DETERRENT: Detecting Trojans Using Reinforcement Learning

Vasudev Gohil, *Graduate Student Member, IEEE,* Satwik Patnaik, Hao Guo, *Graduate Student Member, IEEE,* Dileep Kalathil, *Senior Member, IEEE* and Jeyavijayan (JV) Rajendran, *Senior Member, IEEE*

***Abstract*—The globalized nature of the integrated circuits supply chain has given rise to several security problems. The insertion of malicious components, called hardware Trojans, is one such serious problem. Since Trojans are activated only under extremely rare trigger conditions and the search space is exponentially large, detecting them is arduous. Researchers have attempted to detect Trojans by querying the design-under-test using appropriate test patterns and monitoring its logical or side-channel response. However, techniques in both these categories lack either in terms of detection accuracy or scalability for larger designs. In this work, we investigate why existing techniques fall short and use our findings to propose a new reinforcement learning (RL) framework for detecting Trojans. We carefully design two RL agents (one for each category) that navigate the exponential search space of the test patterns and return minimal sets of patterns that are most likely to detect Trojans. We overcome challenges related to scalability and efficacy through appropriate solutions. Experimental results on a variety of benchmarks demonstrate the scalability and efficacy of our RL agents, which reduce the number of test patterns significantly ($169.68\times$ and $34.73\times$ on average overall and $27.59\times$ and $3.72\times$ on average over large benchmarks) while maintaining or improving the Trojan-detection success rate compared to the state-of-the-art techniques.**

***Index Terms*—Reinforcement Learning, Hardware Trojans, Logic Testing, Side-channel Analysis, Test Generation**

I. Introduction

REINFORCEMENT learning (RL) is a technique where an agent learns through exploring and exploiting an unknown environment to take optimal actions. This approach is highly attractive from a cybersecurity perspective, as the agent can devise optimal defense strategies in an unknown adversarial environment. RL algorithms have recently seen significant improvements, allowing agents to efficiently navigate high-dimensional search spaces to find optimal actions. Researchers have utilized RL agents to develop promising solutions for various security problems, including intrusion detection [1], fuzzing [2], and developing secure cyber-physical systems [3], [4]. Despite these developments, using RL to develop optimal defenses in adversarial environments in hardware security is still in its early stages [5]–[8]. In this work, we demonstrate how RL can be used to detect hardware Trojans (simply called Trojans hereafter) efficiently. We choose the Trojan detection problem because it has significant computational challenges vis-a-vis detection in an unknown environment (i.e., Trojan-infested design).

Manuscript received March 16, 2023; revised June 11, 2023; accepted August 11, 2023. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Swaroop Ghosh. (*Corresponding authors: Vasudev Gohil and Satwik Patnaik*).

This work was partially supported by the National Science Foundation (NSF CNS–1822848 and NSF DGE–2039610). A part of this research was conducted using the advanced computing resources provided by Texas A&M High-Performance Research Computing.

Vasudev Gohil, Hao Guo, Dileep Kalathil, and Jeyavijayan (JV) Rajendran are with the Department of Electrical and Computer Engineering, Texas A&M University, College Station, USA (email: gohil.vasudev@tamu.edu; guohao2019@tamu.edu, dileep.kalathil@tamu.edu, jv.rajendran@tamu.edu).

Satwik Patnaik was with the Department of Electrical and Computer Engineering, Texas A&M University, College Station, USA. He is now with the the Department of Electrical and Computer Engineering, University of Delaware, Newark, USA (email: satwik@udel.edu).

Digital Object Identifier 10.1109/TCAD.2023.3309731

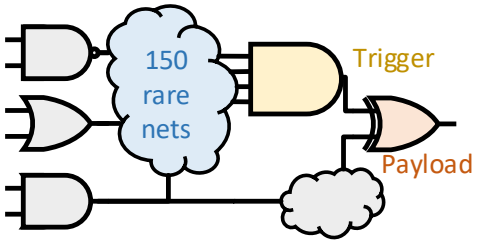

Fig. 1. Example of a Trojan in a design with 150 rare nets.

As integrated circuit (IC) manufacturing costs have risen, semiconductor companies have been compelled to send their designs to untrusted, offshore foundries. Malicious components known as Trojans inserted during the fabrication stage can leak confidential information [9], degrade performance [10], or cause a denial of service [11].

A. Hardware Trojans

A hardware Trojan is composed of two elements: the *trigger* and the *payload*. Once the trigger is activated, the payload initiates a malicious effect in the design. The example in Fig. 1 illustrates a Trojan that flips an output when the trigger is activated. The trigger comprises various nets, known as *select nets*, in the design. For instance, an attacker may pick the select nets to activate the trigger only under rare circumstances. This is accomplished by setting a *rareness threshold*[1] and building the trigger using the corresponding *rare nets*.

Trojan detection is a challenging task as Trojans are deliberately designed to evade detection [12]. For instance, in Fig. 1, the trigger is constructed using four out of 150 rare nets. Checking all possible combinations of rare nets is an exhaustive task, with ${}^{150}C_4 \approx 20\times10^6$ different combinations. Such a large space makes it difficult even for conventional

[1]Rareness threshold is the probability below which nets are classified as rare nets.

automatic test pattern generation (ATPG) tools [13] to activate the trigger.

B. Hardware Trojan Detection Techniques

Trojan detection techniques can be broadly classified into two categories: logic testing and side-channel analysis. Logic testing involves applying test patterns to the Trojan-infested design to activate the trigger [14]–[16]. However, activating an extremely rare trigger is challenging because the possible combinations of rare nets are extensive. In contrast, side-channel-based techniques detect Trojans based on the differences in the side-channel measurements, such as power or timing, between the Trojan-free (i.e., golden) design and a Trojan-infested design [17]–[19]. However, since Trojans have a very small footprint compared to the overall size of the design, their impact on side-channel metrics is usually negligible and concealed under process variation and environmental effects [20]. For a more comprehensive survey on Trojans and their detection techniques, interested readers can refer to [12].

Although detecting Trojans is critical, it is difficult to do so efficiently. Consider Fig. 1; the defender may need up to 20×10^6 test patterns to guarantee trigger activation (which is essential for detecting Trojans) because the defender does not know which rare nets make the trigger. Next, we outline the ideal characteristics required from any technique for detecting Trojans. **(1) High success rate**: The technique should detect a large number of Trojans successfully.[2] **(2) Small test generation time**: The time required to generate the test patterns should not be large; otherwise, the technique will not scale to larger designs. **(3) Compact set of test patterns**: The number of test patterns required to detect the Trojan should be small. A large number of test patterns affect the testing cost adversely. **(4) Feedback-guided approach**: The technique should analyze the test patterns and their impact on the circuit to generate new test patterns, thereby reducing the test generation time and the size of the test set.

C. Our Contributions

Our work proposes novel techniques that aim to meet all four desirable characteristics. To detect Trojans, we approach the problem of test pattern generation as a reinforcement learning (RL) problem. RL algorithms are well-suited to navigate through vast search spaces in search of optimal solutions, making them an ideal choice for test pattern generation, which requires exploration of the vast space of test patterns. However, several challenges must be overcome to implement a practical and scalable RL agent. These challenges include (i) extensive training times for large designs, (ii) efficient action selection, (iii) fine-tuning on challenging benchmarks, and (iv) the need to prune search spaces to scale the agent. Sections III and IV describe how we address these challenges. Our work makes the following primary contributions.

- We develop an efficient RL-based logic testing technique that can activate rare trigger conditions, overcoming the limitations of the current logic-testing-based Trojan detection techniques.
- To make our approach scalable to large designs such as the `MIPS` processor, `AES` core, `GPS` module, and the `mor1kx` processor, we overcome several challenges.
- We conduct an extensive evaluation on diverse benchmarks to demonstrate the superior capabilities of our technique, which outperforms existing logic-testing-based techniques on all benchmarks.
- Our RL-based logic testing technique reduced the test set size by up to two orders of magnitude ($169.68\times$ on average overall and $27.59\times$ on average over large benchmarks) compared to existing techniques.
- We develop another RL agent for side-channel-based Trojan detection by addressing the limitations of the existing side-channel-based techniques.
- Experimental evaluation demonstrates that our RL-based side-channel technique outperforms all prior work, including the state-of-the-art technique.
- Our RL-based side-channel technique provides $34.73\times$ (on average overall) and $3.72\times$ (on average over large benchmarks) reduction in the number of test patterns while improving detection performance.
- We are committed to advancing the field of research, and as such, we will be releasing our benchmarks and test patterns to the community to enable future developments [21].

The remainder of the paper is organized as follows. We present relevant background on Trojans and reinforcement learning in Section II. We also detail our assumptions in Section II. In Section III, we provide a mathematical formulation of the trigger activation problem and the details of our RL agent for logic-testing-based Trojan detection. In Section IV, we explain why the previous agent cannot be used for side-channel-based Trojan detection and why a new agent is needed. Then, we explain our RL agent for side-channel-based Trojan detection. We detail our experimental findings in Section V. Next, we discuss a closely-related work and practicality aspects of our techniques in Section VI. Finally, we present concluding remarks in Section VII.

II. Background and Assumptions

We first provide an overview of logic-testing-based and side-channel-based Trojan detection techniques, followed by an introduction to RL. Finally, we detail the threat model assumed in this work.

A. Logic-testing-based Trojan Detection

Researchers have developed a plethora of logic-testing-based Trojan detection techniques. The objective of all those techniques is to generate test patterns that are likely to activate Trojan triggers.

MERO generates test patterns that activate each rare net N times [14]. The hypothesis is that if all the rare nets are activated N times, the test patterns are likely to activate the trigger. The algorithm starts with a large pool of random test

[2]For logic-testing-based techniques, success is measured as the percentage of Trojan triggers activated, usually called the trigger activation rate or the trigger coverage. For side-channel-based techniques, success is measured as the relative deviation of the side-channel footprint compared to the Trojan-free (i.e., golden) design, and it is called the side-channel sensitivity.

patterns and iteratively performs circuit simulation to keep track of the number of activated rare nets. While MERO provides moderate performance for small benchmarks, it fails for large benchmarks. For instance, MERO's trigger coverage for the `MIPS` processor is $\approx 0\%$ [15], as it violates the ideal characteristics (1), (2), (3), and (4) mentioned in Section I-B.

TARMAC overcomes the limitations of MERO by transforming the problem of test pattern generation into a clique cover problem [15]. It iteratively finds maximal cliques of rare nets that satisfy their rare values. By not relying on brute force, TARMAC outperforms MERO by a factor of $71\times$ on average. However, the performance of TARMAC is sensitive to randomness since the algorithm relies on randomly sampled cliques. Although the test generation time for TARMAC is short, it violates characteristics (3) and (4).

TGRL uses RL along with a combination of rareness and testability measures to overcome the limitations of TARMAC [16]. TGRL achieves better coverage than TARMAC and MERO while reducing the run time. However, it still violates characteristic (3), as evidenced by our results in Section V.

The success rate of logic-testing-based techniques, called the **trigger activation rate** or **trigger coverage**, is defined as the percentage of Trojan triggers activated by the given test patterns.

B. Side-Channel-Based Trojan Detection

Side-channel-based Trojan detection techniques compare the transient current, power consumption, or path delay in the Trojan-free (i.e., golden) design and the design under test. If the difference between the measured side-channel data from the two designs is larger than a threshold, a Trojan is suspected to be present in the design under test.

MERS generates a sequence of test patterns such that collectively, the consecutive pairs of patterns in the sequence cause rare nets to switch from their non-rare to rare values several times and consequently increase the deviation of the measured side-channel signal (dynamic current, in this case) from the expected (i.e., golden) values [18].

MERS-h extends MERS by performing a Hamming distance-based reordering of the test patterns generated by MERS to maximize the activity in the rare nets and minimize the activity in the non-rare nets so that the activity due to the Trojan is not overshadowed by environmental noise [18].

MERS-s is another extension of MERS [18]. Unlike MERS-h, it reorders test patterns generated by MERS using the simulation-based dynamic activity of the circuit instead of Hamming distance-based.

MaxSense, the current state-of-the-art side-channel-based Trojan detection technique uses a combination of a satisfiability modulo theory (SMT) solver and a genetic algorithm [19]. The SMT solver is used to generate test patterns (called first patterns) that are likely to activate Trojans. The genetic algorithm then mutates these first patterns to generate so called second patterns corresponding to the first test patterns with the objective of maximizing switching in the rare nets and minimizing switching in the non-rare nets.

Although the above techniques are designed to detect Trojans, they violate one or more of the four ideal characteristics required from a Trojan detection technique. MERS and its extensions violate all four characteristics: they have limited success, have large runtime, do not generate a compact set of test patterns, and do not leverage feedback. MaxSense, though successful in achieving a high success rate with a small runtime, is unable to generate a compact set of test patterns and is not feedback-guided. On the other hand, our approach, DETERRENT-SC satisfies all four characteristics and outperforms all prior works, as evidenced by our formulation and experiments.

Side-channel-based techniques are evaluated using the **side-channel-sensitivity** metric, which is defined as the switching in a Trojan-infested design relative to the switching in the Trojan-free design.

C. Reinforcement Learning

RL is a powerful machine learning paradigm that allows agents to learn how to take optimal actions in an environment to achieve a specific goal. This methodology is formalized as a Markov decision process. RL is based on the concept of trial-and-error learning, where an agent interacts with its environment and receives feedback in the form of rewards or punishments based on its actions. Usually, in an RL system, the agent interacts with the environment during discrete time steps. During each step, the agent receives information about the current state and the reward, then selects an action to send to the environment. The environment then moves the agent to a new state and provides a reward that corresponds to the action and state transition. The goal of the RL agent is to learn a policy π that maximizes the expected cumulative reward. This policy maps state-action pairs to probabilities of taking a particular action in a given state. The agent learns the optimal or nearly optimal policy by interacting with the environment through trial and error.

D. Threat Model

We assume the standard threat model used in logic-testing-based and side-channel-based Trojan detection techniques [14]–[16], [18], [19]. We assume that the adversary inserts Trojans in rare nets of the design to remain stealthy. The defender's (i.e., our) objective for a logic-testing-based technique is to generate a minimal set of test patterns that activate unknown trigger conditions. Whereas a defender using a side-channel-based technique aims to generate test patterns that maximize the impact of the Trojan on side-channel leakage. In both cases, we generate test patterns using only the Trojan-free (i.e., golden) netlist.

III. DETERRENT-LT: Detecting Trojans using Reinforcement Learning and Logic Testing

In this section, we formulate the problem of generating test patterns for trigger activation as a set cover problem and detail how it can be solved using integer linear programming (ILP). Next, we discuss the practical limitations of this approach and motivate the use of reinforcement learning to solve the Trojan trigger activation problem effectively and efficiently. Then, we

formulate the trigger activation problem as an RL problem, however the formulation still suffers from challenges related to scalability, efficiency, and poor performance. We then address these challenges and devise a final RL agent that outperforms all existing logic-testing-based techniques.

A. Mapping Trigger Activation to Set Cover Problem

Here, we map the trigger activation problem to the set cover problem. The formulation of the set cover problem is as follows. Two sets are given: a set $\mathcal{U}$ of elements (called the universe) and a set $\mathcal{S}$ of subsets of the set $\mathcal{U}$. The union of all the subsets in $\mathcal{S}$ covers the set $\mathcal{U}$. Then, the set cover problem concerns with finding the smallest number of subsets whose union covers the universal set $\mathcal{U}$.

Now, in our case, suppose the circuit under consideration has I primary inputs and R rare nets. We define $\mathcal{U}$ (the universe of elements) as the set of all $2^R - 1$ combinations (all except the empty set) of those R rare nets. Since there are I binary inputs to the circuit, there are 2^I possible input patterns. We define $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{2^I}\}$ as the sets of rare nets that are activated by those 2^I input patterns. Then, finding the smallest set of test patterns that activate all possible valid Trojan triggers is equivalent to finding the set cover with $\mathcal{U}$ as the universe of elements and $\mathcal{S}$ as the set of subsets of $\mathcal{U}$.

Mathematically, the set cover problem can be formulated as an ILP problem as follows:

$$\begin{aligned} \min \quad & \sum_{s=1}^{|\mathcal{S}|} y_s \\ \text{s.t.} \quad & \sum_{s:c\in\mathcal{S}_s} y_s \geq 1, \quad \forall c \in \mathcal{U} \\ & y_s \in \{0,1\}, \quad \forall s \in \{1,2,\ldots,|\mathcal{S}|\} \end{aligned} \tag{1}$$

Here, y_i ($i \in \{1, 2, \ldots, |\mathcal{S}|\}$) is a decision variable:

$$y_i = \begin{cases} 1, & \text{if subset } \mathcal{S}_i \text{ is selected} \\ 0, & \text{otherwise} \end{cases}$$

Next, we prove that the set of test patterns corresponding to the selected subsets from the solution of this ILP is compact and complete.

Let t_i be the input pattern corresponding to the subset $\mathcal{S}_i$. Let $Y^* = \{i : y_i = 1\}$ be the indices of all selected subsets by the solution of the ILP.

Theorem 1. *The test set $T^* = \{t_i : i \in Y^*\}$ is complete and compact, i.e., it activates all valid trigger signals, and it is the shortest test set to do so.*

Proof. First, we prove the completeness of the test set. By definition, a trigger signal is valid if and only if there exists some input pattern (t_i) that can activate the trigger. Now, since the solution of the ILP, Y^*, covers all elements of $\mathcal{U}$ (i.e., valid combinations of rare nets), there must be an index k in Y^* such that $\mathcal{S}_k$ contains the rare nets that make the trigger signal. Thus, the corresponding test pattern t_k must activate the trigger signal.

Now, we prove that the test set is compact. This is easy to prove simply by the objective of the ILP. Since the objective is to minimize the number of selected subsets, the optimal solution will contain the smallest set of subsets $\mathcal{S}_i$, and hence the test patterns t_i that activate all valid trigger signals. ■

Hence, one can obtain a complete and compact test set for trigger activation by solving the above-mentioned ILP. However, there are two practical limitations of the above approach, which are explained next.

B. Practical Limitations

Although solving the ILP is guaranteed to provide a compact set of test patterns that covers all possible Trojan triggers, it is impractical because of the following reasons:

(i) Formulating the ILP problem requires us to compute the subsets ($\mathcal{S}_1, \ldots, \mathcal{S}_{2^I}$), i.e., we need to perform 2^I circuit simulations to find the sets of rare nets that are activated for each input pattern. Since the complexity of this process is exponential in the number of inputs of the design, it is not feasible to do so for realistic designs that have several hundreds of inputs.

(ii) Finding a solution to the ILP (i.e., solving the set cover problem) is an NP-complete problem [22]. Hence, solving the ILP is not scalable to realistic designs.

So, we need an algorithm that can quickly explore the exponentially large space of input patterns efficiently and return a small set of test patterns that is likely to activate all possible trigger signals. To this end, we use an RL agent.

C. A Simple Reinforcement Learning Formulation

As shown in Fig. 1, to activate the trigger, the defender has to apply an input pattern that forces all four rare nets to take their rare values simultaneously,[3] but the defender does not know which four rare nets constitute the trigger. However, as discussed above, one input pattern can activate multiple different combinations of rare nets simultaneously. So, we need to devise an RL agent that can find a minimal set of input patterns that can collectively activate all combinations of rare nets. We define a property, *compatibility*, of a set of rare nets as follows: A set of rare nets, say R, is compatible if there exists an input pattern that can activate all the rare nets in the set simultaneously. This compatibility of a given set of rare nets can be checked using a Boolean Satisfiability (SAT) solver.[4] Thus, our objective is to develop an RL agent that generates maximal sets of compatible rare nets.

To this end, we formulate the Markov decision process that, when solved by our RL agent, will give maximal sets of compatible rare nets.

- **States** $\mathcal{S}$ is the set of all subsets of the rare nets. An individual state s_t represents the set of compatible rare nets at time t. In software implementation, we encode s_t as an R-bit binary vector, with each element of the vector indicating whether or not the corresponding rare net is present.

[3]For the sake of conciseness, henceforth, we shall use the phrase "activate the rare nets" instead of "force the rare nets to take their rare values."

[4]Given a design netlist, we first convert it into a Boolean formula consisting of clauses for all gates in the netlist. Then, we add clauses that activate the rare nets. If the final Boolean formula is satisfiable, the set of rare nets is compatible; otherwise, it is not compatible.

- **Actions** $\mathcal{A}$ is the set of all rare nets. An individual action a_t is the rare net chosen by the agent at time t. In software implementation, a_t is a scalar integer that can take a value from 0 to $R-1$, representing the rare net chosen at time t.
- **The state transition** $P(s_{t+1}|a_t, s_t)$ refers to the probability that action a_t in state s_t will lead to state s_{t+1}. In our case, if including the chosen rare net (i.e., the action) in the current set of rare nets (i.e., the current state) results in a compatible set of rare nets, then we add the selected rare net to the current state, which results in the next state. Otherwise, the next state remains the same as the current state. As a result, the state transition in our model is deterministic, as illustrated below.

$$s_{t+1} = \begin{cases} \{a_t\} \cup s_t, & \text{if } \{a_t\} \cup s_t \text{ is compatible} \\ s_t, & \text{otherwise} \end{cases}$$

- **Reward function** $\mathcal{R}(s_t, a_t)$ is equal to the square of the size of the next state for compatible states, and 0 otherwise.

$$\mathcal{R}(s_t, a_t) = \begin{cases} |\{a_t\} \cup s_t|^2, & \text{if } \{a_t\} \cup s_t \text{ is compatible} \\ 0, & \text{otherwise} \end{cases}$$

  In our model, the reward is structured such that the agent's objective is to maximize the size of the state, which corresponds to the number of compatible rare nets. To achieve this, we square the reward at each step. However, any power greater than 1 would be suitable, as we aim for the reward function to be convex, reflecting the increasing difficulty of increasing the size of compatible rare nets as the state size grows.
- **Discount factor** γ $(0 \leq \gamma \leq 1)$ indicates the importance of future rewards relative to the current reward.

To begin, the agent starts in an initial state s_0, consisting of a single randomly selected rare net. In each time step t, the agent selects an action a_t based on the current state s_t, resulting in a new state s_{t+1} according to predetermined state transition rules, and receives a corresponding reward r_t. This process is repeated T times, forming an *episode*. At the conclusion of each episode, the agent's state reflects the rare nets that are compatible. As both the state and action spaces are discrete, we apply the Proximal Policy Optimization (PPO) algorithm with default parameters (e.g., discount factor of 0.99[5]), unless otherwise specified, to train the agent [23].

After training the agent to return the maximal sets of compatible rare nets, we use a Boolean satisfiability (SAT) solver to generate the corresponding test patterns for the k largest distinct sets, where k is a hyperparameter of our technique.

Our results indicate that while this simple agent performs well on small benchmarks, it struggles to achieve high trigger coverage on larger benchmarks such as the OpenCores `MIPS` processor [24], where we only achieve approximately 70% coverage after training for 12 hours. We conducted a detailed analysis of the architecture and identified several challenges that contribute to this performance limitation. These challenges and our solutions are presented next.

[5]We experimented with discount factors 0.80, 0.85, 0.90, and 0.95 in addition to 0.99 and observed no impact on the performance. Hence, we use the default value of 0.99.

TABLE I
COMPARISON OF TRAINING RATES FOR THE REWARD METHODS FOR THE `MIPS` BENCHMARK: ALL STEPS VS. END-OF-EPISODE

Method	Max. # compatible rare nets	Rate	
		(steps/min)	(eps./min)
Reward at all steps	53	108	0.72
End-of-episode reward	50	9387	63
Improvement	-5.6%	86.91×	87.5×

D. End-of-Episode Reward Computation

Challenge 1: Large training time. In the current architecture, the computation of the reward for each time step involves verifying the compatibility of the selected action with the current state. However, for large benchmarks such as the `MIPS` processor, this can take several seconds due to the large number of gates, and the agent requires millions of steps to learn. Consequently, the training time becomes impractical, making the basic architecture unsuitable for large-scale designs.

Solution 1. We address challenge 1 by reducing the frequency of reward computation, computing it only at the end of each episode. At all intermediate steps, the reward is set to 0. This approach speeds up the training by a factor of $\approx 86\times$, but the rewards become sparse, and it affects the performance of our agent. However, the impact on performance is only 5.6%, as shown in Table I.

E. Masking Actions for Efficiency

Challenge 2: Wasted efforts in choosing actions. Another challenge that the basic architecture suffers from is inefficiency in choosing actions. The available actions for the agent remain fixed at each step, regardless of the current state. Consequently, the agent ends up selecting an action that was previously chosen or will result in a set of rare nets that is not compatible. This leads to the agent wasting time on such steps, which is detrimental to its overall performance.

Solution 2. To improve the efficiency of action selection, we incorporate a masking strategy that limits the available actions to those that lead to new states based on the current state. Thus, we eliminate the time spent on actions that do not lead to new states. This approach also results in less sparse reward computation as the episode length decreases due to masking (the episode ends when no more actions are available). Since we eliminate actions from the state space, one may wonder if this approach may eliminate optimal actions. We now prove that this is not possible for our problem formulation.

Theorem 2. *Masking actions does not prevent our agent from learning anything that it could have learned otherwise.*

Proof. Let $\mathcal{P}'$ and $\mathcal{P}$ denote an agent that masks and does not mask actions, respectively. Suppose both $\mathcal{P}$ and $\mathcal{P}'$ are in state s. Let $\mathcal{A}$ denote the complete set of actions, and $\mathcal{A}_s$ denote the set of masked actions for state s. So, $\mathcal{A}_s = \{i | \{i\} \cup s \text{ is compatible and } i \notin s\}$ and $\mathcal{A}_s \subseteq \mathcal{A}$. If $\mathcal{P}$ chooses an action $a \in \mathcal{A} \setminus \mathcal{A}_s$ (i.e., an action in the set difference), then $\mathcal{P}$ will stay in the same state because the rare net corresponding to such an action a would either result in a set of rare nets that are not compatible or it would already

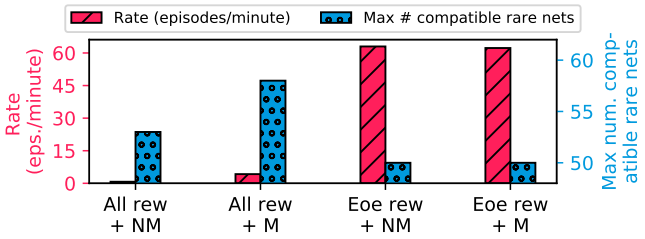

Fig. 2. Combinations of reward and masking methods for `MIPS`. Eoe: End-of-episode, M: Masking, NM: No masking

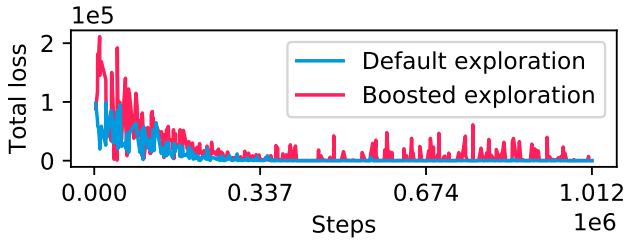

Fig. 3. Total loss trends in `c2670`: default vs. boosted exploration.

be in s. On the other hand, for any action $a' \in \mathcal{A}_s$ chosen by $\mathcal{P}$, agent $\mathcal{P}'$ can also choose the same action a' since it is in $\mathcal{A}_s$. Hence, masking does not prevent our agent from learning anything that the unmasked agent could have learned. ■

To enable masking, we compute pairwise compatibility of all rare nets using a SAT solver before training. For a design with R rare nets, ${}^{R}C_2$ SAT solver calls are required. However, since the compatibility computation for each unique pair is independent, these calls can be trivially parallelized to reduce runtime by a factor of $\mathcal{C}$, the number of available CPU threads. Thus, the runtime can be made arbitrarily small by increasing the number of threads, $\mathcal{C}$, for negligible cost. For instance, even for a large design with 1000 rare nets, using 250 AWS EC2 `t4g.small` instances with 2 vCPUs each would result in an equivalent of ${}^{1000}C_2/500 = 999$ consecutive SAT solver calls, which would cost less than \$4.2 and less than 20 minutes [25]. In our experiments, we parallelize the pairwise compatibility computation across 64 threads. During training, for a given state s (i.e., set of compatible rare nets at current step), all actions (i.e., rare nets) that will result in a set of rare nets that are not compatible with any of the rare nets in s are masked off and hence, are not chosen.

To design the best architecture, we implemented agents with all combinations of reward methods (at all steps and end-of-episode) and masking (with and without). The results in Fig. 2 demonstrate that to obtain the maximum number of compatible rare nets, the optimal architecture should mask actions based on state and provide rewards at each time step.

F. Boosting Exploration

Challenge 3: Convergence to local optima. Since the agent's objective is to generate maximal sets of rare nets, for certain benchmarks (for instance, `c2670`), the agent gets stuck in local optima. In other words, the agent quickly learns to capitalize on sub-optimal sets of compatible rare nets, thereby missing out on the diversity of the sets of compatible rare nets, resulting in poor trigger coverage.

Solution 3. To force the agent to explore, we (1) include an entropy term in the loss function of the agent and (2) control the smoothing parameter, λ, that affects the variance and the bias of the loss calculation.

To implement (1), we modify the total loss function to $l = l_\pi + c_\epsilon \times l_\epsilon + c_v \times l_v$, where l is the total loss, l_π is the loss of the policy network, l_ϵ is the entropy loss, l_v is the value loss, and c_ϵ and c_v are the coefficients for the entropy and value losses, respectively. We set $c_\epsilon = 1$. The entropy loss is inversely proportional to the randomness in the choice of actions. To implement (2), i.e., to increase the variance in the loss calculation, we set the parameter λ for policy loss l_π in PPO to 0.99, a high value. This leads to variance in the loss calculation and hence in the actions chosen by the agent.

Thus, we penalize the agent for having less variance in its choice of actions. Hence, the agent is forced to explore more and is likely to converge to a better state, i.e., a state with more compatible rare nets. Fig. 3 shows that by modifying the loss function and the smoothing parameter in PPO, the loss does not become 0 quickly, forcing the agent to explore more.

G. DETERRENT-LT: *Putting it All Together*

The final architecture of DETERRENT-LT is illustrated in Fig. 4. First, in the offline phase, we identify the rare nets of the design and generate pairwise compatibility information for them in a parallelized manner. Then, in each episode, the agent starts with a random rare net and selects an action according to the policy (a neural network) and the action mask, which limits the available actions based on the current state. The selected action is then evaluated to produce a reward for the agent, and the agent transitions to the next state. This process is repeated for T steps to complete one episode. The PPO algorithm translates the rewards into losses after a certain number of episodes (depending on the output of the policy network, which generates actions, and the value network, which predicts the expected reward of the action), which are used to update the parameters of the policy and value networks. Eventually, the agent learns the task, the losses become insignificant, and the reward saturates. Once the RL agent identifies the maximal sets of compatible rare nets, we choose the k largest sets for each of the R random initial states of the agent, remove duplicates from the combined $R \times k$ sets, and generate the test patterns for the unique sets using a SAT solver. We present the experimental results of DETERRENT-LT in Section V.

Complexity analysis. The computational complexity of DETERRENT-LT depends on the complexity of its three sub-steps: pairwise compatibility computation in the offline phase, the training of the RL agent, and the generation of the test patterns from the maximal sets of compatible rare nets found by the RL agent. Since the offline phase requires calls to a SAT solver and the complexity of the SAT problem is exponential in the number of variables (i.e., the number of nets in the design), the worst-case complexity of DETERRENT-LT is exponential in the size (i.e., the number of nets) of the design. However, in practice, our results on a variety of benchmarks show that DETERRENT-LT is capable of generating high-quality test patterns for large practical designs

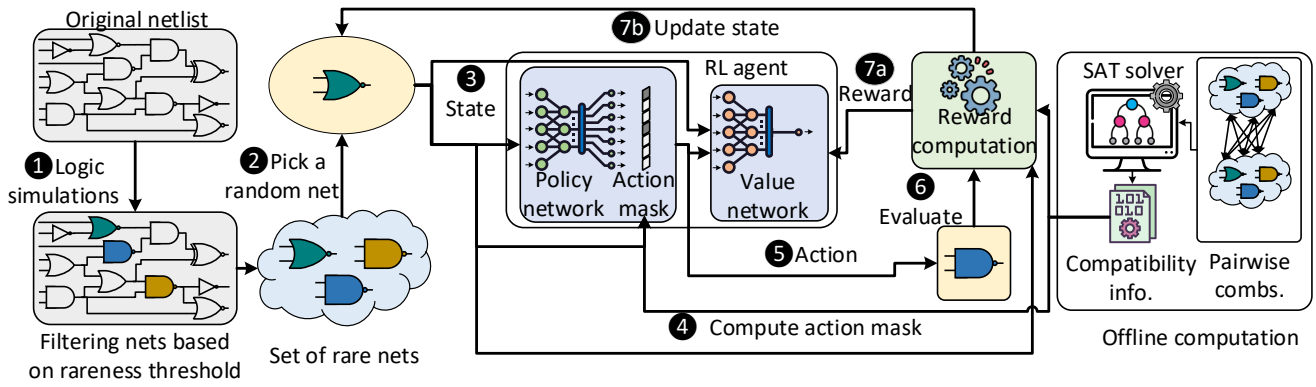

Fig. 4. Architecture of DETERRENT-LT.

well within the time-to-market constraints faced by the design houses (more details in Sec. V-G).

IV. DETERRENT-SC: Detecting Trojans using Reinforcement Learning and Side-channel Analysis

In this section, we demonstrate how we use RL to detect Trojans using side-channel analysis of dynamic current. First, we detail why we cannot simply use our previous RL formulation (developed for logic-testing-based detection) here. Based on our observations, we then design an RL agent specifically for side-channel-based Trojan detection. Next, we illustrate some limitations of this preliminary formulation and provide domain-specific techniques that lead to an effective and efficient RL agent capable of detecting Trojans using side-channel analysis.

A. Differences between Logic Testing and Side-channel Approaches

Here, we explain why we need to design a new RL agent for side-channel-based Trojan detection by discussing the fundamental differences between logic testing and side-channel-based detection techniques.

1) Difference in Objectives: Logic-testing techniques are designed to achieve a high trigger activation rate. They do so by generating test patterns with the objective of maximizing the number of activated rare nets. However, detecting Trojans using side-channel analysis requires generating test patterns that maximize the side-channel sensitivity, which is different from just activating rare nets. Maximizing the dynamic current side-channel sensitivity requires us to balance two competing objectives: (i) Maximize current switching in rare nets and (ii) Minimize current switching in the rest of the circuit (i.e., non-rare nets). Simply generating test patterns that maximize the number of activated rare nets (using logic-testing-based techniques) is likely to maximize switching in rare nets. However, such test patterns also cause more switching in non-rare nets, thereby reducing the side-channel sensitivity.

2) Difference in Test Patterns: Another key difference in side-channel-based Trojan detection stems from the fact that here, the objective relates to switching (i.e., transitions in nets' values) rather than just activation of nets. To cause switching, we need to excite the design-under-test with pairs of dependent test patterns rather than just independent test patterns. In other words, to calculate the side-channel sensitivity, we need to find pairs of test patterns that maximize side-channel sensitivity (i.e., maximize switching in rare nets and minimize switching in non-rare nets). This is in contrast to logic testing, where the response of each test pattern is monitored independently.

Thus, these two differences imply that we cannot simply use the RL agent from Section III to detect Trojans using side-channel analysis. We need a new formulation that generates pairs of test patterns with the objective of maximizing switching in rare nets and minimizing switching in the non-rare nets in the design.

So, our goal is to generate a compact set of ordered pairs of test patterns $\{(u_i, v_i)\}$ (for $i = 1, 2, \ldots, TP_{sc}$) that can maximize the dynamic current based side-channel sensitivity. For each pair of test patterns (u_i, v_i), the switching in the original Trojan-free design is measured by exciting the design with u_i followed by v_i. We denote this switching by $\text{switch}^{O}_{u_i,v_i}$, where O denotes the original, Trojan-free design. We measure the switching in a Trojan-infested design O^T similarly, and we denote it by $\text{switch}^{O^T}_{u_i,v_i}$. Based on these notations, we can define the relative switching in a Trojan-infested design (relative to the original, Trojan-free design) as $|\text{switch}^{O}_{u_i,v_i} - \text{switch}^{O^T}_{u_i,v_i}|/\text{switch}^{O}_{u_i,v_i}$. Then, following prior work, the sensitivity of a Trojan T is defined as the maximum of the relative switching over all test pattern pairs:

$$\text{sensitivity}^T = \max_{i=1,2,\ldots,TP_{sc}} \left(\frac{|\text{switch}^{O}_{u_i,v_i} - \text{switch}^{O^T}_{u_i,v_i}|}{\text{switch}^{O}_{u_i,v_i}} \right) \quad (2)$$

We divide the task of generating a compact set of ordered pairs of test patterns that maximize sensitivity into two subtasks: (i) Generation of a compact set of first patterns (i.e., u_i's of the ordered pairs) that maximize the number of activated rare nets, and (ii) Given u_i's from the previous task, find

Algorithm 1: Generating compact set of first patterns

Input: Circuit netlist, set of rare nets RN, number of iterations M
Output: Set of first patterns U
Initialization: $U \leftarrow \phi$; compute Boolean expressions $r.expr\ \forall r \in RN$; compatible rare nets $C \leftarrow \phi$

1 **for** $k = 1, 2, 3, \ldots, M$ **do**
2 Initialize Boolean expression $B \leftarrow$ True
3 $RN_{tmp} \leftarrow RN$
4 $C_{tmp} \leftarrow \phi$
5 **while** $RN_{tmp} \neq \phi$ **do**
6 randomly pick and remove a rare net r from RN_{tmp}
7 **if** *satisfiable*$(B \wedge (r.expr == r.rv))$ **then**
8 $B \leftarrow B \wedge (r.expr == r.rv)$
9 $C_{tmp} \leftarrow C_{tmp} \cup \{r\}$
10 **end**
11 **if** $|C_{tmp}| > TriggerLimit$ **then**
12 break
13 **end**
14 **end**
15 **if** $C_{tmp} \notin C$ **then**
16 $C \leftarrow C \cup C_{tmp}$
17 solve B to get test pattern u
18 $U \leftarrow U \cup \{u\}$
19 **end**
20 **end**
21 **return** U

the corresponding second patterns (i.e., v_i's) that maximize switching in rare nets and minimize switching in non-rare nets.

For the first task, we use an algorithm similar to MaxSense [19], albeit with a critical post-processing step, to ensure the compactness of the final set of test pattern pairs. For the second task, we devise a novel RL agent. Next, we describe our algorithms for both of these tasks.

B. Task 1: Generation of First Patterns

Before detailing the algorithm, we describe the concept of Boolean expressions for rare nets that will be used in the algorithm. For a rare net, its Boolean expression is defined as an expression that consists of the primary inputs in the fan-in cone of the net. For example, if a net n_1 and n_2 are inputs of a NOR gate and n_3 is the output rare net, the Boolean expression for the rare net n_3 is $!(n_1 \vee n_2)$ (assuming n_1 and n_2 are primary inputs). This Boolean expression is stored in $n_3.expr$, and the rare value of n_3 is stored in $n_3.rv$.

Algorithm 1 details the procedure to generate a compact set of first patterns (U) for a given circuit netlist with a pre-determined set of rare nets and a user-defined number of iterations M. Before the main loop begins, we initialize the set of first patterns as empty, compute Boolean expressions for all rare nets in the design, and initialize an empty set C to keep track of already covered combinations of rare nets. The algorithm has M iterations (line 1). In each iteration, we first create a copy (RN_{tmp}) of the set of rare nets. Then, we run a while loop that iterates over all rare nets in RN_{tmp} (lines 5 to 14). In each iteration of the while loop, we pick and remove a rare net r from RN_{tmp} and evaluate if $B \wedge (r.expr == r.rv)$ is satisfiable. In other words, we check if the chosen rare net r will result in a compatible set of rare nets with the rare nets chosen in the past iterations of the while loop (C_{tmp}). If so, we add the Boolean expression for the rare net r ($(r.expr == r.rv)$) to B and add the rare net r to C_{tmp} (lines 8 and 9). When C_{tmp} contains sufficient compatible rare nets, i.e., $TriggerLimit$, we break out of the while loop. Once we exit the while loop, we check if C_{tmp} has been covered in the past or not (line 15). If not, we solve the corresponding Boolean expression B and add the obtained test pattern u to our set of first patterns U (line 18). The set of first patterns U is returned when the M iterations are done.

Thus, Algorithm 1 generates a set of first patterns U. These first patterns are then provided as inputs to our RL agent, which generates the best second patterns, as explained next.

C. Task 2: Generation of Second Patterns using Reinforcement Learning

The objective of the RL agent is to generate second patterns (v_i's $\in V$) corresponding to the first patterns (u_i's $\in U$) such that the switching in the rare nets is maximized and switching in the non-rare nets is minimized. To that end, we formulate the MDP as follows.

- **States** $\mathcal{S}$ is the set of all binary strings of length I, where I is the number of inputs in the given design. An individual state s_t represents the test pattern chosen by the agent at time t. In software implementation, we encode s_t as an I-bit binary string.
- **Actions** $\mathcal{A}$ is the set of integers $\{0, 1, \ldots, I-1\}$. An individual action a_t is the index of the state s_t that the agent chooses to flip (i.e., 1 to 0 or 0 to 1) at time t. In software implementation, a_t is a scalar integer that can take a value from 0 to $I-1$.
- **State transition** $P(s_{t+1}|a_t, s_t)$ is the probability that action a_t in state s_t leads to the state s_{t+1}. In our case, the state transition is deterministic because the next state s_{t+1} is obtained by flipping the a_tth bit in the current state s_t.
- **Reward function** $\mathcal{R}(s_t, a_t)$ is determined based on whether the time step t is the last step of the episode or not. Let E be the length of the episodes.[6] If t is not the last step i.e., $t \neq E$, the reward is 0. Otherwise, the reward is obtained by evaluating the amount of switching caused by the test patterns (u, s_t). Since the objective here is to maximize switching in rare nets and minimize switching in other nets, we define the reward as the ratio of the switching in rare nets (rare_switch) to the switching (switch) in all nets in the original design O.

$$\mathcal{R}(s_t, a_t) = \begin{cases} \dfrac{\text{rare\_switch}^O_{u,s_t}}{\text{switch}^O_{u,s_t}}, & \text{if } t = E \\ 0, & \text{otherwise} \end{cases}$$

[6] E is a hyperparameter that we decide before training.

TABLE II
PERCENTAGE OF INPUTS PRUNED OFF FOR MIPS, AES, GPS, MOR1KX.

MIPS	AES	GPS	mor1kx
$98.42 \pm 0.09\%$	$95.62 \pm 0.09\%$	$93.85 \pm 0.27\%$	$97.79 \pm 0.18\%$

- **Discount factor** γ $(0 \leq \gamma \leq 1)$ indicates the importance of future rewards relative to the current reward.

The above RL agent is trained to generate the best second pattern for a given first pattern (u) from Task 1. Hence, for all the first patterns from Task 1, we train a separate RL agent. Since each training is independent, we parallelize the trainings across multiple CPU cores. For each individual training, the initial state s_0 is the given first pattern u for which the agent tries to find the best second pattern. At each step t, the agent in state s_t chooses an action a_t, arrives in state s_{t+1} according to the transition function $P(s_{t+1}|a_t, s_t)$, and receives a reward according to $\mathcal{R}(s_t, a_t)$. This cycle of state, action, reward, and next state is repeated E times, where E is the length of the episode determined by the designer before training. At the end of each episode, the state of the agent reflects a candidate for the second pattern corresponding to the given first pattern u. Once this agent is trained, it converges to the best second pattern corresponding to the given first pattern u.

The above RL agent provides test patterns that result in high reward for small benchmarks with a limited number of inputs such as c2670, c7552, s15850, s35932, etc. However, for large benchmarks such as MIPS, which has 4411 inputs, the agent fails to generate test patterns with high reward. We address this shortcoming by pruning the search space for the agent, as explained next.

D. Pruning Search Space

Since, at each time step, the agent needs to decide which input bit to flip, a large number of inputs mean that the search space for the agent is humongous (in fact, the search space grows exponentially with the number of inputs). So, to help the agent find the best second patterns efficiently, we prune out certain inputs, thereby reducing the search space for the agent. However, this pruning needs to be done carefully because pruning inputs randomly can adversely affect the quality of the test patterns and result in low switching. We need to prune only those inputs that are not likely to improve sensitivity. This will allow the agent to explore more patterns that are likely to result in good side-channel sensitivity.

To this end, we characterize the design before training as follows. We find the union of all inputs that influence any rare nets activated by the first pattern, u. This is done by tracing the fan-in cones of all the rare nets activated by u. Then, all inputs not influencing any of the rare nets are pruned off since flipping them is not going to switch the rare nets (i.e., it is not going to increase the numerator in the reward function), but flipping them may or may not cause non-rare nets to switch (i.e., increase the denominator in the reward function). Hence, performing this fan-in-based characterization allows us to prune the search space for the agent, resulting in better second patterns and, thus, better side-channel sensitivity.

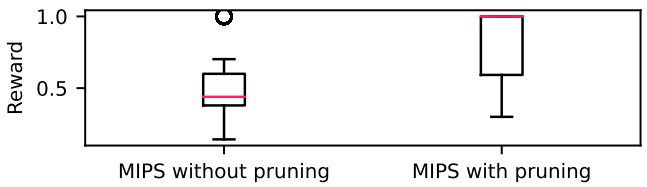

Fig. 5. Distributions of rewards without and with pruning for MIPS.

Table II shows the average percentage of inputs that are pruned off for the large benchmarks, MIPS, AES, GPS, and mor1kx. This analysis shows that pruning is essential for large benchmarks because most ($> 90\%$) of the inputs do not contribute to increasing side-channel sensitivity at all, so they are pruned off from search space for the agent.

Note that although this pruning of search space and masking of actions for DETERRENT-LT (Section III-E) are solutions designed to increase the efficiency of the respective agents, they are fundamentally different concepts. Masking for DETERRENT-LT is concerned with the actions available to the agent, whereas pruning for DETERRENT-SC is concerned with the state space of the agent. Masking reduces the number of actions available to the agent so that at each time step, the agent does not choose actions that do not lead to a new state. This is achieved by pre-calculating the compatibility of rare nets in DETERRENT-LT. On the other hand, pruning here reduces the state space before training even begins by analyzing the structure of the given design. Thus, although they achieve the same goal of increasing the efficiency of the RL agents, their methods are completely different and specific to the requirements of the two agents.

Fig. 5 shows the improvement in the quality of the test patterns (measured through the rewards) generated by the agent when pruning is used as opposed to when it is not used for MIPS. The agent's median reward (i.e., rare_switch/switch) after training increases 56% (from 44% to 100%) when pruning is used.

E. DETERRENT-SC: *Putting it All Together*

Fig. 6 shows the final DETERRENT-SC framework. First, we obtain the rare nets from the original Trojan-free design. Then, we run Algorithm 1 to generate the first patterns U. Then, the fan-in analysis for pruning is performed to prune off the search spaces (represented by the X's in the patterns in Fig. 6) for each first pattern in U. Then, the parallel RL agent trainings begin to generate the second patterns. Each training is for an individual first pattern. The agent starts with that first pattern as the initial state and takes actions to flip an input bit. The action leads the agent to the next state. This procedure repeats for E steps (i.e., an episode). At the end of the episode, the state of the agent is a candidate second pattern v_i, which is evaluated to produce a reward. After a certain number of episodes, the PPO algorithm translates the rewards into losses which are used to update the parameters of the neural networks that make up the agent. After a sufficient number of episodes, the losses diminish, and the reward saturates, i.e., the agent converges. Then, we select the best candidate test pattern (i.e., the one with the highest reward) as the final second pattern.

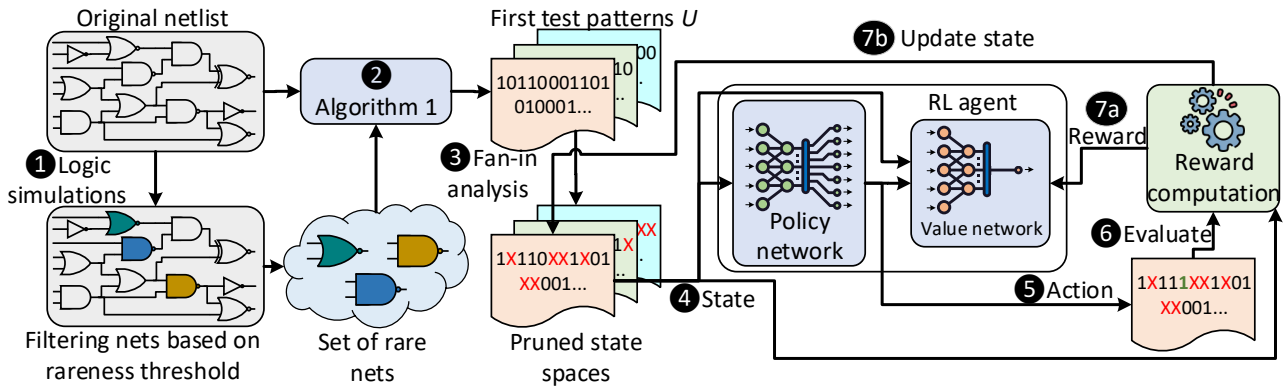

Fig. 6. Architecture of DETERRENT-SC.

Thus, we obtain the first patterns and the corresponding second patterns that maximize the side-channel sensitivity.

Complexity analysis. The computational complexity of DETERRENT-SC depends on the complexity of its two sub-steps: Algorithm 1 for generating a compact set of first patterns and the training of the RL agent. Since Algorithm 1 invokes a SAT solver, the worst-case complexity of DETERRENT-SC is exponential in the size of the design. However, in practice, our results on a variety of benchmarks show that DETERRENT-SC is capable of generating high-quality test patterns for large practical designs well within the time-to-market constraints faced by the design houses (more details in Sec. V-G).

V. Experimental Evaluation

A. Experimental Setup

We implemented our RL agents using *PyTorch1.6* and trained them using a Linux machine with Intel 2.4 GHz CPUs and an NVIDIA Tesla K80 GPU. We implemented the parallelized version of TARMAC in *Python 3.6*. We also implemented MERS and its extensions as well as the parallelized version of MaxSense in *Python 3.6*. We used the SAT solver provided in the *pycosat* library. We used *Synopsys VCS* for logic simulations and evaluating test patterns on Trojan-infested netlists. Similar to prior works (MERO, TARMAC, TGRL, MERS, MERS-h, MERS-s, and MaxSense), for sequential circuits, we assume full scan access [14]–[16], [18], [19]. To enable a fair comparison, we implemented and evaluated all the techniques on the same benchmarks as the prior works. The authors of TGRL provided us with the benchmarks. They also provided us with the TGRL test patterns. Additionally, we also performed experiments on the `MIPS` processor from OpenCores [24], `AES` core [26], `GPS` module [27], and the `mor1kx` processsor [28] to demonstrate scalability. For these large designs, we use vectorized environments with 32 parallel processes to speed up the training of DETERRENT-LT. We set k, the number of top maximal sets of compatible rare nets in DETERRENT-LT, as 30 because we observe empirically that it results in a compact set of test patterns that achieve high trigger coverage. For DETERRENT-SC, we set the hyperparameter E (from the reward function in Sec. IV-C) to be 5 meaning that for each episode, the number of steps is fixed to 5 since prior work has shown that flipping a couple of bits is enough to produce high sensitivity [19]. For evaluation, we generated 100 Trojan-infested netlists for each design, and we verified the validity of the Trojans with a Boolean satisfiability check.

B. Trigger Coverage Performance of DETERRENT-LT

This section provides a comparison of the trigger coverage provided by different techniques (Table III). Apart from evaluating MERO, TARMAC, and TGRL, we also compare the performance of DETERRENT-LT with random test patterns and patterns generated from an industry-standard tool, *Synopsys TestMAX* [13]. To ensure a fair comparison, we used the number of patterns from TGRL as a reference for the random test patterns, MERO, and TARMAC. For TestMAX, the number of patterns is determined by the tool in the default setting (`run_atpg`).

Note that the netlists corresponding to the test patterns provided by TGRL's authors for `s13207`, `s15850`, and `s35932` were not available to us at the time of writing the manuscript. As a result, we could only assess the TGRL test patterns for those circuits on our benchmarks, which led to low trigger coverage for TGRL in these cases. Furthermore, TGRL does not evaluate on the `MIPS` benchmark. Hence the corresponding cells in the table are empty. To perform a fair comparison with TGRL, we have excluded `s13207`, `s15850`, `s35932`, and `MIPS` from the average trigger coverage calculations. Similarly, we have omitted `MIPS` from the average test length calculations for all techniques in Table III.

DETERRENT-LT delivers superior trigger coverage compared to all other methods while using fewer test patterns. On average, it outperforms random patterns (by 68%), TestMAX (by 85.75%), MERO (by 23%), TARMAC (by 12.25%), and TGRL (by 9.25%), and significantly reduces the number of test patterns required by prior work (MERO, TARMAC, and TGRL) by up to two orders of magnitude (a reduction of 169.68× on average overall and 27.59× on average over large benchmarks, i.e., `MIPS`, `AES`, `GPS`, and `mor1kx`).

TABLE III
COMPARISON OF TRIGGER COVERAGE (COV. (%)) AND TEST LENGTH OF DETERRENT-LT WITH RANDOM SIMULATIONS, SYNOPSYS TESTMAX [13], MERO [14], TARMAC [15], AND TGRL [16]. EVALUATION IS DONE ON 100 RANDOM FOUR-WIDTH TRIGGERED TROJAN-INFESTED NETLISTS.

Design	Number of rare nets	# Gates	Random		TestMAX [13]		MERO [14]		TARMAC [15]		TGRL [16]		**DETERRENT-LT (this work)**		
			Test Length	Cov. (%)	Test Length	Cov. (%)	Test Length	Cov. (%)	Test Length	Cov. (%)	Test Length	Cov. (%)	Test Length	Patterns Red./ TARMAC & TGRL	Cov. (%)
c2670	43	775	5306	10	89	27	5306	55	5306	100	5306	96	8	**663.25×**	100
c5315	165	2307	8066	37	103	5	8066	89	8066	61	8066	94	1585	**5.08×**	99
c6288	186	2416	3205	54	38	4	3205	96	3205	100	3205	85	2096	**1.52×**	99
c7552	282	3513	9357	10	137	4	9357	51	9357	73	9357	71	5910	**1.58×**	85
s13207	604	1801	9659	3	106	4	9659	11	9659	80	9659	5	9600	**1.006×**	80
s15850	649	2412	9512	3	110	3	9512	17	9512	79	9512	8	6197	**1.53×**	81
s35932	1151	4736	3083	99	37	68	3083	100	3083	100	3083	58	6	**513.83×**	100
MIPS	1005	23511	25000	0	796	0	25000	0	25000	100	—	—	1304	**19.17×**	97
AES	854	161161	25000	0	192	0	25000	0	25000	52	—	—	953	**26.23×**	50
GPS	853	193141	25000	0	192	0	25000	0	25000	38	—	—	887	**28.18×**	33
morlkx	641	158265	25000	0	864	13	25000	1	25000	99	—	—	679	**36.81×**	98
Avg.	585	50376	6884‡	27.75†	88.57‡	10†	6884‡	72.75†	6884‡	83.5†	6884‡	86.5†	3628.85‡	**169.68×‡**	**95.75†**

For fair comparison with TGRL [16]: †Averaged over c2670, c5315, c6288, c7552; ‡MIPS, AES, GPS, morlkx not included.

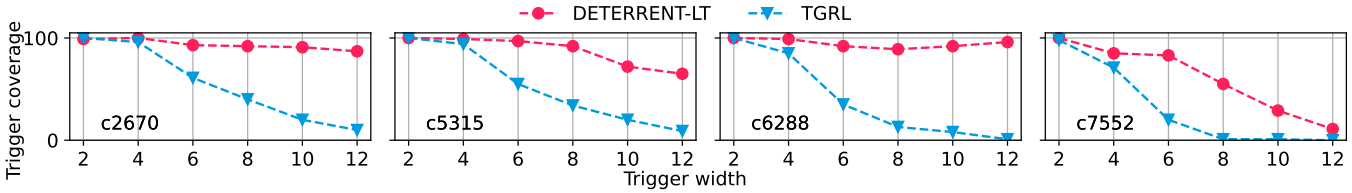

Fig. 7. Impact of trigger width on the trigger coverage of TGRL [16] and DETERRENT-LT for c2670, c5315, c6288, and c7552.

C. *Impact of Trigger Width on* DETERRENT-LT

Trigger width, i.e., the number of rare nets that constitute the trigger, is a critical factor in determining the Trojan's stealth. Increasing the trigger width can make it exponentially more difficult to activate the trigger. For instance, if the trigger width is 4 and the rareness threshold is 0.1, the probability of activating the trigger through random simulation is 10^{-4} (assuming that the 4 rare nets are independent). However, if the trigger width is 12, the probability decreases to 10^{-12} (again, assuming that the 12 rare nets are independent).[7] Thus, it is crucial to maintain performance as the trigger width increases.

Fig. 7 illustrates the results for c2670, c5315, c6288, and c7552, which were chosen due TGRL's high trigger coverage. TGRL's performance decreases significantly with an increasing trigger width. On the other hand, DETERRENT-LT delivers consistent trigger coverage across most benchmarks and consistently outperforms TGRL, indicating its ability to activate extremely rare trigger conditions.[8]

D. *Impact of Rareness Threshold on* DETERRENT-LT

The rareness threshold represents the probability below which nets are considered rare, indicating that their logic values strongly tend towards either 0 or 1. As the rareness threshold increases for a given trigger width (α), the number of rare nets also increases (say, by a factor of β), leading to a greater number of possible combinations for constructing the trigger, which becomes much more difficult to activate (by a factor of β^{α}). Fig. 8 shows that as the rareness threshold increases, the number of rare nets also increases (up to $64\times$ more potential trigger combinations), but DETERRENT-LT still achieves similar trigger coverage (with a drop of no more than 2%) using fewer than 2500 patterns.[9]

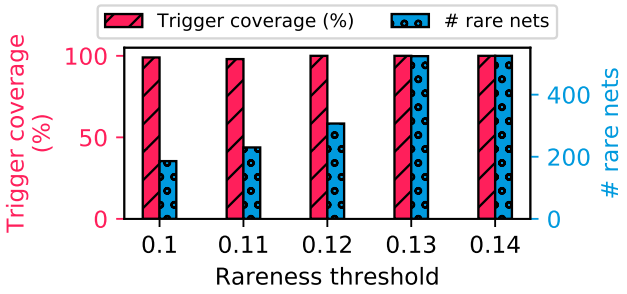

Fig. 8. Impact of rareness threshold on the number of rare nets and the trigger coverage of DETERRENT-LT for c6288.

In a different experiment, we trained the agent using rare nets with a threshold of 0.14 and evaluated the generated test patterns on rare nets with a threshold of 0.1. The results showed that the trigger coverage was as high as 99%. This suggests that we can train the agent for a large set of rare nets and then employ it to generate patterns for a smaller subset of rare nets. Note that as the number of rare nets increases, say from R to $2R$, the number of initial states for DETERRENT-LT also increases to $2R$. So, to have the same number of episodes (i.e., training epochs) for each initial state as before, the total number of episodes doubles,

[7]Note that in general, this assumption does not always hold. Trojans in real designs can be made from rare nets that are not independent. However, this example is meant to illustrate that the difficulty of activating a trigger increases (as high as exponentially) with a linear increase in trigger width.

[8]Following prior work, since the Trojans are generated randomly for each trigger width, it leads to some Trojans being difficult to activate for a trigger width of 8 for c6288. Hence, its trigger coverage is lower than that for trigger widths of 10 and 12.

[9]TGRL test patterns were not provided for thresholds other than 0.1, so no comparison was made with TGRL for other thresholds.

TABLE IV
COMPARISON OF SIDE-CHANNEL SENSITIVITY (SENS. (%)) FOR TEST PATTERNS FROM MERS [18], MERS-H [18], MERS-S [18], MAXSENSE [19], AND DETERRENT-SC. EVALUATION IS DONE ON 100 RANDOM EIGHT-WIDTH TRIGGERED TROJAN-INFESTED NETLISTS.

Design	MERS [18]		MERS-h [18]		MERS-s [18]		MaxSense [19]		**DETERRENT-SC (this work)**		
	Test Length	Sens. (%)	Test Length	Sens. (%)	Test Length	Sens. (%)	Test Length	Sens. (%)	Test Length	Patterns Red./ MaxSense	Sens. (%)
c2670	10125	1.68	10125	1.93	10125	2.32	10000	194.47	28	**357**×	126.35
c5315	16861	0.47	16861	0.57	16861	0.72	10000	18.16	7448	**1.34**×	17.58
c6288	8255	0.84	8255	1.80	8255	1.83	10000	7.93	6404	**1.56**×	7.88
c7552	28070	0.32	28070	0.39	28070	0.45	10000	34.97	7496	**1.33**×	39.66
s13207	28386	0.68	28386	0.73	28386	0.78	10000	62.45	10000	**1**×	58.36
s15850	27380	0.50	27380	0.55	27380	0.58	10000	33.93	10000	**1**×	32.85
s35932	5109	0.39	5109	0.45	5109	0.41	20000	24.62	5146	**3.88**×	28.24
MIPS	289	0.06	289	0.06	289	0.06	20000	5.71	6702	**2.98**×	112.96
AES	16	0.004	16	0.004	16	0.004	30000	0.05	14126	**2.12**×	0.92
GPS	6	0.004	6	0.004	6	0.004	30000	0.19	3874	**7.74**×	12.49
mor1kx	2	0.003	2	0.003	2	0.003	30000	0.85	14752	**2.03**×	34.94
Avg.	11318	0.45	11318	0.59	11318	0.65	17273	34.85	7816	**34.73**×	**42.93**

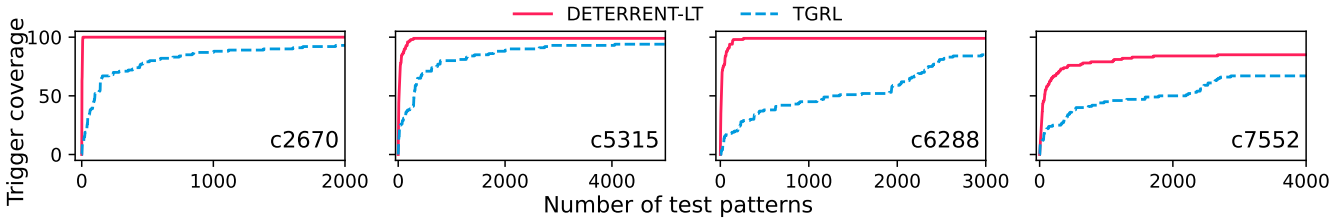

Fig. 9. Trigger coverage vs. number of test patterns comparison of TGRL [16] with DETERRENT-LT.

and hence, the training time doubles. In other words, the training time increases linearly with the number of rare nets. However, as explained above, DETERRENT-LT's trigger coverage performance does not degrade.

E. Trigger Coverage vs. Number of Patterns for DETERRENT-LT

Next, we examine the influence of test patterns on trigger coverage. We perform experiments to measure the increase in trigger coverage provided by each test pattern for both DETERRENT-LT and TGRL. The results are shown in Fig. 9. We observe that DETERRENT-LT achieves a high trigger coverage with only a few test patterns, while TGRL requires significantly more test patterns to obtain similar coverage. This indicates that DETERRENT-LT is better than TGRL in terms of utilizing feedback to explore new areas of the design, even though both techniques use RL.

F. Performance of DETERRENT-SC

In this section, we compare the side-channel sensitivity of DETERRENT-SC with prior techniques (MERS, MERS-h, MERS-s [18]), including the state-of-the-art technique MaxSense [19] (Table IV). The results demonstrate that DETERRENT-SC reduces the number of required test patterns for most of the benchmarks. On average over all benchmarks, it obtains a $34.73\times$ reduction in the number of test patterns over MaxSense, and a $3.72\times$ reduction on average over large benchmarks, i.e., MIPS, AES, GPS, and mor1kx. Moreover, this reduction in test patterns does not affect the side-channel sensitivity adversely. Rather, DETERRENT-SC achieves 42%, 42%, 42%, and 8% higher average side-channel sensitivity than MERS, MERS-h, MERS-s, and MaxSense, respectively.

G. Distribution of DETERRENT-SC *Rewards*

Here, we analyze the performance of DETERRENT-SC in greater detail by observing the rewards it converges to. Recall that the reward of DETERRENT-SC is defined as the ratio of switches in the rare nets (i.e., rare_switch) over the total switching in the design (i.e., switch). A higher value of reward indicates that the test patterns found by the agent are more likely to increase the side-channel footprint of a Trojan, leading to easy Trojan detection. Fig. 10 demonstrates the distribution of the rewards for the final test patterns generated by our technique. The red line indicates the median, the box extends from the first quartile (i.e., 25^{th} percentile) to the third quartile (i.e., 75^{th} percentile) of the data, and the whiskers extend from the box by $1.5\times$ the inter-quartile range (i.e., the difference between the third quartile and the first quartile). DETERRENT-SC converges to test patterns with very high rewards (most benchmarks have a median higher than 0.75),

TABLE V
RUNTIMES (IN HOURS; ROUNDED UP TO 30 MINUTES) OF DETERRENT-LT AND DETERRENT-SC.

Benchmark	c2670	c5315	c6288	c7552	s13207	s15850	s35932	MIPS	AES	GPS	mor1kx
DETERRENT-LT	1	1	2	2	2	2	2.5	13	37	42	31
DETERRENT-SC	1	1	1.5	1.5	1.5	2	3	18	41	45	40

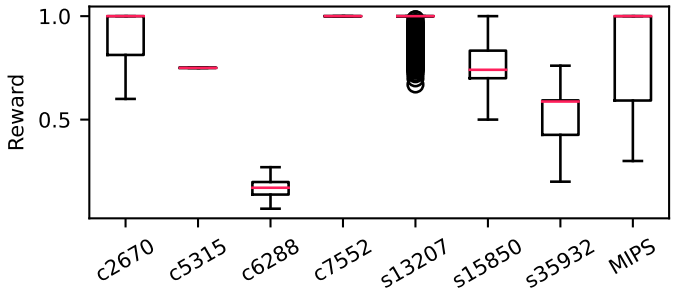

Fig. 10. Distribution of rewards DETERRENT-SC test patterns.

with c6288 being an exception. The reason for the bad performance for c6288 is its structure. c6288, a 16×16 multiplier, has a structure such that activating/deactivating rare nets requires the signal to propagate through several non-rare nets, so switching rare nets also causes significant switching in non-rare nets, leading to a low rare_switch/switch ratio. However, note that despite this, DETERRENT-SC achieves a $1.56\times$ reduction in number of test patterns for c6288 without causing significant reduction in side-channel sensitivity compared to MaxSense (see Table IV).

H. Runtime Analysis

Table V shows the runtimes of DETERRENT-LT and DETERRENT-SC. A small runtime is critical to ensure that test patterns are available by the time the chips are fabricated and ready for testing. Typically, the cycle time for fabrication is $\approx$ 3 months [29]. As the results demonstrate, both our techniques generate high-quality test patterns well within the time constraint the design houses face.

VI. DISCUSSION

Comparison with preliminary work [30]. A preliminary version of this work details an RL-based technique for detecting Trojans using logic testing [30]. However, unlike this work, the preliminary version does not provide a mathematical formulation of the hardware Trojan detection problem. Moreover, in this work, we also provide a method to solve it as an optimization problem and prove the optimality and compactness of the test patterns obtained by solving the problem. We also augment the analyses of the preliminary version with detailed experiments on all relevant benchmarks as well a comparison with one more pioneering prior work, MERO [14]. Additionally, the preliminary version focuses only on logic testing, whereas in this work, we also design a novel RL-based framework for detecting Trojans using side-channel analysis and provide detailed experimental evidence supporting the effectiveness of our novel agent (ref. Sections IV and V).

Comparison with TGRL [16]. The DETERRENT-LT agent architecture differs completely from TGRL. While TGRL aims to maximize a heuristic based on the rareness and testability of nets, we approach the problem of trigger activation as a set-cover problem, seeking maximal sets of compatible rare nets. TGRL generates test patterns by probabilistically flipping bits, while our agent generates maximal sets of compatible rare nets. Due to our formulation, we achieve much better coverage with far fewer test patterns than TGRL (see Section V).

VII. CONCLUSION

Prior works on Trojan detection have shown a reasonable success rate, but they are ineffective, not scalable, or require a large number of test patterns. To address these limitations, we developed a suite of RL agents to detect Trojans using logic testing and side-channel analysis. However, to design these agents, we face several challenges, such as inefficiency and lack of scalability. We overcome these challenges using different features such as masking, boosting exploration, and pruning the state space. As a result, our final RL agents generate compact sets of test patterns for designs of all sizes and complexities, including the MIPS processor, AES core, GPS module, and the mor1kx processor. Experimental results demonstrate that our logic-testing-based RL agent reduces the number of test patterns by $169.68\times$ on average over all benchmarks and $27.59\times$ on average over large benchmarks (MIPS, AES, GPS, mor1kx) while improving trigger coverage. Further evaluations show that our agent is robust against increasing complexity. Our agent maintains steady trigger coverage for different trigger widths, whereas the state-of-the-art technique's performance drops drastically. Our agent also maintains performance against the increasing number of possible trigger combinations. Likewise, our side-channel-based RL agent reduces the number of test patterns by $34.73\times$ on average over all benchmarks and $3.72\times$ on average over large benchmarks while improving the Trojan detection performance by 8%. A deeper analysis of this agent also reveals that it achieves significant switching in Trojan-prone regions of the circuit while minimizing the switching in the rest of the circuit, leading to better side-channel leakage. Although this work demonstrates the power of RL for Trojan detection, the fundamental idea and the challenges related to scalability and efficiency are not specific to the current problem. The ways in which we overcame the challenges can be used to develop better defenses for other hardware security problems.

REFERENCES

[1] G. Caminero, M. Lopez-Martin, and B. Carro, "Adversarial environment reinforcement learning algorithm for intrusion detection," *Computer Networks*, vol. 159, pp. 96–109, 2019.

[2] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. B. Abu-Ghazaleh, "SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning." in *USENIX Security Symposium*, 2021, pp. 2741–2758.

[3] H.-D. Tran, F. Cai, M. L. Diego, P. Musau, T. T. Johnson, and X. Koutsoukos, "Safety verification of cyber-physical systems with reinforcement learning control," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–22, 2019.
[4] T. T. Nguyen and V. J. Reddi, "Deep reinforcement learning for cyber security," *IEEE TNNLS*, pp. 1–17, 2021.
[5] H. Chen, X. Zhang, K. Huang, and F. Koushanfar, "AdaTest: Reinforcement learning and adaptive sampling for on-chip hardware Trojan detection," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 2, pp. 1–23, 2023.
[6] V. Gohil, H. Guo, S. Patnaik, and J. Rajendran, "ATTRITION: Attacking Static Hardware Trojan Detection Techniques Using Reinforcement Learning," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1275–1289.
[7] H. Guo, S. Saha, S. Patnaik, V. Gohil, D. Mukhopadhyay, and J. Rajendran, "Vulnerability Assessment of Ciphers To Fault Attacks Using Reinforcement Learning," *Cryptology ePrint Archive*, 2022.
[8] S. Patnaik, V. Gohil, H. Guo, and J. Rajendran, "Reinforcement Learning for Hardware Security: Opportunities, Developments, and Challenges," in *IEEE 19th International SoC Design Conference*, 2022, pp. 217–218.
[9] L. Lin, M. Kasper, T. Güneysu, C. Paar, and W. Burleson, "Trojan side-channels: Lightweight hardware trojans through side-channel engineering," in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2009, pp. 382–395.
[10] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, "A2: Analog Malicious Hardware," in *2016 IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 2016, pp. 18–37.
[11] A. Baumgarten, M. Steffen, M. Clausman, and J. Zambreno, "A case study in hardware Trojan design and implementation," *International Journal of Information Security*, vol. 10, no. 1, pp. 1–14, 2011.
[12] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: Lessons learned after one decade of research," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 1, pp. 1–23, 2016.
[13] Synopsys, "TestMAX ATPG and TestMAX Diagnosis User Guide," Version S-2021.06-SP1, 2021.
[14] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "MERO: A statistical approach for hardware Trojan detection," in *Cryptographic Hardware and Embedded Systems-CHES 2009: 11th International Workshop Lausanne, Switzerland, September 6-9, 2009 Proceedings*. Springer, 2009, pp. 396–410.
[15] Y. Lyu and P. Mishra, "Scalable Activation of Rare Triggers in Hardware Trojans by Repeated Maximal Clique Sampling," *IEEE TCAD*, vol. 40, no. 7, pp. 1287–1300, 2021.
[16] Z. Pan and P. Mishra, "Automated test generation for hardware Trojan detection using reinforcement learning," in *Proceedings of the 26th ASP-DAC*, 2021, pp. 408–413.
[17] S. Narasimhan *et al.*, "Hardware Trojan detection by multiple-parameter side-channel analysis," *IEEE Transactions on computers*, vol. 62, no. 11, pp. 2183–2195, 2012.
[18] Y. Huang, S. Bhunia, and P. Mishra, "MERS: statistical test generation for side-channel analysis based Trojan detection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 130–141.
[19] Y. Lyu and P. Mishra, "MaxSense: Side-channel Sensitivity Maximization for Trojan Detection Using Statistical Test Patterns," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 26, no. 3, pp. 1–21, 2021.
[20] D. Rai and J. Lach, "Performance of delay-based Trojan detection techniques under parameter variations," in *2009 IEEE HOST*. IEEE, 2009, pp. 58–65.
[21] V. Gohil, "DETERRENT," https://github.com/gohil-vasudev/DETERRENT, 2022.
[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.
[23] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.
[24] OpenCores, "Educational 16-bit MIPS Processor," https://opencores.org/projects/mips_16, 2013, Last accessed the website on 03/14/2023.
[25] "Amazon EC2 On-Demand Pricing," https://aws.amazon.com/ec2/pricing/on-demand/, 2023, [last accessed on 06/04/2023].
[26] Timothy Trippel, "AES," https://github.com/timothytrippel/bomberman/tree/master/third_party/aes_128, 2022, [last accessed on 06/04/2023].
[27] MIT Lincoln Laboratory, "GPS code generator," https://github.com/CommonEvaluationPlatform/CEP/tree/master/generators/mitll-blocks/src/main/resources/vsrc/gps, 2022, [last accessed on 06/04/2023].
[28] Stafford Horne, "mor1kx - an OpenRISC processor IP core," https://github.com/openrisc/mor1kx, 2022, [last accessed on 06/04/2023].
[29] The Washington Post, "Three months, 700 steps: Why it takes so long to produce a computer chip," https://www.washingtonpost.com/technology/2021/07/07/making-semiconductors-is-hard/, 2021, [last accessed on 06/04/2023].
[30] V. Gohil, S. Patnaik, H. Guo, D. Kalathil, and J. Rajendran, "DETERRENT: Detecting Trojans using Reinforcement Learning," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 697–702.

Vasudev Gohil (Student Member, IEEE) is pursuing a doctorate in Computer Engineering at Texas A&M University, College Station, TX, USA. His research interests lie at the intersection of machine learning and hardware security. He is also interested in the game-theoretic aspect of IP protection techniques. He has developed efficient and effective reinforcement learning solutions to improve the security of embedded systems, such as detecting hardware Trojans and identifying fault injection vulnerabilities.

Satwik Patnaik is an Assistant Professor in the Department of Electrical and Computer Engineering at the University of Delaware. His research focuses on developing security-focused computer-aided design tools to realize secure and trustworthy hardware and systems. In addition, his research leverages the 3D paradigm for computer security, exploits security properties of emerging post-CMOS devices, and utilizes machine learning and reinforcement learning techniques for different hardware security problems.

Hao Guo (S'21) received her B. Eng in Electrical Engineering and Automation from Southwest Jiaotong University, Chengdu, China, in 2017 and the M.Sc. in Electrical Engineering from the University of Southern California, Los Angeles, USA, in 2019. She joined the Texas A&M Secure and Trustworthy Hardware (SETH) Lab in the Spring of 2020. Her research focuses on FPGA security and applied machine learning for hardware security.

Dileep Kalathil (Senior Member, IEEE) is an Assistant Professor in the Department of Electrical and Computer Engineering at Texas A&M University (TAMU). His main research area is reinforcement learning theory and algorithms, and their applications in communication networks and power systems. Before joining TAMU, he was a postdoctoral researcher in the EECS department at UC Berkeley. He received his Ph.D. from University of Southern California (USC) in 2014, where he won the best Ph.D. Dissertation Prize in the Department of Electrical Engineering. He received the NSF CRII Award in 2019 and the NSF CAREER award in 2021.

Jeyavijayan (JV) Rajendran (Senior Member, IEEE) is an Assistant Professor in the Department of Electrical and Computer Engineering at the Texas A&M University. Previously, he was an Assistant Professor at UT Dallas between 2015 and 2017. His research interests include hardware security and computer security. His research has won the NSF CAREER Award in 2017, the ACM SIGDA Outstanding Young Faculty Award in 2019, the ACM SIGDA Outstanding Ph.D. Dissertation Award in 2017, and the Alexander Hessel Award for the Best Ph.D. Dissertation in the Electrical and Computer Engineering Department at NYU in 2016, along with several best student paper awards.