2024 Design, Automation & Test in Europe Conference (DATE 2024)

RL-TPG : Automated Pre-Silicon Security Verification through **R**einforcement **L**earning-Based **T**est **P**attern **G**eneration

Nurun Nahar Mondol
ECE, University of Florida
Gainesville, United States
nmondol@ufl.edu

Arash Vafei
ECE, University of Florida
Gainesville, United States
arash.vafaei@ufl.edu

Kimia Zamiri Azar
ECE, University of Florida
Gainesville, United States
k.zamiriazar@ufl.edu

Farimah Farahmandi
ECE, University of Florida
Gainesville, United States
farimah@ece.ufl.edu

Mark Tehranipoor
ECE, University of Florida
Gainesville, United States
tehranipoor@ece.ufl.edu

Verifying the security of System-on-Chip (SoC) designs against hardware vulnerabilities is challenging because of the increasing complexity of SoCs, the diverse sources of vulnerabilities, and the need for comprehensive testing to identify potential security threats. In this paper, we propose RL-TPG, a novel framework that combines traditional verification with hardware security verification using Reinforcement Learning (RL) in Register Transfer Level (RTL) design. Significant research has been done on formal verification, semi-formal verification, automated security asset identification, and gate-level netlist. However, the area of automated simulation using machine learning at RTL is still unexplored. RL-TPG employs an RL agent that generates intelligent test patterns targeting security properties, verification coverage, and rare nodes of the design to achieve security property violation, increase verification coverage, and reach rare nodes. Our framework triggers all embedded vulnerabilities, achieving an average of 90% traditional coverage in an average of 192 seconds for the experimental benchmarks. To demonstrate the effectiveness of the approach, the results are compared with JasperGold by Cadence.

I. Introduction

The use of third-party intellectual property (3PIP) and globalization in the integrated circuit (IC) supply chain has expanded the system-on-chip (SoC) market. SoCs comprise diverse hardware intellectual properties (IPs) collaborating to provide complex functionalities. However, SoCs are vulnerable to security issues due to feature diversity, customization, and intricate IP interactions. These vulnerabilities may arise from designer errors, untrusted 3PIP vendors, insider threats, and a lack of security awareness in electronic design automation (EDA) tools. They can compromise the confidentiality, integrity, and availability of security assets (e.g., cipher keys)Addressing these vulnerabilities requires comprehensive verification and countermeasures, as SoCs may face various adversarial threats throughout their lifetime. Detecting vulnerabilities during the early IC design stage is crucial to reduce re-spins and production costs. Failing to do so can inflate chip repair costs tenfold [1]. Various methods for IP security verification, such as formal verification [2], information flow tracking (IFT) [3], and concolic testing [4], have been explored but face challenges like manual security policy definition and scalability issues. Recent studies have introduced fuzz and penetration testing [5] to enhance SoC-level verification but still rely on manual inspection. Methods like [6, 7] operate at the gate-level instead of the RTL level, using RL on very small benchmarks, focusing only on the rarity of signals, and have scalability and reusability issues.

To address these issues in the existing works, we introduce an automated novel framework for generating test patterns using RL to identify hardware vulnerabilities at the RTL level. RL [8] is a branch of Machine Learning (ML) that involves an agent interacting with an environment to learn through trial and error. RL consists of four main components: an agent, environment, action field, and observation field. The agent receives feedback through rewards or penalties from the observation field and takes a new action through the action field by interacting with the environment. It then learns the best policy to maximize its reward over time. Our approach addresses three critical issues found in existing methods. First, traditional methods for logic testing involve computationally intensive processes that randomly flip bits in test vectors to detect rare activity, which are often not scalable or reusable [9]. Contrarily, our approach utilizes an RL framework, facilitating fast and automated test generation, reducing computational overhead, and providing scalability and reusability. Second, existing methods primarily focus on traditional verification, neglecting security verification [6]. Our proposed approach considers both signal rarity and security assets and significantly enhances test generation by combining traditional coverage metrics obtained from dynamic simulation. These coverage metrics are merged to provide feedback for rewarding and punishing RL models, thereby increasing traditional coverage and detecting security vulnerabilities. Third, cutting-edge automatic test pattern generation (ATPG) tools are widely accessible starting from the gate-level. However, we employ our framework at the RTL level to detect vulnerabilities in the early IC design stage. This paper makes several contributions:

- Introducing an automated security asset-based, reusable, and scalable test generation framework, called RL-TPG, for security verification that increases verification coverage and supports both functional and security verification at the RTL level.

979-8-3503-4859-0/DATE24

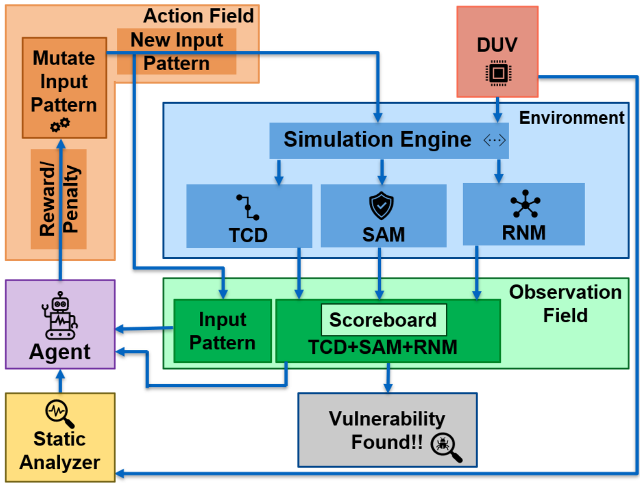


Fig. 1: Overview of RTL-TPG framework.

- Incorporating modern approaches such as static analysis and dynamic mutation methods in conjunction with RL to further enhance test generation efficiency.
- Combining coverage metrics like line, toggle, finite state machine (FSM), conditional, and branch coverage with rare signal coverage and security asset monitor coverage, a novel combination of metrics, to detect vulnerabilities and boost coverage.

The structure of the rest of this paper is as follows: Section II describes our RL-TPG framework in detail. Section III details about the experimental setup, Section IV presents our experimental results, and Section V concludes the paper with future work.

II. RL-TPG Framework Overview

This section discusses the workflow of RL-TPG as illustrated in Figure 1. It is designed for functional and security verification engineers to verify IP security. The framework serves the dual purpose of identifying design functionality errors and detecting potential security vulnerabilities, including integrity, confidentiality, or availability breaches. It comprises five main components: environment, observation field, static analyzer, RL agent, and action field.

The framework's primary objective is to generate intelligent RTL-level test patterns, potentially discovering unverified states that could pose security threats or functional anomalies. It continuously generates new test patterns to improve functional, security, and rareness coverage. Security assets within the Design Under Verification (DUV) are monitored to detect integrity, confidentiality, or availability violations, enabling the RL's learning loop termination.

- **Environment:** The RL environment has two components: the DUV and the simulation engine. The simulation engine generates the coverage data and monitors security assets and rare nodes. The environment gets the test patterns from the action space of the RL agent and stimulates the DUV with those test patterns. We used RTL designs as DUV and Synopsys VCS as the simulation engine to simulate the test patterns generated by RL. Once the simulation is complete, it gives out three parameters - Traditional Coverage Data (TCD), security asset Monitor (SAM), and Rare Node Monitor (RNM).

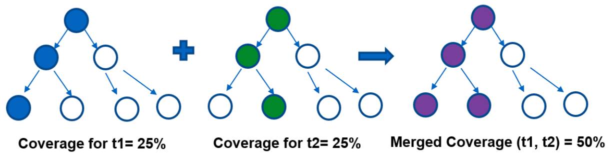


Fig. 2: Two different input patterns with the same coverage percentile.

Details for obtaining TCD, SAM, and RNM are given below:

1) **Traditional Coverage Data (TCD)**: TCD refers to line(l), conditional(c), toggle(tg), branch(b), and finite state machine(fsm) coverage expressed as a percentage. The RL agent's goal is to create new test patterns for achieving higher traditional coverage. While simulating individual test patterns, we generate single coverage data. However, relying solely on single coverage percentiles can be misleading because different test patterns with the same percentile may cover different parts of DUV, as shown in Figure 2. To ensure comprehensive coverage, we track all spaces covered by current and previous patterns. The RL agent generates patterns to cover previously unverified spaces, considering merged_coverage($t1, t2$) rather than individual coverage. We also maintain a weighted summation of all coverage types as $total\_cov$. RL generates test patterns, and as they pass through VCS, we merge all coverage data to provide the scoreboard with individual and comprehensive coverage information as shown in Algorithm. 1. The RL agent aims to maximize the TCD score, as a higher score indicates fewer unverified states and a higher chance of vulnerability detection.

Algorithm 1 TCD calculation

```
Input: $last\_test\_pattern(t_i), new\_test\_pattern(t)$, $DUV$
Output: $TCD$
 1: Initializing the first coverage values:
 2: $l_i, c_i, tg_i, b_i, fsm_i \leftarrow VCS(DUV, t_i)$
 3: $total\_cov_i = l_i + c_i + tg_i + b_i + fsm_i$
 4: Coverage values for rest of the simulation:
 5: while simulation do
 6:     $l, c, tg, b, fsm \leftarrow VCS(DUV, t)$
 7:     $total\_cov = l + c + tg + b + fsm$
 8:     $l_m, c_m, tg_m, b_m, fsm_m, total\_cov_m \leftarrow$
        $merge((l, l_i), (c, c_i), (tg, tg_i), (b, b_i), (fsm, fsm_i),$
        $(total\_cov, total\_cov_i))$
 9:     $l_i = l; c_i = c; tg_i = tg; b_i = b; fsm_i = fsm;$
        $total\_cov_i = total\_cov$
10:     $TCD \leftarrow l_m, c_m, tg_m, b_m, fsm_m, total\_cov_m$
11: end while
```

2) **Security asset Monitor (SAM)**: Security properties associated with DUV-specific security assets work as the standard of correctness or ground truth for that DUV. The DUV is equipped with multiple security-asset-aware cover statements to track violations of these properties and reveal security vulnerabilities during verification. The goal of our RL agent is to generate new test patterns that can invoke security breaches of these properties. During simulation, the cover statements are evaluated in tandem with the simulation flow, and any violations of these statements trigger warnings that are passed

to the scoreboard. The number of cover statements violated by a particular test pattern is also sent to the scoreboard as the SAM value, where:

$$SAM = \frac{(\#cover\ statement\ violated\ by\ t)}{(\#total\ cover\ statements)} * 100\%$$

For example, we have Advanced Encryption Standard (AES) algorithm implemented in benchmark AES-T1300 [10]. One of the security assets of this design is the cipher key. The associated security property is *"cipher key should not be leaked through cipher text"*. Listing 1 shows the cover statement tracking the violation of this property. Our RL agent aims to trigger more cover statement violations, achieving a higher SAM value by generating new test patterns.

3) **Rare node monitor (RNM)**: In graph theory, "rare nodes" are the vertices in a graph that are rarely visited or used. These nodes are critical for identifying influential nodes, detecting anomalies, and understanding network resilience. To ensure the security of the DUV, RL-TPG tries to generate tests that exercise the rare nodes as much as possible. In the case of RTL designs, rare nodes are the lines in the design that have not been activated during the simulation and the signals that have low switching activity. The RL agent aims to generate new test patterns to reach these rare nodes and trigger any potential security breaches denoted by SAM. To achieve this, we first identify initial rare nodes using a random constrained simulation of the DUV for thousands of cycles and keep the unreachable lines and low activity signals in a dictionary. Then, in each iteration, the RL environment monitors the percentile of rare nodes reached by each test pattern compared to the dictionary's total size, which is our RNM value passed to the scoreboard. RL agent's goal is to maximize the RNM score.

```
property key_leak;
  @(posedge clk) turn off iff (reset)
  (data_valid_in == 1 && cipherkey_valid_in
      == 1) |-> (cipher_key == cipher_text) ;
end property
key_leak_c: cover property (key_leak);
```

Listing 1: Cover statement for security asset monitor

• **Observation field**: The observation field is the encapsulated information sent to the RL agent from the environment at each iteration. The choice of observation field is vital as it determines the information available to the agent. This information helps the RL agent in the decision-making process. The environment maintains a scoreboard and tracks the score of each RL-generated test pattern, calculated as $score = (TCD + SAM + RNM)$. The observation field passes the test pattern (t) and its corresponding score, collected from the scoreboard, to the RL agent. The scoreboard also warns about potential security breaches indicated by SAM which will terminate the simulation loop.

• **Static analyzer**: Static analysis is a technique used to detect vulnerabilities in RTL designs without actual hardware simulation. It involves analyzing the design's structure, properties, and code to identify potential malicious components or anomalies. Our framework deploys a static analyzer to guide the RL agent to generate faster SAM violations. Our static analyzer includes design analysis and rule-based detection. Through design analysis, the analyzer extracts all input, output, and intermediate signals, while rule-based detection traces and tracks the places where these signals have been utilized. We keep a database of the extracted signals and their various values so we can use them to generate new test cases and assist the RL agent in initiating directed patterns.

Formal verification methods, such as symbolic execution [4], are powerful for verifying a program's behavior and detecting possible bugs. However, its unscalability and resource-intensive analysis can make it unsuitable for large designs. To overcome the obstacles of scalability and resource-intensive analysis posed by formal verification methods, we have developed a custom static analyzer and coupled it with a dynamic RL agent. This analyzer performs a pattern search to find all the input, output, and intermediate signals and stores all the signal's names in a dictionary. Then, it parses the DUV to find where each signal has been assigned or checked against a value. If some constant numbers are assigned or associated with these signals, they will be stored for further use in the action field's mutation mechanism.

• **Action field**: The action field refers to the actions available to an RL agent in a given environment at each iteration. Our framework uses a discrete set of actions, each representing a new way of generating test patterns. The action field is initialized in the environment with a different set of predefined action types. For RL-TPG, these actions are mutations of the previous test patterns. The agent receives a reward or penalty depending on the $score$ (collected from the observation field's scoreboard) of the previous test pattern and uses these rewards/penalties to select suitable mutation techniques.

1) **Reward/penalty mechanism**: In the RL process, rewards and penalties guide the agent's learning process. These rewards/penalties provide feedback to the agent about the efficiency of its actions, which influences its decision-making in subsequent interactions with the environment. The agent's ultimate goal is to learn how to maximize its expected cumulative reward while minimizing penalties. To achieve this, the agent explores and optimizes its actions, considering the balance between immediate rewards and long-term goals. The agent is rewarded for generating test patterns that increase (TCD) and a higher RNM and SAM score. The higher the TCD, SAM, and RNM score from the previous iteration, the higher the reward. On the contrary, if the agent fails to generate a higher score from the previous iteration, it receives a penalty. The lower the test pattern scores compared to previous test patterns, the higher the penalty it faces.

The RL agent utilizes a model-free RL algorithm such as Advantage Actor-Critic((A2C) and Proximal Policy Optimization(PPO). It then incorporates reward and penalty signals to update its policy and enhance its decision-making abilities. Its goal is to maximize rewards and minimize penalties. The agent tries to learn an efficient policy that maximizes $score$ or cost function while minimizing penalty.

2) **Mutation mechanism**: Based on the policy and the $reward/penalty$ mechanism, the RL agent mutates the previously generated test patterns to create new ones. The agent uses an initial random seed if no previous test pattern exists. Once the framework completes one iteration, it evaluates the $reward\_and\_penalty$. It then applies one mutation method from the diverse mutation methods available to generate a

new test pattern. The RL agent accounts for previous actions and their corresponding feedback in each iteration. The goal of the RL agent is to choose a mutation method from n different options that will maximize $(score, reward)$ and minimize $penalty$ for the mutated test pattern. As shown in the Algorithm 2, we have used different mutation methods as simple as bit-wise operations like XOR to flip bytes/bits, byte shift left/right, bit shift left/right, bit/byte swap on the current test pattern $(t_{current})$ to generate new test patterns (t_{new}). These are the dynamic mutation methods. We also use our static analyzer's dictionary to mutate $(t_{current})$ as the static mutation method. In each iteration of the RL framework, the agent chooses either the static mutation method or one of the dynamic mutation methods as its action. The mutation method the RL agent chooses depends on the magnitude of the reward/penalty for taking that action.

Algorithm 2 Mutation_method

```
Input: $current\ test\ pattern(t_{current})$,
    $constant\_value\_dictornary$
Output: $new\ test\ patterns(t_{new})$
 1: Initialization:
 2: $t_{current} \leftarrow random\_seed$
 3: Mutation for the rest of the simulation:
 4: while simulation do
 5:     mutation 1: $t_{new} \leftarrow xor(t_{current}, 1)$
 6:     mutation 2: $t_{new} \leftarrow byte\_swap(t_{current}, 0)$
 7:     mutation 3: $t_{new} \leftarrow bit\_flip(t_{current})$
 8:     ...
 9:     mutation n: $t_{new} \leftarrow insert(t_{current},$
    $static\_analysis\_dictornary)$
10: end while
```

Sometimes, security vulnerabilities can be triggered by a single input combinationally or after running the simulation for multiple clock cycles sequentially. For example, the hardware vulnerability embedded in the AES-T1300 benchmark is shown in Listing 2. Here, if the first eight input bits are equal to the value of 8'hCC, the cipher key would be leaked through the cipher text. The standard 128-bit long plain text input to the RL agent will take up to 2^{128} mutations to get this input pattern, and the probability of getting this very specific input pattern is $1/2^{128}$, which is computationally infeasible, given the modern technology. The goal of our static analyzer is to guide the RL in detecting this kind of stealthy bug that depends on particular trigger points of the code. The analyzer parses all the constant values and signals and stores them in a dictionary called $constant\_value\_dictornary$. For example, in AES-T1300, the value 8'hcc and the range [7:0] are stored in this dictionary, along with all the other constant values and signals of the DUV. In static mutation, the agent chooses to go through the dictionary of items one by one per iteration and uses each item to mutate the value of $t_{current}$ and gets t_{new}. This dictionary guides the RL to navigate throughout the whole DUV code space and thus increases the probability of generating the cover statements. This increases the SAM score. In this way, the static analyzer helps generate faster cover statement violations, which would unlikely be trigged using other verification methods.

```
always@(clk)
begin
   ...
  assign cipher_text = (plain_text[7:0] ==
      8'hCC) ? cipher_key : cipher_text_2;
  ...
end
```

Listing 2: Code snippet for Hardware Trojan in AES-T1300 benchmark

The above example uses the static mutation method to detect vulnerability through the $static\ analyzer$. However, not all vulnerabilities can be triggered by this method. Therefore, the dynamic mutation methods detect vulnerabilities the static analyzer fails to capture. For example, listing 3 shows a vulnerability listed in Common Weakness Enumeration(CWE) database[11]. In this design, the write operation should not occur when the $lock\_status$ is high. However, the design's vulnerability is that the $scan\_mode$ and $debug\_unlock$ can override the $lock\_status$. RL agents can use dynamic mutation on all the input signals to trigger this vulnerability. As the agent does not know the vulnerability, for each iteration of the framework, it chooses to mutate the test patterns from either one of the dynamic mutation methods or the dictionary of static mutation method. It then gets feedback for t_{new}, and depending on the magnitude of the reward/ penalty, it chooses the same or different mutation method for the next iteration. This is how the RL's learning process continues to generate better test patterns that uncover more unverified states and trigger probable security violations.

- **RL agent**: We used two model-free RL algorithms to build our RL agents. The advantage of using a model-free RL is that this flexible approach does not require an explicit model of the environment dynamics or knowledge of the design's behavior and responses, which works great with RTL designs. We used RL algorithms, such as PPO and A2C, because these algorithms work on both continuous and discrete data, and these data types are compatible with RTL data types. We compare these two algorithms in Section IV. After building the RL agent, we connect it with the observation and action fields. The agent is trained on the following objective function during the learning process: $objective\_function = max(score) + max(reward) + min(penalty)$

```
always @(posedge Clk or negedge resetn)
    ...
    else if (write & (~lock_status |
       scan_mode | debug_unlocked) )
    begin
    Data_out <= Data_in;
    end
    ...
endmodule
```

Listing 3: Code snippet for vulnerability of CWE-1234

III. Experimental setup

In this section, we present the experimental setup of the RL-TPG detection framework. We evaluated the effectiveness of our proposed test generation methodology using a diverse set of benchmark circuits that include externally triggered vulnerabilities that can be detected using logic testing. The

TABLE I: Benchmark, runtime and coverage results of RL-TPG approach

Benchmarks	**CWE-1234**		**RS232-T300**		**RSA-T100**		**RSA-T200**		**AES-T1300**		**AES-T500**		**Average (%)**
#Input Bit	20		17		96		96		256		256		
#Lines of Code	451		892		847		853		1258		1214		
Violation Type	Confidentiality		Integrity		DoS		Integrity		Confidentiality		DoS		
Vulnerability Behaviour	Leaks the input		Replace a bit of plain text		Leaks the cipher key		Disable encryption		Leaks the plaintext		Cipher text will not be updated		
Vulnerability Trigger	Debug overwritten		Counter		Single input		Single input		Single input		Counter		
Vulnerability Trigger Type	Combinational		Sequential		Combinational		Combinational		Combinational		Sequential		
Learning Method	PPO	A2C	PPO	A2C	PPO	A2C	PPO	A2C	PPO	A2C	PPO	A2C	
Line (%)	100	100	90	90	100	100	96.97	100	99.7	99.7	99.7	99.7	97.98
Condition (%)	100	100	NA	NA	100	100	100	100	94.59	94.59	94.59	94.59	97.83
Toggle (%)	92.74	92.74	79.54	77.64	82.42	79.52	70.52	79.52	99.47	99.47	99.47	99.47	89.53
Branch (%)	87.5	87.5	93.94	93.94	100	100	92.5	92.5	99.67	99.67	99.67	99.67	95.86
SAM (%)	100	100	100	100	50	50	33	33	75	75	75	75	66.6
RNM (%)	100	100	6.25	6.25	100	100	40	40	0	0	0	0	48
Iteration time (s)	6.5	6.5	24.9	35.1	32.3	17.2	44.2	40.3	361.9	444.1	597.5	372.4	192.29
#RL Iterations	3	3	15	21	19	9	26	24	121	148	199	124	67.6

experiments were conducted on a dedicated GPU server due to RL's learning process, and the detection performance was assessed and final coverages reported. The benchmarks are obtained from Trust-Hub and contain two cryptographic IPs: RSA and AES, one communication IP core RS232, and a CWE vulnerability obtained from their database. These IPs have been injected with vulnerability designs. For example, The benchmark RSA-T200 gets triggered when a specific pre-defined input plain text is fed to the RSA algorithm. This trigger then forces the private encryption key to become '1', disabling the encryption operation, integrity violation, and denial of service(DoS). Some details of the experimental benchmarks are provided in the first six rows of Table I.

The benchmarks are interfaced with a full SystemVerilog driver and monitor, so we have a test environment that encapsulates the design and its manually defined properties used for SAM evaluation. SystemVerilog test environment provides a useful way of introducing the RL-generated test patterns to the DUV. For simulation, VCS by Synopsys was used. All the scripting for gluing the RL model with DUV, the static analyzer, and the RL model are implemented in Python. For building the RL environment, stable-baseline3 was used.

IV. Results and Discussions

The RL agent populates our test pool with new test cases at each iteration. Iterations can be as many simulation cycles as the user needs. The RL-TPG framework exits from the continuous learning loop whenever one of the following conditions occurs: one is when there is a violation of the cover statement, implying asset leakage or obvious anomalies detected through $SAMs$. Second is when the rareness coverage increases and more corner cases are explored after a user-defined time limit. Table I shows the final number of iterations when one of the two exit criteria gets exercised.

- **Effects of benchmark size:** It is clear from the Figure 3a 3b that as the input size increases, the number of iterations and iteration time to reach cover statement violation increases. Table I shows that the runtime also correlates with the size of the DUV's code (represented by #lines of code) in the same way.
- **Effects of RL algorithm:** Different algorithms for agent training were considered. Table 1 shows that A2C works better with RSA(more efficient, lower iteration time). However, PPO worked better overall, especially with bigger AES benchmarks.

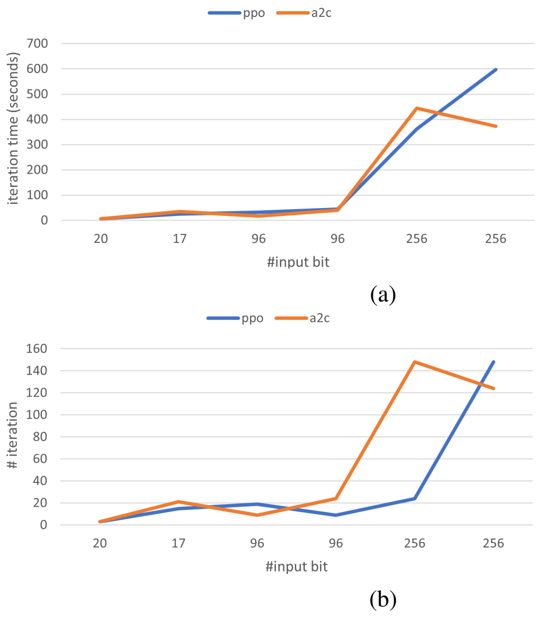


Fig. 3: (a) Iteration time(seconds) vs #input bit (b) #iteration vs #input bit.

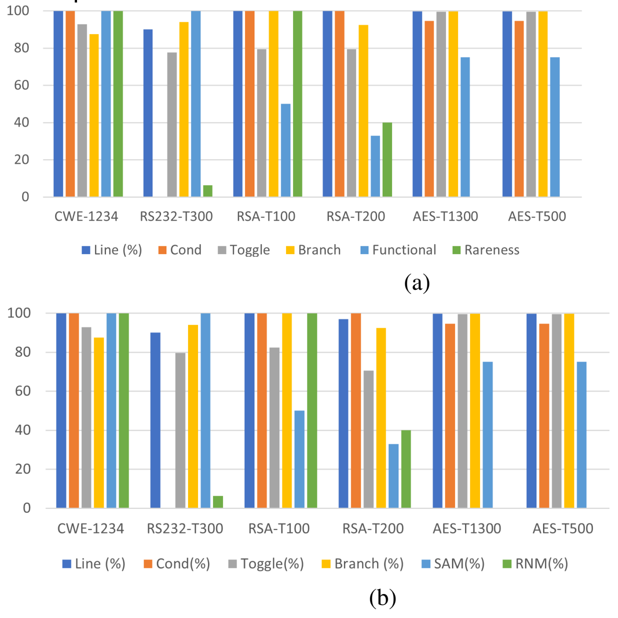


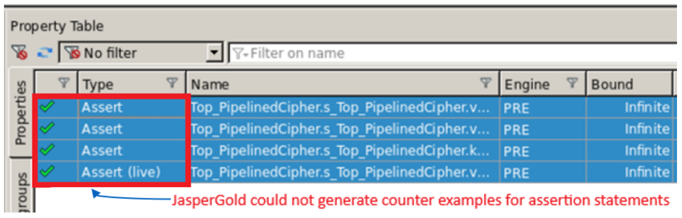
Fig. 4: Coverage using (a) A2C algorithm (b) PPO algorithm.

As different algorithms work differently on different benchmarks, we can experiment with building a hybrid of the two algorithms in the future.

- **Performance of traditional coverages:** RL-TPG worked really well to get an average of 90% TCD in an average time of 192s. This proves that this framework can be utilized for security and functional verification. Figure 4a, 4b shows that among various coverages, line and condition coverages are the easiest to increase, and $toggle$ and RNM are the hardest. Toggle coverage is slightly lower due to the hardness of flipping some of the deeper signals in each design. RNM stayed at zero for some benchmarks, even at the end, because of the diffusion between the functional SAM coverage and RNM coverage. For this reason, we will put more weight on rewards/penalties on test patterns that increase the RNM score in the future.
- **Performance of security asset aware coverage:** The average score for RNM was 48% and the average score for SAM was 66.6%. This means full and partial security leakages were found effectively through this framework.
- **Scalability and reusability:** For verifying each benchmark, we only replaced the DUV part of Figure 1. The whole framework works as a plug-and-play security verification method for the DUVs, proving the high scalability and reusability of the novel framework.
- **Comparison with JasperGold:** We compared our framework's cover statement tracking capability and security violation evoking test pattern generation capabilities with the state-of-the-art formal verification tool JasperGold by Cadence. We found that the cover statements (e.g., Listing 1) we used to track security violations for AES-T300 were unreachable by JasperGold. For generating security violation-evoking test patterns (known as a counterexample), JasperGold uses the negation of the cover statement. However, JasperGold could not find the counterexamples for the bigger benchmarks. For example, for the negation of the cover statement (also known as an assertion) in Listing 1 for AES-T1300, Figure 5a shows that the assertion was not violated. Hence, JasperGold could not produce a counterexample. However, Figure 5b shows that our novel framework could not only track the cover statement but also generate a test case that violates that statement. This proves that formal tools like JasperGold can generate false counterexamples. However, if we remove the reset and clock signal from the Listing 1 and specify input signals, JasperGold can generate counterexamples. However, generating counterexamples in this way is meaningless as the clock and rest signal influence many parts of the DUV, and vulnerability is unknown at the verification state.

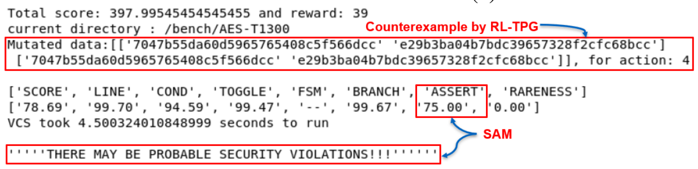
V. Conclusion and Future Work

Traditional verification methods for pre-silicon designs struggle with increasing complexity and time-to-market demands, resulting in unverified designs and security vulnerabilities. Our innovative framework combines hardware security verification with Reinforcement Learning (RL) to create intelligent test patterns that target security flaws. This approach improves verification coverage, identifies potential security issues, and enhances the automation of pre-silicon verification. Future work involves testing the framework at other abstraction levels and finally making it work at the system and device levels. The approach may evolve from a white box to a gray or black box approach. Additionally, we will consider different RL models and algorithms to address complex system challenges and explore automated security cover statement generation and test pattern generation.


Property Table
No filter
Filter on name
Properties
groups
JasperGold could not generate counter examples for assertion statements

	Type	Name	Engine	Bound
✓	Assert	Top_PipelinedCipher.s_Top_PipelinedCipher.v...	PRE	Infinite
✓	Assert	Top_PipelinedCipher.s_Top_PipelinedCipher.v...	PRE	Infinite
✓	Assert	Top_PipelinedCipher.s_Top_PipelinedCipher.k...	PRE	Infinite
✓	Assert (live)	Top_PipelinedCipher.s_Top_PipelinedCipher.v...	PRE	Infinite

(a)



```
Total score: 397.99545454545455 and reward: 39
current directory : /bench/AES-T1300
Mutated data:[['7047b55da60d5965765408c5f566dcc' 'e29b3ba04b7bdc39657328f2cfc68bcc']
 ['7047b55da60d5965765408c5f566dcc' 'e29b3ba04b7bdc39657328f2cfc68bcc']], for action: 4

['SCORE', 'LINE', 'COND', 'TOGGLE', 'FSM', 'BRANCH', 'ASSERT', 'RARENESS']
['78.69', '99.70', '94.59', '99.47', '--', '99.67', '75.00', '0.00']
VCS took 4.500324010848999 seconds to run

'''''THERE MAY BE PROBABLE SECURITY VIOLATIONS!!!''''''
```

(b)

Fig. 5: Security verification using (a) JasperGold (b) RL-TPG.

References

[1] Brendan Davis. "The economics of automatic testing". In: (1994).

[2] Farimah Farahmandi, Yuanwen Huang, and Prabhat Mishra. "Formal approaches to hardware trust verification". In: *The Hardware Trojan War: Attacks, Myths, and Defenses* (2018), pp. 183–202.

[3] W. Hu *et al.* "Hardware information flow tracking". In: *ACM CSUR* 54.4 (2021), pp. 1–39.

[4] Arash Vafaei et al. "SymbA: Symbolic Execution at C-level for Hardware Trojan Activation". In: *2021 IEEE International Test Conference (ITC)*. 2021.

[5] Kimia Zamiri Azar et al. "Fuzz, penetration, and ai testing for soc security verification: Challenges and solutions". In: *Cryptology ePrint Archive* (2022).

[6] Zhixin Pan and Prabhat Mishra. "Automated Test Generation for Hardware Trojan Detection using Reinforcement Learning". In: *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2021.

[7] Huili Chen et al. "AdaTest: Reinforcement Learning and Adaptive Sampling for On-Chip Hardware Trojan Detection". In: (2023).

[8] Yuxi Li. "Deep reinforcement learning: An overview". In: *arXiv preprint arXiv:1701.07274* (2017).

[9] Rajat Subhra Chakraborty et al. "MERO: A Statistical Approach for Hardware Trojan Detection". In: *Cryptographic Hardware and Embedded Systems - CHES 2009*. Ed. by Christophe Clavier and Kris Gaj. 2009.

[10] K Sudeendra Kumar et al. "An improved AES Hardware Trojan benchmark to validate Trojan detection schemes in an ASIC design flow". In: *2015 19th International Symposium on VLSI Design and Test*. 2015.

[11] *CWE - CWE-1234: Hardware Internal or Debug Modes Allow Override of Locks (4.11) — cwe.mitre.org*. https://cwe.mitre.org/data/definitions/1234.html.