2025 Design, Automation & Test in Europe Conference (DATE 2025)

Cute-Lock: Behavioral and Structural Multi-Key Logic Locking Using Time Base Keys

Kevin Lopez
Computer Engineering & Computer Science Department
California State University, Long Beach
Long Beach, CA, USA
Kevin.LopezChavez01@student.csulb.edu

Amin Rezaei
Computer Engineering & Computer Science Department
California State University Long Beach
Long Beach, CA, USA
Amin.rezaei@csulb.edu

***Abstract*—The outsourcing of semiconductor manufacturing raises security risks, such as piracy and overproduction of hardware intellectual property. To overcome this challenge, logic locking has emerged to lock a given circuit using additional key bits. While single-key logic locking approaches have demonstrated serious vulnerability to a wide range of attacks, multi-key solutions, if carefully designed, can provide a reliable defense against not only oracle-guided logic attacks, but also removal and dataflow attacks. In this paper, using time base keys, we propose, implement and evaluate a family of secure multi-key logic locking algorithms called Cute-Lock that can be applied both in RTL-level behavioral and netlist-level structural representations of sequential circuits. Our extensive experimental results under a diverse range of attacks confirm that, compared to vulnerable state-of-the-art methods, employing the Cute-Lock family drives attacking attempts to a dead end without additional overhead.**

***Index Terms*—Logic Locking; SAT Attack; Removal Attack; Multi-Key Locking; Dynamic Locking**

I. Introduction

As chip manufacturing becomes increasingly outsourced, the need for robust protection mechanisms is constantly increasing. In this split of design and manufacturing, one company designs the digital Integrated Circuit (IC), while another handles the physical fabrication phase. Logic locking [1]–[3] has emerged as a promising solution to prevent piracy and overproduction of ICs. Combinational locking is the process of adding additional inputs, called key bits, to an IC to corrupt the output when the incorrect key is inserted. On the other hand, sequential locking is the procedure of adding additional obfuscation states where the user needs to traverse through those states to utilize the circuit.

Traditional logic locking solutions are susceptible to the oracle-guided SAT attack [4] that extracts the key using an oracle (i.e., a working chip bought off the market) and a locked netlist, potentially leaked from an untrustworthy foundry. Although different logic locking solutions have been proposed to protect against the original SAT attack [5], [6], they are still susceptible to approximate versions of oracle-guided SAT attacks [7], [8] as well as dataflow and removal attacks [9], [10] that try to remove or reverse engineer the added lock and extract the original circuit.

While the majority of the logic locking solutions have been focused on single-key methods, we believe that their vulnerabilities against powerful attacks can be effectively mitigated through time-based multi-key logic locking. Thus, in this paper, we introduce the idea of advanced multi-key logic locking for sequential circuits in which the states are locked using a counter and multiple keys that are applied at different times. For the circuit to operate correctly, these key values must be provided in a specific order determined by the corresponding clock cycle. The motivation is to improve security by complicating the decoding process, ensuring that the circuit only operates as intended when the correct key sequence is applied at the correct time. Our proposed approach uses the current hardware to manipulate the state transition and provides a low-overhead solution for sequential logic locking. With the addition of multiple keys, it maintains resilience against oracle-guided attacks with [4], [7], [8] or without [11]–[13] scan chain access assumption. Additionally, it improves structural integrity against dataflow [9] and removal [10] attacks. In this paper, we present the following contributions:

- Proposing **Cute-Lock-Beh**, a **c**ounter-based m**ult**i-k**e**y logic **lock**ing **beh**avioral solution to secure circuits against logic attacks in RTL-level representation;
- Proposing **Cute-Lock-Str**, a **c**ounter-based m**ult**i-k**e**y logic **lock**ing **str**uctural solution to secure circuits against both logic and structural attacks in the gate-level netlist;
- Generating more than 60 benchmarks based on the proposed methods, evaluating their overhead, and testing their security against state-of-the-art oracle-guided as well as removal and dataflow attacks.

II. Background and Related Work

In this section, we begin by examining the development of logic locking methods alongside oracle-guided and removal attacks, followed by an overview of current approaches in multi-key logic locking.

A. Logic Locking Techniques

Logic locking techniques can be categorized into combinational locking and sequential locking. Initial combinational logic locking solutions were XOR-based and MUX-based mechanisms [1], [2]. However, the oracle-guided SAT attack [4] has exposed vulnerabilities in these methods, leading to

978-3-9826741-0-0/DATE25/© 2025 EDAA

the development of more robust techniques such as Anti-SAT [5], SAR-Lock [6], TT-Lock [14], SFLL [15], BLE [16], DLE [17], and others [18]–[21] that increase the time complexity of SAT attacks. Furthermore, HARPOON [3] is a sequential logic locking solution that adds additional states to provide an obfuscation mode; in order to utilize the circuit, one must navigate through the obfuscation mode first. Boosted Finite-State Machine (BFSM) [22] uses a Physical Unclonable Function (PUF) to determine the starting state, which is typically an obfuscated state; to unlock and use the circuit, a specific input sequence must be provided to traverse from the obfuscated state to the functional state. In addition, Dynamic State Reflection (DSR) [23] adds black holes to hide the boundary between obfuscation and functional modes. While all these logic locking methods have been challenged by different attacks such as SAT attack [4], Key-Condition Crunching (KC2) [12], Bounded Model Checking (BMC) [11], and Reverse Assessment of Netlist Encryption (RANE) [13], researchers have explored other combinational and sequential methods [24]–[28] as well.

Adding extra states would incur additional overhead when using the circuit since one would need to transition between the obfuscated states. In addition, single-key solutions remain susceptible once the key is compromised, potentially exposing the entire security of the IC. In our proposed solutions, different keys need to be provided to the circuit at different times, and they re-route users to the wrong states whenever a wrong key is applied. In this case, the circuit starts at the initial state and will maintain the same state transition as the original circuit as long as the correct keys are applied.

B. Logic Locking Attacks

Logic locking attacks are categorized into oracle-guided logic attacks as well as removal attacks. The oracle-guided SAT attack [4] is one of the most powerful attacks against combinational locking and sequential locking with scan access; it uses a SAT solver to iteratively find Discriminating Input Patterns (DIPs) that eliminate incorrect keys, eventually converging on the correct key. This attack has been shown to break almost all of the early logic locking schemes efficiently [1]–[3]. An extension of the SAT attack, AppSAT [7] aims to find approximate keys that work for most input patterns. This attack is particularly effective against schemes such as Anti-SAT [5] with low output corruptibility. The Double DIP attack [8] improves upon the SAT attack by finding two DIPs in each iteration, allowing it to break certain SAT-resistant locking schemes such as SAR-Lock [6].

On the sequential side, the BMC attack [11] targets sequential logic locking by unrolling the circuit for a fixed number of time steps and using a SAT solver to find the key. The KC2 attack [12] improves upon the BMC attack by using incremental SAT solving and dynamic simplification of key conditions. The RANE attack [13] uses API-based invocation of formal verification tools to model the initial state as a secret key variable and find the unlocking sequence.

In addition, the Functional Analysis Attack on Logic Locking (FALL) [10] is primarily a removal attack, with some structural components designed to extract the logic locking key. FALL has shown to be successful against logic locking techniques such as TTLock [14] and SFLL [15]. In addition, the Dataflow-based Netlist Analysis (DANA) attack [9] is designed to assist reverse engineering circuits by structuring an unstructured sea of gates. The key operation is to group registers into distinct clusters, which can then be analyzed to derive the high-level architecture and functionality of the circuit. By focusing on the flow of data between Flip-Flops (FFs), DANA helps to recover meaningful high-level structures from a flatten netlist, making it a crucial first step in the reverse engineering process. Other attacks have also been proposed [29]–[32] targeting a subset of existing logic locking solutions.

C. Multi-Key Approaches

Recently, the idea of using multi-key solutions to mitigate oracle-guided SAT attacks is proposed [33]–[37]. DK-lock [33] is a two-key logic locking solution that uses an activation key and then a functional key. The control logics of both the activation phase and the functional phase then unified into an FSM [34]. DK-Lock is susceptible to attacks like [31] that can expand the key size to reverse the method back to a single-key solution. SLED [35] is another multi-key sequential solution that works by dynamically changing the keys during circuit operation generated by a secure module based on a static seed. However, since it depends on a seed value to operate, it is vulnerable to oracle-guided SAT attacks if the attacker deciphers the seed value as the initial key. Gate-Lock [36] uses an approach focused on locking gates; the resulting key to use the circuit changes depending on the input needed. In addition, K-Gate Lock [37] is based on input encoding and can be fully implemented using combinational logic without the need for state-holder components. However, neither provides any structural benefit against dataflow and removal attacks.

While each state-of-the-art multi-key logic locking method aims to secure against a different attack, our goal in this paper is to propose low-overhead multi-key solutions at both the RTL-level and netlist-level that are secure against all the above-mentioned attacks.

III. Cute-Lock Family

In this section, after explaining the terminology, we discuss our proposed methods of locking a circuit with multiple keys; the first one, called **Cute-Lock-Beh**, is an RTL-level sequential logic locking approach that is secure against oracle-guided attacks and requires different key values based on different clock cycles to operate correctly. The second method, called **Cute-Lock-Str**, is a netlist-level implementation of our behavioral solution that resists not only oracle-guided SAT attacks but also dataflow and removal attacks.

A. Terminology

In the context of **Cute-Lock** family, it is crucial to understand the terminology used to describe the components:

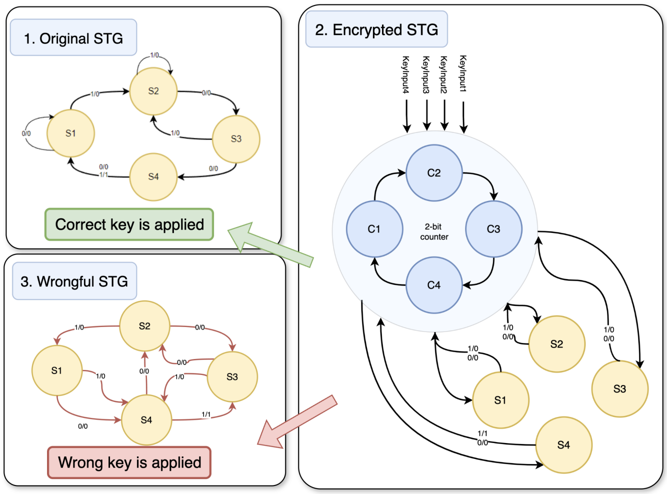


Fig. 1: **Cute-Lock-Beh** STG example

n: Number of inputs to the circuit.

k: Number of key values (multiple keys, in contrast to traditional logic locking algorithms that used a single key).

k$_i$: Number of bits in each key value (i.e., the key size).

c: Number of clock cycles for the counter, determining when specific keys must be provided.

m: Number of layers in the MUX tree equal to $\log_2(k) + 1$.

B. Behavioral Solution

For RTL-level sequential circuit, the core idea is to require a specific key value based on the time count for the circuit to behave as intended. When an incorrect key is provided to a particular clock cycle, the circuit transitions to a random, incorrect state. Fig. 1 demonstrates how **Cute-Lock-Beh** affects the high-level State Transition Graph (STG) using a 1001 sequence detector example. Here, we can see that the keys and counter work together to handle the state transition for the next state, and whenever the wrong key is provided, a wrongful state transition is taken. Implementing **Cute-Lock-Beh** requires minimal changes to the RTL code. The only additions are a counter and the wrongful state transitions when a wrong key is provided. These wrongful state transitions are added to the FF logic, where the present state updates occur. Here are the components of the sequence detector example in Fig. 1:

1. Original STG: The original STG for detecting a 1001 sequence as a mealy machine. Also, it serves as the same STG whenever the correct keys are provided.

2. Encrypted STG: The encrypted STG with four keys, 4 bits each, and a 2-bit counter. The correct state transition would be determined by the counter, along with providing the proper key. If the wrong key is provided, a wrongful state transition is taken.

3. Wrongful STG: The incorrect STG is constructed of random state transitions defined at the RTL-level.

While **Cute-Lock-Beh** does not represent a significant change in the RTL-level representation, if one plans to convert a given netlist back to STG format, they face the state explosion problem [38]. Thus, currently, we have used Xilinx Vivado to implement **Cute-Lock-Beh** that use MUXs instead of redesigning the STG from the ground up. However, in this case, while it provides security against oracle-guided SAT attacks, it does not provide substantial structural benefits against removal attacks. Thus, it requires another step of obfuscation on top of the behavioral method. To address this issue, we discuss a more efficient structural solution with lower overhead in Section III-C.

C. Structural Solution

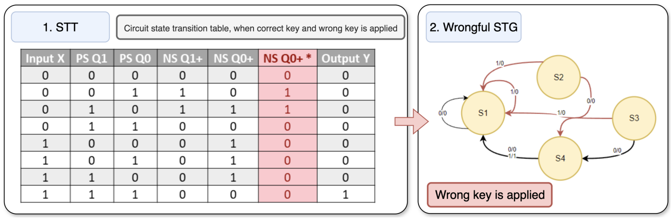
In **Cute-Lock-Str**, instead of transitioning to a random state upon wrong key insertion, it moves to a different state predefined by existing state transitions. Fig. 2 demonstrates how **Cute-Lock-Str** affects the State Transition Table (STT) using the same 1001 sequence detector used in Fig. 1:

1. STT: The STT for the 1001 sequencer detector along with wrongful state transition. The highlighted column (NS Q0+*) indicates values for "NS Q0+" whenever the wrong keys are provided. This will make up the new wrongful state transition. In this case, the hardware from "NS Q1+" is repurposed to be used on wrongful state transition. For example, if the current state (PS Q1, PS Q0) is "00" and input X is 1, the next state (NS Q1+, NS Q0+) should be "01", but when a wrong key-counter combination is given, it will stay at state "00".

2. Wrongful STG: The incorrect STG is constructed from "NS Q1+" and "NS Q0+*".

The FFs in the structural solution are connected through a tree of MUXs illustrated in Fig. 3. The gray clouds show the correct hardware for the FF to perform the state transition correctly, and the red clouds show the hardware of another FF that would accomplish the wrongful state transition. The MUX tree allows the ability to lock an FF with multiple keys synchronized with a counter and enables the FFs to utilize incorrect hardware when the wrong key is provided at a specific time. The MUX tree is made up of m layers, where m is $log_2(k) + 1$. Each layer consists of the following:

1st Layer: The first layer of MUXs verifies that the correct key has been provided, denoted by $keyinput_1$ to $keyinput_{k_i}$. This layer also defines the key size k_i, which is determined by the *select* size of the multiplexer. The number of different wrongful hardware configurations is given by $2^{k_i} - 1$. In Fig. 3, the keys are 00, 10, 00, and 10


1. STT

Circuit state transition table, when correct key and wrong key is applied

Input X	PS Q1	PS Q0	NS Q1+	NS Q0+	NS Q0+ *	Output Y
0	0	0	0	0	0	0
0	0	1	1	0	1	0
0	1	0	1	1	1	0
0	1	1	0	0	0	0
1	0	0	0	1	0	0
1	0	1	0	1	0	0
1	1	0	0	1	0	0
1	1	1	0	0	0	1

Fig. 2: **Cute-Lock-Str** STG & STT example

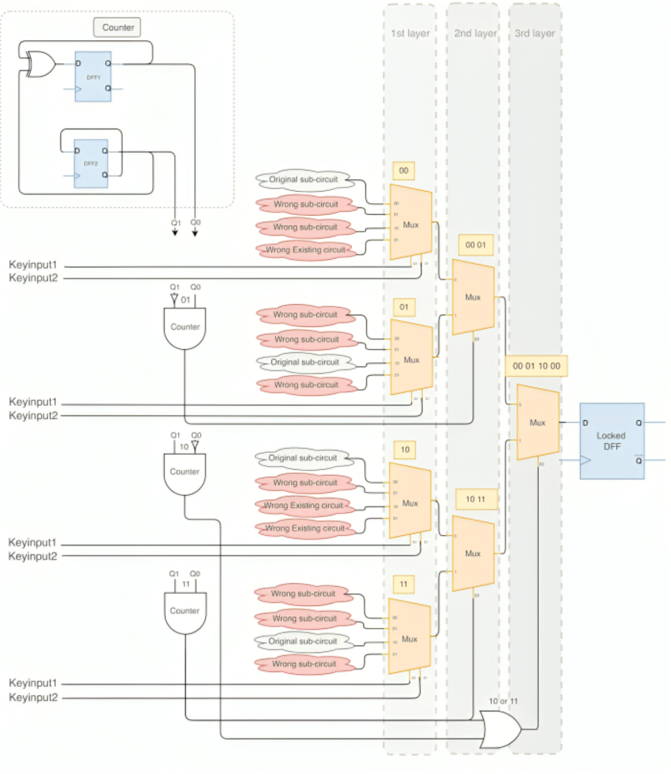


Fig. 3: **Cute-Lock-Str** MUX tree example

in respective counter times. The key sizes and wrongful hardware are denoted by the 4-to-1 MUXs.

2nd, 3rd, ..., (m − 1)th Layers These layers are controlled by the counter to select the correct MUX. The counter values are divided at each subsequent layer. The MUXs at these layers determine whether to use the top or bottom connection based on the previous counter values applied in the bottom MUXs. The check is performed by OR-ing all the counter times in the previous MUXs and feeding the result into the *select* input of the current MUX.

mth Layer The output of the final MUX is fed into the FF.

The MUX-tree has a width of $log_2(k) + 1$ and a height of k where k is the number of distinct keys to lock the FFs. When using **Cute-Lock-Str**, it is possible to lock any number of FFs. While locking one FF with different keys is enough to resist oracle-guided SAT attacks, locking more FFs would provide more resilience against dataflow and removal attacks.

IV. Experimental Results

We conduct experiments on a Windows 11 machine, which accesses Linux Ubuntu 22.04 via WSL2. The machine is an Intel 13900H with 14 cores and 32 threads at 2.6 GHz and 56 GB of DDR5 RAM. First, we validate the **Cute-Lock** family and then test them against oracle-guided SAT attacks. Next we test **Cute-Lock-Str** against dataflow and removal attacks and finally implement it on Cadence Genus to analyze and compare the overhead with state-of-the-art works.

In the tables, different colors are used to indicate specific conditions. The color light red[1] represents the "condition not solvable" status. A deeper red[2] signifies a wrong key, while the darkest red[3] indicates that the attack failed. Green[4] denotes that the correct key has been found. Yellow[5] means the attack did not report any key within the time limit of 20 hours. The source codes and created benchmarks of **Cute-Lock** family are publicly available on our GitHub repository[1].

A. *Algorithm Validation*

The validation of **Cute-Lock** is done in Xilinx Vivado. Locking benchmarks with the same key values (i.e., reduced to a single-key solution) leads to SAT attacks from NEOS [39] and RANE [13] to find the correct key as expected.

TABLE I: **Cute-Lock-Beh** validation

	Inputs	**Outputs**		
Time (ns)	**x[7:0]**	**y[38:0]**	$y_{ck}[38:0]$	$y_{wk}[38:0]$
0	0	0	0	0
60	2aaaa	0	0	400000000
100	3c3c3	2000002007	2000002007	00000000e
120	3c3c3	1800000002	1800000002	00000000e
160	2aaaa	0	0	00000000e
200	3c3c3	1800000002	1800000002	000071
220	2aaaa	400240	400240	91
240	2aaaa	0	0	91
260	2aaaa	0	0	91
280	2aaaa	0	0	2004
300	0	0	0	2004
320	0	0	0	0
340	0	0	0	0
360	0	0	0	0
380	0	20000002007	20000002007	e
380	3c3c3	1800000002	1800000002	e

1) Behavioral Solution: For **Cute-Lock-Beh** algorithm validation, we lock the *bcomp* benchmark from the Synthezza suite [40] with 19 key-bit values. When the correct key values are provided, both the original and the locked circuit behave the same, as shown in Table I, where columns y_{ck} and y_{wk} are the outputs under the correct and wrong keys, respectively.

TABLE II: **Cute-Lock-Str** validation

	Inputs				**Outputs**		
Time (ns)	**G0**	**G1**	**G2**	**G3**	**G17**	$G17_{ck}$	$G17_{wk}$
0	0	1	0	1	x	x	x
20	1	0	1	0	1	1	1
40	1	1	0	0	1	1	1
60	1	1	1	0	1	1	1
80	0	1	0	1	1	1	1
100	1	0	1	0	1	1	1
120	0	0	0	0	1	1	1
140	1	1	1	1	1	1	0
160	0	0	1	1	0	0	0
180	1	0	0	1	0	0	1
200	0	1	1	0	0	0	0
220	0	1	1	1	0	0	1
240	1	1	0	1	1	1	1
260	0	0	0	1	1	1	1
280	1	0	1	1	1	1	0

2) Structural Solution: For the **Cute-Lock-Str** algorithm validation, *s27* from ISCAS'89 [41] is locked using the following keys: 1, 3, 2, 0. When the correct key values are provided, the original and the locked circuit behave equally,

[1] 🟥 CNS, [2] 🟥 x..x, [3] 🟥 FAIL, [4] 🟩 Equal, [5] 🟨 N/A

[1] https://github.com/cars-lab-repo/Cute-Lock

TABLE III: **Cute-Lock-Beh** security against logic attacks

Benchmark and Locking Information				NEOS [39]		
Synthezza [40]	Circuit	# Keys (k)	Key Size (k_i)	BBO	INT	KC2
Small	bcomp	6	18	6m25.446s	0m0.885s	0m1.030s
	bech	6	18	6m4.845s	0m0.723s	0m0.838s
	bridge	5	16	3m28.614s	0m0.100s	0m0.182s
	cat	3	11	15m1.161s	0m0.772s	0m0.680s
	checker9	3	10	3m0.931s	0m0.842s	0m0.803s
	cpu	4	14	2m11.658s	0m0.732s	0m0.799s
	dmac	2	7	1m45.751s	0m0.623s	0m0.681s
	e10	3	10	3m17.832s	0m0.816s	0m1.033s
	e15	4	13	8m59.511s	0m1.361s	0m1.462s
	e16	4	13	7m50.966s	0m0.774s	0m0.918s
	e161	5	16	2m53.761s	0m0.731s	0m0.759s
	e17	2	8	15m0.543s	0m0.522s	0m0.607s
Medium	acdl	5	16	14m47.149s	0m0.641s	0m1.157s
	alf	0	31	0m0.180s	0m0.107s	0m0.469s
	amtz	7	23	14m9.747s	0m2.227s	0m2.727s
	ball	4	44	15m5.744s	0m1.162s	0m4.998s
	bens	7	21	15m3.290s	0m18.365s	0m19.804s
	berg	7	21	10m5.736s	0m1.730s	0m2.531s
	bib	7	21	15m3.795s	0m2.750s	0m3.324s
	big	6	18	11m14.492s	0m0.658s	0m1.163s
	bs	6	19	9m52.679s	0m0.600s	0m0.895s
	codec	2	4	15m2.282s	0m2.252s	0m2.032s
	codec1_2	9	28	15m4.756s	0m3.768s	0m4.005s
	cow	6	49	15m7.930s	0m1.225s	0m4.587s
	cyr	6	20	14m7.341s	0m2.375s	0m3.072s
	dav	6	18	15m3.939s	0m0.519s	0m1.019s
	doron	7	22	13m49.117s	0m2.854s	0m4.004s
Large	absurd	21	65	15m25.370s	0m40.360s	1m9.523s
	bulln	20	61	15m23.190s	1m19.553s	8m7.918s
	camel	19	59	16m29.513s	3m38.506s	14m34.180s
	exxm	15	47	15m52.984s	3m46.605s	3m43.372s
	lion	18	55	15m37.603s	1m31.160s	4m26.537s
	tiger	17	51	15m50.498s	0m37.245s	1m54.818s

TABLE IV: **Cute-Lock-Str** security against logic attacks

Benchmark and Locking Information				NEOS [39]			RANE [13]
	Circuit	# keys (k)	Key Size (k_i)	BBO	INT	KC2	RANE
ISCAS'89 [41]	s1196	4	14	1m20.096s	0m0.694s	0m0.753s	0m1.667s
	s13207	8	31	15m5.270s	0m15.520s	0m19.852s	0m35.909s
	s1488	2	8	1m37.663s	0m0.672s	0m0.723s	0m1.224s
	s15850	4	14	15m9.218s	0m12.460s	0m14.394s	0m20.279s
	s298	2	3	0m0.043s	0m0.474s	0m0.474s	0m0.798s
	s349	4	9	7m21.210s	0m0.672s	0m0.695s	0m0.964s
	s35932	8	35	15m5.694s	3m43.671s	4m9.463s	3m31.814s
	s510	8	19	0m35.772s	0m0.539s	0m0.540s	0m0.942s
	s5378	8	35	7m50.965s	0m1.287s	0m1.486s	0m3.591s
	s641	8	35	0m58.063s	0m0.804s	0m1.191s	0m1.326s
	s713	8	35	0m56.985s	0m0.624s	0m0.659s	0m1.234s
	s832	8	18	0m49.561s	0m0.563s	0m0.603s	0m1.080s
	s9234	8	19	15m4.725s	6h44m50s	7h56m45s	50m 6.04s
	s953	4	15	0m52.608s	0m0.826s	0m0.127s	2h6m4.59s
ITC'99 [44]	b01	2	2	0m0.296s	0m1.023s	0m0.882s	9m6.02s
	b02	2	2	0m0.143s	0m0.487s	0m0.653s	10m39.54s
	b03	2	4	15m0.528s	0m0.473s	0m0.653s	13m6.39s
	b04	4	11	0m52.426s	0m0.820s	0m0.194s	4h5m53.21s
	b05	2	2	0m0.153s	0m0.097s	0m0.089s	0m0.415s
	b06	2	1	0m0.151s	0m0.402s	0m0.165s	0m0.441s
	b07	2	2	0m0.163s	0m0.739s	0m0.863s	0m0.544s
	b08	4	9	0m14.811s	0m0.600s	0m0.186s	14m34.59s
	b09	2	1	0m0.250s	0m0.658s	0m0.698s	0m0.560s
	b10	4	11	0m16.103s	0m0.719s	0m0.206s	16m48.31s
	b11	2	7	1m36.385s	0m1.699s	0m0.256s	23m17.52s
	b12	2	5	16m20.762s	1m24.733s	0m0.261s	1h27m50.43s
	b14	8	32	15m3.473s	1m55.654s	0m1.083s	19m39.38s
	b15	16	36	14m3.219s	20m0.006s	0m4.006s	40m34.59s
	b17	16	37	17m1.496s	20m0.008s	20m0.011s	20h0m0.35s
	b18	16	37	0m0.320s	0m0.258s	0m0.252s	1h59m18.06s
	b19	8	24	0m0.538s	0m0.574s	0m0.752s	18h6m12.74s
	b20	8	32	15m6.045s	6m20.914s	0m4.988s	0m57.046s
	b21	8	32	15m11.598s	6m49.946s	0m5.389s	0m59.616s
	b22	8	32	15m24.620s	20m0.005s	2m41.473s	2m24.392s

as shown in Table II, where $G17_{ck}$ and $G17_{wk}$ represent the output of the circuit when the correct keys and the wrong keys are provided, respectively.

B. Logic Attacks Evaluation

One of the main objectives of **Cute-Lock** family is to generate a locking mechanism that oracle-guided SAT attacks will not be able to decrypt. In this section, we will test **Cute-Lock-Beh** and **Cute-Lock-Str** against SAT-based oracle-guided attacks.

1) Behavioral Solution: To test **Cute-Lock-Beh**, we generate locked versions of the Synthezza benchmark suite [40] in Verilog format, then use Yosys [42] to convert to .blif format. While the files are in .blif format, it is necessary to convert some FFs into latches. Then, ABC [43] is used to convert to .bench format, which is used to run the SAT attacks using NEOS [39]. None of the benchmarks run provide the correct keys as shown in Table III.

2) Structural Solution: To evaluate **Cute-Lock-Str**, we generate locked versions of ISCAS'89 [41] and ITC'99 [44]. The encryption is done in .bench format with our Python implementation of **Cute-Lock-Str** and tested against NEOS [39] and RANE [13] attacks. None of the benchmarks run provide the correct keys as shown in Table IV.

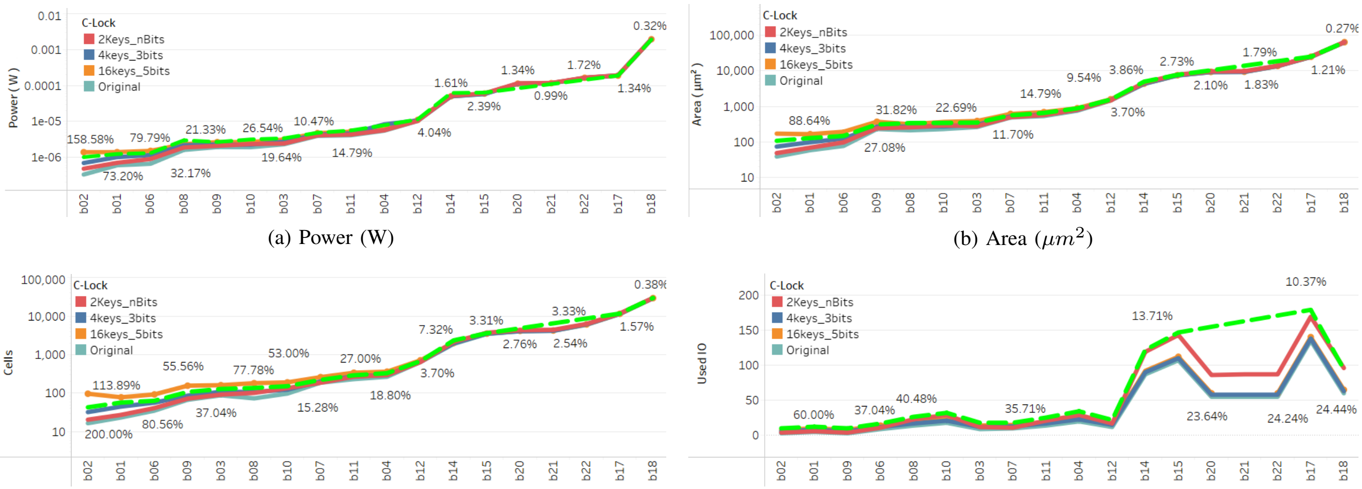
C. Removal Attacks Evaluation

As mentioned before, **Cute-Lock-Beh** does not add much benefit to security against removal attacks; however, **Cute-Lock-Str** allows the circuit to resist against them.

1) Dataflow Attack: To execute DANA [9], we synthesize the ITC'99 [44] benchmarks using Xilinx Vivado. After preparing the netlists, we apply the DANA script to analyze the dataflow and generate the resulting register clusters. DANA does not provide a simple pass/fail output. Instead, it produces clusters that represent potential high-level registers within the circuit. These clusters are then evaluated using the Normalized Mutual Information (NMI) metric, which measures how closely DANA's output matches the ground truth or, in our case, the original circuit. An NMI value of "0" means the tool fails to identify the correct register groupings, while an NMI value of "1" means that DANA's output perfectly matches the reference design. In the original study, DANA was able to get very high NMI scores in the range of 0.87 to 0.99 and an average of 0.95 when compared against the ground truth. When we run DANA against locked benchmarks with **Cute-Lock-Str**, as shown in Table V, it is clear that the NMI scores accuracy drops significantly to a wide range of 0.00 to 0.99 and an average of 0.41. These results demonstrate that **Cute-Lock-Str** changes the dataflow in most of the locked benchmarks compared to the original benchmarks and thus **Cute-Lock-Str** is able to increase resiliency against dataflow attacks and

TABLE V: **Cute-Lock-Str** security against removal attacks

	DANA [9]	FALL [10]		
Circuit	NMI Score	Candidates	Keys	CPU Time (s)
b01	0.00	0	0	0.1
b02	0.00	0	0	0.1
b03	0.00	0	0	0.1
b04	0.00	0	0	0.1
b05	0.00	0	0	0.2
b06	0.00	0	0	0.1
b07	0.74	0	0	0.1
b08	0.99	0	0	0.1
b09	0.43	0	0	0.1
b10	0.00	0	0	0.1
b11	0.76	0	0	0.1
b12	0.99	0	0	0.1
b14	0.60	0	0	10.9
b15	0.89	0	0	16.8
b17	0.93	0	0	97.3
b18	0.93	0	0	1663.6
b19	0.50	0	0	3423.4
b20	0.56	0	0	23.4
b21	0.44	0	0	23.7
b22	0.39	0	0	46.4



(a) Power (W)

(b) Area (μm^2)

(c) Cell Count

(d) Number of IOs

Fig. 4: Overhead comparison of **Cute-Lock-Str** with DK-Lock [33]

deteriorate reverse engineering in sequential benchmarks.

2) Functional Analysis Attack: FALL [10] is designed to work with circuits in .bench format, which we use to test against our locked circuits. In the original study, FALL is reported to be successful against 65 out of 80 locked circuits (81% success rate). When we run FALL against locked circuits with **Cute-Lock-Str** on ITC'99 benchmarks [44], the FALL attack fails to find any key (0% success rate) as shown in Table V. This result demonstrates that our **Cute-Lock-Str** defense is resilient against this type of attack as well.

D. Overhead Analysis

Now, we look at how much overhead **Cute-Lock-Str** adds to circuits and compare it with state-of-the-art multi-key logic locking method DK-Lock [33]. It is worth noting that DK-Lock is not fully secure since it is vulnerable against unrolling attacks such as [31] while as shown in Sections IV-B and IV-C, **Cute-Lock-Str** is secure against all existing attack surfaces.

For the overhead comparison, we focus on four key aspects: power usage, circuit area, number of cells, and number of I/O ports. To evaluate the overhead of **Cute-Lock-Str**, we use circuits from the ITC 99 [44] benchmark set. We convert .bench files to Verilog using the ABC tool [43]. Then, we use Cadence Genus with a 45nm process to synthesize and get the overhead values. We test the following three configurations:

- **Test Run 1:** 2 keys, n bits each ($k = 2, k_i = n$)
- **Test Run 2:** 4 keys, 3 bits each
- **Test Run 3:** 16 keys, 5 bits each

For DK-Lock, we use two setups: one with 10-bit keys and another where the key size changes linearly based on the inputs to the circuit (i.e., $n = k$). In Figure 4, the green dashed line shows the average of these DK-Lock setups. Looking at the figure 4, we can see that as circuits get larger, the extra power, area, and cells needed for **Cute-Lock-Str** get smaller. This means **Cute-Lock-Str** scales well for large circuits. While the smallest circuit might use about 100% more power, for the largest ones, it is less than 1%. In addition, for large circuits (*b14-b22*), both **Cute-Lock-Str** and DK-Lock do not add much overhead. But for smaller and medium-sized circuits (*b01-b11*), our **Test Run 1** and **Test Run 2** do a better job than DK-Lock. For example, for the *b06* benchmark, **Cute-Lock-Str** uses about 30% less power, area, and cells compared to DK-Lock. It is worth noting that the DK-Lock data does not include the *b20*, *b21*, and *b22* benchmarks, and this is why, in the graphs, there is a line jump.

V. Conclusion

In this paper, we introduced **Cute-Lock**, a novel time-based multi-key logic locking family with two variants: **Cute-Lock-Beh** for RTL-level behavioral locking and **Cute-Lock-Str** for netlist-level structural locking. We demonstrated the resilience of the **Cute-Lock** family against state-of-the-art oracle-guided SAT attacks incorporated in NEOS [39] and RANE [13] across a wide range of benchmarks. We showed that **Cute-Lock-Str** improves structural integrity and is resistant to DANA [9] and FALL [10] attacks. In addition, we showed that **Cute-Lock-Str** adds minimal overhead, particularly for large circuits.

Overall, the **Cute-Lock-Str** effectiveness against both oracle-guided and removal attacks, coupled with its low overhead, makes it a promising practical solution for protecting hardware IPs in the semiconductor supply chain. For future works, multi-key solutions can be explored to address other hardware security problems, such as hardware Trojan detection and mitigation.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Award No. 2245247.

References

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

[2] J. Rajendran, H. Zhang, C. Zhang, G. S. Rose, Y. Pino, O. Sinanoglu, *et al.*, "Fault analysis-based logic encryption," *IEEE Transactions on Computers*, pp. 410–424, 2013.

[3] R. Chakraborty and S. Bhunia, "Harpoon: An obfuscation-based soc design methodology for hardware protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1493–1502, 2009.

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

[5] Y. Xie and A. Srivastava, "Anti-sat: Mitigating sat attack on logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 2, pp. 199–207, 2019.

[6] 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)*, pp. 236–241, 2016.

[7] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, "Appsat: approximately deobfuscating integrated circuits," in *International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 95–100, 2017.

[8] Y. Shen and H. Zhou, "Double dip: re-evaluating security of logic encryption algorithms," in *Great Lakes Symposium on VLSI (GLSVLSI)*, pp. 179–184, 2017.

[9] N. Albartus, M. Hoffmann, S. Temme, L. Azriel, and C. Paar, "Dana: Universal dataflow analysis for gate-level netlist reverse engineering," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 4, pp. 309–336, 2020.

[10] D. Sirone and P. Subramanyan, "Functional analysis attacks on logic locking," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 936–939, 2019.

[11] M. E. Massad, S. Garg, and M. Tripunitara, "Reverse engineering camouflaged sequential circuits without scan access," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 33–40, 2017.

[12] K. Shamsi, M. Li, D. Z. Pan, and Y. Jin, "Kc2: Key-condition crunching for fast sequential circuit deobfuscation," in *2019 Design, Automation & Test in Europe Conference & Exhibition*, pp. 534–539, 2019.

[13] S. Roshanisefat, H. Mardani Kamali, H. Homayoun, and A. Sasan, "Rane: An open-source formal de-obfuscation attack for reverse engineering of logic encrypted circuits," *Great Lakes Symposium on VLSI*, 2021.

[14] M. Yasin, B. Mazumdar, J. V. Rajendran, and O. Sinanoglu, "Ttlock: Tenacious and traceless logic locking," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, (Los Alamitos, CA, USA), pp. 166–166, IEEE Computer Society, may 2017.

[15] M. Yasin, O. Sinanoglu, and R. Karri, "Provably-secure logic locking: From theory to practice," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1601–1618, ACM, 2017.

[16] A. Rezaei, Y. Shen, and H. Zhou, "Rescuing logic encryption in post-sat era by locking & obfuscation," in *Design Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 13–18, 2020.

[17] R. Afsharmazayejani, H. Sayadi, and A. Rezaei, "Distributed logic encryption: Essential security requirements and low-overhead implementation," in *Proceedings of Great Lakes Symposium on VLSI (GLSVLSI)*, pp. 127–131, 2022.

[18] A. Rezaei, A. Hedayatipour, H. Sayadi, M. Aliasgari, and H. Zhou, "Global attack and remedy on ic-specific logic encryption," in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 145–148, 2022.

[19] A. Rezaei and H. Zhou, "Sequential logic encryption against model checking attack," in *Design Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1178–1181, 2021.

[20] Y. Aghamohammadi and A. Rezaei, "Cola: Convolutional neural network model for secure low overhead logic locking assignment," in *Great Lakes Symposium on VLSI 2023 (GLSVLSI)*, pp. 339–344, 2023.

[21] A. Rezaei, Y. Shen, S. Kong, J. Gu, and H. Zhou, "Cyclic locking and memristor-based obfuscation against cycsat and inside foundry attacks," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 85–90, 2018.

[22] S. H. Khan, M. Yasin, J. Rajendran, and O. Sinanoglu, "Fsm-based obfuscation for ip protection," in *Proceedings of the Design Automation Conference (DAC)*, pp. 1–6, 2020.

[23] Z. Zhao, T. Meade, and O. Sinanoglu, "Dynamic state reflection for enhanced security in sequential logic locking," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2021.

[24] 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 *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2019.

[25] R. Karmakar, H. Kumar, and S. Chattopadhyay, "Efficient key-gate placement and dynamic scan obfuscation towards robust logic encryption," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 4, pp. 2109–2124, 2019.

[26] R. Muttaki, R. Mohammadivojdan, M. Tehranipoor, and F. Farahmandi, "Hlock: Locking ips at the high-level language," in *Design Automation Conference (DAC)*, pp. 79–84, 2021.

[27] H. Zhou, A. Rezaei, and Y. Shen, "Resolving the trilemma in logic encryption," in *International Conference on Computer Aided Design (ICCAD)*, pp. 1–8, 2019.

[28] N. Limaye, A. Chowdhury, C. Pilato, M. Nabeel, O. Sinanoglu, S. Garg, and R. Karri, "Fortifying rtl locking against oracle-less (untrusted foundry) and oracle-guided attacks," in *Design Automation Conference (DAC)*, pp. 91–96, 2021.

[29] A. Kaur, S. Saha, C. Karfa, and D. Mukhopadhyay, "Corruption exposes you: Statistical key recovery from compound logic locking," in *2022 23rd International Symposium on Quality Electronic Design (ISQED)*, pp. 1–6, 2022.

[30] A. Saha, H. Banerjee, R. S. Chakraborty, and D. Mukhopadhyay, "Oracall: An oracle-based attack on cellular automata guided logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 12, pp. 2445–2454, 2021.

[31] A. Rezaei, R. Afsharmazayejani, and J. Maynard, "Evaluating the security of efpga-based redaction algorithms," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–7, 2022.

[32] Y. Aghamohammadi and A. Rezaei, "Lipstick: Corruptibility-aware and explainable graph neural network-based oracle-less attack on logic locking," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 606–611, 2024.

[33] J. Maynard and A. Rezaei, "Dk lock: Dual key logic locking against oracle-guided attacks," in *2023 24th International Symposium on Quality Electronic Design (ISQED)*, pp. 1–7, 2023.

[34] Y.-C. Chen and S.-H. Huang, "Secure control logic design for dual key logic locking," in *2024 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pp. 800–803, 2024.

[35] Y. Kasarabada, V. Muralidharan, and R. Vemuri, "Sled: Sequential logic encryption using dynamic keys," in *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 844–847, 2020.

[36] V. S. Rathor, M. Singh, K. S. Sahoo, and S. P. Mohanty, "Gatelock: Input-dependent key-based locked gates for sat resistant logic locking," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 32, no. 2, pp. 361–372, 2024.

[37] K. Lopez and A. Rezaei, "K-gate lock: Multi-key logic locking using input encoding against oracle-guided attacks," in *2025 30th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[38] A. Valmari, *The state explosion problem*, pp. 429–528. Springer Berlin Heidelberg, 1998.

[39] K. Shamsi, "Attack tool and benchmarks." https://bitbucket.org/kavehshm/neos, 2019.

[40] Synthezza, "Fsm and logic circuit benchmarks."

[41] M. C. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the iscas-85 benchmarks: a case study in reverse engineering," *IEEE Design & Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.

[42] "Yosys open synthesis suite," 2013.

[43] Berkeley Logic Synthesis and Verification Group, "ABC: A System for Sequential Synthesis and Verification." http://www.eecs.berkeley.edu/~alanmi/abc.

[44] S. Davidson, "Itc'99 benchmark circuits - preliminary results," in *International Test Conference 1999. Proceedings (IEEE Cat. No.99CH37034)*, pp. 1125–1125, 1999.