GALU: A Genetic Algorithm Framework for Logic Unlocking

HUILI CHEN, University of California, San Diego, USA
CHENG FU, University of California, San Diego, USA
JISHEN ZHAO, University of California, San Diego, USA
FARINAZ KOUSHANFAR, University of California, San Diego, USA

Logic locking is a circuit obfuscation technique that inserts additional key gates to the original circuit in order to prevent potential threats such as circuit overproduction, piracy, and counterfeiting. The encrypted circuit generates desired outputs only when the correct keys are applied to the key gates. Previous works have identified the vulnerability of logic locking to satisfiability (SAT)-based attacks. However, SAT attacks are unscalable and have limited effectiveness on circuits with SAT-hard structures. To address the above constraints, we propose GALU, the first *genetic algorithm-based logic unlocking* framework that is *parallelizable* and significantly faster than the conventional SAT-based counterparts. GALU works by formulating circuit deobfuscation (i.e., identifying the correct keys) as a *combinatorial optimization* problem and approaches it using genetic algorithms (GAs). We consider key sequences as individuals in distinct populations and propose an *adaptive, diversity-guided* GA framework consisting of four main steps: circuit fitness evaluation, population selection, crossover, and mutation. In each iteration, the key sequences with high fitness scores are selected and transformed into the offspring key sequences. As a result of evolutionary key searching, GALU is highly scalable, effective, and efficient.

To optimize the runtime overhead of logic unlocking, we integrate the design of GALU's algorithm, software and hardware in a closed loop. In particular, we identify circuit fitness evaluation as the performance bottleneck and employ *hardware emulation* on programmable hardware for runtime optimization. To this end, GALU framework automatically constructs customized auxiliary circuitry to *pipeline* the computation in constraints checking, sorting, crossover, and mutation. GALU is the first adaptive and scalable attack framework that provides the *flexibility/trade-off* between runtime overhead and key usability. This is achieved by producing a group of *approximate* keys with improving quality over time. We perform a comprehensive evaluation of GALU's performance on various benchmarks and demonstrate that GALU achieves up to 1089.2× speedup and 4268.6× more energy-efficiency compared to the state-of-the-art SAT attacks for circuit logic unlocking.

CCS Concepts: • **Security and privacy** → **Malicious design modifications**; **Hardware attacks and countermeasures**; **Embedded systems security**; • **Hardware** → *Hardware test*.

Additional Key Words and Phrases: logic locking, circuit obfuscation, genetic algorithm, co-design

1 INTRODUCTION

A diverse set of real-world applications such as household appliances, biomedical devices, and autonomous systems are empowered by integrated circuits (ICs). Contemporary ICs are vulnerable to potential attacks including IC piracy, overproduction, and counterfeiting [37, 53] since their supply chain involves cooperation between multiple parties. To enhance the security of digital IC, researchers and practitioners have developed various circuit obfuscation techniques. Logic locking and circuit camouflaging are two typical obfuscation methods that target to protect the Intellectual Property (IP) of ICs. IC camouflaging focuses on preventing *layout-level* reverse engineering (RE) attacks by adding dummy contacts/cells to the standard gates [39]. As a result, the purpose of the circuit module is decoupled from its appearance. Logic locking aims to protect the *functionality* of the circuit by inserting additional key gates to the original netlist [56, 58]. Consequently, the locked circuit only generates desired outputs when the correct decryption key is applied to the key gates.

Authors' addresses: Huili Chen, University of California, San Diego, 9500 Gilman Drive, La Jolla, CA, 92093, USA, email:huc044@ucsd.edu; Cheng Fu, University of California, San Diego, 9500 Gilman Drive, La Jolla, CA, 92093, USA, email:cfu@ucsd.edu; Jishen Zhao, University of California, San Diego, 9500 Gilman Drive, La Jolla, CA, 92093, USA, email:jzhao@ucsd.edu; Farinaz Koushanfar, University of California, San Diego, 9500 Gilman Drive, La Jolla, CA, 92093, USA, email:farinaz@ucsd.edu.

A line of research has focused on evaluating the security of logic locking techniques. As the first attempt, [50] demonstrates that satisfiability (SAT)-based attacks can be performed to break circuits encrypted with different logic locking methods. Conventional SAT attacks and their variants work by eliminating incorrect keys with distinguishing input patterns (DIPs) found by SAT solvers. While effective on some circuits, SAT-based attacks have the following limitations: (i) Unscalable to large benchmarks due to the increasing size of DIP constraints over iterations; (ii) SAT solving is non-trivial and challenging to parallelize; (iii) Ineffective on SAT-hard circuit designs (e.g., an internal 'and-tree' structure). This is due to the fact that a DIP can only eliminate a single incorrect key in this case [50]. As such, the runtime of the deobfuscation attack has an exponential dependency on the encryption key length [50]. There are three key challenges to develop an effective and efficient logic unlocking attack. First of all, the logic decryption scheme shall be agnostic to the circuit structure as well as the logic locking method. Secondly, the deobfuscation attack shall be scalable to large and complex benchmarks. Thirdly, the attack is desired to provide a trade-off between the attack success rate and runtime overhead.

To address the above challenges and the limitations of SAT-based attacks, we present GALU, the first genetic algorithm-based framework for circuit logic unlocking. GALU assumes the availability of the encrypted circuit netlist and black-box access to the active IC. The output of GALU is a set of decryption keys that yield high function-level accuracy when applied on the locked circuit. Our framework has two main phases: *(i) Offline training data generation* that queries the active IC and collects the corresponding observable wires. *(ii) Online key evolution* that iteratively updates a group of potential decryption keys using an adaptive genetic algorithm. Compared to prior SAT-based attacks, GALU can efficiently find a set of *approximate* keys that yield correct outputs with high probability. Furthermore, our attack provisions the adversary with the trade-off between unlocking accuracy and attack runtime.

GALU consists of four genetic operators: circuit fitness evaluation, probabilistic population selection, diversity-aware crossover, and dynamic mutation. To ensure stable convergence, we leverage *diversity-guided* GA to adaptively determine the evolutionary transformation of the key sequences. We characterize the overhead of each GA operator and use Algorithm / Software / Hardware co-design for performance optimization. More specifically, GALU deploys *hardware emulation* and *pipelining* to accelerate the computation on FPGAs. To the best of our knowledge, GALU is the first framework that realizes hardware implementation and optimization for logic unlocking. The technical contributions of this paper are summarized as follows:

- **We present the first genetic algorithm-based logic unlocking attack.** GALU's adaptive and diversity-guided key searching technique promotes efficient exploration of large key space. As such, our attack scheme has superior scalability and generalizability compared to traditional SAT attacks.
- **We enable logic unlocking with performance trade-off between attack accuracy and runtime.** GALU provides the adversary with unprecedented flexibility to perform circuit deobfuscation given the attack budget.
- **We optimize the efficiency of GALU using Algorithm/Software/Hardware co-design.** Our attack framework provides a scalable and high-performance hardware design that improves key searching speed to a higher level. Particularly, our hardware design incorporates various optimization techniques including computation pipelining and circuit emulation on programmable hardware.
- **We investigate GALU's performance on various circuit benchmarks.** We corroborate the superior effectiveness and efficiency of GALU compared to the existing SAT attacks with a comprehensive evaluation.

GALU sheds light on the potential of adapting genetic algorithms to address the long-standing challenges in the hardware domain, thus opens a new direction for the growing research in hardware security. Our attack framework is alternative to the existing SAT-based attack methods and discovers a new attack dimension for approximate circuit unlocking. GALU is flexible and able to identify a set of feasible keys for logic decryption with improving effectiveness over time. Furthermore, we demonstrate that the function-level accuracy of the

encrypted circuit can be enhanced by *ensemble learning*. More specifically, the final circuit output is determined by aggregating the individual circuit responses obtained by applying each of the participating unlocking keys returned by GALU after convergence.

2 BACKGROUND AND MOTIVATION

2.1 Circuit Obfuscation

There are two common circuit obfuscation techniques to protect the IP of modern ICs. *IC camouflaging* has been suggested to protect the layout design of the circuit against reverse engineering attacks. Existing IC camouflaging technique includes insertion of dummy connections and/or cells, as well we doping modification [38, 39]. The objective of IC camouflaging is to *decouple* the correlation between the appearance and the corresponding functionality of gates. Figure 1 shows an example of the layout-level circuit camouflaging. The structure of gates with different functionalities (e.g., NAND and NOR) are different by default as can be seen in Figure 1a. However, such a *layout disparity* can be hidden from the attackers by adding redundant connections to both standard gates, resulting in two gates with an identical appearance as shown in Figure 1b. As such, layout-level reverse engineering attacks are invalidated since no useful information is leaked from the appearance of the circuit.

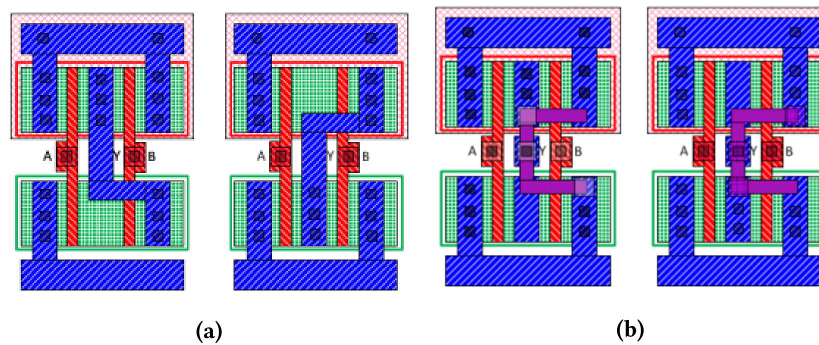

Fig. 1. Demonstration of IC camouflaging [39]. (a). Original layouts of a regular 2-input NAND gate (left) and a NOR gate (right) where their metal layers have different appearances. (b). Camouflaged layouts of the corresponding standard cells in (a) where no structure difference can be captured to differentiate two gates.

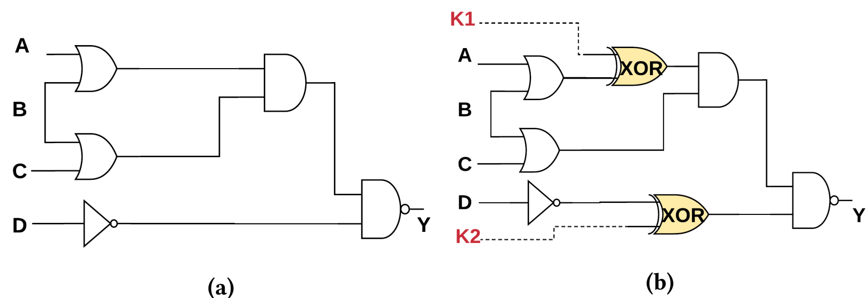

Fig. 2. Example of XOR-based logic locking. The encrypted circuit (b) yields consistent outputs as the original one (a) only when the two-bit key K_1K_2 is set to $2'b00$.

Logic locking has been proposed to protect the intellectual property of ICs by corrupting the functionality of the circuit when incorrect key values are applied to the additional key gates. Compared to IC camouflaging, logic locking is able to prevent attacks from untrusted fabrication foundries. Existing logic locking techniques include performing XOR/XNOR operations of wires with the key inputs [40, 42], substituting a subset of gates

with look-up-tables (LUTs) that stores the key sequence [4], and inserting multiplexers (MUXs) controlled by the key bits [38, 40]. An example of XOR-based logic locking is shown in Figure 2.

While IC camouflaging and logic locking are effective to alleviate the vulnerability of ICs in the semiconductor supply-chain, various attacks have been demonstrated to invalidate these defense techniques. We focus on defeating logic locking in this paper. SAT-based attacks and their variants are able to find a valid decryption key to activate the locked circuit [50]. Removal attacks take a different approach from SAT attacks and deobfuscate the circuit by identifying and removing the protection circuitry that is consisted of the additional key gates [57]. To the best of our knowledge, GALU framework, for the first time, achieves *time-evolving key searching* and alternative *ensemble-based circuit evaluation.* GALU is orthogonal to the state-of-the-art attack methods on logic locking and provides a trade-off between the attack efficiency and effectiveness.

Conjunctive Normal Form. It has been proven in theory that every Boolean function can be converted into an equivalent formula in conjunctive normal form (CNF) [3]. CNF takes the form of a sequence of *clauses* that are connected by AND operator. All variables inside the same clause are connected by OR operator. GALU deploys CNF representation of the encrypted circuit netlist (discussed in Section 5) since such a data structure facilitates the verification of whether the given set of constraints are satisfied. Let us consider the following Boolean expression in the CNF form:

$$O = (x_1 \lor \neg x_3) \land (x_2 \lor x_3 \lor \neg x_1). \tag{1}$$

This circuit is equivalent to the CNF statements (in dimacs format) shown in Equation (2) where each row of numerical sequence corresponds to a clause in Equation (1).

$$\begin{array}{llll} p & \text{cnf} & 3 & 2 \\ 1 & -3 & 0 & \\ 2 & 3 & -1 & 0 \end{array} \tag{2}$$

We refer the readers to [2] for detailed steps of CNF conversion and interpretation.

2.2 Genetic Algorithms

Genetic algorithms are a popular subcategory of evolutionary algorithms (EAs) and have wide applications including feature selection, Knapsack problem, and hardware design optimization [52]. Inspired by *natural evolution* in the real world, a GA routine typically involves the following four genetic operations: fitness evaluation, population selection, crossover, and mutation. In each generation (i.e., iteration), individuals with high fitness scores have a higher probability to 'survive' the selection and produce offsprings. Crossover and mutation increase the diversity of the population, which reduces the probability of GA getting trapped in local optima. The convergence speed and quality of GA depend on the distribution of the population. In particular, the exploration and exploitation of GA shall be balanced dynamically. To this end, *diversity* is introduced to evaluate the difference of individuals' gene representation and control the above balance.

As an instance of heuristic-based optimization methods, GAs feature the following advantages: (i) They are applicable when the objective function is not smooth (in which case derivative-based methods cannot be employed); (ii) They search from a population of solutions simultaneously, thus are able to avoid local optima; (iii) They always yield a solution and the solution gets better over time; (iv) The required computation is inherently parallel, which makes them suitable for hardware acceleration.

2.3 Our Motivation

Prior works on circuit deobfuscation mainly rely on external SAT solvers [7, 48] to find distinguishing input patterns and eliminate incorrect keys [43, 50]. Such an approach has demonstrated reasonable attack performance on many modern ICs. However, the existence of *SAT-hard* problems [13] makes it challenging to apply the SAT attacks in these scenarios. For instance, the SAT attack algorithm proposed in [50] fails to activate the *c*2670

benchmark since the circuit contains an internal 'and-tree' structure. The multiplier circuit *c*6880 is not evaluated in [50] since it is known to be difficult for SAT solvers. To address this limitation of SAT-based attacks, we propose GALU, the first genetic algorithm-based circuit deobfuscation framework that is able to attack circuits with SAT-hard structure.

Real-world Applications. Existing works intend to unlock the obfuscated circuit by finding a decryption key that ensures the correct output for every input pattern. However, such an unlocking scheme may take a prohibitive amount of time to break large circuits with long encryption keys as we investigate in Section 7. However, we want to emphasize that fast, partial decryption of the target circuit can be more threatening than a slow, full decryption. This is particularly true for *fault-tolerant* applications. For example, real-time localization hardware [32] or machine learning accelerators [29] are fault-tolerant since the final result is typically the majority vote from multiple sensors nodes or image frames. As such, a partially incorrect output can be hidden by the majority of other correct outcomes.

Another example is the emerging ASIC accelerators for Deep Neural Networks (DNN) that are inherently fault-tolerant. Many previous hardware designs exploit this property to accelerate DNN after parameter quantization or pruning [21, 44]. As such, minor computation error that came from the approximate key for unlocking the DNN hardware accelerator will cause negligible degradation of accuracy. Let us also consider block-chain mining as another real-world application. In this task, the signature of the cryptocurrency is extracted by performing AES and hashing operations [25] on the hardware miner. The resulting signature is continuously checked against the pre-defined template to determine whether the cryptocurrency is legit. 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.

In summary, the GA primitive of GALU can produce a set of feasible keys for approximately unlocking the functionality of the encrypted circuit, thus enables the adversary to launch the attack with only a few iterations. The approximate deobfuscation attack is also a severe threat for many real-world applications where minor hardware errors can be tolerated.

2.4 Challenges

We identify three main challenges of developing an efficient and effective logic unlocking technique as follows.

(C1) Efficient Solution Space Search. The proposed circuit unlocking scheme shall be able to explore the solution space in an efficient and strategical way since the searching space of the solution has an exponential dependency on the length of the encryption key.

(C2) Agnostic to encryption scheme and circuit design. The deobfuscation attack shall be generic such that it is applicable to the target circuit with arbitrary design structure and unknown logic locking schemes.

(C3) Scalable to large benchmarks. The computation complexity of the attack method shall not scale exponentially with the size of the encrypted circuit nor the key length.

3 RELATED WORK

■ **Conventional Circuit Deobfuscation.** The SAT-based attack on logic locking is first introduced in [50]. In their proposed method, a distinguishing input pattern is found by the external SAT solver [7, 48] 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 [43] 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 the sequential circuit have also been studied [10]. In this paper, we mainly focus on GALU's performance on a 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 GALU on sequential circuits for future work.

■ **Applications of Genetic Algorithms.** Genetic algorithms belong to the heuristic-based search and optimization family [15]. GAs have been widely used in many real-world applications, such as evolvable hardware [49] and code breaking [16]. GAs are also used in parameter selection with large design space, such as DNN structure exploration [46] and non-gradient-based training [51]. However, the deployment of GAs in the hardware security domain has not been explored. To the best of our knowledge, GALU is the first framework that employs GAs for attacking obfuscated circuit.

■ **Hardware Acceleration of Genetic Algorithms.** GAs have been adapted to FPGA platforms for various applications such as cognitive radio processing [17] and design exploration [8]. 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, GALU customizes its hardware design to our particular defined problem (Section 4.1). We tailor the GA for attacking encrypted circuits and develop FPGA design optimizations to accelerate our proposed algorithm.

■ **Circuit Emulation.** FPGAs have been widely used as configurable platforms for emulating certain behaviors of circuits due to their programmable features. One of the most useful application scenarios is logic verification for HDL code before ASIC tape-out [27]. Bhattacharya et al. [5] propose a technique that uses FPGAs for emulating mixed-signal circuits. Experiments on quantum circuit are typically performed on FPGAs using emulation methods [30]. Biancolin et al. [6] propose to accelerate the DRAM simulation process by emulating the behavior of memory using FPGAs.

4 GALU OVERVIEW

We formulate the problem of circuit logic unlocking in Section 4.1. Our threat model and GALU's global workflow framework are introduced in Section 4.2 and Section 4.3, respectively.

4.1 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 $\vec{I}$:

$$OF(\vec{K};\ C_o,\ C_e) = \underset{\forall \vec{I} \in \mathbb{B}^M}{Prob}\ [\ C_e(\ \vec{I},\ \vec{K}) = C_o\ (\vec{I})]. \tag{3}$$

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 [50] if the condition $C_e(\vec{I}, \vec{K}_1) = C_e(\vec{I}, \vec{K}_2)$ is satisfied for any $\vec{I} \in \mathbb{B}^M$.

4.2 Threat Model

We make the following assumptions about GALU 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 stimuli and observe the corresponding outputs. This is the basis of GALU's training data generation phase (Phase I in Figure 3) that will be explained later.

(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 [23]. In this work, we convert the obtained netlist to CNF for circuit fitness evaluation as shown in Phase II of Figure 3. However, we emphasize that GALU is generic and can work with other circuit simulation tools to compute the fitness score defined in Equation (6). The simulation tool does not need to be CNF-based and is only required to be able to obtain values of the observable wires when the input is applied on the encrypted circuit. We assume primary outputs to be observable wires in this work.

4.3 GALU's Global Flow

Figure 3 illustrates the global flow of GALU. We discuss the attack assumptions in Section 4.2. GALU 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.

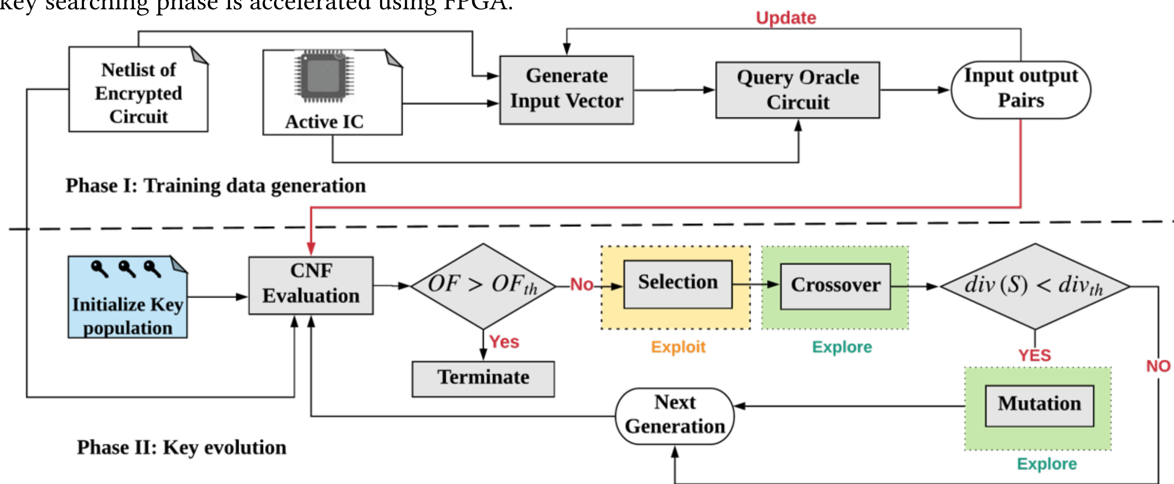

Fig. 3. GALU's global workflow for approximate circuit logic unlocking. Our framework has two key phases: offline training data generation (top part) and online GA-based key evolution (bottom part).

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

- **Generate input vectors.** Given the netlist of the encrypted circuit, GALU crafts input vectors and filter the ones that result in the same circuit outputs when different keys are applied. Such an input filtering step improves the quality of the training data for our logic unlocking task.
- **Query active IC.** The remaining input patterns from step 1 are then used to query the active IC. The training dataset for GALU's key searching is constructed from the collected input/output (IO) pairs.

Phase II: Key evolution. Once the training data for the target circuit is generated in Phase I, GALU performs three subroutines during the key evolution phase as shown in the bottom of Figure 3.

- **Circuit fitness evolution.** Analogous to natural selection, 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 observable nodes (including primary outputs and potential inserted test points) that match with the training dataset when a specific key is applied. Note that GALU's approximate logic unlocking terminates when the output matching accuracy is larger than the threshold.

- **Population diversity computation.** GALU separates genetic operations into two groups (*'exploitation'* or *'exploration'*) and determines which branch to take depending on the population diversity.
- **Diversity-guided GA execution.** GALU 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, GALU's *dynamic, diversity-aware* GA execution demonstrates better convergence.

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 (3)) and the execution time, respectively. GALU, 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. Section 7 presents a detailed, quantitative analysis of these performance metrics across various benchmarks.

5 GALU METHODOLOGY

Prior works have identified that there might be more than one correct key to unlock the given circuit [50]. GALU leverages this fact and processes multiple keys representing different equivalence classes in each iteration, thus features higher efficiency for space exploration. Note that GALU is agnostic to the underlying encryption schemes used by the defender, thus is genetic and applicable to arbitrary ICs. We detail the workflow of GALU's two key phases (shown in Figure 3) in Section 5.1 and 5.2, respectively. To further boost the unlocking accuracy on the target circuit, we show how multiple high-quality keys found by GALU can be leveraged by the ensemble method to construct joint, better decryption in Section 5.3.

5.1 Training Data Generation

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

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); Primary inputs size (M) and the encryption key length (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.

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.

5.2 Adaptive Genetic Algorithm for Key Searching

Algorithm 2 outlines the workflow of GALU's logic unlocking. GALU deploys a *dynamic, diversity-aware* genetic algorithm for efficient and effective solution searching. Prior works have identified population diversity as the key factor that determines the trade-off between the convergence speed and the solution's optimality of genetic algorithms [15, 45, 54]. Diversity-guided GA dynamically alternates between the *'exploitation'* mode (population selection) and the *'exploration'* mode (crossover and mutation) in order to ensure fast and stable convergence. The intuition behind GALU is that, in each generation, we compute the diversity of the current population to determine whether the mutation operation shall be performed for exploration purpose as shown in Figure 3. The formula to compute diversity is shown in Equation (4).

$$div(S_K) = \frac{1}{P}\sum_{i=1}^{P}\left[\sum_{j=1}^{k}[S_K(i,j) - \bar{S_K}(j)]^2\right], \tag{4}$$

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

$$\bar{S_K}(j) = \frac{1}{P}\sum_{i=1}^{P} S_K(i,j). \tag{5}$$

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 GALU's GA methodology as outlined in Algorithm 2.

We detail each step of GALU's online key searching in the following of this section.

❶ **Circuit Fitness Evaluation.** It is critical to develop a fast and accurate computation routine for fitness scores since this step is the backbone of GAs. The definition of fitness value is domain and task-specific. Since our objective is to find (a set of) feasible decryption keys with high OF, we use the matching ratio of circuit response when a specific key is applied on the encrypted circuit as the fitness measurement as shown in Equation (6). We consider both primary outputs and potentially inserted test points [11, 35] as observable circuit responses when evaluating the fitness score of a decryption key. As such, our definition of the fitness value is general and provides more information about the decryption effectiveness compared to output fidelity in the case where internal test points are available for measurement. This work demonstrates the performance of GALU with only primary outputs as observable wires (detailed in Section 7). To facilitate the computation with hardware acceleration, GALU first automatically constructs *auxiliary comparator components* that are added to the netlist of C_e, resulting in an evaluation netlist C_e^{aux}. Particularly, GALU creates a comparator for each *observable wire* in C_e. Each comparator is implemented as an XNOR gate with two inputs where the first input is the wire value in C_e and the second input is the ground-truth wire value obtained from querying C_o. The fitness score of a specific key is then computed using the following equation:

$$F_K = \frac{\#\ matched\ observable\ wires}{\#\ total\ wires} \tag{6}$$

We make the design choice to compute the fitness score instead of the output fidelity defined in Equation (3) for two reasons: (i) The fitness score is a feasible indicator for assessing the quality of a decryption key. A high fitness score suggests that this key has a high probability of yielding a high output fidelity value; (ii) The computation of

Algorithm 2 Genetic Algorithm for Approximate Logic Unlocking.

INPUT: Netlist of target encrypted circuit (C_e); Length of the encryption key (k); Training dataset (S_I, S_O). GA parameters, including the population size (P); maximum number of generations (G); number of maintained individuals (L) for selection; probability of crossover (p_{cross}) and bit-wise element exchanging (p_{exch}), number of children per parent pair (c) for crossover; mutation rate p_{mutate} and bit flipping probability (p_{flip}) for mutation operation. Diversity threshold (div_{th}); Error tolerance of circuit unlocking (ϵ).

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

1: Initialization:
$S_K = \left\{\vec{K}_1, ..., \vec{K}_P\right\} \leftarrow generate_population(k,\ P)$.
$i \leftarrow 0$
2: **while** $i < G$ **do**
3: $F_K \leftarrow evaluate_population_fitness(S_K,\ S_I,\ S_O)$
4: **if** $F_K > 1 - \epsilon$ **then**
5: break ▷ Check termination condition
6: $S_K \leftarrow probabilistic_population_selection(S_K,\ F_K,\ L)$
7: $S_K \leftarrow diversity_driven_crossover(S_K,\ p_{cross},\ p_{exch}),\ c$
8: $div \leftarrow compute_population_diversity(S_K)$
9: **if** $div < div_{th}$ **then**
10: $S_K \leftarrow adaptive_mutation(S_K,\ div,\ p_{mutate},\ p_{flip})$ ▷ Diversity guided evolution
11: $i \leftarrow i + 1$
12: **Return:** Obtained set of circuit deobfuscation keys S_K.

the fitness score can be implemented efficiently with hardware acceleration (detailed in Section 6) compared to the computation of OF.

❷ **Probabilistic Population Selection.** As a step of 'exploitation', the diversity of the population decreases after population selection. GALU selection stage chooses key sequences with high quality based on the ordering of Pareto dominance to generate the next population. In particular, we employ a fitness-proportionate selection scheme. Let us consider a population $S_K = \left\{\vec{K}_i\right\}_{i=1}^{P}$ where $\vec{K}_i \in \{0,1\}^k$ is the ith individual (i.e., key) in the population. The fitness score of each individual is computed based on Equation (6) and denoted as $\{F_K(i)\}_{i=1}^{P}$. GALU's probabilistic population selection strategy chooses superior key sequences using the following probability distribution function (PDF) over the population based on individuals' fitness scores:

$$prob_{sel}(i) = \frac{F_K(i) - F_{min}}{\sum_{i=1}^{P}\left[F_K(i) - F_{min}\right]}. \tag{7}$$

Here, F_{min} is the minimum fitness score in the population. Our PDF construction subtracts the minimum score from the fitness score $F_K(i)$ to ensure that the worst key has zero probability of being selected. As a result, superior decryption keys are selected by performing a *non-uniform random sampling* of the old population where the probability of choosing each individual is obtained using Equation (7). We allow the attacker to specify the total number of individuals (L) in the new population S_K^L to satisfy his attack budget. Also, we use a 'sampling with replacement' policy when applying the PDF in Equation (7). This means that high-quality individuals can

be selected multiple times while low-quality ones are rarely opted. The combination of GALU's circuit fitness evaluation and probabilistic selection preserve superior logic unlocking keys and suppresses weak ones.

❸ **Diversity-driven Crossover.** Crossover (also called 'breeding') is another step of 'exploration' of GA computation. Crossover has two subroutines: parent pairing, and offspring generation. In this process, the 'genome' (encoding) of the parents are *recombined* to produce the offsprings. Note that since GALU targets to find a binary-valued key sequence to unlock the circuit, the encoding of each individual (decryption key) is not required. We design a *diversity-driven crossover* operator to promote the exploration in the crossover step.

More specifically, we build parent pairs as follows. Given the selected population S_K, we first sort them based on their fitness scores. The first parent $\vec{K_1}$ is set as the individual with the highest fitness value and its spouse $\vec{K_2}$ is determined as the one that has the largest distance. Such a *disparity-aware* parent pairing strategy facilitates exploration by increasing the diversity of the new offsprings. Consistent with the diversity definition in Equation (4), we compute the pairwise distance between a pair of individuals as follows:

$$dist(\vec{K_1}, \vec{K_2}) = \frac{1}{2k}\sqrt{\sum_{j=1}^{k}(\vec{K_1}[j] - \vec{K_2}[j])^2}, \tag{8}$$

where k is the key length. Note that since we are only concerned with the relative pairwise distance values, we can neglect the square root and the constant factor $\frac{1}{2k}$ in Equation (8).

GALU controls the diversity-driven crossover step with two hyper-parameters p_{cross} and p_{exch} as shown in line 7 of Algorithm 2. In each iteration, p_{cross} dictates the probability of performing crossover between each pair of parents. If crossover will occur, p_{exch} determines the bit-wise exchanging probability between the parents. Our proposed crossover operator allows the adversary to maintain high-quality individuals in the resulting population while encouraging diverse 'genomes' to exchange knowledge.

❹ **Adaptive Mutation.** Genetic algorithms might be trapped in local optima and yield unsatisfying results. The mutation is of critical importance to maintain a certain level of population diversity to prevent the above premature convergence. Particularly, mutation *explores* the neighbor of candidate solutions in the search space by twisting the individuals in the current population to create a new population for the next GA iteration. As such, *mutation* is performed in the *'exploration'* mode of GALU when the population diversity is lower than the pre-defined threshold. There are two key parameters in the mutation process: the chance of mutation (p_{mutate}) and the level of mutation (p_{flip}). The first parameter determines the probability that mutation occurs on a particular individual. The second parameter dictates the chance of each bit being flipped in the key sequence.

GALU adaptively updates the mutation parameters to maintain the balance between exploration and exploitation. Recall that we define population diversity as the dispersion of individuals in Equation (4). The formulation of diversity closely resembles the notion of statistical variance. We leverage this observation and adjust the mutation parameters by manipulating the bit-wise variance of individuals obtained after crossover. Let us denote an input individual as $\vec{x}$ and its mutated variant as $\vec{y}$. Then the jth bit of the new individual $\vec{y}$ can be considered as a random variable Y_i that is obtained by sampling the original solution $\vec{x}$ with the following probability distribution:

$$prob(Y_i = \vec{y}[i]) = \begin{cases} p_F & \vec{y}[i] = \vec{x}[i] + \eta, \\ 1 - p_F & \vec{y}[i] = \vec{x}[i]. \end{cases} \tag{9}$$

Here, the parameter $p_F = p_{mutate} \cdot p_{flip}$ is the composite probability of flipping a single bit in one individual and η is the random perturbation introduced by the mutation operation. Since the solutions in our problem domain can only take values of 0 or 1, the perturbation has three possible values, i.e., $\eta \in \{0, +1, -1\}$.

Given the PDF defined in Equation (9), we can compute the variance of a single bit as follows:

$$Var[Y_i] = Var[X_i] + p_F \cdot \sigma_\eta^2. \tag{10}$$

The diversity of the mutated population S_K^{t+1} is then computed using Equation (4) by taking the sum of bit-wise variance over the key vector:

$$div(S_K^{t+1}) = div(S_K^t) + k \cdot p_F \sigma_\eta^2, \tag{11}$$

Here, σ_η^2 is the variance of the mutation perturbation. GALU leverages the diversity relation in Equation (11) to determine the mutation parameters. Note that we restrict p_{flip} to a fixed small value and only adjust the mutation probability of each individual p_{mutate}. This is because a greedy bit perturbation might result in the degradation of population fitness, making the GA training unstable. GALU's adaptive mutation operation achieves diversity-aware search space exploration while ensuring stable GA convergence.

5.3 Ensemble-based Logic Unlocking

We discuss how GALU's two-phase framework obtains multiple high-quality keys in the above sections. Here, we focus on the inference stage (i.e., after our GA-based key searching converges) and demonstrate that the ensemble method can further improve the logic unlocking accuracy. More specifically, given a set of high-fitness decryption keys $\left\{\vec{K}_1, \vec{K}_2, ..., \vec{K}_L\right\}$ returned by Algorithm 2 and an unknown input vector $\vec{I}$, we separately apply each key on the encrypted circuit and collect the corresponding output $\vec{O}_i \leftarrow C_e(\vec{I},\ \vec{K}_i)$ where $i = 1, 2, ..., L$. We make the final decision of the circuit C_e's response to the input $\vec{I}$ by taking the element-wise *majority vote* of the output set $\left\{\vec{O}_1, \vec{O}_2, ..., \vec{O}_L\right\}$. Such an ensemble-based logic unlocking strategy yields higher output fidelity compared to the case of using a single key (results given in Section 7.1).

It is worth noting that more advanced decision aggregation rules other than majority vote can be adapted in GALU's ensemble-based logic unlocking scheme. The majority vote does not consider the quality difference of the participating keys and treats each individual's decision equally [28, 31]. This limitation can be alleviated by taking into account the fitness scores of $\left\{\vec{K}_1, \vec{K}_2, ..., \vec{K}_L\right\}$ on the training set and assigning higher weights to the decision of keys with higher fitness.

6 GALU HARDWARE OPTIMIZATION

To improve the attack efficiency of GALU, we develop various hardware optimization techniques to expedite the circuit decryption process. We empirically identify that *circuit fitness evaluation* is the bottleneck of GALU's execution time. To accelerate circuit evaluation, GALU 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, GALU framework automatically constructs the customized auxiliary circuitry to pipeline each computation stage and reduce the runtime. We detail our hardware-level design optimizations as follows.

6.1 GALU Architecture Overview

We take an Algorithm/Software/Hardware co-design approach to speed up the key searching process for the target circuit as outlined in Figure 3. Particularly, GALU maps the netlist of the encrypted circuit with the auxiliary part to the FPGA and performs 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, GALU 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.

GALU Hardware Overview. Figure 4 shows the overview of GALU's architecture. Our hardware design consists of a computing engine for circuit emulation and an auxiliary circuitry for genetic operations. To reduce the data

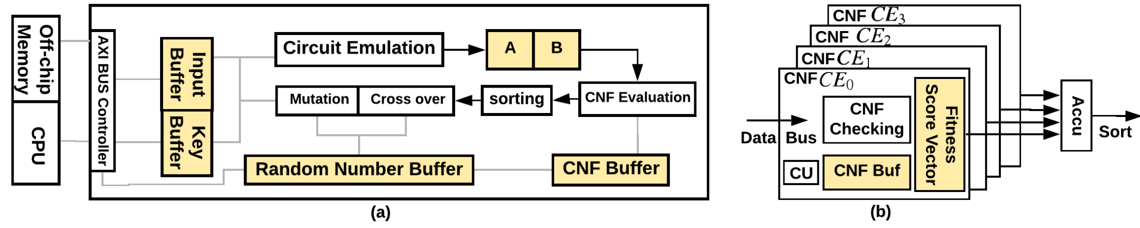

Fig. 4. Overview of GALU hardware design. The overall layout of the hardware system (a) and the implementation of CNF Checking Engines (b) are shown.

communication between the off-chip DRAM and the FPGA, we perform all computations of key evolution on-chip. The non-deterministic nature of our probabilistic GA operators (detailed in Section 5.2) requires random seeds. To this end, GALU stores a set of random numbers pre-computed on CPU using the inherent variation of the operating system. Such a design strategy is more beneficial compared to incorporating a random number generator (RNG) in GALU's hardware design. This is due to the following two reasons: (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. Recall that the fitness score is determined by the ratio of matching observable wires when applying a specific key. Therefore, GALU leverages circuit emulation to inspect how many observable wires have values matching with the expected ones. The results of circuit emulation are used for computing fitness scores using Equation (6) during CNF evaluation. More specifically, in the context of GALU, *CNF evaluation* is defined as the process of computing the ratio of observable wires that have matching values as the expected ones. We evenly partition the workload for each CE offline. The checking process of observable wires in CNF evaluation is parallelized by accommodating multiple Checking Engine (CE) in GALU's design.

Fitness sorting is the main step of our probabilistic population selection step. The sorting engine of GALU permutes the key index based on their corresponding fitness after the accumulation of each individual's fitness score completes. We implement a lightweight sorting engine following the 'even-odd sort' algorithm [12] 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 (4) to reduce computation complexity. Note that this change does not affect the performance of GALU.

We emphasized that GALU does not employ a central control unit to coordinate the entire computation flow. Instead, each part of the design shown in Figure 4 follows a *trigger-based control* mechanism [36]. To be more detailed, each module is controlled by the status flag from its previous computation stage. For example, the sorting engine in GALU 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 3. We detail the design of GALU's circuit emulation and auxiliary circuitry in the following of this section.

6.2 GALU Circuit Emulation

We profile the overhead of GALU's software implementation and empirically observe that circuit evaluation (i.e., obtaining $\vec{O} = C_e(\vec{I}, \vec{K})$) dominates the execution time (results given in Section 7). To improve the attack efficiency, we propose to use circuit emulation for reducing the high latency of evaluating a circuit netlist on CPU. 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, GALU stores the emulation results in a ping-pong buffer and decouples it from the other hardware components as shown in Figure 4. 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.

6.3 GALU Auxiliary Circuitry Design

In this section, we discuss how the auxiliary circuitry is constructed for the target circuit to accelerate the computation in GALU's GA workflow shown in Figure 3.

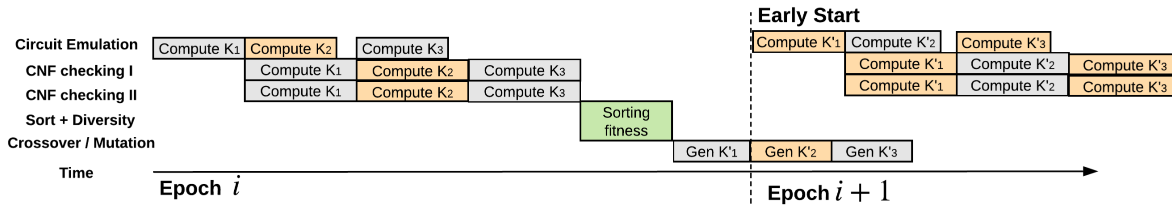

Fig. 5. Pipelining optimization deployed in GALU's genetic algorithm accelerator for logic unlocking.

▪ **Pipeline Evolution Epochs with Early Starting.** GALU's hardware design aims to maximize the time overlapping between each execution stages to increase the throughput of key evolution. As shown in Figure 5, the ping-pong buffer enables the 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 5, 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 $(\vec{I}, \vec{K})$, GALU begins to calculate the fitness score of the key sequence using Equation (6). From the perspective of the hardware, the fitness F_K is computed by accumulating the percentage of observable wires (primary outputs in this work) in C_e that have matching values with the ground-truth (obtained from the active circuit C_o). Independence between different groups of wire signals typically exists in the encrypted circuit. GALU leverages this property by distributing the checking of independent groups of observable wires to different CNF checkers as shown in Figure 4 (b). As such, each CE stores a subset of wires 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 GALU's runtime.

Discussion. It is worth noticing that SAT solving is possible for parallelization and hardware acceleration. Prior works have proposed search space splitting and portfolios for parallel SAT solving [20, 34]. Hardware acceleration for general SAT solving has also been investigated [14, 47]. However, to the best of our knowledge, there is no existing work that demonstrates SAT parallelization or hardware acceleration on the logic decryption problem. As such, we compare GALU's performance with the open-sourced SAT attack proposed in [50] in this paper.

7 GALU EVALUATION

To investigate the performance of GALU, we provide both software and hardware implementation of our framework based on the workflow described in Section 5. Table 1 summarizes the benchmarks evaluated in our experiments, including the Microelectronics Center of North Carolina (MCNC) [9]. We demonstrate the effectiveness and efficiency of GALU compared to prior SAT attacks [50] in Section 7.1 and 7.2, respectively. Furthermore. we corroborate the scalability and generalization capability of GALU in Section 7.3. We discuss how to extend GALU for solving other hardware security tasks in Section 7.4.

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 use python to realize the software implementation of Algorithms 1 and 2. We use the primary outputs of the encrypted circuit as observable wires for fitness computation in this work. The SAT attacks are also executed with only primary outputs observable for comparison. In this work, we use SAT-Compress tool [1] to obtain the CNFs of the encrypted circuits (with Tseitin encoding) for computing the fitness scores F_K in Equation (6). Note that since F_K is defined as the percentage of the observable wires that have matching values with the ground-truths (obtained from the active circuit), GALU can work with any circuit simulation tool (CNF or non-CNF based) for fitness evaluation as long as the simulation tool can obtain values of observable wires when the inputs are applied on the circuit. Experiments are run on an Intel i7-7700k processor with 32 GB of RAM. The energy consumption is measured using *pcm-monitor* utility. Since we empirically identify that the circuit fitness evaluation step is the bottleneck of GALU's online key searching phase, it is beneficial to optimize its computation flow. To this end, we explore the *independence* of computing each individual's fitness scores on the training dataset by *parallelizing* circuit fitness evaluation using *multiprocessing* on CPU. We use the open-sourced code of the SAT attack [50] as our baseline for comparison purposes. Note that [50] is implemented in C++ and assessed on a more powerful CPU (Intel Xeon E31320). As such, our empirical results serve as a *conservative relative speedup* comparison.

We implement our FPGA prototype on the Zynq ZC706 board using the high-level synthesize tool Xilinx SDx 2018.2. GALU's CNF checking engine and the auxiliary GA accelerator discussed in Section 6.1 are implemented using a high-level programming language. Recall that our CNF checking engine obtains the percentage of matching observable wires to compute the fitness score as discussed in Section 6.3. The SDx synthesize tool automatically generates necessary AXI buses for data communication between the off-chip memory and the FPGA. Our design is synthesized using a clock frequency of 100MHz. The power of FPGA implementation is measure at the socket using a power meter during the execution of GALU. Throughout our experiments, we set the number of CEs to $N_{ce} = 16$ and the encryption overhead to 10% with various logic locking schemes in [38] as our default setting.

GA Configurations. As for GALU's GA routine (detailed in Section 5), We generate 100 input/output pairs from the active IC to construct the training data as outlined in Algorithm 1. We use the same approach to generate 500 IO pairs as the validation set. The size of the initial random population is $P = 100$ initial random keys generation and the total number of generations is $G = 50$. The error tolerance of approximate logic unlocking is set to $\epsilon = 0.001$. The total number of maintained individuals in probabilistic population selection is set to $L = 50$. The probability parameters for diversity-driven crossover are set to $p_{cross} = 0.9$ and $p_{exch} = 0.5$. Each pair of parents produces $c = 4$ children during the crossover. To decide whether mutation operation shall be performed in each generation, we set the diversity threshold div_{th} to be half of the value of the initial population. The bit-wise flipping probability is fixed at $p_{flip} = 0.05$. Recall that GALU's adaptive mutation operator dynamically adjusts the mutation probability parameter p_{mutate} using Equation (11) after div_{th} and p_{flip} are set. We determine GA parameters such as L, G, p_{cross} and p_{flip} using *cross-validation*. We adjust GA parameters and run GALU on benchmarks while monitoring the loss values on the training set as well as the validation set. We choose GA parameters that yield a continuously descending validation loss. We empirically found out that the GA parameters work uniformly well across different circuits and do not need to be adjusted for each particular benchmark.

Case Study. To illustrate how GALU creates auxiliary comparator logic for fitness evaluation, we use the $c17$ circuit in ISCAS85 benchmark [23] as an example here. Figure 6 illustrates the structure of the original circuit the logic encrypted variant. We assume the two primary outputs 'PO1' and 'PO2' are observable for our attack in this case. Figure 7 shows the circuit design with GALU's auxiliary comparator logic. The reference values to the comparators ('g_PO1', 'g_PO2' in Figure 7) are obtained by querying the original circuit C_o as explained in Section 5.2. This comparator logic facilitates the computation of the fitness score for a decryption key using Equation (6) in the online key searching stage. We also give an example of the CNF representation for the encrypted circuit $c17$ with auxiliary logic in Figure 8. This CNF description is obtained by SAT-Compress [1] which takes the bench format of the circuit (left part of Figure 8) as input.

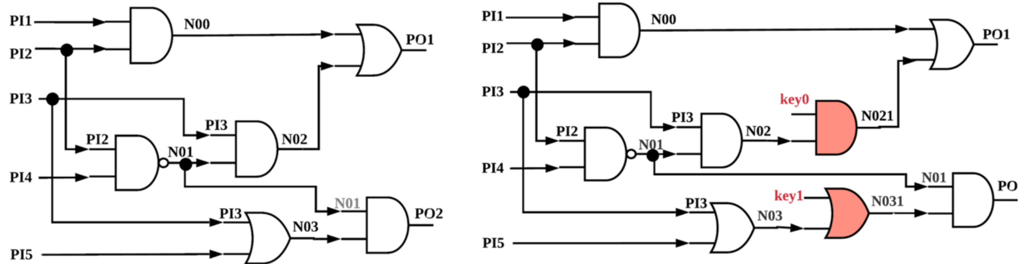

Fig. 6. Circuit diagram of the original benchmark (left) and the encrypted variant (right). The correct decryption key is '10'. The additional key gates are marked in red color and the key inputs are denoted by 'key0', 'key1'.

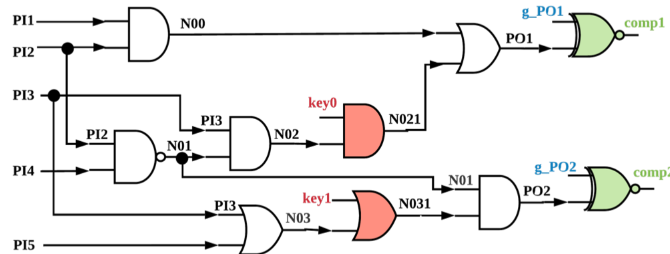

Fig. 7. Encrypted circuit with the auxiliary comparator logic created by GALU. The comparators are marked in green color and the associated ground-truth responses are denoted by 'g_PO1', 'g_PO2'.

```
# key=10                N00 = and(PI1, PI2)
INPUT(PI1)              N01 = nand(PI2, PI4)
INPUT(PI2)              N02 = and(PI3, N01)
INPUT(PI3)              N021 = and(N02, keyinput0)
INPUT(PI4)              N03 = or(PI3, PI5)
INPUT(PI5)              N031 = or(N03, keyinput1)
INPUT(keyinput0)        POO1 = or(N00, N021)
INPUT(keyinput1)        POO2 = and(N01, N031)
INPUT(f_POO1)           comp0 = xnor(POO1, f_POO1)
INPUT(f_POO2)           comp1 = xnor(POO2, f_POO2)
OUTPUT(equal)           equal = and(comp0, comp1)
```

```
c 1 PI1           -1 -2 10 0      10 13 -16 0
c 2 PI2           1 -10 0         -10 16 0
c 3 PI3           2 -10 0         -13 16 0
c 4 PI4           -2 -4 -11 0     -11 -15 17 0
c 5 PI5           2 11 0          11 -17 0
c 6 keyinput0     4 11 0          15 -17 0
c 7 keyinput1     -3 -11 12 0     16 8 18 0
c 8 f_POO1        3 -12 0         -16 8 -18 0
c 9 f_POO2        11 -12 0        16 -8 -18 0
c 10 N00          -12 -6 13 0     -16 -8 18 0
c 11 N01          12 -13 0        17 9 19 0
c 12 N02          6 -13 0         -17 9 -19 0
c 13 N021         3 5 -14 0       17 -9 -19 0
c 14 N03          -3 14 0         -17 -9 19 0
c 15 N031         -5 14 0         -18 -19 20 0
c 16 POO1         14 7 -15 0      18 -20 0
c 17 POO2         -14 15 0        19 -20 0
c 18 comp1        -7 15 0
c 19 comp2
c 20 equal
p cnf 20 35
```

Fig. 8. The bench format of the encrypted circuit (c17) with auxiliary comparator logic (left) and the corresponding CNF representation (right) generated by SAT Compress tool [1]. The auxiliary comparator logic is marked in blue color.

7.1 Attack Effectiveness

We investigate GALU's effectiveness for logic unlocking on the benchmarks in Table 1. Each experiment is repeated 20 times to collect the statistics of the performance metrics (i.e., runtime and energy consumption). The maximum execution time is set to 10 hours (3.6×10^4 seconds). Within this time-bound, GALU is able to unlock 10 out of 10 benchmarks with the best key (i.e., 100% attack success rate), while the baseline SAT method [50] can only break 7 out of 10 benchmarks (70% attack success rate). In other words, GALU framework finds a decryption key that achieves an *ideal output fidelity* $OF = 1$. Figure 14 shows the relationship between GALU's attack success rate (measured by the percentage of the successfully decrypted circuit) and the execution time. We can observe that GALU obtains higher circuit unlocking capability over time.

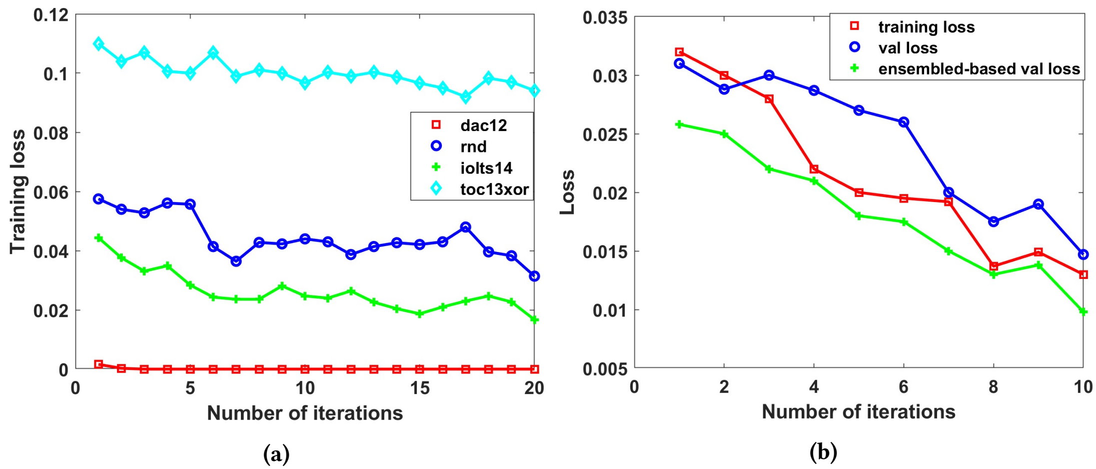

Fig. 9. (a) GALU's circuit unlocking accuracy evolves over time when attacking different logic encryption schemes. (b) Effectiveness of GALU's ensemble-based logic unlocking method using the top three keys.

Conventional SAT-based attack [50] takes very long time (>10 hours) to find distinguishing input patterns on large and complex circuits such as *des*, *c2670* and *c7552* using external SAT solvers [7, 48]. 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 *SAT-hard* (e.g., containing an internal 'and-tree'). Figure 9a shows the learning curve of GALU when attacking four different logic encryption techniques: dac12 [39], rnd [42], iolts14 [18], and toc13xor [40]. The loss of a specific key sequence is computed as $(1 - OF)$ given the IO pairs. We report and visualize the loss metric across 20 runs of all ten benchmarks in Table 1 in Figure 9. One can see from Figure 9a that GALU has intriguing *time-evolving* and *encryption-agnostic* property.

While GALU's convergence speed depends on the adopted logic encryption scheme, our attack framework can always return a set of keys with improving logic unlocking capability over time. As opposed to the SAT attacks, GALU is *generic* and is able to provide approximate keys with high output fidelity for circuits with arbitrary structures. Note that although GALU does not guarantee the global correctness of the decryption key as traditional SAT attacks, our identified approximate key has a high probability of yielding the correct outputs as the active circuit. This generalized unlocking capability of GALU can be seen from Figure 9b where the validation loss has the same trend and similar values as the training loss. Furthermore, our framework provides the adversary with the trade-off between attack accuracy and runtime overhead by tuning the termination parameters G and ϵ in Algorithm 2. Figure 9b shows the effect of GALU's *ensemble-based logic unlocking* discussed in Section 5.3 with the top three key sequences. It can be seen that our ensemble-based unlocking yields a lower error compared to GALU attack with the single best key.

7.2 Attack Efficiency

Figure 10 shows the runtime comparison between GALU's software (Section 5) / hardware (Section 6) implementation and the SAT baseline [50]. Note that we use the average runtime on each benchmark to visualize the performance comparison in Figure 10. Several circuits cannot be decrypted by the baseline algorithm within the pre-defined time-bound (10 hours). In this case, we use 10 hours as the estimated runtime of [50] in Figure 10. With dedicated hardware design support, GALU delivers on average 4.85× speedup compared to the baseline method. For SAT-hard circuits (such as $c2670, c7552, des$), GALU engenders superior performance compared to SAT-based attacks, achieving 236.9×,68.7×, 13.6× speedup on CPU and 1089.2×, 172.4×, 34.8× speedup on the dedicated hardware. Besides the latency comparison, we also measure the power consumption of different circuit deobfuscation methods. The power consumption of 'GALU+HW' on Zynq SoC is measured via the socket when the application is running. On average, GALU with hardware optimization consumes 12.8W power while the software implementation of GALU consumes 51.3W power on CPU. Considering the runtime, the overall energy-efficiency of GALU is 21.8× higher than the SAT-based method.

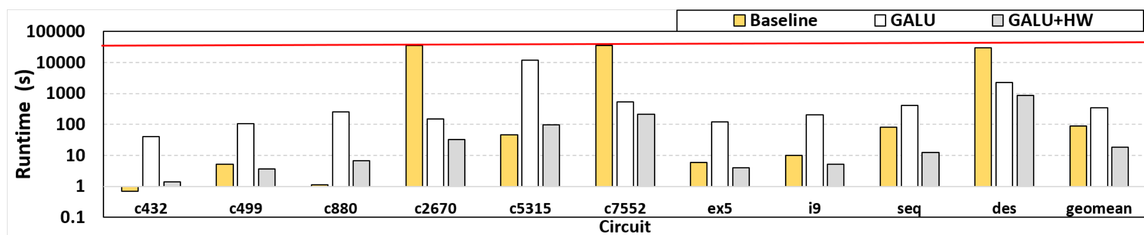

Fig. 10. Comparison of average runtime overhead between GALU and the baseline SAT attack [50]. The software and hardware-accelerated implementation of our framework are denoted by 'GALU' and 'GALU+HW', respectively.

We also compare the performance of GALU and another approximate decryption scheme named AppSAT [43]. AppSAT improves over the basic SAT attack [50] by integrating random queries with distinguishing inputs. The error rate of the decryption is estimated using random patterns. As a result, the attack allows for early stop

and returns an approximate key when the error rate is lower than the pre-defined threshold [43]. We adhere to the attack algorithm and configuration mentioned in [43] for this experiment. Figure 11 illustrates the runtime comparison of GALU and AppSAT on benchmarks locked with dac12 [38] (10% encryption overhead). We use the same decryption error tolerance $\epsilon = 0.001$ for both attacks. The maximal allowed attack time is set to 1000 seconds. One can see from the comparison that GALU performs better than AppSAT on benchmarks *c*2670, *c*5315, *c*7552. In particular, for *c*2670 circuit that has an internal AND tree, GALU identifies an effective approximate key faster than AppSAT [43] since our attack is agnostic to the SAT-hard structure. While AppSAT performs better on the *des* benchmark (also finds the exact correct key) compared to GALU, this is because shrinking the key search space with distinguishing inputs is more effective than our generated training pairs.

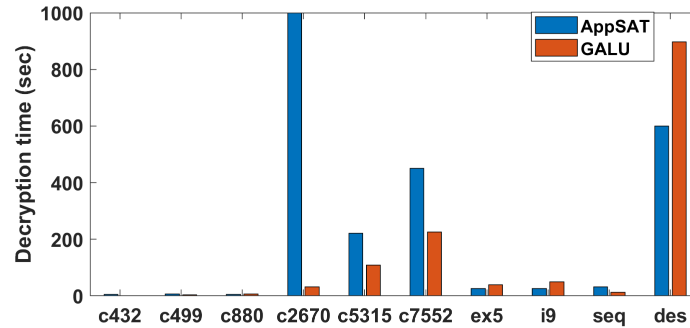

Fig. 11. Runtime comparison of GALU and AppSAT [43] on circuits encrypted with dac12 [38]. The error threshold of approximate decryption is set to $\epsilon = 0.001$ for both attacks.

GALU's resource utilization depends on the key length (k) and the size of the encrypted circuit. Table 2 shows the resource utilization of the assessed benchmark circuits. We evaluate the sensitivity analysis of GALU to these two factors and analyze the underlying reasons in Section 7.3.1.

Table 2. Resource utilization of GALU's auxiliary circuitry under the default setting (10% encryption overhead and 16 CNF checking engines $N_{CE} = 16$) 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)

7.3 Sensitivity Analysis

In this section, we discuss the sensitivity of GALU's performance with respect to the key size, the size of observable wires, the number of CNF checking engines, and the encryption overhead.

7.3.1 Sensitivity to Length of Key and Observable Wires. Figure 12 shows that the resource utilization of GALU with varying key lengths and sizes of observable wires. Note that we only use the primary outputs of the circuit as observable wires in this work. It can be seen that GALU's resource consumption has an approximately *linear* dependency on the length of the encryption key and the length of observable wires. This is due to the fact that a larger number of observable wires require more comparator logic for each CNF checking engine as the index

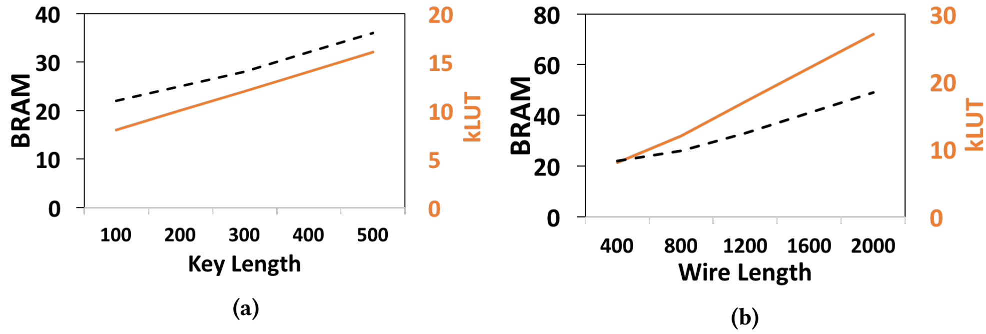

Fig. 12. Resource utilization of GALU's auxiliary circuitry with varying size of the encryption key (a) and observable wires (b). The key length and the wire length is set to 100 and 400 respectively in the initial setting.

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.

7.3.2 Sensitivity to Number of CNF Checking Engines. Figure 13 shows the approximately linear relation between GALU's speedup and the number of CEs. Our system can be scaled up by adding more CNF checking engines to parallel the process of checking observable wires matching as GALU'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. GALU 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 whether the observable wires have matching values compared with the expected ones. 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.

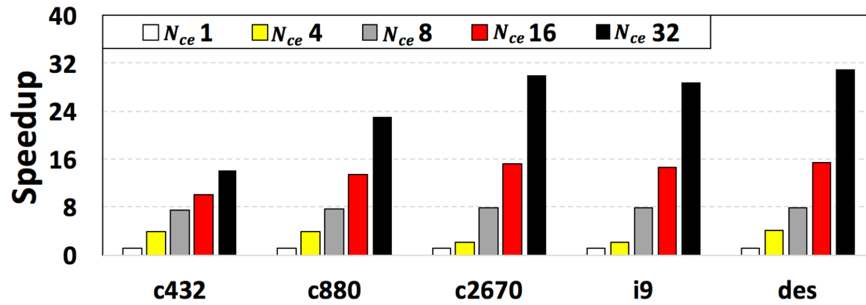

Fig. 13. Scalability of GALU to the number of CNF CEs. The speedup is near-linear with N_{CE} on large circuits where CNF checking is the computation bottleneck.

7.3.3 Sensitivity to Encryption 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 14 shows the execution time of GALU averaged across all ten circuit benchmarks in Table 1 when three different obfuscation overheads are used to encrypt the circuits. One can see that GALU's key searching time does not grow exponentially with the increase of the obfuscation overhead, suggesting the *scalability* of our framework to large circuits.

7.3.4 Sensitivity to Number of Gates. We evaluate the *scalability* of GALU by assessing its runtime sensitivity to the number of gates in the circuit. In addition to the benchmarks listed in Table 1, we created four synthetic circuits with the number of gates equal to 8000, 10000, 15000, and 20000, respectively. Both the number of PIs and

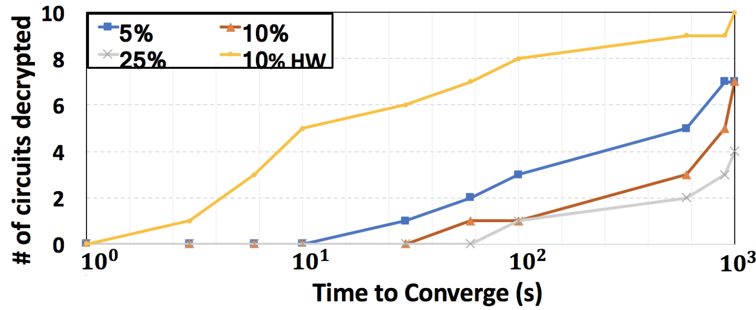

Fig. 14. Sensitivity of GALU's logic unlocking capability (measured by the number of successfully decrypted circuits given a specific GA convergence time) to the logic encryption overhead. The circuits are encrypted using the logic locking technique in [38] with different obfuscation overhead.

the number of POs of these four circuits are set to 250. The gate types and the connections between gates are randomly determined. With this configuration, we create 100 concrete benchmarks for each of the four synthetic circuits and run GALU repeatedly for 20 times to attack each benchmark. We use the same GA parameters mentioned in Section 7 and measure the average runtime of GALU for each synthetic circuit. All circuits are encrypted with the logic locking method in *dac*12 [38]. Figure 15 shows GALU's runtime with varying numbers of gates. It can be seen that the decryption time of our framework increases slowly when the number of gates becomes sufficiently large. We hypothesize that this is because the amount of computation performed by GALU does not increase over generations as opposed to traditional SAT attacks [50].

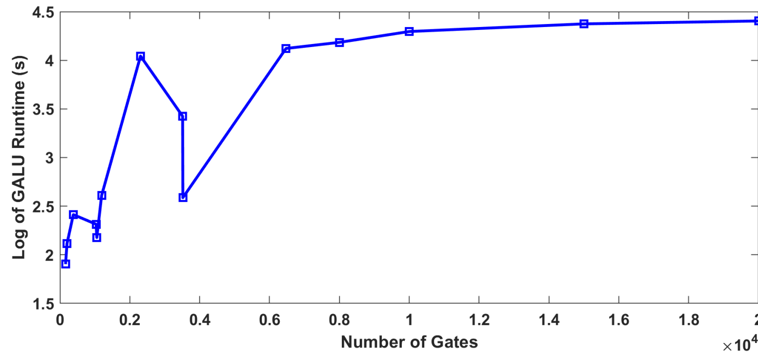

Fig. 15. Evaluation of GALU's runtime with varying number of gates. The circuits are encrypted with DAC'12 [38] using 10% encryption overhead.

7.4 Discussion

Our experimental results demonstrate the effectiveness and efficiency of GALU's framework for approximate logic unlocking. Here, we discuss three future directions to improve GALU.

(i) Advanced Ensemble Methods. Our current framework deploys the majority vote for the proposed ensemble-based logic unlocking as described in Section 5.3. More sophisticated decision aggregation rules can be adapted in GALU to leverage the fitness difference of the participating decryption keys. Such a strategy shall yield higher circuit output fidelity compared to the naive implementation of majority voting.

(ii) Smart Population Initialization. The quality of the initial population has been shown to affect the convergence of genetic algorithms [24, 41]. The current implementation of GALU initializes the population randomly.

Inspired by the above observation, the attacker can design a 'smart initialization' scheme using the information from querying the active IC, thus reducing the key searching time of GALU.

(iii) Domain Adaption. The GA-based framework of GALU can be adapted to resolve other challenges in the hardware security domain. Let us take *functional testing* as an example. Given the golden reference circuit and the unknown circuit under test, we can develop a similar workflow as GALU to generate test input patterns that distinguish the behavior difference between these two circuits.

(iv) Compound Encryption Schemes. We investigate the performance of GALU against four logic locking techniques [18, 39, 40, 42] and show GALU's effectiveness in Figure 9a. Besides these four logic locking methods, point functions are also used to design stronger logic encryption techniques against SAT attacks [55, 56]. The usage of the point functions will make GALU's DIP-inspired training data generation (Section 5.1) more difficult. However, this difficulty is general for the black-box decryption attacks. As the mentioned in [43], a pure point function-based encryption technique has a very low output corruptibility, which is not desired for the IP owner of the circuit. In this scenario, even a random key yields the correct output for all but one input vector, thus GALU is not suitable for an efficient attack. For compound encryption techniques, GALU can be extended to first identify the area locked with traditional logic encryption methods, and then generate distinguishing IO pairs to stimulate the identified area for training data generation.

(v) Attacking Fault-sensitive DNN Hardware. While the large space of DNNs' parameters and activation allows for quantization and pruning with negligible impact on the task accuracy [19, 22], prior works also discover that certain weights of the DNN is sensitive to value perturbation [26, 33]. As a flexible approximate logic unlocking technique, GALU can be extended to attack the DNN hardware if the logic encryption method used by the hardware owner operates on the fault-tolerant portion of the DNN. To this end, the adversary needs to first perform a fault-sensitivity analysis of the victim DNN and identify the regions that feature the fault tolerance property.

8 CONCLUSIONS

Motivated to address the limitations of SAT-based attacks, we present GALU, the first genetic algorithm-based framework for circuit logic unlocking. We develop GALU using an Algorithm/Software/Hardware co-design principle to achieve optimized performance. More specifically, GALU is encryption-agnostic and is generic to arbitrary circuit designs. Unlike SAT that targets to find a single decryption key, our framework yields a set of feasible keys that unlock the obfuscated circuit with an attacker-defined output fidelity. Furthermore, GALU provides the trade-off between attack overhead and output fidelity of the resulting keys. In real-world settings, GALU poses a threat to the rising amount of fault-tolerant applications such as block-chain mining and deep neural networks. We design an *adaptive*, *diversity-guided* genetic algorithm for fast and stable key searching. To further accelerate GALU's implementation, we present *circuit emulation* and *pipelining* as hardware optimization techniques. We perform comprehensive experiments to corroborate the efficiency and effectiveness of GALU across different circuit benchmarks. In future work, we will develop an intelligent decision aggregation policy to improve the accuracy of our ensemble-based logic unlocking, and adapt GALU to address other challenges in the hardware security domain.

REFERENCES

[1] 2012. *SAT-Compress.* https://ddd.fit.cvut.cz/www/prj/SATCompress/man.html

[2] May 22, 2008. *CNF.* https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html

[3] Fahiem Bacchus and Toby Walsh. 2005. *Theory and Applications of Satisfiability Testing: 8th International Conference, SAT 2005, St Andrews, Scotland, June 19-23, 2005, Proceedings.* Vol. 3569. Springer.

[4] Alex Clark Baumgarten. 2009. Preventing integrated circuit piracy using reconfigurable logic barriers. (2009).

[5] R Bhattacharya, S Biswas, and S Mukhopadhyay. 2012. FPGA based chip emulation system for test development of analog and mixed signal circuits: A case study of DC–DC buck converter. *Measurement* 45, 8 (2012), 1997–2020.

[6] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanovic. 2019. Fased: Fpga-accelerated simulation and evaluation of dram. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 330–339.

[7] Armin Biere. 2013. Lingeling, Plingeling and Treengeling entering the SAT competition 2013. *Proceedings of SAT competition* 2013 (2013), 1.

[8] 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

[9] 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.

[10] 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.

[11] Michael Chen, Elham Moghaddam, Nilanjan Mukherjee, Janusz Rajski, Jerzy Tyszer, and Justyna Zawada. 2018. Hardware protection via logic locking test points. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 12 (2018), 3020–3030.

[12] 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.

[13] 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.

[14] John D Davis, Zhangxi Tan, Fang Yu, and Lintao Zhang. 2008. A practical reconfigurable hardware accelerator for Boolean satisfiability solvers. In *2008 45th ACM/IEEE Design Automation Conference*. IEEE, 780–785.

[15] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[16] Bethany Delman. 2004. Genetic algorithms in cryptography. (2004).

[17] 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

[18] Sophie Dupuis, Papa-Sidi Ba, Giorgio Di Natale, Marie-Lise Flottes, and Bruno Rouzeyre. 2014. A novel hardware logic encryption technique for thwarting illegal overproduction and hardware trojans. In *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*. IEEE, 49–54.

[19] Ahmed T Elthakeb, Prannoy Pilligundla, Amir Yazdanbakhsh, Sean Kinzer, and Hadi Esmaeilzadeh. 2018. ReLeQ: A Reinforcement Learning Approach for Deep Quantization of Neural Networks. *arXiv preprint arXiv:1811.01704* (2018).

[20] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. 2010. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation* 6, 4 (2010), 245–262.

[21] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 243–254. https://doi.org/10.1109/ISCA.2016.30

[22] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[23] 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.

[24] Ahmad B Hassanat, VB Prasath, Mohammed Ali Abbadi, Salam Amer Abu-Qdari, and Hossam Faris. 2018. An improved genetic algorithm with a new initialization mechanism based on regression techniques. *Information* 9, 7 (2018), 167.

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

[26] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitraș. 2019. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 497–514.

[27] Mike Hutton, Richard Yuan, Jay Schleicher, Gregg Baeckler, Sammy Cheung, Kar Keng Chua, and Hee Kong Phoon. 2006. A methodology for FPGA to structured-ASIC synthesis and verification. In *Proceedings of the Design Automation & Test in Europe Conference*, Vol. 2. IEEE, 1–6.

[28] Gareth James. 1998. Majority vote classifiers: theory and applications. (1998).

[29] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John

Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 1–12. https://doi.org/10.1145/3140659.3080246

[30] Ahmed Usman Khalid, Zeljko Zilic, and Katarzyna Radecka. 2004. FPGA emulation of quantum circuits. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.* IEEE, 310–315.

[31] Ludmila I Kuncheva, Christopher J Whitaker, Catherine A Shipp, and Robert PW Duin. 2003. Limits on the majority vote accuracy in classifier fusion. *Pattern Analysis & Applications* 6, 1 (2003), 22–31.

[32] Z. Li, Y. Chen, L. Gong, L. Liu, D. Sylvester, D. Blaauw, and H. Kim. 2019. An 879GOPS 243mW 80fps VGA Fully Visual CNN-SLAM Processor for Wide-Range Autonomous Exploration. In *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*. 134–136. https://doi.org/10.1109/ISSCC.2019.8662397

[33] Tao Liu, Wujie Wen, Lei Jiang, Yanzhi Wang, Chengmo Yang, and Gang Quan. 2019. A fault-tolerant neural network architecture. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[34] Ruben Martins, Vasco Manquinho, and Inês Lynce. 2012. An overview of parallel SAT solving. *Constraints* 17, 3 (2012), 304–347.

[35] Elham Moghaddam, Nilanjan Mukherjee, Janusz Rajski, Jerzy Tyszer, and Justyna Zawada. 2016. On test points enhancing hardware security. In *2016 IEEE 25th Asian Test Symposium (ATS)*. IEEE, 61–66.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] Jeyavijayan Rajendran, Huan Zhang, Chi Zhang, Garrett S Rose, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. 2015. Fault analysis-based logic encryption. *IEEE Transactions on computers* 64, 2 (2015), 410–424.

[41] Connie Loggia Ramsey and John J Grefenstette. 1993. Case-Based Initialization of Genetic Algorithms.. In *ICGA*. Citeseer, 84–91.

[42] Jarrod A Roy, Farinaz Koushanfar, and Igor L Markov. 2008. EPIC: Ending piracy of integrated circuits. In *Proceedings of the conference on Design, automation and test in Europe.* ACM, 1069–1074.

[43] 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

[44] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. 2018. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture.* IEEE Press, 764–775.

[45] HISASHI Shimodaira. 1999. A diversity-control-oriented genetic algorithm (DCGA): development and initial experimental results. *Trans. Inf. Process. Soc. Jpn.(Japan)* 40, 6 (1999), 2708–2716.

[46] Takahiro Shinozaki and Shinji Watanabe. 2015. Structure discovery of deep neural network based on evolutionary algorithms. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* IEEE, 4979–4983.

[47] Ali Asgar Sohanghpurwala, Mohamed W Hassan, and Peter Athanas. 2017. Hardware accelerated SAT solvers—a survey. *J. Parallel and Distrib. Comput.* 106 (2017), 170–184.

[48] Niklas Sorensson and Niklas Een. 2005. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT* 2005, 53 (2005), 1–2.

[49] E. Stomeo, T. Kalganova, and C. Lambert. 2006. A Novel Genetic Algorithm for Evolvable Hardware. In *2006 IEEE International Conference on Evolutionary Computation.* 134–141. https://doi.org/10.1109/CEC.2006.1688300

[50] 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.

[51] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. 2017. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. *CoRR* abs/1712.06567 (2017). arXiv:1712.06567 http://arxiv.org/abs/1712.06567

[52] Kit-Sang Tang, Kim-Fung Man, Sam Kwong, and Qun He. 1996. Genetic algorithms and their applications. *IEEE signal processing magazine* 13, 6 (1996), 22–37.

[53] 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.

[54] Rasmus K Ursem. 2002. Diversity-guided evolutionary algorithms. In *International Conference on Parallel Problem Solving from Nature*. Springer, 462–471.

[55] Yang Xie and Ankur Srivastava. 2019. Anti-sat: Mitigating sat attack on logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 2 (2019), 199–207.

[56] 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.

[57] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. 2017. Removal attacks on logic locking and camouflaging techniques. *IEEE Transactions on Emerging Topics in Computing* (2017).

[58] 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.