Evaluating the Security of Logic-Locked Probabilistic Circuits

Michael Zuzak, *Graduate Student Member, IEEE*, Ankit Mondal, and Ankur Srivastava, *Senior Member, IEEE*

***Abstract*—Logic locking is a design-for-security scheme to thwart attacks by an untrusted foundry. Prior work exposed the vulnerability of logic-locked circuits using Boolean Satisfiability (SAT). While these attacks are effective against deterministic circuits, they cannot unlock probabilistic/approximate designs, which have become increasingly popular. In this work, we expand SAT-style attacks to locked circuits with probabilistic behavior. We propose *StatSAT*, an attack incorporating statistical techniques into the SAT attack to unlock probabilistic designs. We then propose a countermeasure called High Error Rate Keys (HERK) to thwart StatSAT and other attacks on probabilistic circuits. HERKs leverage high error wires, caused by probabilistic behavior, to hide the correct key under stochastic noise.**

***Index Terms*—StatSAT Attack, High Error Rate Keys (HERK), Probabilistic Computing, Logic Locking, Untrusted Foundry**

I. Introduction

Integrated circuit (IC) fabrication is frequently outsourced to unaffiliated and untrusted third parties. This prompts security concerns as fabrication facilities can reverse engineer (RE), overproduce, and counterfeit fabricated ICs [1]. As a result, IC designers have shown interest in mitigation schemes to protect design secrets from malicious activity during fabrication [2].

Logic locking (or logic obfuscation) is a design-for-security scheme that obfuscates a design's function by adding auxiliary logic gates and inputs, called keys [2]. The resulting circuit behaves correctly only when the correct key is applied. Otherwise, errant outputs are produced for a deterministic set of inputs. This protects an IC from an untrusted foundry by hiding design function and rendering illegally overproduced copies of an IC unusable. In response, attacks leveraging Boolean satisfiability (SAT) were developed to unlock early locking schemes [3], [4]. This prompted the development of a flurry of SAT-resilient defense techniques [2].

During this time, approximate and probabilistic circuitry have become increasingly relevant for 3 reasons. First, relaxing requirements for functional correctness in circuits reduces the cost of manufacturing, verification, and test [5]. Second, such an approach provides substantial energy savings [6]. Finally, the popularity of big-data, Internet-of-Things, and machine learning applications have widened the scope of approximate computations due to their error-tolerant nature [7].

The uncertainties of IC design at highly scaled-down technology nodes often leads to probabilistic behavior. This is due to a variety of factors including 1) reduced noise margin from voltage scaling, which reduces energy consumption, and 2) larger process variations [5]. The errors arising from the use of probabilistic computing elements are dynamic and transient in nature [8]. They have a certain probability of occurring and can occur anywhere in the circuit.

A. Contributions

In this work, we demonstrate that an untrusted foundry can steal the intellectual property (IP) of locked probabilistic chips. We develop an attack strategy based on Boolean satisfiability (SAT) to find the key of logic-locked probabilistic circuits. The existing SAT attack [3], [4] is good for deterministic circuits, but not probabilistic ones because the latter exhibits inconsistent behavior. The work in [9] proposes a Probabilistic SAT (PSAT) attack to handle this, however, its success is limited to low error rates and its scalability with the number of circuit outputs is poor. We propose **StatSAT**, an attack that is applicable to circuits with higher error than considered by PSAT [10]. We then evaluate the security of the StatSAT attack and develop a locking mechanism called High Error Rate Keys (HERK) to thwart such an attack and protect probabilistic IP.

II. Preliminaries

A. Attack Surface

We consider the adversary outlined in [3], [4]. This model considers an attacker with access to: 1) a netlist of the locked IC and 2) an unlocked version of the IC (i.e. an *oracle*). We consider a circuit error model where each logic gate in the circuit has a certain error probability ϵ_g, which is the probability that a gate outputs the inverse of what it should have. This is realistic from the perspective of non-CMOS gates [9] and dynamic errors [8]. The attacker must find the correct key for the locked circuit despite the probabilistic oracle.

B. Prior Work

The work in [9] shows that probabilistic behavior in an oracle misguides a SAT attacker, tricking them into using a wrong output for a distinguishing input (DI). This causes a SAT attack to fail even at negligible error levels. The authors then propose a new SAT attack, called Probabilistic SAT (PSAT), where the oracle is queried multiple times for each DI (rather than once). If the most common output pattern is *dominant* (defined in [9]), it is assumed correct. Otherwise, an output pattern is sampled with a probability equal to its frequency of occurrence. This attempts to mitigate the inconsistency of the oracle. While the PSAT attack performs better than conventional SAT, it lacks scalability (see Sec. IV).

This work was supported by the the ARCS Foundation, the National Science Foundation grant 1642424, and the Air Force Office of Scientific Research grant FA9550-14-1-0351.

M. Zuzak and A. Srivastava are with the Department of Electrical and Computer Engineering at the University of Maryland, College Park, MD 20742 USA. Email: {mzuzak, ankurs}@umd.edu

A. Mondal is with Intel Corporation, Hillsboro, OR 97124 USA

III. Attack Formulation

The core of the StatSAT attack is the same as the standard SAT attack - a SAT solver finds DIs to limit the key space until only correct keys remain. In this section, we describe the extra steps taken by StatSAT to unlock probabilistic IP.

A. Deviation from the SAT Attack

The SAT attack [3], [4] assumes a deterministic oracle and queries it once in every iteration with a DI to obtain the correct output. In our work, the oracle behaves probabilistically. Because its output is probabilistically approximate (i.e. inconsistent), the output obtained upon applying an input just once cannot be assumed to be the correct output pattern.

Let us define the *Bit Error Ratio* (BER) for an output of a circuit for a given input as the probability that the output is erroneous for that input. Erroneous means that the output of the probabilistic oracle differs from a deterministic version of it. To get around such inconsistent oracle behavior, we apply the same input (DI) multiple times to get multiple output patterns, as was done in [9]. However, we do not consider the oracle's output patterns as a whole. Instead, we take an average of all patterns for each of the circuit's outputs. Thus, for any DI, the oracle is queried not once, but multiple times to obtain the *signal probabilities* of each output bit.

Stated mathematically, let N be the number of POs in the circuit. If the probabilistic oracle is sampled N_s times for an input X, let $Y^{(1)}$, $Y^{(2)} \dots Y^{(N_s)}$ be the observed output patterns, where each $Y^{(j)} \in \{0,1\}^N$. The signal probability at the i^{th} output of the oracle for input X is $P_i^Y = \langle Y_i \rangle = \frac{1}{N_s}\sum_{j=1}^{N_s}(Y^{(j)})_i$ with every $P_i^Y \in [0,1]$ and $i = 1, 2, \dots N$. Rounding all these signal probabilities to 0 or 1 may not yield the correct output because one or more output bits may have higher than 50% BER for an input depending on gate sensitization [11]. As such, let us now discuss how the signal probabilities P_i^Y obtained from the oracle can be provided to the SAT solver. The standard SAT attack uses responses from the oracle to constrain SAT-CNF expressions. For any I/O pair (X, Y) fed to the solver, future iterations ensure that *all* bits of Y are satisfied for input X when DIs are obtained. However, we relax this constraint by not specifying all output bits. This avoids specifying erroneous bits in an output pattern.

B. Output Uncertainty

The signal probabilities P^Y of the oracle response tells us about the certainty of an output bit. Let us define the *uncertainty* U associated with the signal probabilities as:

$$U_i = min(P_i^Y, 1 - P_i^Y) \qquad i = 1 \dots N \tag{1}$$

P_i^Y values close to 0.5 are associated with higher uncertainties in the output bit. Because a Boolean SAT solver accepts only '0' or '1' as variable values, the P_i^Y would have to be converted to '0' or '1'. We opt to leave outputs with high uncertainties **unspecified** in the constraint because these have a high BER. For any DI, the oracle output signal probability vector P^Y could be translated to a binary output vector Y as:

$$Y_i = \begin{cases} round(P_i^Y) & \text{if } U_i \leq U_\lambda \\ \text{x (unspecified)} & \text{if } U_i > U_\lambda \end{cases} \tag{2}$$

where U_λ is a threshold value of uncertainty above which we refrain from specifying the rounded signal probability value.

C. Estimation of Output BERs with DIs

In the previous subsection, we proposed using the uncertainty values U as indicators of high output BERs to avoid providing wrong DIPs to the SAT solver. While this method is effective when the BER at a particular output is low or moderate (close to 0.5), it would not prevent the recording of wrong outputs (Y) if the BER happens to be large. Specifically, if the BER at the i^{th} output is larger than $1 - U_\lambda$, then $U_i < \ U_\lambda$ and Y_i is set to $round(P_i^Y)$. This is wrong. We discuss how to prevent this by considering the possibility of having a large BER at any output for a given DI.

Let us assume that the attacker is aware of the error probability ϵ_g at each logic gate in the circuit[1]. With this knowledge, the attacker can *roughly* estimate the BER at each of the outputs of the circuit with a given input using Boolean Difference Calculus [11]. However, it is not possible to know the BERs *exactly* without the correct key, which is unknown to the attacker. The attacker is in possession of the netlist of the locked circuit that has both primary inputs and key inputs. To determine the output BER for a particular input vector, the attacker can set the signal probabilities to 0 or 1.

This forces us to estimate the BER at each output. We do this with satisfying keys. Each iteration of the SAT attack prunes the key space such that the keys remaining in the solution space at the start of an iteration satisfy all DIs for previous iterations. The BERs from satisfying keys are then averaged to estimate the BER $E \in [0,1]^N$ for that DI. Stated formally, let $(X^1, Y^1) \dots (X^{m-1}, Y^{m-1})$ be the distinguishing I/O pairs (DIP) for the first $m-1$ iterations and X^m be the m^{th} DI. We find a certain number of satisfying keys (N_{satis}) for these $m - 1$ pairs. Using these keys, the output BER E^m of the locked circuit for X^m can be estimated using Boolean Difference Calculus [11]. This identifies any output bits with high error rates, determining whether to pass off the rounded signal probabilities to the CNF formula. The translated binary output vector for DI X^m is defined by

$$Y_i^m = \begin{cases} round(P_i^Y) & \text{if } U_i \leq U_\lambda,\ E_i^m \leq E_\lambda \\ \text{x (unspecified)} & \text{otherwise} \end{cases} \tag{3}$$

where, if the estimated BER E_i^m for the i^{th} output crosses threshold E_λ, we do not specify its value irrespective of the corresponding signal probability P_i^Y. For example, if the signal probability values for a 4-bit output are $P^Y = (0.85, 0.38, 0.20, 0.77)$ and the BER estimate is $E = (0.21, 0.28, 0.27, 0.34)$, then the uncertainty is $U = (0.15, 0.38, 0.2, 0.23)$. If $U_\lambda = 0.25$ and $E_\lambda = 0.30$, then $Y =$ 1x0x is the output vector, with bits 2 and 4 unspecified.

D. Multiple Instances of SAT-CNF Formulas

To this point, we focused on how we can prevent the CNF formula from recording wrong output bits for distinguishing input patterns by holding back specific output bits from the

[1] In [10], we show that this assumption is unnecessary and can be relaxed.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3104270, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

SAT solver. Doing so keeps the correct key within the solution space. However, after a certain number of SAT iterations, the SAT solver may no longer be able to eliminate keys because of the unspecified bits in the output vectors of the DIs. This is evident from the **repetition of a DI** in 2 consecutive iterations. Querying the oracle again with the same DI is unlikely to help since it would yield the same signal probability vector P^Y as in the prior iteration, hence, the same binary output vector Y^m. Next, we describe how to get around this dead-end.

Duplication: Let us call the set of CNF formulas and the distinguishing I/O pairs collectively an *instance* of the SAT formulation. To proceed with the attack, we propose creating a duplicate (clone) of the SAT instance to be able to consider both possibilities of an unspecified bit. Thus, if the DI X^{m+1} is a repeat, we let one of the previously unspecified bits of the binary output vector Y^m be represented differently in Y^{m+1} for these 2 SAT instances - one considers a value of '0' for that bit and the other considers a '1'. They differ only in bit position Y^{m+1} that we were forced to specify. Because a bit position can have a correct value of either '0' or '1', one of these 2 instances is guaranteed to hold correct I/O pairs.

The output bit specified during SAT duplication is chosen by the uncertainty U and estimated BER E. Among all unspecified bits, we choose the one with the largest uncertainty $U_i > U_\lambda$ because that output wire is likely to have a high error. If no $U_i > U_\lambda$, we choose the output bit with the largest estimated BER E_i for the same reason. The original and duplicated instances differ in their Y^{m+1} at bit index j_{dup}:

$$j_{dup} = \begin{cases} j = argmax_i\ (U_i) & \text{if } U_j > U_\lambda \\ argmax_i\ E_i & \text{otherwise} \end{cases} \tag{4}$$

After the original SAT instance has duplicated into 2 instances, the attack proceeds, finding separate DIs for future iterations. These instances do not interact with one another.

Force Proceed: The duplicated instances continue finding DIs. Whenever any of them halt due to another repeated DI, the instance is further duplicated per the above process. However, to avoid exponentiation in the number of SAT instances, we limit the total number of SAT instances to a certain value, N_{inst}. Once this limit has been reached, SAT instances can no longer duplicate. Instead, we forcefully proceed by specifying one of the unspecified bits of Y^m, hoping that this provides sufficient information to the SAT solver to trim down a few wrong keys and not repeat the DI again. To do so, we specify the output bit which had the least estimated BER (hence the least risky) by rounding the observed signal probability. Thus, if $X^{m+1} = X^m$ and the instance cannot duplicate, let $j_{fp} = argmin_i(E_i\ \mathbb{I}(Y_i^m = \text{x}))$ where indicator function $\mathbb{I}(f) = 1$ if f is *true*, and ∞ otherwise. Then, $Y_{j_{fp}}^{m+1} = round(P_{j_{fp}}^Y)$.

Evaluation: Each instance carries out the process of eliminating wrong keys on its own until it cannot find any more DIs. It then returns one key which satisfies all prior DIPs or states that they cannot be satisfied by any key (UNSAT). When an instance becomes UNSAT, the space occupied by it is "freed-up" to allow other instances to duplicate themselves if necessary. After each satisfiable instance returns a key (at most N_{inst} keys), each key is evaluated by comparing the response of the oracle to the locked circuit fitted with the obtained keys. This evaluates the efficacy of a key by looking for discrepancies between oracle and keyed netlist responses.

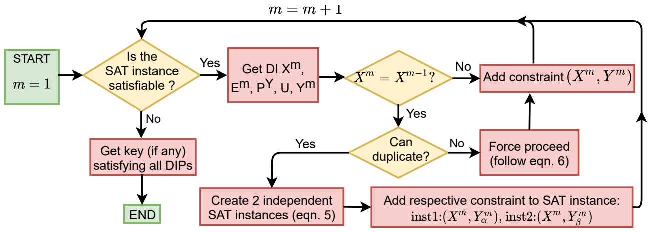
$m = m + 1$
START
$m = 1$
Is the SAT instance satisfiable ?
Yes
No
Get DI X^m, E^m, P^Y, U, Y^m
$X^m = X^{m-1}$?
No
Yes
Add constraint (X^m, Y^m)
Get key (if any) satisfying all DIPs
END
Can duplicate?
Yes
No
Force proceed (follow eqn. 6)
Create 2 independent SAT instances (eqn. 5)
Add respective constraint to SAT instance: inst1:(X^m, Y_α^m), inst2:(X^m, Y_β^m)

Fig. 1: StatSAT attack procedure for one SAT instance.

Let $X_1 \ldots X_j \ldots X_{N_{eval}}$ be randomly chosen input vectors. Let P^{Y_j} and $P^{Y_j}(K)$ denote the output signal probabilities for input X_j in the oracle and the unlocked circuit (with key K). Let $FM(K)$ be the figure of merit for an obtained key K:

$$FM(K) = \frac{1}{N}\sum_{i=1}^{N} \max_j |P^{Y_j} - P^{Y_j}(K)|\ ,\ j = 1..N_{eval} \tag{5}$$

The key with the *smallest* FM is chosen as the best key. In Fig. 1, the full StatSAT attack is outlined for one SAT instance.

IV. Attack Results

We implemented the StatSAT attack[2] by modifying the SAT-attack framework developed and made open-source by [3], [9]. To evaluate StatSAT, we attacked SFLL-locked benchmark circuits [12]. In our attack, N_{inst}, U_λ and E_λ are attacker specified parameters. In addition to the figure of merit $FM(K)$ in Eqn. 5, we calculate the average of the signal probability differences for the best key K^*. This is an average hamming distance in the signal probability domain (i.e. $HD(K) = \frac{1}{N_{eval}}\sum_{j=1}^{N_{eval}} \frac{1}{N}||P^{Y_j} - P^{Y_j}(K)||_1$, where $j = 1..N_{eval}$ and $||.||_1$ is L1-norm). $HD(K)$ is better than $FM(K)$ for measuring the closeness between the behavior of the oracle and the unlocked circuit because it averages over all N_{eval} patterns, whereas $FM(K)$ takes the maximum and is better for measuring discrepancies between a small number of keys. We now present experiments to evaluate StatSAT.

(A) First, we show how the number of instances N_{inst} required to find *the* correct key increases with gate-level error ϵ_g. Table I aggregates attack results for several values of ϵ_g. The gate-level error probability ϵ_g (labeled A, B, ...) was varied and N_{inst} was incremented from 1 in powers of 2 until the correct key was found. For all benchmarks, the smallest ϵ_g in Table I was chosen such that any increase required more than 1 SAT instance. For each ϵ_g, we started the attack with $U_\lambda = 0.25$, $E_\lambda = 0.30$. If the attack did not find a key, we decremented both values. From Table I, notice that StatSAT fully unlocked each benchmark. Moreover, the correct key was found for higher ϵ_g by increasing the allowed SAT instances.

(B) Comparison with PSAT: We assess the performance of the PSAT attack [9] by launching it 20 times on several benchmarks. Table II aggregates the number of times that PSAT ran to completion and found a key. Notice that StatSAT, unlike PSAT, could always find the correct key for those circuits (see Table I), demonstrating StatSAT's superiority.

[2] The code for StatSAT can be found at github.com/mzuzak/StatSAT-attack

Bench+Lock	ϵ_g (in %)	Avg. BER	Max. BER	N_{inst}	$\|K\|$	$HD(K^*)$
c3540 SFLL-HD	1.25 (A)	0.241	0.834	1	1	0.0192
	1.50 (B)	0.269	0.852	8	7	0.0200
	1.75 (C)	0.286	0.865	16	16	0.0207
	2.00 (D)	0.302	0.850	64	64	0.0209
c7552 SFLL-HD	2.00 (A)	0.173	0.784	1	1	0.0122
	2.25 (B)	0.182	0.784	8	1	0.0125
	2.50 (C)	0.189	0.760	16	5	0.0125
	3.00 (D)	0.201	0.758	16	16	0.0128
b14 SFLL-HD	0.50 (A)	0.0520	0.628	1	1	0.0087
	0.75 (B)	0.0668	0.724	4	1	0.0100
	0.80 (C)	0.0750	0.760	4	3	0.0103
	0.85 (D)	0.0759	0.776	16	16	0.0112
b15 SFLL-HD	0.2 (A)	0.0200	0.378	1	1	0.00583
	0.4 (B)	0.0311	0.546	4	4	0.00796
	0.5 (C)	0.0489	0.580	8	1	0.00874
	0.6 (D)	0.0498	0.608	16	10	0.00939

TABLE I: N_{inst} required to find the correct key for varied ϵ_g. Attack parameters: $N_s = 500$, $N_{satis} = 100$, $N_{eval} = 2000$.

Circuit	c880			b15		c3540	b14	c7552
ϵ_g (in %)	1.0	1.5	2.0	0.1	0.2	1.25	0.5	2.0
Valid PSAT Runs	20	5	0	20	0	0	0	0
Valid StatSAT Runs	20	20	20	20	20	20	20	20

TABLE II: # of runs (of 20) that the correct key was located.

V. High Error Rate Keys (HERK)

To continue, we propose a new logic locking scheme, called High Error Rate Keys (HERK), to counter the StatSAT attack. HERKs are key gates inserted on high error rate wires in probabilistic circuits. Such an approach hides the correct key value (i.e. correct HERK function) under high probabilistic noise present at the insertion point. This makes it extremely hard for a SAT solver to infer the correct key (i.e. correct function) of any logic locking influenced by HERK function. We show that this leads to an exponential increase in StatSAT runtime for a linear increase in the number of inserted HERKs.

A. Overview of High Error Rate Keys (HERK)

The proposed HERK structure is shown in Figure 2. To implement the construction in this figure, a designer must first identify locations in the circuit that exhibit a high error rate (i.e. wires where probabilistic behavior makes wrong signal values likely). The error rate at each location in the circuit can be calculated via Boolean Difference Calculus [11]. However, the calculated error rate on a wire is dependent on the considered input pattern. Ideally, HERKs will be inserted at high error rate locations for many inputs. We propose the following approach for HERK insertion.

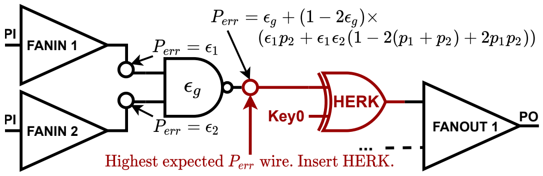
$P_{err} = \epsilon_g + (1 - 2\epsilon_g)\times$
$(\epsilon_1 p_2 + \epsilon_1 \epsilon_2 (1 - 2(p_1 + p_2) + 2p_1 p_2))$
PI
FANIN 1
$P_{err} = \epsilon_1$
ϵ_g
HERK
PI
FANIN 2
$P_{err} = \epsilon_2$
Key0
FANOUT 1
PO
...
Highest expected P_{err} wire. Insert HERK.

Fig. 2: Sample HERK insertion into probabilistic circuit.

First, a set of random inputs is selected and Boolean Difference Calculus is used to calculate the BER at each location in the (correctly-keyed) circuit for each input. The BER is averaged over all inputs for each location. The wires with the highest average BER likely exhibit error for a large subset of inputs, making them ideal for HERK insertion. A XOR gate is inserted on the highest average BER wires, driven by the wire and an added key input. A XOR gate is used due to its minimal error masking compared to other gates (i.e. any single input error guarantees an output error for a XOR gate). This added gate is called a *High Error Rate Key (HERK)*. We propose inserting many HERKs in this way for security.

In probabilistic circuits, high error wires often exhibit error rates that exceed 50% for specific input patterns [11]. This can be observed in our benchmark circuits in Table I where the worst-case BER among only outputs (not all wires) exceeds 50% in 15 of 16 circuits. HERKs inserted at these locations exhibit extremely high error rates. This is by design as HERKs aim to hide their key value under probabilistic noise on a high error rate wire. Thus, in any oracle circuit, the actual functionality of the HERK is extremely hard to infer. This obfuscates not only the key value (correct function) of the HERK gate, but also the key value (correct function) for any locking structures that rely/influence HERK functionality. This property of HERKs is shown to ensure StatSAT resilience.

B. Evaluation of HERK Attack Resilience

The StatSAT attack presents a security threat for locked probabilistic IP. We propose HERKs to counter this threat. To show this, remember that StatSAT assigns a "don't-care" state for any primary output (PO) whose BER exceeds a user-specified threshold for a given DI. This eliminates the possibility of using a wrong value for a PO, which would exclude the correct key from consideration. However, an unspecified PO also cannot be used to eliminate wrong keys either. HERKs exploit this by inserting key gates at high error locations for a set of inputs. Whenever any of these high error inputs are used as a DI during the StatSAT attack, the HERK gate has high output error. This error propagates to one or more POs, resulting in the BER threshold being exceeded and the PO being specified as "don't-care". StatSAT can handle this in 3 ways that each degrade performance.

1) **StatSAT Forks:** High error inputs to a HERK leads to unspecified POs. An unspecified PO cannot be used to eliminate keys. Hence, StatSAT cannot eliminate keys for the HERK. If few/no keys are eliminated, StatSAT forks. Thus, an exponential number of SAT instances in the number of POs affected by HERKs are required.
2) **StatSAT Cannot Fork:** If the StatSAT instance limit (N_{inst}) is reached, further forking is prohibited. StatSAT force proceeds by guessing the most likely PO value. If wrong, the correct key is eliminated from consideration.
3) **Low Quality CNF Clauses:** Defined POs can still be used to eliminate wrong keys. However, because HERK-related POs are unspecified, the resulting CNF cannot be used to infer the key value for the HERK (or any HERK-influenced logic). This reduces the number of keys that can be eliminated for a DI, requiring more DIs be found.

1) HERK Attack Resilience: We consider 3 ways that an attacker could attempt to tune StatSAT (or the locked circuit) to thwart HERKs. HERKs resist each mitigation strategy.

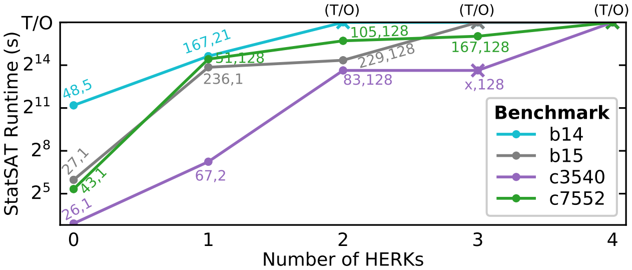

Fig. 3: Runtime required by StatSAT to unlock each HERK-secured benchmark circuit. Numbers beside each data point correspond to *“SAT iterations, SAT instances”* for each attack.

1) **Raise BER Threshold**: The BER threshold can be increased such that the BER at any PO will not exceed it. Thus, high error inputs do not produce an unspecified PO. However, a high BER threshold risks incorrectly specifying a PO, excluding the correct key.
2) **Ignore HERKs (Removal [13])**: Because HERKs correspond to high error points in the circuit, one could assume that ignoring them is acceptable. However, error is input dependent. Nodes may be high error for some inputs and critical for others. HERKs greatly impact function for low-error input patterns, which are likely critical [14], thus they cannot be removed.
3) **Tune Allowable SAT Instances**: To avoid excluding the correct key due to SAT instance limits, one could increase this limit. However, the number of SAT instances is exponential in the number of HERKs. Such an approach quickly becomes infeasible. Conversely, to avoid exponentiation, the instance limit could be kept low. However, once the limit is reached, StatSAT “*force proceeds*”. If a wrong value is assumed, the correct key is eliminated.

2) Experimental Analysis: We implemented HERKs with the existing conventional locking (SFLL [12]) in each StatSAT benchmark in Table I. To do so, we selected 1000 random input patterns and used Boolean Difference Calculus [11] to estimate the error on each wire in the circuit. A HERK (XOR gate) was then inserted, driven by the wire containing the highest average error and an added key input. A separate set of 1000 inputs were used for each HERK insertion and no 2 HERKs were inserted at the same location. We considered only the lowest gate error (ϵ_g) version of each circuit. This maps to the lowest PO BER, hence, it is the hardest for HERKs to secure. HERK security at lower ϵ_g implies security at higher ϵ_g for a circuit.

We launched the StatSAT attack against each benchmark containing 1-4 HERKs. The uncertainty and BER thresholds (U_λ and E_λ) for StatSAT were set to the highest value still able to recover the correct key for the baseline (i.e. 0 HERK) circuit. This ensures that increasing the BER threshold cannot be used to bypass HERK-based locking as the correct key would not be recovered. For each run, if StatSAT did not recover a correct key with its uncertainty and BER threshold, we lowered both in 1% increments and relaunched the attack. This continued until a correct key was found, the SAT instance limit (N_{inst}=128) was reached, or a 30 hour timeout was reached. The resulting StatSAT runtime, SAT iterations, and SAT instances required to unlock each benchmark are in Fig. 3. An ‘x’ for any data point indicates that StatSAT did not find a correct key. We make 3 observations from these results.

1) No correct key could be located in 30 hours when 4 HERKs were applied. This indicates that strong StatSAT resilience can be achieved with only a few HERKs.
2) The 3 mechanisms by which StatSAT responds to a HERK-induced unspecified PO can be observed for each benchmark. 1) The number of SAT instances increases exponentially in the number of HERKs. 2) Once the SAT instance limit is reached, StatSAT cannot find the key (due to wrong POs assumed for suppressed forks). 3) The SAT iterations to locate the key increases with HERKs.
3) StatSAT runtime increases exponentially in the number of HERKs. This is the key takeaway. It indicates that a small number of HERKs causes infeasible StatSAT runtime.

VI. Conclusion

In this work, we proposed StatSAT, a Boolean satisfiability attack against logic-locked probabilistic circuits. We then developed High Error Rate Keys (HERK), a logic locking technique that resists both StatSAT and other prominent attacks. HERKs use high error points in probabilistic IP to hide the correct key (i.e. correct function) under stochastic noise.

References

[1] M. Rostami, F. Koushanfar, and R. Karri, “A primer on hardware security: Models, methods, and metrics,” Proceedings of the IEEE, vol. 102, no. 8, pp. 1283–1295, 2014.

[2] A. Chakraborty et al., “Keynote: A disquisition on logic locking,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2019.

[3] P. Subramanyan, S. Ray, and S. Malik, “Evaluating the security of logic encryption algorithms,” in IEEE International Symposium on Hardware Oriented Security and Trust, 2015, pp. 137–143.

[4] M. El Massad, S. Garg, and M. V. Tripunitara, “Integrated circuit (ic) decamouflaging: Reverse engineering camouflaged ics within minutes.” in NDSS, 2015, pp. 1–14.

[5] K. V. Palem et al., “Sustaining moore’s law in embedded computing through probabilistic and approximate design: retrospects and prospects,” in international conference on Compilers, architecture, and synthesis for embedded systems, 2009, pp. 1–10.

[6] J. Han and M. Orshansky, “Approximate computing: An emerging paradigm for energy-efficient design,” in IEEE European Test Symposium, 2013, pp. 1–6.

[7] C.-Y. Chen et al., “Exploiting approximate computing for deep learning acceleration,” in Design, Automation & Test in Europe. IEEE, 2018.

[8] T. Rejimon, K. Lingasubramanian, and S. Bhanja, “Probabilistic error modeling for nano-domain logic circuits,” IEEE Transactions on Very Large Scale Integration, vol. 17, no. 1, pp. 55–65, 2008.

[9] S. Patnaik et al., “Spin-orbit torque devices for hardware security: From deterministic to probabilistic regime,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2019.

[10] A. Mondal, M. Zuzak, and A. Srivastava, “Statsat: a boolean satisfiability based attack on logic-locked probabilistic circuits,” in ACM/IEEE Design Automation Conference, 2020, pp. 1–6.

[11] N. Mohyuddin, E. Pakbaznia, and M. Pedram, “Probabilistic error propagation in a logic circuit using the boolean difference calculus,” in Advanced Techniques in Logic Synthesis, Optimizations and Applications. Springer, 2011, pp. 359–381.

[12] A. Sengupta et al., “Truly stripping functionality for logic locking: A fault-based perspective,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2020.

[13] M. Yasin et al., “Removal attacks on logic locking and camouflaging,” IEEE Transactions on Emerging Topics in Computing, 2017.

[14] K. He, A. Gerstlauer, and M. Orshansky, “Controlled timing-error acceptance for low energy idct design,” in Design, Automation & Test in Europe. IEEE, 2011, pp. 1–6.