Project HyperAdapt: An Agent-based intelligent sandbox design to deceive and analyze sophisticated malware

Final Group Report

24-25J-025

BSc (Hons) degree in Information Technology
Specialized in Cyber Security

Department of Computer Systems Engineering

Sri Lanka Institute of Information Technology
Sri Lanka

April 2025

DECLERATION

"We declare that this is our 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 our 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, we hereby grant to Sri Lanka Institute of Information Technology, the non-exclusive right to reproduce and distribute our dissertation, in whole or in part in print, electronic or other medium. we retain the right to use this content in whole or part in future works (such as articles or books)."

Name	IT Number	Signature
Perera W M. M. S. D. S.	IT21261046	
Dias M. A. S. S. A.	IT21261664	
Dilhara W. M. A.	IT21299452	
Vithanage V. K. M..	IT21300950	

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to **Mr. Amila Senarathne**, Senior Lecturer and Head of the Industry Engagement Unit at the Sri Lanka Institute of Information Technology (SLIIT), for his valuable guidance, continuous encouragement, and constructive supervision throughout the course of our research. His mentorship was instrumental in directing our efforts and achieving the objectives of this study.

We also extend our heartfelt thanks to our co-supervisor, **Mr. Deemantha Siriwardana**, Assistant Lecturer, for his consistent support, insightful feedback, and dedication to helping us overcome challenges during this project.

Our deepest appreciation goes to our external supervisor, **Ms. Chethana Liyanapathirana**, Assistant Professor at the Department of Mathematics, Computer Science, and Digital Forensics, Commonwealth University of Pennsylvania. Her expert advice, external perspective, and thoughtful suggestions significantly contributed to the quality and depth of our research.

We are also grateful to the **Department of Computer Systems Engineering at SLIIT** for providing the academic resources, facilities, and a conducive environment that enabled us to carry out our research effectively.

Lastly, we would like to acknowledge the valuable contributions of each member of our project group. This research would not have been possible without the teamwork, dedication, and shared commitment of all group members.

We are truly thankful to everyone who supported and inspired us throughout this journey.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Description

1. INTRODUCTION

1.1. Background Literature

Background Literature - Offensive RL Component

Offensive perspectives: reinforcement learning for Malware Evasion

As malware detection techniques become increasingly intelligent through the integration of machine learning, so too have malware authors adopted adaptive mechanisms to evade these detection systems. A growing area of offensive cybersecurity research involves the use of reinforcement learning (RL) to train malware agents capable of learning how to bypass defenses in sandbox environments [3][4][5]. RL differs from traditional supervised approaches by enabling agents to make decisions based on environmental feedback rewarding actions that lead to evasion and penalizing those that result in detection [4].

This model aligns well with the adversarial nature of malware evasion. By interacting with the sandbox (e.g., Cuckoo), an RL-based malware agent can iteratively refine its behavior to minimize the likelihood of detection. Actions such as delaying execution, altering API call sequences, or triggering decoy functionality are commonly rewarded when they lead to a stealthy profile [7][8].

Key frameworks and case studies

Several RL-based malware generation frameworks have emerged in recent years:

- **MERLIN** uses Deep Q-Networks (DQN) to generate malware samples that iteratively modify their behavior to evade static and dynamic classifiers. The

system demonstrated that even minor code transformations could produce significant improvements in evasion rates [3][7].

The figure below illustrates the application of reinforcement learning (RL) in generating adaptable malware. Within this architecture, a reinforcement learning (RL) agent engages with an environment that consists of a harmful Portable Executable (PE) file and a detection solution, such as a security system that use machine learning. The agent obtains the present condition of the environment, formulates decisions regarding the modifications to be made to the malware, and carries out these activities. The changed malware is then assessed by the detection system in the environment, which provides feedback in the form of a reward. A positive reward signifies successful evasion, whereas a negative reward suggests detection. The feedback loop enables the agent to progressively enhance the virus, consistently enhancing its capacity to dodge both static and dynamic analysis.

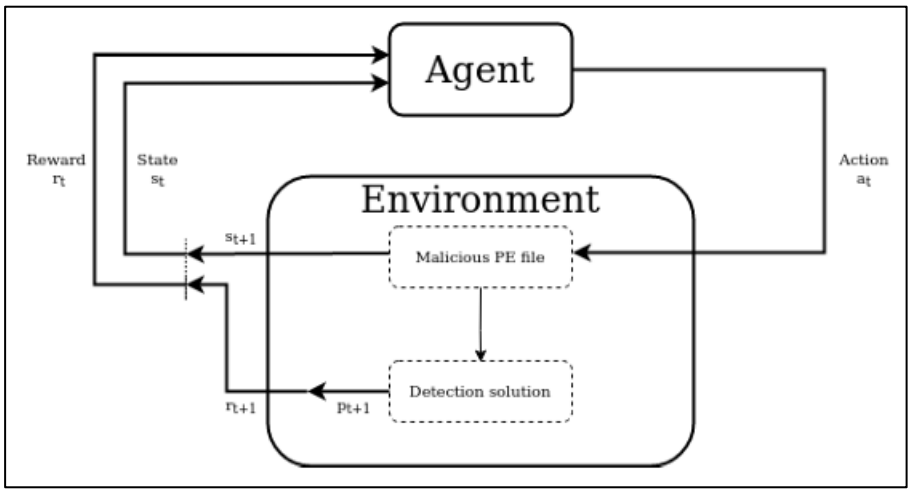


Figure 1: Merlin Reinforcement learning framework representation with detailed environment [7]

The figure above illustrates the application of reinforcement learning (RL) in generating adaptable malware. Within this architecture, a reinforcement learning (RL) agent engages with an environment that consists of a harmful Portable Executable (PE) file and a detection solution, such as a security system that use machine learning. The agent obtains the present condition of the environment, formulates decisions regarding the modifications to be made to the malware, and carries out these activities.

- **MAB-Malware** reframes malware generation as a Multi-Armed Bandit problem, optimizing action selection by balancing exploration and exploitation. This framework showed high success in evading both static classifiers and commercial antivirus software [4][9].

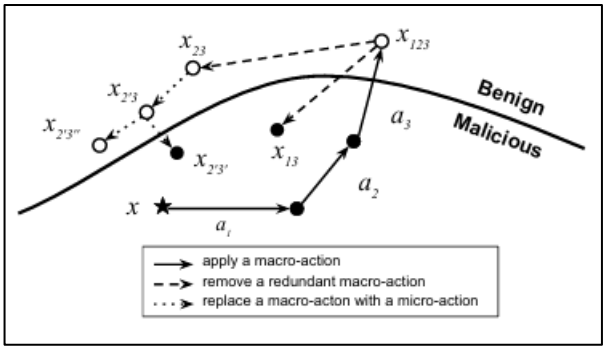


Figure 2: MAB Malware Framework- Action minimization [9]

- **AIMED-RL** employs Distributional Double DQNs (DiDDQN) to minimize the number of behavioral transformations required for evasion. This reduces detection risk while preserving malicious functionality [5][8].

These frameworks highlight the potential of RL to produce malware that evolves in response to detection models and learn in real time how to avoid triggering security systems.

RL Algorithms in evasion (DQN, IRL, DiDDQN)

Among RL algorithms used for malware evasion, Deep Q-Networks (DQN) remain the most widely adopted due to their ability to operate in high-dimensional state spaces. DQNs estimate the action-value function and help the agent select the best possible action at each step [7].

Inverse Reinforcement Learning (IRL) has also been applied in this context. Unlike standard RL where the reward function is defined a priori, IRL infers the reward

structure by observing successful evasion examples. This allows malware to learn from past evasion attempts without explicit programming [6].

DiDDQN extends traditional DQNs by modeling the distribution of possible rewards rather than relying on expected values. This provides a more robust decision-making process under uncertainty, enhancing evasion success rates [8].

Eethical considerations in offensive RL applications

While RL-based malware research provides valuable insights into the limitations of current detection systems, it also introduces important ethical concerns. Offensive RL models could be exploited by malicious actors if publicly accessible or inadequately secured. Therefore, research efforts often include a strong emphasis on **responsible disclosure**, **controlled testing environments**, and parallel work on countermeasures [3][10].

The dual-use nature of these technologies necessitates **guidelines for ethical experimentation** and closer collaboration between academic researchers, cybersecurity professionals, and policymakers. Studies like MERLIN explicitly advocate for simultaneous development of **defensive applications**, using offensive insights to build stronger detection engines [3].

Background Literature - User Behavior Simulation Component

Sandbox evasion and the need for realistic interaction

Sandboxing is an important approach to dynamic malware analysis because it provides a controlled environment to observe malware behaviour without exposing active systems to risk [14]. Sandbox environments can help investigators understand malware interactions with the operating system, networking, and files, while also protecting live systems from exposure [14]. Unfortunately, modern malware increasingly incorporate sandbox evasion techniques which specifically examine artefacts of the sandbox, such as the absence of user input, virtualisation signatures, and inactivity of the system [14][15].

These avoiding measures include delaying execution, inspecting the hardware properties, and looking for debuggers or analysis tools [15]. To work around these measures, sandboxes will need to be realistic simulators of human behaviours, like mouse movement, keyboard input and file manipulation, to be more like actual users [14][15]. If those cues aren't provided, the malware could determine that it is in a virtualised environment, and refuse to carry out its payload [15][16].

Behavioural Indicators and Emulation Strategies

Malware often employs behavioral indicators like mouse movements, keystrokes, application usage, and interaction with the window as part of the process to ascertain whether they are operating in a sandbox environment [16][17]. Even less discernible behavior, such as tab switching, change in focus, or number of times files are accessed can affect malware's opportunity to remain dormant for a period of time and if the malware will either actuate [17][18]. An absence of behavior seen in natural usage significantly leads to lower activation rates of malware during analysis [17][18].

To address this, techniques like environment imitation and dynamic sandbox reconfiguration have been developed to present a more realistic profile to malware [18][19]. These methods dynamically alter system settings and simulate human activity, making detection harder for malware [19][20]. By mimicking genuine system usage, sandboxes can capture more accurate representations of real malware behaviour [19][20].

User behaviour emulation frameworks

User behaviour emulation frameworks are becoming central to modern anti-evasion strategies. These systems aim to replicate realistic user actions to deceive malware and encourage payload execution [14][15]. Frameworks now include mechanisms to simulate various daily activities such as browsing, file editing, and typing to make the sandbox appear more lifelike [14][21].

Studies have shown that such approaches can greatly improve malware detection success rates in sandbox environments [21][22]. Basic emulation using scripted actions can fool less advanced malware but falls short against sophisticated variants that detect fixed patterns [21]. Advanced solutions like UBER generate dynamic user profiles using real interaction data, probabilistic scheduling, and randomised event execution [21][22].

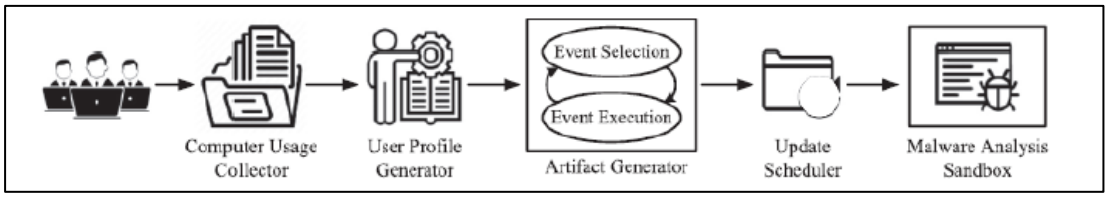


Figure 3: UBER Model Overview

These frameworks also separate behaviour generation into layered components such as event selection, configuration-based randomness, and action execution. This

modularity increases realism and unpredictability, which are essential for bypassing malware evasion mechanisms [21][22].

Role of AI and GANs in behavior simulation

Large Language Models (LLMs) have recently been applied to simulate high-level user interactions, such as email communication or web browsing [23]. These models can understand context and generate plausible user actions in written form, enriching behavioural realism from a narrative or task-level perspective [23]. While LLMs are effective in generating contextual behaviour, they do not yet emulate low-level actions like mouse movements or typing patterns [23].

In contrast, Generative Adversarial Networks (GANs) offer strong potential for producing synthetic data that mimics real user behaviour [24]. GANs are meant to learn the distributions of data and produce new instances that resemble data collected through training. For this reason, they are a good candidate for producing unique user activity profiles [24][25]. Although GANs have recently been used in detection and classification of malware, generating realistic and high-variance data is their most suitable aspect for simulating user profiles in sandboxes [24][25].

Once again, the flexibility GANs develop means that in the sandbox, the behaviours generated would vary, resulting in each sandbox execution looking differently and making it harder for malware to cross reference user behaviour to remain consistent in its objective of reconnaissance through sandbox detection [24]. This aligns with developing a sandbox defence system that is more adaptive and resilient [25].

Background Literature - Hybrid Detection System for Sandbox Evasion Techniques

Limitations of traditional detection approaches

Signature-based and heuristic-based systems are traditional detection methods that have been the foundation of malware detection for many years. While these methods work well for detecting known threats, they often fail to identify malware that can change its code or change its execution behavior to evade detection. This includes polymorphic, metamorphic, and sandbox-evasive malware, designed to get past traditional detection techniques [2][30].

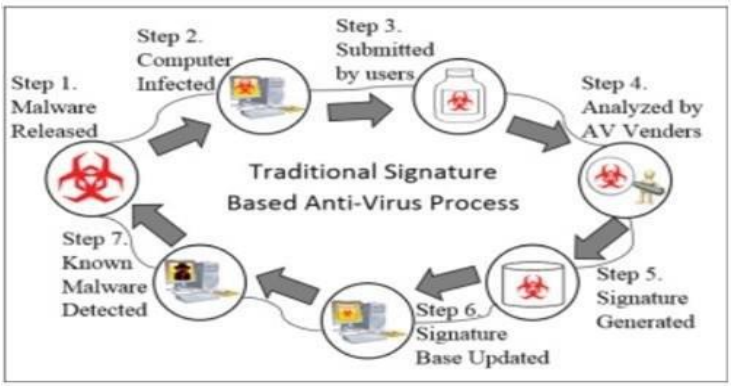


Figure 4: process of Traditional signature based malware Detection

Signature-based detection is particularly unhelpful with zero-day malware, which has not had a signature or pattern created to identify it [36]. While heuristic and rule-based methods can also be used to attempt to identify anomalous behavior, they also suffer from false positive rates that are too high and can also not adapt quickly enough in the fluid nature of modern cyber landscapes [36][42].

Machine Learning in Malware Detection

In response to the shortcomings of conventional techniques, researchers have utilized ML techniques as well as behavioral analysis. ML models can learn from extensive datasets of malware behavior to identify suspicious patterns based on the order of API calls, network activity, and file system changes, etc. [47][48]. Ensemble learning, which combines results from multiple classifiers, has been shown to be highly

effective for minimizing false positives and improving detection accuracy. AdaBoost and Random Forests are examples of approaches in this family [37].

Moreover, hybrid detection frameworks which utilize graph-based analysis are gaining popularity. They take advantage of information derived from system calls and relationships between processes, and furthermore, combine these features with advanced machine learning techniques like support vector machines (SVMs) and graph kernels. This hybrid system enhances detection capabilities for obfuscated malware since these threats specifically alter static features within the present identifier to obfuscate its feature set, or evade detection entirely [38].

LSTM-based multi-label evasion detection

Expanding on the advancements in ML-based detection, this study introduces a hybrid detection system that integrates rule-based techniques with Long Short-Term Memory (LSTM) networks to identify sandbox evasion strategies. LSTMs, a type of recurrent neural network (RNN), are particularly well-suited for analyzing sequential data like API call traces generated during malware execution. This ability to model temporal dependencies in data makes LSTMs ideal for capturing the dynamic patterns of malware behavior during execution [29][46].

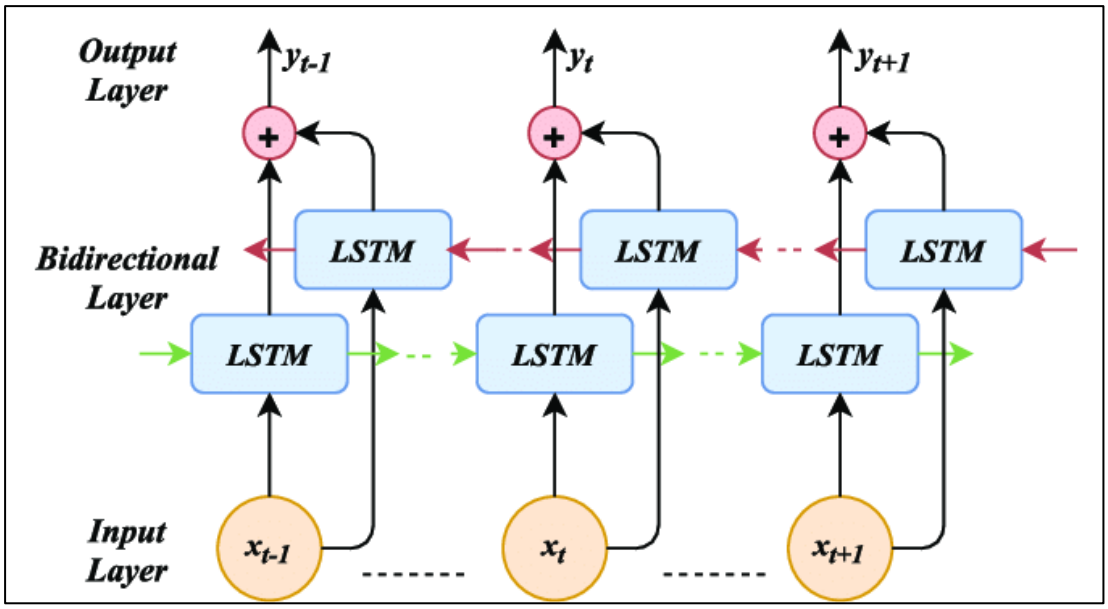


Figure 5: Architecture of a Bidirectional LSTMs

The model takes sequences of API calls obtained from the sandbox logs, and identifies potential evasion tactics based on learned patterns. Evasion tactics can be identified by environment checks, anti-debugging measures, or delay loops. The model has a multi-label classification system that is capable of identifying multiple evasion tactics within a single malware execution trace.

The detection pipeline follows these steps:

- **Data preprocessing:** Tokenization of API sequences, followed by padding to standardize input lengths.
- **Balancing techniques:** Synthetic data generation and oversampling to address class imbalance and ensure minority evasion types are adequately represented.
- **Bidirectional LSTM layers:** Capturing both forward and backward dependencies in the API sequences for more comprehensive pattern recognition.
- **Multi-label classification:** Enabling the detection of multiple concurrent evasion strategies within a single sample.

The training process employs established ML techniques, specifically the Adam optimizer, early stopping, and binary cross-entropy loss. Performance is measured using standard metrics of accuracy, F1-score, precision, and confusion matrices on both balanced and imbalanced datasets [46][47][48].

Training Challenges and Behavioral Analysis

Despite their strengths, LSTM networks present several challenges during training:

- **Class imbalance:** To address this, synthetic data generation and oversampling techniques were applied to ensure that less frequent evasion types were adequately represented in the model's learning process.
- **Sequence variability:** The diverse nature of malware families led to significant variability in API call sequences, necessitating careful feature standardization to improve model consistency.
- **Overfitting:** To mitigate overfitting, dropout layers and early stopping strategies were employed, which helped to prevent the model from becoming too specialized on the training data.

After being trained, the model was combined with sandbox behavior logs to assess unseen examples. The signals produced predicted potential evasion techniques and the confidence scores accompanying those predictions gave the analyst generalized predictive information about the associated likelihood of evasion and malware's sandbox-aware behavior.

This hybrid detection system enhances traditional sandbox-based approaches by enabling real-time identification of evasive malware, even when the malware attempts to hide or suppress its activities during analysis.

Background Litreature - RL agent to dynamically modify sandboxes

Malware detection traditionally involves static and dynamic analysis. Static analysis inspects code without execution, effective for known patterns but vulnerable to evasion tactics like obfuscation. Dynamic analysis runs malware in a sandbox to observe behavior, useful for detecting unknown threats.

Cuckoo Sandbox is a widely used open-source dynamic analysis tool that executes malware in virtual machines and generates detailed behavioral reports. Despite its capabilities, malware authors develop evasion techniques that detect sandbox environments, leading to incomplete or misleading results.

To address this, machine learning (ML) and particularly reinforcement learning (RL) have been explored. RL differs from traditional ML by enabling agents to learn optimal actions through trial and error, using reward-based feedback. This adaptive learning helps agents handle changing, unpredictable environments without requiring exhaustive pre-programmed responses.

RL shows promise in cybersecurity, including phishing detection, wireless spoofing defense, and now malware analysis. By simulating malware and observing its interaction with sandboxes, RL agents can reveal sandbox vulnerabilities and mimic real-world evasion tactics [52].

Traditional ML models rely on labelled historical data, making them reactive and less effective against zero-day threats. In contrast, RL's exploratory nature supports proactive analysis, making it suitable for building intelligent, self-improving malware detection systems [55][56][57].

Combining RL with sandboxing enables dynamic optimization of monitoring features and threat prioritization. It facilitates adaptive, real-time detection systems that evolve with the threat landscape. Furthermore, RL can be used to simulate adversarial

behaviors, helping developers build more resilient and robust cybersecurity defenses against evolving malware strategies.

1.2. Research Gap

1.1.1 Research gap related to Offensive RL Component

The comparative study shown by the research gap matrix exposes numerous important shortcomings in current publications on reinforcement learning-based malware escape. Most previous experiments, including well-known ones including MERLIN, AIMED-RL, and both variants of MAB-Malware, strongly stress evasion against commercial antivirus (AV) systems or static machine learning models. They fail, though, when compared to real dynamic sandbox settings such as Cuckoo Sandbox, which offer a more realistic and developing threat detection system. This dearth of sandbox-oriented testing results in a notable disparity in performance of evasion techniques versus behavior-based detection systems.

Furthermore, the table amply illustrates how many models overlook include a thorough manual malware development component. Although detection bypass methods are under evaluation, they are usually used on synthetic or pre-existing malware instead than custom-developed samples that replicate hostile environments. This reduces the usefulness of the results practically. Furthermore suggesting many approaches are theoretical or offline simulations rather than active escape systems functioning in live, realistic settings is the lack of actual contact between RL agents and sandbox environments. Although adversarial ML techniques are often detached from RL-based dynamic learning systems, limiting the model's responsiveness to changing defenses, they are rather commonly utilized.

Research Paper	Detection ML Models	Manual Malware Development	Check against Commercialized AV	Check against Cuckoo or other sandboxes	Sandbox Evasion techniques	RL Agent Integration	Adversarial ML Techniques
MERLIN - Malware Evasion with Reinforcement Learning (2022)	✓	✗	✗	✗	✓	✓	✓
An IRL-based malware adversarial generation method (2020)	✓	✗	✗	✗	✗	✓	✓
Developing Stealth and Evasive Malware Without Obfuscation (2021)	✓	✓	✓	✗	✓	✓	✓
AIMED-RL	✓	✗	✗	✗	✗	✓	✓
MAB-Malware: A Reinforcement Learning Framework for Attacking Static Malware Classifiers (2020)	✓	✗	✓	✗	✗	✓	✓
MAB-Malware: A Reinforcement Learning Framework for Blackbox Generation of Adversarial Malware (2021)	✓	✗	✓	✗	✗	✓	✓

Evasive Malware via Identifier Implanting	✗	✗	✗	✗	✓	✗	✗
Fang et al. - Evading Anti-Malware Engines with Deep Reinforcement Learning (2019)	✓	✗	✗	✗	✗	✓	✓
Evading malware classifiers using RL agent with action (2020)	✓	✗	✗	✗	✗	✓	✓
Proposed Approach	✓	✓	✓	✓	✓	✓	✓

By tackling all these gaps, the suggested method shines out instead. It uses a totally integrated approach whereby a reinforcement learning agent dynamically interacts with Cuckoo Sandbox via a whole malware modification and deployment path. It sets operating restrictions simulating real-world malware limitations and blends manual C-based malware programming with modular evasion tactics. It questions sandbox analysis in real time by adding state updates depending on actual sandbox feedback and using behavior-driven evasion tactics, especially human behavior simulation. Apart from beyond the limits of present research, this all-encompassing and extremely useful design presents possibilities for more realistic adversarial training systems for sandbox hardening.

1.1.2 Research Gap related to User Behavior Simulation Component

Research Paper	User Behaviour Simulation	Dynamic Data Utilization	Generative Model Usage	Realism of Simulated Behaviour
Combating Sandbox Evasion via User Behaviour Emulators	✔	✘	✘	✘
Enhancing malware analysis sandboxes with emulated user behaviour	✔	✘	✘	✔
Investigating Anti-Evasion Malware Triggers Using Automated Sandbox Reconfiguration Techniques	✔	✘	✘	✘
MalwareEnvFaker A Method to Reinforce Linux Sandbox Based on Tracer Filter and Emulator against Environmental-Sensitive Malware	✔	✘	✘	✘
User Behaviour Simulation with Large Language Model	✔	✘	✔	✔
Proposed Approach	✔	✔	✔	✔

Although sandboxing is commonly used in dynamic malware analysis, many of the modern variants of malware are equipped with sandbox evasion techniques that can recognize artificial environments and suppress malicious behaviour. An important shortcoming of current sandbox solutions is that there is no realistic user behaviour, which assists malware in identifying the environment as noninteractive and non-execution.While previous work includes examples of scripted or rule-based user activity, they are often not diverse, random, or statistically realistic. A review of the current literature shows that the use of generative models (e.g., GANs) to simulate complex and dynamic user behaviour in a malware sandbox is nearly absent.This work proposes to tackle this problem by presenting a generative, behaviour-oriented simulation framework, intended to simulate diverse user profiles, while facilitating the

injection of human-like interactions into existing malware sandboxes. The aim of introducing generative user simulation into a malware sandbox is to produce user behaviour that is significant enough to trigger evasive malware to expose its payload by convincing the malware that it is interacting with a real, interactive system.

1.1.3 Research gap related to Hybrid Detection System for Sandbox Evasion Techniques Component

Sandbox environments are a critical aspect of malware analysis, for they enable the observation of the malware in a controlled environment while monitoring behaviors such as changes to files, system calls, and network traffic. However, while sandboxes have captured these actions, they are unable to differentiate between routine behaviors and tactics specifically intended to evade analysis. Sandbox evasion tactics, which are environmental checks, timing attacks, and process name checks, further obscure the analysis of the malware [22]. As for example a malware sample may exhibit delayed behavior or altered behavior based on the detection that it is being executed in a virtual machine or sandbox. Therefore, the sandbox report would indicate that the malware was benign, even if it was designed to evade detection. Because sandbox reports do not provide clear evidence of malware evasion, analysts are tasked with manually sorting through large amounts of logs to find signs of potential evasion. This process is time-consuming, complicated, and prone to human error. As malware advances and becomes more evasive and complex, the significance of accurately and automatically being able to identify evasion will continue to become a greater issue in cybersecurity [23].

The inefficiency of the ability to detect evasion methods in sandbox reports is not only a problematic issue of detection capability but may also be a broader defect of the malware analysis method itself, particularly in the lack of comparative detection systems with existing malware analysis frameworks. Comparative detection compares the behavior of a specimen of malware to a known dataset of malicious behavior. This detection and analysis technique has not been heavily incorporated into traditional sandbox environments [24]. As a result, malware analysis does not often extend to analyzing malware in isolation from other malware behavioral traits or patterns. The limited ability to detect malware alone lessens existing malware identification methods. If a specimen of malware performs in a fashion similar to the evasive malware specimen, a detection method is available to automate and compare the behavioral patterns of both behavior samples. The accuracy and consistency of

detection would significantly improve by identifying malware samples under observation or analysis if the behavior outputs of the samples were conducted similarly [25]. However, most available detection systems fail to use machine learning, or similar technique ungendered to detect behavior comparisons automatically and recognize similarities in evasion among malware samples based on behavioral sequences.

Also In addition to the use of machine learning models in conjunction with existing malware detection workflows, in particular Long Short-Term Memory (LSTM) networks, remains neglected. While LSTM networks have seen some success in analysis of sequential data, there remains little work applying them directly to sandbox evasion detection. LSTM networks have a unique advantage at modeling time series data, including sequences of API calls made during the malware execution in the sandbox. When given sequences of actions taken by malware, LSTM models can learn long-term dependencies in the data and identify trends associated with evasion strategies, which suggests a fully LSTM model should be able to detect low interaction evasion as well as capture more complex evasion strategies involving API calls. Despite this potential, there is little work in the literature using LSTM models as part of the sandbox analysis process, in particular with the detection of evasion methods or policies. Current practice uses static features or agnostic heuristic rules to collect data in contained environments but does not consider that the evolution of evasion methods in malware execution and the inherent and dynamic nature of automation and evasion during sandbox use. This need for a hybrid model that combines both work being accomplished in dynamic analyses from a sandbox environment with the pattern recognition excellence of LSTM networks is clear, allowing for detection of malware while categorizing its evasion strategies as well [4].

Research	*Analyze & Detect Malware evasion techniques*	*Integrate with a sandbox for analyze*	*Comparative detection*
M. Ficco, "Malware Analysis by Combining Multiple Detectors and Observation Windows," June 2022	✓	✓	✗
Comparative Study of Detection and Analysis of Different Malware with the Help of Different Algorithm,2023	✓	✗	✓
On the Effectiveness of Perturbations in Generating Evasive Malware Variants, 2023	✗	✗	✗
A. K. Sinha and S. Sai, "Integrated Malware Analysis Sandbox for Static and Dynamic Analysis," 2023	✓	✓	✗
X. Jianhua, S. Jing, Z. Yongjing, L. Wei and Z. Yuning, "Research on Malware Variant Detection Method Based on Deep Neural Network," 2021	✓	✗	✗
A. V et al., "Malware Detection using Dynamic Analysis," 2023 International Conference on Advances in Intelligent Computing and Applications (AICAPS), Kochi, India, 2023,	✓	✗	✓
Proposed approach	✓	✓	✓

Figure 4 - Research Gap Analysis

1.1.4 Research Gap related to Reinforcement Learning-Based Dynamic Sandbox Modification for Malware Behavior Analysis

Although various studies have investigated machine learning and reinforcement learning used for malware detection and evasion, a significant gap in the literature is the investigation of runtime sandbox environment adaptation. Quertier et al. and Gibert et al. use reinforcement learning with evasion strategies, however, their methods are offensive and lack run-time sandbox analysis and adaptation. Similarly, other studies, including Bokolo et al., Mail et al., and Jaswinder et al. evaluated static or behavioral detection strategies and implemented machine learning based models, but did not involve RL or sandbox adaptation.

Additionally, although Li et al. and Arif et al. use reinforcement learning or deep learning methods, they do not explore integration with sandboxing environments like Cuckoo Sandbox. While they explore adversary behavior simulation and evasive malware feature generation, they do not simulate real-time or environment-adaptive techniques. Likewise, Chtazoglou et al. proposed several evasive approaches but did not provide a framework for modifying sandbox environments to investigate how these evasive techniques can be detected proactively for environments in a sandbox.

The present research directly addresses this hole and provides a framework for using reinforcement learning to modify sandbox configurations based on malware behavior. The research is the only known effort to integrate real-time sandbox adaptation, anti-evasion measures, and reinforcement learning algorithms to classify malware behavior. It represents an important step in addressing an important gap in the malware analysis segment of security research and provides starting points for future support of intelligent, self-adjusting analysis environments. This also illustrates the distinct contribution of the proposed research and a significant need to invent advanced proactive defenses against various kinds of malware.

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	**✓**	**✓**	**✓**	**✓**	**✓**

1.3. Research Problem

1.3.1 Research Problem Explanation

The underlying issue of this study is detection of malware that evades running in sandbox environments. Traditional sandboxing detects malware running inside a static, virtual sandbox in a passive way. But today's malware employs sophisticated sandbox evasions such as delaying, looking for evidence of virtualization, or waiting for signs of human interaction that hide their true nature. Vulnerabilities result in partial or misleading examination, with advanced threats going undetected.

In proactive detection, this project proposes an adaptive sandbox system that acts ahead of malware. By integrating reinforcement learning (RL) to guide environment adaptation with machine learning (e.g., LSTM) for action classification, the sandbox adapts its analytic techniques based on perceived actions. Further, by simulating human-like behavior (e.g., via GANs or behavior profiles), the system deceives environment-aware malware. With proactive interaction, malware is forced to reveal itself, allowing for earlier and more precise detection of evasive behaviors—a major advance in dynamic malware analysis.

1.4. Research Objectives

The main goal of this study is to design and evaluate an adaptive sandbox system that is capable of effectively detecting, analyzing, and responding to evasive malware in a sandbox. It achieves this by exploring offensive and defensive methods, integrating reinforcement learning (RL), user behavior modeling, and machine learning-based detection to design an adaptive intelligent sandbox system.

Offensive objectives

- To design syntactic-based malware that can evolve in real time to evade dynamic sandbox environments such as Cuckoo Sandbox.
- To develop and experiment with RL algorithms (such as DQN, DiDDQN, IRL) that learn optimal evasive action sequences.
- Simulate adversary environments with sandbox feedback loops and examine malware adaptation to become stealthier.
- In order to assess how effectively existing RL-based systems (for instance, MERLIN, MAB-Malware, AIMED-RL) perform and identify their limitation in dynamic, real-world environments.

Defense Targets

- Attempting to identify sandbox environments by their behavioral patterns and evasive techniques such as lack of human interaction, virtual system fingerprints, and system call irregularities.
- Applying emulation of human-like behavior (i.e., based on principles of UBER approach) to counter environment-aware malware.
- In this paper, we explore how to use probabilistic and generative models (i.e., GANs, LLMs) to make dynamic, human-like, believable interactions possible in the sandbox.

- To make sandbox environments realistic, human behavioral patterns like mouse movement, typing, window switching, and browsing behavior are simulated.

Hybrid Detection Goals

- With an aim to create a hybrid detection model that includes rule-based heuristics and LSTM-based sequence classification to identify sandbox evasion methods.
- Preparing to train the model on sequences of API calls from sandbox logs with multi-label classification of different evasion behaviors.
- Comparing with accuracy, precision, recall, and F1 score on synthetic malware sets, synthetic malware sets, and real-world malware sets.
- Trying to combine detection model with dynamic sandboxing in order to enable real-time prediction and classification of evasive behavior when executing malware.

Adaptive Sandbox Objectives

- To propose an adaptive sandbox architecture that uses reinforcement learning to adapt to change based on feedback from malware behavior analysis.
- In order to train RL agents to adapt sandbox settings—i.e., system state, simulator of users, and observation depth—to induce latent malware activity.
- In order to compare different RL algorithms (DQN, SARSA, TRPO, PPO, SVPG) against one another based on convergence speed, stability, and maximum reward in sandbox environments.
- To illustrate how adaptive sandboxes close the functional gap between dynamic analysis and passive monitoring to enable detection of advanced, evasive, and zero-day malware.

2. METHODOLOGY

2.1 Methodology Overall

2.1.1 Methodology Framework – Offensive RL Component

High-level overview

This study employs a unique malware evasion strategy using reinforcement learning (RL) techniques. The approach combines traditional malware development and sophisticated machine learning techniques to automatically find the best combination of evasion techniques to circumvent today's sandbox detection technologies, specifically using Cuckoo Sandbox. Rather than depending solely on evasion strategies that have been tried previously or using predetermined "tried and true" methods, this approach uses the learning ability of reinforcement learning to explore evasion techniques in a systematic way and find the most effective combinations. This represents a shift from using traditional trial-and-error strategies to assessing and creating an intelligent system that is automated to discover non-obvious evasion techniques as it learns and improves.

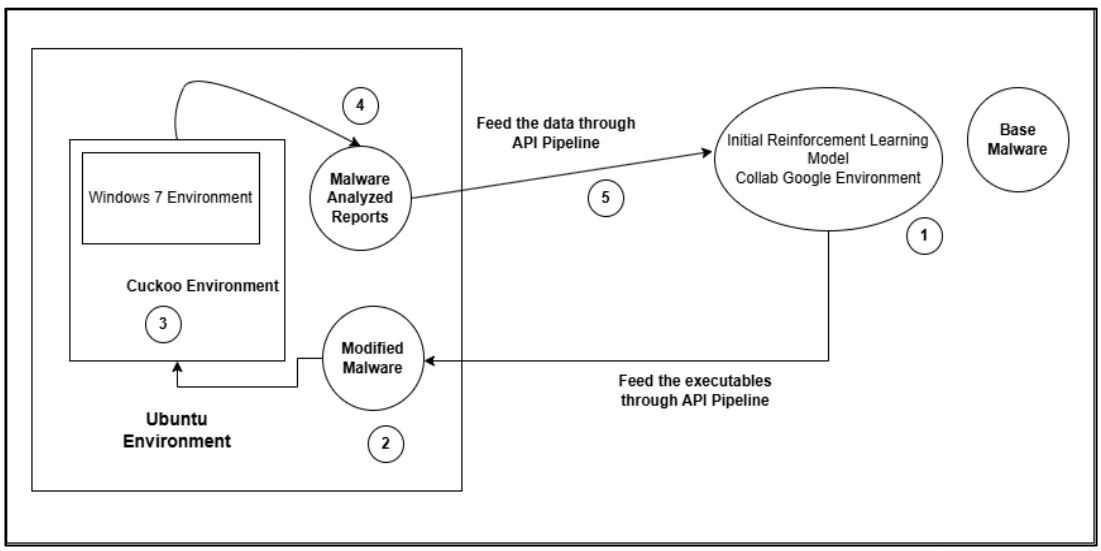


Figure 6: High-Level Infrastructure Overview

Following are the high-level steps of the whole process,

1. **Initial Reinforcement Learning Model Setup (Google Colab) -** The process begins in the Colab environment, where the base malware is paired with the initial RL model. This model is responsible for selecting and applying evasion techniques to generate modified variants of the malware. The agent uses a DQN-based approach to make decisions and adjust actions over training episodes.

2. **Modified Malware Generation** - Once the RL agent decides which evasion actions to take, the base malware is transformed by injecting the selected evasion functions. This results in a modified malware sample, which is saved and prepped for analysis.

3. **Cuckoo Sandbox Execution -** The modified malware is sent from the Colab agent to the Cuckoo Sandbox, hosted in a Windows 7 virtual environment inside an Ubuntu machine. The sandbox executes the sample and monitors its behavior in detail.

4. **Analysis Report Generation** - After execution, the sandbox generates a detailed behavioral analysis report, including indicators, scoring, and detected malicious activity. This report reflects how effectively malware evaded detection.

5. **Feedback Loop via API** - The report is then fed back to the RL agent through an API pipeline. The detection score extracted from the report is used as a reward signal, guiding the agent in its next action selection and enabling it to improve over time.

Parallel Processes in Methodology

The research methodology consists of two parallel.

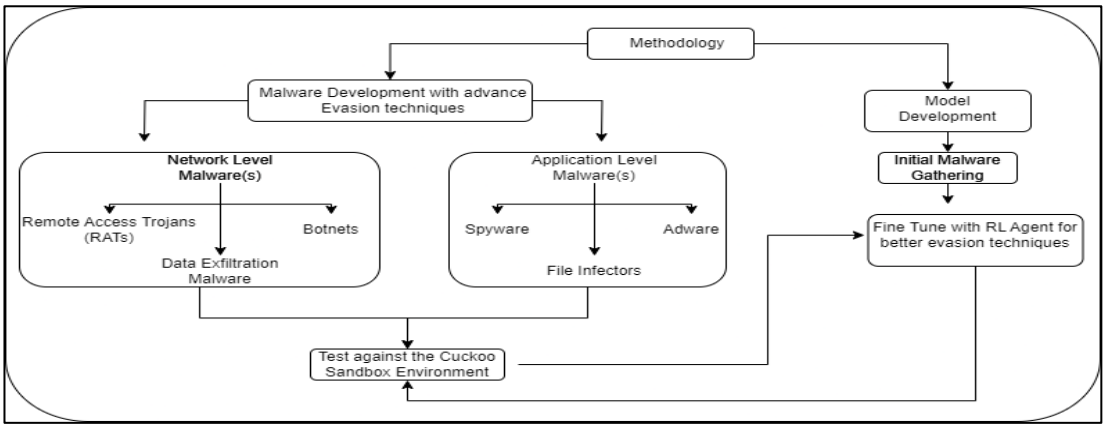


Figure 7: Parallel Processes

- **Manual Malware Development**:

This includes designing a base malware sample in C that exhibits common suspicious behaviors such as network connections, registry modifications, file operations, and process injections.

It also encompasses the implementation of a diverse set of evasion techniques across three categories

  - timing-based evasions
  - filesystem-based evasions
  - human behavior simulations.

These techniques are executed within a modular framework that allows for selective activation and combination. With the modular approach, the reinforcement learning agent can dynamically construct evasion strategies by simply plugging in selected functions into the base malware, as indicated in the figure. Essentially, the base code is transformed into a variant of itself by selecting evasion functions (e.g., T2, F1, H3) as pluggable modules.

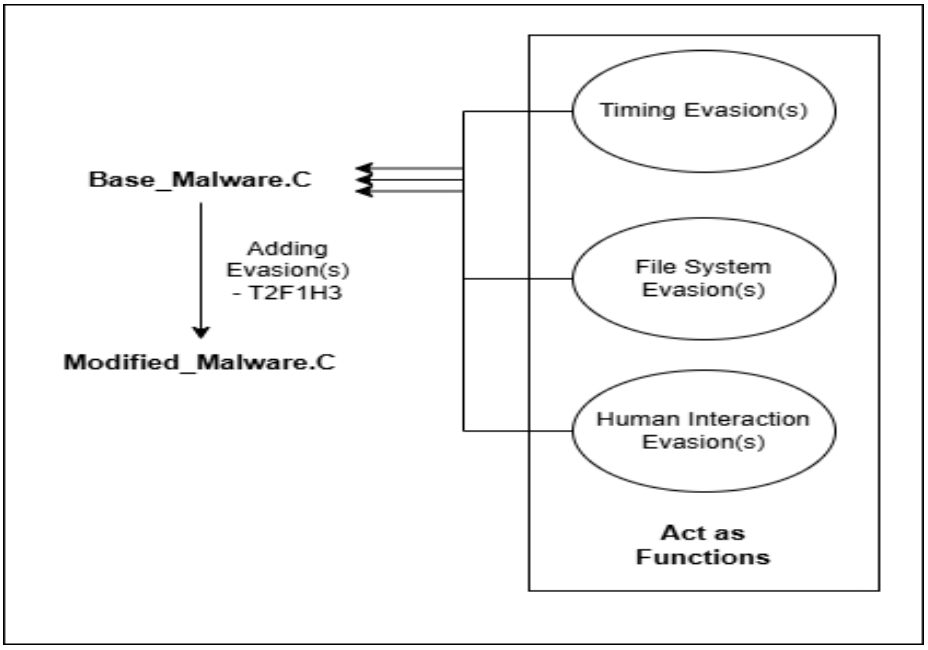


Figure 8: Converting Base Malware

- **Model Development and RL Integration (P2)**

In this study, the reinforcement learning model utilizes a DQN (Deep Q-Network) configuration with a well-defined state representation. Each state is represented as a vector, with the first coordinate being the current Cuckoo detection score, and the remaining indicators being binary (0 or 1) for each applicable evasion technique indicating whether it was deployed to the malware. This compact yet comprehensive representation allows the neural network, with a corresponding input layer to the state size, two hidden layers with 24 neurons per layer, and an output layer that corresponds to the actions taken, to learn which evasion techniques and method combinations were the most effective at reducing the detection scores. The model refines its evasion strategy in Cuckoo Sandbox through multiple training cycles with malware modifications, Cuckoo Sandbox submissions, and monitoring the changes to detection scores. The model will progressively prioritize evasive techniques that provided the greatest rewards (reduced detection scores).

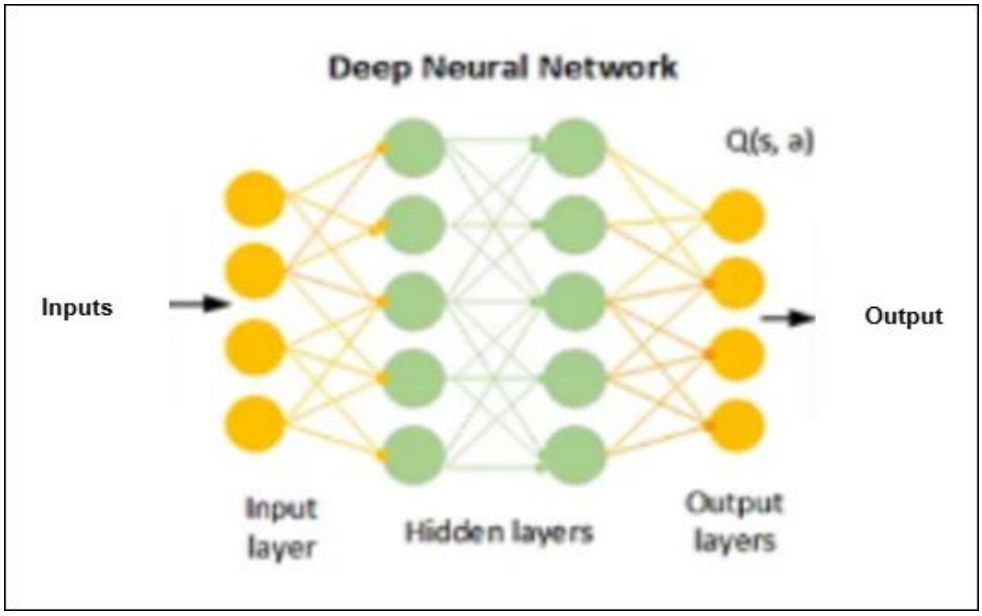


Figure 9: Exampled Deep Neural Network

All of the two above processes involve the design of the environmental interface enabling communication between the RL agent and Cuckoo Sandbox, the definition of valuable state and action space, the creation of a rewarding function based on detection score reduction, and the implementation of a Deep Q-Network agent that can learn optimal evasion strategies. These parallel processes are unified into the API-based workflow, in which the RL agent modifies malware samples and obtains feedback from Cuckoo Sandbox analysis.

Reinforcement learning framework for malware evasion

This framework views the task as a sequential decision process in which the agent alters a malware sample by applying one of a number of evasion techniques and receives feedback based on the detection scores. The agent performs action selection optimization with Q-values that are estimated using a neural approximation and learns via multiple episodes through trial and error.

- **Initial State Assessment:**

The initial stage involves setting a benchmark for the learning mechanism, through an analysis of the non-evasive malware. This process typically nets an approximate Cuckoo detection score of 5.8, in accordance with the sample's alien features before the provisions for evasion were executed. The state vector is then initialized with the Cuckoo detection score formulated as the first digit and the binary flags for each of the 11 evasion techniques set to 0 to signify that none have been administered. Thus, a sterile and regulated start starting point was created to train the reinforcement learning agent. Along the way, environmental constraints were established—for instance, only a single timing-based evasion is allowed, 3 filesystem-based evasions and 2 human-behavior-based evasions are permitted at any time. Forces placed on training environment established the training is setup to initially favor exploration by setting the epsilon (ε) value at 1.0, which maximized the opportunities for the agent to randomly-sample actions and learn more in the early stages of training.

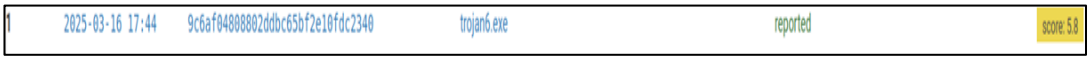
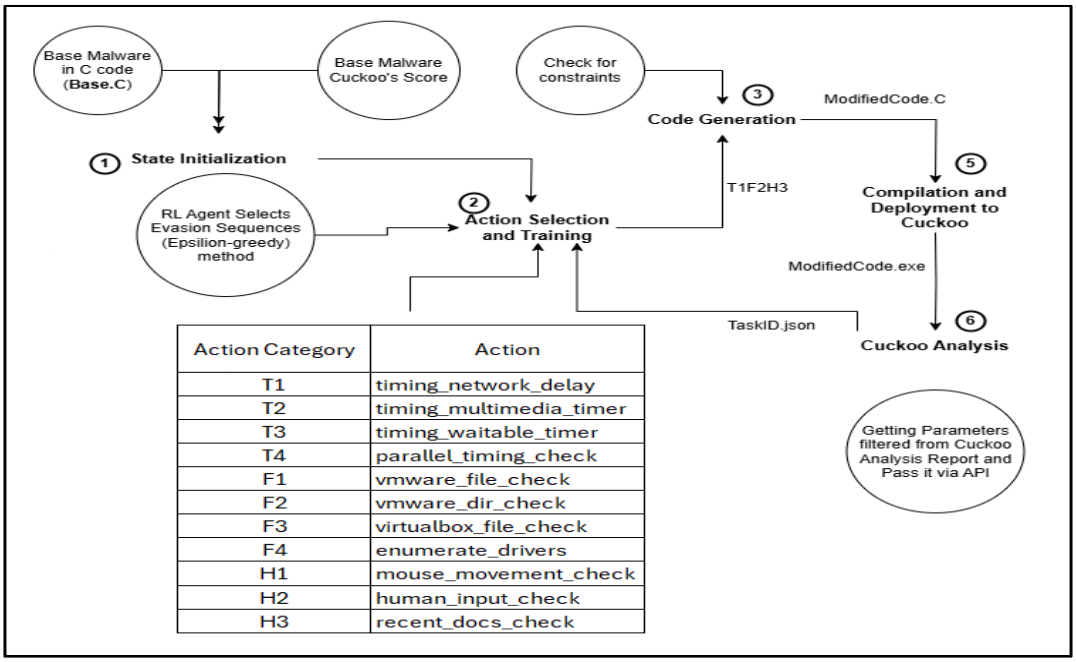


Figure 10: Base Malware Cuckoo Score

- **Action Selection and State Transitions:**

The RL agent functions in a discrete action space encompassing 11 evasion techniques. In every training step, the agent executes an action corresponding to a specific evasion technique, which is determined by the agent's policy. Once the action is executed, the system modifies the malware's source to reflect the selected technique, manages the dependencies and code organization, and builds the unique new sample file with MinGW. The new executable is passed on to the Cuckoo Sandbox Framework for behavioral analysis. After the executable has been run, the sandbox returns a new detection score, which is extracted and utilized in calculating a reward and updating the current policy for the agent. The environment has rigid constraints, e.g., only one timing evasion, three filesystem evasions, and two human interaction evasions maximum. Each transition is

analyzed to ensure the malware is still functional and can even be compiled again, which preserves the breakdown of the build process and avoids noise or invalid learning signals to return to the training process.



Action Category	Action
T1	timing_network_delay
T2	timing_multimedia_timer
T3	timing_waitable_timer
T4	parallel_timing_check
F1	vmware_file_check
F2	vmware_dir_check
F3	virtualbox_file_check
F4	enumerate_drivers
H1	mouse_movement_check
H2	human_input_check
H3	recent_docs_check

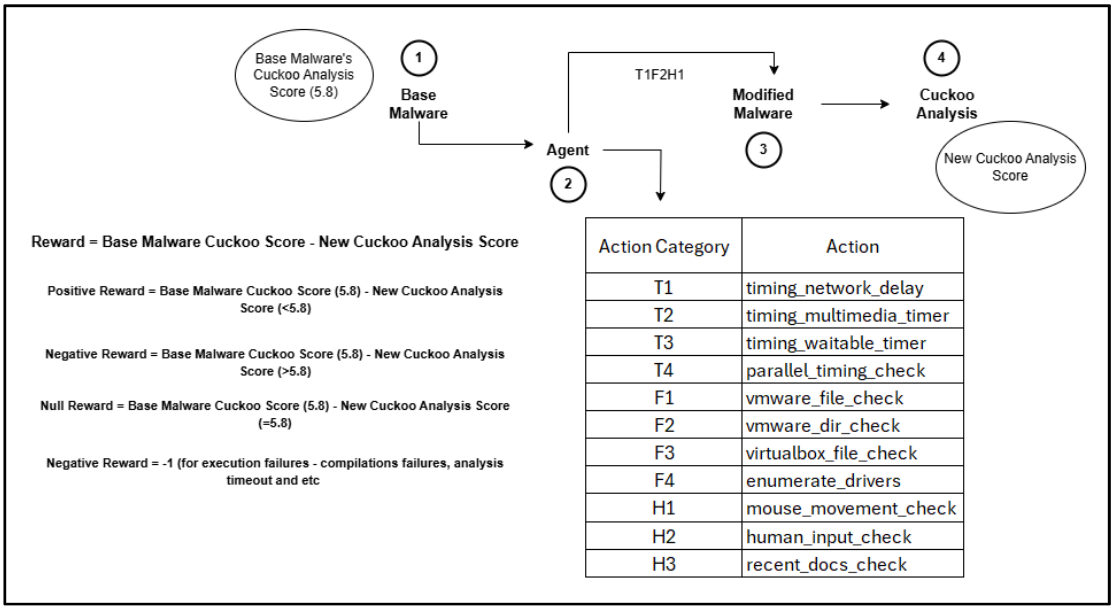
Figure 11: RL Model Workflow Setup

The RL agent functions in a discrete action space characterized by eleven different evasion techniques. In each step of training, the agent will pick an action, which corresponds to an evasion technique based on its current policy. After the action is selected by the agent, the environment adjusts the malware's source code, encapsulating the selected evasion technique, along with handling any necessary dependencies, managing the code, and building the sample with MinGW. The resulting executable is put slice of into a Cuckoo Sandbox to study the malware’s behavior. Once the executable has run through the sandbox, the environment receives a fresh new detection score from the sandbox, which the environment extracts to calculate the reward and filters the new detection score into the agent's policy update. The environment consists of strict constraints: 1 timing evasion, 3 filesystem evasions, and 2 human interaction evasions maximum. Each gained transition is validated to confirm that the resulted malware is functional, compilable, preserving the integrity of the

training cycle, and eliminating noisy or irrelevant observations from the learning signal.

- **Reward Assignment Mechanism:**

The reward mechanism serves as the main feedback loop for the reinforcement learning agent, allowing it to assess the success of the selected evasion actions. The value of the reward is given by the difference in detection scores of the original malware and the evaded version after the evasion actions have been applied.



Action Category	Action
T1	timing_network_delay
T2	timing_multimedia_timer
T3	timing_waitable_timer
T4	parallel_timing_check
F1	vmware_file_check
F2	vmware_dir_check
F3	virtualbox_file_check
F4	enumerate_drivers
H1	mouse_movement_check
H2	human_input_check
H3	recent_docs_check

Figure 12: RL Model's Action table and Reward Assignment

The base malware usually has an initial high detection score as a result of the analysis done by the Cuckoo Sandbox (5.8). Once the RL agent modifies the malware by executing a set of evasion techniques, a modified version of the malware is submitted for further analysis. The detection score from the modified malware is evaluated against the original malware score to compute a reward.

The reward assignment logic is defined as follows:

- **Positive reward** → If the new detection score is **lower** than the base score

- **Negative reward** → If the new detection score is **higher**
- **Null reward** → If the new score is **equal** to the base score
- **Penalty reward** = –1 → For failures like compilation error, timeout, or sandbox crash

- **Q-Value Estimation and Optimization:**

The agent learns an optimal evasion strategy by estimating Q-values using the same neural network architecture described earlier (refer to Figure 1.7). This network receives the malware's state vector as input and outputs expected rewards for each possible evasion action. As mentioned above, it includes:

- 12 input neurons (1 detection score + 11 binary flags),
- Two hidden layers with 24 neurons each (ReLU activation),
- An output layer with 11 neurons representing Q-values for the available actions.

In order to promote strong learning, mini-batch training is employed alongside an experience replay buffer (size: 10,000) which contains previous transitions that can randomly be sampled. A target network is also employed, which periodically gets synchronized with the main network to stabilize updates and help alleviate drifting value estimates.

```
## Step 2: Define Constants

# RL model parameters
MAX_MEMORY_SIZE = 10000
BATCH_SIZE = 32
GAMMA = 0.95  # Discount factor
EPSILON = 1.0  # Exploration rate
EPSILON_MIN = 0.01
EPSILON_DECAY = 0.995
LEARNING_RATE = 0.001
```

Figure 13: Defined RL Model Parameters

- **Evasion Strategy Optimization via Epsilon Decay:**

To balance exploration and exploitation, an epsilon-greedy policy is used. The agent starts training with $\varepsilon = 1.0$ (full exploration) and decays it to $\varepsilon = 0.01$ using a decay factor of 0.995. This enables the model to explore various actions early in training, then shift its focus towards high-reward actions once it develops confidence in the environment.

The discount factor (γ) is established as 0.95, thereby emphasizing the importance of long-term rewards during the training. The goal state is defined by a detection score ≤ 1.0, indicating the malware has circumvented the Cuckoo Sandbox. Transitions (state, action, reward, next state) are added to the replay buffer, and training continues over several episodes until either convergence or repeated evasion success is achieved.

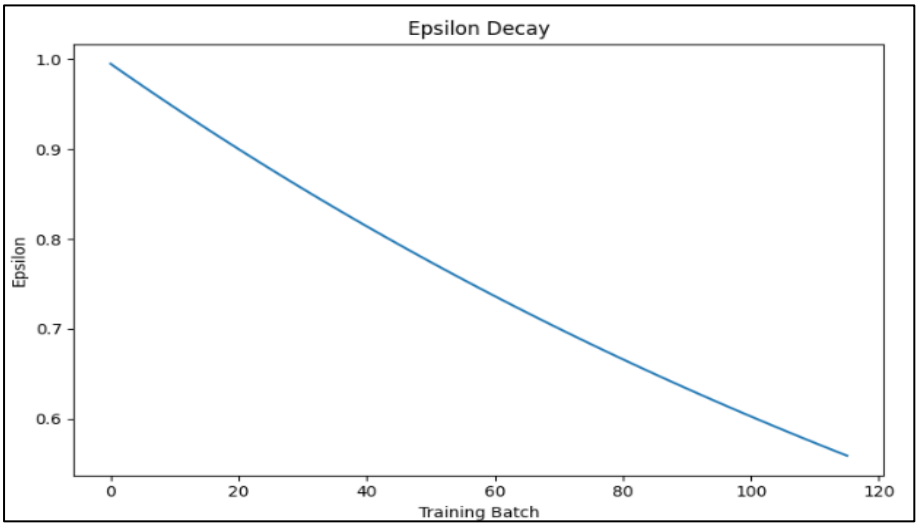


Figure 14: Epsilon Decay

Mathematical Foundation of the RL Framework

This RL-based evasion framework is built upon several mathematical equations that

guide how the agent learns and optimizes its actions. Each equation plays a distinct role in the decision-making and training process,

- **Q-Value Update – Bellman Equation:**

$$Q(s,a)=r+\gamma \cdot a' \max Q(s',a')$$

This equation updates the Q-value for a state-action pair. It combines the immediate reward (r) with the maximum estimated reward of the next state (s') multiplied by the discount factor ($\gamma = 0.95$). It helps the agent consider both short-term and future impact of its actions, encouraging strategies that lead to long-term evasion success.

- **Loss Function for Training:**

$$L=(r+\gamma \cdot a' \max Q(s',a')-Q(s,a))^2$$

This loss function measures how far the predicted Q-value is from the expected Q-value (target). The agent minimizes this loss using gradient descent to improve its predictions. It drives the learning process in the neural network, enabling the agent to make better decisions over time.

- **Epsilon-Greedy Policy:**

$$P(a)=\begin{cases} \epsilon/|A| \text{ (random action)} \\ 1-\epsilon \text{ (greedy action if } a=\arg\max Q(s,a)) \end{cases}$$

Governs the action selection strategy. The agent starts with $\varepsilon = 1.0$ (fully random) and decays it by 0.995 per iteration until it reaches $\varepsilon = 0.01$. This encourages broad exploration in the early training stages and focused exploitation of learned

strategies as training progresses.

- **Reward Function**:

$$\text{Reward} = \text{Base Score} - \text{New Score}$$

The reward is determined by taking the detection score for the variant malware, and subtracting it from the baseline malware score (generally 5.8). A lower final score equates to a successful evasion, which yields a reward. A higher score leads to a negative reward. A small reward or penalty reward if the score is the same or execution fails.

Technological Foundation of the RL Framework

The foundation of the technology in this reinforcement learning framework for evading Cuckoo Sandbox consists of many critical technologies and techniques. Overall, the system uses a Deep Q-Network (DQN) architecture built using TensorFlow and Keras, which is suitable for approximating complex state-action value mappings. The framework incorporates experience replay, storing transitions in a 10,000-size memory buffer, to prevent correlations from sequential experiences and promote stable learning. To create a balance between exploration and exploitation, the framework implements an epsilon-greedy method with an annealed epsilon from 1.0 to 0.01, decayed with a factor of 0.995. The structure of the neural network model consists of two fully-connected hidden layers with 24 neurons, both using ReLU activation functions, and trained with the Adam optimizer at a learning rate of 0.001. For learning stability, the system uses a target network independent of the policy network, which will synchronize for weight updates periodically. To communicate with the Cuckoo Sandbox endpoints, the API communication layer uses the Requests library to submit malware and get the reports back. The malware modification mechanism uses a cross-compiler (MinGW) to develop Windows executables from modified C source code. Finally, throughout the process of training, extended logging

mechanisms and checkpointing allow the resilience of the training progress. Together, these technologies combine to provide a comprehensive framework for autonomously learning how to develop and effectively evade sandboxes.

Evasion technique categories

The system includes a variety of eleven evasions that are split between three complementary categories: timing-based evasions, filesystem-based evasions, and human behavior evasions. Each category focuses on specific characteristics of sandbox environments used to detect and evade analysis. The malware architecture is constructed in a modular manner so that the reinforcement learning agent can dynamically combine evasions during training to generate optimal evasion.

- **Timing-based evasions:**

  These methods are meant to identify artificial time acceleration, which is a frequent method that sandboxes employ to make observing malware behavior one level faster. These techniques purposefully exploit timing errors in order to find these timing manipulations and can ultimately terminate the malware quicker if it identifies a sandbox-like behavior. These methods can range from simple delays caused by the network to more complex timing validation on multi-threaded methods. Each approach provides a slightly different view of the flow of execution and what timing can be validated.

  - **T1: Network Delay** – Creates a deliberate delay using TCP socket connection attempts to unreachable ports. The technique sets up non-blocking socket operations with timeouts to specific IP addresses (Google DNS) and waits for connection attempts to fail, effectively stalling execution while performing legitimate network operations.
  - **T2: Multimedia Timer** – Leverages Windows multimedia timers (timeSetEvent) to create precise delays. This technique uses a different

timing mechanism than standard Sleep functions, making it harder for sandboxes to detect and skip. It registers a callback function that sets a flag when the timer completes, creating a more realistic delay pattern.

- **T3: Waitable Timer** – Uses the Windows waitable timer API to implement delays and detect time manipulation. This technique measures the actual elapsed time against the requested delay time to identify if a sandbox is accelerating time. It can detect "sleep-skipping" techniques commonly used in sandboxes to speed up analysis.
- **T4: Parallel Thread Timing** – Creates multiple threads that each perform timing checks independently. By comparing timing results across threads, it can detect inconsistencies that indicate sandbox time manipulation. This sophisticated approach is particularly effective against sandboxes that don't consistently accelerate time across all execution threads.

- **Filesystem-basedEvasions:**

The goal here is to uncover virtualized environments by identifying the artifacts left by virtualization platforms. The techniques analyze files, folders, and drivers identifiable in sandbox systems like VMware and VirtualBox. Detecting these artifacts provides an extra layer of logic to the malware and an opportunity to exit execution early and avoid being analyzed behaviorally.

- **F1: VMware File Check** – Searches for VMware-specific driver files (vmmouse.sys, vmhgfs.sys) in the Windows system directory. These files are typically present in VMware virtual machines but absent in physical machines, allowing detection of virtualized environments commonly used for sandboxing.
- **F2: VMware Directory Check** – Examines the Program Files directory for VMware installation folders. Uses different methods based on whether the process is running under WOW64, handling both 32-bit and 64-bit Windows environments to detect VMware installations regardless of system architecture.

- **F3: VirtualBox File Check** – Similar to F1, but targets VirtualBox-specific driver files (VBoxMouse.sys, VBoxGuest.sys, VBoxSF.sys, VBoxVideo.sys). These files indicate a VirtualBox virtualized environment, which is commonly used for malware analysis systems.
- **F4: Driver Enumeration** – Performs comprehensive enumeration of all system drivers using Windows API. It searches for virtualization-related keywords (vmware, vbox, virtual, etc.) in driver names, providing broad detection capability for various virtualization platforms including VMware, VirtualBox, Hyper-V, Xen, and QEMU.

- **Human behavior evasions:**

Usually absent in automated sandbox setups, these help to identify the lack of actual user engagement. These methods track cursor movement, keyboard activity, and document history to deduce whether a genuine user is using the system. Should user action not be seen, the virus can decide to stop, therefore avoiding complete behavioral study by the sandbox.

- **H1: Mouse Movement Check** – Captures mouse cursor position at two points in time to detect movement. Genuine user environments typically show mouse movement, while automated sandboxes often lack this interaction. The technique introduces a waiting period between position checks to give a human user time to move the mouse.
- **H2: User Input Check** – Examines the time since the last input event (keyboard, mouse, etc.) using the GetLastInputInfo API. After sleeping for 10 seconds, it checks if any input has occurred. In real user environments, input events are frequent, while sandboxes typically lack user interaction.
- **H3: Recent Documents Check** – Queries the Windows Registry to examine the "RecentDocs" key, which tracks files recently opened by the user. Sandboxes often have minimal or no entries in this registry location. The technique considers an environment suspicious if fewer than 3 recent documents are found.

These evasion categories taken together provide a complete set of capabilities enabling the reinforcement learning agent to intelligibly traverse and control the sandbox detection environment. The ability to mix various methods enables the system to generate malware variants with great stealth potential, able to avoid even sophisticated dynamic analysis tools like Cuckoo Sandbox.

2.1.2 Methodology Framework – User Behavior Simulation Component

Scope of the study

The methodology of this research consists of several key phases, each contributing to the goal of simulating realistic user behaviour in sandbox environments to detect evasive malware. The process is structured into the following main components:

- **Data Preprocessing**: - Datasets accessible to the public, comprising mouse movement and keystroke data, are utilized. The data is subjected to cleansing, normalization, and formatting to guarantee consistency and usefulness.

- **Defining the Schema**: - A structured schema is developed to represent user behaviour profiles. The schema includes key features such as digraphs, trigraphs, and mouse movement characteristics, acting as a blueprint for all generated profiles.

- **Profile Generation using GAN**: - A Generative Adversarial Network (GAN) is trained using the preprocessed data. It generates synthetic user profiles that emulate authentic low-level behaviors exhibited by actual users.

- **Profile Enrichment**: - Four arbitrary user interests are chosen for each profile. These are sent to OpenAI to produce authentic search phrases and URLs, which are incorporated into the profile to enhance realism.

- **Script Generation**: - A scripting engine processes each profile and modifies established behavior templates. It subsequently produces automation scripts to replicate user behavior within the sandbox environment.

- **Timetable Scheduling**: - A timetable engine generates a plausible activity schedule. It designates particular behaviors and allocates them to distinct time intervals to replicate continuous and diverse user actions.

- **Sandbox Integration**: - The completed automation script, encompassing the scheduled actions and revised templates, is executed in Cuckoo Sandbox. This replicates real-time user behavior and incites sandbox-evasive malware to activate.

System Architecture

The system architecture of the proposed solution is designed to simulate realistic user behaviour within a sandbox environment in order to counter sandbox-evasive malware. It is composed of modular and interconnected components that enable the end-to-end generation, management, and simulation of synthetic user activity. Figure 3 illustrates the overall structure of the system.

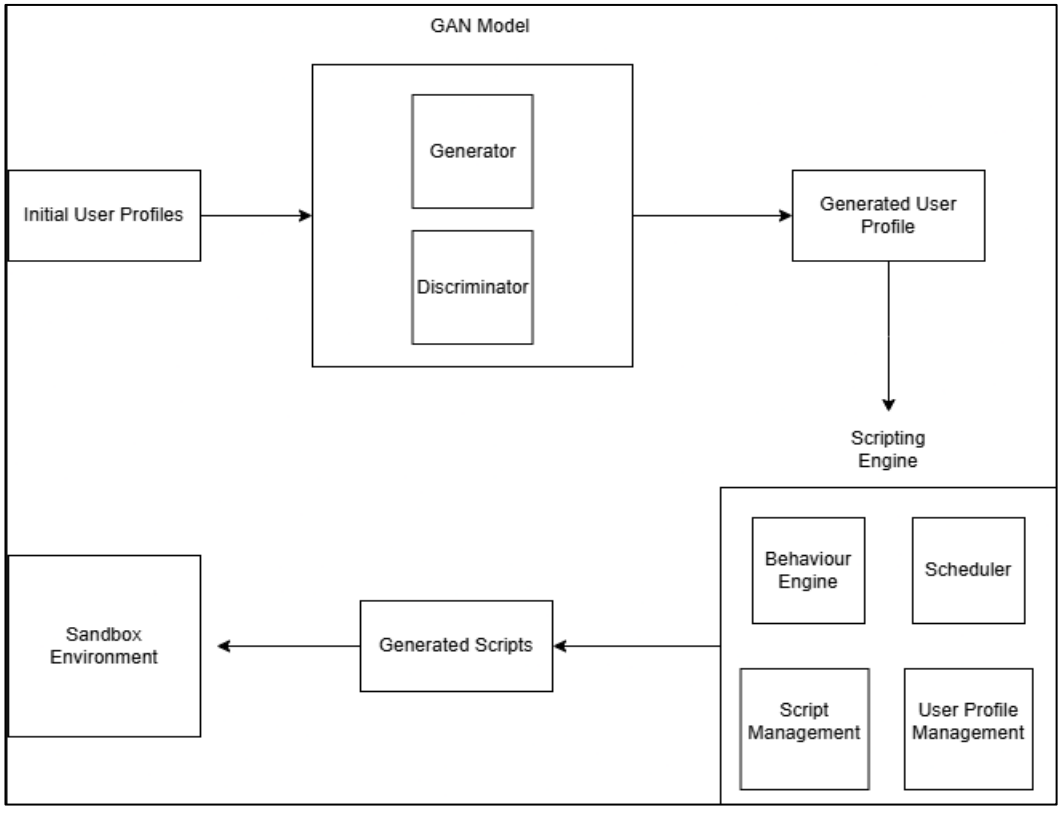


Figure 15: System architecture

- **Initial user profiles**: - First user profiles help to start the process. Derived from publicly accessible databases, these profiles are structured depictions of actual user behavior. Keystroke patterns, digraphs and trigraphs as well as mouse movement metrics are among the features obtained.

- **GAN model**: - The user profiles are sent into a Generative Adversarial Network (GAN). Next neural network components Generator and Discriminator define the GAN. While the Discriminator determines if the user profiles are synthetic or real, the Generator tries to create reasonable synthetic data. By the adversarial training program, the Generator gains knowledge to produce user profiles quite similar to real behavioral patterns.

- **Generated user profile**: - The GAN produces a synthetic yet realistic user behaviour profile. Every profile comprises low-level behavioral information including mouse motion variability, digraph/trigraph sequences, and key stroke dynamics. Search terms for browser behaviour and URLs are created with OpenAI, based on randomly chosen user interests, significantly enrich this profile.

- **Scripting engine**: - The scripting engine is responsible for converting the generated user profile into executable behaviour scripts. It is composed of five key modules:
    - **User Profile Management**: Handles the integration of profile attributes and manages the data used by the engine.
    - **Behaviour Engine**: Updates predefined user activity templates based on profile characteristics.
    - **Script Management**: Selects and builds automation scripts from the updated templates.
    - **Scheduler**: Generates a realistic timetable, distributing activities over time to simulate natural user interaction.

- **Master Automation Logic (internally coordinated)**: Ensures synchronisation and proper execution order of all components.

- **Sandbox environment**: - The final phase of the architecture involves injecting the generated scripts into a virtual environment Cuckoo Sandbox. The scripts are executed within the sandbox, simulating realistic user behaviour such as mouse movements and keyboard input. This activity helps to deceive sandbox-evasive malware into executing, allowing analysts to capture its true behaviour for detection and analysis.

Data Acquisition and Preprocessing

The development of realistic user profiles relies on the acquisition of behavioural data that accurately reflects natural human interaction patterns. This section outlines the types of data used, the preprocessing steps applied, and the construction of initial user profiles that serve as input to the generative model.

- **Data Sources**

  For this research, publicly available datasets were used to obtain user interaction data. The data specifically includes:

  - Mouse movement data – capturing cursor trajectories, pauses, and speed variations.
  - Keystroke data – including keypress durations, time intervals between key events, and sequence patterns (digraphs and trigraphs).

These data types were selected because they represent low-level behavioural traits that are difficult for malware to distinguish from real user actions when simulated properly.

- **Construction of Initial User Schema**

In order to generate consistent and meaningful user profiles, a structured schema was designed to represent the key behavioural attributes extracted from the preprocessed dataset. This schema serves as a blueprint for how user interactions are modelled and later used by the generative model. It ensures that each user profile maintains a standardized format, allowing the GAN to learn and replicate behaviours more effectively.

The schema is built around five main categories of user behaviour: mouse movement, mouse hovering and interaction, keystroke and typing patterns, word-level structure, and digraph/trigraph dynamics. These categories reflect the types of low-level behaviours that are commonly used by sandbox-evasive malware to detect artificial environments.

Each field in the schema is directly derived from measurable attributes present in the dataset. A brief description of the included categories is as follows:

- **Mouse Movement Metrics**: - This category captures the characteristics of mouse activity, including average and variance of movement speed, distance travelled, and scroll behaviour. Features such as path curvature and direction changes are included to reflect complex, human-like cursor movement patterns.

- **Mouse hover and interaction**: - This covers measures of the mouse's interactions with on-screen components. Simulating user attention and responsiveness using hover time and click frequency helps to replicate user response, therefore adding to the realism of simulated behavior.

- **Typing patterns and keystroke**: - This set of characteristics simulates user typing including speed, corrections, key use including modifiers, and

typing pauses. These behavioral features are essential for spotting patterns that show up organically in human input but are often lacking in synthetic simulations.

- **Text and Word Structures**: - Typing habits can be revealed by word-level statistics including word length, variety, and the ratio of capitalized or lengthy words. These elements set apart structured from informal input patterns.

- **Digraph and Trigraph Characteristics**: - These traits derive from combinations of important sequences. Measurement of typing behavior depends critically on digraphs (two-key combinations) and trigraphs (three-key combinations). Strong behavioural indications are the frequency and presence of common combinations.

Figure 16: User profile schema

This schema forms the foundation for all user profiles used in this research. Each preprocessed data record is mapped to this schema to create a structured initial user profile. These profiles are then used as input to the Generative Adversarial Network (GAN), allowing for the generation of synthetic user behaviour that mimics real-world patterns.

User profile generation using GAN

- **Normalization and Feature Vector Structure**

  Before training the GAN model, all features extracted from the initial user schema (see Section 2.3.2) are preprocessed and normalised to a range between -1 and 1. This standardisation is critical for stabilizing GAN training and ensuring all feature dimensions contribute equally during learning.

  The structure of the input data used to train the GAN is based on a fixed-length feature vector, where each element corresponds to a specific metric from the schema. These include values such as:

  - Average mouse speed
  - Variance in hover time
  - Typing speed
  - Frequency of digraph and trigraph sequences
  - Use of keyboard shortcuts and corrections

  Each feature vector represents a complete user session. The GAN is trained to reproduce these distributions without memorizing specific examples.

- **WGAN-GP Architecture and Design Rationale**

To improve training stability and output quality, the system is built using a Wasserstein GAN with Gradient Penalty (WGAN-GP). This framework addresses common GAN training issues such as mode collapse and vanishing gradients.

- **Generator Architecture**

The Generator transforms a 100-dimensional latent vector (sampled from a standard normal distribution) into a high-dimensional user behaviour profile. The network uses multiple fully connected layers with Batch Normalization and Leaky ReLU activations to stabilize gradient flow and accelerate convergence.

Layer Structure:

a) Linear(100 → 256) → BatchNorm → LeakyReLU
b) Linear(256 → 512) → BatchNorm → LeakyReLU
c) Linear(512 → 1024) → BatchNorm → LeakyReLU
d) Linear(1024 → N) → Tanh

(Where N = number of features in the user schema)

- **Critic Architecture**

The Critic (Discriminator) evaluates whether the input profile is real or synthetic. It uses Layer Normalisation (instead of BatchNorm) to avoid feedback instability and includes Dropout layers to improve robustness.

Layer Structure:

a) Linear(N → 1024) → LayerNorm → LeakyReLU → Dropout
b) Linear(1024 → 512) → LayerNorm → LeakyReLU → Dropout
c) Linear(512 → 256) → LayerNorm → LeakyReLU

d) Linear(256 → 1)

Unlike traditional GANs, no sigmoid activation is used in the output. Instead, the model directly estimates the Wasserstein distance between real and fake distributions.

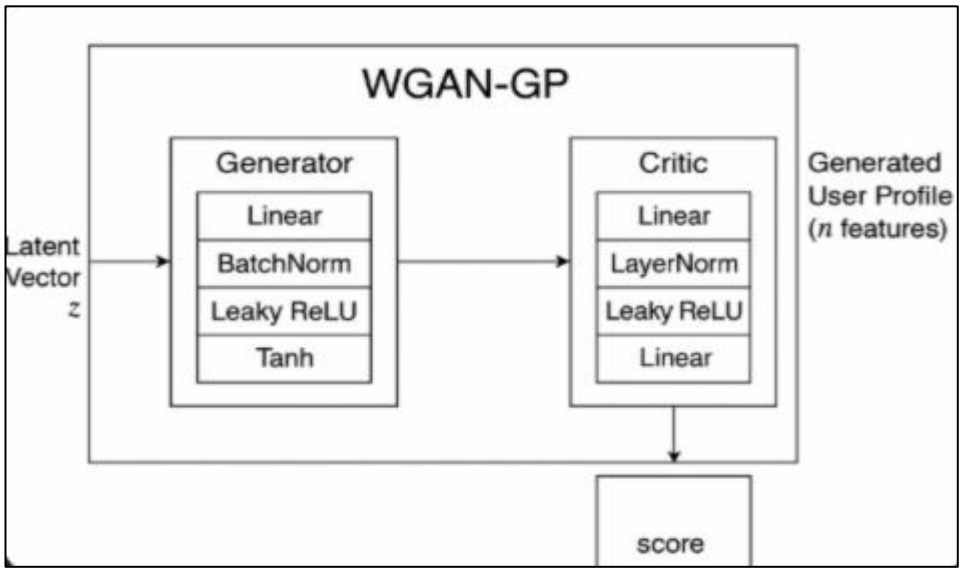


Figure 17: WGAN-GP architecture

- **Synthetic Profile Generation and Postprocessing**

Once the Generator has been trained, it can be used to produce synthetic user profiles by sampling random noise vectors from a latent space. These vectors are passed through the Generator network, which transforms them into feature vectors that resemble real user behaviour.

Each output vector contains a complete set of behavioural attributes such as average mouse speed, typing speed, hover duration, and digraph frequency structured according to the user schema defined earlier. These vectors are still in their normalized form, scaled between -1 and 1, as required by the training process.

To make the synthetic profiles usable by downstream components like the scripting engine, the generated values must be transformed back to their original scale. This process, known as denormalization, uses the stored minimum and maximum values from the original dataset to reverse the scaling. The result is a profile that reflects human-like behaviour patterns in real-world units (e.g., characters per second, pixels per second, milliseconds, etc.).

- **Distribution Matching**

Even after denormalization, there may still be minor differences between the statistical distributions of real and synthetic data. In order to add a layer of realism, a distribution matching method has been added as a postprocessing step that alters the synthetic data by matching distributional properties to that of the real dataset as a postprocessing step.

The synthetic values for each feature are adjusted such that their distribution matches that of the real values for the corresponding feature using a percentile-matching strategy. Therefore, the width of synthetic feature values retains statistical shape compatibility even if the GAN outputs slightly drift. This postprocessing step improves the realism of generated data by correcting small deviations without impacting the relationships or behaviour of the data patterns.

A visual representation of the generation and postprocessing workflow is shown in Figure 18.

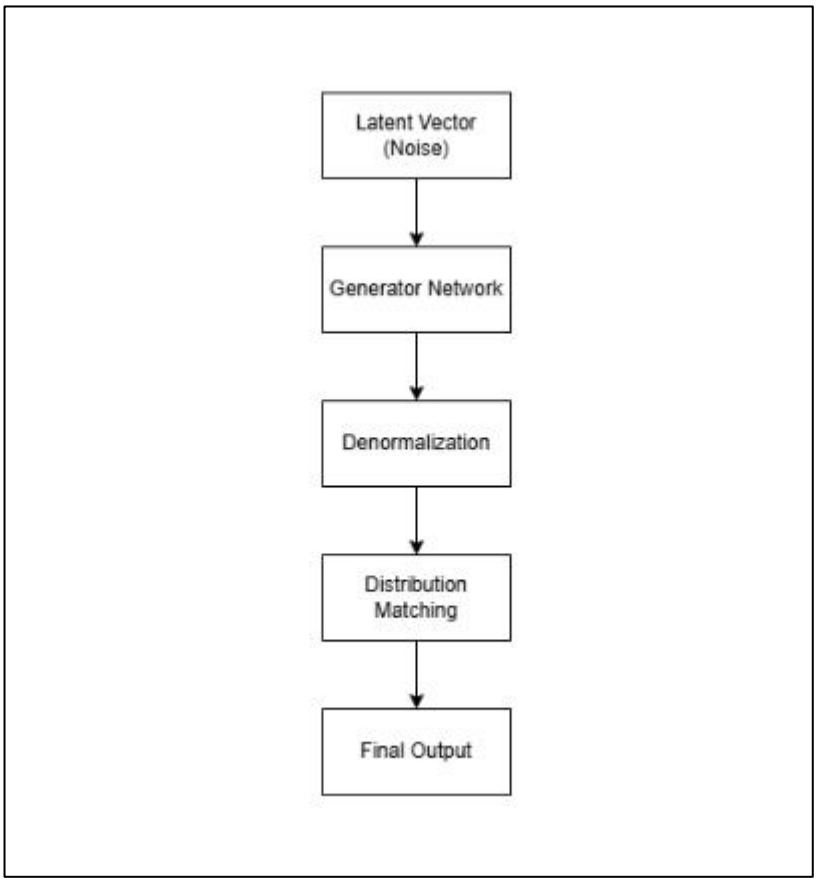


Figure 18: Generation and postprocessing workflow

- **Latent Vector (Noise)** – Random input sampled from a normal distribution
- **Generator Network** – Produces a normalised behavioural profile
- **Denormalisation** – Converts the profile into real-world values
- **Distribution Matching** – Aligns the output with real data distributions
- **Final Output** – A complete, human-like user behaviour profile

Below figure 19 shows a generated user profile using the WGAN-GP model.

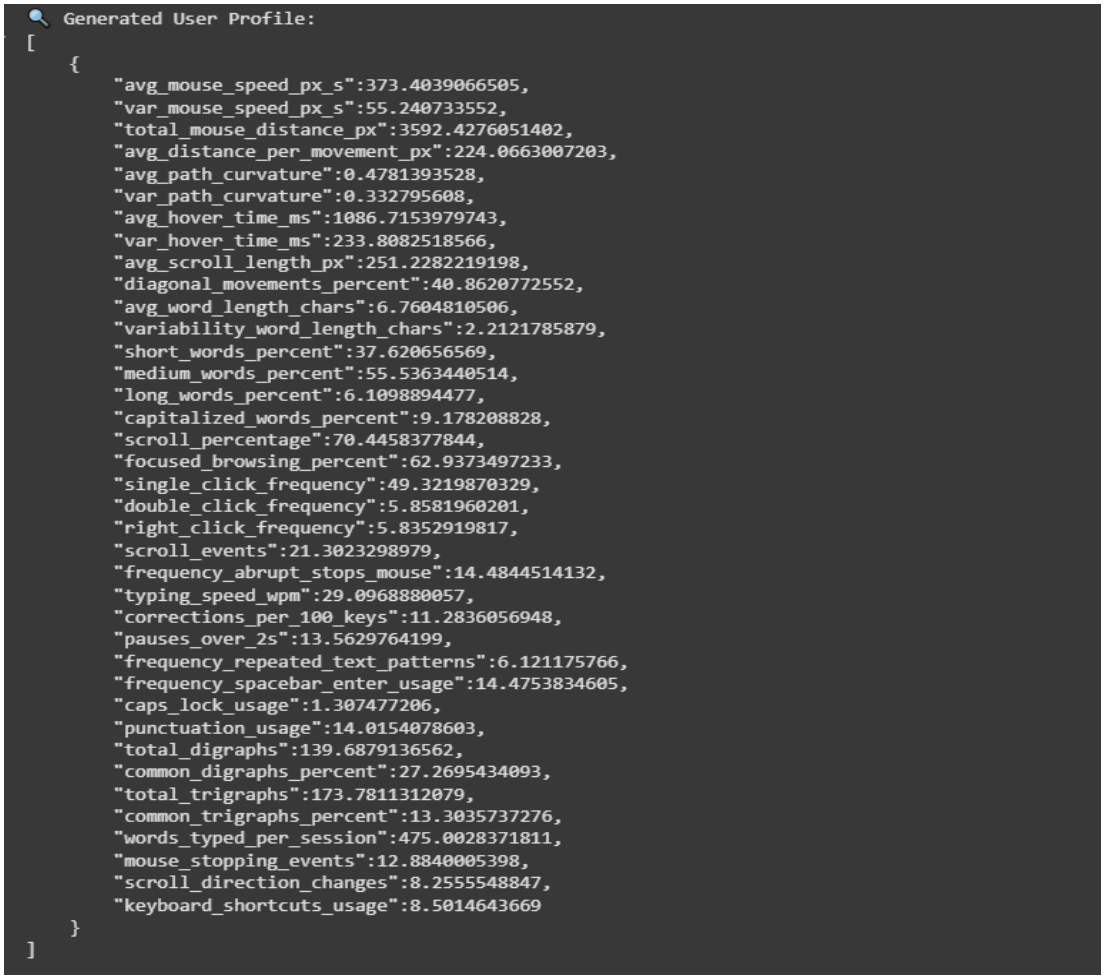


Figure 19: WGAN-GP model generated output

- **Integration with OpenAI for Profile Enrichment**

The GAN model does not consider contextual factors of user activity, including browsing behavior or interest-driven activities, even if it efficiently creates the behavioral traits of a user (such as keyboard patterns and mouse dynamics). The solution uses OpenAI's language model APIs in an enrichment layer to replicate reasonable user context depending on specified interests, hence addressing this.

- **Interest Selection -** From a pre-defined, large pool of categories e.g., technology, fitness, travel, finance each synthetic user is assigned a set of four

random interests. These interests allow one to replicate the variety and precision of actual user intentions

- **OpenAI Integration -** The system uses OpenAI to do two enrichment chores for every given interest**:**

    a) **Search Term Generation** – Realistic queries a user with that interest might search for online.

    b) **URL Generation** – Plausible websites a user might visit, consistent with their interest-driven behaviour.

This integration gives otherwise numerical user profiles semantic and activity context therefore enabling downstream simulation scripts to replicate more complete human activities.

```
Generated Prompt:

You are an expert in internet trends and user behavior. For the following interests:
- Board games, Acrylic painting, Aerobics, Collecting antiques,
generate:
1. Four realistic search terms for each interest.
2. Four popular websites for each interest.
Also, include one trending topic globally and generate:
1. Four search terms for this trending topic.
2. Four popular websites for this trending topic.

Response format (strictly adhere to this structure):
{
    "interests": [
        {
            "interest": "<interest name>",
            "search_terms": ["<term1>", "<term2>", "<term3>", "<term4>"],
            "websites": ["<url1>", "<url2>", "<url3>", "<url4>"]
        },
        ---
    ],
    "trending_topic": {
        "topic": "<trending topic>",
        "search_terms": ["<term1>", "<term2>", "<term3>", "<term4>"],
        "websites": ["<url1>", "<url2>", "<url3>", "<url4>"]
    }
}
```

Figure 20: Generated OpenAI prompt


```
{
  "interests": [
      {
          "interest": "Board games",
          "search_terms": ["best new board games", "classic board games", "board game reviews", "how to play chess"],
          "websites": ["www.boardgamegeek.com", "www.theguardian.com/games/series/board-games", "www.games-workshop.com", "www.zmangames.com"]
      },
      {
          "interest": "Acrylic painting",
          "search_terms": ["acrylic painting techniques", "how to start acrylic painting", "landscape acrylic painting", "best acrylic paints"],
          "websites": ["www.artsy.net", "www.jerrysartarama.com", "www.liquitex.com", "www.goldenpaints.com"]
      },
      {
          "interest": "Aerobics",
          "search_terms": ["aerobics workout for weight loss", "beginner aerobics steps", "best aerobics classes", "Zumba workouts"],
          "websites": ["www.healthline.com", "www.self.com", "www.shape.com", "www.myfitnesspal.com"]
      },
      {
          "interest": "Collecting antiques",
          "search_terms": ["how to value antiques", "popular antique items", "antique shows near me", "restoring antiques"],
          "websites": ["www.antiquesroadshow.com", "www.kovels.com", "www.the-saleroom.com", "www.barnebys.com"]
      }
  ],
  "trending_topic": {
      "topic": "Metaverse",
      "search_terms": ["Metaverse explained", "leading companies in metaverse", "how to experience the metaverse", "real estate in metaverse"],
      "websites": ["www.techcrunch.com", "www.medium.com/tag/metaverse", "www.theguardian.com/technology/metaverse", "www.futureofthemetaverse.com"]
  }
}
```


Figure 21: OpenAI response with search terms & URLs

- **Final Output Profile Format**

The final output of the synthetic profile generation pipeline is a fully enriched user profile that combines:

- Behavioural features generated by the WGAN-GP model
- Interest-based enrichment data (search terms and visiting URLs) obtained via OpenAI

This profile is designed to reflect both the low-level interaction patterns (e.g., keystrokes, mouse dynamics) and high-level interest-driven behaviour, enhancing the realism of simulated users in sandbox environments.

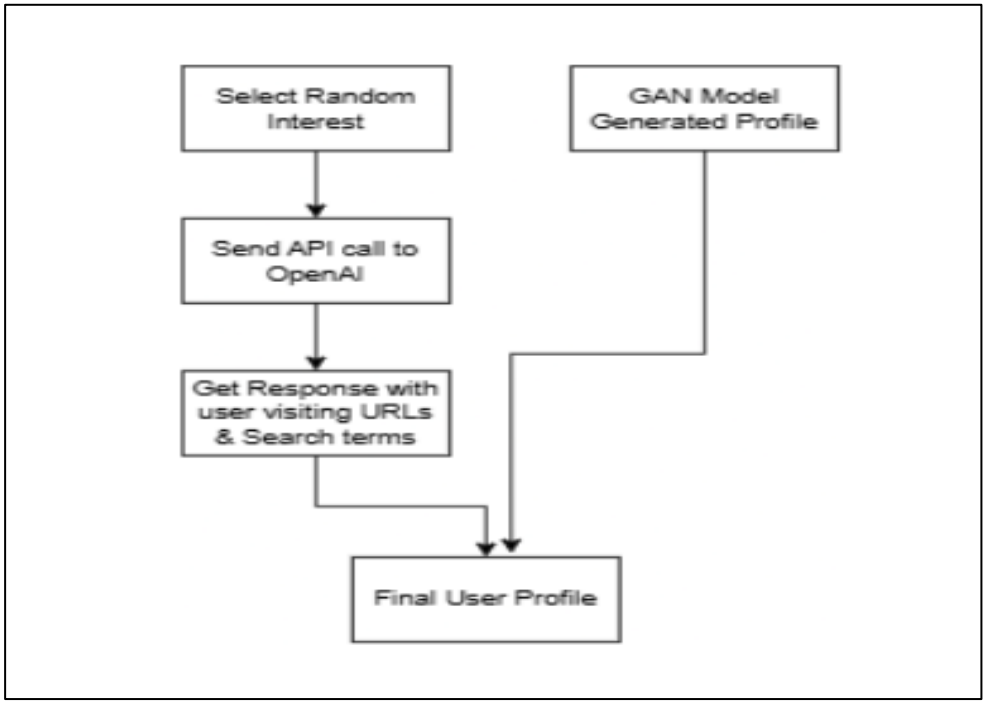


Figure 22: Complete user profile creation workflow

Below figure 23, shows an enhanced final user profile.


```
{
  "avg_mouse_speed_px_s": 373.4,
  "var_mouse_speed_px_s": 55.24,
  "total_mouse_distance_px": 3592.43,
  "avg_distance_per_movement_px": 224.07,
  "avg_path_curvature": 0.48,
  "var_path_curvature": 0.33,
  "avg_hover_time_ms": 1086.72,
  "var_hover_time_ms": 233.81,
  "avg_scroll_length_px": 251.23,
  "diagonal_movements_percent": 40.86,
  "avg_word_length_chars": 6.76,
  "variability_word_length_chars": 2.21,
  "short_words_percent": 37.62,
  "medium_words_percent": 55.54,
  "long_words_percent": 6.11,
  "capitalized_words_percent": 9.18,
  "scroll_percentage": 70.45,
  "focused_browsing_percent": 62.94,
  "single_click_frequency": 49.32,
  "double_click_frequency": 5.86,
  "right_click_frequency": 5.84,
  "scroll_events": 21.3,
  "frequency_abrupt_stops_mouse": 14.48,
  "typing_speed_wpm": 29.1,
  "corrections_per_100_keys": 11.28,
  "pauses_over_2s": 13.56,
  "frequency_repeated_text_patterns": 6.12,
  "frequency_spacebar_enter_usage": 14.48,
  "caps_lock_usage": 1.31,
  "punctuation_usage": 14.02,
  "total_digraphs": 139.69,
  "common_digraphs_percent": 27.27,
  "total_trigraphs": 173.78,
  "common_trigraphs_percent": 13.3,
  "words_typed_per_session": 475,
  "mouse_stopping_events": 12.88,
  "scroll_direction_changes": 8.26,
  "keyboard_shortcuts_usage": 8.5,
  "browser_behaviors": {
    "search_terms": [
      "best new board games",
      "classic board games",
      "board game reviews",
      "how to play chess",
      "acrylic painting techniques",
      "how to start acrylic painting",
      "landscape acrylic painting",
      "best acrylic paints",
      "aerobics workout for weight loss",
      "beginner aerobics steps",
      "best aerobics classes",
      "Zumba workouts",
      "how to value antiques",
      "popular antique items",
      "antique shows near me",
      "restoring antiques",
      "Metaverse explained",
      "leading companies in metaverse",
      "how to experience the metaverse",
      "real estate in metaverse"
    ],
    "websites": [
      "www.boardgamegeek.com",
      "www.theguardian.com/games/series/board-games",
      "www.games-workshop.com",
      "www.zmangames.com",
      "www.artsy.net",
      "www.jerrysartarama.com",
      "www.liquitex.com",
      "www.goldenpaints.com",
      "www.healthline.com",
      "www.self.com",
      "www.shape.com",
      "www.myfitnesspal.com",
      "www.antiquesroadshow.com",
      "www.kovels.com",
      "www.the-saleroom.com",
      "www.barnebys.com",
      "www.techcrunch.com",
      "www.medium.com/tag/metaverse",
      "www.theguardian.com/technology/metaverse",
      "www.futureofthemetaverse.com"
    ]
  }
}
```


Figure 23: Complete user profile

Scripting Engine

- **Overview of the Scripting Engine Module**

The Scripting Engine Module is a fundamental part of the system that takes the generated user profiles and turns them into executable behaviour scripts. Its goal is to imitate real user activity in a sandboxed environment, making it seem as if the malware is activated as if a real user was conducting a real session.

This module acts as the bridge between the output of the user profile generation pipeline and the final execution phase inside the sandbox. It takes enriched user profiles containing both behavioural attributes and interest-based metadata and dynamically generates a time-structured script that reflects genuine user activity.

To support a flexible and scalable simulation system, the scripting engine is designed using a modular architecture. It comprises five specialised submodules:

- **Master Automation Module** – Orchestrates and synchronizes the execution of all submodules in the correct order.
- **User Profile Management Module** – Handles the parsing, validation, and distribution of attributes from the input user profile.
- **Behaviour Engine Module** – Updates predefined behaviour templates using parameters from the user profile (e.g., typing speed, scroll activity).
- **Script Management Module** – Organizes, selects, and compiles appropriate scripts based on the updated templates and simulation context.
- **Scheduler and Executor Module** – Generates the timetable, assigns activities over time, and executes the full script within the sandbox.

The scripting engine ensures that each simulation is context-aware, personalized, and non-repetitive, which is critical for deceiving sandbox-aware

malware. It supports a mix of both static and dynamic activity scripts, blending system-level artefact creation with user-like interaction patterns.

This architecture promotes extensibility, enabling the addition of new script templates or behaviour types without modifying the core pipeline. The following sections describe each of these modules in detail.

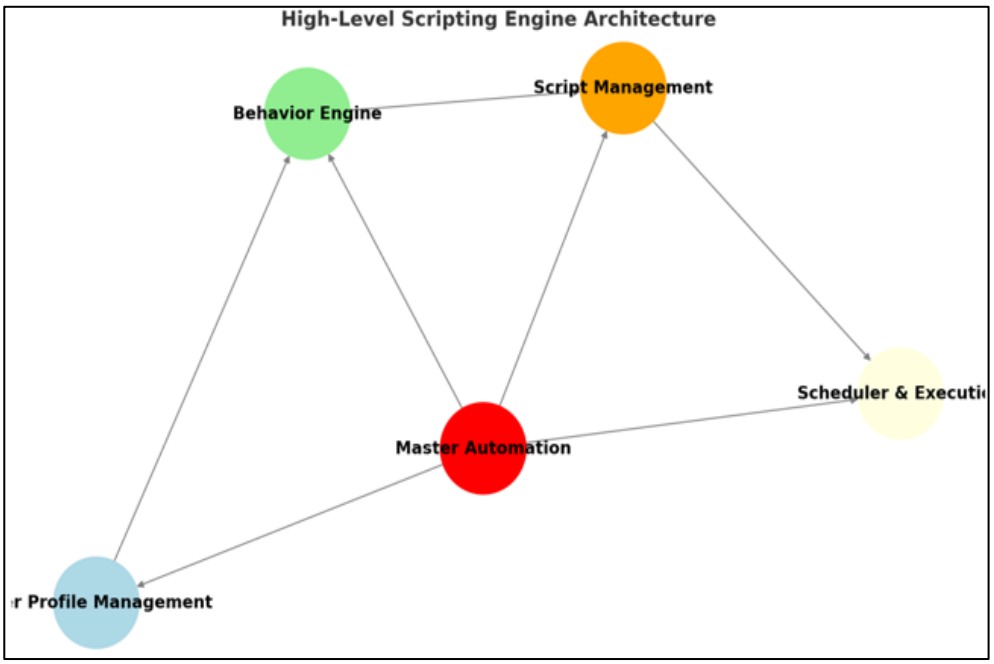


Figure 24: Scripting engine architecture

- **Master Automation Module**

The Master Automation Module is the central controller of the Scripting Engine. Its role is to coordinate and manage the execution flow of all subordinate modules, ensuring that each step in the script generation pipeline is performed in the correct order and with the required input data.

- **User Profile Management Module**

The scripting engine pipeline starts with the User Profile Management Module. Its main duties are parse, validate, and extract pertinent attributes from the enhanced user profiles produced in the past phases of the system.

- **Behaviour Engine Module**

Generating generalised user behaviour patterns based on the enhanced user profile falls to the Behaviour Engine Module. Among these behaviors are mouse movement dynamics, typing qualities, and scrolling action. The Behaviour Engine generates reusable interaction segments reflecting the natural input style of a given user instead of whole scripts.

- **Script Management Module**

Managing, choosing, and compiling the predefined Python-based user behavior scripts needed to replicate reasonable desktop activity falls to the Script Management Module. These models are meant to show typical user behavior in several spheres, including online surfing, file editing, program running, system interactions, multimedia handling, and so on.

- **Scheduler and Executor Module**

The Scheduler and Executor Module is the final component of the scripting engine. Its role is to determine when each user activity should take place during the simulation session and to ensure that the final set of scripts is executed in a timed and controlled manner within the sandbox environment.

- **Scripting Engine Workflow**

This subsection describes the overall workflow that the scripting engine follows once a user profile is generated. The workflow documents how the process is automated behavioural data from users into a scheduled simulation within the sandbox.

The process begins with extracting appropriate behavioural traits (e.g., typing speed, mouse jitter, scrolling intervals, etc.) from the enriched user profile. These parameters are used to generate a realistic set of baseline behaviours. Activity script templates are

chosen and updated with behaviour to ensure these generated actions are consistent with real user behaviours. Finally, the scheduling engine generates an activity schedule for each of the scripts. After time mapping the scripts, these scripts are packaged and loaded into the sandbox.

Figure 25 shows a flow diagram to illustrate this sequence of steps:

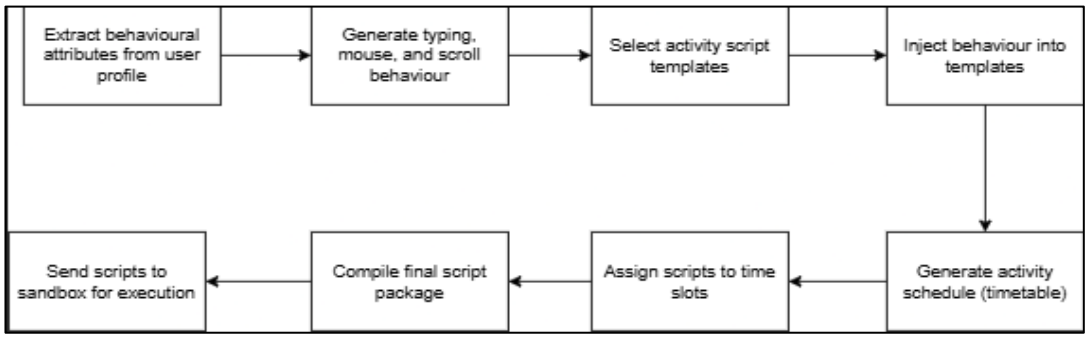


Figure 25: Scripting engine workflow

Sandbox Integration

Once the full set of personalised behaviour scripts and the corresponding timetable have been generated, the final simulation package is sent to the sandbox environment. The sandbox is the execution environment where the synthetic user behaviour is replayed in real time, with the goal of triggering and observing the actions of sandbox-evasive malware.

This research uses Cuckoo Sandbox, a well-established, open-source malware analysis platform that supports virtualised environments and can be extended to support user interaction automation.

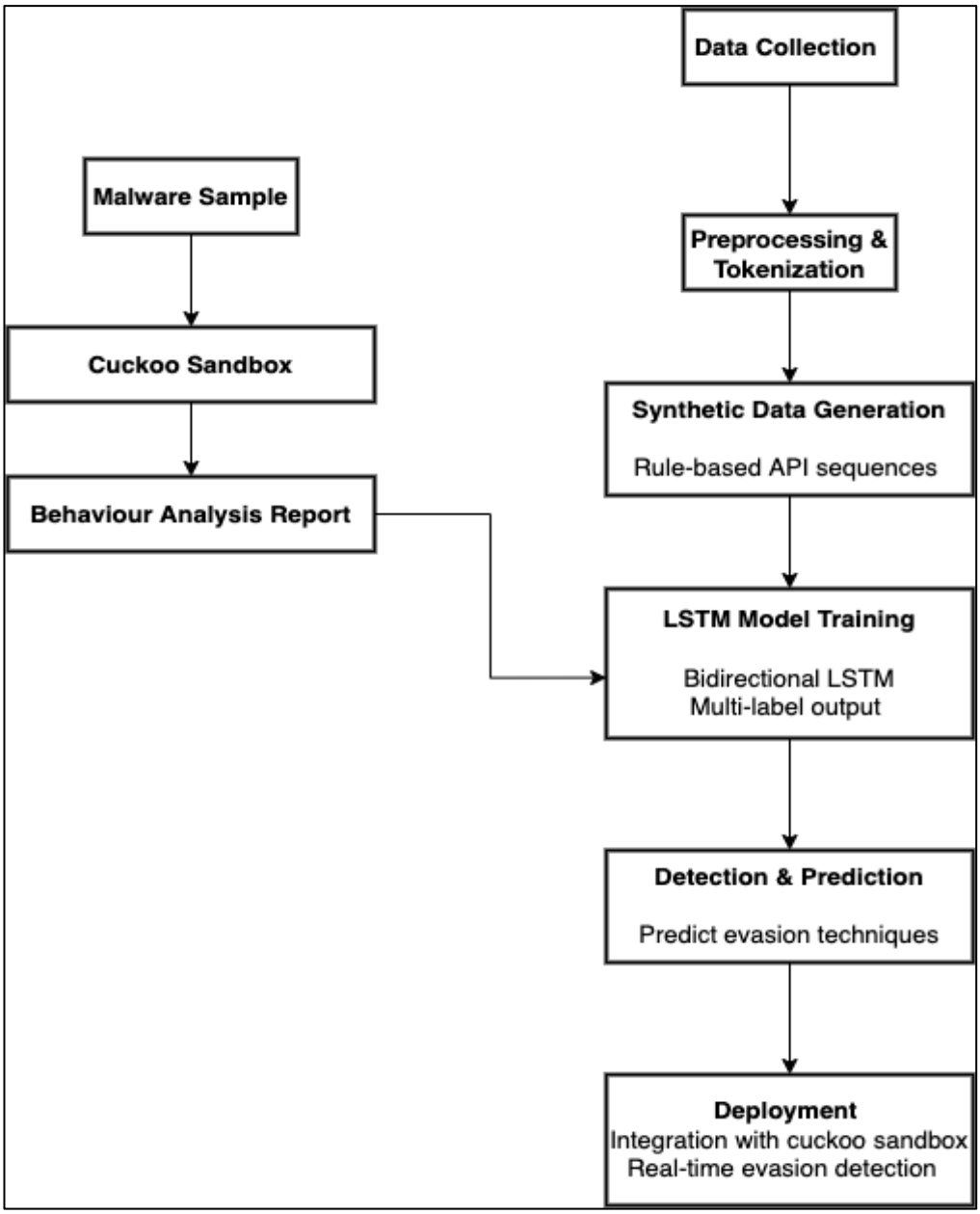
2.1.3 Methodology Framework – Hybrid Detection System for Sandbox Evasion Techniques Component

The selected methodology is consistent with the research objectives in that it seeks to fill knowledge gaps identified in existing detection frameworks while providing a more adaptable and extensible solution. The approach encompasses key aspects, such as data collection in sandbox environments, building off an LSTM model for malware behavior analysis, and integration with sandbox tools for real-time detection. Additionally, this approach consists of a comparative analysis of different evasion techniques that uses machine learning detection models to detect and classify multiple types of sandbox evasion. By addressing not only technical challenges in detection but also practical challenges in real-world implementation, this methodology presents a thorough approach to addressing the state of sandbox evasion detection. The approach in this methodology would likely allow for the system to adjust to developing threats, thus providing a more effectual and efficient solution for cybersecurity professionals.

Overview of the process

The process of identifying sandbox evasion techniques utilized by malware includes several steps addressing the research problem's various components. Firstly, malware (specifically involving the file system and the registry) was collected, and the malware samples were executed in the Cuckoo Sandbox spawning cloud environment. The sandbox is specifically configured and runs samples in a safe, segregated space in which they have no effect on production systems. The sandbox produces behavioral analysis reports for each malware example run on the specific operating system. The reports are critical for compiling the relevant content to understand how the malware behaves in the evaluation environment. The reports contain relevant information, including API calls made during execution, the OS system calls made, which processes and threads were spawned, and any network activity observable at the time of execution [5].

The next step involves collecting, preprocessing, and tokenizing the data to convert raw reports into a format suitable for input into a machine learning model. As discussed earlier, the dataset is imbalanced, so I will utilize synthetic data generation through rule-based methods to create additional examples of the evasion tactics that are underrepresented in the base dataset. Once the data has been prepared, Research is focusing build a Bidirectional LSTM (Long Short-Term Memory) model and then train that model to learn the patterns associated with sandbox evasion techniques and classify/predict multiple evasion techniques, based on the behavior patterns extracted from the sandbox reports and processed into a single input sequence. Once complete, the model is integrated with Cuckoo Sandbox for live evasion detection, where it will predict evasion techniques as malware executes. The system undergoes thorough testing and validation, and gets deployed to ensure the development system can detect an respond to evasion tactics, which is detailed in the following system diagram, which outlines the sequence of process illustration.



The above system diagram outlines the full process for detecting sandbox evasion tactics performed by malware. Every stage in the diagram denotes an important part of the method, starting with the malware samples gathered and ending with the prediction of evasion tactics in a live environment when the model is deployed. The diagram provides an overarching view concerning how data and interaction will flow through the system while demonstrating the relationships among the components. The flow of the system begins with the malware being executed in a Cuckoo Sandbox,

moves on through the collection and preprocessing of behavior analysis reports, will continue to the generation of synthetic data to overcome imbalance, training of the model using LSTM, and ends with the encompassing model running in a live environment. The next sections will delve deeper into each of these steps contributing feedback regarding function, details of processes, and how each step worked towards the overall goal of detecting malware evasion tactics.

Data collection

The first step in the methodology is to gather a rich and balanced data set of malware samples that specifically target a known sandbox evasion technique. Since the model we are developing will use sandbox analysis data, we must ensure that our data set comes from sandbox environments that truly depict the behavior of malware in the wild. Sandboxes, such as Cuckoo Sandbox, provide a way to execute malware in a controlled environment to capture detailed API call sequences, interactions with the underlying operating system, and other behavioral characteristics of malware. These behavioral characteristics are important for the detection of evasion techniques because they represent the malware behavior attempting to avoid detection in a sandbox environment.

In order to create this dataset, samples will be obtained from existing research, curated, publicly available malware repositories, and sandbox reports from actual malware analysis. The API sequences observed during sandbox execution will be the primary data type that will be used for the model because they have a clear and structured way of showing malware actions and interactions within the system. As sandbox environments are deliberately intended to imitate real-world environments in which malware will seek to evade detection, using data collected from sandbox analysis will ensure that the model will be training on data that accurately reflects the environment in which it will operate. Training the model on sandbox analysis data will also increase

its detection capabilities of new and potential evasion techniques, thus improving its accuracy and effectiveness within the detection system.

Four primary datasets will be utilized in this research:

1. **API Call Sequence Dataset**:

Source: This dataset is part of a broader research initiative on malware detection and classification using Deep Learning. It includes 42,797 malware API call sequences and 1,079 goodware API call sequences. Each sequence comprises the first 100 non-repeated API calls associated with the parent process, extracted from the 'calls' elements in Cuckoo Sandbox reports. This data is crucial for understanding the behavior of malware, particularly how it interacts with system APIs during execution [26]

Relevance: The dataset, cited in the work by Oliveira and Sassi (2019), provides a rich foundation for training the generative model to identify and simulate sandbox evasion techniques based on API behaviors [26].
(https://www.kaggle.com/datasets/ang3loliveira/malware-analysis-datasets-api-call-sequences)

2. **Mal-API-2019 Dataset**:

Source: Generated by Cuckoo Sandbox, this dataset is focused on Windows OS API calls and is intended for machine learning-based malware research. It is particularly valuable for studying metamorphic malware, which alters its behavior by modifying API call sequences. The dataset has been used in studies by Yazı and Çatak (2019) and serves as a benchmark for classifying malware based on API calls [27].

Relevance: This dataset will help address gaps in understanding how metamorphic malware changes its behavior to evade detection, a key aspect of this research [27].
(https://www.kaggle.com/datasets/focatak/malapi2019)

3. **CAPEv2 Dynamic Analysis Dataset**:

Source: This dataset consists of dynamic analysis reports generated by CAPEv2, an open-source successor to Cuckoo, widely used for analyzing potentially malicious binaries. It includes 26,200 PE samples (8,600 goodware and 17,675 malware) executed in Windows 7 VMware virtual machines. The dataset spans from 2012 to 2020, with malware samples sourced from VirusTotal and goodware from Chocolatey community-maintained packages. The analysis reports detail system calls, file system interactions, registry changes, and other critical behaviors observed during execution [28].

Relevance: The CAPEv2 dataset is particularly valuable for its comprehensive dynamic analysis reports, which include detailed behavioral information that can be directly leveraged to improve the model's ability to detect and simulate sandbox evasion techniques. The dataset also addresses challenges in malware analysis, such as mitigating evasive checks by malware designed to detect virtualized environments [28].

(https://www.kaggle.com/datasets/greimas/malware-and-goodware-dynamic-analysis-reports)

4. **API Sequences Malware Datasets**

Source: This dataset, available on GitHub, provides a dynamic malware analysis benchmark that is built using hashcodes of malware files, API calls from the **PEFile library in Python**, and the malware types obtained from the **VirusTotal API**. The dataset is presented in CSV format and contains two major sets: **VirusSample** and **VirusShare**. The **VirusSample** dataset includes 9,795 samples obtained from VirusSamples, while the **VirusShare**dataset includes 14,616 samples from VirusShare [29].

Relevance: This dataset is highly relevant for the research as it provides a rich source of dynamic analysis data that captures malware behaviors through API call sequences. The dataset is well-suited for studying the detection of evasive malware, particularly with respect to how different malware families interact with system APIs during execution. By incorporating the **VirusSample** and **VirusShare** datasets, the model will be able to analyze a wide range of malware types and their corresponding API call behaviors. The inclusion of balanced datasets, after rebalancing to address class imbalances, further improves its usefulness for training machine learning models to detect various malware families, including **Adware**, **Trojan**, **Ransomware**, **Worms**, and others, as well as their evasion strategies. This dataset also allows for robust comparisons of performance metrics across machine learning classifiers, with algorithms such as **XGBoost** and **SVM** achieving high accuracy, making it an essential resource for training the proposed detection model [29].

(https://github.com/khas-ccip/api_sequences_malware_datasets)

Data preprocessing and tokenization

Tokenization and data preprocessing are an essential phase of creating malware behavior data for machine learning models. Tokenization and data preprocessing turn raw behavior logs into usable, structured formats suitable for training and testing. Initially, when preprocessing the data, it must be cleaned by removing any rows of times that did not employ even a single API call, which effectively ensures that only fully justified, appropriate API call sequences will go through the pipeline. Addressing evasion labels, is difficult when API calls sometimes I do not get a label or incomplete labels. In pre-processing, those incomplete or missing evasion labels were filled in with a default value, "No Evasion," to standardize the dataset.

Once the data is fully cleaned the next step is to tokenizing the API call sequences. In this process, each unique API call is mapped to a numerical token that is crucial for converting the textual data into a format that can be processed by machine learning algorithms. This is achieved by creating a tokenizer that breaks down the API sequences into individual tokens based on predefined delimiters, such as commas or spaces. The tokenizer then converts the API sequences into numerical sequences of tokens, representing each API call by its corresponding integer. To ensure that all sequences have a uniform length, the sequences are padded to a maximum length, either specified by the user or determined dynamically from the data. Padding ensures that shorter sequences are extended to the same length, facilitating batch processing and avoiding issues during training.

Text to Sequence Conversion: The tokenizer converts each API call in the sequence into an integer based on the word index. The vocabulary size is calculated as the total number of unique API calls encountered during training.

Sequence=Tokenizer.texts_to_sequences(X)

Where:

- X is the list of raw API call sequences.
- texts_to_sequences() method converts each API call into an integer index representing a word in the vocabulary.

Padding: After tokenization, the sequences are padded to a fixed length (max_sequence_length) to ensure uniformity in the input size.

Padded_Sequence = pad_sequences(Sequence, maxlen=max_sequence_length, padding='post')

Padding is performed to ensure all input sequences have the same length, which is required for LSTM input.

Also, the multi-label classification operation requires encoding evasion labels for each malware sample. In this section, multi-label binarization was used on the evasion labels to transform each conceivable evasion approach into a binary vector. This allowed the model to anticipate numerous evasion tactics simultaneously during testing. The combination of tokenised API sequences and multi-label encoded labels produces a data format suited for input into machine learning models, notably LSTM networks, which have demonstrated efficacy in modelling sequential data such as API call sequences. Thus, the preprocessing pipeline achieves the purpose of organising the data for training. The model can learn high-level notions about sandbox evasion tactics from sequential data, as well as how different combinations of malware API calls affect evasion strategy.

```
# Step 1: Load and Preprocess Data
def load_data(filepath):
    data = pd.read_csv(filepath)
    # Drop unsupported class (0 samples)
    data = data.drop(columns=['registry_environment_checks'], errors='ignore')
    # Separate features (X) and labels (y)
    label_columns = ['vm_detection', 'sandbox_process_detection', 'timing_evasion',
                     'user_interaction_detection', 'anti_debugging']
    X = data.drop(columns=label_columns)
    y = data[label_columns]
    return X, y

# Step 2: Split Data into Train/Validation Sets
def split_data(X, y):
    X_train, X_val, y_train, y_val = train_test_split(
        X, y, test_size=0.2, stratify=y, random_state=42
    )
    return X_train, X_val, y_train, y_val

# Step 3: Apply SMOTE to Training Data
def apply_smote(X_train, y_train):
    smote = SMOTE(random_state=42)
    X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
    return X_resampled, y_resampled
```

Figure 26: using SMOTE to fix the class imbalance

To solve the challenge of class imbalance in the dataset, **SMOTE (Synthetic Minority Over-sampling Technique)** and class weighting methods were applied to improve representation of the rare evasion classes and avoid the case for modelling rare evasion techniques. SMOTE is a data augmentation technique that synthesizes samples of the

minority classes in feature space. The method produces new instances by sampling points between existing samples from the minority class, again balancing the dataset to ensure the model has adequate examples to learn from for rare evasion techniques.

In addition, class weighting was employed in the model training process, giving increased weight to each of the minority classes higher weight meant to assign higher importance to the minority classes and reduce the built-in bias towards the majority classes. The combined application of SMOTE for data augmentation as well as class weighting for optimizing the model led the model to be trained on a more balanced dataset and improve the ability to detect and classify rare sandbox evasion techniques. These strategies may also improve model performance and allow for improved generalization to real-world malware samples with different evasion strategies.

Model Train

Selected Long Short-Term Memory (LSTM) networks to detect sandbox evasion techniques based on their nature to detect sequences where order matters. Malware behavior is sequential in nature, especially API call sequences; thus the sequence of API calls provides important contextual clues regarding the malware's intent, behavior, and evasive strategies. LSTM networks are an advancement of recurrent neural networks (RNNs) and specifically, are suited for sequential data due to long-term memory and retention of relevant information over time. This made them ideal for analyzing API sequences produced while malware was executing in a sandboxed environment. By taking advantage of the LSTM's ability to learn temporal patterns, our model can detect subtle behaviors including those that are indicative of evasive tactics that change within the analysis environment [30].

Model Architecture and Design

The model architecture **employs** Bidirectional Long Short-Term Memory (Bi-LSTM) networks to analyze and capture the sequential behavior of malware; in particular, it looks at API call sequences. LSTM networks are most efficient for our needs as they can learn temporally long-range dependencies in sequential data; for example, they can learn how a malware sample interacts with system APIs over time. In malware analysis, the order and context are crucial for understanding the sample's behavior, as well as for uncovering evasive techniques [31].

Bidirectional LSTMs improve this procedure by looking at the input sequence both forwards and backwards. Standard LSTM networks process input data trajectories from the past to the future and make predictions based only on the context provided by the prior input[4]. In contrast to LSTM networks, Bi-LSTM networks have two layers: one runs through the sequence from past to future, and the other runs through the future to past. This type of sequencing allows the networks to learn dependencies in both directions and provide a more comprehensive representation of the input sequence. In the context of malware detection, this enables the model to evaluate the context for previous and future API calls, so it may be better equipped to detect subtle differences and relationships evidence of sandbox evasion techniques [32].

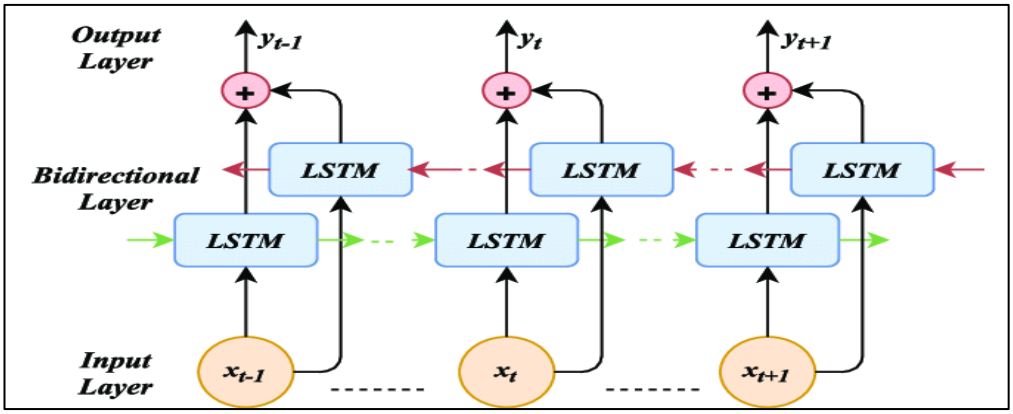


Figure 27: Architecture of a Bidirectional LSTMs

As shown in the diagram, the Bidirectional LSTM layers of the model will apply to each API call sequence in both forward and reverse directions, allowing the model to rely on contextual relationships built throughout the entire sequence. This is particularly beneficial when detecting evasion techniques where action may depend not solely on prior actions (e.g., previous API calls) but even on expected future actions (e.g., the malware may change behaviors when it believes that it has or is being analyzed). This more extensive approach to contextual networks will improve the model's classification accuracy across a variety of classes, especially in complex behaviors such as evasion across malware execution phases [32].

Embedding Layer: The embedding layer projects the input sequence (which consists of integer-encoded API calls) into a dense vector space of size 128. The embedding layer is initialized with random weights that will be trained during the training process. Each token (API call) is mapped to a dense vector of size 128:

$$\text{Embedding Output} = W_{\text{emb}} \cdot \text{Input}$$

Where:

- W_{emb} is the embedding matrix, which is learned during training.

Bidirectional LSTM Layers: Bidirectional LSTM layers are used to capture both past and future context in the sequence. Each LSTM unit computes the hidden state at time t_1,h_{t1} using the following equations:

$$h_t = \text{LSTM}(h_{t-1}, x_t)$$

Where:

- h_t is the hidden state at time tt,
- h_{t-1} is the hidden state from the previous time step,
- x_t is the input at time t (i.e., the tokenized API call at time t).

Since the model is bidirectional, two LSTMs are used:

- One processes the sequence from left to right (forward),
- One processes the sequence from right to left (backward).

Incorporating Bidirectional LSTMs into the model is designed to improve the model's ability to competently manage the complexities associated with malware behavior, allowing the learned patterns to be more resistant to subtle differences in API sequences and recognize novel evasion techniques in real-time, thus becoming a strong method for multi-class classification of evasive techniques in a sandbox. This will fit closely with recent progress in machine learning on sequence data with Bidirectional LSTMs performing better than unidirectional methods.

Model Training

The model training process involves splitting the dataset into three subsets: training, validation, and test sets. The training set is used to train the model, the validation set is used to tune the model's hyperparameters and monitor for overfitting, while the test set is used to evaluate the model's generalization performance. The training process is carried out using the **binary cross-entropy loss function** due to the multi-label classification nature of the problem, where each malware sample could exhibit multiple evasion techniques. The model uses the **Adam optimizer** for efficient gradient-based optimization and is evaluated based on the **accuracy** metric, alongside other metrics such as F1-score to balance precision and recall.

Binary Cross-Entropy Loss: The binary cross-entropy loss is used to compute the error between predicted probabilities and true labels for each class in a multi-label classification setting. The loss for each sample is calculated as:

$$L = -\sum_{i=1}^{N} (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

Where:

- N is the number of classes (sandbox evasion techniques),
- y_i is the true label for class i,
- $\hat{y}_i$ is the predicted probability for class i.

The total loss for the entire dataset is the mean of the individual losses.

Adam Optimizer: The Adam optimizer is used for gradient-based optimization. Adam combines the advantages of both momentum and adaptive learning rates. The update rule for Adam is given by:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$
$$\theta_t = \theta_{t-1} - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Where:

m_t and v_t are the first and second moments (exponentially decaying averages of past gradients),
g_t is the gradient at time t,
α is the learning rate,
β_1 and β_2 are the decay rates for the moment estimates,
ε is a small constant to prevent division by zero.

Early Stopping and Model Checkpoint: Early stopping is used to halt training if the validation loss does not improve after a specified number of epochs (patience). The model checkpoint saves the best-performing model during training based on the validation loss.

```python
# Train the model
def train_model(model, X_train, y_train, X_val, y_val, batch_size=32, epochs=20):
    print("Training model...")

    # Callbacks
    early_stopping = EarlyStopping(
        monitor='val_loss',
        patience=3,
        restore_best_weights=True
    )

    model_checkpoint = ModelCheckpoint(
        'best_model.h5',
        monitor='val_loss',
        save_best_only=True
    )

    # Train model
    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=epochs,
        batch_size=batch_size,
        callbacks=[early_stopping, model_checkpoint]
    )

    return model, history
```

Figure 28: model training process

During the training phase, the model is constructed using a sequential architecture, embedding layers, and bidirectional LSTM layers to capture previous and future context in the sequence of the API calls. The output layer has a sigmoid activation function in order to accommodate for multi-label classification and to make simultaneous predictions for multiple evasion techniques. The model is trained with a combination of callbacks such as early stopping and model checkpoints to monitor the validation loss and reduce overfitting. Early stopping means that training ceases if the performance of the model is not improving, and the best weights are restored, while the model checkpoint saves the best model obtained during the training phase.

The model is trained for a specified number of epochs, incorporating early stopping to prevent overfitting and model checkpoints to save the best model. This allows for optimal model selection based on the validation performance. When evaluating the model on the test set, we generated predictions, compared them to the ground truth labels, and calculated metrics such as accuracy and the classification report to assess the model's ability to detect the different types of sandbox evasion techniques.

2.1.4 Methodology Framework – Reinforcement Learning-Based Dynamic Sandbox Modification for Malware Behavior Analysis

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.

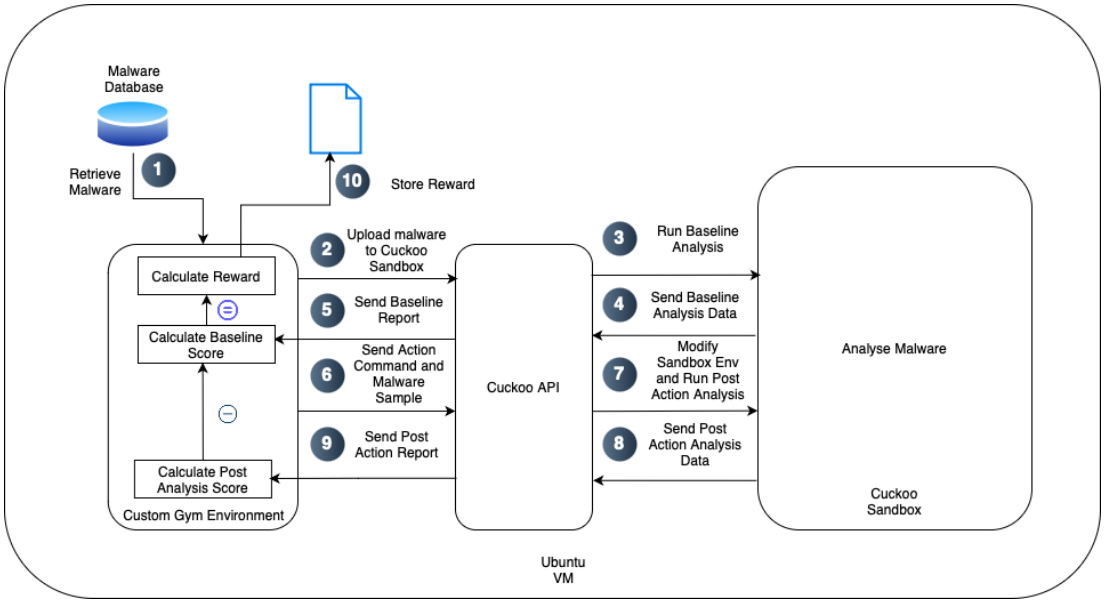


Figure 29.1: System Diagram

- **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.

- **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.

- **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.

- **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.

- **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.

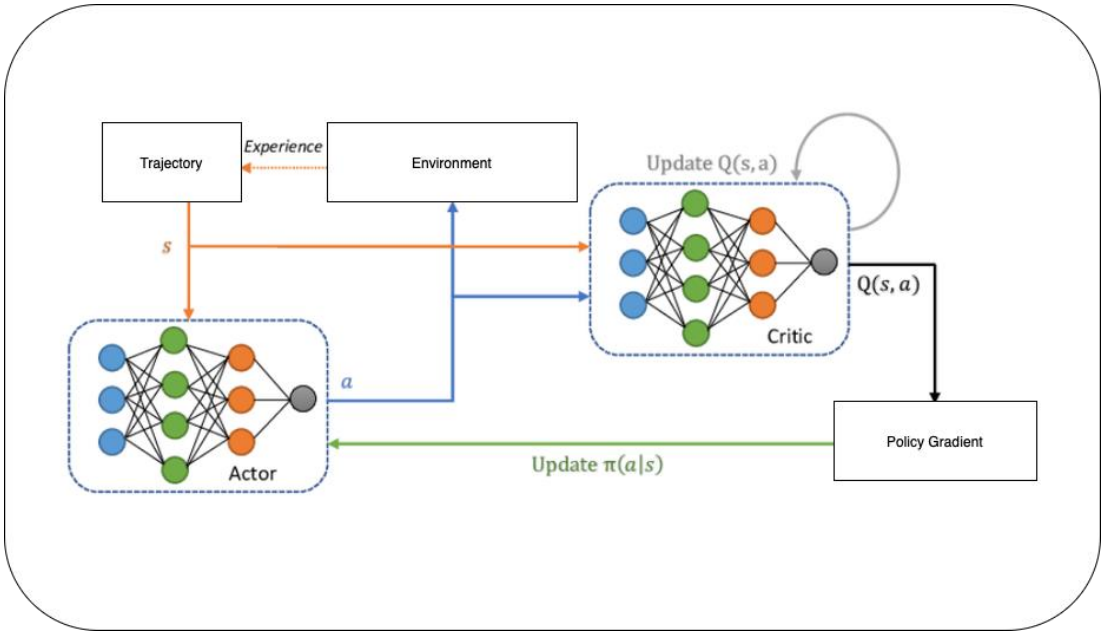


Figure 29.2: PPO Architecture [15]

2.2 Commercialization Aspect of the Product

2.2.1 Commercial value of project HyperAdapt

Project Hyperadapt presents a fresh method for sandbox hardening and malware detection by use of adversarial reinforcement learning. The capacity of the system to replicate real-time attacker-defender dynamics inside a self-adaptive sandbox environment defines its main value. Hyperadapt constantly develops by learning from adversarial input created using offensive reinforcement learning models, unlike traditional detection systems depending on static heuristics or predefined rules specified. This ability to grow personally helps HyperAdapt to be a crucial tool in contemporary cybersecurity architecture.

By combining offensive and defensive RL elements inside a sandbox structure, one creates a tight feedback loop whereby every evasion effort helps the defense system to develop. By reducing the reliance on manual signature updates and threat model tuning, this automated interaction cycle helps companies to stay up with ever more elusive malware strains. Technology like Hyperadapt provides a durable and intelligent solution as security operations centers face continuous pressure to change with the fast moving threat environments.

2.2.2 Core Market Opportunities

Hyper Adapt has commercial potential in many different fields where enhanced threat modeling, automated identification refinement, and adversarial learning are very crucial. Among the main uses is in the field of adversarial testing systems and threat simulation. Without depending on static malware datasets, security teams can utilize the system to assess how effectively their sandbox and detection pipelines withstand new, artificial intelligence-generated evasive behavior.

Additionally in line with enterprise-grade sandboxing solutions is the architecture. HyperAdapt improves classic sandbox technologies with adaptive behavior and real-time policy updates by including dynamic reinforcement learning elements. Security providers seeking to include intelligent defenses into their solutions might combine this architecture to stand out with proactive detection features.

Research and innovation settings where malware analysis, model training, or sandbox benchmarking is standard present still another possibility. HyperAdapt offers a versatile testbed allowing one to experiment with several evasion and detection strategies in a controlled loop. Finally, the system offers real-world-like scenarios where defenders encounter adaptive, non-deterministic attacks more closely resembling today's cyber operations, so enabling advanced analyst training possibilities.

2.2.3 Role and value of the offensive RL component

Though not intended for direct commercialization, the evasive RL agent is a necessary engine in the Hyperadapt system. Its aims are to expose blind areas in the detection logic and stress-test sandbox defenses in real time. By means of ongoing engagement with the sandbox, the agent learns to lower detection scores by means of optimal evasion sequences. Every evasion effort serves as a performance test that lets the system gauge the resilience of the sandbox over a broad spectrum of behavioral approaches.

Using this component to help the defensive RL agent's training magnifies its utility. The evasion model guarantees that the defense does not become overfitted to stationary behavior by offering a stream of varied, hostile samples. It helps the system to learn on the brink of failure, where most successful policy changes can take place. Moreover, the evasion component is valuable for internal evaluation or vendor testing since it can be used separately as a benchmarking tool to assess the robustness of several sandbox platforms.

Systems engineering-wise, this element also helps HyperAdapt to be scalable and modular. Its approach helps to integrate new evasion tactics and sandbox setups without changing the central training loop. This lets Hyperadapt be flexible and upgradable as both threats and analysis tools change.

2.2.4 Role and value of the user behavior simulation component

Aside from its sandbox hardening features, its user behavior simulation engine is of unique commercial value. Traditional sandbox products are often incapable of running context-sensitive malware because they cannot emulate human-like mouse movements, keyboard input, and user interface interaction. The tool fills an important gap in dynamic analysis by its ability to simulate natural mouse movements, keyboard input, and user interface interaction. Its value to users can be found in its ability to trigger malware that would otherwise go undetected and thus improve detection rates.

Sandboxing organizations can achieve considerable benefits from the inclusion of a layer mimicking human behavior. This inclusion reduces the incidence of false negatives associated with environment-aware malware that expects user interaction. In addition, threat intelligence teams are able to define multiple user profiles to study the effects of different behavioral patterns on malware execution. This flexibility is especially useful in APT emulation and targeted malware detection.

The user interaction module further positions itself to be an attractive standalone addition to commercially available sandbox solutions. Security vendors without the capability to simulate GUI-level interactions can utilize this module as an added value module to include with their products. As it operates through externally triggered scripts, integration can be made without significant alterations to current sandbox frameworks. This characteristic makes it an appealing choice to expedite commercialization with cooperative partnerships with vendors.

2.2.5 Role and value of the Hybrid Detection System for sandbox evasion techniques

The hybrid detection system for sandbox evasion techniques plays a vital role in improving malware analysis by combining the strengths of rule-based feature extraction and machine learning classification, specifically using LSTM networks. While not directly commercialized, this component is crucial in enhancing the detection capabilities of sandbox-based malware analysis systems. The rule-based system quickly identifies well-known evasion tactics, such as timing delays, anti-debugging checks, and environment verifications, through predefined patterns. The LSTM model, on the other hand, handles the sequential nature of malware behavior, capturing the temporal dependencies in API call sequences to identify evasions that are not covered by static rules. This dual approach ensures the detection of both known and unknown evasion techniques, making the system adaptable and scalable.

The value of the hybrid detection system is evident in its ability to effectively address both common and novel evasion techniques. The combination of rule-based detection and machine learning ensures that the system can quickly identify well-known evasion strategies while also learning from new behavioral patterns that may indicate previously unseen evasion tactics. The system's modular design allows it to scale, adapt, and integrate easily with other malware analysis tools, enhancing its utility in various cybersecurity applications. By continuously improving its predictive capabilities, this hybrid model is particularly valuable in real-time analysis scenarios, reducing the need for manual intervention and providing more reliable, automated detection of sandbox evasion techniques.

2.2.6 Role and Value of the Sandbox modification RL agent

RL sandbox modifier is tasked with dynamically altering the analyzing environment according to evading steps performed by samples of malware. It is an important component for deactivating sandbox evasions through runtime environmental

alteration measures. It assists the agent in acquiring more realistic behavioral cues from malware that, otherwise, are able to identify and evade static sandbox environments. System intelligence is achieved through assistance provided by reinforcement learning, which allows environmental modifications to be fine-tuned based upon feedback received through detection scores or behavioral results.

From a commercialization point of view, this capability is of huge worth to existing malware analysis platforms. Traditional sandbox platforms use static configurations, which are easily detectable and bypassable for modern malware. Integrating an adaptive RL component to these platforms improves them by making sandbox behavior non-deterministic and harder for malware to fingerprint. Security vendors can incorporate this as a pluggable functionality into platforms, which enhances detection and analytical results provided to customers.

The solution introduces operational efficiency into threat analysis pipelines as well. With fewer missed detections of sandbox-savvy malware, the system reduces false negatives and minimizes analyst effort. Commercial subscribers, including MSSPs, response teams, and SOC operators, can leverage this automation to increase throughput and improve net visibility to threats without compromising accuracy. From a monetization standpoint, RL agent can be sold as a premium solution for sandbox deployments, licensed according to user or endpoint quantities. It can be sold as part of a platform-as-a-service solution for malware analysis, with users being charged according to usage levels. Due to its modular design, the agent can be sold to several verticals, including cybersecurity research labs, large-enterprise security operation centers, and national CERTs, as an intelligent and scalable extension to what they already have for detecting malware.

2.2.7 Competitive Advantages and Differentiators

The most important advantage of HyperAdapt is its use of autonomous learning to sustain resilience against changing hazards. This system adjusts its policy depending on direct contact with hostile input, while conventional sandbox solutions identify malware depending on static signs. It can so guard against evasions not seen in the wild as yet.

The capacity of the framework to generate quantifiable, repeatable, and statistically supported assessment of sandbox strength is another important difference. Hyperadapt benchmarks the performance of other tools and quantifies its own evolution over time using detection score analysis and evasion incentive monitoring. This qualifies not only as a detecting mechanism but also as a validation and research tool.

At last, the modular and extendable character of the architecture enables HyperAdapt to be tailored for various operational surroundings. Applied in a research lab, business SOC, or product integration process, it may increase its capabilities while preserving its central adversarial learning engine. These elements make Hyperadapt a viable contender for commercialization in fields where cyber defense has to keep ahead of fast changing hazards.

2.3 Testing and Implementation

2.3.1 System Architecture Testing and Implementation - Offensive RL Component

Implementing a modular, step-by-step pipeline meant to facilitate the creation, integration, and execution of the RL-based malware evasion framework, the system architecture was set Completing 13 implementation phases overall allowed for smooth interaction between reinforcement learning, sandbox testing, and malware production. The approach started with environment setup including TensorFlow, Keras, MinGW, and pertinent Python libraries. Defining constants helped to regulate API connectivity and RL settings. Developed in C with behaviors meant to set off sandbox detection, the base malware proved.

Every evasion approach was applied as a stand-alone module arranged according to category. After that, these modules loaded into the system and arranged according to a class hierarchy. Built with a neural network backend, a DQN agent was trained to understand evasive patterns via iterative error.

After that, the system was connected using a main RL controller and malware modifying component, so automating the code generation, compilation, submission, analysis process. Along with support features for training orchestration and evasion testing, API connectivity to Cuckoo Sandbox was extensively tested.

Key Implementation steps

Step1: Environment Setup and Dependency Installation

By establishing and importing all relevant dependencies and building the necessary directory structure, this first step lays the groundwork for the DQN-based Cuckoo Sandbox evasion project. TensorFlow and

Keras for the deep learning components, requests for API connectivity, **numpy** for numerical calculations, and matplotlib for visualizing comprise the environment set-up. Including mingw-w64 shows cross-compilation of Windows executables, which is fitting given Cuckoo Sandbox evasion as Cuckoo mostly examines Windows malware.

Three significant directories are created by the code: **saved_executables**" to hold produced binary files, "**saved_evasions**" to save successful evasion techniques or results, and "checkpoints" to save model states all around training. By segregating several kinds of artifacts generated throughout the reinforcement learning process, this company exhibits good software engineering methods. These libraries and directory structure together exhibit readiness of a strong environment for designing, training, and assessing a DQN agent learning to avoid malware detection in the Cuckoo Sandbox.

```
# Cuckoo Sandbox Evasion using Reinforcement Learning

## Step 1: Setup Environment and Dependencies

# Install required packages
!pip install tensorflow keras requests numpy matplotlib

# For mingw installation (Windows cross-compilation)
!apt-get update && apt-get install -y mingw-w64

# Import necessary libraries
import numpy as np
import tensorflow as tf
from tensorflow import keras
from collections import deque
import random
import requests
import json
import time
import os
import re
import subprocess
import matplotlib.pyplot as plt
from datetime import datetime

# Create required directories
os.makedirs("saved_executables", exist_ok=True)
os.makedirs("saved_evasions", exist_ok=True)
os.makedirs("checkpoints", exist_ok=True)

print("Setup complete!")
```

Figure *30*: Importing and required libraries (Step 01)

Step 2: Constants and Configuration Parameters

``

This stage defines the fundamental characteristics of the interaction with the Cuckoo Sandbox environment and the reinforcement learning model. Reinforcement learning hyperparameters, API configuration, operational parameters, and file system organization include many functional groupings among the defined constants.

The reinforcement learning parameters define the behavior of the DQN model including a memory size of 10,000 experiences, batch training on 32 samples, a discount factor of 0.95 for future rewards, and an epsilon-greedy exploration strategy starting at 1.0 with decay to promote exploration early and exploitation later. Along with unique headers to evade browser warnings, API setup consists in ngrok-tunneled endpoints for uploading malware samples and receiving analysis reports. With a maximum wait duration of 20 minutes and regular monitoring every 10 seconds, the operational parameters comprise timeout values to control the asynchronous character of malware analysis. Consistent storing and retrieval of files during the tests is guaranteed by directory paths and naming standards. During development and debugging especially, the use of a simulation flag lets one test without actually making real API calls.

```
## Step 2: Define Constants

# RL model parameters
MAX_MEMORY_SIZE = 10000
BATCH_SIZE = 32
GAMMA = 0.95  # Discount factor
EPSILON = 1.0  # Exploration rate
EPSILON_MIN = 0.01
EPSILON_DECAY = 0.995
LEARNING_RATE = 0.001

# API configuration
UPLOAD_URL = "https://6af7-112-134-198-135.ngrok-free.app/upload"
REPORT_BASE_URL = "https://9cb4-112-134-198-135.ngrok-free.app/api/reports/"

# Ngrok bypass headers
NGROK_HEADERS = {
    'ngrok-skip-browser-warning': 'true',
    'User-Agent': 'CuckooRL-Agent'
}

# Task ID to start from (since task ID 1 is already used for base malware)
STARTING_TASK_ID = 2

# Wait timeouts
MAX_WAIT_TIME = 1200  # 20 minutes in seconds
CHECK_INTERVAL = 10   # Check every 10 seconds

# Optional: Set to True to use simulation instead of real API
USE_SIMULATION = False

# Checkpoint directory
CHECKPOINT_DIR = "checkpoints"
os.makedirs(CHECKPOINT_DIR, exist_ok=True)

# Directory to save malware samples
MALWARE_SAMPLES_DIR = "/home/cuckoo/malware_samples/"

# Malware naming configuration for auto_submit.py compatibility
MALWARE_BASE_NAME = "trojan"  # Base name for all malware files

print("Constants defined!")
```

Figure *31*: Constants and Configuration Parameters

Step 3: Base Malware Creation and defined

This stage concentrates on building the base malware sample meant for modification via the reinforcement learning process to avoid Cuckoo Sandbox detection. The code creates a C application with several harmful actions that security systems usually find easily recognized. To guarantee single instance execution, the base malware combines numerous techniques: PE header alteration, file manipulations, process injection targeting notepad.exe, network connectivity to public DNS servers, Windows registry modifications for persistence, and mutex generation.

Separate functions representing different detection paths for the sandbox help to arrange the malicious capability. Changing PE headers to possibly corrupt file analysis, generating and deleting temporary files to establish presence on the filesystem, injecting code into legitimate processes to hide execution, performing network activity to contact command and control servers, establishing persistence through registry changes, and creating mutexes to prevent multiple infections each function implements a common technique used by real malware. Clear targets for the DQN agent to change when learning evasion techniques are given by this modular architecture. Saving this code to a file called "**base_malware.c**" helps the system get ready for later compilation and Cuckoo Sandbox initial detection baseline.

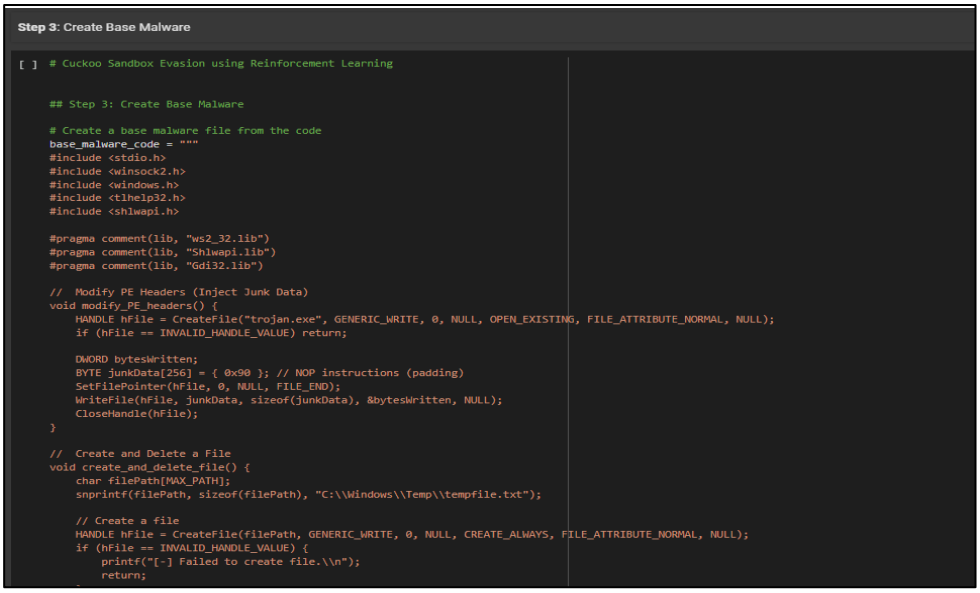


Figure *32*: Base Malware Creation

Step 4: Define Evasion Technique Classes

This stage specifies an object-oriented structure for arranging and using several sandbox evasion methods. Establishing a class hierarchy with a base Evasion Technique class and four specialized subclasses separating various evasion techniques timing-based evasions, file

system checks, human behavior modeling, and anti-debugging techniques the code. While keeping a uniform interface for the reinforcement learning agent to interact with, this design offers a disciplined means to capture the implementation specifics of every evasion technique.

Along with techniques to access common items, the base Evasion Technique class specifies name, code template, needed headers, and libraries. With conditional logic to leave should a sandbox environment be discovered, each subclass uses a specialized **get_main_call()** method to provide the suitable code for calling the technique from the main function of the virus. This design concept lets the produced malware samples have modular construction of evasion strategies. Especially, the **exit_on_detection** value gives the reinforcement learning agent choices for creating more complex evasion strategies that might involve combinations of detections and non-detections - either by immediately terminating execution or continuing despite detection.

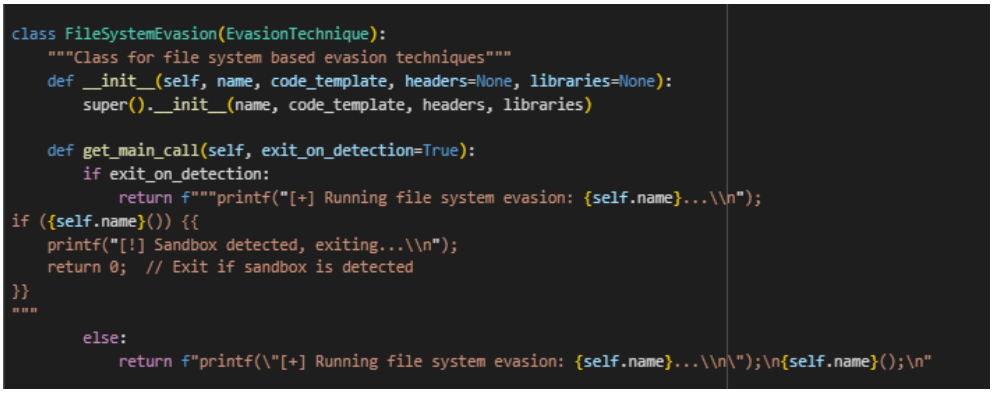
.

```python
class EvasionTechnique:
    """Base class for all evasion techniques"""
    def __init__(self, name, code_template, headers=None, libraries=None):
        self.name = name
        self.code_template = code_template
        self.headers = headers or []
        self.libraries = libraries or []

    def get_function_definition(self):
        return self.code_template

    def get_headers(self):
        return self.headers

    def get_libraries(self):
        return self.libraries

    def get_main_call(self, exit_on_detection=True):
        """Generate code to call this technique from main()"""
        raise NotImplementedError("Subclasses must implement this")
```

Figure *33*: Defining EvasionTechnique Class

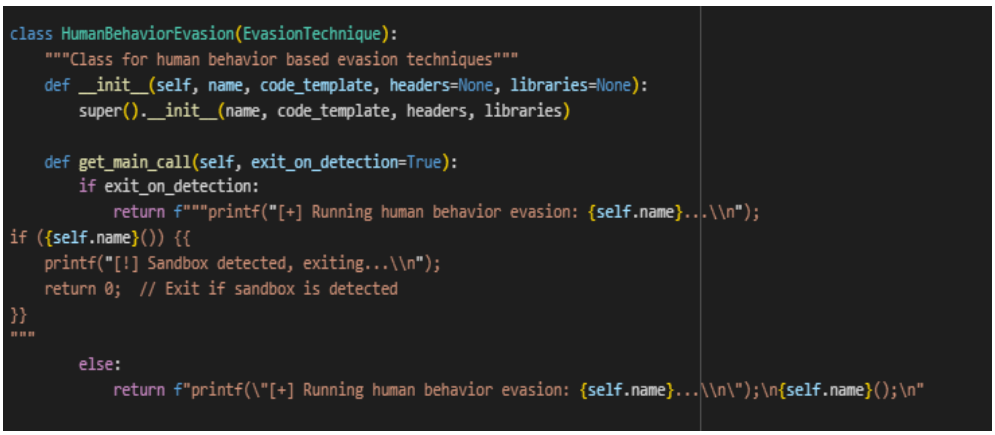
```
class TimingEvasion(EvasionTechnique):
    """Class for timing-based evasion techniques"""
    def __init__(self, name, code_template, delay=5000, headers=None, libraries=None):
        super().__init__(name, code_template, headers, libraries)
        self.delay = delay

    def get_main_call(self, exit_on_detection=True):
        return f"printf(\"[+] Running timing evasion: {self.name}...\\n\");\n{self.name}({self.delay});\n"
```

Figure *34*: Defining TimingEvasion Class



```
class FileSystemEvasion(EvasionTechnique):
    """Class for file system based evasion techniques"""
    def __init__(self, name, code_template, headers=None, libraries=None):
        super().__init__(name, code_template, headers, libraries)

    def get_main_call(self, exit_on_detection=True):
        if exit_on_detection:
            return f"""printf("[+] Running file system evasion: {self.name}...\\n");
if ({self.name}()) {{
    printf("[!] Sandbox detected, exiting...\\n");
    return 0;  // Exit if sandbox is detected
}}
"""
        else:
            return f"printf(\"[+] Running file system evasion: {self.name}...\\n\");\n{self.name}();\n"
```

Figure *35*: Defining FileSystemEvasion Class



```
class HumanBehaviorEvasion(EvasionTechnique):
    """Class for human behavior based evasion techniques"""
    def __init__(self, name, code_template, headers=None, libraries=None):
        super().__init__(name, code_template, headers, libraries)

    def get_main_call(self, exit_on_detection=True):
        if exit_on_detection:
            return f"""printf("[+] Running human behavior evasion: {self.name}...\\n");
if ({self.name}()) {{
    printf("[!] Sandbox detected, exiting...\\n");
    return 0;  // Exit if sandbox is detected
}}
"""
        else:
            return f"printf(\"[+] Running human behavior evasion: {self.name}...\\n\");\n{self.name}();\n"
```

Figure *36*: Defining HumanBehaviourEvasion Class

Step 5: Load Evasion techniques

This stage uses a thorough set of concrete evasion methods to be applied by the DQN agent to produce malware variants able of avoiding Cuckoo Sandbox detection. Eleven different evasion tactics arranged into three categories timing-based techniques (T1-T4), file system inspections (F1-F4), and human behavior simulations (H1-H3) the code specifies. With particular code templates, necessary headers, and library dependencies, every technique is manifested as an object of the suitable class created in the previous stage.

In sandbox contexts, the timing-based evasions identify accelerating time via network delays, multimedia timers, **waitable** timers, and parallel thread timing checks among other approaches. These methods take use of automated analytic environments' inclination to hasten labor-intensive activities. The file system evasions concentrate on identifying virtualization components by counting system drivers and looking for VMware and VirtualBox-specific files and directories, hence spotting **virtualisation** artifacts. Usually lacking in automated analysis environments, the human behavior simulations identify sandbox environments by tracking for indicators of human activity such mouse movement, keyboard input, and the existence of recently opened papers. Every method incorporates thorough logging to monitor its performance and detection outcomes, therefore offering useful input for the reinforcement learning process. Programmatic selection and use of evasion techniques are made possible by the function returning a dictionary mapping technique identification to their implementations.

```
def load_evasion_techniques():
    """Load all available evasion techniques"""
    evasion_techniques = []

    # Timing evasions (T)
    T1 = TimingEvasion(
        name="timing_network_delay",
        code_template="""
void timing_network_delay(DWORD delay) {
    int iResult;
    DWORD timeout = delay; // delay in milliseconds
    DWORD OK = TRUE;
    SOCKADDR_IN sa = { 0 };
    SOCKET sock = INVALID_SOCKET;

    // Initialize Winsock if not already done
    WSADATA wsaData;
    WSAStartup(MAKEWORD(2, 2), &wsaData);

    // This code snippet should take around Timeout milliseconds
    do {
        memset(&sa, 0, sizeof(sa));
        sa.sin_family = AF_INET;
        sa.sin_addr.s_addr = inet_addr("8.8.8.8"); // Route to this IP address
        sa.sin_port = htons(80); // Should not be able to connect to this port
```

Figure *37*: Adding Timing Evasion Code Snippets

```
    # File system evasions (F)
    F1 = FileSystemEvasion(
        name="vmware_file_check",
        code_template="""
BOOL vmware_file_check() {
    TCHAR* szPaths[] = {
        _T("system32\\\\drivers\\\\vmmouse.sys"),
        _T("system32\\\\drivers\\\\vmhgfs.sys"),
    };

    TCHAR szWinDir[MAX_PATH] = _T("");
    TCHAR szPath[MAX_PATH] = _T("");
    GetWindowsDirectory(szWinDir, MAX_PATH);

    printf("[*] Checking for VMware files...\\n");

    for (int i = 0; i < 2; i++) {
        PathCombine(szPath, szWinDir, szPaths[i]);
        printf("[*] Checking %s\\n", szPath);

        DWORD dwAttrib = GetFileAttributes(szPath);
        if (dwAttrib != INVALID_FILE_ATTRIBUTES && !(dwAttrib & FILE_ATTRIBUTE_DIRECTORY)) {
            printf("[!] VMware file detected: %s\\n", szPath);
            return TRUE;
        }
    }
```

Figure *38*: Adding File System Evasion Code Snippets

```
    evasion_techniques.append(("H2", H2))

    # Add H3: Check for recently opened documents
    H3 = HumanBehaviorEvasion(
        name="recent_docs_check",
        code_template="""
BOOL recent_docs_check() {
    printf("[*] Checking for recently opened documents...\\n");

    HKEY hKey;
    DWORD cValues;
    DWORD dwRet;

    if (RegOpenKeyEx(HKEY_CURRENT_USER, TEXT("Software\\\\Microsoft\\\\Windows\\\\CurrentVersion\\\\Explorer\\\\RecentDocs"),
                     0, KEY_READ, &hKey) == ERROR_SUCCESS) {

        // Get number of values
        if (RegQueryInfoKey(hKey, NULL, NULL, NULL, NULL, NULL, NULL, &cValues, NULL, NULL, NULL, NULL) == ERROR_SUCCESS) {
            printf("[*] Found %d recently opened documents\\n", cValues);

            if (cValues < 3) {
                printf("[!] Less than 3 recent documents - potential sandbox\\n");
                RegCloseKey(hKey);
                return TRUE; // Sandbox detected
            }
        }

        RegCloseKey(hKey);
    }

    printf("[*] Recent documents check passed\\n");
    return FALSE; // No sandbox detected
}""",
        headers=["<windows.h>"],
        libraries=[]
    )
    evasion_techniques.append(("H3", H3))

    return dict(evasion_techniques)
```

Figure *39*: Adding File System Evasion Code Snippets

Step 6: Cuckoo Sandbox Interface Implementation

stage defines a complete interface for communicating with the Cuckoo Sandbox environment, therefore offering both a real API client and a development and testing simulation mode. Two primary classes are defined in the code: **SimulatedCuckoo** Interface, which **replics** the behavior of the actual system for offline development, and Cuckoo Interface for real-world interaction with Cuckoo Sandbox API endpoints. Both systems guarantee constant functioning independent of the running mode by using the same interface architecture.

Including uploading executable files with appropriate naming conventions, getting analysis reports, and extracting detection scores, the Cuckoo Interface class oversees the whole lifetime of malware sample analysis. Comprehensive error handling, logging, and debugging tools abound in the implementation to record and document problems during the API contact process. Notable characteristics include recursive score extraction logic to manage several report structures, URL management for the **ngrok-tunneled** endpoints, and wait mechanisms with timeouts to manage the asynchronous character of sandbox analysis. Capture timestamps, request and response data, status codes, and response times; the bundled **APILogger** class offers thorough recording of all API transactions for audit and debugging needs. With customizable effectiveness levels for every strategy to replicate alternative outcomes, the **SimulatedCuckoo** Interface generates realistic report formats and scores based on applied evasion techniques, therefore offering development-friendly capability.

```
class CuckooInterface:
    """Class to interface with Cuckoo Sandbox through custom API endpoints"""

    def __init__(self):
        # Set the API endpoints
        self.upload_url = UPLOAD_URL
        self.report_base_url = REPORT_BASE_URL
        self.logger = APILogger("cuckoo_api.log")
        self.current_task_id = STARTING_TASK_ID

    def update_endpoints(self, upload_url=None, report_base_url=None):
        """Update API endpoints if ngrok URLs expire"""
        if upload_url:
            self.upload_url = upload_url
            print(f"Upload URL updated to: {upload_url}")
        if report_base_url:
            self.report_base_url = report_base_url
            print(f"Report base URL updated to: {report_base_url}")

    def get_next_task_id(self):
        """Get the next task ID to use"""
        task_id = self.current_task_id
        self.current_task_id += 1
        return task_id

    def save_malware_file(self, executable_content, task_id=None):
        """Upload the executable to the Cuckoo API endpoint"""
        if task_id is None:
            task_id = self.get_next_task_id()

        # Format filename for auto_submit.py compatibility
        malware_name = MALWARE_BASE_NAME
        evasion_method = f"task{task_id}"
        iteration = f"{task_id:03d}"
        filename = f"{malware_name}_{evasion_method}_{iteration}.exe"
```

Figure *40*: Defining Cuckoo Sandbox Interface

```
# Calculate score based on evasions if they exist
if task["evasions"]:
    report["data"]["score"] = simulate_cuckoo_score(task["evasions"])

self.logger.log_api_call(
    endpoint=f"SIMULATED/reports/{task_id}",
    method="GET",
    status_code=200,
    response_time=0.3,
    request_data={"task_id": task_id},
    response_data={"score": report["data"]["score"]}
)

return report
```

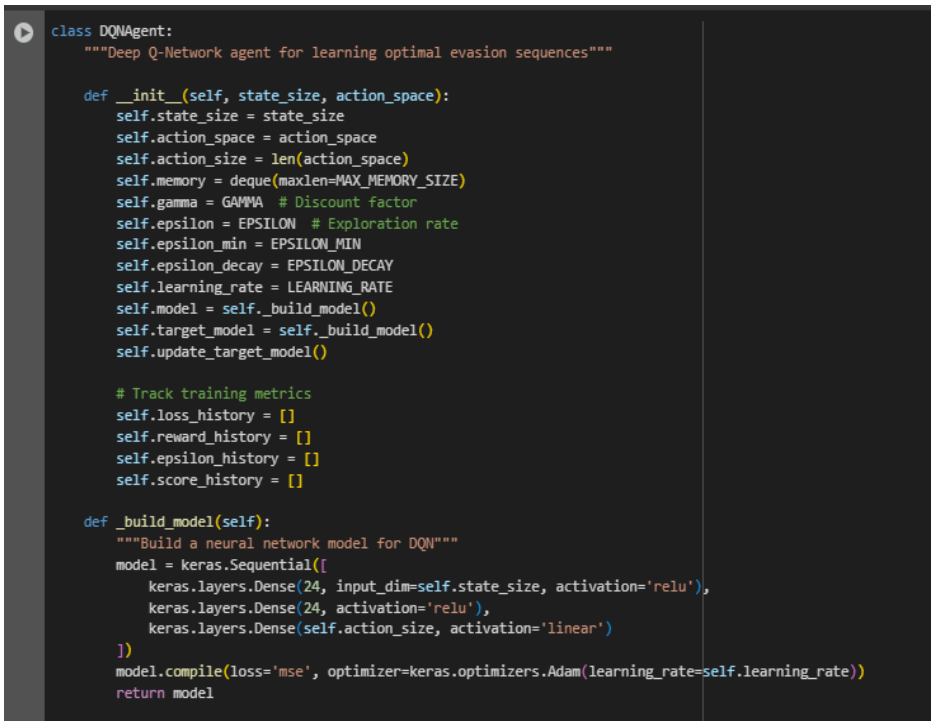
Figure *41*: Defining Reward Calculation Mechanism

Step 7: DQN Agent Implementation

This stage implements the Deep Q-Network (DQN) agent in charge of learning optimal evasion strategies, hence implementing the fundamental reinforcement learning component. Standard DQN architecture with a neural network model for Q-value approximation, a target network for stable learning, experience replay memory, and an epsilon-greedy exploration approach is followed in implementation with several important components. To follow the learning development, the code guarantees thorough monitoring of training statistics.

Using **Keras** with two hidden layers of 24 neurons each with **ReLU** activation functions and a linear output layer matching action values, the **DQNAgent** class builds the neural network model. The agent preserves two identical network architectures: the target model for reliable Q-value estimate during learning and the main model for action selection. Experience replay enhances learning stability by means of a fixed-size memory buffer storing state transitions, therefore helping to

break correlations between successive training samples. Selecting random actions with probability epsilon, which progressively decays over time to move from exploration to exploitation, the epsilon-greedy strategy balances exploration and exploitation. Comprehensive techniques for loading and saving model weights are part of the implementation, therefore allowing both persistence across sessions and pre-trained model deployment capability. With visualization tools to assess agent learning, performance metrics tracking records loss values, rewards, epsilon decay, and Cuckoo scores throughout training.



```python
class DQNAgent:
    """Deep Q-Network agent for learning optimal evasion sequences"""

    def __init__(self, state_size, action_space):
        self.state_size = state_size
        self.action_space = action_space
        self.action_size = len(action_space)
        self.memory = deque(maxlen=MAX_MEMORY_SIZE)
        self.gamma = GAMMA  # Discount factor
        self.epsilon = EPSILON  # Exploration rate
        self.epsilon_min = EPSILON_MIN
        self.epsilon_decay = EPSILON_DECAY
        self.learning_rate = LEARNING_RATE
        self.model = self._build_model()
        self.target_model = self._build_model()
        self.update_target_model()

        # Track training metrics
        self.loss_history = []
        self.reward_history = []
        self.epsilon_history = []
        self.score_history = []

    def _build_model(self):
        """Build a neural network model for DQN"""
        model = keras.Sequential([
            keras.layers.Dense(24, input_dim=self.state_size, activation='relu'),
            keras.layers.Dense(24, activation='relu'),
            keras.layers.Dense(self.action_size, activation='linear')
        ])
        model.compile(loss='mse', optimizer=keras.optimizers.Adam(learning_rate=self.learning_rate))
        return model
```

Figure *42*: Defining DQN Agent

Step 8: Implementation of Malware Modifier

This stage uses the **MalwareModifier** class to act as the link between the real malware code manipulation and the reinforcement learning agent. Applying certain evasion strategies to the basic malware,

controlling code dependencies, and assembling the altered source into Windows executables falls to the class. This part converts the agent's abstract choices into actual malware variants with particular evasion capacity.

With great regard for dependencies and appropriate code structure, the **MalwareModifier** manages the difficult chore of code modification. While guaranteeing correct ordering (e.g., putting winsock2.h before **windows.h** to avoid conflicts), it extracts and maintains needed headers, libraries, function definitions, and main function calls from certain evasion tactics. The implementation covers edge scenarios like CSIDL constants that might be required by some approaches and offers special handling for Windows-specific locations and registry values by appropriately escaping backslashes. With thorough error handling and diagnostic output, the MinGW cross-compiling tool for compiling codes from non-Windows environments requires To further the general development of the system, the class additionally runs a simulation mode that lets one test without actual compilation. This thorough attention to code generating details guarantees that the resultant malware variants incorporate the chosen evasion strategies and

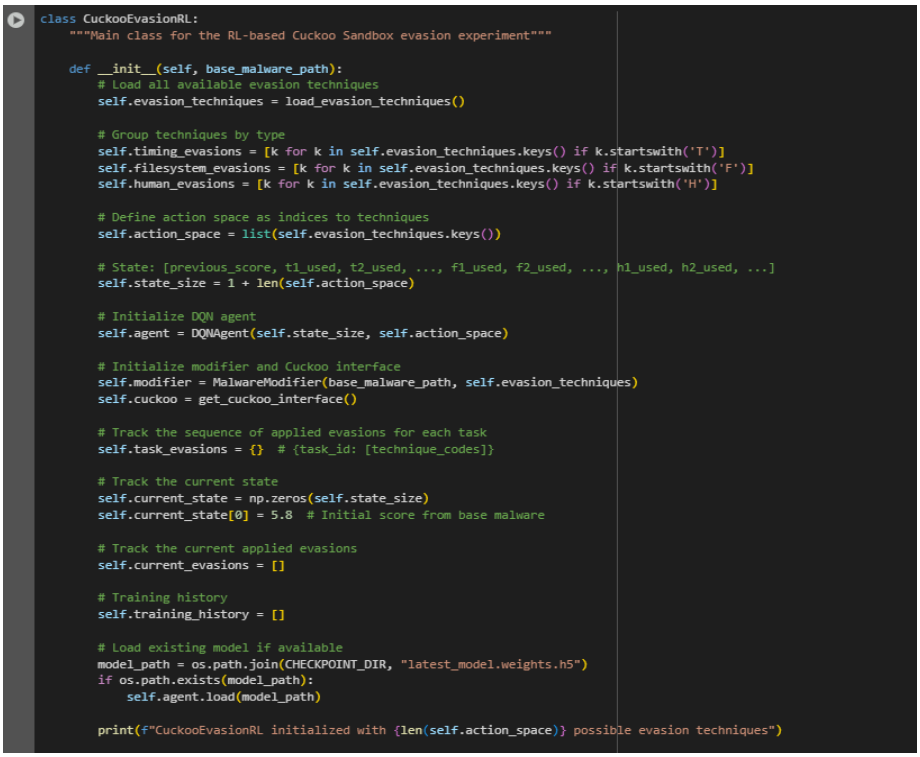
```
class MalwareModifier:
    """Class to modify malware with evasion techniques"""

    def __init__(self, base_malware_path, evasion_techniques):
        self.base_malware_path = base_malware_path
        self.evasion_techniques = evasion_techniques
        self.compiler_path = "x86_64-w64-mingw32-gcc"  # Windows cross-compiler

    def apply_evasion_techniques(self, technique_codes):
        """Apply the specified evasion techniques to the base malware"""
        # Read the base malware code
        with open(self.base_malware_path, 'r') as file:
            base_code = file.read()

        # Collect all headers and libraries needed
        headers = set()
        libraries = set()
        function_definitions = []
        main_calls = []

        for technique_code in technique_codes:
            if technique_code in self.evasion_techniques:
                technique = self.evasion_techniques[technique_code]

                # Add headers
                headers.update(technique.get_headers())

                # Add libraries
                libraries.update(technique.get_libraries())

                # Add function definitions
                function_definitions.append(technique.get_function_definition())

                # Add main calls
                main_calls.append(technique.get_main_call())

        # Sort headers to ensure winsock2.h comes before windows.h
        sorted_headers = sorted(list(headers), key=lambda h: 0 if "winsock2.h" in h else 1)

        # Add special includes for shell functions if needed
        if any("SHGetSpecialFolderPath" in technique.get_function_definition() for technique in [self.evasion_techniques[code] for code in technique_codes]):
            if "<shlobj.h>" not in sorted_headers:
                sorted_headers.append("<shlobj.h>")  # For CSIDL constants and SHGetSpecialFolderPath

        # Insert headers at the beginning of the file
        headers_text = "\n".join([f"#include {header}" for header in sorted_headers])
```

preserve grammatical accuracy.

Figure *43*: Defining Malware Modifier

Step 9: Implementation of Main Reinforcement Learning Controller

This phase runs the central controller class, **CuckooEvasionRL,** which coordinates the whole reinforcement learning process for creating successful sandbox evasion techniques. By means of testing and learning, the controller combines all previously described components into a cohesive system that iteratively enhances malware evasion capacity. Maintaining state representation, tracking performance measures, and guaranteeing constraint satisfaction during technique selection, the class controls the interactions among the DQN agent, malware modification agent, and Cuckoo interface components.



```
class CuckooEvasionRL:
    """Main class for the RL-based Cuckoo Sandbox evasion experiment"""

    def __init__(self, base_malware_path):
        # Load all available evasion techniques
        self.evasion_techniques = load_evasion_techniques()

        # Group techniques by type
        self.timing_evasions = [k for k in self.evasion_techniques.keys() if k.startswith('T')]
        self.filesystem_evasions = [k for k in self.evasion_techniques.keys() if k.startswith('F')]
        self.human_evasions = [k for k in self.evasion_techniques.keys() if k.startswith('H')]

        # Define action space as indices to techniques
        self.action_space = list(self.evasion_techniques.keys())

        # State: [previous_score, t1_used, t2_used, ..., f1_used, f2_used, ..., h1_used, h2_used, ...]
        self.state_size = 1 + len(self.action_space)

        # Initialize DQN agent
        self.agent = DQNAgent(self.state_size, self.action_space)

        # Initialize modifier and Cuckoo interface
        self.modifier = MalwareModifier(base_malware_path, self.evasion_techniques)
        self.cuckoo = get_cuckoo_interface()

        # Track the sequence of applied evasions for each task
        self.task_evasions = {}  # {task_id: [technique_codes]}

        # Track the current state
        self.current_state = np.zeros(self.state_size)
        self.current_state[0] = 5.8  # Initial score from base malware

        # Track the current applied evasions
        self.current_evasions = []

        # Training history
        self.training_history = []

        # Load existing model if available
        model_path = os.path.join(CHECKPOINT_DIR, "latest_model.weights.h5")
        if os.path.exists(model_path):
            self.agent.load(model_path)

        print(f"CuckooEvasionRL initialized with {len(self.action_space)} possible evasion techniques")
```

Figure *44*: Defining Mian RL Controller

The training method is episodic, with many rounds of malware modification, submission, and feedback collecting in every episode. Based on its present policy, the agent chooses an evasion technique for each iteration; validates it against pre-defined constraints (preventing technique duplication and enforcing category limits), applies the technique to the base malware, compiles the modified code, submits it to Cuckoo Sandbox, and compiles rewards based on score reduction. To provide resilience, the solution comprises thorough error handling for compilation failures, API mistakes, and timeout situations. With periodic visualization to measure learning, performance tracking records comprehensive metrics at every level—including rewards, losses, scores, and applied approaches. Through file cleanup, checkpoint saving, and best solution tracking the class also effectively manages artifacts. Additional methods for study are top evasion sequence identifying and technique effectiveness visualizing.

Step 10: API Connection Testing

Before starting the reinforcement learning process, this stage serves a diagnostic purpose verifying connectivity to the Cuckoo Sandbox API endpoints. Early in the process, the **test_api_connection** method acts as a basic check to make sure the system can interact with the upload and report endpoints of the API, therefore helping to spot and fix connectivity problems.

By use of a number of pragmatic tests, the implementation guarantees API connectivity. First it verifies the HTTP response status and the JSON structure of the provided data to guarantee it includes the anticipated score field by requesting data for a known task ID. It then creates a basic text file and tries to send it to the Cuckoo Sandbox to test the upload endpoint, therefore verifying that the submission

process runs as intended. To help solve connection issues, the feature features thorough error reporting covering response status codes, content previews, and exception data. Consistent with the flexibility exhibited across the codebase, a simulation mode bypass is included to enable development and testing without needing an active Cuckoo Sandbox instance.

```
## Step 10: Test the API Connection (Optional)

def test_api_connection():
    """Test the connection to the Cuckoo API endpoints"""
    if USE_SIMULATION:
        print("Using simulation mode. Skipping API connection test.")
        return True

    print("Testing Cuckoo API connection...")

    # Test the report endpoint
    test_task_id = 1  # Use a known task ID for testing
    try:
        response = requests.get(f"{REPORT_BASE_URL}{test_task_id}", headers=NGROK_HEADERS)
        if response.status_code == 200:
            print(f"✓ Successfully connected to report API endpoint")
            print(f"  Sample response: {response.text[:100]}...")

            # Try to parse JSON response
            try:
                report_json = response.json()
                if "data" in report_json and "score" in report_json["data"]:
                    print(f"  Report contains a score: {report_json['data']['score']}")
                else:
                    print("  Report structure doesn't match expected format.")
                    print(f"  Report keys: {list(report_json.keys())}")
                    if "data" in report_json:
                        print(f"  Data keys: {list(report_json['data'].keys())}")
            except json.JSONDecodeError:
                print("  Response is not valid JSON.")
                print(f"  Response content: {response.text[:200]}...")
                return False
        else:
            print(f"✗ Failed to connect to report API. Status code: {response.status_code}")
            print(f"  Response: {response.text}")
            return False
    except requests.exceptions.RequestException as e:
        print(f"✗ Error connecting to report API: {e}")
        return False

    # Test the upload endpoint with a simple file
    try:
        # Create a simple test file
        test_file_path = "api_test_file.txt"
        with open(test_file_path, "w") as f:
            f.write("This is a test file for API connection verification.")

        # Try to upload it
        with open(test_file_path, "rb") as f:
            files = {"file": ("api_test_file.txt", f)}
            response = requests.post(UPLOAD_URL, files=files, headers=NGROK_HEADERS)
```

Figure *45*: Implementing API Connection within the RL Model

Step 11: Evasion Combination Testing

Before starting the whole reinforcement learning process, this stage performs a testing role intended to assess the efficiency of different combinations of evasion techniques. The purpose methodically evaluates pre-defined combinations of evasion strategies and their effects on the Cuckoo Sandbox detection score, therefore generating useful baseline data on approach efficacy both separately and in concert.

Nine alternative combinations from the unmodified base malware to complicated combinations of timing, file system, and human behavior evasion techniques are evaluated in implementation. The function uses the chosen techniques on the underlying malware for every combination, generates the altered code, sends it to Cuckoo Sandbox for examination, and notes the resultant detection score. The method manages both real-world API and simulation modes, therefore enabling testing in development environments without active Cuckoo instance. With the base malware score shown as a reference line, results are shown both as text output and as a visualization demonstrating the relative efficacy of every combination. By means of this methodical approach to testing, significant understanding of which approaches and combinations would be most favorable for the reinforcement learning agent to investigate, hence possibly quickening the learning process by pointing up suitable starting points.

```
## Step 11: Simple Testing Function

def test_evasion_combinations():
    """Test a few evasion combinations to see their effect on the Cuckoo score"""
    print("Testing a few evasion combinations...")

    # Load evasion techniques
    techniques = load_evasion_techniques()

    # Create appropriate interfaces
    cuckoo = get_cuckoo_interface()
    modifier = MalwareModifier("base_malware.c", techniques)

    # Test combinations
    test_combinations = [
        [],                         # Base malware (no evasions)
        ["T1"],                     # Single timing evasion
        ["F1"],                     # Single filesystem evasion
        ["H1"],                     # Single human behavior evasion
        ["T1", "F1"],               # Timing + Filesystem
        ["T1", "H1"],               # Timing + Human behavior
        ["F1", "H1"],               # Filesystem + Human behavior
        ["T1", "F1", "H1"],         # One of each type
        ["T2", "F1", "F2", "H1"]    # Multiple types with constraints
    ]
```

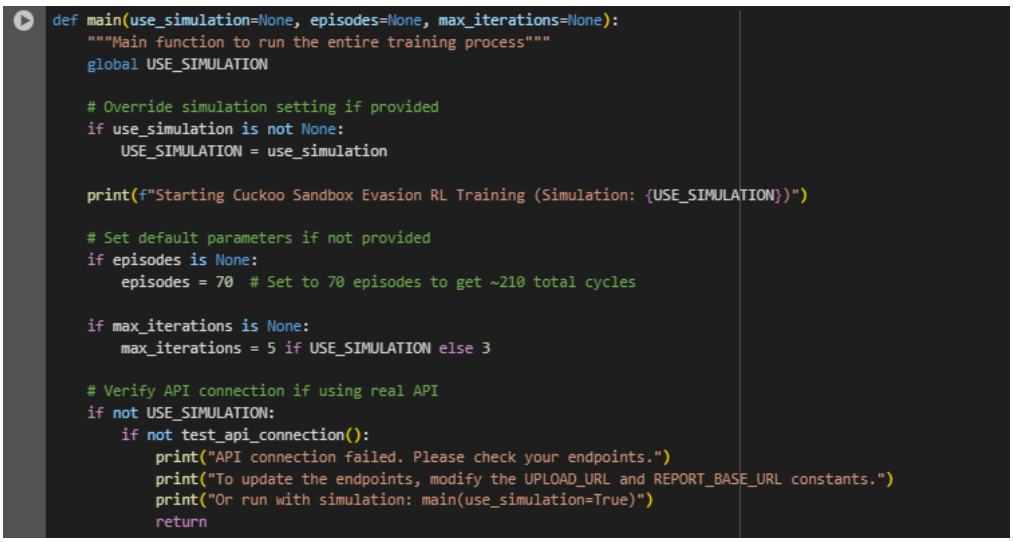
Figure *46*: Defining Evasion Combination Testing

Step 12: Main Training Function Implementation

This stage carries out the primary entrance point role that coordinates the whole Cuckoo Sandbox evasion reinforcement learning process. Coordinating all system components, managing setup parameters, verifying dependencies, running the training process, and presenting the final results in an all-encompassing way mostly serves this purpose.

With reasonable defaults that fit the operating environment, the implementation offers flexibility through customizable parameters for simulation mode, episode count, and iteration restrictions. Operating with the genuine Cuckoo Sandbox API, the function first checks connections to make sure training can go without interruption and, should connection problems be found, provides suitable fallback recommendations. Building the **CuckooEvasionRL** controller and running its training approach with the given parameters starts the training process. When completed, the function generates

thorough training metrics including the best evasion configuration found, reward statistics, and loss progression. Many charts displaying evasion method efficacy and learning evolution over time help to visualize the outcomes. Providing a synopsis of all artifacts produced throughout the training process including file paths to the training history, visualizations, model checkpoints, and stored evasion variants the function ends.



```python
def main(use_simulation=None, episodes=None, max_iterations=None):
    """Main function to run the entire training process"""
    global USE_SIMULATION

    # Override simulation setting if provided
    if use_simulation is not None:
        USE_SIMULATION = use_simulation

    print(f"Starting Cuckoo Sandbox Evasion RL Training (Simulation: {USE_SIMULATION})")

    # Set default parameters if not provided
    if episodes is None:
        episodes = 70  # Set to 70 episodes to get ~210 total cycles

    if max_iterations is None:
        max_iterations = 5 if USE_SIMULATION else 3

    # Verify API connection if using real API
    if not USE_SIMULATION:
        if not test_api_connection():
            print("API connection failed. Please check your endpoints.")
            print("To update the endpoints, modify the UPLOAD_URL and REPORT_BASE_URL constants.")
            print("Or run with simulation: main(use_simulation=True)")
            return
```

Figure *47*: Defining Main Training Function

Step 13: Training Process Execution

This last stage gives customers adaptable choices for running the system in several ways and implements the entrance point for applying the reinforcement learning (RL) training process. Before starting the training process, users of the interactive command-line interface presented by the code can choose the suitable operational mode.

Running with simulation for development and testing, using the real Cuckoo Sandbox API for production training, resuming from a particular task ID to continue interrupted training sessions, or exiting without starting the training process provides four different execution options. Resuming from a certain

task ID, the code changes the global **STARTING_TASK_ID** constant to guarantee appropriate sequence continuation. Comprehensive error handling around the execution catches keyboard interruptions for elegant termination and records unanticipated exceptions using thorough stack traces for debugging needs. Whether for development, testing, or production deployment, this user-friendly interface recognizes the possibly long-running character of the training process and the necessity of several operational modes based on the particular situation of the user.

```
## Step 13: Run the Training Process

if __name__ == "__main__":
    # Ask the user if they want to proceed with training
    print("Do you want to proceed with the training process?")
    print("1: Yes, using simulation mode")
    print("2: Yes, using real Cuckoo API")
    print("3: Yes, resume from a specific task ID (real API)")
    print("4: No, exit now")

    try:
        choice = input("Enter your choice (1-4): ")

        if choice == "1":
            # Run with simulation
            main(use_simulation=True)
        elif choice == "2":
            # Run with real API
            main(use_simulation=False)
        elif choice == "3":
            # Resume from specific task ID
            resume_task_id = int(input("Enter task ID to resume from: "))

            # Update the starting task ID constant
            global STARTING_TASK_ID
            STARTING_TASK_ID = resume_task_id

            print(f"Resuming training from task ID {resume_task_id}")
            main(use_simulation=False)
        else:
            print("Exiting without training.")
    except KeyboardInterrupt:
        print("\nTraining interrupted by user.")
    except Exception as e:
        print(f"\nError during training: {e}")
        import traceback
        traceback.print_exc()

    print("\nProcess complete!")
```

Figure *48*: Overall Training Process

Implementation Environment

The system was deployed using a distributed and isolated architecture to maintain flexibility and safety:

- **Google Colab:** RL agent training with TensorFlow and orchestration scripts
- **MinGW:** Cross-compiled C malware for Windows
- **Cuckoo Sandbox:** Local analysis environment with behavior monitoring
- **Ngrok:** Secured API tunnel between Colab and sandbox system

This setup ensured modularity and control while preserving realistic malware deployment conditions.

Testing Methodology

A structured, multi-phase testing methodology was employed to evaluate both the individual components and the integrated functionality of the reinforcement learning-based malware evasion framework. The testing phases progressed from isolated validation of individual evasion techniques to fully autonomous exploration of evasion sequences by the RL agent.

- **Unit Testing:**

  Validating the accuracy and standalone behavior of every evasion method took front stage in the unit testing stage. This included confirming that, when coupled with the base virus, all source code modules built correctly without faults and ran without runtime issues. Every evasion technique was examined separately to make sure its inclusion had no effect on normal malware operation or cause unneeded behavior.

Every method was also sent to the sandbox to evaluate whether it may lower the detection score or cause environmental reactions. During later phases of testing, this phase provided the basis for verifying that every method was operational and could be consistently triggered upon choice by the RL agent.

- **Predetermined Combination Testing:**

A set of predefined evasion combinations was evaluated to see how various methods affected detection scores before releasing the reinforcement learning agent. The tests started with a baseline submission of the un altered basic malware, which routinely scored 5.8 for detection. Individual testing of every evasive approach to evaluate its solitary impact on evasion performance came next.

To assess interaction effects, further experiments combined two and three approaches across several categories. This phase produced some important realizations. For example, the H3 approach (recent documents check) might get a score of 0.0 when employed by itself, but the combo of F1 and F3 both filesystem-based regularly dropped the score to about 0.6. This step produced a set of reference points against which RL-discovered techniques might subsequently be assessed.

- **RL-driven Testing:**

The last and most thorough testing stage consisted on autonomous strategy discovery by the reinforcement learning agent. This phase was intended to assess the agent's capacity for learning efficient evasion strategies by environmental interaction. Starting with a high exploration rate (epsilon = 1.0), the agent progressively turned toward using acquired methods as training went on and epsilon decreased.

There were seventy episodes overall, three iterations for each, which produced and examined almost 210 different malware strains. Comprehensive metrics were gathered throughout this period including action distributions, reward trends, declining detection scores, and convergence patterns. The capacity of the system to independently investigate the evasion method space and converge on optimal sequences that exceeded manually chosen combinations in many cases was shown by the RL-driven testing phase.

Test Case 1: Validation of Individual Evasion Techniques

- **Objective:**

To verify that each evasion technique, when selected by the RL agent, compiles correctly, integrates with the base malware, and functions without runtime issues during sandbox analysis.

- **Process Explanation:**

High exploration probability ($\varepsilon = 1.0$) drives the RL agent to often choose solitary evasion tactics during early training episodes. These early contacts provide appropriate benchmarks for verifying the execution of every approach. The framework automatically applies a penalty reward of –1 to deter repeated selection in any case when a chosen evasion strategy fails to compile or link properly. Techniques that have been successfully assembled and implemented pass the sandbox evaluation process and their corresponding detection scores are noted for additional learning..

Above is RL Model fail attempt for do the necessary compliation after adding T1 evasion technique to the base malware. This action reward the training process of -1. This test case confirms that the reward mechanism properly penalizes faulty

actions, preventing compilation-related failures from corrupting the learning trajectory

Test Case 3: Enforcement of Action Constraints

- **Objective:**

To ensure that the RL agent respects the enforced constraints on technique combinations during action selection and malware construction.

- **Process Explanation:**

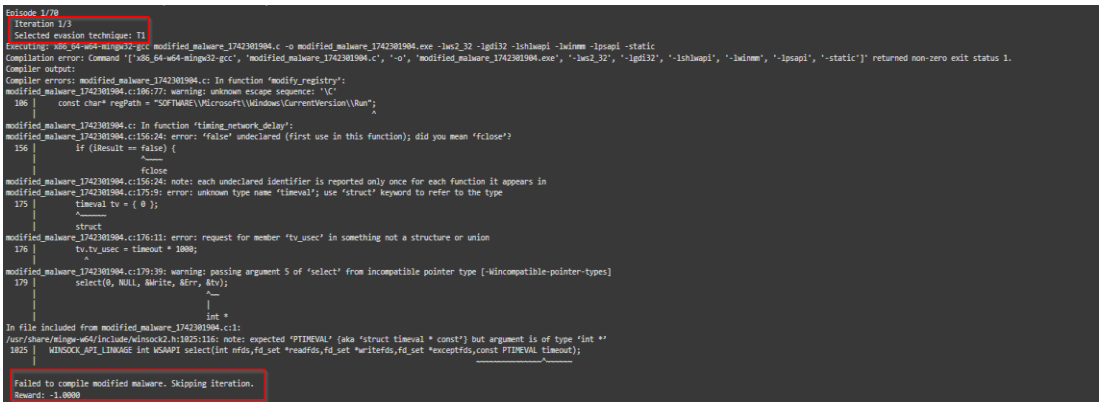


Figure *49*: Compilation Error RL Model Evasion

The system follows rigorous guidelines about the maximum number of concurrently applicable evasion tactics per category. Agents may try to generate invalid combinations that is, choose more than one temporal evasion or violate the limit on human behavior evasions during training. The action filtering system catches these situations.

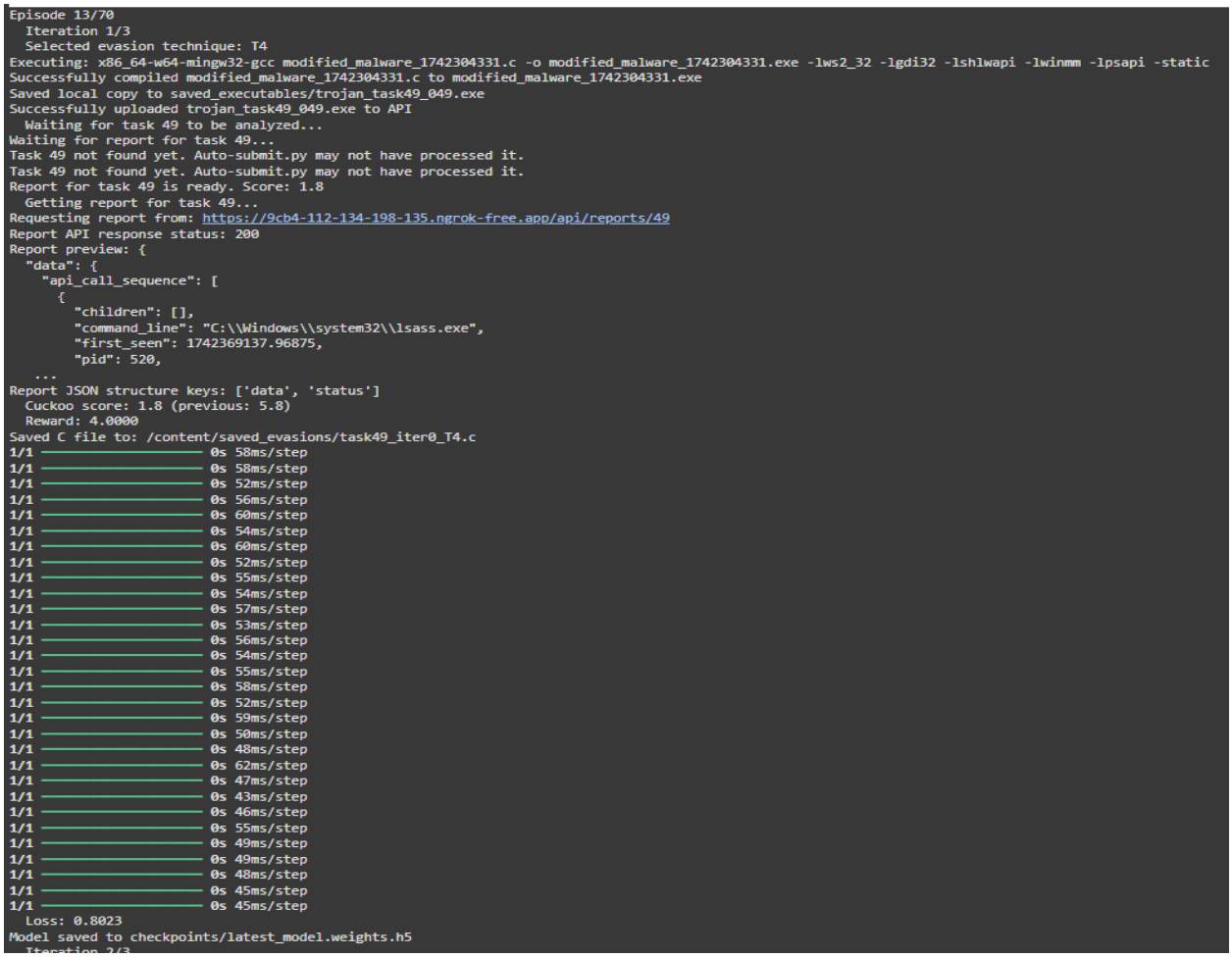


Figure *50*: RL Model Successful Evasion Adding

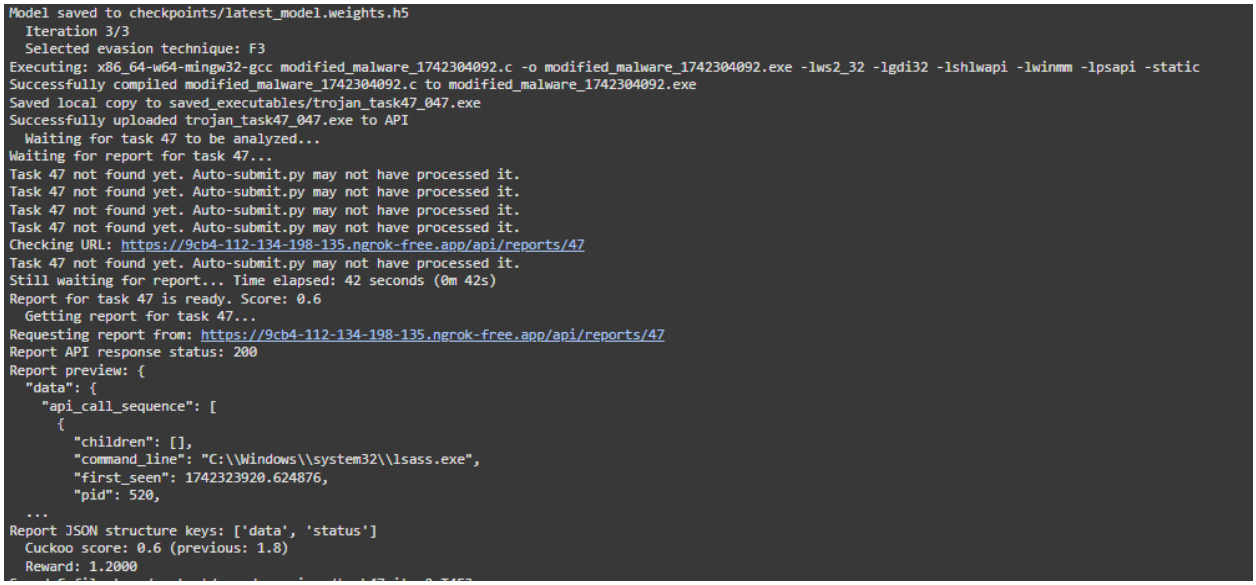


Figure *51*: RL Model Report Receiving from Cuckoo

Above Iterations clarify that the RL agent does following things,

- Invalid combinations are blocked prior to malware compilation
- The agent automatically selects alternative valid actions
- No malformed or rule-violating binaries are passed to the sandbox

By analyzing episodes where multiple evasion techniques are selected, this test case demonstrates the system's ability to enforce logical boundaries while still allowing sufficient exploration of the solution on space.

Test Case 3: Policy Learning and Convergence Evaluation

- **Objective:**

  To evaluate whether the RL agent progressively learns to identify and select effective evasion strategies that reduce sandbox detection scores over time.

- **Process Explanation:**

  The agent moves from exploration to exploitation across 70 training episodes. Key learning indicators including reward trajectory, detection score trends, and Q-value stabilization are tracked on the test. Tracking episodes helps one to see whether the agent converges toward sequences that regularly attain detection scores of ≤ 1.0, therefore indicating a successful evasion.

  Following Episode 55, the training was able to identify the complete evasion sequence approach and it is clear that RL agent have learnt which evasion provides more of the evasion within the sandbox and tries to undertake more exploitation than exploration. Following figure is making clear that

  .

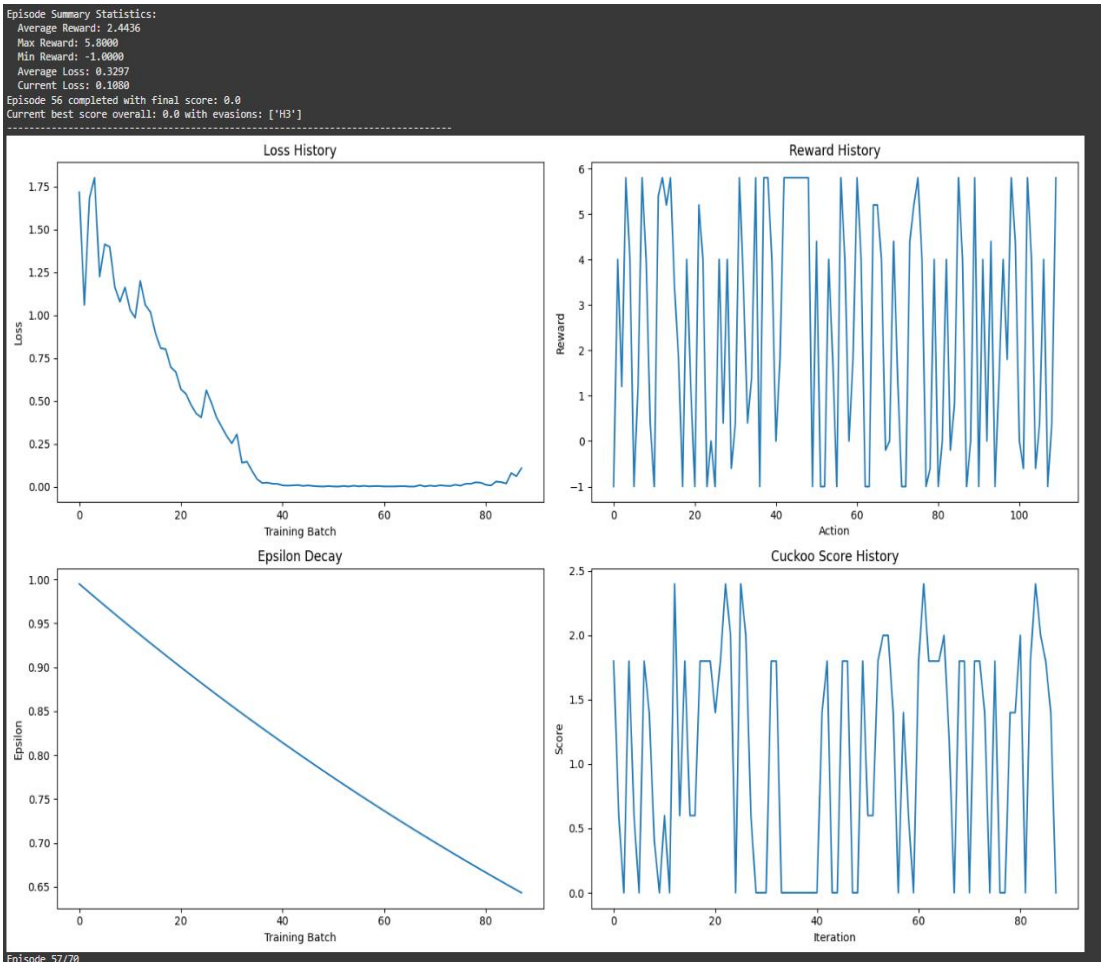


Figure *52*: Episode Overall Information after Execution

So, the success criteria include,

- **A downward trend in average detection scores**
- **Repetition of high-performing evasion combinations**
- **Stabilization of Q-values across episodes**
- **Reduced variance in agent decision-making as epsilon decreases**

This test validates the learning dynamics of the system and confirms its ability to autonomously improve performance through interaction with the sandbox environment.

2.3.2 System Architecture Testing and Implementation - User Behavior Simulation Component (WGAN-GP)

Data Preparation and Normalization

This section handles the initial setup and preparation of behavioural data for training the GAN model. First, the system allows the user to upload a CSV file containing real user behaviour data (such as keystrokes and mouse movements). Once uploaded, the dataset is loaded and basic information is printed for inspection.

To ensure stable training of the neural networks, the raw data is normalised to a standard range between -1 and 1. This transformation helps the model learn more efficiently. Additionally, the minimum and maximum values of each feature are stored so that the generated synthetic data can later be converted back (denormalised) to realistic, original scales.

Finally, the prepared data is converted into PyTorch tensors and made ready for input to the model.

```python
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader, TensorDataset
from scipy import stats
from google.colab import files
import time
from tqdm import tqdm

# Upload file manually
print("Please upload your CSV file with behavioral biometrics data")
uploaded = files.upload()
filename = list(uploaded.keys())[0]  # Get uploaded file name
print(f"Using uploaded file: {filename}")

# Check if running on GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Load dataset
df = pd.read_csv(filename)
print(f"Dataset shape: {df.shape}")
print("First few rows of the dataset:")
print(df.head())

# Normalize dataset between -1 and 1 for stability
def normalize_data(df):
    return 2 * ((df - df.min()) / (df.max() - df.min())) - 1

# Denormalize data back to original range
def denormalize_data(df, ref_df):
    return ((df + 1) / 2) * (ref_df.max() - ref_df.min()) + ref_df.min()

# Store original min/max for later denormalization
df_min = df.min()
df_max = df.max()

# Normalize data
df_scaled = normalize_data(df)

# Convert dataset to tensors
```

Figure 53*: Data preparation & normalization implementation*

GAN Training Hyperparameters

This section defines the critical hyperparameters used to control the training of the WGAN-GP model.

```
# WGAN-GP Hyperparameters
latent_dim = 100  # Size of random noise vector
batch_size = 64
n_critic = 5  # Number of critic updates per generator update
lambda_gp = 10  # Gradient penalty coefficient
epochs = 2000
early_stopping_patience = 200
eval_interval = 50
lr = 0.0001  # Learning rate for both networks
beta1 = 0.5
beta2 = 0.9
```

Figure 54*: WGAN-GP hyperparameter initialization*

Generator and Critic Network Implementation

This sub-section defines the core components of the WGAN-GP architecture: the Generator and the Critic (Discriminator). The Generator takes a latent vector (random noise) and transforms it into a synthetic user profile.

The Critic distinguishes between real and synthetic samples by estimating their Wasserstein distance.

```
# Define the Generator with Batch Normalization
class Generator(nn.Module):
    def __init__(self, latent_dim, feature_dim):
        super(Generator, self).__init__()
        self.latent_dim = latent_dim
        self.feature_dim = feature_dim

        self.model = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),

            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2),

            nn.Linear(512, 1024),
            nn.BatchNorm1d(1024),
            nn.LeakyReLU(0.2),

            nn.Linear(1024, feature_dim),
            nn.Tanh()  # Output in range [-1, 1]
        )

    def forward(self, z):
        # Ensure z has the right shape for BatchNorm
        if z.dim() == 2:
            return self.model(z)
        else:
            z_reshaped = z.view(-1, self.latent_dim)
            return self.model(z_reshaped)

# Define Critic (Discriminator without Sigmoid)
class Critic(nn.Module):
    def __init__(self, feature_dim):
        super(Critic, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(feature_dim, 1024),
            nn.LayerNorm(1024),  # Layer normalization instead of batch norm
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),

            nn.Linear(1024, 512),
            nn.LayerNorm(512),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),

            nn.Linear(512, 256),
            nn.LayerNorm(256),
            nn.LeakyReLU(0.2),

            nn.Linear(256, 1)  # No activation for Wasserstein distance
        )

    def forward(self, x):
        return self.model(x)
```

Figure 55*: Generator & critic network implementation*

Training loop

The training loop constitutes the core iterative process of the WGAN-GP model. During each epoch, real user profile batches are sampled and passed through the critic to compute their scores. Simultaneously, the generator produces synthetic profiles from random latent vectors. The critic is trained using the Wasserstein loss function with a gradient penalty to ensure Lipschitz continuity.

Every n_critic steps, the generator is updated to minimize the critic's score on synthetic data while also incorporating a correlation loss component. This correlation loss ensures the preservation of statistical relationships (feature-wise) between real and generated profiles. Both generator and critic are optimized using Adam optimizer with learning rate schedulers. Losses for each step are logged for performance monitoring and future visualization.

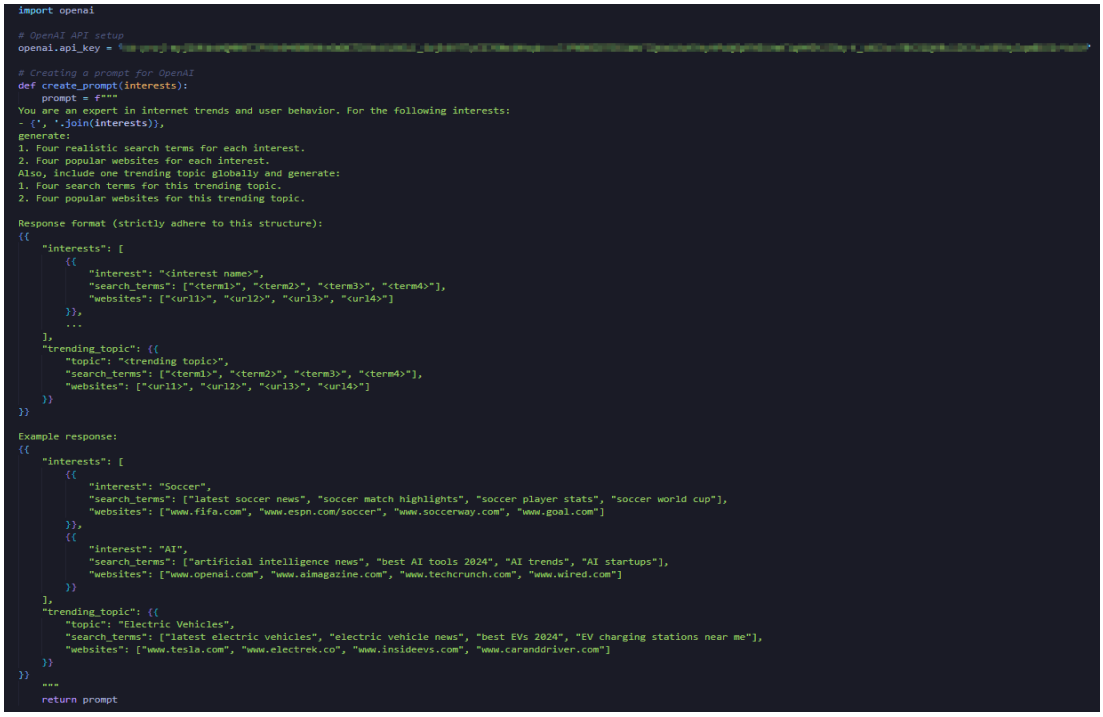
```python
# Training Loop
start_time = time.time()
for epoch in range(epochs):
    for i, real_batch in enumerate(dataloader):
        real_samples = real_batch[0].to(device)
        batch_size = real_samples.size(0)

        # ---------------------
        # Train Critic
        # ---------------------
        optimizer_C.zero_grad()

        # Generate fake samples
        z = torch.randn(batch_size, latent_dim).to(device)
        fake_samples = generator(z).detach()

        # Compute critic loss with gradient penalty
        real_validity = critic(real_samples)
        fake_validity = critic(fake_samples)

        # Wasserstein Loss
        critic_loss = -torch.mean(real_validity) + torch.mean(fake_validity)

        # Gradient penalty
        gp = compute_gradient_penalty(critic, real_samples, fake_samples)

        # Total critic loss
        critic_loss = critic_loss + gp

        critic_loss.backward()
        optimizer_C.step()

        # Store critic loss
        c_loss_history.append(critic_loss.item())

        # ---------------------
        # Train Generator
        # ---------------------
        # Train generator every n_critic steps
        if i % n_critic == 0:
            optimizer_G.zero_grad()

            # Generate fake samples
            z = torch.randn(batch_size, latent_dim).to(device)
            fake_samples = generator(z)

            # Get critic score
            fake_validity = critic(fake_samples)

            # Add correlation loss
            corr_loss = correlation_loss(real_samples, fake_samples)

            # Generator loss (minimize negative critic score with correlation penalty)
            gen_loss = -torch.mean(fake_validity) + 0.1 * corr_loss

            gen_loss.backward()
            optimizer_G.step()
```

Figure 56*: WGAN-GP training loop*

User Profile Enrichment with OpenAI

This section describes the steps taken to enrich synthetic user profiles based on realistic web browsing behaviour using OpenAI. First, the GAN model generates a synthetic profile with certain behavioural characteristics. The methodology then selects a random user interests set from a fixed user interest set. This set of interests will then be collectively used to create a structured prompt, which is sent to OpenAI using the API. For each interest, there are two things the prompt requests:

1. Search terms that are likely realistic for a user to search for on a search engine.
2. Relevant websites or URLs that are likely on topic, and that a user might visit when exploring that topic.



```
import openai

# OpenAI API setup
openai.api_key = "

# Creating a prompt for OpenAI
def create_prompt(interests):
    prompt = f"""
You are an expert in internet trends and user behavior. For the following interests:
- {', '.join(interests)},
generate:
1. Four realistic search terms for each interest.
2. Four popular websites for each interest.
Also, include one trending topic globally and generate:
1. Four search terms for this trending topic.
2. Four popular websites for this trending topic.

Response format (strictly adhere to this structure):
{{
    "interests": [
        {{
            "interest": "<interest name>",
            "search_terms": ["<term1>", "<term2>", "<term3>", "<term4>"],
            "websites": ["<url1>", "<url2>", "<url3>", "<url4>"]
        }},
        ...
    ],
    "trending_topic": {{
        "topic": "<trending topic>",
        "search_terms": ["<term1>", "<term2>", "<term3>", "<term4>"],
        "websites": ["<url1>", "<url2>", "<url3>", "<url4>"]
    }}
}}

Example response:
{{
    "interests": [
        {{
            "interest": "Soccer",
            "search_terms": ["latest soccer news", "soccer match highlights", "soccer player stats", "soccer world cup"],
            "websites": ["www.fifa.com", "www.espn.com/soccer", "www.soccerway.com", "www.goal.com"]
        }},
        {{
            "interest": "AI",
            "search_terms": ["artificial intelligence news", "best AI tools 2024", "AI trends", "AI startups"],
            "websites": ["www.openai.com", "www.aimagazine.com", "www.techcrunch.com", "www.wired.com"]
        }}
    ],
    "trending_topic": {{
        "topic": "Electric Vehicles",
        "search_terms": ["latest electric vehicles", "electric vehicle news", "best EVs 2024", "EV charging stations near me"],
        "websites": ["www.tesla.com", "www.electrek.co", "www.insideevs.com", "www.caranddriver.com"]
    }}
}}
    """
    return prompt
```

Figure 57: Prompt generation for OpenAI

```
# Sending API request
def call_chatgpt(prompt):
    response = openai.ChatCompletion.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": prompt}
        ],
        max_tokens=500
    )
    return response['choices'][0]['message']['content'].strip()

response_text = call_chatgpt(prompt)
print("ChatGPT Response:\n", response_text)
```

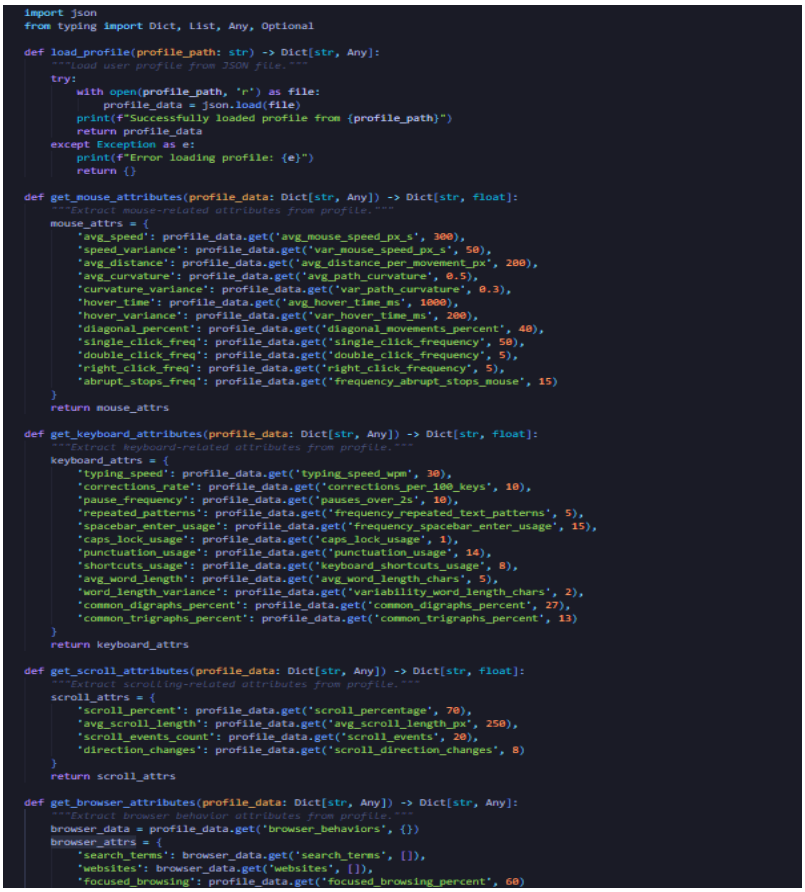
Figure 58: OpenAI API configuration

Scripting Engine Implementation

The Script Generation Engine has the task of turning the enriched synthetic user profiles into executable scripts that mimic a real-world interaction of users in a sandbox environment. This engine consists of four different modules that work in concert with each other to generate human-like activity patterns. Each module has distinct function in the pipeline that contributes to difference parts of the script synthesis process.

User profile management module

This module accepts the enriched user profiles as final outputs, that include behavioural metrics and browsing interests. It processes user-specific such as average typing speed, hover time, and scrolling behaviour. These user-specific attributes are passed downstream to affect the realistic simulation of behaviours in the later modules.



```python
import json
from typing import Dict, List, Any, Optional

def load_profile(profile_path: str) -> Dict[str, Any]:
    """Load user profile from JSON file."""
    try:
        with open(profile_path, 'r') as file:
            profile_data = json.load(file)
        print(f"Successfully loaded profile from {profile_path}")
        return profile_data
    except Exception as e:
        print(f"Error loading profile: {e}")
        return {}

def get_mouse_attributes(profile_data: Dict[str, Any]) -> Dict[str, float]:
    """Extract mouse-related attributes from profile."""
    mouse_attrs = {
        'avg_speed': profile_data.get('avg_mouse_speed_px_s', 300),
        'speed_variance': profile_data.get('var_mouse_speed_px_s', 50),
        'avg_distance': profile_data.get('avg_distance_per_movement_px', 200),
        'avg_curvature': profile_data.get('avg_path_curvature', 0.5),
        'curvature_variance': profile_data.get('var_path_curvature', 0.3),
        'hover_time': profile_data.get('avg_hover_time_ms', 1000),
        'hover_variance': profile_data.get('var_hover_time_ms', 200),
        'diagonal_percent': profile_data.get('diagonal_movements_percent', 40),
        'single_click_freq': profile_data.get('single_click_frequency', 50),
        'double_click_freq': profile_data.get('double_click_frequency', 5),
        'right_click_freq': profile_data.get('right_click_frequency', 5),
        'abrupt_stops_freq': profile_data.get('frequency_abrupt_stops_mouse', 15)
    }
    return mouse_attrs

def get_keyboard_attributes(profile_data: Dict[str, Any]) -> Dict[str, float]:
    """Extract keyboard-related attributes from profile."""
    keyboard_attrs = {
        'typing_speed': profile_data.get('typing_speed_wpm', 30),
        'corrections_rate': profile_data.get('corrections_per_100_keys', 10),
        'pause_frequency': profile_data.get('pauses_over_2s', 10),
        'repeated_patterns': profile_data.get('frequency_repeated_text_patterns', 5),
        'spacebar_enter_usage': profile_data.get('frequency_spacebar_enter_usage', 15),
        'caps_lock_usage': profile_data.get('caps_lock_usage', 1),
        'punctuation_usage': profile_data.get('punctuation_usage', 14),
        'shortcuts_usage': profile_data.get('keyboard_shortcuts_usage', 8),
        'avg_word_length': profile_data.get('avg_word_length_chars', 5),
        'word_length_variance': profile_data.get('variability_word_length_chars', 2),
        'common_digraphs_percent': profile_data.get('common_digraphs_percent', 27),
        'common_trigraphs_percent': profile_data.get('common_trigraphs_percent', 13)
    }
    return keyboard_attrs

def get_scroll_attributes(profile_data: Dict[str, Any]) -> Dict[str, float]:
    """Extract scrolling-related attributes from profile."""
    scroll_attrs = {
        'scroll_percent': profile_data.get('scroll_percentage', 70),
        'avg_scroll_length': profile_data.get('avg_scroll_length_px', 250),
        'scroll_events_count': profile_data.get('scroll_events', 20),
        'direction_changes': profile_data.get('scroll_direction_changes', 8)
    }
    return scroll_attrs

def get_browser_attributes(profile_data: Dict[str, Any]) -> Dict[str, Any]:
    """Extract browser behavior attributes from profile."""
    browser_data = profile_data.get('browser_behaviors', {})
    browser_attrs = {
        'search_terms': browser_data.get('search_terms', []),
        'websites': browser_data.get('websites', []),
        'focused_browsing': profile_data.get('focused_browsing_percent', 60)
```

Figure 59: User profile management module implementation

Behavior Engine

Using the behavioural attributes from the Profile Management Module, the Behaviour Engine generates generalized user action patterns such as:

- Typing sequences with variability and pauses
- Mouse movements including jitter and idle times
- Scroll actions with realistic speed and range

These behaviours are designed to emulate authentic interaction dynamics and are reused across activity templates.

```python
"""
Behavior Engine Module
This module is responsible for generating and executing realistic user behaviors
based on profile attributes extracted by the User Profile Management module.
It fully utilizes all behavioral attributes from the user profile.
"""

import random
import math
import numpy as np
from scipy import interpolate
from typing import Dict, List, Tuple, Any, Optional
import time
import pyautogui

# Configure pyautogui for safety and to work with the behavior engine
pyautogui.PAUSE = 0  # We'll handle timing ourselves
pyautogui.FAILSAFE = True  # Safety feature - move mouse to corner to abort

# Global variables to store generated behavior data
mouse_movements = []
scroll_events = []
keystroke_sequences = []
browser_urls = []
search_terms = []

# Common English digraphs and trigraphs with their relative frequencies
COMMON_DIGRAPHS = {
    'th': 3.56, 'he': 3.07, 'in': 2.43, 'er': 2.05, 'an': 1.99,
    're': 1.85, 'on': 1.76, 'at': 1.49, 'en': 1.45, 'nd': 1.35,
    'ti': 1.34, 'es': 1.34, 'or': 1.28, 'te': 1.20, 'of': 1.17,
    'ed': 1.17, 'is': 1.13, 'it': 1.12, 'al': 1.09, 'ar': 1.07,
    'st': 1.05, 'to': 1.04, 'nt': 1.04, 'ng': 0.95, 'se': 0.93,
    'ha': 0.93, 'as': 0.87, 'ou': 0.87, 'io': 0.83, 'le': 0.83,
    've': 0.83, 'co': 0.79, 'me': 0.79, 'de': 0.76, 'hi': 0.76,
    'ri': 0.73, 'ro': 0.73, 'ic': 0.70, 'ne': 0.69, 'ea': 0.69,
    'ra': 0.69, 'ce': 0.68, 'li': 0.62, 'ch': 0.60, 'll': 0.58
}

COMMON_TRIGRAPHS = {
    'the': 3.51, 'and': 1.59, 'ing': 1.15, 'her': 0.82, 'hat': 0.65,
    'his': 0.65, 'tha': 0.64, 'ere': 0.59, 'for': 0.58, 'ent': 0.56,
    'ion': 0.56, 'ter': 0.54, 'was': 0.53, 'you': 0.51, 'ith': 0.50,
    'ver': 0.50, 'all': 0.49, 'wit': 0.44, 'thi': 0.44, 'tio': 0.44
}

# Keyboard layout for error simulation and digraph timing
KEYBOARD_LAYOUT = {
    'q': (0, 0), 'w': (1, 0), 'e': (2, 0), 'r': (3, 0), 't': (4, 0),
    'y': (5, 0), 'u': (6, 0), 'i': (7, 0), 'o': (8, 0), 'p': (9, 0),
    'a': (0, 1), 's': (1, 1), 'd': (2, 1), 'f': (3, 1), 'g': (4, 1),
    'h': (5, 1), 'j': (6, 1), 'k': (7, 1), 'l': (8, 1), ';': (9, 1),
    'z': (0, 2), 'x': (1, 2), 'c': (2, 2), 'v': (3, 2), 'b': (4, 2),
    'n': (5, 2), 'm': (6, 2), ',': (7, 2), '.': (8, 2), '/': (9, 2)
}

def calculate_key_distance(key1: str, key2: str) -> float:
    """Calculate physical distance between keys on a keyboard."""
    # Default to average distance if key not found
    if key1.lower() not in KEYBOARD_LAYOUT or key2.lower() not in KEYBOARD_LAYOUT:
        return 1.0

    pos1 = KEYBOARD_LAYOUT[key1.lower()]
    pos2 = KEYBOARD_LAYOUT[key2.lower()]

    # Euclidean distance
    return math.sqrt((pos1[0] - pos2[0])**2 + (pos1[1] - pos2[1])**2)
```

Figure 60: Behaviour engine module implementation

Scheduler Module

The scheduler is responsible for temporal coordination. It takes the full set of enriched activity scripts and assigns them to appropriate time slots based on a generated timetable. This ensures that script execution mimics a plausible user workday or activity schedule.

```python
"""
Scheduler & Execution Module
This module is responsible for creating and managing execution schedules for behavior scripts.
"""

import os
import json
import random
import shutil
from typing import Dict, List, Any, Optional
import time

def create_schedule(template_names: List[str], duration_minutes: int = 60) -> Dict[str, Any]:
    """
    Create a schedule for script execution over specified duration.

    Args:
        template_names: List of available template names
        duration_minutes: Total duration in minutes

    Returns:
        A schedule dictionary with activities and their timings
    """
    if not template_names:
        raise ValueError("No templates available for scheduling")

    # Determine how many activities to include based on duration and available templates
    min_activities = max(1, duration_minutes // 15)  # At least 1 activity per 15 minutes
    max_activities = max(min_activities * 2, duration_minutes // 5)  # At most 1 activity per 5 minutes

    # Determine number of activities
    num_activities = min(max_activities, random.randint(min_activities, max_activities))

    # If we have fewer templates than needed activities, we'll reuse templates
    needed_templates = []
    while len(needed_templates) < num_activities:
        # Add templates in random order
        shuffled = random.sample(template_names, min(num_activities - len(needed_templates), len(template_names)))
        needed_templates.extend(shuffled)

    # Start and end times
    start_time = 0  # in seconds
    end_time = duration_minutes * 60  # in seconds

    # Create schedule
    schedule = {
        "duration_minutes": duration_minutes,
        "activities": []
    }

    # Current time pointer
    current_time = start_time

    # Distribute activities across the schedule
    for template_name in needed_templates:
        # Determine activity duration based on remaining time
        remaining_time = end_time - current_time
        if remaining_time <= 0:
            break  # No more time left

        # Calculate max duration to ensure we don't exceed the schedule
        max_duration = min(10 * 60, remaining_time)  # Max 10 minutes or remaining time
        min_duration = min(1 * 60, max_duration)     # Min 1 minute unless less time remains

        # Determine activity duration
        activity_duration = random.randint(min_duration, max_duration)
```

Figure 61: Scheduler module implementation

WGAN-GP Training Process Testing

The first stage of testing involved verifying that the GAN model was implemented and trained correctly without interruption. The model was trained using the real, preprocessed behavioural dataset. During training, both the generator and critic were updated using their respective loss functions, and early stopping was implemented to preserve the best-performing model.

```
Epoch 50/2000 | G Loss: 1.2428 | C Loss: 0.2385 | Avg KS p-value: 0.078973 | Time: 32.4s
New best model saved with avg p-value: 0.078973
Epoch 100/2000 | G Loss: 3.2555 | C Loss: -0.1414 | Avg KS p-value: 0.038452 | Time: 60.6s
Epoch 150/2000 | G Loss: 4.3731 | C Loss: -0.0927 | Avg KS p-value: 0.003238 | Time: 87.8s
Epoch 200/2000 | G Loss: 5.5302 | C Loss: 0.0762 | Avg KS p-value: 0.021210 | Time: 114.7s
Epoch 250/2000 | G Loss: 4.7568 | C Loss: -0.1199 | Avg KS p-value: 0.011818 | Time: 141.5s
Epoch 300/2000 | G Loss: 4.8916 | C Loss: 0.0195 | Avg KS p-value: 0.001704 | Time: 168.5s
```

Figure 62: WGAN-GP training process

After the training process concluded (either after full epochs or due to early stopping), the best generator model was restored from its checkpoint. Using this model, a batch of synthetic profiles was generated by feeding latent vectors through the trained generator.

Figure 63: Generated user profile by WGAN-GP model

OpenAI Enrichment Testing

After the GAN model generates a synthetic user profile, the next step is to enrich the profile with interest-based web activity.

It is expected for the API's response to contain two items for each interest:-

- A list of search terms that a user might enter
- A list of realistic URLs or websites that relate to the topic

Figure 64*: OpenAI response*

Scripting Engine Testing

The scripting engine was launched using a generated synthetic user profile as input. Upon execution, the system sequentially activated each of its core modules—parsing the profile, generating generalised behaviour patterns (mouse movements, keystrokes, scrolling), updating script templates, and creating the session schedule.

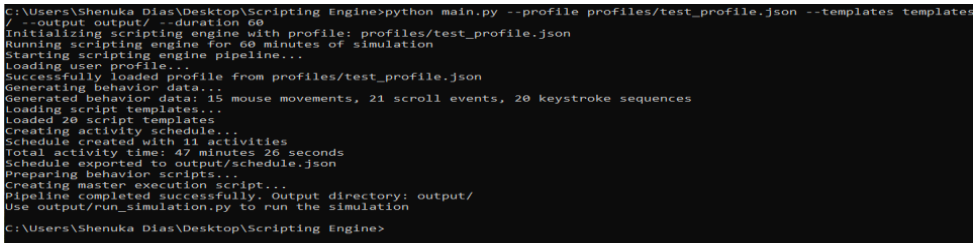


Figure 65*: Scripting engine execution*

The engine correctly populated user behaviour templates using data from the Behaviour Engine. Each selected template was updated with timing and movement logic based on the user's typing speed, scroll rate, and mouse activity.

Below figure 34 shows the scripts that were generated for the user session.

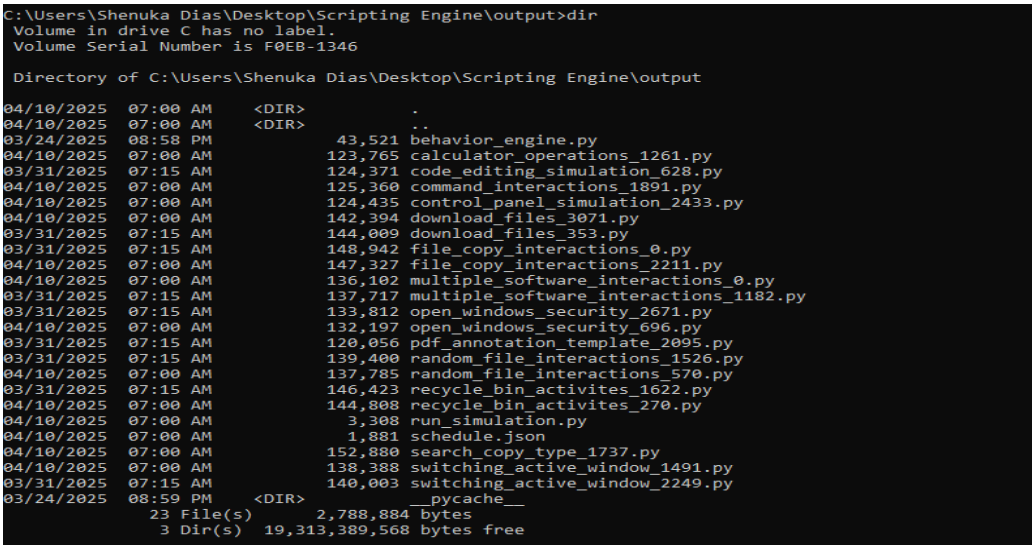


Figure 66*: Generated user behaviour simulation scripts*

Next the scripts were tested execution within the sandbox. Figure 35 shows the logs that got while executing the scripts in the sandbox.

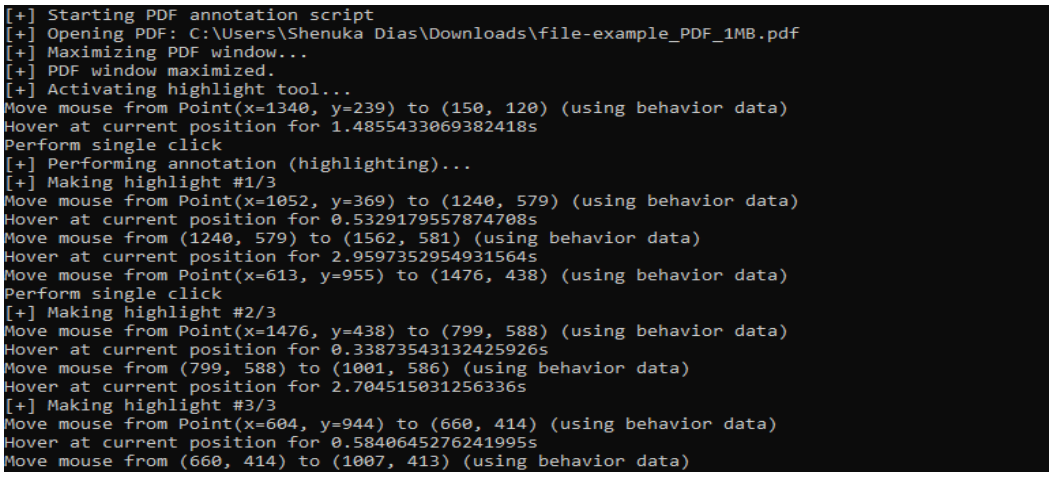


Figure 67*: Successful script execution in sandbox*

2.3.3 System Architecture Testing and Implementation - Hybrid Detection System for Sandbox Evasion Techniques Component

This section describes the testing and deployment stages of the research, concentrating on assessing the trained model and its deployment to detect sandbox evasion techniques using behavior analysis reports. The purpose of these testing stages is to assess whether the model may accurately detect techniques for sandbox evasion. We will measure performance metrics including accuracy, precision, recall, and F1 score. The model will undergo testing based on varying datasets to examine whether it is able to generalize to new, unseen data, and in order to test the robustness of the model against different evasion methods typically observed in malware. By testing the model thoroughly, we can be confident that it will be able to categorize different evasive behavior and make accurate predictions.

The implementation phase involves building and implementing the techniques described in the methodology section of this thesis; specifically data preprocessing, tokenization, and addressing class imbalance, which are necessary to prepare the dataset for model training. Data preprocessing included extracting and cleaning API sequences from the behavior reports of malware samples, tokenizing, and padding the data to make it compatible with the model input requirements. In addition to preprocessing data, to model performance in detecting rare evasion techniques, dataset augmentation and generation of synthetic data for underrepresented classes addressed class imbalance. Once the model is trained, the model is then used to analyze the malware behavior reports to predict sandbox evasion techniques based on the sequences of API calls. The model's predictions are evaluated to measure the performance of the model to ensure it is capable of a functional utility within the context of malware analysis in the real world.

Data Collection and Preprocessing

The dataset utilized in this study was taken from Khas-Ccip's (2023; see Khas-Ccip, 2023) "API Sequences Malware Datasets" on GitHub, which consists of a dynamic malware analysis benchmark based on API calls from the PEFile library in Python and malware types identified from the VirusTotal API. The dataset consists of two main sets: The VirusSample set which has 9,795 samples and the VirusShare set which has 14,616 samples. This dataset has significant value in that it provides an API call sequence documented with the evasive methods malware used during the sequence's execution process, thereby providing researchers with an understanding of the malware's and its interaction with system APIs during execution. Although the dataset provides an example of inherent class imbalance, it was still included in both the training and testing environment because of its value to the study, as it has an example of Adware, Trojan and Ransomware makes, as well as evasive methods employed to deter malicious software detection. Further, the aspect of imbalanced data allows the model to achieve higher accuracy which further strengthens its suitability for training machine-learning models that detect and identify various evasion techniques and malware behavior.

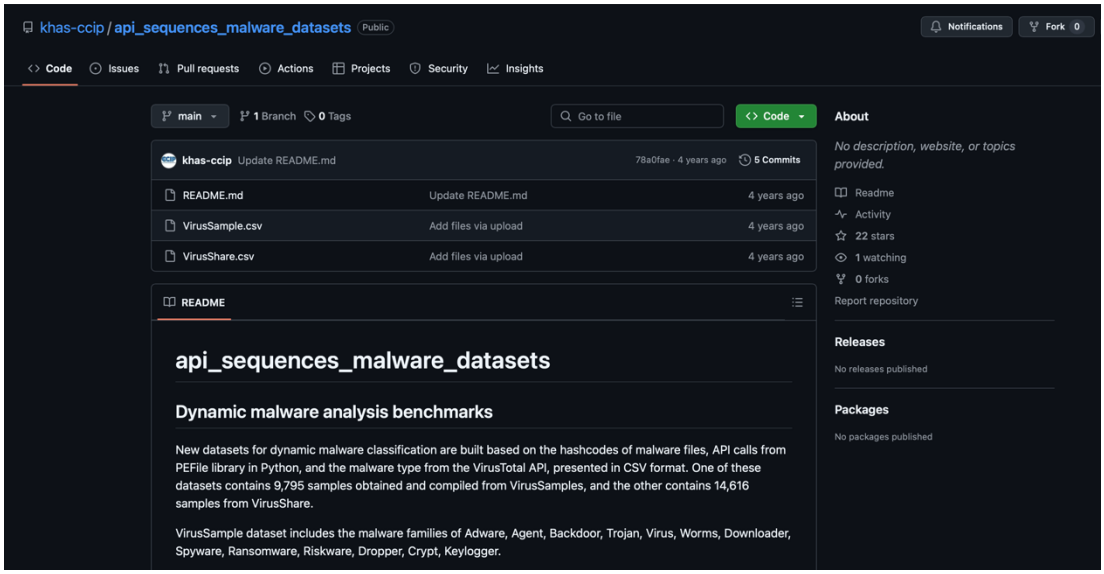


Figure 68 - Evidence for Dataset Source

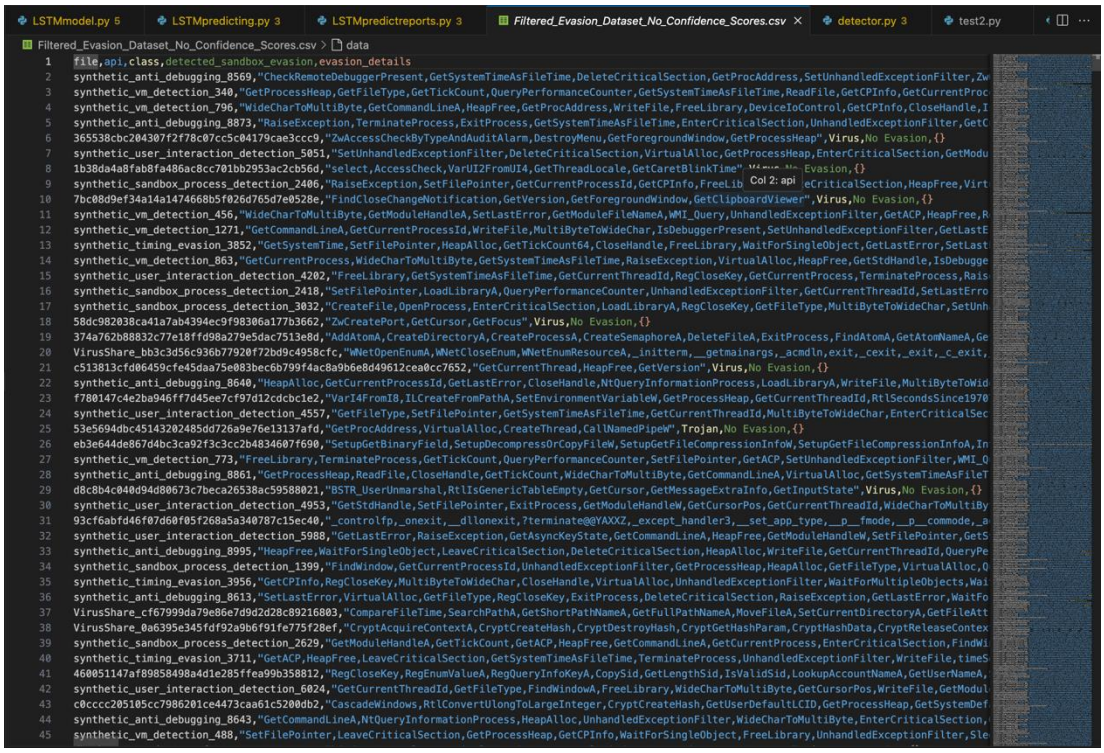


Figure 69 - Preview of the used dataset.

Class Imbalance Handling

Addressing class imbalance is crucial to ensure that the model does not bias its predictions towards the majority class and is able to effectively learn from minority class samples, especially in the case of rare evasion techniques. The following steps were taken to handle the class imbalance in the dataset:

1. **Load and Preprocess Data:** The first step involved loading the dataset, which includes several malware-related features and labels. The features were separated from the labels, and irrelevant or unsupported classes (with zero samples) were dropped. The label columns included different types of sandbox evasion techniques, such as `vm_detection`, `sandbox_process_detection`, and others, which were important for training the model on a multi-label classification task.

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE
from ctgan import CTGAN
from xgboost import XGBClassifier
from sklearn.metrics import classification_report

# Step 1: Load and Preprocess Data
def load_data(filepath):
    data = pd.read_csv(filepath)
    # Drop unsupported class (0 samples)
    data = data.drop(columns=['registry_environment_checks'], errors='ignore')
    # Separate features (X) and labels (y)
    label_columns = ['vm_detection', 'sandbox_process_detection', 'timing_evasion',
                     'user_interaction_detection', 'anti_debugging']
    X = data.drop(columns=label_columns)
    y = data[label_columns]
    return X, y
```

Figure 70 - Loading and Preprocessing of data

2. **Split Data into Train/Validation Sets:** The dataset was split into training and validation sets using an 80/20 ratio. The `train_test_split` function from `sklearn.model_selection` was used to ensure that the data was evenly distributed and that the model could learn from a diverse range of evasion techniques. This step was critical for training the model on the most representative data, which is important when dealing with imbalanced classes.

```python

# Step 2: Split Data into Train/Validation Sets
def split_data(X, y):
    X_train, X_val, y_train, y_val = train_test_split(
        X, y, test_size=0.2, stratify=y, random_state=42
    )
    return X_train, X_val, y_train, y_val

```

Figure 71 - Split_data Function

3. **Apply SMOTE to Training Data:** The **Synthetic Minority Over-sampling Technique (SMOTE)** was applied to balance the training dataset. SMOTE works by generating synthetic samples for the minority class by interpolating between existing instances. This technique was used to generate more samples of the underrepresented evasion classes, thereby ensuring the model could learn from these rare cases and reduce the risk of overfitting on the majority class.

```python

# Step 3: Apply SMOTE to Training Data
def apply_smote(X_train, y_train):
    smote = SMOTE(random_state=42)
    X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
    return X_resampled, y_resampled

```

Figure 72 - Applying SMOTE for the Dataset

4. **Generate Synthetic Data with CTGAN for Minority Classes:** To further address the class imbalance, synthetic data was generated using a **Conditional Generative Adversarial Network (CTGAN)**. This method targets the generation of realistic samples specifically for underrepresented classes, such as the `sandbox_process_detectionclass`. By training the CTGAN on minority class data, new, realistic samples were generated, ensuring that the model had enough data to learn from and improving its ability to generalize.

```python
# Step 4: Generate Synthetic Data with CTGAN for Minority Classes
def generate_ctgan_data(X_train, y_train, target_class, num_samples):
    # Extract minority class samples
    minority_samples = X_train[y_train[target_class] == 1]
    if len(minority_samples) == 0:
        print(f"No samples for {target_class}. Skipping CTGAN.")
        return pd.DataFrame(), pd.DataFrame()
    # Train CTGAN
    ctgan = CTGAN(epochs=100)
    ctgan.fit(minority_samples)
    # Generate synthetic data
    synthetic_X = ctgan.sample(num_samples)
    synthetic_y = pd.DataFrame({target_class: [1]*num_samples})
    return synthetic_X, synthetic_y

```

Figure 73 - Synthetic data generation Process

5. **Train Model with Class Weighting:** After balancing the data, class weights were computed based on the inverse of the class frequencies in the resampled data. This adjustment helps prevent the model from being biased towards the majority class by penalizing it more when it misclassifies minority class instances. The XGBoost classifier was then trained on the resampled dataset, with class weights applied during training to ensure that the model treated each class more equally. This step was essential for improving the model's performance, particularly in detecting evasive malware behaviors that are underrepresented in the dataset.

```
48
49  # Step 5: Train Model with Class Weighting
50  def train_model(X_resampled, y_resampled, X_val, y_val):
51      # Compute class weights (inverse of class frequencies)
52      class_counts = y_resampled.sum(axis=0)
53      class_weights = (len(y_resampled) / (class_counts * len(class_counts))).to_dict()
54      # Train XGBoost (handles multi-label with binary relevance)
55      model = XGBClassifier()
56      model.fit(X_resampled, y_resampled)
57      # Evaluate
58      y_pred = model.predict(X_val)
59      print(classification_report(y_val, y_pred))
60      return model
61
```

Figure 74 - Class rebalancing process

Through these steps, the dataset was effectively balanced, ensuring that the model was trained on a more equitable distribution of evasion techniques, which ultimately improved its detection accuracy for rare and evasive malware behaviors.

Model Architecture and Design

This section describes the architecture and design of the Long short-short term memory (LSTM)-based model employed to detect sandbox evasion techniques. The model taps into the potential of sequential data processing of LSTMs through the use of Bidirectional LSTM layers that are meant to improve its performance identifying both past and future dependencies in sequences of API calls. This particular architecture is suitable for detecting behaviors associated with malware, especially with regard to the detection of more complicated and nuanced evasion techniques when operated in real-time. The rationale of design considerations like added LSTM layers and dropout were used to regularise in layer processing as well as multi-label classification layers were to further optimize adaptable performance in specifying and classifying the many evasive behaviors associated with malware. Let's now discuss the specific design considerations and choices made in the architecture.

Bidirectional LSTM Layer

The **Bidirectional LSTM layer** in the code is crucial for capturing the sequential patterns in the API call sequences. LSTM (Long Short-Term Memory) units are a type of recurrent neural network (RNN) that are well-suited for sequential data, like time-series or text data. In the case of malware detection, the API call sequences act as time-series data, where the order of the API calls and their relationships to one another are important for understanding the malware's behavior. **Bidirectional LSTMs** improve the model's ability to capture these sequential patterns by processing the data in **both forward and backward directions**. This means the model can understand not only what happened before a given API call (past context) but also what might happen afterward (future context). This is important for malware analysis because the behavior of the malware in future time steps can provide additional insights into evasion techniques that may not be fully understood from past behavior alone. In the code, the **Bidirectional(LSTM(128, return_sequences=True))** layer processes the input sequence in both directions, followed by another **Bidirectional(LSTM(64))** layer to further refine the feature extraction. This dual approach ensures that the model benefits from the full context of the sequential data, improving its ability to detect complex evasion tactics.

```python
# Encode multi-labels
def encode_multilabels(evasion_labels):
    print("Encoding multi-labels...")
    mlb = MultiLabelBinarizer()
    encoded_labels = mlb.fit_transform(evasion_labels)

    print(f"Number of target classes: {len(mlb.classes_)}")
    print(f"Classes: {mlb.classes_}")

    return encoded_labels, mlb

# Build LSTM model for multi-label classification
def build_model(vocab_size, max_sequence_length, num_classes):
    print(f"Building LSTM model with {num_classes} output classes...")

    model = Sequential()

    # Embedding layer
    model.add(Embedding(input_dim=vocab_size,
                        output_dim=128,
                        input_length=max_sequence_length))

    # Bidirectional LSTM layers
    model.add(Bidirectional(LSTM(128, return_sequences=True)))
    model.add(Dropout(0.3))
    model.add(Bidirectional(LSTM(64)))
    model.add(Dropout(0.3))

    # Output layer with sigmoid activation for multi-label classification
    model.add(Dense(num_classes, activation='sigmoid'))

    # Compile model
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    model.summary()
    return model

```

Figure 75 - Build_Model function for LSTMs

Layer Design

The **model architecture** is designed with multiple layers to handle the complexity of the malware classification task. The architecture consists of the following layers:

- **Embedding Layer**: The model starts with an **embedding layer** to convert the input integer sequences (the API calls) into dense vectors of fixed size (128 in this case). This allows the model to learn meaningful representations for each API call, helping it to capture the semantic relationships between different API calls. This layer has the dimensions specified by `vocab_size` (the number of unique API calls) and `max_sequence_length` (the maximum length of the API sequences).

- **Bidirectional LSTM Layers**: After the embedding layer, two **Bidirectional LSTM layers** are added. The first LSTM layer uses 128 units and returns sequences to allow the next LSTM layer to continue processing the sequence. The second LSTM layer uses 64 units and outputs the final predictions. The **Bidirectional** wrapper allows the network to process the input sequence in both directions (past and future), which is key for understanding the relationships between sequential API calls.

- **Dropout Layers**: After each LSTM layer, a **Dropout** layer with a rate of 0.3 is used to prevent overfitting by randomly dropping 30% of the neurons during training. This regularization technique helps the model generalize better to unseen data.

- **Dense Layer**: The final layer is a **Dense layer** with a **sigmoid activation function**, which outputs the classification probabilities. Since the model is dealing with multiple evasion techniques, a **multi-label classification** approach is employed, where each output node corresponds to a specific evasion technique (e.g., "VM detection", "timing evasion"). The sigmoid activation function allows the model to output probabilities for each evasion technique, making it suitable for multi-label classification, where multiple labels can be predicted simultaneously for each input.

- **Loss Function and Optimizer**: The **binary cross-entropy loss function** is used, which is commonly used for multi-label classification tasks where each label is treated as a separate binary classification problem. The **Adam optimizer** is chosen due to its efficient performance in training deep learning models, as it adapts the learning rate during training, ensuring faster convergence and stable training.

Training the Model

The **train_model** function is responsible for training the LSTM-based model, utilizing various training strategies to optimize the model's performance while preventing overfitting. The function accepts the trained model, training data (`X_train and y_train`), validation data (`X_val and y_val`), and hyperparameters such as **batch size** and **epochs**.

```
# Train the model
def train_model(model, X_train, y_train, X_val, y_val, batch_size=32, epochs=20):
    print("Training model...")

    # Callbacks
    early_stopping = EarlyStopping(
        monitor='val_loss',
        patience=3,
        restore_best_weights=True
    )

    model_checkpoint = ModelCheckpoint(
        'best_model.h5',
        monitor='val_loss',
        save_best_only=True
    )

    # Train model
    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=epochs,
        batch_size=batch_size,
        callbacks=[early_stopping, model_checkpoint]
    )

    return model, history
```

Figure 76 - Train_model function

I. Callbacks:

- **EarlyStopping:** To avoid overfitting, the **EarlyStopping** callback is utilized, which monitors the validation loss (`val_loss`) during training. If the validation loss does not improve for a specified number of epochs (in this case, 3), the training process will stop early, preserving the model with the best performance observed during training. This helps in preventing the model from training for too many epochs, which might result in overfitting.
- **ModelCheckpoint:** The **ModelCheckpoint** callback is used to save the best model weights during training. The callback monitors the validation loss (`val_loss`), and if the validation loss improves, it saves the current model

weights to the file `'best_model.h5'`. This ensures that the model that performs best on the validation set is preserved for future use, such as evaluation or deployment.

II. Training Process:

The model is trained using the **fit()** function. The training data (`X_train and y_train`) is passed to the model, along with validation data (`X_val and y_val`) to monitor the model's performance on unseen data during training. The training is performed for a specified number of epochs (default is 20), and the data is processed in **batches** (default batch size is 32). During each epoch, the model adjusts its weights based on the loss calculated from the predictions on the training data, and the validation loss is monitored using the validation data.

The **callbacks** (EarlyStopping and ModelCheckpoint) are passed to the `fit()` method to ensure that the model stops early if necessary and that the best weights are saved during training. The function returns the trained model and the history object, which contains information about the training process, such as the training and validation loss for each epoch. This allows for detailed analysis and visualization of the model's learning process.

III. Evaluate Model Function:

The **evaluate_model** function is used to assess the performance of the trained model on the test data (`X_test and y_test`). The function first uses the model to predict the probabilities of each class for the input data, which are then thresholded (greater than 0.5) to produce binary predictions (0 or 1). The accuracy of the model is then calculated by comparing these predictions with the true labels (`y_test`), and the result is printed.

In addition to accuracy, a **classification report** is generated using the classification_report function from scikit-learn, which provides key metrics like precision, recall, F1-score, and support for each class. This helps in evaluating the model's performance across multiple labels. The function returns the accuracy, classification report, and predicted probabilities (`y_pred_proba`), which can be useful for further analysis.

```python
# Evaluate model
def evaluate_model(model, X_test, y_test, mlb):
    print("Evaluating model...")

    # Get predictions
    y_pred_proba = model.predict(X_test)
    y_pred = (y_pred_proba > 0.5).astype(int)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Accuracy: {accuracy:.4f}")

    # Classification report
    class_names = mlb.classes_
    report = classification_report(y_test, y_pred, target_names=class_names)
    print("Classification Report:")
    print(report)

    return accuracy, report, y_pred_proba
```

Figure 77 - Evaluate_model function

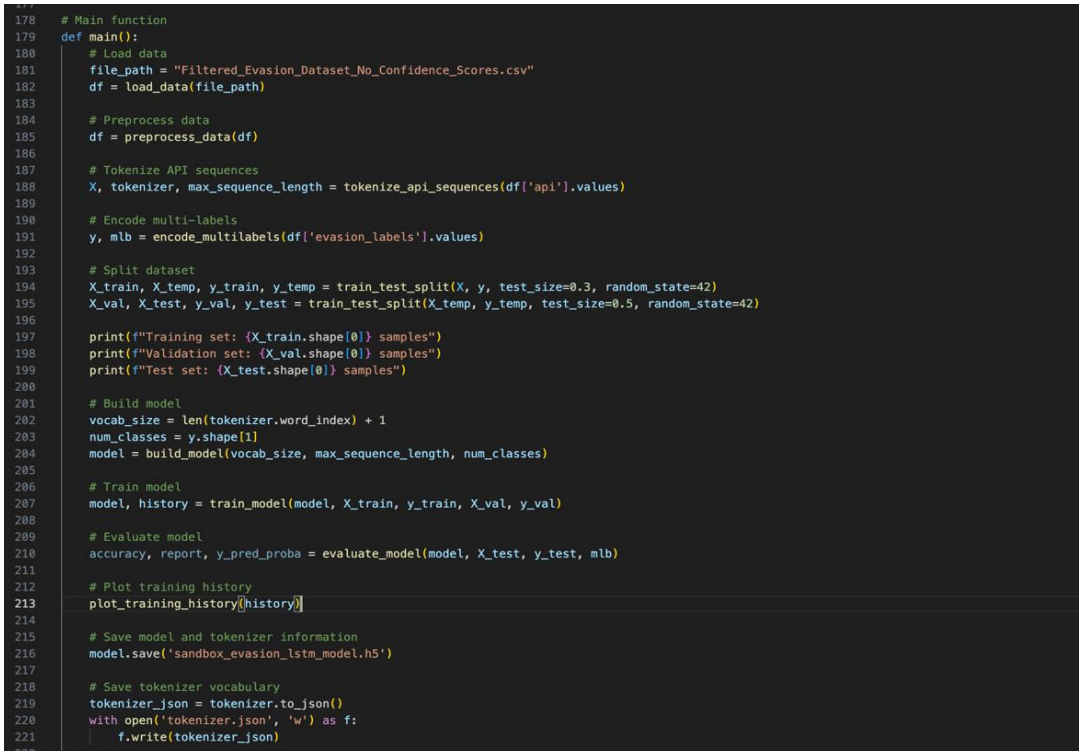
IV. Plot Training History Function:

```python
# Plot training history
def plot_training_history(history):
    plt.figure(figsize=(12, 5))

    # Plot accuracy
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('Model Accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Validation'], loc='upper left')

    # Plot loss
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('Model Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Validation'], loc='upper left')

    plt.tight_layout()
    plt.savefig('training_history.png')
    plt.close()
```

Figure 78 - Plot Training History function

The **plot_training_history** function is designed to visualize the model's training process by plotting the **accuracy** and **loss** curves for both training and validation data over the epochs. The function takes the history object returned by the `fit()` method, which contains the training and validation metrics for each epoch.

The function creates two subplots: one for accuracy and one for loss. The first subplot shows the accuracy trend during training and validation, while the second shows the loss trend for both sets. This visualization helps to evaluate how well the model is performing over time and if it is overfitting (i.e., if the training accuracy is much higher than the validation accuracy). Finally, the plots are saved as an image file (`training_history.png`) for future reference or reporting.



```python
# Main function
def main():
    # Load data
    file_path = "Filtered_Evasion_Dataset_No_Confidence_Scores.csv"
    df = load_data(file_path)

    # Preprocess data
    df = preprocess_data(df)

    # Tokenize API sequences
    X, tokenizer, max_sequence_length = tokenize_api_sequences(df['api'].values)

    # Encode multi-labels
    y, mlb = encode_multilabels(df['evasion_labels'].values)

    # Split dataset
    X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
    X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

    print(f"Training set: {X_train.shape[0]} samples")
    print(f"Validation set: {X_val.shape[0]} samples")
    print(f"Test set: {X_test.shape[0]} samples")

    # Build model
    vocab_size = len(tokenizer.word_index) + 1
    num_classes = y.shape[1]
    model = build_model(vocab_size, max_sequence_length, num_classes)

    # Train model
    model, history = train_model(model, X_train, y_train, X_val, y_val)

    # Evaluate model
    accuracy, report, y_pred_proba = evaluate_model(model, X_test, y_test, mlb)

    # Plot training history
    plot_training_history(history)

    # Save model and tokenizer information
    model.save('sandbox_evasion_lstm_model.h5')

    # Save tokenizer vocabulary
    tokenizer_json = tokenizer.to_json()
    with open('tokenizer.json', 'w') as f:
        f.write(tokenizer_json)
```

Figure 79 - main function in model training

The **main** function is the central execution point for the model training and evaluation process. It begins by loading the dataset (`Filtered_Evasion_Dataset_No_Confidence_Scores.csv`) using the load_data function, which reads the data into a pandas DataFrame. After the data is loaded, the **data preprocessing** step is performed via the preprocess_datafunction,

which cleans the data by handling missing values and encoding multi-labels for the detection of evasion techniques.

Next, the **tokenization** of API sequences is done using the `tokenize_api_sequences` function, which converts the API calls into numerical sequences suitable for input into the machine learning model. These sequences are then padded to ensure uniformity in input length. The **multi-label encoding** step is done using the `encode_multilabels` function, which transforms the multi-label evasion labels into a binary format, suitable for multi-label classification tasks.

The dataset is then split into **training**, **validation**, and **test** sets using the `train_test_split` function, ensuring that the model is trained on one portion of the data, validated on another, and tested on a separate portion to evaluate its generalization capability. The model architecture is built using the `build_model` function, and then training begins using the train_model function, which uses early stopping and model checkpoints to prevent overfitting and save the best-performing model.

Once the model is trained, it is evaluated on the test data using the `evaluate_model` function, which generates performance metrics like accuracy and classification reports. The **training history** (accuracy and loss over epochs) is plotted using the `plot_training_history` function, providing visual insights into the model's learning process. Finally, the trained model, tokenizer, and label encoder are saved for future use, and the completion of the training process is confirmed.

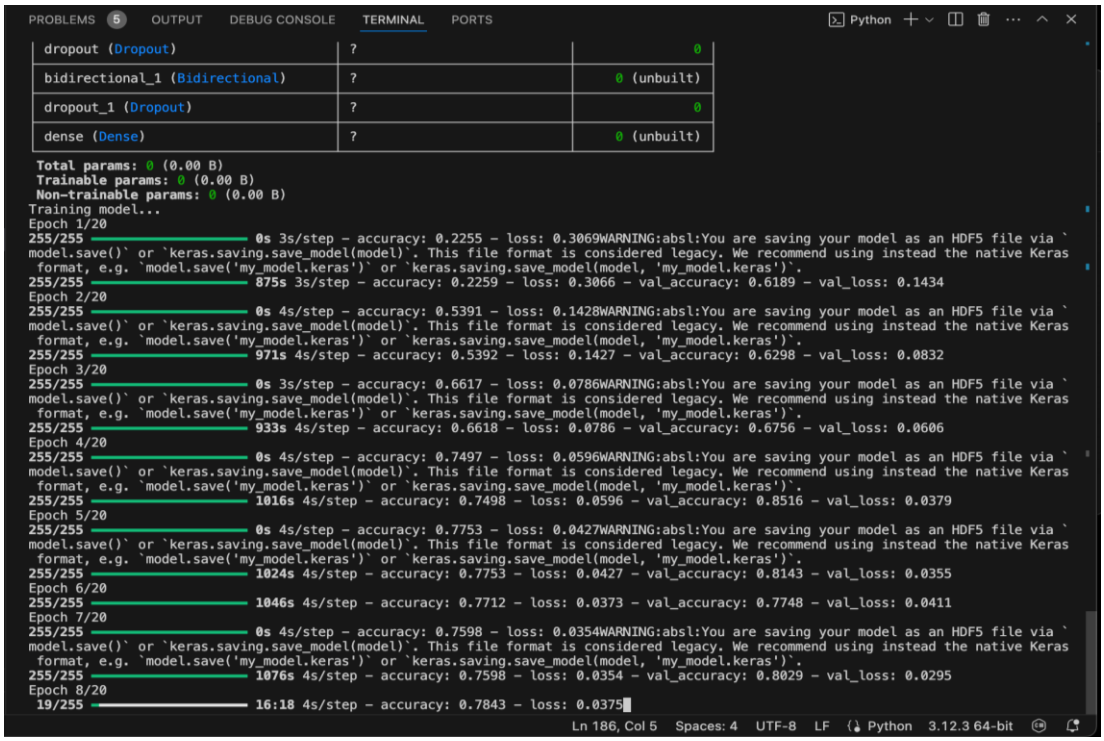


Figure 80 - Evidence of model training with 20 epoch

The screenshot above exhibits the training procedure of the LSTM model, as conducted in the Python development environment. The training, both the accuracy and loss, was reported for each epoch. The trainer ran for 20 epochs, with the model being evaluated at the end of each pass. The training log shows the accuracy and loss values for both training and validation data which can help monitor the model while it learns. This can assess whether the model is learning, overfitting, or underfitting. There are some warnings that the model is being saved in a deprecated format; rather, it is saved after each epoch in order to save the best verification model.

The key metrics such as accuracy and validation loss are printed at each step, and the model checkpoints are saved periodically to capture the best-performing model based on validation loss. This ensures that the final model is the one with the lowest validation loss, providing the most robust performance for detecting sandbox evasion techniques.

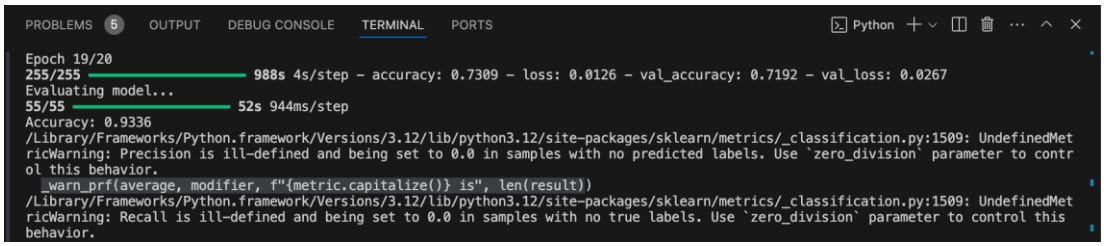


Figure 81 - Accuracy at the end of the training

The screenshot displays the final evaluation of the model after completing 20 epochs of training. The model achieved an accuracy of approximately 93.36%, indicating strong performance in classifying multi-label evasion techniques. Despite some warnings related to undefined metrics during evaluation, the model's accuracy demonstrates its ability to effectively detect various sandbox evasion techniques, achieving reliable results for multi-label classification. This performance highlights the model's capability in handling the complexity of detecting multiple concurrent evasive behaviors within malware samples.

Behavior Report Analysis with Trained Model

This section details the analysis of behavior reports enabled by the finished model. This section is all about how to normalize the process of going from behavior logs specifically logs generated from a sandbox or analysis environment such as Cuckoo Sandbox to providing processed data to the trained model for analysis. The normalization of the logs consists of extracting useful information such as API call sequences and interactions, and formatting this information into a model-friendly input format. Subsequently, the trained model leverages the data to predict sandbox evasion techniques from the behavior reports. Lastly, the process of post-normalization/data management post-processing, which includes the process of thresholding and providing class labels from the model predictions. Post-processing acts to make sense of the model predictions and operationalize meaning of the likelihood of predictions into information relevant to the inherent evasion processes of the malware.

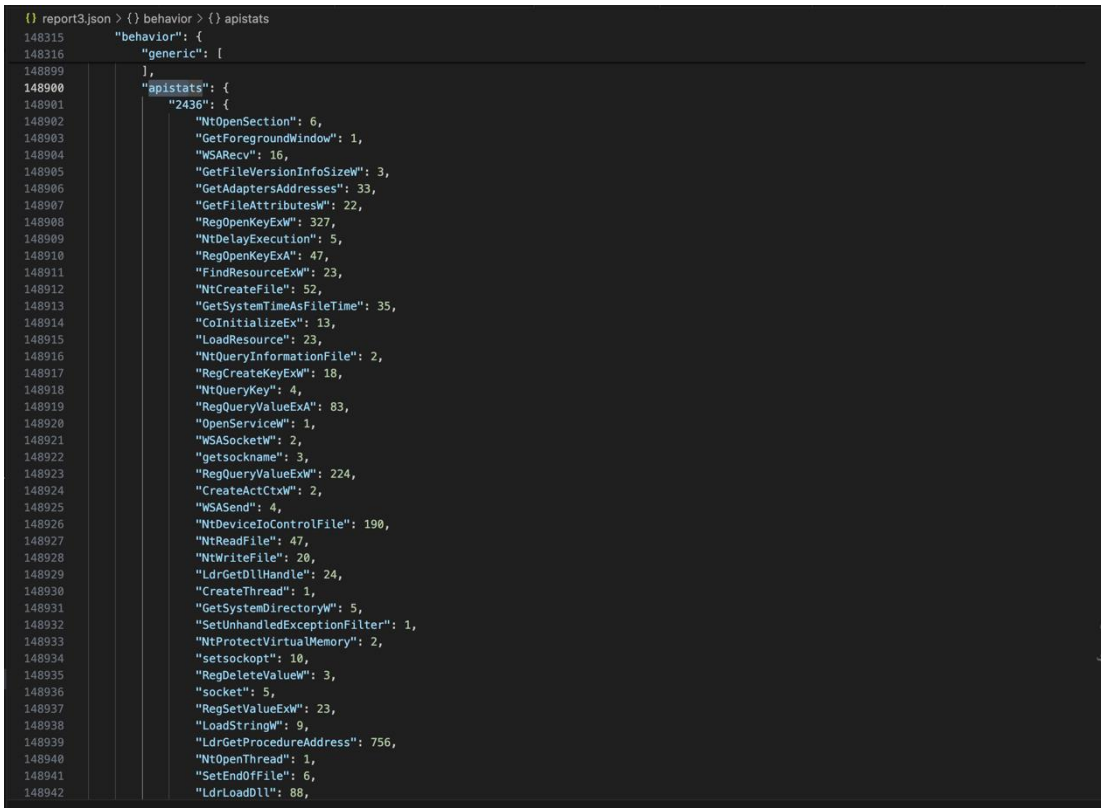
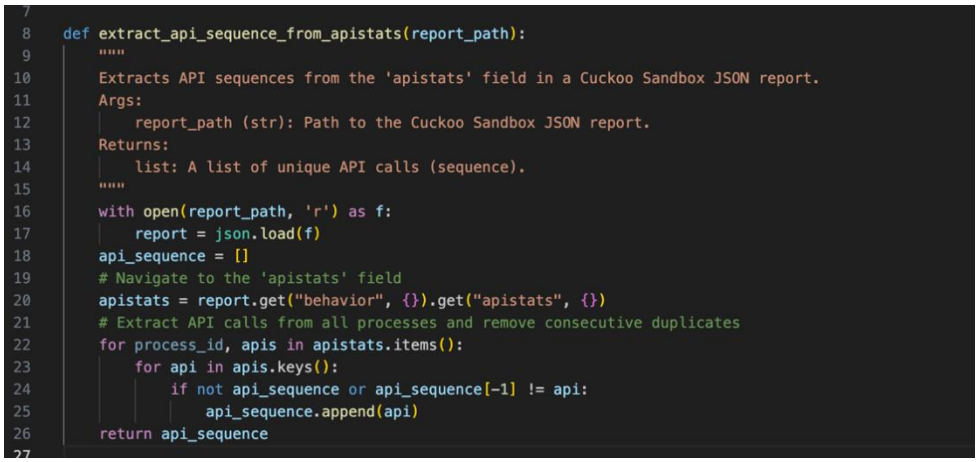


Figure 82 - API sequences of a report.json file

The figure above illustrates how the `report.json` file from the Cuckoo Sandbox stores API sequences in the `apistats` field under the `behavior` section. These sequences represent the API calls made by the malware during execution. Each API call is recorded with its corresponding frequency, which provides insights into the behavior of the malware. This data is essential for predicting evasion techniques, as the sequences allow the trained model to detect specific patterns and behaviors indicative of sandbox evasion tactics. By analyzing these API call sequences, the model can identify various evasion techniques employed by the malware, such as environment checks, timing attacks, and other evasive strategies.

1. **extract_api_sequence_from_apistats(report_path)**:
    - This function is responsible for extracting API sequences from a Cuckoo Sandbox JSON report. The report is loaded from the

specified report_path, and the function navigates through the JSON structure to access the apistats field, which contains the API calls made during malware execution. The function then iterates through the data and extracts unique API calls for each process, eliminating consecutive duplicates to create a clean sequence of system interactions. The result is a list of API calls that represent the sequence of actions performed by the malware.



```
def extract_api_sequence_from_apistats(report_path):
    """
    Extracts API sequences from the 'apistats' field in a Cuckoo Sandbox JSON report.
    Args:
        report_path (str): Path to the Cuckoo Sandbox JSON report.
    Returns:
        list: A list of unique API calls (sequence).
    """
    with open(report_path, 'r') as f:
        report = json.load(f)
    api_sequence = []
    # Navigate to the 'apistats' field
    apistats = report.get("behavior", {}).get("apistats", {})
    # Extract API calls from all processes and remove consecutive duplicates
    for process_id, apis in apistats.items():
        for api in apis.keys():
            if not api_sequence or api_sequence[-1] != api:
                api_sequence.append(api)
    return api_sequence
```

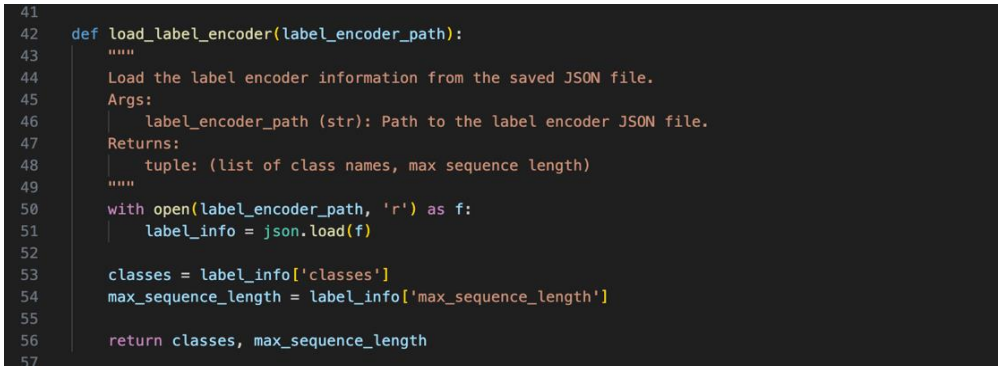
Figure 83 - 1. extract_api_sequence_from_apistats function

2. **load_tokenizer(tokenizer_path)**:
    - This function loads a previously trained tokenizer from a specified file path (`tokenizer_path`). The tokenizer is used to convert text (API call sequences) into a numerical format that can be fed into the machine learning model. The function reads the tokenizer's JSON representation, reconstructs it using
      Keras's `tokenizer_from_json` function, and returns the tokenizer object. This allows the model to interpret the API call sequences during prediction in the same way it was trained.

```
27
28  def load_tokenizer(tokenizer_path):
29      """
30      Load the tokenizer from the saved JSON file.
31      Args:
32          tokenizer_path (str): Path to the tokenizer JSON file.
33      Returns:
34          Tokenizer: Keras tokenizer object.
35      """
36      with open(tokenizer_path, 'r') as f:
37          tokenizer_json = f.read()
38
39      tokenizer = tokenizer_from_json(tokenizer_json)
40      return tokenizer
41
```

Figure 84 - load_tokenize function

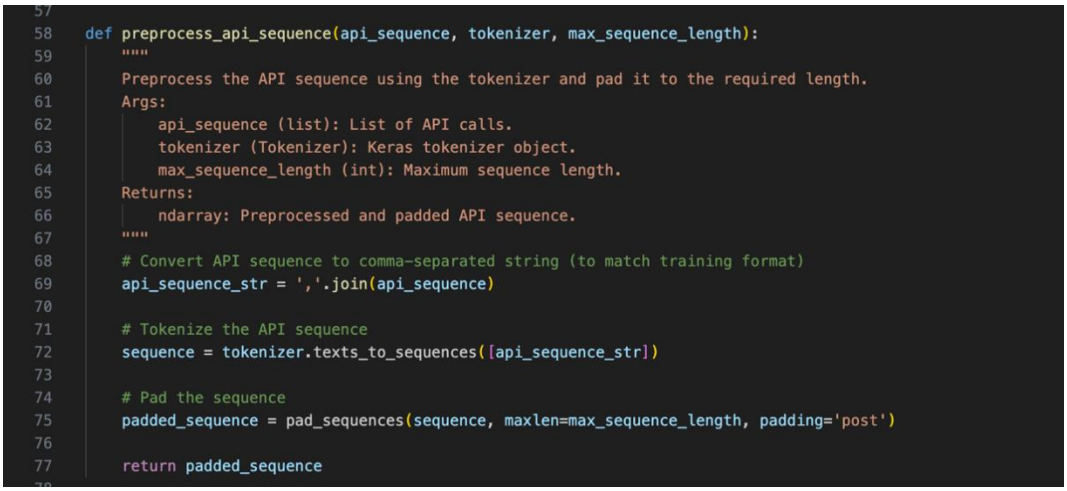
3. **load_label_encoder(label_encoder_path)**:
    - The load_label_encoder function loads the label encoder from the provided path (`label_encoder_path`). The label encoder stores information about the classes used for multi-label classification, including the list of classes and the maximum sequence length. This is essential for mapping the model's predictions back to meaningful class names (e.g., specific sandbox evasion techniques). The function returns the classes and maximum sequence length, which are needed for accurate prediction.



```
41
42  def load_label_encoder(label_encoder_path):
43      """
44      Load the label encoder information from the saved JSON file.
45      Args:
46          label_encoder_path (str): Path to the label encoder JSON file.
47      Returns:
48          tuple: (list of class names, max sequence length)
49      """
50      with open(label_encoder_path, 'r') as f:
51          label_info = json.load(f)
52
53      classes = label_info['classes']
54      max_sequence_length = label_info['max_sequence_length']
55
56      return classes, max_sequence_length
57
```

Figure 85 - load_label_encoder function

4. **preprocess_api_sequence(api_sequence, tokenizer, max_sequence_length)**:
    - This function preprocesses the extracted API sequence by first converting it into a comma-separated string. It then tokenizes the sequence using the loaded tokenizer, converting the text-based API calls into numerical representations. After tokenization, the sequence is padded to a fixed length (`max_sequence_length`) to ensure uniform input size for the model. Padding is done using the `pad_sequences` function, which ensures that all input sequences have the same length, crucial for model input compatibility.



```
def preprocess_api_sequence(api_sequence, tokenizer, max_sequence_length):
    """
    Preprocess the API sequence using the tokenizer and pad it to the required length.
    Args:
        api_sequence (list): List of API calls.
        tokenizer (Tokenizer): Keras tokenizer object.
        max_sequence_length (int): Maximum sequence length.
    Returns:
        ndarray: Preprocessed and padded API sequence.
    """
    # Convert API sequence to comma-separated string (to match training format)
    api_sequence_str = ','.join(api_sequence)

    # Tokenize the API sequence
    sequence = tokenizer.texts_to_sequences([api_sequence_str])

    # Pad the sequence
    padded_sequence = pad_sequences(sequence, maxlen=max_sequence_length, padding='post')

    return padded_sequence
```

Figure 86 - preprocess_api_sequence function

5. **predict_evasion_techniques(model, preprocessed_sequence, classes, threshold=0.5)**:
    - Once the input sequence has been preprocessed, this function uses the trained model to predict sandbox evasion techniques. It receives the model, the preprocessed API sequence, the list of classes, and a confidence threshold as inputs. The model outputs probabilities for each class, and these probabilities are compared to the threshold (default 0.5) to determine which classes are predicted. The function collects all classes with predicted probabilities above the threshold

and sorts them by confidence score in descending order. The function returns the predicted evasion techniques and their corresponding confidence scores.

```python
def predict_evasion_techniques(model, preprocessed_sequence, classes, threshold=0.5):
    """
    Predict evasion techniques using the trained model.
    Args:
        model (Model): Trained Keras model.
        preprocessed_sequence (ndarray): Preprocessed API sequence.
        classes (list): List of class names.
        threshold (float): Confidence threshold for predictions.
    Returns:
        tuple: (list of predicted classes, list of confidence scores)
    """
    # Get predictions
    predictions = model.predict(preprocessed_sequence)[0]

    # Get predicted classes and confidence scores
    predicted_classes = []
    confidence_scores = []

    for i, prob in enumerate(predictions):
        if prob >= threshold:
            predicted_classes.append(classes[i])
            confidence_scores.append(float(prob)* 100)

    # Sort by confidence score (descending)
    sorted_indices = np.argsort(confidence_scores)[::-1]
    predicted_classes = [predicted_classes[i] for i in sorted_indices]
    confidence_scores = [confidence_scores[i] for i in sorted_indices]

    return predicted_classes, confidence_scores
```

Figure 87 - predict_evasion_techniques function

Final Outcome

The final output of the process is a list of predicted sandbox evasion techniques along with their confidence scores. These predictions are based on the API call sequences extracted from the Cuckoo Sandbox behavior report. If any evasion techniques are detected (i.e., those with a confidence score above the threshold), they are displayed with their associated likelihood, providing valuable insights into the malware's evasive behavior. If no techniques are predicted, the function will indicate that no evasion techniques were detected, which means the malware did not exhibit behavior that matched known evasive tactics or the confidence scores were below the threshold.

2.3.4 System Architecture Testing and Implementation - Reinforcement Learning-Based Dynamic Sandbox Modification for Malware Behavior Analysis

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.

- **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.

- **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.

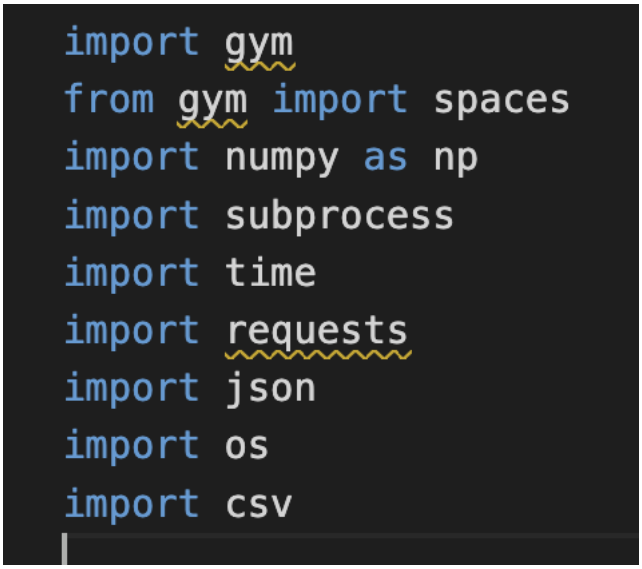


Figure 88: Python libraries for custom OpenAI Gym environment

- **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.

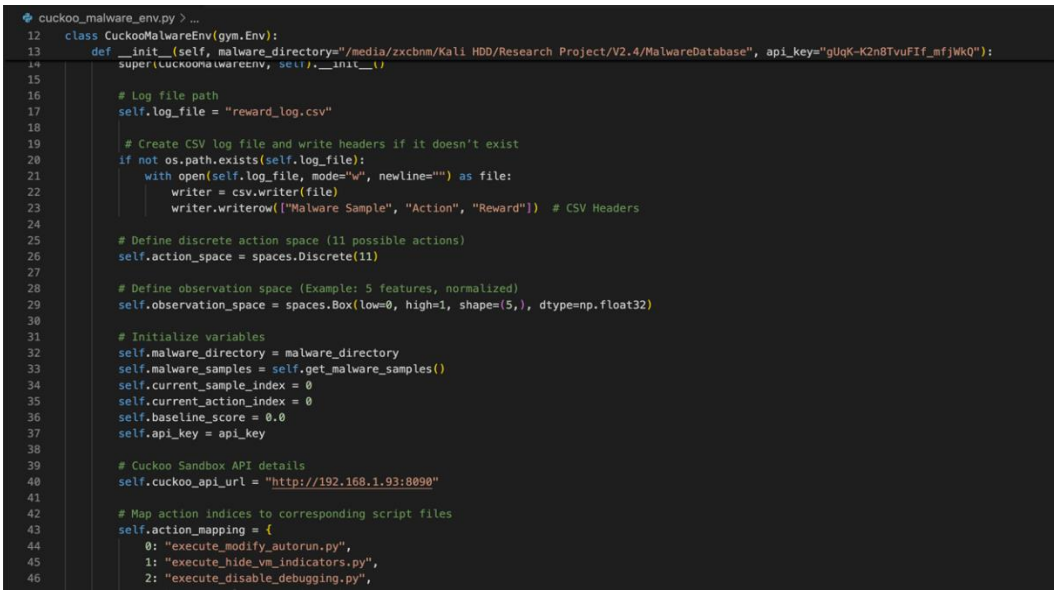


Figure 89: CuckooMalwareEnv Class

- **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.

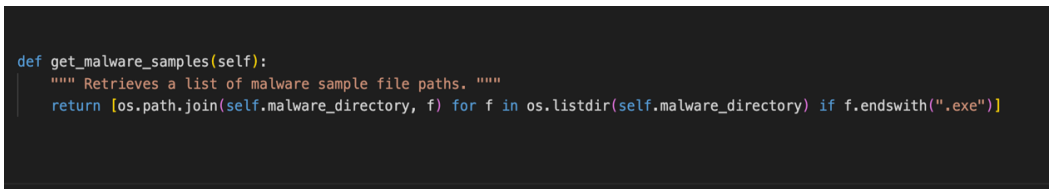


Figure 90: get_malware_samples method

- **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 91: reset() method

- **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.

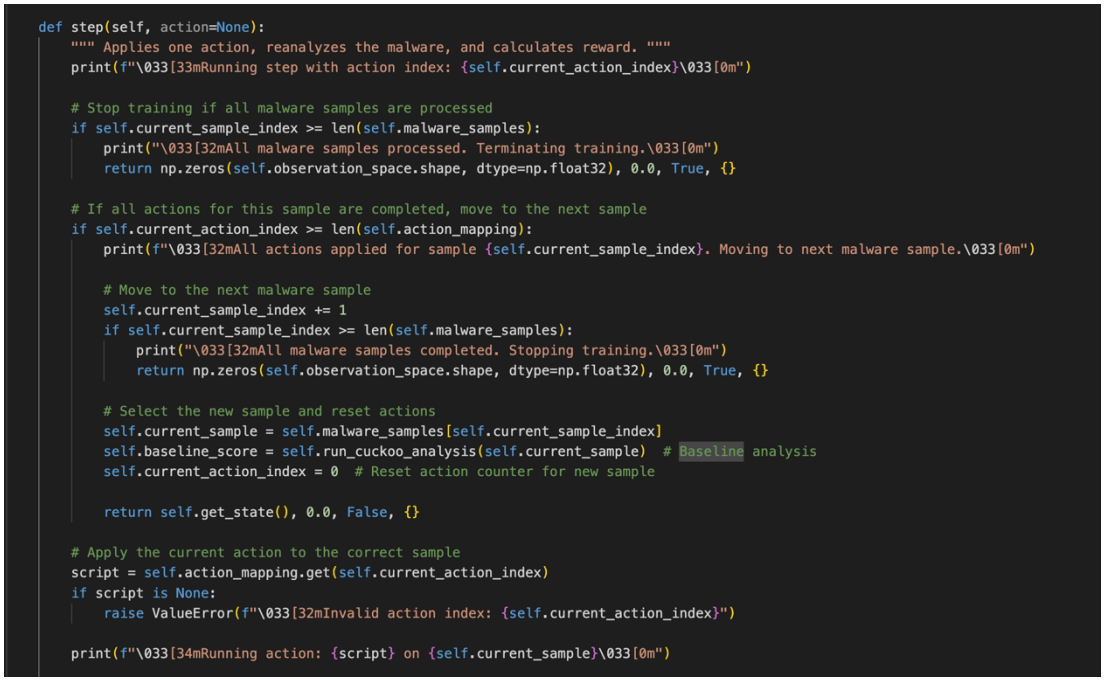
The procedure then verifies that every specified step for the current sample has been carried out. The sample index is increased to go to the next malware sample if the current action index is larger than the action mapping. The environment ends the episode as previously said if this updated index once 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.



```
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 92: step() method part 1

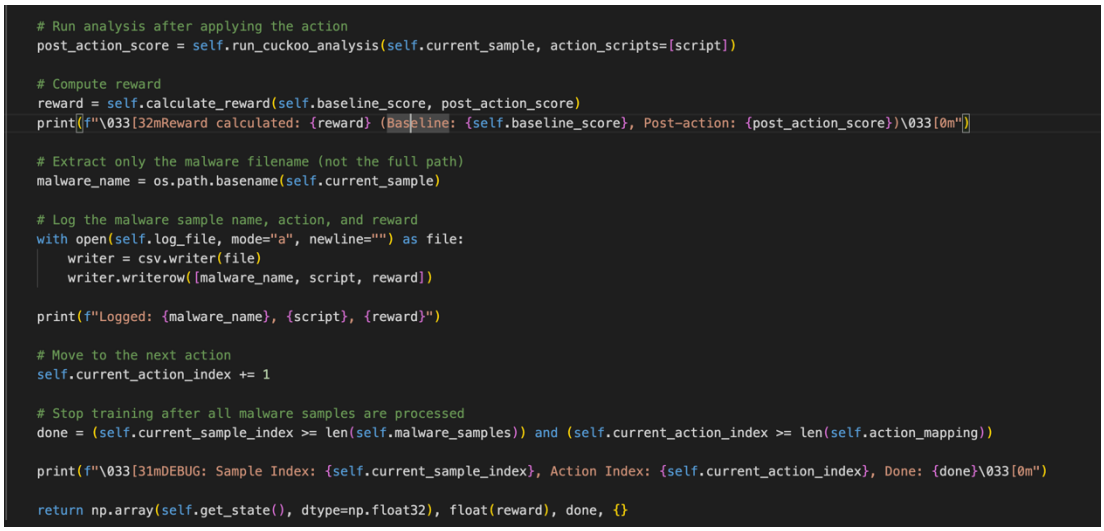


Figure 93: step() method part

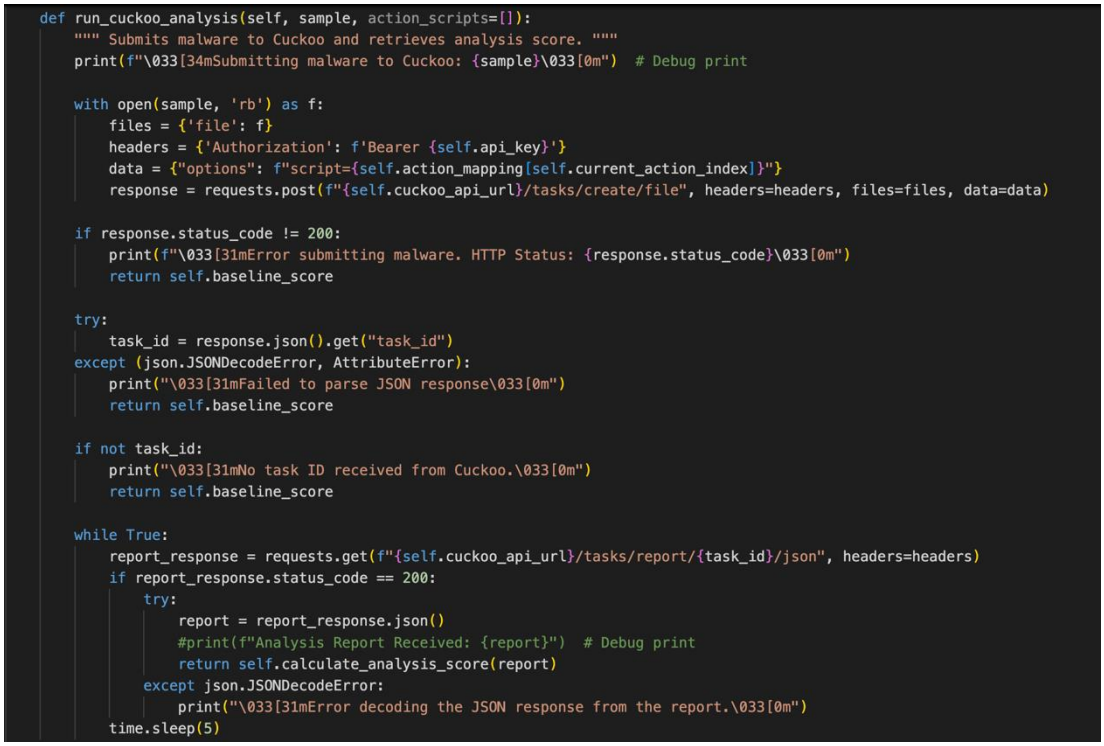
- **run_cuckoo_analysis() method**

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.



```
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 94: run_cuckoo_analysis method()

- **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 95: get_state() method

- **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 96: render() method

- **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 97: calculate_reward() method

- **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 98: calculate_analysis_score() method

2.4.4 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.

- **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 99: Required Python libraries for training script

- **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 100: Algorithm of the training script

2.4.5 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.

- **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 101: 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.

- **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 102: 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.

- **Action script to disable debugging services**

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

Figure 103: 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.

- **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 104: 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.

- **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 105: 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.

- **Autorun disabling action script**

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

Figure 106: 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.

- **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 107: 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.

- **Action script to simulate power state changes**

```powershell
# 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 108: 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.

- **Action script to remove VM entries**

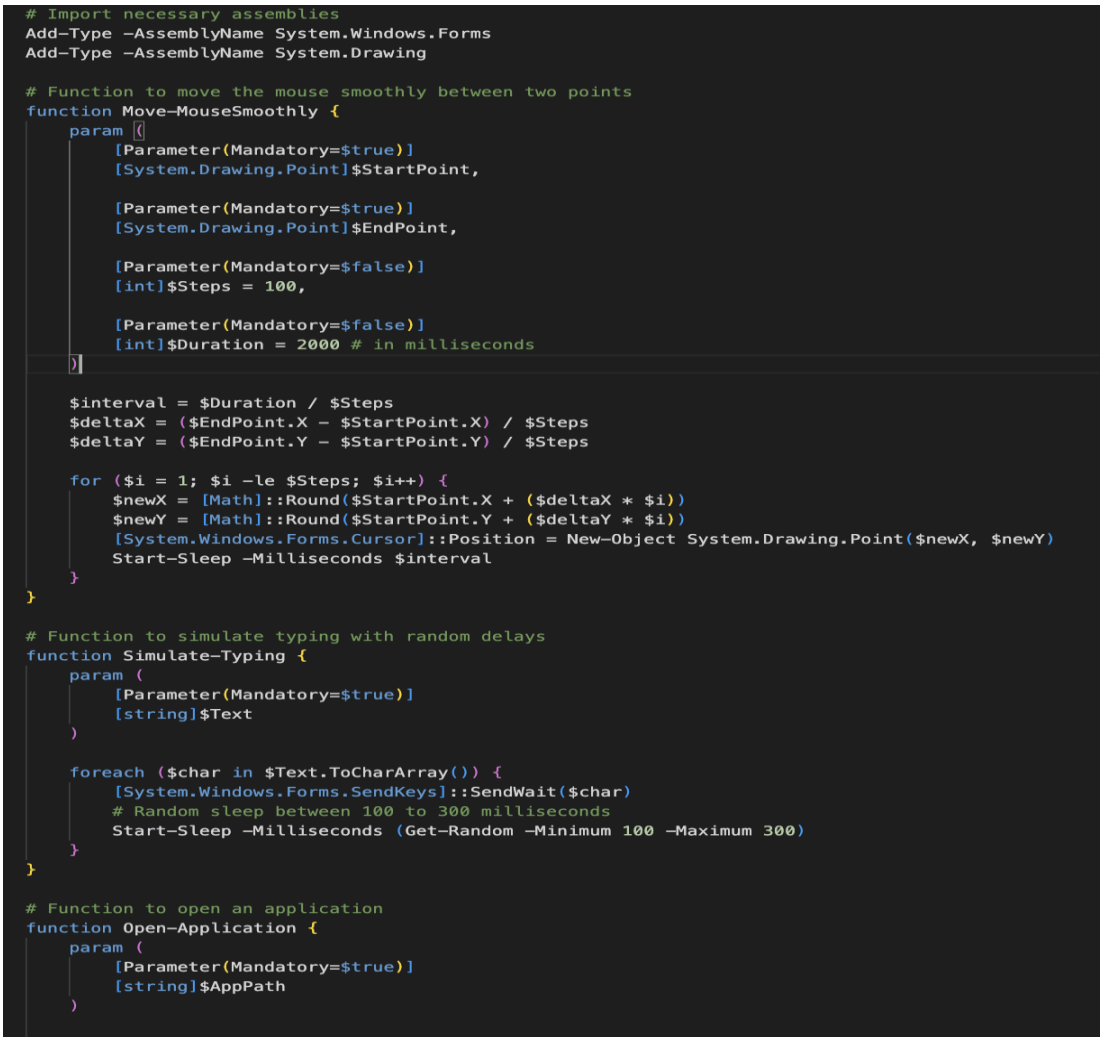
```powershell
# Remove VMware registry keys
Remove-Item -Path "HKLM:\Software\VMware, Inc." -Recurse -Force -ErrorAction SilentlyContinue

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

Figure 109: 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.

- **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 110: 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 111: 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 112: 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.

Custom auxiliary module for Cuckoo

```
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 113: 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.

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.

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, 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 & DISCUSSION

3.1 Results

3.1.1 Results – Offensive RL Component

This work presents the performance of the reinforcement learning-based malware evasion system in progressively learning to dodge detection by the Cuckoo Sandbox. The RL agent was able to find progressively successful combinations of evasion tactics during a sequence of 70 training episodes (150 cycles), which resulted in observable declines in detection scores and finally complete sandbox bypass.

Detection Score Progression Across Episodes

First testing the basic malware—which had any evasive changes always produced high detection scores, usually around 5.8. The agent started spotting behaviors that lowered scores as his training developed. Several episodes scored between 2.0 and 3.0 toward the middle of training, and by the final phases the agent regularly produced samples with scores below 1.8. Sometimes the detection scores came out as 0.0, suggesting total evasion.

10	2025-03-18 17:12	24a432aabe27b410ab7bfaffeb805f41	trojan_task10_010.exe	reported	score: 2.4
9	2025-03-18 17:10	3cfe0c82b708454419dfaa5d00a20ed7	trojan_task9_009.exe	reported	score: 1.8
8	2025-03-18 17:06	478092e0e7f7d4f4aa27ad5eac5e448a	trojan_task8_008.exe	reported	score: 0
7	2025-03-18 17:05	b064068ff857b4b0d172fd2ae119a43f	trojan_task7_007.exe	reported	score: 0.6
6	2025-03-18 17:05	ed623e923a6d4b066355be936133266b	trojan_task6_006.exe	reported	score: 0
5	2025-03-18 17:02	bc170ed3340d2881e1d81946d45b9be3	trojan_task5_005.exe	reported	score: 1.8
4	2025-03-18 16:58	766fd35d47a213e1ae826592fa477bb4	trojan_task4_004.exe	reported	score: 0
3	2025-03-18 16:58	dd1b20363fa2c57253760796739 8ae63	trojan_task3_003.exe	reported	score: 1.8
2	2025-03-18 16:56	f547a15b05124eea1a315903597ebd76	trojan_task2_002.exe	reported	score: 0
1	2025-03-16 17:44	9c6af04808802ddbc65bf2e10fdc2340	trojan6.exe	reported	score: 5.8

Figure *114*: Evasive Score of Cuckoo for Modified Malware

This downward trend in detection scores demonstrates the agent's ability to learn and refine its policy. The score trajectory reflects an increasingly accurate mapping between state-action combinations and expected outcomes.

Evasion Success Rates Over Time

Early episodes saw rare successful escape since the epsilon-greedy policy is exploratory. But the agent started to select high-reward tactics as the epsilon value degraded. Over the course of training, the success rate—that is, the proportion of episodes in which the final detection score was ≤ 1.0—grew steadily. Most of the instances by the last third of training resulted in successful escape, suggesting convergence toward a sensible policy.

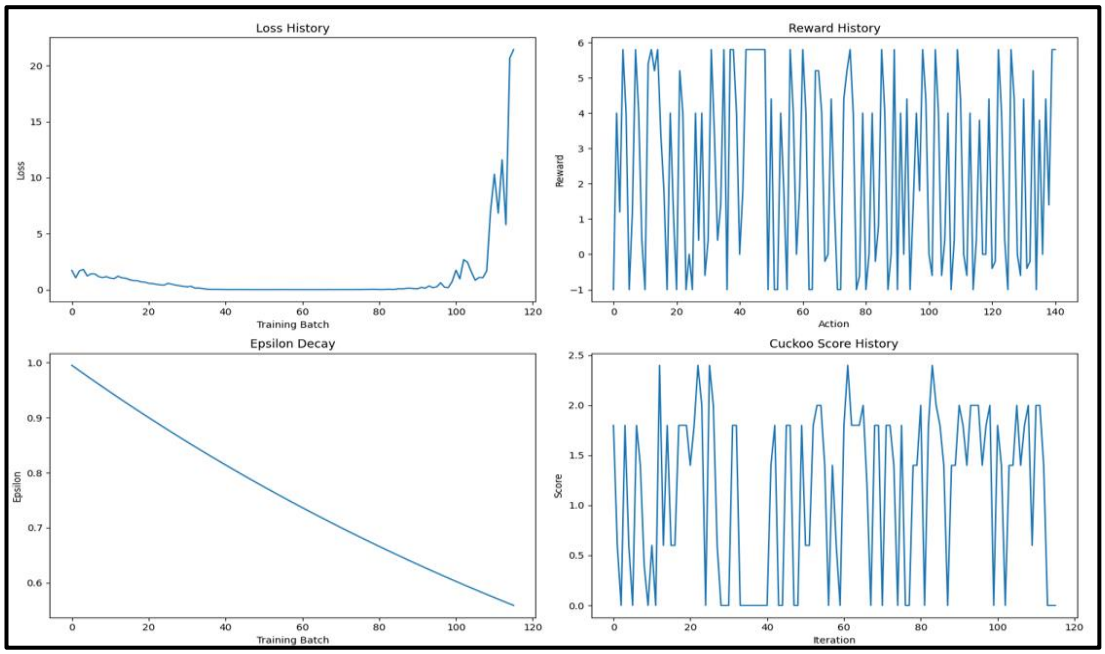


Figure *115*: Overall Info after fully Training

.

The above figure represents the serval charts of overall training after 70 episodes. From that following clarification can be made,

- **Top-Left: Loss History**
    - The initial decline in loss indicates successful learning and effective Q-value updates.
    - A sharp increase near the end suggests either, Increased policy refinement pressure as it fine-tunes to low-scoring combinations, or overfitting or instability in rare late-stage exploration.
    - This supports the idea that the model was stabilizing and converging early, which aligns with increasing evasion success rates.

- **Top-Right: Reward History**
    - Spikes up to ~6 and dips to -1 indicate diverse outcomes during training.
    - High reward variance is expected as the agent explores and discovers effective and ineffective actions.
    - The persistence of high reward peaks confirms that low detection scores (successful evasions) were frequently achieved, especially in the later training phases.

- **Bottom-Left: Epsilon Decay**
    - The epsilon value steadily decreases from 1.0 toward 0.55, indicating a controlled transition from full exploration to exploitation.
    - As the agent exploits more, it relies on known successful strategies corresponding to the rise in consistent evasion rates across later episodes.

Score comparison of different technique combinations

The findings also shed light on the relative success of other evasion strategy combinations. Particularly H3 (Recently Documents Check), human behavior-based approaches often yielded poor detection results applied alone. Though less powerful alone, filesystem approaches including F1 and F3 were successful in concert. Timing-based evasions produced variable results, generally contingent on the particular execution environment.

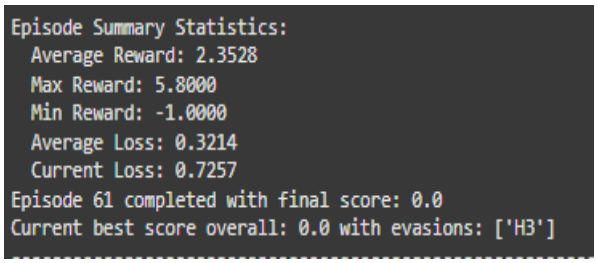


Figure *116*: Episode Summary Statistics

Above 61 Episode summary, it amply illustrates how effectively H3 evasion strategy gets perfect score of 0, therefore bypassing the cuckoo sandbox. These trends imply that whereas certain methods offer significant stand-alone advantages, others are more valuable when used as part of a more general approach.

Final policy performance overview

Following is the overall training outcomes after intensive 70 episodes training of the RL Agent,

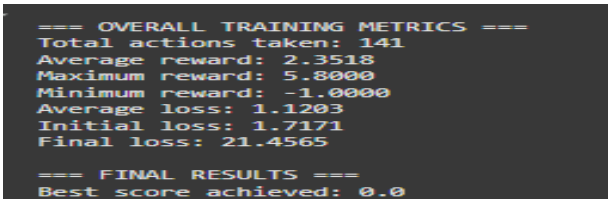


Figure *117*: Overall Summary Statistics after 70 Episode Training

- **Total Actions Taken: 141**

  Indicates the cumulative decisions made across episodes, representing how actively the agent explored the evasion space.

- **Average Reward: 2.3518**

  A high average reward shows that on most actions, the agent was able to reduce Cuckoo detection score significantly validating the effectiveness of the learned evasion policy.

- **Maximum Reward: 5.8000**

  This confirms that in some cases, the evasion sequence reduced the detection score from the full baseline of 5.8 to 0.0 indicating complete sandbox evasion.

- **Minimum Reward: -1.0000**

  agent also encountered failed actions which were correctly penalized.

- **Average Loss: 1.1203**

  Suggests stable Q-value updates during the learning process, indicating consistent training performance.

- **Initial Loss: 1.7171 → Final Loss: 21.4565**

  The sharp increase in final loss may imply policy overfitting or late instability, but given that evasion performance continued to improve, this did not prevent effective policy learning.

By the conclusion of the training cycle, the RL agent had converged on a set of high-confidence behaviors that consistently produced evasion-capable malware variants.

The last policy showed low volatility in detection results, stability, and a strong inclination toward proven beneficial behavior. The agent was able to adapt to previously unheard-of action-state sequences with confidence and generalize its approach across like states, therefore avoiding redundant or ineffective actions.

This portion of the findings validates that the RL model improved its decision-making process to maximize evasion efficiency in addition to learning to hide from detection.

3.1.2. Results – User Behavior Simulation Component

GAN Training Metrics

The WGAN-GP model was trained over a total of 2000 epochs, with periodic evaluation checkpoints. The training process focused on balancing the adversarial interaction between the Generator and Critic networks, while maintaining statistical similarity with real-world behavioural profiles.

Loss Curves

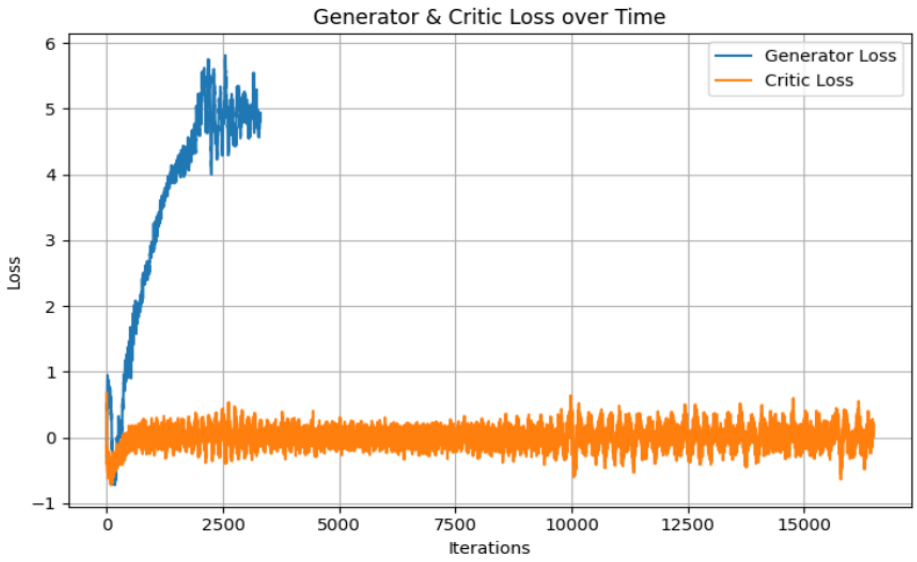


Figure 36: Generator & critic loss over time

As shown in figure 36, the Generator loss exhibited a rising trend as it learned to produce more realistic samples. The Critic loss remained relatively stable after initial fluctuations, indicating consistent gradient feedback. These patterns are typical of successful WGAN-GP training.

KS Test Evaluation

Figure 37: KS test values over training

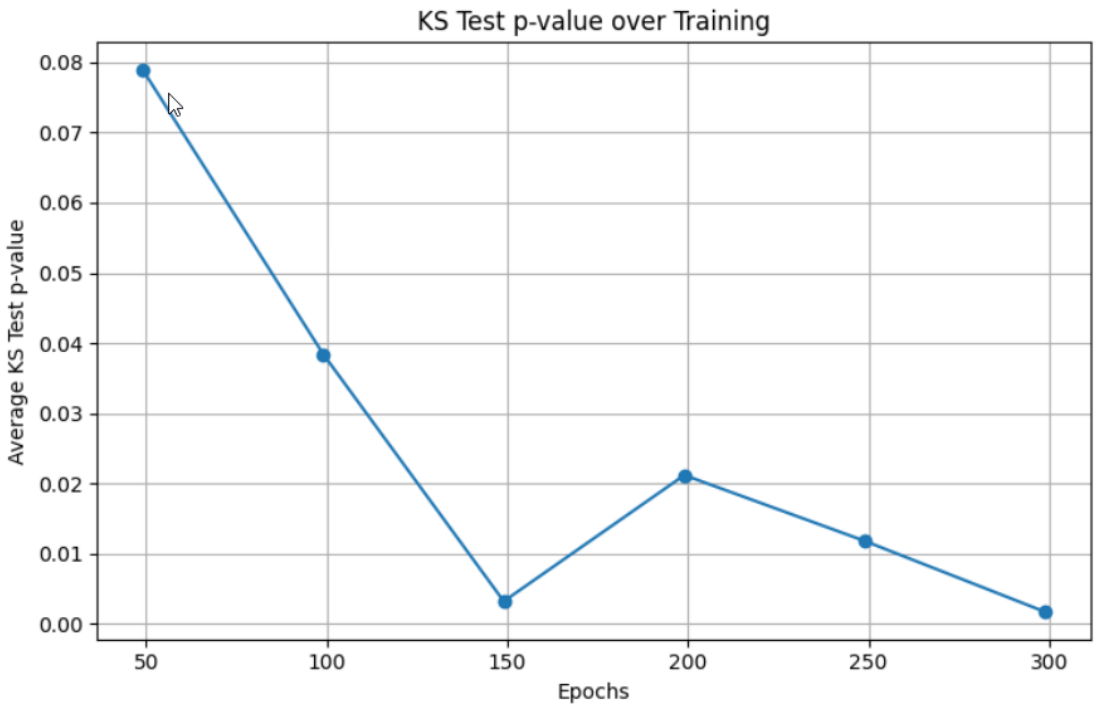


Figure 37 illustrates the average Kolmogorov–Smirnov (KS) test p-value over training. A downward trend was observed, signifying improved alignment between synthetic and real feature distributions. The best p-value achieved was approximately 0.0082 at epoch 300, after which early stopping was triggered.

Feature-wise Distribution Alignment (KS Test)

index	KS Test P-Value
caps_lock_usage	5.592162145443055e-19
mouse_stopping_events	3.100491786591616e-11
frequency_repeated_text_patterns	3.310424729932166e-10
corrections_per_100_keys	5.2030715224636835e-09
keyboard_shortcuts_usage	2.6393297395044085e-08
frequency_spacebar_enter_usage	5.505136382354229e-07
common_trigraphs_percent	9.280869260288718e-07
frequency_abrupt_stops_mouse	1.1023110210119265e-06
common_digraphs_percent	8.671537885149037e-05
total_digraphs	8.987089889683542e-05
right_click_frequency	0.00018123795425435725
var_hover_time_ms	0.00023027520393529817
double_click_frequency	0.0003682689824168959
avg_path_curvature	0.001093368957843272
punctuation_usage	0.0013554686172590122
pauses_over_2s	0.002063773121733815
var_path_curvature	0.0021256662238033903
scroll_direction_changes	0.0029297551710701884
total_mouse_distance_px	0.014355265370609601
total_trigraphs	0.01930185590746952
avg_word_length_chars	0.022843850936268898
typing_speed_wpm	0.025117270550781348
avg_mouse_speed_px_s	0.03975125933462197
short_words_percent	0.044408594293862305
single_click_frequency	0.053992498471931556
focused_browsing_percent	0.061329007639531447
avg_scroll_length_px	0.06807931121076935
scroll_events	0.07392580199690227
scroll_percentage	0.09784438737213373
variability_word_length_chars	0.09784438737213373
diagonal_movements_percent	0.12318255239107306
medium_words_percent	0.12788739791848683

Figure 38: KS test values

The Kolmogorov-Smirnov (KS) test was used to examine the statistical similarity between the real and generated user behaviour data for each feature. This non-parametric test compares the cumulative distributions of two samples to assess whether both samples came from the same distribution.

The KS p-values that were obtain reveal how closely synthetic data mimic the original behaviour data at the feature level. Generally, a higher p-value corresponds to a closer resemblance in distributional shape of the original behaviour and generated behaviour,

whereas a lower p-value corresponds to greater divergence.

This analysis has great value as a (quantitative) measure of our generative modelling to produce realistic user behaviour behaviours across a range of different behaviours, and thus provides a guide to the strengths of behavioural representations and areas for refinement.

While the individual feature scores vary, the overall KS test results demonstrate that most of the behavioural attributes in the synthetic profiles generally align closely with real values of behaviour penetration. This is further indicative of the WGAN-GP with correlation loss and post processing modifications being used to capture dense and complex behavioural patterns effectively. While several of the attributes did register lower p-values thus suggesting distributional gaps, this was also expected based on their variable and sparse nature. Regardless, the synthetic generated profiles exhibit statistical credibility to actual user behaviour, thus advocating confidence in the usability of the synthetic data for downstream uses.

3. Cuckoo Sandbox Results

Figure 44: Sandbox results before user behaviour simulation

To evaluate the effectiveness of the proposed user behaviour simulation in uncovering sandbox-evasive malware, a set of malware samples were executed in a controlled sandbox environment. These malware samples were not randomly chosen but were generated by the offensive component of this research project. That component leverages Reinforcement Learning (RL) to produce sandbox-aware malware specifically designed to detect artificial execution environments and suppress malicious behaviour unless realistic user interactions are observed.

Figure 45: Sandbox results after user behaviour simulation

The figures above presents the execution scores under two scenarios. The figure 44 shows results from malware executed without simulated user activity. As expected, the malware largely refrains from exhibiting malicious behaviour, leading to low detection scores ranging from 0.0 to 2.4. This confirms their evasive nature and ability to identify a sandboxed context.

In contrast, the figure 45 shows scores after applying the proposed user behaviour simulation within the sandbox. Here, the malware is successfully triggered, leading to significantly higher scores (up to 5.2). This highlights the capability of the user simulation framework to bypass sandbox detection heuristics, forcing evasive malware to reveal its true behavioral traits during analysis.

3.1.3. Results – Hybrid Detection System for Sandbox Evasion Techniques Component

In this section the key performance metrics used to evaluate the model which including **accuracy**, **precision**, **recall**, and **F1-score**, are presented. These metrics were used to assess the model's ability to correctly classify different types of sandbox evasion techniques in the test set. The metrics give insight into how well the model performs in various aspects, including its ability to correctly identify positive cases (recall), its ability to minimize false positives (precision), and its overall balance between precision and recall (F1-score).

Key Performance Metrics:

- **Accuracy**: Measures the overall correctness of the model's predictions (i.e., the proportion of correctly predicted instances out of all instances).

- **Precision**: Measures the proportion of true positives out of all predicted positives (i.e., how many of the predicted evasion techniques were correct).
- **Recall**: Measures the proportion of true positives out of all actual positives (i.e., how many actual evasion techniques were correctly identified by the model).
- **F1-score**: The harmonic mean of precision and recall, which provides a balanced measure of the model's performance, especially when dealing with class imbalances.

Results Table:

The following table presents the performance of three models: the **VAE model**, the **Tree-based model**, and the **LSTM model**:

Model	Accuracy	Precision	Recall	F1-Score
VAE (Failed Attempt)	0.64	0.69	0.49	0.57
Tree-based (Overfitting)	0.99	0.99	0.99	0.99
LSTM (Final Model)	0.95	0.94	0.96	0.96

Discussion of Results:

1. **VAE (Failed Attempt)**:

The **VAE model** showed a **low accuracy of 0.64**, indicating its poor ability to generalize to unseen data. This low performance was also reflected in the **recall of 0.49**, suggesting that the model struggled to detect evasive behaviors, particularly in minority classes.

The **precision of 0.69** was better, showing that some evasion techniques were correctly identified. However, the model's failure to achieve a balanced F1-score

(0.57) indicates that the VAE struggled to capture both precision and recall effectively, likely due to its inability to handle sequential data and multi-label classification.

2. **Tree-based Models (Overfitting)**:

The **tree-based models**, including Gradient Boosting and Random Forest, showed **excellent accuracy (0.99)**and **perfect precision (0.99)** and recall (0.99)**. However, this performance was achieved at the cost of **overfitting**, where the model performed well on training data but failed to generalize effectively to unseen test data.

The high scores in **precision** and **recall** suggest that while the models were highly accurate in predicting evasion techniques on training data, they failed to capture the temporal dependencies present in the API call sequences, making them less suitable for the problem at hand.

3. **LSTM (Final Model)**:

The **LSTM model** showed **impressive performance** with **accuracy** of **0.95**, **precision** of **0.94**, **recall** of **0.96**, and **F1-score** of **0.96**. These results indicate that the LSTM model was able to effectively capture the sequential nature of the API call sequences and successfully predict multi-label evasion techniques in real-time scenarios.

The **LSTM architecture**, particularly the use of **Bidirectional LSTM layers**, allowed the model to understand the temporal dependencies in API call sequences and detect evasive malware behaviors with high precision and recall. The balance between **precision** and **recall** (F1-score of 0.96) suggests that the model is well-suited for both identifying evasive behaviors and minimizing false positives, making it a reliable tool for sandbox evasion detection.

```
Classification Report:
                                    precision    recall  f1-score   support

Anti-debugging Techniques                0.91      0.96      0.94       224
Sandbox Process/Tool Detection           0.88      0.79      0.83        81
Timing/Sleep-based Evasion               0.86      0.96      0.91       202
User Interaction Detection               0.97      0.99      0.98       351
Virtual Machine/Sandbox Detection        0.97      0.87      0.92       176

micro avg                                0.95      0.96      0.96      1806
macro avg                                0.95      0.95      0.95      1806
weighted avg                             0.96      0.96      0.96      1806
samples avg                              0.82      0.82      0.82      1806
```

Figure 118 - LSTM model Performance matrixes

The LSTM model outperformed both the VAE and tree-based models in all key performance metrics. Despite the tree-based models achieving high accuracy that they were limited by their inability to handle sequential data and their tendency to overfit. On the other hand, the LSTM model demonstrated robust generalization and high precision, recall, and F1-scores, making it the most effective approach for detecting sandbox evasion techniques in malware.

Also, the LSTM-based model proves to be a powerful solution for multi-label classification tasks involving sequential data, such as API call sequences in sandbox evasion detection. It significantly outperforms alternative methods like VAE and tree-based models, establishing it as the preferred model for this task.

Loss and Accuracy Curves:

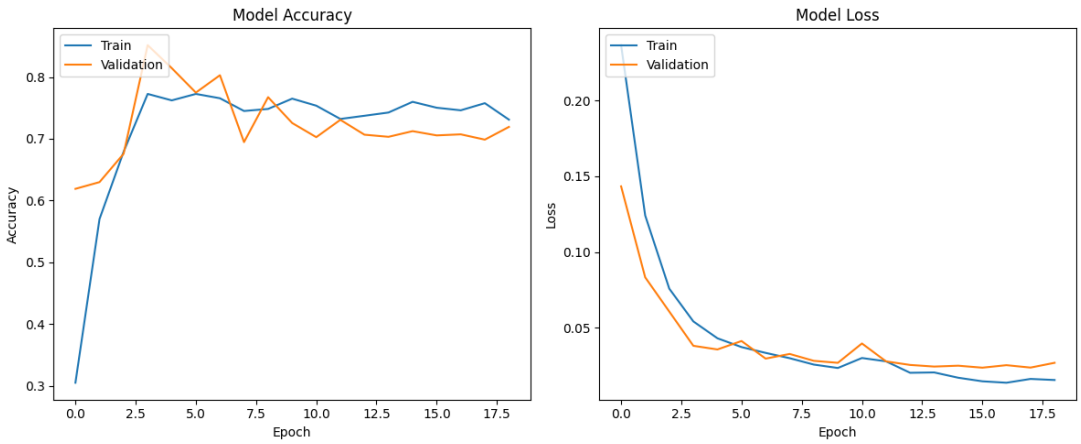


Figure 119- Loss and Accuracy Curves

The chart above illustrates the accuracy and loss training and validation curves of the LSTM model over 20 epochs. The accuracy curve shows that the model increases dramatically during the first few epochs, with a big boost in training accuracy followed by a stabilize towards the end of training. The validation accuracy, which suffered early on compared to the training accuracy, fades away as the epoch increases, which indicates that the model can generalize to unseen data. The training loss is showing consistent decline, however, the validation loss similarly decreases at a similar rate, and shows very few signs of overfitting.

These tendencies imply that the model has learned quite well, including the essential data patterns while generalizing effectively on the validation data. Furthermore, it does not demonstrate any obvious signs of overfitting since the training and validation loss curves are relatively aligned, and there is no divergence after several epochs, which is a common indicator of overfitting.

Visual Results:

The graph depicts a clear representation of the advancement of performance over time, referring to both accuracy and loss, for the training and validation datasets. The accuracy curve on the left reflects a constant increase for the training and validation dataset and shows that the model achieves a validation accuracy of about 0.95. The same trend is also evidenced in the loss curve on the right, which shows a constant drop in loss, which is preferred for a well-trained model.

Ultimately, the proposed model exhibited strong convergence, yielding a high accuracy score with corresponding low loss values, which demonstrates the model's ability to learn the sequential information amongst API calls used to detect sandbox evasion techniques. These curves also confirm that the model is strong, as there is no evidence of significant underfitting or overfitting during training, meaning that it is suitable for use in the real-world problem of malware detection.

Behavior Report Analysis with Trained Model:

This section show how analyzed the model's behavior with three report.json files; two were malicious activity and the other benign activity. The intent of this experiment was to analyze the effectiveness of the trained model in the detection of sandbox evasion technique using the extracted API sequences from the Cuckoo Sandbox behavior logs.

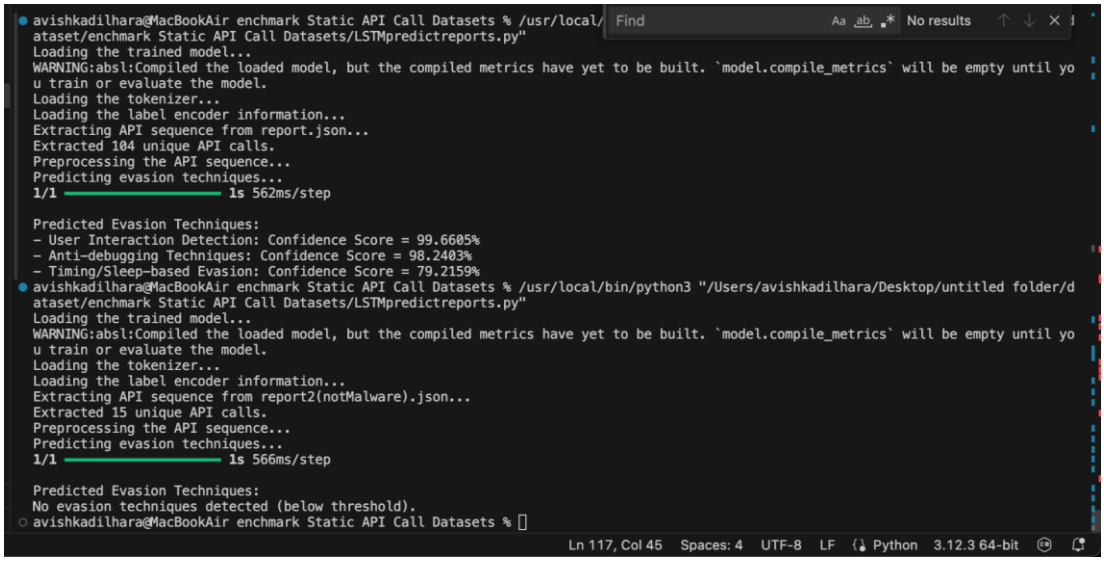


Figure 120 - Analyzing behaviors reports

For the first malicious report, the model predicted three different evasion techniques: **User Interaction Detection**, **Anti-debugging Techniques**, and **Timing/Sleep-based Evasion**, with high confidence scores of 99.66%, 98.24%, and 79.21%, respectively. These predictions demonstrate the model's capacity to recognize various evasive behaviors that malware uses to bypass detection.

On the other hand, the benign report showed that no evasion techniques were detected. This is consistent with the model's ability to correctly identify the lack of evasion techniques, indicating that it is able to distinguish benign from malicious behavior. In addition, this further supports the model's accuracy in discrimination between malware efforts to evade detection and legitimate software.

The analysis reveals that the trained LSTM model achieves favorable results when tested in real-world situations by correctly identifying evasive techniques used by malicious samples while also accurately classifying benign behavior as "non-evasive". This performance corroborates the earlier reported metrics and demonstrates the sandbox evasion capability of the model.

3.1.4 Results – Reinforcement Learning-Based Dynamic Sandbox Modification for Malware Behavior Analysis

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.

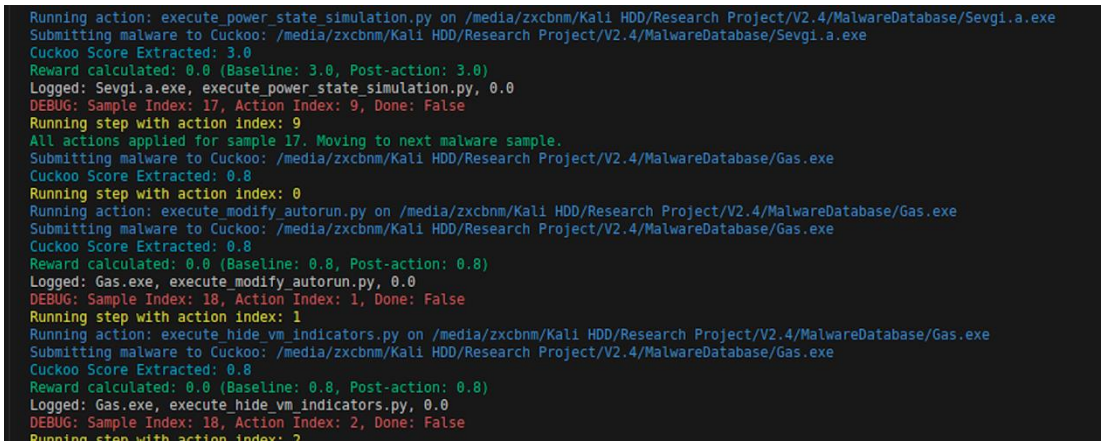


Figure 121: Agent Training Process

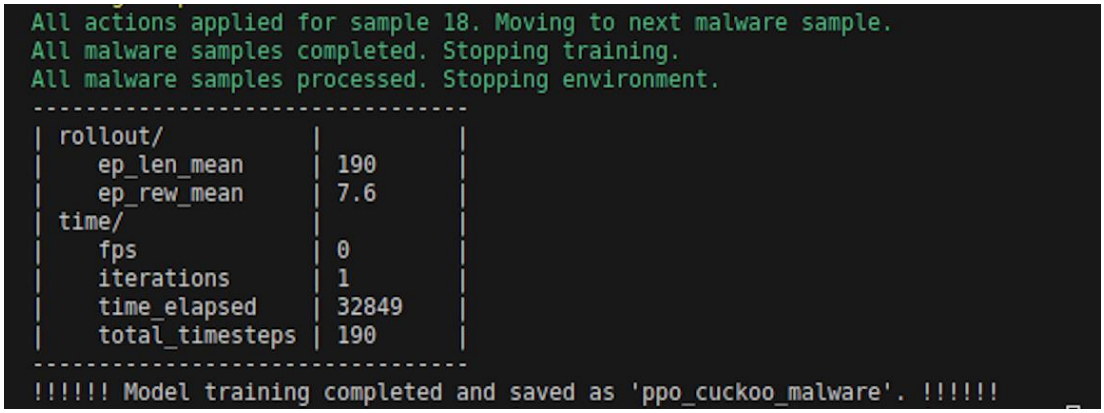


Figure 122: Final Output

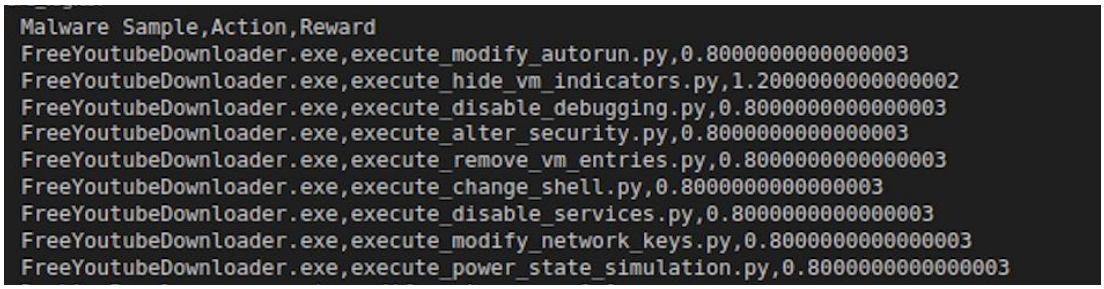


Figure 123: Reward Allocation

3.2 Research Findings

This research presents a multifaceted approach to detecting sandbox-evasive malware through the integration of reinforcement learning, synthetic user behaviour simulation, dynamic sandbox manipulation, and evasion technique classification.

3.2.1 Reinforcement Learning for Malware Evasion Strategy Development

The reinforcement learning (RL) agent exhibited increasing performance in generating malware variants that defeated the Cuckoo Sandbox. Throughout training consisting of 70 episodes, the detection scores were progressively lowered from just under 5.8 to below 1.8 (in some samples there were scores of 0.0 which indicate the malware were completely evaded). The increasing efficacy and the increment of reward values for each episode confirms that the RL agent was trained to learn high-performing evasive behaviours.

Overall, the agent's final policy showed low volatility, clear convergence, and generalised well to unseen state-action sequences. This suggests that the research demonstrated in this paper is a viable method for exploring evasive strategies and combinations of evasive strategies, and for stress-testing sandbox environments with a RL agent.

3.2.2 User Behaviour Simulation with GAN-Based Profiles

The WGAN-GP model, trained using real keystroke and mouse movement data, successfully generated statistically realistic user profiles. The model achieved a KS-test p-value of 0.0082 at its best checkpoint, indicating close distributional alignment between real and synthetic behavioural data. While some features showed variability, post-processing using distribution matching effectively reduced divergence.

Sandbox testing demonstrated that malware designed to avoid static sandboxes became active when exposed to these simulated user behaviours. Detection scores rose

significantly (up to 5.2) when behaviour scripts were injected, compared to scores near 0.0 in sessions without simulation. This result confirms that the simulation framework can trigger evasive malware and enhance visibility into their true behaviour.

3.2.3 Hybrid Detection Model for Evasion Technique Classification

When comparing the three models - VAE, tree-based classifiers, and LSTM - The LSTM-based model was also more suitable for identifying evasion techniques in sandbox logs. The LSTM-based model achieved an accuracy score at 0.95, a precision score at 0.94, recall score at 0.96 and an F1-score of 0.96. This indicates an ability to capture the sequential nature of API call data.

The assessments and behavioural analysis on Cuckoo Sandbox reports established further confidence in the model's precision, as it identified several evasion techniques in the malicious samples while non-evasive behaviour experienced marked outcomes, confirming successful benign behaviour classification. These results suggest the LSTM model has potential in real-world scenarios of malware detection.

3.2.4 Dynamic Sandbox Modification via PPO Agent

The final component evaluated how environment-level changes influenced malware execution. Through 190 timesteps, a PPO-based agent applied a series of counter-evasion actions. Actions like `execute_hide_vm_indicators.py` resulted in the highest behavioural changes, with reward values reaching 1.2, indicating their ability to provoke detectable malware behaviour.

The stable reward tracking and effective environment interaction suggest that targeted sandbox manipulation can expose latent behaviours in evasive malware. This supports the feasibility of dynamic sandbox adaptation as a complementary defence layer.

3.3 Discussions

The results from this study demonstrate that combining offensive and defensive techniques within a sandbox environment offers a promising approach to tackling sandbox-evasive malware. The reinforcement learning (RL) component showed that evasive malware can effectively adapt its behaviour when faced with static sandbox detection. By systematically learning to apply effective evasion strategies, the RL agent was able to produce malware samples with increasingly lower detection scores, in some cases achieving complete sandbox evasion. This illustrates both the capabilities of machine learning in generating evasive threats and the critical need for adaptive defence mechanisms in modern malware analysis systems.

To counteract these threats, the user behaviour simulation component proved to be a pivotal defensive layer. The WGAN-GP-generated user profiles demonstrated strong statistical similarity to real user interactions, with KS test results confirming meaningful distributional overlap. When these simulated behaviours were injected into the sandbox, evasive malware that previously remained dormant began executing its payloads. This result highlights the importance of human-like interaction in fooling advanced malware into revealing its true nature. It also underscores that conventional sandbox environments lacking such simulation are insufficient in detecting context-aware threats.

Further supporting the layered defence approach, the hybrid LSTM-based detection system effectively classified multiple sandbox evasion techniques from behavioural logs. Unlike earlier models, the LSTM architecture was able to capture temporal dependencies within API call sequences, achieving a balanced performance across accuracy, precision, and recall. Real-world behaviour log analysis confirmed its ability to distinguish between malicious and benign samples, suggesting its robustness in production settings. These findings suggest that integrating behaviour-based classification systems within sandbox workflows can improve automated malware triage and reduce analyst burden.

Finally, the dynamic sandbox reconfiguration module offered an innovative strategy for behavioural probing. By manipulating the sandbox environment through targeted actions, the system was able to provoke changes in malware behaviour that might otherwise remain undetected. This shows that both static realism and dynamic variability are essential in overcoming advanced evasion. However, challenges remain, such as ensuring system generalisability across malware families and environments, as well as maintaining low false positives in large-scale deployment. Future work could benefit from the integration of adaptive user simulation with real-time sandbox reconfiguration to maximise behavioural coverage.

CONCLUSION

The rapidly changing threat scene in cybersecurity calls for more than just passive detection mechanisms and reactive responses. Project Hyperadapt's idea was to combine defensive and offensive artificial intelligence models into a single sandbox framework, therefore addressing these flaws. This work effectively shows how to use reinforcement learning not only to replicate real-world evasive malware but also to dynamically strengthen conditions for malware investigation. The system continuously challenges itself using a co-evolutionary learning architecture, hence enhancing resilience and threat detection efficacy with every iteration.

The offensive reinforcement learning component of this project is fundamental and functions as a generative adversary inside the ecosystem. Trained with a DQN-based architecture, this agent was intended to independently find ideal combinations of malware evasion strategies aiming at dynamic analysis systems like Cuckoo Sandbox. Encoding malware evasion as a sequential decision-making challenge allowed the model to iteratively improve its approach by testing. It looked at timing-based, filesystem-based, and human-behavior-based evasions and finally found that sandbox environments like Cuckoo are especially susceptible to human behavior checks. By means of appropriate evasion sequences, the agent's capacity to routinely obtain detection scores as low as 0.0 revealed the feasibility of circumventing even extensive sandbox designs.

The experimental approach confirmed the theory that against Cuckoo's stationary emulation of user activity, behavior-based evasion methods are the most successful. Although filesystem and timing-based approaches sometimes showed some results, their irregularity underscored the requirement of deeper behavioral imitation in sandbox contexts. By means of 70 complete training sessions and more than 140 evasion operations performed, the agent created a strategy emphasizing approaches like recent document registry checks (H3), therefore showing Cuckoo's lack of simulated user interactions as a major vulnerability. These realizations provide useful

intelligence for bettering sandbox architecture, particularly in cases when defensive learning agents are used to resist adversarial tactics.

Defensively, the project combines hybrid detection logic, simulating human-like system interaction, and complementary artificial intelligence agents analyzing evasive behavior patterns. These defensive elements change in tandem as the offensive agent develops—learning from evasion events, changing threat models, and instantly reinforcing sandbox defenses. This continuous arms struggle between detection and evasion produces a strong framework able to adapt to zero-day and polymorphic malware threats without prior signatures. Such flexibility goes much beyond traditional rule-based sandboxes, which stay still following deployment.

One important contribution of the project comes from its end-to- end implementation schedule. From the development of base malware with modular evasion hooks to the evolution of automated testing, cross-compilation, API orchestration, and training cycles, the project provides a whole platform that can act as a basis for further studies. Using cloud environments like Google Colab and sandbox orchestration tools like Cuckoo, which permit distributed analysis and real-time interaction, the infrastructure also offers scalability and robustness. Moreover, built-in checkpointing and recovery systems guarantee fault tolerance and training continuity—qualities needed for long-term implementation in systems of production-level security.

From a more general standpoint, Project Hyperadapt helps the field of cybersecurity not only technically but also conceptually. It adopts a co-adaptive model, therefore transcending the conventional binary contrast of attack against defense. The attacking agent becomes a catalyst pushing defensive systems to get better, not only a testing instrument. Likewise, the protective elements grow valuable via their constant interaction with changing hazards. Suggesting a fresh path in the arms race between malware writers and security practitioners, this ecosystem-oriented viewpoint fits very well with the increasing focus on adversarial learning and cyber threat intelligence.

Finally, our work confirms the necessity of intelligent, flexible, and autonomous

systems in malware detection and protection. Exposing current flaws in sandbox technology and proving the limits of static analysis helps the offensive reinforcement learning model to be rather important. Integrated with defensive learning agents into a feedback loop, the system transforms into a living architecture—always learning, always changing, always protecting. The success of this method affects more than just malware detection and Cuckoo Sandbox. It suggests a time where intelligent cyber systems defend digital ecosystems with resilience and foresight by constant adversarial interaction.

REFERENCES

[1] S. Ž. Ilić, M. J. Gnjatović, B. M. Popović, and N. D. Maček, "A Pilot Comparative Analysis of the Cuckoo and Drakvuf Sandboxes: An End-User Perspective," Vojnotehnički Glasnik / Military Technical Courier, vol. 70, no. 2, pp. 372-392, 2022, doi: 10.5937/vojtehg70-36196.

[2] A. A. R. Melvin and G. J. W. Kathrine, "A Quest for Best: A Detailed Comparison Between Drakvuf VMI-Based and Cuckoo Sandbox-Based Techniques for Dynamic Malware Analysis," in Intelligence in Big Data Technologies - Beyond the Hype, P. J. Fernandes, S. Peter, and A. Alavi, Eds. Singapore: Springer, 2020, pp. 386-395, doi: 10.1007/978-981-15-5285-4_27.

[3] T. Quertier, B. Marais, S. Morucci, and B. Fournel, "MERLIN: Malware Evasion with Reinforcement Learning," arXiv preprint arXiv:2203.12980v4, Mar. 2022.

[4] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, and H. Yin, "MAB-Malware: A Reinforcement Learning Framework for Attacking Static Malware Classifiers," arXiv preprint arXiv:2003.03100v3, Apr. 2021.

[5] R. Labaca-Castro, S. Franz, and G. D. Rodosek, "AIMED-RL: Exploring Adversarial Malware Examples with Reinforcement Learning," in Proceedings of the 16th International Conference on Availability, Reliability and Security (ARES '21), Vienna, Austria, 2021, pp. 1-9, doi: 10.1145/3465481.3470076.

[6] X. Li and Q. Li, "An IRL-based malware adversarial generation method to evade anti-malware engines," Computers & Security, vol. 104, pp. 102118, 2021, doi: 10.1016/j.cose.2020.102118.

[7] T. Quertier, B. Marais, S. Morucci, and B. Fournel, "Malware Evasion with Reinforcement Learning," arXiv preprint arXiv:2203.12980v4, Mar. 2022.

[8] R. Labaca-Castro, S. Franz, and G. D. Rodosek, "AIMED-RL: Exploring Adversarial Malware Examples with Reinforcement Learning," in Proceedings of the 16th International Conference on Availability, Reliability and Security (ARES '21), Vienna, Austria, 2021, pp. 1-9, doi: 10.1145/3465481.3470076.

[9] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, and H. Yin, "MAB-Malware: A Reinforcement Learning Framework for Blackbox Generation of Adversarial Malware," in Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), Nagasaki, Japan, 2022, pp. 990-1000, doi: 10.1145/3488932.3497768.

[10] J. Ferdous, R. Islam, Arash Mahboubi, and Z. Islam, "A State-of-the-Art Review of Malware Attack Trends and Defense Mechanism.," IEEE Access, vol. 11, pp. 121118–121141, Jan. 2023, doi: https://doi.org/10.1109/access.2023.3328351.

[11] M. N. Alenezi, H. K. Alabdulrazzaq, A. A. Alshaher, and M. M. Alkharang, "Evolution of Malware Threats and Techniques: a Review," International Journal of Communication Networks and Information Security (IJCNIS), vol. 12, no. 3, 2020, doi: https://doi.org/10.17762/ijcnis.v12i3.4723.

[12] R. Holzer, P. Wüchner, and Hermann de Meer, "Modeling of Self-Organizing Systems: An Overview," Electronic Communication of The European Association of Software Science and Technology, vol. 27, Apr. 2010, doi: https://doi.org/10.14279/tuj.eceasst.27.385.

[13] M. K. Nishat, O. Gnawali, and A. Abdelhadi, "Adaptive Bitrate Video Streaming for Wireless Nodes: A Survey," *ResearchGate*, Jul. 2020. [Online]. Available: https://www.researchgate.net/publication/343415080_Adaptive_Bitrate_Video_Streaming_for_Wireless_nodes_A_Survey. [Accessed: Aug. 24, 2024].

[14] A. Mohanta and A. Saldanha, Malware analysis and detection engineering: A comprehensive approach to detect and analyze modern malware, 1st ed. Berlin, Germany: APress, 2020.

[15] M. Sikorski and A. Honig, Practical malware analysis: The hands-on guide to dissecting malicious software. No Starch Press, 2012.

[16] A. Chailytko and S. Skuratovich, “Defeating Sandbox Evasion How to increase the successful emulation rate in virtual environment,” 2016.

[17] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, “Detecting Environment-Sensitive Malware,” in Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 338–357.

[18] A. Mills and P. Legg, “Investigating anti-evasion malware triggers using automated sandbox reconfiguration techniques,” J. Cybersecur. Priv., vol. 1, no. 1, pp. 19–39, 2020.

[19] A. Lindorin, Impersonating a sandbox against evasive malware. 2022.

[20] C. Xie et al., “EnvFaker: A method to reinforce Linux sandbox based on tracer, filter and emulator against environmental-sensitive malware,” in 2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2021.

[21] P. Feng, J. Sun, S. Liu, and K. Sun, “UBER: Combating sandbox evasion via user behaviour emulators,” in Information and Communications Security, Cham: Springer International Publishing, 2020, pp. 34–50.

[22] S. Liu, P. Feng, S. Wang, K. Sun, and J. Cao, “Enhancing malware analysis sandboxes with emulated user behaviour,” Comput. Secur., vol. 115, no. 102613, p. 102613, 2022.

[23] L. Wang et al., “User behaviour simulation with large language model based agents,” arXiv [cs.IR], 2023.

[24] Y. Lu and J. Li, “Generative adversarial network for improving deep learning based malware classification,” in 2019 Winter Simulation Conference (WSC), 2019.

[25] A. Dunmore, J. Jang-Jaccard, F. Sabrina, and J. Kwak, “Generative Adversarial Networks for malware detection: A survey,” arXiv [cs.CR], 2023.

[26] T. Porutiu, How Sandbox Security Can Boost Your Detection and Malware Analysis Capabilities, ENDPOINT PROTECTION & MANAGEMENT, March 28, 2024.

[27] "Sandbox detection and evasion techniques. How malware has evolved over the last 10 years,” Ptsecurity.com. [Online]," [Online]. Available: Available: https://global.ptsecurity.com/analytics/antisandbox-techniques. [Accessed: 11-Apr-2025]

[28] "S. G. a. M. Picado, "Agent Tesla amps up information stealing attacks," 02 02 2021. [Online].," [Online]. Available: Available: 02Feb2021.https://news.sophos.com/enus/2021/02/02/agent-tesla-amps-up-information-stealing-attacks.

[29] Anishnama, "Understanding LSTM: Architecture, pros and cons, and implementation,” Medium, 28-Apr-2023. [Online]," [Online]. Available: Available: https://medium.com/@anishnama20/understanding-lstm-architecture-pros-and-cons-and-implementation-3e0cca194094. [Accessed: 11-Apr-2025]..

[30] A. S. Abhijit Mohanta, Malware Analysis and Detection Engineering A Comprehensive Approach to Detect and Analyze Modern Malware, 2022.

[31] Y. S. D. D. X. L. a. M. L. F. Xiao, Novel Malware Classification Method Based on Crucial Behavior, Mathematical Problems in Engineering, no. doi:10.1155/2020/6804290, 2020.

[32] S. Morgan, Cybercrime To Cost The World $10.5 Trillion Annually By 2025"," Cybercrime magazine website. Cybersecurity ventures, 2022.

[33] C. TRAP, "The Evolution of Malware: From Intricacies to Solutions," ANUARY 19, 2024. [Online]," [Online]. Available: https://www.canarytrap.com/blog/malware-evolution/#:~:text=A%20Brief%20History,prank%20than%20a%20security%20threat...

[34] B. Alsulami, A. Srinivasan, H. Dong and S. Mancoridis, Lightweight behavioral malware detection for windows platforms, 2017 12th International Conference on Malicious and Unwanted Software (MALWARE), 2017.

[35] S. O. C. -.. SecureOps, How Malware Uses Encryption To Evade Cyber Defense, Problems And Solutions For The SOC, AUGUST 13, 2019.

[36] B. Alsulami, A. Srinivasan, H. Dong and S. Mancoridis, Lightweight behavioral malware detection for windows platforms, 2017 12th International Conference on Malicious and Unwanted Software (MALWARE), 2017.

[37] I. Alsmadi, B. Al-Ahmad and M. Alsmadi, Malware analysis and multi-label category detection issues: Ensemble-based approaches, 2022 International Conference on Intelligent Data Science Technologies and Applications (IDSTA), 2022.

[38] C. Wagner, G. Wagener, R. State and T. Engel, Malware analysis with graph kernels and support vector machines, 2009 4th International Conference on Malicious and Unwanted Software (MALWARE), 2009.

[39] o. A. Marpaung, M. Sain and H.-J. Lee, Survey on malware evasion techniques: State of the art and challenges, 2012 14th International Conference on Advanced Communication Technology (ICACT), 2012.

[40] S. Jain, T. Choudhury, V. Kumar and P. Kumar, - Detecting Malware & Analysing Using Sandbox Evasion, 2018 International Conference on Communication, Computing and Internet of Things (IC3IoT), 2018.

[41] T.-H. Cheng, Y.-D. Lin, Y.-C. Lai and P.-C. Lin, Evasion Techniques: Sneaking through Your Intrusion Detection/Prevention Systems, IEEE Communications Surveys & Tutorials , 2019.

[42] A. K. Sinha and S. Sai, Integrated Malware Analysis Sandbox for Static and Dynamic Analysis, 2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT), 2023.

[43] S. S. Darshan, M. A. Kumara and C. Jaidhar, Windows malware detection based on cuckoo sandbox generated report using machine learning algorithm, 2016 11th International Conference on Industrial and Information Systems (ICIIS), 2016.

[44] M. P. J, A. C. D, A. AS, S. P. B. R and R. S.M, Malware Detection using Machine Learning, 2024 Second International Conference on Advances in Information Technology (ICAIT), 2024.

[45] B. Bokolo, R. Jinad and Q. Liu, A Comparison Study to Detect Malware using Deep Learning and Machine learning Techniques, 023 IEEE 6th International Conference on Big Data and Artificial Intelligence (BDAI) , 2023.

[46] 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.

[47] 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.

[48] 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

[49] 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.

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

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

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