ISSN: 2754-6659

Journal of Artificial Intelligence & Cloud Computing

SCIENTIFIC Research and Community

Research Article

Open Access

Advancing Moving Target Strategy with Bio-Inspired Reinforcement Learning to Secure Misconfigured Software Applications

Niloofar Heidarikohol and Shuvalaxmi Dass*

School of Computing and Informatics, University of Louisiana at Lafayette, USA

ABSTRACT

Misconfigurations in software systems are a persistent source of security vulnerabilities, particularly within static architectures that fail to adapt over time. Moving Target Defense (MTD) offers a proactive approach by dynamically altering the system's attack surface, thereby reducing exposure. This paper builds upon an MTD model, RL-MTD, which leverages Reinforcement Learning (RL) to generate adaptive secure configurations. Although effective, RL-MTD faces limitations due to an unoptimized and sparse search space. To address this, two hybrid models—GA-RL and PSO-RL—are proposed, integrating Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) into the RL-MTD framework. Experiments on four misconfigured SUTs show both models outperform the baseline. Notably, PSO-RL yields the most secure configurations in most scenarios. The authors present a prototype demonstrating how PSO-RL could be applied on a constrained Windows 10 system to defend against an attack. These findings enhance MTDbased adaptive cybersecurity via optimized search.

***Corresponding author**

Shuvalaxmi Dass, School of Computing and Informatics, University of Louisiana at Lafayette, USA.

Received: October 01, 2025; **Accepted:** October 06, 2025; **Published:** October 15, 2025

Keywords: System Security, Software Configuration, Moving Target Defense (MTD), Reinforcement Learning (RL), Genetic Algorithms (GA), Particle Swarm Optimization (PSO)

Introduction

In today's digital era, software applications are integral to every aspect of our lives, from powering our smartphones to driving complex business operations, driving innovation and efficiency. However, such highly configurable applications if not configured properly can lead to security misconfiguration in software making the system vulnerable to attacks like data breaches. In 2021, Twitch, an interactive livestreaming platform, suffered a massive 125GB data and source code leak due to server misconfiguration [1]. The Open Web Application Security Project (OWASP) lists "Security misconfiguration" among the top 5 web application security risks as of 2021 [2]. Misconfigurations stem from the static nature of application configurations, persisting over time. This static characteristic can result in improper security settings due to missed updates, human errors, or incomplete configurations, leaving systems vulnerable to attacks or unauthorized access by attackers [3].

To counter such threats, we require a dynamic defensive strategy to overcome the static nature of configurations. Traditional solutions like costly antivirus software, which focuses on detection and reaction, are ineffective against evolving attack strategies. A proactive dynamic defense is a more effective approach.

In our preliminary work, we introduced a model for generating dynamic secure configurations using Moving Target Defense (MTD) as our proactive defense solution. We implemented MTD via Reinforcement Learning (RL), termed RL-MTD. However, the performance of our RL-MTD base model was impacted by sparse search space issues. In this paper, we address the performance issue of non-optimized search space by enhancing our base model through integration with bio-inspired algorithms (GA-RL and PSO-RL) to improve performance.

MTD: According to the Department of Homeland Security (DHS), MTD is a military strategy translated to the cybersecurity world that involves dynamically manipulating various system configurations to alter and manage the attack surface, thereby increasing uncertainty and complexity for attackers [4]. This approach reduces opportunities for attackers to identify vulnerable system components and raises the cost of launching attacks or scans. Ultimately, the goal is to make attackers expend time and effort without gaining valuable intelligence about the system [5].

To the best of our knowledge, our work represents the initial endeavor to introduce a proof-of-concept MTD defense strategy focused on software configuration at the individual application level. This paper builds upon preliminary research that laid the groundwork for developing the RL-MTD framework to address software misconfiguration as follows

- We briefly discussed building our foundation work which is base RL-MTD framework.
- The optimization issue of the search space in the RL-MTD model is examined.
- We propose the integration of RL-MTD with bio-inspired algorithms (GA, PSO) to create GA-RL and PSO-RL, offering a solution for optimizing the search space problem.
- A quantitative analysis is conducted to compare the performance of RL-MTD, GA-RL, and PSO-RL across four misconfigured case studies of (SUTs) in terms of generating secure configurations.
- Demonstrative demo on Windows 10.

The paper follows this structure:
An overview of prior work on RL-MTD model (Section 2), addressing the search space issue (Section 3), integrating PSO and GA to RL-MTD (Section 5) after providing background on PSO and GA (Section 4), experimental setup and results (Section 6), discussion of results, attack - defense demo, related work, and conclusion with future work (Sections 7-10).

Preliminary Work
In this section, we will first briefly discuss our preliminary work which forms the foundation of the initial work done in the field of Moving Target Defense for software misconfiguration [6].

We will divide this section into 2 parts where the first part talks briefly about the motivating example/problem statement and the second part demonstrates the MTD approach designed for the problem.

Motivating Scenario
Consider a host machine in an organization running a specific software system with its own configuration space. If some settings were accidentally tampered with due to manual intervention, static analysis tools can detect these misconfigurations using techniques like taint analysis, propagation policy studies, and inconsistency detection [7]. For example, ConfTainter analyzes the impact of configuration options on program behavior, considering data and control flow, and achieves high accuracy in detecting misconfigurations [8]. If the tool identifies a misconfiguration, it could make the system vulnerable to attacks like brute force, code injection, buffer overflow, and cross-site scripting (XSS). Attackers exploit these vulnerabilities by first conducting reconnaissance to understand the misconfiguration. To counter this, the proposed solution uses a Moving Target Defense (MTD) strategy that frequently changes the configuration settings. This approach aims to create a dynamic, secure environment, rendering the attacker's knowledge obsolete and preventing effective exploit development [9]. **The goal is to move towards secure configurations via dynamically changing the configurations from misconfigured state to a secure state as a dynamic defensive measure (MTD) using Reinforcement Learning (RL)**. This will confuse the attackers who rely on outdated or constantly evolving information.

RL-MTD Approach
The idea is to develop a proof-of-concept of a defense approach inspired by the Moving Target Defense (MTD) to defend misconfigured SUT. We modeled MTD in the form of a single-player game implemented using the model-free Monte-Carlo method in Reinforcement Learning (RL) where the goal is to convert a misconfigured SUT to a secure-configured SUT to defend it from potential attacks. This section first talks about the attack surface used in our MTD strategy followed by the game model description using RL.

Attack Surface
We represent the attack surface of a SUT as a configuration *C* which is composed of a series of *P* parameters that belong to SUT. We denote C as:

$$C: = \{P_1: S_1, P_2: S_2, \ldots, P: S_n\}$$

where n represents the number of parameters and S_i is the setting value associated with parameter P_i in a configuration space of SUT. We collected the configuration information of different SUT from the Security Technical Implementation Guide (STIG). The STIG guidelines offer proper checklists to view the "compliance status" of the system's security settings. In other words, the STIG checklists enable us to test whether the underlying system configuration complies with standards (i.e., secure system settings regulations. Figure 1 lists some of the parameters of Windows 10 along with their default values and the domain of values it belongs to. However, the key goal of the MTD technique is to rearrange or randomize system configurations to increase confusion and uncertainty for attackers [5]. Therefore, the attack surface that we use for the RL-MTD model, starts with a misconfigured SUT and the task of the agent is to learn to navigate towards the securely configured SUT.

A misconfigured attack surface would have all the parameter's settings improperly set (i.e randomly drawn from domain of (P))

For instance, *C* for misconfigured Windows would look like:

$C: = \{ ACSettingIndex: 5, AllowBasic:3, \ldots, DoDownloadMode:4\}$

where the settings are a finding as these parameters are not securely set

Parameter Name	Secure Default Values	Domain
ACSettingIndex	1	Integer
AllowBasic	0	Integer, 'None'
AllowDigest	0	Integer
AllowTelemetry	(0, 1)	Integer
AllowUnencryptedTraffic	0	Integer
AlwaysInstallElevated	0	Integer
AutoConnectAllowedOEM	0	Integer
CachedLogonsCount	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	Integer
ConsentPromptBehaviorAdmin	2	Integer
DCSettingIndex	1	Integer
DODownloadMode	[0, 1, 2, 99, 100]	Integer

Figure 1: Windows 10 Configuration Parameters

Note: A subset of Windows 10 configuration parameters associated with default secure settings

RL-Based Game Description of MTD

MTD is modeled as a single-player game played by an RL agent which acts as a defender focusing on the security of misconfigured system. Imagine SUT's attack surface as a board game where the MTD RL agent has a start state, a goal state, and multiple dynamic intermediate states which are dynamically generated and traversed based on the RL agent's action. The overall objective of the MTD-RL agent is to dynamically transition from an insecure state (start) of the misconfigured SUT towards a nearly secure state(goal) of the SUT by taking some actions. Figure 2 shows the RL environment elements used in our MTD approach and how RL elements interact with the environment which is the attack surface of a misconfigured SUT.

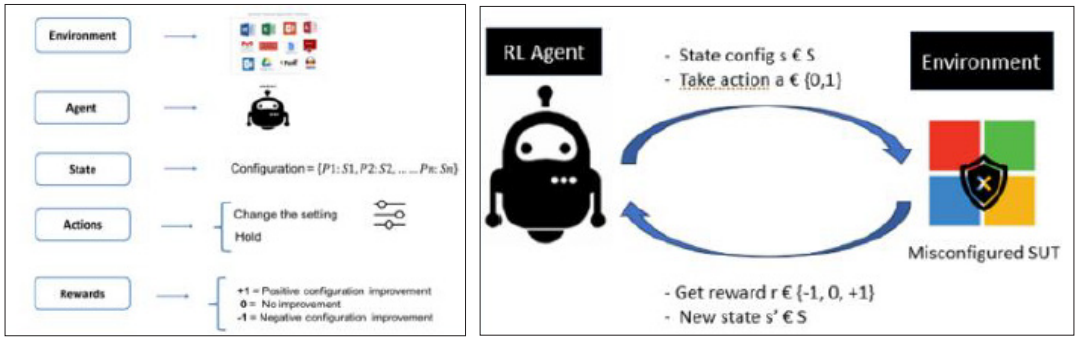

Figure 2: RL-elements and MTD-RL Interaction with Misconfigured SUT

Note: (Left) RL elements: The environment is the misconfigured SUT's attack surface, an agent is the Monte-Carlo-based RL agent, the State represents the configuration (C) instance/state of the misconfigured SUT, actions taken by an agent are to either change a particular parameter setting or hold back and rewards are given based on the improvement(+,- or none) of configuration security score from its previous state. **(Right)** The MTD-RL agent interacts with a misconfigured SUT environment (eg Windows), where the state s is the current config (C) it is in, and it takes an action (0 or 1) based on which the SUT moves the agent to the next state s' and returns a reward (0,1,-1) based on the actions

Starting from the initial state, at every step, the agent must decide whether to alter (action = change (1)) or keep some parameter settings as is (action = Hold (0)) of the current config state. The action taken is based on the fitness score of the current config state, where if it is below a certain threshold value, action = 1 is chosen, other 0. Subsequently, a reward is generated which indicates how good the action is which helps the maximize its choices to progressively achieve a more secure intermediate next config state until it reaches the goal state. Figure 3 shows how the game is played where the ultimate strategy is to continuously and dynamically modify the attack surface towards the direction of the finish/secure state, thereby reducing/changing vulnerabilities in the process and confusing the attackers by increasing the uncertainty. The game's dynamics are implemented through the model-free RL-based Monte Carlo Prediction method, which guides the decision-making process for optimizing security configurations through a defined policy.

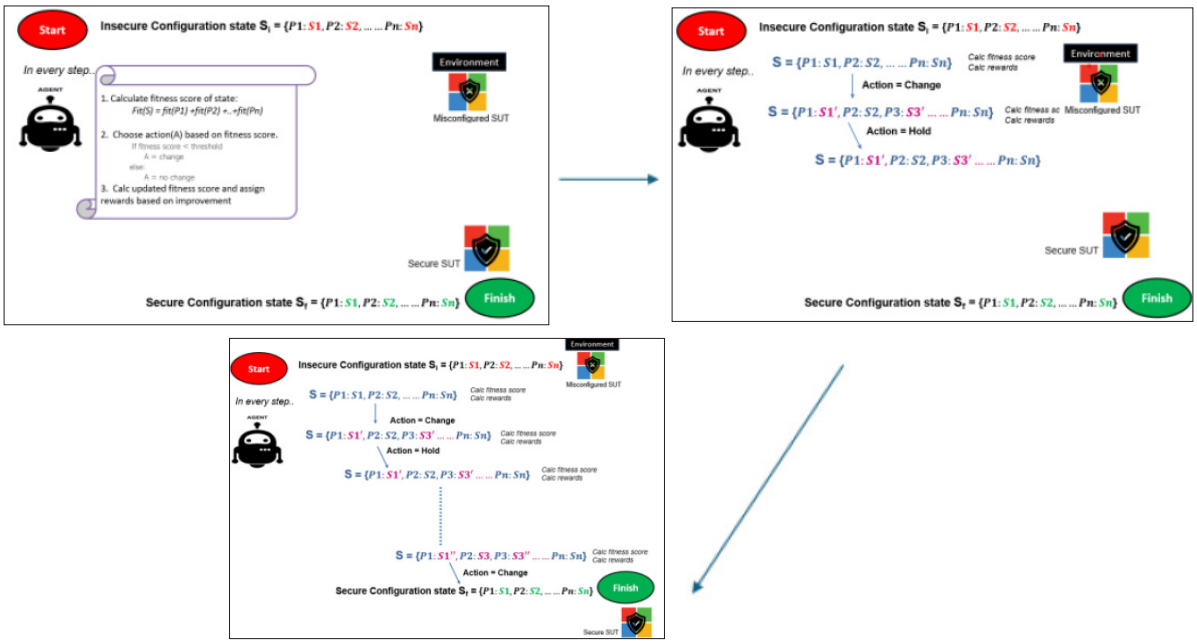

Figure 3: Snapshot of What an RL-MTD Game Looks Like When Played

Note: Initially, it starts from an insecure state **(Top-left)**, takes an action based on the config fitness score, gets a reward, and moves to the next intermediate steps **(Top-right)**. This series of operations is followed in every step until the agent reaches the near-optimal secure config finish state **(Bottom)**.

RL-MTD Game Algorithm

This section outlines a procedure and a series of step-by-step algorithms for implementing the environment required to execute model-free RL Monte Carlo (MC) prediction method in the context of the RL-MTD game. The Monte Carlo method can help improve a supplied policy that is effective at making decisions that lead to winning the game which in our case is reaching as close to the terminal state as possible. In other words, the purpose of this method is to generate secure configurations for a misconfigured SUT platform by assessing the quality of a given policy.

Technically, in MC prediction, the objective is to estimate the state-value function V(s), which represents the expected return from a state S under a given policy. Instead of using the term "expected return" (which is the discounted sum of rewards), we employ the concept of "empirical return." Essentially, MC prediction assesses how well a fixed predefined policy performs by predicting the mean total rewards from any given state, assuming the policy remains constant [10]. The MC prediction pseudocode is shown in Algorithm 1.

Our RL- MTD algorithm is also supplied with a fixed policy and aims to evaluate its performance in terms of the value function. In other words, our goal is to predict the expected total reward from the most secure state it has reached. Consequently, we measure the reward using the fitness score of each model configuration, where higher fitness indicates a more secure configuration. Moreover, we chose a modelfree MC prediction method as the probability of transitioning (i.e., transition probabilities) to the next configuration/state (different set of settings) cannot be gauged from the environment as there is no predefined domain of knowledge known to measure the likelihood of moving from one configuration to another. As a result, the agent learns through running multiple episodes, constantly collecting samples (random values of settings), getting rewards, and thereby evaluating the value function.

RL-MTD approach explained in the previous subsection can be divided into the following steps

1. **Step 1: Set Initial State:** Initial state is a configuration C of the underlying system initialized with random settings for its parameters.
2. **Step 2: Compute Config/Fitness Score:** The fitness score of a configuration state is the total sum of individual fitness scores of the parameters. A parameter receives a definite HIGH = 800 score if it is associated with its secure setting according to the STIG website. Otherwise, a LOW = 8 score is assigned. The fitness score indicates how secure a configuration is. The higher the fitness score, the more secure it is. These are hyperparameter values used to score the severity of these settings.
3. **Step 3: Set Action Policy:** If the overall fitness score of the configuration is below a certain threshold value, then the agent chooses action a either 0(hold) or 1(change) based on the probability distribution p(a) as follows:

$$p(a) = \begin{cases} \{prob_0 = low, prob_1 = high\} : \& \\ \text{if } fitness\_score < Threshold \\ \{prob_0 = high, prob_1 = low\} : \& \\ \text{if } fitness\_score >= Threshold \end{cases}$$

where *a = hold (0)* or *Change (1)*. This ensures the agent chooses action 1 more if the fitness score of the C is not up to the mark (threshold) and vice versa.

1. **Step 4: Generate tuple**. This tuple is indicative of the current position of the agent in the attack surface. Rewards measure how good of an action was taken by calculating the fitness improvement (Fit (new S) - Fit (old S))
2. **Step 5: Generate Episodes**. Collection of tuples used for training the agent.
3. **Step 6: Execute RL MC prediction with the given action policy**. This is executed with multiple episodes and eventually captures the best fitness scores in the form of the value of state V which indicates how secure a given state is:

$$Vs = Epi\ [Rt+1 + \gamma\ Rt+2 + \gamma 2\ Rt+3 ... |\ St = s]$$

Where *E* is the expected mean of the reward for the state *s*.

As there was no pre-existing environment for our problem domain in OpenAI gym at our disposal, we had to create our environment using Step 4 which required Steps 1, 2, and 3 [11].

Step 5 is used to generate an episode of 100 tuples that are used for training the MC prediction algorithm as shown in Algorithm 1 which is a standard algorithm used in literature. Figure 4 shows a highlevel overview of how Steps 1-4 are implemented.

In short, RL-MTD model is composed of 3 main parts: environment(), generate_episode() and MC_prediction method ()

Algorithm 1

Monte Carlo Prediction Pseudocode [10].

1. Procedure mc_prediction(policy, num_ep, df)
2. returns_sum ← { } // Keeps track of the sum of returns for each state to calculate an average.
3. returns_count ← { } // Keeps track of the count of returns for each state to calculate an average.
4. V ← { } // The final value function
5. For i in range (1, num_ep + 1)
6. episode ← generate_episode(policy)
7. states_in_episodes ← Find all states visited in this episode and convert them into tuples
8. For state in states_in_episodes
   a. first_occurrence ← First occurrence of the state in the episode
   b. G ← Sum up all rewards since the first occurrence
   c. returns_sum[state] += G
   d. returns_count[state] += df
   e. V[state] = returns_sum[state] / returns_count[state]
9. End For
10. End For
11. Return V
12. End Procedure

Figure 5 shows the complete execution process flow of our RL-MTD algorithm.

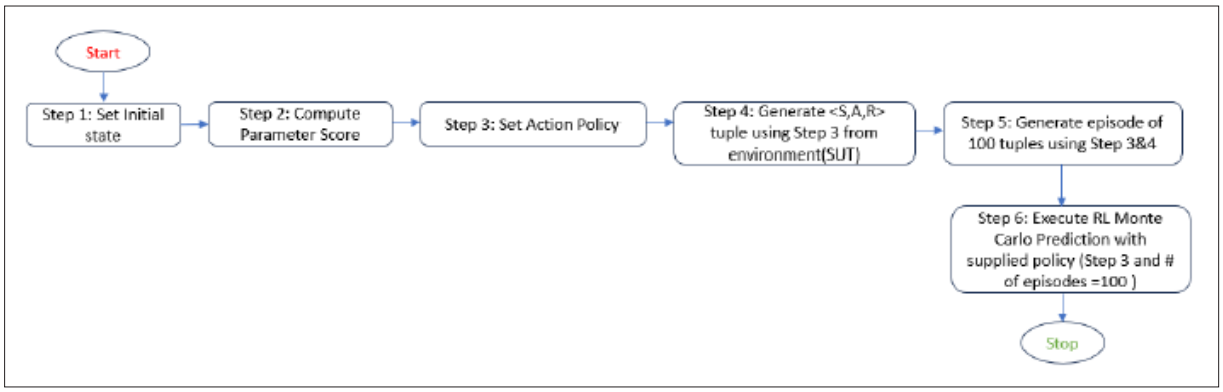

Figure 5: The Execution Flow of Base RL-MTD

Note: The execution flow of base RL-MTD model where Steps 1-4 make up the environment(), step 5 is generate_episode () and step 6 is MC prediction method ().

Search Space Issue in RL-MTD

In the RL-MTD work, there are two instances where the RL agent has to draw random setting values from the search space range of every parameter P present in a configuration: once during the start state (initial state) of the game when it has generated a random configuration state, other times whenever the action = 1 is chosen and it has to change settings of low score parameters.

As per the STIG website, the permissible default settings are defined for any parameter P belonging to a particular SUT. In our work, as this is proof-of-concept, we handpicked only those parameters that mostly had integer values and/or 'None' as their settings for ease of computation.

Figure 6 shows how we designed the search space range for the agent to choose from during the RL-MTD operation for any particular SUT. The agent has to learn to eventually choose the permissible secure setting for every parameter P in configuration C from the given search range to ensure C is close to being secure (goal state). We set the custom space range as follows depending on the data type of setting(P):

1. Numeric: (*v-lim, v+lim*), and
2. List: choice between [(*0, max(v1,v2,v3..) +lim*] and *None*

where *v* is the default setting value as per the STIG website for any SUT, *max(List)* is the maximum setting value if it is a list type, and *lim* is an arbitrary integer value. We set *lim = 10* based on multiple experiments we conducted, and this value seemed to give better secure configurations (as we also covered in the Results section).

Issue: However, finding the right value of *lim* to define the search space range from where the RL agent picks up random values for parameters can be a time-consuming task and often requires us to play around with a bunch of different values before we can find a good enough candidate. As this search space range is majorly responsible for the performance of RL in terms of generating secure configurations, there is a need to find an optimized search space for RL that is best for agents to generate diverse yet secure configurations.

The solution to this problem is to use search optimization algorithms. In section 5, we describe how we integrate the bio-inspired search optimization algorithms namely Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) into RL-MTD and develop Evolutionary RL(E-RL) algorithms for better performance.

We integrate both GA and PSO in our approach because they are widely recognized optimization techniques. PSO operates as a population-based stochastic optimization algorithm, focusing on the collective behavior of swarms, while GA functions as a heuristic search-based algorithm, simulating evolutionary processes like crossover and mutation [12,13]. Our study aims to compare and assess the performance of these two algorithms to determine which one surpasses the base model in effectiveness.

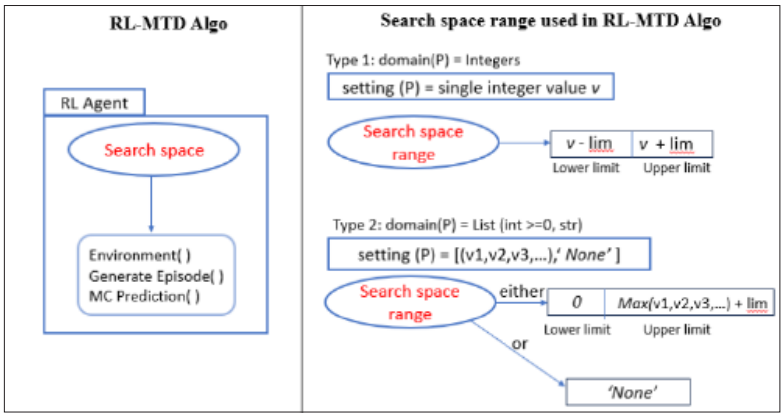

Figure 6: The Diagrammatic View of Search Space in RL-MTD Algorithm

Note: (Left) RL-MTD Algo shows the diagrammatic view of all the important functions that use the search space/domain corresponding to a particular SUT. This search range is used by the agent to randomly draw settings from either during the initial config state or when action=1 is chosen. **(Right)** shows the search space range for 2 types of parameter settings where the agent has to pick a setting from *{v-lim,v+lim}* if default setting(P) is a single integer value *v* and *lim* is a hyperparameter for a limit of int type; and if default setting(P) could be any value from a list consisting of permissible non-neg integers (v1,v2,v3,...) and/or 'None', it gets to choose either a numerical *val* from *{0, max((v1.v2,v3,.)+lim}* or the string value 'None'.

Issue: How to effectively choose value of *lim* that will optimize the search space range for better performance?

Background

In this section, we briefly discuss the general working of bio-inspired algorithms: Genetic Algorithm and Particle Swarm optimization.

Genetic Algorithm

Genetic Algorithms (GA) are based on the biological process of evolution. The idea is that over time, a pool of chromosomes will evolve to be even better (i.e., better fitness value) than the previous generation. A new generation (equal to the pool size) of chromosomes (i.e., configurations) is created with any iteration of the algorithm. This is achieved by the processes of selection, crossover, and mutation [14]. A fitness score metric is adopted as a measure to select the two fittest chromosomes from the pool that are called parent chromosomes. Then crossover takes place between the parents to produce a new child chromosome, which will have the best traits from both the parents followed by mutating of some of the characteristics of the child to introduce new traits. This process is repeated until an entirely new generation gets created. Figures 7 and 8 show the elements and the process of GA respectively.

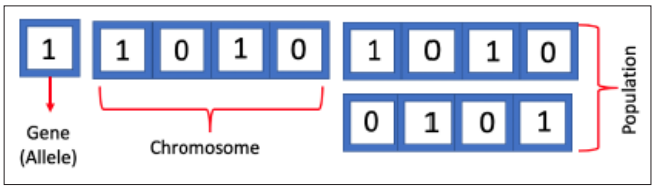

Figure 7: The Elements of Genetic Algorithm

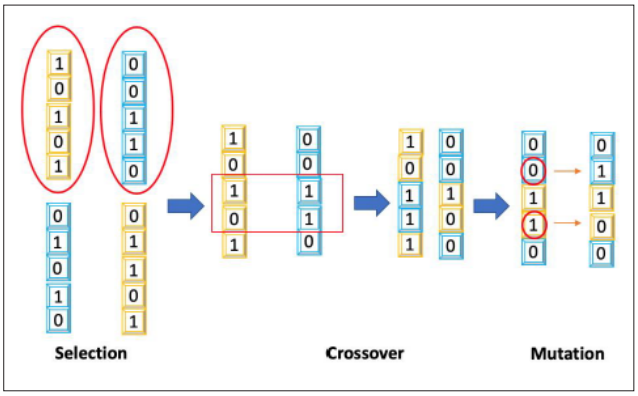

Figure 8: Genetic Algorithm Process

Particle Swarm Optimization

The Particle Swarm Optimization approach is based on natural bio-inspired systems, including bird flocks or schooling fish [15]. Each particle follows certain fundamental rules to navigate the search space effectively or reach optimal values [16]. Individuals should maintain an appropriate distance in standard scenarios, avoid collisions, and remain very close when confronted with threats [15].

PSO stands as an iterative, random, and population-based optimization algorithm for determining the optimal value. This is accomplished by assigning a particle to locate the ideal location or answer within the search space. Each particle's dynamics is influenced by social movements as well as its own internal dynamic. Originally, each particle, irrespective of its peers, can be considered to behave independently to ameliorate its behavior. However, swarms attempt to adjust to the behavior of other particles as the algorithm processes. Consequently, each particle adjusts updates iteratively with other particles to observe the optimal value. Furthermore, the characteristics of each swarm can be determined by the interaction of position and velocity [17]. Figure 9 shows the workflow of a general PSO process [18]. The algorithm starts by initializing a swarm of particles with random positions and velocities. It then enters a loop where it evaluates the fitness of each particle, updates their personal best positions and the global best position, and then adjusts their velocities and positions accordingly. This loop continues until a termination criterion, such as a maximum number of iterations or a satisfactory fitness level, is met.

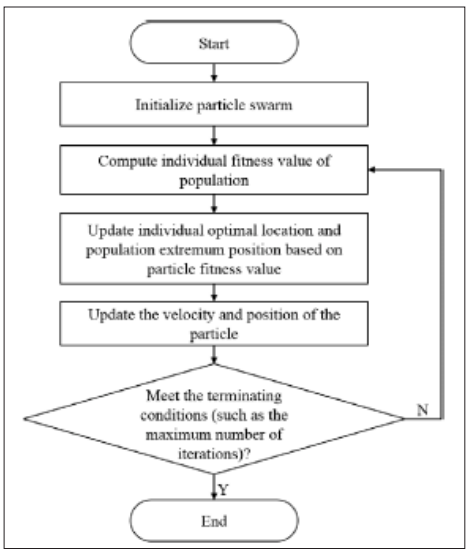

Figure 9: Standard PSO Flowchart Process

Bio-Insoired RL-MTD Modeling

In this section, we delve into the design of the RL-MTD strategy through an integrated approach using a Genetic Algorithm with Reinforcement Learning (GA-RL) and Particle Swarm Optimization with RL (PSORL). We will elucidate how the components of each algorithm - GA-RL, and PSO-RL - are adapted to our specific context of generating an optimized search space for our RL-MTD algorithm to perform better in generating secure configuration. This is followed by a detailed flowchart presentation, illustrating how these individual strategies are operationalized.

Elements Representation for GA-RL, PSO-RL

Before we dive into the algorithm flow of how MTD is realized using each of these methods, we will first show the element representation used for each bio-inspired algorithm.

GA-RL Figure 10 illustrates how the Genetic Algorithm's (GA) components are represented to be integrated into the RL-MTD(RL) framework. In this model, we have:

- **Gene:** It's an initial random *lim* int value used to define the Lower Limit (LL) and Upper Limit (UL) of a search space corresponding to the datatype of setting(P) as mentioned in section 3.
- **Chromosome:** It's an individual search space whose range is composed be gene values.
- **Population:** It is a pool made up of different RL-MTD agents each characterized by its search space range. (where each RL agent acts as a chromosome)

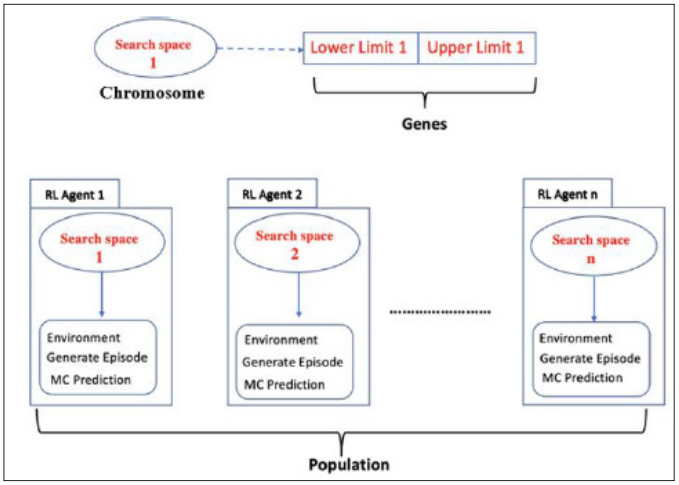

Figure 10: Representation of Genetic Algorithm Elements in Our GA-RL Formulation for MTD.

Note: The search space is considered a chromosome, its range limits (Lower limit, Upper limit) are its gene values. The population is a collection of different GA-RL Agents each with a different search space consisting of the configuration for the same misconfigured SUT.

The objective is to evolve these different search spaces through the GA with multiple generations. Each RL agent draws from its unique settings search space range to generate random initial configuration states, which are then utilized by the RL algorithm. This algorithm encompasses environment setup, episode generation, and Monte Carlo (MC) prediction functionalities. The initial population consists of a variety of RL agents. Through the iterative processes of Selection, Crossover, and Mutation, this population undergoes evolution across generations. At the culmination of these generations, we identify and select the most effective RL agent from this pool. The selection criteria focus on the agent that achieves the maximum rewards or creates the most secure configurations.

A step-by-step construction of incorporating GA into our RL -MTD is mentioned below. We follow the logic for Selection and Crossover steps similar to the neuroevolution algorithm described in [19]:

1. Treat the *search space* the agent draws from as a chromosome. (We will address the agent's search space as an entire agent for ease of use and consistency throughout the paper)
2. Agent's parameter, *lim* of search range will act as its genes.
3. The fitness score of secure configurations generated will act

as the chromosome's fitness (i.e. higher the fitness score, the higher the likelihood of survival).

4. The first iteration starts with n number of agents (search range), all with randomly initialized parameters.
5. **Selection:** By pure chance, some of them will perform better than others. The survival of the fittest option is then implemented by simply excluding the weakest agents from consideration and considering a certain percentage of agents.
6. **Crossover:** It is quite risky to swap parameters/genes for the simple reason that it might disturb the best-performing agents' search space range limits. Hence, we would rather replicate the selected agents for the next iteration until we reach n agents again for the next iteration.
7. **Mutation:** We modify agents produced during the crossover step, by adding or subtracting a small noise (value) to its parameter(limit). This step ensures we get to explore the neighborhood around the parameters of the best agents in the next iteration.
8. To secure the best agents from a probable reduction in performance due to the mutation step, we decided to keep the top-performing agent as is (without adding noise).

PSO-RL

The components in our PSO-RL model as shown in Figure 11 are as follows:

- **Particles / Swarm:** In analogy to Figure 10, a particle corresponds to a chromosome in GA, representing a limit lim used to define the search space range. Similarly, akin to the population in GA, a swarm, comprised of a collection of particles, represents a list of limit values defining various search space ranges, thus forming a swarm of distinct RL agents.
- **Particle Position:** Particle represents an individual search space. We denote the particle's position in terms of its fitness. Here fitness is measured by the performance of the RL-MTD model when supplied with that particle.

  Fitness(particle) = performance (RL-MTD[particle)]

  where performance is the agent's average performance (fitness score) over 60 episodes.
- **Global Best:** In its simplest form, global best refers to the best value of a fitness score among a set of particles. Each particle updates its position at the final stage of the search space exploration, with the best position being identified as the global best.
- **Particle Velocity:** This represents how far a particle position is from the ideal position (fitness). This indicates the moving rates for each particle within the search space. Our study takes advantage of this part to determine the distance between the best particle position and the ideal secure position.

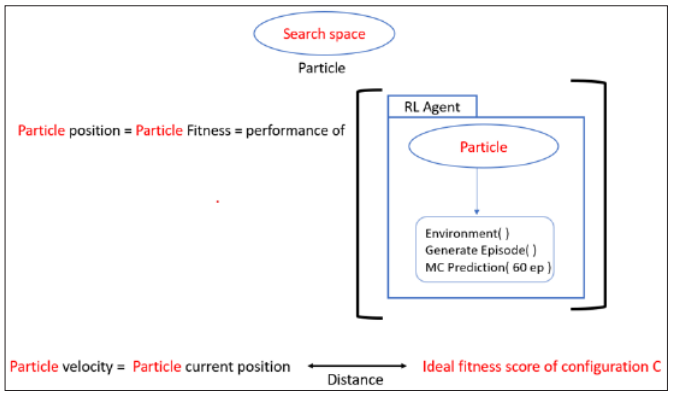

Figure 11: PSO elements in our PSO-RL Formulation for MTD

Note: Representation of Particle, particle position and particle velocity used in PSO-RL MTD.

Just like GA, the goal here is to find the most optimal particle (search space) for our RL-MTD algorithm. The PSO-RL algorithm uses the particle's position, velocity, and global best and is run for 100 generations as per the standard PSO flowchart shown in Figure 9 to find the most optimal search space for the RLMTD model to generate more secure configurations.

Bio-inspired MTD Algorithm Design

In this section, we describe the E-RL algorithms for finding the best-performing agent (search space) to generate a more secure configuration.

GA-RL

Algorithm 2 shows how the GA-RL was adapted to find the best RL agents in generating secure configuration. *(https://github.com/paraschopra/deepneuroevolution/blob/master/openai-gym-cartpoleneuroevolution.ipynb)*

Algorithm 2

GA_RL Algorithm to find best RL agents

Procedure perform_GA() // Initialize n number of agents
1. num_agents ← n
2. agents ← generate_random_agents(num_agents)
3. top_limit ← k // # of top agents to consider as parents
4. For gen in range(X) // Run evolution for X generations
a. rewards ← run_agent(agents) // Return rewards of agents
b. selected_agents ← Select agents of top k rewards // Selection
c. children_agents ← Randomly choose k-1 agents from selected_agents // Crossover
d. mutated_agents ← Mutate(children_agents) // Children agents after mutation
e. agents ← mutated_agents // Replace all agents with mutated children
5. End For
6. Best_agent ← Select agent from agents with maximum reward
7. Return Best_agent
End Procedure

Algorithm 2 follows the pseudo-code mentioned in the previous section.

- With population size set to *n (num_agents)*, we generate agents randomly in the first iteration using the function: *generate_random_agents* described in Algorithm 3.

- We set the maximum number of generations to run the loop to X.
- The *run_agent* function is used in each generation to run all randomly generated agents and get their performance (mean fitness score). (Algorithm 4)
- ***Selection:*** Out of *n,* select only top *k* as parents *(top_limit)* where *k<n*
- ***Crossover:*** As mentioned before, we replicate the selected agents instead of swapping parameters. Among top *k* parent agents, *k-1* agents are randomly chosen to make children for the next iteration.
- ***Mutation:*** In the mutate function, we add or subtract a small noise(value) to the parameter if the value of the agent is greater than a random number. (Algorithm 4)
- After we have child agents as parents, we iterate over the loop again until all generations are done or we find a good performing agent.

Algorithm 3
Generate Agents

Procedure generate_random_agents(num_agents)
1.agents ← [ ]
2.For i from 1 to num_agents
a. lim ← Pick random number from range (1, N) // N is an integer
b. agents.append(lim)
3.End For
4.Return agents
5.End Procedure

In Algorithm 3, function *generate random agents* is used to generate *num agents* number of agents. Here the agent represents limit value *lim* which is an integer value randomly drawn between 1 and N (hyperparameter). This *lim* decides the search space range from which the agent picks from.

Algorithm 4
RUN AGENT

Procedure run_agent(agents)
1. reward_agents ← [ ]
2. For ag in agents do
a. rwrd = run_RL_MTD(ag) // Call the run_RL_mc function which takes the agent (i.e., lim) as input
b. reward_agents.append(rwrd)
3. End For
4. Return reward_agents
End Procedure

In Algorithm 4, the procedure run_agent takes a list of agents agents as input. For each agent *ag* in *agents* list, *run_RL_MTD()* (Figure 5) is called which takes the agent as input and runs the base RL-MTD model with the new search space (ag). The MC prediction (Algorithm 1) method in the RL-MTD model returns the agent's average performance (fitness score) over 60 episodes which is stored in *rwrd.*

Algorithm 5
MUTATION

Procedure mutate(children_agent)
1. mutate_agent ← children_agent
2. If child_agent > random.random()
a. mutate_agent -= noise // hyperparameter
3. Else
a. mutate_agent += noise
4. End If
5. Return mutate_agent
End Procedure

PSO-RL

In Algorithm 6, we initialize all the N particles in a swarm with random integers. Each particle's starting position will have the same random integer and the same goes for the particle's velocity. The optimal difference is the hyperparameter we experiment with which indicates the threshold value. This threshold value is to check how far the particle search space is from the optimal one in terms of fitness scores generated from their corresponding RL-MTD agents.

Algorithm 6
Initializing N Particles

Procedure initialize()
1. swarm ← Pick N random integers // These are the limit values representing each particle
2. For each particle in swarm
a. particle_position ← Start with random integer
b. particle_velocity ← Start with random integer
3. End For
4. global_best ← 0
5. generations ← 100
6. optimal_difference ← d
7. ideal_fitness ← Fitness Score Ideal C // The score of a fully secure configuration
End Procedure

The primary goal of Algorithm 7 is to calculate each particle's current position and update the *gbest* with the maximum value. The procedure *run_RL_MTD()* (base MTD-RL algorithm) is called for each value in the swarm. The obtained value is then contrasted with the preceding value. The global best receives the new value if the outcomes attained are the highest value.

Algorithm 7
Particle Position

Procedure maximum_particle_position()
1. For each particle in swarm
2. new_pp = run_RL_MTD(particle) // Call the run_RL_mc function
3. If new_pp > particle_position[particle]
a. particle_position[particle] = new_pp
4. End If
5. global_best = max(new_pp)
6. End For
7. Return global_best
8. End Procedure

The distance between the ideal fitness and the particle's current position (search space's fitness) is represented by particle velocity. The lesser the distance, the more optimal the particle as it makes the base RL-MTD model generate an almost secure configuration. In Algorithm 8, we are trying to calculate the current distance of the particle (line 3) and we update the particle's old velocity with the current one if the latter is less than the former even though it's still greater than optimal difference (line 4,5). This means it's slowly approaching optimal difference.

Algorithm 8
Particle Velocity Algorithm

Procedure minimum_particle_velocity ()

1. For each particle in swarm
a. new_pv = ideal_fitness - particle_position[particle]
b. If (new_pv > optimal_difference) and (new_pv < particle_velocity[particle])
i. particle_velocity[particle] = new_pv
c. End If
2. End For
3. Return new_pv
End Procedure

In Algorithm 9, *social_influence* seeks to improve the particle value in regard to the intermediate best particle via social influence in a particular generation. It attempts to deduct social influence value) or add it depending on whether the current particle is greater or less than the intermediate best particle, thereby trying to converge all the particles to the near-optimal particle.

Algorithm 9
Social Influence
Procedure social_influence() // Particle value learning to be closer to the best one via social influence
1. For each particle in swarm
a. If particle > best_particle
i. particle -= influence // Random float social influence value
b. Else
i. particle += influence
c. End If
2. End For
End Procedure

To determine the optimal particle (search space), Algorithm 10 which is the PSO algorithm is run over 100 generations.

Algorithm 10
Run PSO Generations
1. For each gen in generation // 100 iterations for finding the best limit_value
a. maximum_particle_position()
b. minimum_particle_velocity()
c. social_influence()
2. End For
3. Return particle

Experiments and Results
In this section, we elucidate the experimental setup for each algorithm used in implementing MTD: RLMTD, GA-RL and PSO-RL where we ran each of them on 4 SUT case studies and compared their performance results. We intend to compare each of these models to determine the most effective approach to generating secure configuration using the MTD approach.

Experiment Set Up
RL-MTD We implemented the algorithm using python 3.6 with libraries numpy and pandas. After much experimentation, we set the following hyperparameters:
• lim = 10 (section 3)
• parameter score = HIGH(secure): 800; LOW(not secure): 8 (Section 2.2.3)
• Threshold= any value in *range: (max_score - val,max_score)*

where *val* is a hyperparameter set to 800 and *max_score* is the ideal total fitness score of C where all parameters are securely set (goal state). In action policy, we discourage the agent to not choosing action change if the fit(C) falls in the threshold fitness range as the range indicates that all parameters of *C* are securely set except 1. Hence, ensuring the likelihood of the agent taking action 0 is high (0.8) in this case.

GA-RL
We implemented GA-RL algorithm using Python 3.6 with libraries *numpy* and pandas. After much experimentation, we set the following hyperparameters:
- number of agents n = 25
- number of top agents k = 5
- number of generations X = 100
- N = 25
- noise = 0.05

Fitness Function for Chromosome: To calculate rewards for different RL agents, we used the entire Rl-MTD algorithm as the fitness function which returns the average of the scores generated from the MC prediction function for the episode count for 20 and 60.

PSO-RL
We implemented PSO-RL algorithm using Python 3.6 with libraries *numpy* and pandas. We set the hyperparameters as follows:
- Swarm size = 30
- Particle values (lim): integers = [1....30]
- Initial Particle position = 0
- Idea Fitness = Maximum fitness score of the most secure configuration based on the selected SUT.
- number of generations = 100
- influence = 0.05

Given that configuration C varies across different SUTs due to its diverse parameters, it became evident that employing identical optimal difference and initial particle velocity values to be implemented for all case studies was inappropriate. This is because the values depended on the specific parameters and their quantity within each SUT's configuration. Consequently, we undertook an exploration of various optimal particle and initial particle velocity values to identify the most effective combinations tailored to each SUT. The ultimate goal is to recognize the optimum balance between Particle Velocity and the Optimal Difference. This leads to the attainment of maximum or highly improved results within the PSO-RL framework.

We elaborate on the ideal set of hyper-parameters examined for PSO, which comprises the Optimal Difference and Particle Velocity for each case study.

The ideal set refers to the optimal combination of parameters that led to the most secure and effective results in our experiments.

Considering these two hyper-parameter values as a tuple (Optimal Difference, Particle Velocity), our best combinations for each case study are as follows:
- Window 10: (20, 300)
- McAfee: (300, 500)
- Microsoft Excel 2016: (160, 200)
- Microsoft Office 2007: (120, 1000)

As mentioned in Section 5-1 and Figure 11, we deemed the search space to correspond to particle position and particle fitness.

We considered the initial search space for the first iteration to be set to zero.

Following running the experiment for over approximately 5000 generations, the optimal search space or optimal particle fitness in the most effective pairing of (Optimal Difference, Particle Velocity) for each scenario was presented as follows:

- Window 10: 34924.0
- McAfee: 7240.0
- Microsoft Excel 2016: 14812.0
- Microsoft Office 2007: 16800.0

Results on Case Studies

This section demonstrates the performance of all three models: RL-MTD, GA-RL, PSO-RL, in generating secure configurations and reports the results. Once we get the optimal search space range for both GARL and PSO-RL, we run the base RL-MTD with those optimized search spaces and capture their results. We then compare their performance with the base RL-MTD with no optimization.

We selected 4 SUT case studies and the corresponding parameters from the STIG website namely:

1. Windows 10 (59 parameters)
2. McAfee (14 parameters)
3. MS Excel (20 parameters)
4. MS Office (21 parameters)

These SUTs contain a good number of configuration parameters whose domains are diverse enough.

We executed our developed scripts for various number of episodes and captured the best fitness scores (i.e., value of state *V* that indicates how secure a given state is). Figures 12 illustrate the trend of fitness scores obtained through the episodes where the x-axis is the episode counts (i.e., between 20-500 episodes); whereas the y-axis holds the normalized values of fitness scores. More specifically, the normalized fitness score value of 0.0 represents the least secure attained by the agent; whereas the value 1.0 is the most secure fitness score. The normalization on fitness scores is performed as follows:

$$\text{normalized}(fs) = \frac{fs - \min(fs)}{\max(fs) - \min(fs)}$$

where *min(fs)* for a given fitness score *fs* is the minimum fitness score of the configuration which is equal to the total sum of the fitness scores for all parameters when they are all set to LOW. Similarly, *min(fs)* for a given fitness score *fs* is the maximum fitness score of the configuration which is equal to the total sum of the fitness scores for all parameters when they are all set of HIGH. More specifically,

$$\min(fs) = \sum_{S_i=1}^{n} \text{LOW} \quad ; \quad \max(fs) = \sum_{S_i=1}^{n} \text{HIGH}$$

where S_i is the i^{th} parameter and *n* is the total number of parameters.

We now analyze the results for each case study which is also summarized in Table 1.

Window 10 Case Study

Upon examining Figure 14a an initial observation reveals that RL-MTD demonstrates the least security performance. However, an instant competition unfolds between GA-RL and PSO-RL (labeled ERL). Ultimately, it becomes evident that in this context, PSO-RL outperforms GA-RL, although a slight difference can be depicted. Moreover, in this case study, PSO-RL outperformed 17 times in comparison to GA-RL.

McAfee Case Study

In this case study, substantial fluctuations are observed across all experiments; however, RL-MTD consistently exhibits inadequate performance compared to the other two methods. Upon initial glance of GA-RL and PSO-RL (labeled ERL) in Figure 14b, notable oscillations are evident. Furthermore, both PSO and GA equally performed better than base model for a count of 8 episodes. A distinct pattern emerges in these two methods, while they promote similar performance. Specifically, in corresponding episodes, both methods experience periods of insecurity, followed by episodes where performance improves. For instance, in episode number 40, the performance of neither algorithm is remarkable, but in the subsequent episode, a notable improvement can be illustrated.

Microsoft Excel 2016 Case Study

In this case study, illustrated in Figure 14c, RL-MTD and GA-RL (labeled ERL) display predominantly similar and fluctuating behaviors. The frequency of episodes where each outperforms the other is comparable. Although RL-MTD may exhibit better performance in specific episodes, the reverse occurs for GA.

Despite the oscillations perceived in PSO-RL as a third graph, it consistently surpasses the other two methods in most instances. Moving to the episode counts analysis, it becomes evident that the results are outstanding when employing the PSO-RL model in the Microsoft Excel 2016 case study.

In the majority of episodes, PSO-RL demonstrates upper-level performance compared to other methods.

Microsoft Office 2007 Case Study

As depicted in Figure 14d, a clear distinction is evident in the performance between RL-MTD and Bioinspired methods (labeled ERL), including GA-RL and PSO-RL, upon initial glance.

Although RL-MTD attempts to reach Bio-inspired methods, within one episode, there is a substantial difference in subsequent episodes. A slight discrepancy is observed by moving to the GA-RL and PSO-RL methods. In most episodes, GA-RL and PSO-RL demonstrate approximate performance.

However, GA-RL slightly exhibits less security in a few instances. It is evident that Bio-inspired methods which are GA-RL and PSO-RL, achieve the highest level of security performance in almost all episodes iteratively. This underlines that both GA-RL and PSO-RL attain higher-rank secure performance.

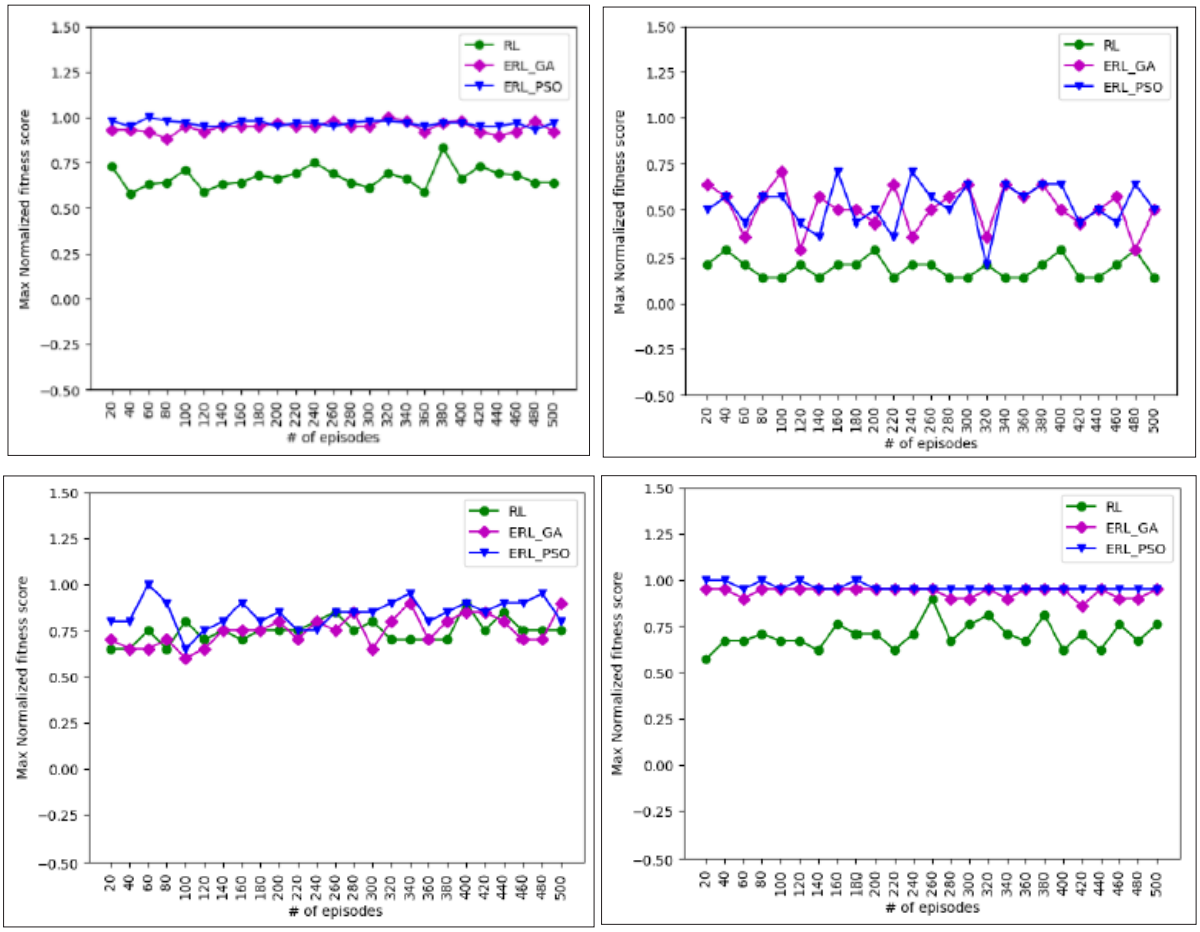

Figure 12: Performance Comparison of the 3 Models: MTD-RL(labeled RL), GA-RL (ERL-GA), and PSO-RL (ERL-PSO) in generating secure configurations for 4 case studies. The x-axis shows the number of episodes each of the models were trained on and the y-axis shows the normalized fitness each of these models was able to achieve. 1 being the max fitness score: (all parameters securely set) and 0 being the most insecure. From the trend, we see both bio-inspired algorithms outperformed the base RL-MTD model for most of the SUTs reaching almost the most secure config across all episodes. However, between GA-Rl and PSORL, it's a close call but as per our analysis in Table 1, PSO-RL was the best performing model by a margin.}

Table 1

This table summarizes the results/analysis (Figure 12) of the 3 models on each SUT and shows PSO-RL was the clear winner among all 3. We considered 25 episodes for each model namely, RL-MTD, GA-RL, and PSO-RL. The values in tuples illustrate the total count of episodes in which each model outperformed the rest two. For instance, for the Windows 10 case study, PSO-RL outperformed others 17 times, while GARL did better 6 times. Having said that, in contrast to PSO-RL and GA-RL, base RL-MTD did not achieve better in generating configuration far nay SU. When the cumulative number of episodes is less than 25, the rest of the episodes show equal values between the methods.

Case Studies	Count of Episodes (RL, GA-RL, PSO-RL)	Best Algorithm
WINDOW 10	(0, 6, 17)	PSO-RL
MCAFEE	(0, 8, 8)	GA-RL, PSO-RL
MS EXCEL 2016	(0, 2, 21)	PSO-RL
MS OFFICE 2007	(0, 0, 12)	PSO-RL

Attack-Defense Model Demo Using Our MTD Approach

For demonstration purposes, consider a **Windows 10** installation on a PC, where the attack surface is defined by three key parameters:

- **ACSettingIndex:** Enforces automatic screen locking (e.g., a short display timeout) to mitigate unauthorized wakeups or physical access risks.
- **AllowBasic:** Controls whether Windows allows remote logins using simple, unencrypted passwords, a method similar to writing your password on a postcard.
- **AllowDigest:** Controls whether Windows permits slightly more secure, but still weak, remote logins that use scrambled passwords.

Reconnaissance

Imagine an attacker targeting the Windows 10 PC. At time **t**, the attacker performs reconnaissance and discovers that the following three parameters are misconfigured:

- **ACSettingIndex = 0:** This results in overly long screen timeout values, increasing the potential for unauthorized access when the system is left unattended.
- **AllowBasic = 1:** This setting enables remote login with unencrypted passwords, which can be exploited by attackers to reuse credentials and break into the system.
- **AllowDigest = 1:** This enables digest authentication, which is vulnerable to credential replay attacks.

These misconfigurations create a vulnerable attack surface, allowing the attacker to exploit the combination of vulnerabilities v1, v2, and v3. This leads to the formulation of an attack strategy A, based on the identified vulnerabilities. A can be ransomware deployment, data exfiltration, etc

Defense Approach Using MTD

Most system administrators (sys admins) are reluctant to change default settings, as doing so can interfere with system functionality and user experience [20]. This resistance often results in the persistence of a vulnerable attack surface (v1, v2, v3), which attackers can exploit for a successful attack **A**.

To mitigate this, we propose using PoC Evolutionary MTD (Moving Target Defense). This defense mechanism dynamically shifts the attack surface by changing the configuration of the three parameters, effectively changing the vulnerabilities at time **t+1**. As the system continuously evolves, it forces the attacker to deal with an ever-changing landscape, making the attack futile.

For example, with each change in the configuration, the vulnerabilities v1, v2, and v3 transform into v1', v2', and v3', creating new but different vulnerabilities which are no longer exploitable to the attacks **A** as the types/properties of the vulnerabilities changed causing the attack to fail. By constantly shifting these settings over t+2, t+3 constantly to outsmart the attackers with new exploits, vulnerabilities also change without being static. Hence the MTD defense mitigates attacks like ransomware deployment, data exfiltration, and others that depend on static vulnerability combinations.

Deployment Process: Overlay Configuration Setting

This defense is not implemented directly at the OS level but through an **overlay configuration setting stage**, called **authoring**. During this phase, sys admins create different configuration profiles, choosing which parameters to harden. These profiles can **include fake configuration keys** that act as an intermediary attack surface, shielding the original surface and misleading attackers.

Mechanism of Evolutionary RL-MTD Defense

The evolutionary RL-MTD algorithm optimizes the attack surface through a dynamic search space, which it learns using **PSO-RL (Particle Swarm Optimization-Reinforcement Learning)**. For the **ACSettingIndex**, the system creates **decoy registry entries** with similar names but different fake values that fetches from the optimized search space through the profiling stage near the original keys, making it harder for attackers to discern legitimate settings. This approach not only disrupts the attacker's ability to exploit the initial vulnerability but also continuously alters it. The following command is used to create a decoy key with a dummy path where the fake value is assigned in d0 (placeholder) by our algorithm periodically.

reg add "HKLM\Software\DummyPath\ACSettingIndex" /v "FakeValue" /t REG_DWORD /d 0

The algorithm performs the same transformation for the other two parameters—**AllowBasic** and **AllowDigest**—by selecting random values from their respective domains, adding an additional layer of confusion for attackers. Figure 13 also shows how you can create duplicate variables graphically.

Results: Wasteful Effort for Attackers

Through continuous profiling and dynamic changes in the settings, the evolutionary MTD approach renders the attacker's efforts futile by directing them toward **fake keys**. The attacker wastes significant time and resources analyzing and attempting to exploit these decoys, effectively neutralizing the potential for a successful attack.

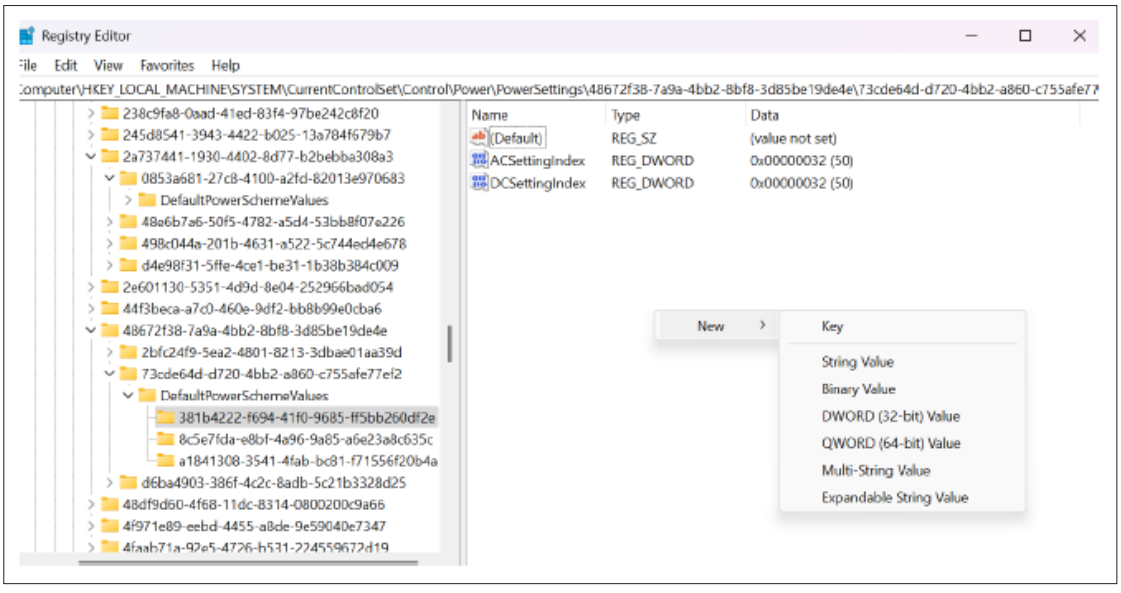

Figure 13

Note: Shows how you can create decoy variables for ACSettingIndex in Windows 10

Discussions

As seen in the results, the base RL-MTD approach exhibits less secure performance across all case studies. The incorporation of the optimal search space derived from both GA and PSO significantly enhanced the performance of our base RL-MTD model compared to its performance without the optimal search space. Interestingly, our analysis revealed that while there was a notable performance improvement when utilizing the optimal search space from either GA or PSO, the difference in performance between GA and PSO was marginal. This suggests that both GA and PSO were effective in searching for an optimal search space to enhance the base model's performance. However, **the clear winner was PSO-RL**. Our findings underscore the efficacy of both optimization techniques in facilitating the identification of an optimal search space conducive to generating more secure configurations within the RL framework.

In some illustrations, such as Windows 10 and Microsoft Office 2007, minor improvements prevailed. Nevertheless, PSO ultimately proves better results in other cases like Microsoft Excel 2016, despite slight fluctuation. It should be noted that in certain case studies, such as McAfee, significant performance fluctuations lead to equal outcomes for GA and PSO. A key point to consider is that in each case study, optimization drives experimentation to enhance outcomes and performance.

Note: As previously noted, the application of MTD to address security challenges in misconfigured software is relatively novel, resulting in the absence of an established benchmark for comparison in existing literature. This study aims to be viewed as the construction of a proof-of-concept (POC), showcasing the potential of an MTD defense strategy from the defender's standpoint in generating dynamic secure configuration. It is crucial to acknowledge that further research and refinement are imperative to bolster this strategy and guarantee its overall effectiveness.

Related Work

Examining the relevant literature closely reveals that there hasn't been adequate discussion of the intended motivation. By scrutinizing the related work can be noticed that the targeted motivation has not been addressed much. C. Gao and Wang countered the moving target defense method by implementing Reinforcement Learning to address the DDoS attack [21]. The findings of their experiments advocate that the reinforcement learning adjustment on MTD affects the results more successfully and enhances the algorithm performance. T Zhang et al., generated the Moving target defense method draws on deep reinforcement learning to protect from cyber security attacks [22]. The Markov decision process (MDP)model was used to design the MTD technique to train on scanning behavior. The analysis of the developed model revealed that the scanning time was diminished effectively. Eghtesad et al. applied the reinforcement learning technique to ameliorate the MTD method performance in their study [23]. The authors evaluated the numerical results demonstrating the performance of the trained model. The outcomes depict that the developed model has the exceptional ability to recognize the optimal policies in a defined environment. Li et al. utilized the MTD technique to mitigate the potential threats and attacks that might be encountered in the container cloud environment [24]. The proposed model exploits the advantage of deep reinforcement learning along with the Markov decision process in the optimized MTD. The model demonstrates the improvement of the efficacy of defense significantly. The authors of John et al, implemented the Genetic Algorithm in Moving Target Defense to find the optimized secure configuration [25]. John et al. stated that the variety of configurations is potentially improved over time regarding the environment. The experiment findings confirm that the GA method is sufficiently capable of identifying the optimal secure configuration. Zhang, et al., introduced the Moving Target Threeway Evolutionary Game Defense Model in network security [26]. The model, which concentrates on offering adjustable defense decisions, combines evolutionary games and signal games. The authors employed the action and rewards strategy to optimize the defense. Findings and analyses reveal that the Monte Carlo simulation performs better compared to former designs. However, defending from various attack behaviors is still ongoing.

Conclusion and Future Work

In conclusion, our in-depth research tackles the pervasive challenge of security misconfiguration within software systems—a problem that leaves systems open to exploitation. By integrating bio-inspired algorithms, specifically the Genetic Algorithm (GA) and Particle Swarm Optimization (PSO), into our previously established RL-MTD model, we've advanced the model's proficiency in continuously generating secure and dynamic configurations. Through rigorous comparative analysis, we've ascertained that both the GA-RL and PSO-RL enhancements not only refine the search space for potential configurations but also surpass the original RL-MTD framework in their ability to produce robust configurations against an array of software systems put to the test (SUTs). Noteworthy is the slightly superior performance of PSORL in the majority of these scenarios, making it the best-performing model due to a nuanced edge in its search strategy. Our findings represent a significant contribution to the domain of proactive cybersecurity measures. By employing Moving Target Defense (MTD) enriched with machine learning and bio-inspired algorithms, we present an innovative and efficacious strategy to proactively MTD defense strategy from a defender perspective, thereby reinforcing the security posture of software configurations against the dynamic landscape of cyber threats.

For future work, we envision expanding our research to transform the current game model into a more complex and realistic scenario. The next step is to develop a two-player version of the game, where one player is the defender, maintaining secure configurations, while the other acts as an attacker, probing and exploiting vulnerabilities. Additionally, since software applications do not operate in isolation but interact with each other, it is crucial to examine these interactions and how they may impact the Moving Target Defense (MTD) strategy. Furthermore, we will run our proposed conceptual MTD framework and measure its efficiency in terms of the perspectives of an attacker and a defender at real time together in ab bigger setup with all SUTs. The attacker’s metric estimates its attack performance, indicating that the attacker’s high performance refers to the defender’s low performance, and vice versa. The defender’s metric measures its performance in achieving security and defense goals of a given system.

(1) Attack success probability (ASP): This metric refers to the probability that attacks are successfully performed. In our case, it refers to the probability that an attack surface is compromised or freezes by an attacker.

(2) Defense success probability (DSP): This metric measures the success of the defender agent in the MTD model by its ability to avoid vulnerability chains, attaining low score interruptions score (QoS high) and number of times it reached the terminal state (secure configuration state).

We also need to study how different configuration parameters influence each other. Ultimately, this research is a stepping stone towards a deeper understanding of how to protect software from misconfiguration threats using MTD, aiming to build a more resilient digital infrastructure.

References

1. Warren T (2021) Twitch news: Data leak breach and security confirmation. Available: https://www.theverge.com/2021/10/6/22712365/twitch-data-leak-breach-securityconfirmation-comments.
2. OWASP (2021) OWASP Top 10. Available: https://owasp.org/www-project-top-ten/.
3. OWASP Top 10 Team (2021) Security misconfiguration. Available: https://owasp.org/Top10/A05_2021Security_Misconfiguration/.
4. DHS (2011) Department of Homeland Security. Available: https://www.dhs.gov/science-and-technology/csd-mtd.
5. Cho JH, Sharma DP, Alavizadeh H, Yoon S, Ben-Asher N, et al. (2020) Toward proactive, adaptive defense: A survey on moving target defense. IEEE Commun Surv Tutor 22: 709-745.
6. Dass S, Siami Namin A (2021) Reinforcement learning for generating secure configurations. Electronics 10: 2392.
7. Mordahl A (2023) Automatic testing and benchmarking for configurable static analysis tools. In: Proc 32nd ACM SIGSOFT Int Symp Softw Test Anal pp: 1532-1536.
8. Wang T, He H, Liu X, Li S, Jia Z, Jiang Y, Li W (2023) Conftainter: Static taint analysis for configuration options. In: Proc 38th IEEE/ACM Int Conf Autom Softw Eng (ASE) Pp: 1640–1651.
9. Jalowski Ł, Zmuda M, Rawski M (2022) A survey on moving target defense for networks: A practical view. Electronics 11: 2886.
10. Sutton RS, Barto AG (2018) Reinforcement Learning: An Introduction. MIT Press.
11. OpenAI (2019) Gym. Available: https://openai.com/research/openai-gym-beta.
12. Sanyal S (2023) An introduction to particle swarm optimization (PSO algorithm). Available: https://www.analyticsvidhya.com/blog/2021/10/an-introduction-to-particle-swarm-optimization-algorithm.
13. Lambora A, Gupta K, Chopra K (2019) Genetic algorithm—A literature review. In: Proc 2019 Int Conf Mach Learn Big Data Cloud Parallel Comput (COMITCon) pp: 380-384.
14. Mirjalili S (2019) Evolutionary algorithms and neural networks. Stud Comput Intell 780: 43-53.
15. Huang W, Xu J (2023) Optimized Engineering Vibration Isolation, Absorption and Control. Springer.
16. Song MP, Gu GC (2004) Research on particle swarm optimization: A review. In: Proc 2004 Int Conf Mach Learn Cybern 4: 2236-2241.
17. Papazoglou G, Biskas P (2023) Review and comparison of genetic algorithm and particle swarm optimization in the optimal power flow problem. Energies 16: 1152.
18. Gao H, Yang Y, Zhang X, Li C, Yang Q, Wang Y (2019) Dimension reduction for hyperspectral remote sensor data based on multi-objective particle swarm optimization algorithm and game theory. Sensors 19: 1327.
19. Chopra P (2019) Neorevolution blog. Available: https://towardsdatascience.com/reinforcement-learning-without-gradients-evolving-agents-using-genetic-algorithms-8685817d84.
20. Dietrich C, Krombholz K, Borgolte K, Fiebig T (2018) Investigating system operators' perspective on security misconfigurations. In: Proc 2018 ACM SIGSAC Conf Comput Commun Secur pp: 1272-1289.
21. Gao C, Wang Y (2021) Reinforcement learning based selfadaptive moving target defense against DDoS attacks. J Phys Conf Ser 1812: 012039.
22. Zhang T, Xu C, Shen J, Kuang X, Grieco LA (2023) How to disturb network reconnaissance: A moving target defense approach based on deep reinforcement learning. IEEE Trans Inf Forensics Secur 18: 5735-5748.
23. Eghtesad T, Vorobeychik Y, Laszka A (2020) Adversarial deep reinforcement learning based adaptive moving target defense. In: Decision and Game Theory for Security: 11th Int Conf, GameSec Proc 11 pp: 58-79.
24. Li Y, Hu H, Liu W, Yang X (2023) An optimal active defensive security framework for the container-based cloud with deep reinforcement learning. Electronics 12: 1598.
25. John DJ, Smith RW, Turkett WH, Cañas DA, Fulp EW (2014) Evolutionary based moving target cyber defense. In: Proc Companion Publ pp: 1261-1268.
26. Zhang Z, Liu L, Zhang C, Ren J, Ma J, Wang L, Liu B (2024) MTTEGDM: A Moving Target Evolutionary Game Defense Model Based on Three-Way Decisions. Electronics 13: 734.