AdaTest: Reinforcement Learning and Adaptive Sampling for On-chip Hardware Trojan Detection

HUILI CHEN, University of California, San Diego, USA
XINQIAO ZHANG, San Diego State University & University of California, San Diego, USA
KE HUANG, San Diego State University, USA
FARINAZ KOUSHANFAR, University of California, San Diego, USA

This paper proposes AdaTest, a novel *adaptive test pattern generation* framework for efficient and reliable Hardware Trojan (HT) detection. HT is a backdoor attack that tampers with the design of victim integrated circuits (ICs). AdaTest improves the existing HT detection techniques in terms of scalability and accuracy of detecting smaller Trojans in the presence of noise and variations. To achieve high trigger coverage, AdaTest leverages Reinforcement Learning (RL) to produce a diverse set of test inputs. Particularly, we *progressively* generate test vectors with high 'reward' values in an *iterative* manner. In each iteration, the test set is evaluated and adaptively expanded as needed. Furthermore, AdaTest integrates *adaptive sampling* to prioritize test samples that provide more information for HT detection, thus reducing the number of samples while improving the samples' quality for faster exploration.

We develop AdaTest with a *Software/Hardware co-design* principle and provide an optimized on-chip architecture solution. AdaTest's architecture minimizes the hardware overhead in two ways: (i) Deploying circuit emulation on programmable hardware to accelerate reward evaluation of the test input; (ii) Pipelining each computation stage in AdaTest by automatically constructing auxiliary circuit for test input generation, reward evaluation, and adaptive sampling. We evaluate AdaTest's performance on various HT benchmarks and compare it with two prior works that use logic testing for HT detection. Experimental results show that AdaTest engenders up to two orders of test generation speedup and two orders of test set size reduction compared to the prior works while achieving the same level or higher Trojan detection rate.

CCS Concepts: • **Hardware → Test-pattern generation and fault simulation**; *Hardware reliability screening*; *Functional verification*; • **Security and privacy → Intrusion/anomaly detection and malware mitigation**; **Embedded systems security**.

Additional Key Words and Phrases: Hardware trojan detection, logic testing, software/hardware co-design

ACM Reference Format:
Huili Chen, Xinqiao Zhang, Ke Huang, and Farinaz Koushanfar. 2022. AdaTest: Reinforcement Learning and Adaptive Sampling for On-chip Hardware Trojan Detection. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2022), 21 pages. https://doi.org/10.1145/3544015

Authors' addresses: Huili Chen, huc044@ucsd.edu, University of California, San Diego, 9500 Gilman Drive, La Jolla, California, USA, 92093; Xinqiao Zhang, San Diego State University & University of California, San Diego, 9500 Gilman Drive, La Jolla, California, USA, 92093, x5zhang@ucsd.edu; Ke Huang, San Diego State University, 5500 Campanile Drive, San Diego, California, USA, 92182, khuang@sdsu.edu; Farinaz Koushanfar, University of California, San Diego, 9500 Gilman Drive, La Jolla, California, USA, 92093, farinaz@ucsd.edu.

1 INTRODUCTION

Integrated circuits (ICs) are indispensable components for a diverse set of real-world applications including healthcare systems, smart home devices, industrial equipment, and machine learning accelerators [5, 7]. The vulnerability of digital circuits may result in severe outcomes due to their deployment in security-critical tasks. The design and manufacturing process of contemporary ICs are typically outsourced to (untrusted) third parties. Such a supply chain structure results in hardware security concerns, such as sensitive information leakage, performance degradation, and copyright infringement [8, 39]. Malicious hardware modifications, a.k.a., *Hardware Trojan (HT)* attack [2, 38] may occur at each stage of the IC supply chain.

There are two main components in a HT attack: Trojan trigger and payload. The HT *trigger* is a control signal that determines when the malicious activity of the HT shall be activated. The Trojan *payload* is the actual effect of circuit malfunctioning which depends on the purpose of the adversary, e.g., stealing private information or producing incorrect outputs [38]. The attacker intends to design a stealthy HT that remains dormant during functional testing and evades possible detection techniques. As such, the HT trigger is typically derived from the rather rare activation conditions that are easier to hide for the intruder.

To alleviate the concerns about malicious hardware modifications, a line of research has focused on developing effective HT detection methods. Existing HT detection techniques can be categorized into two classes based on the underlying mechanisms: *(i) Side-Channel Analysis* (SCA), and, *(ii) Logic Testing*. SCA-based HT detection explores the fact that the presence of the HT on the victim circuit will change its *physical parameters* (e.g., time, power, and electromagnetic radiation), thus can be revealed by side-channel information [18, 19]. Such a mechanism determines that SCA-based approaches can detect *non-functional* HTs, while they may have high false alarm rates when detecting small HTs due to the operational and physical silicon variation, as well as measurement noise. Logic testing-based techniques intend to activate the stealthy Trojan trigger by generating diverse test patterns [4, 25, 29]. The main challenge of logic testing-based HT detection is to increase the *trigger coverage* with a small number of test patterns.

In this paper, we aim to simultaneously address three challenges of logic testing-based HT detection: effectiveness, efficiency, and scalability. To this end, we propose AdaTest, the first automated **adaptive, reinforcement learning-based test pattern generation (TPG)** framework for HT detection with *hardware accelerator design*. Figure 1 demonstrates the high-level usage of AdaTest to inspect if any hardware Trojans are inserted in the CUT. AdaTest takes the netlist of the circuit under test (CUT) and user-defined parameters as its inputs. A set of test vectors with high reward values are returned as the output of AdaTest.

AdaTest framework consists of two main phases: **(i) Circuit profiling**. Given the circuit netlist, we first characterize each node in the CUT from two perspectives: the *transition probability*, and the *SCOAP testability* measures. These two properties are used to identify rare nodes and quantify the fitness of each node, respectively. **(ii) Adaptive test pattern generation.** AdaTest proposes an innovative reward function for test vectors using the following information: the number of times that each rare node is triggered, the SCOAP testability measure of the rare nodes, and the graph-level distance of the circuit (represented as directed acyclic graph) when applying this test input and the historical ones. In each iteration, AdaTest gradually expands the test set by generating candidate test inputs and selecting the ones that have high reward values. AdaTest provisions a flexible *trade-off* between trigger coverage and test generation time. To enable a hardware-assisted solution, we further design an optimized architecture for AdaTest's implementation to reduce the hardware overhead. More specifically, AdaTest architecture pipelines the computation in online TPG and deploys circuit emulation to accelerate reward evaluation.

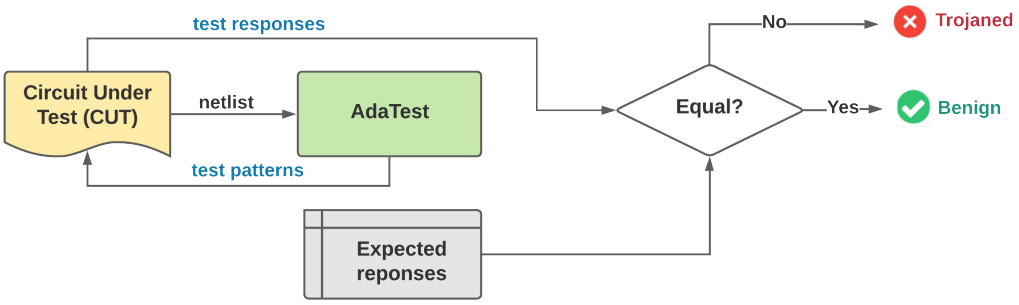

Fig. 1. High-level usage of AdaTest for hardware-assisted security assurance against Trojan attacks.

AdaTest opens a new axis for the growing research in hardware security by exploring the idea of reinforcement learning (RL) and adaptive test pattern generation. The adaptive nature of AdaTest ensures that the quality (measured by our reward function) of our dynamic test set always improves over iterations as new test inputs are added to the test set. Furthermore, AdaTest is *generic* and can be easily extended for other hardware security problems, such as logic verification, efficient ATPG, functional testing, and built-in self-test. For example, the concept of RL and adaptive test pattern generation presented in AdaTest can be used in an efficient ATPG application where the RL reward function is designed to reflect the goal of the ATPG (such as fault coverage of considered fault models).

Organization. Section 2 introduces preliminary knowledge and related works on Hardware Trojan and its detection, as well as reinforcement learning. Section 3 discusses the challenges of HT detection and the overall workflow of AdaTest framework. Section 4 presents our test pattern generation algorithm that combines RL and adaptive sampling for fast exploitation. Section 5 demonstrates our domain-specific architecture design of AdaTest. Section 6 provides a comprehensive performance evaluation of AdaTest on various circuit benchmarks and comparison with prior works on logic testing-based HT detection. Section 7 concludes the paper.

2 PRELIMINARIES AND BACKGROUNDS

2.1 Hardware Trojan Attacks

The security of third-party SoCs has raised an increasing amount of concerns due to the contemporary outsourcing-based supply chain. Hardware Trojans are malicious circuit modifications inserted in the circuit to perform the pre-defined adversarial task ('payload') e.g., circuit malfunction or private information leakage when its control signal ('trigger') is activated. Figure 2 shows an example HT design where a logic-AND gate and an XOR-gate are used as the trigger and payload, respectively. The payload flips the output signal when the trigger is activated, thus disturbing the desired behavior of the original circuit.

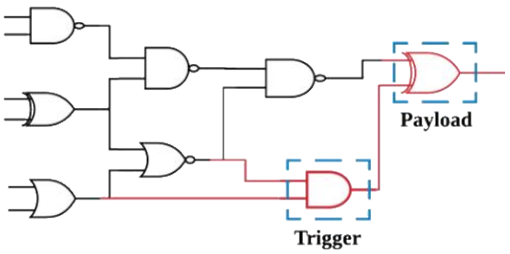

Fig. 2. Demonstration of the Hardware Trojan attack.

The collaborative nature of the supply chain also determines that HTs may be inserted by different parties at different stages of the IC lifecycle. For instance, the untrusted IP provider, the circuit designer, or the manufacturing party might insert HTs in the circuit. Hardware Trojans shall remain *dormant* in most cases to evade functional testing and HT detection, while it should be successfully activated by the trigger to execute the attack. For this purpose, stealthy HTs are designed with two main considerations: (i) Rare conditions are used to construct the trigger signal; (ii) The HT is placed in a non-critical path to minimize its impact on side channels (delay, power, electromagnetic emission, etc.)

2.2 Hardware Trojan Detection

Previous HT detection techniques can be categorized into two broad types: destructive and non-destructive methods. Destructive detection schemes perform de-packaging and de-layering on the manufactured IC to reverse engineer its design layout, thus is prohibitively expensive [9]. Non-destructive HT detection includes two types: run-time monitoring and test-time detection. Run-time approaches monitor the IC throughout its entire operational lifecycle with the goal of detecting Trojans that pass other detection methods, providing the 'last-line of defense'. There are two classes of test-time HT detection techniques. We detail each type as follows:

(i) Side-channel Analysis. SCA-based Trojan detection methods explore the influence of the inserted HT on a particular measurable physical property, such as the supply current, power consumption, or path delay. These physical traces can be considered as the *'fingerprint'* of the circuit and allow the defender to detect both *parametric* and *functional* Trojans [19, 20]. Parametric Trojans modify the wires and/or logic in the original circuit while functional Trojans add/delete transistors or gates in the original chip [14, 24, 42]. However, SCA-based HT detection has two limitations: (i) It cannot detect a small HT that causes a negligible impact on the physical side-channel; (ii) The extracted circuit fingerprint is susceptible to manufacturing variation and measurement noise, thus it might incur high false alarm rates.

(ii) Logic Testing. Compared to the side-channel-based approaches, logic testing methods can only detect *functional* Trojans. However, they yield reliable results under process variation and measurement noise. The main challenge of developing a practical and effective logic testing technique for HT detection is the inordinately large space of possible Trojan designs that the adversary can explore. Since the HT trigger is derived from a very rare condition that is unknown to the defender, attempting to stimulate the stealthy Trojan with a limited number of test inputs is difficult. Existing logic testing methods generate test patterns using simple heuristics, and thus cannot ensure high trigger coverage on complex circuits. Also, such heuristic-driven test generation approaches are inefficient (long test generation time) and unscalable to large benchmarks [2, 4, 38].

Besides SCA and logic testing, other HT detection techniques have also been explored. For instance, FANCI [41] presents a Boolean functional analysis method to identify suspicious wires that are nearly unused in the circuit. For this purpose, FANCI introduces a concept called 'control value' to characterize the influence of a specific wire on other wires. The wires with small control values are flagged as suspicious. However, the wire-wise control value computation in FANCI is unscalable on large circuits. VeriTrust [51] suggests a verification method to detect HT trigger inputs by examining the verification corners. Therefore, VeriTrust is agnostic to the HT implementation styles.

Prior works on logic testing have explored various heuristics to improve trigger coverage while reducing the test generation time. Conceptually similar to the *'N-detection test'* in stuck-at automatic test pattern generation (ATPG), MERO [4] leverages random test vectors and mutates them until each rare node in the circuit is individually triggered at least N times. Such a simple detection heuristic results in an unsatisfying trigger coverage, particularly Trojans that are hard-to-activate. To overcome the limitation of MERO, [29] proposes to use genetic algorithms (GA) and Boolean Satisfiability (SAT) to produce test inputs that excite regular rare nodes and internal *hard-to-trigger* nodes, respectively.

As the end result, [29] achieves a higher trigger coverage compared to MERO, while it is inefficient due to the long test generation time. TRIAGE [25] further improves GA-based test generation by devising a more appropriate 'fitness' function that incorporates the controllability and observability factors of rare nodes. However, the GA nature of TRIAGE limits its efficiency for test input space exploration and the resulting test set might be unnecessarily large. TGRL [26] suggests training a machine learning model for test patterns generation that combines rare signal stimulation as well as controllability/observability analysis. Although TGRL claims to explore reinforcement learning, its test pattern generation pipeline (Alg.3 in [26]) does not involve sequential decision-making in standard RL techniques. Instead, TGRL learns an ML model via stochastic gradient descent for TPG.

2.3 Reinforcement Learning

Reinforcement learning [13, 36, 43] is a machine learning technique that is capable of solving complex problems in various domains. RL works *sequentially* in an environment by taking an action, evaluating its reward, and adjusting the following actions accordingly. In particular, an RL paradigm involves an *agent* that observes the environment and takes *actions* to maximize the *reward* determined by the problem of concern [23, 36]. Figure 3 shows the interaction between the agent and the environment in the RL paradigm.

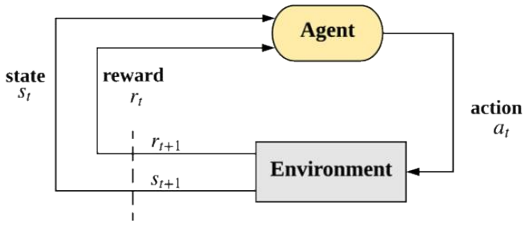

Fig. 3. Illustration of the agent-environment interaction in reinforcement learning.

We introduce the key concepts in an RL system below:

■ **Action Space.** The action space is a set of possible moves that the agent can take to change to a new state. For example, in a video game, an action can be running left/right, or jumping high/low.

■ **Environment.** The environment takes the agent's current state and action as input, and returns the reward and the next state as the output. Depending on the problem domain, the environment might be a set of physical laws or chemical reaction rules that processes the actions and establish the corresponding outcomes.

■ **State.** A state is a concrete and instantaneous situation in which the agent finds itself. This can be an instant configuration, a particular place and a moment that puts the agent in connection with other influential objects in the environment, such as opponents or awards. It is noteworthy that a state needs to contain all information to ensure the system satisfies the *Markov property* [28].

■ **Observations.** The agent can obtain observations (emission of states) from the environment. In particular, the observation is a (stochastic) function of the state.

■ **Reward.** The reward is a numerical value that evaluates the fitness (success or failure) of an agent's actions in a given state. From a given state, an agent takes actions in the environment and acquires the new state as well as the reward from the environment. A *cumulative reward* is defined as the summation of discounted rewards: $G(t) = \sum_{k=0}^{n} \gamma^k R(t+k+1)$. The discount factor γ ($0 \leq \gamma \leq 1$) tunes the importance of future rewards for the current state. The key idea of RL is to find a series of actions that maximize the expected cumulative reward.

▪ **Policy.** The policy of a RL algorithm is typically defined within the context of Markov decision process [36]. Given the state information, policy is the suggested action that the agent shall take in order to obtain a high reward.

Our objective is to develop an adaptive test pattern generation framework for *logic testing* with high Trojan coverage and small test set size. Therefore, AdaTest belongs to the test-time detection category introduced in Section 2.2. We choose RL over other machine learning techniques (e.g. neural networks) since the reward-oriented and progressive nature of RL makes it appealing for our goal. Furthermore, to reduce the complexity of RL, AdaTest integrates adaptive sampling to prioritize test patterns that provide more useful information for HT detection.

3 ADATEST OVERVIEW

In this section, we first discuss the limitations of prior works on Hardware Trojan detection and our motivation (Section 3.1), then introduce our assumptions and threat model for AdaTest framework (Section 3.2). We demonstrate the overall workflow of AdaTest test pattern generation technique in Section 3.3. AdaTest is a hardware-friendly framework and we present our architecture design in Section 5.

3.1 Motivation and Challenges

Prior works have advanced logic testing-based Trojan detection using various techniques [4, 25, 29]. We discuss the limitations of these detection schemes below.

MERO. Inspired by the traditional 'N-detect' test used in stuck-at ATPG, MERO [4] generates random test vectors to activate each rare node (identified as nodes with transition probability smaller than the threshold θ) to the corresponding rare value at least N times. MERO has three main disadvantages: (i) Triggering all rare nodes for N times might be very time-consuming or even impractical; (ii) It yields low trigger coverage for hard-to-trigger Trojans; (iii) It only explores a small number of test vectors in the entire possible space due to its bit mutation and test vector selection policy.

ATPG based on GA+SAT. The paper [29] combines genetic algorithms and SAT in test pattern generation for HT detection. While it improves the trigger coverage compared to MERO, [29] has two constraints: slow test set generation and large memory footprint.

TRIAGE. The paper [25] proposes TRIAGE that integrates the benefits of MERO and [29]. TRIAGE leverages the SCOAP testability parameters and advises the fitness function of GA for HT detection. However, the evolutionary nature of GA determines that TRIAGE might be 'trapped' in the vicinity of a local optimum, thus exploring only a small portion of the full test input space.

We present AdaTest as a holistic solution to address the limitations of the previous works. To this end, we identify three main challenges of developing an efficient and effective logic testing-based HT detection technique as follows:

(C1) High trigger coverage. The test vector set shall yield a high trigger coverage rate to ensure that the probability of activating the stealthy Trojan is large. This property is critical for the *effectiveness* criterion of HT detection.

(C2) Efficient test generation. The runtime overhead of test pattern generation shall be reasonable while attaining a high trigger coverage. For hardware-assisted security, this implies that a test set with a smaller size is preferred. This requirement assures the efficiency and practicality of the HT detection method, particularly on large circuits.

(C3) Scalable to large benchmarks. The runtime consumed by the test pattern generation technique shall not scale exponentially with the size of the examined circuit.

AdaTest tackles the above challenges $(C1) \sim (C3)$ using an *adaptive, RL-based* input space exploration approach. Furthermore, we provide architecture design for AdaTest-based TPG in Section 5 to enable hardware-assisted security. We empirically corroborate the superior performance of AdaTest compared to the above counterparts in Section 6.

3.2 Threat Model

As shown in Figure 2, HTs consists of two parts: trigger and payload. Figure 2 shows an example of HT design. AdaTest is applicable to both combinational and sequential circuits. One can unroll sequential circuits into combinational ones and apply AdaTest for test pattern generation. Without the loss of generality, we assume that the adversary uses a logic-AND gate as the Trojan trigger that takes a subset of rare nodes as its inputs. An XOR gate is used to flip the value of the payload node when the trigger is activated (i.e., each of the trigger nodes has a logical value '1').

We make the following assumptions about AdaTest framework:

(i) The defender knows the netlist of the circuit under test. We assume the party that executes logic testing has the netlist description of the circuit to be examined. This netlist can be obtained by performing de-packaging, de-layering, and imaging [10, 17, 22, 40] on the physical circuit. While hardware obfuscation techniques such as camouflaging [16, 34, 35, 47] and logic encryption [37, 45, 48, 49] could make the trigger design of the Trojan harder to identify, we consider the scenario where the circuit under test is not encrypted in our threat model since this setting is also used in previous Trojan detection papers [4, 26, 33, 46].

(ii) The defender can observe the 'indication signal' when the Trojan is activated. We assume the defender can observe certain *manifestations* of the hidden Trojan when it is activated. In particular, we assume the defender knows the correct response of the CUT to a given test input and observes the primary outputs of the CUT for comparison. Note that AdaTest is compatible with techniques that increase manifestation signals (e.g., test point insertion).

3.3 Global Flow

Figure 4 illustrates the global flow of AdaTest. We discuss the threat model in Section 3.2. AdaTest framework consists of two stages: (i) Circuit profiling phase (offline) that computes the transition probabilities and SCOAP testability parameters of the netlist; (ii) Adaptive RL-based test set generation phase (online) that progressively identifies test vectors with high reward values.

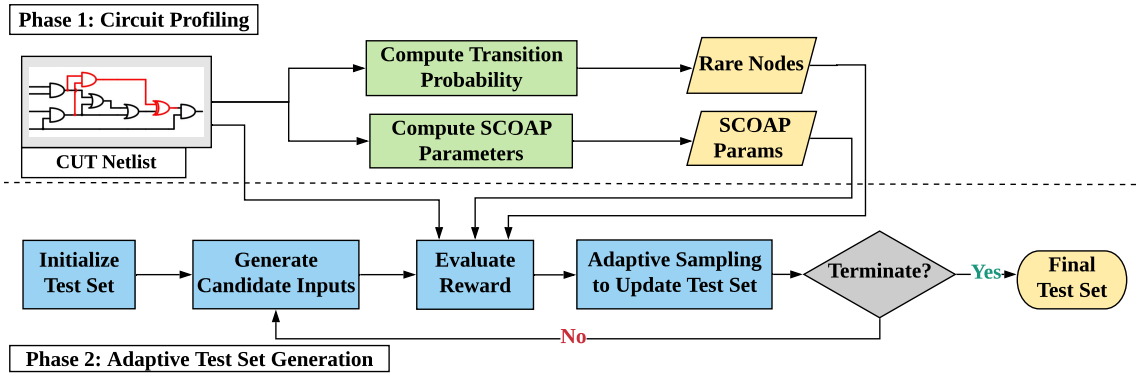

Fig. 4. Global flow of AdaTest framework for Hardware Trojan detection.

Phase I: Circuit Profiling. This stage includes the following:

(1) Compute Transition Probabilities. Given the netlist of the circuit under test, AdaTest first computes the *transition probability* of each internal node in the netlist. In particular, we use the method in [30] and assume that each primary input has an equal probability of taking a logical value of 0 and 1. We make this assumption about the primary input values since previous Trojan detection papers [2, 15, 30, 44] use the same assumption when computing the transition probability. Mathematically, the transition probability of a node is computed as $P_{trans} = p(1-p)$ where

$p = Prob(node = 1)$. P_{trans} of each node is then compared with a pre-defined threshold θ to identify the *rare nodes*. Identifying rare nodes is important for HT detection since the defender does not know the exact set of trigger nodes used by the attacker. As such, the activation status of rare nodes provides guidance to generate test inputs that are likely to trigger the stealthy Trojan.

(2) Compute SCOAP Testability Parameters. Controllability and observability are important testability characteristics of a digital circuit. More specifically, *'controllability'* describes the ability to establish a specific node to 0 or 1 by setting the primary inputs. *'Observability'* defines the capability of determining the value of a node by controlling the circuit's inputs and observing the outputs. The *testability* parameters are useful for Trojan detection since they allow AdaTest to distinguish the quality of different rare nodes.

Phase II: Adaptive RL-based test pattern generation. After the CUT is profiled offline in Phase 1, AdaTest performs adaptive test input generation as shown in the bottom of Figure 4. We outline each step as follows:

(1) Initialize Test Set. AdaTest first generates an initial test vector set that is used in the later steps. A naive way to do so is random initialization, which may not be optimal for HT detection. To improve the trigger coverage in the later runs, AdaTest employs SAT to find a number of test inputs that activate a subset of rare nodes. We call this method *'smart initialization'* and empirically corroborate its effectiveness in Section 6.1.

(2) Generate Candidate Test Inputs. In each iteration of AdaTest's adaptive test vector generation, we first produce a sufficient number of candidate test input patterns that might improve the detection performance when added to the current test set. AdaTest deploys random test generation for this purpose.

(3) Evaluate Reward Function. AdaTest applies the candidate test inputs on the examined circuit and collects the observations, i.e., the netlist status represented as a directed acyclic graph (DAG). We incorporate the transition probabilities and the SCOAP testability parameters from Phase 1 as well as a novel DAG-level diversity measure to define our reward function.

(4) Adaptive Sampling to Update Test Set. Inspired by the selection step in genetic algorithms, we design an adaptive sampling module that picks 'high-quality' test patterns for fast and efficient input space exploration. In particular, after computing the reward value of each test input in the candidate test vectors, AdaTest selects the ones with the highest scores and append them to the current test set.

At the end of each iteration, AdaTest checks the termination condition and decides whether or not the progressive test generation process shall continue.

Performance Metrics. We use *effectiveness* and *efficiency* as two main metrics to assess the performance of a Trojan detection scheme. In particular, we measure the effectiveness from two aspects: trigger coverage and Trojan coverage (i.e. detection rate). The efficiency property is measured by the test set generation time and test set size. AdaTest, for the first time, provides the trade-off between effectiveness and efficiency by adaptively generating a set of test patterns with evolving quality over time. The quantitative analysis of the above metrics is demonstrated in Section 6.

4 ADATEST ALGORITHM DESIGN

The key to ensuring a high probability of Trojan detection using logic testing is to generate a test set that can trigger the circuit to diverse states, in particular, the rare nodes in the circuit. To this end, AdaTest leverages three important characteristics of the circuit: the transition probabilities, the SCOAP testability measures, and the DAG-level diversity. In particular, AdaTest employs an *RL-driven* test pattern generation approach that uses the above three properties to progressively generate test inputs. Inspired by the selection stage in genetic algorithms, we integrate an adaptive

sampling module that progressively expands the current test set (used as historical information) with high-quality test patterns. This **response-adaptive** design is beneficial for statistical search of the HT trigger in the circuit input space, thus improves the efficiency of AdaTest's RL-based pipeline. We detail the two main phases of AdaTest shown in Figure 4 in the following of this section.

4.1 Circuit Profiling

Algorithm 1 outlines the steps of the circuit profiling phase in AdaTest. This stage obtains two informative properties of the circuit: the transition probabilities and testability measures. In particular, we use *random testing* and *logic simulation* to estimate the transition probability P_{trans} of each node in the netlist C_n. To further investigate the rewards of different rare nodes, AdaTest also computes the SCOAP parameters of the nodes using the technique in [11].

AdaTest's circuit profiling stage characterizes the *static reward* properties of the circuit in terms of the transition probabilities of rare nodes and testability measures. We call these two properties *'static'* since they are *independent* of the circuit input for a given circuit netlist. As such, our profiling phase can be performed *offline*. The above two properties are indispensable for the reward computation step in Phase 2 of AdaTest since: (i) Transition probabilities and rare nodes shed light on the potential trigger nodes exploited by the malicious adversary. The defender knows that a subset of rare nodes are used to design the stealthy Trojan while he has no knowledge about the exact trigger set. As such, rewarding the activation of rare nodes encourages the test vectors to stimulate the possible HT. Note that the Trojan activation condition is equivalent to knowledge of the exact trigger set and both are assumed to be unknown to the defender. (ii) Testability parameters provide more fine-grained information about the quality of individual rare nodes in the context of HT detection. One can compare the fitness of two test inputs by counting and comparing the number of activated rare nodes corresponding to each test vector. However, such a naive counting mechanism neglects the intrinsic difference between the quality of individual rare nodes. In principle, a rare node with higher controllability and observability shall be assigned with higher reward values. As such, AdaTest integrates the SCOAP testability measures to quantify the reward of each activated rare node.

Algorithm 1 Circuit Profiling.

INPUT: Netlist of the circuit under test (C_n); Number of random tests (H); Threshold on transition probability (θ) for rare nodes.

OUTPUT: The set of rare nodes (R); Computed testability parameters TP = ($CC0, CC1, CO$).

Initialize rare node set: $R \leftarrow \emptyset$
Generate random inputs: $I \leftarrow RandGen(C_n,\ H)$.
Perform logic simulation: $O \leftarrow LogicSim(C_n,\ I)$.
for node in C_n **do**
    Compute frequency: $p = CountOnes(O, node)/H$
    Estimate transition probability: $P_{trans} = p(1-p)$
    if $P_{trans} < \theta$ **then**
        $R \leftarrow R \cup node$
Obtain SCOAP parameters:
    $(CC0,\ CC1,\ CO) \leftarrow ComputeSCOAP(C_n)$
Return: Obtained rare node set R, SCOAP testability parameters $TP = (CC0, CC1, CO)$.

4.2 Adaptive RL-based Test Pattern Generation

AdaTest deploys a *progressive, reinforcement learning-driven* algorithm for efficient and effective test input space exploration with the goal of HT detection. Section 2.3 introduces the basic concepts of RL. We discuss how we map the Trojan detection problem to the RL paradigm as follows.

AdaTest's RL Formulation of Trojan Detection:

■ **State.** The objective of AdaTest is to adaptively generate test patterns with high effectiveness for Trojan detection in an iterative manner. As such, AdaTest defines a *state* as the *current test set* in the present iteration.

■ **Action Space.** Recall that an action transforms the agent into a new state, which is the new test set according to our definition of the state above. Therefore, a feasible *action* for AdaTest is to *identify a set of new test input vectors* in each iteration that improves the quality of HT detection when added to the current test set.

■ **Environment.** For HT detection, the *netlist of the circuit* (C_n) can be considered as the *environment* that converts the current state and the action, and returns the reward value.

■ **Observations.** The agent makes the observation of the environment before reward computation. For Trojan detection problems, we model the *DAG formed by the values of all nodes* in the netlist given a specific input vector as an *observation* of the circuit state.

■ **Reward.** The definition of the reward function directly reflects the objective of the problem that one aims to solve. As such, for the task of logic testing-based HT detection, AdaTest designs a *composite reward* function to encourage the generation/exploration of test inputs that facilitate the excitation of the potential HT.

The mathematical definition of AdaTest's *dynamic* reward function is given in the equation below:

$$\begin{aligned} Reward(T_i|\ S_i) = \lambda_1 \cdot V_{rare}(T_i,\ R) + \lambda_2 \cdot V_{scoap}(T_i,\ R,\ TP) \\ + \lambda_3 \cdot V_{DAG}(T_i|\ S_i). \end{aligned} \tag{1}$$

Here, S_i and T_i are the current test set (i.e., the state) and the newly generated test inputs in i^{th} iteration, respectively. R and TP are the set of rare nodes and the SCOAP testability parameters identified in Phase 1 (*static attributes*). The hyper-parameters $\lambda_1, \lambda_2, \lambda_3$ determine the relative weighting of the three reward terms. The reward function $Reward(T_i|\ S_i)$ characterizes the fitness of the specific test inputs T_i while considering the current test set S_i. Evaluating the reward value of T_i *in the context of the historical test patterns* (S_i) makes AdaTest's RL framework *adaptive* and intelligent.

We detail how each term in AdaTest's reward function is designed below. Inspired by the 'N-detect' test, the first reward term in Equation (1) aims to activate each rare node in the circuit for at least N times. To this end, we define the **rare node reward** R_{rare} as follows:

$$V_{rare}(T_i,\ R) = -\sum_{r \in R} abs(N - Ctr_i(r)), \tag{2}$$

where $Ctr_i(r)$ is the number of times that the rare node r is activated to its rare value up to the i^{th} iteration.

The second reward term in Equation (1) leverages the SCOAP parameter $TP = (CC0, CC1, CO)$ computed in Phase 1 to encourage the stimulation of rare nodes with high controllability and observability. Given the current test set S_i, we can obtain the set of activated rare nodes Rtr_i (which is a subset of R). The **SCOAP testability reward** V_{scoap} is then computed as follows:

$$V_{scoap}(T_i,\ R,\ TP) = \sum_{r \in Rtr_i} CC(r) + CO(r). \tag{3}$$

Here, $CC(r)$ and $CO(r)$ denote the controllability and observability of the rare node r when set to its rare value. More specifically, $CC(r)$ shall be converted to $CC0(r)$ or $CC1(r)$ depending on the rare value of the node r.

Besides leveraging the static attributes identified in Phase 1 to define the rare node reward R_{rare} and the SCOAP testability reward R_{scoap}, AdaTest further explores the **graph-level diversity** extracted from the circuit netlist. In particular, AdaTest identifies the dynamic fitness property, i.e., the DAG-level diversity that is jointly determined by the circuit netlist and the test vector set. Such a DAG-level distance serves as a *dynamic* fitness measure since it is *input-aware*. Recall that AdaTest leverages an RL paradigm and considers the value assignments of all nodes when given the netlist C_n and a specific test input as the observation. We use the *graph representation* of the circuit to abstract the observed netlist status. To facilitate the computation, AdaTest flattens the DAG to an *ordered sequence* based on the circuit level information. The distance between the two transformed DAG sequences is used as the DAG-level diversity measure. To summarize, we define the **DAG diversity reward** as follows:

$$V_{DAG}(Ti|\ S_i; C_n) = HammDist(DAG(T_i;\ C_n),\ DAG(S_i;\ C_n)). \tag{4}$$

Here, $DAG(T_i; C_n)$ denotes the flattened ordered sequence of the DAG obtained when applying the test inputs T_i to the circuit C_n. The diversity measurement function $HammDist$ computes the normalized pairwise distance of the flattened DAGs using the Hamming distance metric. Since the DAG sequence of the circuit is binary-valued (0 or 1), AdaTest employs XOR function as an efficient implementation of the $HammDist$ function. It's worth noting that this graph reward V_{DAG} is aware of historical test inputs (S_i), thus providing guidance to select new inputs that stimulate different internal nodes structure in the context of current test inputs S_i.

■ **Policy.** The policy component of a RL algorithm suggests actions to achieve a high reward given the current state. Recall that AdaTest defines the state and the action space as the current set of test vectors and the expansion with the new test patterns, respectively. Therefore, the policy module of AdaTest selects the most suitable test pattern candidates and add them to the result test set (line 5&6 in Alg. 2).

Algorithm 2 outlines the procedure of our adaptive test set generation framework. We emphasize that **AdaTest does not require explicit training** on the training set, which is typically required by machine learning models (e.g., gradient descent-based training). The RL nature enables AdaTest to search for distinguishing test inputs with the guidance of the composite reward. This makes our detection method fundamentally different from TGRL [26] that still trains an ML model for test pattern generation. We discuss how AdaTest leverages the RL paradigm formulated above to achieve logic testing-based HT detection in the following of this section.

❶ **Smart Initialization.** Recall that the intuition of logic testing-based Trojan detection is to encourage the generation of test inputs that activate diverse combinations of rare nodes to their corresponding rare values. Random test vectors might be unlikely to yield a high trigger coverage, especially on large circuits. To explore the above intuition, AdaTest leverages SAT to generate the initial test set (line 1 in Algorithm 2) such that it is able to activate diverse rare nodes specified by the defender. We empirically validate the advantage of our smart initialization as opposed to the random variant in Section 6.1. It is worth noticing that while the defender can identify rare nodes in the circuit by thresholding the transition probabilities, it might be infeasible to find an input that stimulates all rare nodes to their rare values. Therefore, AdaTest tries to generate test patterns that stimulate different combinations of rare nodes for Trojan detection.

❷ **Generate Candidate Test Patterns.** AdaTest progressively identifies test inputs that are suitable for HT detection using an iterative approach. To this end, AdaTest first generates a sufficient number of candidate test vectors at the beginning of each iteration (line 3 in Alg. 2). These candidates are responsible for exploring the test input space and aim to find solutions with high rewards. In our experiments, we adopt an adaptive sampling method to generate candidate test patterns at each iteration. In particular, the sampling weights for the test vectors in the initial set S_0 are uniformly assigned at iteration 0. In other words, at iteration 0, we perform a uniform sampling to generate candidate test patterns.

Algorithm 2 Adaptive Reinforcement Learning based Test Input Pattern Generation.

INPUT: Netlist of circuit under test (C_n); Rare node set R; SCOAP testability parameters $TP = (CC0, CC1, CO)$; Size of candidate test inputs per iteration (M); Size of selected test inputs per iteration (L); Maximal number of iterations (I_{max}); Percentage threshold of rare nodes (p); Target activation times (N).

OUTPUT: A set of test patterns S for Trojan detection of the target circuit C_n.

1: Initialization:
   $S_0 = \left\{\vec{S}_0^1, ..., \vec{S}_0^L\right\} \leftarrow SmartInitialize(L)$.
   Iteration counter: $i \leftarrow 0$
2: **while** $i < I_{max}$ and HT is not activated **do**
3: $T_i \leftarrow GenerateTestCandidates(M;\ C_n)$
4: $Reward(T_i|\ S_i) \leftarrow EvaluateReward(T_i,\ S_i;\ C_n)$
5: $T_i^{top} \leftarrow SelectTopCandidates(T_i,\ Reward,\ L)$
6: Update test set: $S_{i+1} \leftarrow S_i \cup T_i^{top}$ ▷ Adaptive sampling to expand test set
7: $A_i \leftarrow CountRareNodeActivation(S_i;\ C_n)$
8: **if** $p\%$ elements in $A_i \geq N$ & $A_i.min() \geq 1$ **then** ▷ Check termination condition
9: break
10: $i \leftarrow i + 1$
11: **Return:** Obtained a test set (S_i) for logic testing-based HT detection of the circuit C_n.

Then the sampling weights of test vectors at iteration $i + 1$ will be updated based on the normalized reward values evaluated at iteration i. Test vectors with higher reward values will result in higher sampling weights, which in turn increases the probability of the test vectors being included in the generated set S. The adaptive sampling method allows us to optimize test pattern generation by favoring test patterns with higher reward values thus enhancing convergence in our test pattern generation.

❸ **Evaluate Reward Function.** The definition of reward is task-specific. Since our objective is to generate test patterns that stimulate the circuit (particularly the rare nodes) to different states for Trojan detection, AdaTest designs an innovative composite reward function as shown in Equation (1). In each iteration, the reward values of the candidate test inputs are evaluated (line 4 of Alg. 2). Our compound reward function captures informative features that are beneficial for HT detection from three aspects: the number of times that each rare node is activated (V_{rare}), the SCOAP testability measures that quantify the fitness of different rare nodes (V_{scoap}), and the graph-level diversity between the current test inputs and historical ones (V_{DAG}).

❹ **Adaptive Sampling to Update Test Set.** Recall that in AdaTest's RL paradigm, the current test set S_i represents the 'state' variable. After obtaining the reward values of individual candidate test input in T_i from Step 3, AdaTest updates the state by selecting a subset of T_i that has the highest reward values and adding them to the current test set S_i. This step is conceptually similar to the selection stage in genetic algorithms. With the domain-specific definition of reward, AdaTest adaptively samples high-quality test patterns from the randomly generated candidate test inputs, therefore facilitating fast exploration of the circuit input space for HT detection.

❺ **Check Termination Condition.** AdaTest's adaptive test set generation terminates if any of the following three conditions is satisfied: (i) $p\%$ of all rare nodes are activated for at least N times and all rare nodes are activated at lease once (line 8 in Alg. 2); (ii) The maximal number of iteration I_{max} is reached (line 2 in Alg. 2); (iii) The current test set S_i activates the hidden Trojan, i.e., all involved trigger nodes are activated to their corresponding rare values by S_i

(line 2 in Alg. 2). Note that we include termination condition (iii) since our threat model assumes that the defender can observe the manifestation of an activated Trojan.

Discussion. As summarized in Alg. 2, our reinforcement learning approach does not require model training. Instead, we progressively generate the set of test vectors using adaptive sampling given the particular circuit with the goal of maximizing the RL rewards for Trojan detection. From this perspective, our RL-based detection tool generates a specific test set for the circuit under test. However, AdaTest is generic in the sense that it is agnostic to the circuit structure and can be applied to various types of circuits. In other words, applying AdaTest to a different circuit does not require any model training since we do not incorporate neural networks in our RL detection pipeline shown in Alg. (2).

5 ADATEST ARCHITECTURE DESIGN

Beyond the novel test generation algorithm discussed in Section 4, we design a Domain-specific systems-on-chip (DSSoC) architecture of AdaTest for its practical deployment. The bottleneck of AdaTest implementation is the computation of the test input's reward $Reward(T_i|S_i)$ according to Equation (1). Given the rare node-set R and SCOAP testability measures of the circuit TP from offline circuit profiling (Algorithm 1), the online reward evaluation of a new test input T_i involves three terms as shown in Equation (1): identifying the rare nodes stimulated by T_i (for V_{rare}), obtaining the SCOAP values corresponding to each active rare node (for V_{scoap}), and computing the DAG-level graph distance (for V_{DAG}). Note that the third component requires us to obtain the DAG with nodes value assignment when applying the test input on the circuit $DAG(T_i;\ C_n)$. This information is also sufficient to compute the first two reward terms. Therefore, the main task for AdaTest's on-chip implementation is to obtain the value-assigned DAG for a new test input on the circuit ($DAG(T_i;\ C_n)$).

To accelerate circuit evaluation, AdaTest deploys *circuit emulation* on the programmable hardware to obtain the response $DAG(T_i;\ C_n)$. Furthermore, AdaTest constructs the customized auxiliary circuitry automatically to pipeline each computation stage and reduce the runtime overhead. We design an optimized DSSoC architecture of AdaTest for efficient implementation of our adaptive TPG method outlined in Algorithm 2.

5.1 Architecture Overview

The overall hardware architecture of AdaTest's online test patterns generation is shown in Figure 5 (a). AdaTest leverages Algorithm/Software/Hardware co-design approach to accelerate the test inputs searching process shown in Figure 4 (phase2). More specifically, AdaTest maps the netlist of the circuit under test (C_n) with the auxiliary part to the FPGA and performs circuit evaluation to obtain the circuit's response ($DAG(T_i;\ C_n)$) to the test input T_i. We make this design decision to develop the hardware accelerator for AdaTest since acquiring the circuit's response from a configured FPGA (circuit emulation) is significantly faster than the same process running on a host CPU (software simulation). In addition, AdaTest parallelizes the computation of circuit emulation and pipelines at each step of the RL process. AdaTest performs reward computation of the candidate test inputs and adaptive sampling in an online fashion to minimize data communication between the off-chip memory and the FPGA.

Note that we do not include a random number generator (RNG) in our architecture design. Instead, AdaTest stores a set of random numbers pre-computed on CPU using the inherent variation of the operating system. This design choice has two benefits: (i) The hardware overhead of a True RNG is non-trivial and not desired; (ii) Random numbers generated from the CPU typically feature stronger randomness compared to the one generated on FPGA. The results of circuit emulation are used for computing the reward values of test inputs using Equation (1) during reward evaluation. The

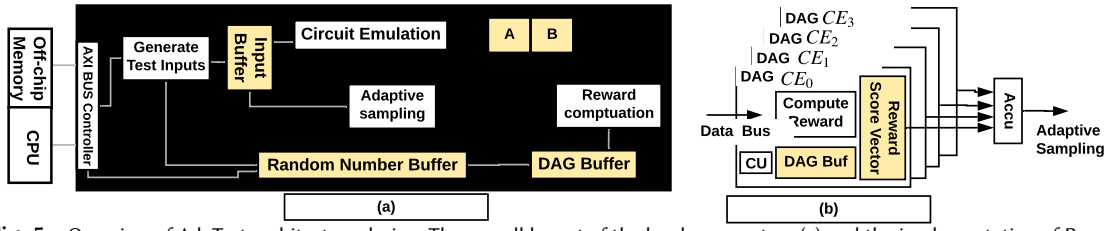

Fig. 5. Overview of AdaTest architecture design. The overall layout of the hardware system (a) and the implementation of Reward Computation Engines (b) are shown.

rare node evaluation and DAG distance computation process in reward evaluation are parallelized by accommodating multiple Computing Engine (CE) in AdaTest's design. We also evenly partition the workload of each CE evenly offline.

After accumulating the reward for each candidate test input, our *adaptive sampling* selects the ones with the highest rewards. This selection process is equivalent to *sorting*. Therefore, AdaTest includes a sorting engine that permutes the key index based on their corresponding rewards. We implement a lightweight sorting engine based on the 'even-odd sort' algorithm [6] for adaptive sampling, incurring a linear runtime overhead with the candidate test set size M.

It is worth noticing that AdaTest does not deploy a central control unit to coordinate the computation flow. Instead, each design component in Figure 5 (a) follows a *trigger-based control* mechanism [27]. Particularly, each module is controlled by the status flag from its previous computation stage. For example, the adaptive sampling module (i.e., the sorting engine) in AdaTest begins to operate when the accumulation of the reward value is detected as completed. Our trigger-based control flow simplifies the control logic while satisfying the data dependency between different components in Figure 4. We detail the design of AdaTest's circuit emulation and auxiliary circuitry as follows.

5.2 AdaTest Circuit Emulation

We empirically observe from AdaTest's software implementation that circuit evaluation (i.e., obtaining $DAG(T_i; C_n)$) dominates the execution time. Motivated to address the high latency issue of evaluating a circuit netlist on CPU, we propose to use *circuit emulation* to improve AdaTest's efficiency. The first step of circuit emulation is to rewrite the netlist of the circuit under test (C_n) such that the values of internal 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_n for any test input by directly applying it to the circuit and collecting the corresponding values in the registers. The collected signal values are used to compute the three reward terms in Equation (1).

Furthermore, AdaTest optimizes the latency of hardware evaluation by storing the emulation results in a ping-pong buffer (consisting of two buffers denoted with A and B) and decoupling it from other hardware components as shown in Figure 5 (a). More specifically, the reward computing engine (CE) calculates the reward of the candidate test input using the data from buffer A. In the meantime, the emulator acquires the states of C_n given the next input T_i and stores the results into buffer B.

5.3 AdaTest Reward Computing Engine

Pipeline with Early Starting. Our architecture design aims to maximize the overlapping time between each execution stage of AdaTest to increase the throughput of TPG. As shown in Figure 6, the ping-pong buffer enables pipelined execution of hardware emulation and reward evaluation. Furthermore, reward evaluation and adaptive sampling can be pipelined across different iterations. We can see from Figure 6 that epoch $(i + 1)$ can start circuit emulation and reward evaluation when the previous epoch begins to generate new test inputs for the next epoch. As such, the latency of candidate test input generation can be hidden by circuit emulation and reward evaluation.

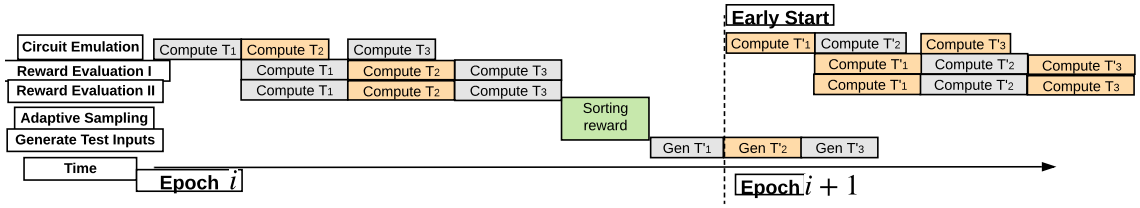

Fig. 6. AdaTest's hardware accelerator employs pipelining optimization to generate test patterns online for HT detection.

Scalable Reward Computing Engine. Once circuit emulation finishes for the current input T_i, AdaTest begins to calculate the reward of this test input using Equation (1). From the hardware perspective, the reward term V_{rare} and V_{scoap} is computed by accumulating the number of activated rare nodes and the corresponding SCOAP values from the circuit C_n, and the reward V_{DAG} is computed by accumulating the Hamming Distance (i.e., XOR) between the values in the current DAG ($DAG(T_i; C_n)$) and the historical ones ($DAG(S_i; C_n)$). Independence between different groups of wire signals typically exists in circuits. AdaTest leverages this property by distributing the computation involving independent groups of nodes to different reward computing engines as shown in Figure 5 (b). As such, each CE stores a subset of DAG nodes' values in the associated DAG buffer. The accumulation of the ultimate reward score completes when the last CE finishes reward computing.

6 EVALUATIONS

We investigate AdaTest's performance for Hardware Trojan detection on various benchmarks, including ISCAS'85 [12], MCNC [21], and ISCAS'89 [3]. The statistics of the evaluated benchmarks are summarized in Table 1. To apply AdaTest on sequential circuits in the ISCAS'89 benchmark, we unroll the circuit for two-time frames and convert it to a combinational one [1, 50]. Note that the unrolling process duplicates the combinational logic blocks, thus increasing the effective circuit size for Trojan detection. The transition probability (P_{trans}) threshold for rare nodes is set to $P_T = 0.1$ for ISCAS'85 and MCNC benchmarks. As for two ISCAS'89 circuits, we use $P_{trans} = 0.0005$ so that the number of rare nodes is at the same level as the previous two benchmarks. The identification results are shown in the last column of Table 1. To compare AdaTest's performance with other logic testing-based Trojan detection methods, we use trigger coverage and Trojan coverage as the metrics to quantify detection effectiveness. To characterize detection efficiency, we use the number of test vectors and the detection runtime as the metrics. We empirically show that AdaTest achieves a higher Trojan detection rate with shorter runtime overhead compared to the counterparts in the rest of this section.

Experimental Setup. Adhering to our threat model defined in Section 3.2, we first design the HT and insert it to each benchmark listed in Table 1. We use a logic-AND gate as the Trojan trigger and select three rare nodes with rare value 1 as the inputs. To fully characterize the performance of AdaTest, we devise various HTs for each circuit (i.e., using different combinations of rare nodes as the trigger) and repeat the insertion for 50 times. Our Trojaned benchmarks include 'hard-to-trigger' HTs with activation probabilities around 10^{-7} (e.g., $c3540$). To compare the performance of AdaTest with prior works, we re-implement MERO [4] and TRIAGE [25] based on the methodology described in the paper using Python. Our experiments are performed on an Intel Xeon E5-2650 v4 processor with 14.5 GiB of RAM.

▪ **MERO Configuration.** We use the parameter selection strategy suggested in MERO [4] for re-implementation. Particularly, we set the size of random patterns to 2,500. The hyper-parameter of MERO is N (desired number of times that each rare node shall be activated). A large value of N achieves a higher detection rate while resulting in a larger test set [4]. We use $N = 1,000$ in our experiments since this value is suggested in MERO [4].

Table 1. Summary of the evaluated circuit benchmarks.

Circuit	dataset	#in	#out	#gate	# of rare nodes ($P_{trans} < P_T$)
c432	ISCAS-85	36	7	160	14
c499	ISCAS-85	41	32	202	48
c880	ISCAS-85	60	26	383	74
c3540	ISCAS-85	50	22	1669	218
c5315	ISCAS-85	178	123	2307	169
c6288	ISCAS-85	32	32	2416	245
c7552	ISCAS-85	207	108	3512	266
des	MCNC	256	245	6473	2316
ex5	MCNC	8	63	1055	432
i9	MCNC	88	63	1035	85
seq	MCNC	41	35	3519	1356
s5378	ISCAS-89	35	49	2958	258
s9234	ISCAS-89	19	22	5825	398

▪ **TRIAGE Configuration.** We use a population size of 100 and select 20 test inputs with the highest fitness score in each generation. The probability of crossover and mutation is set to 0.9 and 0.05, respectively. The termination condition in TRIAGE [25] is used to evolve the test patterns.

▪ **AdaTest Configuration.** In AdaTest's circuit profiling stage, we use the Testability Measurement Tool [31] to compute the SCOAP parameters. The SAT-based smart initialization step of AdaTest's Phase 2 is performed using the pycosat library [32]. Our framework is developed in Python language and does not require extensive hyper-parameter tuning. To ensure the three reward terms in Equation (1) have comparable values within the range of $[0, 10]$, we set the hyper-parameters to $\lambda_1 = 0.05$, $\lambda_2 = 0.0001$, $\lambda_3 = 0.00025$. The candidate test size and the step size in Algorithm 2 are set to $M = 200$ and $L = 80$ for all benchmarks, respectively. We use the percentage threshold $p = 95\%$ to identify rare nodes and set the target activation times to $N = 20$. The maximal iteration time is set to $I_{max} = 500$.

According to the performance metrics in Section 3.3, we use the *trigger coverage* (percentage of trigger nodes identified by the test set) and the *Trojan coverage* (i.e., detection rate) to quantify the effectiveness of HT detection. Meanwhile, we measure the *test set generation time* and test set size of each technique for efficiency comparison. To obtain an accurate and comprehensive performance measurement, we design 50 different HTs for each benchmark in Table 1 while fixing the number of trigger nodes to 3. Each set of devised HTs is inserted into the circuit independently. We run AdaTest detection on each Trojaned circuit for 20 times. The trigger and Trojan coverage for each benchmark are computed as the average value over $50 \times 20 = 1000$ runs.

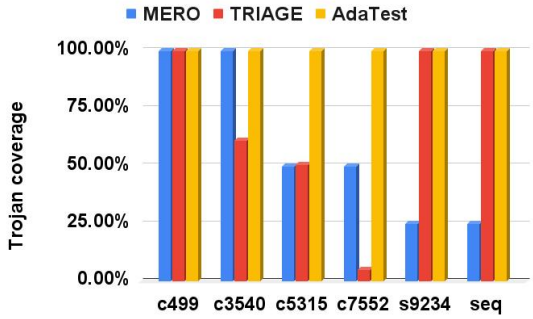

Fig. 7. Trojan detection rates of AdaTest and prior works on various benchmarks.

6.1 Detection Effectiveness

We assess the detection performance of AdaTest, MERO, and TRIAGE using the aforementioned experimental setup. Figure 7 compares the Trojan coverage of the three HT detection techniques on different benchmarks. One can see that our framework achieves uniformly higher detection rates across various circuits. The superior HT detection performance of AdaTest is derived from our definition of *adaptive*, *context-aware* reward functions in Equation (1).

We use two metrics to quantitatively compare the effectiveness of different HT detection techniques: trigger coverage rate and Trojan detection rate. Note that AdaTest determine a Hardware Trojan is present in the circuit if the set of test patterns generated using Alg. 2 result in Trojan activation when the test inputs are applied to the circuit. Therefore, our detection method does not have any false positives and we focus on evaluating the detection rates (which corresponds to the false-negative rate). Table 2 summarizes the HT detection results of three different methods on the benchmarks in Table 1. The trigger coverage and Trojan coverage results are shown in the last two columns of Table 2. It can be seen that AdaTest achieves the highest Trojan coverage while requiring the shortest test generation time across most of the

Table 2. Performance comparison summary of different Trojan detection techniques.

circuit	Method	# test vectors	Runtime (s)	Trigger coverage	Trojan coverage
c499	MERO	1660	136.49	100.00%	100.00%
	TRIAGE	250000	25.91	100.00%	100.00%
	AdaTest	**1010**	13.60	100.00%	100.00%
c880	MERO	1332	352.54	100.00%	100.00%
	TRIAGE	250000	1.75	82.29%	18.00%
	AdaTest	**429**	0.43	100.00%	97.50%
c3540	MERO	1920	1577.36	100.00%	100.00%
	TRIAGE	250000	25.85	100.00%	61.00%
	AdaTest	**905**	22.61	100.00%	**100.00%**
c5315	MERO	9265	1660	100.00%	50.00%
	TRIAGE	250000	37.14	100.00%	50.50%
	AdaTest	**1300**	19.76	100.00%	**100.00%**
c6288	MERO	1906	1867.57	100.00%	100.00%
	TRIAGE	250000	44.11	100.00%	91.50%
	AdaTest	**900**	47.06	100.00%	**99.50%**
c7552	MERO	1916	18650.5	100.00%	50.00%
	TRIAGE	250000	20.93	93.88%	5.00%
	AdaTest	**1600**	39.79	98.08%	**100.00%**
s5378	MERO	1103	30960.11	100.00%	100.00%
	TRIAGE	300	0.45	100.00%	100.00%
	AdaTest	100	11.58	100.00%	**100.00%**
s9234	MERO	11	29737.84	100.00%	25.00%
	TRIAGE	500	35.625	100.00%	100.00%
	AdaTest	140	124.99	100.00%	**100.00%**
des	MERO	1120	34943.41	100.00%	100.00%
	TRIAGE	2500	0.84	100.00%	100.00%
	AdaTest	**156.8**	15.11	92.88%	**100.00%**
ex5	MERO	904	115.22	100.00%	100.00%
	TRIAGE	2500	0.13	99.13%	100.00%
	AdaTest	**500**	12.35	93.81%	**100.00%**
i9	MERO	268	808.56	100.00%	100.00%
	TRIAGE	2500	0.09	100.00%	100.00%
	AdaTest	600	12.15	94.58%	**100.00%**
seq	MERO	1776	3773.3	100.00%	66.67%
	TRIAGE	250000	22.11	95.44%	2.00%
	AdaTest	3700	20.72	94.58%	**82.00%**

benchmarks. More specifically, AdaTest achieves an average of 15.61% and 29.25% Trojan coverage improvement over MERO [4] and TRIAGE [25], respectively. The superior HT detection performance of our logic testing-based approach is derived from the diverse test patterns found by AdaTest adaptive RL-driven input space exploration technique (see Section 4.2). We not only encourage the activation of rare nodes and differentiate their qualities using SCOAP testability parameters but also explicitly characterize the graph-level distance of the CUT status under different test stimuli.

We measure the dynamic rare node coverage versus the number of executed iterations to validate the *time-evolving* property of AdaTest framework. Figure 8 shows the coverage results of AdaTest with random initialization and SAT-based smart initialization on the $c3540$ benchmark. We can make two observations from Figure 8: (i) AdaTest consistently improves the rare node coverage over time (with either initialization method); (ii) SAT-based smart initialization improves the convergence speed of AdaTest, thus reducing our test set generation time. The first observation corroborates the efficacy of our RL-based *progressive* test pattern generation method. The second observation reveals the importance of proper initialization for fast convergence of RL exploration. Note that a shorter convergence time (i.e., a smaller number of iterations in Algorithm 2) indicates s smaller test set returned by AdaTest, which is beneficial to reduce the test generation time for higher detection efficiency.

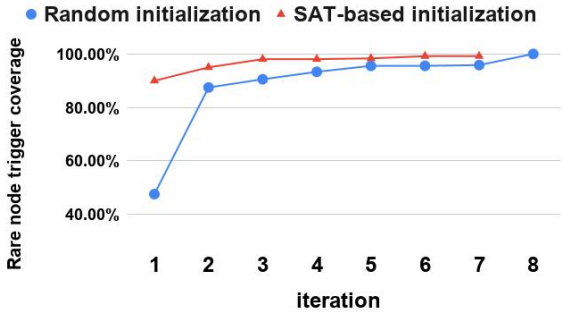

Fig. 8. The rare node coverage of AdaTest versus the number of executed iterations on c3540 benchmark.

6.2 Detection Efficiency

We characterize the efficiency of AdaTest for logic testing based HT detection using two metrics: the test set size (*space efficiency*), and the test set generation time (*runtime efficiency*). The quantitative efficiency measurements of three HT detection methods are shown in the third and fourth columns of Table 2. It can be computed that AdaTest engenders an average of 2.04× and 155.04× reduction of the test set size compared to MERO and TRIAGE across all benchmarks,

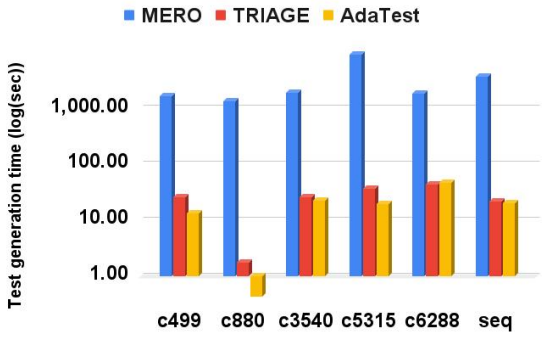

Fig. 9. Test set generation time comparison between AdaTest and prior works. The runtime shown by the y-axis is represented in the log scale.

respectively. The reduction of test set size has two benefits: (i) A smaller test set features a lower memory footprint; (ii) For on-chip test pattern generation, a smaller test set suggests a shorter test generation time.

Figure 9 compares the required test generation time of AdaTest, MERO, and TRIAGE to achieve the coverage results on various benchmarks in Table 2. Note that we use *log-scale* for the vertical axis since the range of runtime is diverse across different circuits. We can observe that AdaTest is the most efficient HT detection method among the three and it also achieves high Trojan coverage (last column of Table 2). More specifically, AdaTest engenders an average of 366.26× and 0.63× test generation speedup compared to MERO [4] and TRIAGE [25], respectively. Note that although the runtime of TRIAGE is smaller, its Trojan detection rate is 30% lower than AdaTest.

6.3 AdaTest Architecture Evaluation

The resource utilization of AdaTest depends on the input length and the circuit size. We report the resource utilization results of the evaluated benchmarks in Table 3. Figure 10 shows that AdaTest architecture achieves approximately linear speedup w.r.t. to the number of CEs. Our hardware design can be scaled up by adding more reward computing engines to parallel the circuit emulation process as AdaTest's computation bottleneck is reward evaluation of the test patterns. Nevertheless, the speedup saturates when N_{CE} is sufficiently high. AdaTest broadcasts the wire values of the circuit response (given a test input) to all CEs via a shared data bus. Each CE scans the DAG buffer and obtains the broadcast wire values to compute the corresponding reward. Therefore, increasing the number of CEs does not lead to extra wire delay. However, more CEs suggest a higher overhead during reward accumulation.

Table 3. Resource utilization of the auxiliary circuitry on *c432*,*c880*, *c2670* and *des* benchmarks with default settings ($N_{CE} = 16$) on Zynq ZC706.

Benchmarks	c432	c880	c2670	des
BRAMS	26	36	65	237
DSP48E1	0	0	0	0
KLUTs (emulator usage)	14.9 (0.5)	25.5 (0.6)	61.1 (3.5)	267.9 (26.1)
FFs (emulator usage)	4,440 (80)	5,743 (160)	6,717 (317)	12,943 (1190)

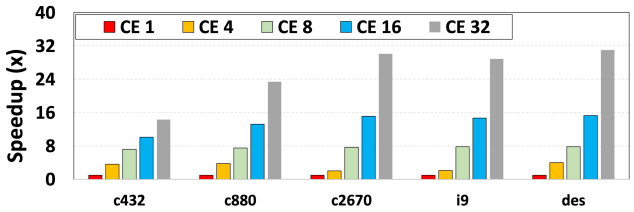

Fig. 10. AdaTest's scalability to the number of DAG reward computing engines. The speedup is near-linear with N_{CE} on large circuits where reward evaluation is the computation bottleneck.

7 CONCLUSION

In this paper, we present a holistic solution to Hardware Trojan detection using adaptive, reinforcement learning-based test pattern generation. To formulate logic testing-based HT detection as an RL problem, we design an innovative reward function to characterize the quality of a test pattern from both static and dynamic aspects. AdaTest progressively expands the test set by identifying test input vectors with high reward values in an iterative approach. AdaTest integrates adaptive sampling to identify and encourage high-reward test patterns, thus accelerating our RL-based input space exploration. We devise AdaTest using a Software/Hardware co-design approach. Particularly, we develop

a domain-specific system-on-chip architecture for efficient hardware implementation of AdaTest. Our architecture optimizes reward evaluation via circuit emulation and pipelines the computation of AdaTest. We perform extensive evaluations of AdaTest on various benchmarks and compare its performance with two counterparts, MERO and TRIAGE. Empirical results corroborate that AdaTest achieves superior effectiveness, efficiency, and scalability for HT detection compared to prior works. AdaTest is a *generic* test pattern generation framework, we plan to investigate its performance on other hardware security problems such as logic verification and built-in self-test in our future work.

ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation (NSF) Trust-Hub under award number CNS-2016737, and NSF TILOS under award number CCF-2112665.

REFERENCES

[1] Rajat Arora and Michael S Hsiao. 2004. Enhancing SAT-based bounded model checking using sequential logic implications. In *17th International Conference on VLSI Design. Proceedings.* IEEE, 784–787.

[2] Swarup Bhunia, Michael S Hsiao, Mainak Banga, and Seetharam Narasimhan. 2014. Hardware Trojan attacks: Threat analysis and countermeasures. *Proc. IEEE* 102, 8 (2014), 1229–1247.

[3] Franc Brglez, David Bryan, and Krzysztof Kozminski. September 22, 2006. *ISCAS89 Benchmark Netlists.* https://filebox.ece.vt.edu/~mhsiao/ISCAS89/

[4] Rajat Subhra Chakraborty, Francis Wolff, Somnath Paul, Christos Papachristou, and Swarup Bhunia. 2009. MERO: A statistical approach for hardware Trojan detection. In *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 396–410.

[5] Shih-Lun Chen, Ho-Yin Lee, Chiung-An Chen, Hong-Yi Huang, and Ching-Hsing Luo. 2009. Wireless body sensor network with adaptive low-power design for biometrics and healthcare applications. *IEEE Systems Journal* 3, 4 (2009), 398–409.

[6] 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.

[7] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 367–379.

[8] Brice Colombier and Lilian Bossuet. 2014. Survey of hardware protection of design data for integrated circuits and intellectual properties. *IET Computers & Digital Techniques* 8, 6 (2014), 274–287.

[9] Mohamed El Massad, Siddharth Garg, and Mahesh V Tripunitara. 2015. Integrated Circuit (IC) Decamouflaging: Reverse Engineering Camouflaged ICs within Minutes.. In *NDSS.* 1–14.

[10] Marc Fyrbiak, Sebastian Strauß, Christian Kison, Sebastian Wallat, Malte Elson, Nikol Rummel, and Christof Paar. 2017. Hardware reverse engineering: Overview and open challenges. In *2017 IEEE 2nd International Verification and Security Workshop (IVSW).* IEEE, 88–94.

[11] Lawrence H Goldstein and Evelyn L Thigpen. 1980. SCOAP: Sandia controllability/observability analysis program. In *Proceedings of the 17th Design Automation Conference.* 190–196.

[12] 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.

[13] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4 (1996), 237–285.

[14] Ramesh Karri, Jeyavijayan Rajendran, and Kurt Rosenfeld. 2012. Trojan taxonomy. In *Introduction to hardware security and trust.* Springer, 325–338.

[15] He Li, Qiang Liu, and Jiliang Zhang. 2016. A survey of hardware Trojan threat and defense. *Integration* 55 (2016), 426–437.

[16] Meng Li, Kaveh Shamsi, Travis Meade, Zheng Zhao, Bei Yu, Yier Jin, and David Z Pan. 2017. Provably secure camouflaging strategy for IC protection. *IEEE transactions on computer-aided design of integrated circuits and systems* 38, 8 (2017), 1399–1412.

[17] Wenchao Li, Zach Wasson, and Sanjit A Seshia. 2012. Reverse engineering circuits using behavioral pattern mining. In *2012 IEEE international symposium on hardware-oriented security and trust.* IEEE, 83–88.

[18] Lang Lin, Markus Kasper, Tim Güneysu, Christof Paar, and Wayne Burleson. 2009. Trojan side-channels: Lightweight hardware trojans through side-channel engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 382–395.

[19] Yu Liu, Ke Huang, and Yiorgos Makris. 2014. Hardware Trojan detection through golden chip-free statistical side-channel fingerprinting. In *Proceedings of the 51st Annual Design Automation Conference.* 1–6.

[20] Yu Liu, Georgios Volanis, Ke Huang, and Yiorgos Makris. 2015. Concurrent hardware Trojan detection in wireless cryptographic ICs. In *2015 IEEE International Test Conference (ITC).* IEEE, 1–8.

[21] Theodore W. Manikas. June 28, 2012. *MCNC Benchmark Netlists.* https://s2.smu.edu/~manikas/Benchmarks/MCNC_Benchmark_Netlists.html

[22] Travis Meade, Shaojie Zhang, and Yier Jin. 2016. Netlist reverse engineering for high-level functionality reconstruction. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC).* IEEE, 655–660.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[24] Samer Moein, Salman Khan, T Aaron Gulliver, Fayez Gebali, and M Watheq El-Kharashi. 2015. An attribute based classification of hardware trojans. In *2015 Tenth International Conference on Computer Engineering & Systems (ICCES)*. IEEE, 351–356.

[25] MA Nourian, Mahdi Fazeli, and David Hély. 2018. Hardware Trojan detection using an advised genetic algorithm based logic testing. *Journal of Electronic Testing* 34, 4 (2018), 461–470.

[26] Zhixin Pan and Prabhat Mishra. 2021. Automated test generation for hardware trojan detection using reinforcement learning. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*. 408–413.

[27] 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.

[28] Fabio Pardo, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. 2018. Time limits in reinforcement learning. In *International Conference on Machine Learning*. PMLR, 4045–4054.

[29] Sayandeep Saha, Rajat Subhra Chakraborty, Srinivasa Shashank Nuthakki, Debdeep Mukhopadhyay, et al. 2015. Improved test pattern generation for hardware trojan detection using genetic algorithm and boolean satisfiability. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 577–596.

[30] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. 2011. A novel technique for improving hardware trojan detection and reducing trojan activation time. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20, 1 (2011), 112–125.

[31] Seyyed Mohammad Saleh Samimi. 2014. Testability Measurement Tool. https://sourceforge.net/projects/testabilitymeasurementtool/

[32] Ilan Schnell. Nov 9, 2017. *pycosat 0.6.3*. https://pypi.org/project/pycosat/

[33] Bicky Shakya, Tony He, Hassan Salmani, Domenic Forte, Swarup Bhunia, and Mark Tehranipoor. 2017. Benchmarking of hardware trojans and maliciously affected circuits. *Journal of Hardware and Systems Security* 1, 1 (2017), 85–102.

[34] Bicky Shakya, Haoting Shen, Mark Tehranipoor, and Domenic Forte. 2019. Covert gates: Protecting integrated circuits with undetectable camouflaging. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), 86–118.

[35] Kaveh Shamsi, David Z Pan, and Yier Jin. 2019. On the impossibility of approximation-resilient circuit locking. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 161–170.

[36] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[37] Benjamin Tan, Ramesh Karri, Nimisha Limaye, Abhrajit Sengupta, Ozgur Sinanoglu, Md Moshiur Rahman, Swarup Bhunia, Danielle Duvalsaint, Amin Rezaei, Yuanqi Shen, et al. 2020. Benchmarking at the frontier of hardware security: Lessons from logic locking. *arXiv preprint arXiv:2006.06806* (2020).

[38] Mohammad Tehranipoor and Farinaz Koushanfar. 2010. A survey of hardware trojan taxonomy and detection. *IEEE design & test of computers* 27, 1 (2010), 10–25.

[39] Mohammad Tehranipoor and Cliff Wang. 2011. *Introduction to hardware security and trust*. Springer Science & Business Media.

[40] 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.

[41] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. 2013. FANCI: identification of stealthy malicious logic using boolean functional analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 697–708.

[42] Xiaoxiao Wang, Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. 2008. Hardware Trojan detection and isolation using current integration and localized current analysis. In *2008 IEEE international symposium on defect and fault tolerance of VLSI systems*. IEEE, 87–95.

[43] Marco A Wiering and Martijn Van Otterlo. 2012. Reinforcement learning. *Adaptation, learning, and optimization* 12, 3 (2012), 729.

[44] Kan Xiao, Domenic Forte, Yier Jin, Ramesh Karri, Swarup Bhunia, and Mohammad Tehranipoor. 2016. Hardware trojans: Lessons learned after one decade of research. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 22, 1 (2016), 1–23.

[45] Yang Xie and Ankur Srivastava. 2018. Anti-SAT: Mitigating SAT attack on logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 2 (2018), 199–207.

[46] Yipei Yang, Jing Ye, Yuan Cao, Jiliang Zhang, Xiaowei Li, Huawei Li, and Yu Hu. 2020. Survey: Hardware trojan detection for netlist. In *2020 IEEE 29th Asian Test Symposium (ATS)*. IEEE, 1–6.

[47] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. 2016. CamoPerturb: Secure IC camouflaging for minterm protection. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[48] Muhammad Yasin, Jeyavijayan JV Rajendran, Ozgur Sinanoglu, and Ramesh Karri. 2015. On improving the security of logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 9 (2015), 1411–1424.

[49] Muhammad Yasin and Ozgur Sinanoglu. 2017. Evolution of logic locking. In *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 1–6.

[50] Zeying Yuan. 2015. *Sequential Equivalence Checking of Circuits with Different State Encodings by Pruning Simulation-based Multi-Node Invariants*. Ph.D. Dissertation. Virginia Tech.

[51] Jie Zhang, Feng Yuan, Linxiao Wei, Yannan Liu, and Qiang Xu. 2015. VeriTrust: Verification for hardware trust. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 7 (2015), 1148–1161.