The Journal of Supercomputing (2024) 80:14295–14329
https://doi.org/10.1007/s11227-024-05963-8

Trojan playground: a reinforcement learning framework for hardware Trojan insertion and detection

Amin Sarihi[1] · Ahmad Patooghy[2] · Peter Jamieson[3] · Abdel-Hameed A. Badawy[1]

Accepted: 3 February 2024 / Published online: 18 March 2024

Abstract

Current hardware Trojan (HT) detection techniques are mostly developed based on a limited set of HT benchmarks. Existing HT benchmark circuits are generated with multiple shortcomings, i.e., (i) they are heavily biased by the designers' mindset when created, and (ii) they are created through a one-dimensional lens, mainly the signal activity of nets. We introduce the first automated reinforcement learning (RL) HT insertion and detection framework to address these shortcomings. In the HT insertion phase, an RL agent explores the circuits and finds locations best for keeping inserted HTs hidden. On the defense side, we introduce a multi-criteria RL-based HT detector that generates test vectors to discover the existence of HTs. Using the proposed framework, one can explore the HT insertion and detection design spaces to break the limitations of human mindset and benchmark issues, ultimately leading toward the next generation of innovative detectors. We demonstrate the efficacy of our framework on ISCAS-85 benchmarks, provide the attack and detection success rates, and define a methodology for comparing our techniques.

Keywords Hardware Trojan · Hardware security · Reinforcement learning

✉ Abdel-Hameed A. Badawy
badawy@nmsu.edu

Amin Sarihi
sarihi@nmsu.edu

Ahmad Patooghy
apatooghy@ncat.edu

Peter Jamieson
jamiespa@miamioh.edu

[1] Klipsch School of Electrical and Computer Engineering, New Mexico State University, Las Cruces, NM, USA

[2] Department of Computer Systems Technology, North Carolina A&T State University, Greensboro, NC, USA

[3] Electrical and Computer Engineering Department, Miami University, Miami, OH, USA

1 Introduction

As per a DoD report [1] released in 2022, 88% of the production and 98% of the assembly, packaging, and testing of microelectronic chips are performed outside of the USA. The growing multi-party production model has significantly raised security concerns about malicious modifications in the design and fabrication of chips, i.e., hardware Trojan (HT) insertion. *HTs* are defined as any design or manufacturing violations in an integrated circuit (IC) concerning the intent of the IC. Upon activation, an HT may lead to erroneous outputs (e.g. , Fig. 1) and possibly leak of information [2]. According to the adversarial model introduced by Shakya et al. [3], HTs can be inserted into target ICs according to the following scenarios:

- Design source code or netlist can be infected with HTs by compromised employees.
- Third-party intellectual properties (IPs) like processing cores, memory modules, I/O components, and network-on-chip [4] are often purchased and incorporated into a design to speed up time-to-market and lower design expenses. However, integrating IPs from untrusted vendors can pose a risk to the security and integrity of the IC.
- An untrusted foundry may reverse-engineer the GDSII physical layout to obtain the netlist and insert HTs inside them.
- Malicious third-party CAD tools may also insert HTs into designs

Researchers have been mostly using established benchmarks reported by Shakya et al. and Salmani et al. [3, 5] as a reference to study the impact of HTs.[1] Subsequently, various HT detection approaches have been developed based on these benchmarks over the past decade [7–10]. Despite the valuable effort to create HT benchmarks for the community, these benchmarks are limited in size and variety needed to push detection tools into more realistic modern scenarios. For instance, the small set of benchmarks makes it hard to leverage and train machine learning (ML) HT detectors, where more training data negatively impacts classification accuracy. Some research studies have tried to alleviate this problem by using techniques to shuffle data for ML-based detectors, e.g. , the leave-one-out cross-validation method [8]; however, it does not solve the problem entirely. The existing HT benchmarks also suffer from an inherent human bias in the insertion phase since they are tightly coupled with the designer's mindset. For instance, the HT benchmarks in [11] only consider signal activity for HT insertion, i.e., HTs are randomly inserted into a pool of available rare nets of the circuit. The flaws in the insertion phase simplify the problem's complexity, leading security researchers to develop HT detectors finely tuned to flawed scenarios [10, 12]. In contrast, adversaries devise new HT attacks that combine different ideas where detectors fall short of exposing them. Another equally important problem in this domain is having almost no HT detectors publicly available. This deprives other researchers of accessing these tools and imposes a considerable latency for newcomers to hardware security.

[1] The benchmarks are available on Trust-Hub [6].

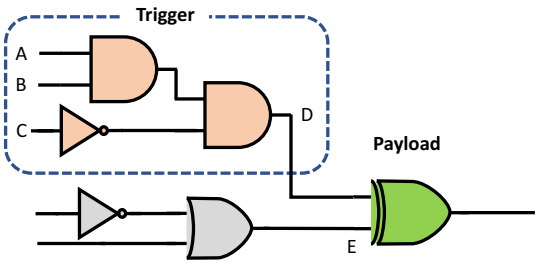


Fig. 1 An HT with a trigger and payload. Whenever $A = 1$, $B = 1$, and $C = 0$, the trigger is activated ($D = 1$) and the XOR payload inverts the value of E

This work attempts to move this research space forward by developing next-generation HT insertion and detection methods based on reinforcement learning. The developed RL-based HT insertion tool creates new HT benchmarks according to the criteria passed to the tool by the user. The insertion criteria is an RL rewarding function modified by a user that relies on the RL agent to insert HTs into designs automatically. The netlist is considered an environment in which the RL agent tries to insert HTs to maximize a gained reward. The rewarding scheme of the proposed insertion tool is tunable, which can push the agent toward a specific goal in the training session. Our insertion tool is a step toward preparing the community for future HTs inserted by non-human agents, e.g. , AI agents. We also propose an RL-based HT detector with a tunable rewarding function that helps detect inserted HTs based on various strategies. To explore this space, we have studied three different detection rewarding functions for the RL detector agent. The agent finds test vectors yielding the highest rewards per each reward function. Then, the generated test vectors activate and find HTs in the IC. The test engineer passes the test vectors to the chip and monitors the output for deviations from the golden model.

Our proposed toolset enables the researchers to experience HT insertion and detection within a unified framework. The framework only requires users to set the parameters to insert and detect HTs without human intervention. There have been previous efforts to automate the HT insertion and detection process [11, 13, 14]; however, they need an intermediate effort hindering us from creating a vast quantity of HTs (more explanation in Sect. 2).

Similar to several previous works [2, 10, 15, 16], this paper's threat model assumes that the perpetrator is capable of inserting HTs into a design's netlist. The netlist can be obtained through state-of-the-art reverse-engineering techniques in the foundry, and HT triggers are constructed and placed in the design layout. On the defense side, we assume a security engineer receives a post-silicon hard IP that may or may not contain malicious HTs. Using a golden model, the security engineer generates a set of minimal test vectors to activate as many HTs as possible. The test engineer does not know the insertion criteria; however, they generate test vectors based on multiple insertion mentalities. If the output(s) of the design-under-test deviate(s) from the golden model, it can insinuate malicious behavior.

We make the following contributions in the paper with respect to our previous publications [18, 19]:

- We developed a tunable RL-based HT insertion tool free of human bias, capable of automatic HT insertion and creating a large population of valid HTs for each design
- We introduce a tunable RL-based multi-criteria HT detection tool that helps a security engineer to better prepare for different HT insertion strategies.
- We introduce and use a generic methodology to compare HT detectors fairly. The methodology is based on the confidence value metric that helps the security engineer select the proper detector based on the chip's application and security requirements.

Our results show that our developed detection tool with all three detection approaches has an average 90.54% detection rate for our HT-inserted benchmarks. We compare these detection results to existing state-of-the-art detection methods and show how our techniques find previously unidentifiable HTs. As we believe that HT detection will be implemented as a variety of detection strategies, the uniquely identified HTs suggest that our detection techniques and framework are important contributions to this space.

The remainder of this paper is organized as follows: Sect. 2 reviews the related work and explains the fundamentals of RL. The mechanics of our proposed HT insertion and detection approaches are presented in Sects. 3 and 4, respectively. We introduce our HT comparison methodology in Sect. 5. Section 6 demonstrates the experimental results, and Sect. 7 concludes the paper.

2 Related work

This section summarizes the previous studies in HT insertion and detection.

2.1 Hardware Trojan insertion and benchmarks

The first attempts to gather benchmarks with hard-to-activate HTs were made by Shakya et al. and Salmani et al. [3, 5]. A set of 96 trust benchmarks with different HT sizes and configurations are available at Trust-Hub [6]. While these benchmarks are a valuable contribution to the research community, they have three drawbacks:

1. The limited number of Trojan circuits represents only a subset of the possible HT insertion landscape in digital circuits, which hampers the ability to develop diverse HT countermeasures,
2. They lack incorporating state-of-the-art Trojan attacks, and
3. They fail to populate a large enough HT dataset required for ML-based HT detection.

Krieg [20] investigates the practicality of the Trusthub benchmark for hardware security study from five different perspectives: Correctness, Maliciousness, Stealthiness, Persistence, and Effectiveness. The paper lists nine main flaws that undermine the feasibility of Trusthub for security evaluations, including pre-/post-synthesis simulation mismatch, unsatisfiable trigger conditions, incorrect original designs,

and buggy wiring. The paper shows that out of the 83 benchmarks, only three hold all the properties, and the rest fail in at least one or more studied aspects.

Various approaches to insert HTs have been attempted. Jyothi et al. [21] proposed a tool called TAINT for automated HT insertion into FPGAs at the RTL-level, gate-level netlist, and post-map netlist. The tool also allows the user to insert HTs in FPGA resources such as look-up tables (LUTs), flip-flops (FFs), block random access memory (BRAM), and digital signal processors (DSP). Despite the claimed automated process, the user is expected to select the trigger nets based on suggestions made by the tool. The results section shows that the number of available nodes in post-map netlists drops significantly, leaving less flexibility for Trojan insertion compared to RTL codes.

Reverse-engineering tools can also identify security-critical circuitry in designs that can direct attackers to insert efficient HTs. Fyrbiak et al. [13] introduced HAL, a gate-level netlist reverse-engineering tool that offers offensive reverse-engineering strategies and defensive measures, such as developing arbitrary Trojan detection techniques. The authors believe that adversaries are more likely to insert HTs through reverse-engineering techniques and are less likely to have direct access to the original HDL codes. A hardware Trojan that leaks cryptographic keys has been inserted with the tool; nonetheless, it requires human effort for insertion, which hinders the production of a large HT dataset [22]. Further endeavors have been made to follow a threat model in which an adversary is located in a foundry with sophisticated reverse-engineering capabilities. Perez et al. [23] targets SCTs (side-channel Trojans), more commonly found in crypto cores. The authors showcase a flow to insert HTs to leak confidential information based on power signatures. During this process, an adversary takes advantage of ECO (engineering change order), a flow originally used to fix bugs in finalized layouts. The work in [24] builds upon the previous study by manufacturing an ASIC prototype with four HT-infected versions of AES and PRESENT. Puschner et al. [25] propose a de-coupled insertion and detection flow where the red team is responsible for inserting ECO-based HTs in design layouts, and the blue team must find the malicious embedding by investigating SEM (Scanning Electron Microscope) images vs GDSII (Graphic Design System II) files. The study shows that the ECO-inserted HTs are less challenging to find. Hepp et al. [26] use the ECO flow to insert HTs in the design layout without prior knowledge of its functionality. The study explores three new criteria for selecting the HT payload and triggers: transition probability, imprecise information flow tracking of selected signals, and the RELIC score. The RELIC score is a metric that provides an attacker with information about the location of a flip-flop relative to the data path or the control path. The authors operate under the assumption of a 24-h time window for the attacker to complete the insertion process.

Cruz et al. [11] tried to address the benchmark shortcomings by presenting a toolset capable of inserting a variety of HTs based on the parameters passed to the toolset. Their software inserts HTs with the following configuration parameters: the number of trigger nets, the number of rare nets among the trigger nodes, a rare net threshold (computed with functional simulation), the number of the HT instances to be inserted, the HT effect, the activation method, its type, and the choice of payload. Despite increasing the variety of inserted HTs, there is no solution for finding the optimal trigger and payload nets. The TRIT benchmark set generated by this tool is available on Trust-Hub [6].

Cruz et al. [22] propose MIMIC, an ML framework for automatically generating Trojan benchmarks. The authors extracted 16 functional and structural features from existing Trojan samples. Then, they trained ML models and generated a large number of hypothetical Trojans called *virtual Trojans* for a given design. The virtual Trojans are then compared to a reference Trojan model and ranked. Finally, the selected Trojan will be inserted into the target circuit using suitable trigger and payload nets. The HT insertion process is highly convoluted, requiring multiple stages and expertise. MIMIC is not released publicly, and rebuilding the tool from their work is an extensive process. MIMIC's HT insertion criteria are very similar to [11], and it suffers the same shortcomings [11].

To deceive machine learning HT detection approaches, Nozawa et al. [27] have devised adversarial examples. Their proposed method replaces the HT instance with its logically equivalent circuit, so the classification algorithm erroneously disregards it. To design the best adversarial example, the authors have defined two parameters: Trojan-net concealment degree (TCD), which is tuned to maximize the loss function of the neural network in the detection process, and a modification evaluating value (MEV) that should be minimized to have the least impact on circuits. These two metrics help the attacker to look for more effective logical equivalents and diversify HTs. The equivalent HTs are inserted in trust-hub benchmarks, and they decrease accuracy significantly.

Sarihi et al. [18] (our prior work) inserted a large number of HTs into ISCAS-85 benchmarks with Reinforcement Learning. The HT circuit is an agent that interacts with the environment (the circuit) by taking five different actions (next level, previous level, same level up, same level down, no action) for each trigger input. Level denotes the logic level in the combinational circuits. The agent moves the Trojan inputs throughout the circuit and explores various locations suitable for embedding HTs. Triggers are selected according to a set of SCOAP (Sandia Controllability/Observability Analysis Program [28]) parameters, i.e., a combination of controllability and observability. The agent is rewarded in proportion to the number of circuit inputs it can engage in the HT activation process.

Gohil et al. [16] proposed ATTRITION, another RL-based HT insertion platform where signal probability is the target upon which the trigger nets are selected. The agent tries to find a set of so-called *compatible* rare nets, i.e., a group of rare nets that can be activated together with an input test vector. The test vector is generated using an SAT-solver. The authors also propose a pruning technique to limit the search space for the agent to produce more HTs in a shorter period. The tool is claimed to be open-source, but only the source code was released.

Table 1 summarizes the existing artifacts and research in the HT insertion space. It represents the target technology (2nd column); summarizes the insertion criteria (3rd column); shows if the tool is automated (4th column) and if the tool or its artifacts are openly released (5th column).

2.2 Hardware Trojan detection

Chakraborty et al. [15] introduced MERO, a test vector generator that tries to trigger possible HTs by exciting rare-active nets multiple times. The algorithm's efficacy

Table 1 Survey of previous HT insertion tools

Tool	Domain	Insertion criteria	Automate	Open-source
Trust-Hub [3]	ASIC/FPGA	Secret leakage, signal prob	✗	✗
HAL [13]	ASIC/FPGA	Neighborhood control value	✗	✓
TAINT [21]	FPGA	Not mentioned	✗	✗
TRIT [11]	ASIC	Signal prob	✗	✗
Yu et al. [14]	ASIC	Transition prob	✓	✗
Nozawa et al. [27]	ASIC	Same as [3]	✗	✗
MIMIC [22]	ASIC	Struct & Funct. Features	✓	✗
Sarihi et al. [18]	ASIC	SCOAP parameters	✓	✗
ATTRITION [16]	ASIC	Signal prob	✓	✗
Perez et al. [23, 24]	ASIC	Power leakage	✗	✗
Puschner et al. [25]	ASIC	No restrictions	✓	✓
BioHT [26]	ASIC	Multiple criteria	✓	✗

is tested against randomly generated HTs with rare triggers. MERO's detection rate significantly shrinks as circuit size grows.

Hasegawa et al. [8] have proposed an ML method for HT detection. The method extracts 51 circuit features from the trust-hub benchmarks to train a random forest classifier that eventually decides whether a design is HT-free. The HT classifier is trained on a limited HT dataset with an inherent bias during its insertion phase.

Lyu et al. [12] proposed TARMAC to map the trigger activation problem to the clique cover problem, i.e., treating the netlist as a graph. They utilized an SAT-solver to generate the test vector for each maximal satisfiable clique. The method lacks scalability as it should run on each suspect circuit separately. Also, the achieved performance is not stable [2]. Implementation of the method is neither trivial nor available publicly to researchers [16].

TGRL is an RL framework used to detect HTs [2]. The agent decides to flip a bit in the test vector according to an observed probability distribution. The reward function, which combines the number of activated nets and their SCOAP [28] parameters, pushes the agent to activate as many signals as possible. Despite its higher HT detection rate than MERO and TARMAC, the algorithm was not tested on any HT benchmarks [16].

DETERRENT, an RL-based detection method [10], finds the smallest set of test vectors to activate multiple combinations of trigger nets. The RL state is a subset of all possible rare nets, and actions are appending other rare nets to this subset. The authors used an SAT-solver to determine if actions are compatible with the rare nets in the subsets, and they only focused on signal-switching activities as their target.

The HW2VEC tool [29] converts RTL-level and gate-level designs into a dataflow graph and abstract syntax tree to extract a feature set that represents the structural information of the design. Extracted features are used to train a graph neural network to determine whether a design is infected with HTs. The authors test the tool with 34 circuits infected by in-house generated HTs.

Table 2 Survey of previous HT detection tools

Study	Detection basis	Open-source
MERO [15]	Switching activity	✗
Hasegawa et al. [8]	Netlist features	✗
TARMAC et al. [12]	Switching activity	✗
TGRL et al. [2]	Switching activity	✗
DETERRENT et al. [10]	Switching activity	✗
HW2VEC [29]	Graph structural info	✓

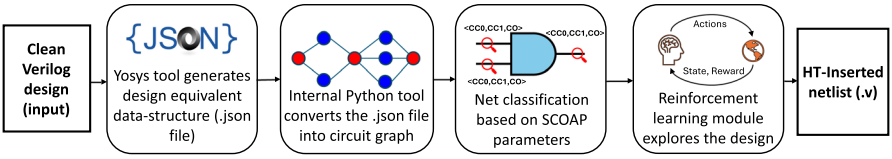


Fig. 2 The proposed RL-based HT insertion tool flow

We note that of the methods reviewed above (and others studied but not discussed here), the only publicly available tool is HW2VEC. Table 2 summarizes the previous works in HT detection where researchers have used various criteria in detecting HTs (2nd column) and the open-source state of the work (3rd column).

3 The proposed HT insertion

Figure 2 shows the flow of the proposed HT insertion tool. The first step creates a graph representation of the flattened netlist from the circuit. Yosys Open Synthesis Suite [30] translates the HDL (Verilog) source of the circuit into a JSON (JavaScript Object Notation) [31] netlist, which enables us to parse the internal graph representation of the circuit. Next, the tool finds a set of rare nets to be used as HT trigger nets (this step is described in detail in Sect. 3.1). Finally, an RL agent uses the rare net information and attempts to insert an HT to maximize a rewarding function as described in Sect. 3.2.

3.1 Rare netS extraction

We use the parameters introduced in [9] to identify trigger nets. These parameters are defined as functions of net *controllability* and *observability*. Controllability measures the difficulty of setting a particular net in a design to either '*0*' or '*1*'. Conversely, observability is the difficulty of propagating a net value to at least one of the circuit's primary outputs [28].

The first parameter is called the HT trigger susceptibility parameter, and it is derived from the fact that low-switching nets have mainly a high difference between their controllability values. Equation (1) describes this parameter:

$$HTS(Net_i) = \frac{|CC1(Net_i) - CC0(Net_i)|}{Max(CC1(Net_i), CC0(Net_i))} \quad (1)$$

where *HTS* is the HT trigger susceptibility parameter of the net; $CC0(Net_i)$ and $CC1(Net_i)$ are the combinational controllability 0 and 1 of Net_i, respectively. The *HTS* parameter ranges between [0, 1) such that higher values correlate with lower activity on the net.

The other parameter, specified in Eq. (2), measures the ratio of observability to controllability:

$$OCR(Net_i) = \frac{CO(Net_i)}{CC1(Net_i) + CC0(Net_i)} \quad (2)$$

where *OCR* is the observability to controllability ratio. This equation requires that the HT trigger nets be hard to control but not so hard to observe. Unlike the *HTS* parameter, *OCR* is not bounded and belongs to the $[0, \infty)$ interval. We will specify thresholds (see Sect. 6) for each parameter and use them as filters to populate the set of rarely-activated nets for our tool.

3.2 RL-Based HT insertion

The RL environment is, in fact, the circuit in which the agent is trying to insert HTs. The agent's action is to insert combinational HTs where trigger nets are ANDED, and the payload is an XOR gate (same as Fig. 1). The RL agent starts from a reset condition, taking a series of actions that eventually insert HTs in the circuit. Different HT insertion options are represented with a state vector in each circuit. For a given HT, the state vector is comprised of $s_t = [s_1, s_2, \ldots, s_{n-2}, s_{n-1}, s_n]$ where s_1 through s_{n-2} are the logic levels of the HT inputs, and s_{n-1} and s_n are the logic levels of the target net and the output of the XOR payload, respectively. Figure 3 shows how we conduct the circuit levelization. Here, the circuit primary inputs (PIs) are considered level 0. The output level of each gate is computed by Eq. (3):

$$Level(output) = MAX(Level(in_1), Level(in_2)) + 1 \quad (3)$$

As an example, the HT in Fig. 4 (in yellow) has the state vector $s_t = [2, 1, 3, 4]$. The action space of the described HT agent is multi-discrete, i.e., each input of the HT may choose an action from a set of five available actions. These actions are:

- *Next level*: the input of the HT moves to one of the nets that are one level higher than the current net level.
- *Previous level*: the input of the HT moves to one of the nets that are one level lower than the current net level.
- *Same level up*: the input of the HT will move to one of the nets at the same level as the current net level. The net is picked by pointing to the next net in the ascending list of net IDs for the given level.

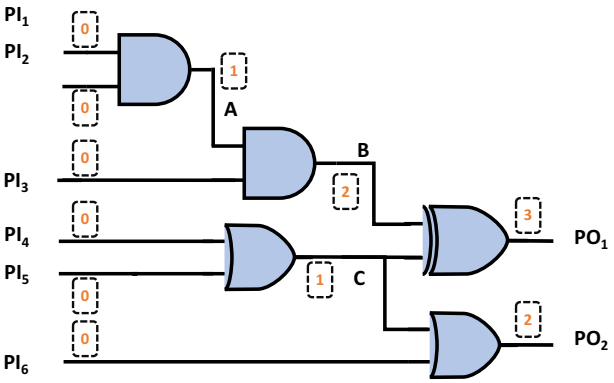


Fig. 3 Levelizing a circuit. The output level of each digital gate is computed by $max(Level(in1), Level(in2)) + 1$

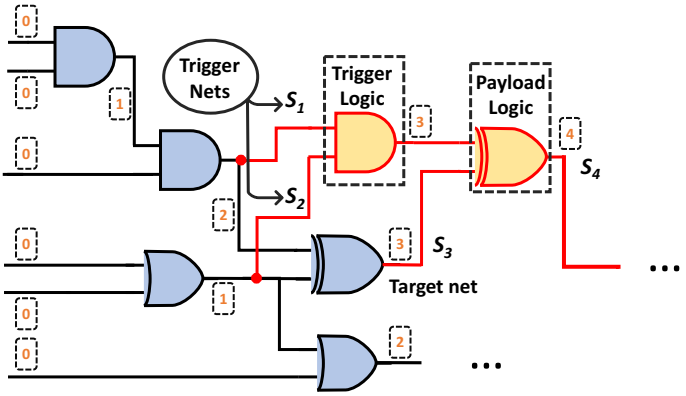


Fig. 4 Obtaining the state vector in the presence of an HT in the circuit

- *Same level down*: the input of the HT will move to one of the nets at the same level as the current net level. The net is picked by pointing to the previous net in the ascending list of nets for the given level.
- *No action*: the input of the HT will not move. If an action leads the agent to step outside the circuit boundaries, it is substituted with a "No action".

The action space is also represented by a vector where its size is equal to the number of the HT inputs, and each action can be one of the five actions above, e.g. , for the HT in Fig. 4, the action space would be $a_t = [a_1, a_2]$ since it has two inputs. Hypothetical actions for the first and the second inputs can be the same level up/down and next/previous level, respectively.

The flow of our RL inserting agent is described in Algorithm 1. The SCOAP parameters are first computed (line 1). We specify two thresholds T_{HTS} and T_{OCR} and require our algorithm to find nets that have higher *HTS* values than T_{HTS} and

lower *OCR* values than T_{OCR} (line 2). These nets are classified as rare nets. The algorithm consists of two nested while loops that keep track of the terminal states and the elapsed timesteps. The latter defines the total number of samples the agent trains on. We have used the OpenAI Gym [32] environment to implement our RL agent.

Algorithm 1 Training of the HT inserting Reinforcement Learning Agent

Input: Graph G, HTS Threshold T_{HTS}, OCR Threshold T_{OCR}, Circuit Inputs in_ports, State Space s_t, Terminal State $Terminal_{state}$, Total Timesteps j;
Output: HT Benchmark $HT_{Benchmark}$;

1: Compute SCOAP parameters: $< CC0, CC1, CO >= computeSCOAP(G)$;
2: Get the set of rare nets: $rare\_nets = Compute\_Rare\_Nets(G, T_{HTS}, T_{OCR})$;
3: $counter = 0$;
4: **while** $(counter < j)$ **do**
5: $HT = reset\_environment()$;
6: $Terminal_{state} = false$;
7: **while** $!(Terminal_{state})$ **do**
8: $G, s_t, Terminal_{state}, HT_{triggers} = action(HT)$;
9: $HT\_activated = PODEM(G)$;
10: $temp_{reward} = (HT_{triggers} \cap rare\_nets).count()$;
11: **if** $(HT\_activated)$ **then**
12: **if** $(temp_{reward} == 1)$ **then**
13: $reward = 8$;
14: **else if** $(temp_{reward} == 2)$ **then**
15: $reward = 16$;
16: **else if** $(temp_{reward} == 3)$ **then**
17: $reward = 100$;
18: **else if** $(temp_{reward} == 4)$ **then**
19: $reward = 1000$;
20: **else if** $(temp_{reward} == 5)$ **then**
21: $reward = 10000$;
22: **else**
23: $reward = -1$;
24: **end if**
25: **end if**
26: $update\_PPO(action, s_t, reward)$;
27: $counter+ = 1$;
28: **end while**
29: **end while**
30: $HT_{Benchmark} = Graph\_to\_netlist(G)$

The first used method is called *reset_environment*(), which resets the environment before each episode and returns the initial location of the agent HT (line 5). The HT is randomly inserted within the circuit according to the following rules.

- Rule (1) Trigger nets are selected randomly from the list of the total nets.
- Rule (2) Each net can drive a maximum of one trigger net.
- Rule (3) Trigger nets cannot be assigned as the target.
- Rule (4) The target net is selected with respect to the level of trigger nets. To prevent forming combinational loops, we specify that the level of the target net should be greater than that of the trigger nets.

In each episode of the training process, we keep the target net unchanged to help the RL algorithm converge faster. Instead of manually specifying a target net, we let the algorithm explore the environment and choose a target net. The terminal state variable *TS* is set to *False* to check the termination condition for each episode. When the trigger nets' level reaches the target net's level, or the number of steps per episode reaches an allowed maximum (lines 6–7), *TS* becomes *True*, which terminates the episode.

The training process of the agent takes place in a loop where actions are being issued, rewards are collected, the state is updated, and eventually, the updated graph is returned. To test the value of an action taken by the RL agent (meaning if the HT can be triggered with at least one input pattern), we use *PODEM* (Path-Oriented Decision Making), an automatic test pattern generator [33] (line 9). This algorithm uses a series of backtracing and forward implications to find a vector that activates the inserted HT. If the HT payload propagates through at least one of the circuit outputs, the action gains a reward proportional to the number of rare triggers on the HT. After the number of rare triggers is counted in line 10, the agent is rewarded in lines 11 through 25. The rewarding scheme is designed such that the agent would start finding HTs with a 1 rare trigger net and adding more rare nets while exploring the environment. Additionally, the exponential reward increase in each case ensures that the agent is highly encouraged to find HTs with at least three or more rare trigger nets. If an HT is not activated with *PODEM* or no rare nets are among the HT triggers, the agent will be rewarded -1. Since the agent is unlikely to find high-reward HTs at the beginning of the exploration stage, the first two rewarding cases ($temp_{reward} = 1$ and $temp_{reward} = 2$) should be set such that the agent sees enough positive, rewarding improvements, yet be more eager to find more HTs that yield higher rewards. After extensive experiments with the RL agent, the reward values are assigned to different cases.

We use the *PPO* (proximal policy optimization) [17] RL algorithm to train the RL agent. PPO can train agents with multi-discrete action spaces in discrete or continuous spaces. The main idea of PPO is that the new updated policy (which is a set of actions to reach the goal) should not deviate too far from the old policy following an update in the algorithm. To avoid substantial updates, the algorithm uses a technique called clipping in the objective function [17]. Using a clipped objective function, PPO restricts the size of policy updates to prevent them from deviating too much from the previous policy. This constraint promotes stability and ensures that the updates are controlled within a specific range, which helps avoid abrupt changes that may negatively affect the agent's performance. At last, when the HTs are inserted, the toolset outputs Verilog gate-level netlist files that contain the malicious HTs (line 30).

4 The proposed HT detection

From a detection perspective, we must determine whether a given circuit is clean or Trojan-infected. To achieve this goal, an RL agent is defined that applies its generated test vectors to circuits and checks for any deviation at the circuits' primary outputs with respect to the expected outputs (golden model). The agent interacts with the circuit (performs actions) by flipping the vector values to activate certain internal nets. The action space is an n-dimensional binary array where n is the number of circuit primary inputs. The action space vector a_t is defined as $a_t = [a_1, a_2, \dots, a_n]$. The agent decides to toggle each a_i to transition to another state or leave them unchanged. $a_i = 0$ denotes that the value of the ith bit of the input vector should remain unchanged from the previous test vector. In contrast, $a_i = 1$ means that the ith input bit should flip. The RL agent follows a π policy to decide which actions should be commenced at each state. The π policy is updated using a policy gradient method [34] where the agent commences actions based on probability distribution from the π policy. The assumption is that attackers are likely to choose trigger nets with a consistent value (0 or 1) most of the time. Thus, a detector aims to activate as many dormant nets as possible. We consider two different approaches for identifying such rare nets:

(1) *Dynamic simulation*: We feed each circuit with 100K random test vectors and record the value of each net. Then, we populate the switching activity statistics during the simulation time and set a threshold θ for rare nets where the switching activity for a net below θ denotes that the net is rare. θ is in the range of [0, 1].
(2) *Static simulation*: We use the *HTS* parameter in Eq. (1) and a threshold to find rare nets. Categorizing rare nets with this approach provides the security engineer with an extra option for detection.

In a circuit with m rare nets, the state space is defined as $State_t = [s_1, s_2, \dots, s_m]$ where s_i is associated with the ith net in the set. If an action (a test vector) sets the ith net to its rare value, s_i will be 1; otherwise, s_i stays at 0. As can be inferred, the action and state spaces are multi-binary.

Attackers tend to design multi-trigger HTs [11], and this should be considered when HT detectors are designed. The final purpose of our detector is to generate a set of test vectors that can trigger as many rare nets as possible. To achieve this goal, a part of the rewarding function should enumerate rare nets. However, we should avoid over-counting situations where a rare net has successive dependent rare nets. An example case is shown in Fig. 5 where four nets net_1, net_2, net_3, and net_4 (with their switching probabilities and their rare values) are all dependent rare nets. Instead of including all four nets in the state space, we choose the rarest net as the representative net since activating the rarest net ensures the activation of the others as well. In this example, net_4 is selected as the set representative. This policy helps accelerate the RL agent to converge on the global minima faster. Figure 6 summarizes our proposed detection flow.

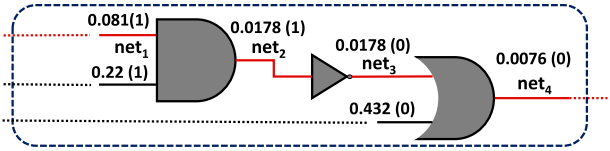


Fig. 5 State pruning identifies nets in the same activation path

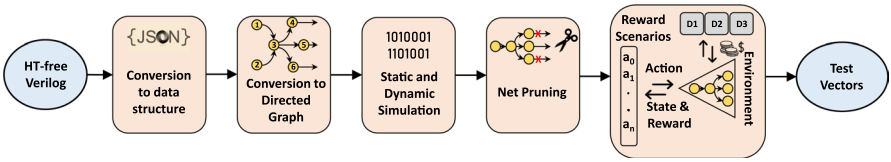


Fig. 6 The proposed detection flow

As for rewarding the agent, we consider three rewarding functions, which we explain here. Our multi-rewarding detector enables security engineers to better prepare for attackers with different mindsets.

4.1 Rewarding function SSD

In our first rewarding function (Algorithm 2) called SSD (Subsequent State Detector), we push the RL agent to build on its current state. We use a copy of the previous state and encourage the agent to generate state vectors that differ from the previous one. The hypothesis is to push the agent toward finding test vectors that lead to various unseen states. To compute the reward, the pruned current and previous state vectors and their lengths are passed as inputs to Algorithm 2. The rewarding function comprises an *immediate* and a *sequential* part, initialized to 0 in lines 1 and 2, respectively. Whenever the state transitions, we iterate through the loop K times. We calculate the sequential reward by making a one-to-one comparison between the nets in the old and new states. In lines 5–11, the highest reward is given when an action can trigger a net not triggered in the previous state, i.e., +40. If a rare net is still activated in the current state, the agent will still get rewarded +20. The worst state transition is whenever an action leads to a rare net losing its rare value, which is rewarded −3. Lastly, if the agent cannot activate a rare net after a state transition, it will be rewarded −1. This process is depicted in Fig. 7.

The immediate award is the number of activated rare nets in the new state. The ultimate reward value is a linear combination of the immediate and sequential rewards with coefficients λ_1 and λ_2, respectively, which are tunable parameters to be set by the user. We build the state vector with the obtained rare nets from functional simulation.

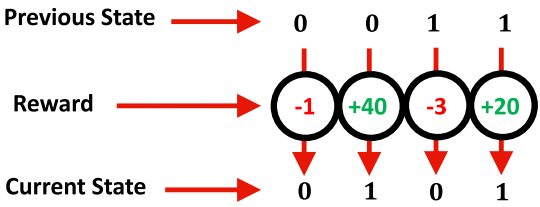


Fig. 7 Rare net transition (state transition) in the current and previous states and corresponding rewards

Algorithm 2 Rewarding Function SSD

Input: $State_{pre}$, $State_{cur}$, State Vector Length K
Output: $Reward_{final}$

1: $Reward_{Imd} = 0$;
2: $Reward_{Seq} = 0$;
3: **for** $k \in \{0, \ldots, K-1\}$ **do**
4: **if** ($State_{cur}[k] = 0$ and $State_{pre}[k] = 0$) **then**
5: $Reward_{Seq} += -1$;
6: **else if** ($State_{cur}[k] = 0$ and $State_{pre}[k] = 1$) **then**
7: $Reward_{Seq} += -3$;
8: **else if** ($State_{cur}[k] = 1$ and $State_{pre}[k] = 0$) **then**
9: $Reward_{Seq} += 40$;
10: **else if** ($State_{cur}[k] = 1$ and $State_{pre}[k] = 1$) **then**
11: $Reward_{Seq} += 20$;
12: **end if**
13: **end for**
14: $Reward_{Imd.} = State_{cur}.count(1)$
15: $Reward_{final} = \lambda_1 \times Reward_{Seq} + \lambda_2 \times Reward_{Imd}$

4.2 Rewarding function SAD

Algorithm 3 describes our second rewarding function called SAD (Switching Activity Detector). In this case, the agent gains rewards proportional to the difficulty of the rare nets triggered. First, the reward vector is initiated with a length equal to the state vector (line 1). Each element in the reward vector has a one-to-one correspondence with rare nets on the state vector. The reward for each rare net is computed by taking the inverse of the net switching activity rate (line 4). In some cases, a net might have a switching probability of 0. In such cases, activating the net would be rewarded 10× times the greatest reward in the vector (line 12). Thus, upon observing every new state, the agent will be rewarded based on the activated nets and the reward vector (line 18). If a rare net is not activated, −1 will be added to the final reward (line 20). The algorithm encourages the agent to trigger the rarest nets in the circuit directly.

Algorithm 3 Rewarding Function SAD

Input: Net switching vector $Switching_{vector}$,
Current state vector $State_{vector}$, State Vector Length K
Output: Final reward $Reward_{final}$

1: $Reward_{vector} = [0] * K$
2: **for** $k \in \{0, \ldots, K-1\}$ **do**
3: **if** $(Switching_{vector}[k]! = 0)$ **then**
4: $Reward_{vector}[k] = Switching_{vector}[k]^{-1}$
5: **else**
6: $Reward_{vector}[k] = 0$
7: **end if**
8: **end for**
9: $reward_{max} = max(Reward_{vector}[\,])$
10: **for** $k \in \{0, \ldots, K-1\}$ **do**
11: **if** $(Switching_{vector}[k] == 0)$ **then**
12: $Reward_{vector}[k] = 10 * reward_{max}$
13: **end if**
14: **end for**
15: $Reward_{final} = 0$
16: **for** $k \in \{0, \ldots, K-1\}$ **do**
17: **if** $(State_{vector}[k] == 1)$ **then**
18: $Reward_{final} += Reward_{vector}[k]$
19: **else**
20: $Reward_{final} += -1$
21: **end if**
22: **end for**

4.3 Rewarding function COD

The third rewarding function is described in Algorithm 4 and is called COD (Controllability Observability Detector). In this scenario, rare nets are populated based on the threshold of the *HTS* parameter computed during the static simulation using Eq. (1). When a rare net in the set is activated, the agent is rewarded with the controllability of the rare value (line 4). Otherwise, it will receive −1 from the environment (line 6). This scenario aims to investigate controllability-based HT detection with the RL agent. Figure 8 shows an example where an RL action is XORed with an old test vector, generating a new test vector. It also shows how activating rare nets (from SAD and COD) leads to state transitions where an activated net corresponds to a '1' in the state vector.

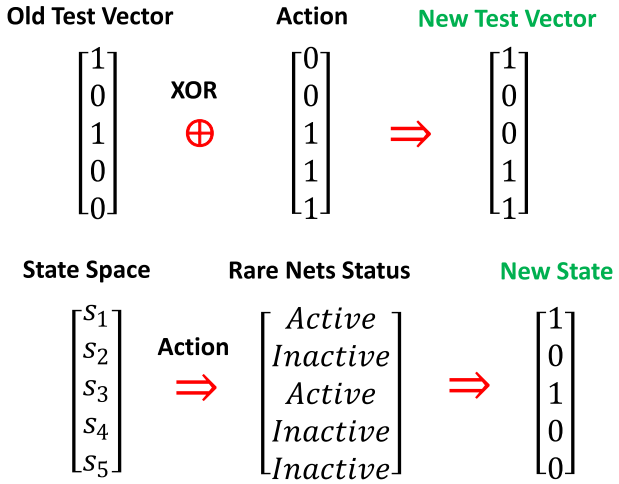


Fig. 8 Test vector generation and state transition for SAD and COD

Algorithm 4 Rewarding Function COD

Input: Controllability reward vector $Reward_{vector}$,
Current state vector $State_{vector}$, State Vector Length K,
Output: Final reward $Reward_{final}$

$Reward_{final} = 0$
for $k \in \{0, \ldots, K-1\}$ **do**
  if $State_{vector}[k] == 1$ **then**
    $Reward_{final} += Reward_{vector}[k]$
  else
    $Reward_{final} += -1$
  end if
end for

5 The proposed generic HT detection metric

We propose the following methodology to the community for fair and repeatable comparisons among HT detection methods. In addition, our methodology can help compare different HT insertion techniques for a given HT detector. This methodology obtains a confidence value that one can use to compare different HT detection methods.

Figure 9 shows four possible outcomes when an HT detection tool studies a given circuit. From the tool user's perspective, the outcomes are probabilistic events. For example, when an HT-free circuit is being tested, the detection tool may either

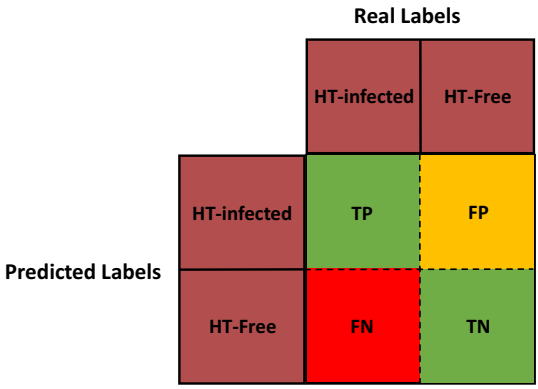


Fig. 9 Possible outcomes of an HT detection trial

classify it as an infected or a clean circuit, i.e., $Prob(FP) + Prob(TN) = 1$ where FP and TN stand for *False Positive* and *True Negative* events. Similarly, for HT-infected circuits, we have $Prob(FN) + Prob(TP) = 1$. FN and FP are two undesirable outcomes at which detectors misclassify the given circuit. However, the FN cases pose a significantly greater danger as they result in a scenario where we rely on an HT-infected chip. In contrast, an FP case means wasting a clean chip by either not selling or not using it. So, we need to know how the user of HT detection tools (might be a security engineer or a company representative) prioritizes FN and FP cases. We define a parameter α as the ratio of the undesirability of FN over FP. The tool user determines α based on characteristics and details of the application that eventually chips will be employed in, e.g. , the risks of using an infected chip in a device with a sensitive application versus using a chip for home appliances. Note that the user sets this value, which is not derived from the actual FP and FN. After α is set, it is plugged into Eq. (4) and a general confidence basis *Conf. Val* is computed.

$$Conf.\ Val = \frac{(1 - FP)}{(1/\alpha + FN)} \tag{4}$$

This metric can compare HT detection methods fairly regardless of their detection criteria and implementation methodology. The defined confidence metric combines the two undesirable cases to their severity from a security engineer's point of view. The *Conf. Val* ranges between $[\frac{0.5\alpha}{1+0.5\alpha}..\alpha]$. The closer the value is to α, the more confidence in the detector. The absolute minimum of the *Conf. Val* $= 1/3$ that happens when $\alpha = 1$ and $FP = FN = 50\%$. This analysis assumes that FN and FP are independent probabilities. We note that, for some detection methods, FP is always 0. For instance, test-based HT detection methods that apply a test vector to excite HTs use a golden model (HT-free) circuit for comparison and decision making, and it is impossible for a non-infected circuit to have a mismatch with the golden model (from the perspective of functional simulation). It is impossible for such methods to detect an HT in a clean circuit falsely. However, our metric is general and captures such cases.

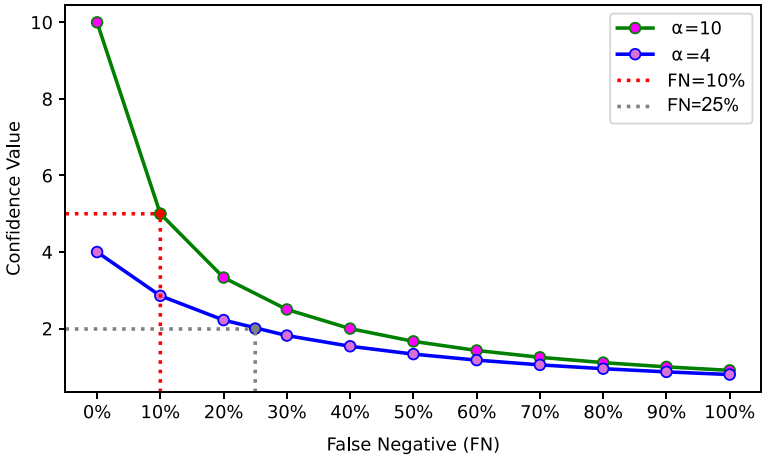


Fig. 10 Confidence value vs. the percentage of *FN* in our detectors assuming $\alpha = 10$ and $\alpha = 4$

Figure 10 shows the relation between the confidence value and the *FN* percentage for $\alpha = 10$ and $\alpha = 4$ for a test-based detector. As can be observed, the slopes of the graphs are different when *FN* approaches zero. The maximum tolerable *FN* is an upper bound for the *FN* value at which we gain at least half the maximum confidence. As shown with the dashed lines in Fig. 10, the maximum tolerable *FN* for $\alpha = 4$ and $\alpha = 10$ is, respectively, $FN = 25\%$ and $FN = 10\%$. Based on the figure, it can be inferred that choosing a higher base α will make it more challenging to attain higher confidence values. This fact should be considered when selecting α and interpreting the confidence values.

In addition to the detection quality, which the proposed confidence value can measure, HT detection methods should also be compared from a computational cost point of view. In particular, we encourage researchers to report the runtime of their methods and the training time, if applicable.

6 Experimental results and discussion

This section demonstrates the efficiency of the developed HT insertion and detection framework. For our experiments, we use an AMD EPYC 7702P 64-Core CPU with 512 GB of RAM to train and test our agents. The training of the RL agents is done using the Stable Baselines library [35] with MLP (multi-layer perceptron) as the PPO algorithm policy [17]. The benchmark circuits are selected from ISCAS-85 [36] and converted into equivalent circuit graphs using NetworkX [37]. Our HT benchmarks and test vectors are available to download from [38]. The HTs are in structural Verilog format, making them easy to use. The input orders of the test vectors are the same as [39]. Our toolset is developed in Python to (1) quickly adopt available libraries and (2) facilitate future expansions and integration with other tools that researchers may develop.

Table 3 Characteristics of different circuits from ISCAS-85 benchmark

Benchmark	# of Inputs	# of Levels	# of nodes	# of nets	T_{OCR}	T_{HTS}	Description
c432	36	40	352	492	14	0.85	27-Channel Interrupt Controller
c880	60	43	607	889	15	0.82	8-Bit ALU
c1355	41	44	957	1416	20	0.75	32-Bit SEC Circuit
c1908	33	52	868	1304	14	0.90	16-bit SEC/DED Circuit
c2670	233	28	1323	1807	20	0.83	12-bit ALU and Controller
c3540	50	60	1539	2527	15	0.84	8-bit ALU
c5315	178	63	2697	4292	21	0.79	9-bit ALU
c6288	32	240	4496	6801	18	0.8	16×16 Multiplier
c7552	207	53	3561	5433	20	0.8	32-Bit Adder/Comparator

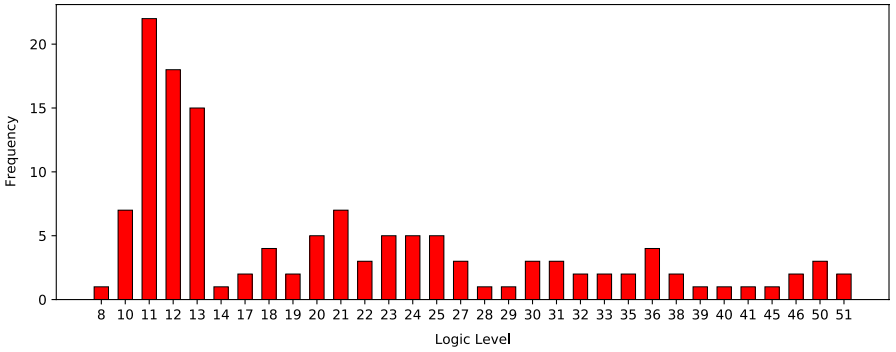
Table 4 Mean HT detection/insertion training time of the RL algorithm for different ISCAS-85 benchmarks

Benchmark	Insertion/detection timesteps	Insertion/detection training time
c432	120K/450K	1 h 40 m/1 h 7 m
c880	132K/495K	2 h 36 m/2 h 7 m
c1355	145K/550K	3 h 10 m/2 h 27 m
c1908	160K/605K	5 h 25 m/2 h 40 m
c2670	175K/665K	8 h 1 m/7 h 23 m
c3540	192K/731K	12 h 1 m/5 h 24
c5315	211K/800K	23 h 16 m/15 h 36 m
c6288	232K/880K	57 h 18 m/59 h 16 m
c7552	255K/970K	26 h 15 m/44 h 15 m

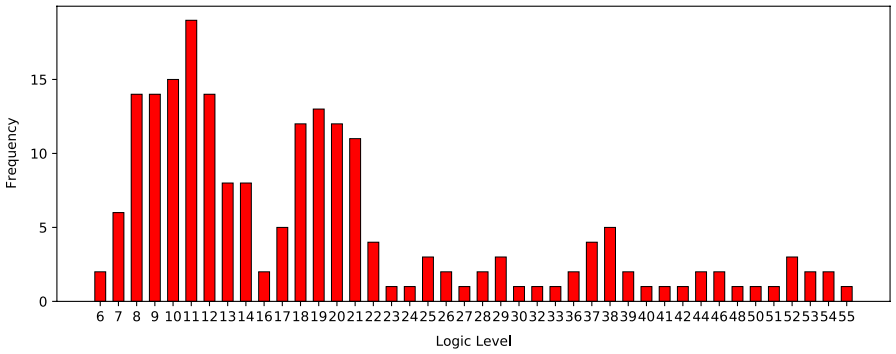
Table 3 provides details of the benchmark circuits used in our experiments. The table represents the number of primary inputs (2nd column), logic levels (3rd column), number of nodes including inputs, outputs, and logic gates (4th column), and nets (5th column). We have specified T_{OCR} and T_{HTS} such that 5% of all nets in each circuit are considered as *rare nets* (6th and 7th columns, respectively). This was done to enable a fair comparison between the circuits. Finally, the circuit functionality is listed in the 8th column.

6.1 Timing complexity and scalability

Table 4 provides timing information on training the HT insertion and detection agents per circuit. The 2nd column shows the total timesteps for insertion/detection, and the 3rd column shows the total spent time. We initialize training the inserting agent in *c*432 with 120K timesteps and an episode length of 450. We increase both values by 10% for each succeeding circuit to ensure enough exploration is made in each circuit as their size grows. As for detection, we start with 450K timesteps and increase it by 10% for subsequent circuits, and we keep the episode length at 10. The



(a) c3540



(b) c5315

Fig. 11 Distribution of rare nets in $c3540$ and $c5315$

short episode length allows the agent to experience different states, thereby increasing the chances of exploration. The test vectors are collected after running the agent for 20K episodes in the testing phase.

In our experiments, $c6288$ takes the most time in both insertion and detection scenarios (2.5 days), which we argue is reasonable for an attacker and the defense engineer. Note that we have not used optimization techniques to reduce the number of gates and nets in the benchmarks. Such techniques can notably decrease the RL environment size and, subsequently, the training time. That being said, the impact of optimization techniques on detection/insertion quality should be investigated, but it is beyond the scope of this paper.

6.2 Insertion, detection, and confidence value figures

Figure 11 illustrates the logical depth distribution of rare nets in $c3540$ and $c5315$ circuits. Although rare nets are primarily found in the lower logic levels, there are still a significant number of rare nets in the higher levels, which could contribute to the creation of stealthier hardware Trojans. As explained in Sect. 3.2, the level

Table 5 Number of inserted HTs under P_{rand} and P_{high} scenarios for ISCAS-85 benchmark circuits

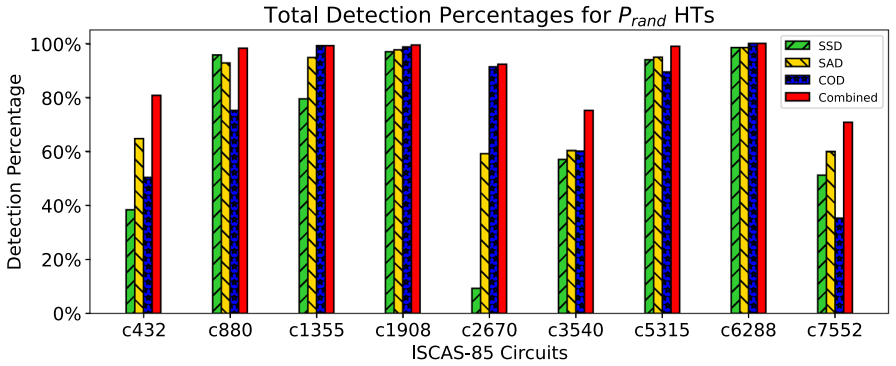
Benchmark	$P_{rand}-Total$	$P_{high}-Total$	$P_{rand}-3$	$P_{high}-3$	$P_{rand}-4$	$P_{high}-4$	$P_{rand}-5$	$P_{high}-5$
c432	1866	2788	1688	2331	160	453	18	4
c880	1954	2116	1595	1736	327	373	32	7
c1355	921	1400	815	1116	86	268	20	16
c1908	1247	1576	1121	1240	126	321	**0**	15
c2670	206	434	188	406	18	28	**0**	**0**
c3540	410	767	367	703	41	64	2	**0**
c5315	434	797	406	719	28	77	**0**	1
c6288	531	475	459	426	67	46	5	3
c7552	769	683	704	615	64	67	1	1

The zeros were bolded to make them easy to spot and stand out

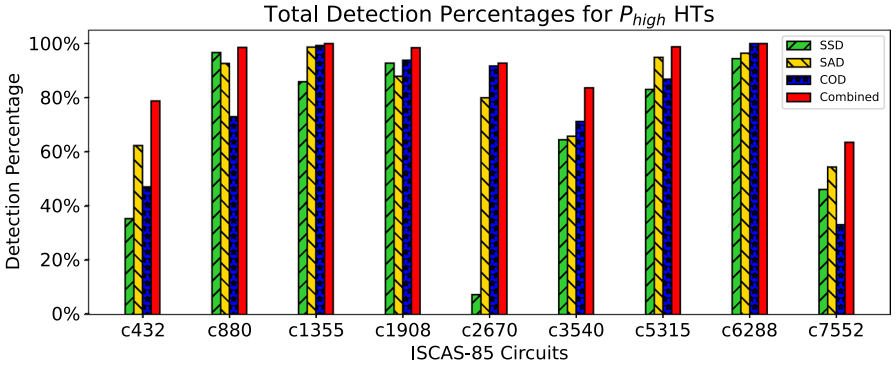
of the HT trigger nets is limited by the payload's level. Suppose a payload is not selected from the higher-level nets. In that case, the agent has less opportunity to explore higher-level trigger nets, which might harm the insertion exploration of new HTs. To enable more exploration, we define the following two payload selection scenarios: (1) P_{rand} in which the agent selects payloads randomly, and (2) P_{high} where payload net is selected such that at least 80% of rare nets are within the agent's sight.

Table 5 provides information about the number of inserted HTs using P_{rand} and P_{high} scenarios for each benchmark circuit. The 2nd and 3rd columns show the total number of HTs successfully inserted by the agent. The numbers followed by each insertion scenario in the remaining columns show the number of rare nets among the five input triggers. For instance, in $c432$, 1866 HTs were inserted under P_{rand} where 1688 of those had 3 rare nets, 160 of those had 4 rare nets, and only 18 of those had 5 rare nets. As can be observed, in most cases, the number of inserted HTs under P_{high} is higher than P_{rand} except for $c6288$ and $c7552$. Also, fewer HTs are inserted as the number of rare triggers increases. In other words, it becomes more difficult for the RL agent to find HTs with higher rare nets. There are some cases under $P_{rand}-5$ and $P_{high}-5$ that the agent could not insert any HTs. These rows in the table are shown as 0, e.g. , in $c2670$.

Figure 12 displays the HT detection accuracy percentages for the studied circuits under P_{rand} and P_{high} insertion scenarios. Figures 13, 14, and 15 provide details about the detection accuracy of each HT group, separately. Besides *SSD*, *SAD*, and *COD*, there is an extra detection scenario called *Combined* where all the test vectors produced by *SSD*, *SAD*, and *COD* are consolidated and applied to the circuits for HT detection. No detection rates are reported in cases where no HTs were inserted. It can be observed from both Table 5 and Figs. 12, 13, 14, and 15 that despite more inserted HTs in the P_{high} scenario, they do not evade detection any better than the random payload selection scenario and the detection rates are almost the same. Nevertheless, the extra inserted HTs under P_{high} can be used to train better ML HT detectors. Figures 12, 13, 14, and 15 also suggest that SSD, SAD, and COD are vital to providing better HT detection coverage. Figure 16 displays the number of times each



(a) P_{rand} HTs

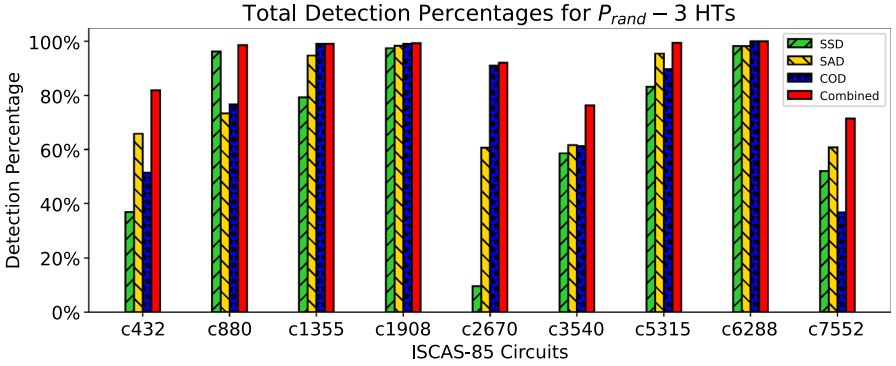


(b) P_{high} HTs

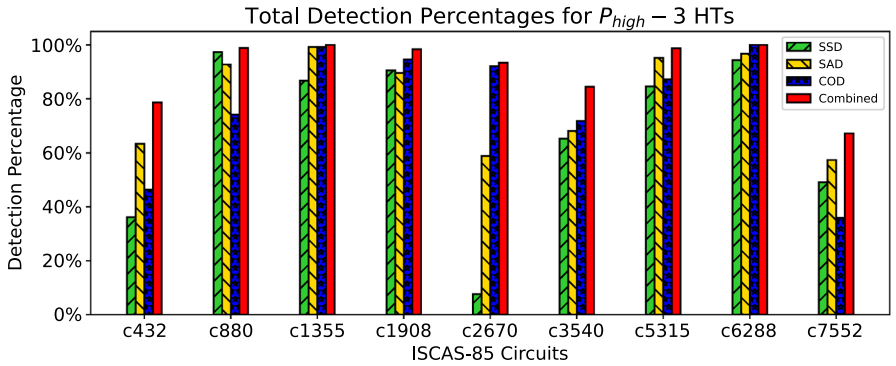
Fig. 12 Detection accuracy of SSD, SAD, COD, and *Combined* scenarios under P_{high} and P_{high} insertion scenarios in ISCAS-85 benchmark circuits

detector was ranked first in nine benchmark circuits under our two insertion strategies. While *COD* ties with *SAD* under P_{rand}, it becomes the best detector under P_{high}. *SSD* only outperforms in 1 benchmark circuit in both scenarios. The figure suggests that solely developing HT detectors based on signal activity might not achieve the expected outcomes. Nevertheless, *SAD* still plays an essential role in overall HT detection accuracy. The impact of the *Combined* scenario is vital as it improves the overall detection accuracy in most cases. For instance, in $c3540$, none of the detectors can perform better than 60% in the P_{rand} scenario while the *Combined* detection accuracy is nearly 75%. It also can be seen that adding more rare nets to the HT trigger does not necessarily lead to stealthier HTs. For example, in $c880$, $c1355$, and $c1908$, there are HTs with five trigger nets that were 100% detected, while the detection accuracy was less for HTs with fewer rare triggers in the same circuits.

Another important observation is the different magnitude of detection accuracy among the benchmark circuits. While we achieve 100% accuracy in $c6288$, it is about 25–30% lower in $c3540$ and $c6288$. Table 3 shows that $c6288$ is a



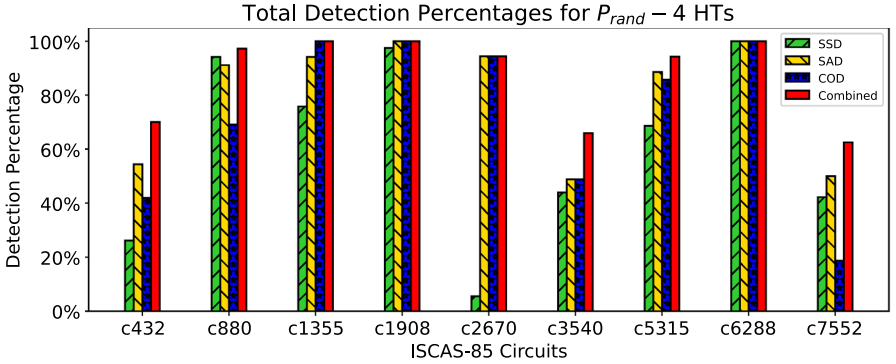
(a) $P_{rand} - 3\ HTs$



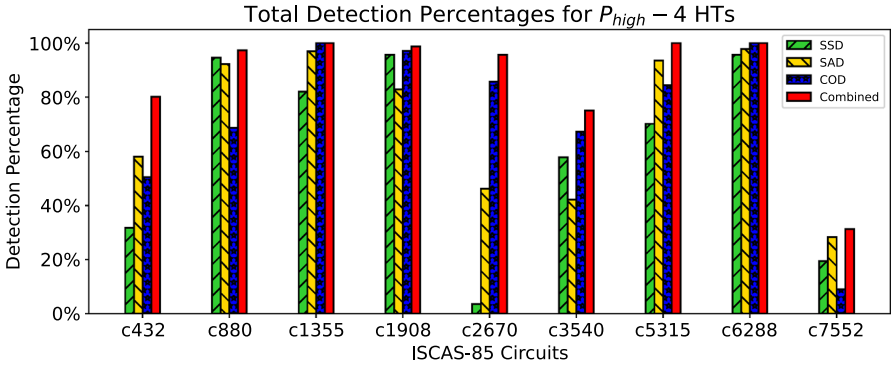
(b) $P_{high} - 3\ HTs$

Fig. 13 Detection accuracy of SSD, SAD, COD, and *Combined* scenarios under P_{high} and P_{high} insertion scenarios for 3-input HTs

multiplier circuit. It contains 240 full and half adders arranged in a 15×16 matrix [39]. $c3540$, on the other hand, has 14 control inputs for multiplexing and masking data. $c7552$ also contains multiple control signals and bit masking operations. We hypothesize that the detection accuracy is higher in $c6288$ due to having fewer control signals that disable circuit components and signals. Accordingly, they get more frequently activated in $c6288$ than $c3540$ and $c7552$. In other words, these results imply that inserting HTs in control paths can lead to stealthier HTs than data paths in circuits. Another interesting finding pertains to the detection rate in $c432$. After administering 100K random test patterns, we discovered that the rarest net in the circuit was triggered 7% of the times, starkly contrasting to other circuits where many nets exhibit less than 1% switching activity. It implies that random test patterns probably more easily activate the inserted HTs in $c432$. We generated 20K random test patterns to prove this hypothesis and passed them to the circuit. These test patterns detected 99% of HTs, indicating that attackers



(a) $P_{rand} - 4\ HTs$



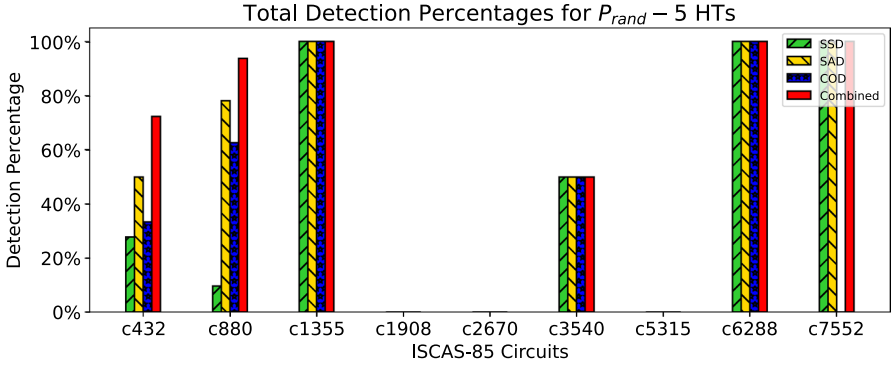
(b) $P_{high} - 4\ HTs$

Fig. 14 Detection accuracy of SSD, SAD, COD, and *Combined* scenarios under P_{high} and P_{high} insertion scenarios for 4-input HTs

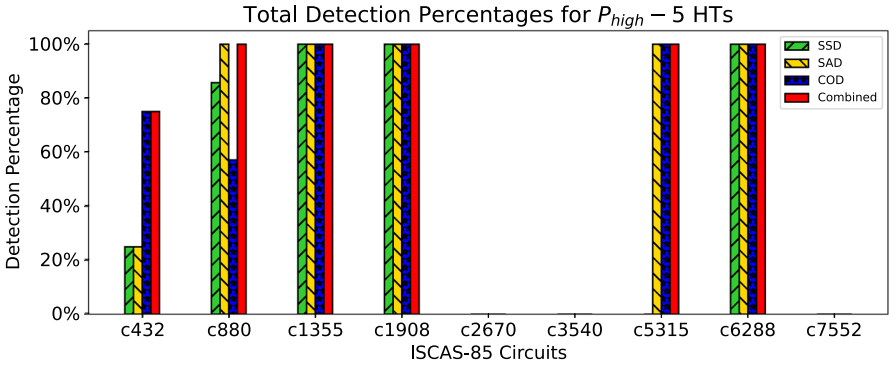
should carefully evaluate the activity profile of the nets before compromising circuits.

To further evaluate the efficacy of our HT detectors, we compare the *Combined* detector with DETERRENT [10] and HW2VEC [29], two state-of-the-art HT detectors. We use the test vectors generated by DETERRENT [10] and collect detection figures for 4 reported ISCAS-85 benchmark circuits, namely *c*2670, *c*5315, *c*6288, and *c*7552.[2] We also replicate the steps in HW2VEC [29] by gathering the *TJ_RTL* dataset, which contains 26 HT-infected (labeled as '1') and 11 HT-Free circuits (labeled as '0'). We train an MLP (multi-layer perceptron) binary classifier to detect the HTs. For the test dataset, we collect the graph embeddings of the HTs generated by the inserting RL agent. Additionally, we add an HT-free version of the original ISCAS-85 circuits and another one synthesized with the academic

[2] We reached out to the authors of TARMAC and TGRL techniques, but we did not receive the test patterns at the time of submission.



(a) $P_{rand} - 5\ HTs$



(b) $P_{high} - 5\ HTs$

Fig. 15 Detection accuracy of SSD, SAD, COD, and *Combined* scenarios under P_{high} and P_{high} insertion scenarios for 5-input HTs

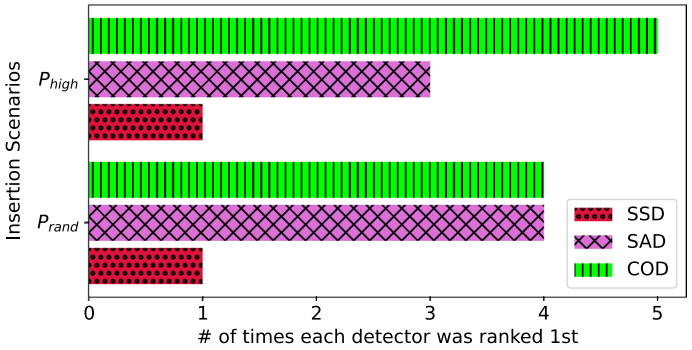


Fig. 16 Comparing the number of times each of *SSD*, *SAD*, and *COD* are ranked as the best detector in our two insertion scenarios

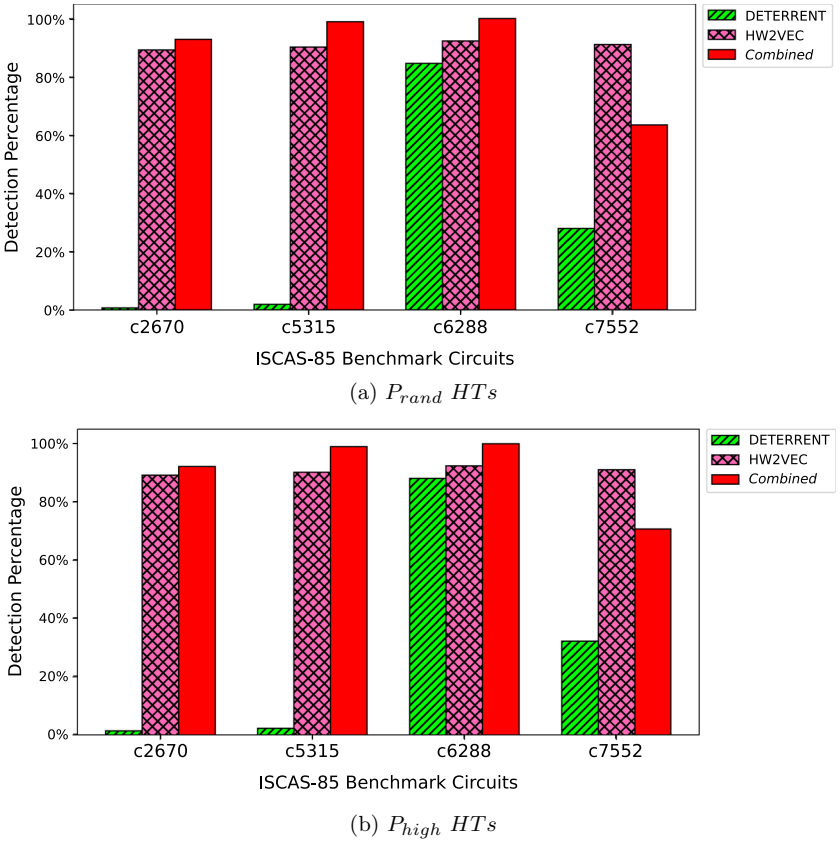


Fig. 17 Comparison of HW2VEC [29], *Combined*, and DETERRENT [10] detection rates under **a** P_{rand} and **b** P_{high} insertion scenarios

NanGateOpenCell45nm library to the test batch to record the number of *TN*s and *FP*s. As shown in Table 2, DETERRENT solely considers signal activity while HW2VEC captures structural information of circuits.

Figure 17 shows the detection accuracy of each HT detector for each benchmark circuit. The detection accuracy is reported for the total inserted HTs in Table 5 for both P_{rand} and P_{high} insertion scenarios. The figure shows that the *Combined* detector outperforms DETERRENT and HW2VEC in 3 of our benchmark circuits. The average detection rate among the 4 benchmarks is 87% percent. While the detection gap between *Combined* and DETERRENT is significant in *c*2670 and *c*5315, it is less evident in *c*6288 and *c*7552. HW2VEC, on the other hand, demonstrates minimal detection variance in all 4 circuits and outperforms *Combined* in *c*7552. Furthermore, HW2VEC illustrates robust performance with HT-Free circuits, correctly classifying them as *TN*s and a *FP* rate of 0.

In another experiment, we train our MLP with *TJ_RTL* + *EPFL* [40] benchmark suites to obtain a more balanced dataset (26 instances labeled as '1' and 30 instances

Table 6 Individual contribution of *SSD*, *SAD*, and *COD* in detection of unique HTs

Circuit	*SSD #*	*SSD%*	*SAD #*	*SAD%*	*COD #*	*COD%*
c432	2	0.1%	275	14.74%	297	15.86%
c880	49	2.52%	16	0.81%	16	0.81%
c1355	0	0%	0	0%	40	4.34%
c1908	1	0.08%	1	0.08%	13	1.04%
c2670	0	0%	1	0.48%	66	32.03%
c3540	7	1.70%	29	7.07%	18	4.39%
c5315	1	0.24%	8	1.93%	9	2.17%
c6288	0	0%	0	0%	8	1.51%
c7552	16	2.08%	29	3.77%	15	1.95%

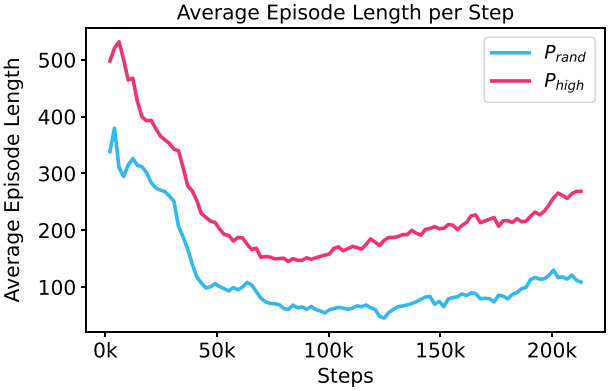
labeled as '0'). While the *FP* remains 0, similar to the previous experiment, the HT detection accuracy drops to 48%. This sheds light on the shortcomings of the current benchmarks used for training ML HT detectors, and it raises the necessity of having a more diverse and larger dataset to attain more dependable results. Overall, these two experiments demonstrate the potential of the RL inserting agent and the advantages of a multi-criteria detector compared to a single-criterion (DETERRENT) HT detector.

Table 6 shows the individual detection contribution of *SSD*, *SAD*, and *COD* toward overall HT detection for each benchmark circuit. The 2nd, 4th, and 6th columns display the number of HTs exclusively detected by each detector followed by their contribution in the overall HT detection in the 3rd, 5th, and 7th columns for *SSD*, *SAD*, and *COD*, respectively. As can be inferred, *COD* has the highest individual contribution, followed by *SAD* and *SSD*. This table is evidence of the importance of the multi-criteria HT detector for higher accuracy.

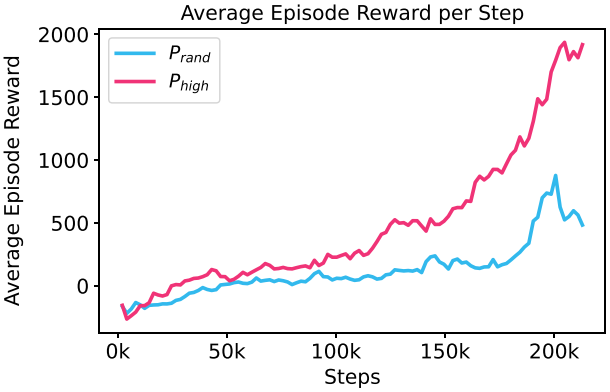
To compute the confidence value of each detector, the overall detection accuracy of each detector is calculated in all nine circuits under both insertion scenarios. Then, each averaged value is plugged into Eq. (4). Assuming $\alpha = 10$, the confidence values for each *SSD*, *SAD*, *COD*, and *combined* scenarios are 2.43, 3.36, 3.09, and 5.13, respectively. Thus, the security engineer can put more confidence in the *Combined* detector since it has the highest confidence values. DETERRENT's and HW2VEC's confidence values are 1.24 and 4.34, respectively.

6.3 Average episode length and reward

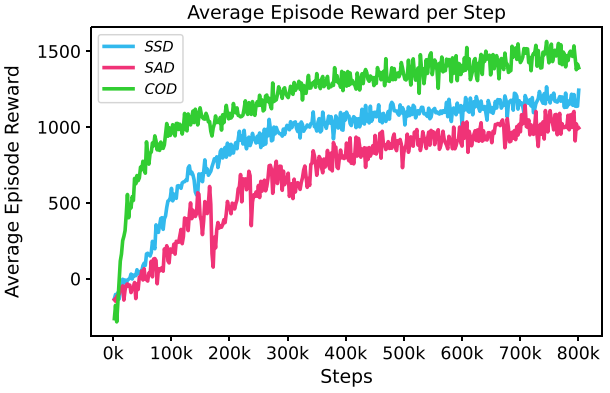
Figure 18 shows the average episode length and reward of the inserting and detector RL agents for the *c*5315 benchmark circuit. As seen from Fig. 18a, initially, the agent leans more toward ending the training episodes to avoid further losses. This trend continues until it gradually increases the episode length, increasing the reward, which can be observed in Fig. 18b. Eventually, the agent collects more and more rewards. Although the agent accumulates higher rewards in P_{high}, the detection rate is not significantly different from P_{rand}. Figure 18c demonstrates the agent's ability to augment rewards in our three detection scenarios at an almost steady pace; it



(a) Average Episode Length per Step in HT insertion for $c5315$



(b) Average Episode Reward per Step in HT insertion for $c5315$



(c) Average Episode Reward per Step in HT Detection for $c5315$

Fig. 18 The average episode length and reward vs. the number of steps in both HT insertion and detection for $c5315$

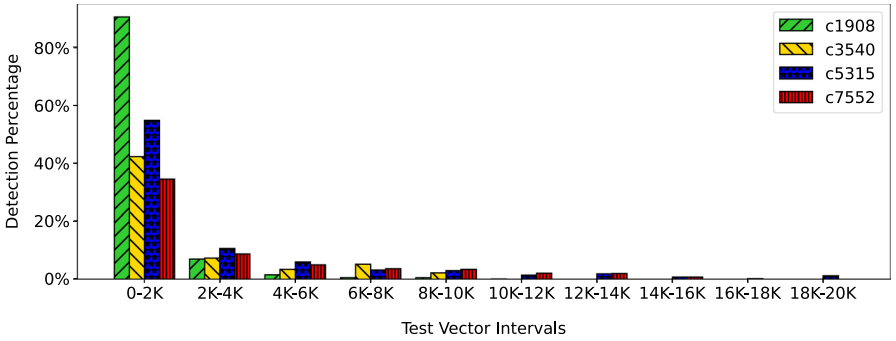


Fig. 19 The number of generated test vectors (x-axis) versus the HT detection accuracy (y-axis)

learns how to increase rewards along the way. It is worthwhile to point out that the proposed RL framework can save the state of the RL models at arbitrary intervals, which helps test the agent's efficacy at different timesteps. Note that since the detector's episode length is always 10, this data was not included in the graph. The agent can always be trained for longer steps, but one should consider the trade-off between the time required and the accuracy achieved.

6.4 Test vector size versus accuracy

We also investigate the relationship between the number of applied test vectors and the HT detection accuracy. For this experiment, we collect a set of test vectors that have obtained a certain minimum reward. We run the trained RL agent for 20K episodes to identify such vectors. We set a cut-off reward of one-tenth of the collected reward in the last training episode. We collect 20K test vectors that surpass this reward threshold. The HT detection distribution of the collected test vectors is shown in Fig. 19 for $c1908$, $c3540$, $c5315$, and $c7552$ under the P_{rand} insertion scenario and the *SAD* detection scenario. The x-axis displays the intervals of the applied test vectors, and the y-axis shows the detection percentage of each particular interval. As can be seen, the first 2K vectors have the greatest contribution toward HT detection. This figure is nearly 90% for $c1908$ and just below 40% for $c7552$. A similar comparison can be made between different HT detectors to help us find the relation between the quantity (number of test vectors) and the quality (the detection accuracy). Such analysis leads us to answer the question, "Does adding more test vectors to the testing batch improve detection?" If the answer is negative, adopting more intelligent rewarding functions might be considered to offset this diminishing returns effect. That being said, in certain instances, adding more test batches leads to higher detection rates. We tested this scenario for $c3540$ where the *Combined* detection rate with 20K test patterns is around 80% in the P_{rand} scenario. We ran the trained detector agents *SSD*, *SAD*, and *COD* for 20K episodes, but this time, we collected all the test patterns that returned positive rewards. Accordingly, we collected 191K, 183K, 121K for *SSD*, *SAD*, and *COD* and the detection rates were 89, 86, and 97%, respectively.

6.5 RL Feasibility in practice

RL agents have been extensively used in various application domains where decision making is required, e.g. , robotics control [41, 42], gaming [43, 44], autonomous driving [45], computer architecture [46, 47], and hardware security [48]. Training RL models requires a large amount of interaction with the environment to learn an optimal policy. This can be costly in many environments (including the HT space) where the interactions with the environment are computationally expensive. The OpenAI RL agent that defeated the DOTA world champions famously took 10 months to train [49]. Despite the training hurdle, RL introduces some valuable advantages in relationship to HTs. First, RL facilitates the exploration of complex environments that humans cannot easily accomplish. It automates the decision-making process and eases automation especially where tasks must be performed repeatedly or in large volumes. RL, as an unsupervised learning technique, can build training sets for other agents that are then trained via supervised learning, for instance, an HT benchmark for training an ML-based HT detector. Moreover, RL removes the human bias stemming from a particular mindset in the process. RL has already proved to be a valuable solution in the HT domain [2, 10, 16]. While utilizing RL helps security engineers produce test vectors, the next generation of malicious actors might be bots designed to compromise security. Hence, despite the added layer of complexity, we believe that utilizing an RL approach for HT insertion and detection is feasible where the sheer complexity of the problem means that we need to explore all potential research avenues.

7 Conclusions and future directions

This paper presented the first framework for joint HT insertion and detection. The inserting and detection RL agents have tunable rewarding functions that enable researchers to experiment with different approaches to the problem. This framework will accelerate HT research by helping the research community evaluate their insertion/detection ideas with less effort. Our inserting tool provides a robust dataset that can be used for developing finer HT detectors, and our detector tool emphasizes the need for a multi-criteria detector that can cater to different HT insertion mindsets. We also presented a methodology to help the community compare HT detection methods, regardless of their implementation details. We applied this methodology to our HT detection and discovered that our tool offers the highest confidence in HT detection when using a combined detection scenario. We aim to explore more benchmarks and create a more diverse HT dataset for the community.

As an extension to this work, we aim to explore more benchmarks and provide support for other circuits, including sequential ones in our flow, e.g. , ISCAS-89 [50] and ITC'99 [51] benchmarks. One solution to tackle these circuits would be utilizing Design for Testability (DFT) techniques such as scan chains [52]. In a full scan design, memory elements are connected to the chain, enabling test engineers to use combinational test patterns instead of sequential ones. Given the existence of this

playground infrastructure, further research questions and more complex ideas can be explored.

Author contributions Initial draft of the manuscript was prepared by A.S.; all authors edited it. All the development of the technical work was done by A.S. The technical feedback, discussions, and ideas were developed as a team.

Funding This work has been partially funded by NSF grants 2219680 and 2219679.

Availability of data and materials Our HT benchmark and test vectors are available on https://github.com/NMSU-PEARL/Hardware-Trojan-Insertion-and-Detection-with-Reinforcement-Learning We will share the hardware Trojan benchmark on a case-by-case basis. Requests can be made on the same GitHub repository.

Declarations

Conflict of interest The authors declare no competing interests.

Ethical approval Not applicable.

References

1. Securing Defense-Critical Supply Chains: An action plan developed in response to President Biden's Executive Order 14017. https://tinyurl.com/3wmddx5d
2. Pan Z, Mishra P (2021) Automated test generation for hardware trojan detection using reinforcement learning. In: Proceedings of the 26th Asia and South Pacific Design Automation Conference, pp 408–413. https://doi.org/10.1145/3394885.3431595
3. Shakya B, He T, Salmani H, Forte D, Bhunia S, Tehranipoor M (2017) Benchmarking of hardware trojans and maliciously affected circuits. J Hardw Syst Secur 1(1):85–102. https://doi.org/10.1007/s41635-017-0001-6
4. Sarihi A, Patooghy A, Khalid A, Hasanzadeh M, Said M, Badawy A-HA (2021) A survey on the security of wired, wireless, and 3d network-on-chips. IEEE Access. https://doi.org/10.1109/ACCESS.2021.3100540
5. Salmani H, Tehranipoor M, Karri R (2013) On design vulnerability analysis and trust benchmarks development. In: 2013 IEEE 31st International Conference on Computer Design (ICCD). IEEE, pp 471–474 https://doi.org/10.1109/ICCD.2013.6657085
6. Trust-Hub. https://trust-hub.org/. Accessed 8 Nov 2023
7. Salmani H (2016) Cotd: Reference-free hardware trojan detection and recovery based on controllability and observability in gate-level netlist. IEEE Trans Inf Forensics Secur 12(2):338–350. https://doi.org/10.1109/TIFS.2016.2613842
8. Hasegawa K, Yanagisawa M, Togawa N (2017) Trojan-feature extraction at gate-level netlists and its application to hardware-Trojan detection using random forest classifier. In: 2017 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, pp 1–4. https://doi.org/10.1109/ISCAS.2017.8050827
9. Sebt SM, Patooghy A, Beitollahi H, Kinsy M (2018) Circuit enclaves susceptible to hardware trojans insertion at gate-level designs. IET Comput Digit Tech 12(6):251–257. https://doi.org/10.1049/iet-cdt.2018.5108
10. Gohil V, Patnaik S, Guo H, Kalathil D, Rajendran J (2022) Deterrent: detecting Trojans using reinforcement learning. In: Proceedings of the 59th ACM/IEEE Design Automation Conference, pp 697–702. https://doi.org/10.1145/3489517.3530518

11. Cruz J, Huang Y, Mishra P, Bhunia S (2018) An automated configurable trojan insertion framework for dynamic trust benchmarks. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, pp 1598–1603. https://doi.org/10.23919/DATE.2018.8342270
12. Lyu Y, Mishra P (2020) Scalable activation of rare triggers in hardware trojans by repeated maximal clique sampling. IEEE Trans Comput Aided Des Integr Circuits Syst 40(7):1287–1300. https://doi.org/10.1109/TCAD.2020.3019984
13. Fyrbiak M, Wallat S, Swierczynski P, Hoffmann M, Hoppach S, Wilhelm M, Weidlich T, Tessier R, Paar C (2018) HAL—the missing piece of the puzzle for hardware reverse engineering, Trojan detection and insertion. IEEE Trans Dependable Secur Comput. https://doi.org/10.1109/TDSC.2018.2812183
14. Yu S, Liu W, O'Neill M (2019) An improved automatic hardware trojan generation platform. In: 2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). IEEE, pp 302–307. https://doi.org/10.1109/ISVLSI.2019.00062
15. Chakraborty RS, Wolff F, Paul S, Papachristou C, Bhunia S (2009) Mero: a statistical approach for hardware Trojan detection. In: International Workshop on Cryptographic Hardware and Embedded Systems. Springer, Berlin, pp 396–410. https://doi.org/10.1007/978-3-642-04138-9_28
16. Gohil V, Guo H, Patnaik S, Rajendran J (2022) Attrition: attacking static hardware Trojan detection techniques using reinforcement learning. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp 1275–1289. https://doi.org/10.1145/3548606.3560690
17. Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O (2017) Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347
18. Sarihi A, Patooghy A, Jamieson P, Badawy A-HA (2022) Hardware Trojan insertion using reinforcement learning. In: Proceedings of the Great Lakes Symposium on VLSI 2022, pp 139–142. https://doi.org/10.1145/3526241.3530379
19. Sarihi A, Jamieson P, Patooghy A, Badawy A-HA (2023) Multi-criteria hardware Trojan detection: a reinforcement learning approach. In: 2023 IEEE 66th International Midwest Symposium on Circuits and Systems (MWSCAS), pp 1093–1097
20. Krieg C (2023) Reflections on trusting TrustHUB. In: 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD). IEEE, pp 1–9
21. Jyothi V, Krishnamurthy P, Khorrami F, Karri R (2017) Taint: tool for automated insertion of Trojans. In: 2017 IEEE International Conference on Computer Design (ICCD). IEEE, pp 545–548. https://doi.org/10.1109/ICCD.2017.95
22. Cruz J, Gaikwad P, Nair A, Chakraborty P, Bhunia S (2022) Automatic hardware Trojan insertion using machine learning. arXiv preprint arXiv:2204.08580
23. Perez T, Imran M, Vaz P, Pagliarini S (2021) Side-channel trojan insertion-a practical foundry-side attack via eco. In: 2021 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, pp 1–5. https://doi.org/10.1109/ISCAS51556.2021.9401481
24. Perez T, Pagliarini S (2022) Hardware Trojan insertion in finalized layouts: from methodology to a silicon demonstration. IEEE Trans Comput Aided Des Integr Circuits Syst. https://doi.org/10.1109/TCAD.2022.3223846
25. Puschner E, Moos T, Becker S, Kison C, Moradi A, Paar C (2023) Red team vs. blue team: a real-world hardware Trojan detection case study across four modern CMOS technology generations. In: 2023 IEEE Symposium on Security and Privacy (SP). IEEE, pp 56–74. https://doi.org/10.1109/SP46215.2023.10179341
26. Hepp A, Perez T, Pagliarini S, Sigl G (2022) A pragmatic methodology for blind hardware trojan insertion in finalized layouts. In: Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design, pp 1–9. https://doi.org/10.1145/3508352.3549452
27. Nozawa K, Hasegawa K, Hidano S, Kiyomoto S, Hashimoto K, Togawa N (2021) Generating adversarial examples for hardware-Trojan detection at gate-level netlists. J Inf Process 29:236–246. https://doi.org/10.2197/ipsjjip.29.236
28. Goldstein LH, Thigpen EL (1980) Scoap: sandia controllability/observability analysis program. In: Proceedings of the 17th Design Automation Conference, pp 190–196. https://doi.org/10.1145/800139.804528

29. Yu S-Y, Yasaei R, Zhou Q, Nguyen T, Al Faruque MA (2021) Hw2vec: a graph learning tool for automating hardware security. In: 2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, pp 13–23. https://doi.org/10.1109/HOST49136.2021.9702281
30. Wolf C, Glaser J, Kepler J (2013) Yosys-a free Verilog synthesis suite. In: Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)
31. Bassett L (2015) Introduction to JavaScript Object notation: a to-the-point guide to JSON. O'Reilly Media, Sebastopol. https://books.google.com/books?id=Qv9PCgAAQBAJ
32. Brockman G, Cheung V, Pettersson L, Schneider J, Schulman J, Tang J, Zaremba W (2016) Openai gym. CoRR arXiv:1606.01540
33. Bushnell ML (2000) Essentials of electronic testing for digital. In: Memory & Mixed-Signal VLSI Circuits. https://doi.org/10.1007/b117406
34. Nguyen TT, Reddi VJ (2019) Deep reinforcement learning for cyber security. IEEE Trans Neural Netw Learn Syst. https://doi.org/10.1109/TNNLS.2021.3121870
35. Raffin A, Hill A, Gleave A, Kanervisto A, Ernestus M, Dormann N (2021) Stable-baselines3: reliable reinforcement learning implementations. J Mach Learn Res 22(268):1–8
36. Bryan D (1985) The ISCAS'85 benchmark circuits and netlist format
37. Hagberg AA, Schult DA, Swart PJ (2008) Exploring network structure, dynamics, and function using networkX. In: Varoquaux G, Vaught T, Millman J (eds) Proceedings of the 7th Python in Science Conference, Pasadena, CA USA, pp 11–15
38. GitHub-NMSU-PEARL/Hardware-Trojan-Insertion-and-Detection-with-Reinforcement-Learning: Reinforcement Learning-based Hardware Trojan Detector—github.com. https://github.com/NMSU-PEARL/Hardware-Trojan-Insertion-and-Detection-with-Reinforcement-Learning. Accessed 27 Dec 2023
39. ISCAS High-Level Models. https://web.eecs.umich.edu/~jhayes/iscas.restore/benchmark.html. Accessed 7 Nov 2023
40. Amarú L, Gaillardon P-E, De Micheli G (2015) The EPFL combinational benchmark suite. In: Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)
41. Tai L, Paolo G, Liu M (2017) Virtual-to-real deep reinforcement learning: continuous control of mobile robots for Mapless navigation. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, pp 31–36. https://doi.org/10.1109/IROS.2017.8202134
42. Hwangbo J, Lee J, Dosovitskiy A, Bellicoso D, Tsounis V, Koltun V, Hutter M (2019) Learning agile and dynamic motor skills for legged robots. Sci Robot 4(26):5872. https://doi.org/10.1126/scirobotics.aau5872
43. Silver D, Huang A, Maddison CJ, Guez A, Sifre L, Van Den Driessche G, Schrittwieser J, Antonoglou I, Panneershelvam V, Lanctot M et al (2016) Mastering the game of go with deep neural networks and tree search. nature 529(7587):484–489. https://doi.org/10.1038/nature16961
44. Silver D, Hubert T, Schrittwieser J, Antonoglou I, Lai M, Guez A, Lanctot M, Sifre L, Kumaran D, Graepel T et al (2018) A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science 362(6419):1140–1144. https://doi.org/10.1126/science.aar6404
45. Talpaert V, Sobh I, Kiran BR, Mannion P, Yogamani S, El-Sallab A, Perez P (2019) Exploring applications of deep reinforcement learning for real-world autonomous driving systems. arXiv preprint arXiv:1901.01536
46. Zhou Y, Roy S, Abdolrashidi A, Wong D, Ma PC, Xu Q, Zhong M, Liu H, Goldie A, Mirhoseini A, et al (2019) Gdp: Generalized device placement for dataflow graphs. arXiv preprint arXiv:1910.01578
47. Yin J, Eckert Y, Che S, Oskin M, Loh GH (2018) Toward more efficient NOC arbitration: a deep reinforcement learning approach. In: Proc. IEEE 1st Int. Workshop AI-assisted Des. Architecture, vol 128
48. Patnaik S, Gohil V, Guo H, Rajendran JJ (2022) Reinforcement learning for hardware security: opportunities, developments, and challenges. In: 2022 19th International SoC Design Conference (ISOCC), pp 217–218. https://doi.org/10.1109/ISOCC56007.2022.10031569
49. Berner C, Brockman G, Chan B, Cheung V, Debiak P, Dennison C, Farhi D, Fischer Q, Hashme S, Hesse C, et al (2019) Dota 2 with large scale deep reinforcement learning. arXiv preprint arXiv:1912.06680
50. WWW: ISCAS89 Sequential Benchmark Circuits—filebox.ece.vt.edu. https://filebox.ece.vt.edu/~mhsiao/iscas89.html. Accessed 22 Jan 2024
51. ITC'99 Benchmark Homepage—cerc.utexas.edu. https://www.cerc.utexas.edu/itc99-benchmarks/bench.html. Accessed 22 Jan 2024

52. Narayanan S, Gupta R, Breuer MA (1993) Optimal configuring of multiple scan chains. IEEE Trans Comput 42(9):1121–1131. https://doi.org/10.1109/12.241600