Corruption Exposes You: Statistical Key Recovery from Compound Logic Locking

Arshdeep Kaur*, Sayandeep Saha†, Chandan Karfa* and Debdeep Mukhopadhyay†
*Indian Institute of Technology Guwahati, India
†Indian Institute of Technology Kharagpur, India
*{akaur, ckarfa}@iitg.ac.in, †{sayandeep.iitkgp@gmail.com, debdeep@cse.iitkgp.ac.in}

2022 23rd International Symposium on Quality Electronic Design (ISQED) | 978-1-6654-9466-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ISQED54688.2022.9806219

***Abstract*—Logic locking (LL) has recently gained significant attention from both VLSI and the security community for preventing intellectual property (IP) piracy and unwanted modifications of hardware circuits. While a continuous development in this area can be observed both in terms of attacks and defenses, practical application of these schemes is still challenging, as several schemes have been found vulnerable against Boolean Satisfiability and functional analysis-based attacks. In this paper, we add yet another attack strategy in the arsenal. The proposed attack is statistical and utilizes Welch's t-test to enable key recovery from logic-locked netlists assuming oracle access to an activated chip. The key fact we utilize is the variation in output corruption for different key bits. Experimental evaluation on state-of-the-art LL benchmarks ensures that the proposed strategy can be a useful aid for conventional SAT and functional analysis-based attacks on LL schemes.**

***Index Terms*—Logic Locking, Compound Logic Locking, Statistical Attack.**

I. Introduction

Logic locking (LL) is a security paradigm that has recently gained popularity as an effective means for protecting Intellectual Properties (IP) associated with digital circuits. The goal of LL is to enable key-dependent correct execution of a circuit, which is commonly realized by adding some extra logic components. The security is achieved from the enhancement of the truth-table due to the addition of this extra logic, which is controlled by the secret key. While such concepts apply to both combinational and sequential parts of a circuit, combinational LLs [10] are more widespread.

In this paper, we focus on the combinational LL schemes. The security of such LL schemes is defined by the hardness of recovering the secret key for an adversary having complete access to the locked netlist and oracle access to an activated chip. Another important aspect is the output corruption resulting from the application of a wrong key. Loosely speaking, a user would expect the chip to be completely non-functional with half of its outputs flipped on average, upon the application of a wrong key. A successful LL technique should prevent key recovery with the aforementioned adversary model, while also providing sufficient output corruption under a wrong key. However, the majority of the contemporary LL schemes fail to satisfy these two criteria, simultaneously. Early LL schemes, such as Random LL (RLL) [4], strong LL (SLL) [12] or Fault-based LL (FLL) [3], achieved high output corruptibility but failed to prevent the key recovery. An SAT-based realization of the aforementioned adversary model is sufficient for recovering keys from such high output corruption LL schemes. On the other hand, techniques like SARLock [13], SFLL [14], CASLock [7], etc. achieve security against SAT-based attacks but provide a very low output corruption which is insufficient for any practical usage. Only a few schemes, such as [5], achieve both SAT-resilience and output corruptability at the same time. The convention is, therefore, to combine RLL and some SAT-resilient schemes to realize a reasonable security margin.

The above-mentioned practice of combining two LL paradigms works fine to strike a balance between security and corruption . However, several attacks have been crafted against such compound LL schemes in the recent past [1], [2], [8]. One of the most prominent approaches is the so-called AppSAT [8], which, in principle, is able to recover all RLL keys. Furthermore, functional-analysis and partial SAT-based techniques have also been found to be practically effective [1], [9]. Almost all of these techniques, however, rely on SAT-solving. While SAT-solving is fairly efficient, thanks to modern SAT-solvers, its performance critically depends on the problem size. Although several partitioning approaches do exist for handling a large circuit in a scalable manner, the performance still gets affected by the efficiency of the partitioned formulations.

In this paper, we present a non-SAT approach for attacking compound LL schemes. Our proposal relies on statistical tests such as Welch's t-test [11]. More precisely, it links the key recovery problem with distinguishing two output distributions and utilizes Welch's t-test to eventually recover some of the key bits from a compound logic-locked design. We present a complete attack heuristic for key recovery with a few tunable parameters along with a procedure to verify the recovered key bits. The main advantage of this proposal is that it can expose certain vulnerable features of a compound LL circuit which is not directly visible through SAT solving. In other words, we recover a few key bits at a time which gives a better understanding regarding the security of individual key bits than an SAT-solver-based approach. The main reason behind the success of our technique is that the output corruption capability (or corruptibility) of different RLL-key bits differ significantly from each other, with some resulting in a very high rate of output corruption and others having a low rate of corruption. Such diversity in output corruption helps in identifying certain

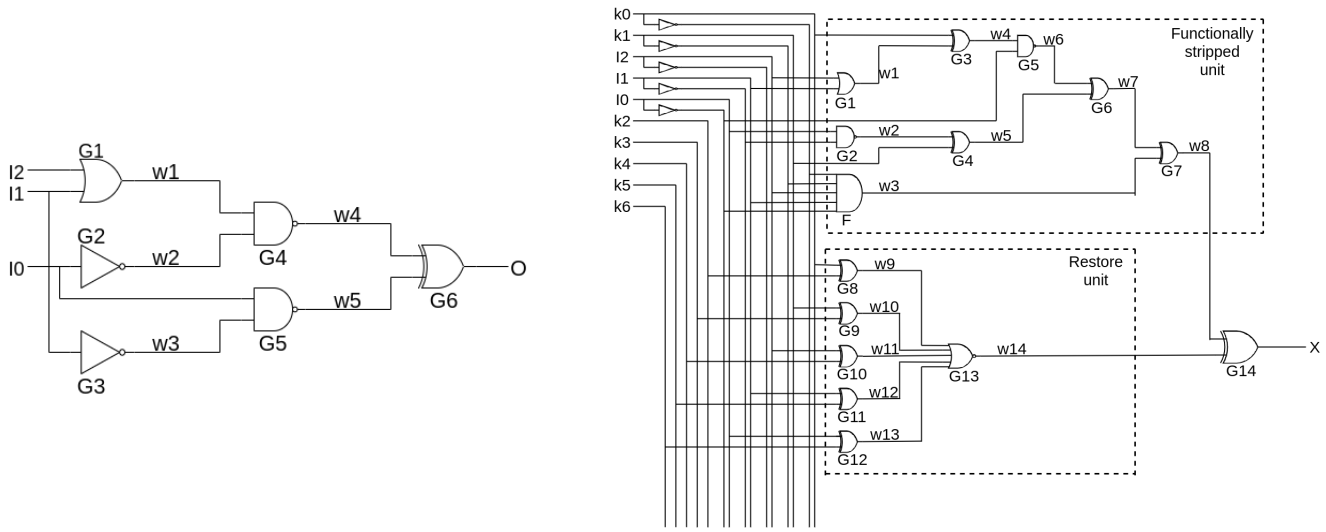

Fig. 1: Motivating example: (a) original circuit; (b) SFLL-RLL locked circuit

TABLE I: Outputs of the example circuit under different keys

I_2	I_1	I_0	O_{K_C}	O_{K_A}	O_{K_B}
0	0	0	0	0	1
0	0	1	1	1	0
0	1	0	1	1	0
0	1	1	0	0	0
1	0	0	1	1	0
1	0	1	1	1	0
1	1	0	1	0	0
1	1	1	0	0	1

key bits in a divide-and-conquer fashion using statistical tests.

One should note that we do not claim to replace the SAT-based techniques with the proposed approach. Our technique only enables the recovery of a few key bits exploiting the feature mentioned in the last paragraph regarding the output corruptibility. However, through our experiments on reasonable-sized LL benchmarks [10], we establish that such circuits indeed contain key bits with the aforementioned feature. Recovery of these bits with statistical tests can aid an SAT-solving-based approach to scale well for large practical cases. Our experimental results show that the attack can recover further locking key bits which are not unearthed by an SAT-based approach [2].

The rest of the paper is organized as follows. In the next section, we present the main intuition behind the statistical approach presented in this work through an example. Sec. III presents the necessary background on Welch's t-test. Sec. IV outlines the proposed attack algorithm in detail. The experiments on LL-benchmarks are reported in Sec. V. We conclude in Sec. VI.

II. Motivation

Most of the conventional LL techniques rely on the insertion of key-gates within a circuit netlist. The output gets corrupted upon the application of the wrong key. The SAT-hardened techniques carefully install some extra logic to achieve resilience. The RLL techniques [4] only insert key-gates at chosen positions without any carefully crafted extra logic and aim to corrupt the output as much as possible. Notably, none of the approaches clearly specify how the keys interact with the outputs in terms of corruptibility, and, in practice, it often depends upon the underlying netlist. As a result, if one considers a specific output net in the circuit, it may happen that this specific net only depends on a few key bits. More generally, due to the structure of the circuit, it may also happen that the impacts of different keys in the cone of influence of an output net are not the same on the output net.

To further illustrate the above-mentioned intuition, let us consider the netlist depicted in Fig. 1(a) containing a total of six gates, three inputs, and one output. The locked version of the circuit is shown in Fig. 1(b). The RLL keys in this locked netlist are (k_0, k_1), while the SFLL keys are marked as $(k_2, k_3, k_4, k_5, k_6)$. SFLL is applied on the top of the RLL keys in this case. The correct key for the entire circuit is given as $(0, 0, 0, 0, 1, 1, 0)$. Let us now consider the truth-table (ref. Table. I) of the locked circuit for three different keys including the correct key. The correct key is denoted as K_C. The first wrong key is given as $K_A = (0, 0, 0, 1, 0, 1, 1)$, while the second one (denoted as K_B) is generated by flipping the second bit of K_A from the left (i.e. the bit k_1). One may clearly observe that the number of wrong outputs with key K_A is fairly low, while with K_B it becomes sufficiently high [1]. This clearly shows the non-uniformity in the output corruption capabilities of different key-bits. Such variation in output corruption can be fatal for a locked circuit as by simply guessing the value of a highly corrupting key-bit (or maybe a small group of such key bits) and observing the output corruption distribution, the correct value of the target

[1]The wrong output is marked as red in the table.

key-bit can be recovered. More precisely, if key-bit results in high output corruption and flipping its value results in a low output corruption , then with a high probability the flipped value is a correct key bit. Although this might seem a little optimistic assumption, we found that this indeed works over large practical benchmarks tested in this work. However, to extend this simple observation for larger circuits, we require a proper quantification of 'high' and 'low' output corruption . In order to do that we interpret the output corruption statistically and take the help of a well-known statistical test described in the next section.

III. Output corruption and t-test

As outlined in the last section, the main crux of our approach lies in statistically distinguishing the corruptibility resulting from two different key bits. To achieve this, we utilize Welch's t-test which can determine if two given distributions are different in terms of their means. Given two samples M_0 and M_1 under test, with M_0 (resp. M_1) having a sample mean μ_0 (resp. μ_1) and sample variance s_0^2 (resp. s_1^2), the t-test statistic t and degrees of freedom v is calculated as:

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}} \tag{1}$$

$$v = \frac{(\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1})^2}{\frac{(\frac{s_0^2}{n_0})^2}{n_0 - 1} + \frac{(\frac{s_1^2}{n_1})^2}{n_1 - 1}} \tag{2}$$

Here $n_0 = |M_0|$ and $n_1 = |M_1|$. We can consider $|t|$ as corruptibility. Large $|t|$ values in the test indicate that the null hypothesis (i.e. $\mu_0 = \mu_1$) should be rejected, whereas the small values of $|t|$ indicates the opposite. Usually, for sufficiently large sample size, $|t| > 4.5$ denotes that the null hypothesis should be rejected with 99.9999% confidence. Therefore, ± 4.5 is often taken as a threshold in t-test [6].

In our attack algorithm (outlined in the next section), we perform a t-test between the output distributions generated from the correct key and a random wrong key on random input sets. Ideally, the t-test should always give a high value in this setting as the correct output distribution is indeed different from the output distribution due to a random wrong key, even if the wrong key differs in a single key bit. However, the statistical distinguishability can only be ensured if the entire truth table corresponding to an output bit is taken into consideration. In practice, the distinguishability (and the high value in the t-test) depends upon the probability of output corruption due to a wrong key. In case the corruptibility is low, the distribution cannot be distinguished from the distribution due to a correct key as we can only enumerate a partial truth table even for a reasonable-sized circuit[2]. On the other hand, high corruptibility ensures distinguishability. While this entire operation can also be performed simply based on output corruption, what the t-test specifically provides us is a well-established threshold, which is otherwise very difficult to determine while analyzing complex circuits. Based on this (t-test assisted) distinction between 'high' and 'low' corruptibility, in the next section, we outline our attack algorithm.

[2]Truth-tables are typically exponential in size over the number of inputs.

Algorithm 1 Procedure STATISTICAL-ATTACK

Input: $\mathcal{C}$, $\mathcal{O}$, $\mathcal{T}$
Output: K_{Rec}: List of recovered key-bits

$M \leftarrow$ `NumIpBits`($\mathcal{C}$) ▷ Get number of input bits
$S \leftarrow$ `NumKeyBits`($\mathcal{C}$) ▷ Get number of key bits
$N \leftarrow$ `NumOpBits`($\mathcal{C}$) ▷ Get number of output bits
$K_{Rec}[S] \leftarrow NULL$
while ($\neg End$) **do**

// Phase-1
$K_{rnd} \leftarrow$ `RandSelectKey`($\mathcal{T}$) ▷ Randomly select $\mathcal{T}$ keys.
$Tmat[\mathcal{T}][N] \leftarrow 0$
for each $k^t \in K_{rnd}$ **do**
$I_{rnd} \leftarrow$ `RandSelectIP`($\mathcal{T}$) ▷ Randomly select $\mathcal{T}$ input vectors.
$o^t \leftarrow \mathcal{C}(k^t, I^t)$ ▷ Simulate the netlist
$oc^t \leftarrow \mathcal{O}(I^t)$ ▷ Oracle query
$ttest_{kt} \leftarrow$ `ttest`(o^t, oc^t)
$Tmat[k^t][:] \leftarrow ttest_{kt}$ ▷ Store t-test results for all key in K_{rnd}
end for

// Phase-2
$HighImp \leftarrow \varnothing$; $HighImpKey[\mathcal{T}] \leftarrow \varnothing$
for each column n of $Tmat$ **do** ▷ For each output bit $0 \leqslant n < N$
$K_n \leftarrow$ `FindHighT`($Tmat[:][n]$) ▷ Keys with $|t| \geqslant 4.5$
$cnt_n \leftarrow |K_n|$ ▷ Save highly corrupted outputs and corresponding keys
if $cnt_n \geqslant Th_{hl}$ **then**
$HighImp \leftarrow HighImp \cup \{n\}$
$HighImpKey[n] \leftarrow HighImpKey[n] \cup K_n$
end if
end for

// Phase-3
for each $h \in HighImp$ **do**
$k_{net} \leftarrow$ `IPConeSearch`(h) ▷ Find key nets in the input cone of h
$H_k \leftarrow HighImpKey[h]$
for each $kb \in k_{net}$ **do**
$cnt1_{kb} \leftarrow$ `CountOne`(H_k, kb) ▷ How many times kb is 1 in H_k
if $cnt1_{kb} \geqslant P_{Th}$ **then**
$K_{Rec}[kb] \leftarrow 0$ ▷ Set kb as 0
else
$cnt0_{kb} \leftarrow (1 - cnt1_{kb})$
if $cnt0_{kb} \geqslant P_{Th}$ **then**
$K_{Rec}[kb] \leftarrow 1$ ▷ Set kb as 1
end if
end if
end for
end for

// Verification
$K_{Rec} \leftarrow$ `VerifyKey`(K_{Rec}) ▷ Verify the consistency of the found keys

// Next Iteration
if "No new keys in this iteration" **then** ▷ No key found
$\mathcal{T} \leftarrow 2 \times \mathcal{T}$
end if

// Termination
if No new key found in two consecutive iterations **then**
$End \leftarrow 1$
else
$End \leftarrow 0$
end if
end while
Return K_{Rec}

IV. Statistical Approach for Robustness Evaluation

In this section, we present the attack on the LL circuit exploiting the variation of corruptibility among different key-bits. Strictly speaking, the approach presented here is a heuristic, which depends on few user-controllable parameters. It only requires oracle access to an activated chip for the correct

TABLE II: Distribution of output set oc^t when I_x is queried to the oracle $\mathcal{O}$ and o^t when I_x with a wrong key is simulated with the circuit $\mathcal{C}$

	output distribution oc^t				output distribution o^t			
$I_{rnd}[1]$	$oc^t[1][1]$	$oc^t[1][2]$	...	$oc^t[1][N]$	$o^t[1][1]$	$o^t[1][2]$	...	$o^t[1][N]$
$I_{rnd}[2]$	$oc^t[2][1]$	$oc^t[2][2]$	...	$oc^t[2][N]$	$o^t[2][1]$	$o^t[2][2]$	...	$o^t[2][N]$
...	...	...	...	...	...	...	...	...
$I_{rnd}[\mathcal{T}]$	$oc^t[\mathcal{T}][1]$	$oc^t[\mathcal{T}][2]$	...	$oc^t[\mathcal{T}][N]$	$o^t[\mathcal{T}][1]$	$o^t[\mathcal{T}][2]$	...	$o^t[\mathcal{T}][N]$

outputs. The circuit netlist is also required for simulation and extraction of the input cones. However, unlike so-called functional analysis attacks [9], we do not need to explore any non-trivial properties from the circuit topology.

The attack-heuristic is outlined in Algorithm 1 and Algorithm 2. Algorithm 1 takes the circuit netlist $\mathcal{C}$, the corresponding oracle $\mathcal{O}$, and a parameter $\mathcal{T}$ as inputs and returns a set of key-bits discovered (K_{Rec}). The $\mathcal{T}$ is the size of the random input vector set and random key set in each iteration of the heuristic. For notational convenience, we define K_{Rec} as a dictionary indexed by the key-nets (or key-bits) and storing their corresponding values. K_{Rec} is initialized with $NULL$ for all the indices. The heuristic begins (Phase-1 of Algorithm 1) with selecting a set (K_{rnd}) of $\mathcal{T}$ random keys of size S bits each. For each of these keys, a random set of input vectors I_{rnd} (with $|I_{rnd}| = \mathcal{T}$) is generated. Next, the correct outputs corresponding to I_{rnd} are generated from the oracle, and the potentially corrupted outputs are generated from the circuit. Finally, a t-test is performed between the correct output set and the corrupted output set. The structure of the correct and faulty output sets (denoted as oc^t and o^t, respectively) are described in Table. II, where each output is an N-bit vector (i.e. the number of outputs of the circuit is N). The t-tests among these outputs are performed point-wise (that is, an independent test for each output bit). The outcomes of these tests are stored in a $\mathcal{T} \times N$ matrix denoted as $Tmat$. Each row of this matrix stores N t-test scores corresponding to a random key k^t. Each column, on the other hand, stores test scores corresponding to an output bit.

In the next step (Phase-2 in Algorithm 1), we iterate $Tmat$ column-wise and find output nets with maximum corruptibility. This is realized by selecting output bits for which most of the keys (decided based on a threshold Th_{hl}) in the matrix $Tmat$ have scored a high t-value ($|t| \geqslant 4.5$). The rationale behind this step is that high t-scores of an output for several random keys indicate that the output is highly impacted by certain key bits. Therefore, focusing on these output nets gives us more information regarding keys compared to relatively less corruptible outputs. The threshold Th_{hl} is a parameter that is to be decided experimentally. It is found that $Th_{hl} = 6$ works universally well for all of our test cases.

After finding the high-corruptibility output nets, in Phase-3 we iterate over these output nets and extract their corresponding input cones. The goal is to find out all the key-nets (denoted as k_{net}) which influence a particular output net h under consideration. Next, we look at the random keys (H_k) resulting in high corruptibility for h. This can be figured out from the keys stored in Phase-2. In these keys of size S, we specifically focus on the bits corresponding to the nets in k_{net}. If any of these key-nets are assuming a specific value (0 or 1) in H_k with high probability (P_{Th}), we store the negation of that value as a candidate's correct key-bit. P_{Th} is set to 1 in our experiments. This step is motivated by the fact that if a specific net is always assuming a fixed value which eventually results in high corruptibility, flipping this value may reduce the corruptibility. Although the efficacy of this heuristic depends on the structure of the netlist and placement of the key-gates to a large extent, we found this to be effective for several key-nets in large benchmarks. However, there might be false alarms. One potential cause can be that a key-net to be flipped have a very low impact on the corresponding output-net. In that case, even though the output is highly corrupted, it is due to the other key-nets. Therefore, even if the target key-net is already set to its correct value, we may end up flipping it. Another case occurs when the value of the key-net in H_k is varying with reasonable probability. For such cases, our heuristic cannot decide anything and leaves the key-net as undiscovered.

Algorithm 2 Procedure VERIFY-KEY

Input: K_{Rec}
Output: K_{Rec} with wrong keys removed
for each $kb \in K_{Rec}$ **do**
  if ($kb \neq NULL$ & (kb already not verified)) **then**
    $k_r \leftarrow$ `RandSelectKey(1)` ▷ Randomly select a key
    $k_r \leftarrow$ `FixVerifiedKeys(`k_r`)` ▷ Fix already verified key-bits in k_r
    $k_x \leftarrow$ `FixKeyBit(`kb`)`
    $k_y \leftarrow$ `FixKeyBit(`$\neg kb$`)`
    $I_r \leftarrow$ `RandSelectIP(`$\mathcal{T}$`)`
    $ttest_{k_x} \leftarrow$ `ttest(`$\mathcal{C}(k_x, I^r)$, $\mathcal{O}(I^r)$`)`
    $ttest_{k_y} \leftarrow$ `ttest(`$\mathcal{C}(k_y, I^r)$, $\mathcal{O}(I^r)$`)`
    Get the output bit h which recovered kb
    if ($|ttest_{k_x}[h]| < 4.5$) $\wedge$ ($|ttest_{k_y}[h]| \geqslant 4.5$) **then**
      Verify Success
    else
      Verify Failure
      $K_{Rec}[kb] \leftarrow NULL$
    end if
  end if
end for
Return K_{Rec}

To get rid of the false alarms corresponding to the first odd case mentioned above, we propose a verification step for the values found. The idea is to use another set of t-tests for a random input set. Here we construct two keys k_x and k_y from a random key k_r. First, we fix all the key-nets with already verified values in k_r. Next, k_x is formed by fixing the value of the key-net to be verified to its found value. k_y is the same as k_x, except the value corresponding to the key-net to be verified. With this setting, we perform two t-tests with the oracle outputs. If one of the tests (with k_x) results in low corruptibility and the other (with k_y) in high corruptibility, we can safely assume the found key-net value to be a correct one.

This step is performed for each key-net for which the heuristic till Phase-3 can decide some value. This step is illustrated separately in Algorithm. 2 (the `VerifyKey` operation from Algorithm. 1). One should notice that the sole purpose of this step is to decide if the key-net under verification strongly impacts the corresponding output or not. In case it fails the verification, it is set to $NULL$ by the verification heuristic.

The final step of the heuristic is to decide the terminating condition. While this can be done in several ways, it is found that terminating after two consecutive failed iterations (i.e. if no keys are recovered in two consecutive iterations) worked reasonably well. Also, we double the size of the test and the key set ($\mathcal{T}$) after each iteration. The initial value of $\mathcal{T}$ is set as 25. It varied up to 1600 in our experiments.

A. Discussion

It is worth mentioning that the attack heuristic critically depends upon the threshold parameters and the size of the key and input sets ($\mathcal{T}$). It is, however, found that setting these parameters to some reasonable values from their respective domains works well over the entire benchmark set. Experimental pieces of evidence regarding these parameter settings are presented in Sec. V, which confirms that we can establish such a generic operating point. It is not necessary to keep random size of the key sets same as that of input sets. We leave this to the hands of experimenter. Another important question in this context is whether the proposed heuristic can recover both RLL and SAT-resilient keys for a compound logic-locked netlist. In principle, our approach should work irrespective of key type. However, in practice, SAT-resilient keys result in extremely low corruptibility and, therefore, the approach should not detect them in the present setting of parameters. Ideally, if some output corruption is observed for an SAT-resilient key, Th_{hl} is set to an even lower value, and $\mathcal{T}$ is large there is a fair chance that some SAT-resilient keys get captured. In our experiments, however, we did not observe any such pathological case. Therefore, it is reasonable to comment that the approach only recovers certain weak RLL keys from a compound logic-locked netlist. It can aid SAT-based attacks on large benchmarks by statistically pruning the key-space thus making the attacks more efficient in terms of time and memory complexity.

V. Experiments and Results

In this section, we present experimental validation of our claims. The benchmark-suite chosen is the one from the 2019 CSAW competition for breaking locked circuits with compound LL [10]. The benchmarks are robust enough that 40+ hackers taking part in LLC 2019 for more than three months could not break them. They remain to be broken till recent time [1]. Details of our benchmarks are provided in Table. III. The experiments were run on a Linux system with 12 Intel-Core i7-8700 CPUs running at 3.20GHz. The overall outcomes of our experiments are summarized in Table. IV, V, and VI. To understand the importance of the verification step, we performed two sets of experiments – one with and one without the verification. Furthermore, to understand the interaction of our proposed methodology with SAT-based attacks, we performed two different sets of experiments. In the first set of experiments, denoted as ($M1$), we only perform the proposed statistical attack initialized with random key and input sets as described in the Algorithm 1. In the second experiment ($M2$), some of the key inputs are assigned with their correct valuations found through the partial SAT attack in [2][3]. Both $M1$ and $M2$ are performed for two cases – with and without verification and the number of keys found is reported (ref. Table. IV).

TABLE III: Description for CSAW 2019 Logic Locking Contest benchmarks

Benchmark	#in	#out	#keys	#gates	#RLL	#SFLL
Small	522	512	80	15995	40	40
Medium	767	757	120	24008	60	60
Large	1452	1445	160	36584	80	80

As it can be observed from Table. IV, the $M1$ experiment (without any key initialization) recovers several keys for all three benchmarks. Some of these keys were also found by the partial SAT attack, and, therefore, we exclude them from the calculation. However, the role of the verification step is crucial here, as several wrong keys can be observed without the verification step. Although the introduction of the verification step reduces the total number of recovered keys but the reliability increases significantly. The most interesting observation for $M2$ is that it recovers certain keys which could not be recovered with the SAT-based approach in [2]. The new keys identified (on top of SAT-based attack [2]) is almost the same in $M1$ and $M2$. This establishes the potential utility of our approach for assisting SAT-based attacks.

A. Finding the Optimal Parameter Settings

As already pointed out in Sec. IV, three parameters determine the key recovery success of the proposed heuristics. The first threshold parameter used is Th_{hl} which decides the high corruptible output-nets. Keeping this value very low will impact the runtime, whereas setting this to a large value will result in insufficient exploration over the potentially recoverable keys. While we found $Th_{hl} = 6$ most effective for all our benchmarks, for large circuits, setting $Th_{hl} = 10$ also recovers most of the key bits recovered in the former case. Another important parameter in our attack is P_{Th}, which is set to 1. The impact of this parameter can be observed in Table. V and VI for experiment $M1$. Each of these tables displays the number of keys found with different settings of P_{Th} and $\mathcal{T}$. The number of wrong key suggestions is also denoted within brackets for each case. While the choice of P_{Th} shows some impact for the cases without verification, the heuristic becomes highly robust to any small change to this parameter value after adding the verification step. The most important parameter, however, is the (initial) input and key set

[3]We note that the keys from [2] are RLL keys only.

TABLE IV: The overall results of our experiments.

BM	#SAT Found Keys	M1						M2					
		Without Verification			With Verification			Without Verification			With Verification		
		#Tot. Key	#Wrong Key	# New Keys	#Tot Key	#Wrong Key	#New Keys	#Tot Keys	#Wrong Keys	#New Keys	#Tot Keys	#Wrong Key	#New Keys
Small	20	16	2	6	9	0	5	13	4	9	5	0	5
Medium	50	22	5	2	11	0	1	4	2	2	2	0	2
Large	45	84	24	36	37	0	14	17	0	17	13	0	13

TABLE V: Trend Analysis for method M1 without verification. Each cell shows the number of keys recovered with the count of wrong keys within brackets.

P_{Th} = 1	$\mathcal{T} = 50$	$\mathcal{T} = 100$	$\mathcal{T} = 400$	$\mathcal{T} = 800$
Small	4 (0)	9 (0)	9 (0)	9 (0)
Medium	13 (3)	12 (2)	11 (0)	11 (0)
Large	50 (8)	37 (0)	36 (0)	36 (0)
P_{Th} = 0.9	$\mathcal{T} = 50$	$\mathcal{T} = 100$	$\mathcal{T} = 400$	$\mathcal{T} = 800$
Small	4 (0)	9 (0)	9 (0)	9 (0)
Medium	12 (2)	11 (1)	12 (1)	12 (2)
Large	40 (4)	41 (3)	36 (1)	36 (1)
P_{Th} = 0.8	$\mathcal{T} = 50$	$\mathcal{T} = 100$	$\mathcal{T} = 400$	$\mathcal{T} = 800$
Small	7 (1)	13 (1)	9 (0)	9 (0)
Medium	10 (0)	13 (3)	13 (2)	13 (0)
Large	64 (17)	38 (1)	44 (6)	36 (1)

TABLE VI: Trend Analysis for method M1 with verification. Each cell shows the number of keys recovered with the count of wrong keys within brackets.

P_{Th} = 1	$\mathcal{T} = 50$	$\mathcal{T} = 100$	$\mathcal{T} = 400$	$\mathcal{T} = 800$	$\mathcal{T} = 1600$
Small	3 (0)	4 (0)	9 (0)	8 (0)	9 (0)
Medium	8 (0)	10 (0)	9 (0)	9 (0)	10 (0)
Large	27 (0)	13 (0)	34 (0)	33 (0)	36 (0)
P_{Th} = 0.9	$\mathcal{T} = 50$	$\mathcal{T} = 100$	$\mathcal{T} = 400$	$\mathcal{T} = 800$	$\mathcal{T} = 1600$
Small	4 (0)	9 (0)	7 (0)	9 (0)	9 (0)
Medium	5 (0)	9 (0)	10 (0)	10 (0)	10 (0)
Large	30 (1)	21 (0)	29 (0)	34 (0)	36 (0)
P_{Th} = 0.8	$\mathcal{T} = 50$	$\mathcal{T} = 100$	$\mathcal{T} = 400$	$\mathcal{T} = 800$	$\mathcal{T} = 1600$
Small	9 (0)	4 (0)	9 (0)	9 (0)	9 (0)
Medium	5 (0)	9 (0)	9 (0)	10 (0)	10 (0)
Large	33 (1)	30 (0)	36 (0)	34 (0)	36 (0)
P_{Th} = 0.7	$\mathcal{T} = 50$	$\mathcal{T} = 100$	$\mathcal{T} = 400$	$\mathcal{T} = 800$	$\mathcal{T} = 1600$
Small	4 (0)	4 (0)	9 (0)	9 (0)	9 (0)
Medium	5 (0)	9 (0)	9 (0)	10 (0)	10 (0)
Large	27 (0)	32 (0)	34 (0)	36 (0)	36 (0)
P_{Th} = 0.6	$\mathcal{T} = 50$	$\mathcal{T} = 100$	$\mathcal{T} = 400$	$\mathcal{T} = 800$	$\mathcal{T} = 1600$
Small	4 (0)	9 (0)	8 (0)	9 (0)	9 (0)
Medium	6 (1)	7 (0)	10 (0)	10 (0)	10 (0)
Large	34 (1)	34 (0)	27 (0)	36 (0)	36 (0)

size ($\mathcal{T}$), which dictates the number of recovered keys. As it can be observed from Table. V and Table. VI, $\mathcal{T} = 800$ is a reasonable value for all the cases considered in this paper. Therefore, these values ($Th_{hl} = 6$, $P_{Th} = 1$ and $\mathcal{T} = 800$) constitute the best operating point for our tool.

VI. Conclusion and Future Directions

Logic locking is a problem with immense practical value. However, ensuring security for LL is a difficult task, especially for the most practical cases, where SAT-resilient LLs are combined with RLL to achieve a practical output corruptibility. Attacking such compound schemes is technically feasible, but still computationally costly for sufficiently large circuits with SAT formulations. In this paper, we presented an SAT-free statistical attack for compound LL schemes which will assist the SAT-based attacks, and can also be used independently. From a broader perspective, this work also shows the feasibility of statistical attacks on LL and will open up a new avenue for such a class of attacks.

References

[1] Zhaokun Han, Muhammad Yasin, and Jeyavijayan (JV) Rajendran. Does logic locking work with EDA tools? In 30th USENIX Security Symposium (USENIX Security 21), pages 1055–1072. USENIX, Aug 2021.

[2] Melbin John, Aadil Hoda, Ramanuj Chouksey, and Chandan Karfa. Sat based partial attack on compound logic locking. In AsianHost2020, pages 1–6, 12 2020.

[3] JV Rajendran, H. Zhang, C. Zhang, G. Rose, Y. Pino, O. Sinanoglu, and R. Karri. Fault analysis-based logic encryption. IEEE Transactions on Computers, 64:410–424, 02 2015.

[4] Jarrod A. Roy, Farinaz Koushanfar, and Igor L. Markov. Ending piracy of integrated circuits. Computer, 43(10):30–38, 2010.

[5] Akashdeep Saha, Sayandeep Saha, Siddhartha Chowdhury, Debdeep Mukhopadhyay, and Bhargab B. Bhattacharya. Lopher: Sat-hardened logic embedding on block ciphers. In 57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020, pages 1–6. IEEE, 2020.

[6] Tobias Schneider and Amir Moradi. Leakage assessment methodology. In Proceedings of 17th International Workshop on Cryptographic Hardware and Embedded Systems (CHES), pages 495–513, Saint-Malo, France, Sep 2015. Springer.

[7] Bicky Shakya, Xiaolin Xu, Mark Tehranipoor, and Domenic Forte. CAS-Lock: A security-corruptibility trade-off resilient logic locking scheme. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020(1):175–202, Nov 2019.

[8] Kaveh Shamsi, Meng Li, Travis Meade, Zheng Zhao, David Z. Pan, and Yier Jin. Appsat: Approximately deobfuscating integrated circuits. In 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pages 95–100, 2017.

[9] Deepak Sirone and Pramod Subramanyan. Functional analysis attacks on logic locking. In 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pages 936–939, 2019.

[10] B. Tan, R. Karri et. al. Rahman, Zhou, Gadfort, Plaks. Benchmarking at the frontier of hardware security: Lessons from logic locking. CoRR, abs/2006.06806, 2020.

[11] Bernard L Welch. The generalization of 'Students's' problem when several different population variances are involved. Biometrika, 34(1-2):28–35, 1947.

[12] 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.

[13] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan J V Rajendran, and Ozgur Sinanoglu. Sarlock: Sat attack resistant logic locking. In 2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pages 236–241, 2016.

[14] Muhammad Yasin, Abhrajit Sengupta, Mohammed Thari Nabeel, Mohammed Ashraf, Jeyavijayan (JV) Rajendran, and Ozgur 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, New York, NY, USA, 2017. ACM.