Re-Pen: Reinforcement Learning-Enforced Penetration Testing for SoC Security Verification

Hasan Al Shaikh, Shuvagata Saha, Kimia Zamiri Azar, Farimah Farahmandi, Mark Tehranipoor, *Fellow, IEEE*, and Fahim Rahman

***Abstract*— Due to the increasingly complex interaction between the tightly integrated components, reuse of various untrustworthy third-party IPs (3PIPs), and security-unaware design practices, there have been a rising number of reports of system-on-chip (SoC) hardware (HW) vulnerabilities that compromise the security of critical assets. SoC security verification, therefore, is an indispensable part of the verification effort. The existing hardware verification methodologies either presuppose white-box knowledge or scale poorly with increasing design complexity. Hardware penetration testing (pentest) is an emerging gray-box security verification methodology at the register-transfer level (RTL) that is applicable across a wide variety of threat models and addresses many shortcomings of the existing methodologies. In this work, we propose Re-Pen, a novel hardware pentest framework that requires minimal gray-box information from the design specification to achieve significantly better security vulnerability (SV) detection performance than state-of-the-art pentest techniques. At the core of this framework lies a mutation engine that combines the strengths of reinforcement learning (RL) and binary particle swarm optimization (BPSO) in its test pattern mutation strategy to generate intelligent test patterns without manual supervision. This framework significantly reduces the requirement for detailed, manual, expertise-driven adaptations specific to the SoC under test. Through extensive experiments conducted on multiple SoCs, we demonstrate that Re-Pen can reduce vulnerability detection time by up to 3× and achieve a markedly improved consistency compared with the state of the art. Furthermore, Re-Pen was able to detect native security bugs in an open-source SoC. It successfully identified a scenario where, despite a functionally correct hardware implementation, a mistake in the architectural specification allowed privilege escalation from the software layer.**

***Index Terms*— Hardware (HW) pentest, particle swarm optimization (PSO), reinforcement learning (RL), system-on-chip (SoC) security verification.**

I. Introduction

SYSTEM-ON-CHIPS (SoCs) have become increasingly popular in various applications, including consumer electronics, autonomous driving, military vehicles and systems, IoT, and embedded devices due to their versatility and capability to provide high computing performance while meeting tight area and power constraints. In fact, according to forecasts, the global SoC market will reach a valuation of more than \$300 billion by 2031 [1]. To satisfy the ever-increasing demand, the design and development time of these devices have kept shrinking dramatically over the past few decades [2]. The verification of SoCs, the most time-consuming stage of the design effort, often takes up to 70%–80% of the total development time [3]. Therefore, verification methodologies must be continuously improved in an effort to keep up with the design complexity and shorten verification time.

Recently, security verification of SoCs has become a pressing concern in light of numerous attacks reported in media and academia [4], [5]. Developing a comprehensive SoC security verification solution is highly challenging due to the following reasons: 1) integration of third-party intellectual property (3PIP) cores and globalization of the supply chain, which have inadvertently introduced untrustworthy entities in the lifecycle; 2) rapidly increasing design complexity; 3) security-unaware design practices and electronic design automation (EDA) tools; 4) vulnerabilities arising from the complex interaction of software–firmware–hardware (HW) layers detection of which require extensive and intelligent exploration of the input state space; and 5) the ever-expanding threat surface [6]. There are several types of security verification methodologies for SoCs that are currently prevalent in both academia and industry, including formal verification [7], [8], [9], concolic testing [10], [11], and machine learning (ML) methods [12]. However, these are inadequate in addressing the aforementioned challenges as they do the following: 1) require significant manual expertise and intervention; 2) can suffer from scalability issues due to path and state explosion problems [13]; 3) assume the availability of white-box knowledge, i.e., detailed knowledge about the structural implementation of the design; 4) can be susceptible to reporting false positives [14]; and 5) sometimes be applicable to only limited threat models.

In recent years, fuzz testing has started to gain traction as a scalable method of SoC security verification [15], [16], [17]. Many of these approaches rely on a golden reference model of a processor, typically a software instruction set simulator (ISS), to compare against and thereby identify anomalous behavior of the device under test (DUT). Such a golden model is not easily definable for SoCs due to the presence of 3PIPs, which need to be modeled as gray-/black-box entities (i.e., limited/no knowledge about the structural implementation, respectively) to properly approximate real-world scenario [18].

Recently, gray-box hardware penetration test (pentest) frameworks for SoC security verification have been proposed

Received 21 May 2024; revised 21 September 2024 and 31 October 2024; accepted 19 November 2024. Date of publication 27 December 2024; date of current version 26 February 2025. *(Corresponding author: Hasan Al Shaikh.)*

Hasan Al Shaikh, Shuvagata Saha, Farimah Farahmandi, Mark Tehranipoor, and Fahim Rahman are with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: hasanalshaikh@ufl.edu; sh.saha@ufl.edu; farimah@ece.ufl.edu; tehranipoor@ece.ufl.edu; fahimrahman@ece.ufl.edu).

Kimia Zamiri Azar is with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816 USA (e-mail: kimia.zamiriazar@ucf.edu).

Digital Object Identifier 10.1109/TVLSI.2024.3510682

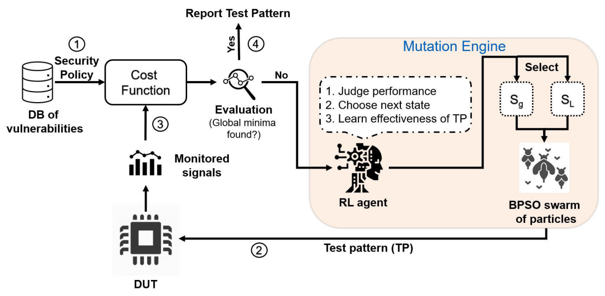


Fig. 1. Proposed RL-enforced hardware penetration testing framework: Re-Pen.

at the register-transfer level (RTL) to solve these limitations [19], [20]. While RTL pentesting draws inspiration from the software domain, it exhibits some noteworthy differences from established practices in that field. In contrast to software development, where penetration testing is performed on already developed software infrastructure [21], RTL pentesting verifies hardware not yet ready for deployment. Postponing penetration testing until the hardware is fully prepared for deployment carries substantial risks, especially the potentially expensive prospect a silicon re-spin if vulnerabilities are detected at such a late stage. Consequently, it has been recommended that hardware pentesting in the presilicon phase should involve a strategic application of test patterns through user-accessible points (software, external debug pins, and interfaces) to propagate the effect of vulnerabilities to an observable point, thereby facilitating early detection [19]. These approaches convert SoC security policies (step ① in Fig. 1) to *cost functions*, which are constructed, such that they evaluate to the global minima in the presence of a vulnerability. Test patterns are applied to the DUT (step ②) and continually mutated with evaluated cost function feedback from signal monitoring (step ③). The mutation is driven by an algorithm that we refer to as the mutation engine, and its objective is to find the global minima in the shortest possible time in the presence of a vulnerability. If the global minima is found, the vulnerability and the corresponding test pattern are reported (step ④).

The framework proposed in [20] used binary particle swarm optimization (BPSO), a binary input space compatible variant of the evolutionary method particle swarm optimization (PSO), to generate these mutations. PSO class of algorithms uses a group of "particles" that iteratively mutate their velocities and positions to find the input vector that minimizes the cost function. The group or collection of these particles, for a particular optimization session, are termed the "swarm." There are two attractive features of PSO class of algorithms for gray-box hardware verification.

1) These algorithms do not require any labeled data (i.e., no training phase) to perform optimization of cost functions. This is especially beneficial, as obtaining ground-truth behaviors for hardware in SoC verification is challenging due to the involvement of 3PIPs.
2) These algorithms have been demonstrably effective in gray- and black-box testing across diverse domains [22], [23], [24].

In a hardware verification context, the positions of these particles represent the value of the test vector, while velocity is a measure of the amount of mutation applied to the test vector for the next iteration. In typical BPSO algorithm, the velocity and position updates occur through a single equation regardless of how individual particles perform. However, it has been demonstrated that the BPSO algorithm can be susceptible to poor convergence performance due to its singular velocity and position update equation for all particles [25].

Contributions: With the above limitations in mind, we introduce Re-Pen, a novel, scalable hardware penetration testing framework shown in Fig. 1. The principal contributions are as follows.

1) We introduce and formalize Re-Pen, a dynamic RTL penetration testing framework facilitating the incorporation of artificial intelligence, specifically reinforcement learning (RL), into its mutation engine (highlighted by a yellow box in Fig. 1). We term the algorithm driving the mutation engine RL-enforced BPSO (RLBPSO). Within the mutation engine, the RL agents choose a mutation strategy for each particle in the BPSO swarm, dynamically selecting different update equations for them instead of using the same one as in the traditional BPSO. They select from sets of candidate global and local search operations ($\mathcal{S}_g$ and $\mathcal{S}_l$, respectively, in Fig. 1), based on each particle's performance toward achieving the global optima.
   The proposed framework does *not* require a training phase with large amounts labeled data (see Section IV for more details) and mutates test pattern through the real-time (i.e., execution time) feedback gathered from DUT.
2) We formulate and demonstrate the generalizability of cost functions for Re-Pen across different SoCs with varied ISAs, IPs, and microarchitectures.
3) We also demonstrate the compatibility of Re-Pen with both RTL simulation and FPGA prototyping *without the use of a golden model*. Note that unlike FPGA-based verification approaches in [26] and [27], Re-Pen is not leveraging *FPGA accelerated simulation*, rather is capable of operating directly on an FPGA prototype of the SoC, inclusive of its software environment.
4) We also demonstrate the efficacy of Re-Pen in detecting security vulnerabilities (SVs) across different SoC test platforms. In the best case, Re-Pen improves the state-of-the-art hardware pentest performance by 3×.
5) In one test platform, *Re-Pen* detected a native SV where privilege escalation from lower privilege software was possible. Furthermore, it detected another bug where lower privilege software could not access all permitted memory regions due to an implementation mistake in hardware.

The rest of this article is organized as follows. Section II surveys and compares previous works in this domain and provides some background information related to the work. Section III provides an overview of hardware pentesting with a focus on limitations of the mutation engine proposed in the prior art. Section IV presents the Re-Pen framework with a focus on the mechanisms of its mutation engine, and Section V discusses implementation. In Section VI, we examine experimental results and conclude the study in Section VII.

II. Preliminaries

A. Prior Work

The following are two pertinent categories of works in the literature that are related to this article: 1) RTL security verification methodologies of SoCs and 2) methods that have been employed to improve the convergence performance of BPSO. We will present an analytical discussion for each of them in the following.

1) Security Verification of SoC at the RT Level: Formal verification, the most widely used SoC security verification methodology in the industry, involves the specification of expertly crafted formal properties to verify security policies of the design [7], [8], [9], [28]. Formal methods are confronted with the challenge of state explosion when dealing with the complexity of modern SoC designs [14], [29]. They also require comprehensive white-box knowledge, and the properties need to be changed from design to design to account for microarchitectural and timing differences.

Symbolic execution and concolic testing have received widespread attention in the hardware security literature in recent years for vulnerability detection at the RTL stage. The method proposed by Zhang et al. [10] was demonstrated to be successful in detecting only processor vulnerabilities and not those that might exist in an SoC with numerous peripherals. Ahmed et al. [11] proposed to detect hardware Trojans through concolic testing. Again, this method might not be scalable to other vulnerabilities, as it relies on targeting suspicious branches in RTL code to increase coverage. For other types of vulnerabilities, there may not be a straightforward distinction between suspicious and nonsuspicious branches. Furthermore, for complex SoCs, the no. of branches in RTL code can easily number in the thousands, which can cause concolic-based testing to suffer from path explosion problems [30].

Although ML methods have shown remarkable scalability in many domains, in the hardware security domain, the objective/reward/loss function utilized to guide their training often makes sense only within the limited threat model the authors considered and cannot necessarily be generalized to the broader threat surface. For instance, Hasegawa et al. [12] utilized neural networks to verify the existence of hardware Trojans. Hasegawa et al. [12] include rare node coverage as a metric in its cost function to train the neural network. In addition to the fact that this method is only applicable to one specific type of threat vector, other types of vulnerabilities also do not necessarily correlate with the notion of rare nodes.

In recent years, hardware fuzzing or fuzz testing has emerged as a dynamic method of SoC security verification. The verification speed of all fuzzing methodologies largely depends on the feedback metric used to guide the mutation process. Tyagi et al. [17] use several functional coverages, such as finite state machine (FSM), branch, condition coverage, and so on, as feedback metrics. These metrics are computed with full white-box knowledge of the design, unlike Re-Pen, which utilizes gray-box specifications and security requirements. Furthermore, it utilizes a golden processor model (i.e., the ISS Spike), which is not readily attainable for SoCs with multiple 3PIPs. The same reliance on golden models is present in several other fuzzing approaches as well [15], [16], [32]. Gray-box fuzz testing methodologies have been proposed in [26] and [27]. They utilize the notion of "mux control coverage" and toggling of control registers for feedback metrics, respectively. The DUT is instrumented with additional logic for each cover point. Obviously, this introduces a tremendous instrumentation burden, which can limit its scalability for very large and complex SoCs [18]. Hossain et al. [33], [34] also introduced gray-box fuzzing techniques that use metrics, such as output activity, behavior, input randomness, and coverage as feedback metric to guide the fuzzing process without the need for a golden model. However, the authors report the applicability of the framework only for a single SoC. Notably, they utilize the Linux kernel to run the fuzzer natively on the SoC under test. There are many SoCs (e.g., those used in resource-constrained embedded/IoT applications), which may not be able to boot the Linux kernel natively *or* require significant modifications to boot it [35]. The authors do not discuss how the framework can be generalized in such cases.

Very recently, large language models (LLMs) and RL have been used for hardware security verification. Rostami et al. [36] combine white-box fuzzing with LLMs and RL to achieve much faster coverage than contemporary processor fuzzers, such as [17] and [32]. However, this method also requires a golden model to detect security bugs, which, as we mentioned before, is difficult to define for 3PIP cores. Next, LLMs are used for assertion generation in security verification in [37]. However, the method is not shown to be scalable to SoC-level complex designs. Saha et al. [38] propose different prompting strategies for detecting SVs in SoC. However, the proposed method requires full access (white box) to DUT's RTL, which may not be available due to the presence of 3PIPs. Nahar Mondol et al. [39] propose RL-TPG that uses functional coverage metric-based reward function for RL agents to identify SVs in HW. Similar to [37], this method is not shown to be scalable to SoC-level complex designs. There are several other works that utilize RL with focused singular threat models, such as hardware Trojans [31], [40], [41], side-channel vulnerability assessment [42], logic locking strength assessment [43], and so on. Consequently, these works are not scalable to the wide threat surface (hardware Trojans, access control, information leakage, hardware–software interactions exposing cross-layer exploits, and so on) that Re-Pen is able to handle.

The work in [20] proposed and formalized hardware penetration testing as an alternative approach for SoC security verification by addressing these shortcomings. It does not rely on white-box coverage metrics, and it does not require a golden reference model. However, its convergence speed is suboptimal, given its use of conventional BPSO. Table I presents a summary of the comparison among these approaches.

2) Improving BPSO: BPSO can suffer from limitations of the algorithm that have been explored extensively in the literature [44], [45], [46], [47] (see Section IV-A for illustrated discussion). There have been multiple studies that report improvement in the performance of BPSO and PSO by introducing the notion of global search and local search operations, which, unlike the conventional variants, assign different update velocity and position update equations depending upon the state and performance of the particles [47], [48], [49]. On the other hand, the integration of RL with various PSO techniques has been shown in multiple studies to enhance their effectiveness. The utilization of RL was implemented in [50] to optimize the parameter tuning of the PSO algorithm. It was also employed autonomously in conjunction with PSO in [51] to augment the estimation of the objective function. To the best of our knowledge, there has been no prior integration of RL with the BPSO technique, as is the case with our proposed methodology. Furthermore, according to the no-free lunch theorem [52], there is no one optimization algorithm that outperforms all others for all classes of problems. The integration of RL and BPSO algorithms is yet to be investigated in the problem of gray-box SoC security verification, which we demonstrate through this work.

TABLE I

Comparison of the Proposed and Established SoC Security Verification Methods

Method	Model	Automation	Scalability	Remarks
Formal methods [7]–[9]	White box	Low	Low	May be susceptible to state explosion problems and may report false positives.
Concolic testing [10], [11]	White box	Moderate	Low	Can suffer from path explosion, applicable to Trojan vulnerabilities only.
AI & ML based [12], [31]	White box	Moderate	High	Utilized metrics in objective/loss/reward function not adaptable to other threat models.
Fuzzing [17], [26]	White/Gray box	High	High	Reliance on golden models limit applicability to SoCs with 3PIPs.
Re-Pen	Gray box	High	High	Tailored for fast gray-box verification of SoCs, even in the presence of 3PIPs, without needing a golden model. Applicable to a wide variety of threat models, including Trojans, information leakage, access control, and Denial-of-Service (DoS).

B. Security Assets and Policies in SoC

Security assets are information stored in a device that, if exposed to a malicious entity or unauthorized party, can cause severe economic and credibility loss for the manufacturer [53]. On the other hand, security policies specify the confidentiality, integrity, and availability requirements of security assets [54]. For example, in an advanced encryption standard (AES) core, the secret key is a security-critical asset. The security policies should dictate that malicious entities cannot extract the key through the data bus or other interfaces. Other security assets include sensitive user data saved in the device, device configuration, the entropy of a true random number generator (TRNG), and so on. Examples of detailed security policies can be found in [19] and [55]. Other than leakage through erroneous hardware implementation, a malicious entity can also extract the assets through hardware Trojans, side channel leakage, security-unaware CAD optimizations, and so on [56], [57]. A device's security policies should ideally encompass all possible threat models in light of adversarial capabilities.

C. Reinforcement Learning

RL is a subfield of ML that involves an agent learning to make decisions by interacting with its environment [58]. Unlike supervised learning, RL does not require a training phase with large amounts of labeled data where the ground truth is known beforehand. Instead, the agent receives feedback as rewards or penalties, as it traverses a state space, and its goal is to maximize its cumulative reward over time through trial and error without explicit instruction or labeled data. Many RL schemes are available, among which Q-learning is especially suitable for discrete state spaces [59]. It is a model-free, off-policy learning algorithm, which means the transition probability of the states of the environment does not need to be specified. Also, the agent predicts the environment's response to progression. In this method, a Q-table that stores the expected rewards of different actions is initialized with 0s or other random values. Then, for each state, the best actions and their associated rewards and Q-scores are calculated (equation for calculation discussed in Section II-C) and populated in the Q-table. These steps are continued until the optimal Q-scores have been achieved. A detailed formal treatment of the Q-learning model utilized in Re-Pen is available in Section IV.

III. Re-Pen: High-Level Overview

In this section, we provide a thorough overview of the proposed hardware penetration testing methodology and its

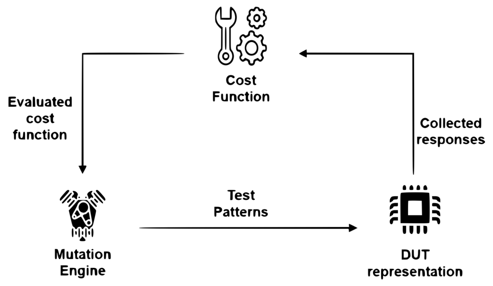


Fig. 2. High-level overview of hardware penetration testing.

different components. We also outline the scope of improvement in this section by discussing the reasons for the suboptimal performance of the previous framework.

From a high level, Re-Pen consists of three main components (see Fig. 2).

1) *Cost Function:* The principal proposition of a hardware pentest is as follows: general security policies, specified from leveraging gray-box knowledge about the DUT, associated with an SoC can be translated to mathematical cost functions [20]. These cost functions can be formulated in such a way that they would evaluate to the global minima in the presence of a vulnerability. The general form of these cost functions is

$$\mathbb{F} = \sum_{i=1}^{N} g(p) \tag{1}$$

where $g : \mathbb{P} \rightarrow f$ is a function that maps a security policy p in $\mathbb{P}$, the set of N security policies of the device, to some real-valued function f. f takes as input the set of observed or monitored hardware signals that are potentially relevant to the vulnerability being tested for. To illustrate this conversion process with an example, let us consider the scenario where an untrusted 3PIP cryptographic core (e.g., Rivest–Shamir–Adleman (RSA) or AES IP) shares a common bus with multiple other IPs in an SoC. One or more of these other IPs may be untrusted. Since the crypto IP itself is untrusted, there is a possibility of it containing an information leaking/DoS Trojan. In such a scenario, the confidentiality of the security asset, the key, must be preserved from other IPs. Furthermore, it must be made certain that the encryption is completed in a timely manner, such that the SoC does not experience interrupted service. Without assuming *any* structural knowledge about the crypto IP, other IPs, or the SoC itself, we can specify the following policies corresponding to these requirements: a) the secret asset/key (K) should not leak to the read ($\mathrm{rd}_{\mathrm{data}}$)/write ($w_{\mathrm{data}}$) channel of the shared bus; b) the key should not be available at memory locations accessible by untrusted IPs (M_{data}); and c) the handshake signal ($h_{o,\text{clk}}$) of the crypto IP, signifying the end of encryption, should be asserted within a certain no. of clock cycles. Under the threat model considered, these three constitute the set $\mathbb{P}$. For the sake of concise representation, let us define a piecewise function, $T(x) = \sum x$, that evaluates to 0 whenever any one of its arguments is 0. The corresponding cost function is

$$\mathbb{F}_1 = T\left(\sum_{j=1}^{N_m} \text{HD}\left(\text{rd}_{\text{data},j}, K\right) + \text{HD}\left(h_{o,\text{clk}}, 0\right) + \sum_{j=1}^{N_m} \text{HD}\left(w_{\text{data},j}, K\right) + \sum_{j=1}^{N_k} \text{HD}\left(M_{\text{data},j}, K\right)\right) \quad (2)$$

where N_m: number of IPs sharing the common bus, N_k: number of memory locations accessible by untrusted IPs, and HD: hamming distance. This cost function will be evaluated using responses collected from the DUT.

Note that if there is a violation of any of the security policies, one of the terms will evaluate to 0, which means $\mathbb{F}_1$ will be 0. Furthermore, the gray-box knowledge does not assume any microarchitectural knowledge about either the SoC itself or the crypto IP. As such, the cost function can be applied to detect vulnerabilities, exactly as-is, for different SoCs regardless of the instruction set architecture (ISA) or the number/type of IPs present as we demonstrate through our experimentations in Section VI.

2) *DUT Representation:* The DUT itself may be represented by an equivalent software model obtained from using a simulator, such as Verilator [60] (in this case, called simulation framework), or by directly prototyping it on an FPGA (emulation framework). The response collection mechanism varies depending on whether the simulation or emulation variant is being implemented.

3) *Mutation Engine:* Test patterns generated from a mutation engine (algorithm) are continually applied to the DUT in an attempt to drive relevant hardware signals toward values that constitute a violation of security policies. Corresponding to the applied test patterns, the values of the monitored signals are fed back to the mutation engine to evaluate the fitness of the generated test patterns.

It should be noted that the verification speed achieved by applying hardware penetration testing largely depends on the effectiveness of the mutation engine. Both pentest frameworks in [19] and [20] utilized BPSO as the mutation engine. The framework shown in Fig. 1 significantly improves the mutation engine and, thus, verification speed, by supplementing the BPSO algorithm with RL agents. We term this novel algorithm RLBPSO, which will be elaborated on in Section IV.

IV. Proposed Algorithm for Mutation Engine: RLBPSO

This section presents the RLBPSO algorithm, the principal driver of the mutation engine in Re-Pen, and how it overcomes the limitations of prior mutation engines used in RTL hardware pentest frameworks.

A. Binary Particle Swarm Optimization and Its Limits

The natural phenomenon of "local adaptation based on global experience" serves as inspiration for PSO and related evolutionary optimization algorithms. The standard PSO considers a D-dimensional feature space where the position of the ith particle can be described by $\boldsymbol{x_i} = [x_{i1}, x_{i2}, x_{i3}, \ldots, x_{iD}]$. Here, $x_{i1}, x_{i2}, x_{i3}, \ldots, x_{iD}$ are real-valued quantities. Each particle, in the context of hardware penetration testing, is essentially an input test pattern. For example, if the applied input pattern is an RISC-V assembly program, each 32-bit instruction can be considered a particle. N such particles, collectively called the "swarm," can be represented by $\boldsymbol{X} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}, \ldots, \boldsymbol{x_N}]$. Each particle goes through a predefined number of iterations ($\text{it}_{\max}$), in each of which the position of the particle is updated according to the equations

$$v_{i,j}(\text{it}+1) = wv_{i,j}(\text{it}) + \phi_1 U_1\left(p_{\text{best}i} - X_{i,j}(\text{it})\right) + \phi_2 U_2\left(g_{\text{best}} - X_{i,j}(\text{it})\right) \quad (3)$$

$$X_{i,j}(\text{it}+1) = X_{i,j}(\text{it}) + v_{i,j}(\text{it}+1). \quad (4)$$

U_1, U_2: uniformly distributed random numbers between 0 and 1, ϕ_1, ϕ_2: acceleration coefficients, and w: positive inertia constant. $v_{i,j}(\text{it})$ is the velocity of the particle i of the swarm at iteration it at index j. The $p_{\text{best}i}$ and g_{best} represent the positions that achieved the best performance for the ith particle and the global best performance of the entire swarm, respectively. For BPSO, introduced in [61], all of the above equations apply with the following clamping done on the position of the particles after they have been calculated using (4):

$$X_{i,j}(\text{it}+1) = \begin{cases} 0, & \text{if rand}() \geq \xi\left(v_{i,j}(\text{it}+1)\right) \\ 1, & \text{otherwise} \end{cases} \quad (5)$$

where $\xi(x)$: the sigmoid function.

A great advantage of using BPSO (or its derivatives) lies in the nature of SoC circuitry. SoC circuitry, especially CPUs, are binary in nature. Hence, vast majority of typical hardware inputs (e.g., reset signals, processor instructions, joint test action group (JTAG) input signals, and so on) can be modeled by a binary representation. Thus, all such inputs can be directly mutated by BPSO.

However, note that there is one update equation for updating velocity and position (see (3) and (4), respectively) for *all* particles regardless of how good or bad they are performing. This can directly lead to two unintended consequences.

L1: All particles are influenced by the global best. Therefore, they may explore the pockets around the region of global best attained (so far) instead of the full state space.

L2: Well-performing particles that are approaching the minima, might get ejected out of the 'valley' that contains the global minima as there is no concept of refined local search wherein only subvectors of the particle are mutated.

RL can overcome these inefficiencies and, thus, streamline the mutation generation process.

B. RLBPSO Algorithm

Unlike traditional BPSO, RLBPSO will allow particles to choose from numerous states (or operations) that correspond to different velocity and position update equations. Particles transition states to explore and exploit the search space. Corresponding to the limitations described in Section IV-A, we divide these states or operations into two major categories: global and local search operations. RLBPSO uses four global search operations, namely, exploration (E), convergence (C), high jump (H), and low jump (L), and one local search operation, namely, fine search ($\mathcal{FS}$) as candidate states for RLBPSO particles.

Role of RL: The challenge resides in selecting the proper update equation in a context-aware manner, as there are multiple candidate operations that the particles could perform. More specifically, while global search operations address limitation L1 and local search operations address L2 mentioned in Section IV-A, it is still a difficult optimization decision to determine the optimum sequence of these operations when finding the global minima for a cost function. Rather than basing this decision on a predefined or anticipated statistical distribution, we employ the artificial intelligence of RL agents. Specifically, we employ one RL agent for each particle in the BPSO swarm. The RL agent associated with each particle receives a reward based on that particle's performance.

1) Formalism of Reinforcement Learning: At the beginning of the learning process, an RL agent can take an action $a \in \mathcal{A}$ that changes its state $s \in \mathcal{S}$ over some unfamiliar environment $\mathbb{E}$. Over time, the goal of the agent is to maximize the reward function $\mathcal{R}$ provided to it. Note that the specification of an RL problem can be completely defined by identifying the set of possible states $\mathcal{S}$, the set of actions $\mathcal{A}$, and the reward function $\mathcal{R}$. In the RLBPSO algorithm, an RL agent is associated with each BPSO particle. For each RL agent, the following hold.

1) $\mathcal{S}$*:* The states can be combined set of global and local search operations, i.e., $\mathcal{S} = \{E, C, H, L, \mathcal{FS}\}$. Each element in $\mathcal{S}$ is associated with separate velocity and position update equations.
2) $\mathcal{A}$*:* The actions are those, which can switch from one element in $\mathcal{S}$ to another. In the case of RLBPSO, the way to achieve this is to change the update equation. Therefore, the number of elements in $\mathcal{S}$ is the same as that in $\mathcal{A}$.
3) $\mathcal{R}$*:* A simple scheme is followed to give reward to the RL agent, which is represented mathematically as follows:

$$\mathcal{R} = \begin{cases} 1, & \text{if } \mathbb{F} \text{ is reduced from prev. it} \\ -1, & \text{otherwise.} \end{cases} \tag{6}$$

Since the state space is discrete for the RL agents, Q-learning is an appropriate RL scheme. In Q-learning, the Q-score associated with each RL agent is updated following the below equation and stored in the Q-table:

$$Q_{\text{it}+1}(\mathcal{S}_{\text{it}}, \mathcal{A}_{\text{it}}) = \text{lr}\left[d \max_{\mathcal{A}} Q(\mathcal{S}_{\text{it}+1}, \mathcal{A}) + (r_{\text{it}+1})\right] + (1 - \text{lr})Q(\mathcal{S}_{\text{it}}, \mathcal{A}_{\text{it}}) \tag{7}$$

where $Q(\mathcal{S}, \mathcal{A})$: the Q-score associated with action $\mathcal{A}$ when the state of the particle is $\mathcal{S}$, lr: learning rate, d: discount factor, and r_{it} is the reward to agent at iteration it. It is trivial to note that the Q-table of each agent is a 5×5 matrix holding all Q-scores for all. Furthermore, in the proposed algorithm, the reward for an action can vary based on the sequence of previous actions. More specifically, the reward for a given action depends on the cost function value from the previous iteration, which is influenced by the particle's state at that time. This means that identical actions may produce different rewards depending on the particles' prior trajectory and agents' prior actions. This sequential dependency highlights the need for a more sophisticated RL approach than simpler RL approaches, such as multiarmed bandits, to handle the temporal dynamics effectively [62].

Note that for each verification session (i.e., for each cost function constructed for the DUT), the RL agents' parameters (i.e., the Q-table entries) are initialized anew. As the BPSO swarm explores the search space represented by the cost function, Q-table entries are dynamically updated, which, in turn, dictates the agents' decision of the operation to select. Furthermore, note that there is no requirement for large amounts of labeled data for this dynamic updating to happen. The dynamic decisions will be made utilizing execution time feedback gathered from the DUT representation.

2) Formalism of Global Search Operations: In RLBPSO, we define the global search operations as those operations that update the *entire* position and velocity vectors of the particle. In updating the entire vectors, particles have a movement component along every dimension of the state space.

1) *Exploration and Convergence Operations:* The idea behind exploration (E) is that the acceleration coefficient parameters ϕ_1 and ϕ_2, and the inertia parameter w in (3) can be tuned to search for a large number of optima. On the other hand, these parameters can also be tuned to let the swarm converge faster to global optima in which case it will be called the convergence (C) operation. It has been reported that allowing particles to transition freely between these two states at all iterations of the algorithm is better than confining them to the early and late stages [45]. For this work, the values of these parameters are set as $(\phi_1, \phi_2, w) = (2.5, 0.5, 0.9)$ for E and $(\phi_1, \phi_2, w) = (0.5, 2.5, 0.4)$ for C following the recommendation of [45], [63], and [64].
2) *High and Low Jump Operations:* We introduce high (H) and low (L) jump operations for binary spaces where the primary objective is to allow the particles to escape from potential local optima. While performing a high or low jump, velocities of the particles are not updated; instead, their positions ($\boldsymbol{X_i}$) are updated directly via the equation

$$\boldsymbol{X_i} = p_{\text{best},i} + \text{rand}_n(X_{\max} - X_{\min}) \tag{8}$$

where rand_n: random number drawn from the normal distribution and $X_{\max}$, $X_{\min}$: the maximum and minimum possible values of the input binary vector. If the normal distribution has a std. deviation of 0.9 and 0.1, the particles will perform a high and low jump, respectively.

3) Formalism of the Local Search Operation: In RLBPSO, we define the notion of fine search ($\mathcal{FS}$) inspired by Ji et al. [65], which mutates only a subvector $\tilde{\boldsymbol{X}}_i$ consisting of consecutive highly correlated elements of $\boldsymbol{X_i}$. To put it formally, if $\boldsymbol{X_i}$ is of length L and m is some number of highly correlated consecutive elements, $\tilde{\boldsymbol{X}}_i$ is defined by the equation, $\boldsymbol{X_i} = [x_1, x_2, \ldots, x_{L-m}] \frown \tilde{\boldsymbol{X}}_i$, where $\frown$ denotes the concatenation operation. To give an example of a highly correlated subvector, consider the case where $\boldsymbol{X_i}$ represents the binary encoding of RISC-V instructions. The first 6 bits together represent the opcode, while the other bits are data fields or encode special information about the instruction. Hence, the first 6 bits of each instruction are greatly correlated, which is the same case for the data fields of respective instructions, and so on. The associated velocity of the subvector is $\tilde{\boldsymbol{V}}_i$. The

Algorithm 1 RLBPSO Algorithm

```
Input: $\mathbb{F}$, $it_{max}$, $\theta$, $lr$, $it_{fs}$
Output: $g_{best}$
1  Initialize a swarm with $N$ particles, each with random
     initial position $\boldsymbol{X_i}$ and random velocity $v_i$;
2  while $i \ngtr N$ do /*Iterate over the swarm*/
3      $\boldsymbol{F_i} \leftarrow \mathbb{F}(X_i)$; // compute fitness for
         each particle
4      $p_{besti} \leftarrow \boldsymbol{X_i}$, $\mathcal{S}_i \leftarrow E$;
5      $Q(\mathcal{S}_i, :) \leftarrow 0$;
6  end
7  $g_{best} \leftarrow min(\boldsymbol{p_{best}})$;
8  while $it \ngtr it_{max}$ and $it > 1$ do
9      while $i \ngtr N$ do
10         $\boldsymbol{F_{i,it}} \leftarrow \mathbb{F}(X_i)$
11         if Fitness improves by $\theta$ then $\mathcal{A}_i \leftarrow \mathcal{FS}$ ;
12         else $\mathcal{A}_i \leftarrow \max_{\mathcal{A}}(Q(\mathcal{S}_i, \mathcal{A}))$ ;
13         switch $\mathcal{A}_i$ do
14             case $E$ or $C$ or $H$ or $L$ do
15                 Update $\boldsymbol{V_i}$ and $\boldsymbol{X_i}$ using appropriate
                     equations and parameters
16             end
17             otherwise do
18                 Update $\tilde{\boldsymbol{X}}_i$ and associated $\tilde{\boldsymbol{V}}_i$;
19                 if $\boldsymbol{F_{i,it+1}} \geq \boldsymbol{F_{i,it}}$ then
20                     Continue selecting $FS$ for at least
                         $it_{fs}$ iterations;
21             end
22         end
23         $\mathcal{S}_{i_{it+1}} \leftarrow \mathcal{A}_i$;
24         Get maximum Q value for $\mathcal{S}_{i_{it+1}}$;
25         Update Q-table entry using equation 7;
26         $\mathcal{S}_{i_{it}} \leftarrow \mathcal{S}_{i_{it+1}}$;
27         Clamp $\boldsymbol{X_i}$ using equation 5;
28         if $\boldsymbol{F_{i,it}} < \mathbb{F}(p_{besti})$ then $p_{besti} \leftarrow \boldsymbol{X_i}$ ;
29         if $\boldsymbol{F_{i,it}} < \mathbb{F}(g_{best})$ then $g_{best} \leftarrow \boldsymbol{X_i}$ ;
30     end
31 end
```

velocity update of the itth iteration is calculated as follows:

$$\tilde{\boldsymbol{V}}_{\boldsymbol{i},\mathbf{it}} = \left(\tilde{V}_{i,\text{it}-1} + \text{rand}_n\right)/k \tag{9}$$

where k: decaying factor (> 1). The position is updated by the equation: $\tilde{\boldsymbol{X}}_{\boldsymbol{i},\mathbf{it}} = \tilde{\boldsymbol{V}}_{\boldsymbol{i},\mathbf{it}} + \tilde{\boldsymbol{X}}_{\boldsymbol{i},\mathbf{it-1}}$. Note that in cases where highly correlated subvectors are not readily apparent, this operation may be skipped.

C. Algorithmic Description of RLBPSO

The overall RLBPSO is presented in the pseudocode form in Algorithm 1. In lines 1–7, the initialization phase is performed. N particles are initialized, each with their own randomized position, and Q-table rows are all set to 0, and the state of all particles is set to exploration. In subsequent iterations, for each particle, at first, the fitness is evaluated (lines 8–10). The agent chooses FS only when a particle has managed to reduce the cost function between consecutive iterations by a predefined factor θ. This allows the algorithm to do refined local searches around promising optima. Otherwise, the agent chooses the action that maximizes its reward (lines 11 and 12). Subsequently, corresponding to the action chosen, the position and velocity of the particle are updated (lines 13–18).

If the chosen operation is FS, the agent is incentivized to keep selecting FS for at it_{fs} iterations (lines 19 and 20). This is because it might happen that during the first few local searches around the optima, the performance does not improve. In such a case, the algorithm should keep reducing the velocity to do finer searches around the optima [through (9)]. From lines 23–26, the Q-learning algorithm and update equations are implemented. Using line 27, we clamp the $\boldsymbol{X_i}$ to a binary space. The last two lines update p_{best} and g_{best}. Our experiments use $\text{it}_{\text{fs}} = \text{it}_{\text{max}}/10$ & $\theta = 0.8$. The algorithmic complexity for the given algorithm is $O(N \times \text{it}_{\text{max}})$. This is because the worst case computation is determined by the loops in lines 9–31. Here, the algorithm initializes a swarm of N particles and then enters a loop that runs for it_{max} iterations. Within each iteration, it processes each particle individually, performing constant-time operations, such as fitness evaluations, updates, and assignments. Since the inner loop over particles runs N times per iteration and there are it_{max} iterations, the nested loop gives us the time complexity $O(N \times \text{it}_{\text{max}})$.

V. Implementation of the Re-Pen Framework

In this section, we will elaborate on how we implemented the Re-Pen framework and our experimental setup.

A. Test Platforms

In order to assess the applicability of our proposed methodology across different SoCs, we have chosen two testing platforms for experimentation. The two test platforms under consideration are as follows: 1) the Ariane RISC-V SoC; 2) a MicroBlaze (MBlz)-based SoC; and 3) Neorv32 SoC [66].

Ariane is an open-source SoC containing a six-stage, single-issue central processing unit that utilizes the 64-bit RISC-V instruction set [67]. The system incorporates M, S, and U privilege levels to facilitate the execution of a Unix operating system. We customized the open-source Ariane SoC by integrating AES and RSA IP in addition to the IPs already present.

Our second test platform SoC contains the MBlz soft-core processor that utilizes a DLX ISA [68]. For the creation of this test platform, we integrated the following IPs as peripherals: AES, RSA, ROM, General Purpose Input Output (GPIO), Universal Asynchronous Receiver/Transmitter (UART), and an Advanced eXtensible Interface (AXI) controller.

Our third test platform SoC is an RISC-V Harvard architecture embedded SoC that has two privilege levels: M-mode and U-mode. The platform has a collection of security-relevant features, such as a physical memory protection (PMP) unit and a TRNG IP. The PMP unit is a hardware feature that enables access control of physical memory regions. It is configured, at runtime, through a set of control and status registers (CSRs) (`pmpcfg` and `pmpaddr`) by M-mode software (typically, the boot code or start-up code) to give read (i.e., load), write (store), or execute permissions to lower privilege (in this case, U-mode) software, which, by default, has none. Note that if U-mode software attempts to load/store from/to memory regions where M-mode did not grant it read/write permission, the RISC-V spec mandates the raising of a synchronous exception called load/store access faults, respectively. Similarly, if U-mode software attempts to fetch instruction from a memory region where it does not have execution access, the hardware should raise an instruction access fault. Notably, this SoC also contains a direct memory access (DMA) controller that can transfer data between any two memory locations, without involving the processor, to improve latency in data transfer.

TABLE II
VULNERABILITIES PRESENT IN SOC TEST PLATFORMS

Index	**Description**	**Mutated input**	**Monitored output**	**Ref.**
SV1, SV2, SV3	A Trojan in AES (SV1) and RSA (SV3) leaks the secret key across the common bus or slows encryption (SV2) when a specified plaintext pattern is used.	Plaintext pattern in a baremetal C program	Shared bus, memory, handshake signals	AES T-1000, T-11000 [69]
SV4, SV5	Illegal CSR read/write operation (SV4) or WFI instruction (SV5) execution allowed in lower privilege mode	RISC-V instructions	*mstatus* register, illegal instruction exception	CWE-1256 [70]
SV6	Implementation error in password check logic in debug module	JTAG data register values	Password check signal	CWE-1191 [70]

We inserted the SVs shown in Table II in the first two SoCs to create a set of synthetic benchmarks. Each of these benchmarks, therefore, has underlying vulnerabilities that constitute various security requirement violations. All of these vulnerabilities have been based on open-source vulnerability databases. For instance, SV1, SV2, and SV3 are the implementations of TrustHub Trojan benchmarks [69]. SV1 and SV3 are Trojans inserted in the AES and RSA crypto IP, respectively, that cause confidentiality violations by leaking the secret key through the common bus shared by all IPs. This secret key could then be leaked directly to the outside world through the UART terminal. SV2 is also caused by a Trojan that inflicts denial of service in the system by delaying the assertion of the handshake signal at the end of encryption.

The CWE database [70] lists several hardware weaknesses described at a high level. For example, CWE-1256 stands for "Improper Restriction of Software Interfaces to Hardware Features" but does not contain any implementation example. SV4 and SV5 represent the implementation of these high-level weaknesses as applicable to the RISC-V-based SoCs. RISC-V ISA describes the privilege levels that are used to properly isolate the software stack during execution, and SV4/SV5 is related to these specifications. In particular, an attacker may leverage SV4 to trigger invalid or major page faults in the CPU by illegally writing to a CSR and SV5 to cause an unauthorized processor halt from the software layer. By leveraging SV6, which is caused by improper implementation of password check logic in the debug module, an attacker can bypass the password authentication mechanism. The latter three vulnerabilities are inserted only in the Ariane SoC.

We did not insert any vulnerabilities in the Neorv32 benchmark. We used the M-mode software to give read/write permissions to U-mode to all IPs and memory regions in the system, barring the regions where M-mode software executes (i.e., a small portion of the instruction memory) and the TRNG IP. U-mode software was also given execution access to the rest of the instruction memory. In effect, this created the scenario where all user-level IPs (UART, GPIO, and so on) are accessible to lower privilege software, but IPs that might be security critical (such as TRNG) are closed off to it.

B. Implementing the Re-Pen Emulation and Simulation Frameworks

For the simulation framework, the open-source simulator Verilator [60] was used to build the software model. We developed a custom VCD parser to parse through the VCD file generated from the simulation and collect the values of monitored signals. In the emulation framework, monitoring of the Ariane and Neorv32 SoCs's relevant registers and memory locations was gained via a JTAG debugging software (e.g., OpenOCD and UrJTAG). The FPGA board for this and the Neorv32 platform was the Digilent Genesys 2. For the MBlz SoC, Xilinx ILA IP cores were used to insert observation points in the design [71]. Xilinx Vitis was utilized to run baremetal code on the platform. This SoC was emulated on the Digilent Basys 3 FPGA board.

C. Cost Functions Utilized

Here, we briefly describe the gray-box cost functions used for detecting the SVs. These cost functions are utilized by turn to detect the present vulnerabilities in the test platforms. To construct these cost functions, it is assumed that the prerequisites are met beforehand, originally presented in [20], which also applies to Re-Pen. For the sake of brevity, we do not present these prerequisites here but refer the reader to that work.

$\mathbb{F}_1(SV1, SV2, SV3)$*:* The security policies associated with these vulnerabilities have already been discussed in Section III. We reiterate that these security policies do *not* assume any implementation details about the SoC ISA or microarchitecture (gray box). Hence, the cost function is the same as that in (2) across both test platforms.

$\mathbb{F}_2(SV4, SV5)$*:* The associated security policies are taken directly from RISC-V privileged architecture specification [72]. In supervisor mode, the following hold: 1) if the trap virtual memory (TVM) bit in `mstatus` CSR is set, an illegal instruction exception should be raised when trying to write to or read from `satp` register and 2) if the timeout wait (TW) bit in `mstatus` is set, an illegal instruction exception should be flagged when trying to execute the wait for interrupt (WFI) instruction. The cost function can be written as follows[1]:

$$\mathbb{F}_2 = \mathrm{HD}(\mathrm{Il}_{\mathrm{exc}}, 1) + T\left(\sum_{b\epsilon C} \mathrm{HD}(b, 1)\right) + T\left(\sum_{k\epsilon W_c} \mathrm{HD}(\mathrm{Ins}_2, k)\right) + \mathrm{HD}(\mathrm{Ins}_{3\mathrm{rd/rs}}, 0\times 180) + T\left(\sum_{m\epsilon W_i} \mathrm{HD}(\mathrm{Ins}_3, m)\right) + \mathrm{HD}(\mathrm{Ins}_{2\mathrm{rd}}, 0\times 300) \quad (10)$$

where $\mathrm{Il}_{\mathrm{exc}}$: the value of illegal instruction exception, Ins_x: binary representation of assembly instruction x, $\mathrm{Ins}_{x\mathrm{rd/rs}}$: binary encoding of destination and source registers of instruction x, W_l: set of load immediate instructions, C: set of configuration bits (e.g., TVM and TW) of the `mstatus` register, W_c: set of CSR read/write instructions, and W_i: set of candidate instructions (e.g., WFI). Note that this function will also evaluate to global minima of 0 when a vulnerability is present in *any* RISC-V processor-based SoC (gray box).

$\mathbb{F}_3(SV6)$*:* The security policies regarding a debug module with a password checking mechanism can be stated as follows: the password check signal (p_{chk}) should be asserted: 1) only

[1] 0×300 and 0×180 are the hex encodings of `mstatus` and `satp` registers.

after checking for the entire password; 2) at both read and write time; and 3) even after reset. The cost function is

$$\mathbb{F}_3 = \mathrm{HD}\left(p_{\text{chk\_rst}}, 0\right) + \mathrm{HD}\left(p_{\text{chk\_rd}}, 0\right) + \mathrm{HD}\left(p_{\text{chk\_wr}}, 0\right) + \mathrm{HD}\left(p_{\text{sub}}, 0\right) \tag{11}$$

where $p_{\text{chk\_}x}$: value of p_{chk} signal during or after x operation. p_{sub}: the value of p_{chk} after only a substring (e.g., 10 of 32 bits) of the correct password is applied.

$\mathbb{F}_4$*:* Given Neorv32 test platform's security features and setup as explained in Section V-A, we can formulate the following security policies: 1) load/store access fault exception should *not* be raised when U-mode software accesses permitted regions; 2) load/store access fault exception *should* be raised when U-mode software attempts to access restricted regions; 3) instruction access fault exception *should* be raised when U-mode software tries to execute/fetch instructions from execution restricted memory regions; and 4) U-mode should not be able to use DMA to transfer data from load/store restricted memory region to permitted region. We note that the last policy is not directly mentioned in the RISC-V specification but can be readily inferred from interpreting CWEs (namely, CWE-1189 and CWE-1274) and CVEs (CVE-2018-15383). If this policy is violated, memory protections that should be guaranteed by PMP setup would be bypassed, and U-mode software could escalate its privileges (similar to CVE-2024-21823, CVE-2008-4279, and so on). Corresponding to these policies, the constructed cost function was

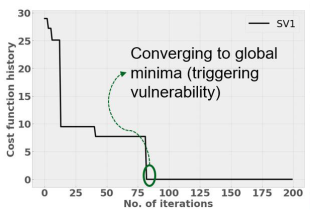
$$\mathbb{F}_4 = T\left(\sum_{p\_\text{reg}} \mathrm{HD}\left(\mathrm{ER}_{p\_\text{reg}}, 1\right)\right) + T\left(\sum_{r\_\text{reg}} \mathrm{HD}\left(\mathrm{ER}_{r\_\text{reg}}, 0\right)\right) + T\left(\sum_{r\_\text{ex\_reg}} \mathrm{HD}\left(\mathrm{ER}_{r\_\text{ex\_reg}}, 0\right)\right) + \mathrm{HD}\left(\mathrm{pmem}_d, \mathrm{rmem}_d\right) \tag{12}$$

where ER: exception raised, p_reg: load/store permitted memory regions, r_reg: load/store restricted memory regions, r_ex_reg: execution restricted memory regions, pmem$_d$: value of load/store permitted memory region *before* a DMA transfer has been requested from U-mode, and rmem$_d$: value of load/store restricted memory region *after* a DMA transfer has been completed from U-mode.

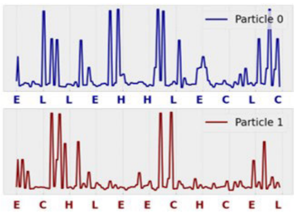
VI. Results and Analysis

We modified the open-source *pyswarms* library to implement RLBPSO [73]. For SV1–SV3 and SV6, since there were no highly correlated subvectors, the FS operation was not included in $\mathcal{S}$. For SV4 and SV5, certain subfields (e.g., bits 7–11 are r_d register subfield, and bits 15–19 are r_s register subfield) within the 32-bit instruction field are considered as the highly correlated subvector(s). For comparison of performance speedup achieved by Re-Pen, we compare it with the framework in [20], since, to the best of our knowledge, it is the only gray-box hardware pentest framework in the literature to report results on SoC security verification.

Re-Pen successfully triggered all of the inserted vulnerabilities in both test platforms. Fig. 3(a) illustrates the swarm cost history when detecting SV1 vulnerability in Ariane SoC. It can be observed that the swarm gradually converges to the minima, i.e., finds the vulnerability triggering pattern. Fig. 3(b) shows the experience of two individual particles in the swarm,

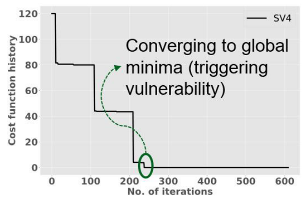


(a)

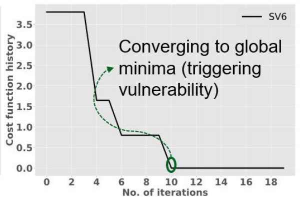


(b)

Fig. 3. History of cost function and particle states in detecting SV1 in Ariane SoC. (a) History of the swarm. (b) History of two individual particles and their states.

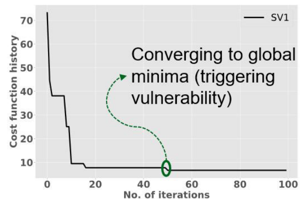


(a)

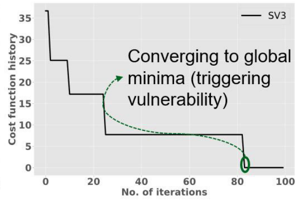


(b)

Fig. 4. History of cost function in detecting (a) SV4 and (b) SV6 in Ariane SoC.



(a)



(b)

Fig. 5. History of cost function in detecting (a) SV1 and (b) SV3 in MBlz SoC.

and on the bottom, we show the states of these two particles (every 20th iteration), as the RL agents try to find the minima through controlling these particles. We see that both particles start at the E state. However, both of them automatically get switched to other states by the associated RL agents. Although each particle jumps in and around the global minima, the RL agents of the swarm learn to hit the global minima after around 80 iterations. We show the swarm cost history obtained by applying Re-Pen on the Ariane SoC test platform with SV4 and SV6 vulnerability in Fig. 4. For SV4 and SV5, the mutated pattern is the assembly code instructions. Fig. 5 shows the same for detecting SV1 and SV3 vulnerability in the MBlz-based SoC. In both Figs. 6 and 7, we observe how Re-Pen is able to gradually and quickly converge to the global minima and, thereby, find the trigger test pattern to the vulnerabilities.

Re-Pen provides the verification engineer with information regarding the trigger condition(s), which are the mutated test patterns that, when applied to the DUT, caused the cost function to reach global minima. For example, upon mutating the set of RISC-V instructions identified in Section V-C, the SV5 trigger code snippet reported by Re-Pen is shown in Listing 1.

A. Discovering Native Vulnerabilities and Bugs

Recall from Section V-A that we had not inserted any vulnerabilities in the Neorv32 system. Starting from the fundamental principle that a system must follow security policies to ensure compliance with specifications and prevent violations of security requirements, we developed $\mathbb{F}_4$ for this test platform. Rather than targeting specific vulnerabilities,

```
main:
    jal ra, switch_to_sprvsr
    wfi
switch_to_sprvsr:
    /*setup_CSR sets the TW bit*/
    csrrw x0, mepc, ra
    li a1, 0x200888
    csrrw x0, mstatus, a1
    mret
```

Listing 1: Code snippet to trigger **SV5**.

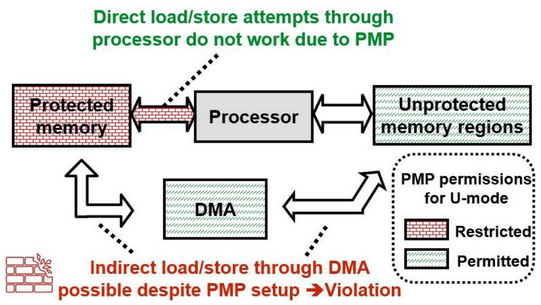

Direct load/store attempts through processor do not work due to PMP
Protected memory
Processor
Unprotected memory regions
DMA
PMP permissions for U-mode
Restricted
Permitted
Indirect load/store through DMA possible despite PMP setup ➔Violation

Fig. 6. U-mode view of system in Neorv32. A vulnerability was found where U-mode could escalate privileges through leveraging DMA.

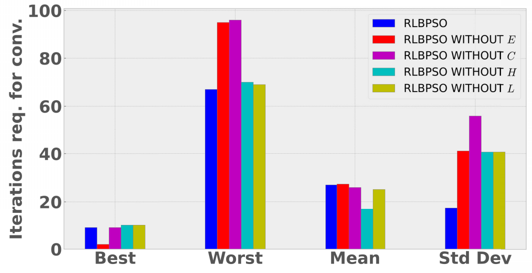


Fig. 7. No. of iterations required by Re-Pen in finding minima measured across different initializations. The plot shown is for detecting SV1 in Ariane.

we formulated broad policies that could potentially be violated due to inherent bugs or oversights, threatening the confidentiality, integrity, or availability of assets.

Neorv32 is still under active development, but there have been stable releases. We tested a few of the stable releases with Re-Pen optimizing for the cost function. Re-Pen discovered a critical security vulnerability in all tested releases of Neorv32.

Native Vulnerability (NV1): Re-Pen was asked to mutate a U-mode software. Re-Pen discovered that it was possible to successfully complete a transfer of data from a permitted region to a restricted region (i.e., the instruction memory reserved for M-mode) from the U-mode. The discovery was indicated by the fourth term in $\mathbb{F}_4$ going to 0. The vulnerability is graphically illustrated in Fig. 6.

On closer investigation of the specification documents, we noticed that the DMA IP performed *all* transactions with elevated M-mode privileges regardless of the mode of the software issued the request. As a result, it was possible to successfully complete a transfer of data from a permitted region to any restricted region from the U-mode. Also, note that since the U-mode could write or read to any protected memory region, escalation of privilege happens by it gaining access to memory of security-critical IPs and memory locations. We emphasize that there was not any implementation error/bug in hardware or software. Rather, during architectural specification, the system designer made an *oversight* in not realizing that having DMA perform transactions in elevated privilege would allow U-mode to bypass PMP restrictions.

Native Bug (NB1): Re-Pen detected that it was not possible for the U-mode to read/write from/to all permitted memory regions despite having proper access permissions from M-mode. This was detected when the first term went to 0 in optimizing for $\mathbb{F}_4$. It was possible for the U-mode software to have execution access to permitted memory regions that were implemented in the physical instruction and data memory. However, if U-mode attempted to load/store access other regions (UART and GPIO) outside them, an exception was being raised *despite* having proper permissions from M-mode. This is an implementation error or security bug, as the test platform was not behaving as specified.

Note that although four policies were included in the cost function, only two were violated. Thus, from a high level, to verify any new design, we suggest creating a set of cost functions from a stipulated list of policies and then using those cost functions, by turn, in Re-Pen for vulnerability detection.

B. Impact of Initialization on Performance

All PSO algorithms begin with initial randomization (lines 1–6 in Algorithm 1). By coincidence, the initialization may place one particle very near the global minima, allowing the swarm to find it rapidly. The opposite may also happen. Thus, to objectively compare the RLBPSO mutation engine in Re-Pen to the BPSO mutation engine [20], we should use several different initial random seeds. In light of this, we introduce four distinct performance measures: the best and worst case performance in converging to the minima is the least and most number of iterations required, respectively; the mean is the average number of iterations required, and the standard deviation (denoted by σ) is the number of iterations across a number of initializations. σ essentially measures the consistency in performance, since it denotes the deviation from the mean despite the variability in randomizations.

We compare the performance of Re-Pen against the BPSO-based approach by Al-Shaikh et al. [20] in Table III. The no. of iterations required in converging to the global minima is reported in Table III when detecting vulnerabilities in both the test platforms across 50 different initial random seeds. As mentioned previously, best and worst case values reported in Table III represent the least and most no. of iterations required by the framework, while mean and σ measure the average and consistency in performance across these 50 initializations. Obviously, the lower the number of iterations required, the faster the framework will trigger the vulnerability. It can be observed that across both SoC test platforms, RLBPSO hits the global minima more consistently (indicated by the lower value of σ) in a lower number of iterations. The average no. of iterations required also remains lower for Re-Pen. For some cases, the initialization of particles may be particularly favorable for BPSO, in which case, the best-case performance is better (for example, in detecting SV2 in Ariane SoC). This does not necessarily mean RLBPSO is performing worse, since verification engineers cannot rely on having extremely favorable initializations.

We also compared the execution time of the entire Re-Pen framework against that of [20]. We show the average speedup, as measured in seconds, gained by Re-Pen in detecting SV1 and SV2 in Ariane SoC in Table IV. As can be seen, Re-Pen improves the SV1 vulnerability detection speed in Ariane SoC by almost 3×. Note further that the average improvement in detection speed (measured in seconds) is roughly equal to the corresponding improvement in mean minimum iterations, as reported in Table III. Hence, Table III

TABLE III
ITERATIONS REQUIRED BY RE-PEN AND AL-SHAIKH et al. [20] FOR CONVERGENCE

PM	SV1				SV2				SV3				SV4		SV6	
	Ariane		MBlz		Ariane		MBlz		Ariane		MBlz		Ariane		Ariane	
	[20]	Re-Pen	[20]	Re-Pen	[20]	Re-Pen	[20]	Re-Pen	[20]	Re-Pen	[20]	Re-Pen	[20]	Re-Pen	[20]	Re-Pen
Best	32	**21**	**24**	32	**17**	21	20	**19**	17	**16**	**22**	25	2588	**761**	10	**4**
Worst	224	**195**	278	**250**	**230**	267	251	**245**	212	**197**	218	**192**	5681	**2295**	31	**11**
Mean	120	**41.4**	111.2	**37.2**	127.2	**47.3**	131.6	**42.9**	109.5	**61.1**	111.8	**66.3**	3450.67	**1511.27**	21.83	**8.47**
σ	45.6	**13.3**	38.7	**20.9**	51.2	**15.3**	45.8	**16.6**	47.3	**12.6**	57.2	**15.1**	870.23	**554.24**	6.31	**3.72**

PM: Performance Measure; SV4-SV6 not applicable for MBlz platform, results for SV5 shown in Table V

TABLE IV
DETECTION SPEED FOR RE-PEN AND [20] FOR ARIANE SoC

Index	Framework	[20]	Re-Pen	Speedup
SV1	Simulation	3050-3150 s	1030-1120 s	∼ 3x
	Emulation	180-190 s	65-75 s	∼ 2.8x
SV2	Simulation	3000-3150 s	1040-1080 s	∼ 2.9x
	Emulation	190-200 s	75-80s s	∼ 2.5x

TABLE V
MINIMUM NO. OF ITERATIONS REQUIRED IN SV5 DETECTION

Method	**Best**	**Worst**	**Mean**	σ
[20]	2123	5352	3150.28	916.9
Re-Pen without $\mathcal{FS}$	872	2803	1259.57	836.52
Re-Pen with $\mathcal{FS}$	652	1716	949.41	596.12

can also be regarded as a measure of vulnerability detection speed improvement.

C. Impact of Global and Local Search Operations

To measure the significance of each global and local search operation in RLBPSO, we engaged Re-Pen on the same test platforms without each operation. The results are shown in the form of a bar chart in Fig. 7 when detecting SV1 in the Ariane SoC. Again, the lower the no. of iterations, the better the performance. We can observe that when E was not included in detecting SV1 (as shown in Fig. 7), the mean no. of iterations required increased slightly, which indicates a slightly worse average performance. Interestingly, the best-case performance improves by a good margin. This is expected, since the E operation propels the particles to explore the state space extensively. As such, even if the random initialization of the particles is favorable, due to this initial propelling, the particle may miss the minima, which results in a deterioration in the best-case performance. However, note that the standard deviation of iterations required also increased significantly, implying that the consistency decreased significantly. Table V shows the impact of including $\mathcal{FS}$ in Re-Pen to achieve faster convergence in detecting SV5. It can be seen that without $\mathcal{FS}$, the performance of Re-Pen in detecting SV5 deteriorates across every metric.

The C operation is the most essential in extracting good performance from the algorithm. However, as shown in Fig. 7, the most consistent performance is achieved when all operations are present. Therefore, having all members in $\mathcal{S}$ present is optimal for Re-Pen.

D. Comparison With Fuzz Testing Methods

Hardware fuzz testing is an emerging verification method gaining significant traction in processor and SoC verification domain. We briefly discussed the limitations of a few hardware fuzzing techniques in Section II. We will now conduct a more thorough comparison with various hardware fuzzing methods, especially with regards to their capability in finding vulnerabilities that Re-Pen was able to detect.

A host of hardware fuzzers have been proposed for fuzzing processors. This includes works like PSOfuzz [75], TheHuzz [17], MABFuzz [74], MorFuzz [76], Cascade [77], and HyPFuzz [32]. Most of them are white-box fuzzing techniques. They were not designed to verify SoCs, which present a more complex verification challenge than processors due to the presence of 3PIP peripherals, associated software, and the difficulty of defining a reliable golden model. For example, SV1–SV3 are Trojan vulnerabilities that exclusively originate within peripherals (AES and RSA) of the processor. As these methods' threat models do not include hardware Trojans in peripherals of the processor, these are not detectable by them. Similarly, SV6 is a vulnerability that originates in the peripheral debug module. Furthermore, note that NV1 arises from an oversight in the specification of the DMA. The hardware and software were behaving correctly as specified, but the vulnerability results from an oversight in architecture specification. The vulnerability is also a result of complex interaction of the DMA, processor's PMP unit, peripherals, and associated software stack with the system memory. Again, none of these methods are shown to be able to detect such processor-external vulnerabilities that arise from an interaction of the processor, the peripheral, and the software stack. On the other hand, SV4, SV5, and NB1 are processor internal vulnerabilities and, hence, are detectable by these methods.

There are other fuzzers that have been proposed, which are equally applicable to target IP and SoC vulnerabilities. Such methods include RFUZZ [26], Difuzzrtl [79], the work in [78], SoCFuzzer [33], and TaintFuzzer [34]. Although the first three are applicable to detecting all of the vulnerabilities presented here, there are some implementation challenges when it comes to their generalizability and adaptibility. For example, Trippel et al. [78] convert the RTL to an equivalent software model, which is then fuzzed by a software fuzzer. However, to be able to apply upstream conversions required by the software fuzzer, the method requires the DUTs to be communicating exclusively using the TL-UL bus protocol. None of the presented DUTs in our study utilize this protocol. Hence, none of the test platforms are readily verifiable (i.e., as is) with this method unless significant changes to the RTL are made or a very involved and complicated conversion is undertaken. Next, RFUZZ [76] and Difuzzrtl [79] methods utilize coverage metrics, such as mux control coverage to guide their mutation. However, both of these require the translation of the source HDL into an intermediate representation language called FIRRTL IR [80] before being fuzzed. As far

TABLE VI
VULNERABILITY DETECTABILITY COMPARISON BETWEEN DIFFERENT HARDWARE FUZZING TECHNIQUES AND RE-PEN

Method	**SV1**		**SV2**		**SV3**		**SV4**	**SV5**	**SV6**	**NV1**	**NB1**
	Ariane	**MBlz**	**Ariane**	**MBlz**	**Ariane**	**MBlz**	**Ariane**	**Ariane**	**Ariane**	**Neorv32**	**Neorv32**
HyPFuzz [32]	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
MABFuzz [74]	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
PSOFuzz [75]	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
TheHuzz [17]	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
MoRFuzz [76]	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
Cascade [77]	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
[78]	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
SoCFuzzer [33]	✓	✗	✓	✗	✓	✗	✓	✓	✓	✗	✗
TaintFuzzer [34]	✓	✗	✓	✗	✓	✗	✓	✓	✓	✗	✗
Re-pen	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

as we are aware, only the Chisel HDL compiler has built-in support for converting HDL into FIRRTL IR. As none of the test platforms are written in Chisel HDL, none of the test platforms we have tested here are verifiable by these two fuzzers unless significant effort is expended in converting the RTL into FIRRTL IR.

SoCFuzzer and TaintFuzzer are both gray-box fuzzers that adopt system-level view of the SoC hardware, such as Re-Pen. Similar to Re-Pen, they also rely on cost functions to detect vulnerabilities. However, there are two key differences: first, both utilize more abstract coverage metrics (such as input coverage, output activity, and so on) to guide their test pattern mutation. On the other hand, Re-Pen utilizes security policies to construct its cost functions. The advantage of relying on security policies to construct cost function enables Re-Pen to detect oversights in DUT specification (as in NV1). Such oversights are not detectable by relying solely on coverage metrics, because as mentioned earlier, both the hardware and software were implemented correctly. Second, they both require their fuzz engine [namely, American Fuzzy Loop (AFL)] to run natively on the DUT. To achieve this, the DUT is booted with a general purpose OS (e.g., Linux), and then, AFL is run to analyze natively running application. However, MBlz and Neorv32 platforms do not have a virtual memory management unit and, hence, are not capable of booting a general purpose OS, such as Linux. As such, AFL cannot be run natively on these platforms, and their vulnerabilities are not detectable by these two techniques. We have summarized the preceding discussion in Table VI to show the detectability comparison between Re-Pen and these methods.

1) Quantitative Comparison: As Table VI illustrates, SoCFuzzer and TaintFuzzer provide the most detectability for the vulnerabilities and test platforms studied in this work. We also quantitatively compared the relative effectiveness of Re-Pen against these two methods in detecting SV1, SV2, SV4, and SV5 in Ariane test platform based on the no. of iterations to reach the vulnerability trigger point—the minima. Our findings are tabulated in Table VII.

We note that Re-Pen is significantly faster than SoCFuzzer and TaintFuzzer in detecting SV1, SV2, SV3, and SV6 vulnerabilities. Re-Pen is more directed in its vulnerability detection approach in that it specifically targets the design corners that are relevant to the policies associated with these vulnerabilities. However, since TaintFuzzer and SoCFuzzer use more abstract metrics for their cost functions, they have to explore a larger portion of the design before honing in on the corners relevant for these vulnerabilities that originate in processor external IPs. This larger exploration space requires more time, which is reflected in the larger number of iterations. For processor-internal vulnerabilities, such as SV4 and SV5, both outperform Re-Pen. This suggests that both of these methods explore processor internal corners of the design in the initial iterations much more effectively and, hence, can quickly converge to the minima.

TABLE VII
PERFORMANCE COMPARISON OF [33] AND [34] WITH RE-PEN

	No. of iterations required to detect		
Index	SoCFuzzer [33]	TaintFuzzer [34]	Re-pen (Mean)
SV1	2862	5261	**41.4**
SV2	2992	2597	**47.3**
SV3	2999	5292	**61.1**
SV4	351	**253**	3450.67
SV5	491	**353**	3150.28
SV6	2093	1945	**8.47**

E. Cost Function Requirements for Vulnerability Detection

As per the formulations in Section V-C, three cost functions were required to detect the vulnerabilities present in the test platforms. One might conclude from this that a separate cost function is needed to detect different vulnerabilities, which would restrict Re-Pen's scalability and generalizability. However, we point out that SV1–SV3 could be detected with the same cost function across the test platforms. This is because the underlying security policies that would be violated in the presence of these vulnerabilities are the same. Similarly, we could detect SV4 and SV5 on Ariane SoC using a single cost function, since the underlying security policies are the same for both. Hence, the no. of cost functions required is upper bounded by the set of security policies of the device. Furthermore, many of the policies can be grouped together to form a smaller number of cost functions.

In constructing and evaluating the cost functions, we observe that a certain set of prerequisites had to be met. First, through consulting an online (such as [69] and [70]) or in-house vulnerability database, a high-level knowledge is required of the security policies of the SoC, the vulnerabilities that might violate them, and associated hardware signals in the system that might potentially be impacted in the presence of these vulnerabilities, We argue that it is a reasonable requirement to be met, since as hardware security research matures, the cataloging of hardware vulnerabilities and corresponding security policies will continue to get better. Second, it should be made possible to observe or monitor the signals in the design that are associated with the vulnerability. If the simulation variant of Re-Pen is used, meeting this observability requirement is straightforward, as all RTL simulation provides extensive internal visibility of hardware signals. However, for the emulation variant, the instrumentation of proper signals and nets through ILA IPs needs to be performed provided JTAG access is not possible. Finally, the tester should have reasonable (but not necessarily exact) knowledge of how to trigger the vulnerability. This also implies that the tester should have controllability access to relevant signals. For example, it can be reasonably assumed that a baremetal C code can be used that exercises the crypto IP inputs for different corner cases for detecting SV1–SV3. Therefore, such a software code (or relevant parts of it) is considered as the input to mutate. We point out that these prerequisites have to also be met for some of the popular SoC security verification methodologies to differing extents. For example, to use formal assertion-based verification for these three test platforms, separate security properties would need to be written for each policy. Each property would also need to be potentially rewritten if the IPs, the ISA, or the microarchitecture changes to account for microarchitectural and IP structure variation as well as timing behaviors. On the other hand, as we just demonstrated, so long as a vulnerability violates these general gray-box requirements, regardless of the ISA or microarchitecture of the SoC under test, the cost function would not need to be rewritten. Therefore, Re-Pen significantly reduces the burden of manual expertise compared with popular formal-based verification methods.

VII. Conclusion

Through this work, we introduced Re-Pen, an enhanced hardware pentest framework that combines the advantages of RL and BPSO to explore and exploit the search space more effectively. We developed and formalized the novel RLBPSO algorithm, which employs RL agents to choose update equations for BPSO particles autonomously. We also presented an extensive performance comparison between Re-Pen and a previous hardware pentest framework on a variety of benchmark SoCs to demonstrate the increase in SV detection speed made possible by Re-Pen.

References

[1] *System-on-chip Market Outlook 2031*. Accessed: Feb. 6, 2024. [Online]. Available: https://www.transparencymarketresearch.com/soc-market.html

[2] U. Kamath and R. Kaundin, "System-on-chip designs: Strategy for success," Wripo Technol., White Paper, Jun. 2001, pp. 1–5.

[3] S. K. Roy and S. Ramesh, "Functional verification of system on chips–practices, issues and challenges," in *Proc. ASP-DAC/VLSI Design 7th Asia South Pacific Design Autom. Conf. 15h Int. Conf. VLSI Design*, Jan. 2002, pp. 11–13.

[4] E. Kovacs. *Unpatchable Hardware Vulnerability Allows Hacking of Siemens Plcs*. SecurityWeek. Accessed: Mar. 12, 2024. [Online]. Available: https://www.securityweek.com/unpatchable-hardware-vulnerability-allows-hacking-siemens-plcs/

[5] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "PACMAN: Attacking ARM pointer authentication with speculative execution," in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, Jun. 2022, pp. 685–698.

[6] F. Farahmandi, Y. Huang, and P. Mishra, *System-on-Chip Security*. Cham, Switzerland: Springer, 2020.

[7] X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, and P. Mishra, "Pre-silicon security verification and validation: A formal perspective," in *Proc. 52nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2015, pp. 1–6.

[8] J. Rajendran, A. M. Dhandayuthapany, V. Vedula, and R. Karri, "Formal security verification of third party intellectual property cores for information leakage," in *Proc. 29th Int. Conf. VLSI Design, 15th Int. Conf. Embedded Syst. (VLSID)*, Jan. 2016, pp. 547–552.

[9] P. Subramanyan and D. Arora, "Formal verification of taint-propagation security properties in a commercial SoC design," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2014, pp. 1–2.

[10] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, "End-to-end automated exploit generation for validating the security of processor designs," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 815–827.

[11] A. Ahmed, F. Farahmandi, Y. Iskander, and P. Mishra, "Scalable hardware trojan activation by interleaving concrete simulation and symbolic execution," in *Proc. IEEE Int. Test Conf. (ITC)*, Oct. 2018, pp. 1–10.

[12] K. Hasegawa, Y. Shi, and N. Togawa, "Hardware trojan detection utilizing machine learning approaches," in *Proc. 17th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun. 12th IEEE Int. Conf. Big Data Sci. Eng. (TrustCom/BigDataSE)*, Aug. 2018, pp. 1891–1896.

[13] Y. Lyu and P. Mishra, "Scalable concolic testing of RTL models," *IEEE Trans. Comput.*, vol. 70, no. 7, pp. 979–991, Jul. 2021.

[14] A. Nahiyan, M. Sadi, R. Vittal, G. Contreras, D. Forte, and M. Tehranipoor, "Hardware trojan detection through information flow security verification," in *Proc. IEEE Int. Test Conf. (ITC)*, Oct. 2017, pp. 1–10.

[15] N. Bruns, V. Herdt, E. Jentzsch, and R. Drechsler, "Cross-level processor verification via endless randomized instruction stream generation with coverage-guided aging," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2022, pp. 1123–1126.

[16] K. Ruep and D. Große, "SpinalFuzz: Coverage-guided fuzzing for SpinalHDL designs," in *Proc. IEEE Eur. Test Symp. (ETS)*, May 2022, pp. 1–4.

[17] A. Tyagi et al., "TheHuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities," 2022, *arXiv:2201.09941*.

[18] M. Rostami, C. Chen, R. Kande, H. Li, J. Rajendran, and A.-R. Sadeghi, "Fuzzerfly effect: Hardware fuzzing for memory safety," *IEEE Secur. Privacy*, vol. 22, no. 4, pp. 76–86, Jul. 2024.

[19] K. Z. Azar et al., "Fuzz, penetration, and AI testing for SoC security verification: Challenges and solutions," *Cryptol. ePrint Arch.*, 2022. [Online]. Available: https://eprint.iacr.org/2022/394

[20] H. Al-Shaikh et al., "SHarPen: SoC security verification by hardware penetration test," in *Proc. 28th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2023, pp. 579–584.

[21] P. Engebretson, *The Basics of Hacking and Penetration Testing: Ethical Hacking and Penetration Testing Made Easy*. Amsterdam, The Netherlands: Elsevier, 2013.

[22] Q. Zhang, K. Wang, W. Zhang, and J. Hu, "Attacking black-box image classifiers with particle swarm optimization," *IEEE Access*, vol. 7, pp. 158051–158063, 2019.

[23] H. M. Allawi, W. Al Manaseer, and M. Al Shraideh, "A greedy particle swarm optimization (GPSO) algorithm for testing real-world smart card applications," *Int. J. Softw. Tools Technol. Transf.*, vol. 22, no. 2, pp. 183–194, Apr. 2020.

[24] L. Yang, Z. Li, D. Wang, H. Miao, and Z. Wang, "Software defects prediction based on hybrid particle swarm optimization and sparrow search algorithm," *IEEE Access*, vol. 9, pp. 60865–60879, 2021.

[25] X.-H. Wang and J.-J. Li, "Hybrid particle swarm optimization with simulated annealing," in *Proc. Int. Conf. Mach. Learn. Cybern.*, vol. 4, Jun. 2004, pp. 2402–2405.

[26] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2018, pp. 1–8.

[27] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi, "DirectFuzz: Automated test generation for RTL designs using directed graybox fuzzing," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, Dec. 2021, pp. 529–534.

[28] N. Farzana, F. Farahmandi, and M. M. Tehranipoor, "SoC security properties and rules.," *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 1014, Jan. 2021.

[29] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *Tools for Practical Software Verification: LASER, International Summer School 2011*. Elba Island, Italy: Springer, 2012, pp. 1–30.

[30] V. Herdt and R. Drechsler, "Advanced virtual prototyping for cyber-physical systems using RISC-V: Implementation, verification and challenges," *Sci. China Inf. Sci.*, vol. 65, no. 1, Jan. 2022, Art. no. 110201.

[31] Z. Pan and P. Mishra, "Automated test generation for hardware Trojan detection using reinforcement learning," in *Proc. 26th Asia South Pacific Design Autom. Conf.*, Jan. 2021, pp. 408–413.

[32] C. Chen and R. Kande, "HyPFuzz: Formal-assisted processor fuzzing," in *Proc. 32nd USENIX Secur. Symp. (USENIX Secur. 23)*, 2023, pp. 1361–1378.

[33] M. M. Hossain, A. Vafaei, K. Z. Azar, F. Rahman, F. Farahmandi, and M. Tehranipoor, "SoCFuzzer: SoC vulnerability detection using cost function enabled fuzz testing," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Apr. 2023, pp. 1–6.

[34] M. M. Hossain, N. F. Dipu, K. Z. Azar, F. Rahman, F. Farahmandi, and M. Tehranipoor, "TaintFuzzer: SoC security verification using taint inference-enabled fuzzing," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2023, pp. 1–9.

[35] U. Popovic. *789 Kb Linux Without Mmu on Risc-v*. Accessed: Mar. 19, 2024. [Online]. Available: https://popovicu.com/posts/789-kb-linux-without-mmu-riscv/

[36] M. Rostami, M. Chilese, S. Zeitouni, R. Kande, J. Rajendran, and A.-R. Sadeghi, "Beyond random inputs: A novel ML-based hardware fuzzing," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2024, pp. 1–6.

[37] R. Kande et al., "(Security) assertions by large language models," 2023, *arXiv:2306.14027*.

[38] D. Saha et al., "LLM for SoC security: A paradigm shift," *IEEE Access*, vol. 12, pp. 155498–155521, 2024.

[39] N. Nahar Mondol, A. Vafei, K. Zamiri Azar, F. Farahmandi, and M. Tehranipoor, "RL-TPG: Automated pre-silicon security verification through reinforcement learning-based test pattern generation," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2024, pp. 1–6.

[40] V. Gohil, S. Patnaik, H. Guo, D. Kalathil, and J. Rajendran, "DETERRENT: Detecting trojans using reinforcement learning," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, Jul. 2022, pp. 697–702.

[41] H. Chen, X. Zhang, K. Huang, and F. Koushanfar, "AdaTest: Reinforcement learning and adaptive sampling for on-chip hardware trojan detection," *ACM Trans. Embedded Comput. Syst.*, vol. 22, no. 2, pp. 1–23, Mar. 2023.

[42] Z. Pan, J. Sheldon, and P. Mishra, "Test generation using reinforcement learning for delay-based side-channel analysis," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2020, pp. 1–7.

[43] A. Shelton and J. Mellor, "Attacking logic locked circuits using reinforcement learning," Santa Clara Univ., Santa Clara, CA, USA, Tech. Rep. 58, 2021.

[44] Z.-H. Zhan, J. Zhang, Y. Li, and H. S.-H. Chung, "Adaptive particle swarm optimization," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 39, no. 6, pp. 1362–1381, Dec. 2009.

[45] A. Ratnaweera, S. K. Halgamuge, and H. C. Watson, "Self-organizing hierarchical particle swarm optimizer with time-varying acceleration coefficients," *IEEE Trans. Evol. Comput.*, vol. 8, no. 3, pp. 240–255, Jun. 2004.

[46] W. H. Lim and N. A. Mat Isa, "An adaptive two-layer particle swarm optimization with elitist learning strategy," *Inf. Sci.*, vol. 273, pp. 49–72, Jul. 2014.

[47] B. Ji, X. Lu, G. Sun, W. Zhang, J. Li, and Y. Xiao, "Bio-inspired feature selection: An improved binary particle swarm optimization approach," *IEEE Access*, vol. 8, pp. 85989–86002, 2020.

[48] H. Wang, I. Moon, S. Yang, and D. Wang, "A memetic particle swarm optimization algorithm for multimodal optimization problems," *Inf. Sci.*, vol. 197, pp. 38–52, Aug. 2012.

[49] G. Wu, D. Qiu, Y. Yu, W. Pedrycz, M. Ma, and H. Li, "Superior solution guided particle swarm optimization combined with local search techniques," *Expert Syst. Appl.*, vol. 41, no. 16, pp. 7536–7548, Nov. 2014.

[50] P. Rakshit et al., "Realization of an adaptive memetic algorithm using differential evolution and Q-learning: A case study in multirobot path planning," *IEEE Trans. Syst., Man, Cybern. Syst.*, vol. 43, no. 4, pp. 814–831, Jul. 2013.

[51] L. Jiao, M. Gong, S. Wang, B. Hou, Z. Zheng, and Q. Wu, "Natural and remote sensing image segmentation using memetic computing," *IEEE Comput. Intell. Mag.*, vol. 5, no. 2, pp. 78–91, May 2010.

[52] D. Wolpert and W. Macready, "No free lunch theorems for optimization," *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 67–82, Apr. 1997.

[53] N. Farzana, A. Ayalasomayajula, F. Rahman, F. Farahmandi, and M. Tehranipoor, "SAIF: Automated asset identification for security verification at the register transfer level," in *Proc. IEEE 39th VLSI Test Symp. (VTS)*, Apr. 2021, pp. 1–7.

[54] S. Ray and Y. Jin, "Security policy enforcement in modern SoC designs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2015, pp. 345–350.

[55] F. Farahmandi, M. S. Rahman, S. R. Rajendran, and M. Tehranipoor, *CAD for Hardware Security*. Cham, Switzerland: Springer, 2023.

[56] A. Stern, H. Wang, F. Rahman, F. Farahmandi, and M. Tehranipoor, "ACED-IT: Assuring confidential electronic design against insider threats in a zero-trust environment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 10, pp. 3202–3215, Oct. 2022.

[57] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *Proc. 30th IEEE Symp. Secur. Privacy*, Sep. 2009, pp. 45–60.

[58] M. A. Wiering and M. Van Otterlo, "Reinforcement learning," *Adaptation, Learn. Optim.*, vol. 12, no. 3, p. 729, 2012.

[59] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, "Q-learning algorithms: A comprehensive classification and applications," *IEEE Access*, vol. 7, pp. 133653–133667, 2019.

[60] Accessed: Feb. 7, 2024. [Online]. Available: https://www.veripool.org/verilator/

[61] J. Kennedy and R. C. Eberhart, "A discrete binary version of the particle swarm algorithm," in *Proc. IEEE Int. Conf. Syst., Man, Cybern. Comput. Cybern. Simul.*, vol. 5, Sep. 1997, pp. 4104–4108.

[62] R. S. Sutton and A. G. Barto, "Reinforcement learning," *J. Cognit. Neurosci.*, vol. 11, no. 1, pp. 126–134, 1999.

[63] T. M. Shami, A. A. El-Saleh, M. Alswaitti, Q. Al-Tashi, M. A. Summakieh, and S. Mirjalili, "Particle swarm optimization: A comprehensive survey," *IEEE Access*, vol. 10, pp. 10031–10061, 2022.

[64] M. Z. Shirazi, T. Pamulapati, R. Mallipeddi, and K. C. Veluvolu, "Particle swarm optimization with ensemble of inertia weight strategies," in *Proc. 8th Int. Conf. Adv. Swarm Intell. (ICSI)*, Fukuoka, Japan. Springer, Jul. 2017, pp. 140–147.

[65] Z. Ji, H. Liao, Y. Wang, and Q. H. Wu, "A novel intelligent particle optimizer for global optimization of multimodal functions," in *Proc. IEEE Congr. Evol. Comput.*, Sep. 2007, pp. 3272–3275.

[66] *The Neorv32 SoC*. Accessed: Feb. 11, 2024. [Online]. Available: https://github.com/stnolting/neorv32

[67] J. Balkind et al., "OpenPiton+Ariane: The first open-source, SMP Linux-booting RISC-V system scaling from one to many cores," in *Proc. Workshop Comput. Archit. Res. RISC-V (CARRV)*, 2019, pp. 1–6.

[68] *Microblaze Processor Reference Guide*, document ug984, X. Inc, 2015.

[69] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, and M. Tehranipoor, "Benchmarking of hardware Trojans and maliciously affected circuits," *J. Hardw. Syst. Secur.*, vol. 1, no. 1, pp. 85–102, Mar. 2017.

[70] *The Common Weakness Enumerations*. Accessed: Feb. 23, 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/1245.html

[71] *Xilinx Ila Logic Debug Core*. Accessed: Feb. 23, 2024. [Online]. Available: https://www.xilinx.com/products/intellectual-property/ila.html

[72] A. Waterman and K. Asanovi. (2017). *The Risc-v Instruction Set Manual: Privileged Architecture*. [Online]. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf

[73] Accessed: Mar. 12, 2024. [Online]. Available: https://pyswarms.readthedocs.io/en/latest/intro.html

[74] V. Gohil, R. Kande, C. Chen, A.-R. Sadeghi, and J. Rajendran, "MABFuzz: Multi-armed bandit algorithms for fuzzing processors," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2024, pp. 1–6.

[75] C. Chen, V. Gohil, R. Kande, A.-R. Sadeghi, and J. Rajendran, "PSOFuzz: Fuzzing processors with particle swarm optimization," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Oct. 2023, pp. 1–9.

[76] J. Xu, Y. Liu, S. He, H. Lin, Y. Zhou, and C. Wang, "MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation," in *Proc. 32nd USENIX Secur. Symp. (USENIX Secur. 23)*, 2023, pp. 1307–1324.

[77] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: CPU fuzzing via intricate program generation," in *Proc. 33rd USENIX Secur. Symp.*, 2024, pp. 1–18.

[78] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," 2021, *arXiv:2102.02308*.

[79] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "DifuzzRTL: Differential fuzz testing to find CPU bugs," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 1286–1303.

[80] A. Izraelevitz et al., "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," in *IEEE/ACM Int. Conf. Comput.-Aided Design Dig. Tech. Papers*, Nov. 2017, pp. 209–216.