influence the output corruption of POP. Formally, $\{KI, PIP\} = startpoints(cn)$ and $POP = endpoints(cn)$. If multiple candidates for cn exist, then the

---

Algorithm 1: Attack on hard-coded PSLL techniques

Input: Locked circuit ($\mathbb{C}_{lock}$), Oracle ($\mathbb{C}_{act}$), Primary inputs (PI),
Key-inputs (KI), Number of protected patterns ($|PP|$)
Output: Secret key (K)

1 procedure ExtractNets(*keyin,in*):
2 $POP \leftarrow endpoints(keyin)$
3 $N \leftarrow net\_conn(POP)$
4   **return** N

5 procedure KeyInputMapping(*N*):
6 $PIP, KEY \leftarrow \emptyset$
7   **for** $net \in N$ **do**
8 $ins \leftarrow startpoints(net)$
9   **if** $|ins| = 2$ **then**
10 $(key,pip) \leftarrow ins$ where $key \in KI$
11 $PIP.append(pip)$
12 $KEY.append(key)$

13   **return** PIP, KEY

14 procedure CandidateNets(*N,PIP*):
15 $CN \leftarrow \emptyset$
16   **for** $net \in N$ **do**
17 $in \leftarrow startpoints(net)$
18   **if** $in \in PIP$ **then**
19 $CN.append(net)$

20   **return** CN

21 procedure KeyExtraction(TP,*PIP,K,PI*):
22   **for** k in $\{0,|$K$|\}$ **do**
23 $pip \leftarrow PIP[k]$
24 $idx \leftarrow get\_index(pip, PI)$
25 $val \leftarrow$ TP$[idx]$
26   **if** $val \neq x$ **then**
27 $key[k] \leftarrow val$

28   **return** key

29 function KeyRecovery_HC:
30 $\{N\} \leftarrow$ ExtractNets(KI, PI)
31 $\{KEY, PIP\} \leftarrow$ KeyInputMapping(N)
32 $\{CN\} \leftarrow$ CandidateNets(N, PIP)
33   **for** cn in CN **do**
34   TP $\leftarrow \mathcal{T}(cn, i, |PP|+1)$; $i \in \{0,1\}$
35   **if** $|$TP$| = |$PP$|$ **then**
36   K $\leftarrow$ KeyExtraction(TP, PIP, KEY, PI)
37   **if** $|$K$| < |KI|$ **then**
38 $key_final \leftarrow SAT(\mathbb{C}_{lock}, \mathbb{C}_{act}, K)$
39   K $\leftarrow key\_final$

40   **return** K

---

wire closest (shortest distance measured in levels of logic) to the KI is chosen for extracting the locking unit.

After successfully extracting the locked unit, the next step involves recovery of the secret key from the locking unit. To that end, we first provide the definition of key-gate mapping.

Recall the construction of non-hard-coded PSLL techniques where a key-controlled locking unit is XORed with the original circuit to obtain a locked circuit (Fig. 1(b)). For Anti-SAT, blocks f and g are complementary to each other and denoted by g and $\overline{g}$. The blocks are controlled by the same $PIPs$ but different KIs ($K = \{\mathcal{K}_1, \mathcal{K}_2\}$), where $\mathcal{K}_j$ consists of key-inputs k_{ji}; $i \in \{0, \ldots, |K|/2 - 1\}$, $j \in \{1, 2\}$. Locking unit can be formally defined as $Y = g(PIP_i \oplus k_{1i} \oplus r_{1i}) \wedge \overline{g(PIP_i \oplus k_{2i} \oplus r_{2i})}$, $\forall i \in \{0, |K|/2 - 1\}$; r as 0(1) indicates an X(N)OR key-gate. This construction forces I/O-based attacks to query at least $2^{|K|/2}$ input patterns, where $|K| = |\mathcal{K}_1| + |\mathcal{K}_2|$ is the total key-size. For instance, for a circuit locked

using Anti-SAT with key-size $|K|$, it takes $2^{|K|/2}$ queries to recover the secret key; each query to an oracle eliminates $2^{|K|/2} - 1$ incorrect keys. This means, there are $2^{|K|/2}$ correct keys, and when analyzed further, we identify a unique mapping between $\mathcal{K}_1$ and $\mathcal{K}_2$ portions of the correct keys, *i.e.,* $\mathcal{K}_1 \oplus \mathcal{K}_2$ across all the correct keys is unique. We define this unique mapping as *key-gate mapping* (KGM) and seek to find this mapping to recover the secret key(s) from the hardware implementation of a non-hard-coded PSLL technique. Finally, we introduce our last property, which considers the gate-type for r (X(N)OR) in the successful recovery of the unique mapping. Unlike the properties discussed so far, this is an assumed property.

Property 6. Each KI must drive exactly one X(N)OR logic gate. Formally, $\Pr[gate\_conn(k_i) = \text{X(N)OR}] = 1$, $k_i \in KI$.

Theorem 2. *A key-gate mapping exists between the two sets of keys in non-hard-coded PSLL techniques.*

Proof. As per the aforementioned definition of locking unit, Y depends on $PIPs$ and KIs (k_{1i} and k_{2i}) and will only be 0 when ($PIP_i \oplus k_{1i} \oplus r_{1i}$) equals ($PIP_i \oplus k_{2i} \oplus r_{2i}$). Operation $k_{ji} \oplus r_{ji}$ denotes either XOR or XNOR key-gate $\forall j \in \{1, 2\}$ and the corresponding key-values can be either $\langle 0, 0 \rangle$ or $\langle 1, 1 \rangle$ when $r_{1i} = r_{2i}$, or $\langle 0, 1 \rangle$ or $\langle 1, 0 \rangle$ when $r_{1i} \neq r_{2i}$. Thus, there is a definite mapping between the key-gates in $\mathcal{K}_1$ and $\mathcal{K}_2$ bins, which can derive the secret key(s). $\square$

In a pre-synthesized locked circuit, the construction of the non-hard-coded PSLL technique is retained ($g \wedge \overline{g}$). Thus, the correlation between individual key-inputs can be derived. For example, consider Fig. 5(a), $k0$ is correlated with $k4$ since both the KIs are XORed with the same PIP $I0$. Similarly $k2$ is correlated with $k6$ since they are both XORed with the same PIP $I2$. Subsequently, all the correlated KIs are distributed into two bins, $\mathcal{K}_1$ and $\mathcal{K}_2$. Once the correlation between KIs is identified, and the subsequent binning process is completed, our KGM attack extracts the following attributes per KI.

- Bin it belongs to ($\mathcal{K}_1$ or $\mathcal{K}_2$)
- Gate it is connected to (XOR or XNOR)?
- Is there an inversion (or not) on its output path?

Table II describes the recovery of the secret key using the aforementioned attributes. Considering columns 2 and 3, keeping key-bin and gate-type constant, the effect of inversion can be observed on the key-value. Similarly, consider columns 6 and 8; the effect of gate-type can be observed on the key-value, while with columns 5 and 9, we can observe the effect of key-bin on the key-value. Next, we discuss the application of the KGM attack using two examples for Anti-SAT.

Example 3. Consider the pre-synthesized Anti-SAT locking unit in Fig. 5(a), where $k0$ and $k4$ are connected to an XOR and XNOR gate. As per the proof of Theorem 2, key-values for these two key-bits should be either $\langle 0, 1 \rangle$ or $\langle 1, 0 \rangle$. Next, we check the key-values obtained using our attack. Before applying key-gate mapping, we distribute the KIs, $k0$ and $k4$, corresponding to the same PI ($I0$), into two bins, $\mathcal{K}_1$ and $\mathcal{K}_2$. Observe that $k4$ sees inversion on its signal path (belongs to $\overline{g}$), whereas $k0$ sees no inversion (belongs to g). Thus, as per Table II, $k0$ is 0 and $k4$ is 1, consistent with
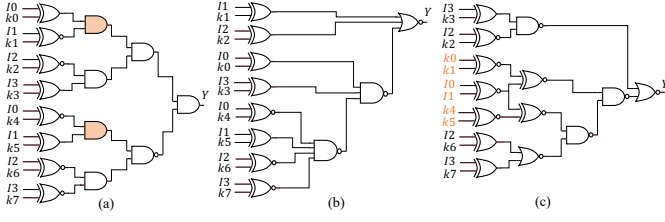
Fig. 5. a) Pre-synthesis Anti-SAT block. (b) Post-synthesis Anti-SAT block (all standard gates). (c) Post-synthesized Anti-SAT-DTL block when gates in orange in (a) are replaced with XOR gates.

TABLE II
KEY-GATE MAPPING TABLE. KEY-BIT VALUE IS DEPENDENT ON KEY BIN, GATE-TYPE, AND INVERSION IN THE FANOUT

	Key-inputs							
Key-bin	1	1	1	1	2	2	2	2
Gate-type	XOR	XOR	XNOR	XNOR	XOR	XOR	XNOR	XNOR
Inversion?	No	Yes	No	Yes	No	Yes	No	Yes
Key-value	0	1	1	0	1	0	0	1

TABLE III
EXTRACTING ATTRIBUTES OF EACH KEY-INPUT AND RECOVERING SECRET KEY FROM LOCKING UNIT FOR EXAMPLE 4.

Key-input	k0	k1	k2	k3	k4	k5	k6	k7
Key-bin	1	1	1	1	2	2	2	2
Gate-type	XOR	XOR	XOR	XOR	XNOR	XOR	XNOR	XNOR
Inversion?	No	Yes	Yes	No	Yes	Yes	Yes	Yes
Key-value	0	1	1	0	1	0	1	1

our result above. Note that the key-mapping between the two sets must be unique and hence there are two correct values corresponding to $k0$ and $k4$, $\langle 0, 1 \rangle$ and $\langle 1, 0 \rangle$. Similarly, we recover the remaining key-values, leading to the secret key $\langle k0, k1, k2, k3, k4, k5, k6, k7 \rangle$ as $\langle 0, 1, 1, 0, 1, 0, 1, 1 \rangle$ and the secret mapping between $\mathcal{K}_1$ and $\mathcal{K}_2$ ($\mathcal{K}_1 \oplus \mathcal{K}_2$) as $\langle 1, 1, 0, 1 \rangle$. All the ($2^4$) keys satisfying this mapping are the secret keys.

Example 4. Consider the post-synthesized Anti-SAT locking unit in Fig. 5(b), where the KIs are connected to XOR and XNOR gates. First, we perform key-binning by distributing the KIs into two bins. Thus, $\{k0, k1, k2, k3\}$ are binned into $\mathcal{K}_1$, while $\{k4, k5, k6, k7\}$ are binned into $\mathcal{K}_2$. We utilize the strategy outlined in Table II and traverse the signal path of the logic gates connected to each KI to keep track of any signal inversions. Note that NAND/NOR gates also account for inversions in addition to INV gates. We recover the secret key value K as $\langle 0, 1, 1, 0, 1, 0, 1, 1 \rangle$ by scrutinizing the attributes of each KI as per Table III.

Corner cases. To address **C4**, we identify various corner cases and address them in our attack. Some synthesis-induced optimizations may challenge an attacker in successfully identifying the correlation between KIs. We can address this challenge by performing another round of logic optimization on the extracted locking unit using only standard logic gates. Once correlation between the KIs is obtained and correlated KIs are distributed into distinct bins, the value of key-bits can be recovered using Table II in an oracle-less setting.

Example 5. Consider the post-synthesized circuit in Fig. 5(c), which is obtained when gates marked in orange in Fig. 5(a) are replaced with XOR gates (Anti-SAT-DTL) and subsequently synthesized. First, we perform key-binning by distributing the KIs into two bins depending on their connected $PIPs$. Thus, $\{k0, k1, k2, k3\}$ are binned into $\mathcal{K}_1$, while $\{k4, k5, k6, k7\}$ are binned into $\mathcal{K}_2$. However, $\{k0, k1, k4, k5\}$ do not have a definite correlation. We utilize the strategy outlined in Table II for the correlated KIs. Note that due to inconclusive correlation between key-inputs $\{k0, k1, k4, k5\}$, key-bits corresponding to $k0$ and $k1$ remain undeciphered using our KGM attack. We assign $k4$ and $k5$ random key-bits since they lie in a different bin than $k0$ and $k1$. We invoke the SAT-based attack to recover these two key-bits. We successfully recover the secret key value K as $\langle 1, 1, 1, 0, 0, 0, 1, 1 \rangle$ by scrutinizing the attributes of correlated KIs as per Table IV and invoking SAT-based attack for the undeciphered key-bits.

Our KGM attack recovers the key in an oracle-less setting if all KIs are successfully correlated and binned into distinct sets. However, assume KIs are (i) connected to the

same XOR/XNOR gate, (ii) connected to gates other than XOR/XNOR, or (iii) driving multiple logic gates. In such cases, binning is unsuccessful, and the attack annotates them as undeciphered key-bits (x). Our KGM attack recovers these key-bits by executing the SAT-based attack [7].

C. Algorithm

We outline our generic attack on non-hard-coded PSLL techniques in Alg. 2. It consists of five steps, viz., (i) identification of critical wire, (ii) extraction of the locking unit, (iii) re-synthesis of extracted logic cone, (iv) extraction of attributes, and (v) recovering key-bits. Identification of the critical wire (cn) and subsequent extraction of the locking unit ($\mathcal{C}_{cn}$) takes place in `ExtractLogicCone()`. The extracted logic cone, $\mathcal{C}_{cn}$, is re-synthesized using $\mathcal{S}$ with only the standard gates to generate $\mathcal{C}_{cn\_syn}$. Next, `GetAttribute()` is executed on $\mathcal{C}_{cn\_syn}$ which extracts attributes such as the set each KI belongs to, the type of logic gates connected to each KI, and the presence of inversion on the output path. These attributes are passed to `KeyMapping()` to recover the secret key corresponding to the KIs. Suppose the algorithm cannot conclusively determine the attribute of any KI. In that case, the value of that KI is annotated as x. Finally, we invoke $SAT()$ to recover the undeciphered KIs using an oracle, $\mathcal{C}_{act}$.

Time complexity. Table V showcases the time complexity of functions and procedures used in our key-recovery attack algorithms (Alg. 1 and 2). Although ATPG is an NP-complete problem, efficient heuristics have been developed for practical circuits, which reduces this complexity to polynomial in the number of gates in the circuits [30]. Synthesis approaches have undergone decades of research, with worst-case complexity being $\mathcal{O}(E^3)$ [31]. In this work, since we use a commercial, closed-source synthesis tool (*i.e., Synopsys DC*), the time complexity of our KGM attack algorithm cannot be conclusively obtained. The overall time complexity of our attacks is contingent on the underlying structure (graph representation) and the number of gates in the locked circuit.

V. EXPERIMENTAL INVESTIGATION

In this section, we demonstrate the efficacy of our key-recovery attacks on the hardware implementation of PSLL

TABLE IV
EXTRACTING ATTRIBUTES OF EACH KEY-INPUT AND RECOVERING
SECRET KEY FROM LOCKING UNIT FOR EXAMPLE 5

Key-input	k0	k1	k2	k3	k4	k5	k6	k7
Key-bin	1	1	1	1	2	2	2	2
Gate-type	-	-	XOR	XOR	-	-	XNOR	XNOR
Inversion?	-	-	Yes	No	-	-	Yes	Yes
Key-value	x	x	1	0	0	0	1	1

- denotes unknown value

TABLE V
TIME COMPLEXITY OF THE FUNCTIONS AND PROCEDURES USED IN OUR
KEY-RECOVERY ATTACK ALGORITHMS

Function	Time complexity
$startpoints()$	$\mathcal{O}(|V| + |E|)$
$endpoints()$	$\mathcal{O}(|V| + |E|)$
$net\_conn()$	$\mathcal{O}(|V| + |E|)$
$get\_index()$	$\mathcal{O}(1)$
$append()$	$\mathcal{O}(1)$
$extract\_cone()$	$\mathcal{O}(|V| + |E|)$
$gate\_conn()$	$\mathcal{O}(1)$
$tech\_mapping()$	$\mathcal{O}(1)$
$fanout\_cells()$	$\mathcal{O}(|V| + |E|)$
ExtractNets()	$\mathcal{O}(|V| + |E|)$
KeyInputMapping()	$\mathcal{O}(|V| + |E|)$
CandidateNets()	$\mathcal{O}(|N| * (|V| + |E|))$
KeyExtraction()	$\mathcal{O}(|K|)$
ExtractLogicCone()	$\mathcal{O}(|N| * (|V| + |E|))$
GetAttribute()	$\mathcal{O}(|V| + |E|)$
KeyMapping()	$\mathcal{O}(1)$

V: vertices (gates)   E: edges (nets)   N: candidate nets (N $\subset$ E)

techniques across different parameters such as choice of (i) technology library, (ii) synthesis tool, (iii) synthesis commands, (iv) type of logic gates used for synthesis, and (v) key-size. In addition, we also elucidate some important findings.

A. Experimental Setup

Locking techniques. We implement all the PSLL techniques considered in this work using *Perl* and *Python* on three abstraction levels (*BENCH*, RTL, and synthesized *Verilog*). We lock the circuits with a key-size of 128 for SARLock, SARLock-DTL, SFLL-HD^0, SFLL-flex, SFLL-rem, CAC, CAC-DTL, and ECE. We choose a key-size of 256 for Anti-SAT, Anti-SAT-DTL, CASLock, SAS, and the variants of Gen-Anti-SAT. The number of PP is 16 for SFLL-flex and we diversify the AND-tree by replacing 16 gates in the AND-tree with OR gates for DTL techniques. We consider 4 SAS blocks for [23].
Circuits. We demonstrate the efficacy of our key-recovery attacks on eight combinational circuits from the ITC-99 suite. We lock each circuit 100 times to capture variations in the selection of $PIPs$ and locking different $POPs$.
Tool setup. We perform synthesis of locked circuits using two commercial synthesis tools (*Synopsys Design Compiler (DC)* and *Cadence Genus*) and one academic synthesis tool (*ABC* [32]). We obtain TPs using an academic TPG tool, *ATALANTA* [33]. We use two synthesis recipes,[7] *synth_A* and *synth_B*, when synthesizing circuits using *Synopsys DC*. While *synth_A* comprises {compile_ultra, compile_ultra

[7]Denotes a sequence of logic optimization commands.

Algorithm 2: Attack on Non-hard-coded Techniques

Input: Locked design ($\mathcal{C}_{lock}$), Oracle ($\mathbb{C}_{act}$), Key-inputs (KI)
Output: Secret key ($\mathbb{K}$)

1 **procedure** ExtractLogicCone(*c,keyin*):
2     $POP \leftarrow endpoints(keyin)$
3     $N \leftarrow net\_conn(POP)$
4   **for** $net \in N$ **do**
5        $ins \leftarrow startpoints(net)$
6   **if** $(keyin \subseteq ins)$ **then**
7           $cn \leftarrow net$
8     $\mathcal{C}_{cn} \leftarrow extract\_cone(\mathcal{C}, cn)$
9   **return** $\mathcal{C}_{cn}$

10 **procedure** GetAttribute(*key,PIP,gate*):
11     $bin \leftarrow 1$
12     $\{ins\} \leftarrow startpoints(gate)$
13   **if** $ins \subseteq PIP$ **then**
14        $bin \leftarrow 2$
15        $PIP.append(ins)$
16     $lib \leftarrow tech\_mapping(gate)$
17     $inv \leftarrow 0$
18   **if** 'INV' $\subseteq fanout\_cells(key)$ **then**
19        $inv \leftarrow 1$
20   **return** bin,lib,inv

21 **procedure** KeyMapping(*bin, gateType, inv*):
22     $key\_value \leftarrow 0$
23   **if** $(bin = 1$ & $inv = 1$ & $lib = `XOR') | (bin = 2$ & $inv = 0$ & $lib = `XOR') | (bin = 1$ & $inv = 0$ & $lib = `XNOR') | (bin = 2$ & $inv = 1$ & $lib = `XNOR')$ **then**
24        $key\_value \leftarrow 1$
25   **return** key_value

26 **function** KeyRecovery_NHC:
27     $\mathcal{C}_{cn} \leftarrow$ ExtractLogicCone($\mathcal{C}_{lock}, KI$)
28     $\mathcal{C}_{cn\_syn} \leftarrow \mathcal{S}(\mathcal{C}_{cn})$
29     $PIP \leftarrow \emptyset$
30   **for** key in KI **do**
31        $idx \leftarrow get\_index(key, KI)$
32        $key\_value[idx] \leftarrow$ x
33        $gate \leftarrow gate\_conn(key)$
34        $\{bin,lib,inv\} \leftarrow$ GetAttribute($key, PIP, gate$)
35        $key\_value[idx] \leftarrow$ KeyMapping(bin, lib, inv)
36   **if** 'x' $\subseteq key\_value$ **then**
37        $\mathbb{K} \leftarrow SAT(\mathcal{C}_{lock}, \mathbb{C}_{act}, key\_value)$
38   **else**
39        $\mathbb{K} \leftarrow key\_value$
40   **return** $\mathbb{K}$

-incremental}, *synth_B* comprises of three instantiations of compile_ultra followed by three instantiations of compile_ultra -incremental. We use the command "synthesize -to_mapped" with different efforts (medium and high) for *Cadence Genus*. Furthermore, we utilize six synthesis recipes (*resyn*, *resyn2*, *resyn2a*, *resyn3*, *compress*, and *compress2*) within the ABC tool [32] to verify the efficacy of our proposed attacks.

Attack setup and evaluation metrics. We implement our key-recovery attacks using *TCL* and C++ scripts integrated with *Synopsys DC* and *ATALANTA*. We use two metrics to assess the efficacy of our attacks, viz., (i) accuracy, by computing the number of correct key-bits divided by the key-size and (ii) precision, by checking the correctness of each key-bit. We perform verification of the recovered key using a combinational equivalence checker within the *ABC* tool. We

TABLE VI

AVERAGE EXECUTION TIME (IN SECONDS) FOR **KEY-RECOVERY ATTACK** ACROSS HUNDRED RANDOM TRIALS. LOCKED CIRCUITS ARE SYNTHESIZED USING SYNOPSYS DC WITH ALL LOGIC GATES FROM THE NANGATE 45NM LIBRARY

Defense Circuit	Hard-coded								Non-hard-coded					
	SARLock	SFLL			CAC	ECE	DTL		Anti-SAT	CASLock	Gen-Anti-SAT		DTL	SAS
		HD^0	flex	rem			SARLock	CAC			Comp.	Non-comp.	Anti-SAT	
b14_C	35	17	16	104	22	37	34	111	34	38	44	38	32	25
b15_C	42	37	37	157	43	47	43	128	37	37	48	39	34	26
b20_C	83	31	31	100	52	86	83	138	67	73	110	68	73	72
b21_C	80	32	32	159	58	85	81	164	65	73	106	70	72	69
b22_C	121	39	37	212	82	123	119	161	93	103	168	95	87	106
b17_C	158	69	62	166	122	164	158	266	137	131	317	141	107	133
b18_C	556	361	365	405	587	570	565	743	444	540	624	620	436	1,013
b19_C	637	513	545	676	557	543	668	931	681	1,034	868	966	829	1,345

TABLE VII

AVERAGE EXECUTION TIME (IN SECONDS) FOR **KEY-RECOVERY ATTACK** ACROSS HUNDRED RANDOM TRIALS. LOCKED CIRCUITS ARE SYNTHESIZED USING SYNOPSYS DC WITH ALL LOGIC GATES FROM GLOBALFOUNDRIES 65NM LIBRARY

Defense Circuit	Hard-coded								Non-hard-coded					
	SARLock	SFLL			CAC	ECE	DTL		Anti-SAT	CASLock	Gen-Anti-SAT		DTL	SAS
		HD^0	flex	rem			SARLock	CAC			Comp.	Non-comp.	Anti-SAT	
b14_C	48	20	20	98	28	50	46	114	38	42	47	42	35	75
b15_C	64	43	53	134	50	65	66	138	37	46	48	36	34	98
b20_C	103	31	41	113	62	104	103	143	73	87	96	80	72	115
b21_C	100	33	40	178	63	106	100	143	75	94	87	76	67	120
b22_C	142	40	49	243	86	140	146	169	96	126	137	110	85	153
b17_C	192	71	83	276	125	202	193	234	110	144	169	110	109	224
b18_C	643	395	401	527	659	614	670	789	629	611	599	602	615	971
b19_C	667	579	511	632	711	579	723	887	777	1,161	907	911	1,120	1,459

TABLE VIII

AVERAGE EXECUTION TIME (IN SECONDS) FOR **KEY-RECOVERY ATTACK** ACROSS HUNDRED RANDOM TRIALS. LOCKED CIRCUITS ARE SYNTHESIZED USING ABC WITH ONLY 2-INPUT AND GATES AND INVERTERS

Defense Circuit	Hard-coded								Non-hard-coded					
	SARLock	SFLL			CAC	ECE	DTL		Anti-SAT	CASLock	Gen-Anti-SAT		DTL	SAS
		HD^0	flex	rem			SARLock	CAC			Comp.	Non-comp.	Anti-SAT	
b14_C	36	27	27	114	27	34	35	133	42	43	53	46	39	72
b15_C	48	54	56	147	51	45	51	158	48	51	61	51	43	87
b20_C	70	46	49	104	45	71	72	158	81	90	98	86	77	103
b21_C	68	48	50	133	48	72	72	155	78	87	99	84	72	106
b22_C	92	59	60	134	59	89	90	176	108	116	140	114	98	121
b17_C	124	98	100	154	102	118	129	219	175	205	216	179	146	204
b18_C	160	229	269	514	230	152	165	430	255	769	312	415	306	929
b19_C	345	551	449	917	545	347	368	912	444	972	784	595	958	1,101

execute our attacks on a 128-core Intel Xeon processor running at 2.4 GHz having, 512 GB of RAM.

B. Results of Key-Recovery Attacks

Applicability. Our proposed attacks successfully recover the secret key (with accuracy and precision of 100%) from the hardware implementation of all 14 PSLL techniques considered in this work. *More importantly, our attacks highlight security vulnerabilities in nine previously unbroken locking techniques* (CAC [22], CAC-DTL, SARLock-DTL, Anti-SAT-DTL [22], SFLL-flex [3], ECE [28], Gen-Anti-SAT (Comp. and Non-comp.) [24], and SAS [23]). We perform a detailed comparison with other key-recovery attacks in §V-D.

Execution time. We document the execution time of our key-recovery attacks across 14 locking techniques for ITC-99 circuits in Table VI, Table VII, and Table VIII, respectively. We derive the execution time for each locking technique and

locked circuit by averaging attack runtimes across 100 random trials. We follow three setups, as explained next. First, we synthesize the locked circuits using *Synopsys DC* with all Boolean logic gates available in a technology library (full-library) for *Nangate 45nm* and *GlobalFoundries 65nm*. We document the average attack execution time for the aforementioned setup in Table VI and Table VII. Next, we synthesize the locked circuits in a technology-agnostic manner by using an academic synthesis tool, *ABC* [32]. We document the attack execution time for this setup in Table VIII. Across all the considered PSLL techniques, benchmarks, and the aforementioned setups, our attack recovers the secret key (accuracy and precision of 100%) in a maximum of 1,013 and 1,459 seconds for the two largest circuits (b18_C with 117,941 gates and b19_C with 237,962 gates) from the ITC-99 suite.

Oracle-less versus Oracle-guided attacks. Here, we provide further details regarding the efficacy of our key-recovery

TABLE IX
NUMBER OF SAT-BASED ATTACK ITERATIONS (OR ORACLE QUERIES) REQUIRED TO RECOVER UNDECIPHERED KEY-BITS. KEY-SIZE IS 128

Defense Circuit	SARLock				SARLock-DTL				CAC				CAC-DTL				ECE				SFLL-flex			
	Synth_A		Synth_B		Synth_A		Synth_B		Synth_A		Synth_B		Synth_A		Synth_B		Synth_A		Synth_B		Synth_A		Synth_B	
	min	max	min	max	min	max	min	max	min	max	min	max	min	max	min	max	min	max	min	max	min	max	min	max
b14_C	2	16	2	128	2	16	2	32	2	32	2	8	2	8	2	64	2	16	2	32	2	64	4	128
b15_C	2	16	8	256	2	16	2	128	2	2	2	8	2	8	2	128	2	16	8	64	2	4	2	16
b20_C	2	16	2	64	2	16	2	64	2	8	2	8	2	32	2	32	2	16	4	64	2	4	2	8
b21_C	2	16	2	64	2	16	2	64	2	2	2	8	2	16	2	16	2	32	2	64	2	2	4	32
b22_C	2	16	8	64	2	16	2	64	2	2	2	8	2	32	2	64	2	32	4	64	2	16	2	64
b17_C	2	8	2	256	2	16	2	128	2	2	2	16	2	16	2	64	2	16	2	32	2	4	2	16
b18_C	2	16	2	64	2	16	2	64	2	2	2	8	2	16	2	64	2	16	4	128	2	4	2	4
b19_C	2	16	2	256	2	16	2	256	2	2	2	8	2	16	2	32	2	16	4	512	2	4	2	8

attacks by distinguishing whether the attack recovers the key in an *oracle-less* or an *oracle-guided* setting. Recall that in an oracle-less setting, an attacker has access to only the locked circuit, while in an oracle-guided setting, an attacker has access to a working chip and the locked circuit (§II-B). We depict the minimum (and maximum) number of SAT-based attack iterations required to recover the secret key for two different synthesis settings (*synth_A* and *synth_B*) for some PSLL techniques in Table IX. Recall that we leverage an oracle (by launching the SAT-based attack [7]) to recover the undeciphered key-bits (Alg. 1 and 2). On average, our attacks correctly recover a high percentage (94.5%) of key-bits using only structural analysis of the locked circuit, *i.e.,* in an oracle-less setting. For instance, the maximum number of undeciphered key-bits across all locking techniques and circuits is nine for a key-size of 128 (Table IX).

Note that our key-recovery attacks do not target vulnerabilities in the underlying PSLL algorithm, rather we identify and leverage structural vulnerabilities in the hardware implementation of the considered PSLL techniques to recover the secret key. Our attacks perform structural analysis of the locked circuit and recover the secret key by either (i) leveraging TPs in hard-coded PSLL techniques or (ii) utilizing attributes about KIs in non-hard-coded PSLL techniques. Since any Boolean function can be realized using different structural representations, it is imperative that we evaluate the efficacy of our attacks on locked circuits with varied structural representations. Therefore, we perform a thorough analysis with regards to the choice of (i) technology libraries (academic/commercial), (ii) synthesis tools (academic/commercial), (iii) synthesis commands, (iv) logic gates used for synthesis, and (v) protected pattern, for different circuits and PSLL techniques. All the aforementioned parameters dictate the underlying structure (or graph-based representation) of locked circuits. Although we do not showcase the attack execution time for all cases (due to limited space), we illustrate the distribution of oracle-less (oracle-guided attacks) for four PSLL techniques.

Effect of technology library. To evaluate the effect of using different technology libraries (academic versus commercial), we synthesize the locked circuits using *Synopsys DC* using only two-input gates with the same synthesis commands. The variable parameter is the technology library and the timing constraints for synthesis. We illustrate the distribution of oracle-less to oracle-guided attacks (stacked bar graphs) for four PSLL techniques in Fig. 6. When locked circuits are

synthesized using *Nangate 45nm* library, our key-recovery attack recovers the secret key in an oracle-less setting in 50% of cases for CAC-DTL, 35% for ECE, 43.12% for Gen-Anti-SAT (Comp.), and 51.25% for Gen-Anti-SAT (Non-comp.). These numbers are 38.75% for CAC-DTL, 39.38% for ECE, 53.75% for Gen-Anti-SAT (Comp.), and 47.5% for Gen-Anti-SAT (Non-comp.) when circuits are synthesized using *GlobalFoundries 65nm* library. The remaining circuits are broken in an oracle-guided setting. **This analysis highlights the efficacy of our attacks across technology libraries.**

Effect of synthesis tool. To evaluate the effect of using different synthesis tools, we fix the technology node (45nm) and the type of gates used for synthesis (2-input gates). We showcase the distribution of oracle-less to oracle-guided attacks in Fig. 7. A majority of trials (78.75% and 62.5%) for CAC-DTL and Gen-Anti-SAT (Comp.) require oracle access when locked circuits are synthesized using *Cadence Genus*. The percentage of trials requiring oracle access dropped to 43.13% and 56.88% when locked circuits are synthesized using *Synopsys DC*. This analysis demonstrates the efficacy of our attacks across different commercial synthesis tools.

Effect of types of gates. To evaluate the effect of utilizing different types of gates (2-input gates versus full-library) for synthesis, we fix the synthesis tool (*Synopsys DC*), technology library (*Nangate 45nm*), and synthesis commands (`compile_ultra` + `compile_ultra -incremental`). We showcase the distribution of oracle-less and oracle-guided attacks in Fig. 8. Most trials (78.13%, 76.88%, and 65%) for CAC-DTL, ECE, and Gen-Anti-SAT (Comp.) require oracle access when synthesis is accomplished using the full library. The percentage of trials requiring oracle access dropped to 41.88%, 65%, and 56.88% when synthesis is performed using two-input gates. The availability of diverse logic gates during full library synthesis enables the synthesis tool to optimize the structure of locked circuits. Such optimizations hinder the recovery of a *few* key-bits when the attacker only has access to a locked circuit. However, an attacker can decipher the value of these unknown key-bits using an oracle.

Effect of synthesis commands. To evaluate the effect of synthesis commands, we fix the synthesis tool (*Synopsys DC*), type of logic gates used for synthesis (2-input gates), and the technology library (*Nangate 45nm*). We observe aggressive synthesis optimizations (*synth_B*) lead to more undeciphered key-bits (when an attacker attempts to recover the secret key in an oracle-less setting), thereby requiring access to an oracle to recover the secret key (Fig. 9).
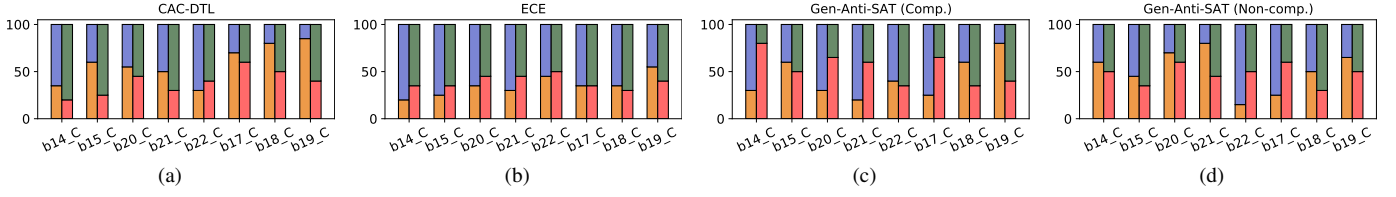
Fig. 6. Distribution of oracle-less (oracle-guided) attacks for different technology libraries. Orange (violet) denote oracle-less (oracle-guided) results for *Nangate 45nm* technology library. Red (green) denote oracle-less (oracle-guided) results for *GlobalFoundries 65nm* technology library.
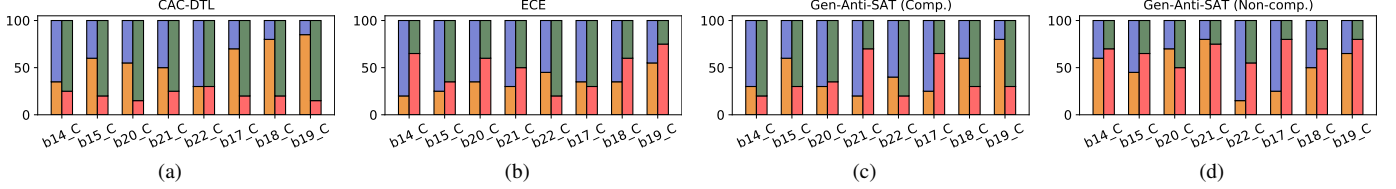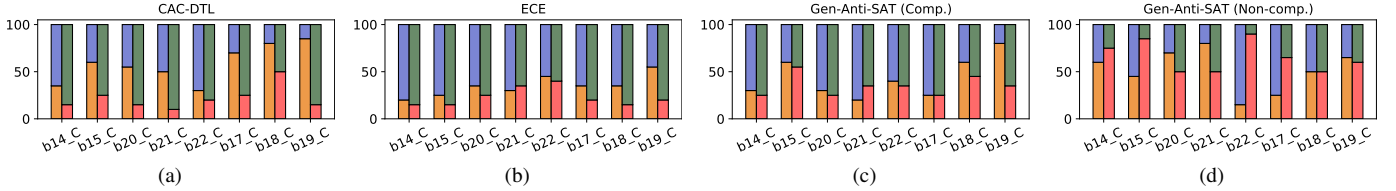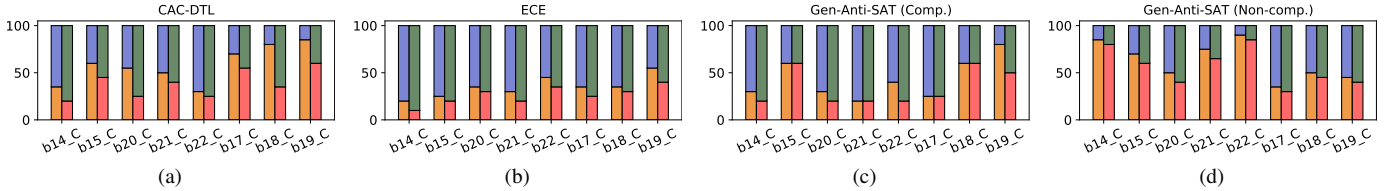

Fig. 7. Distribution of oracle-less (oracle-guided) attacks for different synthesis tools. Orange (violet) denote oracle-less (oracle-guided) results for circuits synthesized using *Synopsys DC*. Red (green) denote oracle-less (oracle-guided) results for circuits synthesized using *Cadence Genus*.

Fig. 8. Distribution of oracle-less (oracle-guided) attacks for different type of gates used in synthesis. Orange (violet) denote oracle-less (oracle-guided) results for circuits synthesized using two-input gates. Red (green) denote oracle-less (oracle-guided) results for circuits synthesized using full library.

Fig. 9. Distribution of oracle-less (oracle-guided) attacks for different synthesis settings. Orange (violet) denote oracle-less (oracle-guided) results for circuits synthesized using *synth_A*. Red (green) denote oracle-less (oracle-guided) results for circuits synthesized using *synth_B*.
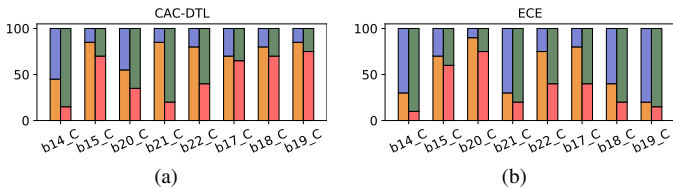

Fig. 10. Distribution of oracle-less (oracle-guided) attacks for different PP for two hard-coded PSLL techniques. Orange (violet) denote oracle-less (oracle-guided) results for circuits synthesized using *synth_A*. Red (green) denote oracle-less (oracle-guided) results for circuits synthesized using *synth_B*.

Effect of PP. To evaluate the impact of PP, we fix the synthesis tool (*Synopsys DC*), the type of logic gates used for synthesis (2-input gates), and the technology library (*Nangate 45nm*). The $PIPs$ and POP are also kept constant across all locked circuits; the only variable parameter is the choice of PP. We illustrate the distribution of oracle-less to oracle-guided

attacks for two hard-coded PSLL techniques in Fig. 10. While we recover the secret key (accuracy of 100%) in all locked circuits, we observe that the role of PP determines whether our attacks recover the secret key in an oracle-less setting (or not). Furthermore, synthesis-induced logic optimizations, *i.e.,* the use of different synthesis recipes also play a crucial role, as evidenced next. While we recover the secret key in an oracle-less setting for 73.12% (CAC-DTL) and 54.38% (ECE) of the trials when locked circuits are synthesized using *synth_A*, these numbers drop to 48.75% (CAC-DTL) and 35% (ECE) when using *synth_B*.

Effect of key-size. We illustrate the impact of increasing key-size on the attack execution time in Fig. 11. Increasing key-size does not affect the accuracy of our key-recovery attacks except for an increase in the attack execution time.

Note that, we do not identify anything wrong in the security proofs of the PSLL techniques considered in our work. The security proofs were directed primarily toward resilience
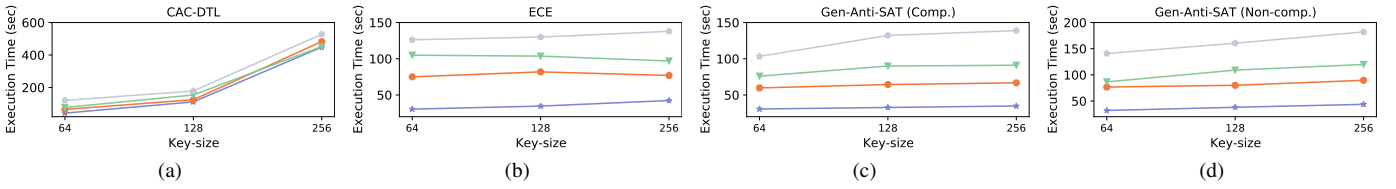
Fig. 11. Effect of increasing key-size ($|K|$=64, $|K|$=128, and $|K|$=256) on the attack execution time for ITC-99 circuits (b14_C, b15_C, b17_C, and b22_C). The gray, green, orange, and blue colors correspond to b17_C, b22_C, b15_C, and b14_C circuits (top to bottom), respectively.

against I/O-based attacks and assumed that an attacker would only query a working chip to recover the secret key. The scenario of an attacker aiming to recover the secret key *only* through input/output pairs from a working chip corresponds to *black-box cryptanalysis*. Just as cryptographic algorithms provide security against an attacker with only black-box access to cryptographic devices, PSLL techniques provide provable-security against an attacker with only black-box access. *However, such a (black-box) model does not always correspond to the realities of hardware implementations.* In a realistic and practical setting, an attacker has access to the locked circuit and the activated chip (§II-B) and attempt to recover the secret key through structural and functional analysis.

> **Takeaway Message:** Our attacks successfully recover the secret key (with 100% accuracy and 100% precision) from the hardware implementation of all the considered PSLL techniques for all circuits across different parameters that include variations in (i) technology libraries, (ii) synthesis tools, (iii) synthesis commands, (iv) type of logic gates used, (v) protected patterns, and (vi) key-sizes.

C. Important Observations from Key-Recovery Attacks

1) Our attack recovers the secret key (with 100% accuracy and 100% precision) in the following hard-coded PSLL techniques (SARLock, SARLock-DTL, ECE, CAC, CAC-DTL, SFLL-HD^0, SFLL-flex) in an *oracle-less* setting when the defender synthesizes locked circuits using an academic synthesis tool *ABC* [32]. An exception is SFLL-rem, where the attack requires access to an oracle to recover some key-bits. *Recovering the secret key in an oracle-less setting is powerful since it undermines the security guarantees of logic locking techniques during fabrication.*

2) Our KGM attack recovers the secret key (100% accuracy and 100% precision) for non-hard-coded PSLL techniques such as Anti-SAT, Anti-SAT-DTL, and Gen-Anti-SAT (Comp.) in an *oracle-less* setting when circuits are synthesized using *ABC*. In addition, the attack recovers the secret key for most cases for CASLock (92.5%) and Gen-Anti-SAT (Non-comp.) (90%) in an *oracle-less* setting. All locked circuits are broken using an oracle for SAS.

3) Our attacks recover the secret key for most PSLL techniques in an *oracle-less setting* across six different synthesis recipes using *ABC*. *In a nutshell, synthesis operations carried out by ABC do not aid in the dissolution of key-revealing logic gates(s) with the original circuit for both hard-coded and non-hard-coded PSLL techniques.*

4) Through our experiments, we observe that synthesis optimizations, usage of different technology libraries, etc., lead to a reduction of point-functions (Fig. 3). *Our attack recovers the secret key successfully independent of procedures used by designers to generate the modified circuit (§II-C).*

5) To increase the output corruption of PSLL techniques and resist approximate attacks (*Double-DIP* [9] and *App-SAT* [8]), researchers proposed DTL [22] and ECE [28]. Our attacks recover the secret key for both techniques since the underlying construction either hard-codes the PP using (i) a point-function (ECE), or (ii) a point-function diversified with OR/NAND gates. The structural hints from key-revealing logic gate(s) (either point-function or reduced point-functions) are captured through TPs, aiding an attacker to recover the secret key.

6) It has been established that hard-coded passwords in software artifacts are indicators of weakness in software security. For example, the CWE-798 mentions that "*if hard-coded passwords are used, it is almost certain that malicious users will gain access to the account in question.*" Our key-recovery attacks attest to this from a hardware perspective, *i.e.,* the hard-coding of secrets in PSLL techniques lead to structural vulnerabilities that attackers can exploit to recover the secret key.

7) Our experimental analysis reveals that the locking unit remains disjoint from the original circuit for all considered non-hard-coded PSLL techniques.

8) Finally, we observe that the choice of PP plays a role in the ability of a hard-coded point-function to merge with the original circuit. During our experiments, we came across a few examples where the considered PP led to a significant dissolution of the point-function. For instance, we observed that hard-coding a specific PP using a 64-input point-function led us to recover only 35 bits using our attack. This (empirical) finding highlights the role of PP toward structural security for hard-coded PSLL techniques.

D. Comparison with State-of-the-Art Key-Recovery Attacks

Generality. Recall that almost all the key-recovery attacks proposed by researchers cater towards a specific locking technique and do not generalize to other techniques (§I-B). On the other hand, our key-recovery attacks apply to a broad category of PSLL techniques (both hard-coded and non-hard-coded), including *nine previously unbroken techniques*.

Scalability. Now we discuss the scalability of our attacks compared to prior key-recovery attacks. We executed the FALL attack [17] on CAC-DTL, Gen-Anti-SAT, SFLL-rem, and SFLL-flex and observed that it was unsuccessful in recovering the
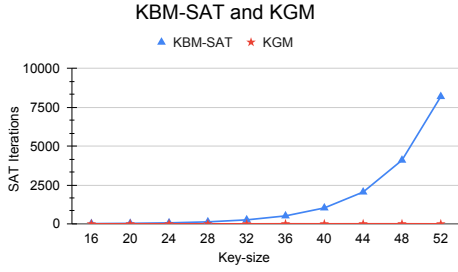
Fig. 12. Efficacy of key-bit mapping (KBM)-SAT attack [21] compared with our KGM attack on b14_C circuit locked using CASLock [20].

secret key. The key-bit mapping (KBM)-SAT attack [21] does not work for all cascaded-chain configurations of AND/OR gates.[8] We showcase the performance of the KBM-SAT and our attack for certain cascaded-chain configurations in Fig. 12. On the contrary, our KGM attack recovers the secret key for all cascaded-chain configurations. The SPI attack [19] breaks SFLL-flex for only one PP. However, SFLL-flex allows a designer to protect multiple PP. Our attack recovers all PP for SFLL-flex in an *oracle-less* setting. Further, our attack is agnostic to the implementation style of the restore unit, *i.e.,* whether constructed using look-up tables [3] or logic gates.

E. Application as a Diagnostic Tool for Designers

Designers can utilize our key-recovery attacks to determine the lower bounds of security achieved by the hardware implementation of a PSLL technique. Recall that our experimental analysis reinforces that implementing a PSLL technique (on hardware) with a key-size of $|K|$ does not necessarily imply $|K|$-bit security. If an attacker recovers a portion of the secret key, $|K^*|$, then the actual security drops to $|K| - |K^*|$ [26]. Since the design IP is the "secret sauce" for industries and defense establishments, the ramifications are immense if the entity in question does not utilize a suitable diagnostic tool to ascertain the structural signatures emanated from the hardware implementation of a PSLL technique.

VI. DISCUSSION

A. Other IP Protection Solutions

Design IPs can either be state-less (combinational) or state-full (sequential). Researchers have proposed several logic locking techniques that lock the combinational portion of the design IP. Furthermore, since industrial design IPs are

---

[8]To evaluate the efficacy of KBM-SAT attack on cascaded-chain configurations of AND/OR gates, we chose a smaller key-size ($|K|$=16) to iterate over all possible configurations. Then, we executed the attack over all configurations, *i.e.,* 2^7 configurations across seven AND/OR gates. We observed many configurations where KBM-SAT attack failed to recover the secret key. Notation-wise, let us represent the cascaded-chain configuration of AND-OR-AND as $\langle 0, 1, 0 \rangle$. As per our experimental analysis for $|K|$=16, a cascaded-chain configuration $\langle 1, 0, 0, 0, 1, 1, 1 \rangle$, KBM-SAT attack requires at least $2^{|K|/4} - 1$ iterations, where $|K|$ is 16. Let us denote the configuration $\langle 1, 0, 0, 0, 1, 1, 1 \rangle$ as $\langle 1, 0^m, 1^m \rangle$, where m is calculated as $16/4 - 1 = 3$. Extrapolating this to a key-size of 128 (512) for the aforementioned configuration, the KBM-SAT attack will require *at least* $2^{32} - 1$ ($2^{128} - 1$) iterations to recover the secret key. We verified our hypothesis by executing the attack for seven days—KBM-SAT was unsuccessful in recovering the secret key. However, our KGM attack successfully recovers the secret key in an *oracle-less setting* for both key-sizes.

inherently sequential, sequential locking (sequential obfuscation) is another promising direction to prevent piracy of design IPs and unauthorized overproduction of ICs. Sequential designs consist of finite state machines (FSMs), which are protected using FSM-based obfuscation. The original FSM is protected by (i) inserting additional states known as obfuscated states [34] and black hole states [35], (ii) locking combinational logic cones of FSM states [36], and (iii) re-encoding states to obscure the boundary between original state registers [37]. Hardware redaction is another approach where sensitive portions of the circuit are replaced with an embedded FPGA [38]. Another direction is eradicating key leakage in the scan mode [36], [39]. Such techniques enable the use of high-corruption locking techniques. Furthermore, provably secure block ciphers or pseudo-random functions can be adopted to thwart cryptanalysis and SAT-based attacks.

B. Comparison with Password Selection and Cracking

A logic-locked circuit can be considered analogous to a password-protected computer system. In logic locking, the secret key is chosen either based on a seed or is a random string of $|K|$ bits. Unlike human-chosen passwords/PINs, the key search space is extensive ($2^{|K|}$) in logic locking [40]. The key-recovery attacks discussed in our work aim to recover secret keys from the hardware implementation of PSLL techniques; this can be considered analogous to password cracking. Analogous to personally identifiable information in targeted online password guessing [41], a partially recovered (correct) secret key from a locked circuit aids an attacker in reducing the security-level of a locked circuit. However, unlike targeted password guessing [41], there is no limit imposed on the number of queries an attacker makes to an oracle.

C. Future Work

Our investigation highlights that vulnerabilities of the considered PSLL techniques stem from their hardware implementation. Although one can argue that synthesis tools prioritize logic optimization over security-driven goals, our observations about specific PP that lead to the dissolution of key-revealing logic gate(s) are important. This means an interplay exists between the structure (and/or functionality) of the underlying circuit and the PP we wish to protect. Searching for this elusive set of PP is computationally challenging due to the exponential complexity of the number of input patterns. Recall that our analysis of non-hard-coded PSLL techniques reveals that the locking unit remains disjoint from the original circuit. Addressing the (i) dissolution of the locking unit, (ii) integration of security algorithms with CAD tools, and (iii) extension of attacks presented in this paper toward sequential obfuscation techniques is reserved for future work.

VII. CONCLUSION

In this work, we conceptualize and develop generalized attacks that recover the secret key from the hardware implementation of PSLL techniques. We extract various structural and functional properties contingent on the underlying hardware

implementation and use (i) principles of test pattern generation and (ii) Boolean transformations to develop two attack algorithms that recover the secret key from locked circuits. We evaluate the efficacy of our attacks across different parameters such as the choice of technology libraries, synthesis tools, synthesis settings, key-size, etc., and observe 100% accuracy for 14 PSLL techniques (including nine previously unbroken techniques). Besides demonstrating the security-obliviousness of current academic and commercial computer-aided design tools, our attacks provide several important insights, viz., (i) the structural security of hard-coded PSLL techniques are contingent on the choice of the secret key, and (ii) the locking unit remains disjoint from the original circuit for non-hard-coded PSLL techniques. Additionally, developers of PSLL techniques can utilize our attacks to ascertain the lower-bound security achieved by hardware implementations. Finally, we release our locked circuits and attack binaries with the hope that (i) it would foster the development of secure hardware implementation of PSLL techniques and (ii) future attackers can benchmark the performance of their developed attacks on common datasets, thereby enabling reproducibility.

REFERENCES

[1] R. Zafar, "TSMC's Total 3nm Investment Will Equal At Least $ 23 Billion," https://wccftech.com/tsmc-3nm-investment-23-billion-project-end/, 2021, [Online; accessed 14-March-2022].

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

[3] 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 SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 1601–1618.

[4] J. A. Roy, F. Koushanfar, and L. Igor, "EPIC: Ending piracy of integrated circuits," in *Proc. Des. Autom. Test Europe*, 2008, pp. 1069–1074.

[5] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Security Analysis of Logic Obfuscation," in *Proc. Des. Autom. Conf.*, 2012, pp. 83–89.

[6] J. Rajendran *et al.*, "Fault Analysis-Based Logic Encryption," *IEEE Transactions on Computer*, vol. 64, no. 2, pp. 410–424, 2015.

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

[8] 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 Secur. Trust*, May 2017, pp. 95–100.

[9] Y. Shen and H. Zhou, "Double DIP: Re-evaluating security of logic encryption algorithms," in *Proc. Great Lakes Symp. VLSI*, May 2017, pp. 179–184.

[10] M. Yasin, B. Mazumdar, J. J. Rajendran, and O. Sinanoglu, "SARLock: SAT attack resistant logic locking," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust*, May 2016, pp. 236–241.

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

[12] H. M. Kamali, K. Z. Azar, H. Homayoun, and A. Sasan, "Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks," in *Proc. 56th Annu. Design Autom. Conf*, Jun. 2019, pp. 1–6.

[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*, 2017, pp. 173–178.

[14] R. Karmakar, S. Chatopadhyay, and R. Kapur, "Encrypt flip-flop: A novel logic encryption technique for sequential circuits," *arXiv preprint arXiv:1801.04961*, 2018.

[15] X. Xu, B. Shakya, M. M. Tehranipoor, and D. Forte, "Novel bypass attack and bdd-based tradeoff analysis against all known logic locking attacks," in *Proc. Int. Conf. Cryptograph. Hardw. Embedded Syst.*, 2017, pp. 189–210.

[16] F. Yang, M. Tang, and O. Sinanoglu, "Stripped functionality logic locking with hamming distance-based restore unit (SFLL-hd)–Unlocked," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 10, pp. 2778–2786, 2019.

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

[18] A. Sengupta, M. Nabeel, N. Limaye, M. Ashraf, and O. Sinanoglu, "Truly stripping functionality for logic locking: A fault-based perspective," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 4439–4452, Dec. 2020.

[19] Z. Han, M. Yasin, and J. J. Rajendran, "Does logic locking work with EDA tools?" in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 1055–1072.

[20] B. Shakya, X. Xu, M. Tehranipoor, and D. Forte, "CAS-Lock: A security-corruptibility trade-off resilient logic locking scheme," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, no. 1, pp. 175–202, Jul. 2020.

[21] A. Sengupta, N. Limaye, and O. Sinanoglu, "Breaking CAS-lock and its variants by exploiting structural traces," *IACR Trans. Cryptograph. Hardw. Embedded Syst*, vol. 2021, no. 3, pp. 418–440, Jul. 2021.

[22] K. Shamsi, T. Meade, M. Li, D. Z. Pan, and Y. Jin, "On the Approximation Resiliency of Logic Locking and IC Camouflaging Schemes," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 2, pp. 347–359, 2018.

[23] Y. Liu, M. Zuzak, Y. Xie, A. Chakraborty, and A. Srivastava, "Strong anti-sat: Secure and effective logic locking," in *Proc. 21st Int. Symp. Quality Electron. Design*, Mar. 2020, pp. 199–205.

[24] J. Zhou and X. Zhang, "Generalized SAT-Attack-Resistant Logic Locking," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 2581–2592, 2021.

[25] N. Limaye, S. Patnaik, and O. Sinanoglu, "Valkyrie: Vulnerability assessment tool and attack for provably-secure logic locking techniques," *IEEE Trans. Inf. Forensics Security*, vol. 17, pp. 744–759, 2022.

[26] F. Guo, W. Susilo, and Y. Mu, *Introduction to Security Reduction*. Springer, 2018.

[27] J. Buchmann, *Introduction to Cryptography*. Springer, 2004, vol. 335.

[28] Y. Shen, A. Rezaei, and H. Zhou, "A comparative investigation of approximate attacks on logic encryptions," in *Proc. 23rd Asia South Pacific Design Autom. Conf.*, Jan. 2018, pp. 271–276.

[29] M. Bushnell and V. Agrawal, *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*. Springer Science & Business Media, 2004, vol. 17.

[30] M. Prasad, P. Chong, and K. Keutzer, "Why is ATPG easy?" in *Proc. Des. Autom. Conf.*, 1999, pp. 22–28.

[31] T. Sasao, *Logic synthesis and optimization*. Springer, 1993, vol. 2.

[32] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-strength Verification Tool," in *Proc. Int. Conf. Comput. Aided Verification*. Springer, 2010, pp. 24–40.

[33] H. Lee and D. Ha, "Atalanta: an Efficient ATPG for Combinational Circuits," in *Technical Report*, 1993.

[34] R. S. Chakraborty and S. Bhunia, "Harpoon: An obfuscation-based soc design methodology for hardware protection," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 10, pp. 1493–1502, 2009.

[35] Y. Alkabani and F. Koushanfar, "Active hardware metering for intellectual property protection and security." in *USENIX Secur. Symp.*, 2007, pp. 291–306.

[36] L. Li and A. Orailoglu, "Janus-HD: Exploiting FSM sequentiality and synthesis flexibility in logic obfuscation to thwart sat attack while offering strong corruption," in *Proc. Des. Autom. Test Europe*, 2022, pp. 1323–1328.

[37] Y. Zhang, Y. Hu, P. Nuzzo, and P. A. Beerel, "TriLock: IC protection with tunable corruptibility and resilience to sat and removal attacks," in *Proc. Des. Autom. Test Europe*, 2022, pp. 1329–1334.

[38] P. Mohan, O. Atli, J. Sweeney, O. Kibar, L. Pileggi, and K. Mai, "Hardware redaction via designer-directed fine-grained efpga insertion," in *Proc. Des. Autom. Test Europe*, 2021, pp. 1186–1191.

[39] N. Limaye, E. Kalligeros, N. Karousos, I. G. Karybali, and O. Sinanoglu, "Thwarting all logic locking attacks: Dishonest oracle with truly random logic locking," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 9, pp. 1740–1753, Sep. 2021.

[40] D. Wang, Q. Gu, X. Huang, and P. Wang, "Understanding human-chosen pins: characteristics, distribution and security," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 372–385.

[41] D. Wang, Z. Zhang, P. Wang, J. Yan, and X. Huang, "Targeted online password guessing: An underestimated threat," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1242–1254.