A Novel Reinforcement Learning Model for Post-Incident Malware Investigations

Dipo Dunsin, Mohamed Chahine Ghanem *, Karim Ouazzane, Vassil Vassilev,

***Abstract*—This Research proposes a Novel Reinforcement Learning (RL) model to optimise malware forensics investigation during cyber incident response. It aims to improve forensic investigation efficiency by reducing false negatives and adapting current practices to evolving malware signatures. The proposed RL System leverages techniques such as Q-learning and the Markov Decision Process (MDP) to train the system to identify malware patterns in live memory dumps, thereby automating forensic tasks. The RL model is based on a detailed malware workflow diagram that guides the analysis of malware artefacts using static and behavioural techniques as well as machine learning algorithms. Furthermore, it seeks to address challenges in the UK justice system by ensuring the accuracy of forensic evidence. We conduct testing and evaluation in controlled environments, using datasets created with Windows operating systems to simulate malware infections. The experimental results demonstrate that RL improves malware detection rates compared to conventional methods, with the RL model's performance varying depending on the complexity and learning rate of the environment. The study concludes that while RL offers promising potential for automating malware forensics, its efficacy across diverse malware types requires ongoing refinement of reward systems and feature extraction methods.**

***Index Terms*—Cyber Incident, Digital Forensics, Artificial Intelligence, Machine Learning, Reinforcement Learning, Malware, Incident Response.**

I. Introduction

In post-incident malware forensics investigations, the detection and classification of malware are critical processes for reconstructing evidence files. This is particularly important because malware, being a malicious program, can lead to unauthorised access to confidential information, jeopardising the security and integrity of data or information systems, thereby posing a significant threat to the involved systems and institutions. He and Sayadi [1] highlight that malware attacks have become a pervasive threat, affecting homes, education, businesses, government, and healthcare by "finding vulnerabilities in networks and applications to launch attacks." In the healthcare field, particularly within the Internet of Medical Things (IoMT), such malicious attacks are especially dangerous [2]. A minor misclassification or failure to detect malware can seriously compromise patient medical records, potentially leading to incorrect diagnoses or treatments and, in extreme cases, resulting in paralysis or death. Machine Learning (ML) has gained widespread adoption to detect various types of malware. However, increasingly complex malware can circumvent ML techniques and models. The paper by Wu et al., [4] discusses the use of reinforcement learning (RL) as a more advanced model that improves malware detection accuracy, surpassing traditional machine learning methods. According to Liu et al., [5], machine learning in malware detection involves extracting features from data to classify malware and learn from past data to identify new threats. As a result of leveraging algorithms and data analysis, machines can improve their accuracy in identifying malware. However, newer and more complex malware can deceive ML models by masquerading as benign software, evading detection [6] [7]. Ebrahimi et al., [8] suggest that reinforcement learning can help defenders detect sophisticated adversarial threats. Adversarial malware relies on perturbation methods to evade ML-based detectors [9] [10]. A significant challenge is the need for constant updates with new malware behaviours.

Reinforcement learning, on the other hand, enables models to generate adversarial malware that can bypass detection by portable executable (PE) malware classifiers. Quertier et al., [10], note that the RL System "Gym-Malware" achieves an evasion rate of up to 16%, while the RL System "MAB-malware" can achieve an evasion rate of over 75% in a black-box setting. Furthermore, according to Quertier et al., [10], adversarial knowledge defines attacks as either "white box," where the adversary has complete access to the model, or "black box," where the adversary has no knowledge of the model and can only obtain classification results through a limited number of attempts [13]. These adversarial samples, which are falsely classified as benign, can improve the detection accuracy of malware detectors by 16% to 94% [4]. Reinforcement learning's ability to automate malware evasion shows how detailed sequences of adversarial actions can train antivirus and malware detection systems, enhancing their effectiveness in combating malware [10].

A. Research Aim and Question

This research aims to improve malware forensics investigations by utilising reinforcement learning (RL) techniques. The primary focus is on identifying, analysing, and enhancing models for post-incident investigations. As a result of this, the goal is to expedite forensic processes and mitigate the miscarriage of justice within the UK legal system [3]. Additionally, this research seeks to improve heuristic- and signature-based analysis methods through the application of

* Mohamed C. Ghanem is the corresponding author. email: m.ghanem@londonmet.ac.uk

Mr. D. Dunsin, Dr. M.C. Ghanem, Prof. K. Ouazzane and Prof. V. Vassilev are with the Cyber Security Research Centre, London Metropolitan University, London, UK

RL, thereby enhancing overall cybersecurity measures after a security breach. Building on these objectives, the research also aims to address a central question: How effective are reinforcement learning models in distinguishing between benign and malicious software, and what are the areas for potential improvement? The study investigates RL's role in enhancing malware analysis within post-incident forensics, particularly in identifying patterns that traditional tools may struggle to detect. Furthermore, it examines RL's adaptability to evolving malware signatures and explores the feasibility of combining RL techniques with heuristic methods for more reliable and comprehensive malware analysis.

II. Related work

A. Reinforcement Learning Improves ML Malware Detection

Wu et al. [4] explore reinforcement learning (RL) to enhance malware evasion against machine learning detection models. They propose the "gym-plus" model, an extension of "gym-malware," which generates evasive malware samples. Using the EMBER dataset, they retrain detection models, improving detection rates for unknown malware. However, the study lacks a clear theoretical explanation of RL and omits a discussion on limitations like agent selection and action space. Moreover, there is no comparative analysis with other detection methods. Despite these gaps, the paper demonstrates practical effectiveness, offering a detailed evaluation of the RL model. Future work should address these limitations to further improve malware detection.

B. Adversarial RL for Malware Detection

Ebrahimi et al., [8] research improves cybersecurity by addressing adversarial attacks on ML-based malware detection. Specifically, their innovative use of adversarial reinforcement learning (RL) enhances malware detector robustness in dynamic environments. The method improves dynamic interactions between the detector and the adversary by including adversarial agents that create false samples. However, reliance on MalVAC and lack of comparative assessment limit the research's generalizability. Furthermore, more theoretical exploration of convergence properties could improve the System. Despite these flaws, their focus on future work like optimisation solvers and reducing false positives shows forward-thinking.

C. Reinforcement Learning: Uncovering Control

Wang et al., [11] propose an automated approach for identifying command and control (C2) attack paths in large networks using reinforcement learning (RL). They address manual C2 detection methods' limitations by emphasizing efficiency and automation. By integrating cyberdefense terrain into the RL model, they enhance practical relevance. While their detailed attack simulation, which covers infection to exfiltration phases, demonstrates rigor, reliance on a simulated environment limits real-world applicability. Furthermore, the assumption of full knowledge of host characteristics may be unrealistic, and the RL model's complexity poses practical challenges. Despite these weaknesses, the paper contributes valuable insights into RL-based intrusion detection, laying the groundwork for future cybersecurity research and applications in network defence.

D. Reinforcement Learning for Grid Security

Yu [12] introduces a hierarchical deep reinforcement learning-based (HDRAD) scheme for detecting Advanced Persistent Threats (APT) in data management systems. The thorough simulations demonstrating HDRAD's superior performance over RS and HP schemes in detection delay and data protection are a key strength. However, the study's complexity and reliance on technical illustrations make it difficult for non-experts. While the research outlines its objectives, it lacks simplified explanations of how the HDRAD scheme functions. Additionally, brief introduction, discussion, and conclusion sections limit comprehensive understanding. Despite these limitations, the paper significantly contributes by presenting a novel and efficient APT detection method for improving data security in smart grids.

E. Deep Learning Techniques for Malware Obfuscation

Gao and Fang [14] present a novel malware evasion approach using deep reinforcement learning. They generate adversarial examples by extracting bytes from benign files and injecting them into malware, achieving an 85% evasion rate against EMBER, a state-of-the-art malware classifier. The research's strengths include the innovative combination of reinforcement learning and malware evasion, along with a clear presentation. However, weaknesses arise from a lack of discussion on ethical implications and potential limitations, such as effectiveness across diverse malware types. Despite the robust experimental design, real-world validation requires further research. Nevertheless, the study significantly contributes to cybersecurity by proposing a promising yet ethically complex method for enhancing malware evasion techniques.

F. A3C Algorithm for Malware Identification

Xue et al., [16] describe a novel approach to improving malicious code detection using reinforcement learning and the Asynchronous Advantage Actor-Critic (A3C) algorithm. They highlight the limitations of traditional detection methods and position their work within the broader context of machine learning advancements. The research's strengths include its detailed methodology and contribution to generating anti-detection adversarial samples. However, the study lacks in-depth discussion of broader implications, such as ethical concerns and scaling challenges. Exploring alternative adversarial techniques and potential biases could improve robustness. Despite these limitations, the article significantly contributes to cybersecurity through its novel application of reinforcement learning techniques.

III. Research Methodology

A. Experimental Setup and Dataset Generation

The London Metropolitan University Digital Forensics Laboratory created a comprehensive malware dataset to implement

and validate the proposed reinforcement learning malware investigation system [17]. We set up thirteen virtual machines in a secure network to prevent unintended malware spread. We uploaded different ISO files of the Windows operating system to ensure a diverse test environment. We introduced malware to each virtual machine, took snapshots of infected and uninfected states, and produced 26 RAM files. To uncover malware behaviours, we analysed these files using the Volatility System. Finally, we created a detailed workflow diagram to facilitate the training and validation of the model.

B. Malware Workflow Diagram Creation

The research methodology covers the experimental setup, dataset generation, and development of a malware analysis workflow. This workflow is central to the reinforcement learning System, integrating techniques such as data collection, examination, and analysis [18]. We analysed live memory dumps from *13* versions of Windows, both infected and uninfected, using the Volatility System to detect anomalies and malware artefacts. The analysis phase employs static, signature-based, behavioural techniques, and machine learning algorithms. The workflow diagram maps typical malware behaviours and improves post-incident forensic investigations, supporting the reinforcement learning model's training and validation.

C. Q-Learning Terminologies

We implement the proposed Reinforcement Learning Post-Incident Malware Investigation Model in the following sections. Key terminologies are briefly defined. The ***Environment*** is the world where the agent operates. The ***Agent*** learns by interacting with the environment. ***States*** (s) represent the agent's position, while an ***Action (a)*** is any move the agent can take, leading to either a reward or penalty. ***Episodes*** signify the end of a stage, either through success or failure. For each state-action pair, the agent manages ***Q-values*** in a ***Q-Table***. ***Temporal Differences (TD)*** compare current and previous state-actions. The ***learning rate*** controls how new information replaces old. A ***policy*** maps states to actions, while the ***Discount Factor*** weighs future rewards. The ***Bellman Equation*** relates Q-values across state-action pairs, and the ***Epsilon-Greedy*** strategy balances *exploration* and *exploitation*.

D. The Unified Markov Decision Process

The Unified Markov Decision Process (MDP), illustrated in ***Figure 1***, brings together various MDP components into a single framework, offering a complete overview. This integration enables the agent to efficiently explore the environment and make well-informed choices in the context of malware analysis.

E. Proposed RL Post-Incident Malware Investigation System

The main components of the Reinforcement Learning Post-Incident Malware Investigation System consist of six key elements, as shown in ***Figure 2***. These elements include data acquisition, workflow diagram mapping, implementation of the MDP model, environmental dependencies, the MDP solver, and continuous learning with adaptation. Initially, data acquisition involves capturing live memory dumps from Windows operating systems. Subsequently, data analysis focuses on examining the acquired data to detect anomalies, indicators of compromise, and potential malware artifacts. The workflow diagram provides a detailed approach for identifying malware infections through static analysis, signature-based techniques, behavioural analytics, and machine learning algorithms. The AWK module carries out feature extraction from identified processes. In addition, listing DLLs is important for monitoring the DLLs loaded by each process. Monitoring open handles is also critical for tracking the open handles associated with each process. Network data collection ensures the capture of all relevant network-related information. We conduct registry hive analysis to identify registry hives and enumerate their keys. We duplicate processes into executable files and compare them with known malware databases to determine their malicious or benign nature. We also duplicate addressable memory to conduct searches using specific keywords. The state spaces are constructed to match the malware workflow diagram, encompassing ***67*** unique states. Actions are defined based on this workflow, ranging from ***three*** to ***ten***, which exposes the agent to ***109*** different actions within a defined environment. As part of our work on the MDP solver, we create unified Markov Decision Process (MDP) models from the proposed RL Post-Incident Malware Investigation Model. Lastly, the environment setup dependencies are categorised into three sections: establishing dependencies and gym environments, importing the necessary libraries, and implementing training data for continuous learning and adaptation.

F. Algorithm 1 - Implementation of the Q-Learning Algorithm

Algorithm 1 implements Q-learning to train an agent for optimal decision-making. It initialises a zero-valued Q-table representing the agent's understanding of the environment, where rows are states and columns are actions. Key parameters like learning rate, discount factor, and exploration probability (initially set to 0.9) are established, along with decay schedules and data storage structures (*storage*, *storage new*, *reward list*). The main loop runs over a specified number of episodes, resetting the environment and variables at each episode's start. Within each episode, a ***greedy policy*** selects actions: if the probability is high, it explores by choosing a random action (***exploration***); otherwise, it exploits by selecting the action with the highest Q-value for the current state (***exploitation***). The environment executes the action, providing the next state, reward, and a done flag indicating the episode's end. Q-values are updated using the ***Bellman equation***, taking into account the immediate reward and maximum future Q-value, and stored in the Q-table. The state updates to the next state, and data such as Q-values, episode number, and action are appended to the storage list. The exploration probability decays according to a predefined schedule. We optionally check convergence by comparing the absolute difference between new and current Q-values; if it falls below a threshold, we can terminate the loop early. After each episode, we append

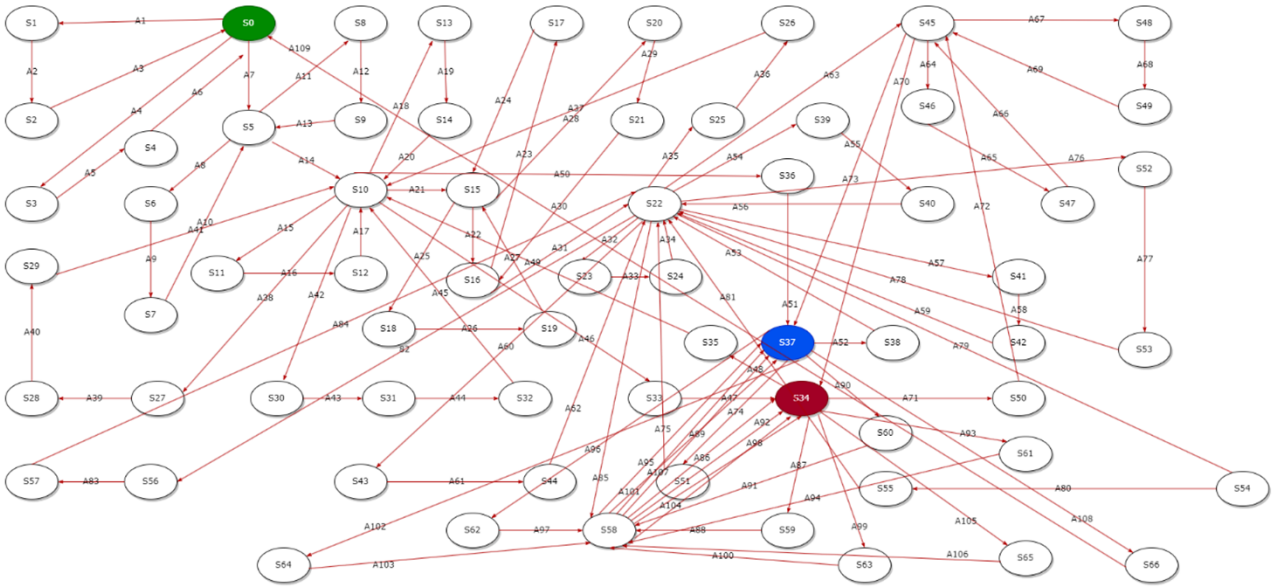

Fig. 1: *Overall Markov Decision Process (MDP) Model*

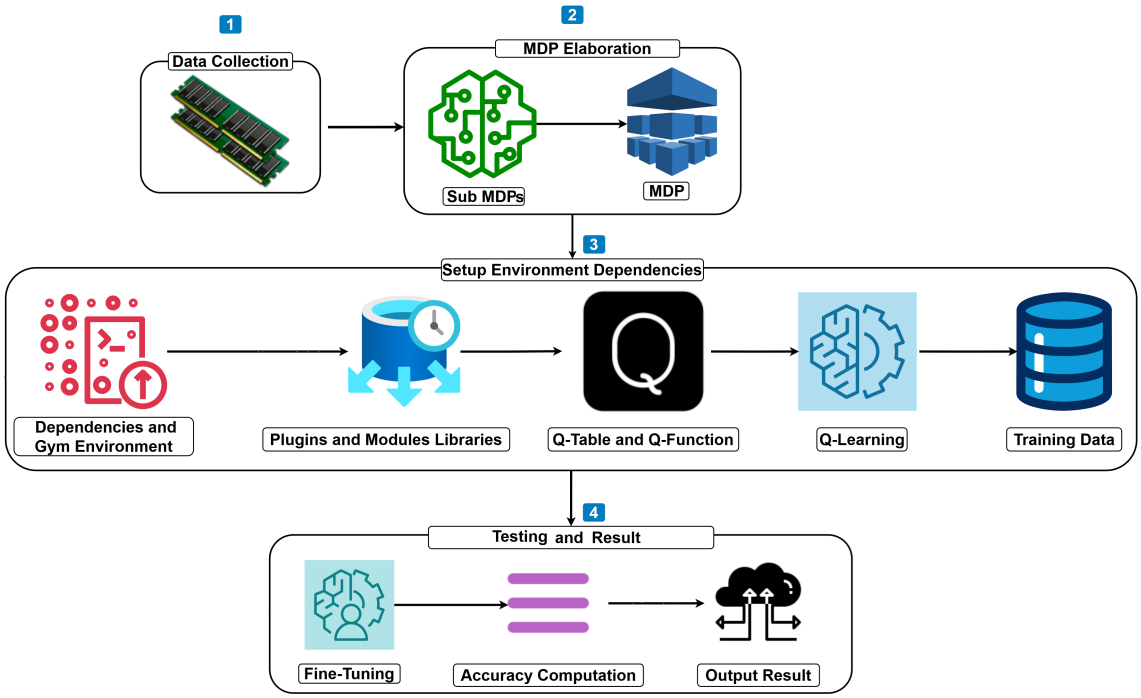

Fig. 2: *The Proposed RL Post-Incident Malware Investigation System*

episodic rewards and other data to the reward list. Finally, we return the Q-table and storage data, which summarise the learnt policy and training data. The process begins with the initialisation phase, where three custom environments (*env_new1*, *env_new2*, and *env_new3*) are defined using a specified *MDP function*. The algorithm iterates over a list of names (*name_list*), and for each name, it assigns the appropriate environment by configuring the Markov Decision Process (MDP) with specific transition probabilities and rewards.

G. Algorithm -2 Iterating Learning Rates Variation over MDP Environments

Algorithm 2 is a method that *trains* and *saves* models by utilising various *learning rates (LRs)* across different environments. The algorithm starts with an initialisation phase, where it defines multiple environments (envs) and creates an empty dictionary called '*final dict*' to hold the results. Training parameters are then set, which include a list of learning rates ranging from ***0.001 to 0.9***, along with storage for the corresponding Q-tables, intermediate data, rewards, and other storage structures. For each learning rate (***lr***) in the list of

Algorithm 1 *Q-Learning Implementation*

Initialization:
- Initialize Q-table with zeros. (defines the state of the agent)
- Set parameters: learning rate (α), discount factor (γ), exploration probability ($\epsilon = 0.9$), and decay schedule.
- Initialize storage structures: storage, storage_new, reward_list.

for episode = 0, 1, . . . , episodes do

Reset environment and variables:
- Reset the environment to obtain the initial state.
- Initialize episodic reward and step counter.
- Store current epsilon value.

while not done, do

Select action using ϵ-greedy policy:
- if $\epsilon <$ rand() then
- action $\sim$ Uniform(noA)
- else
- action = $\max_a Q(\text{state}, a)$

Execute action:
- Act in the environment and observe the next state, reward, and done flag.
- Update episodic reward.
- Increment steps counter.

Update Q-value using Bellman equation

Compute the maximum future Q-value for the next state.

Calculate the new Q-value

Update the Q-table with the new Q-value.

Store Q-value updates if specific conditions (e.g., state, action) are met.

Update state: Set the current state to the next state.

Append the (current_q, new_q, episode, action) to the storage list.

Decay ϵ: Reduce epsilon based on the decay schedule.

Check convergence: if $|\text{new\_q} - \text{current\_q}| <$ threshold and new_q $\neq$ current_q then

Break the loop

Append (episodic_reward, episode, steps) to reward_list.

Update ϵ:
- $\epsilon \leftarrow \epsilon - (\epsilon\text{\_decay\_value} \times 0.5)$

Return results:
- **Return Q-table, storage, reward_list, and storage_new.**

end for

Algorithm 2 *Iterating Learning Rates Variation over MDP Environments*

Initialization: Defining different envs and empty final dict

for name **in** name_list **do**

1) Assigning the right env (MDP): Using diff transition probs and rewards to create the MDP
2) Defining and resetting the training params:
   - Learning rates list
   - outputs, store and rewards dictionaries
3) **for** lr **in** lrs **do**
   a) Performing the new_q_learning algorithm
   b) Storing everything by appending in the final_dict
4) **end for**

end for

```python
class BlankEnvironment(gym.Env):
    def __init__(self):
        # Define action space and observation space
        self.action_space = gym.spaces.Discrete(10)
        self.observation_space = gym.spaces.Discrete(67)

        #self.action_space = np

        self.state = 0

        self.P = dict()
```

Fig. 3: *Initialise and Implement the BlankEnvironment Class*

learning rates (***lrs***), the algorithm applies the '***new q learning***' algorithm. Afterward, the results are appended to the '***final dict***', ensuring an organised process for model training and results recording across various environments.

IV. MDP Models Integration and Implementation

A. An Overview of Our Three Proposed MDP Environments

The **`BlankEnvironment`** represents the proposed Markov Decision Process using the Malware Workflow Diagram, encompassing *states, actions, rewards, transition probabilities*, and the completion status of an *episode*. It consists of a discrete action space with ***10*** possible actions and an observation space comprising ***67*** observations, applying a default step penalty of ***-0.04*** and awarding a reward of ***2*** for detecting malware. The **`BlankEnvironment_with_Rewards`** provides a reward of ***2*** for correctly identifying malware in all terminal states and ***4*** for correct identification at earlier stages, promoting accurate classifications. On the other hand, the **`BlankEnvironment_with_Time`** enforces a stricter penalty of ***-0.01*** per step to encourage timely malware detection by deterring the agent from making excessive moves. In both environments, rewards are treated as hyperparameters, fine-tuned to achieve optimal agent performance.

B. Implementation of the BlankEnvironment

In ***Figure 3***, we introduce a new class called '**`BlankEnvironment`**', which extends from '***gym.Env***', signifying its function as a gym-compatible environment. Within the `BlankEnvironment` class, the constructor sets up the instance of the class. A subsequent variable specifies the environment's action and observation spaces. The action space is discrete, containing ***10*** possible actions, while the observation space is also discrete, with ***67*** possible observations. The variable '***self.state = 0***' initialises the environment's starting state as '0'. We also initialise '***self.P = dict()***' as an empty dictionary to store the transition probabilities in the later code sections.

C. Implementation of the BlankEnvironment with Rewards

The ***BlankEnvironment with Rewards*** represents a distinct implementation in comparison to the ***BlankEnvironment***. In ***BlankEnvironment with Rewards***, actions that lead to terminal states receive a reward of *2*, unlike the *-0.04* reward given in the ***BlankEnvironment***. The reward function in ***BlankEnvironment_with_Rewards*** is adjusted at the end of an episode, as indicated by the done flag. The done flag assigns the fourth element's value to the variable named done, which reflects whether the episode has concluded. When the done flag detects that the done variable is set to True, the reward variable is assigned a positive value of *4*. This modification accounts for the impact of changing the reward at the conclusion of the episode.

D. Implementation of the BlankEnvironment with Time

In ***BlankEnvironment_with_Time***, the agent encounters a harsher negative reward of ***-0.1*** per step, in contrast to the usual penalty of ***-0.04*** present in the other two environments. The design of this approach encourages the agent to swiftly identify malicious files by taking the shortest possible route, thereby avoiding any unnecessary actions. Moreover, if the agent prolongs the episode by taking extra steps, it incurs substantial penalties. Crucially, we treat this incentive as a hyperparameter, enabling continuous adjustments. The expression '*done = k[3]*' assigns the fourth element from the tuple '*k*' to the variable 'done', signifying whether the episode has concluded. If 'done' is 'True', the agent receives a reward of ***+4***; otherwise, it is penalised with ***-0.1***. Subsequently, the tuple 'new k' is generated by retaining the original values from '*k*' but updating the reward. This modified tuple is then returned for further interactions with the environment.

E. Iterating MDP Environments over Learning Rates

We implemented a Python code and iterated the three MDP environments over a range of learning rates (*0.001–0.9*). The `name_list = ['env_new1', 'env_new2', 'env_new3']` defines a list containing the names of the environments. We initialise an empty dictionary to store the final results and iterate over each environment using a `for` loop. We use the *Q-learning function* to convert the current learning rate to a float and store the results in dictionaries. We convert the output into a list and save it in the output dictionary. Finally, we group the collected data into a tuple and store it in the final dictionary, consolidating all results for further insight.

V. Testing and Evaluation

A. Comparing the Speed of Convergence

We implemented a Python code to visualise the speed of convergence across the three MDP environments, (`env_new1`, `env_new2`, and `env_new3`) representing `BlankEnvironment`, `BlankEnvironment_with_Rewards()`, and `BlankEnvironment_with_Time()`, respectively. Each dictionary maps learning rates to the number of episodes required for convergence. The code line `x = [float(key)` `for key in env_new1.keys()]` creates a list of floating-point learning rates from `env_new1`. The command `plt.figure(figsize=(10, 6))` initialises a 10x6-inch plot, where we create scatter plots for each environment, using different colours (blue, red, and green) for distinction and adding lines to illustrate convergence trends. ***Figure 4*** shows that `BlankEnvironment_with_Rewards` (`env_new2`) has the smoothest and fastest convergence. In contrast, `BlankEnvironment_with_Time` (`env_new3`) converges slowly due to a higher negative reward function, necessitating larger learning rates and more computational time. `BlankEnvironment` (`env_new1`) also performs well, but it converges slower due to learning rate fluctuations. As a result, `BlankEnvironment_with_Rewards` is the best MDP environment, with a 0.4 learning rate.

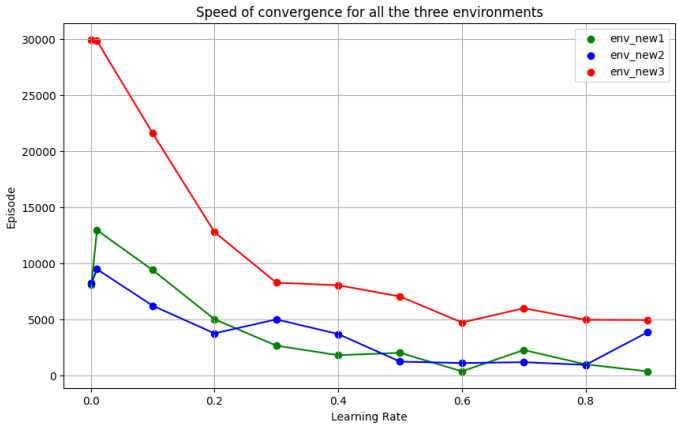

Fig. 4: The speed of convergence across the three MDP environments

B. Command Definitions for State-Based Actions

We imported the `Subprocess` module to allow the Python script to spawn new processes and manage their input/output/error pipes and return codes. We initialise and populate an empty dictionary, `my_dict`, with **key-value** pairs, each representing a ***state*** and each value a list of ***commands*** for that state. For example, *state 0* includes a command to clone a GitHub repository, while *state 10* has commands for Windows system information and the registry. *States 15 to 45* have various commands, some including special characters and options like `-pid` and `-o`.

VI. Results and Discussion

A. The Agent Decision-Making Processes

We implemented a Python script that initialises two lists, `ideal_list` and `pred_list`, containing integer values representing ***actions*** for specific ***states*** within our reinforcement learning MDP environment. The `ideal_list` assigns optimal actions for states ***0 to 66***, while the `pred_list` contains predicted actions for the same states. For example, *state 0* has an ideal *action of 0* and a predicted *action of 2*. Each index in both lists corresponds to a specific state,

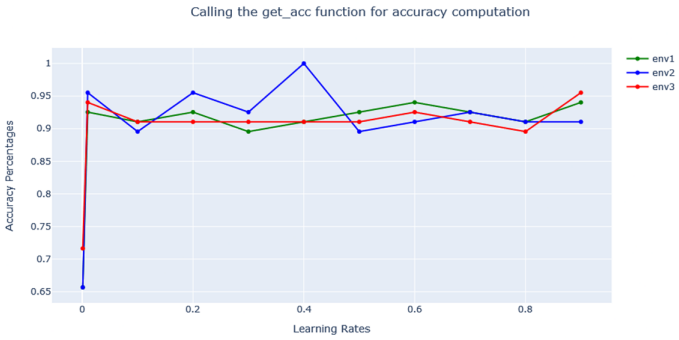

Fig. 5: *Calling the* `get_acc` *function for accuracy computation*

facilitating the comparison of predicted actions against ideal outcomes to measure model performance across the three environments using varying learning rates.

B. Python Function to Evaluate Predictive Model Accuracy

To compare the accuracy of predicted actions against ideal actions, we implemented a Python function named `get_acc`. Initially, the function sets two variables, `true` and `false`, to zero to count correct and incorrect predictions, respectively. It iterates through the `ideal_list` and `pred_list` simultaneously using the `zip()` method, comparing each element; if they match, it increments `true`; otherwise, it increments `false`. After the iteration, the function computes the accuracy by dividing `true` by the total number of comparisons (`true + false`), formats the result to five decimal places, and prints the ***accuracy***. This function is useful for evaluating prediction accuracy in reinforcement learning settings, and upon execution, it shows an accuracy of 94%.

C. `get_acc` function for accuracy computation

The implemented Python function named `get_acc` processes multiple environments (`env1`, `env2`, and `env3`) represented by dictionaries (`q1_dict`, `q2_dict`, and `q3_dict`). We defines x and y coordinates for three sets of data representing different environments: `env1`, `env2`, and `env3`. Each environment's data is stored in respective lists, such as `env1_x` and `env1_y`, `env2_x` and `env2_y`, and `env3_x` and `env3_y`. The code then creates three scatter plot traces using `go.Scatter`, specifying the data points, mode (lines and markers), names, and marker colours for each environment. A layout is defined for the plot, including a title, x-axis and y-axis labels, and hover mode configuration. A figure object is created by combining the traces and the layout, and the plot is displayed using `fig.show()`. This code effectively visualises the accuracy computation for different learning rates across *three MDP environments*, as shown in ***Figure 6***. Consequently, it demonstrates that ***env2***, with a ***learning rate of 0.4***, is the best-performing environment.

D. Plotting the proposed Model command execution timings

As a result of keeping track of state changes using our proposed reinforcement learning post-incident malware investigation model, we obtain a trajectory based on actions and landing states, which control a series of state changes in the environment. We utilised the Google Collaborative Environment's execution timings to plot the proposed model's command execution timings. To store keys and values related to states and action trajectories, we created a new `command_timings_dict`. We then defined new Python code to create a multi-plot figure using Plotly to analyse the execution time of different malware commands (WannaCry, Cerber, and Cridex). This code initialises a figure with three vertical subplots, each with a title and increased vertical spacing. We add line plots for WannaCry, Cerber, and Cridex to the first, second, and third subplots, respectively, ensuring each has distinct colours and markers. We update the figure's layout to set its dimensions and centre the title. We customize the X-axis labels for each subplot and label the y-axes with 'Time (seconds)'. The resulting graph is displayed using `fig.show()`, as illustrated in ***Figure 7*** below.

Fig. 6: *Malware Analysis Execution Time for Different Commands Executed Using the Agent (RL Model)*

E. Research Findings and Recommendations

This research examines post-incident malware forensics using reinforcement learning (RL), emphasising RL's growing importance in adapting to evolving malware threats. By automating forensic tasks, particularly malware artefact identification, through a structured RL workflow based on Q-learning, the

model effectively identifies and classifies malware. However, its performance depends on the diversity of malware samples, necessitating broader datasets to improve accuracy. Optimising RL models for computational efficiency is also critical, especially in resource-limited environments. Integration with existing security infrastructure is recommended, combining RL's adaptive learning with traditional forensic processes for a more robust defense. Furthermore, the ethical implications of AI deployment in cybersecurity must be considered to protect against adversarial attacks. Finally, this model could mitigate miscarriages of justice in the UK's legal system [19] by introducing a data-driven and unbiased approach to digital forensic analysis, enhancing judicial outcomes by improving accuracy and reducing errors.

VII. Conclusion

This paper presents a novel reinforcement learning (RL) model and System for post-incident malware forensics investigations, designed to surpass the capabilities of human forensic experts. The model accelerates malware analysis and identifies both known and unknown threats by integrating various techniques, such as live memory dumps from Windows systems. A unified Markov Decision Process (MDP) System was developed, featuring three environments: *BlankEnvironment*, *BlankEnvironment_with_Rewards*, and *BlankEnvironment_with_Time*, each with distinct reward mechanisms to optimise malware analysis. The RL agent, utilising Q-learning and epsilon-greedy exploration, iteratively refines its policy and decision-making process, improving malware identification accuracy. Experimental tests, including simulations with malware like WannaCry and Cerber, demonstrated that performance depends on learning rates and environment complexity. The study focuses on hyperparameter tuning and continuous learning to improve the performance of RL models. It shows that reward systems, feature extraction, and hybrid analysis could all use more optimisation. In future work, we intend to integrate a rule-based expert system (RBES) to capture expertise generated by our system, generalise it and use it directly in future investigations where the case investigated involves similar machine architecture and configurations, this will certainly improve the performance in case of re-investigation similar plarform (many targeted computers with similar buits) and maintain consistency [20].

References

[1] He, Z., and Sayadi, H. (2023, April). Image-Based Zero-Day Malware Detection in IoMT Devices: A Hybrid AI-Enabled Method. In 2023 24th International Symposium on Quality Electronic Design (ISQED) (pp. 1-8). IEEE.

[2] Nguyen, T.T. and Reddi, V.J., 2021. Deep reinforcement learning for cyber security. IEEE Transactions on Neural Networks and Learning Systems, 34(8), pp.3779-3795. https://doi.org/10.1109/TNNLS.2021.3121870

[3] Demontis, A., Melis, M., Biggio, B., Maiorca, D., Arp, D., Rieck, K., Corona, I., Giacinto, G. and Roli, F., 2017. Yes, machine learning can be more secure! a case study on android malware detection. IEEE transactions on dependable and secure computing, 16(4), pp.711-724. https:doi.org/10.1109/TDSC.2017.2700270

[4] Wu, C., Shi, J., Yang, Y., and Li, W. (2018, November). Enhancing machine learning based malware detection model by reinforcement learning. In Proceedings of the 8th International Conference on Communication and Network Security (pp. 74-78).

[5] Liu, L., Wang, B. S., Yu, B., and Zhong, Q. X. (2017). Automatic malware classification and new malware detection using machine learning. Frontiers of Information Technology and Electronic Engineering, 18(9), 1336-1347.

[6] Penmatsa, R. K. V., Kalidindi, A., and Mallidi, S. K. R. (2020). Feature reduction and optimization of malware detection system using ant colony optimization and rough sets. International Journal of Information Security and Privacy (IJISP), 14(3), 95-114.

[7] Song, W., Li, X., Afroz, S., Garg, D., Kuznetsov, D., and Yin, H. (2020). Mab-malware: A reinforcement learning System for attacking static malware classifiers. https://arxiv.org/abs/2003.03100

[8] Ebrahimi, M. R., Li, W., Chai, Y., Pacheco, J., and Chen, H. (2022, November). An Adversarial Reinforcement Learning System for Robust Machine Learning-based Malware Detection. In 2022 IEEE International Conference on Data Mining Workshops (ICDMW) (pp. 567-576). IEEE.

[9] Zhong, F., Hu, P., Zhang, G., Li, H., and Cheng, X. (2022). Reinforcement learning based adversarial malware example generation against black-box detectors. Computers and Security, 121, 102869.

[10] Quertier, T., Marais, B., Morucci, S. and Fournel, B., 2022. MERLIN–Malware Evasion with Reinforcement LearnINg. arXiv preprint arXiv:2203.12980. Available at: https://arxiv.org/abs/2203.12980

[11] Wang, C., Kakkar, A., Redino, C., Rahman, A., Ajinsyam, S., Clark, R., ... and Bowen, E. (2023, April). Discovering Command and Control Channels Using Reinforcement Learning. In SoutheastCon 2023 (pp. 685-692). IEEE.

[12] Yu, S., 2022. Fast Detection of Advanced Persistent Threats for Smart Grids: A Deep Reinforcement Learning Approach. In ICC 2022-IEEE International Conference on Communications (pp. 2676-2681). IEEE.

[13] Ghanem, M.C., Mulvihill, P., Ouazzane, K., Djemai, R. and Dunsin, D., 2023. D2WFP: a novel protocol for forensically identifying, extracting, and analysing deep and dark web browsing activities. Journal of Cybersecurity and Privacy, 3(4), pp.808-829. https://doi.org/10.3390/jcp3040036

[14] Gao, J., and Fang, Z. (2022, October). Utilizing benign files to obfuscate malware via deep reinforcement learning. In 2022 4th International Conference on Intelligent Information Processing (IIP) (pp. 293-297). IEEE.

[15] Ghanem, M., Mouloudi, A. and Mourchid, M., 2015. Towards a scientific research based on semantic web. Procedia Computer Science, 73, pp.328-335. https://doi.org/10.1016/j.procs.2015.12.041

[16] Xue, Y., Shu, H., Bu, W., and Qu, W. (2020, October). Malicious Code Detection Technology Based on A3C Algorithm. In 2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS) (pp. 116-120). IEEE.

[17] Dunsin, D., Ghanem, M.C., Ouazzane, K. and Vassilev, V., 2024. Reinforcement Learning for an Efficient and Effective Malware Investigation during Cyber Incident Response. https://doi.org/10.48550/arXiv.2408.01999.

[18] Dunsin, D., Ghanem, M.C., Ouazzane, K., Vassilev, V., 2024. A comprehensive analysis of the role of artificial intelligence and machine learning in modern digital forensics and incident response. Forensic Science International. Digital Investigation 48, 301675. https://doi.org/10.1016/j.fsidi.2023.301675

[19] Moore, L., 2009. Review: Rethinking Miscarriages of Justice: Beyond the Tip of the Iceberg Michael Naughton Palgrave Macmillan, Houndsmills, 2008, 233pp, ISBN 978—0—230—01906—5, £45.00 (hbk). Critical Social Policy 29, 296–297. https://doi.org/10.1177/02610183090290020702

[20] Ghanem, M.C., Chen, T.M., Ferrag, M.A. and Kettouche, M.E., 2023. ESASCF: expertise extraction, generalization and reply framework for optimized automation of network security compliance. IEEE Access. https://doi.or/10.1109/ACCESS.2023.3332834