CAS-Lock: A Security-Corruptibility Trade-off Resilient Logic Locking Scheme

Bicky Shakya[1*], Xiaolin Xu[2*], Mark Tehranipoor[1] and Domenic Forte[1]

[1] ECE Department, University of Florida
bshakya@ufl.edu,{tehranipoor,dforte}@ece.ufl.edu
[2] ECE Department, University of Illinois at Chicago
xiaolin8@uic.edu

Abstract. Logic locking has recently been proposed as a solution for protecting gate-level semiconductor intellectual property (IP). However, numerous attacks have been mounted on this technique, which either compromise the locking key or restore the original circuit functionality. SAT attacks leverage golden IC information to rule out all incorrect key classes, while bypass and removal attacks exploit the limited output corruptibility and/or structural traces of SAT-resistant locking schemes. In this paper, we propose a new lightweight locking technique: CAS-Lock (cascaded locking) which nullifies both SAT and bypass attacks, while simultaneously maintaining non-trivial output corruptibility. This property of CAS-Lock is in stark contrast to the well-accepted notion that there is an inherent trade-off between output corruptibility and SAT resistance. We theoretically and experimentally validate the SAT resistance of CAS-Lock, and show that it reduces the attack to brute-force, regardless of its construction. Further, we evaluate its resistance to recently proposed approximate SAT attacks (i.e., *AppSAT*). We also propose a modified version of CAS-Lock (mirrored CAS-Lock or M-CAS) to protect against removal attacks. M-CAS allows a trade-off evaluation between removal attack and SAT attack resiliency, while incurring minimal area overhead. We also show how M-CAS parameters such as the implemented Boolean function and selected key can be tuned by the designer so that a desired level of protection against all known attacks can be achieved.

Keywords: Logic locking; SAT attack

1 Introduction

Globalization of the semiconductor industry has led to the outsourcing of integrated circuit (IC) fabrication to untrusted, off-shore foundries. As a result, semiconductor companies as well as government agencies are now facing threats of IP piracy, counterfeiting, and overproduction [1]. Therefore, new techniques are required for combating these issues of untrusted foundries. Towards this end, logic locking has emerged as a promising solution. A majority of logic locking schemes insert extra *key-gates* into the netlist of the circuit design. The locked circuit works correctly only when the correct key is provided. However, recent work has shown that most of these locking techniques are vulnerable to *Boolean satisfiability (SAT) based attacks* [2]. In SAT attack, a set of *distinguishing input patterns (DIPs)* are collected from the locked circuit to rule out incorrect keys that do not satisfy the DIPs and the corresponding known-good responses from an unlocked IC. In order to mitigate SAT attacks, several SAT-resistant countermeasures have been proposed [3] [4], which aim to limit the ability of the attack to rule out wrong keys, given the golden

*These authors contributed equally to this work.

observations. However, they have been proven to be vulnerable to bypass attacks [5], which exploit the low corruptibility of these locking schemes. In the bypass attack, functionality of the locked circuit can be fully restored with a linear-sized bypass circuitry. This avoids the difficulty of recovering the correct key and allows the circuit to operate in a functionally correct manner *even in the presence of an incorrect key.*

To mitigate the threats from bypass attack and at the same time, ensure robustness against SAT attack, we propose a new logic locking technique: CAS-Lock (Cascaded Locking). In CAS-Lock, a logic block comprising of a cascade of key controlled AND/OR gates is stitched into the original circuit. The block exponentially increases the complexity of SAT attacks while simultaneously allowing the locked design to maintain non-trivial output corruptibility for defeating bypass attacks. Note that this property contradicts the results from recent literature, which show an unavoidable trade-off between output corruptibility and SAT resistance. Specifically, we show that the CAS-Lock scheme is the only locking technique proposed so far that can ensure SAT resistance with non-trivial output corruptibility, and can remain secure under a black-box attack model, where the attacker aims to recover the key using input-output observations. Our main contributions in this paper can be summarized as follows:

- We adopt the merits of two SAT-resistant techniques: SARLock [3] and Anti-SAT [4], and propose a more secure countermeasure – CAS-Lock – which is simultaneously resistant against SAT and bypass attacks.
- We provide a proof to show that breaking CAS-Lock with SAT attack requires, *at a minimum*, brute force through the entire input space of the circuit. We also show that its non-trivial and tunable output corruptibility leads to high overheads for bypass attack and incomplete bypass pattern extraction, as well as resistance against approximate SAT (AppSAT) attacks. The newly proposed countermeasure is also lightweight, as its overhead is only dependent on the number of inputs used.
- We also propose an extension to CAS-Lock (termed Mirrored-CAS or 'M-CAS') to protect against removal attacks from white-box adversaries such as untrusted foundries. We show that some trade-offs exist in this regard: strong resiliency to removal (through increased output corruptibility) leads to reduced SAT resistance (and vice-versa). We compare M-CAS to a recently proposed stripped functionality logic locking (SFLL) technique [6] and show that it achieves similar SAT and bypass resistance at reduced area overheads.

The rest of the paper is organized as follows. Section 2 reviews the background on logic locking, attacks on logic locking, and several recent countermeasures. Section 3 introduces CAS-Lock and its security properties. It also provides a series of proofs to show the SAT attack resiliency against CAS-Lock. Section 4 describes a modified version of CAS-Lock (M-CAS) that is required for preventing removal attacks. Section 5 provides a comparison of CAS-Lock and M-CAS with other pre-existing logic locking techniques. Finally, Section 6 concludes the paper.

2 Background and Related Work

Logic locking (also referred to as logic encryption) locks a logic circuit by adding key-controlled gates, and correct functionality of the circuit is ensured only when the correct key is applied. An incorrect key leads to corrupted outputs, thereby preventing unauthorized parties from making use of the locked circuit. In the case of an untrusted foundry, the locking prevents them from engaging in overproduction or IC piracy. Note that logic locking usually assumes that there is one master key per design, and the security of the design is dependent on this single key. This is because there is only one mask set per

design, from which many chips can be produced. However, there may be unique per-chip keys related to public key protocols, that are used in conjunction with the master key to unlock the design [7]. In this paper, we are mainly concerned with the master key that is used to implement the gate-level locking mechanism.

In recent years, several techniques have been proposed to perform logic locking, based on the position and impact of the inserted key gate logic; this includes techniques such as random insertion [8] and fault analysis-based key gate insertion [9]. Unfortunately, all these approaches have been shown to be vulnerable to SAT attacks [2].

2.1 SAT Attacks on Logic Locking

In SAT attacks [2], the threat model is as follows: an attacker is assumed to have access to: 1) *A locked netlist*: This can be either obtained from a malicious foundry or through reverse-engineering a chip from the open market [10]. The netlist can be simulated to derive the outputs for given inputs. 2) *Unlocked IC*: An unlocked 'golden' IC can be purchased from the open market or obtained through a malicious insider in the design house. Such a chip can be used by the attacker to check whether the output for a given key from the locked netlist is correct, i.e., he/she can perform chip-level functional/structural tests to obtain golden responses. The goal of the attacker is to find the correct key by inquiring the least number of input patterns from the unlocked IC. Note that only combinational circuits (or sequential circuits with scan capabilities) are considered in such attacks [2].

Using the same threat model, test-based attacks have been developed, which use automatic test pattern generation (ATPG) techniques [11] to generate a set of inputs that can propagate the correct key to observable outputs in the circuit. The problem of finding the correct key is modeled after the problem of generating patterns to detect stuck-at faults in a circuit [12]. In SAT-based attacks, such propagations are not required. Instead, the attacker constructs a 'miter circuit' with two copies of the locked netlist, which are loaded with two different wrong keys, respectively. The miter helps in finding a set of *distinguishing input patterns* (DIPs) for which the two circuits produce different outputs; since the outputs are different, it is assured that at least one of the keys chosen for the miter are wrong. Since the unlocked IC is available to the attacker, she can then apply this DIP to the unlocked IC and decide which of the keys is incorrect. Further, the known good input-output pair is also added as an additional constraint to the key formulation. The algorithm then iteratively uses these DIPs (and the added constraints) to guide a SAT solver to a correct key value, by ruling out all incorrect key classes. The algorithm terminates when no more DIPs can be found and as a result, the remaining key(s) is guaranteed to be the functionally correct key(s). The results in [2] show that the algorithm converges in a short time, when applied on a variety of logic locked circuits.

2.2 SAT-Resistant Logic Locking

Various SAT-resistant techniques have been recently developed, most notably SARlock [3] and Anti-SAT [4], to counter SAT attacks. Both of these techniques attach additional logic to the circuit in order to reduce the ability of the DIPs to rule out wrong keys. In other words, the attack is only able to rule out a negligible amount of the entire key space in a single attack iteration, forcing it to take an exponential time to find the correct key.

2.2.1 SARLock

SARLock ensures that *at most* one incorrect key is ruled out by each DIP [3]. This is realized with a comparator circuit that inverts the circuit output for only one input pattern for a wrong key. Thus, the SAT attack algorithm is forced to observe at least 2^N DIPs (where N is the number of inputs used for the SARLock logic) to rule out all incorrect key

classes. While SARLock renders SAT attack ineffective, it cannot protect against attacks exploiting gate-level structural traces. In *removal attack*, an attacker can analyze the netlist and then identify and remove the SARLock gates from the design. The removal attack is feasiblee since the SARLock logic has an extremely low signal probability on its output, which makes it distinct from other gates in the netlist. To counter such vulnerabilities, the authors in [3] proposed a hybrid logic-locking mechanism: SARLock + strong logic locking (SLL)[12]. This hybrid technique combines SARLock with regular logic locking (i.e., insertion of XOR/XNOR/MUX key-gates into the netlist), and also combines the two keys (SARLock key and SLL key) using permutations.

2.2.2 Anti-SAT

Anti-SAT is another locking technique proposed to counter SAT attacks [4][13]. The Anti-SAT block is illustrated in Figure 1, which is composed of the logic blocks $B_1 = g_{l1}(X, K^{l1})$ and $B_2 = \overline{g_{l2}(X, K^{l2})}$. The blocks share a common input X but are locked by two different keys K^{l1} and K^{l2}. Circuit integration is done by stitching the Anti-SAT output Y to a high observability net in the design (e.g., the primary output). The two blocks g_{l1} and $\overline{g_{l2}}$ are designed to be complementary in nature. They can also be denoted by g and $\bar{g}$. The output signal Y is generated by the logical AND of B_1 and B_2. Similar to SARLock, a wrong key applied on Anti-SAT will enable $Y = 1$ for some input pattern(s), and flip the correct outputs. Assuming the Boolean function g has N inputs, the number of input patterns that make g evaluate to "1" is denoted as p. The authors in [4] prove that the ability of the SAT attack to obtain the correct Anti-SAT key is greatly limited if p is sufficiently close to 1 (or $2^N - 1$); this choice of Anti-SAT logic, along with choosing the primary inputs as inputs to the Anti-SAT block, is termed as 'secure integration'. An Anti-SAT block satisfying $p = 1$ forces the conventional SAT attack to enumerate the largest number of possible input patterns to reveal the correct ones. They also note that natural candidates for g and $\bar{g}$ that satisfy $p = 1$ can be AND and NAND, respectively. 'Random integration' is also proposed, where random internal signals of the circuit (instead of the primary inputs) are used as inputs to the Anti-SAT block. The usage of 'Random integration' offers increased output corruptibility, albeit at reduced SAT resistance, as not all input pattern combinations are possible at the input of the Anti-SAT block.

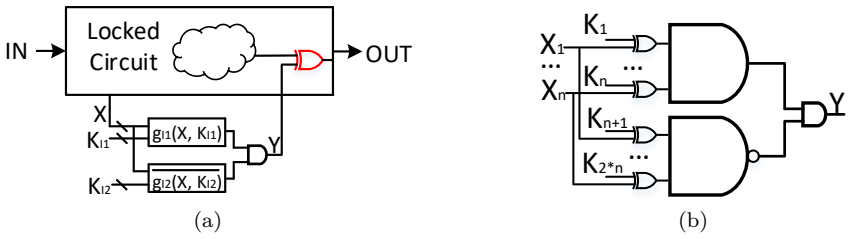


Figure 1: (a) Integration of Anti-SAT into a circuit. (b) Anti-SAT implemented with AND and NAND gates.

2.2.3 Removal Attack on Anti-SAT

Although the Anti-SAT block can be integrated in the whole netlist, it has been shown that an attacker can still identify the flip signal Y generated by the Anti-SAT block. This is accomplished by analyzing the signal probability skew (SPS) of the g and $\bar{g}$ blocks in the circuit [14]. In an SPS-based attack, the attacker computes the static probabilities (P) and skew ($s = P - 0.5$) of all the gates in the design. Due to the complementary construction between g and $\bar{g}$, the Y signal in the Anti-SAT block demonstrates the highest

absolute difference in signal skew (ADS), i.e., $ADS(Y) = |s(g) - s(\bar{g})|$, of all the gates in the design. As a result, the attacker could identify the AND gate with the Y signal and remove all the gates in the transitive-fanin cone of this net. This would effectively remove the entire Anti-SAT logic from the design. Additionally, the attacker could also set the flip signal of the Anti-SAT block to 0 and then apply the conventional SAT attack if there are additional key gates (e.g., XOR/XNOR) in the design.

2.2.4 AppSAT

A modified version of the original SAT attack (*'AppSAT'*) that approximately de-obfuscates a locked circuit was proposed in [15]. The motivation behind AppSAT is that hybrid obfuscation schemes (e.g., SARLock + SLL) cause the original SAT attack algorithm to get stuck in resolving the point function [1] (e.g., AntiSAT/SARLock). This leads to an exponential number of iterations for the attack to rule out all incorrect keys. AppSAT, on the other hand, forces the original SAT algorithm to terminate early, when the error rate (i.e., number of patterns that are incorrect among a sampling of input-output patterns, calculated periodically between iterations) settles below a given threshold. Further, multiple distinguishing input patterns and their correct outputs are added as constraints in a single iteration. Therefore, given a sufficient number of iterations/DIPs, AppSAT is able to resolve the correct XOR/XNOR keys while not being impeded by the point function. Therefore, the error rate of the key resolved by AppSAT is claimed to be approximately 2^{-n}.

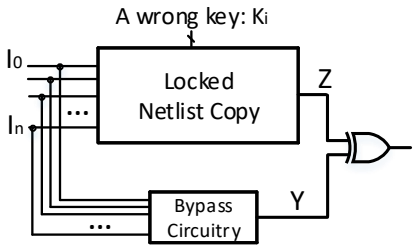
2.2.5 Bypass Attack

Bypass attack was proposed to exploit the low corruptibility of SAT resistant logic locking schemes such as SAR-Lock and Anti-SAT [5]. In this technique, a miter is formed between two copies of the circuit locked with two different wrong keys, in the same way as SAT attacks. After this, the set of input patterns whose outputs disagree with each other (i.e., distinguishing input pattern) are collected. These patterns are used to create a bypass circuitry, which is then stitched back into the locked circuit, as shown in Figure 2. As these limited set of input patterns are the only ones on which the circuit is corrupted, the bypass circuitry corrects these errors on the original circuit and restores the circuit's functionality. The effectiveness of bypass attack is mainly dependent on the ability of the miter circuit to find the DIPs, as well as the corruptibility of the SAT resistant scheme. Lower corruptibility implies smaller bypass attack overhead, whereas high corruptibility forces the attacker to create a large bypass circuit, which might not be feasible. For example, there exists only one DIP for every wrong key in 'secure integration' Anti-SAT, leading to a bypass circuit that only has to correct for one wrong input pattern. On the other hand, 'random integration' Anti-SAT ensures a large number of DIPs per wrong key but renders drastically reduced SAT resistance.

2.2.6 Stripped Functionality Logic Locking (SFLL)

Stripped functionality logic locking (SFLL) is an approach that has been proposed to combat SAT, bypass, and removal attacks [6]. In this approach, select logic cones of the original design are modified, such that they are corrupted on a pre-defined set of input patterns. A logic block is then stitched into the design (in a fashion similar to Anti-SAT or SARLock), which corrects the errors in the original design once the correct key is provided. The assumption is that even if the logic block that provides SAT resistance is removed, the design is still non-functional. This is because the block is also required to correct the injected errors in the design. A suggested choice for the logic block is a

[1] A point function is a function which evaluates to 1 at a particular input, and 0 elsewhere



(a)

	I_0	I_1	I_2	I_3	Z	$Z(K_i)$	$Z(K_j)$	...	...
	0	0	0	0	0	0	0	...	...
I_a	0	0	0	1	0	1	0	...	...
I_b	0	0	1	0	1	1	0	...	...
	0	0	1	1	0	0	0	...	...
	:	:	:	:	:	:	:	:	:
	:	:	:	:	:	:	:	:	:

(b)

Figure 2: (a) shows that for a locked netlist, a bypass circuit can be inserted to detect the DIP for the wrong key K_i. (b) shows an example truth table for finding the two DIPs with two wrong keys K_i and K_j for SARLock and Anti-SAT (Secure Integration).

hamming distance(HD)-based comparator, which computes the HD between the input and the key and corrupts/corrects the circuit output if the HD is equal to a publicly known parameter h. An overhead improvement to SFLL, that uses fault injection and failing pattern detection to modify the original circuit, is proposed in [16]. A functional attack on SFLL ('FALL') has been recently proposed in [17], where the logic that corrupts the original design (which is also a HD comparator) is identified. After identification, the hard-coded key value in the netlist, which drives the comparator and is supposed to be hidden after re-synthesis, is extracted. The key is decoded by specifically exploiting the logical properties of the HD comparator block, such as non-overlapping errors between two input pairs. Results showed that as much as 81% of SFLL locked combinational benchmark circuits could be decoded using FALL attack.

2.3 Other Countermeasures and Attacks

Yasin *et al.* have proposed to use one-way functions (such as AES) for combating SAT attacks [18]. However, as the cipher and the original circuit are functionally and structurally independent, it becomes trivial for the attacker to identify and circumvent the AES block. To prevent similar vulnerabilities, Xie *et al.* proposed structural obfuscation techniques to secure the Anti-SAT block against removal [4]. This is done by adding MUX/XOR key gates into the design, so that the original circuit and the Anti-SAT block are entangled with each other and structural traces are removed. Some approaches have also been proposed to restrict or prevent scan access to a circuit, so that SAT attacks can be rendered infeasible in the first place [19]. However, these approaches follow a weaker security notion i.e., they consider a much weaker adversary who cannot access the scan chain. Further, there is no formal treatment of the security of such approaches. Logic locking based on re-use of DFT test points has also been proposed [20]. Unfortunately, no concrete security evaluation (in terms of SAT and/or structural attacks) has been provided for such approaches. The work in [21] proposes a locking mechanism based on the insertion of routing blocks that are inherently hard for SAT solvers, similar to [18]. However, the technique makes use of lookup tables (LUTs), which are not compatible with ASIC designs and are high in overhead. All locking approaches proposed so far also assume the secrecy of the key, and do not take into account attacks such as invasive/direct probing of the key (e.g., as they are being loaded from memory, or directly from fuses). Such attacks require their own set of countermeasures, and are beyond the scope of this paper.

3 CAS-Lock

3.1 Requirements for Attack-Resilient Logic Locking

Based on the numerous attacks discussed above, it can be concluded that any single countermeasure against a certain attack is not secure enough. Therefore, to comprehensively protect the locked circuit, all the aforementioned attacks need to be considered in unison to formulate a new secure logic locking scheme.

In [4], Xie *et al.* thoroughly evaluated the security of the Anti-SAT block when changing the p value (output-one count) of the $g/\bar{g}$ function from 1 to 2^N, where N is the number of inputs used in the Anti-SAT block. It was found that by changing more AND gates of the $g/\bar{g}$ function in the AND-tree structure to OR gates (or conversely, changing more of the OR gates in the OR-tree structure to AND gates) decreased the resiliency to SAT attacks. However, we've found that for certain structures of the AND/NAND tree, it is still possible to maximize SAT attack resistance (i.e., reduce it to brute force) even when $p \neq 1$ or $p \neq 2^N - 1$.

Before discussing these structures, we outline some requirements for an ideal Anti-SAT block.

(1) **Lightweight**: The overhead (i.e., the number of gates in Anti-SAT) should be a linear function of the number of circuit inputs [4]. Otherwise, if the overhead increases exponentially with the number of inputs used or is large, it would not be a feasible solution for industry to adopt.

(2) **Strong resilience against bypass attack**: There are two scenarios that can limit the successful application of bypass attack: (a) The hardware overhead of the bypass circuitry is prohibitively high; or (b) The number of output errors across all possible input patterns for a wrong key is random, undetectable and hard to calculate as N scales. For example, the baseline Anti-SAT (with $p = 1$) was *always* incorrect on only one unique input pattern for a given wrong key, which made it easily defeated by bypass attack [5].

(3) **Strong resilience against SAT-based attacks**: The attacker should be forced to iterate through a very large number of input patterns to discover the correct key(s) (i.e., near brute force[2]).

(4) **Resistance to removal attacks**: It should be difficult for the attacker to use structural information to isolate and remove the Anti-SAT block from the locked netlist (e.g., by using the signal probability skew attack proposed in [14]).

As was shown in [5], there exists a trade-off between SAT and bypass attack resistance. This is because a countermeasure is most robust against SAT attack if the number of wrong keys that can be ruled out by each distinguishing input pattern is minimized. Conversely, such low corruptibility facilitates bypass attack. To mitigate both these attacks with one single countermeasure, we propose a more general constraint and rewrite requirements (2) and (3) as: ***For any input pattern X_i, there exists at least one wrong key WK^i that can only be ruled out by this input pattern, and the number of corrupted outputs for a given wrong key is neither constant nor unique across all possible keys.*** The first half of the new constraint implies that there should exist unique wrong key(s) for any input pattern. Therefore, to rule out all possible wrong keys, the attacker is forced to iterate through all possible input patterns. The second half of the constraint relates to bypass attack difficulty. If the circuit is incorrect on a random (and large) number

[2]If the no. of inputs used is N, the number of input-output observations needed to prune all wrong keys is 2^N.

of input patterns for any given wrong key, the bypass overhead can be unpredictable (and large). Further, if the output errors are not unique across different wrong keys, it becomes hard to detect and correct all of them.

3.2 CAS-Lock Analysis

To fulfill these requirements, we propose a new countermeasure called CAS-Lock, which is shown in Figure 3. It can be seen that CAS-Lock adopts a structure similar to Anti-SAT, where the outputs of two complementary Boolean functions g_{cas} and $\bar{g}_{cas}$ are ANDed together to produce the output (Y). The difference lies in the structure of the logic gates implementing g and $\bar{g}$, where instead of a tree structure, we adopt a daisy-chained or cascaded structure for the AND/OR gates.

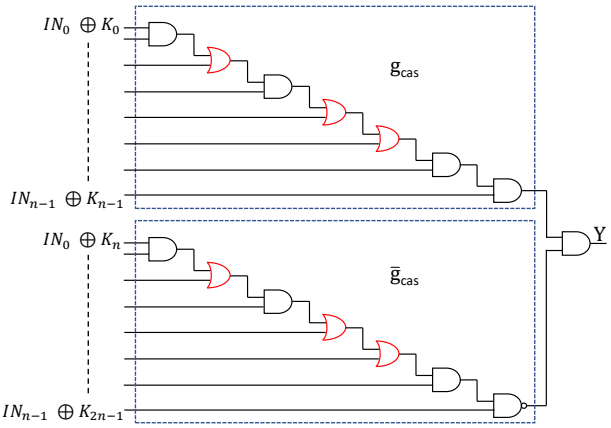


Figure 3: One possible instantiation of CAS-Lock (Note that g and $\bar{g}$ are still symmetric and complementary.)

It can be noted that for input length N, the gate count of CAS-Lock is the same as that of Anti-SAT i.e., $\approx 2 \cdot (N-1)$. This satisfies **Requirement (1)** listed above (i.e., the overhead increases linearly with input size). **Requirement (2)**, which relates to bypass attack resistance, is ensured as the number of *unique wrong keys* is controllable by changing the location and number of OR/AND gates in g_{cas} and $\bar{g}_{cas}$. Further, the number and location of incorrect input/output patterns for a given wrong key can only be resolved by brute force (i.e., by evaluating the CAS-Lock structure for all possible patterns or by iterative SAT solving). This becomes prohibitive for large N. **Requirement (3)** is fulfilled as CAS-Lock ensures the existence of *at least one unique* wrong key for every possible input pattern. A proof of this property, along with some precursor lemmas, is given below. For the proof, we show the existence of 2^N wrong keys, each of which can only be eliminated by a unique DIP. More importantly, we show that regardless of the output-one count (p) of g_{cas}, such keys always exist in CAS-Lock (Lemma 1). This is due to its logical behavior, whereby a certain number of input patterns cause g_{cas} to output logic 1 and $\bar{g}_{cas}$ to output logic 0 (Lemma 2). Since there are an odd number of such patterns (shown by Lemma 3), g_{cas} and $\bar{g}_{cas}$ both evaluate to 1 (and thus, produce an incorrect output) only on one unique input pattern when one of the 2^N wrong keys is applied. Note that requirement (4), i.e., resistance to removal attack, will be considered in Section 3.5.

Following the notation in [4], we denote the n-bit inputs to the CAS-Lock component with $\mathbf{X}$ and the $2n$-bit key with $\mathbf{K}$. $|\mathbf{X}|$ refers to the size of the inputs used in CAS-Lock, and $|\mathbf{K}|$ refers to the key size of $2n$. Similar to [13], $\mathbf{X}$ is assumed to be directly connected

to the primary inputs **IN**[3]. We also define $\mathbf{L} = \mathbf{X} \oplus \mathbf{K}$, i.e., XOR or XNOR of the inputs with the key, as shown in Figure 1.

Lemma 1. *Given the countermeasure built with Boolean functions g_{cas} and $\bar{g}_{cas}$, there exists 2^N wrong keys, each of which can only be ruled out by a unique input pattern X_i. Thus, to rule out all the wrong keys, the attacker has to iterate through all possible 2^N input patterns (i.e., brute force through the entire input space).*

Proof. Lemma 1 is true if we can find *at least one unique* wrong key that can only be ruled out by a specific input pattern X_i.

Based on the design scheme of CAS-Lock, we denote a correct $2n$-bit key as $\mathbf{K}=< CK_i^1, CK_i^2 >$, where CK_i^1 and CK_i^2 are fed into Boolean functions g_{cas} and $\bar{g}_{cas}$, respectively. Then, for any input pattern X_i, we have:

$$Y_i = g_{cas}(X_i \oplus CK_i^1) \wedge \bar{g}_{cas}(X_i \oplus CK_i^2) = 0 \tag{1}$$

Therefore, for the correct key, the output Y_i of the AntiSAT block is ***always zero*** and the circuit output is never inverted.

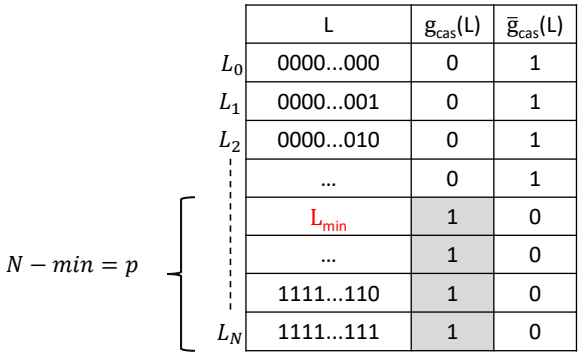


Figure 4: A truth table for g_{cas} and $\bar{g}_{cas}$, where L_{min} stands for the smallest input pattern that makes the Boolean function g_{cas} equal to 1.

As shown in Figure 4, there always exists the smallest input pattern L_{min} where, for $\forall L_i < L_{min}$, we have $g_{cas}(L_i) = 0$[4]. In other words, if we count the input patterns incrementally from $0000\ldots000$ to $1111\ldots111$ and observe the corresponding output of function g_{cas}, L_{min} is the smallest input pattern that makes $g_{cas} = 1$. For example, if g_{cas} is built with all AND gates, then $L_{min} = 1111\ldots111$. Thus, we now have the expression in Eqn. 2, where $(L_{min} - 1) \oplus 1 = L_{min}$ [5]. These two scenarios are highlighted in green in Figure 5(a), where the output Y_i is always zero.

$$\begin{aligned} g_{cas}(L_{min}) &= \bar{g}_{cas}(L_{min} - 1) = 1 \\ \bar{g}_{cas}(L_{min}) &= g_{cas}(L_{min} - 1) = 0 \end{aligned} \tag{2}$$

Now, let us assume a wrong key $WK_j =< WK_j^1, WK_j^2 >$ that satisfies Eqn. 3.

$$\begin{aligned} WK_j^1 \oplus CK_i^1 &= 000\ldots00 \\ WK_j^2 \oplus CK_i^2 &= 000\ldots01 \end{aligned} \tag{3}$$

[3] **X** could also be connected to random internal wires instead of **IN**. However, this reduces SAT resistance, as shown in [4] and as such, is not considered.

[4] This is proven in Lemma 2

[5] This is proven in Lemma 3

(a)

	X_i	$X_i \oplus CK_i^1$	$X_i \oplus CK_i^2$	$g_{cas}(X_i \oplus CK_i^1)$	$\bar{g}_{cas}(X_i \oplus CK_i^2)$	Y_i
	0000...000	0000...000	0000...000	0	1	0
	0000...001	0000...001	0000...001	0	1	0
	...	...	...	0	1	0
$L_{min}-1$	0	0	0	0	1	0
L_{min}	1	1	1	1	0	0
	...	...	...	1	0	0
	1111...110	1111...110	1111...110	1	0	0
	1111...111	1111...111	1111...111	1	0	0

$X_i \oplus WK_j^2 = X_i \oplus CK_i^2 \oplus 000 \ldots 001$

(b)

	X_i	$X_i \oplus WK_j^1$	$X_i \oplus WK_j^2$	$g_{cas}(X_i \oplus WK_j^1)$	$\bar{g}_{cas}(X_i \oplus WK_j^2)$	Y_i
	0000...000	0000...000	0000...001	0	1	0
	0000...001	0000...001	0000...000	0	1	0
	...	...	...	0	1	0
$L_{min}-1$	0	0	1	0	0	0
L_{min}	1	1	0	1	1	1
	...	...	...	1	0	0
	1111...110	1111...110	1111...111	1	0	0
	1111...111	1111...111	1111...110	1	0	0

Figure 5: Example Truth Table for g_{cas} and $\bar{g}_{cas}$, where the wrong key WK_j can only be ruled out by applying the input pattern L_{min}.

Such a wrong key causes the outputs of $\bar{g}_{cas}(L_{min}-1)$ and $\bar{g}_{cas}(L_{min})$ (in fact, any two consecutive patterns) to be swapped, as shown in Figure 5(b). Assuming that the least significant bit (LSB) of L_{min} is 1[6], we now have the scenario highlighted in red in Figure 5(b), at the input patterns $L_{min}-1$ and L_{min}, which can be written as:

$$\begin{aligned} g_{cas}(X_i \oplus WK_j^1) &= g_{cas}(L_{min}) = 1 \\ \bar{g}_{cas}(X_i \oplus WK_j^2) &= \bar{g}_{cas}(L_{min}) = 1 \end{aligned} \tag{4}$$

In other words, the output $Y_i = g_{cas}(X_i \oplus WK_j^1) \wedge \bar{g}_{cas}(X_i \oplus WK_j^2)$ for input X_i is 1 when the wrong key $WK_j =< WK_j^1, WK_j^2 >$ is applied. Moreover, for the wrong key $WK_j =< WK_j^1, WK_j^2 >$ that satisfies Equation 3, the Y output of CAS-Lock will be 0 for all other input patterns that are not equal to X_i. In other words, **the corruptibility of the circuit given WK_j is only 1, and this incorrect key can only be ruled out by applying input pattern X_i and no other input pattern**. Since there are 2^n correct keys in CAS-Lock, there are 2^n wrong keys with this property. This is because such wrong keys are obtained by flipping the least significant bit (LSB) of the 2^n correct keys, as shown in Equation 3. Therefore, to rule out all 2^n such incorrect keys that satisfy Equation 3 from the total key space of 2^{2n}, the attacker has to apply *at least* 2^n input patterns (i.e., brute force through the entire input space of the circuit). □

Lemma 2. *For a Boolean function g_{cas} that is built with the cascaded tree structure, there exists the smallest input pattern L_{min}, for which $g_{cas}(L_{min}) = 1$. Then $\forall L_i$, if $L_i \geq L_{min}$, we have $g_{cas}(L_i) = 1$.*

[6]In Lemma 3, we prove that the input pattern L_{min} is an odd number, which implies that its LSB must always be 1.

Proof. Lemma 2 can be proven if we can show that $\forall L_i(L_i < 1111\ldots111)$ that makes $g_{cas}(L_i) = 1$, we have $g_{cas}(L_{i+1}) = 1$, where $L_{i+1} = L_i + 1$.

We start by introducing a generic CAS-Lock structure shown in Figure 6, where the gate at index k (denoted by $GATE_k$) is an AND gate. The logic cone before the AND gate is denoted by g^{up}_{cas}, and the logic cone after it is denoted by g^{down}_{cas}. Therefore, we would have:

$$g_{cas} = g^{down}_{cas}(g^{up}_{cas}(l^0_i, \ldots, l^{k-1}_i) \wedge l^k_i, l^{k+1}_i, \ldots, l^{n-1}_i)$$

Note that g^{up}_{cas} and g^{down}_{cas} can be any cascaded combination of AND's and OR's.

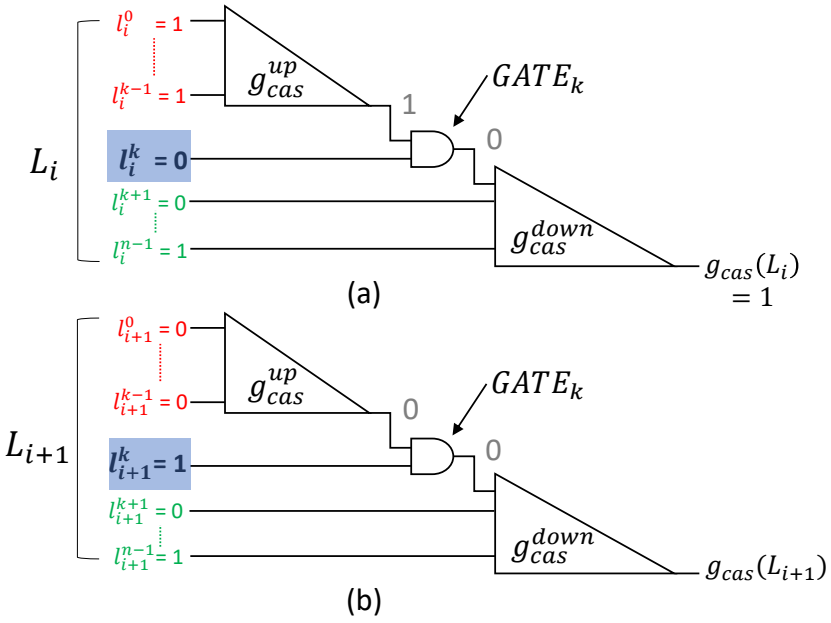


Figure 6: Input vectors L_i and L_{i+1} applied to g_{cas}

In Figure 6(a), we can see that the input pattern L_i is applied on g_{cas} while in Figure 6(b), the pattern L_{i+1} is applied on the same g_{cas}. For L_i, bit l^0_i is the least significant bit (LSB) and l^{n-1}_i is the most significant bit (MSB). Similarly, for L_{i+1}, bit l^0_{i+1} is the LSB and l^{n-1}_{i+1} is the MSB. For both input vectors L_i and L_{i+1}, let us assume that the bits at index k are the left-most bits (i.e., towards the MSB) that are not equal to each other (i.e., $l^k_i \neq l^k_{i+1}$). Figure 7 shows an example of this, with two vectors for L_i, L_{i+1} and $k = 2$. At $k = 2$, we have $l^k_i \neq l^k_{i+1}$ as:

- $l^k_i = 0$
- $l^k_{i+1} = 1$

k

Index	6	5	4	3	2	1	0
L_i	1	0	0	1	0	1	1
L_{i+1}	1	0	0	1	1	0	0

Figure 7: Example of L_i and L_{i+1} input vectors

Since $L_{i+1} = L_i + 1$ (i.e., they are consecutive input vectors), there are now two implications.

1. The bits after index k i.e., $\{l_i^{n-1}$ ... $l_i^{k+1}\}$ and $\{l_{i+1}^{n-1}$ ... $l_{i+1}^{k+1}\}$ will always be equal to each other. For example, in Figure 7, the vectors marked in green (from index 6 to 3) are the same in L_i as well as L_{i+1}.

2. For the bits that are at indices $< k$, we can see that they are always going to be the inverse of each other. In Figure 7, the bits marked in red at indices 1 and 0 are flipped across L_i and L_{i+1}.

Note that this trend is true for any two consecutive input vectors L_i and L_{i+1}, and also for any value of k.

Now, in Figure 6 (a) and (b), we can see that $l_i^k = 0$ and $l_{i+1}^k = 1$, since k, as we defined above, is the left-most index (i.e., towards the MSB) on which the bits are different. Figure 6 (a) shows that the inputs to the logic cone g_{cas}^{up} are all 1's. This will cause the output of logic cone g_{cas}^{up} to be 1, regardless of how many AND's and OR's are there in g_{cas}^{up}. Therefore, the output of the AND gate in Figure 6 (a) (i.e., $GATE_k$) will be 0. Since the definition of $L_{i,min}$ (i.e., L_{min} at index i) requires $g_{cas}(L_i) = 1$, the input bits to g_{cas}^{down} will have to 'cancel out' the effect of the 0 output of $GATE_k$ so that $g_{cas}(L_i) = 1$. Thus, in some sense, the inputs to g_{cas}^{down} (l_i^k to l_i^{n-1}), along with the g_{cas}^{down} logic, serve as 'controlling values', forcing the output of the logic cone to 1.

For L_{i+1}, the opposite of L_i happens. In Figure 6 (b), we see that the input bits to g_{cas}^{up} produce an output of 0, after which the output of $GATE_k$ will also be 0. Since the Boolean logic in g_{cas}^{down} is the same for L_i and L_{i+1} and so are the inputs to g_{cas}^{down}, the output of $g_{cas}(L_{i+1})$ will also be forced to 1. From these observations, we can conclude that the output of $GATE_k$ is always the same for L_i and L_{i+1}. Further, since $g_{cas}(L_i) = 1$, the logic in g_{cas}^{down} will always force the output of g_{cas} to be 1 for both L_i and L_{i+1}. While we used the example of an AND gate for $GATE_k$, it is straight-forward to prove the same if $GATE_k$ is an OR gate. Further, the proof also implies that if $g_{cas}(L_i) = 1$ and thus $g_{cas}(L_i + 1) = 1$, $g_{cas}(L_i + 2) = 1$ and $g_{cas}(L_i + 3) = 1$ and so on (since the relationship holds for arbitrary i and can be shown by proof of induction). Hence, Lemma 2 is proven. □

Lemma 3. *The total number of input patterns that make the Boolean function g_{cas} equal to 1 (i.e., p) is an odd number.*

Proof. From Lemma 2, we saw that there always exists the smallest input pattern L_{min}, for which all input vectors greater than L_{min} cause the output of Boolean function g_{cas} to be 1. We also know that the LSB l_i^0 for an input pattern L_i is 0 (if L_i is an even number) or 1 (if L_i is odd). Therefore, to prove Lemma 3, we just need to show that the LSB of L_{min} is always 1. Alternatively, following proof by contradiction, Lemma 3 is also true if we can prove that the LSB of L_{min} can never be 0.

First, we begin the proof by assuming that $l_{min}^0 = 0$ (i.e., L_{min} is an even number). Therefore, the output of $GATE_0$ shown in Figure 8 will be 0 if $GATE_0$ = AND, and 1 if $GATE_0$ = OR and $l_i^1 = 1$. All of the possibilities for $GATE_0$ are shown in Figure 9 with $l_{min}^0 = 0$. In cases (a) (b) and (c), the output of $GATE_0$ is 0. Since $g_{cas}(L_{min}) = 1$ (by the definition of L_{min}), the gates and inputs in g_{cas}^{cone} must compensate for the 0 output from $GATE_0$ and make the output $g_{cas}(L_{min}) = 1$. Thus, the inputs l_{min}^2 to l_{min}^{n-1} and g_{cas}^{cone} control the output to 1, regardless of the output of $GATE_0$. In case (d), the 1 output from $GATE_0$ would contribute to making $g_{cas}(L_{min}) = 1$.

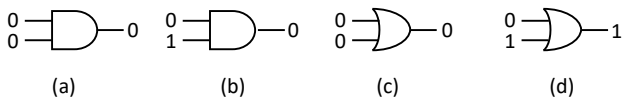


Figure 9: Four cases for $GATE_0$, the top bit is l^0_{min} and the bottom bit is l^1_{min}.

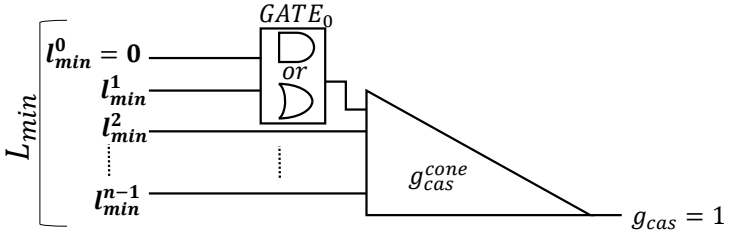


Figure 8: L_{min} with $l^0_{min} = 0$ applied to g_{cas}.

Since the definition of L_{min} requires it to be the *smallest* input pattern that can ensure $g_{cas} = 1$, the cases (b) and (d) are ruled out because they are not the smallest values that can generate the same outputs of $GATE_0$. For example, the input pattern $\{l^1_{min} l^0_{min}\} = 01$ is smaller than $\{l^1_{min} l^0_{min}\} = 10$ (the pattern in (b)), but can generate 0 as well in case (b). By excluding (b) and (d), we can see that if the $l^0_{min} = 0$, then:

1. The output of $GATE_0$ is always 0, regardless of whether $GATE_0$ is an AND gate (case (a)) or an OR gate (case (c)).

2. Both l^0_{min} and l^1_{min} must be 0 to ensure the definition of L_{min}, i.e., the smallest pattern that results in $g_{cas} = 1$.

Note that if $l^0_{min} = l^1_{min} = 0$ and the output of $GATE_0 = 0$, this output of $GATE_0$ would not contribute to make $g_{cas}(L_{min}) = 1$. Thus, there must exist OR gate(s) in g^{cone}_{cas}, and one or more of the bits from $l^2_{min}...l^{n-1}_{min}$ must be 1. This would result in L_{min} to be patterns such as 0110010...00 or 100000...00, where at least one of the bits at index > 2 would be 1, while $l^0_{min} = l^1_{min} = 0$. However, this presents a contradiction with the definition of L_{min} (i.e., it should be the *smallest* input pattern that causes $g_{cas} = 1$). This is because we can always find another assignment for L_{min} that is smaller than the L_{min} pattern we assumed above (where $l^0_{min} = l^1_{min} = 0$). For example, if assuming $L_{min} = 0110010..100$ with $l^0_{min} = 0$, we can find another value 0110010..011 smaller than this assumed L_{min}, which will also set $g_{cas} = 1$.

Thus, due to the contradiction we achieved in the definition of L_{min} when $l^0_{min} = 0$, it must be true that $l^0_{min} = 1$. □

3.3 Bypass Attack Analysis

The above proofs show why CAS-lock is resistant to SAT attacks. To fulfill requirement (2) for attack-resilient logic locking, we must also determine resistance to bypass attack. In order to perform a bypass attack, a miter circuit is first formed with two copies of the locked circuit[7]. We denote the output of the two locked circuit copies with Y_A and Y_B. The first copy is locked under wrong key K_A and the second under wrong key K_B. Primary inputs I are fed into both copies. The miter output Y_{miter} can be expressed as:

$$Y_{miter} = Y_A(I, K_A) \oplus Y_B(I, K_B) \tag{5}$$

[7]For simplicity, we consider a standalone CAS-Lock block as the locked circuit

The miter is then fed to a SAT solver (to solve for $Y_{miter} = 1$) and the disagreeing patterns I are collected to construct the bypass circuit.

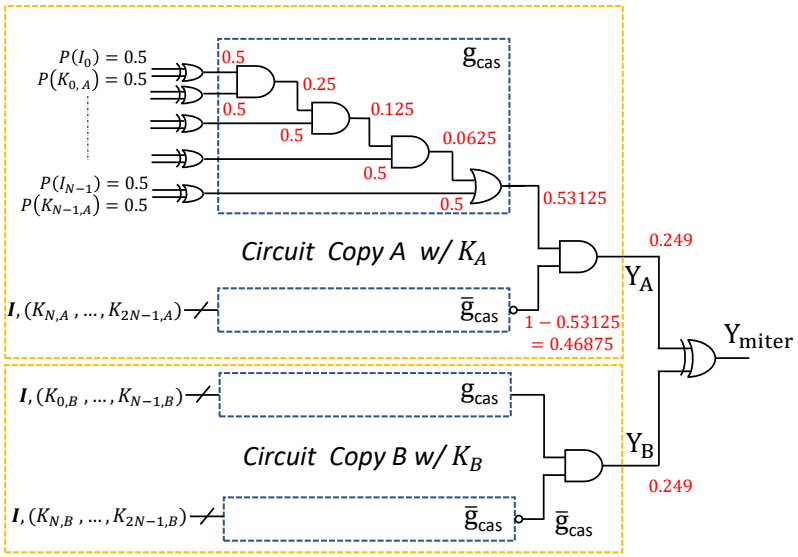


Figure 10: Miter circuit used for bypass attack and corresponding probabilities of logic-1 at various nets.

CAS-Lock is able to thwart bypass attack as the number of input patterns I to bypass can be prohibitively large for large p values (i.e., close to $2^N/2$) and a large input size $|I|$. The maximum corruptibility (at $p \approx 2^N/2$) is attained with a cascade of AND gates and one OR gate at the output of g_{cas}. The OR gate at the output ensures $g_{cas} = 1$ for almost half the entire input space. In addition to the high corruptibility, bypass attack is also thwarted when the miter in Equation 5 is unable to discover the input patterns I to bypass. This scenario arises when $Y_A(I, K_A) = 1$ and $Y_B(I, K_B) = 1$, leading to $Y_{miter} = 0$. In other words, both K_A and K_B produce the same wrong output but the miter can only detect input patterns I when $Y_A(I, K_A) \neq Y_B(I, K_B)$. Effective bypass resistance is ensured when large p values are used and, thus, a large number of such undetectable patterns are generated.

To see how p affects bypass resistance, we evaluate the probability of getting $Y_A(I, K_A) = 1$ and $Y_B(I, K_B) = 1$ as p is varied. As an example, we consider the miter circuit in Figure 10, where Y_A is the output of the CAS-Lock circuit locked under the key K_A, and Y_B is the output of CAS-Lock under key K_B. For this g_{cas} logic (where the last gate in the cascade is an OR gate), we get $p = 17$, which is equivalent to a probability of $\frac{17}{2^5} = 0.53125$. Consequently, this gives us a probability of $Y_A = 1$ and $Y_B = 1$ (i.e., the probability of being unable to detect a corrupted input pattern) as $0.249 * 0.249 = 0.062$. Similarly, for a g_{cas} logic composed of all ANDs (which gives us $p = 1$, or probability of $\frac{1}{2^5} = 0.03215$), the probability of $Y_A = 1$ and $Y_B = 1$ becomes $0.0303 * 0.0303 = 0.00092$. Thus, larger p values (till $\approx 2^N/2$) tend towards a higher probability of bypass attack failure.

This probability of bypass failure (P_{fail}) can also be generally expressed as:

$$\begin{aligned} P_{fail} &= P(Y_A = 1) \cdot P(Y_B = 1) \\ &= pr \cdot (1 - pr) \cdot pr \cdot (1 - pr) \\ &= pr^2 \cdot (1 - pr)^2 \end{aligned}$$

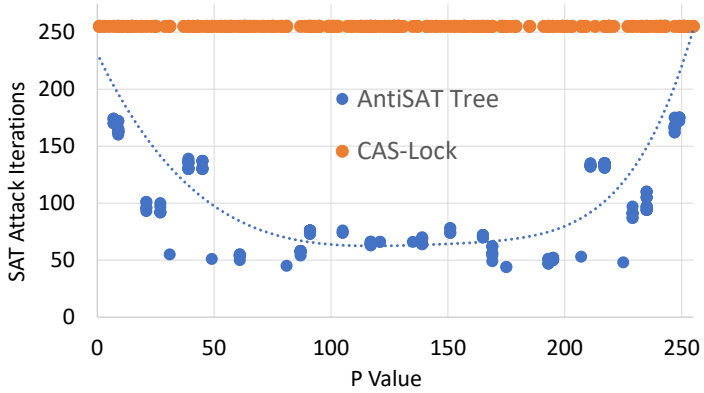


Figure 11: SAT Attack Time and Number of Iterations for CAS-Lock and Anti-SAT Structures with Varying Output-One Count

Here, the probability of getting logic 1 at the output of g_{cas} is expressed as $pr = \frac{p}{2^N}$. From the expression and the example above, we can see that as pr increases (towards ≈ 0.5), the probability of getting 1 at the outputs of both Y_A and Y_B increases, thus leading to bypass attack error.

3.4 Experimental Evaluation of CAS-Lock

3.4.1 Resistance Against SAT Attack

To experimentally evaluate resistance against SAT attacks, we used the tool provided in [2] on variations of CAS-Lock. The results from the experiment (Figure 11) show that regardless of p (output-one count of $g_{cas}/\bar{g}_{cas}$), the number of iterations required to break CAS-Lock is always $2^N - 1$. In these experiments, we used $N = 8, |K| = 2 * N = 16$ and varied p by changing between AND's and OR's in $g_{cas}/\bar{g}_{cas}$[8]. We can see that the number of SAT iterations required to break CAS-Lock is always $2^8 - 1 = 255$. In the same figure, we can also see that the number of iterations required to break AntiSAT varies with p, while CAS-Lock's SAT resistance is maximized regardless of p. Although we performed this analysis on standalone CAS-Lock structures, the SAT attack runtime is expected to be even longer when the block is stitched into a circuit, as the size of the overall CNF formula increases. Further, increasing N and K to larger values will make SAT attacks infeasible (e.g., when $N = 128/K = 256$, number of SAT iterations would be $2^{128} - 1 = 3.4e + 38$).

3.4.2 Resistance Against Bypass Attack

As mentioned before, CAS-Lock can ensure protection against bypass attack if (a) there are random (and often large) number of patterns to bypass for any given wrong key and (b) it is very likely that for any given wrong key, some erroneous patterns remain undetected. In order to verify these properties, we generated several variants of CAS-Lock and evaluated the number of times CAS-Lock evaluated to 1 for a sample of wrong keys. In these experiments, we set $N = 12$ and thus, $K = 2N = 24$. When $p = 1$ or $p = 2^N - 1$, CAS-Lock reduces to Anti-SAT. Therefore, any given wrong key Ki', there is one (and only one) input pattern which 'triggers' the block. In other words, the error count or number of input patterns for which the output is flipped, **denoted by** ϵ, is equal to 1 for all K' wrong keys across all input patterns. On the other hand, for CAS-Lock with $p = 1857$, we

[8]Remember that when $p = 1$ or $p = 2^8 - 1 = 255$, CAS-Lock reduces to Anti-SAT in secure integration mode with g as AND and g' as NAND (or vice-versa).

obtained $\epsilon_{K1'} = 1857$, $\epsilon_{K2'} = 191$, $\epsilon_{K3'} = 65$, $\epsilon_{K4'} = 63$ and so on. In bypass attack, any two random keys are chosen to form a miter and all resulting 'distinguishing patterns' are chosen for bypass. If $K1'$ and $K2'$ were chosen and all patterns for $K2'$ overlapped with $K1'$ (i.e., for these set of patterns, both $K1'$ and $K2'$ produce errors), the miter would only discover $1857 - 191 = 1666$ patterns for correction. Thus, the resulting bypassed circuit would be erroneous. Further, this would translate to $38,319$ gates for the bypass circuitry (assuming $N = 12, N_{DIP} = 1666$ and $N_{out} = 1$), when following the equation given in [5] for converting between number of DIPs and bypass circuitry gate count. Of course, re-synthesis would be able to reduce this overhead, but it would not be viable to do such 'retroactive correction of logical errors' for a large number of input patterns. This problem would also be substantially aggravated as N increases. For example, when $N = 16$ and $p = 34043$, we obtained $\epsilon_{K1'} = 31493, \epsilon_{K2'} = 1275$ and so on.

3.5 Analysis of Removal and AppSAT Attacks

Most of the attacks on logic obfuscation can be categorized into two types.

- **Black-box attack**: Attacks such as SAT or AppSAT rely on a black-box model of the original design. While the attacker does have possession of the netlist, he/she is able to attack the circuit without any analysis of the netlist/structure of the design or how the logic blocks have been integrated at the gate-level. Rather, the locked circuit is fed into the solver as a CNF formula, its input/output behavior is analyzed and correct key assignments are returned.
- **White-box attack**: Attacks such as removal specifically target the implementation of the locking technique at the netlist-level. Structural metrics such as signal probability skew and fan-in analysis are computed to identify the gates implementing the locking and subsequently remove them.

While CAS-Lock provides security against black-box attacks such as SAT and bypass, it also needs to be resilient against white-box attacks such as removal. Unfortunately, CAS-Lock, by itself, is prone to removal in the same way as Anti-SAT, especially when low or high p values are used [14] [22]. This is because an attacker can identify the block in the design netlist using signal probability skew (SPS) [14], as mentioned in Section 2.2.3. To prevent such attacks, it is necessary to hide CAS-Lock so that, to an attacker, the block becomes indistinguishable from other logic cones in the design and cannot be removed while keeping the original design intact. If the attacker is a reverse engineer in the supply chain, this hiding of the CAS-Lock logic block can be efficiently implemented using camouflaging [23][24][25]. The key idea would be to conceal the identity of the gates in the CAS-Lock block, so that signal probability/skew values or any structural metrics cannot be correctly computed by the attacker. Therefore, the AND/OR gates in the original design as well as CAS-Lock could be replaced by camo-cells. Due to this, the reverse engineer would be limited to a black-box attack, which CAS-Lock is resilient against.

While camouflaging coupled with CAS-Lock would ensure maximum resiliency against black-box attacks such as SAT and removal, it would not be effective in the case of an untrusted foundry. This is because foundry assistance is required in order to build the camo cells, due to which their identity is already known to the foundry. Therefore, the foundry would be able to mount white-box attacks such as removal. In order to protect the design against such adversaries, it is necessary to insert additional key gates (XOR, XNOR, MUX, AND, OR etc.) into the CAS-Lock block in a non-symmetric fashion [3] [4]. This would ensure that the SPS-based trait of the CAS-Lock block would be hidden (i.e., the net Y will not necessarily have the highest SPS difference value). As a result, an attacker will not be able to identify and isolate the CAS-Lock block, unless he or she knows the key values for the key gates. Note that the security achieved with such obfuscation

will depend on how many candidate nets can be isolated by the attacker. Without key gate insertion, there will always be a single candidate (the CAS-Lock output net). With obfuscation, there should be many such candidate nets (or none) in the design, which would make it hard for the attacker to perform CAS-Lock removal.

After performing random XOR/XNOR insertion into CAS-Lock (hereafter referred to as 'obfuscated CAS-Lock'), we performed a set of experiments to evaluate if there was any change in its resistance to SAT and AppSAT attacks. While SAT resistance remained the same ($2^N - 1$ iterations to resolve the correct key), we noticed that AppSAT (with the default parameters in [15]) was able to find the correct key assignments to the obfuscated CAS-Lock in a few iterations. In our experiments, we generated 100 random CAS-Lock structures (with $N = 16$), obfuscated with 32 random XOR key gates. In 50 of those structures, AppSAT was able to return the correct key. We also generated 100 Anti-SAT blocks, obfuscated them with random XOR gates and attacked them with AppSAT as well. In these experiments, AppSAT managed to successfully attack 51 out of 100 structures. Therefore, roughly half the time, AppSAT was able to obtain the full secret key for obfuscated CAS-Lock as well as Anti-SAT structures. On the other hand, for standalone CAS-Lock, AppSAT was only able to find the correct key 5 out of 100 times.

The above phenomenon happens because random XOR/XNOR insertion into the CAS-Lock/AntiSAT structure increases its output corruptibility. During its constraint addition phase, AppSAT is thus able to add in more distinguishing patterns (i.e., multiple DIPs in one iteration). This allows the attack to use these DIPs to rule out a significantly large portion of the wrong key space (including the relatively low output corruptibility keyspace of CAS-Lock/Anti-SAT). Note that AppSAT attack success is also increased when standalone CAS-Lock constructions with very high output corruptibility are used (e.g., a cascade of AND's with OR at the end, which ensures that almost half the input space is corrupted). While AppSAT attack success is not 100% guaranteed, a success rate of almost $\sim 50\%$ indicates a significant vulnerability. Therefore, XOR/XNOR insertion to protect CAS-Lock against removal attacks is not a fool-proof solution. It is recommended that whenever standalone or obfuscated CAS-Lock is employed, the structure be tested against AppSAT to make sure that the correct key cannot be easily recovered.

4 Mirrored CAS-Lock (M-CAS)

In order to protect against removal attacks without the need for XOR/XNOR-based obfuscation, we propose a modification of the original netlist before the insertion of CAS-Lock. The key idea is that even if CAS-Lock is removed using attacks such as SPS, the attacker would be left with a non-functional design. A similar idea was proposed in [26], where gate-level modifications were made to the netlist during the design phase and corrected later on using post-fabrication edit. Instead of the need for edit, we mirror the CAS-Lock structure in the original design. An illustration of this concept is shown in Figure 12. The original design is first locked using CAS-Lock with the secret key K_{secret} hardcoded into the netlist. During re-synthesis, the K_{secret} key values can be propagated to the design using *sweep* (a common constant propagation routine in most logic synthesis algorithms). Therefore, it would not be possible to inspect the netlist and obtain K_{secret}. Note that this is similar to a fundamental assumption in all logic encryption techniques i.e., re-synthesis should make it hard to simply inspect the key gate/design and decide the key bit (e.g., XOR corresponding to key-bit 0 or XNOR corresponding to 1) [27].

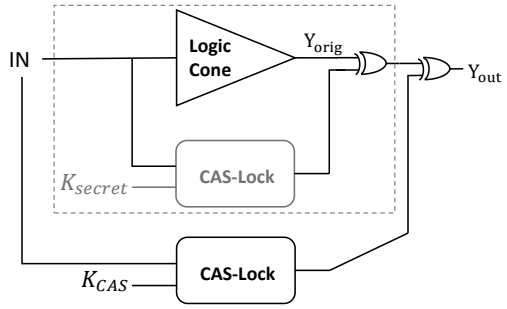
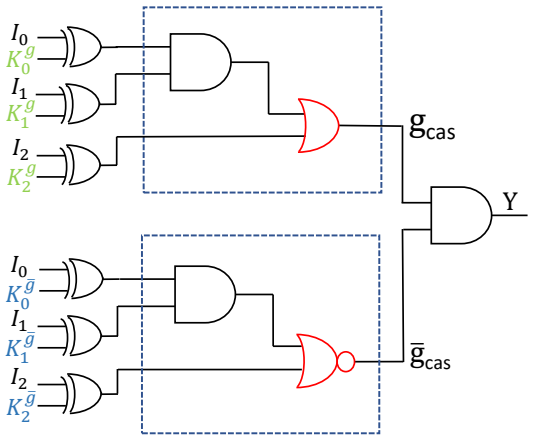


Figure 12: Mirrored CAS-Lock.

After integrating the first CAS-Lock, the design is again locked with a second CAS-Lock structure that is identical to the first one. However, in the second structure, K_{CAS} is assigned as the key input (i.e., one of the primary inputs). Correct functionality of the circuit is achieved as follows:

$$\begin{aligned} Y_{out} &= Y_{orig} \oplus CASLock(K_{secret}) \oplus CASLock(K_{CAS}) \\ &= Y_{orig} \oplus CASLock(K_{secret}) \oplus CASLock(K_{secret}) \\ &= Y_{orig} \oplus F \\ Y_{out} &= Y_{orig} \end{aligned}$$

In the above expressions, we can see that the locked function Y_{out} equals the original function Y_{orig} only when $K_{CAS} = K_{secret}$. Security against removal is achieved as removal of the first CAS-Lock with key K_{CAS} does not guarantee correct functionality; the second CAS-Lock with key K_{secret}, that is embedded inside the original circuit, must also be removed.



I_0	I_1	I_2	g_{cas} ✓K^g	$\bar{g}_{cas}$ ✓$K^{\bar{g}}$	Y ✓K	$\bar{g}_{cas}$ ×$K^{\bar{g}}$	Y ×K
0	0	0	0	1	0	0	0
0	0	1	1	0	0	1	1
0	1	0	0	1	0	0	0
0	1	1	1	0	0	1	1
1	0	0	0	1	0	0	0
1	0	1	1	0	0	1	1
1	1	0	1	0	0	0	0
1	1	1	1	0	0	0	0

Figure 13: CAS-Lock with Maximum Output Corruptibility

4.1 M-CAS Security Analysis

While M-CAS provides protection against removal attacks, it leads to an unavoidable decrease in SAT resistance[9]. In order to understand why this happens, we first consider an example CAS-Lock structure shown in Figure 13. In this construction, the last AND gate

[9]Note that M-CAS is still resistant to bypass attacks, for the same reasons as those mentioned in Section 3.4.2 for CAS-Lock

in the cascade is changed to an OR. For any correct key ($\checkmark K$), g_{cas} and $\bar{g}_{cas}$ are always complementary. Therefore, the CAS-Lock trigger signal Y is always zero and the output is never inverted. However, when the wrong key is provided for g_{cas}, $\bar{g}_{cas}$ or both, Y will be triggered. In Figure 13 (last column), we can see that for nearly half the input patterns, $Y = 1$. Therefore, if we choose this particular wrong key for M-CAS, the error count for the original circuit will be $\frac{2^N}{2} - 1$. When the second CAS-Lock block (with the key inputs) is added, the error count for any given wrong key will vary from 1 to $2 \cdot (\frac{2^N}{2} - 1)$ (when the wrong key introduces $\frac{2^N}{2} - 1$ errors along with the $\frac{2^N}{2} - 1$ errors from the corrupted original circuit).

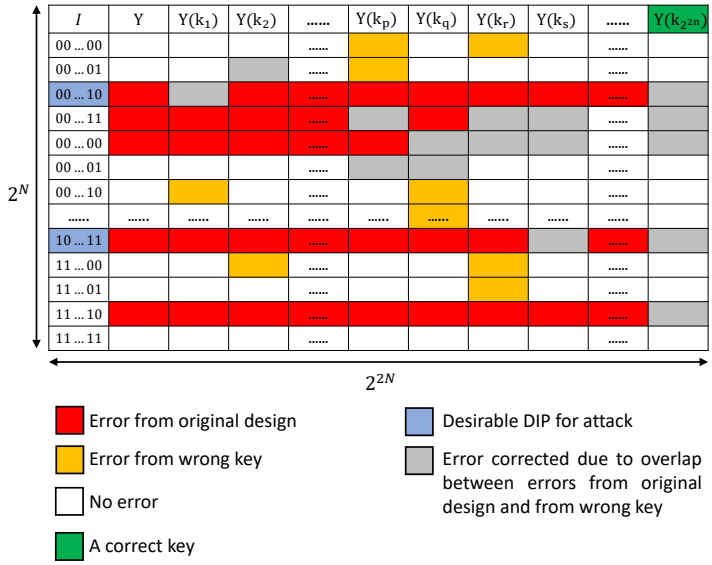


Figure 14: Truth table example for effect of M-CAS

While this high corruptibility is desirable to effectively prevent black-box usage, it also leads to a much faster SAT attack. Figure 14 shows a sample truth table, where the original function Y is corrupted on a number of input patterns due to M-CAS. As seen in the figure, these errors carry onto all of the wrong keys, where they either persist (colored in red) or are muted (colored in grey) due to the additional error from the wrong keys. If the SAT solver happens to pick DIPs such as those colored in blue (*'Desirable DIP for attack'*), it will be able to rule out all the incorrect keys and converge onto the correct key(s) in a single iteration. While this is not guaranteed to always happen, it becomes more likely as the output corruptibility in the original design increases. For structures such as those in Figure 13, the probability would be $\approx \frac{2^N/2}{2^N} \approx 50\%$ (since almost half the input space for every wrong key is corrupted). Thus, the inherent trade-off between corruptibility and SAT attack resistance (which did not exist in CAS-Lock) returns in M-CAS. Picking a high corruptibility key for the original design leads to reduced SAT resistance (and vice-versa).

4.1.1 Analysis of re-synthesis

As with regular logic locking, M-CAS relies on re-synthesis to ensure that the CAS-Lock logic block with the hard-coded K_{secret} (as shown in Figure 12) is not trivially bypassed or removed. The logic block corrupts the functionality of the locked logic cone, with the circuit being unlocked only when $K_{CAS} = K_{secret}$. Re-synthesis is required so that the

attacker cannot (1) deduce the hard-coded K_{secret} by simply observing the gates in the netlist, or (2) nullify the logic block by removal. Re-synthesis helps in transforming the netlist in the following ways.

- Bubble push: Logical inversions in the CAS-Lock block and the original design logic cone are propagated to different parts of the netlist, where they result in the transformation of pre-existing gates (due to DeMorgan's Law).
- Technology mapping: During re-synthesis, the logic synthesis tool maps gates in the netlist to various other gates available in a standard cell library, to meet area, timing and/or power constraints. This also has the direct impact of changing gates in the locked logic cone, and thus altering the logical structure of the gate-level netlist.

An example of a re-synthesized logic block, locked with CAS-Lock and K_{secret}, is shown in Figure 15. From an attacker's perspective, two removal attack strategies can be pursued:

- The attacker could remove all gates in the transitive fan-in of the output of the CAS-Lock block (marked by a red cross in Figure 15). However, this would not be possible as re-synthesis (which involves a combination of bubble push, technology mapping and other logic optimization techniques) would create shared logic between the original logic cone and the CAS-Lock logic cone with K_{secret}. For example, multiple gates in the two separate logic cones can be mapped to a single gate. This can also be visualized with the cone overlap shown in Figure 15. Any attempt to remove the transitive fan-in of CAS-Lock would also remove the logic belonging to the original design logic cone.
- The attacker could also simply set the output of the CAS-Lock block (shown with a red cross in Figure 15) to 0. In theory, this should de-activate CAS-Lock and restore the original circuit functionality. However, due to re-synthesis (and specifically, bubble pushing), the unlocking value is not guaranteed to be 0. In fact, for the circuit shown on the right in Figure 15, the cross marked net actually had to be set to 1 to unlock the circuit, implying that re-synthesis had altered the unlocking value for this net. Further, as also explained in [13], it is not necessary to directly stitch the CAS-Lock output directly to the circuit output. Instead, high observability internal circuit nodes could be utilized without any significant drop in SAT or bypass resistance. This would make it harder to look at the immediate periphery of the locked output pin, and isolate a single net for attack.

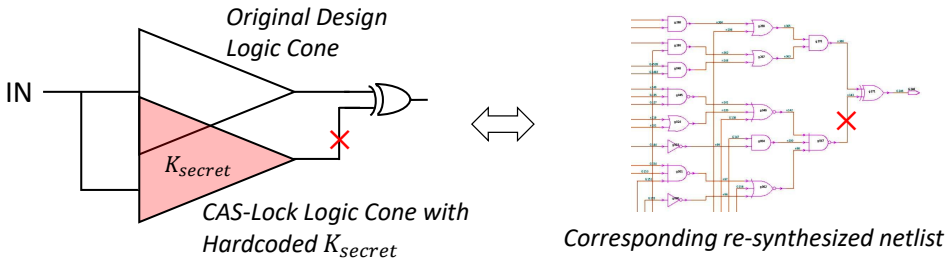


Figure 15: Gate-level netlist obtained after re-synthesis of a logic cone integrated with CAS-Lock with K_{secret}

4.2 Key selection for M-CAS

Unfortunately, while it would be ideal to assess the exact error count (i.e., number of flipped input-output pairs) for every choice of K_{secret}, this would entail evaluating the

entire truth table for all 2^N combinations of the circuit. Since we are mainly concerned with 'excessive' error count (max. of $2^N/2 = 2^{N-1}$), we can set a desired threshold and use any model counting tool (#SAT)[10] or iterative SAT solving to see if the error count for any key exceeds the threshold. Further, we know that any construction of CAS-Lock is going to be better, in terms of lower output corruptibility, than the one shown in Figure 13, where the last gate in the AND (OR) cascade is an OR (AND) gate. This is because it has the highest output corruptibility (maximum p or output-1 count).

In addition, the structure selected for M-CAS also gives us some hints about what kind of key can be chosen to ensure an appropriate level of output corruptibility. For example, in Figure 13, if we set the OR gate key bits $K_2^g = K_2^{\bar{g}}$ while setting $K_1^g = K_1^{\bar{g}}$ and $K_0^g \neq K_0^{\bar{g}}$, the output corruptibility would be $\frac{1}{2^3}$. Conversely, if we set $K_2^g \neq K_2^{\bar{g}}$, we would get the results shown in the last column of Figure 13, where the error rate is $\frac{2^{N-1}-1}{2^N} = \frac{3}{2^3}$. Therefore, by making the OR key gate bits in g_{cas} equal to the OR key gate bits in $\bar{g}_{cas}$ while perturbing a few of the AND gate keys, we can roughly select K_{secret}, whose exact corruptibility can then be verified using a model counter.

In order to get more fine-grained control over output corruptibility, we can use the following heuristics:

- For higher output corruptibility, replace AND gates with OR gates towards the end of a cascade of AND's (i.e., near the output Y). For lower corruptibility, use OR's towards the start of the cascade (i.e., near the PI's). Likewise, if we have a cascade of OR's, replace the OR gates with AND's.

- For each OR gate in the AND cascade, make sure the respective key bit in g_{cas} and $\bar{g}_{cas}$ are not equal. This will cause the output-1 count of CAS-Lock (i.e., corruptibility) to go up. Alternatively, set the OR key bits to be equal to lower the output-1 count.

After these techniques are used to select a CAS-Lock structure and key, we can then use a model counter to determine the exact output corruptibility (since it is guaranteed to be low). Note that while the key selection guideline can help to select an appropriate key, it introduces another vulnerability; if re-synthesis does not change the netlist sufficiently, it could help the attacker to observe the structure in the netlist and deduce key candidates. In that case, inverter pairs can be inserted into the netlist to perform 'bubble pushing' and some of the XOR/XNOR gates can be flipped (i.e., XOR $\Longleftrightarrow$ XNOR). In our experiments, we noticed that synthesis tools such as Design Compiler merged gates from the original design and CAS-Lock, and masked the identity of the CAS-Lock gates (e.g., by replacing them with multi-input gates and/or other complex gates such as AOI, OAI etc.). This would make the task of deciding key bits by observing the CAS-Lock structure to be much harder.

4.3 Comparison to SFLL

A similar approach to M-CAS was also proposed in [6], where a modified version of the original circuit was locked with a hamming distance (HD)-based logic block. This approach is termed SFLL-HD. The HD block receives the circuit inputs (of length N) and key inputs (also of length N), and flips the output whenever the HD between the input and the key is h. The original circuit is modified in such a way that, on application of the correct key, all the errors introduced in the original circuit are corrected. If the incorrect key is applied, the circuit is corrupted on anywhere from 1 input pattern to $2 \cdot \binom{N}{h}$ input patterns (where,

[10]Model counting or sharp-SAT/#SAT is the problem of counting the total number of satisfying assignments to a Boolean formula, usually expressed in conjunctive normal form (CNF) [28]. While #SAT is known to be much harder than SAT, there are efficient solvers that use extensions of well known DPLL heuristics to find/approximate the model count efficiently [29][30].

$\binom{N}{h}$ errors are introduced by the wrong key, and $\binom{N}{h}$ additional errors are introduced due to the modification of the original circuit). Therefore, the mode of operation of M-CAS is similar to SFLL-HD, and both are resilient to removal attacks. However, there are some important differences:

- SFLL-HD allows the designer to set the output corruptibility of the original design from 1 to $\binom{N}{N/2}$ (when $h = N/2$). In contrast, M-CAS can achieve a much higher maximum output corruptibility of 2^{N-1}. However, note that output corruptibility and SAT resistance are inversely related, and a high output corruptibility significantly speeds up the SAT attack (as well as AppSAT).

- M-CAS (and CAS-Lock) allow fine-tuning of the output corruptibility, as there are two modes of control: (1) the design of g_{cas} and $\bar{g}_{cas}$, and (2) the choice of key K_{secret} selected in M-CAS. In contrast, output corruptibility can only vary in discrete ranges from $\binom{N}{1}$ to $2 \cdot \binom{N}{N/2}$ in SFLL-HD.

- M-CAS only requires 2X instances of CAS-Lock. On the other hand, SFLL-HD requires a HD calculation logic, comprised of (i) N XOR's to compute bit-wise difference between the input and the key, (ii) $N - 1$ full adders to add the result of input $\oplus$ key, (iii) XNORs for comparing the XOR sum to the hamming distance parameter h and (iv) AND gates to finally decide if HD(input,key) $= h$. Therefore, M-CAS is expected to be slightly lower in overhead than SFLL-HD. This is also experimentally confirmed in Section 4.4.

- A functional analysis-based attack has also been recently proposed against SFLL [17]. In the 'FALL' attack, the hard-coded key value in the modified logic cone of the design is decoded. This is done by developing a set of functional properties of the hamming distance logic, and deducing key values that satisfy these properties. First, the attack identifies potential gates in the design that are part of the comparator logic (i.e., logic that compares the input to the key - this is also present in M-CAS in the form of the XOR/XNOR of the input with the key). However, straightforward removal of these gates is not possible, as the design has been re-synthesized. This is also the case in M-CAS. After the candidate gates and their support set have been identified, three specific properties of the HD logic (which are used based on the range of the hamming distance) are exploited to deduce the key value: unateness, non-overlapping errors and sliding window. These properties help the attacker to intelligently guess the correct key bits, based on the applied input pattern and response from the HD logic. For example, when HD=1, the attacker first finds two input patterns such that the hamming distance between them is 2, and their outputs match. Since the HD between the hard-coded key and the input patterns must be 1 (as HD was set to 1), any bits that are the same in the two input patterns are the correct key bits. Further bits are decoded by extending this observation.

  Thus, in SFLL, the parameter h (the hamming distance) and input patterns applied to the comparator logic are sufficient to exploit the relationship between the key and the inputs, and deduce the key bits. In contrast, if two or more input patterns are found whose errors overlap in M-CAS, the hard-coded key cannot be directly deduced from the patterns in the same way. Further, the Boolean functions $g_{cas}/\bar{g}_{cas}$ - which can vary depending on the designer-chosen level of corruptibility - must first be identified. However, the design has been re-synthesized and $g_{cas}/\bar{g}_{cas}$ cannot be readily identified from the netlist.

Table 1: SAT and AppSAT resistance of various M-CAS locked logic cones, along with area, power and delay comparisons to SFLL.

Benchmark	|K|	# Error Patterns	# Added Gates	SAT Iterations	SAT Time (Hrs)	AppSAT Success	Area (μm^2)			Delay (ns)			Power (μW)		
							SFLL	M-CAS	% Diff	SFLL	M-CAS	% Diff	SFLL	M-CAS	% Diff
c432	64	22701	580	28182	13.2	No	1926.95	904.81	72.19	3.07	3.17	-3.21	110.80	85.27	26.05
c1908	64	19409	556	N/A	Timeout	No	1771.61	825.03	72.91	2.95	3.32	-11.80	48.57	21.36	77.84
c5315	64	53123	566	N/A	Timeout	No	2015.64	986.00	68.61	3.08	2.93	4.99	53.57	26.05	69.14
c7552	64	53153	550	N/A	Timeout	No	2451.62	1448.73	51.43	3.24	3.17	2.18	48.95	22.44	74.28
sqrt	56	12413	489	N/A	Timeout	No	3937.90	3095.03	23.97	16.74	17.00	-1.54	52.17	23.65	75.23
k2	64	32535	534	N/A	Timeout	No	2380.76	1405.08	51.54	3.11	3.08	0.97	52.27	24.61	71.96
seq	64	24463	554	N/A	Timeout	No	2138.60	1151.66	59.99	3.01	3.03	-0.66	47.97	20.88	78.7

4.4 M-CAS Experimental Analysis

In order to assess overhead, we locked the logic cone of a set of benchmark circuits, from the MCNC and EPFL set [31, 32], using M-CAS. For each benchmark, we selected the CAS-Lock structure and K_{secret} such that the output corruptibility given K_{secret} was at least $10,000$ input patterns and less than $100,000$. The number of patterns was determined using the *dSharp* model counter [29]. For our tested benchmark circuits, it only took a few seconds to find the number of the patterns listed in Table 1. The input length N used was the total number of primary inputs of the logic cone if it was less than 32. Otherwise, we randomly chose $N = 32$ primary inputs. For area/delay evaluation, we synthesized the locked netlists using Synopsys Design Compiler and a generic 90 nm library [33].

After performing locking, we used the SAT and AppSAT attack platforms from [2] and [15] respectively, to evaluate the security of M-CAS for logic cones from various benchmarks. The time-out for SAT was set to 20 hours, and AppSAT was used with the default parameters (as mentioned in [15]). The results are shown in Table 1. We can see that for most of the benchmarks, the SAT attack fails to find the correct key within the allotted time. For C432, SAT attack manages to converge on the correct key. However, larger key sizes and lower error rate in the original design should drastically decrease SAT attack success probability. AppSAT fails on all the benchmarks, as it relies on random patterns to generate additional constraints. Since the errors are sparsely distributed across the entire input space of the circuit, random patterns are unable to 'cover' these errors, leading to pre-mature termination and a wrong key.

We can also see that the number of gates required to implement M-CAS, shown in the Table as '# Added Gates' and calculated using *GateCount*(Logic Cone + M-CAS) - *GateCount*(Logic Cone), is more or less similar across various benchmarks. This is because the number of gates in M-CAS, much like CAS-Lock, mostly depends on N and is independent of the original circuit size. However, gates of CAS-Lock (K_{secret}) can be shared with the original design, due to which we see slight differences in the number of added gates across various benchmarks. In Table 1, we can also see the average area, delay and power difference between M-CAS and SFLL-HD (where $HD = 4$). In all cases, M-CAS is more economical, as SFLL requires a more elaborate circuit (composed of full adders) for the HD calculation, as discussed in Section 4.3. Across all the benchmarks, we can see that there is, on average, an area saving of 57.23% for M-CAS, which becomes especially significant if a large number of logic cones are locked in a circuit. We also obtained the delay and power difference between M-CAS and SFLL-HD. The negative delay difference in Table 1 indicates slightly better SFLL-HD timing performance for some benchmarks (e.g., c432, c1908, sqrt and seq). From the last column in Table 1, we can see that M-CAS fares better in terms of power consumption. This is expected as M-CAS can be implemented with fewer gates than SFLL-HD, resulting in lower leakage and dynamic power consumption. Thus, when area, power and timing are considered together, we can see that M-CAS performs better. This is because the percentage difference in area and power is much higher than the percentage difference in timing. In other words, M-CAS saves more area and power than it loses out on timing to SFLL-HD.

Figure 16 shows how the number of SAT iterations required to break M-CAS varies with

the model count (i.e., the number of patterns for which the original circuit is corrupted) for the apex2 benchmark. In this experiment, the number of inputs used was $N = 10$, and we generated 5000 instances of M-CAS, by varying K_{secret} as well as the CAS-Lock structure (by changing AND's $\leftrightarrow$ OR's). Note that each point in Figure 16 is a separate circuit with varying model counts, on which SAT attack was conducted. Given $N = 10$, the maximum number of SAT iterations is $2^{10} - 1 = 1023$ and the maximum output corruptibility for the modified original design can be $2^{N-1} = 2^9 = 512$ (analogous to 50% Hamming Distance)[11]. We can see that when the model count is either low (i.e., 1) or high (1023), the number of SAT iterations is equal to brute force (i.e., 1023). However, when the model count is varied between these ranges, we can see a sharp decline in SAT resistance (< 400 iterations). In Figure 11, we saw that regardless of the corruptibility of CAS-Lock, the number of SAT iterations **was always maximum** (i.e., equal to $2^N - 1$). However, for M-CAS, we can see a sharp decline in SAT resistance as the output corruptibility of the original circuit is varied. Thus, by adopting M-CAS, we are inevitably sacrificing SAT attack resistance for removal attack resistance.

Also, given the same model count, we can observe some variability in SAT iterations in Figure 16. This is due to the inherent randomness of the SAT solver (e.g., random restarts of the underlying DPLL engine). In any case, as the model count (and thus, output corruptibility) of the original circuit is varied between 1 and 2^N, the SAT attack requires far less iterations to resolve the correct key. In addition, very high corruptibility (e.g., Model Count = 511) leads to a higher success probability for SAT attack. This is evidenced in Figure 16, where we can see that only a few instances of designs with output corruptibility equal to ≈ 511 requires a high number of SAT iterations (e.g., ≈ 500 iterations). Most instances in that corruptibility range terminate in less than 100 iterations.

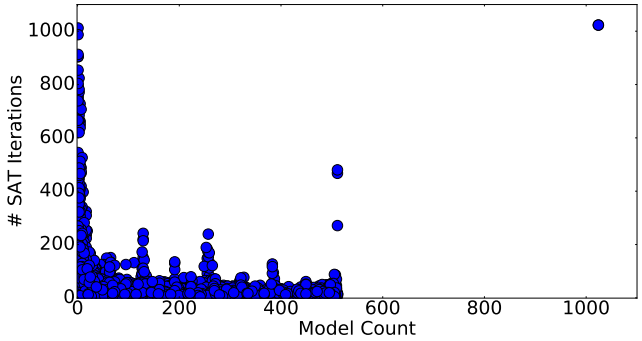


Figure 16: Model Count (i.e., number of incorrect input patterns in the original circuit) vs SAT Iteration for **apex2** Logic Cone with $N = 10$.

5 Discussion

Table 2 shows how each logic locking technique performs in the face of four known attacks. In the table, *Low* refers to the fact that a given attack has low (and almost zero) success probability against a given countermeasure. In other words, the countermeasure is robust against the given attack. *Medium* implies that the attack has a non-negligible success probability and could be successful under some circumstances. For example, M-CAS

[11] Model count when the last gate in the AND tree is an OR gate is 512. A model count in the range $512 < N < 1023$ cannot be produced with any combination of ANDs and ORs in M-CAS. However, with all ORs in the cascade, we can produce a model count of 1023, which is why there is one data point when N=1023. Thus, valid values for model count are $1 < N < 512$ and $N = 1023$, which is why the figure appears truncated between $512 \leq N < 1023$

Table 2: A comparison of various logic locking techniques, attacks and countermeasures. 'Low' means that the attack is not successful on the countermeasure i.e., it is reduced to brute force in the input or key length, or is inapplicable, 'Medium' implies that the attack works with limited success, while 'High' implies that the countermeasure is extremely vulnerable to the attack.

Attacks		*Countermeasures*							
		SLL	**SARLock**	**AntiSAT (SI)**	**AntiSAT (RI)**	**CAS-Lock**	**CAS-Lock + SLL**	**M-CAS**	**SFLL-HD**
Black Box	**SAT**	High	Low	Low	Medium	Low	Low	Medium	Medium
	AppSAT	High	Low	Low	Medium	Low	Medium	Low	Low
	Bypass	Low	High	High	Medium	Low	Low	Low	Low
	FALL	N/A	Low	Low	Low	Low	Low	Low	High
White Box	**Removal**	Low	High	High	High	High	Low	Very Low	Low

configured with sufficiently high output corruptibility leads to a higher SAT attack success rate. *High* means that the countermeasure is extremely vulnerable to the attack.

From the table, we can see that CAS-Lock is well-protected against black-box attacks such as SAT, AppSAT and Bypass. Here, black-box refers to oracle-guided attacks i.e., the attacker uses input-output responses from an unlocked IC to find satisfying assignments. Unlike *white box* attacks, there is no need to compute structural metrics or analyze the gate-level implementation. Therefore, CAS-Lock, coupled with camouflaging (as mentioned in Section 3.5), is sufficient against adversaries such as reverse engineers in the supply chain, who would be forced to take a black-box approach. However, CAS-Lock needs to be fortified with M-CAS to protect against white-box adversaries. This includes untrusted foundries who have a full view of the netlist and gate identities. Similar to MCAS, SFLL-HD is also suitable in the scenario of untrusted foundries.

From the comparisons, we can also see that bypass attack success is linked to output corruptibility. Schemes such as AntiSAT (SI) and SARLock which have extremely low output corruptibility are highly susceptible to bypass attacks. Conversely, SLL, CAS-Lock and similar schemes which can be configured for high output corruptibility are harder to compromise using bypass. Another observation from the comparisons is that output corruptibility is a fundamental limitation for both M-CAS and SFLL, i.e., high corruptibility leads to lower SAT resistance. However, a sufficient level of corruptibility is also desired. This is because a design with little corruptibility is approximately equivalent to the original, which might be good enough for counterfeiters.

6 Conclusion

In this paper, we presented CAS-Lock, a new logic locking scheme which simultaneously combats bypass and SAT attacks, while maintaining non-trivial output corruptibility. We also showed that it can be used as a secure logic locking scheme under a black box attack model. Thorough proofs as well as simulation-based demonstrations were provided to validate CAS-Lock. We also proposed M-CAS, an extension to CAS-Lock, which modifies the original design in order to prevent removal attacks against white-box adversaries such as untrusted foundries. We also evaluated the trade-off between SAT resistance and output corruptibility for M-CAS.

References

[1] M. M. Tehranipoor, U. Guin, and D. Forte. Counterfeit integrated circuits. In *Counterfeit Integrated Circuits*, pages 15–36. Springer, 2015.

[2] P. Subramanyan, S. Ray, and S. Malik. Evaluating the security of logic encryption algorithms. In *Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on*, pages 137–143. IEEE, 2015.

[3] M. Yasin, B. Mazumdar, J. J. V. Rajendran, and O. Sinanoglu. Sarlock: Sat attack resistant logic locking. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 236–241, May 2016.

[4] Y. Xie and A. Srivastava. Mitigating sat attack on logic locking. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 127–146. Springer, 2016.

[5] 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 *International Conference on Cryptographic Hardware and Embedded Systems*, pages 189–210. Springer, 2017.

[6] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. J. Rajendran, and O. Sinanoglu. Provably-secure logic locking: From theory to practice. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1601–1618. ACM, 2017.

[7] Y. Alkabani and F. Koushanfar. Active hardware metering for intellectual property protection and security. In *USENIX security*, pages 291–306. Boston MA, USA, 2007.

[8] J. A. Roy, F. Koushanfar, and I. L. Markov. Epic: Ending piracy of integrated circuits. volume 43, pages 30–38. IEEE, 2010.

[9] J. Rajendran, H. Zhang, C. Zhang, G. S. Rose, Y. Pino, O. Sinanoglu, and R. Karri. Fault analysis-based logic encryption. *IEEE Transactions on Computers*, 64(2):410–424, 2015.

[10] R. Torrance and D. James. The state-of-the-art in ic reverse engineering. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 363–381. Springer, 2009.

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

[12] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri. Security analysis of logic obfuscation. In *Proceedings of the 49th Annual Design Automation Conference*, pages 83–89. ACM, 2012.

[13] Y. Xie and A. Srivastava. Anti-sat: Mitigating sat attack on logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2018.

[14] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran. Security analysis of anti-sat. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pages 342–347. IEEE, 2017.

[15] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin. Appsat: Approximately deobfuscating integrated circuits. In *Hardware Oriented Security and Trust (HOST), 2017 IEEE International Symposium on*, pages 95–100. IEEE, 2017.

[16] A. Sengupta, M. Nabeel, M. Yasin, and O. Sinanoglu. Atpg-based cost-effective, secure logic locking. In *2018 IEEE 36th VLSI Test Symposium (VTS)*, pages 1–6, April 2018.

[17] D. Sirone and P. Subramanyan. Functional analysis attacks on logic locking. *arXiv preprint arXiv:1811.12088*, 2018.

[18] M. Yasin, J. J. Rajendran, O. Sinanoglu, and R. Karri. On improving the security of logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(9):1411–1424, 2016.

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

[20] M. Chen, E. Moghaddam, N. Mukherjee, J. Rajski, J. Tyszer, and J. Zawada. Hardware protection via logic locking test points. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2018.

[21] H. M. Kamali, K. Z. Azar, H. Homayoun, and A. Sasan. Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 89. ACM, 2019.

[22] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran. Removal attacks on logic locking and camouflaging techniques. *IEEE Transactions on Emerging Topics in Computing*, (1):1–1, 2017.

[23] J. Rajendran, M. Sam, O. Sinanoglu, and R. Karri. Security analysis of integrated circuit camouflaging. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 709–720. ACM, 2013.

[24] M. Li, K. Shamsi, T. Meade, Z. Zhao, B. Yu, Y. Jin, and D. Z. Pan. Provably secure camouflaging strategy for ic protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.

[25] B. Shakya, H. Shen, M. Tehranipoor, and D. Forte. Covert gates: Protecting integrated circuits with undetectable camouflaging. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 86–118, 2019.

[26] B. Shakya, N. Asadizanjani, D. Forte, and M. Tehranipoor. Chip editor: Leveraging circuit edit for logic obfuscation and trusted fabrication. In *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD '16, pages 30:1–30:8, New York, NY, USA, 2016. ACM.

[27] S. M. Plaza and I. L. Markov. Solving the third-shift problem in ic piracy with test-aware logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(6):961–971, 2015.

[28] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[29] C. Muise, S. A. McIlraith, J. C. Beck, and E. I. Hsu. D sharp: Fast d-dnnf compilation with sharpsat. In *Canadian Conference on Artificial Intelligence*, pages 356–361. Springer, 2012.

[30] M. Thurley. sharpsat–counting models with advanced component caching and implicit bcp. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 424–429. Springer, 2006.

[31] S. Yang. *Logic synthesis and optimization benchmarks user guide: version 3.0.* Microelectronics Center of North Carolina (MCNC), 1991.

[32] L. Amarú, P.-E. Gaillardon, and G. De Micheli. The epfl combinational benchmark suite. In *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, number EPFL-CONF-207551, 2015.

[33] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, et al. Freepdk: An open-source variation-aware design kit. In *Microelectronic Systems Education, 2007. MSE'07. IEEE International Conference on*, pages 173–174. IEEE, 2007.