GenUnlock: An Automated Genetic Algorithm Framework for Unlocking Logic Encryption

Huili Chen, Cheng Fu, Jishen Zhao, Farinaz Koushanfar
University of California, San Diego
{huc044,cfu,jzhao,farinaz}@ucsd.edu

ABSTRACT

Logic locking inserts additional key gates to the original circuit for protecting the intellectual property (IP) of modern integrated circuits (ICs). Prior works have identified the vulnerability of logic locking to satisfiability (SAT)-based attacks. However, SAT attacks are ineffective on circuits with SAT-hard structures. In this paper, we propose GenUnlock, the first genetic algorithm-based logic unlocking attack framework addressing the above limitation of SAT attacks. GenUnlock formulates logic unlocking (i.e., identifying the correct keys) as a combinatorial optimization problem and tackles it using genetic algorithms (GAs). Multiple key sequences form the individuals in the population and undergo the following main operations: circuit fitness evaluation, population selection, crossover, and mutation. The key sequences with high fitness scores 'survive' the selection and are transformed into the offspring. GenUnlock's evolutionary process of key searching features high scalability, exploration efficiency, and parallelizable fitness evaluation.

We take an Algorithm/Software/Hardware co-design approach to optimize GenUnlock's runtime overhead. More specifically, GenUnlock (i) Pipelines each computation stage by automatically constructing auxiliary circuitry for constraints checking, sorting, crossover, and mutation; (ii) Employs *hardware emulation* on programmable hardware for accelerating circuit fitness evaluation. We perform a comprehensive evaluation of GenUnlock's performance on various benchmarks and demonstrate that GenUnlock achieves up to 1014.1× speedup and is 3974.3× higher energy efficiency compared to the state-of-the-art SAT attacks for logic unlocking.

1 INTRODUCTION

Integrated circuits (ICs) are indispensable for various real-world applications ranging from domestic electronics, autonomous vehicles to medical devices and deep learning systems [6, 17]. The supply chain of modern ICs involves the participation of multiple parties, thus is vulnerable to potential attacks such as IC piracy, overproduction, and counterfeiting [13, 19]. Logic locking and circuit camouflaging have been suggested as obfuscation techniques to protect the Intellectual Property (IP) of ICs. IC camouflaging aims to prevent layout-level reverse engineering (RE) attacks by adding dummy contacts/cells to the standard gates [15]. Logic locking intends to protect the functionality of the circuit by inserting additional *key gates* to the original circuit [20, 21] such that the output is correct only when the decryption key is applied. Figure 1 illustrates an example of the XOR-based logic locking scheme.

The security of logic locking has been discussed in prior works. Satisfiability (SAT)-based attacks and their variants can break obfuscated circuits with the state-of-the-art logic locking methods. Traditional SAT attacks work by eliminating incorrect keys with distinguishing input patterns (DIPs) found by SAT solvers [18]. However, SAT-based attacks have the following drawbacks: (i) They are not scalable to large benchmarks since the size of the DIP constraints increases over iterations; (ii) The computation of SAT solving is difficult to parallelize; (iii) They cannot activate circuits with SAT-hard designs (e.g., an internal 'and-tree' structure) since a DIP can only eliminate a single incorrect key in this case [18]. Developing an efficient and effective logic unlocking methodology is challenging since the approach is desired to: (i)*Be generic to negate arbitrary logic encryption techniques and unknown circuit structures*; (ii) *Provide the trade-off between the attack success rate and runtime overhead*; (iii) *Demonstrate scalability on large circuits*.

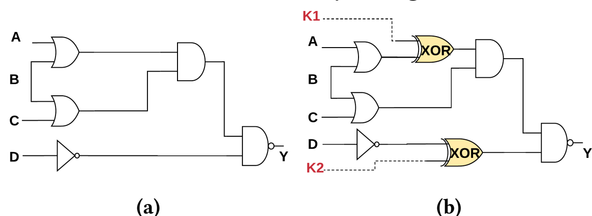

Figure 1: Example of XOR-based logic locking. The encrypted circuit (b) yields consistent outputs as the original one (a) only when the two-bit key K_1K_2 is set to $2'b00$.

We propose GenUnlock, the first genetic algorithm-based framework for logic unlocking. GenUnlock takes the netlist of the encrypted circuit and the corresponding black-box accessible active IC as its inputs. A set of feasible key sequences are returned as the outputs of GenUnlock. Our framework consists of two main phases: *(i) Training data generation.* We first generate the training dataset by querying the active IC and collecting corresponding outputs. *(ii) Key evolution.* A set of keys are instantiated as individuals in the population and *'evolve'* with GA training. GenUnlock provisions the trade-off between the unlocking accuracy and execution time. Compared to the existing SAT attacks, GenUnlock can efficiently find *approximate* keys that yield correct outputs with high probability. The approximate keys can be used to attack fault-tolerant applications such as deep learning systems and block-chain mining.

GenUnlock is devised based on an Algorithm / Software / Hardware co-design approach. We deploy a *diversity-guided* genetic algorithm to ensure the stable convergence of the GA. Furthermore, GenUnlock incorporates *hardware emulation* and *pipelining* as optimization techniques to accelerate GenUnlock's computation on FPGAs. To the best of our knowledge, GenUnlock is the first logic unlocking framework that provides hardware design and optimization. This paper makes the following contributions:

- **Demonstrating the first genetic algorithm-based key searching method to invalidate logic locking.** GenUnlock's diversity-guided key evolutionary facilitates the exploration of key space, thus is more scalable and generic than the traditional SAT attacks.

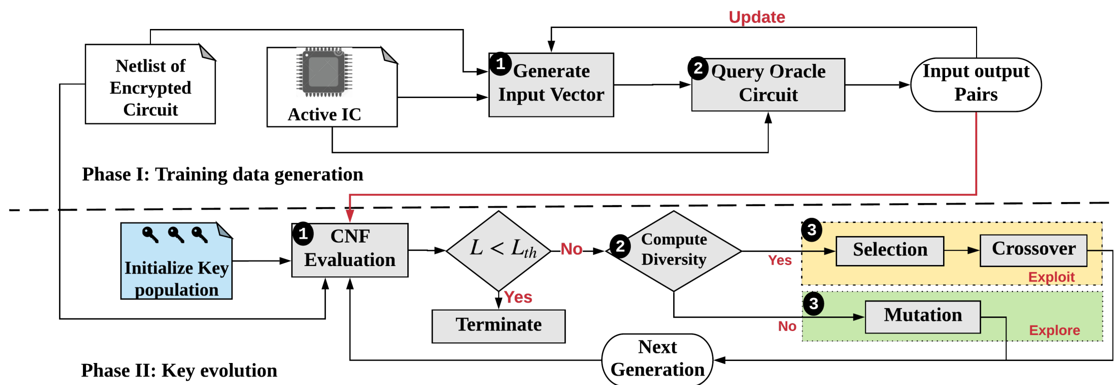

Figure 2: Global flow of GenUnlock framework for logic unlocking.

- **Enabling logic unlocking with performance trade-off.** GenUnlock allows the adversary to explore the trade-off between attack effectiveness and runtime, thus revealing an undiscovered threat on fault-tolerant applications.
- **Leveraging an Algorithm/Software/Hardware co-design approach to devise an efficient attack scheme.** GenUnlock provides a scalable and high-performance hardware design that advances the deobfuscation speed to a higher level. Our hardware design incorporates various optimization techniques including computation pipelining and circuit emulation on programmable hardware.
- **Investigating the performance of GenUnlock on various circuits.** We conduct an extensive evaluation of GenUnlock and compare the results with the state-of-the-art SAT attacks to corroborate its efficacy and efficiency.

GenUnlock opens a new axis for the growing research in hardware security by shedding light on the potential of deploying genetic algorithms to address challenging problems in the hardware domain. Our approach is alternative to the existing SAT-based attacks and provide a new attack dimension for (approximately) unlocking the encrypted circuit. GenUnlock provides a flexible attack mechanism that yields a set of feasible keys for circuit unlocking with improving effectiveness over time. Furthermore, the returned population after convergence (i.e., key sequences) enables *ensemble-based circuit evaluation* that exhibits superior unlocking performance compared to one when a single key is used.

2 GENUNLOCK OVERVIEW

Figure 2 illustrates the global flow of GenUnlock. We discuss the attack assumptions in Section 2.3. GenUnlock framework consists of two stages: (i) Offline pre-processing phase that generates training data for GA; and (ii) Key searching phase that performs key evolution. The one-time pre-processing phase is performed via oracle access while the key searching phase is accelerated using FPGA.

Phase I: Training data generation. This phase consists of the following two tasks:

❶ **Generate input vectors.** Given the netlist of the encrypted circuit, GenUnlock crafts input vectors and filter the ones that result in the same circuit outputs when different keys are applied.

❷ **Query active IC.** The remaining input patterns from step 1 are then used to query the active IC. The collected input/output (IO) pairs form the training dataset for GenUnlock's logic unlocking.

Phase II: Key evolution. Once the training data for the target circuit is generated in Phase I, GenUnlock performs three subroutines during the key evolution phase as shown in the bottom of Figure 2:

❶ **Circuit fitness evolution.** Analogous to natural selection, the key sequences with higher fitness scores are maintained and transformed to offsprings at each iteration. The fitness of each key is evaluated by the ratio of output matching on the training dataset when the specific key is applied. We convert the netlist representation to conjunctive normal form (CNF) to facilitate fitness evaluation.

❷ **Population diversity computation.** GenUnlock separates genetic operations into two groups (*'exploitation'* or *'exploration'*) and determines which branch to take depending on the population diversity. Since key sequences are binary-valued in logic locking, we use the dispersion (i.e., variance) of the population as the measurement of diversity.

❸ **Diversity-guided GA execution.** GenUnlock applies genetic operations on the current population (i.e., key sequences) based on the computed diversity. As opposed to traditional GAs that perform all genetic operations in each iteration, GenUnlock's *dynamic, diversity-aware* GA execution demonstrates better convergence.

2.1 Motivation

Prior works on circuit deobfuscation heavily rely on external SAT solvers to find distinguishing input patterns and eliminate incorrect keys [1, 16, 18]. However, the existence of *SAT-hard* problems [7] makes it challenging to apply SAT attacks in these scenarios. For instance, the SAT attack proposed in [18] fails to unlock the *c*2670 and *c*6880 benchmark since these circuit contains an internal 'and-tree' structure. To address the above limitation, we propose GenUnlock framework that is able to attack circuits with SAT-hard structures.

Real-world Use Cases. Existing works focus on unlocking the circuit with perfect accuracy, thus may incur prohibitive runtime overhead to break large circuits (Section 5). Here, we want to emphasize that *fast, approximate* decryption of the target circuit can be more threatening than slow, full decryption. This is particularly true for *fault-tolerant* applications. Let us consider block-chain mining as a real-world example where the signature of the cryptocurrency is extracted from AES and hashing operations [10] on the hardware miner. The resulting signature is continuously checked against the pre-defined template to determine whether the cryptocurrency is legitimate. As such, it is sufficient for the user to find a key that yields

correct outputs with high probability in order to obtain financial benefits. Emerging ASIC accelerators for Deep Neural Networks (DNN) are also inherently fault-tolerant, which has been exploited for parameter quantization or pruning.

2.2 Notations and Metrics

Problem Statement and Notation. Our objective is to design a systematic methodology for unlocking arbitrary unknown, encrypted circuit. We denote the original unlocked circuit and its encrypted version as C_o and C_e. The primary input, output vector and the encryption key of the circuit are denoted as $\vec{I} \in \mathbb{B}^M$, $\vec{O} \in \mathbb{B}^N$, and $\vec{K} \in \mathbb{B}^k$, respectively. The functionality of the circuit is represented by the following *deterministic mapping*: $C_o(\vec{I}) = \vec{O}$ and $C_e(\vec{I},\ \vec{K}) = \vec{O}$. The quality of a decryption key is quantified by the output fidelity (OF) that defines the probability of the output vector of C_e being consistent with the one of C_o given any input $\vec{I}$:

$$OF(\vec{K};\ C_o,\ C_e) = \underset{\forall \vec{I} \in \mathbb{B}^M}{Prob}\ [\ C_e(\ \vec{I},\ \vec{K}) = C_o\ (\vec{I})]. \tag{1}$$

We consider logic unlocking as successful if the OF of the identified key is higher than the attacker-defined threshold $OF > (1 - \epsilon)$. Note that two different key sequences might result in the same circuit behavior (i.e., same mapping C_e). We define that $\vec{K}_1$ and $\vec{K}_2$ belong to the same *equivalence class* of keys [18] if the condition $C_e(\vec{I}, \vec{K}_1) = C_e(\vec{I}, \vec{K}_2)$ is satisfied for any $\vec{I} \in \mathbb{B}^M$.

Performance Metrics. We use *effectiveness* and *efficiency* as two main metrics to assess the performance of a logic unlocking scheme. These two metrics are quantified by the attack success rate (defined in Equation (1)) and the execution time, respectively. GenUnlock, for the first time, provides the trade-off between effectiveness and efficiency by generating a set of keys with evolving quality over time. In addition, we also use *resource consumption* as a metric to evaluate our hardware design. A detailed, quantitative analysis of these metrics is given in Section 5.

2.3 Threat Model

We make the following assumptions about GenUnlock framework:
(i) The attacker has black-box access to the active IC. We assume that the adversary can purchase the unlocked circuit from the market and obtains oracle access to it. As a result, the attacker is able to query the active IC with arbitrary input challenges and observe the corresponding outputs, which is the basis of GenUnlock's training data generation phase (Phase I in Figure 2).

(ii) The attacker knows the netlist of the encrypted circuit. We assume the attacker can reverse engineer the netlist of C_e from a physical circuit by performing depackaging, delayering and imaging [9]. The obtained netlist is converted to CNF and used in circuit fitness evaluation (Phase II in Figure 2).

3 GENUNLOCK METHODOLOGY

Prior works have identified that there might be more than one correct keys to unlock the given circuit [18]. GenUnlock leverages this fact and processes multiple keys representing different equivalence classes in each iteration, thus features higher efficiency for space exploration. Note that GenUnlock is oblivious of the underlying encryption schemes used by the defender, thus is genetic and applicable to arbitrary ICs. In the following of this section, we detail the two key phases of GenUnlock framework.

3.1 Training Data Generation

Algorithm 1 outlines the steps of the one-time, offline training data generation phase in GenUnlock. Ground-truth input/output pairs (S_I, S_O) are generated using oracle access to the active IC.

Algorithm 1 Training Data Generation.

INPUT: Active circuit (C_o) with oracle access; Netlist of target encrypted circuit (C_e); Number of desired IO pairs (T); Size of primary inputs (M) and the encryption key (k).

OUTPUT: A set of input/output pairs (S_I, S_O) as the training data for genetic algorithms.

1: Initialization: $S_I \leftarrow \emptyset$, $S_O \leftarrow \emptyset$, $i \leftarrow 0$.
2: **while** $i < T$ **do**
3: $\quad \vec{I} \leftarrow generate_random_inputs(M)$
4: $\quad \vec{K}_1, \vec{K}_2 \leftarrow generate_random_keys(k)$
5: $\quad \vec{O}_1 \leftarrow C_e(\vec{I},\ \vec{K}_1), \vec{O}_2 \leftarrow C_e(\vec{I},\ \vec{K}_2)$
6: $\quad$ **if** $\vec{O}_1 \neq \vec{O}_2$ **then**
7: $\quad\quad i \leftarrow i + 1$
8: $\quad\quad S_I \leftarrow add_element(S_I,\ \vec{I})$
9: $\quad\quad \vec{O} \leftarrow C_o(\vec{I})$
10: $\quad\quad S_O \leftarrow add_element(S_O,\ \vec{O})$
11: **Return:** Obtained IO pairs (S_I, S_O) for GA training.

Note that a naive implementation of challenge-response pairs collection is not desirable since the resulting training data may not be able to distinguish different key sequences in Phase II. To alleviate this concern, we estimate the distinguishing capability of each input ($\vec{I}$) by comparing the outputs of the encrypted circuit (C_e) when two different random keys are applied. Only inputs that result in different outputs are maintained in the final training set (line 4-10 in Algorithm 1). The attacker can obtain a more accurate approximation of the input's distinguishing capability using more keys at the cost of higher computation complexity.

3.2 Genetic Algorithm for Key Searching

The workflow of GenUnlock's logic unlocking is detailed in Algorithm 2. GenUnlock deploys a *dynamic, diversity-aware* genetic algorithm for efficient and effective solution searching. *Diversity* evaluates the difference of individuals' gene representation and affects the convergence of GAs. The intuition behind GenUnlock is that we compute the diversity of the current population at the beginning of each epoch (iteration) to determine the 'gene flow' as shown in Figure 2. Diversity-guided GA dynamically alternates between the 'exploitation' mode (population selection and crossover) and the 'exploration' mode (mutation) in order to ensure a fast and stable convergence. We use the *dispersion* of the key sequences in the population as the measurement of the diversity metric. The formula of computing diversity is given in Equation (2).

$$div(S_K) = \frac{1}{P} \sum_{i=1}^{P} \sqrt{\sum_{j=1}^{k} [S_K(i,j) - \bar{S_K}(j)]^2}, \tag{2}$$

where $\bar{S_K}(j)$ is the sample average of all individuals at j^{th} bit:

$$\bar{S_K}(j) = \frac{1}{P} \sum_{i=1}^{P} S_K(i,j). \tag{3}$$

Here, P is the population size, k is the key length, $S_K \in \mathbb{B}^{P \times k}$ is the current population, and $S_K(i,j)$ denotes the jth bit of the ith individual in the population S_K. In the following of this section, we discuss the four main steps involved in GenUnlock's GA methodology as outlined in Algorithm 2.

Algorithm 2 Genetic Algorithm for Logic Unlocking.

INPUT: Netlist of target encrypted circuit (C_e); Size of the encryption key (L); Training dataset (S_I, S_O); GA parameters, including the population size (P), maximum number of generations (G), number of high-fitness (h) and low-fitness individual (l) for selection, number of child (c) for each pair of parent, and mutation rate m; Diversity threshold (d_{low}, d_{high}); Error tolerance of the attack (ϵ).

OUTPUT: A set of feasible key values ($\{\vec{K}\}$) that can unlock the circuit C_e.

1: Initialization:
   $S_K = \vec{K}_1, ..., \vec{K}_P \leftarrow generate_population(L,\ P)$.
   $i \leftarrow 0$
2: **while** $i < G$ and $F_K < 1 - \epsilon$ **do**
3: $\quad F_K \leftarrow evaluate_population_fitness(S_K,\ S_I,\ S_O)$
4: $\quad div \leftarrow compute_population_diversity(S_K)$
5: $\quad$ **if** $div < d_{low}$ **then**
6: $\quad\quad GA_mode \leftarrow$ '*explore*'
7: $\quad$ **else if** $div > d_{high}$ **then**
8: $\quad\quad GA_mode \leftarrow$ '*exploit*'
9: $\quad$ **if** $GA_mode ==$ '*exploit*' **then**
10: $\quad\quad S_K \leftarrow select_next_generation(S_K,\ F_K,\ h,\ l)$
11: $\quad\quad S_K \leftarrow crossover(S_K,\ c)$
12: $\quad$ **else if** $GA_mode ==$ '*explore*' **then**
13: $\quad\quad S_K \leftarrow mutate_population(S_K,\ m)$
14: $\quad$ **if** $F_K > 1 - \epsilon$ **then**
15: $\quad\quad$ break $\quad$ ▷ Check termination condition
16: $\quad i \leftarrow i + 1$
17: **Return:** Obtained a set of circuit deobfuscation keys S_K.

❶ **Fitness Evaluation.** The fast and accurate computation of fitness scores is the backbone of genetic algorithms. The definition of fitness is task-specific. Since our objective is to find (a set of) feasible decryption keys with high OF, we use the matching ratio of the specific key on the training data as the fitness measurement as shown in Equation (4). To facilitate the computation, GenUnlock first automatically constructs *auxiliary comparator components* that are added to the netlist of C_e, resulting in an evaluation netlist C_e^{aux}. Each comparator is implemented as an XNOR gate with two inputs where one of them comes from the ground-truth output in the training dataset. The auxiliary netlist is then converted to CNF to compute the fitness score based on Equation (4).

$$F_K = \frac{\#\ matched\ CNF\ clauses}{\#\ total\ CNFclauses} \tag{4}$$

❷ **Population Selection.** As a step of 'exploitation', the diversity of the population decreases after population selection. GenUnlock determines high-fitness individuals using the *tournament selection* technique [11]. A random subset of the current population is selected to participate in each round of the tournament. The individual with the highest fitness score is maintained in the next generation. Such a selection process repeats until the size of the resulting new generation reaches the desired number of high-fitness individuals (h). GenUnlock also incorporates several (l) 'lucky' individuals with relatively low fitness in the next generation in order to increases the randomness and help GA escape local optima.

❸ **Crossover.** Crossover (also called 'breeding') is the other step in 'exploitation'. In this process, the 'genome' (encoding) of the parents are *recombined* to produce the offsprings. Crossover consists of the following two subroutines: (i) Parent pairing: given the current population, GenUnlock randomly assigns two individuals as a pair of parents without repeating the use of an individual. (ii) Offspring generation: each bit of the child sequence is obtained from a uniform random sampling of the corresponding bit from its parents (i.e., 50% probability inheriting the bit from either of the parents).

❹ **Mutation.** mutation is of critical importance to maintain a certain level of diversity of the population. As such, *mutation* is performed in the *'exploration'* mode of GenUnlock when the population diversity is lower than the pre-defined threshold. There are two key parameters in the mutation process: the chance of mutation and the level of mutation. The first parameter determines the probability that mutation occurs on a particular individual. The second parameter dictates how many bits in the key sequence will be *flipped* as a result of mutation. A high chance and/or a large magnitude of mutation will result in large fluctuation of the fitness scores of the population, making the GA training unstable.

4 GENUNLOCK OPTIMIZATION

We empirically identify that *circuit fitness evaluation* is the bottleneck of GenUnlock's execution time. To accelerate circuit evaluation, GenUnlock deploys *circuit emulation* on the programmable hardware to obtain the response of the encrypted circuit (C_e) for the given input signals and the tested key. Furthermore, GenUnlock framework automatically constructs the customized auxiliary circuitry to pipeline each computation stage and reduce the runtime. GenUnlock framework integrates innovative hardware-level design to ensure attack efficiency. We explicitly discuss our hardware design optimizations as follows.

4.1 GenUnlock Architecture

GenUnlock leveraged an Algorithm/Software/Hardware approach to accelerate the key searching process for the target circuit as outlined in Figure 2. Particularly, GenUnlock maps the netlist of the encrypted circuit with the auxiliary part to the FPGA and perform circuit evaluation $\vec{O} = C_e^{aux}(\vec{I},\ \vec{K})$ directly. Given the input vector from the training data and the key sequence from the population, acquiring the circuit's response from the configured FPGA (circuit emulation) is significantly faster than the same process running on a host CPU (software simulation). In addition, GenUnlock parallelizes the computation of circuit emulation and pipelines each stage of GA operations. Population fitness evaluation and key evolving are performed in an online approach to minimize data communication between the off-chip DRAM and the FPGA.

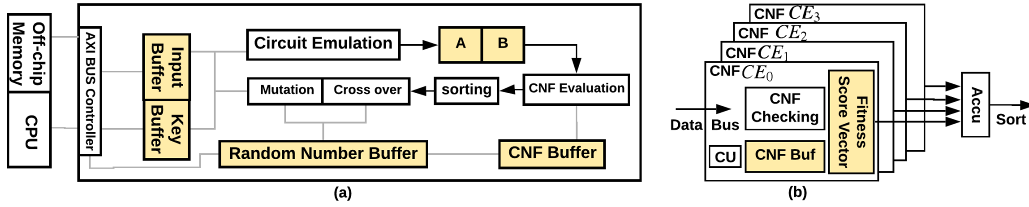

Figure 3: Overview of GenUnlock hardware design. The overall layout of the hardware system (a) and the implementation of CNF Checking Engines (b) are shown.

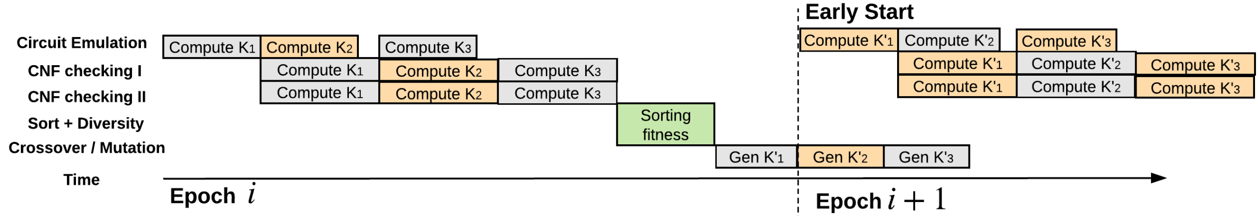

Figure 4: Pipelining optimization deployed in GenUnlock's genetic algorithm accelerator for logic unlocking.

GenUnlock Hardware Overview. Figure 3 illustrates the overview of GenUnlock's hardware architecture consisted of a computing engine for circuit emulation and an auxiliary circuitry for genetic operations. To reduce the data communication between the off-chip DRAM and the FPGA, we perform all computations of key evolution on-chip. Note that we do not include a random number generator (RNG) in GenUnlock's hardware design. Instead, GenUnlock stores a set of random numbers pre-computed on CPU using the inherent variation of the operating system. There are two main reasons behind our design choice: (i) The hardware implementation of a True RNG incurs non-trivial overhead, thus is not desired; (ii) Offloading random number generation to CPU typically provides stronger randomness compared to the one generated on FPGA. The results of circuit emulation are used for computing fitness scores using Equation (4) during CNF evaluation. The clause checking process in CNF evaluation is parallelized by accommodating multiple Checking Engine (CE) in GenUnlock's design. The workload for each CE is partitioned evenly offline.

After accumulating the fitness for each key sequence, the sorting engine permutes the key index based on their corresponding fitness. We implement a lightweight sorting engine following the 'even-odd sort' algorithm [5] for genetic selection, incurring a linear runtime overhead with the population size P. The population diversity is computed as follows. First, the average key is calculated along with circuit emulation as every key is read from the buffer. The *div* metric is then computed during sorting using l_1 norm instead of l_2 norm in Equation (2) to reduce computation complexity. Note that this change does not affect the performance of GenUnlock.

It is worth noting that GenUnlock does not employ a central control unit to coordinate the entire computation flow. Instead, each part of the design shown in Figure 3 follows a *trigger-based control* mechanism [12]. More specifically, each module is controlled by the status flag from its previous computation stage. For example, the sorting engine in GenUnlock begins to function when the fitness accumulation process is detected to be completed. Such a trigger-based control flow simplifies the control logic while respecting the data dependency between different modules shown in Figure 2. We detail the design of GenUnlock's circuit emulation and auxiliary circuitry in the following of this section.

4.2 GenUnlock Circuit Emulation

We empirically observe from GenUnlock's software implementation that circuit evaluation (i.e., obtaining $\vec{O} = C_e(\vec{I}, \vec{K})$) dominates the execution time (Section 5). Due to the high latency of evaluating a circuit netlist on CPU, we propose to use circuit emulation to improve the attack efficiency. The first step of circuit emulation is rewriting the netlist of the target encrypted netlist such that the values of all observable nodes can be recorded by registers. The rewritten circuit is then connected with the auxiliary circuitry and mapped onto FPGA. In this way, we can emulate the response of the target circuit C_e for any given input and key by directly applying the known signals (including $\vec{I}$ and $\vec{K}$) on the circuit and collecting the corresponding values in the registers. To further hide the latency of hardware evaluation, GenUnlock stores the emulation results in a ping-pong buffer and decouples it from the other hardware components as shown in Figure 3. More specifically, the CNF checking engine (CE) computes the fitness score of the population using the data from one buffer. In the meantime, the emulator acquires observable outputs of C_e given the next input/key pair $(\vec{I}, \vec{K})$ and stores the results into the other buffer.

4.3 GenUnlock Auxiliary Circuitry Design

In this section, we discuss how the auxiliary circuitry is constructed for the target circuit to accelerate the computation in GenUnlock's GA workflow as shown in Figure 2.

▪ **Pipeline Evolution Epochs with Early Starting.** GenUnlock's hardware design aims to maximize the time overlapping between each execution stages to increase the throughput of key evolution. As shown in Figure 4, the ping-pong buffer enables pipelined execution of hardware emulation and CNF evaluation. Furthermore, fitness evaluation and cross-over/mutation of each key in the population can be pipelined across different epochs. As illustrated in Figure 4, epoch $(i+1)$ starts circuit emulation and CNF evaluation when the previous epoch begins to breed new keys for the next epoch. As such, the latency of crossover and/or mutation can be hidden by circuit emulation and CNF evaluation.

■ **Scalable CNF Checking Engine.** Once circuit emulation is completed for the given input/key pair $(\vec{I}, \vec{K})$, GenUnlock begins to calculate the fitness score of the key sequence using Equation (4). From the perspective of the hardware, the fitness F_K is computed by accumulating the ratio of satisfied CNF clauses for the encrypted circuit C_e. Independence between different groups of wire signals typically exists in the encrypted circuit. GenUnlock leverages this property by distributing the checking of independent groups of clauses in CNF evaluation to different CNF checkers as shown in Figure 3 (b). As such, each CE stores a subset of CNF clauses in the associated CNF buffer. The accumulation of the ultimate fitness score completes when the last CE finishes CNF checking.

■ **Crossover and Mutation Logic.** The crossover logic exchanges random elements among two parent keys. The mutation logic randomly selects a subset of key bits and flip them (i.e. XOR the key sequence with a binary random mask vector). The execution of crossover and mutation can also be paralleled using multiple crossover and mutation processing units. In this case, each of the unit handles different segments of the key sequence and performs crossover and/or mutation. We use a default value of 1 for the number of the crossover/mutation unit since this step is not the bottleneck of GenUnlock's runtime.

5 GENUNLOCK EVALUATION

We investigate GenUnlock's performance on various benchmarks, including ISCAS'85 and Microelectronics Center of North Carolina (MCNC) [3] as summarized in Table 1.

Table 1: Summary of the evaluated circuit benchmarks.

Circuit	dataset	#in	#out	#gate	Key Length (5%, 10%, 25%)
c2670	ISCAS-85	233	140	1193	(60,119,298)
c432	ISCAS-85	36	7	160	(8,16,40)
c499	ISCAS-85	41	32	202	(48,51,101)
c5315	ISCAS-85	178	123	2307	(115,231,577)
c7552	ISCAS-85	207	108	3512	(176,351,878)
c880	ISCAS-85	60	26	383	(38,96,192)
des	MCNC	256	245	6473	(324,647,1618)
ex5	MCNC	8	63	1055	(106,264,528)
i9	MCNC	88	63	1035	(104,259,518)
seq	MCNC	41	35	3519	(132,265,660)

Experimental Setup. We demonstrate the software implementation of Algorithms 1 and 2 in python. Experiments are run on an Intel i7-7700k processor with 32 GB of RAM and the energy consumption is measured using *pcm-monitor* utility. We use the open-sourced code of the SAT attack [18] as our baseline comparison. Note that [18] is implemented in C++ and tested on a more powerful CPU (Intel Xeon E31320). As such, our empirical results serve as a *conservative relative speedup* comparison.

Our FPGA prototype is implemented on Zynq ZC706 board using the high-level synthesize tool Xilinx SDx 2018.2. GenUnlock's CNF checking engine and the auxiliary GA accelerator discussed in Section 4.1 are implemented using high-level programming language. Our design is synthesized using a clock frequency of 100MHz. The power of FPGA is measure at the socket using a power meter during the execution of the GenUnlock. Throughout our experiments, we set the number of CEs to $N_{ce} = 16$ and the encryption overhead to 10% with [14] as our default setting. As for GenUnlock's GA, we use a key population size $P = 80$ and the total number of generations $G = 50$. The number of high-fitness and low-fitness individuals are set to $h = 54$ and $l = 6$ for selection. Each pair of parents produces $c = 4$ children during crossover. The mutation rate is set to 2% (see Algorithm 2 for details). We generate 50 input/output pairs from the active IC to construct the training data as outlined in Algorithm 1.

5.1 Unlocking Capability

We assess the effectiveness of GenUnlock for logic unlocking on the benchmarks in Table 1. Each experiment is repeated 20 times to collect the statistics of the performance metrics. The maximum execution time is set to 10 hours (3.6×10^4 seconds). During this period, GenUnlock is able to unlock 10 out of 10 benchmarks (100% attack success rate) with the best key, while the baseline method [18] can only break 7 out of 10 benchmarks (70% attack success rate). In other words, GenUnlock framework finds a decryption key that yields an *ideal output fidelity* $OF = 1$. Figure 9 shows the runtime statistics of GenUnlock software implementation on CPU for unlocking various circuit benchmarks. One can see that GenUnlock's capability of logic unlocking increases over time.

For large and complex circuits such as *des*, *c2670* and *c7552*, traditional SAT-based method [18] takes very long to find distinguishing input patterns using the external SAT solvers (>10 hours). As such, SAT attacks fail to unlock the circuit with a very high probability when the design of the encrypted circuit turns out to be a *SAT-hard* problem (e.g., containing an internal 'and-tree'). Figure 5a shows the *encryption-agnostic* property of GenUnlock. The loss is computed as $(1 - OF)$. The convergence speed of GenUnlock depends on the adopted logic encryption scheme while the GA can always return a set of keys with improving quality over time. As opposed to the SAT attacks, GenUnlock is *generic* and is able to provide approximate keys with high output fidelity for circuits with arbitrary structures. Figure 5b shows the effect of GenUnlock's ensemble-based logic unlocking with the top three key sequences. The validation set is generated following the steps in Algorithm 1. It can be seen that the ensemble-based unlocking yields a small error compared to GenUnlock attack using the single best key.

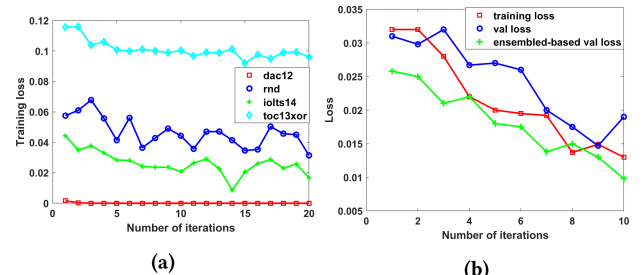

Figure 5: (a) Learning curve of GenUnlock for different logic encryption methods. (b) Effect of GenUnlock's ensemble-based logic unlocking using the top three keys.

5.2 Efficiency

Figure 6 shows the comparison between GenUnlock's software (Section 2.2) / hardware (Section 4.1) implementation with the baseline [18]. Note that we use the average runtime on each benchmark to visualize the performance comparison in Figure 6. Several circuits cannot be decrypted by the baseline algorithm within 10 hours.

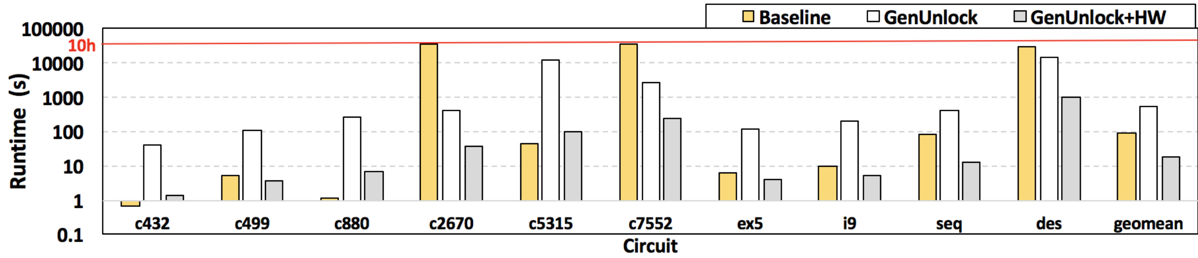

Figure 6: Average runtime comparison between GenUnlock and the baseline SAT attack [18]. 'GenUnlock' and 'GenUnlock+HW' denotes the latency of our software implementation and accelerated FPGA implementation, respectively.

In this case, we use 10 hours as the estimated runtime of [18] in Figure 6. With dedicated hardware design support, GenUnlock delivers on average 4.68× speedup compared to the baseline method. For SAT-hard circuits (such as *c2670*, *c7552*, *des*), GenUnlock engenders superior performance compared to SAT-based attacks, achieving 90×,13×, 2.1× speedup on CPU and 1014×, 153×, 31.2× speedup on the dedicated hardware. Besides the latency comparison, we also measure the power consumption of different circuit deobfuscation methods. The power consumption of 'GenUnlock+HW' on Zynq SoC is measured via the socket when the application is running. On average, GenUnlock with hardware optimization consumes 13.6W power while the software implementation of GenUnlock consumes 53.3W power on CPU. Considering the runtime, the overall energy-efficiency of GenUnlock is 18.3× higher than the SAT-based method.

GenUnlock's resource utilization depends on the key length (k) and the size of the original circuit. Table 2 shows the resource utilization of the assessed benchmark circuits. We present the sensitivity analysis of GenUnlock's performance in Section 5.3.

Table 2: Resource utilization of the auxiliary circuitry on *c432*,*c880*, *c2670* and *des* benchmarks with default settings (10% overhead and N_{CE} = 16) on Zynq ZC706.

Benchmarks	c432	c880	c2670	des
Data Transfer (Kbits/epoch)	1.3	3.0	9.5	51.8
BRAMS	22	27	37	86
DSP48E1	0	0	0	0
KLUTs (emulator usage)	9.4 (0.3)	12.1 (0.3)	19.4 (1.1)	41.1 (4.6)
FFs (emulator usage)	4,397 (80)	5,734 (160)	6,689 (316)	12,972 (1176)

5.3 Sensitivity Analysis

5.3.1 Sensitivity to Size of Key and Observable Wires. Figure 7 shows that the resource utilization of GenUnlock demonstrates an approximately *linear* dependency on the length of the encryption key length and the observable wires (which are primary outputs in our case). This is because a larger number of observable wires requires more comparator logic for each CNF checking engine as the index used in CNF checking requires a longer bitwidth, thus resulting in a higher LUT utilization. We tune the depth of the wire buffer and key buffer to accommodate the entire netlist.

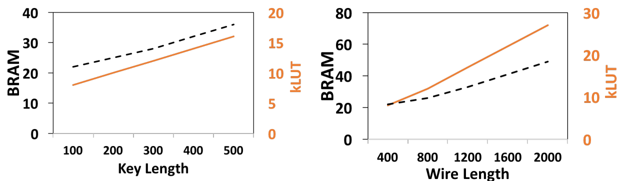

Figure 7: Resource utilization of the auxiliary circuitry with varying size of the encryption key (a) and observable wires (b). The key length and the wire length is set to 100 and 400 respectively in the initial setting.

5.3.2 Sensitivity to Number of CNF Checking Engines. Figure 8 shows the approximately linear relation between GenUnlock's speedup and the number of CEs. Our system can be scaled up by adding more CNF checking engines to parallel the clause checking process as GenUnlock's computation bottleneck is CNF evaluation. Nevertheless, the speedup saturates when N_{CE} is sufficiently high such that the computation overhead is dominated by crossover operation instead. GenUnlock broadcasts the observed wire values to all the CEs via a shared data bus. Each CE scans the CNF buffer and obtains the broadcast wire values for checking the satisfiability of the clauses. As such, increasing the number of CEs does not lead to extra wire delay. However, more CEs suggests a higher overhead during the fitness accumulation stage.

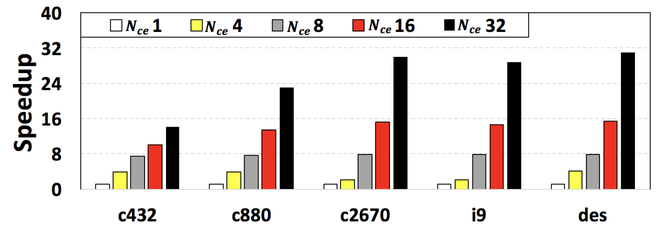

Figure 8: Scalability of GenUnlock to the number of CNF CEs. The speedup is near-linear with N_{CE} on large circuits where CNF checking is the computation bottleneck.

5.3.3 Sensitivity to Obfuscation Overhead. *Encryption overhead* is defined as the ratio of the additional key gates to the total number of gates in the original circuit. Larger encryption overhead suggests that a longer key sequence is used to encrypt the circuit. Figure 9 shows the execution time averaged across all assessed benchmarks with varying obfuscation overhead. One can see that GenUnlock's execution time does not grow exponentially with the increase of the obfuscation overhead, suggesting the scalability of GenUnlock framework to large circuits.

6 RELATED WORK

▪ **Conventional Circuit Deobfuscation.** The SAT-based attack on logic locking is first introduced in [18]. In their proposed method,

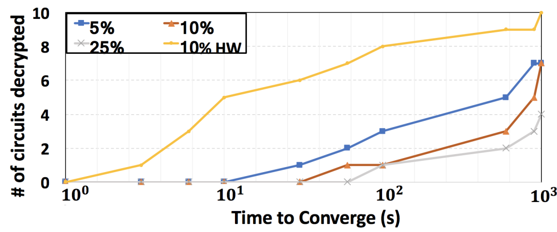

Figure 9: Execution time of GenUnlock averaged across all benchmarks. The circuits are encrypted using the logic locking technique in [14] with different obfuscation overhead.

a distinguishing input pattern is found by the external SAT solver in each iteration and is added as the constraints on the correct keys. The SAT algorithm terminates when no DIPs can be found, ensuring the full unlocking of the encrypted circuit. Later on, an active learning-based approached called 'AppSAT' is suggested in [16] where random queries are incorporated as constraints on the key in the iterative algorithm in addition to DIPs found by the SAT solver. As a result, AppSAT alleviates the limitation of SAT attacks on 'SAT-hard' circuits. Various attacks targeting at sequential circuit have also been studied [4]. In this paper, we mainly focus on GenUnlock's performance on combinational benchmark. Note that our attack framework and hardware optimization techniques are generic and applicable to the encrypted sequential circuits. We leave the implementation of GenUnlock on sequential circuits for future work.

▪ **Hardware Acceleration of Genetic Algorithms.** GAs have been adapted to FPGA platforms for various applications such as cognitive radio processing [8] and design exploration [2]. Existing works mainly focus on accelerating particular GA benchmarks and their fitness functions do not characterize the goal of logic unlocking. Also, fitness evaluation is not the bottleneck of computation latency in the existing GA acceleration benchmarks. As opposed to these works, GenUnlock customizes its hardware design to our particular defined problem (Section 2.2). We tailor the GA for attacking encrypted circuits and develop FPGA design optimizations to accelerate our proposed algorithm.

7 CONCLUSIONS

In this paper, we introduce GenUnlock, the first genetic algorithm-based framework for logic unlocking. GenUnlock leverages an Algorithm/Software/Hardware co-design approach and engenders superior performance improvement compared to traditional SAT-based attacks. More specifically, GenUnlock is encryption-agnostic and is generic to arbitrary circuit designs. Our framework yields a set of feasible keys that unlock the obfuscated circuit with an attacker-defined output fidelity. Furthermore, GenUnlock, for the first time, provides the trade-off between runtime overhead and output fidelity of the resulting keys. In real-world settings, GenUnlock poses a threat to the rising amount of fault-tolerant applications such as block-chain mining and deep neural networks. Circuit emulation and pipelining are presented as hardware optimization techniques to further accelerate the computation of GenUnlock. We perform comprehensive experiments to corroborate the efficiency and effectiveness of GenUnlock across different circuit benchmarks. In future work, we will consider using the learning-based algorithms to guide the mutation and crossover operations.

REFERENCES

[1] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. 2019. SMT Attack: Next Generation Attack on Obfuscated Circuits with Capabilities and Performance Beyond the SAT Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), 97–122.

[2] C. Bolchini, P. L. Lanzi, and A. Miele. 2010. A multi-objective genetic algorithm framework for design space exploration of reliable FPGA-based systems. In *IEEE Congress on Evolutionary Computation*. 1–8. https://doi.org/10.1109/CEC.2010.5586376

[3] Franc Brglez, David Bryan, and Krzysztof Kozminski. 1989. Combinational profiles of sequential benchmark circuits. In *IEEE international symposium on circuits and systems*, Vol. 3. 1929–1934.

[4] Rajat Subhra Chakraborty and Swarup Bhunia. 2008. Hardware protection and authentication through netlist level obfuscation. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 674–677.

[5] TC Chen, Kapali P Eswaran, Vincent Y Lum, and C Tung. 1978. Simplified odd-even sort using multiple shift-register loops. *International Journal of Computer & Information Sciences* 7, 3 (1978), 295–314.

[6] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138.

[7] Stephen A Cook and David G Mitchell. 1997. Finding hard instances of the satisfiability problem. In *Satisfiability problem: theory and applications: DIMACS workshop*, Vol. 35. 1–17.

[8] P. V. dos Santos, J. C. Alves, and J. C. Ferreira. 2013. A framework for hardware cellular genetic algorithms: An application to spectrum allocation in cognitive radio. In *2013 23rd International Conference on Field programmable Logic and Applications*. 1–4. https://doi.org/10.1109/FPL.2013.6645599

[9] Mark C Hansen, Hakan Yalcin, and John P Hayes. 1999. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers* 16, 3 (1999), 72–80.

[10] Dominiek Ter Heide. Feb, 2018. *A Closer Look At Ethereum Signatures*. https://hackernoon.com/a-closer-look-at-ethereum-signatures-5784c14abecc

[11] Brad L Miller, David E Goldberg, et al. 1995. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems* 9, 3 (1995), 193–212.

[12] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, et al. 2013. Triggered instructions: a control paradigm for spatially-programmed architectures. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 142–153.

[13] Michael J Pennock, Douglas A Bodner, and William B Rouse. 2018. Lessons learned from evaluating an enterprise modeling methodology. *IEEE Systems Journal* 12, 2 (2018), 1219–1229.

[14] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. 2012. Security analysis of logic obfuscation. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 83–89.

[15] Jeyavijayan Rajendran, Michael Sam, Ozgur Sinanoglu, and Ramesh Karri. 2013. Security analysis of integrated circuit camouflaging. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 709–720.

[16] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin. 2017. AppSAT: Approximately deobfuscating integrated circuits. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 95–100. https://doi.org/10.1109/HST.2017.7951805

[17] Yao Shi, Myungjoon Choi, Ziyun Li, Gyouho Kim, Zhiyoong Foo, Hun-Seok Kim, David Wentzloff, and David Blaauw. 2016. 26.7 A 10mm3 syringe-implantable near-field radio system on glass substrate. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 448–449.

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

[19] Randy Torrance and Dick James. 2009. The state-of-the-art in IC reverse engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 363–381.

[20] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan JV Rajendran, and Ozgur Sinanoglu. 2016. SARLock: SAT attack resistant logic locking. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 236–241.

[21] Muhammad Yasin, Jeyavijayan JV Rajendran, Ozgur Sinanoglu, and Ramesh Karri. 2016. On improving the security of logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 9 (2016), 1411–1424.