electronics

MDPI

Article

Reinforcement Learning for Generating Secure Configurations

Shuvalaxmi Dass and Akbar Siami Namin *

Department of Computer Science, Texas Tech University, Lubbock, TX 79409, USA; shuva93.dass@ttu.edu
* Correspondence: akbar.namin@ttu.edu

Abstract: Many security problems in software systems are because of vulnerabilities caused by improper configurations. A poorly configured software system leads to a multitude of vulnerabilities that can be exploited by adversaries. The problem becomes even more serious when the architecture of the underlying system is static and the misconfiguration remains for a longer period of time, enabling adversaries to thoroughly inspect the software system under attack during the reconnaissance stage. Employing diversification techniques such as Moving Target Defense (MTD) can minimize the risk of exposing vulnerabilities. MTD is an evolving defense technique through which the attack surface of the underlying system is continuously changing. However, the effectiveness of such dynamically changing platform depends not only on the goodness of the next configuration setting with respect to minimization of attack surfaces but also the diversity of set of configurations generated. To address the problem of generating a diverse and large set of secure software and system configurations, this paper introduces an approach based on Reinforcement Learning (RL) through which an agent is trained to generate the desirable set of configurations. The paper reports the performance of the RL-based secure and diverse configurations through some case studies.

Keywords: secure software configuration; moving target defense; reinforcement learning

Citation: Dass, S.; Siami Namin, A. Reinforcement Learning for Generating Secure Configurations. *Electronics* **2021**, *10*, 2392. https://doi.org/10.3390/electronics10192392

Academic Editor: Nurul I. Sarkar

Received: 28 August 2021
Accepted: 24 September 2021
Published: 30 September 2021

1. Introduction

We live in a world where our lives are predominantly dependent on machines and the Internet. From activities as trivial as accessing social media for entertainment to as confidential as storing our bank account details, we rely on the machines and a wide-ranging software that cater to our different needs. Such heavy dependency makes these machines rich in user data to such an extent that they become prone to all sorts of cyber attacks where the attackers search the network for vulnerable machines [1]. In order to defend the security of the target machines, most people will try to install various heavy priced anti-virus software or download their upgraded patches to prevent some vulnerability. These defense actions can turn futile especially when attackers are constantly updating their attack mechanisms.

The inclining complexity of software applications encourages designers to consider integrating third-party APIs, tools, and utilities to their systems with the goal of minimizing the risk of failures and mitigating vulnerability exposures. Given the enormous number of features offered by these packages, libraries, and even operating systems, the integration process may need additional effort in testing the combined system, also called configuration and integration testing. The strategy employed in conducting configuration and integration testing often targets detecting conventional defects. As a result, scrutinizing some other aspects of the integrated system, such as security, may not be addressed properly.

One of the key risk factors affecting security of software systems is *misconfiguration*. Improperly configured software lead to a multitude of vulnerabilities, which makes the underlying system prone to various types of attacks. Different classes of vulnerabilities pertaining to varied software systems can be found in National Vulnerability Database (NVD) [2]. The database reports the Common Vulnerability Scoring System (CVSS) scores, an open and free standard framework, which rates the vulnerabilities based on their severity.

In the case of vulnerabilities that are exposed due to poorly configured applications or systems, where configuration parameters govern the proper functionality of the systems, it is not only tedious but also impractical for a system administrator to manually identify and fix parameters that are misconfigured. Moreover, the administrator might not be aware of the problematic parameters that are causing the vulnerability. This delay in defense time could result in successful execution of a launched attack leading to a compromised target system. The Open Web Application Security Project introduces "*Misconfiguration*" among the top 10 Web application security risks [3].

A potentially effective approach is to continuously change the configuration of systems with the goal of rapidly changing the attack surface and thus confusing the attacker during the reconnaissance stage. More specifically, by rapidly changing the settings and configurations of a given system, the attacker's data that are collected during the exploration will be invalidated, hurdling the attackers of launching successful attacks and exploiting outdated vulnerabilities. However, the challenging issue is the generation of a set of "*diverse*" and "*good*" configurations to shape the settings of the underlying system on a random basis. The former (i.e., diverse) refers to the diversity of configurations produced in order to prevent prediction of configurations; whereas, the latter (i.e., good) is designated to measure the security posture and level of the proposed configurations (i.e., "*adequate security*").

A game-changing approach to cyber security, called Moving Target Defense (MTD), has emerged as a potential solution to the security challenges associated with the static nature of vulnerable software system [4]. MTD is interpreted as a strategy by which the underlying system is constantly changing to reduce or shift the attack surface available for exploitation by attackers. We view the attack surface of a misconfigured system as the result of improper settings of system parameters that together can introduce vulnerabilities that can be exploited by attackers.

A platform enabling MTD can help in invalidating the data collected by an attacker during the reconnaissance stage. However, in addition to physical implementation and deployment of an MTD-based platform, the *generation*, *goodness of security*, and the *diversity* of configurations still remain large and open challenging problems.

In this paper, we introduce a proof-of-concept approach to generate a set of secure configuration for any given system using Reinforcement Learning (RL). In the model, an agent is trained to learn about the permissible values of parameters that lead the system to be more secure.

In this work, we develop a MTD framework-inspired single-player game prototype using Reinforcement Learning (RL), where we train an agent to generate secure configurations. We then demonstrate the effectiveness of the RL game prototype by applying it on a misconfigured Windows 10 system as a case study. We will be using Monte Carlo Prediction in RL method to assess the agent's performance.

This paper is structured as follows: the literature of this line of research is reviewed in Section 2. Section 3 presents the methodology of how RL is adapted to formulated MTD through a game. The application of RL-based MTD game on Windows case study is presented in Section 4. The algorithm details are presented in Section 5. The implementation details are provided in Section 6. The results are present in Section 7. A discussion about our reinforcement learning-based approach in comparison with some other techniques to this problem is presented in Section 8. The conclusion of the paper and some future work are provided in Section 9.

2. Related Work

2.1. Moving Target Defense

Moving Target Defense is a game changing idea that was introduced in the National Cyber Leap Year Summit in 2009 [4]. The basic idea is to carry security through diversification by continuously changing the configuration of a system. Such dynamic changing of the settings of a system invalidates the data collected about a system and thus prevents

attackers in utilizing the information they have captured about the target system during the reconnaissance stage.

There exist several research works that utilize MTD-based strategies to protect computers on a network such as IP address randomization, virtualization, decoy-based deployment models, software defined networking-based infrastructure, and lightweight MTDs [4] through which the dynamic changing of operational behavior of the underlying target system is changing.

It is important to deploy MTD-based systems on a flexible, mobile, and programmable environment so the operations are automated and performed in a more efficient way. Software Defined Networking (SDN) is a transformative paradigm in networking that can enable us to create such a flexible and automated platform for MTD systems. In SDN architecture, the control logic and decision are separated from the network devices, enabling us to have more control on the flow of packets across the network through programming.

Hyder and Ismail proposed an Intent-based Moving Target Defense (INMTD) framework in which Software Defined Networks (SDNs) are utilized [5]. In their framework, the concept of shadow servers is used to counter the reconnaissance stage of cyber-attacks where the servers running in SDN networks are the prime target.

Recently, the use of MTD in addressing security challenges of IoT networks has been explored [6]. Mercado et al. [7] introduce a MTD-based strategy for implementing Internet of Things (IoT) cybersecurity. The proposed strategy randomly shuffles the communication protocols to enable communication of a particular node to the gateway in an IoT network. The framework intends to make a balance between the cost associated with performance overhead, business impact, and at the same time reduce the likelihood of at attack being successful. The MTD strategy parameters will be identified after several iterations.

Although the deployment techniques developed for implementing MTD-based systems are needed, these techniques cannot be so effective if the new configuration settings are not secure enough or there are some redundant configurations that can enable attackers to estimate the next configuration settings. Therefore, it is crucial to strengthen the operation and thus effectiveness of MTD-based systems through generating a good number of candidate secure configurations automatically. For a more complete discussion of the state-of-the-art of Moving Target Defense (MTD), please refer to [4].

2.2. Secure Configuration Generation

The problem of identifying a set of secure and diverse configurations automatically is essentially a search problem. Accordingly, several techniques borrowed from machine learning, genetic algorithms, and input fuzzing have been adapted to address this problem.

Dai et al. [8] proposed the concept of "*configuration fuzzing*" for the purpose of examining vulnerabilities that emerge only at certain conditions. These conditions were created by dynamically changing the configuration of the running application at specific execution points. The authors have employed different approaches to auto-tune the configurations of a SUT in order to optimize the performance under a specific workload.

Crouse and Fulp [9] and John and Fulp [10] used genetic algorithm (GA) to implement a Moving Target Defense (MTD) platform that makes computer systems more secure through temporal and spatial diversity in configuration parameters that govern how a system operates. They also further introduced mutation changes to GA namely, Parameter Values Mutation (PVM) where mutation operator mutates the parameter values based on its type (integer, option, and bit) and Parameter Domain Modifier (PDM) where Mutation operator modifies the domain of the parameter by eliminating the insecure setting from the parameter's domain. The performance of these two genetic algorithms (GA + PDM and GA + PVM) were then compared and contrasted on the basis of generating secure configuration for Moving Target Defense.

Bei et al. [11] proposed a random forest-based configuration tuning, called RFHOC, to automatically recommend the configurations for tuning the Hadoop which optimizes

the performance for a given application running on a given cluster. The authors also employ evolutionary algorithm like Genetic Algorithm (GA) to actively search the Hadoop configuration space.

Zhu et al. [12,13] investigated the challenges in tuning the right configuration for systems and formalized the challenges into the Automatic Configuration Tuning with Scalability (ACTS) problem. The ACTS problem is to find a configuration setting for a System Under Tune (SUT) that optimizes the SUT's performance under a specific workload, given the limited resource. The authors proposed an ACTS solution by implementing a flexible architecture, which provides easy incorporation of various SUTs, deployment environments, workloads, and scalable sampling methods and optimization algorithms.

Zhang et al. [14,15] proposed an end-to-end automatic configuration tuning system for cloud databases called CDBTune. It uses the reinforcement learning policy-based Deep Deterministic Policy Gradient (DDPG) algorithm, which is a combination of Deep Q Network (DQN) and actor-critic algorithm to tune the configuration settings for improving the performance of cloud databases.

Bao et al. [16] approached the configuration tuning problem in an unconventional manner. They proposed Automatic Configuration Tuning using Generative Adversarial Network (ACTGAN) tool to generate potentially better configurations by focusing on capturing the hidden structures of good configurations instead of improving the performance estimation of a given configuration.

Our motive is to build a proof-of-concept for auto-tuning the configurations in order to optimize the security of the underlying system irrespective of any workload. In this paper, we employ a Monte-Carlo based Reinforcement Learning technique. To the best of our knowledge, RL has not been adapted in the domain of generating secure configurations to auto-tune the configurations and ensure the security of software platforms.

3. Problem Formulation

This section provides a motivating example and then demonstrates the proposed approach designed for the problem.

3.1. A Motivating Scenario

Consider a host machine with a misconfigured underlying software system in an organization whose vulnerabilities are exposed to adversaries. The adversary is anticipated to launch attacks such as: Brute force/credential stuffing, Code injection, Buffer overflow, Cross-site scripting (XSS) to exploit the vulnerabilities caused by the misconfiguration.

This problem depicts the interaction between a misconfigured software present in the host machine and the attacker. Through the scanning acquisition stage (i.e., the reconnaissance stage), the attacker gains certain knowledge about the configuration settings that are causing the vulnerabilities in the targeted host. The attacker then uses the acquired knowledge to construct an exploit. In order to prevent attacks, it is crucial to make the acquired knowledge futile. Such invalidation of acquired data can be achieved through changing the configuration settings frequently. This way attackers will have a hard time to keep up with a constantly changing attack surface. As a result, the attackers will not have sufficient time and knowledge to craft an effective exploit.

In solution to the aforementioned problem, we need to devise a MTD-based strategy that a set of not only secure but also diverse configurations are generated. The intent is to create a dynamic environment by constantly changing the underlying vulnerable system configurations towards secure configuration settings, so that an attacker acting on false or constantly changing information is confused.

3.2. Why Reinforcement Learning for Generating Secure Configuration?

There exists a great number of techniques to generate a set of populations that have similar properties. However, the choice of technique employed for a problem explicitly depends on the nature of the problem being solved. Evolutionary algorithms and techniques

such as Genetic Algorithms (GA) and Particle Swarm Optimization (PSO) [17,18] have already been applied to secure configuration problems. In addition to these evolutionary algorithms, it is possible to adapt machine learning-based techniques to the problem and understand the pros and cons of these approaches. However, as mentioned earlier, the type of techniques employed for addressing the problem relies on the nature of the problem. Therefore, it is important to understand what machine learning techniques are and how they can be adapted for generating secure configurations.

Generally speaking, there are three types of machine learning techniques [19]: supervised learning, unsupervised learning, and reinforcement learning. In supervised learning, the machine learning module is trained through a set of labeled data. Basically, the machine learning module is being told about the pattern of data; whereas, in unsupervised learning, a set of unlabeled data is provided to the training module without anything in mind, except the discovery of the structural patterns of data. The supervised and unsupervised approaches are suitable for problems such as clustering, classification, regression, and prediction, to name a few. These approaches are less common in generating a set of sample data.

Reinforcement learning is a type of machine learning through which the training agent is provided with some partial information about the domain, context, and model. The partial information enables the reinforcement learning agent to interact with the surrounding environment and learn what actions and in what conditions the maximum rewards are returned.

In the context of generation of secure configurations, the intention is to generate a new sample of data (i.e., secure configurations) with some common properties (i.e., highly secured configurations). The ultimate goal is not to cluster, or classify, or regress the security level of a given configuration. Although it is possible to randomly generate a configuration and then utilize some labeled configuration data to predict the security level of the generated configuration, it is more desirable to generate more "*secure*" configuration at the first place than measuring the security level of blindly generated configuration settings.

3.3. Reinforcement Learning Algorithm

There exists two possible approaches to formulate a problem through Reinforcement Learning (RL) [19]:

1. *Model-Based Markov Decision Process (MDP)* approach where the system is required to have complete knowledge of the environment. More specifically, the reward and transition probabilities between states should be known in order to find the best policy.
2. *Model-Free Monte Carlo (MC)* approach. Unlike MDP, Monte Carlo (MC) methods are model-free algorithms that do not require prior knowledge of the model's dynamics in order to find optimal behavior and policy. Instead, these stochastic methods rely on learning from actual experience derived from interaction with the environment. An experience can be viewed as repeated sequences of states, actions, and rewards.

In our problem formulation of MTD-based strategy model, the Model-Free Monte Carlo (MC) method is more suited for the following reason: The model's environment that we deal with is that of the SUT where an individual configuration (i.e., series of settings corresponding to different parameters) acts as a state. 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 pre-defined 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.

3.4. Monte Carlo Methods for MTD

The Monte Carlo (MC) method [19] finds approximate solutions through random sampling. It approximates the probability of an outcome by running multiple trails. MC is a stochastic technique to find an approximate answer through sampling.

Generally, Monte-Carlo approaches consist of two methods: (1) Prediction and (2) Control. In the formulation of MC for MTD problem, we will only use the prediction method.

3.4.1. MC Prediction

This method is used to measure the *goodness* of a *given policy*. It measures the goodness by learning the state-value function `V(s)`, which is essentially the *expected* return from a state S under a given policy π. Here, instead of "*expected return*" which is equal to discounted sum of all rewards, we use "*empirical return*". In a nutshell, a prediction task in MC is where a fixed pre-defined policy is supplied, and the goal is to measure how well it performs. That is, to predict the mean total rewards from any given state assuming the function is fixed. With respect to Figure 1, Algorithm 1 shows the steps involved in the Monte Carlo prediction in the context of reinforcement learning. With respect to Figure 1, the steps involved in the Monte Carlo prediction procedure in the context of reinforcement learning are as follows:

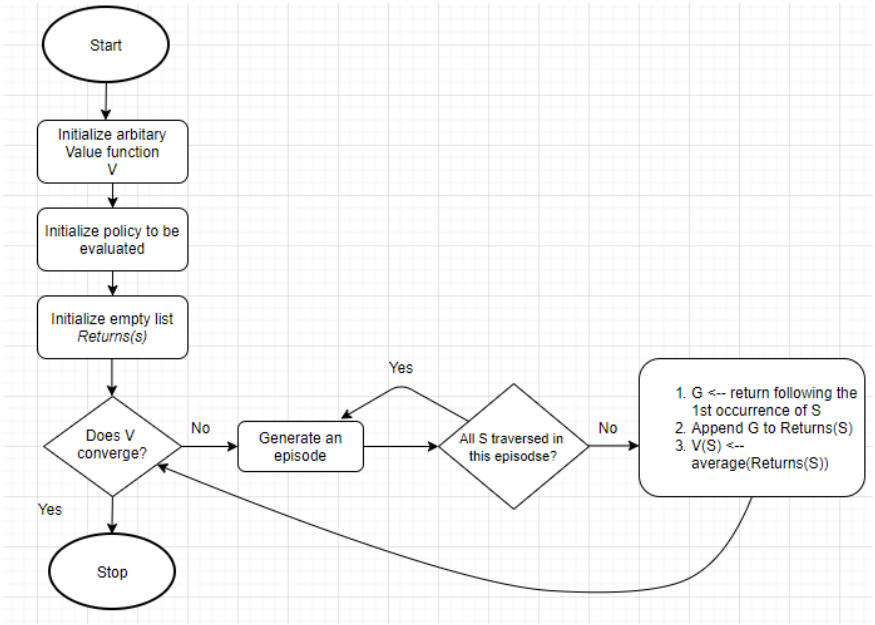


Figure 1. Steps of Monte-Carlo prediction in the context of reinforcement learning.

1. Initialize a value function V with a random value.
2. Define a policy π to be evaluated.
3. Initialize a return variable G with an empty list to store the returns.
4. Calculate return R for each state visited in each episode.
5. Append the calculated return to the return variable.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{+\infty} \gamma^k R_{t+k+1}$$

where γ is the discount factor, a constant number. It is multiplied by reward at each time step with an increasing power. The main idea behind the discount factor is to provide more priority to the immediate actions and less priority to rewards attained in later time steps.

6. Calculate the average returns and assign it to the value function.

$$V_s = E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} ... | S_t = s]$$

where E is the expected mean of the rewarded G_t for the state s.

Algorithm 1 MC Prediction.

procedure MC_PREDICTION($policy, num\_ep, df = 1.0$)
    $returns\_sum \leftarrow defaultdict(float)$ ▷ Keeps track of sum of returns for each state to calculate an average.
    $returns\_count \leftarrow defaultdict(float)$ ▷ Keeps track of count returns for each state to calculate an average.
    $V \leftarrow defaultdict(float)$ ▷ The final value function
    for i in $range(1, num\_ep + 1)$ **do**
        $episode \leftarrow generate\_episode(policy)$
        $states\_in\_episodes \leftarrow$ Find all states visited in this episode and convert them into tuple
        for $state$ in $states\_in\_episodes$ **do**
            $first\_occurrence \leftarrow$ First occurrence of the state in the episode
            $G \leftarrow$ Sum up all rewards since the first occurrence
            $returns\_sum[state] += G$
            $returns\_count[state] += 1.0$
            $V[state] = returns\_sum[state] / returns\_count[state]$
        end for
    end for
    return V
end procedure
$V = mc\_prediction(get\_action\_policy, num\_ep = 100)$ ▷ Calculating state value function V by calling the procedure $mc\_prediction()$

The procedure ends once the value function V converges or when an episode ends.

3.4.2. MC Control

A control task in reinforcement learning is where the policy is not fixed, and the goal is to find the optimal policy. That is, the objective is to find the policy $\pi(a|s)$ (with action a in a given state s) that maximizes the expected total rewards from any given state [20].

The reason behind choosing only the prediction method is because, in our game formulation of MTD-based strategy explained in Section 4, we are supplying a fixed policy and our goal is to measure the performance of the supplied policy in terms of value function. That is, to predict the expected total reward from any given state as explained before. Hence, in our methodology, we measure the reward in terms of the fitness score of each configuration of the model. The greater the fitness, the more secure is the configuration.

3.5. MC Single-Player Game for MTD

We model the MTD problem using a single player game description. The game starts with an insecure configuration state, where a configuration represents a chain of different configuration parameters and settings:

$$C := < S_1, S_2, ...S_n > \quad (1)$$

where n is the total number of configuration parameters and S_i is the setting of parameter P_i. The goal is to reach near or almost near to a secure configuration state (i.e., terminal state). An agent can either change the value of the settings or do nothing. The agent has to decide which one out of the two actions (i.e., replace or keep the setting for a given parameter) to take in every state in order to move towards the goal state. The deriving idea is to ensure that the attack surface keeps changing in the direction of a nearly secure configuration, thereby minimizing the vulnerabilities as a result. The following section describes formulation of this game into the Monte Carlo Prediction problem.

4. MC-Game for Secure Configuration

This section describes the main elements used in problem formulation of Monte Carlo prediction. We present the general scheme through a running case study in which parameters of Windows 10 operating systems are explored. First, let us provide some definitions:

1. **State**. Each state is a configuration of the underlying system. A configuration can be represented by a vector (Equation (1)), whose elements are the configuration parameters of the system and thus they hold concrete values for setting each parameter. For instance, Table 1 lists the parameters of Windows 10 along with their default values and the domain of values it belongs to. A configuration for the Windows operating system consists of multiple parameters along with their setting value such as:

```
State: < 'ACSettingIndex': 1,
         'AllowBasic': 9,
         'AllowDigest': 6,
         'AllowTelemetry': 3,
         'AllowUnencryptedTraffic': 5,
         'AlwaysInstallElevated': 4,
         'AutoConnectAllowedOEM': 4,
         ...>
```

   This vector shows a configuration for a Windows 10 system with only seven parameters along with a setting value for each.
2. **Fitness Score**. The fitness score of a configuration state is the total sum of score of the settings of parameters, denoted by `pscore`. In this paper, the `pscore` of a parameter receives a definite $HIGH$ score if it is associated to its secure setting according to STIG website [21]. Otherwise, a LOW score is assigned to the `pscore` of the parameter. For example, the fitness score of the aforementioned configuration state for Windows 10 system will be the sum of:

```
score(state) =
    pscore('ACSettingIndex') +
    pscore('AllowBasic') +
    pscore('AllowDigest') +
    pscore('AlwaysInstallElevated') +
    pscore('AutoConnectAllowedOEM').
```

3. **Actions**. In our model, two actions are defined: 1) Change ($action = 1$) the settings of the parameters, and 2) Hold/No Change ($action = 0$) which means do nothing about the setting of the underlying parameter.
4. **Policy**. We defined a policy function as follows: if the overall fitness score of the configuration is below a certain *threshold* value, then the agent is given a probability distribution of $\{p, 1-p\}$ for choosing actions *Hold/No Change* and *Change*, respectively where p is a probability. If the action *change* (i.e., $action = 1$) is chosen, the agent changes only the configuration setting of those parameters whose `pscore` is below $HIGH$ score. Otherwise, if the action *Hold* is chosen (i.e., $action = 0$), no parameter settings are changed. Formally, for a certain *threshold* value, the probability distribution $p(a)$ for action a is defined as:

$$p(a) = \begin{cases} \{prob_{NC} = lowp, prob_C = highp\} & \text{if } fitness\_score < Threshold \\ \{prob_{NC} = highp, prob_C = lowp\} & \text{if } fitness\_score >= Threshold \end{cases} \text{ where } a = NC \text{ or } C$$

5. **Rewards**. A reward value is decided on the value of fitness improvement, which is the change in the fitness score of configuration after an action takes place.

```
fitness_improvement =
    current_fitness_score - previous_fitness_score
```

Thus the reward can be computed as:

$$Reward = \begin{cases} 0 & \text{if } fitness\_improvement = 0 \\ +1 & \text{if } fitness\_improvement > 0 \\ -1 & \text{if } fitness\_improvement < 0 \end{cases}$$

Table 1. A subset of configuration parameters of Windows 10 [21].

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
DeepHooks	1	Integer
DevicePKInitEnabled	[1, 'None']	Integer, 'None'
DisableAutomaticRestartSignOn	1	Integer
DisableHTTPPrinting	1	Integer
DisableIpSourceRouting	2	Integer
DisableRunAs	1	Integer
DisableWebPnPDownload	1	Integer
DontDisplayNetworkSelectionUI	1	Integer
DriverLoadPolicy	[1,3, 8, 'None']	Integer, 'None'
EnableICMPRedirect	0	Integer
EnableInstallerDetection	1	Integer
EnableSecuritySignature	1	Integer
EnableUserControl	0	Integer
EnumerateLocalUsers	0	Integer
FormSuggest Passwords	0	Integer
fMinimizeConnections	[1, 'None']	Integer, 'None'
fBlockNonDomain	1	Integer
fMinimizeConnections	[1, 'None']	Integer, 'None'
InactivityTimeoutSecs	900	Integer
LimitBlankPasswordUse	1	Integer
LmCompatibilityLevel	5	Integer
LocalAccountTokenFilterPolicy	0	Integer

Table 1. *Cont.*

Parameter Name	Secure Default values	Domain
LoggingDisabled	[1, 'None']	Integer, 'None'
MaxSize	32768	Integer
MinEncryptionLevel	3	Integer
MinimumPINLength	[6,infinity)	Integer
NTLMMinClientSec	537395200	Integer
NoAutoplayfornonVolume	1	Integer
NoDataExecutionPrevention	[0, 'None']	Integer, 'None'
NoDriveTypeAutoRun	255	Integer
NoGPOListChanges	0	Integer
NoLMHash	1	Integer
NoLockScreenCamera	1	Integer
NoNameReleaseOnDemand	1	Integer
NoWebServices	1	Integer
RequireSecurityDevice	1	Integer
RequireSecuritySignature	1	Integer
RequireSignOrSeal	1	Integer
RestrictNullSessAccess	1	Integer
RequireStrongKey	1	Integer
SupportedEncryptionTypes	2147483640	Integer
SCENoApplyLegacyAuditPolicy	1	Integer
SCRemoveOption	[1,2]	Integer
SafeForScripting	[0,'None']	Integer, 'None'
ScriptBlockLogging	1	Integer
SealSecureChannel	1	Integer
SignSecureChannel	1	Integer
SupportedEncryptionTypes	2147483640	Integer

5. The MC Algorithm for MTD

This section presents a procedure along with the several stage-wise algorithms designed to implement an MC-based game for generating secure configurations for a given software platform.

5.1. MC Prediction Environment Setting

An environment is the surrounding context with which the agent interacts whose functions is to return the environment's next state and rewards when the agent performs an action on the current environment state. The environment implementation procedure consists of several steps, as follows:

1. Step 1: *Set Initial State*.
2. Step 2: *Compute Parameter Score*.
3. Step 3: *Set Action Policy* (Algorithm 2).
4. Step 4: *Perform the Steps* (Algorithm 3).
5. Step 5: *Generate Episodes* (Algorithm 4).

followed by *Monte Carlo Prediction* (Algorithm 1) to calculate the V. In the following sections, we discuss these steps in further details.

5.1.1. Step 1: Set Initial State

This step involves randomly assigning initial values for a configuration. The random value for each parameter is drawn from the parameter's domain. For instance, Table 1 lists the parameters and the most secure value along with the domain for each parameter. As an example, in the initial setting stage, we randomly choose an integer value for the `ACSettingIndex` (e.g., 25) whose domain is integer and most secure setting is 1.

5.1.2. Step 2: Compute Parameter Score

In this step, the procedure takes the initial randomly generated configuration in Step 1 and decides for each parameter of the configuration whether the random setting is the most secure value or not. If the initial setting for a parameter turned out to be the most secure one, then its score will be the highest value, otherwise, the lowest value will be considered as its score.

5.1.3. Step 3: Action Policy

As shown in Algorithm 2, we pass a dict *state* which is an intermediate configuration to the policy function to determine what action to take for the given state or intermediate configuration. Since we have only two actions defined in our domain: Change(C) settings (action = 1) and Hold or Not Change (NC) ($action = 0$), the function returns a tuple which consists of a action value *action_val* (0 or 1) and a list called *change_params*, which stores name of parameters if action = 1 and otherwise returns an empty list. The choice of action depends on the value of probability where $probs_{NC}$ corresponds to probability of taking action 0 and the $probs_C$ for action 1. If the value of the overall fitness score of *state* is $<$ threshold, we give high probability (i.e., p_H; e.g., $= 0.8$) to $probs_C$, or else, it is given a low probability (i.e., $1 - p_H$; e.g., $= 0.2$).

Algorithm 2 GET_ACTION_POLICY: Define Sample Policy

```
procedure GET_ACTION_POLICY(state)
    change_params ← []                                   ▷ Create an empty list
    fitness_score ← 0          ▷ To store the total fitness score of the complete state
    for key, value in state.items() do
        fitness_score+ = pscore(key, value)
        if pscore(key, value) < HIGH then
            change_params.append(key)
        end if
    end for
    if fitness_score > threshold then ▷ Check if the total score is greater than threshold
which is a hyperparameter)
        prob_NC = p_H; prob_C = 1 − p_H;
    else
        prob_NC = 1 − p_H; prob_C = p_H;
    end if
    action_val ← Randomly choose (action_NC, action_C) based on probability values
end procedure
return (action_val, change_params)
```

5.1.4. Step 4: Perform the Steps

As shown in Algorithm 3, the *Step* procedure's functionality is inspired by that of `env.step` function in OpenAI gym platform. The `env.step` function takes an action at each step. This function returns four parameters:

1. *Observation.* An environment-specific object representing the observations of the environment;
2. *Reward.* Amount of reward achieved by previous action;

3. *Done.* A Boolean value indicating the needs to reset the environment, and;
4. *Info.* Additional information for debugging.

The `STEP` procedure we developed takes two inputs: (1) state and (2) the action taken by the agent in that state. The procedure then returns three values: (1) next state the agent reached, (2) the rewards it collected, and (3) a Boolean variable "`done`" whose True value indicates that the terminal/secured state is reached (i.e., the state has reached the maximum fitness score). More specifically, this procedure takes in the current state and the action and based on the action, it changes the parameters of the given state. If $action = 1$, agent picks the action *change* where it changes the settings of the parameter mentioned in *change_params*, or else, do nothing. The reward is assigned according to the *fitness_improv* value which is the difference between the given state's total new fitness score (if action 1 is chosen) and its total old fitness score (before taking any action). For a positive, negative, and neutral fitness improvement, $+1$, -1, and 0, reward is awarded, respectively.

Algorithm 3 STEPS

procedure STEP($state, action$)
  $old\_fit\_score, new\_fit\_score \leftarrow 0, 0$
  for key, $value$ in $state.items()$ **do** ▷ Iterating over state's keys (parameter names) and values (settings)
    $old\_fit\_score+ = pscore(key, value)$
  end for
  if action ==1 **then** ▷ If action is 1, change settings, or else print hold and do nothing
    for *params* in *change_params* **do**
      $state[params] \leftarrow$ random values based on domain (params)
    end for
  else
    print('Hold')
  end if
  for key, $value$ in $state.items()$ **do**
    new_fit_score+=pscore(key,value)
  end for
  $fit\_improv = new\_fit\_score - old\_fit\_score$
  if $fit\_improv == 0$ **then**
    $reward \leftarrow 0$
  else
    if $fit\_improv > 0$ **then**
      $reward \leftarrow 1$
    else
      $reward \leftarrow -1$
    end if
  end if
  if $new\_fit\_score == len(state) * HIGH$ **then** ▷ Check if the terminal state (secured state) is reached.
    $done \leftarrow True$ ▷ Reset the environment
  else
    $done \leftarrow False$
  end if
  return $(state, reward, done)$
end procedure

5.1.5. Step 5: Generate Episode

As shown in Algorithm 4, this procedure is used to generate an episode based on policy given as a parameter. An episode is an array of $\langle state(S), action(A), rewards(R) \rangle$ tuples. It begins with an arbitrary initial state, which is assigned to *current_state*. It returns

an episode consisting maximum of 100 tuples unless it reaches the terminal sate (*done* = True) before reaching the 100 tuple limit. In that case, it ends the episode there. We set the limit to 100 tuples as a threshold to account for those cases when an episode might go through a lot of tuples before reaching the terminal state. This way we ensure the agent is time efficient and not stuck. Value of threshold is determined experimentally.

Algorithm 4 GENERATE_EPISODE

```
1: procedure GENERATE_EPISODE(policy)   ▷ Generate an array of <S,A,R> tuples for 1 episode
2:     episode ← []
3:     current_state ← get_state()   ▷ Start with an arbitrary initial state
4:     for t = 1to100 do
5:         action ← policy(current_state)
6:         next_state, reward, done ← step(current_state,action)
7:         episode.append(next_state, action[0], reward)
8:         if done == True then
9:             break
10:        end if
11:        current_state ← next_state
12:    end for
13:    return episode
14: end procedure
```

5.2. Monte Carlo Prediction

Algorithm 1 shows the Monte Carlo prediction algorithm [] for generating a set of secure configurations. The *mc_prediction* procedure takes in three parameters: (1) policy (*policy*), (2) number of episodes (*num_ep*), and (3) the discount factor (*df*). The discount factor in our case study and evaluation is set to 1.0.

The algorithm returns the state value function by taking policy as *get_action_policy* and *num_ep* = 100. In lines (2–4), we initialize three variables, `returns_sum, returns_count and V`, as a dictionary for storing the sum of rewards returned, count of occurrence, and values of each state, respectively.

In lines (5–7), for `num_ep` number of episodes, we do the following: first, in line (6), we generate an episode using the policy we provide as an input. Then in line (7), in the generated episode, we implement the first visit Monte Carlo method, finding all the states visited and store it in the variable `first_occurrence`. In the first visit Monte Carlo method, the return is averaged only after the state is visited the first time in the episode. An agent, for example, who plays snakes and ladder games, is very likely to return to the state if bitten by a snake. A state's average return is not taken into consideration when an agent revisits it. Only the first visit to the state by an agent is used to calculate an average return.

From lines (8–12), For each state in the list of visited states, we find the first occurrence of the state in the episode. We then sum up all the rewards since the first occurrence and store it in variable G such that,

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{+\infty} \gamma^k R_{t+k+1} \tag{2}$$

where γ is the discount factor. It determines the weight we assign to future and immediate rewards. Discount factor values range from 0 to 1. If the discount factor is 0 then immediate rewards are more important, whereas one would mean future rewards are more important. For calculating V_s, we compute average return for this state over all sampled episodes. more specifically,

$$V_s = E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3}...|S_t = s] \tag{3}$$

where E is the expected mean of the rewarded G_t for the state s.

6. Implementation

To the best of our knowledge, there is no public environment library available in the OpenAI Gym (https://gym.openai.com/envs/, (accessed on 1 May 2021)) pertaining to our problem at hand. Therefore, we built our own environment and customized it according to our problem.

In the *Initial State (Step 1)*, a permissible range of values is defined for every parameter in $configuration$ from which the value can be drawn randomly from the domain of the underlying parameter (i.e., $domain(p)$). Using this mechanism and to accelerate the search performed by the agent, we narrow down the search space for the agent and thus the agent can identify a secure parameter setting faster. In our case study, the default range varies according to the permissible values for each parameter takes. More specifically, depending on the *type* of parameter, the ranges of choices are limited according to the following constraints:

1. Numerical type: $(v - lim, v + lim)$
2. List Type: integer/string values. Choice between
$$\begin{cases} [(0, max(List) + lim] \\ String\ type\ value \end{cases}$$

where v is the secure setting value, lim is an arbitrary integer value, and $max(List)$ is the maximum setting value if it is a list type. We set $lim = 10$ based on multiple experiments we conducted and this value seemed to give better results in lesser time.

In terms of implementation, we can define two global variables $HIGH$ and LOW that can represent the scores obtained for each parameter. In our case, we set $HIGH$ and LOW to 800 and 8, respectively.

In Algorithm 2, to decide which action to assign more probability we set the *threshold* as a *range*: $(max\_score - value, max\_score)$ where max_score is the overall maximum fitness score of the configuration and $value$ is a hyperparameter set to 800 which denotes maximum score of one parameter. This implies that we tell the agent to not do many changes (action = NC or No Change) to the settings as that threshold range denotes that all the parameters in the configuration are set correctly except one. Otherwise, the agent chooses action = C if it less than the lower limit of threshold change, implying the fitness score of configuration is low.

We implemented all algorithms using Python version 3.6 and major libraries like numpy and pandas. Algorithms 1 and 4 are adapted from github repository [22].

7. Evaluation and Results

This section demonstrates the performance of the proposed RL-based approach in generating secure configurations and reports the results. We selected six applications and the corresponding parameters from the STIG website, namely Windows 10 (59 setting parameters), Adobe (20 setting parameters), Chrome (19 setting parameters), MS Office (21 setting parameters), Ms Excel (20 setting parameters), and McAfee (14 setting parameters). These programs contain a good number of configuration parameters whose domains are diverse enough.

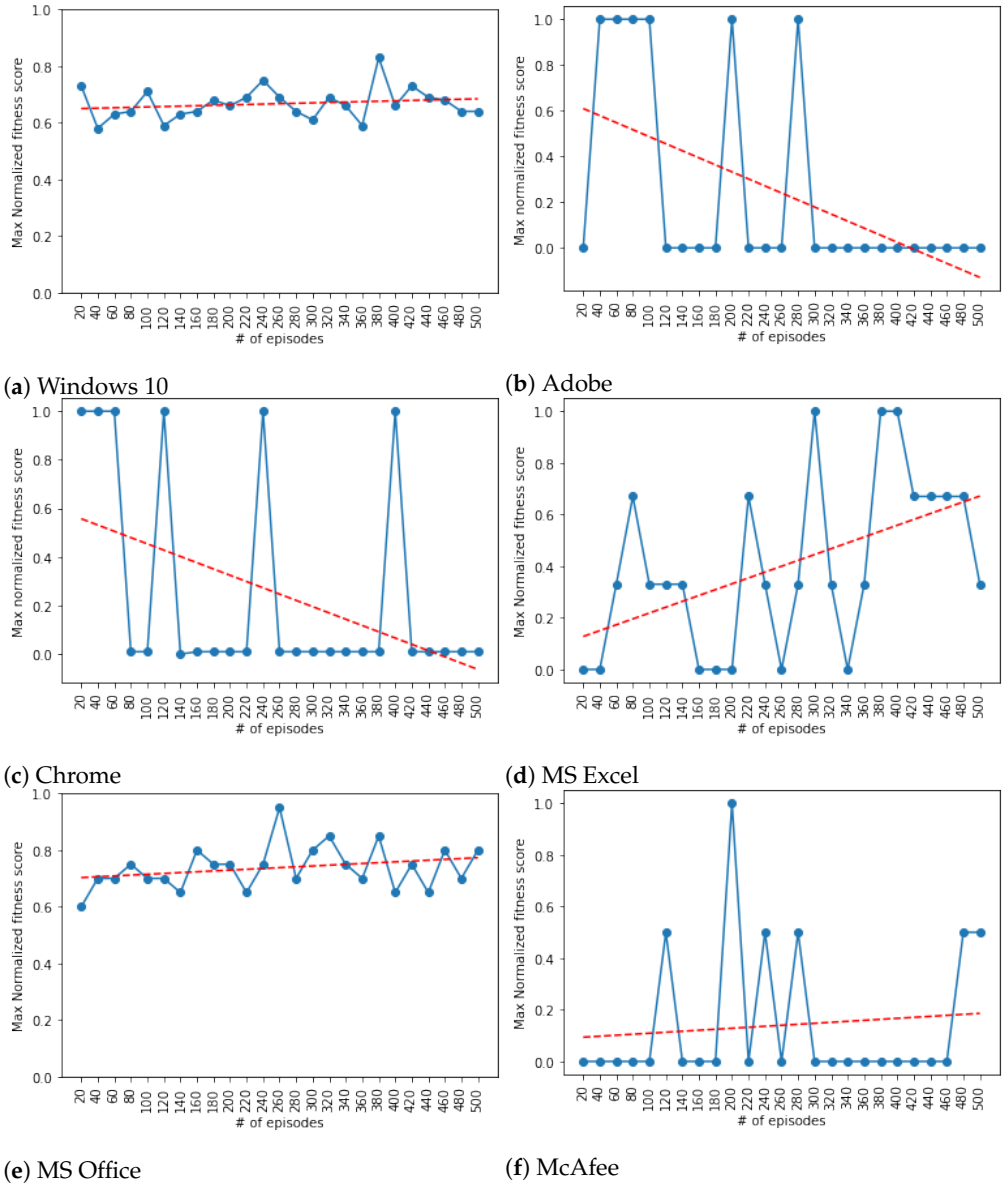
We executed our developed scripts for various number of episodes and captured the best fitness scores (i.e., value of state V that indicates how secure a given state is). Figure 2a–f illustrates the trend of fitness scores obtained through the episodes where x-axis is the episode counts (i.e., between 20 and 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 are performed as follows:

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

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,:

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

where S_i is the ith parameter and n is the total number of parameters.



(**a**) Windows 10 (**b**) Adobe

(**c**) Chrome (**d**) MS Excel

(**e**) MS Office (**f**) McAfee

Figure 2. The fitness scores obtained for each number of episodes.

Performance. To depict the trends of the fitness scores across the episodes, the plots are annotated (i.e., the red line) with fitted linear models. As Figure 2a–f indicate, we observe an increasing trends in fitness scores (most of cases) and overall good performance of generating secure configurations with very high fitness scores.

As shown in Figure 2a for Windows 10, the RL agent overall managed to attain fitness level greater than 0.6. As seen in the plot, roughly between 60% to 80% of parameters in

the windows configuration have been securely set by the agent. The highest fitness score achieved by the agent was in episode count 400 with a value slightly greater than 0.8.

Figure 2b illustrates the results obtained for Adobe. Unlike the case for Windows 10 and other applications that we studied, the Adobe software exhibited a different pattern. As depicted in Figure 2b, the trend line decreases as the episode counts increases. The fitness score reaches its highest value when the episode counts are between 40 and 100. It then shows a strange pattern where the fitness score reaches either its highest peak (i.e., =1.0) or drops to its lowest value (i.e., =0.0).

Unlike the Adobe software application, Chrome demonstrated a rational behavior as shown in Figure 2c. The trend line steadily increases as the episode counts increases, an expected behavior in training the agent. The fitness scores are between 60% and 90% where the maximum fitness score is achieved when the number of episodes is 480.

Similarly, For MSExcel and MSOffice application, as shown in Figure 2d,e, an increasing trend in improving security level is observable as episodes progress.

The only case where we observe poor performance is the output obtained for McAfee application (Figure 2f) where the security level remains in the lowest fitness score range (10–20%).

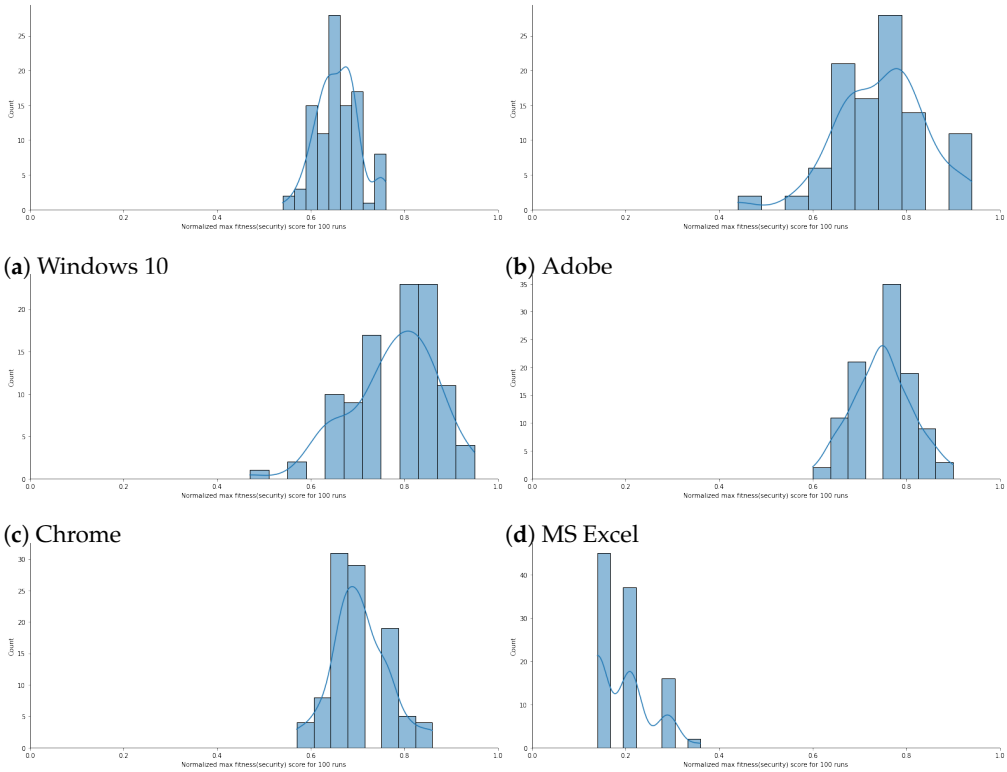
Probability Distributions. In order to analyze the probability distribution and range of fitness score values the generated configurations fall under for each case study, we re-executed the RL model 100 times with the episode count equal to 200. We then demonstrated the probability distribution of the fitness score values through histograms. Figure 3a–f visually represent the histograms for the case studies where x-axis shows the normalized fitness score values; whereas, the y-axis holds the count of configuration falling in each range,

Figure 3a shows the histogram plot for Windows 10 where 100 configurations are generated and each execution comprised of 200 episodes. We observe that most of the configuration generated after every execution falls in the range (>60% to 80%). A similar pattern is observable for Adobe, Chrome, MS Excel, and MS Office where the probability distributions are on the upper part (i.e., >0.6) of normalized fitness scores. The only subject program that exhibits a different and poor performance is McAfee where the observed probability distribution of secure configuration falls in the lower part. This might be due to less diverse set and low number of parameters (14 parameters for McAfee) for the McAfee application defined. McAfee had a few parameters composing the configuration along with as few as three possible settings to deal with: binary (i.e., $(0, 1)$) and string.

Descriptive Statistics. We also calculated the statistical metrics: maximum, minimum, mean, and standard deviation of the best fitness scores of 100 configurations collected over 100 executions of RL script for each application. As Table 2 lists, RL performed better with configurations generated for most of the cases. The best performance is observed for MSExcel with all the metrics having the best values; whereas, the worst case is demonstrated for the McAfee program. As indicated earlier, the problem with McAfee case study might be due to the lack of diversity of parameters of this application.

Table 2. Statistical analysis of maximum fitness scores of 100 configurations collected after 100 runs of RL script executed on different SUTs

SUT	Max	Min	Mean	Std Dev
Windows10	0.76	0.54	0.65	0.04
Chrome	0.95	0.58	0.77	0.085
Adobe	0.94	0.5	0.74	0.100
McAfee	0.43	0.14	0.202	0.069
MS Excel	0.95	0.6	0.74	0.066
MS Office	0.9	0.57	0.699	0.061



(**a**) Windows 10 (**b**) Adobe

(**c**) Chrome (**d**) MS Excel

(**e**) MS Office (**f**) McAfee

Figure 3. Histograms.

8. Discussion

The reinforcement learning-based schemes have already been adapted for some other similar problems such as tuning configuration of systems [11,12,14]. The problem of tuning configurations technically has its own characteristics and thus a direct comparison with the problem discussed in this paper might not be meaningful.

The closest research work to address the automatic generation of secure configurations is related to the application of evolutionary algorithms to this problem [9,10,17,18]. Such greedy algorithms are used to maximize the optimality of solutions. However, the main drawback of these evolutionary algorithms is the scalability issue in the presence of a large number of constraints in the environment [23]. As a result, obtaining reasonably good and relatively optimal solutions might not be feasible when these evolutionary algorithms are used. On the other hand, reinforcement learning-based models are more robust to such limitation and are scalable even when there are large number of constraints in the system that the agent needs to enforce and learn.

A second limitation with evolutionary algorithms is that these techniques may produce some populations that may eventually be ignored in computation. More specifically, unlike reinforcement learning where an agent learns to avoid generating such a less useful set of samples, in evolutionary algorithms there is no training notion of any agent and thus the resulting samples may be discarded.

Another distinguished difference between reinforcement learning and evolutionary algorithms is that the former utilize Markov Decision Process (MDP), whereas the latter are heuristic-based. Hence, the goodness of the reinforcement learning-based models depends on the completeness of the underlying MDP; whereas, the effectiveness of evolutionary algorithms relies on the goodness of the heuristics defined.

According to our experience with evolutionary algorithms [17,18], techniques such as Genetic Algorithms (GA) and Particle Swarm Optimizations (PSO) are very effective when the search space is small; whereas, Reinforcement Learning is more viable for large search spaces due to the fact they need large amount of training data to be effective. As a

result, the evolutionary algorithms may result in generating less diverse set of populations and samples and thus some of the samples might be repeated. On the other hand, due to a larger search space, reinforcement learning models may produce a more diverse and less redundant set of samples.

9. Conclusions and Future Work

This paper presents a technique based on reinforcement learning to generate a secure set of configurations for a given software application. The presented work is a proof-of-concept approach to moving target defense strategy in the context of configuration security. This approach leverages the AI capabilities through the application of RL in auto-tuning a vulnerable configuration setting to a secure one. We formulated the MTD strategy as a single player game using Monte Carlo Prediction where the goal of the game is to reach the terminal state (secure configuration states) through the process of auto-tuning. We then then evaluated the performance of the game on different subject applications to check how well the RL-based agent is able to attain secure configurations across different episode counts.

The presented work has some limitations regarding the number and type of parameters involved in the computation. For instance, the McAfee software application did not exhibit satisfactory results. The poor performance demonstrated by this subject software might be because of the very limited number of parameters (14 for McAfee) with very restricted choices for each parameter. Hence, the number of parameters and the diversity of settings for each parameter have a direct impact on the performance of the agent.

The fitness function and thus fitness scores were computed and optimized according to the secure parameter settings as described by Security Technical Implementation Guide (STIG) [21]. The STIG guidelines offer proper checklists in order 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 is in compliance with standards (i.e., secure system settings regulations). It is important to note that complying with some standards and thus having high fitness scores does not necessarily mean the system will be protective against various types of vulnerabilities and attacks. The research question of whether complying with security standards and checklists reduces the vulnerabilities of systems against various types of attacks need extensive experimentation and analysis.

As future work, this work can be extended and improved by integrating evolutionary algorithms like Genetic Algorithms and Particle Swarm Optimizations to find the optimum range of search space for the agent to choose from so that it can choose more rewarding actions and hence generate and thus reach the most secure configurations as a result. Moreover, this game can be turned more realistic by extending it to a 2-player game, where this time the second agent can be an attacker responsible for conducting simulated attacks on the targeted vulnerabilities while the first agent (defender) is trying to attain secure configurations.

Author Contributions: Data curation, S.D.; Implementation, S.D.; Investigation, A.S.N.; Project administration, A.S.N.; Writing—original draft, S.D.; Writing—review & editing, A.S.N. All authors have read and agreed to the published version of the manuscript.

Funding: This work is funded by National Science Foundation (NSF) under Grant No: 1821560.

Data Availability Statement: The data are available upon request.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Zhou, X. Measurements Associated with Learning More Secure Computer Configuration Parameters. Ph.D. Thesis, Wake Forest University, Winston-Salem, NC, USA, 2015.
2. National Vulnerability Database: The National Institute of Standards and Technology (NIST). Available online: https://nvd.nist.gov/ (accessed on 1 May 2021).
3. Top 10 Web Application Security Risks. Available online: https://owasp.org/www-project-top-ten/ (accessed on 1 May 2021).

4. Zheng, J.; Namin, A.S. A Survey on the Moving Target Defense Strategies: An Architectural Perspective. *J. Comput. Sci. Technol.* **2019**, *34*, 207–233.
5. Hyder, M.F.; Ismail, M.A. INMTD: Intent-based Moving Target Defense Framework using Software Defined Networks. *Eng. Technol. Appl. Sci. Res.* **2020**, *10*, 5142–5147.
6. Navas, R.E.; Cuppens, F.; Cuppens, N.B.; Toutain, L.; Papadopoulos, G.Z. MTD, Where Art Thou? A Systematic Review of Moving Target Defense Techniques for IoT. *IEEE Internet Things J.* **2021**, *8*, 7818–7832.
7. Andres A. Mercado-Velazquez, Ponciano J. Escamilla-Ambrosio, A.F.O.R. A Moving Target Defense Strategy for Internetof Things Cybersecurity. *IEEE Access* **2021**, *9*, 118406–118418.
8. Dai, H.; Murphy, C.; Kaiser, G. Configuration Fuzzing for Software Vulnerability Detection. In Proceedings of the 2010 International Conference on Availability, Reliability and Security, Kraków, Poland, 15–18 February 2010; pp. 525–530. doi:10.1109/ARES.2010.22.
9. Crouse, M.; Fulp, E.W. A moving target environment for computer configurations using Genetic Algorithms. In Proceedings of the Symposium on Configuration Analytics and Automation, Arlington, VA, USA, 31 October–1 November 2011; pp. 1–7.
10. John, D.J.; Smith, R.W.; Turkett, W.H.; Cañas, D.A.; Fulp, E.W. Evolutionary Based Moving Target Cyber Defense. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*; ACM: New York, NY, USA, 2014; pp. 1261–1268. doi:10.1145/2598394.2605437.
11. Bei, Z.; Yu, Z.; Zhang, H.; Xiong, W.; Xu, C.; Eeckhout, L.; Feng, S. RFHOC: A Random-Forest Approach to Auto-Tuning Hadoop's Configuration. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *27*, 1470–1483. doi:10.1109/TPDS.2015.2449299.
12. Zhu, Y.; Liu, J.; Guo, M.; Ma, W.; Bao, Y. ACTS in Need: Automatic Configuration Tuning with Scalability Guarantees. In Proceedings of the 8th Asia-Pacific Workshop on Systems, Mumbai, India, 2 September 2017.
13. Zhu, Y.; Liu, J.; Guo, M.; Bao, Y.; Ma, W.; Liu, Z.; Song, K.; Yang, Y. BestConfig. In Proceedings of the Symposium on Cloud Computing, Santa Clara, CA, USA, 25–27 September 2017, doi:10.1145/3127479.3128605.
14. Zhang, J.; Liu, Y.; Zhou, K.; Li, G.; Xiao, Z.; Cheng, B.; Xing, J.; Wang, Y.; Cheng, T.; Liu, L.; Ran, M.; Li, Z. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In Proceedings of the 2019 International Conference on Management of Data, Amsterdam, Netherlands, 30 June–5 July 2019.
15. Zhang, J.; Zhou, K.; Li, G.; Liu, Y.; Xie, M.; Cheng, B.; Xing, J. CDBTune: An efficient deep reinforcement learning-based automatic cloud database tuning system. *VLDB J.* **2021**, 1–29, doi:10.1007/s00778-021-00670-9.
16. Bao, L.; Liu, X.; Wang, F.; Fang, B. ACTGAN: Automatic Configuration Tuning for Software Systems with Generative Adversarial Networks. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 465–476. doi:10.1109/ASE.2019.00051.
17. Dass, S.; Namin, A.S. Evolutionary algorithms for vulnerability coverage. In Proceedings of the S2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC), Madrid, Spain, 13–17 July 2020; pp. 1795–1801.
18. Dass, S.; Namin, A.S. A Sensitivity Analysis of Evolutionary Algorithms in Generating Secure Configurations. In Proceedings of the 2020 IEEE International Conference on Big Data (Big Data), Atlanta, GA, USA, 10–13 December 2020; pp. 2065–2072.
19. Sutton, R.S.; Barto, A.G. *Reinforcement Learning, Second Edition: An Introduction (Adaptive Computation and Machine Learning Series)*; MIT Press: Cambridge, MA, USA, 2018.
20. Sutton, R.S.; Barto, A.G. *Introduction to Reinforcement Learning*, 1st ed.; MIT Press: Cambridge, MA, USA, 1998.
21. STIG Viewer. Available online: https://www.stigviewer.com/stigs (acccessed on 1 May 2021).
22. Weissman, J. Reinforcement Learning. 2019. Available online: https://github.com/dennybritz/reinforcement-learning/tree/master/MC (acccessed on 1 May 2021).
23. Cho, J.H.; Sharma, D.P.; Alavizadeh, H.; Yoon, S.; Ben-Asher, N.; Moore, T.J.; Kim, D.S.; Lim, H.; Nelson, F.F. Toward Proactive, Adaptive Defense: A Survey on Moving Target Defense. *Commun. Surv. Tutor.* **2020**, *22*, 709–745.