Automated Test Generation for Hardware Trojan Detection using Reinforcement Learning

Zhixin Pan and Prabhat Mishra
University of Florida, Gainesville, Florida, USA

ABSTRACT

Due to globalized semiconductor supply chain, there is an increasing risk of exposing System-on-Chip (SoC) designs to malicious implants, popularly known as hardware Trojans. Unfortunately, traditional simulation-based validation using millions of test vectors is unsuitable for detecting stealthy Trojans with extremely rare trigger conditions due to exponential input space complexity of modern SoCs. There is a critical need to develop efficient Trojan detection techniques to ensure trustworthy SoCs. While there are promising test generation approaches, they have serious limitations in terms of scalability and detection accuracy. In this paper, we propose a novel logic testing approach for Trojan detection using an effective combination of testability analysis and reinforcement learning. Specifically, this paper makes three important contributions. 1) Unlike existing approaches, we utilize both controllability and observability analysis along with rareness of signals to significantly improve the trigger coverage. 2) Utilization of reinforcement learning considerably reduces the test generation time without sacrificing the test quality. 3) Experimental results demonstrate that our approach can drastically improve both trigger coverage (14.5% on average) and test generation time (6.5 times on average) compared to state-of-the-art techniques.

ACM Reference Format:
Zhixin Pan and Prabhat Mishra. 2021. Automated Test Generation for Hardware Trojan Detection using Reinforcement Learning. In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3394885.3431595

1 INTRODUCTION

A vast majority of semiconductor companies rely on global supply chain to reduce design cost and meet time-to-market deadlines. The benefit of globalization comes with the cost of security concerns. For example, a typical automotive System-on-Chip (SoC) consists of about 100 Intellectual Property (IP) cores, some of these cores may come from potentially untrusted third-party suppliers. An attacker may be able to introduce malicious implants in one of these third-party IPs. Hardware Trojan (HT) is a malicious modification of the target integrated circuit (IC) with two critical parts, trigger and payload. When the trigger is activated, the payload enables the malicious activity. For example in Figure 1, when the output of the trigger logic is true, the output of the payload XOR gate will invert the expected output. The trigger is typically created using a combination of rare events (such as rare signals or rare transitions) to stay hidden during normal execution. The payload represents the malicious impact HT will inflict to the target design, commonly resulting in information leakage or erroneous execution. Due to stealthy nature of these Trojans, it is infeasible to detect them using traditional functional validation methods. It is vital to detect HTs to enable trustworthy computing using modern SoCs.

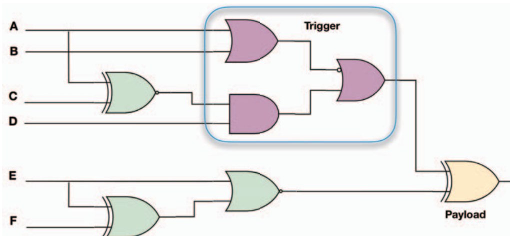


Figure 1: An example hardware Trojan constructed by a trigger logic (purple gates). Once the trigger condition is satisfied, the payload (yellow XOR gate) will invert the expected output. The gates of the original design are shown in green color.

There are many promising research efforts for Trojan detection. These approaches can be broadly classified into two categories: side-channel analysis and simulation-based validation (logic testing). Side-channel analysis focuses on the difference in side-channel signatures (such as power, path delay, etc.) between the expected (golden specification) and actual (Trojan-inserted implementation) values [6, 10, 14]. A major drawback in side-channel analysis is that it is difficult to detect the negligible side-channel difference caused by a tiny Trojan (e.g., few gates in a multi-million gate design) since the difference can easily hide in process variation and environmental noise. In contrast, logic testing is robust against process variation and noise margins [3]. However, it is a fundamental challenge to activate an extremely rare trigger without trying all possible input sequences. Due to exponential input space complexity, traditional logic testing is not suitable for Trojan detection in large designs. Existing logic testing based Trojan detection approaches have two fundamental limitations: high computation complexity (long test generation time) and low Trojan detection accuracy (low trigger coverage).

In this paper, we propose an efficient logic testing approach for HT detection that addresses the above two challenges. (1) Existing logic testing approaches suffer from high computation complexity due to the fact that they require continuously flipping bits [5] of test vectors in an ad-hoc manner to maximize the number of triggered rare activities. In contrast, we utilize a stochastic reinforcement learning framework to enable fast and automated generation of effective tests. (2) Existing approaches provide poor trigger coverage since they only focus on rare signals. Our approach considers both rareness and the testability of signals using a combination of Sandia Controllability/Observability Analysis Program (SCOAP) measurement and dynamic simulation. It is expected to significantly improve the coverage of suspicious nodes with high stability. Specifically, this paper makes the following major contributions:

This work was partially supported by the National Science Foundation grant 1908131.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPDAC '21, January 18–21, 2021, Tokyo, Japan
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-7999-1/21/01...$15.00
https://doi.org/10.1145/3394885.3431595

- Unlike previous works that focuses on rare signals, our approach also exploits the controllability and observability of signals. As a result, the generated test patterns can maximize the trigger coverage in suspicious regions.
- We utilize reinforcement learning to find the profitable test patterns to drastically reduce the test generation complexity.
- Extensive evaluation shows significant improvement in both trigger coverage (14.5% on average) and test generation time (6.45x on average) compared to state-of-the-art approaches.

The paper is organized as follows. Section 2 surveys related efforts in hardware Trojan detection and reinforcement learning. Section 3 describes our test generation framework. Section 4 presents experimental results. Finally, Section 5 concludes the paper.

2 BACKGROUND AND RELATED WORK

2.1 Logic Testing for Hardware Trojan Detection

The basic idea of logic testing for Trojan detection is to generate test patterns that are likely to activate the trigger conditions. In early days, random test generation was widely explored in industry due to its simplicity. However, there is no guarantee for activating stealthy Trojans using millions of random or constrained-random tests. MERO [5] proposed a statistical test generation scheme, which adopts the N-detect idea [15] to achieve better coverage. The heuristic behind is that if all rare signals are activated for at least N times, it is likely to activate the rare trigger conditions when N is sufficiently large. The left side of Figure 2 shows an overview of MERO. It starts with random test generation followed by a brute-force process of flipping bits to increase the number of rare values being satisfied. It provides promising result for small benchmarks, but it introduces long execution time and scalability concerns, making it unsuitable for large benchmarks [9].

To address these issues, Lyu et al. proposed TARMAC [9, 11] as shown on the right side of Figure 2. Like MERO, TARMAC also starts with random simulation to identify rare signals in the netlist. Next, it maps the design to a satisfiability graph, and converts the problem of satisfiability into a clique cover problem, where the authors use an SMT solver [12] to generate test patterns for each maximal clique. Although TARMAC performs significantly better than MERO in evaluated benchmarks, its performance is very unstable. This is due to the fact that TARMAC relies on random clique sampling, making its performance dependent on the quality of sampled cliques. In summary, the existing approaches have inherent limitations in terms of Trojan detection accuracy as well as test generation complexity.

2.2 Reinforcement Learning

Reinforcement learning [16] has earned its reputation as an efficient tool solving problems with large complex searching space. [8]. Unlike traditional supervised learning schemes, training process of reinforcement learning is similar to the nature of human learning. Basically, reinforcement learning works in an *adaptive* way as shown in Figure 3. There are five core components in reinforcement learning: *Agent, Action, Environment, State and Reward.* Reinforcement learning starts with the interaction between agent and environment. At each step, agent utilizes its inner strategy to decide the action to take, and the environment reacts to this action to update the current state, which accordingly provides reward value as feedback. By giving positive reward for beneficial actions and penalty for inferior choices, it allows the machine to distinguish the merits of certain action. Moreover, the agent's strategy gets updated after receiving the

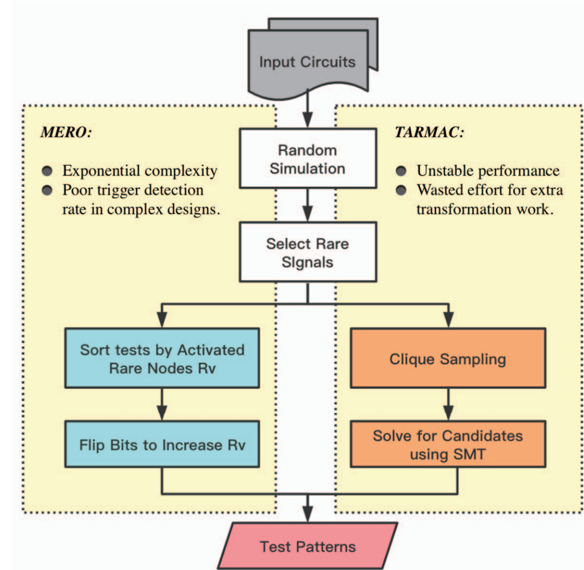


Figure 2: Overview of state-of-the-art logic testing techniques: MERO [5] and TARMAC [9].

feedback, and tries to maximize possible reward next time. Through continuous trials and rewards, the system gradually adapts itself to make the most beneficial decisions, which quickly leads to a desirable solution.

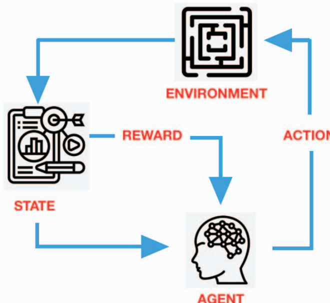


Figure 3: Reinforcement learning consists of five important components: agent, action, environment, state and reward.

There are two key obstacles in directly applying this naive framework in test generation for Trojan detection.

1. *Reward Function:* Explicitly setting up proper reward for actions in test generation is difficult. For example, just counting the number of activated rare nodes is not a good metric to assign reward because an attacker may take multiple dimensions (such as rareness, controllability, observability, etc.) into account while designing a trigger condition.
2. *Action Space:* For a given n-bit test pattern, there are $2^n - 1$ possible ways to produce variations. It is impractical to meet both time and space requirement for dealing with such exponential action space.

In our proposed approach, we address these challenges and provide a fast and efficient learning algorithm, as discussed in Section 3.5.

3 TGRL: TEST GENERATION USING REINFORCEMENT LEARNING

3.1 Motivation

In order to motivate the need for our proposed work on Test Generation using Reinforcement Learning (TGRL), let us take a closer

look at prior works in logic testing based Trojan detection [5, 9, 13]. There are two major problems that affect the performance of existing efforts: rareness heuristic and test generation complexity.

Weakness of Rareness Heuristic: Existing methods rely on rareness heuristic for activating HT triggers. However, in [17], the author rigorously discussed the inconsistency between rare nodes and trigger nodes. According to their experimental evaluation, rare nodes are not necessarily trigger nodes, and vice versa. Reliance on rareness hurts the genuine nodes with rare attribute (e.g., low switching activity). Moreover, a smart implementation of HT can exploit the mixture of both rare nodes and genuine (non-rare) nodes to obfuscate Trojan detection. In our work, we utilize SCOAP testability measurement to address this issue (Section 3.4).

Test Generation Complexity: Another major drawback of existing approaches is high computation complexity. Existing efforts ignores the interaction between intermediate test vectors and circuit that typically provides useful feedback. For example, if a newly generated test vector significantly decreases the number of triggered rare nodes, then the current parameters of the test generation algorithm needs to get adjusted to avoid wasted effort (time). While this intuition is likely to help in guiding the test generation process, it is ignored by both MERO and TARMAC. MERO generates new test patterns by blindly flipping bits in a brute-force manner using random strategy, and TARMAC performs random sampling of cliques without taking the feedback into consideration. In [13], the authors also observed this problem and proposed a genetic algorithm [4] based approach. However, their evaluation shows that they require even longer test generation time. This is due to the combined effects of time-consuming training and slow convergence of genetic algorithm in the later stages of evolution.

Based on the discussion above, we consider an ideal test generation algorithm should satisfy these two crucial requirements to address the presented challenges. Our proposed approach effectively fulfils these requirements as outlined in the next section.

- **Test Effectiveness:** Exploiting not only the rareness, but also the testability of signals to improve trigger coverage.
- **Test Generation Efficiency:** Efficiently making use of feedback in intermediate steps to save test generation time.

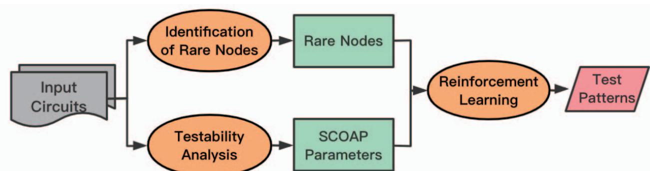


Figure 4: Overview of our proposed test generation framework that consists of three major activities: identification of rare nodes, testability analysis, and reinforcement learning.

3.2 TGRL Overview

Figure 4 shows an overview of our proposed test generation scheme using reinforcement learning (TGRL) that satisfies the two requirements outlined in Section 3.1. For a given circuit design, we first apply a combination of static testability analysis and dynamic simulation, where the simulation provides us with information of rare nodes (Section 3.3) and testability analysis computes SCOAP testability parameters (Section 3.4) of each node in the circuit. Next, these intermediate results are fed into the machine learning model as primary inputs. We utilize reinforcement learning (RL) as the learning model due to its outstanding potential in efficiently solving problems with large and complex solution space [8]. The reinforcement learning model is trained with a stochastic learning scheme (Section 3.5) to generate test vectors, and it continuously improves itself to cover as many suspicious nodes as possible. After sufficient iterations of training, the trained RL model is utilized for automatic test generation. It starts with initial input patterns, and continuously generates a set of test patterns until we get the required number of test patterns.

3.3 Identification of Rare Nodes

Like existing approaches, we utilize dynamic simulation of the benchmark to identify rare nodes. First, the design needs to be simulated using reasonable number of random or constrained-random test patterns. Next, the trace needs to be analyzed to determine how many times each node (signal) is assigned a value '0' or '1' during the simulation. Finally, we need to select the signals (with specific values) as rare nodes that are below a specific threshold. For example, if the output of the NOR gate in Figure 1 was '0' 96% of the time (i.e. '1' with 4% of the time) during simulation and threshold is 5%, the output of the NOR gate with value '1' will be marked as a rare node. A threshold is considered reasonable if the trigger constructed by the respective rare nodes cannot be covered by traditional simulation based validation using millions of random tests. The above process is described in Algorithm1.

Algorithm 1: Identification of Rare Nodes

```
Input  : Design(D), threshold ρ, number of epochs k
Output : Rare nodes set RN
1  repeat
2      randomSim(D)
3      for each s ∈ D do
4          if s.val = 1 then s.cnt1 = s.cnt1 + 1
5          else s.cnt0 = s.cnt0 + 1
6      i = i + 1
7  until i < k;
8  for each s ∈ D do
9      if min{s.cnt1, s.cnt0} ≤ ρk then
10         RN = RN ∪ {s}
```

3.4 Testability Analysis

According to Section 3.1, while majority of existing techniques mainly consider rareness to evaluate suspicious signals, it remains the responsibility of the defender to come up with a more comprehensive measurement. In our approach, we exploit Sandia Controllability/Observability Analysis Program (SCOAP), which takes both *controllability* and *observability* attributes of signals into consideration. In essence, controllability indicates the amount of effort required for setting a signal to a specific value, while observability weighs up the difficulty of propagating the target signal towards observation points.

The testability measurement naturally fits the demand of HT detection from a security perspective. Clearly, signals with low controllability are more likely to be chosen as trigger signals. Because low controllability guarantees the difficulty of switching these signals with a limited number of test patterns. Similarly, targeting signals with low observability as payload are favorable for attackers, since it coincides with HT's clandestine property, avoids them from frequently generating observable impact on design outputs.

The SCOAP method quantifies the controllability and observability of each signal in the circuit with three numerical values.

- **CC0**: Combinational 0-controllability, the number of signals must be manipulated to set '0' value for target.
- **CC1**: Combinational 1-controllability, the number of signals must be manipulated to set '1' value for target
- **CO**: Combinational observability, the number of signals must be manipulated to observe target value at primary outputs.

The SCOAP computation can be performed in a recursive manner. First, the boundary conditions are the primary inputs (PI) and primary outputs (PO), where

$$CC0(PI) = CC1(PI) = 1$$
$$CO(PO) = 0$$

This is straightforward, since only one manipulation is required for controlling primary input (itself), while no extra operation needed for observing primary output. Next, the circuit is converted into a directed acyclic graph (DAG) and further levelized by topological sorting. For each gate, the output controllability is determined by controllability of its inputs, while the input observability is determined by observability of output and all the other input signals. Figure 5 shows the computation formula for three fundamental logic gates. Consider the $CC1$ measurement of AND gate as an example, in order to control the output signal c as '1', both of its input signals a and b should be manipulated as '1' at the same time. Therefore, we have $CC1(c) = CC1(a) + CC1(b) + 1$, where the '+1' is for counting the level depth. It is worth noting that since the controallability parameters of inputs are necessary for computing that of the output signal, the SCOAP procedure starts from calculating controllability values for all signals in a direction from PI toward PO. Afterwards, signals' observability are measured in the reverse direction, which is described in Algorithm 2.

Logic gates	Testability Measure
a, b → c	$CC0(c) = min\{CC0(a), CC0(b)\} + 1$ $CC1(c) = CC1(a) + CC1(b) + 1$ $CO(a) = CO(c) + CC1(b) + 1$
a, b → c	$CC0(c) = CC0(a) + CC0(b) + 1$ $CC1(c) = min\{CC1(a), CC1(b)\} + 1$ $CO(a) = CO(c) + CC0(b) + 1$
a → b	$CC0(a) = CC1(b) + 1$ $CC1(a) = CC0(b) + 1$ $CO(a) = CO(b) + 1$

Figure 5: Formula of SCOAP testability measurement for three fundamental logic gates.

Algorithm 2: Testability Analysis (*getSCOAP*)

Input : Design(D)
Output : SCOAP Parameters of all nodes in D

1. Transfer design into DAG: $G = DAG(D)$
2. Topological Sort: $G^* = topo(G, PI \rightarrow PO)$
3. $CC0(PI) = CC1(PI) = 1, CO(PO) = 0$
4. **for** *each gate* $g \in G^*$ **do**
5. $g.out.SCOAP = computeCC(g.in.SCOAP, type(g))$
6. $G^* = reverse(G^*)$
7. **for** *each gate* $g \in G^*$ **do**
8. $g.in.SCOAP = computeCO(g.out.SCOAP, type(g))$

The task of SCOAP testability analysis can be performed in parallel with identification of rare signals in the circuit. The computed attributes (SCOAP parameters, and rare signal values) will be fed into an reinforcement learning model to fulfill automatic test generation as discussed in the next section.

3.5 Reinforcement Learning

Based on the workflow and challenges for reinforcement learning as discussed in Section 2.2, we present our learning paradigm by listing the mapping from objects in test generation task onto the five crucial components of reinforcement learning.

Agent: Agent usually refers to the object interacting with the environment. In our test generation problem, it is chosen as the current test vector under processing and we denote it as t.

Environment: Circuit design is mapped into environment, which receives the input test vector to produce meaningful results. We denote it as D.

State: State refers to information presented by the environment that can be perceived by the user, such as conditions and parameters. We map the SCOAP parameters and rare signal values of the entire circuit as state. They are encoded by two functions rv and $scoap$, where rv returns the rare value for a specific signal, and $scoap$ is defined as follows.

$$scoap(s) = | < CO(s), CC(rv(s)) > |$$

For a given signal s, $CO(s)$ is the combinational observability of s, and $CC(rv(s))$ is the combinational controlability corresponding to the rare value of s. We are utilizing the $L-1$ norm of SCOAP parameters to measure the synthesized testability of signal s. The state records the basic information of the interaction between current test vector and circuit, which can be further utilized in reward computation.

Action: Action space consists of all possible operations that make changes to the system. For test generation problem, a natural choice is the total set of possible bit flipping operations. However, as mentioned in Section 2.2, in that case the action space (size) for a vector o length n is $2^n - 1$, which is impractical for encoding and manipulation. We apply a stochastic approach to address this challenge as described in Figure 6. In our approach, for each bit in the current test vector, a probabilistic selection will determine whether to flip it or not. In other words, the action is chosen randomly at each step. This non-deterministic action is not completely arbitrary but determined by the given probability distributions, which guarantees the coverage of all possible flipping operations.

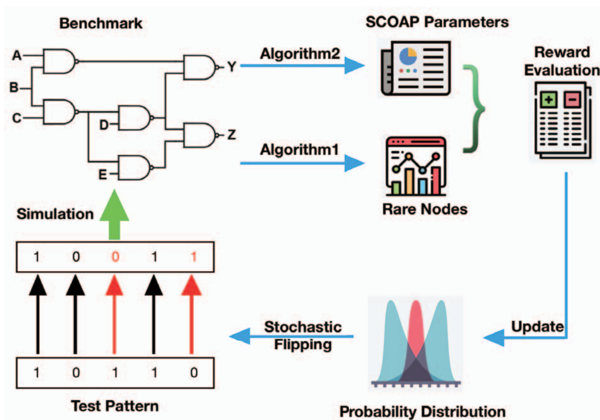


Figure 6: Overview of stochastic reinforcement learning

This approach sheds light on drastically reducing the cost for encoding actions. The probabilistic selection happened on each bit is a binary selection, which can be encoded by one floating-point number. Therefore, an n-bit test pattern requires a vector function $P(\theta) = [\theta_1, \theta_2, ..., \theta_n]$ to formulate the entire space of probability selection.

Reward: In general, reward value is the most important feedback information from the environment that describes the effect of the latest action. For optimization problems, it often refers to the benefit of performing the current operation. In our framework, we exploit a composite reward evaluation scheme consisting of two components, ***rare reward*** and ***testability reward***.

Given current test vector t and action space P, we first denote the newly generated test vector as $t_p = act(t, P)$, and $D(s, t_p)$ as the value of signal s after applying t_p for D, by which we can further define the reward value R in test generation as follows:

$$R_r(t_p) = size(\{s|D(s, t_p) = rv(s)\})$$
$$R_t(t_p) = \sum scoap(s) \quad w.r.t \; D(s, t_p) = rv(s)$$
$$R(t_p) = R_r(t_p) + \lambda \cdot R_t(t_p)$$

Here, $R_r(t_p)$ is the rare reward, which is based on counting of the number of triggered rare signals, and $R_t(t_p)$ is the testability reward defined as the summation of *scoap* measurement of corresponding signals. Finally, we put $\lambda \in \mathbb{R}^+$ as a regularization factor to balance the weight of two components. This reward value is exploited in reinforcement learning model to update hyperparameters at each iteration representing interaction between 'agent' and 'environment'. Specifically, we apply propagation [7] with the computed reward value to adjust those probability distributions, when positive reward is obtained, the probability of the corresponding action is increased, and vice versa. The entire procedure is presented in Algorithm 3.

Algorithm 3: Training of Reinforcement Learning Model

```
Input  : Design(D), Parameter (θ), learning rate (α),
         number of epochs (k)
Output : Optimal Model Parameter θ*
1  Initialize Random test set T = RandomTest()
2  Initialize probability distributions P = P(θ)
3  Compute SCOAP parameters
   (CC0, CC1, CO) = getSCOAP(D)
4  i = j = 0, n = size(T)
5  repeat
6      Initialize Reward: R = 0
7      repeat
8          for each t ∈ T do
9              t_p = act(t, P)
10             R_r(t_p) = size({s|D(s, t_p) = rv(s)})
11             R_t(t_p) = Σ scoap(s)   w.r.t D(s, t_p) = rv(s)
12             R(t_p) = R_r(t_p) + λ · R_t(t_p) R = R + R(t_p)
13         Update parameter : θ = θ + α∇_θ J(R)
14     until j ≥ n;
15 until i ≥ k;
16 Return θ
```

4 EXPERIMENTAL EVALUATION

4.1 Experimental Setup

To enable fair comparison with existing approaches, we deploy the experiment on the same benchmarks as [5, 9] from ISCAS-85 and ISCAS-89 [1]. Also, we preserve the parameter configuration applied in those papers, where rareness threshold is set to 0.1, and total number of sampled Trojans is 1000. The code for benchmark parsing and identification of rare nodes is written in C++17. To perform SCOAP analyses, we use open-source Testability Measurement Tool from [2]. The reinforcement learning model in our approach was conducted on a host machine with Intel i7 3.70GHz CPU, 32 GB RAM and RTX 2080 256-bit GPU. We choose Python (3.6.7) code using PyTorch (1.2.0) with-https://www.overleaf.com/project/5f179de1547e3b0001d694d2 cudatoolkit (10.0) to implement the machine learning framework. The training process consisted of 500 epochs, where we initialize the learning rate α as 0.02 at the beginning, and lower it down to 0.01 after 200 epochs. We compare performance in terms of trigger coverage and test generation time between the following methods:

- **MERO:** Statistical test generation for Trojan detection utilizing multiple excitation of rare occurrences [5].
- **TARMAC:** State-of-the-art test generation method for Trojan detection using clique cover [9].
- **TGRL:** Our proposed test generation technique for Trojan Detection using reinforcement learning.

4.2 Results on Trigger Coverage

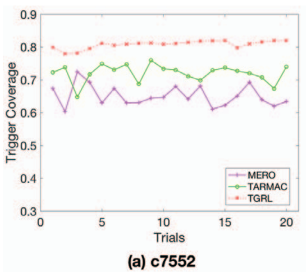
Table 1 demonstrates the effectiveness of our proposed methods compared to the state-of-the-art methods. The first column lists the benchmarks. The second column shows the number of signals in those designs. The third, fifth and seventh columns provide the number of tests generated by MERO [5], TARMAC [9] and our approach, respectively. The fourth, sixth and eighth columns show the trigger coverage using the tests generated by MERO [5], TARMAC [9] and our approach, respectively. The last two columns present the improvement in trigger coverage provided by our approach compared to the state-of-the-art methods. Clearly, MERO provides decent trigger coverage on tiny designs such as c6288, while its trigger coverage drastically drops to less than 10% when applied to large designs like s15850. TARMAC provides promising improvement compared with MERO, but we can observe that it does not have a consistent outcome. For example, in case of c6288 and c7552 with comparable size, the trigger coverage drastically differs (86.1% versus 58.7%). Such huge gap clearly indicates TARMAC's instability with respect to various benchmarks. In contrast, our approach achieves 100% trigger coverage for the first three benchmarks. When we consider large designs, our approach still maintains a high trigger coverage. Overall, our approach outperforms both MERO (up to 92.4%, 77.1% on average) and TARMAC (up to 38.6%, 14.5% on average) in trigger coverage.

Table 1 also reveals the weakness of previous works in terms of "stability" in trigger coverage. To confirm our observation, we further evaluate the stability of all approaches. We choose c7552 and s15850 as target benchmarks, where we repeat each approach for 20 trials and record the trigger coverage, in order to study the extent of variations. The results are shown in Figure 7. As we can see from the figure, our proposed method preserves stable performance across 20 trials. However, there are drastic variations in trigger coverage for the other two approaches. Especially when applied to larger benchmark like s15850, this phenomenon becomes more obvious. The standard deviation of TARMAC is high (0.1876), while it is negligible for our

Table 1: Comparison of trigger coverage with existing approaches

		MERO [5]		TARMAC [9]		Proposed Approach (TGRL)			
Benchmarks	**# Signals**	**# Tests**	**Trigger-Cov(%)**	**# Tests**	**Trigger-Cov(%)**	**# Tests**	**Trigger-Cov(%)**	**improv./ MERO(%)**	**improv./ TARMAC(%)**
c2670	2747	6820	33.1	6820	100	6820	100	66.9	0
c5315	5350	9232	54.3	9232	84.6	9232	100	45.7	15.4
c6288	7744	5044	68.9	5044	86.1	5044	100	31.1	13.9
c7552	7580	14914	4.9	14914	58.7	14914	97.3	92.4	38.6
s13207	8772	44534	2.6	44534	84.2	44534	93.4	90.8	9.2
s15850	10470	39101	2.2	39101	66.3	39101	88.5	86.3	22.2
s35932	12204	34041	8.6	34041	91.5	34041	93.7	85.1	2.2
Average	7838	**21955**	**24.99**	**21955**	**81.62**	**21955**	**96.12**	**77.13**	**14.5**

proposed method (0.0237). In reality, a stable performance is desirable, otherwise a user needs to try numerous times to obtain an acceptable result, which can be infeasible for Trojan detection in large designs.



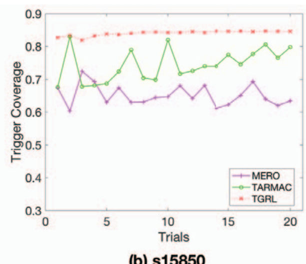


Figure 7: The variation of trigger coverage in 20 trials. TARMAC and MERO demonstrate unstable performance, while our approach provides consistently high trigger coverage.

4.3 Results on Test Generation Time

Table 2 compares the test generation time for the three methods. The first columns lists the benchmarks. The next three columns provide the test generation time for MERO [5], TARMAC [9] and our approach, respectively. The last two columns show the time improvement provided by our approach compared to the other methods.

Table 2: Comparison of Test Generation Time (in seconds).

Design	MERO	TARMAC	TGRL	MERO/TGRL	TARMAC/TGRL
c2670	1149	301	74	15.52x	4.06x
c5315	3791	643	126	30.08x	5.11x
c6288	826	666	108	7.64x	6.16x
c7552	7423	2809	169	43.92s	16.62x
s13207	16508	6022	1328	12.4x	4.53x
s15850	16429	12580	1204	13.64x	10.44x
s35932	53171	23446	4092	12.99x	5.72x
Average	**14185**	**6638**	**1014**	**14.1x**	**6.54x**

Clearly, our approach provides the best results across benchmarks, while MERO is the worst. Not surprisingly, MERO lags far behind the other two in time efficiency due to its brute-force bit-flipping method. While TARMAC provides better test generation time than MERO, our approach is significantly (6.54x on average) faster than TARMAC. There are three major bottlenecks that slow down TARMAC. First, TARMAC requires extra transformation to map the circuit design into a satisfiability graph. Next, the clique sampling in TARMAC is compute-intensive, it repeatedly removes nodes from circuit and re-computes logic expression for each potential trigger signal. Finally, TARMAC exploits an SMT solver to generate each candidate test vector, which determines the upper-bound of its time efficiency. In contrast, our proposed approach does not use any satisfiability solver. Only overhead in our approach is the training time - the model training is composed of 500 iterations where each iteration is basically a one-step test mutation and evaluation. When the model is well-trained, it can automatically generate all the remaining test vectors without extra efforts. *Note that our reported test generation time includes the model training time.* Overall, our proposed approach drastically (up to 16.6x, 6.54x on average) improves the test generation time compared to state-of-the-art methods.

5 CONCLUSION

Detection of hardware Trojans is an emerging and urgent need to address semiconductor supply chain vulnerabilities. While there are promising test generation techniques, they are not useful in practice due to their inherent fundamental limitations. Specifically, they cannot provide reasonable trigger coverage. Most importantly, they require long test generation time and still provides unstable performance. To address these serious challenges, we proposed an automated test generation scheme using reinforcement learning for effective hardware Trojan detection. The proposed method made several important contributions. It explored an efficient combination of rareness of signals and testability attributes to provide a fresh perspective on improving the coverage of suspicious signals. We also developed an automated test generation scheme utilizing reinforcement learning model trained with stochastic methods which is able to drastically reduce the test generation time. Experimental results demonstrated that our approach can drastically reduce the test generation time (6.54x on average) while it is able to detect a vast majority of the Trojans in all benchmarks (96% on average), which is a significant improvement (14.5% on average) compared to state-of-the-art methods.

REFERENCES

[1] [n.d.]. ISCAS Benchmarks. https://filebox.ece.vt.edu/~mhsiao/iscas89.html.
[2] [n.d.]. SCOAP. https://sourceforge.net/projects/testabilitymeasurementtool/.
[3] Alif Ahmed et al. 2018. Scalable hardware Trojan activation by interleaving concrete simulation and symbolic execution. In *International Test Conference.* 1–10.
[4] Thomas Back. 1996. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford university press.
[5] R. Chakraborty et al. 2009. MERO: A Statistical Approach for Hardware Trojan Detection. In *CHES.* 396–410.
[6] Yuanwen Huang et al. 2018. Scalable test generation for Trojan detection using side channel analysis. *IEEE TIFS* 13, 11 (2018), 2746–2760.
[7] Henry J Kelley. 1960. Gradient theory of optimal flight paths. *Ars Journal* 30, 10 (1960), 947–954.
[8] Sami Khairy et al. 2019. Reinforcement-Learning-Based Variational Quantum Circuits Optimization for Combinatorial Problems. *CoRR* abs/1911.04574 (2019).
[9] Yangdi Lyu and Prabhat Mishra. 2020. Automated Trigger Activation by Repeated Maximal Clique Sampling. In *ASPDAC.* 482–487.
[10] Yangdi Lyu and Prabhat Mishra. 2020. MaxSense: Side-Channel Sensitivity Maximization for Trojan Detection using Statistical Test Patterns. *TODAES* (2020).
[11] Yangdi Lyu and Prabhat Mishra. 2020. Scalable Activation of Rare Triggers in Hardware Trojans by Repeated Maximal Clique Sampling. *IEEE TCAD* (2020).
[12] L. Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS.* 337–340.
[13] M. Nourian et al. 2018. Hardware Trojan Detection Using an Advised Genetic Algorithm Based Logic Testing. *JETTA* 34, 4 (2018), 461–470.
[14] Zhixin Pan, Jennifer Sheldon, and Prabhat Mishra. 2020. Test Generation using Reinforcement Learning for Delay-based Side-Channel Analysis. ICCAD.
[15] Irith Pomeranz and Sudhakar M. Reddy. 2004. A Measure of Quality for n-Detection Test Sets. *IEEE Trans. Computers* 53, 11 (2004), 1497–1503.
[16] Stuart J Russell and Peter Norvig. 2016. *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,.
[17] H. Salmani. 2017. COTD: Reference-Free Hardware Trojan Detection and Recovery Based on Controllability and Observability in Gate-Level Netlist. *TIFS* (2017).