# A REINFORCEMENT LEARNING AGENT TO DYNAMICALLY MODIFY THE BEHAVIOUR OF A SANDBOX BASED ON SPECIFIC MALWARE SAMPLES

Vishwadinu Kisaja Muthukumara Vithanage

(IT21300950)

Dissertation submitted in partial fulfillment of the requirements for the Bachelor of Science (Hons) in Information Technology Specialized in Cyber Security

Department of Computer Systems Engineering

Sri Lanka Institute of Information Technology
Sri Lanka

April 2025

## DECLERATION

“I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for a degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to Sri Lanka Institute of Information Technology, the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).”

Signature: Date: 11/05/2024

# ABSTRACT

This work investigates the integration of reinforcement learning with malware behavioral analysis through instantiating an environment to interface with Cuckoo Sandbox. The system accommodates applying counter-evasion techniques on the environment level by applying predefined PowerShell scripts onto the VM prior to analyzing malware behavior. The scripts mimic typical techniques used to trigger malware behavior manipulation, including disabling debug services, hiding virtualization signatures, updating autorun entries, and configuring networking options. The reinforcement learning agent is trained under the Proximal Policy Optimization (PPO) algorithm and acts on this environment by selecting and applying one among the predefined actions to one malware specimen. The system runs the manipulated environment with the sandbox and returns the behavior scoring against scoring one without manipulation. The scores' difference is employed to act as the signal to the agent to identify actions that are efficient in evoking perceptible behavior. Experiments on applying the proposed framework on an array of malware specimens against an array of predefined actions demonstrated repeated patterns on some scripts, i.e., hiding virtualization presence. Action-reward mappings and train metrics indicated that the agent could identify successful strategies from variations on behavior that are perceptible. The environment recorded all actions, reward, and pairing with samples to be traceable and support-able. The system accommodates modularity in extension by accommodating the addition of more malware family types or evasion techniques and is suitable to be applied as the foundation to automate evasive behavior discovery through adaptive sandboxes configurations.

**Keywords:** Reinforcement Learning, Malware Analysis, Cuckoo Sandbox, Evasion Techniques, Sandbox Automation

# ACKNOWLEDGEMENT

I would like to extend my heartfelt gratitude to **Mr. Amila Senarathne**, Senior Lecturer and Head of the Industry Engagement Unit at the Sri Lanka Institute of Information Technology, for his invaluable guidance, constant encouragement, and thoughtful supervision throughout the course of this research. His expertise and mentorship played a vital role in shaping the direction and quality of this study.I am equally grateful to my co-supervisor, **Mr. Deemantha Siriwardana**, Assistant Lecturer, for his continuous support, constructive feedback, and timely insights that helped me overcome various challenges during this project.A special word of thanks goes to our external supervisor, **Ms. Chethana Liyanapathirana**, Assistant Professor at the Department of Mathematics, Computer Science, and Digital Forensics, Commonwealth University of Pennsylvania, for her valuable contributions, external perspectives, and expert advice which greatly enriched the research process.I also wish to sincerely acknowledge the **Department of Computer Systems Engineering at Sri Lanka Institute of Information Technology (SLIIT)** for providing the academic environment and necessary resources to carry out this research successfully.Moreover, I would like to express my appreciation to my project group members for their collaboration, commitment, and shared efforts throughout the research journey. Their support and teamwork were essential to the successful completion of this project.Finally, I thank everyone who supported this endeavor, both directly and indirectly, and contributed to the successful completion of this thesis.

# TABLE OF CONTENT

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF ABBREVIATIONS

| Abbreviation | Description |
|---|---|
| IT | Information Technology |
| ML | Machine Learning |
| RL | Reinforcement Learning |
| VM | Virtual Machine |
| DRL | Deep Reinforcement Learning |
| GAN | Generative Adversarial Network |
| PPO | Proximal Policy Optimization |
| DQN | Deep Q-Network |

# 1. INTRODUCTION

In the modern digital world, Information Technology is the driving force of day-to-day work of people. Information Technology has been able to make lives of people easier. But at the same time, threats that can impact in many different methods started to emerge in the field of IT. As a result, and solution for this cyber security started to become an essential for all the systems related to IT. Cyber security plays a crucial role for the tasks that are being conducted with the help of information technology.

In the past few years cyber security sector started to be developed with both offensive and defensive capabilities. Malware, one of the most popular categories in the field of cyber security has become more sophisticated as the offensive strategies became developed. Malware can be described as any software which are designed to violate any of the three pillars of cyber security that are confidentiality, integrity, and availability in a particular information system. As malware has been sophisticated during recent years, they have become stealthier which implies that malware can hide malicious behaviour according to the situation. As a counter action for this malware detection methods and system must be improved to find the true malicious behaviour of malware.

### 1.1. Background Literature

Traditional methods for malware detection fall into two main categories which are static and dynamic analysis. Static analysis involves analyzing the source code or binary code of application software without running it. This technique excels at detecting known malware patterns and signatures but will be vulnerable to evasion methods including obfuscation, encryption, or polymorphism. On the other hand, dynamic analysis involves running the suspicious malware in a controlled environment which is often referred as sandbox and tracking what it does. This will make it possible to identify previously unknown malware by observing what the malware does when run.

Sandboxes provide an isolated environment to safely analyze malware [1]. Among the sandboxes that are widely used, Cuckoo can be mentioned as a well-known sandbox for malware analysis. It is an open-source automated analysis framework that can analyze different types of malicious files dynamically within seconds. Cuckoo creates detailed behavioural reports by running malware in a VM and monitoring changes in file systems, network traffic, registry modifications, and other metrics. One of the valuable advantages of using Cuckoo is its customizable architecture which allows users to modify the sandbox according to their wish [2].

Despite the effectiveness of sandboxes, malware developers have found ways to detect and bypass these sandbox environments. Various evasion techniques have been employed with malware to make the detection difficult for the sandboxes. It thus leads to the inference that sandboxes are likely to create incomplete or even misleading behavioural information, which ultimately compromises detection accuracy [1].

Researchers have investigated using ML and RL in malware analysis to combat these evasion techniques. Although the main goals of supervised and unsupervised learning techniques are pattern recognition and classification RL provides a distinct benefit by allowing agents to discover the best course of action through interaction with their surroundings.

RL can be identified as a subdomain within the larger category of machine learning drawing from concepts in behavioural psychology. This refers to the process by which a self-contained agent learns to differentiate between optimal decision-making approaches through direct interaction with its environment, adopting a trial-and-error process in a bid to achieve set objectives. The agent repeatedly executes random behaviours under diverse environmental conditions and gains evaluative feedback in the form of rewards or penalties, based on the outcomes of its behaviours [5]. The reinforcement signals are used by the agent in a bid to induce behaviours relating to desirable outcomes while suppressing behaviours resulting in negative outcomes. The agent incrementally forges its decision-making strategy, moving through a system in a bid to maximize the cumulative number of rewards received during its interaction. The iterative learning process, combining the exploration of new strategies with the leveraging of learned experience, allows the agent to adapt to changing and stochastic environments, gain insights from experience, and create more productive behaviours in line with a reduction in the burden of comprehensive, detailed programming for each possible situation [5]. RL has been used in cybersecurity for tasks like autonomous vehicle systems security, spoofing identification in wireless systems, detection of phishing etc [4].

The connection between RL and malware analysis is a new and growing field with a huge potential. By simulating the behaviour of malware and observing how malware interacts with detection systems, RL can help to uncover sophisticated evasion techniques. Moreover, RL-trained agents can identify sandbox vulnerabilities and develop techniques that mimic evasion behaviours in the real world.

Scholarly communities from academia and industry conduct research to introduce several counter-malware strategies. Notable among them has been the use of artificial intelligence, which has spawned smart detection mechanisms that go beyond traditional antivirus programs. These systems take advantage of huge datasets to learn models that can identify malicious behavioural patterns. According to Brown et al. [6], it was possible to detect malware using machine learning in static and cloud IaaS environments. Several more studies [7], [8], also proved that the use of machine

learning will be helpful in detection of malware. But these approaches are contradicted by the reactive nature of such models and their dependence on historical data for their effectiveness against new, zero-day attacks.

This limitation imposes the value of proactive analysis techniques such as sandboxing, supplemented by adaptive approaches such as reinforcement learning. Supervised models require labelled datasets, whereas reinforcement learning can learn with minimal a priori knowledge and enhance through exploration and interaction. This aspect makes RL a good candidate for constructing robust and intelligent malware analysis frameworks.

In addition, RL's iteratively learning capability allows it to be optimized continuously, thereby making it well-adapted for real-time application in threat detection and malware analysis. In combination with sandboxing, RL can drive intelligent decision-making within dynamic analysis by optimizing what to monitor for features, what to alert on for behaviour, and how to order threats. This convergence can lead to creating robust, independent malware analysis frameworks that can continue to evolve according to the threat landscape, eventually enhancing cybersecurity defences in a more scalable and intelligent manner.

Also, the malware creators' competition with security professionals has spawned adversarial machine learning. In this context, criminals create inputs to deceive machine learning algorithms. Similarly, malware can be seen as an adversarial actor that tries to manipulate the analysis ecosystem. Applying RL to emulate such activity allows defenders to anticipate and make their detection systems more resilient against such techniques.

## 1.2. Research Gap

Nonetheless there are reactive approaches for malware detection, proactive approaches are very hard to be found in malware analysis and detection. Quertier et al. [3] performed studies on evading multiple malware detection engines via RL. The investigation indicates that several commercial antivirus software was also circumvented. The researchers adopted an offensive approach instead of a defensive one, resulting in the absence of anti-evasion tactics and malware analysis in the research report [3].

Bokolo et al. [8] has suggested a technology using machine learning to identify malware. A training dataset and a testing dataset were used to train a model, which will successively produce a classifier according to that study. The malware test will initialize, and the final categorization result will be acquired when a user inputs a file. Malware samples have been classified and detected using deep learning models as K-nearest neighbor, Support Vector Machine, and SGD [8]. There is no indication that RL was used. The article does not discuss how to identify malware by altering a sandbox environment.

A method that employs machine learning and a cloud-based sandbox to identify malware samples has been proposed by Mail et al. [9]. According to the article, the suggested method will use key attributes found in the cloud sandbox to run a machine learning classifier that will identify and categorize the different types of malware attacks. This technique uses machine learning to detect malware, and it is unable to find any evidence of reinforcement learning or real-time sandbox modification.

Gibert et al. [10] suggested a framework for obfuscating malware using Reinforcement Learning by wrongly altering the label. This has been accomplished by using an intelligent agent that uses reinforcement learning to insert NOP command. Evasion techniques and the application of reinforcement learning can be found in this study. However, it is also possible to identify the absence of a technology for real-time sandbox change.

Jaswinder et al. [11] was able to propose a method for identifying malware based on their behaviours. Malware samples have been found using behaviour artifacts like PSI, API calls, and network activity. Additionally, machine learning algorithms have been used to train malware classifiers. Furthermore, real-time sandbox modification or reinforcement learning is not covered in this study. Only the artifacts from the malware have been extracted using the Cuckoo Sandbox.

The suggested framework of Li et al. [12] can be considered as the first adversarial network that operates on principles. This research's technique is based on Deep Neural Networks, which is a subclass of RL. However, the disparity in altering a sandbox environment persists even in this study.

The creation of malware samples that can elude machine learning based IoT malware detectors can be found in the research conducted by Arif et al. [13]. In this proposed framework, malware samples were produced with adversarial features by using DRL. GAN model was used to generated adversarial features. Although DRL was employed in this study, it is not possible to view an agent that is integrated with the Cuckoo sandbox.

Chtazoglou et al. [14] investigated the development of new evasion tactics for commercially accessible antivirus systems. Overall, the researchers were able to produce seven evasion tactics and integrate them with 16 malware samples. According to the researchers, the prior malware samples were undetectable. While evasion tactics were developed in this study, the area on altering a sandbox environment and using RL are not covered.

The following table will demonstrate the differences between the current research and conducted research studies in this area. It is highly clear that sandbox environments modification procedures cannot be found in the conducted research studies. This explains that this research is highly important for the malware analysis field.

*Table 1.1: Research Gap*

| **Research Paper** | **Use of Reinforcement Learning (RL)** | **Sandbox Environment Modification** | **Anti-Evasion Strategies** | **Focus on Evasion Techniques** | **Real-Time Malware Detection** |
|---|---|---|---|---|---|
| *Quertier et al. [3]* | ✓ | ✗ | ✗ | ✓ | ✗ |
| *Bokolo et al. [8]* | ✗ | ✗ | ✗ | ✗ | ✓ |
| *Mail et al. [9]* | ✗ | ✗ | ✗ | ✗ | ✓ |
| *Gibert et al. [10]* | ✓ | ✗ | ✓ | ✓ | ✗ |
| *Jaswinder et al. [11]* | ✗ | ✗ | ✓ | ✗ | ✓ |
| *Li et al. [12]* | ✓ | ✗ | ✗ | ✓ | ✓ |
| *Arif et al. [13]* | ✓ | ✗ | ✗ | ✓ | ✗ |
| *Chtazoglou et al. [14]* | ✗ | ✗ | ✓ | ✓ | ✗ |
| ***Proposed Research*** | **✓** | **✓** | **✓** | **✓** | **✓** |

According to the current research conducted, it can be proven that the use of RL for dynamically modifying the sandbox environments is in a good value when comparing with other research studies. This research has paved the way for dynamic customization of sandboxes related to this area.

### 1.3. Research Problem

Malicious software often tend to include various evasion techniques and as a result of this matter, malware can be identified as very sophisticated. These evasion strategies can detect virtualized environments. Due to this issue most of the defensive approaches have been ineffective towards the evasion strategies used by malware samples. As mentioned in the previous section, most of the approaches to address this issue have been developed with the use of machine learning. Even though they address the issue for a certain level, the capability of modifying the sandbox in real-time is absent.

The main problem that has been addressed by this research is the absence of dynamic modifications of sandbox environments to find the exact malicious behaviour of specific sandbox evasive malware samples via an intelligent agent. Precisely, how an agent that has been implemented with RL can pick the correct anti evasion strategy for the given specific malware samples?

This research explores the possibility of developing an RL agent to engage with a sandbox environment that can apply diverse system-level changes and monitor the effect on malware detection. Thus, it tries to detect actions that minimize detection scores and reveal possible blind spots in sandbox analysis.

The research problem consists of machine learning, cyber security, and malware analysis. The solution for the problem must be developed by involving software engineering, artificial intelligence, and threat intelligence skills. Addressing this issue involves the creation of an experimental framework, the selection of appropriate RL algorithms, the design of action scripts, integration with the Cuckoo API, and systematic examination of the results.

Also, the research explores the adversarial behaviour of RL agents. By allowing an RL agent to experiment with different conditions with the environments, the research measures the efficiency of how malware can be manipulated to evade detection. This

adversarial simulation gives insightful information regarding the resilience of existing security infrastructures and areas for improvement.

## 1.4. Research Objectives

### 1.4.1 Main Objective

- **To develop an RL agent to dynamically modify the sandbox environment based on evasion strategies of specific malware**

The main objective of this research is to develop an RL agent which is capable of dynamically modifying the sandbox environments in response to the evasion strategies used by malware samples.

The agent is built to automatically discover and apply changes that render the detection less likely, providing insight into potential vulnerabilities in sandbox analysis techniques and facilitating the creation of more concrete defenses.

### 1.4.2 Sub Objectives

- **Clearly define the problem statement of malware sandbox evasion and list down the requirements for a solution using reinforcement learning.**

First and foremost, objectives is to clearly define the problem statement, which includes identify and detail the problem of sandbox evasion behaviour of specific malware samples. The focus of this sub-objective is to formalize the evasion problem and identify the core challenges faced by the malware analyzing sandboxes.

- **Designing and organizing a simulated environment inside which the RL agent is going to run, including specifying possible actions, states, and incentive systems that will guide the development process.**

A well-constructed environment must be developed with OpenAI Gym framework. This is done to incorporate the correct functions which are required for the RL agent to operate correctly. The environment must include state spaces that can capture

relevant system features such as registry values, running services, debugging settings etc. Also, the action space must be created with counter-evasion strategies such as changing specific values in registry, simulate user behaviour etc. The reward function also be designed with the ability to assign rewards and penalties according to the score generated by Cuckoo sandbox.

- **Design an RL agent from this structured environment and train it on the identification and mitigation of malware evasion strategies.**

The target of this sub-objective is to implement the RL agent by selecting an appropriate RL training algorithm. Few different algorithms are available such as PPO, DQN etc. The agent must be trained with different actions and different malware samples to gain the sufficient experience. Training will involve specific number of episodes and according to the number of episodes, the amount of experience that the agent will gain will be different.

- **Run the trained RL agent for the evaluation of performance in malware detection, clearance of evasion techniques, and efficiency in different scenarios.**

The performance of RL agent must be evaluated upon the training. This will be based on how the trained agent is able to increase the score of Cuckoo for a malware sample. This indicates that hidden malicious behaviour of that malware sample is discovered with the modification of the sandbox. Performance will be evaluated according to the average reward collected by the agent.

- **Iteratively optimize the performance of the RL agent so that it will finally fit within the specifications of deployment.**

The final sub-objective is to fine-tune the agent. This will be done with changing the action scripts, number of episodes and number of actions scripts.

# 2. METHODOLOGY

In this chapter, the methodology of the RL agent will be discussed.

## 2.1 System Overview

The basic idea of this research is to discover hidden malicious behaviours of a malware sample. To achieve this there are several components and algorithms have been used. The following figure will demonstrate the overall process of the system diagram.

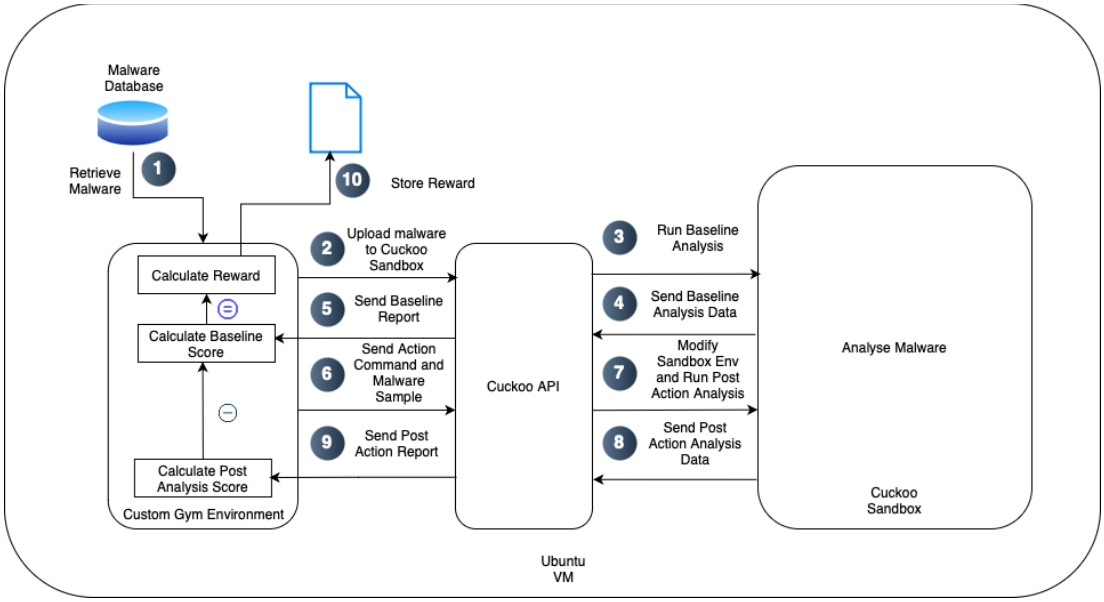


*Figure 2.1: System Diagram*

### 2.1.1 Baseline analysis

The baseline analysis is done as reference point for the agent. Before any modifications to the sandbox are performed, all malware samples are run in their default sandbox environment first. This approach documents how the sandbox naturally detects and classifies malware behaviour in an unaltered state. This will ensure a consistent reference point for measuring the effects of introducing modifications afterward. This is essential in ascertaining whether the effectiveness of the sandbox is improved or

degraded after the reinforcement learning agent introduces counter-evasion techniques.

The resulting procedure allows for the standardization of a broad range of malware specimen samples. Since individual malware families exhibit inherently different detection signatures, having a baseline allows for a target, sample-specific reference as opposed to relying on static detection values. This process separates environmental variables' impact from those of the malware, thus maintaining the scientific integrity of the analysis. If these normalized systems were not in place, it would be impossible to determine whether detection improvements were due to direct changes in the sandbox or were merely random variations in malware activity.

### 2.1.2 Extract baseline score from the report

The Cuckoo Sandbox generates a detailed behavioural report after the execution of every malware sample. The report includes a section known as *info.score*, which summarizes the overall severity and detection confidence of the behaviour witnessed. The numeric value of this score is obtained by using Python's requests library to interface with the API, in combination with the json module to parse the formatted data. The automation in this extraction process ensures consistency and reproducibility, essentially removing the possibility of human error in the scoring system. The calculated score is immediately recorded and set as the baseline score for the malware sample in question.

The accurate retrieval and storage of this score form the basis on which rewards are determined in each episode of reinforcement learning. Inconsistencies as well as latency in the availability of reports are overcome through ongoing polling of the sandbox's REST API, thus ensuring the system is ready while waiting for report generation. In addition, the extracted score is written out in CSV format for future use, aiding auditing, visualization, as well as analysis of temporal trends. This systematic retrieval of the raw JSON output converts it into useful information that can be utilized by the reinforcement learning agent in making future decisions.

### 2.1.3 Apply counter-evasion strategies

Having determined the baseline score, the RL agent begins deploying counter-evasion measures to harden the sandbox environment against sophisticated malware threats. This implementation consists of a set of Python scripts that remotely run PowerShell commands on the target Windows virtual machine. The actions represent a broad range of defensive measures, such as enabling security services that malware might attempt to disable, modifying system-level registry keys, exposing virtualization artifacts, or mandating improved process visibility. These scripts automate the manual tuning process that a human analyst would normally perform in a sandbox, with the added advantages of automation and reinforcement learning feedback.

Each action is taken by the reinforcement learning agent and is carried out sequentially. After applying changes within the sandbox, the same malware sample is re-run, leading to the creation of a new detection rating. This approach allows for automated testing of the effectiveness of each individual action. The permutations and combinations of these changes increasingly help in determining those methods that maximally boost the detection rating. This process mimics the work of real-world malware analysts who would be refining a sandbox environment in response to increasingly advanced malware tactics; however, this is done systematically and adaptively via the use of reinforcement learning.

### 2.1.4 Reward allocation

Reward allocation is an important element in the framework of reinforcement learning since it provides evaluation feedback on the effectiveness of the action of an agent. In this research framework, reward is ascertained through the evaluation of the difference between the revised detection score (after making an amendment) and the initial baseline score. In cases where the amendment causes the detection score to increase, the agent is rewarded with a positive reward, thus encouraging the use of the amendment in the environment. In cases where the detection score does not change or

drops, the agent is penalized through either zero or negative reward, encouraging it to avoid ineffective or detrimental configurations in future situations.

The reward system design is inherently linked to the main goal of this research: enhancing the visibility of the sandbox environment. By framing the reward in terms of detection improvement, the agent is encouraged to select actions that help in the advancement of the sandbox's analytical coverage. Every reward is carefully logged in a file that is associated with the malware sample and the action taken, allowing researchers to see trends and determine which methods provide consistent effectiveness. This feedback system, realized through rewards, is at the heart of the learning system, distilling raw detection data into useful reinforcement that increasingly influences the agent's policy.

### 2.1.5 Training the agent

The agent is trained with the help of the Proximal Policy Optimization (PPO) algorithm, known for balancing training stability with efficiency of performance. PPO works towards making step-by-step improvements in the agent's policy without risking very large updates that can destabilize the learning progress. During the period of training, the agent interacts with sets of malware samples. It performs a sequence of sandbox modification operations on each sample and is awarded rewards based on detection score changes. PPO updates the agent’s policy systematically based on the rewards, thus ensuring improvements in its decision-making in a logical, systematic way.

Unlike value-based algorithms like DQN, PPO maintains a stochastic policy that encourages both exploration and exploitation. This allows the agent to discover novel sequences of actions that it may not have encountered before while still leveraging actions that are known to be effective. The training is conducted over many episodes, with each episode involving a new malware sample and a fresh analysis cycle. As the agent is exposed to a broader range of malware behaviour and learns from thousands of interactions, it generalizes better and becomes capable of adapting the sandbox

configuration dynamically to strengthen detection. All training interactions, including state transitions, rewards, and selected actions, are recorded for later analysis and *validation.sandbox* environments to be more accurate at detecting evasive malware attacks.

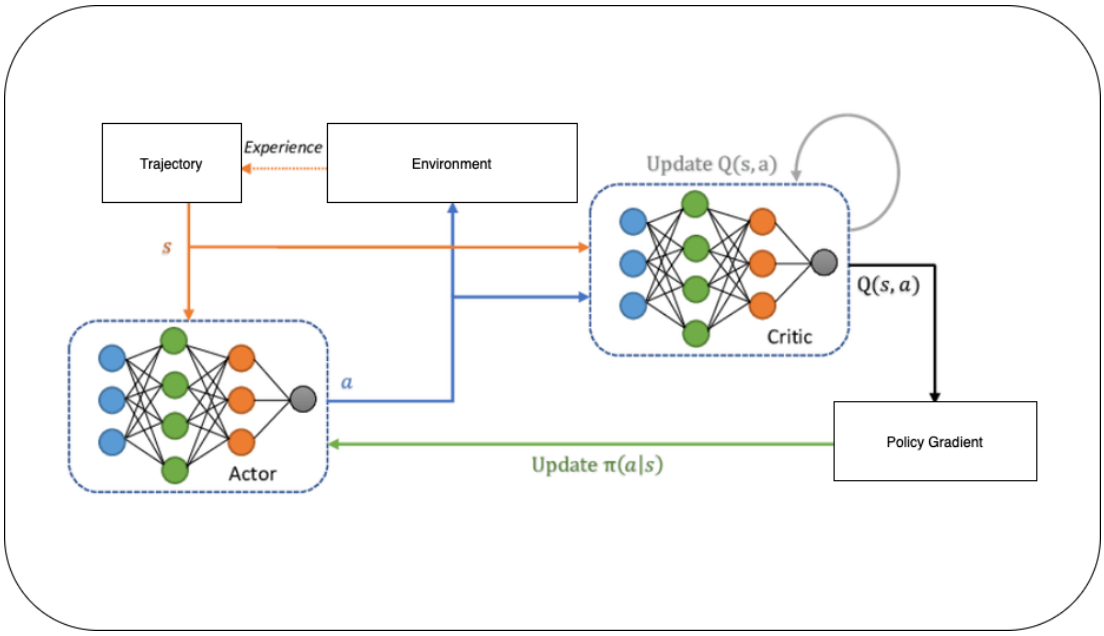


*Figure 2.2: PPO Architecture [15]*

## 2.2 Commercialization Aspects of the Product

Though initially created for research purposes, the reinforcement learning-based sandbox hardening system holds immense commercial prospects in the field of cybersecurity and academic research. Its unique selling point is its automation and optimization of sandbox settings using reinforcement learning, which brings immense benefit to organizations and institutions involved in malware research, threat monitoring, and research in cyber defenses. As today's malware evolves continuously using sophisticated evasion tactics, traditional static and dynamic analysis software is increasingly struggling to remain effective. A system allowing for real-time adaptation to counter such tactics would bring immense competitive advantage in proactive threat identification and sandbox hardening.

One of the most promising areas for commercialization is integrating this structure within commercial sandboxing products. Currently, there are various organizations and antivirus companies relying upon static sandbox configurations with limited flexibility. Integrating the reinforcement learning agent within products would make such companies capable of dynamically modifying the analysis environment in accordance with evasion mechanisms of the detected samples. This would effectively increase their detection rates, primarily for sophisticated malware variants specially crafted to detect and evade virtualization environments.

The suggested methodology can serve as an internal asset in Security Operation Centers (SOCs) and threat-intelligence platforms for improving sandbox security and conducting red teaming drills. From within, analysts can use the system to mimic real-life evasive attacks, thus unmasking weaknesses in current sandbox configurations. In its plug-in module or external service deployment mode, the system can run as an always-on assessor for current sandbox configurations susceptible to anti-analysis tactics. This proactive approach would give an advantage to organizations against adversaries in terms of discovering and fixing weaknesses in sandboxes before they can be used in real attacks.

In addition, the suggested framework offers opportunities for commercialization based on a shift in mindset, known as Research-as-a-Service (RaaS), to sandbox resilience testing. Research centers and advisory companies have the possibility of offering sandbox resilience test services to customers who want to analyze the effectiveness of malware identification systems. The software can generate in-depth logs and reports detailing the operation of a sandbox in countering evasive threats, along with recommendations for improving configurations in light of experiential knowledge. Such insights can inform infrastructure and staff training program changes.

Another fascinating market is in research and academic institutions that can use this product as an experimentation and teaching tool. The framework is a real-world example of reinforcement learning for cybersecurity, which would be ideal to use in computer science, cyber defense, and machine learning courses. It also offers a basis for further research to be conducted—such as investigating multi-agent systems, zero-day malware tolerance, or adversarial machine learning in dynamic analysis.

Finally, long-term potential also remains for incorporation into cloud-based malware analysis platforms, where the sandbox environment is virtualized and scaled across multiple customers. Within these platforms, the RL agent could be leveraged to intelligently assign hardened sandbox profiles to samples based on metadata, malware family, or threat level. This would optimize resource allocation and analysis accuracy in a high-throughput system. Licensing this module as a plug-in to VirusTotal, Any.Run, or hybrid cloud sandboxes can unlock recurring revenues while contributing to global threat intelligence.

## 2.3 Testing and Implementation

In this section, the code level implementation and also the testing part will be discussed. Implementation was done in a modular fashion, with each module being developed and tested separately before integrating into the full system. Integration of the reinforcement learning agent into the Cuckoo Sandbox was under the requirement of the development of strong communication protocols and synchronization mechanisms. Implementation also involved managing system dependencies, runtime performance optimization, and the development of a reward function capable of guiding the learning process of the agent within the sandboxed environment efficiently.

Testing was done through unit and integration testing. Unit testing verified the correctness of individual modules, and integration testing verified that components function reliably together as part of the overall system. Controlled samples of malware were executed to test the functionality of the system to monitor and respond to dynamic behaviour. Measure metrics such as action trace accuracy, sample execution consistency, and agent performance were recorded and compared to guide iterative improvements in both implementation and reinforcement learning framework.

### 2.3.1 Custom OpenAI Gym environment

The cuckoo_malware_env.py is the Python script that defines a custom OpenAI Gym environment for malware behavioural analysis using reinforcement learning and the Cuckoo Sandbox. The environment enables an agent to interact with malware samples through a defined set of actions and receive feedback in terms of rewards based on the observed behaviour when the sandbox is executed. It supports discrete actions and normalized observation spaces, provides automatic logging of actions and rewards, and is designed to facilitate experimentation and agent training for dynamic malware analysis tasks.

#### 2.3.1.1 Required Python Libraries for custom environment

The Python code relies on several libraries to interact with the reinforcement learning environment and other external systems. gym and spaces module of OpenAI Gym are utilized to define the environment, action space, and observation space of the reinforcement learning agent. numpy is utilized for numeric computations, i.e., to handle observation data. ti me and subprocess modules enable interaction with the operating system in order to run and monitor malware samples in the Cuckoo Sandbox. requests and json enable HTTP communication and data processing in order to interact with the Cuckoo API and parse JSON responses. os and csv are also used for interaction with the file system and logging agent behaviour and rewards so that experiments are traceable and reproducible.

```
import gym
from gym import spaces
import numpy as np
import subprocess
import time
import requests
import json
import os
import csv
```

*Figure 2.3: Python libraries for custom OpenAI Gym environment*

#### 2.3.1.2 CuckooMalwareEnv class

An OpenAI Gym environment created for reinforcement learning studies in the context of malware investigation is the CuckooMalwareEnv class. An RL agent can view and engage with malware samples by performing pre-programmed behaviours that mimic system-level behaviour changes thanks to this environment's interface with the Cuckoo Sandbox. In this environment, the agent receives observations and rewards based on

the results of its interactions, allowing automated investigation of malware behaviour through guided action choices.

The class configures key setup options, including the directory containing the malware sample and an API key for using the Cuckoo Sandbox. If one does not already exist, it also ensures that a log file (reward_log.csv) is created, where each interaction is recorded for future review. There are eleven distinct actions in the environment, each of which represents a different system modification script (e.g., turning off debugging, altering autorun settings, or altering shell behaviour). By mapping these scripts using an action mapping dictionary, the agent can choose from a variety of behaviour-changing actions.

The observation space is defined as a five-dimensional continuous space which is normalized between 0 and 1. This represents abstracted features extracted from the sandbox output. These features are designed to capture relevant indicators of malware behaviour even though the exact nature of the features is abstracted in the code snippet provided. Several internal variables are also initialized such as sample and action indices, a baseline score for reward comparison, and the list of malware samples retrieved from the specified directory.

During agent interactions, the class also configures the Cuckoo Sandbox API URL, which will be used to submit and oversee analytic tasks. Asynchronous retrieval of behavioural reports and submission of updated samples to the sandbox are made possible by this configuration. Overall, this class offers a structured framework with features for dynamic interaction, behavioural manipulation, and logging for reward-based learning that makes it possible to automate the analysis of malware samples using reinforcement learning.

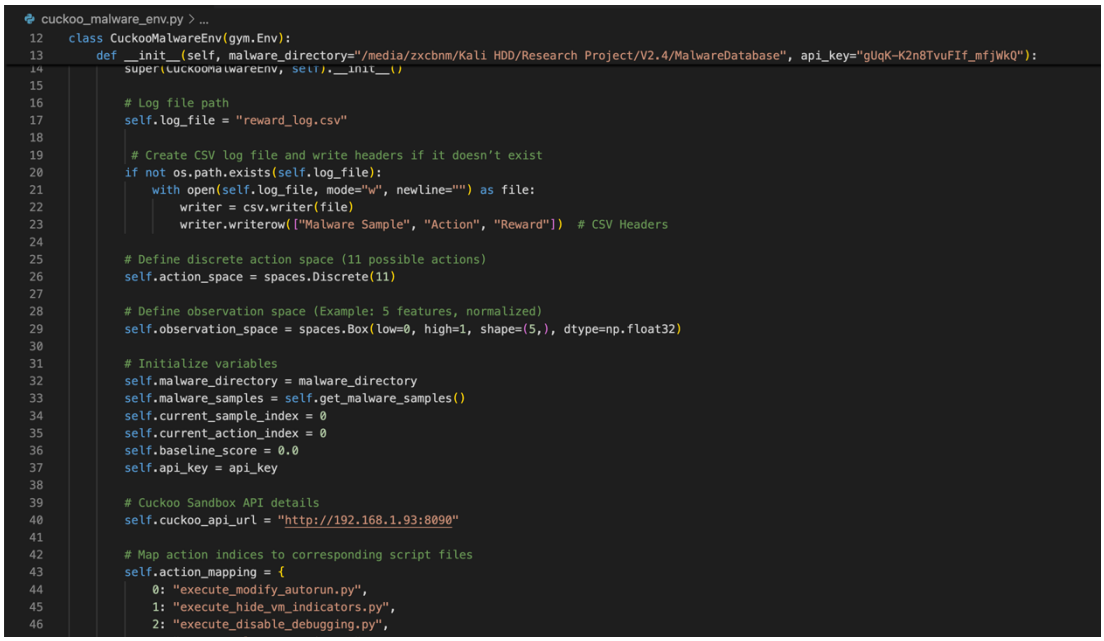


*Figure 2.4: CuckooMalwareEnv Class*

### 2.3.1.3 get_malware_samples() method

Obtaining the list of malware sample file paths from the designated directory is the responsibility of the get_malware_samples() method. It restricts the files in the malware_directory to those with the.exe extension, which are usually executable malware binaries. For every legitimate executable, it creates the whole file path by connecting the directory path and the file name using a list comprehension. This technique makes sure that the environment only runs on pertinent executable samples, which are the Cuckoo Sandbox's intended targets for dynamic analysis. For agent interaction and behavioural monitoring, the generated list is saved and then used to feed samples into the environment one after the other.

```
def get_malware_samples(self):
    """ Retrieves a list of malware sample file paths. """
    return [os.path.join(self.malware_directory, f) for f in os.listdir(self.malware_directory) if f.endswith(".exe")]
```

*Figure 2.5: get_malware_samples method*

### 2.3.1.4 reset() method

The environment is reset for the next malware sample using the reset() technique. It compares the current sample index to the total number of samples to see if all malware samples have been processed. A zeroed observation vector that matches the shape and data type of the observation space is returned by this method and it also prints a message signaling completion if all samples have been processed. This signal is to terminate the process.

The method assigns the next malware sample to self.current_sample if there are any leftover samples choosing it depending on the current index. The run_cuckoo_analysis() function is then used to do a baseline analysis, and the outcome is stored as self.baseline_score. To begin a new interaction cycle for the chosen sample, the action index is reset to zero.

Also a message is printed indicating the environment has been reset for the selected malware sample and returns the current state by calling self.get_state().

```
def reset(self):
    """ Establishes a baseline score for the next malware sample or stops training when all samples are processed. """
    if self.current_sample_index >= len(self.malware_samples):
        print("\033[32mAll malware samples processed. Stopping environment.\033[0m")
        return np.zeros(self.observation_space.shape, dtype=np.float32)  # Prevents unnecessary resets

    # Get the next malware sample
    self.current_sample = self.malware_samples[self.current_sample_index]
    self.baseline_score = self.run_cuckoo_analysis(self.current_sample)  # Run baseline analysis once
    self.current_action_index = 0

    print(f"Reset environment for malware sample: {self.current_sample}")
    return self.get_state()
```

*Figure 2.6: reset() method*

### 2.3.1.5 step() method

The step() method is in charge of moving the environment along by one step, which is equivalent to the agent doing a single action. It performs several tasks, such as termination checks, action execution, environment interaction, reward computation,

logging, and state transition, and it accepts an optional action input (albeit the current implementation depends on an internally managed action index).

For debugging, the method starts by reporting the current action index. Next, it determines if every malware sample has been handled. An empty info dictionary, terminal indications (done=True), and a zeroed observation space are returned by the environment together with a termination message if the current sample index is greater than the number of available samples. This guarantees that when the dataset runs out, the agent will cease interacting.

The procedure then verifies that every specified step for the current sample has been carried out. The sample index is increased to go to the next malware sample if the current action index is larger than the action mapping. The environment ends the episode as previously said if this updated index once more surpasses the sample size that is provided. If not, it chooses the subsequent sample, uses run_cuckoo_analysis() to perform a baseline analysis, resets the action index, and provides the beginning state for the new sample with done=False and a reward of 0.

Run_cuckoo_analysis() is then used to execute the chosen script once more, but this time as a post-action analysis to record the malware's behaviour following the application of the particular behavioural modification by the environment. After that, the environment determines a reward by contrasting the post-action score with the previously acquired baseline score. The calculate_reward() method is used to calculate this reward, and the results are printed for validation. The purpose of the difference between the two scores is to show how the applied action affected the malware's behaviour as seen by the sandbox.

The procedure then gets ready to record the interaction information. After removing the directory path from the malware sample's filename using os.path.basename(), it adds a new row to the log file (reward_log.csv) with the sample name, the executed script, and the calculated reward. Reproducibility and accountability of all environment-agent interactions are guaranteed by this stage.

The method prepares for the following step by increasing the current action index before ending. The done flag is then set in accordance with the results of a reevaluation of whether the current sample and action indices reflect the end of the dataset or action set.

```python
def step(self, action=None):
    """ Applies one action, reanalyzes the malware, and calculates reward. """
    print(f"\033[33mRunning step with action index: {self.current_action_index}\033[0m")

    # Stop training if all malware samples are processed
    if self.current_sample_index >= len(self.malware_samples):
        print("\033[32mAll malware samples processed. Terminating training.\033[0m")
        return np.zeros(self.observation_space.shape, dtype=np.float32), 0.0, True, {}

    # If all actions for this sample are completed, move to the next sample
    if self.current_action_index >= len(self.action_mapping):
        print(f"\033[32mAll actions applied for sample {self.current_sample_index}. Moving to next malware sample.\033[0m")

        # Move to the next malware sample
        self.current_sample_index += 1
        if self.current_sample_index >= len(self.malware_samples):
            print("\033[32mAll malware samples completed. Stopping training.\033[0m")
            return np.zeros(self.observation_space.shape, dtype=np.float32), 0.0, True, {}

        # Select the new sample and reset actions
        self.current_sample = self.malware_samples[self.current_sample_index]
        self.baseline_score = self.run_cuckoo_analysis(self.current_sample)  # Baseline analysis
        self.current_action_index = 0  # Reset action counter for new sample

        return self.get_state(), 0.0, False, {}

    # Apply the current action to the correct sample
    script = self.action_mapping.get(self.current_action_index)
    if script is None:
        raise ValueError(f"\033[32mInvalid action index: {self.current_action_index}")

    print(f"\033[34mRunning action: {script} on {self.current_sample}\033[0m")
```

*Figure 2.7: step() method part 1*

```python
# Run analysis after applying the action
post_action_score = self.run_cuckoo_analysis(self.current_sample, action_scripts=[script])

# Compute reward
reward = self.calculate_reward(self.baseline_score, post_action_score)
print(f"\033[32mReward calculated: {reward} (Baseline: {self.baseline_score}, Post-action: {post_action_score})\033[0m")

# Extract only the malware filename (not the full path)
malware_name = os.path.basename(self.current_sample)

# Log the malware sample name, action, and reward
with open(self.log_file, mode="a", newline="") as file:
    writer = csv.writer(file)
    writer.writerow([malware_name, script, reward])

print(f"Logged: {malware_name}, {script}, {reward}")

# Move to the next action
self.current_action_index += 1

# Stop training after all malware samples are processed
done = (self.current_sample_index >= len(self.malware_samples)) and (self.current_action_index >= len(self.action_mapping))

print(f"\033[31mDEBUG: Sample Index: {self.current_sample_index}, Action Index: {self.current_action_index}, Done: {done}\033[0m")

return np.array(self.get_state(), dtype=np.float32), float(reward), done, {}
```

*Figure 2.8: step() method part*

#### 2.3.1.6 run_cuckoo_analysis() method

In this technique, run_cuckoo_analysis() function takes the corresponding analysis report after providing a malware sample to the Cuckoo Sandbox in order to determine a behaviour-based score. This score examines the malware's activity either in its original state or after running an action script. The malware sample to execute and an optional list of action_scripts are the two arguments the current code takes, although it basically relies on the current_action_index to determine what script to execute. The execution starts by initializing the data to submit and opening the malware sample file in binary mode.

This involves building a payload (data) that instructs the script to run on analysis, such as HTTP headers with required API authorization token, and an indication of files dictionary with the sample. The current index of action is used to choose the script from the action_mapping dictionary. The Cuckoo API endpoint responsible for task generation is then invoked via a POST request. It displays an error message and the process returns the previously calculated baseline score if the API fails to return a successful HTTP status code (200), meaning that analysis cannot proceed. The method checks for the task_id in the JSON response when successfully submitted. The method returns the baseline_score and logs the error when the task_id is not present or the JSON decoding fails. The procedure goes into a polling loop to get the analysis report if a valid task ID is received.

A GET request of the task ID is made in the loop to obtain the task analysis report. The function next decodes the JSON content upon having the report available (HTTP 200). A behaviour score is obtained through invoking the calculate_analysis_score() method with the report if successful in decoding. The behaviour score obtained is returned as the output of the run_cuckoo_analysis() method. An error is logged and the loop pauses for five seconds before attempting again if the JSON will not process. This polling method also controls report generation delays by allowing the method to pause and wait for the Cuckoo Sandbox to finish the analysis before continuing.

```python
def run_cuckoo_analysis(self, sample, action_scripts=[]):
    """ Submits malware to Cuckoo and retrieves analysis score. """
    print(f"\033[34mSubmitting malware to Cuckoo: {sample}\033[0m")  # Debug print

    with open(sample, 'rb') as f:
        files = {'file': f}
        headers = {'Authorization': f'Bearer {self.api_key}'}
        data = {"options": f"script={self.action_mapping[self.current_action_index]}"}
        response = requests.post(f"{self.cuckoo_api_url}/tasks/create/file", headers=headers, files=files, data=data)

    if response.status_code != 200:
        print(f"\033[31mError submitting malware. HTTP Status: {response.status_code}\033[0m")
        return self.baseline_score

    try:
        task_id = response.json().get("task_id")
    except (json.JSONDecodeError, AttributeError):
        print("\033[31mFailed to parse JSON response\033[0m")
        return self.baseline_score

    if not task_id:
        print("\033[31mNo task ID received from Cuckoo.\033[0m")
        return self.baseline_score

    while True:
        report_response = requests.get(f"{self.cuckoo_api_url}/tasks/report/{task_id}/json", headers=headers)
        if report_response.status_code == 200:
            try:
                report = report_response.json()
                #print(f"Analysis Report Received: {report}")  # Debug print
                return self.calculate_analysis_score(report)
            except json.JSONDecodeError:
                print("\033[31mError decoding the JSON response from the report.\033[0m")
        time.sleep(5)
```

*Figure 2.9: run_cuckoo_analysis method()*

### 2.3.1.7 run_cuckoo_analysis() method

This method is returning the current observation of the environment of the state of the system after taking some action. Its current instantiation is creating a placeholder state as one-dimensional NumPy array of size five whose elements are five random floating-point numbers ranging from 0 to 1. They are converted into float32 data type to support the given observation space of the environment.

The placeholder state is used as a temporary stopgap until an actual state description can be created. In an actual implementation, this approach would be replaced or extended to derive salient features from Cuckoo Sandbox analysis reports, e.g., changes in files, changes in traffic, system calls, or process tree structure. Such features would enable better learning and intelligent selection of actions through context provision to the agent concerning what activity of interest is under execution by the malware sample.

Now, the agent learns without accurate feedback from the environment because values are created at random. In order to make learning outputs meaningful and understandable, this must be replaced by a deterministic, behaviour-driven state vector.

```
def get_state(self):
    """ Returns a placeholder state (Replace with meaningful metrics). """
    return np.random.rand(5).astype(np.float32)
```

*Figure 2.10: get_state() method*

#### 2.3.1.8 render() method

To fulfill requirements of the API for the OpenAI Gym environment, the render() method is offered. Its purpose is to offer an option for displaying the current state of the world in which an agent resides. With the mode argument passed to the method, rendering in the format perceivable by humans (usually in terms of textual and graphical output) occurs when set to 'human' as the default.

The render() function is defined in this current implementation is not utilized through the pass statement. That is, it generates no visual output. It is possible to add useful information in future iterations such as the current malware sample being processed, the action performed, reward earned, and the corresponding changes in the environment. Debugging, training monitoring, and checking of the agent's decision-making process can all be enabled by such visualizations.

A concrete functional implementation of render() can prove to be highly beneficial in creating and debugging reinforcement learning agents, especially when the environment's dynamics prove to be complicated or hard to interpret using observations in numerical values.

```
def render(self, mode='human'):
    pass
```

*Figure 2.11: render() method*

#### 2.3.1.9 calculate_reward() method

The calculate_reward() function considers malware sample behaviour before, and after, executing an action in an effort to determine the reward signal for the reinforcement learning agent.
Baseline and post_action, numeric values from sandbox analyzer outputs, must be input arguments.
The post_action indicates to you resulting behaviour after executing an explicit action script, while the baseline indicates to you initial malware sample behaviour before an action ever takes place.

The calculation gives the difference between the baseline and post_action. That serves to constitute the reward. A positive outcome rewards the agent in terms of an increase in intensity of activity or an adjustment, which is positive in terms of its closeness to achieving the environment's goal. A negative outcome penalizes the actor in terms of implying decreased or less desirable outcome. It is 0 for an absence of adjustment in activity.

```
def calculate_reward(self, baseline, post_action):
    """ Rewards improvement; penalizes degradation. """
    return post_action - baseline
```

*Figure 2.12: calculate_reward() method*

#### 2.3.1.10 calculate_analysis_score() method

The numerical score of an Cuckoo Sandbox's analysis report is accessed using the calculate_analysis_score() function. The score is utilized directly by the reward calculation system of the platform and is used to measure the behavioural characteristics of an instance of malware. The function can accept just one parameter, report, and this should be so defined as to be a JSON object which is returned by Cuckoo's API upon completion of sandboxing an instance of malware.

The first thing the function does is check whether report_dict contains an info field with a nested score key and an info key. It extracts the value of the score in report['info']['score'] and assigns it to detected_score if they do. As a part of its debugging output, it also prints the score to the command line. This result, which is an indirect measurement of malware activity, is returned.

The function returns 0 and also prints an error message when unable to get the score in case the necessary keys in the report do not exist. This provides for resilience of the environment despite Cuckoo Sandbox failure in generating an acceptably valid and complete report.

```
def calculate_analysis_score(self, report):
    """ Extracts the 'score' parameter from Cuckoo's report as the detection score. """
    #print(f"Full Cuckoo Report: {json.dumps(report, indent=4)}")  # Debugging output

    # Extract the score directly
    if 'info' in report and 'score' in report['info']:
        detected_score = report['info']['score']
        print(f"\033[36mCuckoo Score Extracted: {detected_score}\033[0m")  # Debugging output
        return detected_score  # Use the score from the report

    print("\033[31mNo 'score' found in the Cuckoo report. Defaulting to 0.\033[0m")
    return 0  # Default to 0 if nothing is found
```

*Figure 2.13: calculate_analysis_score() method*

### 2.3.2 Agent training script

The PPO algorithm in the Stable Baselines3 library is used in train_rl_agent_new.py to create a reinforcement learning training pipeline. It initializes the PPO agent, creates a custom Gym environment, defines hyperparameters, and begins training. Model checkpointing is also added in the script and TensorBoard logging is done.

The model is saved from time to time during training and validated at periodic intervals. Periodic actions are executed by a custom callback. By supporting command-line arguments for loading paths and model saving, training is restarted from a previous checkpoint if one is provided.

#### 2.3.2.1 Required Python libraries for training script

In order to build and train an agent using reinforcement learning, the script begins by loading necessary modules and libraries. EvalCallback to perform evaluations at regular intervals during training, PPO from Stable Baselines3 for training, and gym for use in environments. CuckooMalwareEnv, a specialized environment used for malware investigation, is loaded from the cuckoo_malware_env package. It begins logging training results using configure from Stable Baselines3's logger module.

```
import gym
from stable_baselines3 import PPO
from stable_baselines3.common.callbacks import EvalCallback
from cuckoo_malware_env import CuckooMalwareEnv
from stable_baselines3.common.logger import configure
```

*Figure 2.14: Required Python libraries for training script*

#### 2.3.2.2 Algorithm of agent train script

The script configures a logger to output logs to TensorBoard and the console (stdout), and it sets the logging directory to./ppo_cuckoo_tensorboard/. Performance data and training progress can be tracked with this configuration.

The environment in which the reinforcement learning agent will interact is initialized as the custom environment CuckooMalwareEnv. Multiplying the number of malware samples in the environment by the number of possible actions plus one yields the total number of training steps. This guarantees that the training steps given to the agent cover the whole spectrum of potential interactions with any malware sample.

The 'MlpPolicy' is used for initializing the PPO agent, verbosity is enabled, and training steps are calculated. Agent saves data in given TensorBoard folder. Best model is saved while training and results from the evaluation are saved to)./logs/ folder every 5000 training steps using an evaluation callback (EvalCallback).

The total training steps and the evaluation callback are passed as arguments when calling model.learn() in order to begin training. TensorBoard logging is marked as "PPO_Cuckoo". When training is finished, the trained model is written to ppo_cuckoo_malware and the logger is initialized. A message is printed to mark training and saving of model as completed.

```
log_dir = "./ppo_cuckoo_tensorboard/"
new_logger = configure(log_dir, ["stdout", "tensorboard"])

# Initialize the custom environment
env = CuckooMalwareEnv()

# Calculate total timesteps
num_malware_samples = len(env.malware_samples)
num_actions = len(env.action_mapping)
total_training_steps = num_malware_samples * (num_actions + 1)  # Matches required Cuckoo analyses

print(f"!!!!!! Total Training Steps: {total_training_steps} !!!!!!")

# Initialize PPO RL agent
model = PPO('MlpPolicy', env, verbose=1, n_steps=total_training_steps, tensorboard_log="./
ppo_cuckoo_tensorboard/")

# Setup evaluation callback for Tensorboard
eval_callback = EvalCallback(env, best_model_save_path='./logs/',
                             log_path='./logs/', eval_freq=5000,
                             deterministic=True, render=False)

# Train PPO agent with the exact number of steps
model.learn(total_timesteps=total_training_steps, callback=eval_callback, tb_log_name="PPO_Cuckoo")

# Save the trained model
model.set_logger(new_logger)
model.save("ppo_cuckoo_malware")
print("!!!!!! Model training completed and saved as 'ppo_cuckoo_malware'. !!!!!!")
```

*Figure 2.15: Algorithm of the training script*

### 2.3.3 Action Scripts (Counter evasion strategies)

The action scripts give direct access to the underlying operating system upon which the malware samples run. Each of them represents an analog for some specific system-level operation or change typically found within malware in-the-wild, such as disabling security features, altering autorun settings, or emulating network communications. Such scripts can optionally be called by the agent at training time because each script is translated into distinct actions in the environment's action space. Performing such operations has the consequence of modifying the execution context and observing malware's response as an outcome. This is intended for training the agent to learn activity patterns motivating or inhibiting malicious behaviour.

#### 2.3.3.1 Alter security settings action script

```
# Disable User Account Control (UAC)
Set-ItemProperty -Path "HKLM:\Software\Microsoft\Windows\CurrentVersion\Policies\System" -Name "EnableLUA"
-Value 0

# Disable Windows Defender
Set-MpPreference -DisableRealtimeMonitoring $true
```

*Figure 2.16: execute_alter_security.ps1 script*

Two actions are taken by this script with the intention of changing the default Windows security settings. By setting the EnableLUA registry key to 0 under the path HKLM\Software\Microsoft\Windows\CurrentVersion\Policies\System, the first command turns off User Account Control (UAC). By turning off UAC, the operating system won't ask for elevation when carrying out administrative tasks, potentially enabling processes to operate with more rights without requiring user input. The second command uses the Set-MpPreference cmdlet with the -DisableRealtimeMonitoring flag set to $true to turn off Windows Defender's real-time monitoring. By doing this, the integrated antivirus engine is prevented from actively detecting and reacting to dangerous activities while it is running.

### 2.3.3.2 Action script to change the shell value

```
# Change the default shell to cmd.exe
Set-ItemProperty -Path "HKLM:\Software\Microsoft\Windows NT\CurrentVersion\Winlogon" -Name "Shell" -Value
"cmd.exe"
```

*Figure 2.17: execute_change_shell.ps1 script*

This script changes the default system shell to cmd.exe by modifying the Windows registry. It sets "cmd.exe" as the value of the Shell key and targets the registry path HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon. This alteration causes the command prompt executable to be used by the system in place of the default shell, which is usually explorer.exe. This change causes the user session to start cmd.exe instead of the default graphical desktop environment when the user logs in. In order to avoid UI-based security measures or to design a simple interface, this behaviour is frequently employed.

### 2.3.3.3 Action script to disable debugging services

```
# Disable Windows Debugging Service
Stop-Service -Name "Dbgsvc" -Force -ErrorAction SilentlyContinue
Set-Service -Name "Dbgsvc" -StartupType Disabled
```

*Figure 2.18: execute_disable_debugging.ps1 script*

The Windows Debugging Service (Dbgsvc) is disabled with this script. The first line prevents any errors with -ErrorAction SilentlyContinue and stops the service right away with Stop-Service and the -Force option. The second line prevents the service from automatically starting on system boot by setting the service's startup type to Disabled with the Set-Service cmdlet. The system's capabilities to perform debugging operations may be compromised if this service is disabled and may impede monitoring systems or analyzing tools. Without available debugging techniques to use, this practice can be used to assess malware behaviour.

### 2.3.3.4 Action script to disable specific services

```
# Disable Windows Search Service
Stop-Service -Name "WSearch" -Force -ErrorAction SilentlyContinue
Set-Service -Name "WSearch" -StartupType Disabled

# Disable Superfetch Service
Stop-Service -Name "SysMain" -Force -ErrorAction SilentlyContinue
Set-Service -Name "SysMain" -StartupType Disabled
```

*Figure 2.19: execute_disable_services.ps1 script*

The Windows Search (WSearch) and Superfetch (SysMain) Windows services are disabled by this script. Both the services are stopped immediately with the Stop-Service cmdlet and with the -Force option and error messages are ignored when run with the -ErrorAction SilentlyContinue option. The Set-Service cmdlet sets the StartupType to Disabled to prevent the services from being run when the system boots up. Disk search operations are decreased when WSearch is disabled and background improvement operations are suspended when SysMain is disabled. The operations can be performed to inspect malware behaviour on systems with decreased background operation or to reduce system noise.

### 2.3.3.5 Action script to hide VM indicators

```
# Remove VirtualBox registry keys
Remove-Item -Path "HKLM:\Software\Oracle\VirtualBox Guest Additions" -Recurse -Force -ErrorAction
SilentlyContinue

# Remove Microsoft Virtual Machine registry keys
Remove-Item -Path "HKLM:\Software\Microsoft\Virtual Machine" -Recurse -Force -ErrorAction SilentlyContinue
```

*Figure 2.20: execute_hide_vm_indicators.ps1 script*

The objective of this script is to remove registry keys associated with virtual machine environments. The first one removes the HKLM:\\\Software\\\Oracle\\\VirtualBox Guest Additions registry path with VirtualBox guest tools identifiers. The second one removes the HKLM:\\\Software\\\Microsoft\\\Virtual Machine registry path which may signify the presence of a Microsoft virtual machine installation. Both commands use the Remove-Item cmdlet with the -Recurse and -Force switches to remove the keys

and their values silently and don't show errors with -ErrorAction SilentlyContinue. Removing these entries may be employed to conceal the virtualized environment from executables that attempt to identify sandboxing or virtualization.

#### 2.3.3.6 Autorun disabling action script

```
# Disable Autorun for all drives
Set-ItemProperty -Path "HKLM:\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer" -Name
"NoDriveTypeAutoRun" -Value 0xFF
```

*Figure 2.21:execute_modify_autorun.ps1 script*

This script disables Autorun on all drive types in Windows. It alters the registry entry at HKLM:\\\\Software\\\\Microsoft\\\\Windows\\\\CurrentVersion\\\\Policies\\\\Explorer by setting the value to 0xFF on NoDriveTypeAutoRun. 0xFF disables Autorun on all drive types including removable, fixed, network, CD-ROM, and RAM disks. It does this with the Set-ItemProperty cmdlet. Disabling Autorun is one countermeasure to prevent autorunning potentially malicious software on removable storage media.

#### 2.3.3.7 Action script to modify network keys

```
# Set network location to Private
Set-NetConnectionProfile -Name "Ethernet" -NetworkCategory Private

# Disable network discovery
Set-NetFirewallRule -DisplayGroup "Network Discovery" -Enabled False
```

*Figure 2.22: execute_modify_network_keys.ps1 script*

This script configures network-related options on a Windows platform. The first command configures the networking category of the "Ethernet" connection to Private using the Set-NetConnectionProfile cmdlet. Classification determines how the firewall and sharing options are applied to the connection. The second command disables all the rules on the display group "Network Discovery" by applying the -Enabled parameter to False with the Set-NetFirewallRule cmdlet. Network discovery disabled will stop the system from being able to see or be visible to others on the network. Those modifications alter the visibility and connect profile of the system and can impact detection or response capabilities related to the network.

### 2.3.3.8 Action script to simulate power state changes

```
# Simulate Sleep Mode
Add-Type -AssemblyName System.Windows.Forms
[System.Windows.Forms.Application]::SetSuspendState('Suspend', $false, $false)

# Simulate Hibernate Mode
Add-Type -AssemblyName System.Windows.Forms
[System.Windows.Forms.Application]::SetSuspendState('Hibernate', $false, $false)
```

*Figure 2.23: execute_power_state_simulation.ps1 script*

This code emulates power state changes on a Windows system. The first block triggers sleep mode by calling the SetSuspendState function with the 'Suspend' argument to place the system into a power state without turning off the system. The second block triggers hibernate mode by calling the function with 'Hibernate'. Both call the.NET System.Windows.Forms.Application class by utilizing PowerShell's Add-Type to import the required assembly. The arguments $false, $false are passed to indicate hibernation is not forced and wake events are not disabled. The commands may be used to see how malware will react when system power states are being changed during the course of execution.

### 2.3.3.9 Action script to remove VM entries

```
# Remove VMware registry keys
Remove-Item -Path "HKLM:\Software\VMware, Inc." -Recurse -Force -ErrorAction SilentlyContinue

# Remove VirtualBox registry keys
Remove-Item -Path "HKLM:\Software\Oracle\VirtualBox" -Recurse -Force -ErrorAction SilentlyContinue
```

*Figure 2.24: execute_remove_vm_entries.ps1 script*

This script removes registry keys for virtual platforms to reduce a virtual environment's traceability. The first one removes the HKLM:\\\\Software\\\\VMware, Inc. key that contains metadata and configuration for VMware. The second one removes the HKLM:\\\\Software\\\\Oracle\\\\VirtualBox key containing identifiers for VirtualBox. Both commands make use of the Remove-Item cmdlet along with -Recurse and -Force parameters to delete the registry keys and their subkeys quietly and suppress errors using -ErrorAction SilentlyContinue. The actions are typically

used to conceal virtualization artifacts that are looked for by software attempting to determine execution in a sandbox or virtual machine.

### 2.3.3.10 Action script simulate user behaviour

```powershell
# Import necessary assemblies
Add-Type -AssemblyName System.Windows.Forms
Add-Type -AssemblyName System.Drawing

# Function to move the mouse smoothly between two points
function Move-MouseSmoothly {
    param (
        [Parameter(Mandatory=$true)]
        [System.Drawing.Point]$StartPoint,

        [Parameter(Mandatory=$true)]
        [System.Drawing.Point]$EndPoint,

        [Parameter(Mandatory=$false)]
        [int]$Steps = 100,

        [Parameter(Mandatory=$false)]
        [int]$Duration = 2000 # in milliseconds
    )

    $interval = $Duration / $Steps
    $deltaX = ($EndPoint.X - $StartPoint.X) / $Steps
    $deltaY = ($EndPoint.Y - $StartPoint.Y) / $Steps

    for ($i = 1; $i -le $Steps; $i++) {
        $newX = [Math]::Round($StartPoint.X + ($deltaX * $i))
        $newY = [Math]::Round($StartPoint.Y + ($deltaY * $i))
        [System.Windows.Forms.Cursor]::Position = New-Object System.Drawing.Point($newX, $newY)
        Start-Sleep -Milliseconds $interval
    }
}

# Function to simulate typing with random delays
function Simulate-Typing {
    param (
        [Parameter(Mandatory=$true)]
        [string]$Text
    )

    foreach ($char in $Text.ToCharArray()) {
        [System.Windows.Forms.SendKeys]::SendWait($char)
        # Random sleep between 100 to 300 milliseconds
        Start-Sleep -Milliseconds (Get-Random -Minimum 100 -Maximum 300)
    }
}

# Function to open an application
function Open-Application {
    param (
        [Parameter(Mandatory=$true)]
        [string]$AppPath
    )
```

*Figure 2.25: simulate_user_activity.ps1 script part 1*

```
    Start-Process $AppPath
    # Wait for the application to open
    Start-Sleep -Seconds 2
}

# Function to close Calculator reliably
function Close-Calculator {
    # Attempt to close using 'Calculator' process name
    $calcProcess = Get-Process -Name "Calculator" -ErrorAction SilentlyContinue

    if ($null -ne $calcProcess) {
        Stop-Process -Id $calcProcess.Id -Force
        return
    }

    # If 'Calculator' not found, attempt to close using 'Calculator.exe'
    $calcProcess = Get-Process -Name "Calculator.exe" -ErrorAction SilentlyContinue

    if ($null -ne $calcProcess) {
        Stop-Process -Id $calcProcess.Id -Force
        return
    }

    # As a fallback, attempt to close using 'ApplicationFrameHost' with specific window title
    $appFrameHost = Get-Process -Name "ApplicationFrameHost" -ErrorAction SilentlyContinue | Where-Object {
        $_.MainWindowTitle -eq "Calculator"
    }

    if ($null -ne $appFrameHost) {
        Stop-Process -Id $appFrameHost.Id -Force
    }
}

# Main Function to simulate user interactions
function Simulate-UserInteractions {
    # Define screen boundaries
    $screenWidth = [System.Windows.Forms.SystemInformation]::PrimaryMonitorSize.Width
    $screenHeight = [System.Windows.Forms.SystemInformation]::PrimaryMonitorSize.Height

    # Initial mouse position
    $currentPos = [System.Windows.Forms.Cursor]::Position

    while ($true) {
        # Generate random target position
        $targetX = Get-Random -Minimum 0 -Maximum $screenWidth
        $targetY = Get-Random -Minimum 0 -Maximum $screenHeight
        $targetPos = New-Object System.Drawing.Point($targetX, $targetY)

        # Move mouse smoothly to the target position
        Move-MouseSmoothly -StartPoint $currentPos -EndPoint $targetPos -Steps (Get-Random -Minimum 50 -Maximum 150) -Duration (Get-Random -Minimum 1000 -Maximum 3000)
```

*Figure 2.26: simulate_user_activity.ps1 script part 2*

```
            $currentPos = $targetPos

            # Random sleep between movements to mimic human behavior
            Start-Sleep -Seconds (Get-Random -Minimum 1 -Maximum 3)

            # Randomly decide to type something
            if ((Get-Random -Minimum 0 -Maximum 10) -lt 3) { # 30% chance
                # Open Notepad
                Open-Application -AppPath "notepad.exe"

                # Simulate typing
                Simulate-Typing -Text "This is a simulated user input for testing purposes."

                # Close Notepad after typing
                Stop-Process -Name "notepad" -Force
            }

            # Randomly decide to open another application
            if ((Get-Random -Minimum 0 -Maximum 10) -lt 2) { # 20% chance
                # Example: Open Calculator
                Open-Application -AppPath "calc.exe"

                # Simulate some activity in Calculator
                Start-Sleep -Seconds (Get-Random -Minimum 2 -Maximum 5)

                # Close Calculator reliably
                Close-Calculator
            }
        }
    }

    # Start simulating user interactions
    Simulate-UserInteractions
```

*Figure 2.27: simulate_user_activity.ps1 script part 3*

This PowerShell script imitates user interaction on a Windows system to create the illusion of human behaviour. It includes options to move the mouse, type keys, open and close windows, and introduce random pauses in between operations. The script begins by importing .NET assemblies used, which are System.Windows.Forms and System.Drawing and are used to provide system input and graphics capabilities.

The script contains the function Move-MouseSmoothly that linearly interpolates from one point on the screen to an arbitrarily defined set of endpoints and uses a user-provided number and length of steps to smoothly move the mouse on the screen with patterns that are human-like in nature. A function called Simulate-Typing simulates the typing by moving through an input string character by character with random pause lengths in between inputs and mimics human-like typing. Open-Application and Close-Calculator functions open and close applications like Notepad and Calculator in an efficient and stable way.

The main execution block, Simulate-UserInteractions, relentlessly mimics user behaviour by selecting random points on the display and drags the mouse between them with breaks in-between and with conditional logic to open programs randomly and simulate activity there. As an illustration, it could open Notepad and type in a message or open Calculator and pause and close with several fallback methods to complete the task. The entire setup is designed to create an artifact trail of legitimate user input and system behaviour patterns and can be applied to something like anti-evasion detection or sandbox interaction enhancement.

### 2.3.4 Custom auxiliary module for Cuckoo

```python
import os
import subprocess
import logging
from lib.common.abstracts import Auxiliary

log = logging.getLogger(__name__)

class RunScript(Auxiliary):
    """Executes a PowerShell script before or during analysis."""

    def __init__(self):
        Auxiliary.__init__(self)

    def start(self):
        """Executes the specified PowerShell script before malware execution."""
        log.info("🔹 RunScript auxiliary module started successfully! 🔹")

        script_path = self.task.options.get("script")  # Extract script from options

        if script_path:
            log.info("Executing auxiliary PowerShell script: {}".format(script_path))

            if os.path.exists(script_path):
                try:
                    result = subprocess.run(
                        ["powershell.exe", "-ExecutionPolicy", "Bypass", "-File", script_path],
                        capture_output=True, text=True, check=True
                    )
                    log.info("Script Output: {}".format(result.stdout))
                    log.error("Script Error: {}".format(result.stderr))
                except Exception as e:
                    log.error("Error executing script {}: {}".format(script_path, e))
            else:
                log.warning("Script {} not found. Skipping execution.".format(script_path))

        return True  # ✅ Ensure the function returns a boolean
```

*Figure 2.28: RunScript auxiliary module to execute powershell scripts in guest OS*

The auxiliary module RunScript which is a custom addition to Cuckoo Sandbox. This will execute any provided PowerShell script in the guest OS alongside or before malware analyzing. It originates from Cuckoo's plugin system foundation Auxiliary class and is triggered through the start() method. It allows analysts to set up pre-analytical environments automatically through automated execution or counter-evasion techniques and user interaction simulations directly through the workflow in the sandbox.

On being called, the module takes the script path from the task options by calling self.task.options.get("script"). It calls the script if the path and file are both available

by calling subprocess.run() with the powershell execution policy set to Bypass. The execution output and error are captured and logged through Python's logger module. In the event that the execution fails or the file is not found, corresponding warning or error messages are logged. In any case, the function returns True to signify successful execution of the auxilary module to aid the plugin interface of the sandbox.

### 2.3.5 Testing the agent

Each of the significant components of the RL agent, environment actions, the reward system, and sandbox communication were tested individually before doing end-to-end system testing. Unit testing tried to make sure state retrieval, reward computation, and execution of PowerShell scripts worked and produced the correct results under typical and edge-case conditions.

Integration testing assessed the extent to which the custom Gym environment integrated with the Cuckoo analysis pipeline. Integration tests comprised malware sample submission, execution of corresponding PowerShell-based counter-evasion actions and retrieving behavioural scores from the sandbox. Attention to details was given to ordering and operation timing issues, i.e., making sure that baseline and follow-on action analysis occurred in proper ordering and that operations were reliably executed within the guest VM before beginning analysis.

Execution logs, reward logs, and Cuckoo reports were examined during testing to confirm data flow and traceability. Environmental logging capabilities were used to verify that action was being performed in the correct way and that the outcome of the analysis was being logged correctly. Controlled test instances, which consisted of known benign and malware samples, were used to observe the assignment behaviour of the agent under controlled conditions. The tests confirmed that the environment was performing as intended and could support reinforcement learning with behaviour-based feedback with dynamic analysis.

#### 2.3.5.1 Test Cases

*Table 2.1: Test Case 1*

| **Test Case ID** | **01** |
|---|---|
| **Test Case Scenario** | Verify that the environment executes a PowerShell action and logs the correct reward. |
| **Inputs** | Malware Sample: freeyoutubedownloader.exe,<br>Action: execute_disable_debugging.ps1 |
| **Expected Output** | Reward is correctly computed and logged based on post-analysis score. |
| **Action Result** | Action script was executed; reward was calculated and saved in reward_log.csv |
| **Status (pass/fail)** | Pass |

*Table 2.2: Test Case 2*

| **Test Case ID** | **02** |
|---|---|
| **Test Case Scenario** | Confirm that the RL environment skips execution if malware samples are exhausted. |
| **Inputs** | All available malware samples processed (22 total samples) |
| **Expected Output** | Environment returns zeroed state and terminates further training. |
| **Action Result** | Reset returned zeroed observation; agent training was halted as expected. |
| **Status (pass/fail)** | Pass |

*Table 2.3: Test Case 3*

| **Test Case ID** | **03** |
|---|---|
| **Test Case Scenario** | Verify that the correct script file is selected for a given action index. |
| **Inputs** | Action Index: 5, Expected Script: execute_change_shell.py |
| **Expected Output** | Script path resolves to correct filename, and script executes in guest VM. |
| **Action Result** | execute_change_shell.py was correctly selected and executed remotely. |
| **Status (pass/fail)** | Pass |

*Table 2.4: Test Case 4*

| | |
|---|---|
| **Test Case ID** | **04** |
| **Test Case Scenario** | Verify that the reward calculation correctly reflects a reduction in detection score. |
| **Inputs** | Baseline Score: 8.5, Post-Action Score: 6.2 |
| **Expected Output** | Reward = 3.2 |
| **Action Result** | calculate_reward() returned 3.2 and value was logged with correct sample and action |
| **Status (pass/fail)** | Pass |

*Table 2.5: Test Case 5*

| | |
|---|---|
| **Test Case ID** | **05** |
| **Test Case Scenario** | Check that the environment handles missing PowerShell script paths gracefully. |
| **Inputs** | Script path does not exist on the guest VM |
| **Expected Output** | Error is logged, and script is skipped without interrupting analysis. |
| **Action Result** | Warning message was logged; execution continued to next step |
| **Status (pass/fail)** | Pass |

# 3. RESULTS AND DISCUSSION

This chapter will demonstrate the results of the conducted research. The intention is to experiment with the system to see if it can use counter-evasion methods through scripted behaviour and extract behaviour-based feedback to train agents from it. Results are organized by patterns in reward, the behaviour of agents to diverse malware behaviour, and the system-level difference impact. Results are supplemented by logs and sandbox reports and training performance metrics too.

## 3.1 Results

The behaviour in the training set demonstrates that numerous counter-evasion actions were properly executed on malware samples and respective rewards were recorded on the basis of detected behaviour changes being found in the sandbox. In the FreeYoutubeDownloader.exe sample, execute_modify_autorun.py, execute_disable_debugging.py, and execute_remove_vm_entries.py more frequently resulted in an award with an average award value of 0.8. The maximum award experienced in this sample was 1.2 with the action execute_hide_vm_indicators.py. The inference being that hiding virtualization indicators resulted in the maximum behavioural change in the malware execution behaviour most likely to make it undetectable or more behaviourally deviant.

Reward values were calculated by comparing the post-action analysis score with the baseline analysis score in one single script. More value in reward corresponds to more divergence from the baselining behaviour and means that the action performed did impact the malware in some manner that was detected by the sandbox. Static reward values on more actions further confirm that some malware is deterministic under controlled manipulation and variance in scores show sensitivity to controlled system-level manipulation.

As can be inferred from the training logs of the PPO model, 190 timesteps were faced during the training and this amounts to 190 action uses on all malware samples. ep_len_mean and ep_rew_mean were 190 and 7.6, respectively. Both are equal to the logged per-action values and so this implies that the environment had effectively tracked the rewards throughout several episodes. The training completed approximately 32,849 seconds and one iteration elapsed in the logged session.

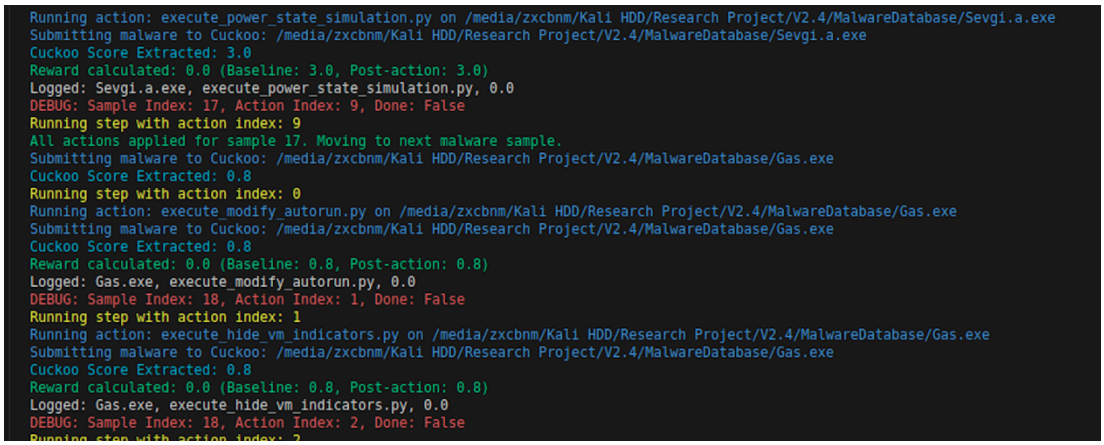


*Figure 3.1: Agent Training Process*

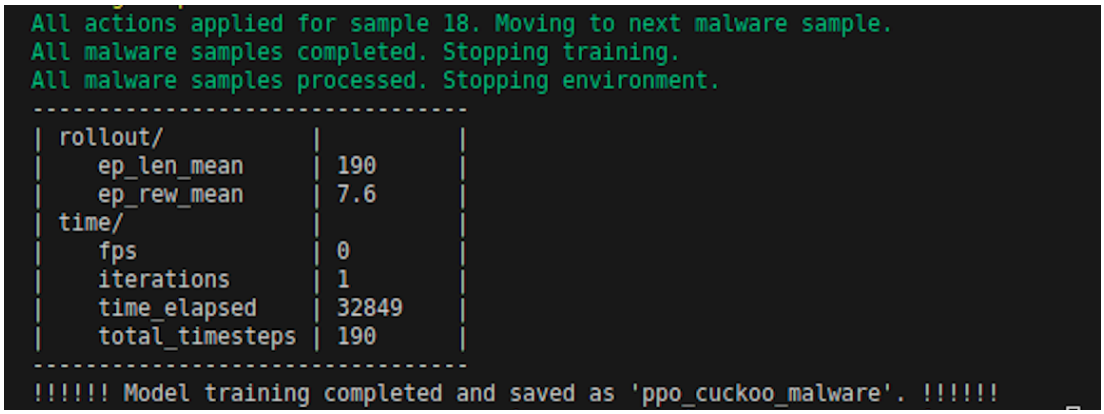


*Figure 3.2: Final Output*

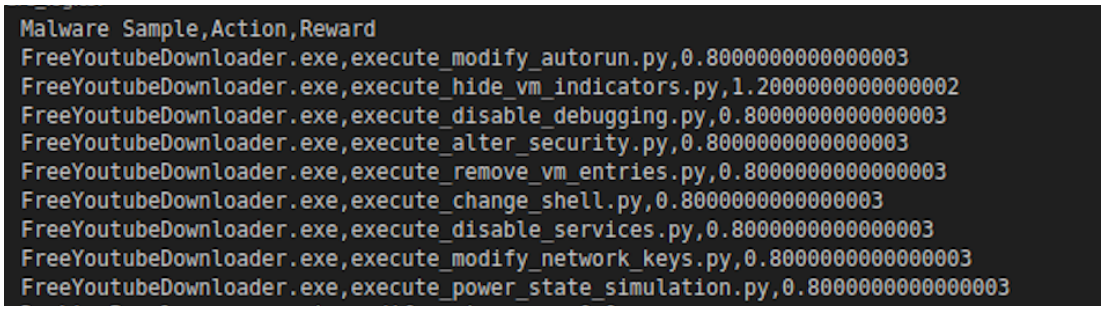


*Figure 3.3: Reward Allocation*

### 3.2 Research findings

The research demonstrated that some evasion techniques implemented ahead of malware execution produced measurable behavioural variations that were identified by the sandbox. Behaviours such as hiding virtualization artifacts (execute_hide_vm_indicators.py) provided increased reward values in the majority of the samples and proved that malware behaviour changes when virtual environment checks are hidden. This supports the presumption that malware tends to perform environmental checks and adapt execution to avoid detection when the environment is virtual or sandboxed.

The second notable observation was that the assignment of reward by most action-script pairs was stable and reproducible. Commands like execute_disable_debugging.py, execute_modify_network_keys.py, and execute_change_shell.py elicited similar rewards around 0.8 when run on identical instances and thus show these modifications reliably impact the visible behaviour of malware. Scripts with negligible or no impact on execution visibility or system protection garnered smaller or similar rewards, confirming the environment effectively distinguishes influential and irrelevant modifications.

Aside from action-specific outcomes, the testing showed the role of ordering and timing in applying changes to the analysis environment. Pre-submission scripts affected the detection capacity of the sandbox reliably, with late or in-mid-analysis injected scripts having smaller or intermittent effect. This points to evasive malware being more probable to adapt and or impede its behaviour in the early stages of execution, highlights the significance of preparation to the environment prior to execution. The study promotes the implementation of these evasion-conscious modifications on the initialization of the sandbox with an aim to realizing maximum behaviour observability. The experiment also demonstrated how the application of reinforcement learning in this context enables the system to discover what modifications provide the biggest behaviour transformation. The first version itself could learn to establish a stable reward profile and demonstrated how it recognized

and rewarded proper behaviour. It finds use in automated evasion platforms that train agents to incrementally adjust environment configurations to adapt to evolving malware behaviour. In general, the experiment's findings confirm that behaviour-aware scripting with learning-based decision making offers an effective platform to tap into dynamic malware analysis.

### 3.3 Discussion

The experience of applying reinforcement learning to the Cuckoo Sandbox demonstrates how system modifications created by it are found to produce measurable and tangible changes in malware behaviour. Replicable patterns of reward on specific actions validate that scoring through the sandbox can be effectively used as an RL loop back to the agent. The approach allows the environment to adjust itself automatically by learning what system modifications uncover concealed or disguised behaviours of evasive malware. Focus on environment modification rather than on malware manipulation allowed realism to be maintained with increased analytical insight.

One of the more notable discoveries through testing was the effectiveness of particular counter-evasion techniques, namely against virtualization and debugging artifacts. Execute_hide_vm_indicators.py and execute_disable_debugging.py scripts would produce high behavioural scores. This is characteristic of real-world evasion logic used by many malware families where functionality is disabled when these artifacts are detected. The reinforcement learning model learning to identify and prefer these behaviours demonstrates that it can generalize beneficial behaviour with experience given a comparative small initial set of samples and actions.

However, this work was constrained by runtime performance and scalability issues too. As each action-sample pair requires the full-cycle sandboxing analysis, learning is labor-intensive—highlighted by the 32,849-second runtime one round takes. Furthermore, rewards are based on one scalar value (ssandboxed score), which in some malware may not capture the complete behavioural extent. Subsequent versions of this system can incorporate more high-fidelity state features from API calls, file system

operations, or hosts' network traffic to enrich the observation and learning space. The project in its entirety shows that reinforcement learning can effectively optimize sandboxes and uncover otherwise latent malicious activity. The structure of the action being modular makes extensibility easy and opens the way to having future work take advantage of more complex or adaptive evasion countermeasures. Though performance and dataset diversity are both areas to be improved, current implementation offers an empirically validated basis on which learning-based malware behaviour discovery can be explored.

# 4. CONCLUSION

The objective of this study was to investigate the application of reinforcement learning to enhance dynamic malware analysis through environment manipulation. An environment that had been set up to interface with the Cuckoo Sandbox permitted scripted system modifications to be made before samples of malware were run. A proportionally related reward signal to the behavioural scores being monitored by the sandbox was given to the environment to initialize and presented to an existing learning agent. This gave the agent the facility to identify what modifications were effective to cause visible behaviour by potentially evasive malware and to identify an optimal observability-improving policy to maximize the observability of these behaviours

The experimental setup consisted of a set of PowerShell scripts applied with ordinary counter-evasion techniques like disabling debug services, eliminating virtualization-based indicators, modifying autorun properties, and emulating user interaction. The scripts were used as atomic operations in the action set of the agent. In all malware samples, the agent called once on an individual script, uploaded the sample to the sandbox to be analyzed and retrieved a score from the sandbox and compared it to a baseline to compute a value for the reward. The experiment iterated on several samples and actions so that the agent could acquire experience and adapt its behaviour based on empirical learning.

Proximal Policy Optimization (PPO) was used to train the model due to its stability and suitability with environments with continuous nature. PPO operates by batching experiences and employing clipped-policy updates to make it robust against the nature of sparse and noisy reward signals characteristic in environments in sandboxes. The train setup was tuned to work with short episodes to simulate the fixed action count by instance malware. The train process could capture interaction traces, values in the form of rewards, and selection actions, which confirmed the performance under controlled environments of the agent-environment loop.

There were consistent behaviour patterns in the reward on several samples of malware under testing. Highest among these was the high reward on the virtualization hint obfuscation script whose malware alters behaviour when it does not realize it is being run under virtualization. Other scripts disabling debugging capabilities or modifying initialization options were moderately high-rewarding scoring. What this means is that these modifications need to impact execution context or visibility in some way and yield noticeable differences captured by Cuckoo's behaviour scoring system.

Reward logs showed several malware samples behaved similarly when responding to some changes, which showed shared evasion methods. The finding supports the conjecture that modern malware frequently employs sandbox-detection logic and conditionally executed behaviour. Executing some counter-evasion scripts beforehand allowed the system to bypass or anticipate these checks and achieve greater behavioural insight. The feasibility of environment-level action-scripting as a mean to enhance sandboxed malware analysis is thus validated by the finding.

A notable observation was the reproducibility of the value of the reward across runs. As an illustration point, actions that reliably evaded detection in one sample were similarly valued when applied to others. Reproducibility is an advantageous characteristic because it promotes convergence by reducing noise in the reward signal. It further implies that the model should generalize policies trained on malware families in the case of similar evasion logic.

While the system performed according to expectations, some limitations became apparent once implemented and put to the test. The most significant limitation is the duration an action-sample pair in the sandbox takes to complete an analysis cycle. Because the environment must wait on the computation of the sandbox on each given sample, training is computationally expensive and takes very long to complete. In the current implementation, it took over 32,000 seconds to complete one run with 190 timesteps, inhibiting training scalability and restricting experimentation.

Further, the observation space used placeholder data rather than dynamic aspects of the sandbox reports. Even if the reward function itself took its origins from the real Cuckoo reported score, the state returned to the agent was an array rather than some representation of structured malware behaviour encoded state. It keeps the agent from conditioning on relevant input and makes the learning partially blind. The future enhancements need to include structured state representation in the form of API call counts, process tree depth, system registry changes, or file events.

The experiments showed that there was an impact on the result by the actions taken in a specific sequence. Certain behaviours of malware were only invoked if there were certain environmental conditions present at the time of execution. This is indicative of the need for sequence-aware agents or multi-action policies that can execute a sequence of modifications prior to samples being executed. A future iteration of this system may investigate action chaining or recurrent policy networks to address state changes due to the elapse of time.

Aside from being applied to this work on calculating reward, sandbox scores are one unified measure and do not automatically capture all aspects of malware behaviour. Some adversarial behaviour is evasive or low scoring without being significant statistically. Adding other measures i.e., persistence tools, anti-analysts check, or network abnormalities—towards the reward system would provide more informative feedback to the learner. It would allow the agent to better identify evasive behavioural patterns and to distinguish highly evasive and moderately evasive samples. The modularity in the action-script interface and the environment allows the system to scale to ongoing experimentation. New action capabilities can be added by adding new scripts and casting them into the environment so that the system can expand with the discovery of newer evasion techniques. The policy of the agent can be stored and tested with and without retraining on progressively harder datasets. The reproducibility of the system is similarly well-fitted to the system being used in real-world adaptive malware pipeline workloads.

In conclusion, the project demonstrated that reinforcement learning could be used to enable environment-level transformation in evasive malware analysis. With the integration with Cuckoo Sandbox and the custom RL environment, the system enabled data driven exploration for counter-evasion techniques. Aside from the requirement to continue improving state representation and enhancing performance, the discovery affirms the feasibility of the paradigm. The work presents avenues open to future systems that can discover hidden behaviour in advanced malware through adaptive sandbox setups through learning automatically.

# REFERENCES


[1] E. Debas, N. Alhumam, and K. Riad, "Unveiling the dynamic landscape of malware sandboxing: A comprehensive review," *Preprints*, 2023. doi: 10.20944/preprints202312.1009.v1.

[2] Essien and Ele, "Cuckoo Sandbox and Process Monitor (Procmon) performance evaluation in large-scale malware detection and analysis," *British Journal of Computer, Networking and Information Technology*, vol. 7, no. 4, pp. 8–26, 2024, doi: 10.52589/bjcnit-fcedoomy.

[3] T. Quertier, B. Marais, S. Morucci, and B. Fournel, "MERLIN -- malware evasion with reinforcement LearnINg," *arXiv [cs.CR]*, 2022. [Online]. Available: http://arxiv.org/abs/2203.12980

[4] T. T. Nguyen and V. J. Reddi, "Deep reinforcement learning for cyber security," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 8, pp. 3779–3795, 2023, doi: 10.1109/TNNLS.2021.3121870.

[5] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: Bradford Books, 1998.

[6] A. Brown, M. Gupta, and M. Abdelsalam, "Automated machine learning for deep learning based malware detection," *Comput. Secur.*, vol. 137, no. 103582, p. 103582, 2024, doi: 10.1016/j.cose.2023.103582.

[7] M. S. Akhtar and T. Feng, "Evaluation of machine learning algorithms for malware detection," *Sensors (Basel)*, vol. 23, no. 2, p. 946, 2023, doi: 10.3390/s23020946.

[8] B. Bokolo, R. Jinad, and Q. Liu, “A Comparison Study to Detect Malware using Deep Learning and Machine learning Techniques,” in *2023 IEEE 6th International Conference on Big Data and Artificial Intelligence (BDAI)*, IEEE, 2023, pp. 1–6.

[9] M. A. EfendyMail, Department of Information Technology and Communication, Politeknik Mersing Johor, 86800 Johor, Malaysia, M. F. Ab Razak, M. Ab Rahman, Faculty of Computing, Universiti Malaysia Pahang, 26600 Pahang, Malaysia, and Department of Information Technology and Communication, Politeknik Mersing Johor, 86800 Johor, Malaysia, “Malware detection system using Cloud Sandbox, machine learning,” *Int. J. Comput. Syst. Softw. Eng.*, vol. 8, no. 2, pp. 25–32, 2022, doi: 10.15282/ijsecs.8.2.2022.3.0100.

[10] D. Gibert, M. Fredrikson, C. Mateu, J. Planes, and Q. Le, “Enhancing the insertion of NOP instructions to obfuscate malware via deep reinforcement learning,” *Comput. Secur.*, vol. 113, no. 102543, p. 102543, 2022, doi: 10.1016/j.cose.2021.102543.

[11] J. Singh and J. Singh, “Detection of malicious software by analyzing the behavioral artifacts using machine learning algorithms,” *Inf. Softw. Technol.*, vol. 121, no. 106273, p. 106273, 2020, doi: 10.1016/j.infsof.2020.106273.

[12] D. Li, S. Cui, Y. Li, J. Xu, F. Xiao, and S. Xu, “PAD: Towards principled adversarial malware detection against evasion attacks,” *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 2, pp. 920–936, 2024, doi: 10.1109/tdsc.2023.3265665.

[13] R. M. Arif *et al.*, “A Deep Reinforcement Learning Framework to Evade Black-box Machine Learning based IoT Malware Detectors using GAN-generated

influential features," *IEEE Access*, pp. 1–1, 2023, doi: 10.1109/access.2023.3334645.

[14] E. Chatzoglou, G. Karopoulos, G. Kambourakis, and Z. Tsiatsikas, "Bypassing antivirus detection: old-school malware, new tricks," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, New York, NY, USA: ACM, 2023.

[15] A. Unnikrishnan, "Financial news-driven LLM reinforcement learning for portfolio management," *arXiv [q-fin.CP]*, 2024. [Online]. Available: http://arxiv.org/abs/2411.11059